I. Introduction

a. Présentation du candidat

Je m'appelle Thomas Touhey, j'ai 21 ans, et je suis stagiaire au GRETA de l'Essonne en BTS SNIR pour la session qui a commencé le 18 septembre 2017 et se termine le 13 juin 2018 avec notre oral final de projet.

J'ai trouvé ma voie, à savoir l'informatique, dès mes 13/14 ans ; cependant, j'ai connu un parcours assez atypique dans mes études. En effet, après avoir obtenu un bac S en 2014 avec mention bien, j'ai fait une année de MPSI (math sup), puis, après un bref passage en DUT que j'ai apprécié mais interrompu pour ce qui suit, une année à tenter l'école 42 à Paris, suivi d'une première année de licence de sciences et de technologie à l'Université Pierre et Marie Curie (UPMC), à distance avec le CNED. Bien que ces années m'ont toutes permis d'acquérir des compétences, il me faut obtenir un diplôme pour me placer dans le monde professionnel rapidement ; c'est ce qui m'a mené à ce BTS.

Concernant mon expérience en informatique, je suis parvenu à un niveau assez avancé en auto-didacte grâce à une grande curiosité que j'ai développée en allant toujours plus loin, et en ayant commencé, comme beaucoup dans le milieu, par le développement de jeux. En effet, l'une des premières tâches que je me suis donné dans ce domaine était le portage d'un jeu de rôle assez simple pour lequel je réalisais des maquettes en papier sur ordinateur, d'abord avec Game Maker puis, quand cela n'a plus suffi à satisfaire ma curiosité, en C avec la SDL à l'aide du tutoriel de M@teo21 sur le Site du Zéro.

J'ai ensuite enchaîné, pour des serveurs du jeu auquel je jouais principalement en seconde et en première, à réaliser des petits sites web personnels en PHP, jusqu'à ce que mon année de MPSI me mette au Python pour faire un peu d'algorithmie. Quand j'ai commencé à 42, j'ai rencontré quelqu'un là-bas qui m'a introduit au développement bas niveau sur les calculatrices CASIO, et j'ai combiné cela avec du développement système pour faire un set d'utilitaires pour gérer les protocoles de communication et formats de fichiers relatifs à ces machines. J'ai également, en parallèle, continué le développement web en Python avec un micro-framework nommé Flask, et ai continué tout un tas de petits projets que j'ai finis ou abandonnés au fur et à mesure.

J'ai également changé d'environnement de façon assez conséquente : en effet, du Windows XP sur lequel j'ai commencé, après quelques années sous Windows 7, je suis passé à GNU/Linux avec d'abord une année sur Debian GNU/Linux, puis bientôt deux ans sous Manjaro GNU/Linux. J'ai aussi eu quelques serveurs sur lesquels j'ai expérimentés l'installation de services type Mastodon, etc. En général, j'ai lu beaucoup d'informations sur ce qui se présentait et que je trouvais intéressant, et je continue de le faire.

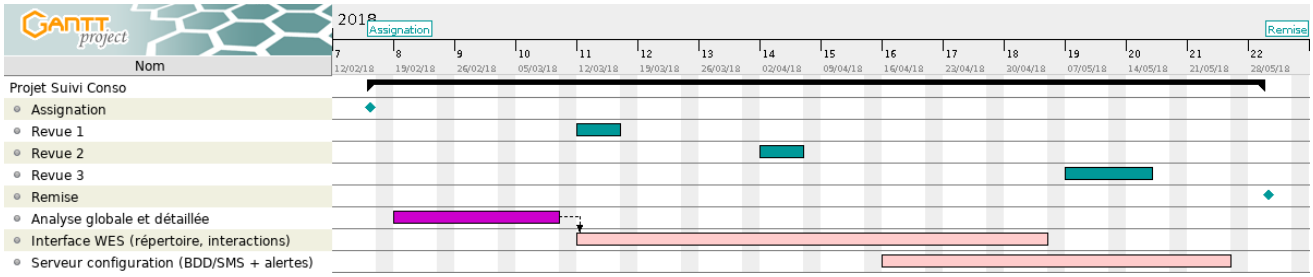
b. Projet USIMARIT

Concernant le projet actuel, j'ai un peu dévié de mon plan de départ pour plusieurs raisons :

- j'ai poussé la maîtrise des serveurs WES un peu plus loin car je pensais que j'aurais le temps de produire quelque chose de plus tenace face au temps ;

- j’ai repris une partie du travail du candidat 2 (à savoir le serveur du système d’échange Tonique) pour qu’on aie un ensemble fonctionnel d’ici le jour de notre oral, le 13 juin.

Mon travail s’est plus ou moins organisé de la façon suivante :



J’ai également assumé une partie du *leadership* du projet, ayant un *background* déjà technique et des connaissances que j’ai pu mettre en œuvre pour le projet, notamment en termes de développement système et en protocoles de communication, ainsi qu’en développement bas niveau (surtout C, j’ai appris le C++ en cours d’année, avec quelques bases et surtout une raison d’apprendre fournie par l’équipe pédagogique).

c. Sommaire

<insérer le sommaire de la partie ici>

II. Gestion des serveurs WES

Comme décrit dans la partie commune, au niveau de chaque logement et des parties communes se trouve un serveur WES, qu'il s'agit donc de manipuler. J'ai choisi de représenter le serveur WES comme une machine bardée de différents éléments, parmi lesquels on peut trouver :

- Des **compteurs**, qui comptent l'énergie consommée (e.g. l'électricité) ;
- Des **capteurs**, qui captent régulièrement une grandeur donnée (e.g. la température) ;
- Un **afficheur**, qui permet d'interagir avec l'utilisateur.

Ainsi, une telle abstraction permettra aux élèves suivants, s'ils le souhaitent, de remplacer le serveur WES par une autre machine tout en conservant une partie du travail actuel, sur le sous-projet *weshd* lui-même (que je décrirai plus tard) comme sur les autres sous-projets qui se servent de ce composant comme le système d'échange *Tonique*. On pourrait imaginer par exemple un autre Raspberry Pi reliée directement à la sortie téléinfo d'un compteur électrique, qui déclarerait donc un seul compteur (téléinfo), aucun capteur et aucun afficheur.

