GNU LINUX MAGAZINE / FRANCE

DÉVELOPPEMENT SUR SYSTÈMES UNIX, OPEN SOURCE & EMBARQUÉ

N°209
NOVEMBRE 2017

FRANCE MÉTRO. : 7,90 €
DOM/TOM : 8,50 €
BEL/LUX/PORT.CONT. : 8,90 €
CH : 13 CHF
CAN : 14 $CAD

L 19275 - 209 - F: 7,90 € - RD

Communication / Python

ENVOI - RECEPTION - AUTHENTIFICATION

MAÎTRISEZ LA GESTION AVANCÉE DE SMS

... SANS VOUS RUINER ! p.58

- Configurez votre modem USB 3G
- Envoyez des SMS avec Gammu
- Interfacez votre système avec Google Calendar
- Gérez la réception de SMS
- Mettez en place une authentification 2 facteurs

Microcontrôleur / STM32

DÉCOUVREZ L'ENVIRONNEMENT DE DÉVELOPPEMENT OPENSTM32 SUR STEVAL-3DP001 p.46

Web / Go

RÉALISEZ RAPIDEMENT UNE API REST EN GO p.76

Reverse Engineering / Désassemblage

ANALYSEZ STATIQUEMENT ET DYNAMIQUEMENT DES CODES AVEC CAPSTONE, UNICORN ET KEYSTONE p.92

InfluxDB / Grafana / Glances

AFFICHEZ VOS DONNÉES DE MONITORING DANS UNE INTERFACE DIGNE DE CE NOM p.36

Cybercriminalité / Vol de données

SOCIAL ENGINEERING : SCÉNARIO D'UN VOL DE FICHIERS CLIENTS p.06

ET AUSSI... IP VIRTUELLE - SYSTÈME DE FICHIERS AVEC GLUSTERFS - CODE PHASE-OUT

ikoula
HÉBERGEUR CLOUD

PRÉSENTE

CLOUD**IKOULA**ONE

Le succès est votre prochaine destination

MIAMI SINGAPOUR PARIS
AMSTERDAM FRANCFORT ___

*CLOUD**IKOULA**ONE* est une solution de Cloud public, privé et hybride qui vous permet de déployer **en 1 clic et en moins de 30 secondes** des machines virtuelles à travers le monde sur des infrastructures SSD haute performance.

www.ikoula.com sales@ikoula.com 01 84 01 02 50

ikoula
HÉBERGEUR CLOUD

NOM DE DOMAINE | HÉBERGEMENT WEB | SERVEUR VPS | SERVEUR DÉDIÉ | CLOUD PUBLIC | MESSAGERIE | STOCKAGE | CERTIFICATS SSL

GNU/Linux Magazine France
est édité par Les Éditions Diamond

10, Place de la Cathédrale - 68000 Colmar - France
Tél. : 03 67 10 00 20 – **Fax :** 03 67 10 00 21
E-mail : lecteurs@gnulinuxmag.com
Service commercial : abo@gnulinuxmag.com
Sites : www.gnulinuxmag.com – www.ed-diamond.com

Directeur de publication : Arnaud Metzler
Chef des rédactions : Denis Bodor
Rédacteur en chef : Tristan Colombo
Résponsable service infographie : Kathrin Scali
Réalisation graphique : Thomas Pichon
Responsable publicité : Valérie Fréchard,
Tél. : 03 67 10 00 27 – v.frechard@ed-diamond.com
Service abonnement : Tél. : 03 67 10 00 20
Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne

Distribution France : (uniquement pour les dépositaires de presse)
MLP Réassort : Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier. Tél. : 04 74 82 63 04

IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution, **N° ISSN :** 1291-78 34
Commission paritaire : K78 976

Périodicité : Mensuel
Prix de vente : 7,90 €

SUIVEZ-NOUS SUR :

https://www.facebook.com/editionsdiamond

@gnulinuxmag

p.41/42 DÉCOUVREZ TOUS NOS ABONNEMENTS MULTI-SUPPORTS !

ENCART JETÉ

LES ABONNEMENTS ET LES ANCIENS NUMÉROS SONT DISPONIBLES !

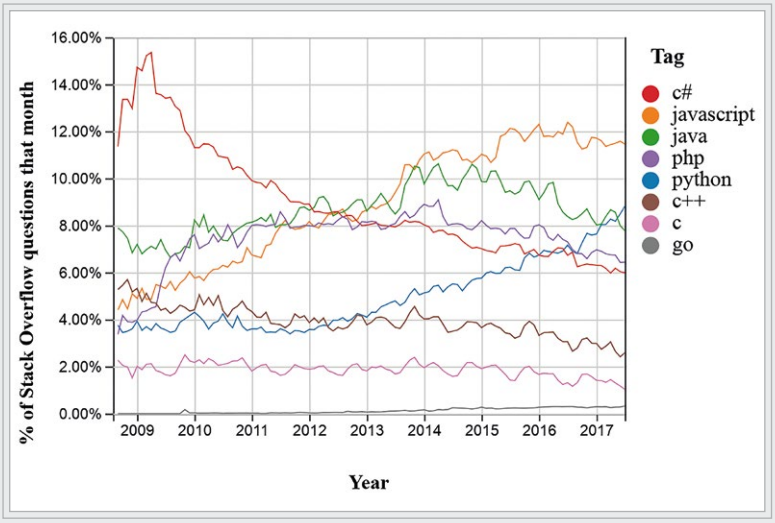
EN VERSION PAPIER ET PDF :
www.ed-diamond.com

Codes sources sur https://github.com/glmf

ÉDITO

Python, on aime ou on n'aime pas, mais le langage ne laisse pas indifférent. Il y a les fans, ceux qui ne jurent que par ce langage et qui vont jusqu'à l'extrémisme en oubliant tous les autres langages qui ne sont pas pour autant à jeter à la poubelle et il y a les opposants farouches, eux aussi sombrant souvent dans l'extrémisme et dénigrant un « pseudo-langage » qui n'est même pas compilé et qui envahit de plus en plus les magazines et les articles sur le net.

Python n'est certes pas compilé, mais il permet de développer rapidement, sans se préoccuper de la gestion de la mémoire. D'un autre côté, suivant les développements, son aspect de langage semi-interprété est pénalisant, car entraînant un gaspillage de mémoire et une lenteur des programmes. Alors, puisque nous ne pourrons départager les « pros » des « antis », essayons de comprendre avec les outils qui sont à notre disposition pourquoi on parle tant de Python. Utilisons par exemple les tendances de questions sur **StackOverflow** :



On peut constater que Python vient de dépasser **Java** et n'est plus devancé que par **JavaScript**. Ceci s'explique par le nombre de sites web contenant du JavaScript, mais montre que malgré les développements d'applications **Android** en Java, les questions sont plus nombreuses sur Python. Les mauvaises langues pourront bien sûr arguer du fait que s'il y a beaucoup de questions, c'est que le langage n'est pas clair... passons et retenons seulement la croissance imbattable sur les deux dernières années.

Pour confirmer cela, penchons-nous sur d'autres données. Sur **Google Trends** les recherches comme « python » ou « java » sont parasitées par d'autres significations que des langages de programmation mais le projet **PYPL** (*PopularitY of Programming Language* sur http://pypl.github.io/PYPL.html) l'utilise néanmoins en se basant sur les recherches de tutoriels. Je vous invite à consulter le site et à utiliser les langages qui vous intéressent, pour moi ce seront les mêmes que les précédents.

Cette fois Python ne dépasse pas Java, mais sa croissance de 9,5 % sur les dix dernières années est là encore remarquable et l'étude des données fournies par GitHub (http://githut.info/) fournira là encore la même tendance.

Ces informations montrent donc qu'il y a un engouement de plus en plus fort pour ce langage et qu'on le veuille ou non, il faudra le connaître un minimum pour comprendre, pouvoir interagir avec les futurs développements. Donc oui, Python est de plus en plus utilisé, et vous découvrirez même dans ce numéro comment vous en servir pour envoyer/recevoir des SMS, interagir avec **Google Calendar** et mettre en place une authentification deux facteurs, mais dans *GNU/Linux Magazine* nous n'en oublions pas pour autant les autres langages et vous pourrez ainsi trouver par exemple dans ce numéro également du **C** et du **Go** (oui, le bon dernier de toutes les tendances si l'on ne tient pas compte des langages « exotiques »).

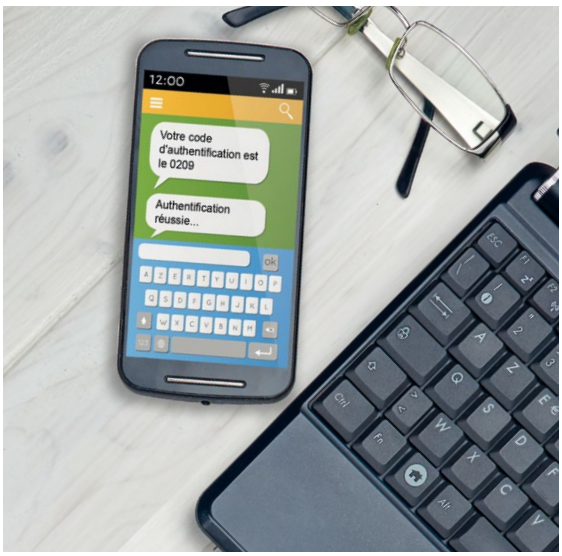
Surtout, n'oubliez pas que ce n'est pas parce qu'un langage est populaire qu'il est le meilleur. Le meilleur langage est celui le plus adapté à la tâche que vous devez réaliser ! Sur ce, bonne lecture !

Tristan Colombo

SOMMAIRE

GNU/LINUX MAGAZINE FRANCE

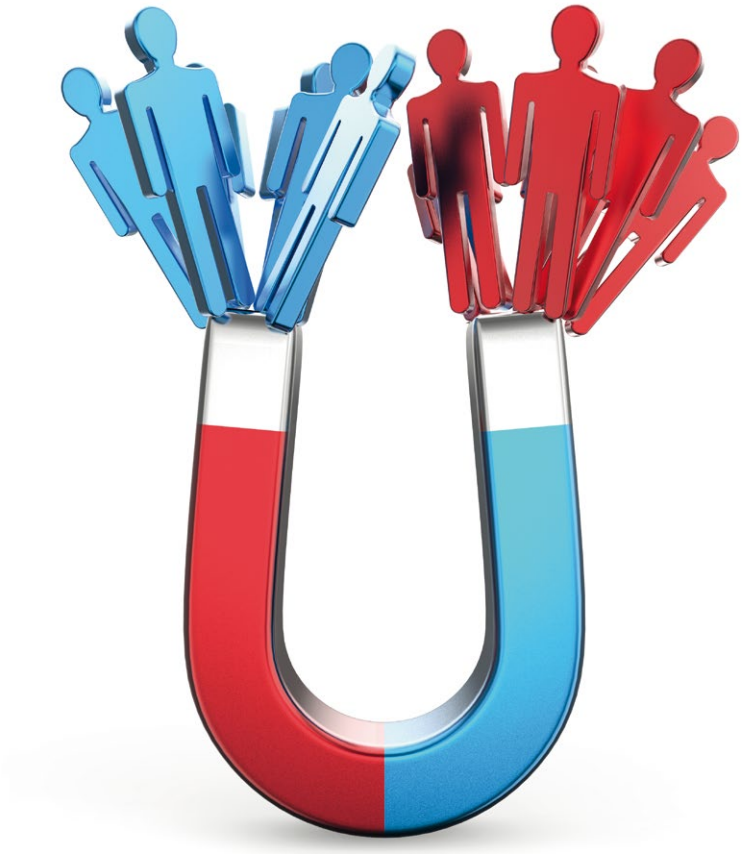
N°209



PETITE LEÇON FICTIVE DE SOCIAL ENGINEERING

GUILLAUME DUALÉ
[Administrateur système]

MOTS-CLÉS : SOCIAL ENGINEERING, CYBERCRIMINALITÉ



Le social engineering, ou l'ingénierie sociale en Français, est un ensemble de techniques qui permettent d'obtenir d'une personne ce que l'on souhaite. Cela peut s'apparenter à de la manipulation dont le but le plus fréquent est de soutirer des informations confidentielles à son interlocuteur ou de lui faire faire des actions à son insu.

Avertissement : Les personnages et les situations de ce récit étant purement fictifs, toute ressemblance avec des personnes ou des situations existantes ou ayant existé ne saurait être que fortuite. De plus, l'histoire racontée est hautement illégale (et immorale) et les faits relatés sont passibles de prison (Art. 323-x du Code pénal).

Un entrepreneur d'une société de fabrication de peinture perd des parts de marché alors que son concurrent direct, que nous appellerons **Painting-360**, a mis au point une nouvelle formule chimique de peinture blanche permettant de réduire les coûts de fabrication de 15%.

Il a alors deux objectifs : récupérer les secrets de fabrication de cette peinture et les fichiers clientèles de son concurrent.

Dans cet article, nous allons nous glisser dans la peau de cet entrepreneur peu moral pour décrire comment il procèderait avec quelques outils et des pratiques de *social engineering*.

1. PRISE D'INFORMATIONS SUR LA SOCIÉTÉ

La majorité des sociétés sont référencées dans les moteurs de recherche, car elles ont souvent un site officiel ou

sont présentées dans des sites spécialisés, nous devrions donc aisément trouver des informations telles que l'activité de l'entreprise, une adresse mail, des numéros de téléphone et de fax, ainsi que l'adresse de son siège.

Cependant, les informations trouvées ne sont pas suffisantes, il faut effectuer une recherche plus approfondie, avec des **Google Dorks [1]** par exemple, afin de trouver des traces d'adresses mail que des employés auraient pu laisser sur Internet. L'action est payante, dans un article traitant d'expériences chimiques, nous trouvons l'adresse mail professionnelle d'un scientifique de la société.

Cela nous permet de déduire la première information, les adresses mail internes sont composées de la sorte : **<pnom@painting-360.com>**.

Des recherches supplémentaires à l'aide de Google Dorks révèlent qu'un ancien site web de la société est encore présent dans un répertoire du site actuel. Sur ce site obsolète figurent plus d'informations et notamment les adresses des bureaux annexes. En continuant à parcourir ces anciennes pages, le menu ***Nos équipes*** permet de prendre connaissance des noms des neuf départements qui composent la société.

NOTE

Le site est hébergé dans un sous-répertoire du site actuel, mais il n'est pas accessible de manière classique via le site principal. C'est uniquement avec les Google Dorks que le site a pu être trouvé. Un webmaster consciencieux aurait vu avec la direction pour supprimer ce vieux site qui révèle beaucoup d'informations que visiblement l'entreprise ne souhaitait plus exposer.

La pêche aux informations continue, sur le site www.painting-360.com nous apprenons que la société est composée d'environ 50 salariés. C'est une information importante pour du social engineering, car il ne faut pas se comporter de la même manière dans une grosse boite de plus de 1000 personnes et dans une petite PME de 50 personnes où tout le monde se connaît ou presque. Dans le cas d'une tentative de social engineering dans une petite PME, si l'attaquant tente de se faire passer pour « le gars du support informatique », la cible risque de détecter la supercherie. En effet, habituée à son interlocuteur elle pourrait par exemple se rendre compte que les voix ou le comportement diffèrent.

Le second lot d'informations est organisationnel. Il y a donc un siège, un site pour la logistique et un site pour la production. La société est une PME composée de cinq départements pour environ 50 salariés.

Une autre recherche sur Internet nous apporte un complément d'information intéressant, leurs adresses mail peuvent aussi être formées de la manière : **prénom.nom@paiting-360.com**. Ce sont probablement des alias des boîtes aux lettres de type **pnom@painting-360.com**, ou l'inverse.

Sur le même site, se trouvent également le nom et le prénom d'une employée qui était le contact privilégié lors d'un salon dans lequel l'entreprise a tenu un stand, madame Stéphanie MONA. La persévérance permet finalement de révéler aussi le nom du directeur général du site de production, monsieur Roger BERNARD.

Les recherches web ont révélé d'intéressantes informations, mais il est possible d'aller un peu plus loin en utilisant le WHOIS du nom de domaine par exemple.

Dans le WHOIS du nom de domaine de la société, rien n'est caché. Nous apprenons que Mr Michel DUBOIS est le propriétaire. Son profil **LinkedIn** révèle qu'il a effectivement travaillé à Painting-360, mais qu'il est actuellement retraité. Une piste intéressante qui sera peut-être utile lors du social engineering.

Les informations récoltées lors de cette troisième phase sont plus précises, elles nous apportent des prénoms et noms qui pourront servir de points d'entrées ou pour légitimer des demandes.

Nous avons maintenant une idée assez précise de la structuration de l'entreprise et même quelques contacts, nous avons désormais un bon nombre d'informations pour commencer l'attaque.

NOTE

Chez la plupart des registraires de noms de domaine, il existe une option afin de masquer les informations personnelles du détenteur du nom de domaine. Cette fuite d'information aurait donc pu être évitée.

2. PHASE 1 - UNE PREMIÈRE PRISE DE CONTACT

Afin d'atteindre les deux objectifs fixés, récupérer la formule et le fichier client, nous allons utiliser un programme malveillant. Pour le construire, il est important de connaître le système d'exploitation et le client de messagerie utilisés par les employés de painting-360. Une possibilité simple pour obtenir ces informations est d'envoyer un mail à l'adresse de contact en posant une question. En effet, nous trouverons dans le code source du mail de réponse toutes les informations qui nous intéressent.

Voici un exemple :

Nous allons envoyer un mail sur l'adresse publique **contact@painting-360.com** avec une question anodine. La réponse importe peu du moment que nous en avons une, l'intérêt est d'afficher alors le code source du courrier électronique avec les en-têtes pour relever le User-Agent.

Voici un exemple de courrier pour notre cas :

```
Bonjour,
J'ai entendu dire du bien sur la qualité de vos
peintures et je souhaiterais savoir si vous vendiez
au grand public ou seulement aux entreprises.
En vous remerciant par avance pour votre réponse.
Bien cordialement,
Mr MARNET.
```

Au bout de quelques jours, une réponse arrive dans notre boîte aux lettres :

```
Bonjour,
Je suis au regret de vous annoncer que nous livrons
uniquement les professionnels.
En vous souhaitant bonne réception.
Cordialement,
Mme Isabelle DUPONT.
Chargé des relations clientèle.
```

Voici ce que l'User-Agent contient :

```
User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:31.0)
Gecko/20100101 Thunderbird/31.6.0
```

Nous savons maintenant qu'ils utilisent **Thunderbird** et que potentiellement tout le parc machine est sous **Windows 7**, s'ils ont un parc homogène.

À noter que nous avons aussi récupéré un nouveau contact avec ses fonctions, Mme DUPONT.

3. PHASE 2 - INSTALLATION DU MALWARE

L'objectif est d'amener Mme DUPONT à ouvrir la pièce jointe malicieuse d'un mail que nous allons lui envoyer. Si tout se passe bien, cette pièce jointe devrait installer une porte dérobée qui nous permettra de prendre le contrôle distant de sa machine de manière silencieuse (pour mémoire, c'est illégal...).

Une fois le contrôle pris, nous rechercherons des fichiers bureautiques sur sa machine en espérant trouver la formule et les fichiers clients. Le cas échéant, nous chercherons sur le réseau la présence de serveurs de fichiers et nous continuerons notre quête, même s'il est rare que cela aboutisse dès le premier essai.

3.1 Emballage du « paquet cadeau »

Sachant que la cible est un poste sous Windows 7, il est possible de construire grâce à une boîte à outils un malware qui permettra de prendre le contrôle de la machine. Il ne faut pas négliger les détails de construction ni hésiter à tester sur une machine virtuelle équipée du même OS le bon fonctionnement du malware. Il faut aussi s'assurer qu'avec un antivirus classique installé sur le poste, le malware ne se fasse pas attraper à la volée. Un peu d'obfuscation et de *packing*, puis un test sur virustotal.com : si tous les indicateurs passent au vert, le malware est prêt !

Pour dissimuler sa malveillance, le malware a une double extension **Plaquette.pdf.exe**. Avec un peu de chance, le poste est dans sa configuration par défaut et n'affiche pas les extensions. Bien entendu, l'exe affiche une icône de PDF, style **Adobe Acrobat**.

3.2 Le social engineering par mail adapté à la saison

Nous sommes bientôt en été, les vacances approchant, il est possible de créer un mail de type « promo de vacances » avec la fameuse pièce jointe pour tenter d'appâter la cible. Bien entendu, il faut forger l'adresse expéditrice pour qu'elle ressemble à quelque chose de probable, comme **nepasrepondre@balneoclub.fr**.

Voici un mail potentiel :

```
Sujet : Promotion sur les vacances en
Méditerranée
Bonjour cher client,
Il est temps pour vous de profiter de
vos vacances d'été 2017 pour prendre le
repos que vous méritez. Balnéo'Club c'est
un vaste choix de destinations à travers
toute la France.
[...]
<PHOTO ICI>
N'hésitez pas à consulter notre belle
brochure en pièce jointe de ce courrier.
Bien cordialement,
Mr le directeur Alphonse DUPUIS.
```

L'intégralité du mail n'y est pas, mais le principe est là. Bien entendu, à ce mail est attaché la pièce jointe frauduleuse.

3.3 Constat de l'échec de la première tentative

Malheureusement, cinq jours après l'envoi du mail, l'ordinateur n'a toujours pas été infecté. Plusieurs raisons peuvent l'expliquer :

- Mme DUPONT ayant déjà réservé ses vacances, elle n'a pas ouvert la pièce jointe.
- L'antivirus a détecté quelque chose.
- Mme DUPONT ne s'est pas fait prendre au piège, a considéré ceci comme un spam et a supprimé le message.
- Autres empêchements techniques divers.

Parmi ces raisons il y a donc un défaut de réaction de la part de Mme DUPONT, voyons comment améliorer cet axe.

3.4 Un second scénario de persuasion

La cible est toujours la même personne, Mme DUPONT, mais cette fois-ci nous allons nous faire passer pour un membre de l'entreprise afin d'augmenter les chances de réussite et nous allons l'interpeller directement à propos d'une question professionnelle liée à sa fonction de chargée de clientèle.

Étant donné qu'il y a trois sites, le plus discret n'est pas de se faire passer pour un collègue proche, cela serait un peu risqué, mais plutôt pour une personne d'un autre site, avec un peu de chance ils se connaissent moins et la supercherie pourra fonctionner.

De plus, afin d'effectuer un peu de pression psychologique, le mail envoyé va sembler venir d'une autorité, le directeur du site de production.

Voici un nouveau mail forgé en se faisant passer pour Roger BERNARD, le directeur du site de production.

```
Expéditeur : roger.bernard@painting-360.com
Sujet : Plaquette pour le stand du prochain salon
Bonjour Isabelle,
J'ai réalisé une plaquette pour le prochain
salon, peux-tu me dire si elle te convient?
Merci à toi,
Roger.
```

Il faut miser sur la simplicité du mail pour qu'il soit efficace, plus on donne de détails plus on risque de se tromper et d'éveiller la suspicion. De plus, le manque de détails permet d'éveiller la curiosité et pousse la victime à ouvrir la pièce jointe qui se dira certainement quelque chose du genre « mais de quoi il parle celui-là encore ? ».

Bien entendu, la pièce jointe est le malware préparé pour le précédent scénario. Cette fois-ci le résultat ne se fait pas (trop) attendre, environ 45 minutes après l'envoi du mail, Mme DUPONT ouvre la pièce jointe et la porte dérobée à sa machine.

Nous avons désormais un accès distant à sa machine, que la chasse aux fichiers commence !

4. PHASE 3 - RÉCUPÉRATION DES FICHIERS

Il faut maintenant agir vite, car nous ne pouvons pas savoir comment va réagir la victime, si elle va prévenir le directeur du site de production, si elle va parler de « ce mail étrange » à son service informatique, ni comment le service informatique va réagir. Vont-ils débrancher le poste du réseau ? Il y a plusieurs issues possibles et il faut se dépêcher pour éviter de perdre la main.

Une recherche dans tout son disque dur de fichiers bureautiques de type **Microsoft Office**, **LibreOffice**, **OpenOffice**, etc. donnera probablement de bons résultats. De même pour les fichiers de base de données bureautique, comme **Microsoft Access** et l'équivalent en version libre.

Le résultat du scan est plutôt bon, il y a environ 150 fichiers qu'il faut rapatrier au plus vite tant que nous avons la main sur la machine, l'analyse se fera plus tard.

Il est aisé de remarquer que des lecteurs réseau sont montés, **Active Directory** est bien pratique dans notre cas. Les lecteurs montés automatiquement à l'ouverture de session facilitent la vie des utilisateurs... et celle des pirates.

Le parcours du lecteur réseau s'avèrerait un peu long, car il y a beaucoup de dossiers et sous dossiers, et surtout pléthore de fichiers bureautiques qui semblent s'accumuler depuis plusieurs années sur ce serveur.

Il est donc préférable de lancer des recherches récursives sur des motifs précis tels que **`compta`**, **`client`** et **`contrat`** qui semblent être un bon point de départ. Après quelques dizaines de secondes, les résultats s'affichent à l'écran et parmi ceux-ci, il y a **`2016_Fichier_clients_PBE4.xls`**.

Cela semble bien parti pour l'un des deux objectifs, mais après rapatriement du fichier sur la machine locale, lors de l'ouverture, nous pouvons voir un pop-up demandant : « Veuillez saisir le mot de passe afin de déverrouiller le fichier ». C'est un peu frustrant !

NOTE

C'est une bonne chose qu'un fichier Excel contenant des informations importantes soit protégé par un mot de passe, cela prouve qu'ils ont quand même conscience que ces données sont sensibles. Cependant, si le serveur de fichier était correctement configuré, il n'aurait pas été possible d'accéder à ce fichier qui concerne la comptabilité depuis le poste de la personne chargée des relations clientèles.

Maintenant, il faut ouvrir ce fameux fichier : nous lançons une attaque locale par dictionnaire et brute-force. L'opération prit du temps, mais s'avéra payante, au bout de cinq jours le mot de passe (fort) est enfin trouvé : `PaintinG360ClienTS2016`.

Premier objectif atteint : avec ce mot de passe, nous avons enfin les coordonnées complètes de tous les clients de la société.

Pendant que le dictionnaire engrainait ses possibles mots de passe, l'épluchage des autres documents récupérés ne donnait pas grand-chose. Toujours pas de formule chimique, mais quelques noms d'employés et notamment celui de la personne chargée de l'informatique sur le site principal.

À ce stade, il va nous falloir un peu « d'aide » pour savoir où chercher la formule et souvent, il suffit de demander.

4.1 Étape de social engineering par téléphone

Pour tenter de soutirer des informations sur l'emplacement du fichier de la formule chimique, il est plus efficace de faire du social engineering par téléphone, mais cela demande aussi plus d'assurance et d'habileté.

Le scénario préétabli va être d'appeler l'accueil du site de production en se présentant comme Xavier MAULE et demander à parler à Roger BERNARD qui n'est pas à son bureau.

Au téléphone, la conversation ressemble à ceci :

- *Bonjour, c'est Xavier de l'informatique, est-ce que je pourrais parler à Mr BERNARD s'il vous plaît ? Je ne suis pas à mon bureau et je n'ai plus son numéro en tête.*
- *Oui, je vous le passe.*
- *Allo ?*
- *Oui bonjour Roger, c'est Xavier de l'informatique, vous allez bien ?*
- *Bien merci et vous ?*
- *Bien merci, dites-moi, je voulais confirmer quelque chose avec vous, les formules chimiques sont bien sur le PC de production, celui relié à la machine ? Car je ne trouve plus toutes les sauvegardes et je voudrais vérifier tout cela.*
- *On a seulement une copie des données ici pour le bon fonctionnement des machines, mais les fichiers originaux sont sur l'ordinateur de Bertrand de la R&D.*
- *Ah d'accord merci Roger, je vais voir avec lui pour m'assurer des bonnes sauvegardes, et sinon tout va bien chez vous ?*

Fin de conversation cordiale et classique.

NOTE

On remarquera que nous avons récupéré des informations précieuses durant cet appel, mais il y a eu un facteur chance tout de même, vu que Mr BERNARD, ne s'est aperçu de rien. Xavier et Roger ne doivent pas très bien se connaître dans la réalité, ce qui a permis de ne pas se faire démasquer. Si Mr BERNARD avait détecté la supercherie, il aurait fallu trouver une petite excuse rapide, du type « désolé j'ai dû me tromper de numéro » puis raccrocher et trouver un autre scénario.

Une nouvelle recherche sur LinkedIn est nécessaire pour obtenir des informations sur ce fameux Bertrand. La fiche de la société Painting-360 nous renvoie vers un seul Bertrand pour lequel nous trouvons d'autres détails sur cette personne et entre autres : sa photo, nom, prénom et ses anciens postes.

Une recherche **Google** sur son nom et prénom renvoie sur une page **Facebook**, la photo permettant de confirmer qu'il s'agit bien de la même personne. Il y a peu de données publiques, mais dans le peu qui est disponible, deux choses attirent l'attention : cette personne aime la pêche et a « liké » la page d'un magasin de pêche situé dans une ville voisine de celle du siège de l'entreprise.

La pêche au pêcheur est lancée, nous tentons la même approche avec un mail publicitaire sur la pêche, puis par mail professionnel, mais rien n'y fait. Afin d'éviter de nous faire repérer, nous n'insistons pas trop, il va falloir tenter une nouvelle méthode.

4.2 Nouveau scénario personnalisé

Nous allons envoyer par la poste à cette personne un courrier semblant provenir du magasin de pêche, avec à l'intérieur, une carte de visite préalablement récupérée dans la boutique, une clé USB et un courrier expliquant qu'il a été parrainé et qu'il gagne en cadeau une clé USB avec un logiciel de gestion des prises de pêche et des concours, ainsi que de la documentation sur les poissons de méditerranée. Bien entendu, la clé contient le même malware que pour Mme DUPONT avec auto exécution, mais aussi pour mettre en confiance la victime, les documentations d'ichtyologie appropriées.

Nous créons donc une enveloppe à bulles avec tous les éléments à l'intérieur et l'expédions par la poste à :

Société Painting-360

Mr Bertrand LEGER

Service Recherche et développement

96130 MIRUE

Au bout de 7 jours à patienter, le poste de Mr LEGER s'est connecté au serveur de contrôle, l'accès shell à sa machine est enfin opérationnel. Encore une fois il ne faut pas perdre de temps, lancer un scan des fichiers bureautique sur son ordinateur, rapatrier le tout et analyser la pêche.

Pas de chance, rien de probant concernant la formule chimique. Au vu des documents, cela semble pourtant être l'ordinateur de la bonne personne, les contenus sont typiquement de la R&D en chimie, mais la formule recherchée n'est pas récupérée. Le poisson continue à passer entre les mailles du filet.

Il faut élargir la recherche. Tout ne se limite pas aux logiciels de bureautique, peut-être existe-t-il des logiciels spécifiquement dédiés à la chimie ? Un parcours de la liste des programmes installés sur sa machine en révèle un grand nombre classiques tels que des drivers, des logiciels de chat et quelques jeux. Cependant, un outil sort du lot, il s'appelle **OpenBabel**. Google nous indique que c'est une boite à outils pour tout ce qui touche à la chimie.

La documentation officielle d'OpenBabel permet d'identifier la liste des extensions prises en compte. Nous modifions le filtre de recherche et nous repartons à la pêche. Le résultat est mince, neuf fichiers seulement, mais bon, c'est déjà mieux que rien.

Une fois ces fichiers rapatriés, nous installons OpenBabel et nous ouvrons ces fichiers un par un. Même si la chimie n'est pas notre domaine, nous comprenons que dans chaque fichier il y a un grand champ de prise de notes et un champ « titre de formule ». Dans le septième fichier, son champ de note contient un texte assez clair expliquant le gain de production sur la peinture blanche et le titre est « Formule finale pour mise en production » ... banco ! Le deuxième objectif est atteint.

Nous savons maintenant produire au même coût et en rognant un tout petit peu les marges (ce qui est facile, vu que nous n'avons pas investi dans la R&D), nous allons pouvoir démarcher les clients de la liste récupérée.

Il ne faut pas oublier d'effacer ses traces sur les machines que nous avons infectées afin d'être le plus discret possible. Si possible, il faudra également faire disparaître les mails envoyés aux victimes.

5. DEUX ANS PLUS TARD

Notre concurrent, la Société Painting-360 n'existe plus et nous sommes bien positionnés sur le marché. Par contre, il paraît que la société **Peinture nature** aurait développé une formule sans solvant et au même prix. Ils sont en train de gagner des parts de marché... à défaut d'avoir investi dans la R&D, il va peut-être falloir repartir à la pêche.

EN CONCLUSION DE CE DOSSIER

Avec ces techniques (illégales) de social engineering, le temps passé par l'usurpateur et le coût engagé peuvent être très largement rentables au regard des bénéfices potentiels obtenus lorsque les attaques fonctionnent et que les objectifs visés sont atteints.

A contrario, pour l'entreprise attaquée, le préjudice consécutif à des attaques réussies pouvant être vraiment important, il semble aujourd'hui primordial pour elles de se prémunir contre le social engineering.

Si le social engineering peut se révéler redoutablement efficace avec un peu d'habileté et de chance, il est grandement facilité lorsque les victimes sont peu méfiantes et coopératives.

La première mesure de protection est donc la sensibilisation. Il faut informer très régulièrement son personnel et encore mieux, l'entraîner. Il est possible de prévoir un petit budget pour des formations dédiées aux risques liés à la sécurité informatique et faire faire des campagnes de tests par un prestataire.

Le coût de cet investissement serait sans aucun doute bien inférieur au coût résultant d'une ou de plusieurs attaques réussies ! ■

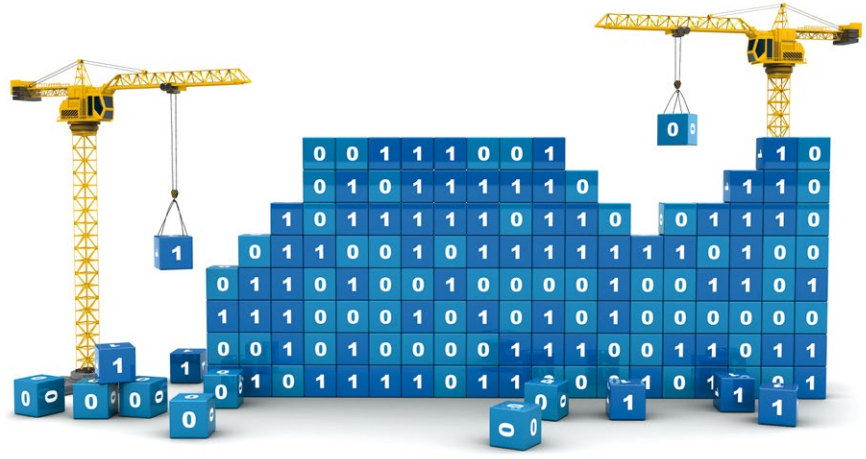
RÉFÉRENCE

[1] Découverte des Google Dorks : https://www.information-security.fr/decouverte-google-dorks/

LE CODE PHASE-OUT : L'AUTRE CODE BINAIRE TRONQUÉ

YANN GUIDON
["C'est pourtant pas compliqué"]

MOTS-CLÉS : COMPRESSION, CODAGE BINAIRE, CODES À LONGUEUR VARIABLE, CODE PHASE-IN, PHASE-OUT, OPTIMISATION



La compression est cet art fascinant de représenter les informations avec le moins de bits possible. Une des premières façons de compresser des données est d'éviter de gaspiller de la place à cause de représentations redondantes ou inutiles. Les codes binaires classiques sous-utilisent souvent l'espace de codage occupé, alors que les techniques de codage efficaces augmentent considérablement la complexité du code. Nous allons étudier un compromis, appelé Code Binaire Tronqué, et plus particulièrement une version peu connue : les codes phase-out.

Les codes phase-out sont dérivés des codes phase-in, qui m'ont été suggérés en 2003 par Steven Pigeon **[1]**, l'auteur d'une thèse (en français) sur le codage binaire **[2]** (voir en particulier la partie 5.3.4.2 « Codes phase-in »). Ils résolvent un petit souci rencontré lors de l'utilisation de codes binaires classiques : lorsqu'on connaît la limite que peut prendre un nombre, jusqu'à la moitié des codes peuvent être inutilisés. La figure 1 montre qu'en moyenne, un quart de l'espace de codage est perdu, ce qui représente jusqu'à un demi-bit par nombre.

1. QUELQUES REPÈRES

Pour situer le contexte, précisons d'abord que nous voulons créer un flux de données, qui représentent dans notre cas des signaux échantillonnés, tels que des sons ou des images. Chaque échantillon est transformé en une suite de bits, qui sera ensuite concaténée au flux binaire, afin d'être écrit dans un fichier ou transmis.

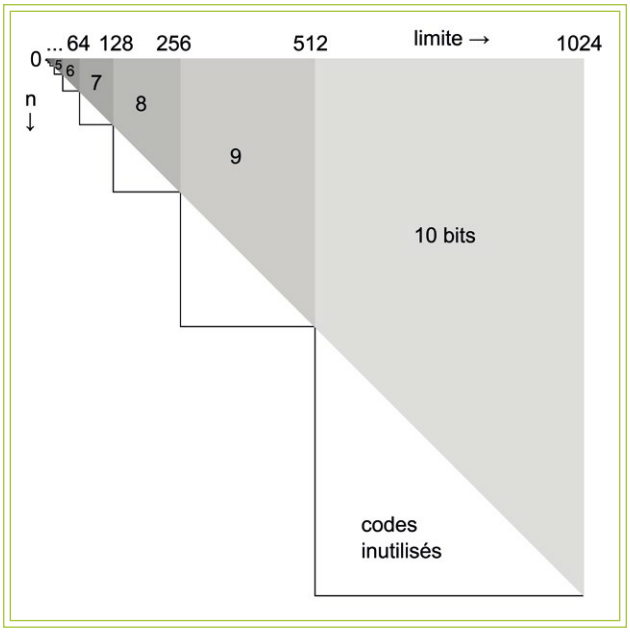


Fig. 1 : L'espace de codage inutilisé en codage binaire classique est représenté par les triangles blancs.

Le sport consiste ici à réduire la taille totale que ce flux de bits occupe, sans perdre d'information : chaque nombre encodé doit redonner le même en sortie après décodage. Il existe de nombreuses techniques avec des traitements plus ou moins compliqués, mais nous allons ici considérer que chaque nombre à coder (appelé **n**) est indépendant des voisins. On traite ainsi chaque échantillon sans se soucier du précédent ou du suivant, ce qui fait de **n** un nombre un peu abstrait, mais pour les besoins de l'expérience, c'est *toujours un entier supérieur ou égal à zéro*.

Là où l'expérience commence à devenir intéressante est que le décodage des bits de notre nombre **n** nécessite de connaître au préalable, d'une manière ou d'une autre, une information auxiliaire : la limite (appelée **L**) qui indique quelle valeur maximale notre nombre **n** peut prendre. On peut donc écrire l'inéquation suivante :

$$0 \leq n \leq L$$

L'art de l'entropiste (ou de l'entropologue ?) consiste donc à transmettre cette limite sans prendre plus de place que l'original, mais cet article se concentre sur comment ces deux valeurs **n** et **L** peuvent se combiner, pour que **n** utilise juste le nombre idéal de bits.

Un code binaire classique utilise **k** bits pour représenter un nombre entre 0 et 2^k-1. Par exemple, avec **k=4**, on peut représenter 2^4=16 symboles, qui sont le plus souvent associés aux nombres 0 à 15. Si notre limite est 15 alors notre nombre **n** utilisera 4 bits, ni plus ni moins. En règle générale, si la limite **L** est égale à 2^k-1, alors on utilise **k** bits. Inversement, quand on connaît **L**, alors on peut calculer le nombre de bits $k=\log2(L)$.

Mais une limite exactement égale à une puissance de deux est un cas assez rare. La plupart du temps, notre limite **L** se situera entre deux puissances de deux : $2^k-1 < L < 2^k$. Nous devons alors toujours allouer **k** bits, mais il restera 2^k-L codes inutilisés, des codes qui prendront une place virtuelle, une fraction de bit qui finit par peser sur l'ensemble des données. Cet espace de codage perdu est représenté par les triangles blancs de la figure 1, car un bit n'est pas divisible.

Il existe des techniques proches de l'idéal, telles que les codes arithmétiques **[3]** et le codage par intervalle **[4]** avec lesquels un symbole se retrouve mélangé aux précédents, ce qui affranchit les nombres de la limite arbitraire des bits individuels. L'inconvénient est que ce type de code est relativement complexe et délicat à mettre en œuvre. En particulier, les codes arithmétiques effectuent des calculs sur les entiers avec des multiplications, ce qui occupe des ressources matérielles conséquentes dans le cas d'une réalisation matérielle. Et comme je développe un CODEC destiné à utiliser le moins de portes logiques possible, ce type d'approche est écarté.

Les codes de Huffman (voir Wikipédia) sont une autre classe de codes binaires répandus, qui ne nécessitent pas de multiplication. Ils sont capables eux aussi de gérer des probabilités complexes, au moyen d'une table qu'il faut stocker et construire. Donc pour utiliser un code de Huffman classique, il faut d'abord collecter, encoder et reconstituer des statistiques et histogrammes, avant même de pouvoir commencer à transmettre des données utiles... Sinon il y a aussi les codes de Huffman adaptatifs ou dynamiques.

Nous allons explorer ici les codes *phase-in*, parfois aussi appelés *codes binaires tronqués* : c'est un cas spécial des codes de Huffman, à mi-chemin entre les codes arithmétiques et les codes binaires classiques. Comme les codes de Huffman, les codes phase-in ne se mélangent pas entre mots consécutifs, donc chaque mot binaire reste toujours bien distinct. Pour réduire la taille, les nombres sont représentés par une chaîne de bits de longueur variable alors que la longueur est fixe dans un codage binaire classique.

La simplicité est à la fois un avantage et un inconvénient. Le codage n'est pas optimal, car il ne modélise pas les probabilités, on suppose ainsi que tous les nombres sont équiprobables. En retour, l'algorithme ne nécessite pas de gérer des tables en mémoire, qui sont une source de complexité, de latence et de bugs. Le codage phase-in utilise juste quelques additions ou soustractions, ce qui promet des réalisations simples et très rapides, en logiciel comme en matériel.

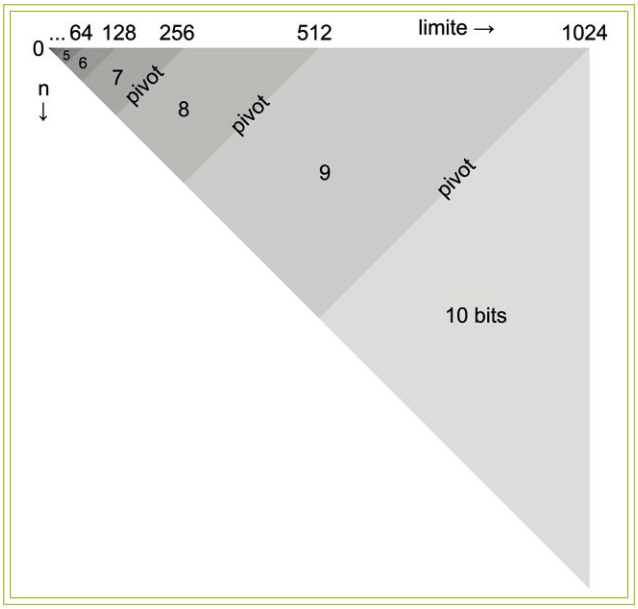


Fig. 2 : Utilisation des bits et taille progressive des mots avec un code phase-in.

La figure 2 montre la différence de comportement avec le code binaire classique. Il n'y a plus de changement brutal de la taille du code en fonction de la limite, au contraire la longueur du mot varie linéairement, suivant une diagonale, ce qui évite tout gâchis de code.

Comme on le voit sur la figure 2, les codes correspondant à un nombre sont divisés en deux zones, selon la limite :

- une moitié des codes utilise k bits ;
- l'autre moitié utilise k-1 bits.

Ces deux moitiés (qui ne sont pas forcément de taille égale) sont séparées par une valeur appelée ici *pivot*, qui permet de décider si le code binaire de **n** peut être raccourci d'un bit. Sur la figure 2, les pivots sont les lignes diagonales parallèles qui séparent deux zones grises contiguës.

Ainsi, dans le pire des cas, les codes phase-in n'augmentent pas le nombre de bits utilisés. Dans le meilleur des cas, un bit est économisé. Le gain n'est pas énorme en pratique, mais puisqu'il n'y a pas de risque de perte de place, et comme l'algorithme correspondant est très simple, il peut être ajouté sans risque à un système de compression. Sur un bloc entier de données, les fractions de bits ainsi économisées s'accumulent et peuvent grappiller approximativement (selon les données) quelques pourcents de la taille totale d'un fichier, avec peu d'efforts, donc c'est toujours bienvenu.

Comme tous les codes, ils ne peuvent être utilisés seuls, ils ont besoin d'un contexte, c'est-à-dire d'informations auxiliaires permettant leur décodage. Ils s'inscrivent idéalement dans un algorithme tel que 3R (présenté dans **[5][6][7]**) et tout système utilisant des nombres entiers positifs dont on connaît implicitement ou explicitement les bornes.

L'idée des codes phase-in semble avoir été cristallisée vers 1995, année de la publication des articles dénichés par Steven Pigeon (**[2]** p.104) :

[Les auteurs] Acharya et Já Já **[8][9]** *référèrent aux codes phase-in sous le nom de economy codes. C'est l'une des améliorations qu'ils proposent au codage des index dans l'algorithme de compression LZW.*

[...]

La documentation sur les codes phase-in et récursivement phase-in, outre les articles d'Acharya et Já Já, est essentiellement anecdotique.

Cela fait donc trois noms pour une même technique : la méthode de codage phase-in est aussi appelée « Code binaire tronqué » par Wikipédia **[10]**, mais nous utiliserons les termes *phase-in* et *phase-out*, car ils permettent de distinguer la sous-méthode spécifique.

2. EXEMPLE NUMÉRIQUE

Pour encoder et décoder les nombres, l'algorithme doit calculer d'abord le pivot, puis la valeur ajustée du code. Cela nécessite juste un peu d'arithmétique linéaire. Rien de bien compliqué, comme nous allons le voir avec un exemple simple où nous codons les nombres de 0 à 10.

