GNU LINUX MAGAZINE / FRANCE

DÉVELOPPEMENT SUR SYSTÈMES UNIX, OPEN SOURCE & EMBARQUÉ

N°203
AVRIL 2017

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

L 19275 - 203 - F: 7,90 € - RD

Python / OpenCV

Mettez en place un système de RECONNAISSANCE FACIALE ! p.16

- Installez OpenCV
- Comprenez et réglez les paramètres
- Détectez et découpez les visages présents sur une image
- Construisez votre base de visages
- Identifiez automatiquement des personnes sur des photos

Embarqué / Bas niveau

INITIEZ-VOUS À LA PROGRAMMATION BARE-METAL SUR RASPBERRY PI p.48

Sysadmin / Multi-distributions

DÉCOUVREZ LA GESTION DE PAQUETS FONCTIONNELLE AVEC Nix ET NixOS p.32

Bash / Script

REPRENEZ LE CONTRÔLE DE VOS SAUVEGARDES AVEC LE SHELL ET RSYNC p.70

Sécurité / Assembleur

COMPRENEZ LA TECHNIQUE DU RETURN ORIENTED PROGRAMMING p.94

Python / Pyo

ACCÉLÉREZ VOS TRAITEMENTS AUDIOS AVEC LA PROGRAMMATION « MULTITHREADÉE » p.06

DÉPLOIEMENT CONTINU AVEC OPENSHIFT - REMOTESTORAGE.JS : STOCKAGE DISTANT...

* WARGAME * CONFERENCES * CHALLENGES * WORKSHOPS *

NUIT DU HACK

XV

24->25 Juin 2017

New York Hôtel Convention Center

Disneyland Paris

www.nuitduhack.com

@hackerzvoice

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 €

ÉDITO

Il est temps de penser au grand nettoyage de printemps !

Comment ? Vous n'avez encore rien fait ? Alors il est possible que votre machine s'en charge pour vous ! C'est la mésaventure qui m'est arrivée lors de la réalisation de ce numéro, du hors-série sur la programmation Shell et du hors-série de *Hackable* sur l'apprentissage de la programmation Python sur Raspberry Pi... lorsque le destin veut se rappeler à votre bon souvenir, il choisit généralement toujours le meilleur moment (une variante de la loi de Murphy sans doute). Or donc, une veille de week-end, avant d'éteindre la machine, j'effectue de manière tout à fait fortuite une sauvegarde des derniers travaux en cours et c'est là, juste à la fin de cette sauvegarde, après avoir effacé certains fichiers de la corbeille, que le destin s'est dit « Tiens, allons nous amuser un peu... » : tous les fichiers de **`/home`** ont disparu, tous ! Sans doute une erreur mal placée et irrécupérable sur le disque, mais le résultat est là : la perte de tous les fichiers importants de l'ordinateur. Dans ces cas-là, on ne s'affole pas, il reste :

1. l'armada **`fsck`**, **`testdisk`** et autres ;
2. les sauvegardes automatiques dans le *cloud*.

Oui... sauf que là, après une nuit passée à tester différents outils, il n'était possible de récupérer que des fichiers sans intérêt. Du coup, je me suis tourné vers la sauvegarde dans le *cloud*, un service auquel j'avais souscrit il y a une dizaine d'années et qui moyennant une cinquantaine d'euros par an me proposait de chiffrer et sauvegarder mes fichiers dans trois *datacenters* distincts. C'était la solution idéale, car après l'avoir testée, elle était très simple à installer, à utiliser et surtout non chronophage : tout était déjà prêt à l'emploi. Sauf que depuis 5 ans elle n'enregistrait plus rien... sans un message d'erreur ou un mail rapportant une anomalie (les clients de cette société aiment sans doute payer un service pour ne pas l'utiliser...).