Les types de compteurs représentables dans l'abstraction que j'ai posée sont les suivants :

- Les **compteurs d'impulsions**, où chaque impulsion signifie qu'une certaine quantité d'énergie, connue à l'avance par les deux parties, a été consommée. Ces impulsions peuvent être mécaniques (ILS) ou électroniques. Par exemple, un calorimètre peut envoyer une impulsion au serveur WES chaque fois que quatre calories a été consommée (on a alors 0.25 impulsion/calorie, avec une précision de 4 calories).
- Les **pinces ampèremétriques**, qui détectent l'intensité d'un courant électrique qui passe dans la pince par rapport au champ magnétique qu'il produit. Cette méthode n'est pas utilisée dans le cadre du projet USIMARIT.
- La sortie de **télé-information** client défini par Enedis (souvent abrégée « téléinfo »), utilisable avec les compteurs électriques modernes qui possèdent une telle sortie. Cette liaison est une

liaison série où le transfert est unidirectionnel, du compteur au serveur qui récupère les données, sur laquelle le compteur envoie des trames d'information de façon régulière, trames qui contiennent un ensemble d'informations structurées sous la forme d'un tableau associatif. Parmi ces informations, on trouve par exemple l'adresse du concentrateur de téléreport (soit l'identifiant du compteur électrique) via la clé « ADCO », ou la quantité d'énergie consommée pour chaque période tarifaire, comme celle consommée lors de la période « Heures Creuses » d'un format « Heures Creuses, Heures Pleines » exprimée en kVA (kilovoltampères) via la clé « HCHC ».

On trouve également, dans mon abstraction, les types suivants de capteurs :

- Capteurs de température et d'humidité reliés au serveur WES par le bus **1-Wire**, défini par Dallas Semiconductor. Ce bus peut techniquement accueillir bien plus que ces deux éléments, mais le serveur WES ne supporte que ces deux-là.

Par-dessus ces éléments, le système (micrologiciel) présent sur le serveur WES pose ses propres interfaces destinées à l'utilisateur, dont notamment une interface web pour que l'utilisateur puisse piloter la machine et contrôler sa consommation à l'aide de graphiques divers depuis son navigateur web. Mais dans le cadre du projet USIMARIT, nul n'interagit avec l'interface web du serveur WES ; en effet, les interactions que nous avons avec le serveur WES sont les suivantes :

- les résidents peuvent interagir avec l'afficheur du serveur WES pour visualiser leur consommation actuelle directement ;
- les serveurs centraux récupèrent les données de consommation depuis les serveurs WES à but de conservation et de traitement ;
- les serveurs centraux modifient les tarifs et interagissent avec l'afficheur du serveur WES pour, par exemple, faire clignoter l'écran en rouge lorsqu'un seuil de consommation d'énergie a été dépassé.

Nous nous intéressons ici aux second et troisième points. Dans la partie suivante, je décrirai quelles interfaces j'ai implémentées pour le faire, et dans celle d'après, de la façon dont j'ai choisi de le faire dans le sous-projet associé, intitulé « **weshd** » (pour « **WES H**andling **D**aemon »).

a. Interfaces proposées par les serveurs WES

Le central a quelques interfaces à sa disposition parmi ce qu'offre le serveur WES pour remplir ses missions :

- Pour la **récupération** d'informations :
  - **Récupération de fichiers CSV contenant les données des compteurs** : si l'option correspondante est activée dans l'interface web, le système du serveur WES se met à écrire les données de façon historique dans différents dossiers (« teleinfo » pour les données des entrées téléinfo, « PCE » pour celles des pinces ampèremétriques, et « PLS » pour celles des compteurs d'impulsions) dans des fichiers CSV. Cette méthode était utilisée lors des années précédentes du projet que nous avons repris sous le nom de projet USIMARIT, et nous l'utiliserons aussi pour les données non-instantanées.
  - **Récupération des données de graphes** : le serveur WES stocke systématiquement des données pour générer les graphes visibles via l'interface web. Les données correspondantes sont stockées dans le dossier « GRAPH/ », et sont sous la forme de fichiers « .csm » et « .dat» organisés en dossiers concernant la date des mesures. Ces fichiers sont probablement des médiums plus sûrs que les fichiers CSV, mais leur format n'est pas documenté publiquement et je n'ai pas eu le temps de m'attarder dessus.
  - **Utilisation du protocole M2M** : un protocole machine à machine a été implémenté par Cartelectronic comme protocole d'application juste au-dessus de TCP ou UDP (avec un port personnalisé, 1500 par défaut). Pour l'utiliser, il faut activer l'option (désactivée par défaut) sur l'interface web du serveur WES en indiquant, pour le protocole over TCP et le même protocole over UDP, le port à utiliser. On ne peut récupérer beaucoup de données via ce protocole, qui n'est d'ailleurs pas encore très stable, et il va surtout nous servir pour du contrôle.
  - **« Web scraping »**, soit récupération des pages web comme le ferait un utilisateur puis extraction dans le document HTML obtenu des informations : cette méthode est très peu pratique et très instable, puisque toute modification des pages web destinées à être vues par un utilisateur humain, ce qui est fait sans préavis généralement (comme le récent changement à Bootstrap opéré par Cartelectronic), peut casser cette technique. Comme partout, on n'utilise donc cette technique qu'en dernier recours.
  - **Utilisation de la fonctionnalité des « triggers »** (déclencheurs, en résumé : « si ceci alors fais cela ») : fait que le serveur WES peut effectuer, sous certains conditions, des requêtes HTTP ou des envois de courriels (requêtes SMTP) pour envoyer des informations sous un format donné de façon régulière. Cette méthode requiert de mettre en place un serveur HTTP ou SMTP, ce qui est assez lourd, et ne permet pas d'accéder à beaucoup d'informations ; elle reste cependant intéressante pour la détection de seuils depuis le serveur WES directement (et non pas depuis les Raspberry Pi comme actuellement).