Le nombre 10 est représenté en binaire par 1010, donc il faut quatre bits. Cependant 4 bits permettent de coder des nombres de 0 à 15 et les nombres de 11 à 15 ne sont pas utilisés, ce qui est une perte de 5 codes, soit près d'un tiers de l'espace de codage. D'un autre côté, 3 bits permettent de représenter seulement 8 codes (de 0 à 7), ce qui est insuffisant. Voici la table de codage en binaire classique :

```
Code  valeur
0000    0
0001    1
0010    2
0011    3
0100    4
0101    5
0110    6
0111    7
1000    8
1001    9
1010   10
1011    *
1100    *
1101    *
1110    *
1111    *
```

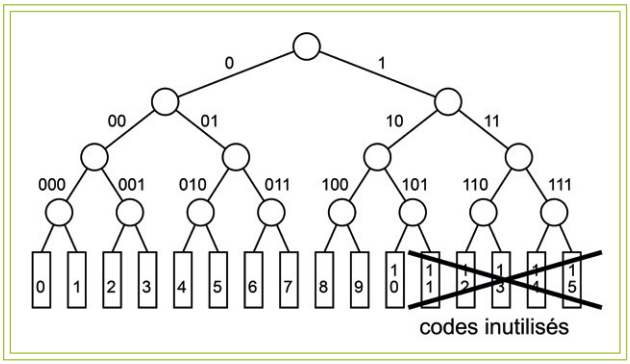


Fig. 3 : Les codes de 0 à 10 sont projetés sur un arbre binaire équilibré de profondeur 4, laissant 5 codes inutilisés.

Tout l'enjeu est alors de faire un mélange équilibré entre les deux tailles de mots binaires. La figure 3 représente la table ci-dessus sous forme d'un arbre binaire : les codes 11 à 15 prennent de la place que nous allons maintenant tenter de récupérer.

La figure 4 montre la première étape, où les codes sont d'abord décalés vers la fin. Ensuite, on déplace le premier élément vers un niveau supérieur de l'arborescence : le nombre 0 est donc représenté avec le code **000** au lieu de **0000**. En remontant ainsi, le code « perd » un bit, mais aussi occupe la place de deux codes plus longs, donc il ne reste que 4 codes perdus.

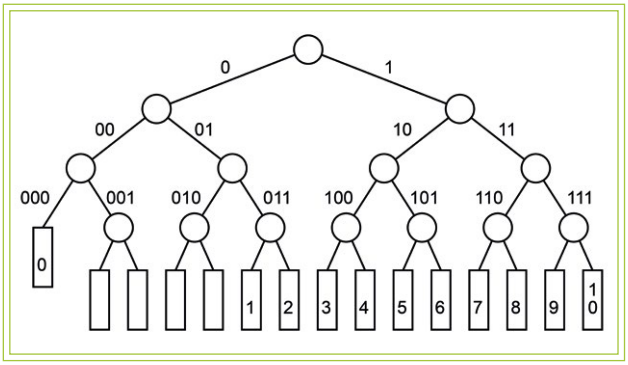


Fig. 4 : Décalage des codes et élévation du code 0.

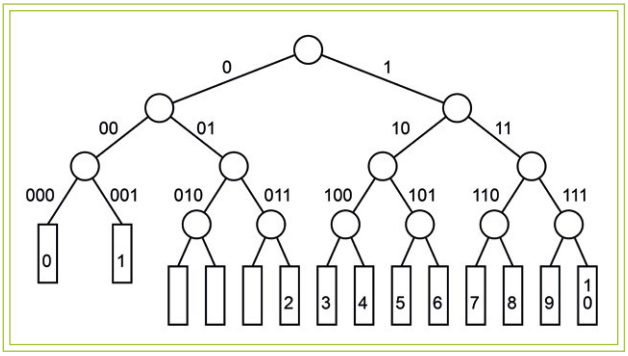


Fig. 5 : Élévation du code 1.

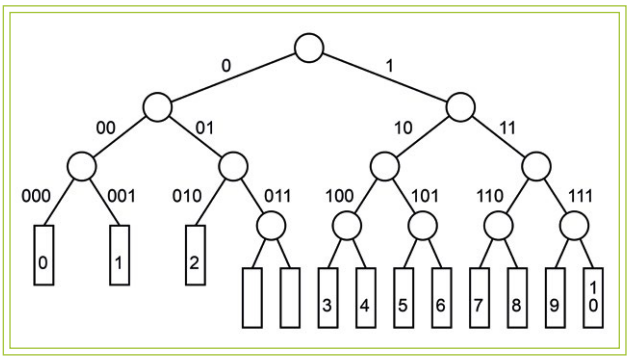


Fig. 6 : Élévation du code 2.

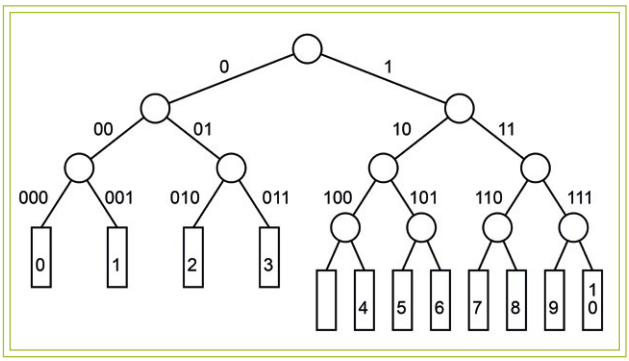


Fig. 7: Élévation du code 3.

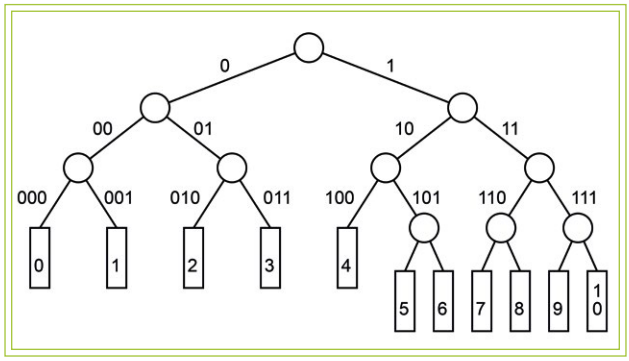


Fig. 8 : Élévation du code 4.

Les figures 5, 6, 7 et 8 répètent cette opération de substitution, déplaçant un code vers un nœud plus élevé et éliminant ainsi un code mort. L'arbre final de la figure 8 représente la table de codage suivante :

```
Code  valeur
000      0
001      1
010      2
011      3
100      4
 ------------ pivot
1010     5
1011     6
1100     7
1101     8
1110     9
1111    10
```

5 opérations d'élévation sont donc nécessaires pour compenser les codes morts (si on ne compte pas le décalage). Ainsi il y a autant de codes « promus » que de codes morts, c'est-à-dire 2^k-L. Cette valeur est donc notre fameux nombre *pivot* :

- en dessous de 2^k-L, on écrit le nombre directement avec $k-1$ bits ;
- sinon, on utilise k bits. Mais il faut encore ajuster le nombre et tenir compte du décalage, qui était de 2^k-L positions (le pivot, encore une fois).

L'algorithme d'encodage s'écrit donc facilement. Le dernier détail consiste à déterminer k, le nombre de bits nécessaires pour encoder la limite L, mais on va considérer pour l'instant que ce calcul est pris en charge par le reste du programme. Le pivot dépend de L donc son calcul peut aussi être factorisé si les conditions le permettent.

```
pivot = (1 << k ) - L;
bits = k;

if n < pivot then
   bits = bits - 1;
else
  n = n + pivot;
endif;

envoyer n avec k bits.
```

Le décodage est plus subtil, car on ne connaît pas à l'avance le nombre de bits à lire. L'approche sûre consiste à lire $k-1$ bits et, s'il s'avère que le nombre fait k bits, lire un bit supplémentaire. Mais c'est lent. Surtout s'il faut réaliser l'opération avec des circuits électroniques : il faut au moins deux cycles pour décoder un mot.

L'autre approche est un peu plus expéditive : on lit directement k bits. On décode le nombre obtenu, on l'ajuste s'il le faut, et éventuellement on « remet le bit en trop » dans le flux binaire. En pratique, cela revient à incrémenter un compteur, ou même sélectionner entre k et $k-1$.

Qui dit grande vitesse, dit risque de foncer dans les murs et il faut correctement traiter le cas particulier de la fin du flux : le décodeur est susceptible de lire un bit de trop, à la fin du flux de données. Il faut donc s'attendre à un dépassement, même s'il ne devrait avoir aucun effet. Mais à part cela, l'ajustement est toujours assez simple : on doit détecter si le nombre n (long de k bits) est inférieur au pivot.

L'autre subtilité est que le pivot doit être multiplié par deux puisqu'il s'applique maintenant à un nombre de $k-1$ bits. En effet, puisqu'on lit k bits, les codes courts ont été doublés :

```
Code  valeur
0000   \ 0
0001   /
0010   \ 1
0011   /
0100   \ 2
0101   /
0110   \ 3
0111   /
1000   \ 4
1001   /
 ------------- pivot
1010    5
1011    6
1100    7
1101    8
1110    9
1111   10
```

Le pivot n'est plus au code $101_2=5_{10}$, mais au code $1010_2=10_{10}$. Le décodage s'effectue donc presque aussi simplement que l'encodage :

```
pivot = (1 << k ) - L;
n = lire k bits;

if n < (pivot << 1) then
  renvoie le dernier bit dans le flux;
  n = n >> 1;
else
  n = n - pivot;
endif;
```

Cet algorithme est tellement simple, pourquoi donc s'en passer ? Une fois toutes ces petites subtilités traitées, le plus compliqué consiste à déterminer le nombre k.

3. INVERSER LES CODES

En jouant avec les codes phase-in en 2008, j'ai testé une variation, que j'ai nommée *phase-out*, puisqu'elle reprend le principe du phase-in, mais l'applique à l'envers. On retrouve cette idée à la fin du chapitre sur les codes récursivement phase-in de la thèse (**[2]** p. 105). La différence est simple : contrairement aux codes habituels qui allouent moins de bits pour les petits nombres, on va allouer moins de bits aux grands nombres. Bien que ce ne soit pas parfaitement juste, on peut le justifier ainsi :

- Les petits nombres prennent déjà peu de bits, donc on ne gagne pas grand-chose à enlever un bit.
- Par contre, si on suppose une probabilité uniforme, les grands nombres, par définition, sont beaucoup plus nombreux, donc plus probables, et enlever un bit a plus d'impact.

Steven Pigeon aborde ainsi la problématique dans sa thèse :

En fait, dans l'article d'Acharya et Já Já, les codes sont inversés dans la mesure où ce sont les grands nombres qui reçoivent les codes les plus courts. Cela est parfaitement raisonnable puisque dans l'algorithme LZW qu'ils se proposent d'améliorer, les index qui sont les plus susceptibles d'être utilisés correspondent aux dernières concordances trouvées, lesquelles viennent d'être ajoutées au dictionnaire et ont reçu par conséquent un index près de la fin du dictionnaire. On peut transformer trivialement les codes récursifs présentés ici en codes qui assignent des codes courts aux grands entiers : $C'_{\rho(N)}(i) = C_{\rho(N)}(N-1-i)$

Les tests préliminaires **[7]** ont montré que ce simple changement de perspective ou d'heuristique peut apporter un autre gain supplémentaire, marginal, mais pas insignifiant, et sans ajouter trop de complexité par rapport aux codes phase-in.

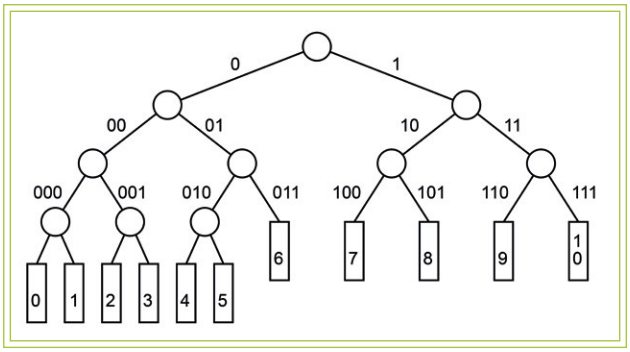


Figure 9 : Pour déterminer les codes promus à utiliser moins de bits, on reprend la figure 8 puis on applique un miroir vertical.

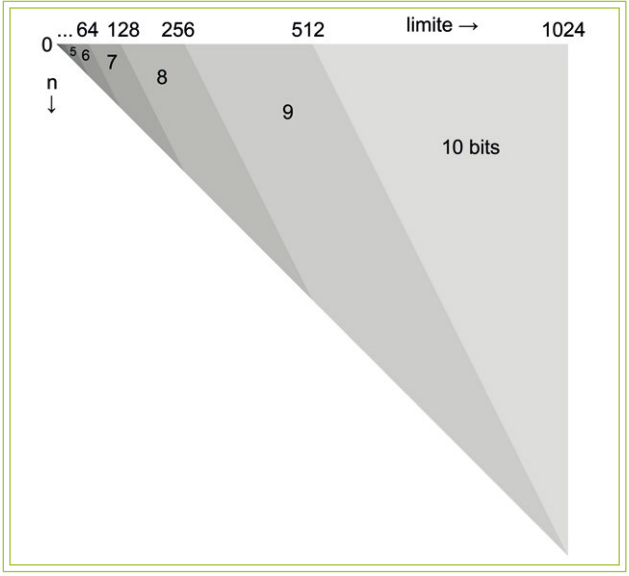


Figure 10 : Les codes phase-out sont un corollaire des codes phase-in, mais ils favorisent les nombres élevés.

La figure 9 doit être comparée à la figure 8 : la forme est identique, mais l'ordre des valeurs a été inversé. D'ailleurs la manière de réaliser le code phase-out est d'inverser le nombre avec la formule `N = N-1-i` donnée dans la citation précédente.

Cette inversion a un effet important sur la répartition des tailles, que l'on observe sur la figure 10 : la forme est différente de celle de la figure 2. Et comme les codes phase-in sont des codes à préfixes donc parfaitement réversibles, les codes phase-out le sont aussi.

Vous pouvez jouer avec le code grâce à la page interactive sur **Hackaday [11]** ou **GitHub**. Le projet **3RGB [12]** fournit aussi un autre fichier JS/HTML **[13]** qui montre en détail l'intégration du phase-out dans un flux de bits.

4. COMPARAISON DES PERFORMANCES

Le comportement des codes phase-in et phase-out dépend essentiellement des propriétés statistiques des données à encoder. Il est donc impossible a priori de prédire quelle version fonctionnera le mieux sans un minimum d'informations sur les données. Nous disposons déjà d'une heuristique :

- Si on traite surtout des nombres de faible amplitude (par rapport à la limite), alors on utilise le phase-in.
- Si on traite surtout des nombres de grande amplitude, alors on utilise le phase-out.

Évidemment, tout est relatif et les termes « faible amplitude » et « grande amplitude » changent selon le contexte et d'un bloc de données à l'autre... Donc le choix entre phase-in et phase-out dépend des propriétés et de la distribution des nombres.

Par exemple, on pense naturellement au codage phase-in pour compacter les résidus de prédictions (c'est-à-dire les erreurs par rapport aux valeurs calculées à partir des précédentes). Si les prédictions sont bonnes, donc si les données correspondent au modèle et aux heuristiques, alors les erreurs à encoder auront de faibles valeurs. Cela permet de gagner encore un peu de place dans le flux binaire. Mais en pratique, on ne peut pas savoir à l'avance, on peut juste faire des suppositions !

Une stratégie plus avancée consiste à tester les deux versions, valeur par valeur, puis indiquer par un bit quelle version est retenue. C'est la stratégie *in-out* qui a été testée en même temps que les deux versions sur **[7]**, mais le résultat est décevant. Pour indiquer quel encodage prend le moins de place, il faut ajouter un bit par valeur dans le flux. Puisque ce bit supplémentaire prend autant de place que le bit économisé, la stratégie in-out est mécaniquement toujours la plus mauvaise.

On peut appliquer la stratégie in-out sur des blocs de différentes longueurs, mais là encore, le compromis est difficile à trouver et n'est pas bénéfique puisqu'un avantage en chasse un autre. Transmettre la taille du bloc prend encore de la place que nous essayons justement d'économiser.

On se retrouve donc avec des blocs de taille fixe, qui sont à la merci des variations des données à traiter. Un bloc trop petit va ajouter un surcoût avec ce bit supplémentaire, alors qu'un bloc trop gros diluera les statistiques fines et réduira l'avantage d'une stratégie sur une autre. Il n'y a donc pas de taille de bloc idéale sans une bonne connaissance des données et des heuristiques fiables.

Les mesures effectuées dans le cadre de l'algorithme 3R mettent en lumière les transformations que les nombres subissent et le changement de contexte qui s'opère. En effet, l'algorithme 3R a comme raison d'être l'affinage des limites, ce qui augmente les chances d'obtenir des limites proches des nombres à encoder. Ainsi, 3R biaise les statistiques d'une manière qui favorise le phase-out, puisqu'il tente de rapprocher `L` de `n`.

Encore une fois, c'est la simplicité qui gagne : si on évite de se focaliser sur quelques cas particuliers, on traite les cas généraux moins mal. Tant qu'on ne doit pas encoder une information supplémentaire, on empêche ainsi les données de grossir dans le pire des cas (qui ne manque jamais d'arriver).

5. OPTIMISATION

La partie précédente concernait la performance de codage, ou plutôt le gain de place apporté par telle ou telle version. Maintenant nous allons nous occuper de la vitesse d'exécution, qui dépend de différents facteurs et surtout du contexte. Dans ce cas particulier, le développement du code répond aux objectifs et contraintes spécifiques suivants :

- Le décodeur est plus important que l'encodeur (pour l'instant), donc nous allons nous focaliser dessus.
- Le décodeur doit être réalisable en circuits logiques dans un FPGA, ce qui limite les opérations utilisables aux plus basiques : opérations booléennes et décalages. Les additions et soustractions doivent être limitées autant que possible à cause de la longueur des chaînes de retenue, qui sont la cause principale de lenteur d'un circuit.
- Le circuit logique doit contenir le moins de portes logiques possible, même si cela augmente légèrement le nombre de lignes de code pour la réalisation logicielle.
- Le CODEC utilise la version phase-out, car il est plus efficace que phase-in dans l'algorithme 3R.
- Les nombres à encoder en phase-out sont limités à 16 bits.

La version électronique est préparée et prototypée en logiciel, ce qui pourrait avoir des effets bénéfiques (comme ce fut le cas en optimisant l'algorithme 3R **[14]**).

Le fait d'utiliser phase-out avec 3R implique que la limite change à chaque nombre à coder et décoder, donc le pivot doit être recalculé à chaque fois, ce qui augmente la quantité de calculs et rend l'optimisation plus importante. Ici nous utiliserons essentiellement deux techniques : la parallélisation (réaliser plus d'une opération en même temps, en superscalaire ou avec des circuits parallèles) et la réduction du nombre d'opérations.

5.1 Le logarithme

La première étape consiste à déterminer combien de bits doivent être lus dans le flux. En termes mathématiques, on calcule le logarithme en base 2 de la limite : `k=log2(L)`. En termes informatiques, on compte simplement les bits :

```
k=0;
n=L;
while n != 0
  k = k+1;
  n = n>>1;
end while
```

Cette approche itérative est sympathique, mais fonctionne avec une complexité en *O(n)* et la vitesse d'exécution dépend de la taille du nombre, ce qui n'est pas très efficace. Quand on réalise l'algorithme 3R, on peut partir de la taille précédente et la diminuer puisqu'on sait qu'elle est inférieure ou égale : c'est un peu plus rapide, mais toujours aléatoire.

Le logarithme peut être calculé en *O(log n)* étapes et comme nous savons que le nombre de bits est limité à 16, il faudra au maximum $\log_2(16)=4$ itérations, que nous pouvons dérouler. En logiciel, un test supplémentaire court-circuite le tout si `L=0` :

```
k=0;

if L>0
  n=L;
  k=1;
  if (n & ~255) k+=8; n=n>>8; end if;
  if (n & ~ 15) k+=4; n=n>>4; end if;
  if (n & ~  3) k+=2; n=n>>2; end if;
  if (n & ~  1) k+=1; n=n>>1; end if;
end if;
```

Le lecteur attentif remarquera qu'on additionne des puissances de 2 à **k**, donc les 3 premières étapes peuvent être réalisées avec un **OU** logique au lieu d'une addition : en matériel, cela revient tout simplement à mettre des fils côte à côte pour former un bus. Le dernier **+1** est une simple incrémentation. Côté logiciel, on peut même remplacer la première addition par une assignation :

```
k=0;

if L>0
  n=L;
  k=1;
  if (n & ~255) k =9; n=L>>8; end if;
  if (n & ~ 15) k|=4; n=n>>4; end if;
  if (n & ~  3) k|=2; n=n>>2; end if;
  if (n & ~  1) k+=1; n=n>>1; end if;
end if;
```

Un circuit électronique sera plus efficace et rapide grâce à d'autres techniques, mais nous avons déjà une bonne idée de sa complexité. Les quelques petites modifications du code ci-dessus devraient rendre la vie un peu plus facile aux processeurs à exécution dans le désordre, tels les processeurs Intel actuels ou les derniers ARM.

5.2 Le masque

La partie suivante est une étape importante qui n'a pas encore été abordée, mais qui prend une nouvelle dimension quand on examine plus profondément la réalisation de l'algorithme. Il s'agit de créer un « masque » (appelé **m**), qui est un champ de bits contigus qui sont tous à **1**, dont la taille est égale à la limite : **m≥L>(m>>1)**. Comme tous les bits sont à **1**, on obtient aussi la relation booléenne **L&m=L**.

Ce masque a un premier usage : il permet de filtrer les bits que nous recevons du flux binaire, ce qui isole ceux qui nous intéressent.

```
n = lecture de k bits;
n &= m; // garde juste k bits de poids faible
```

Le masque a un autre usage grâce à une relation très intéressante : $m+1=2^k$, donc le masque sert à calculer le pivot !

En logiciel, il existe deux manières de calculer ce masque. La version en *O(log n)* est recommandée, car elle montre comment cela peut être réalisé en matériel. Cela consiste à effectuer un **OU** bit à bit avec tous les bits de poids fort, comme dans le code suivant :

```
m = L | (L >> 1);
m = m | (m >> 2);
m = m | (m >> 4);
m = m | (m >> 8);
```

Ainsi, le bit de poids faible sera un **OU** logique de tous les autres bits. Cependant, nous avons déjà calculé **k** avec un algorithme en *O(log n)* et la relation $m+1=2^k$ nous permet de calculer le masque d'un seul coup, pour réduire la quantité de code de la version logicielle :

```
m=(1<<k)-1;
```

Pour ce qui est de la version matérielle, le calcul du masque est déjà effectué par le circuit qui calcule le logarithme. Il a été étudié et présenté dans l'article qui détaille la conception de l'algorithme 3R **[14]**, nous faisons ainsi d'une pierre deux coups !

Dans **[15]**, la partie 2 décrit un encodeur de priorité qui utilise quasiment les mêmes idées, à la différence que nous cherchons la position du bit de poids fort au lieu du bit de poids faible. Le sens des bits est changé, mais on retrouve les mêmes opérations :

- D'abord, une cascade binaire de **OU** logiques, pour mettre à **1** tous les bits suivant le bit de poids fort. C'est une traduction directe du code à 4 étapes ci-dessus.

```
 le mot 0000111101011010
devient 0000111111111111
```

- Ensuite, un **XOR** ou un **ANDN** entre les bits consécutifs afin de déterminer où se trouve le bit de poids fort : un seul bit est à **1**. En logiciel, cela s'écrit en incrémentant le masque : l'instruction **m2 = m+1** propage la retenue et met tous les LSB à zéro, et ne garde plus qu'un bit à **1**. En matériel, le code équivalent est juste booléen : **m2 = m & ~(m>>1)** ce qui est facile à câbler, mais cela nécessite trois instructions de processeur au lieu d'une seule.

```
 le mot 0000111111111111
devient 0000100000000000
```

- Enfin, la position est encodée en binaire naturel au moyen de portes logiques **OU**. Pour un calcul sur **n** bits, chaque bit du résultat est à **1** une fois sur deux, donc chaque porte **OU** a **n/2** entrées. Pour des mots de 16 bits, il faut donc $Log_2(16)=4$ portes avec **16/2=8** entrées.

```
 le mot 0000100000000000
devient 1011
```

Le fait de disposer du masque très vite après la lecture de la limite permet de calculer rapidement le pivot. Encore mieux : si le masque est nul, le circuit peut sauter un cycle de lecture du flux de bits. C'est la fonction du signal **/skip** sur la figure 11.

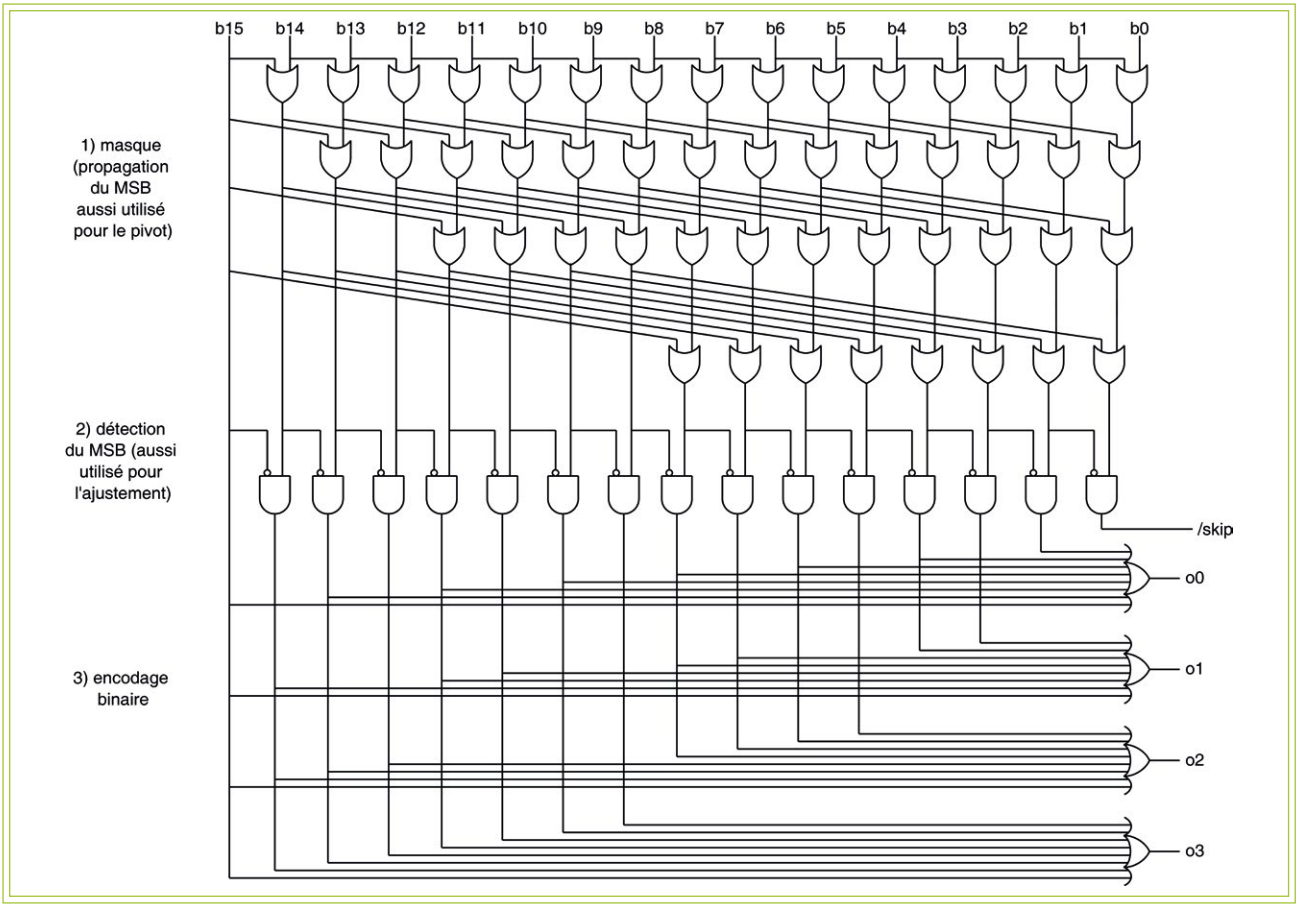


Figure 11 : Schéma d'un circuit de calcul du logarithme d'un nombre binaire. Les étapes intermédiaires génèrent aussi des résultats utiles pour les autres parties de l'algorithme.

5.3 Bernard

Si la limite n'est pas nulle, il faut encore calculer le pivot. C'est là qu'une relation binaire très heureuse apparaît, pour l'encodage comme pour le décodage. Je vous épargne les détails du pourquoi du comment, et je vous livre le résultat :

```
if (n > (((L<<1) & m) |1 )) then
   (ajustement)
endif;
```

En gros, l'utilisation du phase-out au lieu du phase-in a économisé (miraculeusement ?) une soustraction, puisque le calcul d'origine du pivot est `(1<<k)-L`. Où est passée la soustraction ?

Le tour de passe-passe se révèle quand on remarque que `1<<k` est égal à `m+1`, donc on peut réécrire le pivot ainsi : `(1<<k)-L=m+1-L`.

L'autre astuce utilise la représentation binaire en complément à deux, où `-N=(~N)+1`. Autrement dit, le calcul du complément revient à inverser tous les bits puis incrémenter. Mais même l'incrémentation est économisée puisque le pivot du phase-out est inversé par rapport au pivot du phase-in, donc le circuit équivalent ne comporte qu'une seule chaîne de retenue, celle de la comparaison entre `n` et la limite modifiée.

Le calcul du pivot peut encore être amélioré. D'abord on peut étudier l'expression `(((L<<1)&m)|1)` qui revient à effectuer une rotation de `L` sur `k` bits. Le bit de poids fort de `L` est par définition toujours à 1, et il se retrouve effacé par le décalage à gauche puis le masquage par `m`. Mais ce bit est de nouveau réécrit dans le bit de poids faible… Malheureusement, il n'est pas évident d'effectuer une telle rotation plus efficacement donc il faut trouver une autre méthode.

L'autre chose à remarquer est que le bit à `1` réinjecté dans le bit de poids faible est une constante, ce qui n'affecte pas la comparaison. `L` est décalé à gauche, mais on pourrait tout aussi bien décaler `n` à droite, ce qui économiserait ainsi une opération `OR` avec la constante. D'autre part, l'ajustement (voir plus loin) utilise le masque décalé à droite `(m>>1)`, ce qui est une valeur qu'on peut réutiliser pour le calcul du pivot. La condition de pivot s'écrit donc ainsi :

DISPONIBLE LE 3 NOVEMBRE

GNU/LINUX MAGAZINE HORS-SÉRIE N°93 !

SÉCURISEZ VOTRE INFRA-STRUCTURE LINUX !

NE LE MANQUEZ PAS

CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

https://www.ed-diamond.com

```
if (n>>1) > (L&(m>>1)) then
  (ajustement)
endif;
```

La comparaison s'effectue habituellement au moyen d'une soustraction (c'est-à-dire une addition modifiée), donc au niveau électronique, c'est la seule opération qui va consommer des portes logiques et du temps. Le masquage ne consomme presque rien et les décalages logiciels se traduisent en ignorant le bit de poids faible, donc on ne compare que 15 bits (au lieu de 16).

5.4 L'ajustement

On y est presque ! Le dernier morceau de code s'occupe de modifier **n** lorsqu'il dépasse le pivot.

Dans le cas de l'envoi comme de la réception, la taille du mot doit être décrémentée. Par contre, l'envoi diffère de la réception par deux détails :

- L'ajustement de **n** se fait par une addition à l'envoi, et une soustraction à la réception (puisque l'un est la réciproque de l'autre). La formule magique à l'encodage est **n=(n+(m>>1))-L** donc le décodage s'effectue avec **n=(n+L)-(m>>1)**.
- À la réception, avant l'ajustement, il ne faut pas oublier d'enlever le bit qui a été spéculativement lu (sinon le résultat n'aura pas de sens). Le décodage devient ainsi **n=((n>>1)+L)-(m>>1)**.

Encore une fois, il existe une astuce qui permet de simplifier le circuit logique correspondant puisque **m** est directement lié à **L**. Pour l'encodage, en appliquant de nouveau la relation **-N=(~N)+1**, l'expression **(m>>1)-L** peut être réécrite ainsi : **(m>>1)+(1+ ~L)**. Cette deuxième expression est composée de **(m>>1)+1**, que nous savons déjà équivalente à **1<<k** (à un décalage fixe près, ce qui ne consomme pas de porte logique). Donc même si la représentation logicielle contient une addition et une soustraction, le circuit logique ne réalise qu'une addition **(n + ~L)**. Le masque n'est pas utilisé directement, on réutilise le deuxième étage du circuit de logarithme (où un seul bit est à **1** pour indiquer le MSB de la limite) pour ensuite faire un XOR de **~L**. En fait tout cela revient à enlever le MSB de **L**, et l'inversion de ses bits finit de calculer l'expression du pivot 2^k**-L**.

On peut aussi en profiter pour effectuer quelques simplifications supplémentaires. Par exemple, dans le cas de l'encodage, seule l'expression **(m>>1)** est utilisée (**m** n'est pas utilisé seul) alors qu'on calcule **m** au moyen de l'expression **1<<k** (moins un). On peut économiser les décalages de **m** en calculant le logarithme afin de retourner **k-1** au lieu de **k**. En propageant cette modification, le code source de l'encodeur se retrouve alors apparemment plus clair, même si c'est trompeur. Afin d'éviter les confusions, la valeur **(m>>1)** est appelée **m1**.

6. LE CODE COMPLET

Tous ces éléments algorithmiques sont assemblés pour former une fonction d'encodage et une fonction de décodage réciproque. Le code source a été développé en JavaScript et traduit ici en langage C.

L'encodage a l'air assez simple, reprenant tous les morceaux discutés précédemment :

```
// encodage :
int send_phaseout(unsigned int val, unsigned
int lim) {
  uint32_t k=0, l=lim, m1; // m1 = m >> 1

  if (lim) {
    // génère m avec lim, démarre avec k=0
    if (l & ~255) { k =8; l=lim>>8; }
    if (l & ~ 15) { k+=4; l>>=4; }
    if (l & ~  3) { k+=2; l>>=2; }
    if (l & ~  1) { k+=1; l>>=1; }
    m1=(1<<k)-1;

    if ( (val>>1) > (lim & m1) ) // lim sans MSB
      val+=m1-lim;
    else
      k++;

    send_bits(val, k); // à voir plus tard
  }

  return k; // cette valeur peut être utile
}
```

Le seul point nouveau est la fonction **send_bits()** que nous devrons étudier dans un prochain article. Le décodage est similaire, mais diffère par la méthode de calcul du masque, qui commence avec **k=1**.

```
// décodage :
unsigned int receive_phaseout(unsigned int lim) {
  uint32_t
    val=0,    // valeur par défaut si retour
immédiat
    val1,     // val >> 1
    k,        // nombre de bits
    l=lim,    // limite temporaire
    m,
    m1;       // m >> 1
```

```
  if (lim) {
    // génère le masque, mais départ avec k=1
    k=1;
    if (l & ~255) { k =9; l=lim>>8; }
    if (l & ~ 15) { k+=4; l>>=4; }
    if (l & ~  3) { k+=2; l>>=2; }
    if (l & ~  1) { k+=1; l>>=1; }
    m = (1<<k)-1;
    m1 = m>>1;
    val =  receive_bits(k) & m;  // à voir
plus tard
    val1 = val>>1;

    if (val1 > (lim & m1)) {
      val = (val1+lim)-m1;
      adjust_bits_queue(); // à voir plus tard
    }
  }

  return val;
}
```

On retrouve les mêmes éléments qu'à l'encodage, mais l'algorithme nécessite **val** et **mask** ainsi que leurs valeurs décalées d'une position à droite, ce qui donne l'impression que le code est plus lourd. Les fonctions **receive_bits()** et **adjust_bits_queue()** seront étudiées en même temps que **send_bits()** dans un prochain article.

CONCLUSION

Les codes binaires tronqués sont un outil supplémentaire dans la grande boîte à outils des entropistes. Ils ne permettent pas de gagner beaucoup de place, mais complémentent avec peu d'efforts des algorithmes existants. En supposant des limites et des nombres aléatoires, on peut gagner environ un quart de bit par nombre, ce qui fait un octet économisé pour 32 nombres. Un bloc de 64 pixels monochromes perd ainsi environ deux octets, mais il ne faut pas oublier que la gestion de la limite requiert aussi des informations auxiliaires...

Ce n'est pas la panacée, mais ce petit gain est d'autant plus notable qu'il ne nécessite aucun calcul compliqué et ignore toute considération de statistique, de distribution ou de contexte par exemple. Ce codage est idéal, entre autres, pour améliorer l'algorithme de compaction 3R, dans lequel il s'intègre facilement.

Dans un prochain article, je présenterai d'ailleurs comment le codage phase-out s'inscrit dans des fonctions d'insertion et d'extraction de flux binaire. L'article suivant rebondira sur le sujet, car tout cela permet de construire les codes sigma-alpha, qui vont encore améliorer l'algorithme 3R et d'autres choses assez surprenantes... ■

RÉFÉRENCES

[1] PIGEON S., page personnelle : http://stevenpigeon.org/

[2] PIGEON S., *« Contributions à la compression de données »*, thèse de l'Université de Montréal, décembre 2001 : http://www.stevenpigeon.org/Publications/publications/phd.pdf

[3] Wikipédia, *« Codage Arithmetique »*, https://fr.wikipedia.org/wiki/Codage_arithmétique

[4] Wikipédia, *« Codage Par Intervalle »*, https://fr.wikipedia.org/wiki/Codage_par_intervalle
Voir aussi la version anglaise : Wikipedia, *« Range Coding »*, https://en.wikipedia.org/wiki/Range_encoding

[5] GUIDON Y., *« Compactez une suite de nombres avec peu d'efforts grâce à l'algorithme 3R »*, *Open Silicium* n°16, octobre 2015, https://connect.ed-diamond.com/Open-Silicium/OS-016/Compactez-une-suite-de-nombres-avec-peu-d-efforts-grace-a-l-algorithme-3R

[6] GUIDON Y., *« Data compression: the 3R algorithm »*, octobre 2003, http://ygdes.com/ddj-3r/ddj-3r_compact.html

[7] GUIDON Y., *« Lossless data compression with the Recursive Range Reduction algorithm »*, 2006, http://ygdes.com/3r/

[8] ACHARYA (Tinku) et Já (Joseph Já), *« Enhancing LZW Coding Using a Variable-Length Binary Encoding. »* Rapport technique n° TR-95-70, *Institute for Systems Research*, 1995

[9] Acharya (Tinku) et Já (Joseph Já), *« An Online Variable Length Binary Encoding of Text. »*, *Informatics & Computer Science*, vol. 94, 1996, pp. 1–22

[10] Wikipedia, *« Codage binaire tronqué »*, https://fr.wikipedia.org/wiki/Codage_binaire_tronqué

[11] GUIDON Y., Démonstration interactive et graphique du codage phase-out https://cdn.hackaday.io/files/248341062497856/test-phase-out.html (voir aussi sur le compte GitHub du magazine)

[12] GUIDON Y., *« Le projet 3RGB »*, https://hackaday.io/project/24834-3rgb-image-lossless-compression-format

[13] GUIDON Y., Banc de test pour l'insertion et l'extraction de codes phase-out dans un flux de bits https://cdn.hackaday.io/files/248341062497856/exercise-phase-out.html (voir aussi sur le compte GitHub du magazine)

[14] GUIDON Y., *« Optimisation de l'algorithme de décompression de flux 3R »*, *Open Silicium* n°17, janvier 2016, https://connect.ed-diamond.com/Open-Silicium/OS-017/Optimisation-de-l-algorithme-de-decompression-de-flux-3R

[15] GUIDON Y., *« Décompressez un flux de données 3R avec un circuit écrit en VHDL »*, *Open Silicium* n°18, avril 2016, https://connect.ed-diamond.com/Open-Silicium/OS-018/Decompressez-un-flux-de-donnees-3R-avec-un-circuit-ecrit-en-VHDL

UN SYSTÈME DE FICHIERS HAUTE DISPONIBILITÉ AVEC GLUSTERFS !

JULIEN MOROT
[Sysadmin passionné par les logiciels libres depuis 1998]

MOTS-CLÉS : GLUSTERFS, HAUTE DISPONIBILITÉ, SYSTÈME DE FICHIERS RÉSEAU, NAS

GlusterFS est un système de fichiers réseau client/serveur permettant d'agréger différents nœuds de stockage afin de fournir un environnement NAS hautement disponible.

1. PRÉSENTATION

1.1 Pour quoi faire ?

Admettons que j'ai une application web lambda, je vais pouvoir déployer plusieurs instances **Apache** ou **Nginx** qui se trouveront derrière un équilibreur de charge, lui-même hautement disponible. Sur chaque instance de serveur web, il me sera facile de déployer l'application. Toutefois chaque instance aura besoin d'accéder à des fichiers communs, générés ou non par l'application. Bien souvent, je vais rencontrer dans ce cas un serveur NFS qui va donc lui-même constituer un point de faiblesse dans l'architecture.

Gluster permet de mettre en cluster plusieurs nœuds de stockage (a minima deux), ce qui permet de répondre à deux problématiques majeures dès qu'une application a besoin de pouvoir monter en charge : la parallélisation et la réplication du stockage. Pour fournir ces fonctionnalités sur un volume, une « brick » en langage Gluster, le système s'appuie sur des systèmes de fichiers traditionnels, XFS ou EXT4 au-dessus d'un périphérique en mode bloc (partition, LVM, RAID, etc.). Gluster travaille donc principalement au niveau fichier.

Contrairement à un certain nombre d'autres systèmes de fichiers de ce type, Gluster offre l'immense avantage de ne pas nécessiter de serveur de métadonnées pour fonctionner. De fait, cette absence ne constitue pas un point de faiblesse ou un élément supplémentaire à maintenir dans l'infrastructure de stockage. De plus, chaque fois que l'on ajoute un nœud au cluster, le système devient plus performant et l'augmentation de la performance est linéaire avec l'extension de l'infrastructure. Dernier point pour mettre en évidence cette simplicité de conception, il n'existe pas de notion de maître ou d'esclave avec **GlusterFS**.

1.2 Les volumes GlusterFS

Un volume est une agrégation de plusieurs bricks réparties sur différents nœuds de stockage. Le choix du type de volume se fait en fonction des attentes de performances, de sécurité ou de la combinaison des deux. Voyons les types de volumes standards, sachant qu'il existe des modes géorépliqués, strippés ou basés sur l'*erasure coding* pour des *workflows* spécifiques.

1.2.1 Volume distribué

Ce mode est le mode par défaut de GlusterFS. Les fichiers sont répartis sur l'ensemble des bricks du volume sans redondance aucune. Par conséquent, lors de la perte d'un nœud, les données de celui-ci sont perdues et il faudra se baser sur des mécanismes complémentaires pour assurer la reprise après incident. La volumétrie utile est celle de l'ensemble des nœuds du cluster. Ce mode permet une croissance aisée de la volumétrie en ajoutant simplement des nœuds au volume. Il faut donc au minimum deux nœuds, et la distribution peut se faire sur autant de nœuds du cluster (voir figure 1).

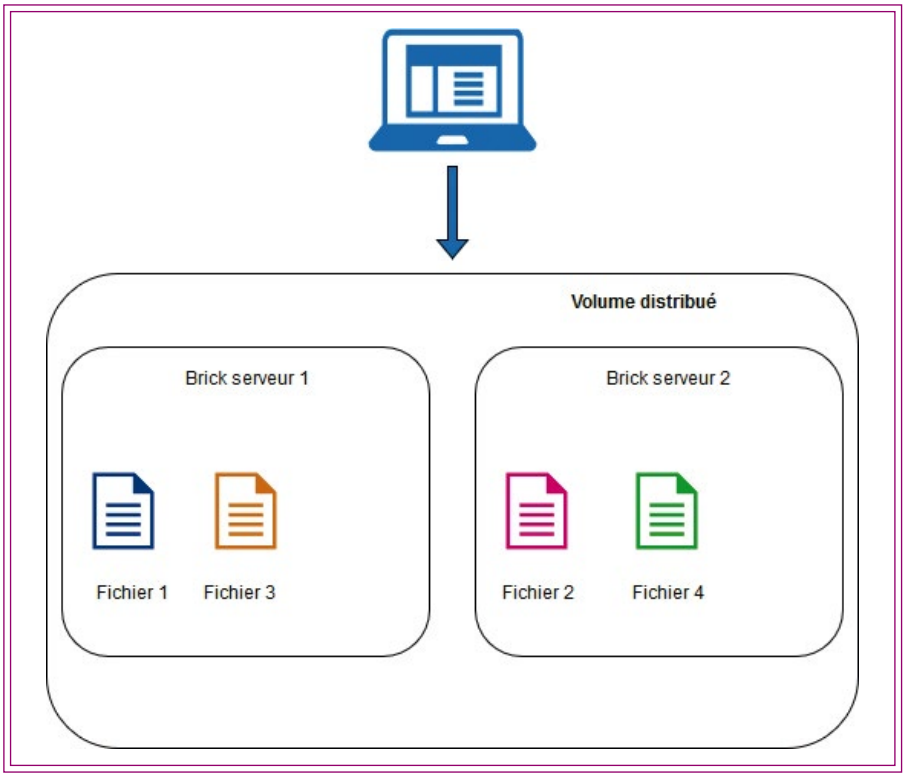


Fig. 1 : Volume distribué.

1.2.2 Volume répliqué

Ce mode permet de répondre au problème de la sécurité de la donnée posé par le mode distribué. Dans ce mode opératoire, le système maintient *n* copies de chaque fichier au sein des « bricks » spécifiées. Il faut donc autant de nœuds au cluster que de réplicas désirés. De la même façon que sur du RAID1, la volumétrie utile est la moitié de la volumétrie allouée (voir figure 2).