J'ai perdu beaucoup de choses, et même sans doute plus que ce que je pense, car on se rend compte qu'au fur et à mesure des fichiers manquants. Après la période de frustration intense, on relativise : j'avais écrit ces documents et programmes, je peux donc le refaire. Certains articles sont définitivement perdus, mais d'autres prendront une nouvelle direction que je n'avais pas envisagée au moment de leur rédaction. Et puis, surtout, je vais pouvoir écrire tous ces petits outils qui facilitent la vie de tous les jours et pour lesquels on se dit toujours « c'est une bonne idée, mais je le ferai plus tard, là je n'ai pas le temps ». Pour commencer, j'ai donc écrit mon script de sauvegarde avec un suivi constant des opérations. Les autres scripts suivront au fur et à mesure et permettront de repartir sur des bases saines sans pâtir d'une architecture vieille de plus de dix ans et de scripts-rustines écrits à la hâte. Et puis cela m'a déjà permis d'écrire un article sur **`rsync`** que vous trouverez dans ce numéro... il faut bien trouver des points positifs à cette histoire :-)

Enfin, s'il fallait une ultime preuve que tout cela n'a pas été cataclysmique, tous les magazines ont pu sortir... même s'il faut reconnaître que la petite sauvegarde imprévue du vendredi soir a bien aidé :-) Votre *GNU/Linux Magazine* est bien là ! Je vous en souhaite une bonne lecture !

Tristan Colombo

CONNECT ÉVOLUE !

LISEZ CE NUMÉRO ET PLUS DE 150 AUTRES EN LIGNE !

ACTUELLEMENT SUR CONNECT :

- **CE NUMÉRO**
- **et + de 150 autres numéros de GNU/Linux Magazine**

+

- **72 numéros Hors-Séries de GNU/Linux Magazine**

TOUT CELA À PARTIR DE 199 € TTC*/AN !

* Tarif France Métropolitaine

OFFRE DÉCOUVERTE CONNECT

1 MOIS GRATUIT, RÉSERVÉE AUX PROFESSIONNELS

Appelez le 03 67 10 00 28 et donnez le code « GLMF203 » pour découvrir Connect gratuitement pendant 1 mois !

Pour tous renseignements complémentaires, contactez-nous via notre site internet : **www.ed-diamond.com**, par téléphone : **03 67 10 00 28** ou envoyez-nous un mail à **connect@ed-diamond.com** !

SOMMAIRE

GNU/LINUX MAGAZINE FRANCE
N°203

LA PROGRAMMATION AUDIO MULTICŒUR AVEC PYTHON

OLIVIER BÉLANGER
[Développeur de logiciels audios]

MOTS-CLÉS : PROGRAMMATION AUDIO, PYTHON, MULTICŒUR, MÉMOIRE PARTAGÉE, PYO, MULTIPROCESSING

Dans le numéro 157 de GNU/Linux Magazine, j'ai présenté « pyo », un module offrant une multitude d'outils permettant de créer des chaînes de traitements audios de façon native avec le langage de programmation Python. Cet article présente les derniers développements de la librairie ainsi que différentes stratégies permettant d'écrire des programmes audios utilisant à pleine capacité la puissance des ordinateurs multicœurs, qui sont devenus la norme de nos jours.

Le code présenté dans cet article est disponible sur le compte GitHub de **pyo** ainsi que dans la documentation en ligne :

- https://github.com/belangeo/pyo
- http://ajaxsoundstudio.com/pyodoc/examples/18-multicore/index.html

NOTE

Cet article portant sur la programmation audio avec le module Python pyo [1], une connaissance de base du langage et de la librairie sont assumés de la part du lecteur. Les exemples [2] disponibles dans la documentation en ligne constituent un bon point de départ!