    - **Récupération de fichiers CGX générés à chaque requête** : le serveur WES propose des données de façon instantanée aux applications sous la forme de fichiers XML dont la structure est plus ou moins décrite par la documentation officielle. Ce moyen est sympathique et on peut récupérer une multitude de données, et peut servir de solution de repli pour ces données si les méthodes meilleures échouent.
    - **Exécution de scripts CGI** : cette méthode est un détournement du fonctionnement de l'interface web du système des serveurs WES. Pour rendre dynamiques les pages web du serveur WES, celles-ci sont en réalité des scripts dans un langage inventé par Cartelectronic où chaque ligne représente une instruction du type « afficher du texte brut », « afficher une donnée de telle façon », ou autre. Cette méthode permet d'accéder à toutes les données affichables sur les pages web (donc toutes les options de configuration et toutes les données instantanées), et même si elle est plus lourde en opérations que les autres, elle est plus efficace et, implémentée correctement, facile à utiliser. Inconvénient, comme elle est assez peu documentée par Cartelectronic, certains éléments de l'interface, comme les noms des données, peuvent changer, et c'est effectivement arrivé durant mes tests avec la variable qui permettait d'accéder au nom d'hôte du serveur NTP utilisé. C'est cette méthode que nous allons tout de même utiliser, d'une façon que je détaillerai après.
- Pour le **contrôle** du serveur WES et de sa configuration :
    - **Utilisation du protocole M2M** : le protocole M2M décrit plus tôt peut servir pour le contrôle de certains éléments du serveur WES. Il nous intéressera surtout pour la manipulation de l'afficheur : en effet, le contenu de l'écran et son clignotement sont définissables via ce moyen.
    - **Édition des fichiers de configuration présents dans le dossier « CFG/ »**, via FTP ou en récupérant la carte SD du serveur WES à froid, la branchant au central, modifiant les fichiers qui vont bien, puis en la remettant dans le serveur WES et en démarrant celui-ci à nouveau (requiert un humain, on écarte donc cette possibilité immédiatement) : dans les deux cas, pour que la nouvelle configuration prenne effet, il faut que la machine soit redémarrée, cette interface est donc très peu pratique.
    - **Formulaires GET et POST** : dans la même veine que les scripts CGI évoqués précédemment, pour avoir le même niveau de contrôle que l'utilisateur via l'interface web, autant imiter ce qui se passe lorsqu'il clique sur « Valider » sur les différents formulaires sur les pages web correspondantes ! C'est la méthode que nous choisirons pour la plupart des actions.

1. Interface CGI

Le format des scripts utilisé par le serveur web du serveur WES est un format propriétaire décrit de façon minimale dans la documentation du WES. Ce format fonctionne par instructions codées en UTF-8 non normalisé qui occupent chacune une ligne, où les lignes sont séparées par des CRLF (soit le

retour chariot « CR », de code U+000D et représenté par '\r' en C, suivi du retour à la ligne « LF », code U+000A et représenté par '\n' en C). Si des caractères doivent être lues de la ligne (comme l'instruction ou le code de donnée pour une commande) mais que le retour à la ligne a été atteint, ces caractères sont lus comme des espaces.

Une ligne commence toujours par deux caractères représentant l'instruction. Les instructions disponibles sont les suivantes :

- **« # » (U+0023 puis U+0020)** : commentaire, la ligne ne sera pas envoyée au client HTTP.
- **« t » (U+0074 puis U+0020)** : texte, le reste de la ligne sera envoyée sans modification au client HTTP suivie d'un CRLF.
- **« i » (U+0069 puis U+0020)** : inclusion, le fichier dont le reste de la ligne contient le nom, e.g. « BANDEAU.INC », sera inclus sans modification. Le fichier inclus ne doit pas contenir d'instructions, mais bel et bien le contenu directement (autrement, j'aurais appelé ça une importation).
- **« c » (U+0063 puis U+0020)** : commande (voir ci-dessous).
- **« . » (U+002E puis U+0020)** : fin de fichier, cette commande doit être présente, et en dernière. Le reste de la ligne est ignoré, et peut être vide.

Une commande affiche une donnée. Les deux caractères d'instruction sont suivis par quatre caractères représentant le code de la donnée à afficher, où souvent (mais pas toujours), les deux premiers représentent la section de la donnée et les deux derniers la donnée elle-même. Dans le cadre de cette instruction, il y a trois types de données :

- Les données représentables avec un format : après les quatre caractères représentant le code de la donnée, on trouve un format printf-like où sont introduits divers éléments qui dépendent de la donnée. Par exemple, pour l'heure du serveur (code « H h »), trois entiers sont fournis, le premier représentant les heures, le deuxième les minutes, et le troisième les secondes, on peut donc utiliser le format « Heure du serveur : %02d:%02d:%02d ».
- Les données sans format : ce sont généralement des données contenant les lignes d'un tableau, soit zéro, une ou plusieurs lignes, dont on ne peut pas commander le format (le reste de la ligne est donc ignoré). Par exemple, le tableau de l'état des sockets utilisées (code « r o ») renvoie plusieurs lignes de tableau où chaque ligne ressemble à « <tr><td>4</td>…<td>120</td></tr> ».
- Les données inconnues : celle-là ne produisent rien. Lorsqu'une donnée du premier type ne renvoie aucune ligne, cela veut probablement dire que la donnée n'existe plus ou a changé de code.

Certaines variables sont documentées dans un document fourni par Cartelectronic intitulé « *CGI_WES.pdf* » justement produit pour que les clients puissent produire leurs propres scripts. D'autres variables ont été obtenues directement depuis les pages web existantes, via rétro-ingénierie en

examinant ce que chaque donnée produit et en examinant son contexte dans la page (si le code est précédé par un texte décrivant ce à quoi correspond la commande, etc).

Pour exploiter ces scripts CGI, au moins quatre phases sont nécessaires : la production d'un script, le téléversement (upload) de celui-ci via FTP, l'exécution de celui-ci (en récupérant la réponse) via HTTP et décodage de la réponse. Pour le téléversement via FTP et l'exécution via HTTP, nous exploiterons l'interface « easy » de la **libcurl**, qui essaie d'uniformiser l'ensemble des protocoles que celle-ci gère en un système de requêtes dont on définit les options.

Dans l'implémentation faite dans *weshd*, en ce qui concerne le téléversement du script via FTP, nous produisons le script à la volée tandis qu'il est téléversé, via une fonction de lecture de flux et une donnée qui lui est associée, opaque à la *libcurl* (ce qui veut dire qu'elle transmettra le pointeur sans savoir ce vers quoi il pointe). L'utilisation de la libcurl se fait ainsi :

```
 1 CURL *handle;
 2 CURLcode result;
 3
 4 handle = curl_easy_init();
 5
 6 curl_easy_setopt(handle, CURLOPT_URL, "ftp://192.168.1.111/wh1.cgi");
 7 curl_easy_setopt(handle, CURLOPT_USERNAME, "adminftp");
 8 curl_easy_setopt(handle, CURLOPT_PASSWORD, "wesftp");
 9 curl_easy_setopt(handle, CURLOPT_UPLOAD, 1L);
10 curl_easy_setopt(handle, CURLOPT_READDATA, &cookie);
11 curl_easy_setopt(handle, CURLOPT_READFUNCTION, sendscript_read);
12
13 result = curl_easy_perform(curl);
14 if (result != CURLE_OK) {
15       fprintf(stderr, "curl error: %s\n", curl_easy_strerror(result));
16
17       /* gestion d'erreur… */
18 }
```

La fonction de génération générera un script comme celui-ci :

```
 1 c h d %02d/%02d/%04d
 2 t --- REQUEST SEPARATOR ---
 3 c h h %02d:%02d:%02d
 4 t --- REQUEST SEPARATOR ---
 5 c h N %.1s
 6 t --- REQUEST SEPARATOR ---
 7 c Mua %.1s
 8 t --- REQUEST SEPARATOR ---
 9 c P A1%.2f
10 t --- REQUEST SEPARATOR —
11 c h I %s
12 t --- REQUEST SEPARATOR ---
13 .
```