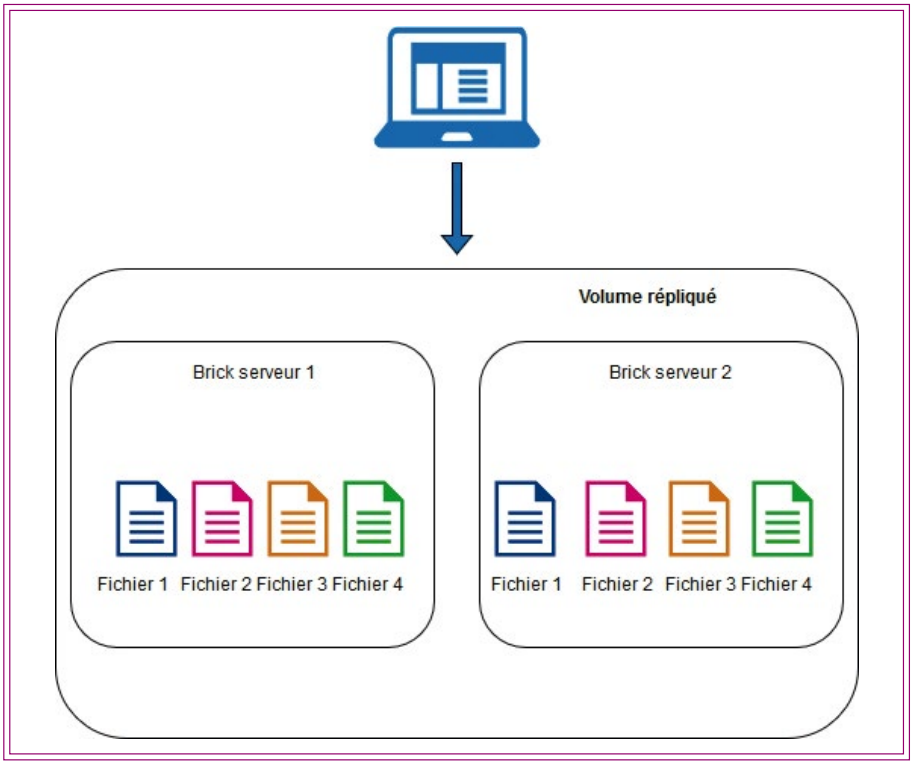


Fig. 2 : Volume répliqué.

1.2.3 Volume distribué répliqué

Vous l'aurez compris, ce mode est une combinaison des deux modes précédents. Cela permet de traiter des *workflows*

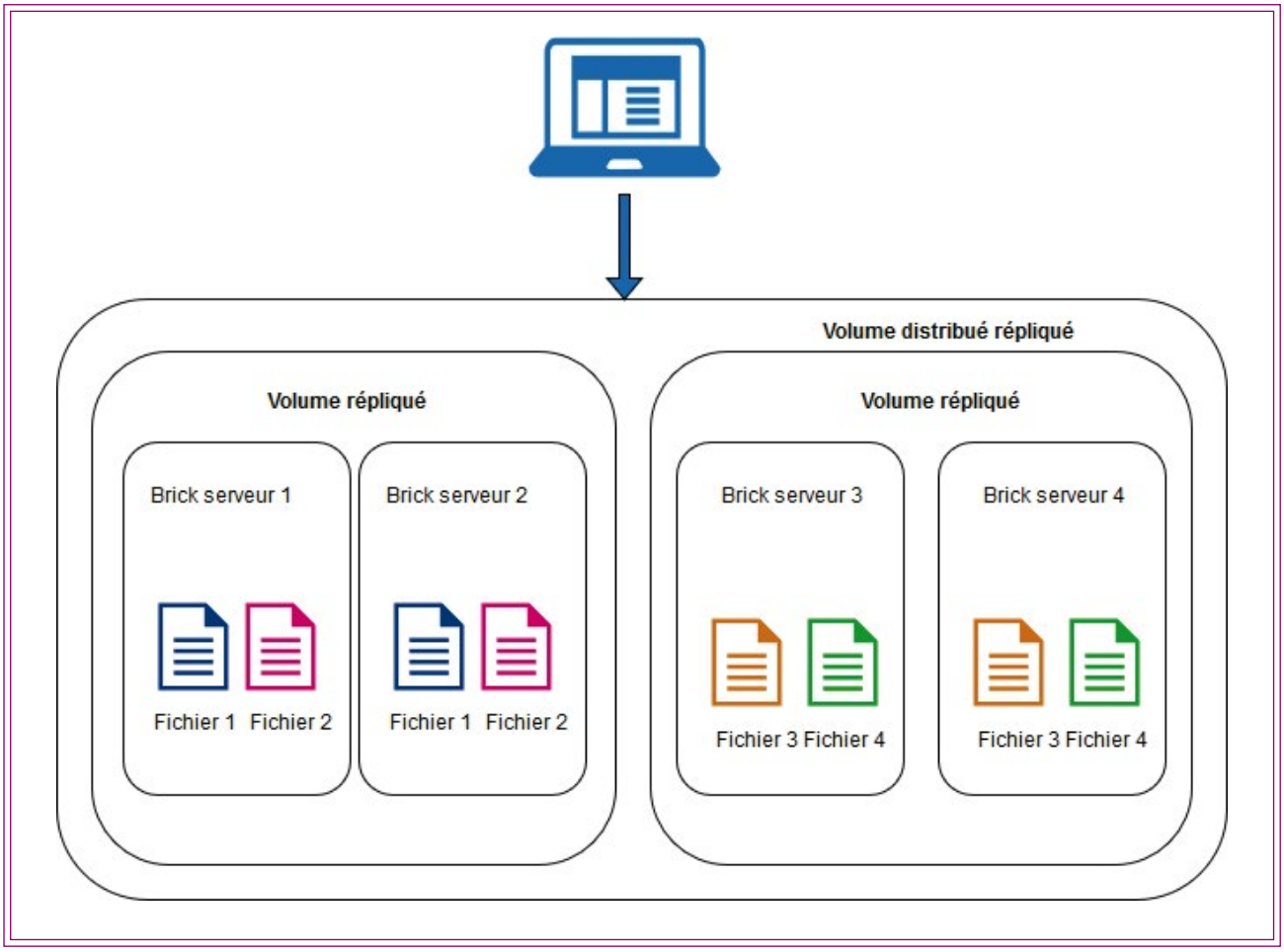


Fig. 3 : Volume distribué répliqué.

nécessitant disponibilité et capacité à monter en charge. Le nombre de bricks nécessaires est un multiple du niveau de réplication attendu. De plus, la réplication entre les bricks est définie par leur ordre de déclaration à la création du volume. Pour quatre bricks avec deux réplicas, les deux premières bricks répliquent ensemble et de même pour les deux suivantes (voir figure 3). Si nous souhaitions quatre réplicas, il nous faudrait donc huit bricks et les quatre premières répliqueraient entre elles.

2. UN PEU DE PRATIQUE

2.1 L'environnement

GlusterFS est assez agnostique par rapport à l'environnement et à la distribution. Pour ma part, les démonstrations suivantes seront toutes réalisées sous **Ubuntu 16.04**. Un point indispensable étant que les nœuds soient capables de discuter par leur nom, qu'il soit résolu par DNS ou par le fichier **hosts**, mais pas par adresse IP. Pour la suite, je partirai sur deux VMs **stor0** et **stor1** afin de monter un cluster à deux nœuds.

Chaque serveur dispose d'un second disque virtuel de 10 Gio pour la démonstration. Le périphérique doit par contre impérativement disposer d'un système de fichiers supportant les attributs étendus, ext4 ou XFS sachant qu'XFS est de loin le système de fichiers recommandé. La convention de nommage veut, mais n'impose pas, que les données soient placées dans **/data/glusterfs/volume/brick**. La création des volumes peut se faire simplement comme ceci :

```
$ apt-get -y install lvm2 acl attr
xfsprogs
$(echo o; echo n; echo p; echo 1; echo ;
echo; echo t; echo 8E; echo w) | fdisk
/dev/sdb
$ pvcreate  /dev/sdb1
```

```
$ vgcreate VG-Brick0 /dev/sdb1
$ lvcreate  -l 100%VG -n LV-Brick0 VG-Brick0
$ mkfs.xfs -i size=512 -L Brick0 /dev/VG-Brick0/LV-Brick0
$ mkdir -p /data/glusterfs/vol0/
$ echo "/dev/VG-Brick0/LV-Brick0 /data/glusterfs/vol0/ xfs defaults 1 2" >> /etc/fstab
$ mount /data/glusterfs/vol0/
$ mkdir /data/glusterfs/vol0/brick0
```

Le LVM n'est pas obligatoire, mais s'y tenir permet d'avoir les bons réflexes pour de la production. Pour le reste, l'installation des *packages* est très simple, cette simple commande suffit :

```
$ apt-get -y install glusterfs-server
```

Par souci de simplification, aucun pare-feu n'est activé sur les différentes machines. Point de vigilance, il ne faudra pas cloner les machines avec le disque de données supplémentaire.

2.2 Le trusted pool

Avant d'être en mesure de gérer des volumes de stockage, les membres d'un cluster GlusterFS doivent se reconnaître entre eux et faire partie d'un même *trusted pool*. Tant que cette opération n'est pas réalisée, il n'est pas possible pour un hôte de joindre le réseau de stockage. Pour cela, c'est très simple, il suffit depuis un nœud de sonder avec la commande **`gluster peer probe`** et d'ajouter les autres nœuds :

```
root@stor1:~# gluster peer probe stor0
peer probe: success.
```

Pour vérifier :

```
root@stor1:~# gluster peer status
Number of Peers: 1

Hostname: stor0
Uuid: a920b020-9e5a-46f6-b073-1cc8ec00ba0e
State: Peer in Cluster (Connected)
```

2.3 Un volume répliqué

On va poursuivre notre itinéraire au sein de GlusterFS en créant un volume répliqué à deux nœuds. J'ai donné en introduction un exemple basé sur des attentes de haute disponibilité du stockage, il me semble pertinent de poursuivre sur cet exemple qui parlera sans doute davantage. Notre cluster ayant deux nœuds, avec conservation de deux copies, cela nous fait donc un système en miroir. Sur chaque serveur, on indique le dossier dans lequel se trouvent les données. Par sécurité, il est préconisé de créer le volume dans un sous-répertoire du point de montage afin qu'en cas d'échec de montage du volume, cela n'ait pas d'incidence sur la réplication gluster. Du fait du risque d'avoir un dossier vide sur un membre du cluster lors du démarrage des services, le comportement ne serait pas forcément prévisible.

```
root@stor1:~# gluster volume create repl-vol replica 2 transport tcp stor0:/data/glusterfs/vol0/brick0/ stor1:/data/glusterfs/vol0/brick0/
volume create: repl-vol: success: please start the volume to access data
root@stor1:~# gluster volume start repl-vol
volume start: repl-vol: success
```

On peut donc vérifier que tout est en ordre avec la commande ci-dessous. Le volume doit être marqué comme *online* sur l'ensemble des nœuds :

```
root@stor1:~# gluster volume status

Status of volume: repl-vol
Gluster process                                 TCP Port  RDMA Port  Online  Pid
------------------------------------------------------------------------------
Brick stor0:/data/glusterfs/vol0/brick0         49152     0          Y       2536
Brick stor1:/data/glusterfs/vol0/brick0         49152     0          Y       2338
NFS Server on localhost                         2049      0          Y       2359
Self-heal Daemon on localhost                   N/A       N/A        Y       2364
NFS Server on stor0                             2049      0          Y       2557
Self-heal Daemon on stor0                       N/A       N/A        Y       2562

Task Status of Volume repl-vol
------------------------------------------------------------------------------
There are no active volume tasks
```

2.4 Connexion d'un client

Il existe trois mécanismes d'accès principaux côté client :

- le client natif accédé au travers de FUSE, le système permettant de créer des pilotes de *filesystem* au niveau *userland*. Il suffit pour cela d'installer les *packages* nécessaires.
- via NFS, Gluster implémentant nativement le support NFS. Si vous avez été vigilant lors de l'installation du *package* **glusterfs-server**, vous avez sûrement remarqué certaines dépendances. Le serveur NFS n'est pas activé par défaut cependant.
- en CIFS, avec un serveur **Samba**.

Dans les deux derniers cas, il est souhaitable d'associer les serveurs à un système de type CTDB pour fournir de la haute disponibilité. NFS et Samba ne savent en effet pas tirer parti de l'ensemble des fonctionnalités contrairement au client natif. Connectons donc un premier client :

```
$ apt-get -y install glusterfs-client
$ mkdir /data
$ mount -t glusterfs stor1:/repl-vol /data
```

Créons un fichier aléatoire avec par exemple la commande ci-dessous.

```
$ dd if=/dev/urandom of=/data/toto bs=1024 count=10240
```

Pour confirmer que la réplication est fonctionnelle, il suffit de vérifier avec une simple commande **ls** que le fichier est présent sur les bricks de chacun des deux serveurs GlusterFS :

```
$ ls -l /data/glusterfs/vol0/brick0/toto
-rw-r--r-- 2 root root 10485760 sept.  9 19:58 /data/
glusterfs/vol0/brick0/toto
```

Un point qui a dû vous surprendre est la commande de montage. On a en effet explicitement spécifié l'un des serveurs alors que l'on est censé avoir déployé un stockage hautement disponible. En pratique, le client natif **glusterfs** ne fait que récupérer lors de la commande de **mount** les informations de configuration du cluster. Il communiquera directement avec l'ensemble des serveurs définis dans les *volfile* (dans le répertoire **/var/lib/glusterd/vols/repl-vol** sur les nœuds de stockage). Un bon moyen de vérifier est d'arrêter le nœud vers lequel on a réalisé le montage (un **halt -p** sur **stor1** dans ce cas) : le client doit continuer à fonctionner. Côté client, la perte de connexion doit être visible dans le fichier **/var/log/glusterfs/data.log**.

```
[2017-09-09 18:01:18.933835] W [socket.c:588:__socket_
rwv] 0-glusterfs: readv on 192.168.69.61:24007 failed
(Aucune donnée disponible)
[2017-09-09 18:01:37.954070] W [socket.c:588:__socket_
rwv] 0-repl-vol-client-1: readv on 192.168.69.61:49152
failed (Connexion terminée par expiration du délai
d'attente)
```

Un point que vous aurez noté également, c'est que la bascule n'est pas immédiate. En pratique, le délai est de 42 secondes. Pour ramener ce délai à une valeur plus raisonnable de 5 secondes, modifions notre nœud comme suit :

```
root@stor1:~# gluster volume set repl-vol
network.ping-timeout 5
volume set: success
```

Ce changement est tracé dans le log **/var/log/glusterfs/glustershd.log** avec une ligne par nœud comme celle-ci :

```
[2017-09-09 18:33:19.591108] I [rpc-clnt.c:1823:rpc_
clnt_reconfig] 0-repl-vol-client-0: changing ping
timeout to 5 (from 42)
```

2.5 Un brin de sécurité

Jusqu'ici, on a pu monter le volume simplement en contactant l'un des serveurs du *pool* GlusterFS, mais aucune sécurité supplémentaire n'a été imposée. Il est possible de restreindre l'accès à notre volume en définissant une ACL similaire à ce qui existe en NFS via le fichier **/etc/exports**.

```
root@stor2:~# gluster volume set repl-vol
auth.allow 192.168.69.104
volume set: success
```

Il est également possible de définir une *wildcard*, par exemple **192.168.69.*** afin d'autoriser tout un réseau. Dans cet exemple, nous avons autorisé explicitement une adresse IP à se connecter au volume.

Nous aurions également pu autoriser un nom d'hôte ou plusieurs adresses IP ou noms séparés par des virgules. Le fait de définir l'attribut **auth.allow** a comme effet immédiat d'interdire toutes les autres machines qui n'ont pas été explicitement autorisées. Pour revenir au comportement par défaut, il faut autoriser le caractère *wildcard* (*) tout simplement. À l'inverse, l'attribut **auth.reject** n'interdit aucune machine par défaut (**auth.reject** avec comme valeur **NONE**). Il sert comme vous l'avez deviné à interdire explicitement une machine. Pour résumer, le contrôle d'accès a une logique similaire avec ce qui existe côté TCP Wrappers.

2.6 Une corbeille sur le volume

GlusterFS sait gérer une corbeille au niveau volume pour conserver les fichiers supprimés. Le dossier est créé automatiquement par gluster et ne peut être supprimé. Fait intéressant, gluster sait si on le lui dit, tirer parti de cette corbeille pour ses opérations internes. Activons donc une corbeille pour les fichiers de moins de 10 Mio :

```
$ gluster volume set repl-vol features.trash on
$ gluster volume set repl-vol features.trash-
dir "Corbeille"
$ gluster volume set repl-vol features.trash-
max-filesize 10485760
$ gluster volume set repl-vol features.trash-
internal-op on
```

2.7 Node HS ? Pas de panique !

Un incident majeur sur un équipement sensible d'un système d'information, c'est bien entendu quelque chose auquel on se doit d'être préparé. Dans un système hautement disponible, tout élément qui n'est pas considéré comme un point

unique de défaillance (SPOF) doit pouvoir être indisponible sans impacter fortement le bon fonctionnement du système. Nous nous retrouvons dans un état de fonctionnement dégradé. Si le système défaillant ne peut être dépanné, un processus de reconstruction doit être mis en œuvre.

Nous allons considérer que le nœud **stor0** est irrémédiablement défaillant, la VM est même supprimée. Cela se vérifie par la commande suivante :

```
root@stor1:~# gluster volume heal repl-vol info
Brick stor0:/data/glusterfs/vol0/brick0
Status: Noeud final de transport n'est pas
connecté
Brick stor1:/data/glusterfs/vol0/brick0
Number of entries: 0
```

Voyons étape par étape comment le nouveau serveur nommé **stor2** va prendre de relais de celui-ci. Pour cela, la première étape que je ne vais pas détailler consiste à provisionner un nouveau serveur avec le disque de données et les dépendances comme indiqué précédemment.

Premièrement, on ajoute le nouveau nœud et on va confirmer qu'on a bien un nouveau nœud présent, et un ancien toujours connu du cluster, mais manquant :

```
root@stor1:~# gluster peer probe stor2
peer probe: success.
root@stor1:~# gluster peer status
Number of Peers: 2

Hostname: stor0
Uuid: a920b020-9e5a-46f6-b073-1cc8ec00ba0e
State: Peer in Cluster (Disconnected)

Hostname: stor2
Uuid: f2a03465-11bb-4c2a-a882-22933cfa2d08
State: Peer in Cluster (Connected)
```

Remplaçons maintenant la brick du **stor0** par celle de notre nouveau serveur **stor2** et vérifions son état de santé :

```
root@stor1:~# gluster volume replace-
brick repl-vol stor0:/data/glusterfs/
vol0/brick0 stor2:/data/glusterfs/vol0/
brick0 commit force
volume replace-brick: success: replace-
brick commit force operation successful
```

On réconcilie le volume :

```
root@stor1:~# gluster volume heal repl-vol full
Launching heal operation to perform full self
heal on volume repl-vol has been successful
Use heal info commands to check status
```

```
root@stor1:~# gluster volume heal repl-vol info
Brick stor2:/data/glusterfs/vol0/brick0
Number of entries: 0

Brick stor1:/data/glusterfs/vol0/brick0
Number of entries: 0
```

Et depuis le nouveau node, lançons une synchronisation :

```
root@stor2:/data/glusterfs/vol0/brick0# gluster volume
sync stor1 repl-vol
Sync volume may make data inaccessible while the sync
is in progress. Do you want to continue? (y/n) y
```

Il nous reste une dernière étape : répliquer le *volume id* dans les attributs étendus du système de fichiers et le propager au second serveur. Pour le récupérer, il faut lancer la commande suivante :

```
root@stor1:~# getfattr  -n trusted.glusterfs.
volume-id /data/glusterfs/vol0/brick0/
getfattr: Suppression des " / " en tête des
chemins absolus
# file: data/glusterfs/vol0/brick0/
trusted.glusterfs.volume-
id=0seEhN1zXZTFOXmRGV92ibvw==
```

Sur le nouveau serveur, on applique l'ID du volume sur la brick :

```
root@stor2:/data/glusterfs/vol0/brick0#
setfattr -n trusted.glusterfs.volume-id -v
'0seEhN1zXZTFOXmRGV92ibvw==' /data/glusterfs/vol0/brick0/
service glusterfs-server restart
```

La configuration de notre volume est bien mise à jour comme on peut le voir ci-dessous. Dans le cadre d'un volume distribué, il faudrait lancer un rééquilibrage (*rebalance*) du volume :

```
root@stor1:~# gluster volume info repl-vol

Volume Name: repl-vol
Type: Replicate
Volume ID: 1c493043-9c2d-4be6-afcd-8512577342c9
Status: Started
Number of Bricks: 1 x 2 = 2
Transport-type: tcp
Bricks:
Brick1: stor2:/data/glusterfs/vol0/brick0
Brick2: stor1:/data/glusterfs/vol0/brick0
Options Reconfigured:
performance.readdir-ahead: on
cluster.self-heal-daemon: enable
network.ping-timeout: 5
```

Enfin, il ne reste plus qu'à retirer l'ancien nœud des *peers* autorisés dans le *trusted pool* :

```
root@stor1:~# gluster peer detach stor0
peer detach: success
root@stor1:~# gluster peer status
Number of Peers: 1

Hostname: stor2
Uuid: f2a03465-11bb-4c2a-a882-22933cfa2d08
State: Peer in Cluster (Connected)
```

Il ne doit plus apparaître dans la liste des nœuds :

```
root@stor1:~# gluster pool list
UUID                                    Hostname        State
0cb0f3b6-10e5-41c4-ad7e-cb9ca794db9e    stor2           Connected
d5ff9617-6989-48ae-be3a-3e1286060ea1    localhost       Connected
```

Vous savez désormais comment remplacer un nœud défaillant, sachant que ce processus s'applique également en cas de migration de la brick de **stor0** vers un nouveau serveur.

2.8 Étendre le volume

Quand l'espace disque commence à manquer, une première solution peut être d'étendre l'espace libre sur les bricks, d'où l'intérêt d'être parti au départ sur du LVM. Une autre solution est d'étendre le cluster avec de nouveaux nœuds afin d'améliorer la disponibilité du système dans son ensemble. Cette extension du cluster se fait en outre sans interruption de service.

Pour étendre un cluster répliqué, il faut ajouter un nombre de bricks avec un nombre multiple du nombre de réplicas. Nous avons monté un volume à deux réplicas, il nous faut donc ajouter deux bricks supplémentaires. La commande **gluster volume info repl-vol** nous permet de le confirmer (**1x2**). Nous allons ajouter donc deux serveurs **stor3** et **stor4**, avec le volume disque préparé et le *package* glusterfs-server installé.

La première étape consiste à autoriser les deux nœuds avec la commande **gluster peer probe** vue précédemment. On peut donc ensuite ajouter des bricks au volume en spécifiant les bricks de nos deux nouveaux serveurs :

```
root@stor2:~# gluster
volume add-brick repl-vol
stor3:/data/glusterfs/
vol0/brick0 stor4:/data/
glusterfs/vol0/brick0
volume add-brick: success
```

Vérifions notre volume, nous devons retrouver nos deux bricks supplémentaires.

```
root@stor2:~# gluster volume
info repl-vol

Volume Name: repl-vol
Type: Distributed-Replicate
Volume ID: 78484dd7-35d9-4c53-
9799-1195f7689bbf
Status: Started
Number of Bricks: 2 x 2 = 4
Transport-type: tcp
Bricks:
Brick1: stor2:/data/glusterfs/
vol0/brick0
Brick2: stor1:/data/glusterfs/
vol0/brick0
Brick3: stor3:/data/glusterfs/
vol0/brick0
Brick4: stor4:/data/glusterfs/
vol0/brick0
Options Reconfigured:
performance.readdir-ahead: on
```

Notre volume à deux réplicas comportant quatre nœuds se comporte donc désormais comme un volume distribué répliqué par la magie de l'extension du volume. Seul problème, il n'y a aucune donnée sur les serveurs **stor3** et **stor4**, ce qui n'a pas eu pour effet de libérer de l'espace disque sur les deux premiers serveurs. Il est donc nécessaire de répartir la volumétrie sur l'ensemble des bricks qui composent le volume :

```
root@stor2:~# gluster volume rebalance repl-vol start
volume rebalance: repl-vol: success: Rebalance on repl-vol has been started successfully. Use
rebalance status command to check status of the rebalance process.
ID: dffbed2e-3a0c-4d7d-9f43-9d978a546b04
```

Pour vérifier, il suffit de lancer la même commande avec le paramètre **status** :

```
root@stor2:~# gluster volume rebalance repl-vol status
        Node Rebalanced-files          size       scanned      failures       skipped        status   run time in secs
   ---------      -----------   -----------   -----------   -----------   -----------   ------------     --------------
   localhost                5        0Bytes            10             0             0     completed              2.00
       stor1                0        0Bytes             0             0             0     completed              1.00
       stor3                0        0Bytes             2             0             0     completed              1.00
       stor4                0        0Bytes             0             0             0     completed              1.00
volume rebalance: repl-vol: success
```

2.9 Les quotas

GlusterFS dispose d'un mécanisme permettant de définir des quotas au niveau dossier. Ils ne sont pas activés par défaut. Pour changer ce comportement :

```
root@stor2:~# gluster volume quota repl-vol enable
volume quota : success
```

Nous allons appliquer une limite à 1Gio sur le sous-dossier **subdir** de notre volume. Ce dossier devra avoir été impérativement créé depuis le client glusterfs ajouté précédemment. Pour créer ce quota :

```
root@stor2:~# gluster volume quota repl-vol limit-usage /subdir 1GB
volume quota : success
```

Si nous avions souhaité créer un quota au niveau du volume, il suffit d'indiquer **/** dans le chemin. Créons un fichier approchant le quota depuis notre client GlusterFS :

```
root@desktop:/data/subdir# dd if=/dev/zero of=/data/subdir/toto bs=1024 count=1024000
1024000+0 enregistrements lus
1024000+0 enregistrements écrits
1048576000 bytes (1,0 GB, 1000 MiB) copied, 223,044 s, 4,7 MB/s
```

Et voyons l'état du quota :

```
root@stor1:~# gluster volume quota repl-vol list
          Path                   Hard-limit  Soft-limit      Used  Available  Soft-limit exceeded? Hard-limit exceeded?
-------------------------------------------------------------------------------------------------------------------------------
/subdir                            1.0GB     80%(819.2MB) 1000.0MB  24.0MB            Yes                  No
```

Reprenons notre commande précédente, en créant un fichier au nom différent, la création est bien interrompue sur le dépassement de quota *hard* :

```
root@desktop:/data# dd if=/dev/zero of=/data/subdir/tata bs=1024 count=1024000
dd: erreur d'écriture de '/data/subdir/tata': Débordement du quota d'espace disque
dd: fermeture du fichier de sortie '/data/subdir/tata': Débordement du quota d'espace disque
```

CONCLUSION

Il est difficile d'être exhaustif sur un sujet aussi vaste. J'espère toutefois avoir aiguisé votre appétit sur GlusterFS et vous avoir donné l'envie de tester ce qui n'a pas été détaillé ici. ■

MISE EN PLACE D'UNE IP VIRTUELLE AVEC COROSYNC ET PACEMAKER

SÉBASTIEN REUILLER
[Chef de projet informatique]

MOTS-CLÉS : HAUTE DISPONIBILITÉ, PACEMAKER, COROSYNC, FAILOVER, HAPROXY, DEBIAN

Dans la course à la haute disponibilité, le basculement automatique en cas de défaillance est un mécanisme incontournable pour assurer la continuité de service. Par exemple, vous pouvez avoir déployé le meilleur des répartiteurs de charge, si vous n'en déployez qu'un, vous créerez par la même occasion un point unique de défaillance. La mise en place d'une IP virtuelle va vous permettre d'éviter cela, suivez le guide...

Pour illustrer l'article, nous prendrons l'exemple d'un serveur **HAProxy** que nous souhaitons redonder en mode Actif/Passif (voir figure 1).

1. CRÉER LE CLUSTER

1.1 Installation de Corosync

La première chose à faire est la mise en place du cluster avec **Corosync**. Ce dernier permet d'assurer la communication entre les nœuds et leur permet de s'informer mutuellement de leurs états respectifs et également des états de leurs ressources (Ex. HAProxy, IP virtuelle). Dans notre exemple, nous aurons donc deux nœuds, à savoir les deux serveurs où sont installés les HAProxy.

Corosync étant dans les paquets **Debian** depuis la version 9, son installation est une formalité (à réaliser sur les deux nœuds) :

```
Proxy-1:~# apt-get update
Proxy-1:~# apt-get install
corosync
```

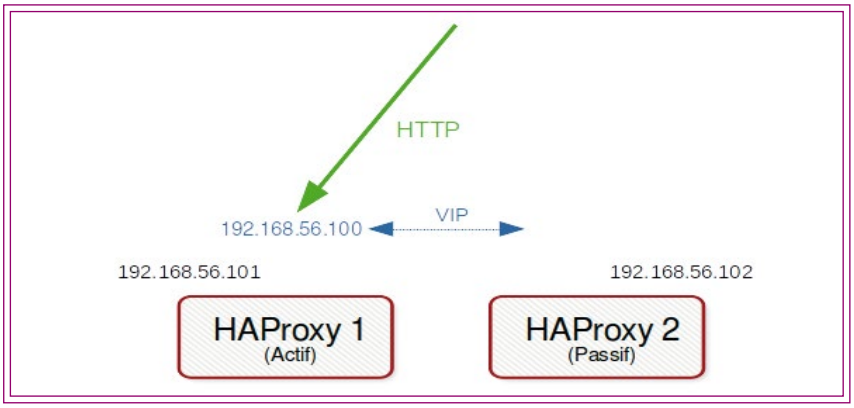


Fig. 1 : Schéma de principe.

1.2 Configuration du cluster Corosync

Une fois Corosync installé, toute la configuration se fait dans le fichier **/etc/corosync/corosync.conf**.

Dans notre exemple, le rattachement au cluster se fera sans authentification. Il conviendra donc de changer uniquement les directives **cluster_name** (le nom que vous souhaitez donner au cluster), **bindnetaddr** (adresse réseau de l'interface d'écoute, **192.168.56.101** pour le premier nœud et **192.168.56.102** pour le deuxième nœud) et **mcastaddr** (identique sur les deux nœuds, c'est l'adresse de *multicast* sur laquelle les nœuds vont échanger).

```
totem {
 version: 2
 cluster_name: CLT_
HAProxy
 token: 3000
 token_retransmits_
before_loss_const: 10
 clear_node_high_bit: yes
 crypto_cipher: none
 crypto_hash: none
 interface {
  ringnumber: 0
  bindnetaddr:
192.168.56.101
  mcastaddr:
239.255.56.56
  mcastport: 5405
  ttl: 1
 }
}
```

Une fois cette configuration effectuée sur les deux nœuds, il suffit de relancer les services Corosync :

```
Proxy-1:~# systemctl
restart corosync.service
```

1.3 Contrôler l'état du cluster

Le premier contrôle à effectuer est de vérifier l'état des services Corosync qui doivent être « active (running) » :

```
Proxy-1:~# systemctl
status corosync.service
```

Ensuite, contrôler l'état du cluster (*Ring number 0*) :

```
Proxy-1:~# corosync-cfgtool -s
```

Voici la sortie que vous devriez avoir sur le premier nœud :

```
Printing ring status.
Local node ID 1084766309
RING ID 0
 id = 192.168.56.101
 status = ring 0 active with
no faults
```

Et sur le deuxième nœud :

```
Printing ring status.
Local node ID 1084766310
RING ID 0
 id = 192.168.56.102
 status = ring 0 active with
no faults
```

Enfin, vous pouvez vous assurer que les nœuds ont bien rejoint le cluster avec la commande :

```
Proxy-1:~# corosync-
cmapctl runtime.totem.
pg.mrp.srp.members
```

La sortie doit être similaire sur les deux nœuds et ressembler à la sortie suivante :

```
runtime.totem.pg.mrp.srp.
members.1084766309.config_version (u64) = 0
runtime.totem.pg.mrp.srp.
members.1084766309.ip (str) = r(0)
ip(192.168.56.101)
runtime.totem.pg.mrp.srp.
members.1084766309.join_count (u32) = 1
runtime.totem.pg.mrp.srp.
members.1084766309.status (str) = joined
runtime.totem.pg.mrp.srp.
members.1084766310.config_version (u64) = 0
runtime.totem.pg.mrp.srp.
members.1084766310.ip (str) = r(0)
ip(192.168.56.102)
runtime.totem.pg.mrp.srp.
members.1084766310.join_count (u32) = 1
runtime.totem.pg.mrp.srp.
members.1084766310.status (str) = joined
```

Notre cluster est désormais fin prêt pour accueillir ses ressources !

2. CRÉER L'IP VIRTUELLE

2.1 Installer Pacemaker

Maintenant que notre cluster est en place, nous pouvons passer à la création de notre IP virtuelle. C'est alors que **Pacemaker** entre en scène. Son installation est également une formalité sous Debian Stretch :

```
Proxy-1:~# apt-get update
Proxy-1:~# apt-get install pacemaker crmsh
```

Une fois l'installation faite sur les deux nœuds, la configuration doit être faite uniquement sur un nœud.

Pour débuter la configuration, entrez :

```
Proxy-1:~# crm configure
```

Vous pouvez dès lors afficher la configuration à tout moment :

```
crm(live)configure# show
```

Voici la sortie que vous devriez obtenir :

```
node 1084766309: Proxy-1
node 1084766310: Proxy-2
```

```
property cib-bootstrap-options: \
 have-watchdog=false \
 dc-version=1.1.16-94ff4df \
 cluster-infrastructure=corosync \
 cluster-name=CLT_HAProxy
```

Notez que les nœuds et le nom du cluster ont été repris.

Avant de commencer la configuration de notre IP virtuelle, nous désactivons l'option **STONITH** (ce composant de Pacemaker permet de forcer l'arrêt d'un nœud dès lors qu'il est en défaut vis-à-vis du cluster). Cette fonctionnalité très utile dans un environnement où les ressources ont des actions concurrentes ne sera pas abordée ici.

```
crm(live)configure# property
stonith-enabled=false
```

Validez ensuite la configuration :

```
crm(live)configure# commit
```

2.2 Créer la ressource VIP

L'IP virtuelle est une ressource gérée nativement par Pacemaker sous le nom de **IPaddr2**.

Voici comment créer la ressource **vip100** grâce à la directive **primitive** :

```
crm(live)configure# primitive
vip100 IPaddr2 params
ip=192.168.56.100 nic=ens3
```

L'IP **192.168.56.100** sera déclarée sur l'interface **ens3** (paramètre à ajuster selon la configuration de votre serveur).

2.3 Configurer l'emplacement par défaut de la ressource

Définissez ensuite l'emplacement par défaut de l'IP virtuelle grâce à la directive **location** :

```
crm(live)configure# location default-
location-vip100 vip100 inf: Proxy-1
```

3. MISE EN PLACE DE LA BASCULE AUTOMATIQUE

3.1 Créer la ressource à surveiller (Ex. HAProxy)

Le service à surveiller est une ressource au même titre que l'IP virtuelle, nous la créons donc avec la directive **primitive** :

```
crm(live)configure# primitive srv_haproxy service:haproxy
op monitor interval=2s timeout=15s on-fail=standby
```

3.2 Configurer la règle de basculement

Les deux ressources sont désormais créées, mais sans aucune interaction entre elles. Nous souhaitons basculer la ressource **vip100** si la ressource **srv_haproxy** bascule. Nous devons donc créer une règle de **colocation**. Une subtilité existe sur cette règle à savoir que si vous la créez directement sur la ressource **srv_haproxy**, le service HAProxy ne sera actif que sur un seul nœud. Dans notre cas, nous préférerons qu'il soit toujours actif sur les deux nœuds afin que la bascule soit le plus rapide possible. Pour que cela soit possible, nous devons cloner la ressource **srv_haproxy** comme suit :

```
crm(live)configure# clone srv_haproxy_clone srv_haproxy
```

Nous pouvons ensuite créer la règle de **colocation** :

```
crm(live)configure# colocation coloc_vip100_srv_haproxy
inf: vip100 srv_haproxy_clone
```

Le paramètre **inf:** signifie que les deux ressources doivent toujours être sur le même nœud.

Enfin, vous pouvez valider la configuration :

```
crm(live)configure# commit
```

La configuration est dès lors active et l'IP virtuelle apparaît sur **Proxy-1**.

3.3 Tester le basculement automatique

Pour commencer les tests, assurez-vous que l'IP virtuelle est bien présente sur le premier nœud avec la commande :

```
Proxy-1:~# ip a
```

Et qu'elle n'est pas sur le deuxième nœud :

```
Proxy-2:~# ip a
```

Pour tester la bascule, il suffit d'arrêter le service **haproxy** sur le premier nœud :

```
Proxy-1:~# systemctl stop haproxy.service
```

La bascule est quasi instantanée grâce à l'intervalle de 2 secondes que nous avons configuré sur la ressource **srv_haproxy**. Afin de valider le test, exécutez

de nouveau la commande **ip a** sur les deux nœuds pour vous assurer que l'IP **192.168.56.100** est bien passée sur le deuxième nœud.

Pour revenir à l'état normal, relancez **HAProxy** :

```
Proxy-1:~# systemctl start haproxy.service
```

Et forcez l'actualisation de l'état de la ressource **srv_haproxy** :

```
Proxy-1:~# crm resource cleanup srv_haproxy
```

L'IP virtuelle doit alors repasser sur le premier nœud.

4. SURVEILLANCE

4.1 Les commandes utiles

Pour surveiller notre cluster, plusieurs commandes sont à notre disposition. La principale commande est :

```
Proxy-1:~# crm_mon
```

Elle permet d'avoir un état des lieux du cluster et de l'état des ressources en temps réel.

Ensuite, vous pouvez consulter les paramètres de Corosync avec la commande :

```
Proxy-1:~# corosync-cmapctl
```

Enfin, pour avoir une vue plus détaillée de l'état du cluster, mais uniquement du cluster (sans l'état des ressources), exécutez la commande :

```
Proxy-1:~# crm corosync status
```

4.2 Les notifications

En cas de basculement automatique à la suite d'un problème, il est évident qu'il faille intervenir et pour cela, en être informé. Un des moyens les plus simples à mettre en œuvre pour recevoir l'alerte est de configurer les notifications par mail.

Pour cela, configurez une nouvelle ressource comme suit :

```
crm(live)configure# primitive notification
MailTo params email="mon@adressemail.fr"
subject="[Cluster Corosync] Notification"
op monitor timeout=10 interval=10 depth=0
```

Puis, configurez une règle de **colocation** pour que cette nouvelle ressource suive l'IP virtuelle :

```
crm(live)configure# colocation notification_
vip100 inf: notification vip100
```

Ainsi, à chaque changement d'état de la ressource **notification,** vous recevrez un mail.

ATTENTION !

Cette configuration fonctionne si vous avez correctement configuré un serveur de mail sur les deux nœuds. Si ce n'est pas le cas, vous pouvez installer le paquet **ssmtp** qui vous permettra de configurer un serveur de mail externe.

CONCLUSION

Le couple Corosync et Pacemaker est une merveille pour la haute disponibilité. Vous avez vu tout au long de cet article que sa configuration est plutôt simple, vous avez également aperçu ses possibilités, je vous laisse maintenant apprécier sa stabilité au sein de vos environnements de production les plus critiques... ■

RÉFÉRENCES

[1] Site officiel de *Corosync* : http://corosync.github.io/corosync/

[2] Site officiel de *Pacemaker* : http://www.clusterlabs.org/

POUR ALLER PLUS LOIN

crmsh offre des possibilités infinies et vous permet de les explorer avec la commande :

```
Proxy-1:~# crm help
```

Vous y trouverez notamment des informations sur la multitude d'agents de ressource proposés en natif :

```
Proxy-1:~# crm ra classes
lsb
ocf / .isolation heartbeat pacemaker redhat
service
stonith
systemd
Proxy-1:~# crm ra list ocf
ASEHAagent.sh     AoEtarget       AudibleAlarm
CTDB              ClusterMon      Delay
Dummy             EvmsSCC         Evmsd
Filesystem        HealthCPU       HealthSMART
ICP               IPaddr          IPaddr2
IPsrcaddr         IPv6addr        LVM
LinuxSCSI         MailTo          ManageRAID
...
```

Et les informations sur les agents :

```
Proxy-1:~# crm ra info ClusterMon
```

À vous de jouer !

INFLUXDB, GRAFANA ET GLANCES,
LE MONITORING QUI BRILLE

DENIS GOURMEL
[Linux SysAdmin]

MOTS-CLÉS : INFLUXDB, GRAFANA, GLANCES, DOCKER, FREENAS

Dans l'article qui va suivre, je vais vous guider, si vous l'acceptez, dans la mise en place d'un système de monitoring peu conventionnel. En nous basant sur des programmes très spécifiques, mais complémentaires, nous allons créer un outil capable de surveiller vos serveurs tout en affichant vos données dans une interface digne du XXIème siècle.

Le monde du monitoring est sûrement plus stable que les services et produits qui sont eux-mêmes « monitorés ». C'est pourquoi j'ai eu envie de voir ce que l'on pouvait faire à ce niveau-là avec les technologies d'aujourd'hui... Force est de constater que je ne m'attendais pas à un tel résultat.

Allons-y, je vais vous conter mes péripéties.

1. PRÉPARATION DES PAQUETAGES AVANT L'ASCENSION

Plutôt que d'utiliser une solution de monitoring « tout-en-un », je préfère une fois de plus construire mon système grâce à différentes briques que je juge complémentaires. Ma distribution serveur du moment étant **Ubuntu**, cet article se base sur la dernière version « LTS » (soit Ubuntu 16.04). Cependant, vu que nous allons utiliser **Docker**, la distribution de l'OS n'aura au final que très peu d'importance.

1.1 Installation de Docker

J'espère que vous êtes bien harnachés, car c'est parti.

Nous allons installer Docker « Community Edition » avec la dernière méthode en date.

Tout d'abord les « prérequis » :

```
$ sudo apt-get install -y apt-transport-https software-properties-common  ca-certificates curl
```

Ajoutons ensuite le dépôt adéquat :

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
$ sudo add-apt-repository  "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

Il faut désormais mettre à jour la liste de nos dépôts afin de prendre en compte l'ajout :

```
$ sudo apt-get update
```

Et enfin, nous pouvons installer la dernière version de Docker ainsi que **docker-compose** que nous utiliserons pour gérer nos containers :

```
$ sudo apt-get -y install docker-ce docker-compose
```

L'étape suivante n'est pas obligatoire, mais j'aime bien configurer mes installations de Docker de la sorte. Il s'agit simplement d'ajouter son utilisateur au groupe **docker** afin de pouvoir lancer Docker sans passer « sudo » :

```
$ sudo usermod -aG docker MonUser
```

NOTE

Afin de prendre en compte les changements, vous devrez vous déconnecter / reconnecter.

1.2 Configuration du réseau Docker

Dans la partie qui va suivre, nous allons voir comment créer un réseau « dédié » à nos containers de monitoring. Pour une fois, nous allons pouvoir ajouter une couche de sécurité tout en ajoutant aussi une couche de praticité. Qui a dit qu'il fallait choisir entre sécurité et praticité ?

En effet, par défaut Docker crée ses containers avec des IP dynamiques et les changent à chaque redémarrage des containers.

Nous ajoutons donc le côté pratique d'avoir une IP fixe et un nom d'hôte pour chaque container tout en isolant les containers du réseau principal. Cette isolation va permettre à nos containers d'ouvrir des ports uniquement visibles par eux-mêmes et non pas par le reste du monde.

```
$ docker network create --driver bridge br0
```

Vérifions que notre nouveau réseau est disponible avec la commande suivante :

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
f5fa89b94de4        br0                 bridge              local
193d033921f1        bridge              bridge              local
49e3217679f9        host                host                local
d7e622968ebb        none                null                local
```

Et… voilà, c'est tout ! Docker est installé et fonctionnel, voilà le premier palier franchi.

2. RÉCUPÉRATION ET LANCEMENT DES IMAGES

Attaquons-nous au palier suivant, mais n'ayez crainte, l'ascension n'en sera pas pour autant plus compliquée. En cas de chute, je suis là et je tiens la corde.

J'ai pour habitude de placer mes images Docker dans un répertoire dédié :

```
$ sudo mkdir -p /srv/data/apps/{glances,influxdb,grafana,nginx}
$ chown -R MonUser: /srv/data
```

2.1 InfluxDB

À partir du bon répertoire, construisons un fichier **/srv/data/apps/influxdb/docker-compose.yml** :

```
# docker-compose for influxDB
version: '2'

services:
  influxdb:
    image: influxdb
    container_name: influxdb
    ports:
#     - "8083:8083"
      - "8086:8086"
      - "2003:2003"
    volumes:
      - influxdb-storage:/var/lib/influxdb
    environment:
      - INFLUXDB_GRAPHITE_ENABLED=true
```

```
    restart: always
    networks:
      default:
        aliases:
          - influxdb

networks:
  default:
    external:
      name: br0

volumes:
  influxdb-storage:
```

Nous pouvons désormais lancer notre premier container. Vu qu'il s'agit du premier lancement, la dernière image en date d'InfluxDB sera aussi téléchargée :

```
$ docker-compose up -d
```

Afin de vérifier que tout est en ordre, nous pouvons exécuter la commande suivante :

```
$ docker ps
CONTAINER
ID01c53fe77784

IMAGE         COMMAND                  CREATED
influxdb      "/entrypoint.sh in..."   20 minutes ago

STATUS          PORTS                                NAMES
Up 20 minutes   0.0.0.0:2003->2003/tcp, 8086/tcp     influxdb
```

NOTE

Les sorties console ont été réarrangées pour une meilleure lisibilité.

Vous constaterez que le statut du container nommé **influxdb** indique **Up** : cela veut dire que tout fonctionne.

Ensuite, nous devons créer une base de données au sein d'InfluxDB afin d'y stocker les statistiques venant de **Glances**. La première étape consiste à entrer dans le container d'InfluxDB :

```
$ docker exec -it influxdb bash
```

Une fois dans le container, connectons-nous à InfluxDB :

```
$ influx
```

La dernière étape consiste à créer la base, l'utilisateur, et lui donner les droits :

```
Connected to http://localhost:8086 version 1.2.2
InfluxDB shell version: 1.2.2
> create database glances
> create user "glances" with password 'MyPass'
> grant all on "glances" to "glances"
> exit
```

Attaquons-nous désormais à la collecte des statistiques avec Glances **[1]**.

NOTE

Pour quitter le container et revenir à notre hôte, un simple **<Ctrl>** + **<d>** suffit.

2.2 Glances

Même procédure que précédemment ; nous nous plaçons dans le bon répertoire et créons notre fichier de configuration principal **/srv/data/apps/glances/docker-compose.yml** :

```
# docker-compose for Glances
version: '2'

services:
  glances:
    image: docker.io/nicolargo/glances
    container_name: glances
    volumes:
      - /var/run/docker.sock:/var/run/
docker.sock:ro
      - ./conf/glances.conf:/glances/
conf/glances.conf
      - ./scripts:/scripts
  #    - /home/MonUser/fake:
/srv/fake
    environment:
      - GLANCES_OPT=-q --export-influxdb
    pid: "host"
    restart: always
    networks:
      default:
        aliases:
          - glances

networks:
  default:
    external:
      name: br0
```

ATTENTION !

En fonction de votre partitionnement, il se peut que le container de Glances ne voit pas toutes vos partitions. Dans ce cas, je vous recommande de créer un dossier « bidon » sur les partitions en question et de le monter via le fichier **docker-compose.yml** (cf. le dossier **/srv/fake** dans ma config).

Ensuite, nous allons créer les dossiers, fichiers de configuration et points de montage nécessaires au lancement de Glances en nous basant sur le fichier que nous venons de créer :

```
$ mkdir {conf,scripts}
```

Pour ce qui est de la configuration de Glances, elle se trouve dans **/srv/data/apps/glances/conf/glances.conf**. Je vous présente uniquement les parties les plus « intéressantes », le reste étant les valeurs par défaut. De plus, vous trouverez tous mes fichiers de configuration sur mon **GitHub [2]**.