Dans les années 90, l'architecture monocœur constitue la norme de fabrication des ordinateurs personnels et l'augmentation de la puissance de calcul repose sur une élévation de la fréquence du processeur. Cette fréquence a atteint une limite, entre 3 à 4 GHz, où la surchauffe du processeur était telle que le refroidissement à air ne fût plus suffisant

(ou trop bruyant) et le refroidissement par eau trop complexe et coûteux. La solution pour répondre à la demande toujours croissante en ce qui concerne la puissance de calcul de l'ordinateur fut de se tourner vers les processeurs multicœurs **[3]**, où plusieurs unités de calcul indépendantes résident sur une même puce. Ceci permet d'effectuer plusieurs tâches de façon simultanée, donc d'augmenter l'efficacité de travail, tout en gardant la fréquence de chacun des processeurs à un niveau raisonnable. Depuis le début des années 2000, le nombre de cœurs présents dans nos processeurs ne cesse d'augmenter et il est devenu impératif pour les développeurs de logiciels d'élaborer des algorithmes permettant d'utiliser à leur plein potentiel ces architectures multicœurs. Les programmeurs utilisant le langage Python rencontrent en ce domaine un obstacle de taille en la présence d'un mécanisme appelé le *Global Interpreter Lock* (GIL).

1. GIL, FONCTIONNALITÉ OU LIMITATION

Le GIL, présent dans la plupart des variantes de l'interpréteur Python, et assurément dans l'implémentation de référence, **Cpython**, est depuis longtemps une source de désaccord dans la communauté des développeurs Python. Le rôle du GIL consiste à assurer que les instructions d'un programme Python soient toujours exécutées en série. Même un programme multitâche, mettant en place des calculs parallèles à l'aide du module **threading**, effectuera les opérations des différents fils à tour de rôle, et ce, même si ce module utilise en interne les *threads* natifs du système d'exploitation. Les détracteurs du GIL lui reprochent d'être un obstacle à la création de réels programmes multitâches (exécution parallèle utilisant plusieurs cœurs) en Python. Les partisans du GIL, quant à eux, mettent en l'avant la très grande stabilité du langage comme principale raison de la présence du GIL. Il ne s'agit pas ici de faire le procès ou la propagande du GIL, mais d'en comprendre l'impact sur la performance des programmes. Il ne faut pas oublier non plus que la première version officielle de Python date de 1991, époque où les processeurs étaient encore principalement monocœurs.

Le module **threading**, bien que soumis à la loi du GIL, s'avère très utile, voire indispensable dans plusieurs situations. Prenons en exemple un programme qui ouvre un port série et observe continuellement si ce dernier a reçu des données à traiter :

```
import serial

serv = serial.Serial('/dev/ttyUSB0', 9600)

while True:
    if serv.inWaiting() > 0:
        data = serv.readline()
        # do something with the data...
```

Le problème principal de ce programme est qu'une fois entré dans la boucle **while**, en attente d'une action à poser, le programme ne peut plus effectuer d'autres opérations (mise à jour d'une interface graphique, initialisation d'une base de données, création d'une chaîne de traitement audio, etc.). Afin de garder le contrôle sur la boucle d'exécution du programme, il serait préférable de placer la boucle d'écoute du port série dans un fil parallèle :

```
import threading
import serial

class SerialServer(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self.daemon = True
        self.serv = serial.Serial('/dev/ttyUSB0', 9600)

    def run(self):
        while True:
            if self.serv.inWaiting() > 0:
                data = self.serv.readline()
                # do something with the data...

server = SerialServer()
server.start()
```

Cette variante permet de lancer une boucle infinie pour la détection de données en entrée du port série sans pour autant bloquer le programme principal, qui a fort probablement d'autres commandes à exécuter.

Une erreur courante consiste à croire qu'un programme multitâche est forcément plus performant que sa contrepartie à fil unique. La présence du GIL dans le langage Python aura généralement pour effet de provoquer un concours d'acquisition et de relâche du GIL par les différentes tâches, concours dont le résultat sera bien souvent une charge supplémentaire en temps et en CPU. Un programme Python monotâche, s'il n'attend pas après une communication réseau ou un accès au disque dur, sera presque toujours plus rapide que sa version multitâche puisque l'interpréteur est optimisé pour qu'il n'ait aucune nécessité d'acquérir ou de relâcher le GIL, il peut donc fonctionner à son plein rendement. Pour une description détaillée des effets du GIL sur les programmes multitâches, consultez les présentations de David Beazley sur le sujet **[4, 5]**.