Une fois exécuté, la réponse obtenue de façon similaire avec la *libcurl* sera de cet acabit :

```
1 27/05/2018
2 --- REQUEST SEPARATOR ---
3 19:59:01
4 --- REQUEST SEPARATOR ---
```

```
 5
 6 --- REQUEST SEPARATOR ---
 7 c
 8 --- REQUEST SEPARATOR ---
 9 0.00
10 --- REQUEST SEPARATOR ---
11 --- REQUEST SEPARATOR ---
```

Ici, on a les réponses pour chacune des commandes :

- La réponse à « h d » (date actuelle) avec le format « %02d/%02d/%02d » a donné une ligne de réponse contenant « 27/05/2018 » ;
- La réponse à « h h » (heure actuelle) avec le format « %02d:%02d:%02d » a donné une ligne de réponse contenant « 19:59:01 » ;
- La réponse à « h N » (l'heure est-elle mise à jour via NTP ? « checked » si oui, rien sinon ») avec le format « %.1s » (on n'affiche la chaîne avec une précision de 1, soit simplement le premier caractère de la chaîne) renvoie une ligne vide, ce qui n'est **pas** équivalent à aucune ligne (voir pour « h I » ci-après), ce qui signifie que l'heure n'est pas mise à jour via NTP ;
- La réponse à « Mua » (le protocole M2M est-il activé en UDP ? « checked » si oui, rien sinon) avec le format « %.1s » renvoie une ligne simplement composée du premier caractère de « checked », ce qui veut dire que le protocole M2M est bien activé au-dessus du protocole de transport UDP ;
- La réponse à « P A1 » (intensité instantanée mesurée par la pince ampèremétrique numéro 1 en ampères) avec le format « %.2f » donne une ligne contenant « 0.00 », ce qui signifie que l'intensité instantanée mesurée la première pince ampèremétrique est nulle.
- La réponse à « h I » (anciennement nom d'hôte du serveur NTP, code obsolète) avec le format « %s » n'a renvoyé aucune ligne, ce qui veut dire que la donnée n'existe probablement plus ou a été renommée.

La liste des données accessibles connues à ce jour, avec leur format s'il y a lieu, est documentée dans le fichier « doc/CGI.rst » du projet *weshd*.

2. Formulaires GET et POST

Pour agir sur le WES, la méthode que l'on utilisera la plupart du temps imite la validation de formulaires par l'utilisateur sur l'interface web. En effet, lorsque cela se produit, le formulaire complété est envoyé au WES via deux différentes méthodes :

- **Formulaires GET** : les champs et leur valeur sont envoyés dans l'URL d'une requête de type POST, e.g. « http://192.168.1.111/page.cgi?nm=western&TIC1=ON&TIC2=OFF » ;
- **Formulaires POST** : les champs et leur valeur sont envoyés dans le contenu d'une requête de type POST, formatté de la même façon, e.g. « LAL=5&LBL=0.5 ».

Par souci de ne pas verrouiller les formulaires à un set de pages particulier, les deux types de formulaires marchent pour toutes les pages, y compris celles pour lesquels le fichier n'a pas été trouvé (chemins qui produisent une erreur 404). Les deux formulaires, pour l'interface web du serveur WES, ne permettent pas de définir les mêmes options, et autant les paramètres des formulaires GET peuvent être utilisés indépendamment ou ensembles, autant les paramètres des formulaires POST doivent être utilisés par des combinaisons fixes, même si certains paramètres ne sont pas à redéfinir (il faudra récupérer les données actuelles auparavant pour pouvoir renvoyer la donnée actuelle, comme ce qui se produit pour l'utilisateur).

L'implémentation actuelle dans *weshd* a été fusionnée avec la récupération de variables CGI (on exécute un script et on en profite pour passer des paramètres au passage – et comme les deux utilisent la *libcurl*, c'est plus logique). La liste des options disponibles connues à ce jour est documentée dans le fichier « doc/HTTP.rst ».

3. Protocole de machine à machine (M2M)

Nous nous servons également du protocole M2M, protocole d'application au-dessus de TCP ou UDP, posé par Cartelectronic pour ses serveurs WES, notamment à des fins de commande. Ce protocole est basé sur un modèle requête/réponse (souvent abrégé « reqrep »), où la requête et la réponse tiennent en une ligne qui se termine par un LF (U+000A). En TCP, cela se fait sur un canal sans souci, et en UDP, la requête doit tenir en un paquet et la réponse tiendra en un paquet (bien que le comportement exact à adopter sur de l'UDP n'est pas documenté par Cartelectronic). Si la commande est inconnue, « ??? » suivi d'un retour à la ligne LF est retourné.

Il n'y a pas de format fixe qui marche pour toutes les commandes (j'ai une exception pour toutes les règles que j'ai essayé de poser). Mais l'idée générale est que si on met le nom d'une donnée, on récupère sa valeur, parfois avec un préfixe constitué du nom suivi par un '=', parfois non, et si on met le nom de la donnée puis un '=' et une valeur, on affecte une valeur à la variable.

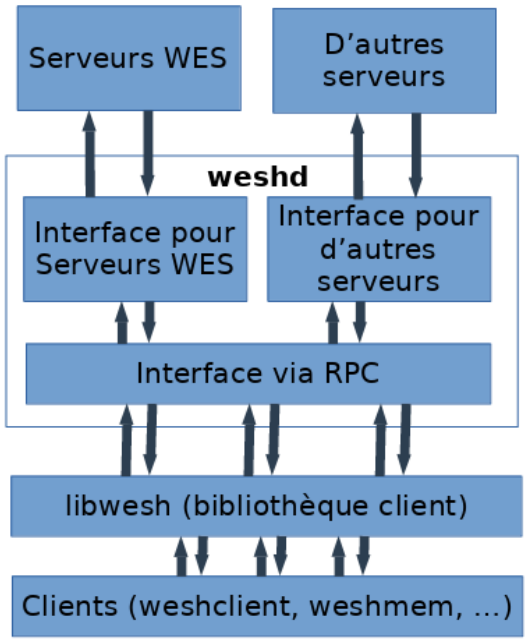
Mais certaines données rendent le principe obscur. Par exemple, pour la variable correspondant à l'alarme, `lcd_a` (`lcd_alm` en pré-V0.7A), on doit lui affecter 0 ou 1 pour éteindre et allumer l'alarme de l'écran LCD respectivement, et si on récupère son contenu, on a une valeur comme 536876024 qui augmente perpétuellement, donc la valeur de la donnée et celle qu'on lui affecte n'ont rien à voir. D'autres exemples de données problématiques sont l'adresse MAC (`gmac`) ou la version du firmware (`gfw`) qu'on ne peut pas affecter et auxquelles on peut d'ailleurs concaténer d'autres caractères, e.g. `gmachaha`, sans que ça ne change quelque chose.Pour l'écran, il faut définir les données `lcd_1`, `lcd_2`, `lcd_3` puis `lcd_4` avec le contenu de chaque ligne.