```
[load]
careful=0.7
warning=1.0
critical=5.0
critical_action=sh /scripts/send_sms.sh
"Load:: `uptime`"

[mem]
careful=50
warning=70
ceitical=90
critical_action=sh /scripts/send_sms.
sh "Free mem: 'free -m | grep Mem | awk
'{print $4}'`"

[fs]
careful=50
warning=70
critical=90
critical_action=sh /scripts/send_sms.sh
"Disk: 'df -h | grep md | grep 9[0-9]'"

[influxdb]
host=influxdb
port=8086
user=glances
password=MyPass
db=glances
prefix=ubunmonit

[amp_nginx]
enable=true
regex=nginx
refresh=60
one_line=false
status_url=http://nginx/nginx_status
```

Comme nous pouvons le voir dans la config ci-dessus, j'utilise un script « maison » pour envoyer des alertes par SMS lorsqu'un seuil critique est atteint. Voici le script **scripts/send_sms.sh** en question :

```
#!/bin/bash
curl -k "https://smsapi.free-mobile.fr/se
ndmsg?user=MonUser&pass=MyPass&msg=Ubunmo
nit - $1"
```

N'oubliez pas de rendre votre script exécutable :

```
$ chmod +x scripts/send_sms.sh
```

NOTE

Ce script permet d'envoyer des SMS via l'**API Free mobile [3]**. Dans le cas où vous n'utiliseriez pas Free, il existe des méthodes similaires (ou basées sur l'envoi de mails) chez les autres opérateurs, voire chez des prestataires externes.

Construisons notre container :

```
$ docker-compose up -d
```

Encore une fois, nous pouvons vérifier que tout fonctionne avec **docker ps**. En cas d'erreur, pour « débugger » , nous pouvons utiliser la commande **docker logs -f glances**, ou encore **docker exec -it glances bash** (si le container est **Up**) pour aller « fouiller » dans le container.

Admettons que tout fonctionne, et allons voir dans notre base InfluxDB si nous avons récolté des données :

```
$ docker exec -it influxdb bash
root@01c53fe77784:/# influx
Connected to http://localhost:8086 version 1.2.2
InfluxDB shell version: 1.2.2

> show databases
name: databases
name
----

_internal
glances

> use glances
Using database glances
> show measurements
name: measurements
name
----
ubunmonit.cpu
ubunmonit.diskio
ubunmonit.docker
ubunmonit.fs
ubunmonit.ip
ubunmonit.load
ubunmonit.mem
ubunmonit.memswap
ubunmonit.network
ubunmonit.percpu
ubunmonit.processcount
ubunmonit.sensors
ubunmonit.uptime
```

Parfait ! Glances communique avec InfluxDB et la base commence à stocker nos différentes statistiques.

J'ai beau adorer mon terminal, quand il s'agit de statistiques je préfère quand même avoir des graphiques et une représentation plus « visuelle ». Ça tombe bien, c'est exactement le rôle de **Grafana**.

2.3 Grafana

Pour ne pas déroger à la règle, nous allons créer le fichier suivant **/srv/data/apps/grafana/docker-compose.yml** :

```
# docker-compose for Grafana
version: '2'
services:
  grafana:
    image: grafana/grafana
    container_name: grafana
#    ports:
#      - "3000:3000"
```

```
    volumes:
      - grafana-storage:/var/lib/grafana
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=An0th3rP4ss
    restart: always
    networks:
      default:
       aliases:
          - grafana

networks:
  default:
    external:
      name: br0

volumes:
  grafana-storage:
```

Et nous lançons le container :

```
$ docker-compose up -d
```

Comme vous pouvez le remarquer, j'ai décidé de ne pas « exposer » le port **3000**. C'est-à-dire qu'à l'heure actuelle, notre interface graphique n'est pas visible depuis l'extérieur de notre serveur.

Vérifions :

```
$ netstat -tulpn
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address   Foreign Address   State    PID/Program name
tcp        0      0 0.0.0.0:22      0.0.0.0:*         LISTEN   -
tcp6       0      0 :::2003         :::*              LISTEN   -
tcp6       0      0 :::22           :::*              LISTEN   -
udp        0      0 0.0.0.0:68      0.0.0.0:*                  -
```

Effectivement, seuls SSH (22/tcp), DHCP (68/udp) et 2003/tcp sont visibles depuis l'extérieur. Pour ce qui est du port 2003/tcp, il provient du container InfluxDB et correspond à une base **Graphite** dans laquelle je stocke les informations provenant de mon NAS (nous verrons cela plus tard).

Un simple **docker ps** nous confirme tout de même que le port **3000** (par exemple) est utilisé :

```
6a061326eb47    grafana/grafana              "/run.sh"
01c53fe77784    influxdb                     "/entrypoint.sh in..."
93e983755929    docker.io/nicolargo/glances  "/bin/sh -c 'pytho..."

6 minutes ago       Up 6 minutes      3000/tcp                          grafana
32 minutes ago      Up 32 minutes     0.0.0.0:2003->2003/tcp, 8086/tcp  influxdb
43 minutes ago      Up 21 minutes     61208-61209/tcp                   glances
```

Mais comment allons-nous y accéder ? La réponse à cette question après la pu... ou plutôt après un petit bonus : récupérer les stats de **FreeNAS**.

2.4 Bonus : FreeNAS

J'utilise un NAS « maison » basé sur FreeNAS. Ce dernier propose d'envoyer ses statistiques vers une « base de données » Graphite.

Côté FreeNAS 9.x, il faut aller dans ***System > Advanced***, et renseigner l'adresse de votre serveur InfluxDB dans la partie ***Remote Graphite Server Hostname***.

Côté InfluxDB, la configuration est faite via le fichier **docker-compose.yml** que nous avons utilisé. Voilà, c'est tout ! En admettant qu'il n'y ait pas de pare-feu entre votre NAS et le serveur qui fait tourner votre container InfluxDB, cela fonctionnera.

Il est grand temps d'afficher tout ce que nous sommes en train de collecter. Pour ce faire, nous allons utiliser un reverse proxy **NGiNX** avec des certificats « SSL » (notez les guillemets).

3. CONSULTATION DES STATS VIA UN REVERSE PROXY

Troisième et avant-dernier palier. Le sommet est en vue, mais l'oxygène se raréfie. Essayons tout de même de rester concentrés.

3.1 Du httpS pour tous !

Tout d'abord, un grand merci à toute l'équipe de **Let's Encrypt [4]**. Grâce à leur projet, nous pouvons désormais utiliser des certificats « SSL » facilement et gratuitement. Il n'y a donc aucune raison de s'en passer. Installons le paquetage **certbot** pour gérer nos certificats :

```
$ sudo add-apt-repository
ppa:certbot/certbot
$ sudo apt-get update
$ sudo apt-get install -y
letsencrypt
```

Ensuite, nous générons le certificat pour notre virtual host Grafana. J'utilise un sous-domaine du genre **grafana.monserveur.com** :

```
$ sudo letsencrypt
certonly --standalone -d
grafana.monserveur.com
```

Abonnez-vous !

PARTICULIERS,

➡ Rendez-vous sur :

www.ed-diamond.com

pour consulter toutes les offres !

➡ ...ou renvoyez-nous le document au verso complété !

PROFESSIONNELS,

➡ Rendez-vous sur :

proboutique.ed-diamond.com

pour consulter toutes les offres dédiées !

➡ ...ou renvoyez-nous le document au verso complété !

VOICI LES OFFRES D'ABONNEMENT AVEC GNU/LINUX MAGAZINE !

CHOISISSEZ VOTRE OFFRE ! Prix TTC en Euros / France Métropolitaine*

Offre	ABONNEMENT	PAPIER Réf	PAPIER Tarif TTC
LM	11n° GLMF	☐ LM1	69 €
LM+	11n° GLMF + 6n° HS	☐ LM+1	125 €
	LES COUPLAGES AVEC NOS AUTRES MAGAZINES		
A	11n° GLMF + 6n° LP	☐ A1	105 €
A+	11n° GLMF + 6n° HS + 6n° LP + 3n° HS	☐ A+1	189 €
B	11n° GLMF + 6n° MISC	☐ B1	109 €
B+	11n° GLMF + 6n° HS + 6n° MISC + 2n° HS	☐ B+1	185 €
C	11n° GLMF + 6n° LP + 6n° MISC	☐ C1	149 €
C+	11n° GLMF + 6n° HS + 6n° LP + 3n° HS + 6n° MISC + 2n° HS	☐ C+1	249 €
J	11n° GLMF + 6n° HK*	☐ J1	105 €
J+	11n° GLMF + 6n° HS + 6n° HK*	☐ J+1	159 €
L	11n° GLMF + 6n° HK* + 6n° LP + 6n° MISC	☐ L1	189 €
L+	11n° GLMF + 6n° HS + 6n° HK* + 6n° LP + 3n° HS + 6n° MISC + 2n° HS	☐ L+1	289 €

Les abréviations des offres sont les suivantes : GLMF = GNU/Linux Magazine France | HS = Hors-Série | LP = Linux Pratique | HK = Hackable

J'indique l'offre si différente que celles ci-dessus :

J'indique la somme due (Total) : €

Je choisis de régler par :

❑ **Chèque bancaire ou postal à l'ordre des Éditions Diamond (uniquement France et DOM TOM)**

❑ **Pour les règlements par virements, veuillez nous contacter via e-mail : cial@ed-diamond.com ou par téléphone : +33 (0)3 67 10 00 20**

SÉLECTIONNEZ VOTRE OFFRE DANS LA GRILLE CI-DESSUS ET RENVOYEZ CE DOCUMENT COMPLET À L'ADRESSE CI-DESSOUS !

Voici mes coordonnées postales :	
Société :	
Nom :	
Prénom :	
Adresse :	
Code Postal :	
Ville :	
Pays :	
Téléphone :	
E-mail :	

❑ Je souhaite recevoir les offres promotionnelles et newsletters des Éditions Diamond.

❑ Je souhaite recevoir les offres promotionnelles des partenaires des Éditions Diamond.

En envoyant ce bon de commande, je reconnais avoir pris connaissance des conditions générales de vente des Éditions Diamond à l'adresse internet suivante : http://**boutique.ed-diamond.com/content/3-conditions-generales-de-ventes** et reconnais que ces conditions de vente me sont opposables.

LINUX MAGAZINE / FRANCE

Les Éditions Diamond
Service des Abonnements
10, Place de la Cathédrale
68000 Colmar – France
Tél. : + 33 (0) 3 67 10 00 20
Fax : + 33 (0) 3 67 10 00 21

Vos remarques :

.....

.....

RETROUVEZ TOUTES NOS OFFRES SUR : www.ed-diamond.com !

*Les tarifs hors France Métropolitaine, Europe, Asie, etc. sont disponibles en ligne !

ATTENTION !

Afin que Let's Encrypt puisse vous attribuer un certificat, il faut que votre sous-domaine soit valide. C'est-à-dire qu'un enregistrement DNS doit exister et pointer vers l'IP de votre serveur.

Répondez aux questions et voilà, si tout s'est passé comme prévu, vous trouverez vos certificats et clefs dans :

```
$ sudo ls -l /etc/letsencrypt/
total 24
drwx------  4 root root 4096 Oct 27 07:31 accounts
drwx------ 14 root root 4096 Nov 28 05:37 archive
drwxr-xr-x  2 root root 4096 Apr 22 02:50 csr
drwx------  2 root root 4096 Apr 22 02:50 keys
drwx------ 14 root root 4096 Nov 28 05:37 live
drwxr-xr-x  2 root root 4096 Apr 22 02:50 renewal
```

Le seul « inconvénient » avec Let's Encrypt, c'est que les certificats ne sont valables que trois mois. Il est donc nécessaire de mettre en place un renouvellement automatique. C'est que nous allons faire avec **systemd** (oui, oui, l'horrible systemd).

Il faut tout d'abord créer un service **/etc/systemd/system/renew-letsencrypt.service** :

```
[Unit]
Description=Renew Let's Encrypt certificates

[Service]
Type=oneshot
# check for renewal, only start/stop nginx if certs need to be renewed
ExecStart=/usr/bin/letsencrypt renew --agree-tos --pre-hook "/usr/bin/docker stop nginx" --post-hook "/usr/bin/docker start nginx"
```

Ensuite un *timer* **/etc/systemd/system/renew-letsencrypt.timer** :

```
[Unit]
Description=Daily renewal of Let's Encrypt's certificates

[Timer]
# once a day, at 2AM
OnCalendar=*-*-* 02:00:00
# Be kind to the Let's Encrypt servers: add a random delay of 3600 seconds
RandomizedDelaySec=3600
Persistent=true

[Install]
WantedBy=timers.target
```

Indiquons à systemd que nous venons d'ajouter un service :

```
$ sudo systemctl daemon-reload
```

Ensuite, démarrons le service :

```
$ sudo systemctl start renew-letsencrypt.timer
```

Et si tout se passe bien, alors activons ce service pour qu'il se lance automatiquement à chaque démarrage du serveur :

```
$ sudo systemctl enable renew-letsencrypt.timer
```

Vous pouvez vérifier que votre *timer* a bien été pris en compte avec la commande suivante :

```
$ sudo systemctl list-timers
NEXT                         LEFT          LAST
Thu 2017-04-27 07:49:57 PDT  1h 17min left Thu 2017-04-27 01:01:20 PDT
Thu 2017-04-27 12:06:28 PDT  5h 34min left Thu 2017-04-27 00:48:40 PDT
Fri 2017-04-28 02:29:20 PDT  19h left      Thu 2017-04-27 02:29:20 PDT

PASSED       UNIT                         ACTIVATES
5h 30min ago apt-daily.timer              apt-daily.service
5h 43min ago certbot.timer                certbot.service
4h 2min ago  systemd-tmpfiles-clean.timer systemd-tmpfiles-clean

3 timers listed.
```

3.2 Mise en place du reverse proxy

Encore une fois, et cette fois-ci, promis, c'est la dernière : nous allons créer un nouveau fichier **docker-compose.yml** dans **/srv/data/apps/nginx** qui ressemblera à cela :

```
# docker-compose for Nginx
version: '2'

services:
  nginx:
    image: nginx
    container_name: nginx
    ports:
      - "443:443"
      - "80:80"
    volumes:
      - ./conf.d:/etc/nginx/conf.d
      - /etc/letsencrypt:/etc/nginx/ssl
      - /srv/data/ssl/dhparam.pem:/etc/nginx/cert/dhparam.pem
    #environment:
    restart: always
    networks:
      default:
        aliases:
          - nginx

networks:
  default:
    external:
      name: br0
```

Les configurations de nos virtual hosts résident dans un dossier dédié :

```
$ mkdir -p /srv/data/apps/nginx/conf.d/vhost
```

Préparons la voie pour la config « SSL » :

```
$ mkdir /srv/data/ssl
$ cd /srv/data/ssl/ && openssl dhparam -out dhparam.pem 4096
```

Commençons par créer un vhost pour obtenir le statut de NGiNX lui-même afin que Glances puisse collecter des statistiques sur son usage (fichier **/srv/data/apps/nginx/conf.d/vhost/status.conf**) :

```
server {
    listen   80;

location /nginx_status {
        stub_status on;
        access_log   off;
        allow 172.0.0.0/8;
        deny all;
   }
}
```

Ensuite le vhost pour Grafana (fichier **grafana.conf** dans le même répertoire) :

```
server {
       listen         80;
       server_name    grafana.monserveur.com;
       return         301 https://$server_name$request_uri;
}


server {
    listen 443 ssl;
    ssl_certificate /etc/nginx/ssl/live/grafana.monserveur.
com/fullchain.pem;
    ssl_certificate_key /etc/nginx/ssl/live/grafana.
monserveur.com/privkey.pem;
    ssl_dhparam /etc/nginx/cert/dhparam.pem;
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
    ssl_prefer_server_ciphers on;
    ssl_ciphers ECDH+AESGCM:DH+AESGCM:ECDH+AES256:DH+AES256
:ECDH+AES128:DH+AES:ECDH+3DES:DH+3DES:RSA+AESGCM:RSA+AES:RS
A+3DES:!aNULL:!MD5:!DSS;

    add_header Strict-Transport-Security "max-age=31536000;
includeSubDomains" always;

    server_name  grafana.monserveur.com;
    access_log on;

    location / {
        proxy_pass      http://grafana:3000;
        proxy_set_header    Host            $host;
        proxy_set_header    X-Real-IP       $remote_addr;
        proxy_set_header    X-Forwarded-for $remote_addr;
        port_in_redirect off;
        proxy_connect_timeout 300;
    }

    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/local/nginx/html;
    }
}
```

NOTE

Notez bien les valeurs **ssl_certificate** et **ssl_certificate_key** qui pointent vers votre dossier Let's Encrypt qui a été monté dans le container via le fichier **docker-compose.yml**.

N'oublions pas de créer des liens symboliques depuis le dossier **conf.d** vers le dossier contenant nos vhosts. De la sorte, il sera plus facile d'ajouter/supprimer des vhosts à notre serveur web :

```
$ cd /srv/data/apps/nginx/conf.d
$ ln -s vhost/status.conf status.conf
$ ln -s vhost/grafana.conf grafana.conf
```

Et on démarre notre container :

```
$ cd /srv/data/apps/nginx
$ docker-compose up -d
```

L'heure de vérité a sonné, essayons de nous connecter à Grafana !

4. VISUALISATION DES STATS DANS GRAFANA

Ouvrez votre navigateur et rendez-vous sur : https://grafana.monserveur.com ! Le *login* par défaut est **admin**, et le mot de passe **An0th3rP4ss** (ou celui que vous avez entré dans le fichier **docker-compose.yml** correspondant à Grafana).

Cerise sur le gâteau, votre site passe parfaitement le test du certificat « SSL », et vous obtenez la meilleure note possible, A+ ! **[5]**

4.1 Ajout des sources

La toute première chose à faire (sans parler de changer le mot de passe et/ou créer des utilisateurs) consiste à définir vos « Data sources ». Pour ce faire, il faut cliquer dans le coin supérieur gauche sur le ***logo > Data Sources > Add data source***.

La première source que nous allons ajouter sera Glances. Nommez donc la source comme bon vous semble puis sélectionnez ***InfluxDB*** comme type. Pour l'URL, il s'agit de http://influxdb:8086, l'*access* sur proxy, et enfin les informations concernant la base, à savoir **glances** comme nom de base et d'utilisateur et le mot de passe défini dans la section **influxdb**.

Si vous utilisez FreeNAS, répétez la même opération avec les informations adéquates.

4.2 Aperçu des possibilités

Il serait trop long d'expliquer Grafana en détail, je vais donc simplement donner un aperçu du fonctionnement ainsi que de ses possibilités.

Tout d'abord, Grafana fonctionne par *dashboard* (tableau de bord dans notre langue). J'utilise par exemple un tableau de bord par périphérique. Pour illustrer cet article, nous

aurions donc un tableau de bord pour notre serveur surveillé via Glances et un second pour notre NAS.

Au sein de chaque tableau de bord, vous allez pouvoir ajouter le suivi de différentes statistiques. Admettons que vous vouliez voir l'usage de la mémoire au fil du temps, alors vous allez probablement utiliser un graphe. Par contre, si vous voulez obtenir le nombre de processus à un instant *t*, alors vous préfèrerez probablement avoir une simple valeur en évidence sur votre tableau de bord. Ces différents éléments sont appelés des *panels* (panneaux en français, même si cela ne sonne pas très bien). Il existe tout un tas de panneaux, comme des tableaux, des listes, des cartes du monde, des camemberts, etc.

Ces panneaux ainsi que les tableaux de bord peuvent être téléchargés via le site officiel et sont produits par la « communauté ».

4.3 Gestion des dashboards

Venons-en donc aux fameux tableaux de bord. Il s'agit d'un ensemble de panneaux qui forment un tout plus ou moins cohérent.

Pour vous donner un exemple, le tableau de bord correspondant à mon serveur me donne :

- des graphes d'utilisation CPU/mémoire/disque/ressources par containers ;
- un instantané du nombre de processus/espace disque, etc.

Ces tableaux de bord peuvent être facilement importés/exportés sous forme de fichier .json.

NOTE

Vous remarquez que la « data source » est souvent indiquée directement dans le fichier .json, ainsi que le nom des interfaces réseaux par exemple. Pensez donc à modifier ce fichier avant de l'importer afin de ne pas avoir à le faire graphiquement (ce qui sera plus contraignant).

CONCLUSION

Félicitations, nous voici arrivés au sommet, et en un seul morceau !

Durant notre ascension, nous avons vu comment tirer profit de technologies assez diverses et récentes pour construire un système de monitoring sur mesure. Je ne prétends pas que ce système est parfait, ni même viable en production. Je l'utilise uniquement sur un serveur personnel et j'ai mis en place cette solution dans le but de me familiariser avec ces technologies. Ayant trouvé l'expérience intéressante, j'ai décidé de la partager avec vous.

Les possibilités sont nombreuses. Il est par exemple possible de surveiller des appareils envoyant des données via SNMP (comme **pfSense** par exemple, ou des **APs Cisco**).

Malheureusement, mettre au point un tel système prend un temps certain. On ne parle clairement pas d'une solution « clef en main » comme du **Zabbix**. C'est de la « bidouille », mais c'est aussi pour cela qu'on aime le faire, n'est-ce pas ? ■

POUR ALLER ENCORE PLUS HAUT…

Comme je le disais précédemment, ce système est très modulaire et les possibilités sont nombreuses, tout comme les améliorations. Vous avez peut-être retenu que nous avions utilisé un léger « hack » avec des points de montage bidons dans le container de Glances pour afficher les partitions manquantes. Il reste cependant un problème avec Glances dans un container, c'est qu'il ne voit pas l'interface réseau physique de notre serveur ! Impossible donc de surveiller en l'état cette interface.

De plus, aucun de nos containers n'est capable (tels qu'ils sont configurés) d'envoyer des e-mails, cela peut poser problème pour envoyer un message de création de compte et/ou oubli de mot de passe depuis Grafana par exemple…

Comme souvent dans ce domaine, le travail n'est jamais fini… et il y aura encore de la place pour améliorer et optimiser au fur et à mesure des installations et/ou des nouveaux outils disponibles.

RÉFÉRENCES

[1] Site officiel de Glances par nicolargo : https://nicolargo.github.io/glances/

[2] Les différents fichiers de configuration sur GitHub : https://github.com/Nesousx/glaraflux

[3] Fonctionnement de l'API SMS Free : http://www.universfreebox.com/article/26337/Nouveau-Free-Mobile-lance-un-systeme-de-notification-SMS-pour-vos-appareils-connectes

[4] Let's Encrypt - le httpS pour tous : https://letsencrypt.org/

[5] Testez votre certificat « SSL » : https://www.ssllabs.com/ssltest/

PRÉSENTATION ET UTILISATION DE LA CARTE STEVAL-3DP001 POUR LE PILOTAGE DES IMPRIMANTES 3D

BRUNO DIRAISON
[Ingénieur firmware et Linux embarqué, utilisateur GNU/Linux depuis 2008]

MOTS-CLÉS : IMPRESSION 3D, EVAL-3DP001, STM32, L6474, OPENSTM32, MARLIN4ST

Avec un doublement des ventes entre 2015 et 2016 [1], l'impression 3D continue sa progression fulgurante. Il n'est pas rare de trouver une imprimante 3D chez un technophile. La baisse du prix des imprimantes entrée de gamme à monter soi-même n'y est pas étrangère. Au cœur de ces imprimantes, une carte de contrôle assure le pilotage des différents organes de l'imprimante. Le but de cet article est de vous guider dans la mise en œuvre de la carte de contrôle open-hardware STEVAL-3DP001, de la société STMicroelectronics.

La société Franco-Italienne **STMicroelectronics**, spécialisée dans la fabrication de composants électroniques, a mis en vente une carte d'évaluation permettant le contrôle des mécaniques d'imprimantes 3D.

Cette carte, développée en France par le centre R&D de STMicroelectronics Grand-Ouest de Rennes, assure le pilotage des moteurs pas-à-pas et des organes chauffants (têtes d'impression, plateau). Elle offre également de nombreuses autres fonctionnalités comme le montre le schéma de la figure 1.

Après une introduction rapide à l'impression 3D, je vais décrire dans cet article la carte **STEVAL-3DP001**, ses fonctionnalités ainsi que sa mise en œuvre dans le cadre du pilotage d'une imprimante de type **Prusa**.

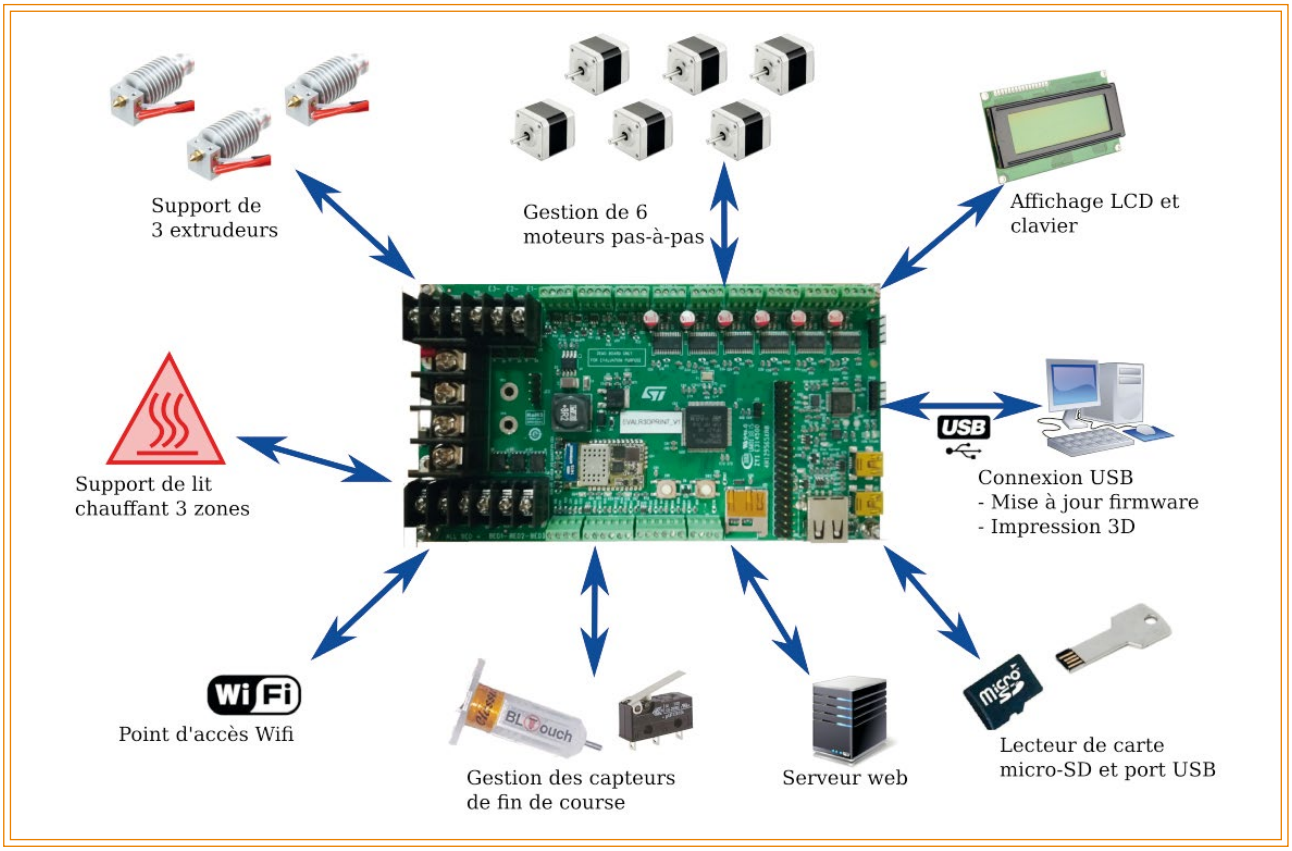


Fig. 1 : Vue générale des différentes interfaces offertes par la carte STEVAL3DP001.

1. HISTOIRE ET FONCTIONNEMENT DES IMPRIMANTES 3D

Les débuts de l'impression 3D datent du milieu des années 1980. La société 3D-Systems a été la première à commercialiser une imprimante 3D en 1988, la SLA-250. Elle utilisait le procédé technique de stéréolithographie.

Toutes les techniques d'impression 3D reposent sur la même méthode : la superposition de tranches fines de matière. La différence se fait ensuite au niveau du procédé utilisé pour la réalisation des tranches de matière comme le montre la figure 2.

Quatre familles de technologies existent :

- la photo polymérisation consiste en la solidification d'un liquide par une lumière, le plus souvent un Laser ;
- la pulvérisation consiste à pulvériser de la matière liquide (ou poudre) ainsi qu'un liant qui va durcir en séchant ;
- la fusion consiste à déposer une fine couche de matière sous forme de poudre. Un laser est ensuite utilisé pour faire fondre et fusionner la poudre aux endroits où on veut de la matière ;
- et enfin la déposition de matière en fusion (*Fused Deposition Modeling* : FDM).

C'est dans cette dernière famille que l'on retrouve le procédé par extrusion qui nous intéresse. Pour cette méthode, on fait fondre un filament de matière qui passe dans une buse d'extrusion et qui est déposé sur l'objet en construction. La tête d'impression se déplace suivant les axes X et Y pour « dessiner » la couche de matière sur la précédente. En refroidissant, cette nouvelle couche va durcir. Pour réaliser une nouvelle couche, soit la tête d'impression va monter, soit le plateau supportant l'objet va descendre.

La carte STEVAL-3DP001 est conçue pour piloter ce dernier type d'imprimante 3D.

Si vous souhaitez en connaître davantage sur les technologies d'impression 3D, je vous invite à vous rendre sur la page dédiée à l'impression 3D sur le site d'**Aniwaa [1]**.

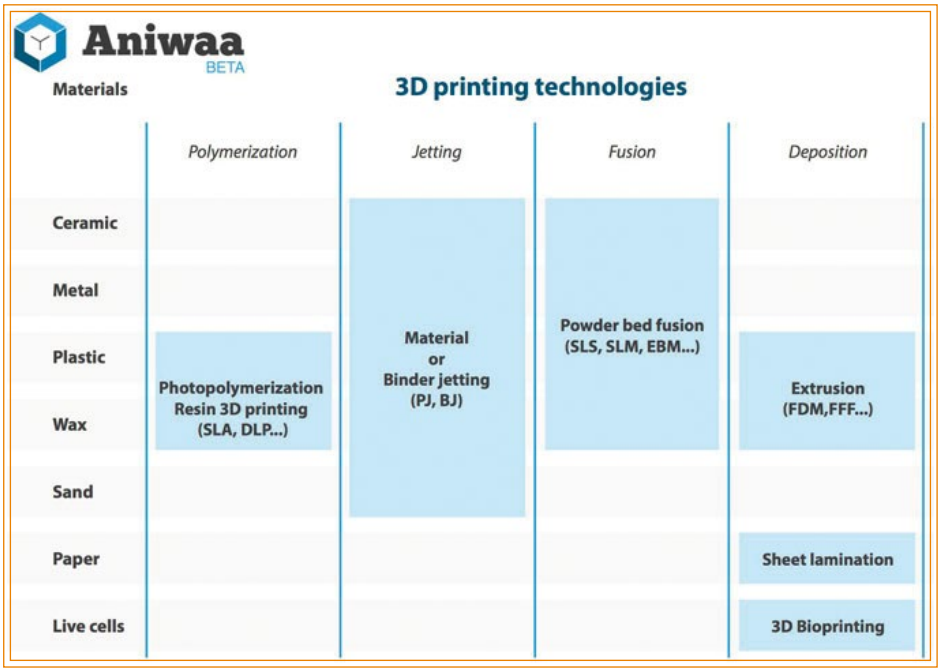


Fig. 2 : Technologies d'impression 3D. Source [1].

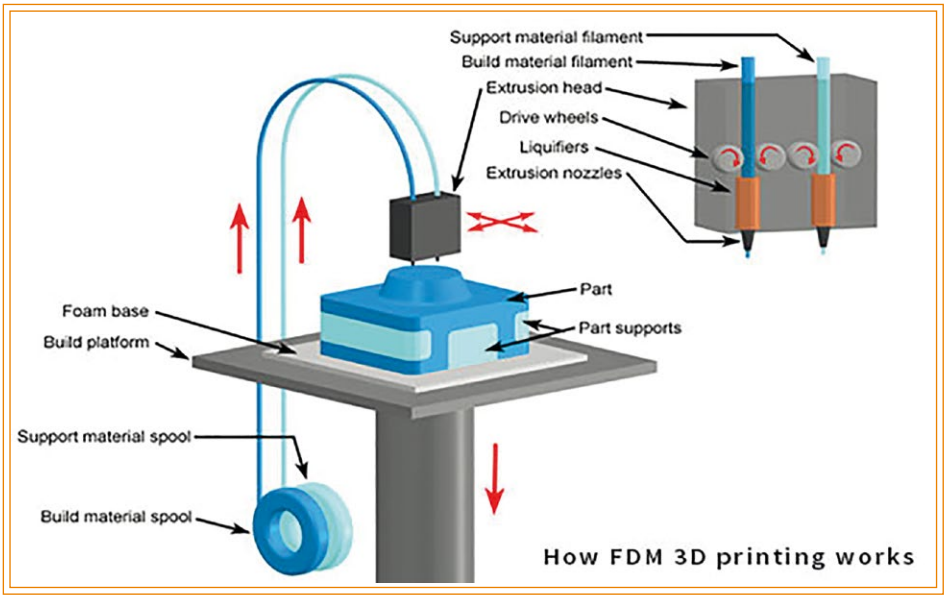


Fig. 3 : Principe de la technologie par déposition de matière en fusion (FDM). Source [10].

2. PRÉSENTATION DE LA CARTE D'ÉVALUATION STEVAL-3DP001

La carte 3DP001 est architecturée autour d'un processeur **ARM Cortex M4** sous la forme d'un **STM32F401**. Elle est alimentée en 12v et peut également recevoir du 24v pour le lit chauffant.

Elle est équipée de 6 contrôleurs de moteur pas-à-pas **STSPIN L6474 [3]**, pouvant délivrer jusqu'à 3A chacun. Le réglage des contrôleurs est entièrement numérique, ce qui assure une meilleure précision par rapport aux composants à réglage analogique.

Elle permet de piloter jusqu'à trois têtes de chauffe, trois ventilateurs de refroidissement et trois zones de chauffe du lit.

Elle intègre un module Wifi permettant de lancer et de contrôler une impression à partir d'un équipement mobile (*laptop*, tablette ou smartphone).

Elle dispose également d'une interface USB master et microSD permettant l'impression d'objet directement à partir de média portable.

Une interface de débuggage, sous la forme d'un **STLink-V2**, est présente. Cette interface offre bien sûr la possibilité de débugger le *firmware* tournant sur la carte à partir de l'IDE **OpenSTM32**, mais elle permet également de charger un nouveau binaire sur la carte par un simple *drag and drop*.

Enfin, un bus d'extension est disponible. Il donne accès à certains GPIO et interfaces (I2C, SPI) du STM32 et permet d'y connecter par exemple un **Raspberry Pi**, sur lequel on aura préalablement installé **Octoprint**. La mise en place d'un tel système fera l'objet d'un prochain article.

Les différentes interfaces de cette carte sont représentées par la figure 4.

NOTE

À noter qu'il s'agit d'une carte Open Hardware. Le schéma, la BOM ainsi que les fichiers Gerber sont fournis et peuvent être librement réutilisés ou modifiés. Ils sont disponibles en suivant le lien **[2]**.

3. MISE EN ŒUVRE DE LA CARTE STEVAL-3DP001

Maintenant que nous avons décrit les différentes possibilités de la carte STEVAL-3DP001, nous allons nous attaquer au câblage de la carte.

3.1 Pilotage par USB ou par UART

Dans un premier temps, une « configuration » matérielle de la carte peut être nécessaire. En effet, la carte intègre un convertisseur UART-USB dans le ST-Link, permettant le pilotage à partir d'un PC connecté au port USB esclave. Par défaut, ce convertisseur est utilisé. Si vous souhaitez piloter la carte directement via l'UART, pour la connecter au port série d'un Raspberry Pi par exemple, ou pour simplement augmenter la vitesse de transfert, il vous faudra enlever deux shunts (résistances de 0Ω), marquées R35 et R36. L'emplacement de ces deux résistances est indiqué sur la figure 5.

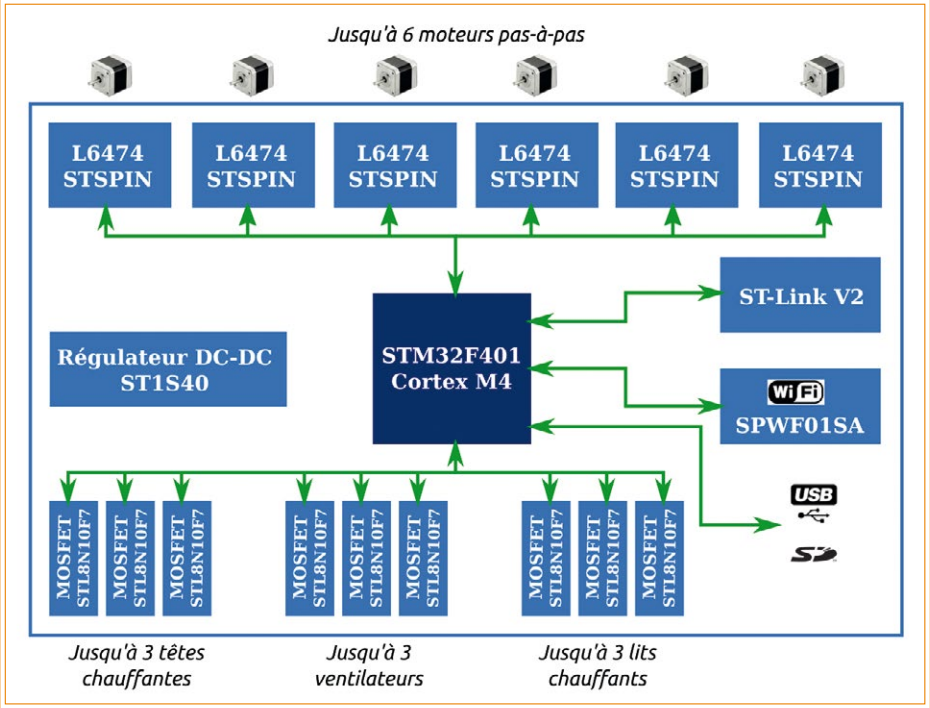


Fig. 4 : Architecture fonctionnelle de la carte STEVAL3DP001V1. Source [2]

3.2 Câblage de la carte STEVAL-3DP001

La figure 6 présente le câblage de la carte STEVAL-3DP001 pour une imprimante du type Prusa i3, équipée d'une tête d'extrusion, d'un lit chauffant à une zone, d'un ventilateur de refroidissement de pièce, de deux capteurs de fin de course pour les axes X et Y, et d'un capteur d'auto-positionnement pour l'axe Z (auto-leveling).

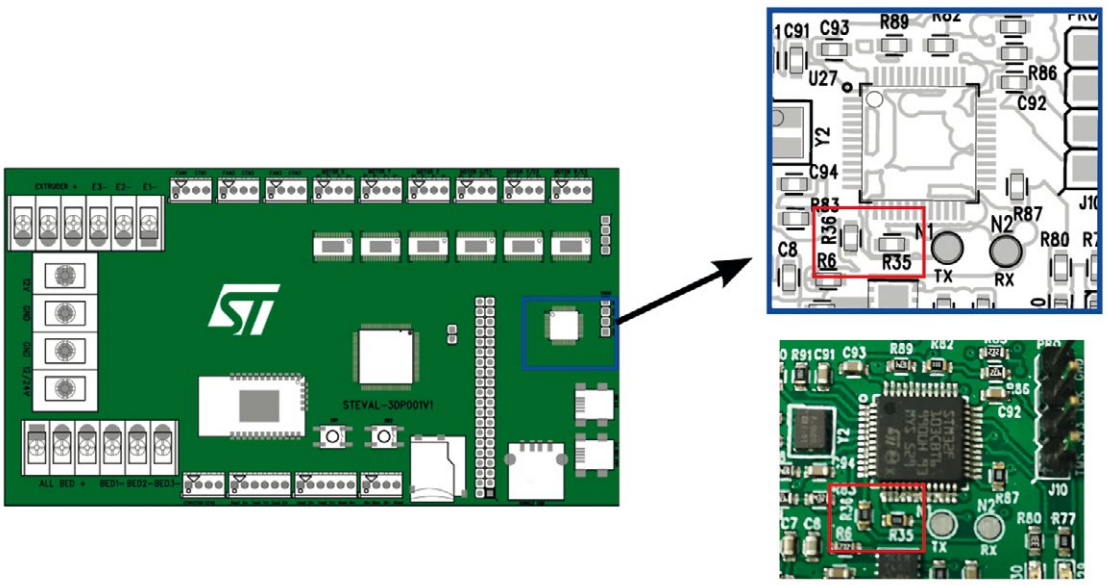


Fig. 5 : Localisation des résistances R35 et R36.

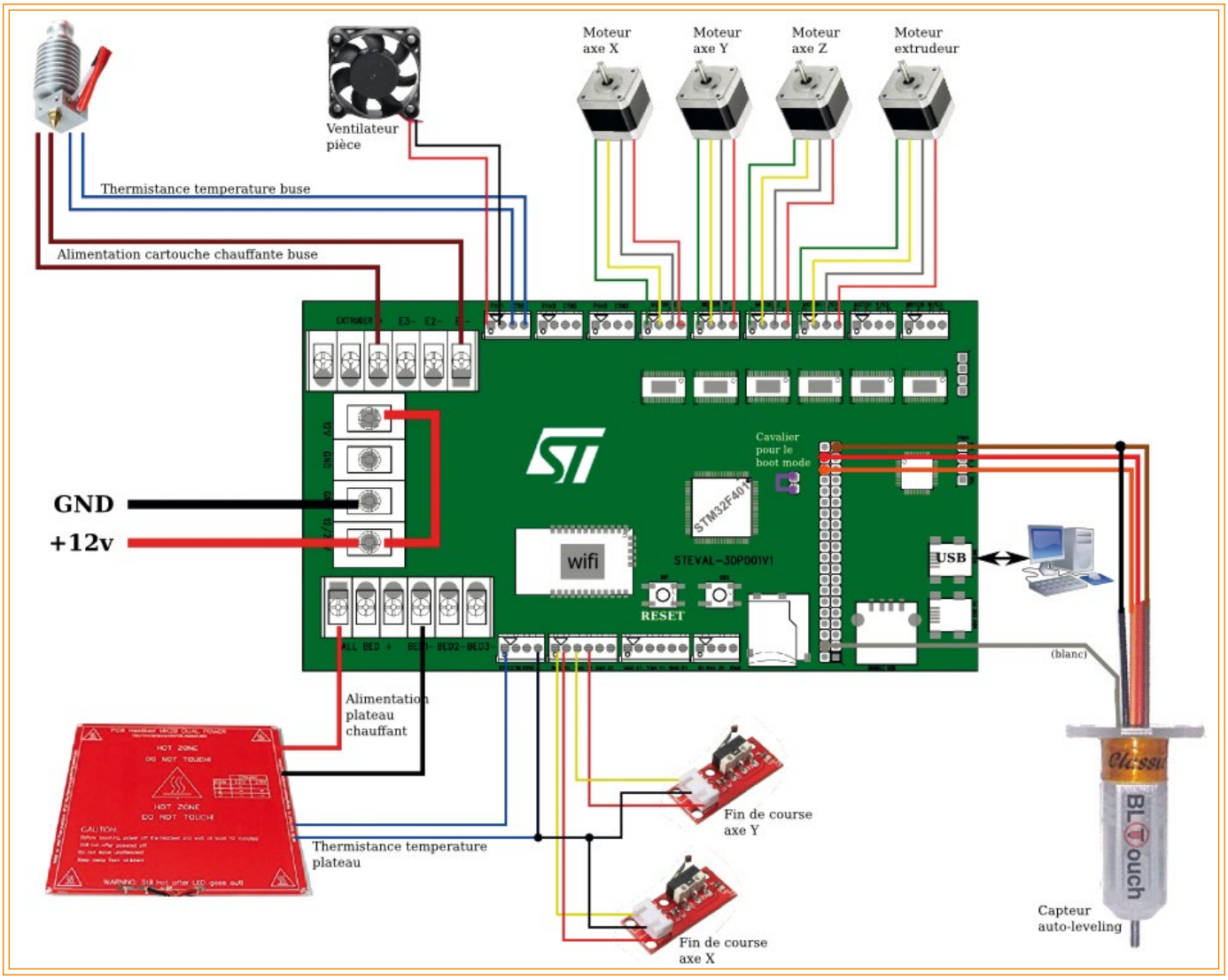


Fig. 6 : Câblage de la carte STEVAL3DP001.

Cette carte est également équipée d'un connecteur d'extension, dont le brochage est donné par la figure 7. La grande majorité des entrées/sorties présentes sur les borniers à vis de la carte sont présentes sur ce connecteur. On y retrouve également les signaux de la carte SD et 4 GPIOs laissés libres pour l'utilisateur (**USER_1** à **USER_4**).

3.2.1 Moteurs pas-à-pas

Le câblage des moteurs pas-à-pas ne présente aucune difficulté. Il faut juste veiller à ne pas se tromper dans la polarité des enroulements (A-, A+, B-, B+). Cette polarité est sérigraphiée sur le PCB, et un coup d'œil à la documentation du moteur vous permettra de faire le lien avec la couleur des fils.

3.2.2 Tête d'impression et plateau chauffant

Là non plus, aucune difficulté à signaler. Les thermistances ainsi que la cartouche de chauffe de la tête d'impression ne sont pas polarisées. Ce n'est pas le cas pour le plateau chauffant.

Il est intéressant de signaler que le pilotage des alimentations de chauffe du plateau ainsi que de la tête d'impression se fait par l'intermédiaire de MOSFET reliés à la masse. La tension d'alimentation en +12v (ou +24v pour le plateau) est donc toujours présente.

La figure 6 présente le câblage dans le cas d'une alimentation unique en 12v. Si vous disposez d'une alimentation à double sortie 12v et 24v, alors vous pouvez

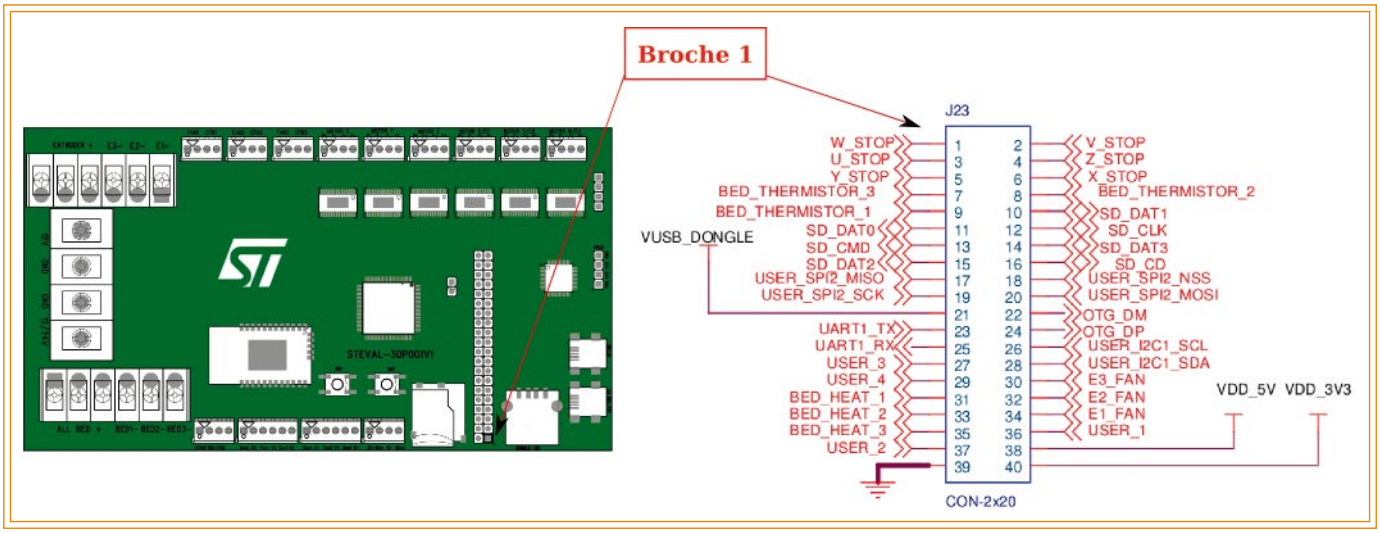


Fig. 7 : Brochage du connecteur d'extension de la carte EVAL3DP001.

alimenter le plateau en 24v, ce qui permet une montée en température plus rapide. Le plateau prend en effet plus de temps que la tête d'impression pour monter en température.

3.2.3 Capteurs de fin de course

Les capteurs de fin de course utilisés sont les mêmes que ceux présents sur les imprimantes Prusa. Le connecteur est constitué de 3 fils :

- fil rouge : alimentation du capteur en 3v3 (tension logique de la carte EVAL-3DP001) ;
- fil noir : masse ;
- fil jaune : signal de fin de course.

Le câblage est indiqué sur la figure 6.

3.2.4 Capteur de l'axe Z

Le capteur sur l'axe Z permet au *firmware* de réaliser une correction automatique de la perpendicularité du lit chauffant par rapport à l'axe Z de l'imprimante. Si le lit chauffant n'est pas entièrement perpendiculaire, le logiciel intègrera une correction aux déplacements sur l'axe Z en fonction de la position de la tête d'impression en X et Y.

Pour pouvoir effectuer cette correction, le logiciel va, avant de commencer l'impression, effectuer un ensemble de mesures en différents points du plateau. Le nombre de points de la grille est défini dans le fichier de configuration. La position du capteur d'*auto-leveling* par rapport à la tête d'impression ainsi que le type de capteur doivent également être configurés dans ce fichier.

Il existe différents types de capteurs d'*auto-leveling* : *microswitch*, induction, BLTouch, IR, etc. Dans mon cas, j'utilise un BLTouch de la société **Antclabs [4]**. Il s'agit d'un système fonctionnant par contact mécanique. Il est à la fois précis et, surtout, indépendant de la nature du matériel du lit de chauffe. Dans le cas de mon imprimante, j'ai positionné au-dessus du plateau chauffant une vitre en verre sur laquelle est imprimée la pièce. L'utilisation d'un système par induction n'est donc pas possible pour moi.

Pour ce qui est de l'installation du capteur BLTouch sur la carte, on le connecte tout simplement au port d'extension. Mais avant cela, il faut vérifier que le capteur est bien configuré en logique 3v3. Sur le PCB du capteur BLTouch, la liaison « LOGIC » ne doit pas être connectée, comme le montre la figure 8. La page dédiée au BLTouch **[4]** sur le site Antclabs contient la notice d'utilisation.

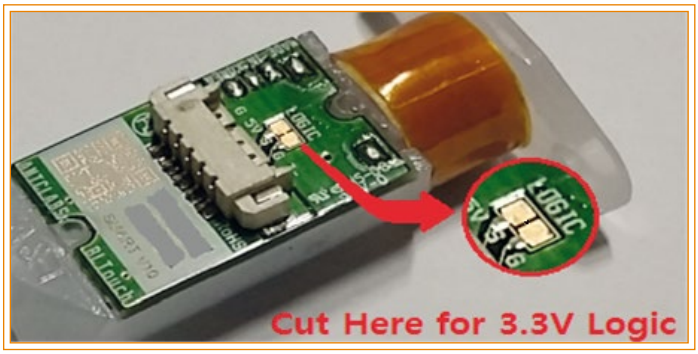


Fig. 8 : Configuration de la logique 3v3 du BLTouch. Source [4].

La connectique du capteur BLTouch se compose de 5 fils séparés en 2 groupes.

Le premier groupe de 3 fils (marron, rouge et orange) est utilisé pour l'alimentation et le contrôle du capteur. Ce dernier est contrôlé de la même manière qu'un servomoteur, grâce à un signal PWM.

Le second groupe composé de 2 fils (noir et blanc) donne le signal de contact avec le plateau et sert également à informer la carte d'une erreur dans le capteur (alarme).

La connexion du capteur sur le connecteur d'extension de la carte est donnée par le tableau suivant :

Couleur du fil du capteur BLTouch	Description	Correspondance sur le connecteur d'extension
Marron	Masse (GND)	Broche 39
Rouge	+5v	Broche 38
Orange	Signal de contrôle	Broche 36
Noir	Masse (GND)	Broche 39
Blanc	Signal Zmin	Broche 4 (correspond au Z_Stop)

Nous en avons maintenant terminé avec le câblage de la carte, il est temps de passer à la partie logicielle avec en premier lieu la mise en place de l'environnement de développement.

4. MISE EN PLACE DE L'ENVIRONNEMENT OPENSTM32

4.1 Installation d'OpenSTM32

OpenSTM32 est un outil de développement intégré (IDE) basé sur **Eclipse**. Il est disponible pour les trois OS les plus répandus, à savoir **Linux**, **Mac OS** et **Windows**. Je ne traiterai ici que de la partie installation sur Linux. Le système hôte utilisé est un Ubuntu 14.04.5 LTS.

L'IDE OpenSTM32 est disponible sur http://www.openstm32.org/. Pour pouvoir récupérer l'outil, il faut en premier lieu se créer un compte. Une fois cette étape passée et après vous être authentifié, il vous est possible d'accéder à certaines parties du site, et en particulier la page de documentation du projet « System Workbench for STM32 ». Sur cette page vous trouverez un lien sur l'installeur qui nous intéresse. Dans le cadre de cet article, la version v1.8 est utilisée.

L'installation de l'IDE ne devrait vous poser aucun problème. L'installeur dispose d'une interface graphique permettant de suivre la progression de l'installation. Il va également créer les règles udev pour permettre au débugger de se connecter au STLink-v2 de la carte en USB.

Une fois installé, et avant d'ouvrir l'IDE OpenSTM32, nous allons récupérer le code source du *firmware* **Marlin4St**, disponible sur **GitHub**.

4.2 Installation des sources Marlin4ST

Le *firmware* Marlin est le *firmware* officiel développé pour les imprimantes de la famille **RepRap**. Les informations relatives à ce *firmware* peuvent être trouvées à l'adresse **[5]**.

L'objectif principal de ce *firmware* est de transformer des commandes **G-Code [6]** en actions mécaniques. Il offre également d'autres fonctionnalités (affichage sur LCD, lecture de fichier G-Code à partir d'une carte SD ou d'une clé USB, etc.) au risque de s'éloigner de sa fonction première et de gagner en lourdeur. Ce *firmware* a été conçu pour tourner sur des cartes de la famille Arduino, utilisant des processeurs 8 bits.

La dernière version en date de ce firmware est la 1.1.0-RC8. Elle est disponible sur **GitHub [7]**. Le *firmware* Marlin4ST est lui basé sur la version 1.1.0-RC7 qui date de juillet 2016.

Le code source de Marlin4ST est également disponible sur GitHub **[8]** :

```
$ git clone  https://github.
com/St3dPrinter/Marlin4ST.git
```

4.3 Importation et compilation du projet sous OpenSTM32

Il reste maintenant à importer le projet sous OpenSTM32.

Pour rappel, OpenSTM32 est basé sur l'IDE Eclipse. La première chose qui va vous être demandée est donc de choisir le *workspace* du projet. Cette étape est primordiale. En effet, les chemins d'accès au code source sont tous relatifs au *workspace*. Le choix d'un mauvais chemin pour le *workspace* induira donc une incapacité à compiler le *firmware*.

Le répertoire **SW4STM32** doit être choisi comme *workspace*.

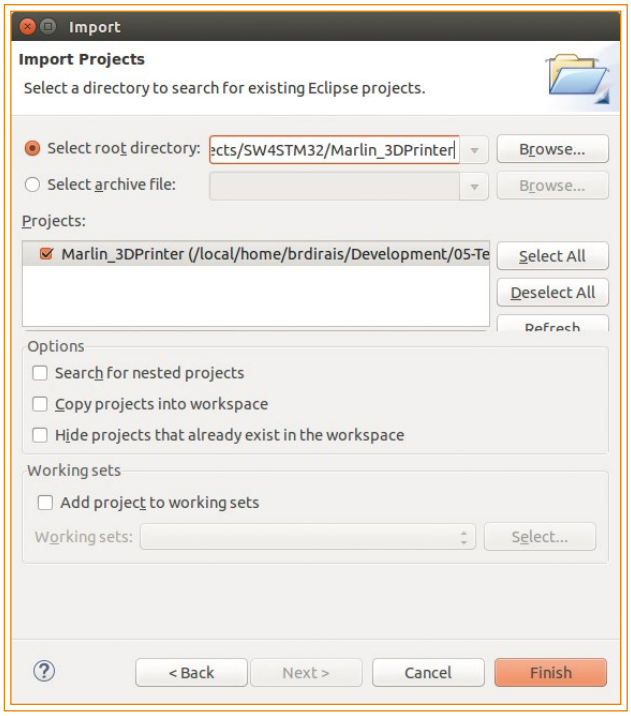


Fig. 9 : Fenêtre d'importation du projet.

Une fois l'IDE ouvert, dans la fenêtre ***Project Explorer***, ouvrez le menu contextuel par un clic droit et choisissez ***Import...***.

Le type de source à importer est ***General > Existing Projects into Workspace***.

Dans la fenêtre ***Import***, choisir comme **`root directory`** le répertoire **`Marlin_3DPrinter`**. Vous devez ensuite sélectionner le projet *Marlin_3DPrinter*. Pour importer le projet, cliquez sur ***Finish***.

Le projet est maintenant visible dans l'onglet ***Project Explorer*** comme présenté par la figure 10.

Il ne reste plus qu'à générer le binaire pour la cible qui nous intéresse et à le charger sur cette cible.

Pour cela, nous allons d'abord effectuer un *clean* du projet par un clic droit sur le nom du projet et sélectionner ***Clean Project***. La compilation du code source suit la même procédure, mais cette fois, il faut sélectionner ***Build Project***. Au préalable, vous aurez pris soin de choisir le type de compilation désiré : soit ***Debug***, soit ***Release***, via le menu ***Build Configurations > set Active***.

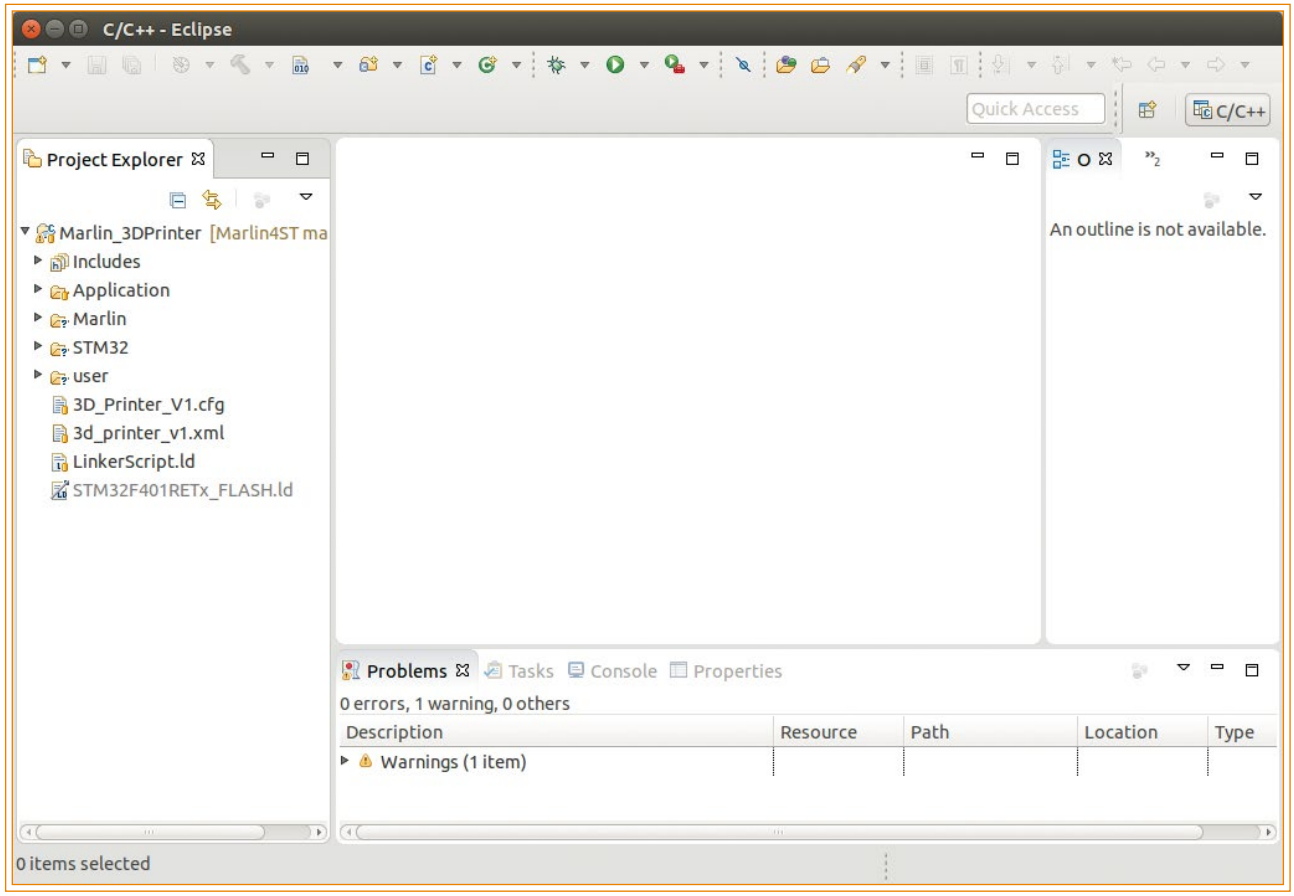


Fig. 10 : Workspace Eclipse avec le projet Marlin_3DPrinter.

Fig. 11 : Console Eclipse après compilation.

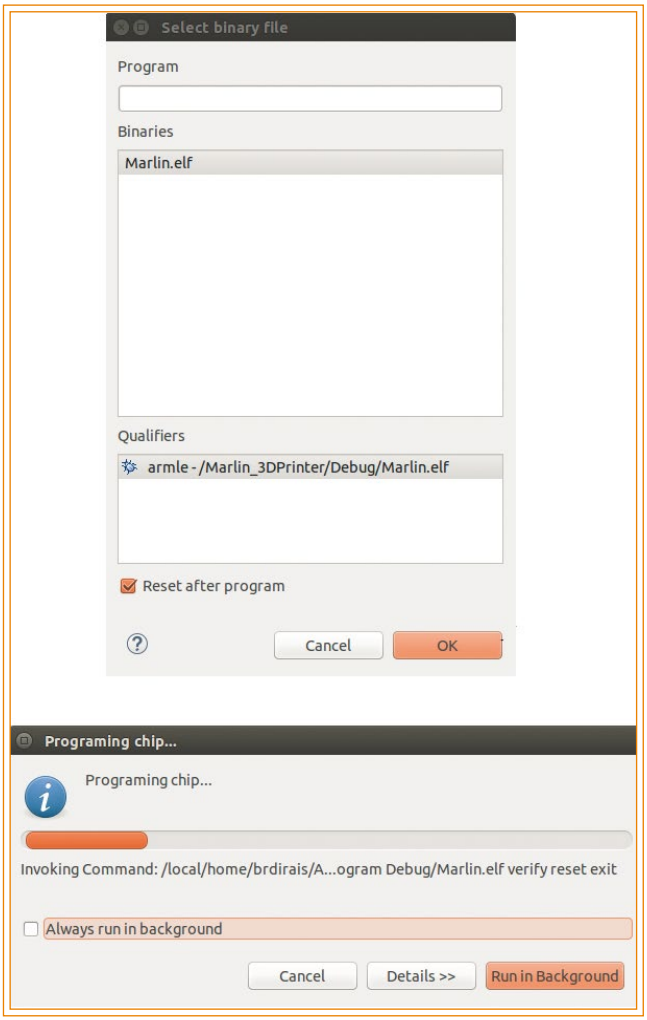


Fig. 12 : Chargement binaire par l'IDE Eclipse.

À la fin de la compilation, qui doit durer entre 10 et 20 secondes, le compilateur vous indique qu'il a généré les binaires **`Marlin.elf`** et **`Marlin.bin`**, comme le montre la figure 11.

NOTE

Attention : Entre les différentes étapes (*clean*, *build*), pensez à effectuer un ***Refresh*** pour qu'Eclipse prenne bien en compte les changements au niveau des fichiers.

4.4 Chargement du binaire sur la carte et débuggage

Il existe deux solutions pour charger le binaire sur la carte : le *drag and drop* et le chargement via Eclipse.

4.4.1 Le drag and drop.

Une fois la carte EVAL-3DP001 connectée par USB à l'hôte, un nouveau disque va être monté. Ce disque virtuel permet de charger sur la carte le *firmware* par un simple *drag and drop*. Pour cela, il suffit de prendre le binaire fraîchement créé (**`Marlin.bin`**) qui est présent dans le répertoire **`Debug`** (ou **`Release`**) et de le copier (copier-coller ou *drag-and-drop*) dans le disque virtuel de la carte, qui porte le nom de **`STPRINTER`**.

Pour que le nouveau *firmware* soit pris en compte, il faut effectuer un *power off/on* complet de la carte en débranchant toutes les alimentations, câble USB compris.

Une fois l'alimentation de la carte rétablie, le fichier **`Marlin.bin`** doit avoir disparu du répertoire virtuel. Cela signifie que le binaire a bien été flashé dans le stm32 de la carte.

4.4.2 Le chargement via l'IDE4

L'IDE OpenSTM32 offre la possibilité de directement charger le fichier `Marlin.elf` généré, en mémoire flash du stm32.

Pour cela, une fois la compilation terminée, il suffit, dans le menu contextuel du projet, de sélectionner ***Targe > Program chip...*** et de choisir le bon fichier `Marlin.elf` (au cas où une version debug et une version release existent). Le flashage du stm32 va ensuite commencer.

5. ANALYSE DU CODE SOURCE

5.1 Présentation de l'architecture firmware

Le code source du *firmware* est structuré en couches, comme présenté par la figure 13.

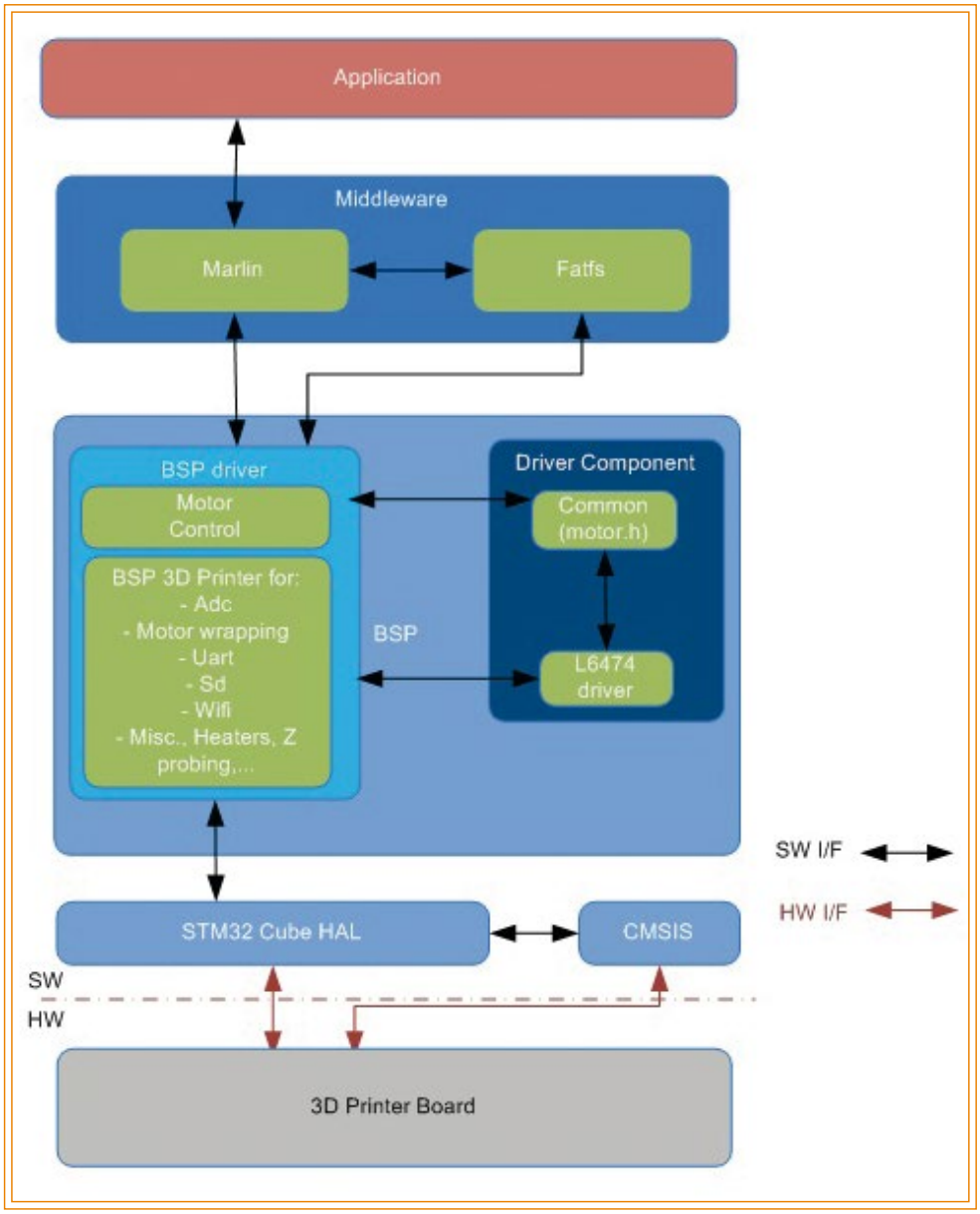


*Fig. 13 : Architecture du firmware. Source **[9]**.*

Au plus près du *hardware* (matériel), on trouve le **CMSIS** et le **HAL** (STM32F4xx_HAL_Driver). Le CMSIS (*Cortex Microcontroller Software Interface Standard*) contient les fonctions permettant d'initialiser les registres du cœur ARM lors du démarrage. Le HAL (*Hardware Abstraction Layer*) est spécifique au processeur utilisé. Dans notre cas, un STM32F4xx. Ce module contient les API génériques permettant d'accéder à tous les périphériques supportés par les processeurs de la famille STM32F4xx. Il peut être adressé directement, mais est surtout utilisé par le BSP pour offrir à l'application une interface de haut niveau.

Vient ensuite le BSP (*Board Support Package*). Cette couche offre une API de haut niveau permettant d'adresser directement le matériel présent sur la carte. Le répertoire `BSP/STM32F4xx-3dPrinter` contient les méthodes offertes par cette API. On y trouve la gestion des moteurs pas-à-pas, de la carte SD, du Wifi, de l'UART, etc. Le driver de bas niveau pilotant les contrôleurs de moteurs pas-à-pas se trouve sous `BSP/Components/l6474`. En fonction des moteurs utilisés, il vous faudra venir configurer à ce niveau la puissance maxi de ces moteurs.

Au-dessus du BSP on trouve la couche *middleware* contenant le code Marlin adapté pour la carte STEVAL-3DP001 ainsi qu'un gestionnaire de Fat. Dans le répertoire Marlin, le nom des différents fichiers est assez explicite quant à leurs rôles respectifs. Vous trouverez dans ce répertoire certains

fichiers apparaissant sous Eclipse en grisé. Cela signifie qu'ils ont été sortis du projet. Ces fichiers devraient tout logiquement disparaître dans une version future.

5.2 Structure du projet

La structure du projet apparaissant sous Eclipse est présentée en figure 14. On y trouve les modules BSP, CMSIS et HAL sous le répertoire **`STM32/Drivers`** et le module **Fatfs** sous le répertoire **`STM32`**. Le module Marlin va lui être situé à la racine du projet.

On y trouve d'autres dossiers qui sont :

- Le dossier **`Binaries`** contient un binaire généré par STMicroelectronics pour un test rapide de la carte ;
- Les dossiers **`Debug`** et **`Release`** sont les dossiers de génération des binaires par l'IDE OpenSTM32 ;
- Le dossier **`Includes`** contient les liens vers les différents répertoires d'*include*. Le lien vers ces répertoires est configuré dans les préférences du projet, via l'onglet ***C/C++ Build > Settings*** ;
- Le dossier **`Application/SW4STM32`** est l'emplacement où sont stockés les paramètres du projet. Vous n'aurez pas à y toucher ;
- Le dossier **`Application/User`** contient le point d'entrée de l'application **`main()`** dans le fichier **`main.c`**. On y trouve également la configuration du *system clock* (**`clock_f4.c`**) ainsi que la gestion des interruptions (*stm32f4xx_it.c*).

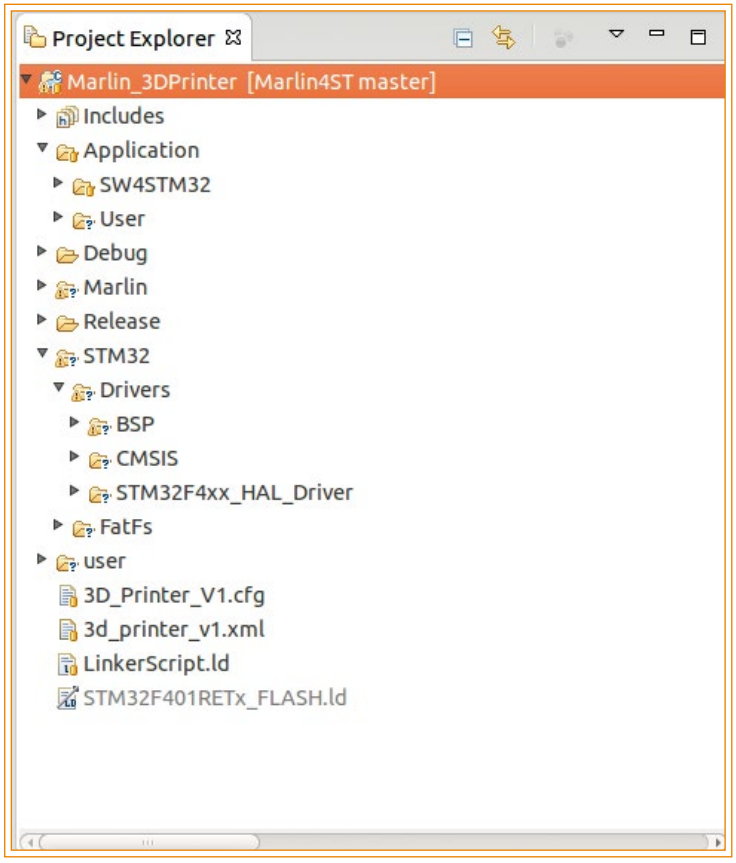


Fig. 14 : Structure du projet dans l'IDE Eclipse.

6. CONFIGURATION

La configuration du projet dépend du matériel que devra piloter la carte STEVAL-3DP001. Elle s'opère en trois endroits :

6.1 Dans le fichier de configuration du projet

Les options de compilation vont permettre de sélectionner certaines fonctionnalités. Comme le projet contient du code **C++** (partie Marlin) et du code **C** (tout le reste), les onglets ***Symbols*** du ***MCU GCC Compiler*** et ***Preprocessor*** du ***MCU G++ Compiler*** doivent être complétés. Le tableau suivant donne la signification de chacun de ces *switchs*.

6.2 Dans le fichier l6474_target_config.h

Ce fichier, localisé dans **`Drivers/BSP/Components/l6474`**, permet de configurer les pilotes des moteurs pas-à-pas utilisés dans l'imprimante. En fonction des moteurs que vous utilisez, vous serez probablement amenés à venir changer la configuration.

On y trouve en particulier les constantes **`L6474_CONF_PARAM_TVAL_DEVICE_x`** qui définissent le courant utilisé par chaque moteur, et les constantes **`L6474_CONF_PARAM_OCD_TH_DEVICE_x`** qui définissent le courant maximal à ne pas dépasser par chaque moteur.

6.3 Dans le répertoire Marlin

Le fichier **`temperature.h`** contient les tables de température permettant une conversion entre la valeur analogique reçue du capteur et la température réelle. Normalement, vous devriez trouver la table correspondant à vos capteurs dans ce fichier, et dans ce cas le nom de cette table doit être reporté

Symbole	Description
`USE_HAL_DRIVER`	Utilisé pour activer le driver HAL du STM32.
`STM32F401xE`	Permet de choisir le processeur à utiliser. Utilisé en particulier par le HAL.
`STM_3DPRINT`	Permet de désactiver les parties de Marlin non utilisées par la carte STEVAL-3DP001.
`MARLIN`	Les drivers BSP et HAL peuvent être utilisés en association avec d'autres *firmwares* que Marlin. Dans le cas d'une utilisation avec Marlin, il faut l'indiquer.
`WIFI`	Permet d'activer ou non la fonctionnalité Wifi.
`SD_WIDE_BUS`	Fixe la taille du bus de la carte SD. Pour une utilisation en 4 fils, ce *switch* doit être utilisé. S'il n'est pas positionné, la SD fonctionnera en 2 fils.
`WAIT_FOR_RPI`	Dans le cas d'une utilisation de la carte avec un pilotage par un Raspberry Pi, ce *switch* permet d'attendre le Raspberry Pi au démarrage. Cette attente peut être d'une durée fixe de 30 secondes ou peut se faire sur détection du Raspberry Pi (voir `WITH_RPI_DETECTION`).
`WITH_RPI_DETECTION`	Permet d'attendre un signal en provenance du Raspberry Pi (`gpio`) avant de démarrer le *firmware*. Le *switch* `WAIT_FOR_RPI` doit être activé.

dans le fichier `Configuration.h`. Si ce n'est pas le cas, vous aurez à générer par vous-même une table de température.

Les fichiers `Configuration.h` et `Configuration_adv.h`, présents sous Marlin, permettent une configuration précise du *firmware* en fonction de votre mécanique. On y trouve la direction des moteurs X, Y et Z ; le nombre et la position des capteurs de fin de course, etc. L'explication de ces deux fichiers fera l'objet d'un prochain article.

CONCLUSION

Nous arrivons au terme de cet article. Son objectif était de vous faire découvrir la carte d'évaluation STEVAL-3DP001. J'espère qu'il a répondu à vos attentes et vous a également permis de découvrir l'environnement de développement OpenSTM32, basé sur l'IDE Eclipse et dédié aux plateformes de la société STMicroelectronics. ■

RÉFÉRENCES

[1] Page dédiée aux technologies d'impression 3D sur le site d'*Aniwaa* : http://www.aniwaa.fr/les-technologies-dimpression-3d

[2] Site officiel de STMicroelectronics. Page dédiée à la carte STEVAL-3DP001 : http://www.st.com/content/st_com/en/products/evaluation-tools/solution-evaluation-tools/computer-and-peripherals-solution-eval-boards/steval-3dp001v1.html

[3] Pilote de moteur pas-à-pas L6474 sur le site de STMicroelectronics : http://www.st.com/en/motor-drivers/l6474.html

[4] Page BLTouch sur le site ANTCLABS : https://www.antclabs.com/bltouch

[5] Site officiel de Marlin : http://marlinfw.org/

[6] Page dédié au G-Code sur le site officiel de *RERAP :* http://reprap.org/wiki/G-code/fr

[7] Page Marlin sur GitHub : https://github.com/MarlinFirmware/Marlin

[8] Page Marlin pour la carte STEVAL-3DP001 sur GitHub : https://github.com/St3dPrinter/Marlin4ST

[9] Site officiel de STMicroelectronics. Page dédiée au firmware de la carte STEVAL-3DP001 : http://www.st.com/content/st_com/en/products/embedded-software/evaluation-tool-software/stsw-3dp001.html

[10] Page dédiée à la technologie d'impression 3D sur le site de Kul 3D : http://www.kul3d.com/what-is-fdm-3d-printing/

VOUS DONNEZ DES RENDEZ-VOUS ET ON VOUS OUBLIE ? ENVOYEZ AUTOMATIQUEMENT DES SMS DE RAPPEL !

TRISTAN COLOMBO

MOTS-CLÉS : RASPBERRY PI, GOOGLE CALENDAR, SMS, CLÉ USB 3G

Lorsque l'on prend régulièrement des rendez-vous, il arrive que parfois les gens vous oublient... et quand on a peu de temps, c'est assez pénible ! Pourquoi donc ne pas mettre en place une solution envoyant automatiquement un SMS pour être sûr que votre rendez-vous sera honoré ? Nous profiterons de l'utilisation des SMS pour voir également comment mettre en place un système d'authentification deux facteurs.

Le point de départ de cet article n'est pas issu d'une problématique qui m'est propre, mais d'une réflexion permettant de résoudre un problème rencontré par bon nombre de professions libérales (psychologues, médecins, etc.) : comment éviter un maximum de fixer des rendez-vous qui ne seront pas honorés ? Des solutions existent déjà sur le marché et vous avez déjà sans doute reçu des SMS de rappel d'un centre médical ou encore d'un hôpital. Toutefois, chacune de ces solutions est propriétaire et vous devrez vous acquitter d'un forfait de SMS ou d'un coût au SMS envoyé. En dehors du fait qu'en adoptant l'une de ces solutions vous êtes lié au fournisseur du service, estimons le coût de mise en place de ce système sur un an dans le cadre d'un exemple pratique.

Supposons que nous nous trouvions dans un petit centre médical hébergeant 4 professionnels et que chaque professionnel donne 7 rendez-vous par jour (ce qui est un minimum puisque pour un kinésithérapeute, les séances durent moins d'une heure). Nous arrivons donc à 6720 SMS par an (4 professionnels x 7 rendez-vous par jour x 5 jours par semaine x 4 semaines par mois x 12 mois dans l'année). Par mois, cela

fait 560 SMS (important pour la suite). J'ai pris au hasard les deux premières solutions que j'ai trouvées et dont le nom sera modifié : **Pianotier** qui permet d'automatiser des actions sur différentes applications et **SMSSystem** qui fournit un service lié à **Google Calendar**. Pianotier permet d'utiliser un compte gratuit jusqu'à 100 SMS par mois... nous sommes donc largement au-delà et nous devrons prendre un abonnement à 20$ par mois (soit environ 17€ par mois et donc 204€ par an). Avec SMSSystem, il faudra payer 0,045€ HT par SMS. Tous les professionnels de notre exemple ne peuvent pas tous déduire la TVA, mais nous conserverons tout de même le montant HT pour nos calculs. Sur l'année, SMSSystem reviendra à 302,40€ et il est donc bien plus cher que Pianotier. Penchons-nous maintenant sur notre solution.

Dans cet article, je vais installer un serveur **Raspberry Pi** sous **Raspbian** équipé d'un dongle USB 3G, mais la démarche est bien entendu la même sur n'importe quel PC sous **Debian**. Tous les soirs, un programme ira lire le contenu d'un agenda de Google Calendar pour le lendemain et, en fonction de ce qu'il lira comme rendez-vous, enverra un SMS de rappel (un rendez-vous devra nécessairement suivre un format spécifique pour indiquer un numéro de téléphone). Avant de nous lancer dans les aspects techniques, estimons le coût de notre réalisation : un Raspberry Pi (dans le cadre de cet article, il s'agit d'un Raspberry Pi 3 à 40€ puisque c'est ce que je possédais, mais on peut en utiliser un moins puissant et moins cher voire utiliser un serveur PC déjà en fonction), un chargeur 5V quelconque (environ 10€), une clé USB 3G (en l'occurrence ici une **Huawei E169** à 37€), une carte SIM jumelle (gratuite selon les abonnements) et enfin un câble réseau dont le coût dépendra en fonction de la longueur. Nous arrivons environ à 87€ en marge haute et 37€ en marge basse (il y a toujours un Raspberry Pi et une alimentation qui traînent quelque part ou vous pouvez utiliser un serveur PC). Nous voyons bien que notre solution est la plus rentable, d'autant plus sur le temps comme le montre le graphe de la figure 1.

L'aspect financier ne laissant plus planer l'ombre d'un doute sur la justification de notre démarche, passons au plus intéressant : l'aspect technique et la mise en pratique ! Et puisque nous allons manipuler des SMS, nous en profiterons pour faire une petite digression et voir comment réaliser une authentification deux facteurs (qui n'a absolument rien à voir avec le projet initial, mais qui peut toujours servir...).

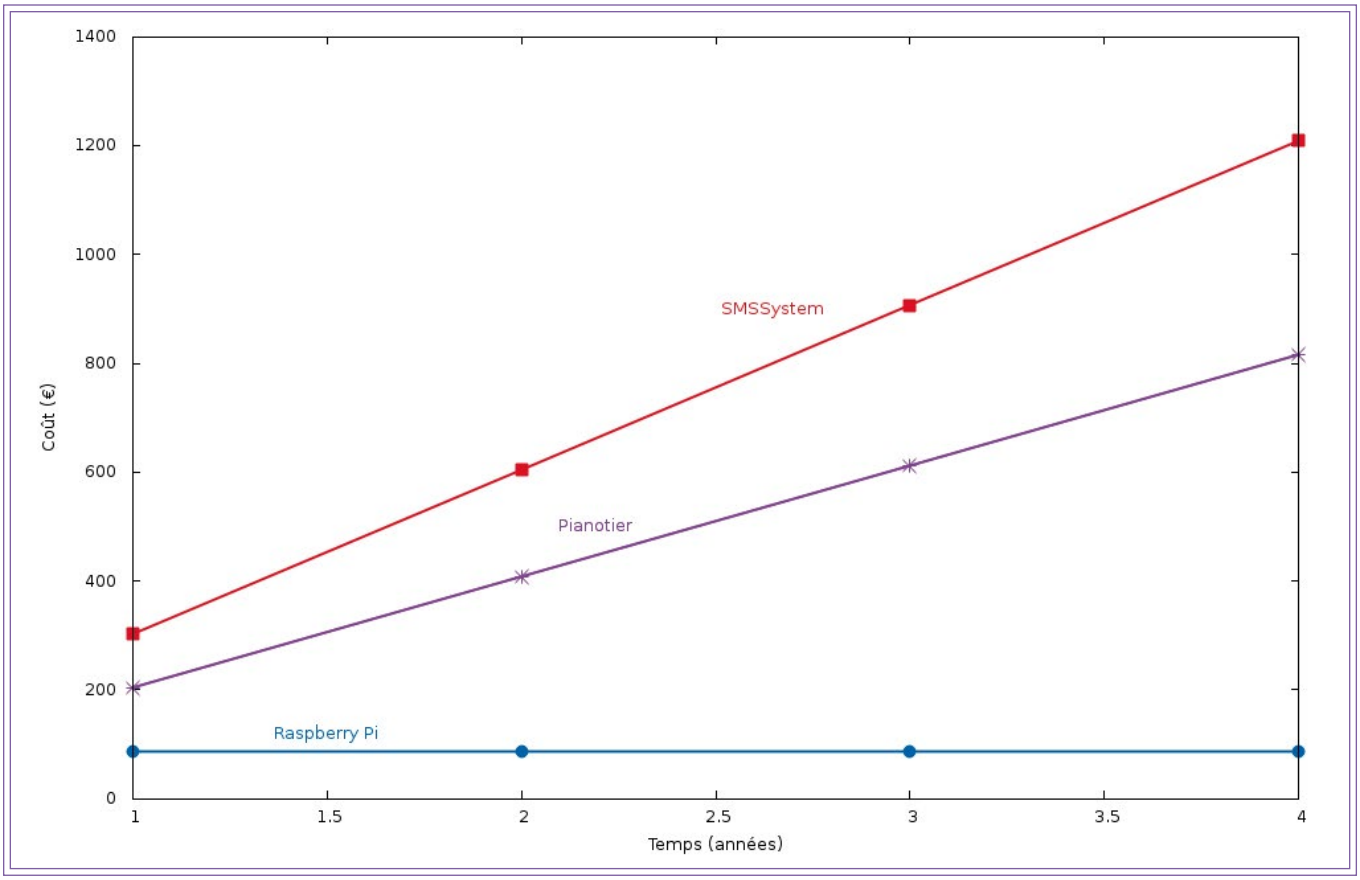


Fig. 1 : Données cumulées du coût de chaque solution en fonction du nombre d'années.

NOTE

Le graphe de la figure 1 a été réalisé avec **GnuPlot**. Je détaille ici rapidement comment utiliser cet outil fort pratique et qui permet de générer simplement de petits graphes.

Les données sont préparées dans un fichier **money.dat** :

```
# Raspberry Pi (index 0)
  1    87
  2    87
  3    87
  4    87


# SMSSystem (index 1)
  1    302,4
  2    604,8
  3    907,2
  4    1209,6


# Pianotier (index 2)
  1    204
  2    408
  3    612
  4    816
```

Notez bien les deux lignes vides entre chaque bloc de données : c'est ce qui va permettre de déterminer des index et de traiter les données indépendamment.

Et le script de génération du graphe, **graph.plt**, contient :

```
01: set xlabel 'Temps (années)'
02: set ylabel 'Coût (€)'
03:
04: unset key
05:
06: set style line 1 lc rgb '#0060ad' lt 1 lw 2 pt 7 ps 1.5   # --- bleu
07: set style line 2 lc rgb '#dd181f' lt 1 lw 2 pt 5 ps 1.5   # --- rouge
08: set style line 3 lc rgb '#c200f2' lt 1 lw 2 pt 3 ps 1.5   # --- violet
09:
10: set label 1 'Raspberry Pi' at 1.5,120 front center textcolor ls 1
11: set label 2 'SMSSystem' at 2.5,900 textcolor ls 2
12: set label 3 'Pianotier' at 2.3,500 right textcolor ls 3
13:
14: plot 'money.dat' i 0 u 1:2 with linespoints ls 1 title "Raspberry Pi", \
15:      ''          i 1 u 1:2 with linespoints ls 2 title "SMSSystem", \
16:      ''          i 2 u 1:2 with linespoints ls 3 title "Pianotier"
```

Les lignes 1 et 2 permettent de modifier les étiquettes des axes x et y et la ligne 4 désactive l'affichage de la légende (on peut par exemple utiliser **set key left** pour placer la légende à gauche, etc.). Les lignes 6 à 8 définissent des styles de ligne en leur associant un identifiant (**1**, **2** ou **3** ici). On utilise des commandes raccourcies pour définir précisément le style : **lc** pour **linecolor**, **lt** pour **linetype** (code numérique pour une ligne pleine (**1**), ou différents styles de pointillés), **lw** pour **linewidth**, **pt** pour **pointtype** (là encore un code numérique indiquant le type de point : **7** pour un disque, etc.) et **ps** pour **pointsize**. Ces styles peuvent ensuite être utilisés en y faisant référence par **ls <identifiant>**.

Dans les lignes 10 à 12, on affiche des étiquettes au-dessus des courbes en indiquant des coordonnées et enfin, dans les lignes 14 à 16 on affiche les droites. **i 0** correspond au premier index (les données pour le Raspberry Pi) et **u** (pour **using**) indique que nous travaillons avec la colonne **1** pour les abscisses et **2** pour les ordonnées.

La génération du graphe se fait ensuite en lançant **gnuplot** :