Pour tester tout cela, sous GNU/Linux, une fois l'option M2M over TCP activée et en supposant que le port TCP utilisé soit le port 1500, rien de tel que l'utilitaire en lignes de commande `ncat` pour offrir une interface simple (le texte en bleu est celui que je rentre) :

```
1 user@host :: ~ » ncat 192.168.1.111 1500
2 gmac                                  » On récupère l'adresse MAC.
```

```
001EC09497DB
gfw                                     » On récupère la version du
V0.7A                                   micrologiciel du serveur WES.
lcd_a
LCD_ALM=536876024
lcd_a=1                                 » On active l'alarme de l'afficheur.
LCD_ALM=1
lcd_b=0                                 » On désactive la backlight.
LCD_BKL=0
lcd_b=1                                 » On active la backlight.
LCD_BKL=1
^D                                      » On quitte l'utilitaire en
user@host :: ~ »                        provoquant une fin de fichier sur
                                        l'entrée standard (Ctrl+D)
```

b. Architecture du projet *weshd*



Le projet *weshd* en lui-même est basé sur le principe provenant d'UNIX : « une ressource, un démon ».

Un démon est un service sur les systèmes type UNIX (respectant les standards POSIX/Single Unix Specification), soit un logiciel tournant en arrière plan, qui s'est détaché de son processus parent pour devenir autonome et continuer de tourner quand son parent (le processus qui a lancé le logiciel) meurt.

L'intérêt de concentrer la gestion de ressources tels que les serveurs WES dans un seul logiciel plutôt que dans de multiples clients (ou de multiples instances d'un même client, ou même une combinaison des deux) permet d'éviter les conflits (« race condition » dans la langue de Shakespeare) : si deux clients essaient de définir une même propriété d'un serveur WES donné en même temps, l'issue est incertaine, alors que si un démon est là pour arbitrer les requêtes et leur ordonnancement (voire même si elles peuvent être lancées en même temps ou non), cela élimine ce genre de risques.

Seulement, faire un démon complexifie quelque peu l'architecture du sous-projet, puisqu'en plus du démon, il faudra donc des clients et un moyen d'interfacer les deux pour que les clients puissent envoyer des requêtes et pour que le démon puisse leur répondre, ainsi qu'une bibliothèque pour que les clients n'aient pas à tous redéfinir l'interface dans leur code source.

1. Choix de l'interface entre les clients et le démon

Au début, parce que je n'avais jamais fait ça et que je ne connaissais pas quels étaient les moyens les plus pratiques et efficaces de faire de la communication inter-processus (en anglais : IPC, « Inter-Process Communication) sur le modèle requête/réponse (souvent abrégé « reqrep »), j'avais commencé par proposer **nanomsg**, un système de transport de messages proposant plusieurs modèles dont le modèle « reqrep » que je cherchais, léger, multi-plateformes (tourne aussi bien sous Microsoft Windows® que des systèmes type UNIX) et supportant plusieurs méthodes de transport en-dessous, permettant même de communiquer entre processus de différentes machines avec certains modes de transport (comme le TCP). Mais nanomsg ne s'occupant que du transport, il me fallait un moyen de coder l'information par-dessus, et comme j'avais récemment regardé du côté de l'**ASN.1** et que je souhaitais expérimenter avec en posant les définitions dont j'avais besoin pour ce cas, c'est ce que j'ai commencé à faire.

Mais plus tard, alors que je tentais de mettre en place mon projet avec nanomsg et **asn1c** (générateur de décodeur et d'encodeur en ASN.1 à partir d'un set de définitions), j'ai ré-entendu parler d'un mode de communication beaucoup plus simple : le **RPC**, « Remote Procedure Call », soit de l'appel de

procédure distante. Il existe au moins une dizaine de systèmes de ce genre, adaptés à différents contextes, mais puisque mon environnement de test, comme la Raspberry Pi, est sous GNU/Linux, j'ai choisi l'un des premiers systèmes de RPC qui tourne toujours sur cet OS de nos jours (bien qu'il ne soit plus beaucoup utilisé), c'est celui développé par Sun (depuis racheté par Oracle) dans les années 70/80, officiellement appelé « ONC RPC » mais j'y fais référence par son nom officieux, « **SunRPC** ».

Petite anecdote avant de continuer : pendant le développement de *weshd*, l'implémentation de SunRPC dans la GNU C Library, bibliothèque de base du système pour le développement en C présente sur notre système, a été retirée. Nous avons donc dû utiliser une bibliothèque externe, et avons choisi la **libtirpc** qui a quelques différences mineures pour lesquelles nous avons dû adapter mon code, comme le fait que l'adresse IP source des requêtes ne soit plus une adresse IPv4 mais une adresse IPv6, mais qui fonctionne comme l'originale.

Pour le développement d'une interface avec SunRPC, tout commence avec la définition du protocole SunRPC, qui va contenir la définition du programme, de ses versions et des procédures disponibles pour chaque version, ainsi que celle des types utilisés : structures, énumérations, constantes, unions. Ce protocole doit être écrit dans un fichier dont l'extension est « .x », comme, pour *weshd*, à la racine du projet, « weshd.x ». À partir de ce fichier, l'utilitaire `rpcgen` génère divers éléments :

- une en-tête (d'extension « .h ») utilisable en C/C++ ;
- la définition des fonctions à utiliser pour le client ;
- la définition des fonctions à utiliser pour le serveur ;
- la définition des fonctions d'encodage et de décodage des types utilisés dans le protocole en XDR (« eXternal Data Representation ») pour le protocole.