```
$ gnuplot

        G N U P L O T
        Version 4.6 patchlevel 6    last modified September 2014
        Build System: Linux x86_64
...
gnuplot> load 'graph.plt'
gnuplot> q
```

1. MISE EN PLACE DE LA CLÉ USB 3G

L'étape de mise en place de la clé n'est pas particulièrement compliquée et je décrirai donc cette partie rapidement.

Pour commencer, insérez votre carte SIM dans la clé... et si vous avez une carte SIM nano, n'oubliez pas de vous procurer des adaptateurs de cartes SIM (nano vers micro ou standard). Il s'agit d'une simple pièce de plastique qui coûte quelques euros (voir figure 2). Connectez ensuite la clé au Raspberry Pi et démarrez celui-ci.

Pour commencer, nous vérifions si la clé est bien reconnue :

```
$ dmesg | grep tty | grep usb
[    8.785663] usb 1-1.5: GSM modem (1-port)
converter now attached to ttyUSB0
[    8.786142] usb 1-1.5: GSM modem (1-port)
converter now attached to ttyUSB1
[    8.786613] usb 1-1.5: GSM modem (1-port)
converter now attached to ttyUSB2
[   10.096642] usb 1-1.5: GSM modem (1-port)
converter now attached to ttyUSB0
[   10.103533] usb 1-1.5: GSM modem (1-port)
converter now attached to ttyUSB1
[   10.108148] usb 1-1.5: GSM modem (1-port)
converter now attached to ttyUSB2
```

Fig. 2 : La clé USB 3G Huawei E169 (même s'il y a des choses bizarres écrites dessus, c'est bien le modèle indiqué) et un adaptateur de SIM.

Tout paraît correct et on peut même confirmer qu'il s'agit bien de notre modem :

```
$ lsusb | grep -i Huawei
Bus 001 Device 005: ID 12d1:1001 Huawei
Technologies Co., Ltd. E169/E620/E800
HSDPA Modem
```

Installons maintenant Gammu pour pouvoir utiliser ce périphérique :

```
$ sudo apt install gammu
```

Passez ensuite en root pour récupérer le fichier de configuration généré par **gammu-detect** et le placer dans **/etc/gammurc** (attention, les commandes gammu prennent souvent quelques secondes) :

```
$ sudo su
# gammu-detect > /etc/gammurc
```

NOTE

Vous pouvez bien entendu choisir de créer un fichier **.gammurc** dans votre répertoire personnel au lieu du **/etc/gammurc**. Pour l'usage que nous allons faire de ce Raspberry Pi, je vois difficilement d'autres comptes avec des configurations différentes, voilà pourquoi j'ai choisi l'option du fichier de configuration dans **/etc**.

Si vous le souhaitez, vous pouvez vérifier que le contenu du fichier de configuration correspond bien à votre matériel :

```
$ gammu --identify
Périphérique       : /dev/ttyUSB0
Fabricant          : Huawei
Modèle             : E169 (E169)
Firmware           : 11.314.13.51.156
IMEI               : 35963**********
```

On pourrait se contenter de cette configuration et passer directement à l'envoi d'un SMS, mais si vous rebootez votre Raspberry Pi, il n'est pas garanti que le modem soit de nouveau associé à **/dev/ttyUSB0** (c'est ce que nous avons déclaré dans **/etc/gammurc**). Nous allons donc créer une règle **udev** pour associer un lien fixe à ce périphérique dont le nom est alloué dynamiquement. Pour cela, relancez la commande suivante :

```
$ lsusb | grep -i Huawei
Bus 001 Device 005: ID 12d1:1001 Huawei
Technologies Co., Ltd. E169/E620/E800 HSDPA Modem
```

Après **ID** vous voyez deux identifiants : le premier correspond à l'identifiant du fabricant (**idVendor**) et le second à l'identifiant du produit (**idProduct**). Ceci va nous aider à créer le fichier de règles **/etc/udev/rules.d/98-usb-serial.rules** (en tant que *root*) :

```
SUBSYSTEMS=="usb", ATTRS{modalias}=="usb:v12D1p1001*",
KERNEL=="ttyUSB*", ATTRS{bInterfaceNumber}=="00", ATTRS
{bInterfaceProtocol}=="ff", SYMLINK+="ttyUSB_3G_modem"
SUBSYSTEMS=="usb", ATTRS{modalias}=="usb:v12D1p1001*",
KERNEL=="ttyUSB*", ATTRS{bInterfaceNumber}=="01", ATTRS
{bInterfaceProtocol}=="ff", SYMLINK+="ttyUSB_3G_diag"
SUBSYSTEMS=="usb", ATTRS{modalias}=="usb:v12D1p1001*",
KERNEL=="ttyUSB*", ATTRS{bInterfaceNumber}=="02", ATTRS
{bInterfaceProtocol}=="ff", SYMLINK+="ttyUSB_3G_pcui"
```

NOTE

Le Huawei E169 possède trois interfaces (visibles par **gammu-detect**). Vous devrez adapter ce fichier en fonction de votre modem.

Modifiez le fichier **/etc/gammurc** pour remplacer les noms de *devices* par les nouveaux noms (par exemple **ttyUSB0** par **ttyUSB_3G_modem**, **ttyUSB1** par **ttyUSB_3G_diag**, etc.).

Rechargez les règles udev :

```
$ sudo udevadm control --reload
```

Débranchez et rebranchez le modem et vous devriez voir celui-ci reconnu en tant que **ttyUSB_3G_modem** :

```
$ gammu --identify
Périphérique       : /dev/ttyUSB_3G_modem
...
```

Nous pouvons maintenant tester l'envoi d'un SMS. Auparavant, nous allons composer le code PIN associé à la carte SIM... sinon ça ne fonctionnera pas (bien entendu, si vous n'avez pas de code PIN, vous pouvez sauter cette étape... ou en ajouter un, ça pourrait être une bonne idée) :

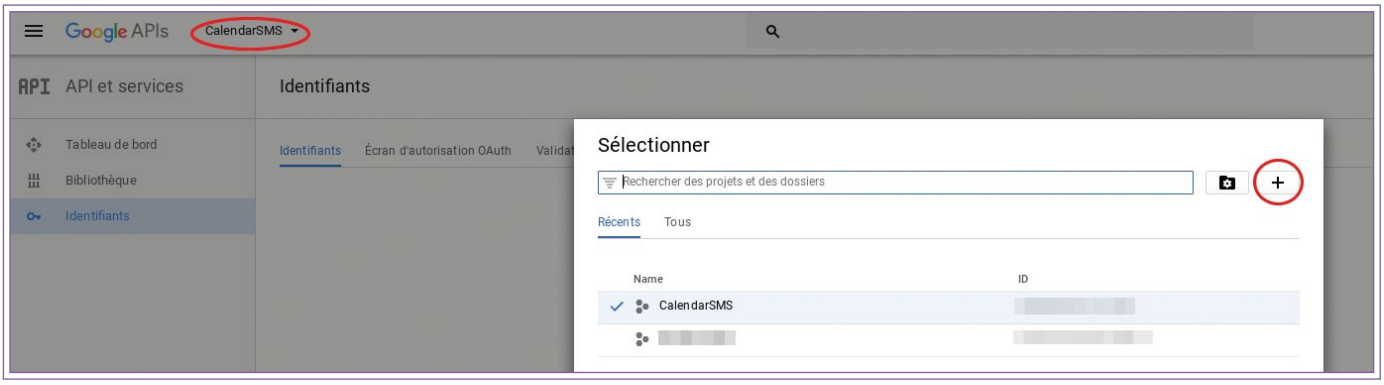


Fig. 3 : Création d'un projet.

```
$ gammu getsecuritystatus
En attente du PIN.
$ gammu entersecuritycode PIN 1234
$ gammu getsecuritystatus
Rien à faire entrer.
```

Et on envoie un SMS :

```
$ gammu sendsms TEXT 0612345678
-text "Premier SMS du serveur"
If you want break, press Ctrl+C...
Sending SMS 1/1....waiting for
network answer..OK, message
reference=123
```

Ça marche ! Malgré tout ce que j'ai pu lire sur Internet ou même dans **[1]** sur de mauvaises expériences, tout a fonctionné du premier coup ! Nous verrons plus loin comment envoyer les sms depuis un script.

2. UTILISER L'API GOOGLE CALENDAR

J'ai choisi la facilité dans cet article en employant Google Calendar, mais l'intérêt de la solution présentée est que l'on n'est lié à aucun logiciel ou service ! Vous pourrez très bien l'adapter à n'importe quel service proposant une API décente.

Partant donc sur une API Google, il va falloir activer l'API depuis la console développeurs de Google, comme nous l'avions fait pour **Gmail [2]**. Rendez-vous donc sur https://console.developers.google.com/ et à droite du titre « Tableau de bord », cliquez sur le lien ***Activer les API et les services*** puis sur le nouvel écran, dans la colonne ***API G Suite*** (à côté du logo de Gmail), cliquez sur ***Calendar API*** et enfin sur ***Activer***.

Si vous ne disposez pas encore de projet, il va falloir en créer un, ainsi qu'un identifiant client. Pour le projet, cliquez en haut à gauche sur le lien se trouvant à côté de « Google APIs » puis sur le signe « + » qui apparaît sur la nouvelle fenêtre (voir figure 3). Ici notre projet s'appellera `CalendarSMS` (que j'ai transformé au fil de la rédaction de cet article en `SMSRDV...` c'est plus « classe » :-)).

Rendez-vous ensuite sur l'onglet ***Écran d'autorisation Oauth*** et donnez un nom à votre application, comme le montre la figure 4.

Pour créer un identifiant, cliquez ensuite sur l'onglet ***identifiants*** puis sur ***Créer des identifiants > ID Client Oauth***. Remplissez ensuite le formulaire comme le montre la figure 5.

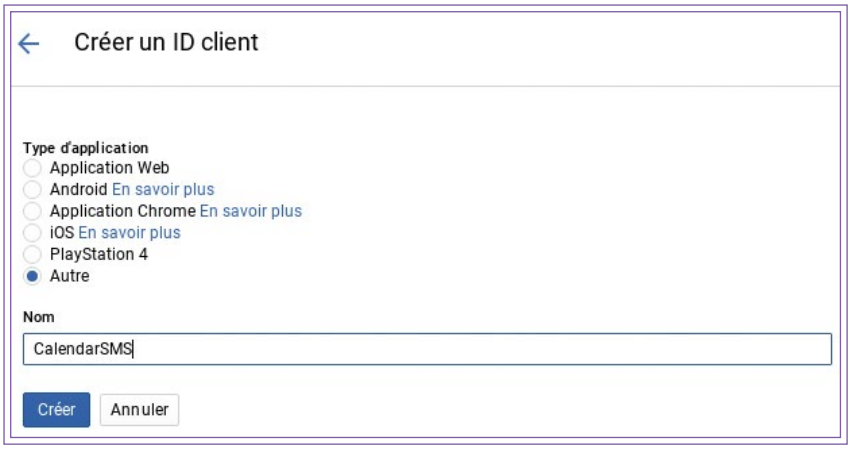


Fig. 5 : Création des identifiants client Oauth.

Lorsque vous revenez au menu général des identifiants, cliquez sur l'identifiant que vous venez de créer (comme si vous vouliez le modifier) et cliquez sur ***Télécharger JSON***. Vous obtenez un fichier `client_secret_....json` à placer dans le répertoire de votre projet (et éventuellement à renommer en `client_secret.json` sans tous les caractères superflus).

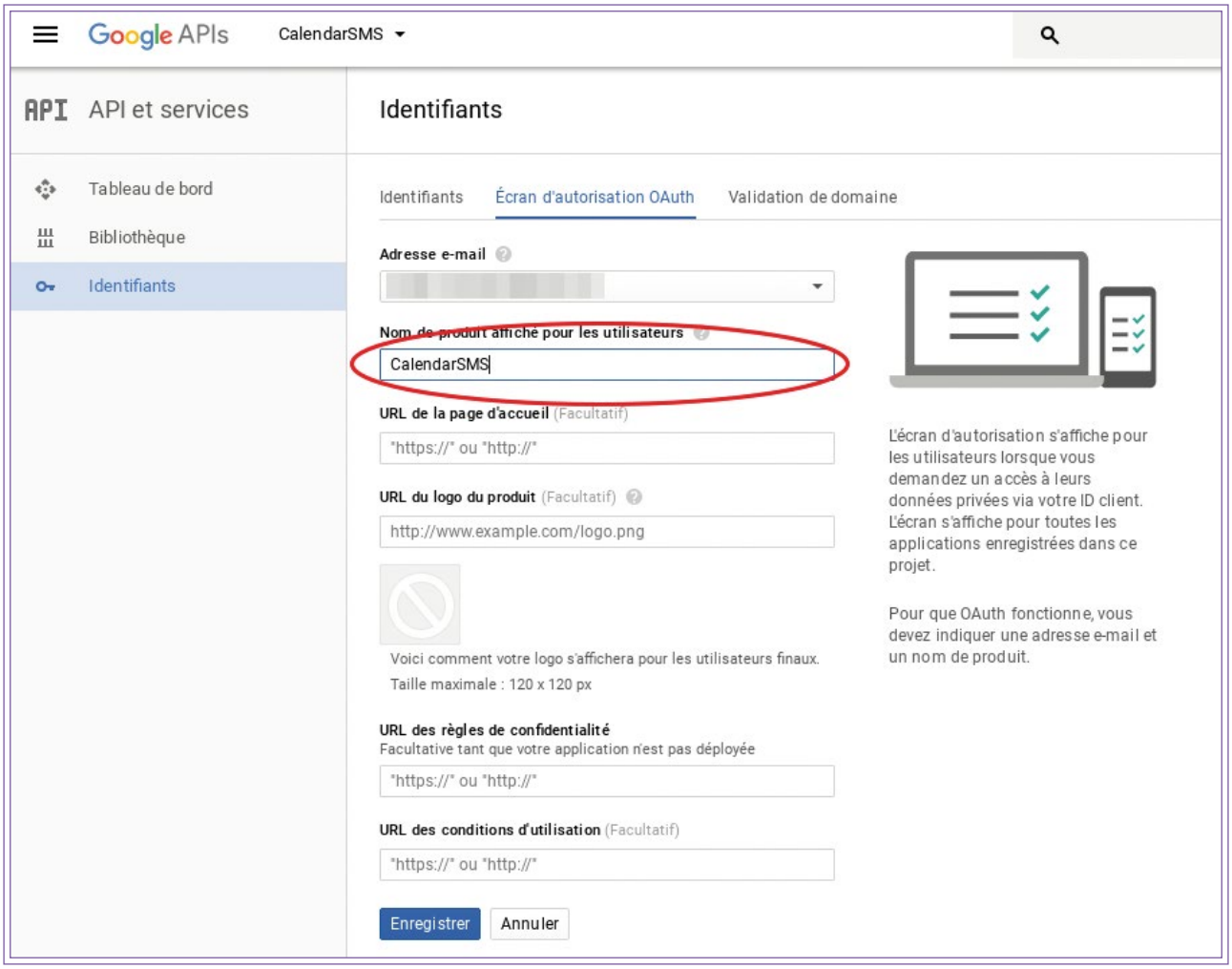


Fig. 4 : Configuration de l'écran d'autorisation Oauth.

Nous allons maintenant installer les modules de l'API Google :

```
$ sudo pip3 install google-api-python-client
```

Google propose un exemple sur https://developers.google.com/google-apps/calendar/quickstart/python. Cet exemple n'est pas forcément très bien structuré et n'exploite qu'une infime partie de l'API, mais il permet au moins de se mettre les pieds à l'étrier. Je suis parti de celui-ci pour élaborer le code permettant de récupérer les événements du lendemain, liés à un agenda donné.

Tout d'abord, j'ai utilisé un fichier de modèle (*template* pour ceux qui veulent absolument de l'anglais) qui permet de définir le format du SMS à envoyer. Ce fichier, nommé **default.tpl**, a la forme suivante :

```
Bonjour,
Je vous rappelle que nous avons rendez-vous demain à [heure].
En cas d'empêchement, veuillez me prévenir au [montel].
Cordialement,
[signature]
```

Les éléments en rouge sont ceux qui seront remplacés par notre programme avant d'envoyer le SMS (il existe des modules gérant tout cela très bien, mais pour ce que nous allons faire il n'est pas vraiment utile de sortir l'artillerie lourde).

Nous allons utiliser un fichier de configuration pour indiquer notre numéro de téléphone (**montel**) et le nom (**signature**) à utiliser dans le SMS. Ce fichier se nommera **default.ini** :

```
[perso]
signature=Tristan Colombo
phone_number=+33 6 11 22 33 44
```

Enfin, avant de nous lancer dans le code permettant de récupérer les événements d'un agenda particulier, nous devons créer cet agenda. Pour cela, rendez-vous sur la page internet de votre agenda Google, cliquez sur la flèche se trouvant à côté de ***Mes Agendas*** dans la colonne de gauche, puis sur ***Créer***

un agenda. Donnez-lui ensuite un nom avant de l'enregistrer (j'ai utilisé **Test**). Notez que pour accéder à un agenda depuis l'API nous n'utiliserons pas ce nom, mais l'identifiant associé à l'agenda (que vous pouvez voir sur la ligne ***Adresse URL de l'agenda***).

Passons donc au code pour comprendre comment utiliser le nom d'un agenda pour se connecter à celui-ci. Nous créerons un objet **Calendar** que nous pourrons manipuler ainsi plus simplement. Celui-ci utilisera des exceptions personnalisées que nous placerons dans un répertoire **exceptions** (qui contiendra bien entendu un fichier **__init__.py** pour que les fichiers s'y trouvant soient accessibles). Nous définissons trois exceptions dans les fichiers **CalendarNameNotFoundException.py**, **CalendarNoEventException.py** et **CalendarNoDescriptionException.py**. Ce ne sont que des renommages de la classe **Exception** et je ne donnerai donc que le code d'une classe, il suffit de modifier le nom pour les autres :

```
01: class CalendarNoEventException
(Exception):
02:     pass
```

Nous avons maintenant tous les éléments nous permettant d'aborder le fichier **Calendar.py** :

```
001: import httplib2
002: import os
003:
004: from apiclient import discovery
005: from oauth2client import client
006: from oauth2client import tools
007: from oauth2client.file import
Storage
008:
009: from exceptions.
CalendarNameNotFoundException import
CalendarNameNotFoundException
010: from exceptions.
CalendarNoEventException import
CalendarNoEventException
011: from exceptions.
CalendarNoDescriptionException import
CalendarNoDescriptionException
012:
013: import configparser
014: import datetime
015: import re
```

Nous commençons par la (longue) liste des imports de modules. Les lignes 1 à 7 seront utiles pour l'API Calendar, les lignes 9 à 11 sont les exceptions que nous avons définies et les lignes 13 à 15 permettront de lire le fichier de configuration ini (**configparser**) et de manipuler les données.

```
018: class Calendar:
019:     SCOPES = 'https://www.googleapis.com/auth/
calendar.readonly'
020:     APPLICATION_NAME = 'smsrdv'
021:     CLIENT_SECRET_FILE = None
022:     RE_TIME=r'T\d\d:\d\d'
```

Nous définissons quatre attributs de classe (accessibles en les préfixant par le nom de la classe, même si aucune instance n'a été créée). **SCOPES** contient l'URL définissant de quelle manière nous allons utiliser l'API Calendar. Ici il s'agit simplement d'une lecture, mais nous aurions pu y accéder aussi en écriture **[3]**. **APPLICATION_NAME** contient le nom de l'application (nécessaire pour se connecter via l'API Calendar), **CLIENT_SECRET_NAME** contient le nom du fichier json d'authentification (le fichier **client_secret.json** que nous avons téléchargé précédemment), et enfin **RE_TIME** contient l'expression régulière permettant d'extraire une heure d'un rendez-vous.

```
025:     def __init__(self, name='primary', template='default.tpl',
config_file='default.ini', client_secret_file='client_secret.json'):
026:         Calendar.CLIENT_SECRET_FILE = client_secret_file
027:         self.credentials = Calendar.getCredentials()
028:         self.http = self.credentials.authorize(httplib2.Http())
029:         self.service = discovery.build('calendar', 'v3',
http=self.http)
030:         self.events = None
031:         self.perso = {}
032:         try:
033:             with open(template, 'r') as fic:
034:                 self.template = fic.read()
035:         except:
036:             print('Unable to read template file', template)
037:             exit(1)
038:         try:
039:             self.name = self.getIdFromName(name)
040:         except CalendarNameNotFoundException:
041:             print('Calendar "{}" not found'.format(name))
042:             exit(2)
043:
044:         config = configparser.ConfigParser()
045:         config.read(config_file)
046:         for key in config['perso']:
047:             self.perso[key] = config['perso'][key]
```

Le constructeur accepte quatre paramètres : **name** qui est le nom de l'agenda (par défaut **'primary'** qui correspond à l'agenda « primaire »), **template** qui est le nom du fichier contenant le modèle de message à envoyer, **config_file** qui contient le nom du fichier ini de configuration, et **client_secret_file** qui, lui, contient le nom du fichier json d'authentification. En ligne 26, nous mettons à jour l'attribut de classe **CLIENT_SECRET_FILE** puis dans les lignes 27 à 29, nous définissons les attributs permettant la connexion à l'API Calendar. En ligne 30 et 31 se trouvent les définitions de l'attribut **events** qui contiendra les événements extraits de l'agenda et **perso** qui contiendra les éléments de configuration.

Dans les lignes 32 à 37, nous lisons le fichier de modèle et plaçons son contenu dans l'attribut **template**. Ensuite, dans les lignes 38 à 42 nous extrayons l'identifiant de l'agenda **name** grâce à la méthode **getIdFromName()** (décrite dans la suite) et le plaçons dans l'attribut **name**. Enfin, nous lisons le fichier de configuration dans les lignes 44 à 47 en appliquant une simple recette trouvée dans un très bon magazine **[4]**. Le contenu du fichier est placé dans l'attribut **perso** sous forme de dictionnaire (nous aurons les clés **signature** et **phone_number**).

```
050:     @staticmethod
051:     def getCredentials():
052:         home_dir = os.path.expanduser('~')
053:         credential_dir = os.path.join(home_dir,
'.smsrdv')
054:         if not os.path.exists(credential_dir):
055:             os.makedirs(credential_dir)
056:         credential_path = os.path.
join(credential_dir, 'credential.json')
057:
058:         store = Storage(credential_path)
059:         credentials = store.get()
060:         if not credentials or credentials.
invalid:
061:             flow = client.flow_from_
clientsecrets(Calendar.CLIENT_SECRET_FILE, Calendar.
SCOPES)
062:             flow.user_agent = Calendar.
APPLICATION_NAME
063:             credentials = tools.run_flow(flow,
store)
064:             print('Storing credentials to ' +
credential_path)
065:         return credentials
```

La méthode de classe **getCredentials()** permet la connexion à l'API Calendar (ouverture d'un navigateur pour autorisation de l'application à accéder en lecture aux agendas) et la sauvegarde des clés dans un fichier **~/.smsrdv/credential.json**.

NOTE

Si vous effectuez des tests et que des erreurs se glissent dans votre code pour l'authentification, il peut être nécessaire de supprimer **credential.json** pour redémarrer sur des bases saines...

```
068:     @staticmethod
069:     def getFutureDate(days):
070:         time = datetime.datetime.
today() + datetime.timedelta(days)
071:         time = time.replace(hour=0,
minute=0, second=0, microsecond=0)
072:         return time.isoformat() + 'Z'
```

La méthode de classe **getFutureDate()** permet d'obtenir une date au format iso d'un jour prochain (**days** définit le nombre de jours). Par exemple, pour obtenir la date du lendemain, nous utiliserons **Calendar.getFutureDate(1)**. Cette fonction n'est présente que pour permettre à notre programme d'évoluer par la suite, car nous n'aurons besoin que de la date du lendemain dans notre cas de figure.

```
075:   def getIdFromName(self, name):
076:       page_token = None
077:       while True:
078:           calendar_list = self.service.calendarList().
list(pageToken=page_token).execute()
079:           for calendar_list_entry in calendar_
list['items']:
080:               if calendar_list_entry['summary'] == name:
081:                   return calendar_list_entry['id']
082:           page_token = calendar_list.get('nextPageToken')
083:           if not page_token:
084:               break
085:       raise CalendarNameNotFoundException()
```

La méthode **getIdFromName()** nous permet de récupérer l'identifiant correspondant à l'agenda **name**. Si l'agenda n'est pas trouvé, nous levons une exception **CalendarNameNotFoundExeption** (ligne 85).

NOTE

Il est possible d'adapter le code précédent pour afficher la liste des agendas disponibles ainsi que leur identifiant :

```
    def listAgendas(self):
        page_token = None
        while True:
            calendar_list = self.service.calendarList().
list(pageToken=page_token).execute()
            for calendar_list_entry in calendar_
list['items']:
                print(calendar_list_entry['summary'] +
' - ' + calendar_list_entry['id'])
            page_token = calendar_list.get('nextPageToken')
            if not page_token:
                break
```

Nous n'aurons pas besoin de cette méthode ici, mais dans certains cas ça peut être pratique...

```
089:     def getTomorrowEvents(self):
090:         timeMin = Calendar.getFutureDate(days=1)
091:         timeMax = Calendar.getFutureDate(days=2)
092:         eventsResult = self.service.events().list(
093:             calendarId=self.name, timeMin=timeMin,
timeMax=timeMax, singleEvents=True,
094:             orderBy='startTime').execute()
095:         self.events = eventsResult.get('items', [])
096:
097:         if not self.events:
098:             raise CalendarNoEventException()
```

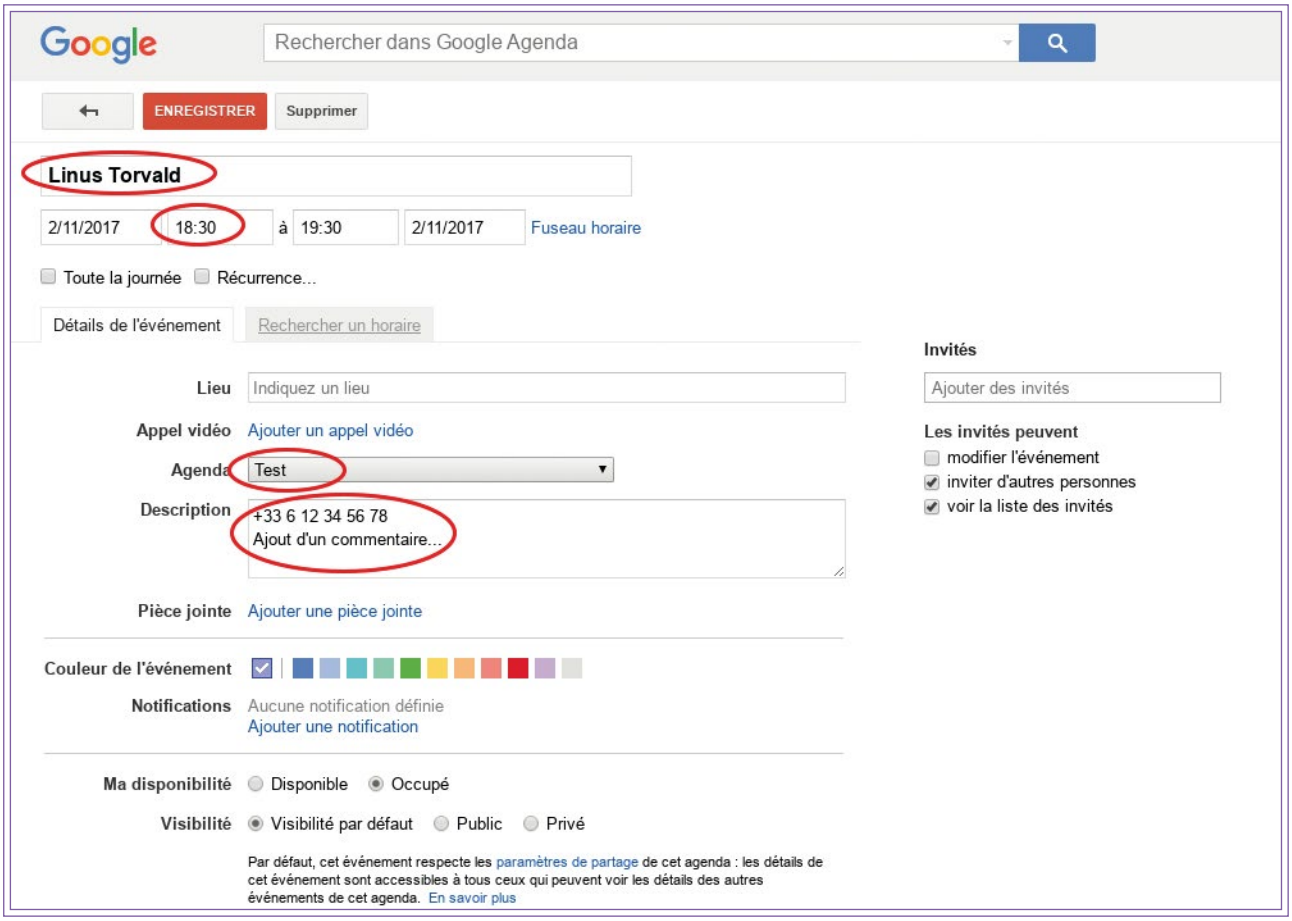


Fig. 6 : Création d'un nouveau rendez-vous dans l'agenda « Test ».

`getTomorrowEvents()`, comme son nom l'indique, permet de récupérer la liste des événements du lendemain et de les stocker dans l'attribut **`events`**. Pour borner notre recherche, nous utilisons les paramètres **`timeMin`** et **`timeMax`** de la méthode **`list()`** (lignes 92 à 94). Ces paramètres sont calculés et placés dans des variables de même nom dans les lignes 90 et 91 : nous prenons la date du lendemain à minuit et la date du surlendemain à minuit (ce qui nous donne la date du lendemain de 0h à 23h59). Si aucun événement n'est trouvé, nous levons une exception **`CalendarNoEventException()`**.

```
101:    def sendSMS(self):
102:        for event in self.events:
103:            date = event['start'].get('dateTime', event['start'].get('date'))
104:            time = re.search(Calendar.RE_TIME, date).group(0)[1:]
105:            name = event['summary']
106:            if 'description' in event:
107:                phoneNumber = event['description'].split('\n')[0].replace(' ', '')
108:            else:
109:                raise CalendarNoDescriptionException()
110:            msg = self.template.replace('[heure]', time) \
111:                               .replace('[nom]', name) \
112:                               .replace('[montel]', self.perso['phone_number']) \
113:                               .replace('[signature]', self.perso['signature'])
114:            print(msg)
```

NOTE

La méthode **`list()`** appliquée sur des **`Events`** accepte également un paramètre **`maxResults`** permettant de limiter le nombre de résultats. Les autres paramètres sont visibles sur la fiche de la méthode dans la documentation officielle **[5]**.

Une fois les événements récupérés, c'est la méthode **`sendSMS()`** qui sera chargée d'envoyer les SMS. Pour cela, nous récupérons l'heure des rendez-vous (lignes 103 et 104), le nom associé au rendez-vous (ligne 105) et la description (ligne 106 à 109) dont nous extrayons la première ligne qui est censée contenir le numéro de téléphone. Ensuite le message est composé depuis le modèle contenu dans l'attribut **`template`** par remplacements successifs (lignes 110 à 113). Pour l'instant, nous affichons seulement les messages, l'envoi sera ajouté par la suite.

NOTE

Les champs disponibles pour chaque événement sont donnés dans la documentation **[6]**.

```
118: if __name__ == '__main__':
119:     c = Calendar('Test')
120:     c.getTomorrowEvents()
121:     c.sendSMS()
```

Pour tester notre code, nous créons un petit programme principal qui se connecte à l'agenda **Test**, récupère les événements du lendemain et affiche les messages.

Pensez à ajouter quelques événements sur le jour suivant le jour de vos tests et complétez bien tous les champs comme le montre la figure 6.

Au premier lancement, il vous sera demandé de vous connecter avec votre compte Google et de valider l'autorisation d'accès à votre Agenda (voir figure 7, page suivante).

Si vous lancez votre script peu de temps après avoir activé l'API et que vous obtenez un message d'erreur, lisez-le bien, il suffit peut-être d'attendre : *« If you enabled this API recently, wait a few minutes for the action to propagate to our systems and retry »*. Si le problème persiste, vérifiez que vous avez bien activé l'API *Calendar API* (voir en début de section).

Finalement, vous devez obtenir quelque chose de similaire à ceci :

```
$ python3 Calendar.py
Bonjour,
Je vous rappelle que nous avons
rendez-vous demain à 16:00.
En cas d'empêchement, veuillez me
prévenir au +33 6 11 22 33 44.
Cordialement,
Tristan Colombo

...

Bonjour,
Je vous rappelle que nous avons
rendez-vous demain à 18:00.
En cas d'empêchement, veuillez me
prévenir au +33 6 11 22 33 44.
Cordialement,
Tristan Colombo
```

Nous savons maintenant que les deux éléments fondamentaux de notre solution d'envoi automatique de SMS fonctionnent… il ne nous reste plus qu'à les utiliser au sein du même programme.

NOTE

Si l'un des événements ne respecte pas le format fixé (une description contenant un numéro de téléphone sur la première ligne), nous sortons du programme avec une exception. Ce n'est peut-être pas forcément l'objectif fixé ! Il est alors possible d'ajouter un test pour détecter si la description du rendez-vous contient bien un numéro de téléphone et d'afficher seulement un message d'erreur en cas d'absence de description :

```
...
018: class Calendar:
...
022:     RE_TIME=r'T\d\d:\d\d'
023:     RE_PHONE_NUMBER=r'^\d{10}$|^\+33\d{9}$'
...
101:     def sendSMS(self):
102:         for event in self.events:
103:             date = event['start'].get('dateTime', event['start'].
get('date'))
104:             time = re.search(Calendar.RE_TIME, date).group(0)[1:]
105:             name = event['summary']
106:             if 'description' in event:
107:                 phoneNumber = event['description'].split('\n')[0].
replace(' ', '')
108:                 if re.search(Calendar.RE_PHONE_NUMBER, phoneNumber)
is None:
109:                     print('Format de numéro de téléphone invalide :',
phoneNumber)
110:                     continue
111:                 msg = self.template.replace('[heure]', time) \
112:                                    .replace('[nom]', name) \
113:                                    .replace('[montel]', self.
perso['phone_number']) \
114:                                    .replace('[signature]', self.
perso['signature'])
115:                 print(msg)
116:             else:
117:                 print('Événement "{}" sans description !'.format(name))
...
```

3. PREMIÈRE MISE EN PLACE DE NOTRE SOLUTION

Il ne nous reste plus beaucoup de travail maintenant et la plus « grosse » tâche sera de modifier la méthode **sendSMS()** pour qu'elle envoie réellement un SMS à l'aide de Gammu que nous avons installé au début de cet article. Toutefois, nous n'avons pas installé le module Python permettant d'utiliser Gammu, ce que nous allons faire maintenant.

Sachez tout d'abord que le paquet **python3-gammu** n'est pas disponible pour **Raspbian Jessie**. Si vous êtes déjà sous **Stretch**, pas de problème, pour les autres,

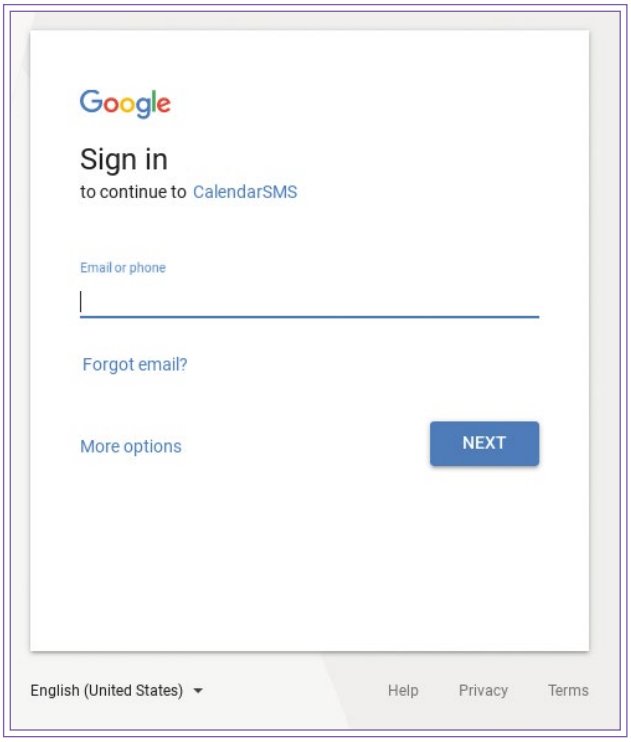


Fig. 7 : Premier démarrage du script, demande de connexion pour valider les autorisations d'accès.

il vous faudra effectuer la migration. Bien entendu, n'oubliez pas de sauvegarder vos fichiers avant la mise à jour, on ne sait jamais à moins que, comme moi vous ne travailliez sur une machine de test : au pire vous aurez à réinstaller une Raspbian... ce qui est finalement une meilleure idée, car bien plus rapide ! Dans ce cas, vous devrez penser à installer à nouveau gammu et tout ce que nous avions installé en début d'article.

Je suppose maintenant que votre système est à jour ; nous pouvons installer python3-gammu :

```
$ sudo apt install python3-gammu
```

Il faut reprendre le code de **Calendar.py** pour y ajouter la gestion des SMS avec gammu :

```
...
016: import gammu
017: from time import sleep
018: from random import randint
```

Nous aurons besoin bien entendu du module **gammu**, mais également des fonctions **sleep()** et **randint()** issues respectivement des modules **time** et **random**. Ces fonctions nous permettront d'effectuer des pauses de quelques secondes (durée aléatoire). Nous verrons plus loin pourquoi...

```
...
021: class Calendar:
...
029:     def __init__(self, name='primary',
template='default.tpl', config_file='default.
ini', client_secret_file='client_secret.json'):
...
053:         self.gammu = gammu.StateMachine()
054:         self.gammu.ReadConfig()
055:         self.gammu.Init()
056:         if self.gammu.GetSecurityStatus() ==
'PIN':
057:             print('On entre le PIN')
058:             self.gammu.
EnterSecurityCode('PIN', self.perso['pin'])
059:         else:
060:             print('Rien à faire')
061:
062:         print('Attente détection SIM')
063:         sleep(10)
```

Pour envoyer nos SMS, nous utiliserons un objet **GSM_StatMachine** créé en lisant la configuration que nous avions créée précédemment (lignes 53 à 55). S'il faut spécifier un code PIN, nous indiquons celui du fichier de configuration **default.ini** (lignes 56 à 60). Enfin, nous attendons 10 secondes en ligne 63. Pourquoi ? Si nous n'introduisons pas un délai, lors du premier lancement du programme, nous obtenons une erreur **ERR_NOSIM**. Au deuxième lancement, tout fonctionne bien... La solution que j'ai trouvée (pas vraiment satisfaisante) est d'introduire un délai d'attente avant d'envoyer des SMS, ce qui visiblement laisse le temps de détecter la carte SIM et supprime l'erreur dès le premier lancement.

```
...
116:   def sendSMS(self):
117:       for event in self.events:
118:           date = event['start'].get('dateTime',
event['start'].get('date'))
119:           time = re.search(Calendar.RE_TIME,
date).group(0)[1:]
120:           name = event['summary']
121:           if 'description' in event:
122:               phoneNumber =
event['description'].split('\n')[0].replace(' ', '')
123:               if re.search(Calendar.RE_PHONE_
NUMBER, phoneNumber) is None:
124:                   print('Format de numéro de
téléphone invalide :', phoneNumber)
125:                   continue
126:               msg = self.template.
replace('[heure]', time) \
127:                       .replace('[nom]', name) \
128:                       .replace('[montel]',
self.perso['phone_number']) \
129:                       .replace('[signature]',
self.perso['signature'])
130:               msgToSend = {
```

1ER ÉVÉNEMENT EUROPÉEN
LIBRE & OPEN SOURCE

ENABLING DIGITAL EVERYWHERE

opensourcesummit.paris

#OSSPARIS17

PARIS REGION SMART WEEKS

La FRENCH TECH

PARIS OPEN SOURCE SUMMIT

6 & 7 DÉCEMBRE 2017

DOCK PULLMAN

Email : cjardon@weyou-group.com – Tel : 01 41 18 60 52

SPONSOR DIAMOND

Inria

SPONSORS PLATINUM

alter way Microsoft

Smile OPEN SOURCE SOLUTIONS

SPONSORS GOLD

BlueMind Messagerie & espaces collaboratifs

henix

IBM

redhat

SensioLabs Créateur de Symfony

SPONSORS SILVER

axelor

centreon

cozy.io

AXELTI ZABBIX

devoteam

XiVO

ORGANISÉ PAR

Systematic Paris Region Digital Ecosystem

weyou group

EN PARTENARIAT AVEC

CNLL Les entreprises du numérique ouvert

îledeFrance

```
131:                     'Text' : msg,
132:                     'SMSC' : {'Location' : 1},
133:                     'Number' : phoneNumber,
134:                 }
135:                 print('- On envoie le message :')
136:                 print(msg)
137:                 self.gammu.SendSMS(msgToSend)
138:                 delay = randint(3, 15)
139:                 print('Attente de : {}s'.format(delay))
140:                 sleep(delay)
141:             else:
142:                 print('Événement "{}" sans description
!'.format(name))
```

Pour l'envoi d'un SMS, nous retrouvons la création du message et la récupération du numéro de téléphone dans les lignes 118 à 134 puis nous créons un dictionnaire **msgToSend** contenant les informations d'envoi du SMS. La seule clé qui peut poser question est la clé **SMSC** : il s'agit des informations sur l'émetteur du SMS et en fixant sa valeur à **1** elles sont déterminées automatiquement. Nous envoyons ensuite le message sous forme de SMS à l'aide de la méthode **SendSMS()** en ligne 137. Avant d'éventuellement envoyer un autre SMS, nous attendons entre 3 et 15 secondes (lignes 138 à 140), de manière à ce que votre opérateur ne vous prenne pas pour un spammeur...

Et voilà, notre système répond à toutes nos contraintes de départ ! Il n'y a plus qu'à créer des crons et des fichiers de configuration pour toutes les personnes qui souhaitent utiliser notre système au sein d'une même organisation.

J'aurais pu arrêter cet article ici, mais il est tentant d'améliorer le système en autorisant une interaction avec celui-ci via des SMS... et c'est là que les choses vont se compliquer et devenir intéressantes !

4. ALLER PLUS LOIN AVEC LA RÉCEPTION ET LE TRAITEMENT DE SMS

C'est un démon qui est chargé de réceptionner les SMS. Nous aurons un nouveau paquet à installer :

```
$ sudo apt install gammu-smsd
```

Vous devez ensuite configurer le fichier **/etc/gammu-smsdrc** (n'oubliez pas que nous avons créé un nom fixe de *device*) :

```
...
[gammu]
port = /dev/ttyUSB_3G_modem
connection= at19200
...
[smsd]
...
PIN = xxx # À spécifier si votre carte SIM est protégée par un code de sécurité
RunOnReceive = sudo /home/pi/SMSRDV/sms_analysis.py
...
```

Ajoutez le groupe **gammu** à la liste des *sudoers* pouvant exécuter **sms_analysis.py** sans mot de passe en lançant :

```
$ sudo visudo
```

Puis ajoutez la ligne suivante (à adapter en fonction du chemin de votre programme) :

```
...
gammu ALL=(ALL) NOPASSWD: /home/pi/SMSRDV/sms_analysis.py
...
```

Redémarrez votre Raspberry Pi. À partir de maintenant, si vous ne notez aucune erreur dans votre syslog (**sudo tail /var/log/syslog**), les SMS vont se trouver dans **/var/spool/gammu/inbox** (valeur par défaut de **/etc/gammu-smsdrc** que vous pouvez modifier) sous la forme de fichiers dont le nom sera du type **INAAAAMMJJ_HHMMSS_oo_+33PPPPPPPPP_xx.txt** où :

- **AAAAMMJJ** représente la date (année, mois et jour) ;
- **HHMMSS** représente l'heure d'envoi (heure, minutes et secondes) ;
- **oo** pour l'ordre du message si des messages sont reçus en même temps (donc la plupart du temps ce champ vaudra **00**) ;
- **+33PPPPPPPPP** est le numéro de téléphone de l'expéditeur ;
- **xx** est le numéro du SMS (**00** pour un SMS simple et pour un SMS découpé la valeur ira de **00** à **nn**).

Créons un fichier **sms_analysis.py** qui est le fichier appelé par **gammu-smsd** à la réception d'un SMS en lui passant en paramètre le nom du fichier associé à ce SMS :

```
01: #!/usr/bin/python3
02:
03: import sys
04: import re
05:
06: BASE_PATH = '/var/spool/gammu/inbox/'
07:
08: def getPhoneNumber(filename):
09:     return re.search('\+33\d{9}',
filename).group(0)
10:
11: if __name__ == '__main__':
```

```
12:   filename = BASE_PATH + sys.argv[1]
13:   with open(filename, 'r') as fic:
14:       msg = fic.read()
15:       with open('/home/pi/SMSRDV/inbox.log', 'a')
as fic2:
16:           response = '***************\n'
17:           response += 'Message : ' + msg + '\n'
18:           response += 'Expéditeur : ' +
getPhoneNumber(sys.argv[1]) + '\n'
19:           if msg.find('SMSRDV-CMD') == 0:
20:               response += 'Commande détectée !\n'
21:           response += '***************\n'
22:           fic2.write(response)
```

Il ne s'agit ici que d'un petit programme démontrant ce qu'il est possible de réaliser. À la réception d'un SMS, nous ajouterons dans le fichier **inbox.log** le message et le numéro de téléphone de l'expéditeur (lignes 15 à 18). Si le message commence par **SMSRDV-CMD** (position **0** renvoyée par **find()** en ligne 19), alors nous indiquons qu'une commande a été détectée : à vous de paramétrer des commandes et leur action (et même de les protéger par un mot de passe éventuellement).

N'oubliez pas de rendre le fichier exécutable :

```
$ chmod ugo+x sms_analysis.py
```

Après réception de SMS, vous obtiendrez quelque chose similaire à :

```
$ more inbox.log
***************
Message : SMSRDV-CMD pour tester
Expéditeur : +33611229900
Commande détectée !
***************
***************
Message : Message sans commande
Expéditeur : +33611229900
***************
```

Tout va pour le mieux ! Confiants, après de multiples tests nous pouvons mettre notre projet en production... et là, patatras ! Éventuellement la première slave de SMS partira, mais vraisemblablement pas la seconde. Mais pourquoi ?

5. CHERCHEZ L'ERREUR

En analysant les logs de **/var/log/syslog** et **/var/log/cron.log**, on s'aperçoit que le programme est bien appelé à l'heure fixée... mais les SMS ne partent pas. Nous avons ajouté un système signalant les erreurs sur les événements lus, mais ce système envoie un SMS, donc si la machine est dans l'incapacité d'envoyer un SMS, nous n'en saurons rien.

La première idée est donc d'implémenter un mécanisme pour envoyer un message d'erreur par mail (je ne traiterai pas cette partie dans cet article). Mais il y a forcément quelque chose qui se joue dans le temps puisque sur un lancement manuel tout fonctionne correctement...

Le programme n'envoie plus de SMS, mais peut-il en recevoir ? Ah, là non plus ça ne marche plus. En regardant du côté de **gammu-smsd**, on s'aperçoit que le démon ne tourne plus lorsque l'on vérifie son état par :

```
$ sudo service gammu-smsd status
```

Ni une, ni deux, on le relance :

```
$ sudo service gammu-smsd start
```

Ouf ! Nous recevons à nouveau les SMS. Qu'en est-il de l'émission ?