Pour vous démontrer le principe, voici un protocole d'exemple pour faire une addition :

```
 1 struct add_params {
 2       int a;
 3       int b;
 4 };
 5
 6 program EXAMPLE_PROG {
 7       version EXAMPLE_VERS1 {
 8             /* Fonction recommandée, qui ne fait rien, pour vérifier
 9              * l'état du service. */
10
11             void EXAMPLE_NULL(void) = 0;
12
13             /* Fonction qui fait une addition. */
14
15             int EXAMPLE_ADD(add_params params) = 1;
16       } = 1;
17 } = 0x20012345;
```

Ici, nous définissons le service `EXAMPLE_PROG` de code `0x20012345` (les codes en dehors de la zone `0x20000000 - 0x2FFFFFFF` sont standardisés, j'utilise ici un code non standardisé) qui, dans sa version première (de code 1), définit deux procédures :

- `EXAMPLE_NULL` (de code 0) : ne fait rien et un équivalent est présent dans beaucoup de protocoles standards défini dans le dossier `/usr/include/rpcsvc/`. Elle permet de mesurer le délai de réponse du service.
- `EXAMPLE_ADD` (de code 1) : fait une addition et renvoie le résultat. L'on remarquera que dans la définition d'un protocole SunRPC, historiquement, les possibilités sont beaucoup plus limitées qu'en C, et l'on ne peut avoir qu'un argument ; pour contrecarrer cela, il faut donc faire une structure contenant les multiples arguments.

De plus, comme vous l'aurez remarqué, les `typedef` sont faits automatiquement dans les fichiers sources et d'en-tête et ne font de toute manière pas partie de la syntaxe des protocoles SunRPC.

Une fois le protocole défini, l'étape d'après est de gérer le serveur. Tout d'abord, on commence par poser la fonction qui va lancer le serveur. Dans l'ordre, il faudra qu'on :

- Crée le transport pour l'interface (ici, ça sera du TCP), via `svctcp_create()`.
- Crée l'application et l'enregistre auprès du *portmapper* avec le numéro de service et le numéro de version, le *portmapper* étant un service via lequel, par défaut, toutes les requêtes SunRPC transitent (c'est lui qui occupe le port `tcp/113`), via `svc_register()`.
- Entre en mode serveur en donnant le contrôle du fil d'exécution aux fonctions de la *libtirpc*, via `svc_run()`.

Les fonctions de la bibliothèque ne gérant pas le signal `SIGKILL` (`^C`, `Ctrl+C`) pour interrompre le service, si nous souhaitons nous interrompre proprement, nous devons gérer ce signal pour qu'il aille à la fin de la fonction qui lance le serveur pour déinitialiser les ressources. Pour cela, nous définissons notre fonction pour gérer le signal et l'enregistrons avec la fonction `signal()`, et nous utilisons le système de sauvegarde et restauration d'environnement défini par les fonctions `setjmp()` et `longjmp()` pour aller au bon endroit.

Aussitôt dit, aussitôt fait, voici donc le code de la fonction de lancement du serveur :

```
 1 #include <signal.h>
 2 #include <setjmp.h>
 3 #include <arpa/inet.h>
 4 #include <rpc/pmap_clnt.h>
 5 #include "example.h"
 6
 7 static jmp_buf endjmp;
 8
 9 static __attribute__((noreturn)) void its_the_signal(int sig)
10 {
11       (void)sig;
12       longjmp(endjmp, 1);
13 }
14
15 int run_server(void)
16 {
17       SVCXPRT *tcp = NULL;
```

```
18
19      /* On libère le port sur le service portmap (portmapper) pour
20       * pouvoir nous enregistrer auprès de lui. */
21
22      pmap_unset(EXAMPLE_PROG, EXAMPLE_VERS1);
23
24      /* On crée le transport via TCP, et on crée l'application sur
25       * ce transport. */
26
27      if (!(tcp = svctcp_create(RPC_ANYSOCK, 0, 0))) {
28            fprintf(stderr, "error: could not create the transport\n");
29            return (-1);
30      } else if (!svc_register(tcp, EXAMPLE_PROG, EXAMPLE_VERS1,
31        example_prog_1, IPPROTO_TCP)) {
32            fprintf(stderr, "error: could not register the service\n");
33            svc_destroy(tcp);
34            return (-1);
35      }
36
37      /* Gestion du signal d'arrêt (^C). */
38
39      signal(SIGINT, its_the_signal);
40
41      /* Lancement de l'application. */
42
43      if (setjmp(endjmp) == 0)
44            svc_run();
45
46      /* Libération de mémoire.
47       * Il n'y a malheureusement pas moyen de libérer les
48       * ressources allouées en interne par `svc_run()` à ce stade… */
49
50      svc_destroy(tcp);
51      return (0);
52 }
53
```

Les différentes fonctions appelées par le serveur sont d’ores et déjà prototypées par le fichier d’en-tête généré par rpcgen, il suffit donc de les définir. Ces fonctions ont toujours deux arguments, le premier étant toujours un pointeur vers l’argument fourni (s’il n’y a pas d’argument, le pointeur n’est pas à utiliser) et le second étant les données de la requête, dont le transport et l’adresse IP du client qui a fait la requête. La fonction doit retourner un pointeur sur l’instance de la réponse, ou `NULL` si la fonction a échoué à renvoyer une réponse pour une raison x ou y. Si la procédure n’est rien censée renvoyée, elle doit quand même retourner un pointeur valide, peu importe sur quoi puisque le résultat n’est pas utilisé.

Cette interface pour les fonctions n’est pas faite pour des données allouées dynamiquement : en effet, il faut le retourner pour que l’objet soit traité, et il n’y a pas de fonction permettant de définir une fonction pour libérer la ressource après que la réponse aie été retournée au client. Historiquement, la façon de retourner une réponse avec cette interface est de déclarer une variable statique dans la fonction pour retourner son adresse, puisque les variables statiques sont des emplacements valables même en dehors du contexte d’exécution de la fonction. Les variables statiques sont aujourd’hui passées de mode puisque dans un contexte de multi-threading, elles sont au pire la source de conflits d’accès en lecture/écriture (les fameuses « race conditions » décrites plus tôt dans ce document) et au mieux une

source de lenteur puisqu'un fil doit attendre que l'autre aie fini de l'utiliser pour pouvoir l'utiliser à son tour, mais comme notre serveur n'a qu'un seul fil d'exécution, leur usage est valable dans notre cas d'utilisation.

Par exemple ici, nos deux fonctions sont définies de la façon suivante :

```
 1 void *example_null_1_svc(void *argp, struct svc_req *req)
 2 {
 3       static char val[4] = "ok!";
 4
 5       (void)argp;
 6       (void)req;
 7       return ((void*)&val);
 8 }
 9
10 int *example_add_1_svc(add_params *params, struct svc_req *req)
11 {
12       static int result;
13
14       (void)req;
15       result = params->a + params->b;
16       return (&result);
17 }
```

Du côté du client, pour faire appel à la fonction d'addition, on utilisera là une fonction déclarée dans le fichier d'en-tête, mais cette fois, définie dans le fichier des définitions des fonctions clients généré par `rpcgen`. Dans un premier temps, nous créerons le client (type `CLIENT *`) puis nous l'utiliserons pour faire la requête :

```
 1 int add(int a, int b)
 2 {
 3       CLIENT *clnt;
 4       add_params params;
 5       int *ptr, ret;
 6
 7       clnt = clnt_create("localhost", EXAMPLE_PROG, EXAMPLE_VERS1, "tcp");
 8       if (!clnt)
 9             return (-1);
10
11       params.a = a;
12       params.b = b;
13       if (!(ptr = example_add_1(&params, clnt))) {
14             clnt_destroy(clnt);
15             return (-1);
16       }
17
18       ret = *ptr;
19       clnt_destroy(clnt);
20       return (ret);
21 }
```

2. Architecture de la bibliothèque client

Le protocole que j'ai posé est basé organise les opérations en objet pour la bibliothèque client, la *libwesh*, dont l'interface est en C++. Les ressources suivantes sont implémentées en tant que classes

dans la *libwesh* avec les appels correspondant dans le démon et suivent les types de ressources décrites dans l'introduction :

- Les serveurs WES, type nommé `wes_base` dans la *libwesh*, dont plusieurs types de ressources sont héritées :
  - Les serveurs WES réels, connectés par IP (IPv4 pour le moment), et dont le type dans la *libwesh* est `wes` ;
  - Les serveurs WES factices, présentant des données d'entraînement pour pouvoir développer l'interface client/démon sans nécessairement avoir un serveur WES sous la main, et dont le type dans la *libwesh* est `dummy_wes`.
- Les compteurs, type nommé `meter_base` dans la *libwesh*, associé à une ressource de type WES, dont plusieurs types de ressources sont héritées :
  - Les pinces ampèremétriques, dont le type dans la *libwesh* est `clamp` ;
  - Les compteurs d'impulsions, dont le type dans la *libwesh* est `pulse_meter` ;
  - Les compteurs reliés par téléinfo, dont le type dans la *libwesh* est `teleinfo_meter`.
- Les ressources de type capteur, type nommé `sensor_base` dans la *libwesh*, associé à une ressource de type WES, dont plusieurs types de ressources sont héritées :
  - Les capteurs reliés par le bus 1-Wire, dont le type dans la *libwesh* est `onewire_sensor`.

Les classes définissent également un ensemble de getters (méthodes dont le but est de récupérer la valeur d'une propriété) et de setters (méthodes dont le but est de définir la valeur d'une propriété).

Les objets sont récupérés, directement ou indirectement, depuis un objet de type `interface` qui gère la connexion au démon et sert de parent aux objets créés (qui y sont donc associés). Voici le diagramme de classe correspondant à l'interface de la bibliothèque :

Le type `WESH_CLIENT_TYPE` est un type qui n'est défini qu'à l'intérieur de la bibliothèque et correspond à l'objet utilisé pour communiquer avec l'interface RPC du démon (équivalent à l'objet `CLIENT` de la *libtirpc*) ; il est opaque à l'utilisateur (défini comme `void`).

Pour le reste, certains mécanismes n'ont pas été représentés entièrement dans le diagramme ci-dessus par souci de simplicité, dont notamment la gestion des parents et de leurs enfants. En effet, rien que pour la récupération de l'objet client nécessaire à la réalisation d'une requête au démon, depuis une instance de la classe `pulse_meter` par exemple, la méthode `get_clnt()` appelle la méthode du même nom de l'objet parent de cette instance, d'un type hérité de `wes_base` (par exemple `wes`), qui va elle-même chercher la méthode du même nom de l'interface parente ; d'où le fait que ces méthodes soient publiques et non réservées à la classe et à ses classes filles. Lors de la création d'un objet fils, lui est passé un pointeur vers son parent et l'identifiant qu'il utilisera à son niveau ; seulement, le pointeur vers le parent pourrait expirer lorsque celui-ci est déinitialisé, et l'on souhaite éviter que les objets fils appellent un objet déinitialisé ; il faut donc qu'ils soient prévenus lorsque cet évènement arrive.

Pour cela, au niveau des classes des objets pères, l'on tient une liste de pointeurs vers les différents objets fils à prévenir lorsque l'on est déinitialisés, avec laquelle les objets fils peuvent interagir en utilisant les méthodes du type `add_*_ref(type_objet_fils *fils)` et `del_*_ref(type_objet_fils *fils)` qui ajoutent et retirent des pointeurs à un set (liste où les éléments sont garantis d'être uniques). Lorsqu'un objet fils est déinitialisé et que son objet père ne l'est pas, il le prévient de ne plus le prévenir lorsqu'il est déinitialisé en retirant sa référence (`this`). Lorsque l'objet père est déinitialisé, il appelle les méthodes `orphan()` des objets fils pour leur dire qu'ils sont orphelins et qu'ils ne peuvent plus utiliser les méthodes de leur père (à eux de choisir s'ils soulèvent une exception ou ignorent cela après). Tout cela est transparent à l'utilisateur, car passe par les constructeurs et destructeurs principalement, et ne fait que soulever une exception s'il a déinitialisé le parent et qu'il tente d'utiliser le fils.

Voici un exemple d'utilisation pour récupérer le nom de la première pince ampèremétrique d'un serveur WES relié par IP :

```
1 wesh::interface iface;
2 wesh::wes wes = iface.get_wes("192.168.1.111");
3 wesh::clamp cl = wes.get_clamp_no(1);
4
5 std::cout << "Clamp #1 name: " << cl.get_name() << std::endl;
```

Dans ce code, les exceptions suivantes peuvent être soulevées :

- `wesh::connexion_exception` : la connexion au démon a échoué. Cela peut être dû au fait qu'il ne soit tout simplement pas lancé.
- `wesh::daemon_exception` : le démon a connu une erreur interne.

- `wesh::not_implemented_exception` : le démon n’a pas encore implémenté l’un des appels correspondants.
- `wesh::unavailable_wes_exception` : le serveur WES est injoignable.
- `wesh::no_meter_exception` : le compteur que l’on a essayé de récupérer n’existe pas.

Les compteurs ont, pour un serveur WES donné, un même espace d’identifiants, pour pouvoir tous les traiter d’une façon égale pour récupérer les éléments communs (ce qui est mesuré, etc).

La mise en place et l’assemblage de tous ces éléments relevant du domaine de l’opérationnel, passons donc au serveur Tonique.

III. Serveur Tonique

Je me suis également chargé, sur ce projet, du « serveur » du système d'échange Tonique. L'architecture du système d'échange est en réalité du producteur/consommateur avec messages de contrôle, pour définir le tarif ou d'autres options diverses.