```
$ python3 Calendar.py
Traceback (most recent call last):
...
  File "/home/pi/SMSRDV/Calendar.py", line 72, in __init__
    self.gammu.Init()
gammu.ERR_DEVICEOPENERROR: {'Where': 'Init', 'Text':
"Erreur à l'ouverture du périphérique : périphérique
inconnu, occupé ou problème de permissions.", 'Code': 2}
```

Ça ne marche pas... Si on débranche le modem et qu'on le rebranche, en refaisant un test ça fonctionne... mais plus la réception des SMS, **gammu-smsd** a encore planté ! Je penche donc pour une hypothèse simple : **gammu-smsd** et **gammu** rentrent en conflit. Pour valider mon hypothèse, je relance le démon :

```
$ sudo service gammu-smsd start
```

Puis je regarde ce qui se passe du côté de **gammu** :

```
$ gammu --identify
Aucune réponse dans le temps d'attente spécifié:
Le téléphone n'est peut-être pas connecté.
```

Jusque-là l'hypothèse est vérifiée, faisons la démarche inverse. J'arrête **gammu-smsd** :

```
$ sudo service gammu-smsd stop
```

Puis je regarde **gammu** :

```
$ gammu --identify
Périphérique       : /dev/ttyUSB_3G_modem
Fabricant          : Huawei
Modèle             : E169 (E169)
Firmware           : 11.314.13.51.156
IMEI               : 35963**********
```

Bingo ! Lorsque **gammu-smsd** est actif, il ne faut traiter qu'avec lui pour éviter des conflits. Du coup, nous devons réviser notre méthode d'envoi de SMS. Testons en ligne de commandes :

```
$ sudo gammu-smsd-inject TEXT 0611223344 -text
"Message de test"
gammu-smsd-inject[14356]: Created outbox message
OUTC20170920_102721_00_0611223344_sms0.smsbackup
Written message with ID /var/spool/gammu/outbox/
OUTC20170920_102721_00_0611223344_sms0.smsbackup
```

Ça marche !!! Et en Python ? Faisons un petit test avec **sms_test.py** :

```
01: import gammu.smsd
02:
03: smsd = gammu.smsd.SMSD('/etc/gammu-smsdrc')
04:
05: msg = {'Text' : 'Petit message de test',
'SMSC' : {'Location' : 1}, 'Number' : '0611223344'}
06:
07: smsd.InjectSMS([msg])
```

Puis :

```
$ sudo python3 sms_test.py
```

Et là encore, ça fonctionne !

ATTENTION !

Notez bien qu'avec **gammu-smsd**, vous devez lancer vos commandes d'envoi de SMS en tant que root (d'où le **sudo** devant l'appel à **sms_test.py**) ! Donc si vous avez installé des crons, il faut les passer en root (**sudo crontab -e** au lieu de simplement **crontab -e**).

Pensez également que si vous utiliser dans vos codes des chemins vers des fichiers, ceux-ci devront être notés en chemins absolus !

Modifions donc rapidement le code de **Calendar.py** :

```
...
016: import gammu.smsd
...
021: class Calendar:
...
029:     def __init__(self, user,
name='primary', template='default.tpl', config_
file='default.ini', client_secret_file='client_
secret.json', report=True):
...
053:         self.gammu_smsd = gammu.smsd.
SMSD('/etc/gammu-smsdrc')
...
116:     def sendSMS(self):
...
130:         msgToSend = {
131:             'Text' : msg,
132:             'SMSC' : {'Location' : 1},
133:             'Number' : phoneNumber,
134:         }
135:         print('- On envoie le message ({}):'.
format(phoneNumber))
136:         self.gammu_smsd.InjectSMS([msgToSend])
...
```

NOTE

Puisque désormais **gammu-smsd** est le Grand Maître des SMS sur notre serveur, assurons-nous qu'il est toujours actif en ajoutant une règle de surcharge par :

```
$ sudo systemctl edit gammu-smsd
```

La règle est :

```
[Service]
Restart=always
```

6. VERS L'INFINI ET AU-DELÀ !

L'objectif initial est atteint et nous avons une solution réellement fonctionnelle... mais puisque nous manipulons les SMS, comment ne pas avoir envie de réaliser réellement un tour complet de la technologie (ou presque, car nous reviendrons sans doute dans un autre article sur les SMS de Classe 0). Je vous propose donc maintenant de voir comment envoyer des SMS longs et comment réaliser une authentification deux facteurs.

6.1 Envoyer des SMS longs

Il existe une méthode dans le module gammu qui permet de gérer directement l'envoi de SMS longs sans avoir à découper « à la main » les messages par paquets de 160 caractères :

```
msg = 'xxx' # Un message long de plus de 160
caractères

smsinfo = {
            'Class': -1,
            'Unicode': True,
            'Entries':  [
                {
                    'ID': 'ConcatenatedTextLong',
                    'Buffer': msg
                }
            ]
        }
encoded = gammu.EncodeSMS(smsinfo)

print('- On envoie le rapport :')
print(msg)
```

```
for msgToSend in encoded:
    msgToSend['SMSC'] = {'Location': 1}
    msgToSend['Number'] = self.perso['phone_
number']
    self.gammu_smsd.InjectSMS([msgToSend])
```

Comme vous le voyez, il s'agit simplement d'une modification de la structure contenant le message (maintenant **smsinfo** qui contient **msg**). Si vous comptez expédier des messages longs, vous n'avez qu'à remplacer la méthode précédente par celle-ci.

6.2 Réaliser une authentification deux facteurs

Le principe de l'authentification à deux facteurs est simple : vous devez fournir deux informations pour vous authentifier. En général, la première information est un mot de passe et la deuxième information peut être un code que vous recevez par SMS. Dans cette partie, nous allons réaliser une petite page de connexion illustrant ce concept. Il ne s'agit donc que d'une preuve de concept, d'un petit script cgi posant les bases de l'authentification deux facteurs, mais qui ne fait pas intervenir de base de données par exemple.

Pour commencer, nous aurons besoin du serveur proprement dit. Ce sera le fichier **server.py** :

```
01: #!/usr/bin/python3
02:
03: import http.server
04:
05: PORT = 8000
06: server_address = ('', PORT)
07:
08: server = http.server.HTTPServer
09: handler = http.server.CGIHTTPRequestHandler
10: handler.cgi_directories = ['/']
11: print('Serveur actif sur le port : ', PORT)
12:
13: httpd = server(server_address, handler)
14: httpd.serve_forever()
```

Le fichier principal, gérant les connexions et les différents affichages de pages sera **connexion.py** :

```
01: #!/usr/bin/python3
02:
03: import cgi
04: import random
05: import gammu
06:
07: LOGIN = 'glmf'
08: PWD   = 'L1nu><'
09: TEL   = '0611223344'
10:
11: def generatePassword(length=6):
12:     code = ''
13:     for i in range(length):
14:         code += str(random.randint(0, 9))
15:     with open('.code_storage', 'w') as fic:
16:         fic.write(code)
17:     return code
18:
19:
20: def sendSMS(code, phoneNumber):
21:     sm = gammu.StateMachine()
22:     sm.ReadConfig()
23:     sm.Init()
24:     if sm.GetSecurityStatus() == 'PIN':
25:         sm.EnterSecurityCode('PIN', '0000')
26:     msgToSend = {
27:             'Text' : 'Votre code de connexion : ' + code,
28:             'SMSC' : {'Location' : 1},
29:             'Number' : phoneNumber,
30:     }
31:     sm.SendSMS(msgToSend)
32:
33:
34: if __name__ == '__main__':
35:     form = cgi.FieldStorage()
36:     print('Content-type: text/html; charset=utf-8\n')
37:
38:     access = False
39:     login = form.getvalue('login')
40:     pwd = form.getvalue('password')
41:     code = form.getvalue('code')
42:
43:     if code is not None:
44:         with open('.code_storage', 'r') as fic:
45:             code_sms = fic.read()
46:         if code == code_sms:
47:             print('ACCÈS AUTORISÉ')
48:             access = True
49:         else:
50:             print('Erreur dans le code SMS. Retour à la
case départ')
51:
52:     if login == LOGIN and pwd == PWD:
53:         with open('sms.html', 'r') as fic:
54:             html = fic.read()
55:         code_sms = generatePassword()
56:         sendSMS(code_sms, TEL)
57:     else:
58:         if login is not None or pwd is not None:
59:             print('Identifiants incorrects !')
60:         with open('index.html', 'r') as fic:
61:             html = fic.read()
62:
63:     if not access:
64:         print(html)
```

Dans ce code exemple, les identifiants de connexion sont stockés en dur et non chiffrés (lignes 7 à 9). On trouve une fonction de génération de codes basique (lignes 11 à 17) : par défaut, le code sera composé de 6 chiffres. Dans les lignes 20 à 31, nous retrouvons l'envoi de SMS (première méthode sans **gammu-smsd** de manière à s'affranchir du lancement en root). Le programme principal consiste à récupérer les éventuelles valeurs fournies par les formulaires (lignes 39 à 41) et, en fonction de leurs valeurs, à générer la bonne page à afficher. Notez qu'ici le code généré et envoyé par SMS

en ligne 55 est sauvegardé dans un fichier **.code_storage** (lignes 15 et 16) puisqu'il n'y a pas persistance des données.

Il nous manque ensuite seulement les deux fichiers de modèles html. Pour **index.html**, il s'agit du formulaire demandant les identifiants de connexion :

```
01: <!doctype html>
02:
03: <html lang="fr">
04:     <head>
05:         <meta charset="utf-8" />
06: <title>Authentification deux facteurs</title>
07:     </head>
08:
09:     <body>
10:         <form action="/connexion.py" method="post">
11:      <fieldset>
12:          <legend>Identification</legend>
13:    <label for="login">Login :</label>
14:    <input type="text" name="login" id="login"
placeholder="Votre identifiant" /><br />
15:    <label for="password">Password :</label>
16:    <input type="password" name="password" id="password"
placeholder="Votre mot de passe" /><br />
17:    <input type="submit" name="send" value="Connexion"/>
18:      </fieldset>
19: </form>
20:     </body>
21: </html>
```

Et pour **sms.html**, il s'agit du formulaire de demande du code envoyé par SMS :

```
01: <!doctype html>
02:
03: <html lang="fr">
04:     <head>
05:         <meta charset="utf-8" />
06: <title>Authentification deux facteurs</title>
07:     </head>
08:
09:     <body>
10:         <form action="/connexion.py" method="post">
11:      <fieldset>
12:          <legend>Identification</legend>
13:    <label for="code">Code :</label>
14:    <input type="password" name="code" id="code"
placeholder="Le code envoyé par sms" /><br />
15:    <input type="submit" name="send" value="Connexion"/>
16:      </fieldset>
17: </form>
18:     </body>
19: </html>
```

Pour lancer ce code, n'oubliez pas de rendre exécutable les fichiers **server.py** et **connexion.py** :

```
$ chmod ugo+x server.py connexion.py
```

Lancez **server.py** :

```
$ server.py
Serveur actif sur le port : 8000
```

Puis connectez-vous à votre machine depuis un navigateur (par exemple sur localhost ce sera localhost:8000/connexion.py). Les bases du système sont en place, il n'y a plus qu'à les adapter sur un véritable serveur, à améliorer la génération de code, à intégrer une durée limite de validité des codes, etc. Il y a de quoi faire !

CONCLUSION

Faites des recherches sur Internet : je pense que 99,9% des documents traitant de l'envoi de SMS n'ont été rédigés qu'à des fins de tests (et je n'ai pas trouvé les 0,1% restant, mais j'espère quand même qu'ils existent). Il est impossible avec les informations fournies par ces sites de mettre en place un projet stable utilisant les SMS en émission et réception. J'espère donc avoir été dans cet article le plus clair et le plus complet possible. Ce qui est écrit dans ces pages tourne actuellement en production depuis quelques semaines et fonctionne plutôt bien.

Une dernière remarque : si vous souhaitez exploiter l'analyse de SMS reçus, préférez un nouvel abonnement à une carte SIM jumelle : vous perdriez tous les SMS qui ne sont pas destinés au système puisque c'est ce dernier qui les réceptionnerait tous !

À vous maintenant de faire preuve d'imagination et d'utiliser tout ce que nous avons vu pour automatiser tout et n'importe quoi... ■

RÉFÉRENCES

[1] ARMAND J.-M., *« Envoyez des SMS avec un Raspberry Pi et Python »*, *GNU/Linux Magazine HS n°90*, mai 2017 : http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMFHS-090/Envoyez-des-SMS-avec-un-Raspberry-Pi-et-Python

[2] COLOMBO T., *« Lire des mails avec l'API Gmail »*, *GNU/Linux Magazine n°185*, septembre 2015, http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-185/Lire-des-mails-avec-l-API-Gmail

[3] Les différents (deux) scopes de l'API Calendar : https://developers.google.com/identity/protocols/googlescopes#calendarv3

[4] COLOMBO T., *« Lire un fichier ini »*, *GNU/Linux Magazine HS n°86*, septembre 2016 : http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMFHS-086/Lire-un-fichier-ini

[5] Méthode list() sur Events : https://developers.google.com/google-apps/calendar/v3/reference/events/list

[6] Objet Events : https://developers.google.com/google-apps/calendar/v3/reference/events

ACTUELLEMENT DISPONIBLE

MISC N°94 !

CERT, CSIRT ET SOC EN PRATIQUE :

COMMENT S'ORGANISER ET QUELS OUTILS METTRE EN PLACE

NE LE MANQUEZ PAS

CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

https://www.ed-diamond.com

RÉALISER UNE API REST AVEC GO

YANNICK HEINRICH
[Ingénieur passionné]

MOTS-CLÉS : GO, GIN, HTTP, WEB, REST, JSON

Choisir une architecture REST pour développer son API constitue rarement une décision difficile. Sélectionner une technologie se révèle plus cornélien devant l'immense catalogue de bibliothèques et langages disponibles. Ne vous trompez plus et choisissez Go.

De plus en plus d'entreprises se laissent séduire par le petit géomyidé pour développer leurs API : géants de l'Internet, de la pharmaceutique, des médias et bien d'autres encore **[1]**. **Go** est également au cœur de plusieurs outils visant à simplifier le déploiement, l'architecture et l'orchestration d'infrastructures. Parmi eux se trouvent des technologies largement adoptées par le plus grand nombre : **Docker**, **SwarmKit**, **consule**, **etcd** ou **Terraform**.

L'évocation de ces utilitaires soulève une question : pour une problématique infiniment moins ambitieuse, Go n'est-il pas trop complexe à mettre en œuvre ? La réponse est négative.

Le code des différents exemples se base sur la version 1.8 du langage et est accessible sur **GitHub [2]**.

1. PRÉSENTATION BRÈVE DU LANGAGE

Nous allons résumer les points forts de Go. Pour une introduction plus détaillée, nous invitons le lecteur à consulter les différentes ressources disponibles, dont l'excellente présentation faite dans le hors-série n°63 **[3]**.

À l'heure où les langages tendent vers la richesse des concepts et des écritures, les concepteurs de Go ont parié sur la simplicité pour favoriser un code simple à comprendre et à adopter. Bien qu'imaginé originellement pour des applications serveurs, Go est largement sorti de ce cadre.

1.1 Généralités

Go est un langage fortement typé qui se rapproche du langage **C** tout en soulageant le développeur de la gestion de

la mémoire par l'utilisation d'un ramasse-miette. Il n'inclut pas de générique et n'utilise pas d'exceptions pour les erreurs.

Toujours dans un souci de simplification, l'application de la programmation orientée objet est réduite à la composition et à la création de méthodes sans surcharge.

Toutes ces notions sont regroupées dans le programme ci-dessous à l'utilité toute relative :

```
package main

import (
 "fmt"
 "os"
)

type Utilisateur struct {
 Email      string
 motDePasse string // Champ privé (minuscule)
}

func (u Utilisateur) modifieCopie(email string)
{
 u.Email = email
}

func (u *Utilisateur) modifieReference(email
string) {
 u.Email = email
}

type Administrateur struct {
 Utilisateur
 Niveau int
}

func main() {

 u := Utilisateur{Email: "linux@linux.com",
motDePasse: "secret"}

 u.modifieCopie("autre@linux.com")
 fmt.Println(u.Email)
 // Affiche: linux@linux.com

 u.modifieReference("autre@linux.com")
 fmt.Println(u.Email)
 // Affiche: autre@linux.com

 a := &Administrateur{Utilisateur: u, Niveau: 0}

 file, err := os.Create("administrateur.txt")
 if err != nil {
  panic(err)
 }

 defer file.Close()
 file.WriteString(a.Email)
}
```

1.2 Interfaces

Le polymorphisme est traduit en Go par le concept d'interface. Celui-ci est similaire aux interfaces rencontrées en **Java** : une interface est un ensemble de méthodes qu'un objet doit implémenter s'il veut être considéré comme compatible avec cette même interface. À la différence de Java, il n'est pas nécessaire d'importer la définition de l'interface lors de son implémentation :

```
 Package main

type Animal interface {
 Noise() string
}

type Dog struct{}

func (d *Dog) Noise() string {
 return "waf"
}

type Cat struct{}

func (c Cat) Noise() string {
 return "miaou"
}

func main() {
 var animalA Animal = &Dog{}
 var animalB Animal = Cat{}
 fmt.Println("Animal A:", animalA.Noise())
 fmt.Println("Animal B:", animalB.Noise())
}
```

1.3 Goroutine

Une des forces de Go réside dans son modèle de gestion de la concurrence. L'unité d'ordonnancement n'est pas le *thread*, mais la *goroutine*. Nous pouvons la voir comme un *green thread* au faible coût qui s'exécute dans un bassin géré par le moteur d'exécution de Go. La synchronisation entre différentes goroutines se fait par des *channels* qui échangent de l'information à la manière de ce que l'on retrouve en **Erlang** ou en **Rust** :

```
package main

func work(ch chan bool, n time.Duration) {
 time.Sleep(n * time.Millisecond)
 fmt.Printf("%d ms\n", n)
 ch <- true
}

func main() {
 ch := make(chan bool)
 go work(ch, 1000)
 go work(ch, 600)
 go work(ch, 300)
 <-ch
 <-ch
 <-ch
 fmt.Println("Finished!")
}
```

NOTE

Il est possible d'utiliser les exclusions mutuelles si les goroutines et canaux ne conviennent pas à votre problématique (protection de ressources par exemple). Vous les trouverez dans le *package* **sync**.

2. UN SERVEUR HTTP AVEC LA BIBLIOTHÈQUE STANDARD

La bibliothèque standard intègre une multitude de composants : compression, base de données **SQL**, **JSON**, **XML**, **TLS**, chiffrement... et HTTP bien évidemment.

Les briques élémentaires résident dans le composant **net/http**. Ainsi un serveur minimaliste prend la forme suivante :

```
package main

import (
 "fmt"
 "net/http"
 "log"
)

// Définition d'un HandlerFunc
func helloHandler(w http.
ResponseWriter, r *http.Request)
{

 fmt.Fprintf(w, "Hello world
\n")
}

func main() {
 // Démarrage du serveur
 log.Panicln(http.
ListenAndServe(":8080", http.
HandlerFunc(helloHandler)))
}
```

Lancez le programme avec **go run hello_world.go** et dans une autre fenêtre de votre terminal :

```
$ curl http://localhost:8080
Hello world
```

2.1 Les composants élémentaires

Le serveur HTTP est créé à partir de la méthode **ListenAndServe** :

```
func ListenAndServe(addr string, handler Handler) error
```

Celle-ci prend en paramètre l'adresse réseau du serveur sous forme d'une *string* et un objet qui implémente l'interface **Handler**.

Sa définition est la suivante :

```
type Handler interface {
        ServeHTTP(ResponseWriter, *Request)
}
```

Dès qu'une requête est reçue par notre serveur, celui-ci démarre une nouvelle goroutine dans laquelle la méthode **ServeHTTP** est appelée.

L'objet **Request** représente la requête HTTP reçue et l'objet **ResponseWriter** nous permet de renvoyer une réponse à cette requête.

Il n'est pas nécessaire de créer un nouveau type pour chaque **Handler**. Nous pouvons définir une fonction de type **HandlerFunc**. Ce type, défini dans la bibliothèque standard, satisfait l'interface **Handler** de la manière suivante :

```
type HandlerFunc func(ResponseWriter, *Request)

type HandlerFunc func(ResponseWriter, *Request) {
 f(w, r)
}
```

NOTE

Il faut bien noter que nous recevons un pointeur vers un objet de type **Request** et une interface de type **ResponseWriter** (notez l'absence de *).

2.2 Rediriger les requêtes vers les handlers

Le serveur ne gère qu'un seul point d'entrée pour le moment. La bibliothèque standard offre la structure **ServeMux** pour gérer de manière minimaliste la redirection des requêtes. Elle permet d'associer des chemins d'URL à différents **Handlers** :

```
package main

import (
 "fmt"
 "net/http"
)

func helloHandler(w http.ResponseWriter, r *http.Request) {
 fmt.Fprintf(w, "Hello world \n")
}

func main() {
 // Source : https://golang.org/pkg/net/http/
 mux := http.NewServeMux()
 mux.Handle("/hello/", http.HandlerFunc(helloHandler))

 mux.HandleFunc("/", func(w http.ResponseWriter, req *http.Request) {
```

```
    // Le motif "/" correspond à n'importe quel
chemin,
    // c'est pourquoi il faut vérifier si la
requête correspond au chemin racine.
    if req.URL.Path != "/" {
      http.NotFound(w, req)
      return
    }
    fmt.Fprintf(w, "Bienvenue sur la page racine!")
  })
  http.ListenAndServe(":8080", mux)
}
```

Le package **net/http** possède une instance de **ServeMux** par défaut (variable **DefaultServeMux**) qui nous permet d'alléger légèrement le code :

```
package main

import (
  "fmt"
  "net/http"
)

func helloHandler(w http.ResponseWriter, r *http.
Request) {
  fmt.Fprintf(w, "Hello world \n")
}

func main() {
  // Équivaut à http.DefaultServeMux.Handle
  http.Handle("/hello/", http.
HandlerFunc(helloHandler))

  http.HandleFunc("/", func(w http.ResponseWriter, req
*http.Request) {
    if req.URL.Path != "/" {
      http.NotFound(w, req)
      return
    }
    fmt.Fprintf(w, "Bienvenue sur la page racine!")
  })
  // Passer nil équivaut à passer http.DefaultServeMux
  http.ListenAndServe(":8080", nil)
}
```

Nous pouvons utiliser **curl** pour interroger notre API :

```
$ curl http://localhost:8080/hello/
Hello world
$ curl http://localhost:8080/
Bienvenue sur la page racine!%
$ curl http://localhost:8080/hello
<a href="/hello/">Moved Permanently</a>.
```

Il faut remarquer que pour **ServeMux**, l'appel **GET /hello** et **GET /hello/** ne sont pas les mêmes.

2.3 Répondre aux requêtes

L'interface **ResponseWriter** nous permet de formuler notre réponse. L'interface possède trois méthodes :

- **Write([]byte) (int, error)** : cette méthode nous permet de créer le contenu de la réponse HTTP ;
- **WriteHeader(int)** : cette méthode nous permet de spécifier un code de statut HTTP. Si elle n'est pas spécifiquement appelée, la valeur **200** sera retournée par défaut ;
- **Header() Header** : cette méthode nous permet de spécifier la valeur des entêtes HTTP de la réponse.

Avec ces méthodes, il est assez facile de renvoyer une réponse au format JSON ou XML en utilisant les paquets **encoding/json** ou **encoding/xml** :

```
package main

import (
  "encoding/json"
  "encoding/xml"
  "net/http"
  "time"
)

// Un objet représentant une nouvelle
type News struct {
  XMLName      xml.Name  'json:"-" xml:"nouvelle"'
  Titre        string    'json:"titre" xml:"titre"'
  Contenu      string    'json:"contenu" xml:"contenu"'
  DateCreation time.Time 'json:"date" xml:"date,attr"'
}

var (
  // Notre liste de nouvelles
  listNews = []News{
    News{Titre: "Titre 1", Contenu: "Contenu 1", DateCreation:
time.Now()},
    News{Titre: "Titre 2", Contenu: "Contenu 2", DateCreation:
time.Now().Add(50 * time.Minute)},
  }
)

func main() {
  http.HandleFunc("/nouvelles.json", func(w http.
ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    if err := json.NewEncoder(w).Encode(listNews); err != nil {
      http.Error(w, "Impossible de créer le JSON", http.
StatusInternalServerError)
    }
  })

  http.HandleFunc("/nouvelles.xml", func(w http.ResponseWriter,
r *http.Request) {
    w.Header().Set("Content-Type", "application/xml")
    if err := xml.NewEncoder(w).Encode(listNews); err != nil {
      http.Error(w, "Impossible de créer le XML", http.
StatusInternalServerError)
    }
  })

  http.ListenAndServe(":8080", nil)
}
```

Nous pouvons valider notre API en utilisant **curl** à nouveau :

```
$ curl http://localhost:8080/nouvelles.json
[{"titre":"Titre 1","contenu":"Contenu
1","date":"2017-04-
02T17:21:31.658807289+02:00"},{"titre":"Tit
re 2","contenu":"Contenu 2","date":"2017-04-
02T18:11:31.658807386+02:00"}]
$ curl http://localhost:8080/nouvelles.xml
<nouvelle date="2017-04-02T17:20:09.210342869+
02:00"><titre>Titre 1</titre><contenu>Contenu
1</contenu></nouvelle><nouvelle date="2017-04-
02T18:10:09.210342922+02:00"><titre>Titre 2</
titre><contenu>Contenu 2</contenu></nouvelle>
```

NOTE

La sérialisation se base par défaut sur l'introspection (utilisation du paquet **reflect**). Les éléments entre *backquotes* dans la déclaration de l'objet **News** définissent la correspondance entre les propriétés de la structure et celle du fichier JSON ou XML. Il est possible de personnaliser cette étape en en implémentant l'interface **json.Unmarshaler** ou **xml.Unmarshaler**.

2.4 Limitations de la bibliothèque standard

Les programmes d'exemple présentés plus haut peuvent largement se satisfaire de la bibliothèque standard.

Ses limites se retrouvent très vite franchies lors de la réalisation d'applications plus complexes comme une **API REST**. D'après les exemples précédents, il est déjà possible d'en observer trois :

- L'entité **ServeMux** est limitée : vous ne pouvez associer un **Handler** qu'à un chemin d'URL. Ce que nous souhaitons, c'est associer un couple chemin d'URL et méthode HTTP à un **Handler**.
- Aucune trace ne nous permet de suivre l'historique des requêtes.
- Si une goroutine dans laquelle un **Handler** s'exécute panique (**panic**), le serveur s'arrête immédiatement sans répondre aux autres requêtes en cours.

Les concepteurs du langage n'ont pas imposé de solution, car différentes réponses existent selon les cas.

Une réponse au premier problème serait de créer notre propre entité **Routeur** plus flexible que celui par défaut. GitHub regorge de paquets de ce type et nous pouvons nous reposer sereinement sur les plus testés d'entre eux **[4]**.

Une réponse aux deux autres serait l'utilisation d'un système de *middlewares* à l'instar de ce que l'on retrouve dans **Rack** en Ruby **[5]** ou **ExpressJS** avec node **[6]**. Un *middleware* gèrerait les appels à **panic** et un autre tracerait l'historique des requêtes reçues.

De la même manière, de nombreux *micro-framework* permettant d'architecturer nos serveurs sous forme de *middlewares* se trouvent sur GitHub **[7]**.

En combinant les deux, on obtient une base souple et solide pour démarrer sereinement une application.

On retrouve aussi des *frameworks* HTTP qui combinent les deux en plus d'offrir méthodes simplifiées pour la génération des réponses **[8]**.

3. GIN POUR ACCÉLÉRER LE DÉVELOPPEMENT

3.1 Installer Gin

Nous pouvons installer **Gin** dans notre *workspace* en lançant la commande suivante :

```
$ go get -u -v gopkg.in/gin-gonic/gin.v1
```

Créons un fichier **main.go** et reprenons l'exemple proposé par la documentation :

```
package main

import "gopkg.in/gin-gonic/gin.v1"

func main() {
 r := gin.Default()
 r.GET("/ping", func(c *gin.Context) {
  c.JSON(200, gin.H{
   "message": "pong",
  })
 })
 r.Run() // écoute automatiquement sur le
port 8080
}
```

Ouvrons un terminal pour lancer le serveur :

```
$ go run main.go
```

Testons avec **curl** dans un autre :

```
$ curl http://localhost:8080/ping
{"message":"pong"}%
```

Nous pouvons démarrer l'écriture de notre API REST.

3.2 Une API basique avec Gin

Nous allons réaliser une API REST pour gérer un fil de nouvelles :

- **GET /news** renverra l'ensemble des nouvelles ;
- **GET /news/{id}** renverra la nouvelle identifiée par id ;
- **PUT /news** ajoutera une nouvelle à la liste. L'objet à rajouter est envoyé en utilisant JSON ;
- **POST /news/{id}** modifiera la nouvelle identifiée par id. Les modifications seront envoyées sous forme de JSON ;
- **DELETE /news/{id}** supprimera la news identifiée par id.

Le projet sera créé dans le *workspace* de l'utilisateur. Nous découpons le projet en plusieurs paquets :

```
$ mkdir -p $GOPATH/src/glmf.org/api_news
$ tree
.
├── models  # Les entités manipulées
├── server  # Le code principal du serveur
└── store  # Le code principal du store
```

3.2.1 L'objet News

Dans le dossier **models**, créons un fichier **news.go** qui contiendra notre objet **News**:

```
package models

import "time"

// News représente notre nouvelle
type News struct {
 ID           int       'json:"id"'
 Title        string    'json:"title"'
 Content      string    'json:"content"'
 CreatedAt    time.Time 'json:"created_at"'
 LastModified time.Time 'json:"last_modified"'
}
```

La sérialisation des objets de type **Time** utilise la RFC 3339 par défaut **[9]**.

3.2.2 L'objet Store

Nous allons créer une interface pour représenter l'objet qui gèrera la persistance de nos objets. Cette abstraction présente l'avantage de rendre les tests plus aisés par la suite.

```
package store

import (
 "errors"
 "github.com/yageek/glmf-go-api/models"
)

var (
 ErrNewsNotFound     = errors.New("The news was not found.")
 ErrCanNotDeleteNews = errors.New("Can not delete news")
)

type Store interface {
 GetAllNews() ([]models.News, error)
 CreateNews(title, content string) (models.News, error)
 UpdateNews(ID int, news models.News) (models.News, error)
 DeleteNews(ID int) error
 GetNews(ID int) (models.News, error)
}
```

3.2.3 Le contexte Gin

La redirection utilise des éléments de type **gin.HandlerFunc**. Celui-ci correspond à une fonction prenant en paramètre une instance de type ***gin.Context**.

L'instance ***gin.Context** nous permettra d'interagir avec la requête et la réponse HTTP. Il possède également des fonctions utilitaires pour répondre à la requête sans avoir à manipuler directement l'instance **ResponseWriter**. L'extrait de code est un simple aperçu de toutes les fonctions disponibles.

```
// Extrait du package gin
type HandlerFunc func(*Context)

type Context struct {
      /* ... */
 Request   *http.Request // La requête reçue
 Writer    ResponseWriter // L'instance du ResponseWriter
 Params   Params // Les paramètres de la requête. Dans notre
cas, l'élément :id pour
 /* ... */
}

// Répondre avec une erreur.
func (c *Context) AbortWithError(code int, err error) *Error {
/* ... */
}
// Répondre avec un fichier JSON
func (c *Context) JSON(code int, obj interface{}) {
/* ... */
}
// Répondre avec un simple statut
func (c *Context) Status(code int) {
}
```

Nous pouvons définir l'API comme précisé en introduction. Les différents *handlers* renverront pour le moment un code de statut **404**.

```
package main

import (
 "errors"
```

```
  "net/http"

  "github.com/yageek/glmf-go-api/store"
  "gopkg.in/gin-gonic/gin.v1"
)

var (
  newsStore          store.Store
  ErrNotImplemented = errors.New("Not implemented yet.")
)

// API News
func GetAllNews(c *gin.Context) {
  c.AbortWithError(http.StatusNotFound, ErrNotImplemented)
}
func DeleteNews(c *gin.Context) {
  c.AbortWithError(http.StatusNotFound, ErrNotImplemented)
}
func UpdateNews(c *gin.Context) {
  c.AbortWithError(http.StatusNotFound, ErrNotImplemented)
}

func GetNews(c *gin.Context) {
  c.AbortWithError(http.StatusNotFound, ErrNotImplemented)
}

func CreateNews(c *gin.Context) {
  c.AbortWithError(http.StatusNotFound, ErrNotImplemented)
}

func main() {
  r := gin.Default()

  // Routeur
  r.GET("/news", GetAllNews)
  r.GET("/news/:id", GetNews)
  r.PUT("/news", CreateNews)
  r.POST("/news/:id", UpdateNews)
  r.DELETE("/news/:id", DeleteNews)

  // Lancement
  r.Run()
}
```

3.2.4 Implémentation sans persistance

Nous allons commencer par implémenter un store sans persistance. Nous devons seulement nous souvenir que chaque requête HTTP est exécutée dans une nouvelle goroutine, ce qui implique l'utilisation d'une exclusion mutuelle.

Nous utilisons un tableau associatif pour stocker les nouvelles selon leur identifiant. Celui-ci est simplement un entier qui est incrémenté lors de la création d'une nouvelle instance de **News**.

```
package store

import (
  "sync"
  "time"

  "github.com/yageek/glmf-go-api/models"
)
```

```
type MemoryStore struct {
  Contents map[int]models.News
  // Le mutex nécessaire à la
synchronisation entre goroutines.
  mu       sync.Mutex
  counter int
}

func NewMemoryStore() *MemoryStore {
  return &MemoryStore{Contents: map[int]
models.News{}}
}

func (m *MemoryStore) GetAllNews() ([]
models.News, error) {
  m.mu.Lock()
  defer m.mu.Unlock()

  resultSet := make([]models.News, len(m.
Contents))
  index := 0
  for _, value := range m.Contents {
    resultSet[index] = value
    index++
  }
  return resultSet, nil
}

func (m *MemoryStore) CreateNews(title,
content string) (models.News, error) {
  m.mu.Lock()
  defer m.mu.Unlock()

  t := time.Now()
  news := models.News{
    ID:           m.counter,
    Title:        title,
    Content:      content,
    CreatedAt:    t,
    LastModified: t,
  }
  m.Contents[news.ID] = news
  m.counter++
  return news, nil
}

func (m *MemoryStore) UpdateNews(ID int,
news models.News) (models.News, error) {
  m.mu.Lock()
  defer m.mu.Unlock()

  storeNews, exist := m.Contents[ID]
  if !exist {
    return models.News{}, ErrNewsNotFound
  }
  storeNews.Content = news.Content
  storeNews.Title = news.Title
  storeNews.LastModified = time.Now()
  m.Contents[ID] = storeNews
  return storeNews, nil
}

func (m *MemoryStore) DeleteNews(ID int)
error {
  m.mu.Lock()
  defer m.mu.Unlock()
```

```
 if _, exist := m.Contents[ID]; !exist {
  return ErrNewsNotFound
 }
 delete(m.Contents, ID)
 return nil
}

func (m *MemoryStore) GetNews(ID int)
(models.News, error) {
 m.mu.Lock()
 defer m.mu.Unlock()

 storeNews, exist := m.Contents[ID]
 if !exist {
  return models.News{}, ErrNewsNotFound
 }
 return storeNews, nil
}
```

Nous pouvons créer une instance dans notre serveur dans **server/main.go** :

```
var (
 newsStore          store.Store = store.
NewMemoryStore()
 ErrNotImplemented             = errors.
New("Not implemented yet.")
)
```

Pour l'ajout de nouvelles, nous avons simplement besoin de connaître le titre et le contenu de la nouvelle.

Pour ce faire, nous déclarons un nouvel objet qui contiendra le contenu de la requête sur **PUT /news** et nous l'utilisons dans notre **Handler** :

```
type CreateRequest struct {
 Title   string 'json:"title" binding:"required"'
 Content string 'json:"content"
binding:"required"'
}

func CreateNews(c *gin.Context) {
 news := CreateRequest{}
 if err := c.BindJSON(&news); err != nil {
  c.AbortWithError(http.StatusBadRequest, err)
  return
 }

 dbNews, err := newsStore.CreateNews(news.Title,
news.Content)
 if err != nil {
  c.AbortWithError(http.StatusNotFound,
ErrNotImplemented)
  return
 }
 c.JSON(http.StatusCreated, dbNews)
}
```

La fonction **BindJSON** de l'objet **Context** se sert des tags *bindings* pour déterminer quels sont les éléments obligatoires. Si l'un venait à manquer, la fonction retournerait une erreur et nous informerions l'appelant via un code de statut **400** (*Bad Request*).

```
$ curl -i -H "Content-Type: application/json" -X PUT
-d '{"title": "Titre"}' http://localhost:8080/news
HTTP/1.1 400 Bad Request
Content-Length: 0
Content-Type: text/plain; charset=utf-8
Date: Thu, 13 Apr 2017 13:35:38 GMT
```

Si nous envoyons tous les champs requis, une nouvelle est bien créée par notre **Store** :

```
$ curl -i -H "Content-Type: application/json"
-X PUT -d '{"title": "Titre", "content": "Du
contenu"}' http://localhost:8080/news
HTTP/1.1 201 Created
Content-Length: 152
Content-Type: application/json; charset=utf-8
Date: Thu, 13 Apr 2017 13:36:38 GMT

{"id":0,"title":"Titre","content":"
Du contenu","created_at":"2017-04-
13T15:36:38.660871616+02:00","last_
modified":"2017-04-13T15:36:38.660871616+02:00"}%
```

Nous pouvons maintenant implémenter le *handler* **GET /news** :

```
// API News
func GetAllNews(c *gin.Context) {
 news, err := newsStore.GetAllNews()
 if err != nil {
  c.AbortWithError(http.StatusInternalServerError, err)
  return
 }

 c.JSON(http.StatusOK, news)
}
```

Les trois autres *handlers* nécessitent un paramètre d'URL qui représente l'identifiant de la nouvelle. Nous pouvons retrouver ce paramètre via la méthode **Param** de l'objet **Context**. Cette méthode renvoie un objet de type *string* que nous convertissons en entier pour satisfaire l'interface **Store** :

```
func DeleteNews(c *gin.Context) {
 id, err := strconv.ParseInt(c.Param("id"), 10, 32)
 if err != nil {
  c.AbortWithError(http.StatusBadRequest, err)
  return
 }

 if err := newsStore.DeleteNews(int(id)); err != nil {
  c.AbortWithError(http.StatusNotFound,
ErrNotImplemented)
  return
 }
```

```
 c.JSON(http.StatusOK, nil)

}

func UpdateNews(c *gin.Context) {
 id, err := strconv.ParseInt(c.
Param("id"), 10, 32)
 if err != nil {
  c.AbortWithError(http.
StatusBadRequest, err)
  return
 }

 news := models.News{}
 if err := c.BindJSON(&news); err !=
nil {
  c.AbortWithError(http.
StatusBadRequest, err)
  return
 }

 updatedNews, err := newsStore.
UpdateNews(int(id), news)
 if err != nil {
  c.AbortWithError(http.
StatusNotFound, ErrNotImplemented)
  return
 }

 c.JSON(http.StatusOK, updatedNews)
}

func GetNews(c *gin.Context) {
 id, err := strconv.ParseInt(c.
Param("id"), 10, 32)
 if err != nil {
  c.AbortWithError(http.
StatusBadRequest, err)
  return
 }

 news, err := newsStore.
GetNews(int(id))
 if err != nil {
  c.AbortWithError(http.
StatusBadRequest, err)
  return
 }
 c.JSON(http.StatusOK, news)
}
```

Nous pouvons vérifier que l'API fonctionne correctement. Redémarrons le serveur dans une session :

```
$ go run server.go
```

Dans une autre session, utilisons **curl** pour réaliser les requêtes. Nous allons ajouter deux nouvelles puis demander la liste de toutes les nouvelles :

```
$ curl -i -H "Content-Type: application/json" -X PUT -d '{"title":
"Titre", "content": "Du contenu"}' http://localhost:8080/news
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Date: Thu, 20 Apr 2017 14:10:55 GMT
Content-Length: 152
Proxy-Connection: Keep-alive

{"id":0,"title":"Titre","content":"Du contenu","created_at":"2017-
04-20T16:10:55.258941744+02:00","last_modified":"2017-04-
20T16:10:55.258941744+02:00"}%
$ curl -i -H "Content-Type: application/json" -X PUT -d '{"title":
"Titre", "content": "Du contenu"}' http://localhost:8080/news
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Date: Thu, 20 Apr 2017 14:11:15 GMT
Content-Length: 152
Proxy-Connection: Keep-alive

{"id":1,"title":"Titre","content":"Du contenu","created_at":"2017-
04-20T16:11:15.023942753+02:00","last_modified":"2017-04-
20T16:11:15.023942753+02:00"}%

$ curl -i -X GET http://localhost:8080/news
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Thu, 20 Apr 2017 14:14:35 GMT
Content-Length: 307
Proxy-Connection: Keep-alive

[{"id":0,"title":"Titre","content":"Du contenu","created_at":"2017-
04-20T16:10:55.258941744+02:00","last_modified":"2017-04-20T16
:10:55.258941744+02:00"},{"id":1,"title":"Titre","content":"Du
contenu","created_at":"2017-04-20T16:11:15.023942753+02:00","last_
modified":"2017-04-20T16:11:15.023942753+02:00"}]
```

Nous pouvons essayer de modifier la news **0** et supprimer la news **1** :

```
$ curl -i -H "Content-Type: application/json" -X POST -d
'{"title": "Titre modifie", "content": "Du contenu modifie"}'
http://localhost:8080/news/0
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Thu, 20 Apr 2017 14:16:14 GMT
Content-Length: 168
Proxy-Connection: Keep-alive

{"id":0,"title":"Titre modifie","content":"Du contenu
modifie","created_at":"2017-04-20T16:10:55.258941744+02:00","last_
modified":"2017-04-20T16:16:14.603211241+02:00"}%
$ curl -i -X DELETE http://localhost:8080/news/1
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Thu, 20 Apr 2017 14:16:42 GMT
Content-Length: 4
Proxy-Connection: Keep-alive
```

```
null%
$ curl -i -X GET http://localhost:8080/news/0
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Thu, 20 Apr 2017 14:17:14 GMT
Content-Length: 168
Proxy-Connection: Keep-alive

{"id":0,"title":"Titre modifie","content":"Du
contenu modifie","created_at":"2017-
04-20T16:10:55.258941744+02:00","last_
modified":"2017-04-
20T16:16:14.603211241+02:00"}%
curl -i -X GET http://localhost:8080/news/1
HTTP/1.1 400 Bad Request
Date: Thu, 20 Apr 2017 14:17:28 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8
Proxy-Connection: Keep-alive
```

3.3 Les middlewares Gin

À l'instar de nombreux *frameworks* HTTP, Gin propose le concept de *middlewares*. Dans Gin, un *middleware* est simplement une fonction **`gin.HandlerFunc`** dans laquelle la méthode **`Next`** de **`gin.Context`** est appelée.

Dans le code précédant, le code récupérant l'identifiant de la nouvelle est dupliqué trois fois. En créant notre propre *middleware*, nous pouvons réduire un petit peu cette duplication :

```
var newsIDKey = "news_request_key"
func newsIDMiddleware(c *gin.Context) {
 id, err := strconv.ParseInt(c.Param("id"), 10, 32)
 if err != nil {
  c.AbortWithError(http.StatusBadRequest, err)
  return
 }
 c.Set(newsIDKey, int(id))
 c.Next()
}
```

Ce qui nous permet de réduire le code de nos trois *handlers* :

```
func DeleteNews(c *gin.Context) {
 id, _ := c.Get(newsIDKey)
 if err := newsStore.DeleteNews(id.(int));
err != nil {
  c.AbortWithError(http.StatusNotFound,
ErrNotImplemented)
  return
 }

 c.JSON(http.StatusOK, nil)
}
```

```
func UpdateNews(c *gin.Context) {
 id, _ := c.Get(newsIDKey)
 news := models.News{}
 if err := c.BindJSON(&news); err != nil {
  c.AbortWithError(http.StatusBadRequest, err)
  return
 }
 updatedNews, err := newsStore.UpdateNews(id.(int), news)
 if err != nil {
  c.AbortWithError(http.StatusNotFound,
ErrNotImplemented)
  return
 }

 c.JSON(http.StatusOK, updatedNews)
}

func GetNews(c *gin.Context) {

 id, _ := c.Get(newsIDKey)
 news, err := newsStore.GetNews(id.(int))
 if err != nil {
  c.AbortWithError(http.StatusBadRequest, err)
  return
 }
 c.JSON(http.StatusOK, news)
}
```

Nous déclarons ensuite le *middleware* dans les règles de routage dans **`server/main.go`** :

```
func main() {
 r := gin.Default()
 r.GET("/news", GetAllNews)
 r.PUT("/news", CreateNews)

 r.GET("/news/:id", newsIDMiddleware, GetNews)
 r.POST("/news/:id", newsIDMiddleware, UpdateNews)
 r.DELETE("/news/:id", newsIDMiddleware, DeleteNews)
 r.Run()
}
```

4. PERSISTANCE AVEC MARIADB

Le paquet **`database/sql`** de la bibliothèque standard nous permet d'interagir avec les bases de données transactionnelles. Ce paquet est agnostique vis-à-vis de la technologie utilisée (**MySQL**/**MariaDB**, **PostgreSQL**, etc.). Dans le cadre de cet article, nous utiliserons MariaDB **[10]**.

4.1 Préparation de la base de données

Par souci de simplicité, une image **Docker** a été publiée sur le GitHub avec le fichier SQL suivant :

```
CREATE DATABASE glmf_go;
USE glmf_go;
CREATE TABLE news (
    id bigint auto_increment primary key,
    title tinytext,
    content text,
    created_at datetime not null,
    modified_at datetime not null
)
```

La commande suivante permet de la télécharger et de la démarrer en arrière-plan :

```
$ docker run -p 127.0.0.1:3306:3306 -d yageek/glmf-go
```

NOTE

L'option **-p** détermine le lien que Docker doit réaliser entre le conteneur et le système d'exploitation. En l'occurrence, nous faisons un lien entre le port **3306** du conteneur et le port **3306** du système d'exploitation. L'option **-d** lance le conteneur en arrière-plan (pour *daemon*).

4.2 Importation d'un driver

La liaison avec la base de données se fait via l'import d'un paquet qui implémente différentes interfaces définies dans **database/sql/driver**. Nous utiliserons le driver **github.com/go-sql-driver/mysql** pour notre programme.

Pour l'installer :

```
$ go get -u -v github.com/go-sql-driver/mysql
```

Un driver se charge en utilisant l'identifiant vide (*blank*) **[11]**.

```
import _ "github.com/go-sql-driver/mysql"
```

Ce type d'importation est utilisé uniquement pour forcer l'initialisation d'un paquet. Dans sa fonction **init**, le driver s'enregistre auprès de la bibliothèque standard en précisant un nom unique via une chaîne de caractères.

```
func init() {
 sql.Register("mysql", &MySQLDriver{})
}
```

4.3 Un store avec MariaDB

Nous pouvons maintenant créer un nouvel objet **store** qui se connectera à MariaDB pour stocker nos nouvelles. La connexion à la base de données se fait en utilisant la fonction **sql.Open** en précisant le nom du driver ainsi que les informations nécessaires à la connexion :

```
package store
import (
 "database/sql"
 _ "github.com/go-sql-driver/mysql"
)

type MariaDBStore struct {
 db *sql.DB
}

func NewMariaDBStore(address string)
(*MariaDBStore, error) {

 if db, err := sql.Open("mysql", address);
err != nil {
  return nil, err
 } else {
  return &MariaDBStore{db}, nil
 }
}
```

La méthode **Exec** de l'objet **sql.DB** nous permet d'exécuter des requêtes SQL ne retournant pas d'enregistrement. Nous allons l'utiliser pour la création et la suppression des news :

```
func (s *MariaDBStore) CreateNews(title,
content string) (models.News, error) {
 now := time.Now()
 result, err := s.db.Exec("INSERT INTO news
(title, content, created_at, modified_at)
VALUES (?, ?, ?, ?)", title, content, now, now)
 if err != nil {
  return models.News{}, err
 }

 id, err := result.LastInsertId()
 if err != nil {
  return models.News{}, err
 }
 return models.News{int(id), title, content,
now, now}, nil
}
func (s *MariaDBStore) DeleteNews(ID int) error
{
 _, err := s.db.Exec("DELETE FROM news WHERE
id=?", ID)
 return err
}
```

Pour récupérer des données, nous devons utiliser la méthode **Query** (ou **QueryRow** pour une requête ne concernant qu'un seul résultat). La valeur de retour nous permet d'itérer sur les enregistrements.

Les requêtes SQL n'étant pas typées, il convient à l'auteur du code de correctement associer les valeurs de l'enregistrement aux bonnes valeurs : l'ordre des champs demandés dans la requête correspond à l'ordre dans lequel la méthode **Rows.Scan** va remplir ses arguments.

```
func (s *MariaDBStore) GetAllNews() ([]
models.News, error) {
 rows, err := s.db.Query("SELECT
id,title,content,created_at,modified_at
FROM news")
 if err != nil {
  return []models.News{}, err
 }
 defer rows.Close()

 type row struct {
  ID       int64
  News     string
  Content  string
  Title    string
  Created  time.Time
  Modified time.Time
 }
 got := []models.News{}

 for rows.Next() {
  var r = row{}
  err := rows.Scan(&r.ID, &r.Title,
&r.Content, &r.Created, &r.Modified)
  if err != nil {
   return []models.News{}, err
  }
  news := models.News{int(r.ID), r.Title,
r.Content, r.Created, r.Modified}
  if err != nil {
   return []models.News{}, err
  }
  got = append(got, news)
 }
 return got, nil
}
```

L'implémentation pour la méthode **GetNews** est similaire :

```
func (s *MariaDBStore) GetNews(ID int)
(models.News, error) {
 type row struct {
  ID       int64
  News     string
  Content  string
  Title    string
  Created  time.Time
  Modified time.Time
 }
 var r = row{}
 err := s.db.QueryRow("SELECT
id,title,content,created_at,modified_at FROM
news WHERE id=?", ID).Scan(&r.ID, &r.Title,
&r.Content, &r.Created, &r.Modified)
 if err != nil {
  return models.News{}, err
 }
 return models.News{int(r.ID), r.Title,
r.Content, r.Created, r.Modified}, nil
}
```

La méthode **UpdateNews** renvoie les nouvelles valeurs de la *news* une fois les modifications effectuées. Nous devrons exécuter deux requêtes SQL : une pour modifier la *news* et une autre pour récupérer la *news* après modification. Afin d'éviter tout problème de concurrence, nous devons utiliser des transactions SQL. Nous pouvons créer des transactions via la fonction **Begin** de l'objet **sql.db** :

```
func (s *MariaDBStore) UpdateNews(ID int, news
models.News) (models.News, error) {
 tx, err := s.db.Begin()
 if err != nil {
  return models.News{}, err
 }
 _, err = s.db.Exec("UPDATE news SET title = ?,
content = ?, modified_at = ? WHERE id=?", news.
Title, news.Content, time.Now(), ID)
 if err != nil {
  return models.News{}, err
 }
 updatedNews, err := s.GetNews(ID)
 if err != nil {
  return models.News{}, err
 }
 return updatedNews, tx.Commit()
}
```

Nous pouvons maintenant utiliser notre nouveau **store** dans notre application :

```
func main() {

 store, _ := store.NewMariaDBStore("root:password@/
glmf_go?parseTime=true")
 ...
}
```

4.4 Protéger son API avec JWT

En utilisant le système des *middlewares*, il est assez rapide de mettre en place un mécanisme d'authentification pour une API. L'utilisation des **JSON Web Token** (JWT) est très répandu et offre la possibilité de se soustraire à la gestion de sessions par le serveur HTTP **[12]**.

Un JWT est fait de trois composantes séparées par un **.** et de la forme **xxxx.yyyy.zzzz**. Chacune représente un élément JSON formaté en base64. Nous avons dans l'ordre l'entête, le corps et la signature.

L'entête définit le type de jeton ainsi que l'algorithme utilisé pour générer la signature. Par exemple :

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Les propriétés du corps regroupent des informations à propos du destinataire du jeton ainsi que certaines métadonnées. On parle de « Claims » pour chacun des couples clef/valeur. Certains « Claims » sont réservés par la norme comme **iss** (*issuer*) ou **exp** (*expiration time*).

```
{
  "iss": "glmf.org",
  "exp": 1300819380
  "admin": true
}
```

La signature est générée à partir des deux sections précédentes de la manière suivante :

```
ALGORITHME_SIGNATURE(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Le processus de demande et création d'un JWT est présenté en figure 1.

Une fois obtenu par le client, celui-ci le transmet dans sa requête via un entête HTTP pour la forme : **Authorization: Bearer <token>**.

Nous allons utiliser le *package* **github.com/square/go-jose** pour manipuler les JWT :

```
$ go get -u -v gopkg.in/
square/go-jose.v2/jwt
```

Différentes méthodes existent pour créer la signature, nous utiliserons simplement un mot de passe partagé avec une signature basée sur l'algorithme HS256 **[13]**.

Nous créons un paquet dédié à l'authentification :

```
$ mkdir authentication
$ touch authentication/jwt.go
```

Nous créons l'URL **POST /login** pour récupérer le *token* depuis l'API. Nous utiliserons un nom d'utilisateur et un mot de passe par défaut :

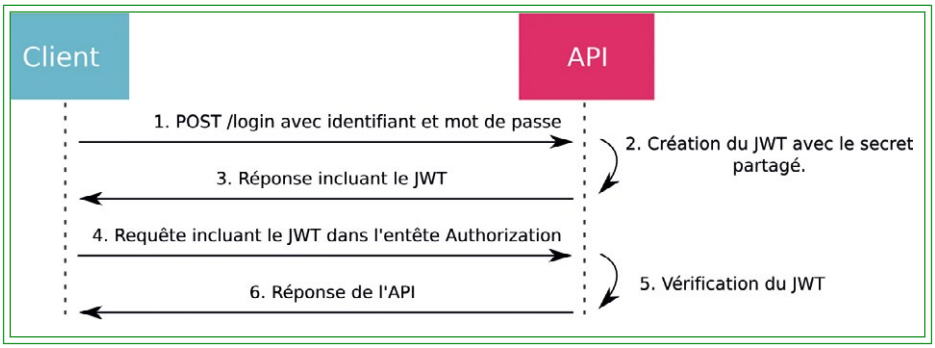


Fig. 1 : Processus d'authentification via JWT.

```
var (
 password  = "password"
 login     = "login"
)

func Login(c *gin.Context) {

 // Récupération de l'identifiant et du mot de passe
 loginRequest := LoginRequest{}
 err := c.BindJSON(&loginRequest)
 if err != nil {
  c.AbortWithError(http.StatusBadRequest, err)
  return
 }

 if loginRequest.Login != login || loginRequest.Password !=
password {
  c.AbortWithError(http.StatusUnauthorized, errors.
New("Invalid credentials"))
 }

 // Création du JWT à l'aide du mot de passe partagé
 signer, err := jose.NewSigner(jose.SigningKey{Algorithm:
jose.HS256, Key: jwtSecret}, (&jose.SignerOptions{}).
WithType("JWT"))
 if err != nil {
  c.AbortWithError(http.StatusInternalServerError, err)
  return
 }

 cl := jwt.Claims{
  Issuer:   "glmf-go-api",
  Audience: jwt.Audience{"glmf-go-api-client"},
 }

 raw, err := jwt.Signed(signer).Claims(cl).CompactSerialize()
 if err != nil {
  c.AbortWithError(http.StatusInternalServerError, err)
  return
 }
 c.String(http.StatusOK, raw)

}
```

Nous allons ensuite créer une fonction utilitaire qui extraira le JWT de la requête ainsi que le mot de passe partagé utilisé pour la signature :

```
package authentication

import (
 "errors"
 "net/http"
 "strings"

 "gopkg.in/square/go-jose.v2/jwt"
)

var jwtSecret = []byte("JWTSecret")

func extractToken(r *http.Request)
(*jwt.JSONWebToken, error) {
 authorizationHeader := r.Header.
Get("Authorization")
 if len(authorizationHeader) > 7
&& strings.EqualFold(authorization
Header[0:7], "BEARER ") {
   return jwt.ParseSigned(authoriza
tionHeader[7:])
 }
 return nil, errors.New("Token not
found")
}
```

Nous pouvons ensuite écrire le *middleware* :

```
func JWTMiddleware(c *gin.Context) {

 // Extraction du JWT
 token, err := extractToken(c.
Request)
 if err != nil {
   c.AbortWithError(http.
StatusUnauthorized, err)
   return
 }

 // Validation du JWT
 cl := jwt.Claims{
   Issuer:   "glmf-go-api",
   Audience: jwt.Audience{"glmf-go-
api-client"},
 }
 if err := token.Claims(jwtSecret,
&cl); err != nil {
   c.AbortWithError(http.
StatusUnauthorized, err)
   return
 }

 c.Next()
}
```

Nous utilisons ce code dans notre routeur :

```
func main() {

 store, err := store.NewMariaDBStore("root:password@/
glmf_go?parseTime=true")
 if err != nil {
  panic(err)
 }
 newsStore = store

 r := gin.Default()
 // URLs ne nécessitant pas d'authentification
 r.POST("/login", authentication.Login)

 r.GET("/news", GetAllNews)
 r.GET("/news/:id", newsIDMiddleware, GetNews)

 // URL ne nécessitant pas d'authentification
 r.PUT("/news", authentication.JWTMiddleware, CreateNews)
 r.POST("/news/:id", authentication.JWTMiddleware,
newsIDMiddleware, UpdateNews)
 r.DELETE("/news/:id", authentication.JWTMiddleware,
newsIDMiddleware, DeleteNews)
 r.Run()
}
```

Il est désormais impossible d'ajouter, de modifier ou de supprimer une nouvelle sans fournir de jeton d'authentification :

```
$ curl -i -H "Content-Type: application/json" -X PUT -d '{"title":
"Titre", "content": "Du contenu"}' http://localhost:8080/news
HTTP/1.1 401 Unauthorized
Date: Mon, 29 May 2017 15:24:35 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8
```

Nous devons dans un premier temps récupérer le *token* :

```
$ curl -i -H "Content-Type: application/json" -X POST -d '{"login":
"login", "password": "password"}' http://localhost:8080/login
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Date: Mon, 29 May 2017 15:55:31 GMT
Content-Length: 153

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOlsiY2xpZW50Il0sImlzcyI
6ImdsbWYtZ28tYXBpIiwic3ViIjoic3ViamVjdCJ9.hvFfGyFCgNQ5BUnLsbAnLLqCOq
KNNGYB1L-_YObzQlk
```

Et finalement, nous pouvons utiliser l'API en fournissant le token :

```
$ curl -i -H "Content-Type: application/json" -H "Authorization:
Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOlsiY2xp
ZW50Il0sImlzcyI6ImdsbWYtZ28tYXBpIiwic3ViIjoic3ViamVjd
CJ9.hvFfGyFCgNQ5BUnLsbAnLLqCOqKNNGYB1L-_YObzQlk" -X PUT -d '{"title":
"Titre", "content": "Du contenu"}' http://localhost:8080/news
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Date: Mon, 29 May 2017 15:59:30 GMT
Content-Length: 152
```

```
{"id":1,"title":"Titre","content":
"Du contenu","created_at":"2017-05-
29T17:59:30.918315115+02:00","last_
modified":"2017-05-
29T17:59:30.918315115+02:00"}%
```

CONCLUSION

L'écosystème du langage Go nous a permis de mettre en place rapidement une API REST avec un mécanisme d'authentification basé sur les JWT. Nous avons pu voir comment les interfaces nous permettent d'abstraire la couche de persistance et quels sont leurs rôles dans la redirection des requêtes au sein de la bibliothèque standard. ■

RÉFÉRENCES

[1] Page wiki *GoUser* : https://github.com/golang/go/wiki/GoUsers

[2] Code source des exemples : https://github.com/yageek/glmf-go-api

[3] COLOMBO T., ARMAND J.-M., *« Apprenez à programmer en Go! », GNU/Linux Magazine HS n°63*, décembre 2012 : https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMFHS-063

[4] *Go HTTP request router and web framework benchmark* : https://github.com/julienschmidt/go-http-routing-benchmark

[5] *Rack : a Ruby Webserver Interface* : http://rack.github.io

[6] Page d'accueil du projet ExpressJS : http://expressjs.com

[7] Projet awesome-go (middlewares) : https://github.com/avelino/awesome-go#middlewares

[8] Projet awesome-go (frameworks) : https://github.com/avelino/awesome-go#web-frameworks

[9] RFC3339 *Date and Time on the Internet: Timestamps* : https://www.ietf.org/rfc/rfc3339.txt

[10] Site officiel du projet MariaDB : https://mariadb.org

[11] La spécification du langage Go : https://golang.org/ref/spec#Blank_identifier

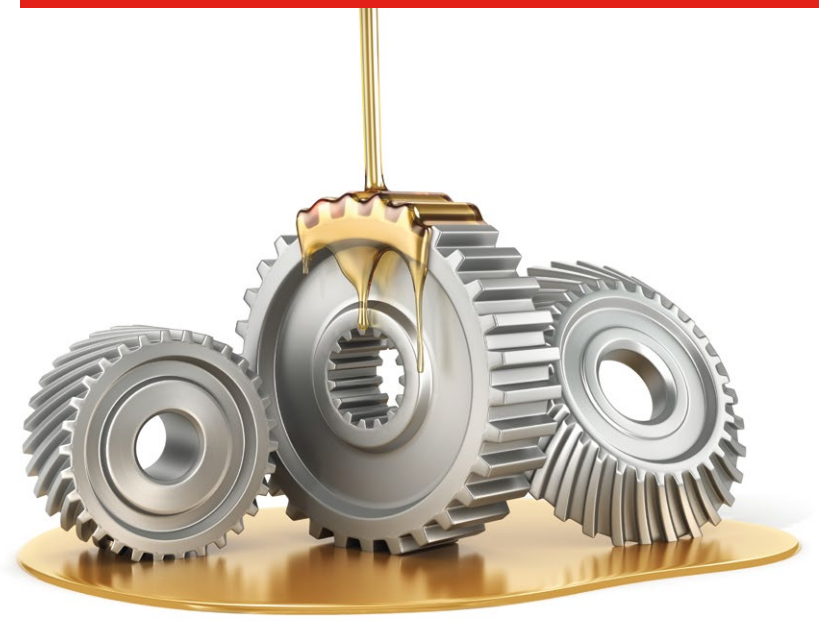
[12] RFC7519 *JSON Web Token (JWT)* : https://tools.ietf.org/html/rfc7519

[13] Page Wikipédia HMAC : https://en.wikipedia.org/wiki/Hash-based_message_authentication_code

LA TRILOGIE DU REVERSE ENGINEERING

SYLVAIN NAYROLLES
[Security Software Engineer]

MOTS-CLÉS : CAPSTONE, UNICORN, KEYSTONE, REVERSE ENGINEERING, LINUX, LLVM, QEMU



Le reverse engineering est une discipline qui souffre d'un manque réel d'outils open source faisant référence, et fédérant une communauté suffisante pour lui assurer un avenir pérenne. Dans cet article, nous allons parler de trois projets initiés durant les cinq dernières années, et qui ont l'avantage d'avoir réussi à créer une réelle dynamique au sein du microcosme de l'analyse de binaire.

La communauté du reverse engineering open source souffre réellement de l'absence d'un roi, d'un ou plusieurs projets faisant référence. **Radare 2** commence à créer un réel engouement auprès des puristes, après des années d'instabilité chronique. La stabilité de Radare 2 est venue de l'appui d'un moteur de désassemblage bien plus stable : **Capstone**. Il peut s'avérer surprenant qu'en 2014, il n'existait pas de projet de moteur de désassemblage, constat partagé par Nguyen Anh Quynh, l'initiateur du projet capstone. Mais le sujet peut s'avérer bien plus complexe qu'il n'y paraît. Par la suite, ce dernier a voulu compléter son *framework* par deux autres outils ; un pour l'émulation, et un dernier pour l'assemblage de code. C'est de ces trois projets que cet article traite, mais nous verrons qu'il était impossible d'initier des projets aussi complets en si peu de temps.

1. LIVRE I : CAPSTONE

Capstone est le projet fondateur de cette trilogie ; c'est un *framework* de désassemblage. Un désassembleur n'est **pas** un projet facile, et nécessite de nombreuses années « d'expérience », car malgré l'aspect normé des langages assembleurs, il existe de nombreux **opcodes** non documentés dans les processeurs, ce qui complique énormément la tâche. Partant de ce constat, Nguyen Anh Quynh ne s'est pas lancé dans la réécriture complète d'un nouveau désassembleur. Il voulait faire de Capstone un projet multi-architecture, multi-plateforme et multi-langage. Réaliser un tel *framework* en moins d'un an n'était possible qu'en repartant d'un projet déjà avancé.

Et c'est là que les développeurs du projet ont été très inspirés, car ils sont repartis d'une sous-partie du projet LLVM, le *framework* de compilation, qu'ils ont par la suite adapté, en partie réécrit en langage **C**, afin de le *packager* sous forme de *framework*.

Le projet LLVM-MC (pour *Machine Code*) fût introduit dans le *framework* afin de palier des problèmes propres à la manipulation des langages assembleurs. LLVM-MC est donc un désassembleur utilisé afin de générer des fichiers **.S** ou encore des fichiers objet, dans le but d'enrichir le *backend* x86, celui en charge de la génération du code binaire pour les architectures cibles du même nom.

Cette utilitaire est disponible dans le paquet **llvm** de votre distribution préférée :

```
$ sudo apt-get install llvm
```

llvm-mc est un utilitaire permettant de faire du désassemblage en ligne de commandes :

```
$ echo "0xCD 0x21" | llvm-mc --disassemble
  .text
  int $33
```

Cet utilitaire est identique à celui proposé dans radare2 **rasm2**.

L'idée donc des initiateurs du projet capstone était de récupérer tout le travail autour du projet llvm-mc. Malheureusement, ce dernier était écrit en **C++**, et ces derniers voulaient un *framework* en **C**, très certainement pour simplifier les *bindings* depuis d'autres langages. Ils ont donc décidé de réécrire les parties exportées afin de pouvoir garder les évolutions futures sur le cœur.

Par la suite, ils ont réalisé quelques *bindings* avec des langages populaires, tels que **Python**, et ont réussi à fédérer une communauté pour des langages moins populaires. Aujourd'hui, la liste des *bindings* est impressionnante : **Common Lisp**, **Visual Basic**, **PHP**, **PowerShell**, **Haskell**, **Perl**, **Python**, **Ruby**, **C#**, **NodeJS**, **Java**, **GO**, **C++**, **OCaml**, **Lua**, **Rust**, **Delphi**, et **Free Pascal.**

La découverte de capstone au travers de Python est aisée et nous allons réaliser deux exemples à la difficulté croissante qui vont, par la même occasion nous faire découvrir encore un peu plus les structures utilisées à bas niveau.

1.1 Résolution des symboles de fonctions

Dans cet exemple, nous allons réaliser le désassemblage du segment **.text**, contenant le code exécutable, le tout entièrement en Python. Au travers de cet exemple, nous allons démontrer la simplicité de mise en œuvre du *framework*.

Pour cela, nous allons commencer par installer le capstone via **pip** :

```
$ pip install capstone
```

Ce dernier va compiler les sources en plus de l'installation du *binding* Python associé. Pour la suite de notre exemple, nous allons aussi installer un paquet qui va nous permettre de lire les formats exécutables sous Linux, le format ELF. **Pyelftools** est un *framework* très complet et simple d'utilisation :

```
$ pip install pyelftools
```

Le but de notre script va être de réaliser le désassemblage de n'importe quelle section d'un fichier binaire, auquel nous allons ajouter la résolution des symboles des fonctions si ces derniers sont présents dans le fichier.

```
from capstone import *
from capstone.x86 import *
from elftools.elf.elffile import ELFFile


class InvalidSectionName(Exception):
    def __init__(self, message):
        super().__init__("section <%s> is not
present"%message)


def section_dump(file, section_name):
    # lecture du fichier ELF
    elf = ELFFile(file)

    # construction de la table des symboles
    symbol_section = elf.get_section_by_name(".dynsym")
    symbol_table = {}
    for symbol in symbol_section.iter_symbols():
        symbol_table[symbol["st_value"]] = symbol.name

    # récupération de la section contenant le code
exécutable
    text_section = elf.get_section_by_name(section_name)
    if text_section is None:
        raise InvalidSectionName(section_name)

    # offset de la section dans le fichier source
    offset = text_section["sh_offset"]

    # taille de la section
    size = text_section["sh_size"]

    # adresse où la section sera chargée
    address = text_section["sh_addr"]

    # dump de la section
    file.seek(offset)
    text_section_raw = file.read(size)
```

```
    #initialisation du désassembleur
    md = Cs(CS_ARCH_X86, CS_MODE_64)

    # on dump la section
    for inst in md.disasm(text_section_raw, address):
        if inst.id == X86_INS_CALL:
          try:
              # résolution des symboles de fonction
              func_address = int(inst.op_str, 16)
              print("0x%x:\t%s\t%s" % (inst.address,
inst.mnemonic, symbol_table[func_address]))
              continue
          except:
              pass

      print("0x%x:\t%s\t%s" % (inst.address, inst.
mnemonic, inst.op_str))
```

Au tout début de la fonction **section_dump**, nous tentons de lire le fichier binaire passé en paramètre.

Ensuite, nous récupérons la table des symboles se situant, sur les systèmes Linux, dans la section **.dynsym** ou bien **.symtab** pour les symboles non utiles. Nous créons une table faisant correspondre la valeur du symbole (l'adresse) à son nom, car dans le cadre de symboles de type fonction le nom représente bien le nom de la fonction.

Ensuite, nous allons extraire la section cible, son offset dans le fichier (adresse de départ), sa taille, et son adresse théorique de chargement, bien sûr si ce segment n'est pas soumis au PIE (*Position Independent Executable*). Nous récupérons la section au sein de la variable **text_section_raw**.

Enfin, nous initialisons notre désassembleur capstone auquel nous fournissons notre architecture (que nous aurions pu récupérer du fichier ELF) et notre section au format binaire. Enfin, nous passons au désassemblage à proprement parler. À chaque instruction désassemblée, nous tentons de réaliser une résolution des symboles dans le cas d'une instruction de type **call**, en vérifiant que l'adresse cible est présente ou non dans la table précédemment chargée.

Si nous testons notre script sur **/usr/bin/python** sur la section **.text**, nous obtenons la sortie suivante :

```
$ python sectiondump.py --file-path /usr/bin/python
--section-name .text
...
0x41788b:       lea     rcx, qword ptr [rsp + 0x20]
0x417890:       xor     edx, edx
0x417892:       mov     rsi, r15
0x417895:       mov     rdi, rax
0x417898:       call    PyRun_SimpleFileExFlags
0x41789d:       call    PyErr_Clear
0x4178a2:       mov     rdi, r12
0x4178a5:       call    fclose
...
```

Très peu de lignes de Python pour arriver à un résultat déjà avancé.

1.2 Détermination du graphe d'appel d'une fonction

Nous pouvons appliquer des algorithmes déterminant les blocs basiques du programme, afin d'en calculer le graphe d'appel d'une fonction.

Pour cela rien de plus simple, il suffit d'instrumenter un peu plus capstone et d'utiliser la puissance des structures de données de Python. Nous allons donc réaliser un programme simple avec une boucle et une conditionnelle :

```
#include <stdio.h>

int sum_pair(void) {
 int sum = 0;
 for(int i = 0; i < 10; i++) {
  if (i % 2 == 0) {
   sum += i;
  }
 }
 return sum;
}

int main(void) {
 printf("sum = %d\n", sum_pair());
}
```

Puis le compiler :

```
$ gcc -o sum sum.c
```

Dans notre script Python, nous allons réaliser le désassemblage de la section **.text**, comme dans l'exemple précédent, et nous allons récupérer l'adresse de notre fonction via la table des symboles comme précédemment, mais cette fois nous allons inverser notre structure pour que la résolution se fasse du nom vers les adresses.

```
def graph_dump(file, section_name):
  # lecture du fichier ELF
  elf = ELFFile(file)

  # récupération de la section contenant
le code exécutable
  section = elf.get_section_by_name(".
text")

  # construction de la table des symboles
  symbol_section = elf.get_section_by_
name(".symtab")
  symbol_table = {}
```

```
  for symbol in symbol_section.iter_
symbols():
      symbol_table[symbol.name] =
symbol["st_value"]

  # offset de la section dans le fichier
source
  offset = section["sh_offset"]

  # taille de la section
  size = section["sh_size"]

  # adresse où la section sera chargée
  address = section["sh_addr"]

  # dump de la section
  file.seek(offset)
  section_raw = file.read(size)

  engine = Engine(section_raw, address)

  graph = engine.bloc_fill(symbol_
table["sum_pair"])
```

Ensuite, nous réalisons une classe **Engine** :

```
class Bloc:
    def __init__(self):
        self.inst = []
        self.sons = {}

def parse_address(inst):
    try:
        return int(inst.op_str, 16)
    except:
        return None

class Engine:
    def __init__(self, section_raw, section_
address):
        self.md = Cs(CS_ARCH_X86, CS_MODE_64)
        self.section_raw = section_raw
        self.section_address = section_address
        self.all_address_bloc = {}

    def disasm(self, address):
        offset = address - self.section_address
        if offset < 0 or offset > len(self.section_
raw):
            return []

        return self.md.disasm(self.section_
raw[offset:], address)

    def bloc_fill(self, address):
        # permet de détecter les boucles
        if address in self.all_address_bloc:
            return self.all_address_bloc[address]
```

```
            current_bloc = Bloc()
            self.all_address_bloc[address] = current_bloc

        for inst in self.disasm(address):
            if inst.id in [X86_INS_CALL, X86_INS_JMP]:
                target = parse_address(inst)
                if target:
                    current_bloc.sons[target] = self.bloc_fill(target)
                else:
                    current_bloc.inst.append(inst)
            elif inst.id in [X86_INS_JLE, X86_INS_JBE, X86_INS_JNE, X86_INS_JE]:
                # deux blocs à ajouter car c'est une jonction
                target = parse_address(inst)
                current_bloc.sons[target] = self.bloc_fill(target)
                current_bloc.sons[inst.address + inst.size] = self.bloc_
fill(inst.address + inst.size)
            elif inst.id == X86_INS_RET:
                # fin de la récursion
                break
            else:
                # on remplit le bloc courant
                current_bloc.inst.append(inst)
        return current_bloc
```

Nous allons décrire un peu le fonctionnement de la méthode **fill_bloc** : cette dernière va tenter de déterminer les blocs basiques de la fonction et traite trois cas différents :

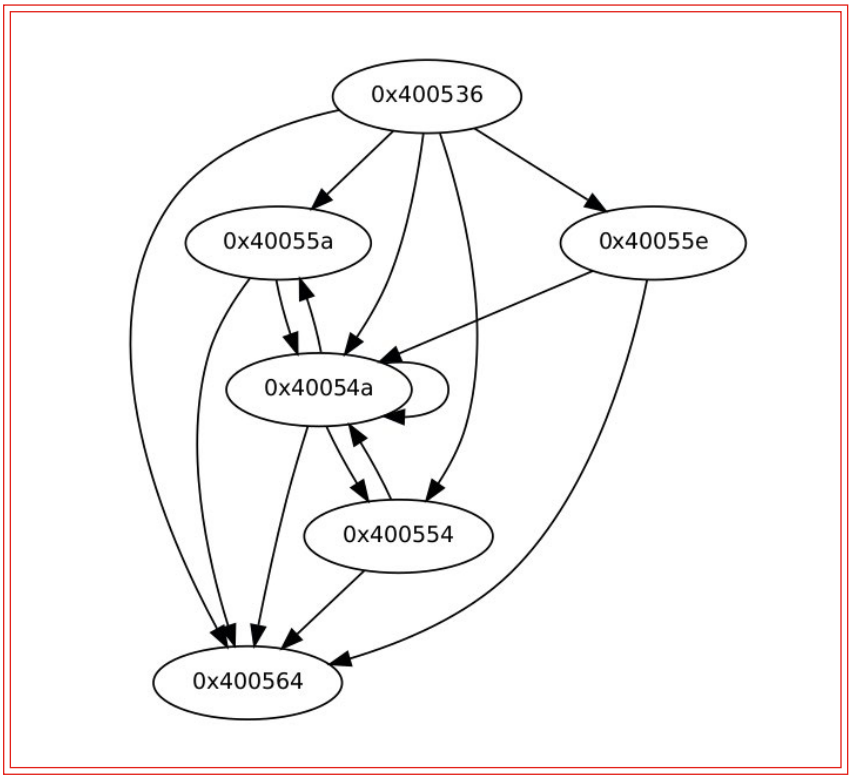


Fig. 1 : Graphe d'appel de la fonction sum_pair.

- le cas de l'appel à une fonction qui va créer un nouveau bloc. Il faudra juste se prémunir des fonctions externes, non encore résolues et qui ne se trouvent pas dans ce segment de code ;
- le cas du *jump* où l'on va créer aussi un nouveau bloc comme le *call* en tentant de résoudre l'adresse. Cette dernière peut être dynamique, se trouvant dans un registre par exemple, donc on tente de la *parser* via une fonction ;
- les cas des jumps conditionnels, qui eux vont créer deux nouveaux sous blocs : le cas juste et le cas faux ;
- le cas de l'instruction **ret** qui va déterminer que le bloc est une feuille ;
- et pour toutes les autres instructions, remplir le bloc courant.

Ensuite, nous avons réalisé une fonction permettant de créer le fichier au format DOT suivant la partie de l'objet **graph** :

```
digraph call {
"0x400536" -> "0x400564";
"0x400536" -> "0x40055a";
"0x40055a" -> "0x40054a";
"0x40054a" -> "0x400564";
"0x40054a" -> "0x40055a";
"0x40054a" -> "0x400554";
"0x400554" -> "0x40054a";
"0x400554" -> "0x400564";
"0x40054a" -> "0x40054a";
"0x40055a" -> "0x400564";
"0x400536" -> "0x400554";
"0x400536" -> "0x40054a";
"0x400536" -> "0x40055e";
"0x40055e" -> "0x40054a";
"0x40055e" -> "0x400564";
}
```

Nous pouvons visualiser ce dernier via **xdot** en figure 1.

Ce graphe n'est pas optimisé, mais nous pouvons déjà constater la présence des feuilles, ainsi que la boucle et la condition.

Avec très peu d'outils et de code, on peut déjà réaliser des outils avancés dans l'analyse de code source.

2. LIVRE II : UNICORN

Avec **Unicorn**, nous allons sortir du désassemblage répondant aux problématiques d'analyses statiques, afin de nous plonger dans l'analyse dynamique.

Unicorn repart sur les mêmes bases et problématiques de capstone avec les mêmes objectifs. Mais malheureusement LLVM ne propose pas d'outils d'émulation.

Les fondateurs du projet, qui ne sont autres que ceux du projet capstone, ont réalisé le même constat concernant les *frameworks* d'émulation. Ils se sont donc tournés vers **qemu** afin de repartir d'une base avancée dans l'émulation de CPU.

Le projet qemu possède de nombreuses architectures déjà implémentées, et l'immense avantage d'être écrit en langage C, ce qui simplifie énormément l'objectif premier de pouvoir proposer des *bindings* dans de nombreux langages de très haut niveau.

Maintenant que nous sommes habitués à capstone, nous allons voir que unicorn est très semblable et tout aussi simple d'utilisation.

Malheureusement, unicorn n'est pas encore disponible pour Python 3, au contraire de capstone ; donc tous les exemples qui vont suivre seront réalisés en Python 2. Nous allons installer unicorn via pip :

```
$ pip install unicorn
```

Durant l'installation, vous constaterez les nombreuses sources de qemu avec leurs architectures associées.

Afin d'illustrer nos propos, nous allons générer un MBR (*Master Boot Record*) qui va afficher un message sur la sortie standard. Un MBR s'exécute dans le contexte du BIOS.

```
bits 16    ; mode 16 bit
org 0x7C00   ; origine du programme
 jmp boot
print:
 push bp
 mov bp, sp
 mov si, [bp + 4]; place le premier argument de la
fonction dans le registre si
.loop:
 lodsb
 cmp al, 0 ; verifie si la chaine est finie par un 0
 je .end
 mov ah, 0eh
 mov bx, 0
 int 10h
 jmp .loop ; continue
.end:
 mov sp, bp
 pop bp
 ret
linux_magazine: db 'GNU Linux Magazine France', 0
boot:
 sti
 push linux_magazine
 call print
 add sp, 2
times 510 - ($ - $$) db 0  ; octets de bourrage
dw 0xAA55    ; mbr magic number
```

Ce dernier est vraiment simple, car il fait appel au bios via l'interruption **0x10** afin d'utiliser les fonctionnalités du mode console de ce dernier. Et il va afficher sur cette dernière, le texte **GNU Linux Magazine France**. Un MBR doit faire une taille de 512 octets, dont les deux derniers doivent être **0xAA55** qui représentent un *magic*. Nous notons aussi que le MBR s'exécute en mode 16 bits et qu'il est chargé à l'adresse **0x7C00**. Le code précédent comporte des instructions compréhensibles uniquement par le logiciel **nasm**, tel que la partie bourrage, et qui ne viennent pas du standard **asm**.

Nous allons ensuite générer le code binaire associé via nasm :

```
$nasm -f bin mbr.asm -o mbr.img
```

Nous allons maintenant générer un script qui va réaliser une émulation du bios pour ce type de programme, via unicorn en Python :

```
# adresse de démarrage dans le cadre
d'un MBR
ADDRESS = 0x7000
OFFSET = ADDRESS + 0xc00

def hook_int(uc, int_num, user_data):
    # on récupère l'interruption
    if int_num == 0x10:
        # ah contient la commande
        ah = uc.reg_read(x86_const.
UC_X86_REG_AH)
        # al contient la donnée
        al = uc.reg_read(x86_const.
UC_X86_REG_AL)
        # on demande d'afficher un
caractère
        if ah == 0xe:
            sys.stdout.write(chr(al))
            sys.stdout.flush()

def emulate(file):
    code = file.read()
    # initialise l'émulateur en x86
16 bits
    mu = Uc(UC_ARCH_X86, UC_MODE_16)

    # on alloue 2mb de mémoire
    mu.mem_map(ADDRESS, 0x10000)
```

```
    # on écrit dans la mémoire
    mu.mem_write(OFFSET, code)

    # on installe un hook sur les interruptions pour simuler le BIOS
    mu.hook_add(UC_HOOK_INTR, hook_int)

    try:
        # démarrage de l'émulation
        mu.emu_start(OFFSET, OFFSET + len(code))
    except UcError as e:
        print("ERROR: %s" % e)

    print(">>> Fin de l'emulation")
```

Dans la fonction **emulate**, nous initialisons notre émulation. Tout d'abord, nous lisons le fichier **mbr.img** en son entier.

Ensuite, nous initialisons le mode d'émulation en x86 en mode 16 bits, car le mode réel dans lequel s'exécutent les MBR est en 16 bits ; ce mode est historique en x86.

On alloue la zone qui va servir de mémoire pour la simulation. Le MBR doit être chargé à l'adresse **0x7C00**, mais pour des raisons d'alignement, nous devons initialiser notre mémoire à **0x7000**.

Nous installons un *hook* sur les interruptions afin de simuler le bios, surtout son mode terminal. Dans la fonction **hook_int**, nous ne traitons que les interruptions de type **0x10** (terminal) et la commande **0xe** (affichage d'un caractère). Pour les autres, je ne saurais que vous conseiller de vous référer à la documentation du BIOS **[1]** et **[2]**. Et c'est bien là le réel atout de unicorn, pouvant donc instrumenter l'émulation. Il est tout à fait possible de mettre des *hooks* à différents niveaux :

- les interruptions comme nous venons de le voir ;
- des instructions ;
- du code source ;
- ainsi que sur l'accès en lecture ou écriture à la mémoire.

Si nous lançons notre script, nous obtenons la sortie suivante :

```
> bios.py --file-path /tmp/mbr.img
GNU Linux Magazine France
ERROR: Invalid memory write (UC_ERR_WRITE_UNMAPPED)
>>> Fin de l'emulation
```

Notre émulateur n'est pas complet, mais sa réalisation est aisée via unicorn.

3. LIVRE III : KEYSTONE

Keystone est le projet pendant de capstone, car il réalise l'opération inverse de ce dernier, l'assemblage. Repartant du même constat que son homologue, et reprenant exactement les mêmes techniques. Keystone est le dernier projet de la trilogie qui a vu le jour.

Pour l'installer, il suffit d'utiliser pip comme les autres :

```
$ pip install keystone-engine
```

Il existe de nombreux projets tels que nasm réalisant les mêmes opérations, et que nous avons déjà pu voir dans le cadre de la section précédente. Nous allons illustrer keystone via un exemple simple :

```
ks = Ks(KS_ARCH_X86, KS_MODE_16)
encoding, count = ks.asm(b"INC ecx; DEC edx")
```

À l'instar de capstone et unicorn, keystone mise sur la simplicité de mise en œuvre. Il serait tout à fait possible de combiner des bibliothèques telles que Pyelftools et **grako** (un *parser* de grammaire EBNF) pour réaliser un compilateur complet, mais ceci constituerait un article à part entière.

CONCLUSION

Nous avons vu trois outils, simples, complets et stables permettant de laisser libre cours aux esprits les plus créatifs de s'amuser avec des concepts fondamentaux de la programmation, voire de s'appuyer dessus afin de proposer des outils avancés et stables pour la communauté. En effet, nous constatons ces derniers temps un ralentissement dans l'innovation des outils que peuvent proposer les chercheurs en sécurité. On peut expliquer cela par le manque de projets open source faisant une réelle différence, et permettant de construire des outils toujours plus performants ; comme Linux le fut pour les systèmes d'exploitation, GCC pour les compilateurs et les logiciels en règle générale, et comme LLVM commence à l'être pour tous ceux qui s'intéressent aux structures d'un programme binaire. ■

RÉFÉRENCES

[1] Bios Interrupt Call sur Wikipédia : https://en.wikipedia.org/wiki/BIOS_interrupt_call

[2] Documentation BIOS : ftp://ftp.embeddedarm.com/old/saved-downloads-manuals/EBIOS-UM.PDF

SERVEURS DÉDIÉS Synology®

AVEC ikoula HÉBERGEUR CLOUD

Votre serveur dédié de stockage (NAS)
hébergé dans nos Data Centers français.

POUR LES LECTEURS DE
LINUX MAG*

OFFRE SPÉCIALE -60 %
À PARTIR DE
5,99 € HT/MOIS
~~14,99 €~~
CODE PROMO
SYLIM17

- Bande passante **100 Mbit/s**
- Station de **surveillance**
- Support technique **en 24/7**
- Trafic réseau **illimité**
- Système d'exploitation **DSM 6.0**
- Hébergement dans **nos Data Centers**

*Offre spéciale -60 % valable sur la première période de souscription avec un engagement de 1 ou 3 mois. Offre valable jusqu'au 31 décembre 2017 23h59 pour une seule personne physique ou morale, et non cumulable avec d'autres remises. Prix TTC 7,19 €. Par défaut les prix TTC affichés incluent la TVA française en vigueur.

CHOISISSEZ VOTRE NAS
https://express.ikoula.com/promosyno-lim

ikoula HÉBERGEUR CLOUD — /ikoula — @ikoula — sales@ikoula.com — 01 84 01 02 50

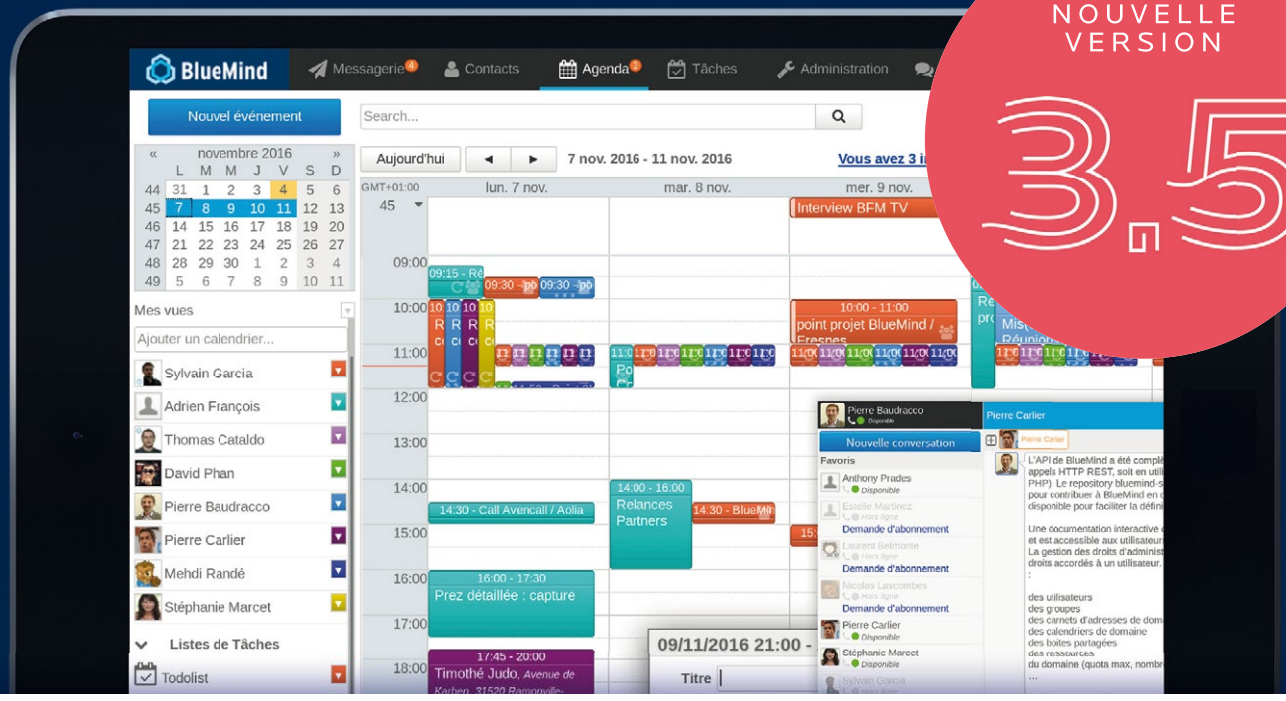
NOM DE DOMAINE | HÉBERGEMENT WEB | SERVEUR VPS | SERVEUR DÉDIÉ | CLOUD PUBLIC | MESSAGERIE | STOCKAGE | CERTIFICATS SSL

BlueMind

SOLUTION OPENSOURCE
PROFESSIONNELLE DE MESSAGERIE
COLLABORATIVE

LIBÉREZ VOTRE MESSAGERIE

EMAILS
AGENDAS
CONTACTS
TÂCHES
CHAT
MOBILES
DOCUMENTS & PIÈCES JOINTES
ENVOI FICHIERS VOLUMINEUX
...
...



NOUVELLE VERSION
3.5

FRANCAIS / NOMBREUSES RÉFÉRENCES / ERGONOMIQUE / ÉVOLUTIF / ÉCONOMIQUE

Découvrez l'écosystème BlueMind et toutes les fonctionnalités sur
www.bluemind.net