Le rôle du serveur est de consommer les données envoyées par les centraux, contenant notamment les données des parties communes et les données de chaque appartement (récupérées depuis les serveurs WES correspondants), et d'envoyer les paquets de contrôle lorsqu'une modification est opérée dans la base de données. Les deux tournent en tant que service (tâche de fond) et fonctionnent sur une approche évènementielle : on surveille l'état de divers éléments du système, et si un évènement surveillé se produit, on a la marche à suivre dans ce cas.

Le serveur est plus facile à construire que le client notamment pour une histoire de temps et de régularité : autant le client doit faire des tâches régulières (en plus de tâches irrégulières comme exécuter les ordres donnés dans les paquets de contrôle), ce qui nécessite une certaine maîtrise du temps du client qui en plus est hors-ligne et n'a pas de serveur NTP à interroger (synchronisation grâce au modem GSM ? On ne nous a pas demandé d'aborder la question) et des timezones avec, difficulté supplémentaire puisque nous nous basons sur l'heure française pour découper les jours, gestion des changements d'heures : tous les jours ne font pas 24h, il y en a au moins un qui fait 23h et un autre qui fait 25h. Sans compter les décisions politiques qui peuvent avoir lieu pour ignorer tel ou tel jour, comme le 30 décembre 2011 qui n'a pas existé à Samoa. Cet article (en anglais) devrait vous donner une idée d'à quel point les humains ne savent pas gérer le temps, et d'à quel point le concept de tâches régulières en informatique est une tâche ardue.

Le serveur n'ayant pas de concept de tâches régulières, il peut se contenter de ne réagir qu'à deux types d'évènements assez simples à détecter :

- **Un message a été reçu d'un central.** Il suffit de, régulièrement (toutes les minutes par exemple), voir si quelque chose a été reçu pour chaque central, et de traiter les paquets de données parmi ce qui a été reçu pour intégrer les données à la base de données.
- **Une option a été mise à jour dans la base de données.** L'on peut par exemple imaginer qu'un tarif a été mis à jour pour un site donné, et que grâce à un trigger dans la base MySQL, dans une table contenant les ordres soit insérée une entrée disant d'envoyer un paquet de contrôle à tel site pour mettre à jour ses afficheurs.

Je m'attarderai sur le premier cas. La libtonique offrant une interface assez simple, il suffit, sur l'interface de la bibliothèque que nous utilisons, après avoir défini le numéro de téléphone du client (central) comme le numéro avec lequel nous souhaitons communiquer grâce à `set_default_number()`, d'appeler la méthode `recv_packet()` qui renvoie `true` si un paquet a été reçu et `false` sinon. Si un paquet a été reçu, les données qu'il contient ont été stockées dans les propriétés de l'interface, et l'on peut donc y accéder via des getters. On commence par accéder au type du paquet via la méthode `get_type()` qui peut donc renvoyer plusieurs choses :

- `pt_data_appart` s'il s'agit d'un message de données qui concerne un appartement.
- `pt_data_site` s'il s'agit d'un message de données qui concerne un site.
- Un autre type (e.g. message de contrôle), que nous ignorons car les autres types, nous les envoyons, nous ne les recevons pas ici.

On fait donc une boucle par client avec, dedans, une boucle par paquet reçu :

```
 1 for (auto it = clients.begin(); it != clients.end(); it++) {
 2       iface.set_default_number(it->second);
 3       while (iface.recv_packet()) switch (iface.get_type()) {
 4             case tonique::pt_data_appart:
 5                   /* Stockage des données de l'appartement. */
 6
 7                   // ...
 8                   break;
 9
10             case tonique::pt_data_site:
11                   /* Stockage des données des parties communes du site. */
12
13                   // ...
14                   break;
15
16             default:
17                   /* On se moque des autres types de paquets, on est le
18                    * serveur ici. */
19
20                   break;
21       }
22 }
```

Pour les deux cas des paquets de données, il faut extraire les données de l'interface puis faire une requête d'insertion avec. Voici un exemple pour les données correspondant à un appartement, avec le connecteur MySQL en C++ :

```
 1 std::stringstream dt;
 2 sql::Connection *conn = nullptr;
 3 sql::PreparedStatement *req = nullptr;
 4
 5 conn = get_driver_instance()->connect("tcp://[::1]:3306/usimarit",
 6       "usimarituser", "usimaritpw");
 7
 8 req = conn->prepareStatement("INSERT INTO consommation(con_appid, "
 9       "con_date, con_kVA_elec, con_m3_gaz, con_m3_efs, con_m3_ecs, "
10       "con_calorie, con_temp_min, con_temp_moy, con_temp_max) "
11       "VALUES(?, DATE(?), ?, ?, ?, ?, ?, ?, ?, ?)");
12
13 dt.str("");
14 dt << std::setfill('0') << iface.get_year() << "-";
15 dt << std::setw(2) << iface.get_month() << "-";
16 dt << std::setw(2) << iface.get_month_day();
17
18 req->setInt(1, iface.get_app());
19 req->setString(2, dt.str());
20 req->setDouble(3, iface.get_elec());
21 req->setDouble(4, iface.get_gaz());
```

```
22 req->setDouble(5, iface.get_efs());
23 req->setDouble(6, iface.get_temp_min());
24 req->setDouble(7, iface.get_temp_moy());
25 req->setDouble(8, iface.get_temp_max());
26
27 req->execute();
28
29 delete req;
30 delete conn;
```

La suppression manuelle de la requête et de la connexion sont importantes car sinon, les objets ne sont pas déinitialisés et cela peut poser problème pour la base de données. De plus, pour former une date en MySQL à partir d'une année, d'un mois et d'un jour du mois, il faut créer une chaîne du type `YYYY-MM-DD` et la passer à la fonction `DATE`. Par ailleurs, si vous ne saviez pas, il s'agit là du format de date international à utiliser absolument dans ce genre de situations, comme XKCD le décrit avec humour dans sa planche ci-contre (en anglais).

À partir de là, il suffit de mettre en place les triggers et une surveillance d'une table de messages de contrôle à envoyer dans la base de données, et le serveur est fait !

IV. Conclusion

En conclusion, avec ce projet, j’ai appris beaucoup de choses sur la gestion de l’énergie, autant par les particuliers que, du point de vue des clients, des fournisseurs, et j’ai pu expérimenter beaucoup de choses avec une machine que je connaissais pas, à savoir les serveurs WES de Cartelectronic.

Ce projet s’intègre avec les autres pour une connaissance plus approfondie et variée de l’informatique, grâce à ma curiosité qui m’a fait explorer ça en détail. Avec ce projet, je pense pouvoir obtenir le diplôme du BTS SNIR pour que mes compétences soient reconnues et que je puisse accéder à un métier intéressant dans le domaine de l’informatique via, si possible, une licence professionnelle en alternance.