Si les tâches parallèles du module **threading** sont soumises à la loi du GIL, est-il alors possible d'écrire un programme Python exploitant à son plein potentiel les multiples cœurs qui sont aujourd'hui systématiquement présents dans nos ordinateurs personnels? La réponse est « oui », grâce à l'apparition, depuis la version 2.6, du module **multiprocessing**.

2. DU MULTITÂCHE AU MULTIPROCESSUS

Le module **multiprocessing** **[6]** partage une API (*Application Programming Interface*) similaire au module **threading**, mais plutôt que de lancer des tâches parallèles, il utilise des sous-processus (indépendants du processus principal), ce qui permet au programmeur de contourner les limitations imposées par la présence du GIL. L'objet central du module, **Process**, possède un équivalent pour chacune des méthodes de l'objet **Thread**. Il est donc très aisé de remplacer les « tâches » d'un programme Python par des « sous-processus ». Il suffit de remplacer **threading.Thread** par **multiprocessing.Process**, comme dans l'exemple suivant :

```
import multiprocessing
import serial

class SerialServer(multiprocessing.Process):
    def __init__(self):
        multiprocessing.Process.__init__(self)
        self.daemon = True
        self.serv = serial.Serial('/dev/ttyUSB0',
9600)

    def run(self):
        while True:
            if self.serv.inWaiting() > 0:
                data = self.serv.readline()
                # do something with the data...

server = SerialServer()
server.start()
```

Dû au fait que les différentes sections du code fonctionnent à l'intérieur de processus indépendants, la communication s'en trouve quelque peu complexifiée. Une variable globale du programme ne sera pas accessible au sous-processus comme elle peut l'être pour une tâche. Par exemple :

```
import threading

value = 0

class SerialServer(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
    def run(self):
        global value
        value = 1

server = SerialServer()
server.start()
print(value)
```

Ce programme affichera la valeur « 1 » en sortie de la fonction **print**. Par contre :

```
import multiprocessing

value = 0

class SerialServer(multiprocessing.Process):
    def __init__(self):
        multiprocessing.Process.__init__(self)
    def run(self):
        global value
        value = 1

server = SerialServer()
server.start()
print(value)
```

Celui-ci affichera la valeur « 0 » puisque la ligne qui modifie la valeur de la variable *value* est exécutée dans un processus indépendant.

Le module offre différentes méthodes pour mettre en place une communication interprocessus. Les objets **Queue**, **Pipe**, **Value** ou **Array** permettent tous de partager des données de façon sécurisée entre les processus actifs. Dans les exemples concrets illustrés plus bas, nous utiliserons l'objet **Pipe**, qui permet une communication dans les deux sens, pour contrôler nos sous-processus à partir du programme principal.

Le module **multiprocessing** offre plusieurs autres fonctionnalités qui dépassent le cadre de cet article et dont nous ne nous servirons pas dans la mise en place de nos programmes audios multicœurs. Le lecteur est invité à lire la documentation du module pour plus de détails.

Un problème auquel nous ferons face - dont le module ne fournit pas les outils nécessaires pour y remédier - est la communication audio interprocessus. Comment faire pour échanger des signaux audios entre deux processus indépendants ? Un signal audio stéréo consiste en deux *arrays* de nombres à virgule flottante contenant 44100 échantillons par seconde (fréquence d'échantillonnage par défaut du serveur audio de pyo). Le transfert des échantillons d'un processus à l'autre ne peut souffrir aucun délai puisque la carte de son (qui transfère des blocs d'échantillons du pilote audio aux sorties physiques) n'attendra pas. Si les échantillons n'arrivent pas

dans les temps, il y aura alors des artefacts dans le son. Aucun des procédés fournis par le module **multiprocessing** n'est assez rapide pour assurer une communication audio interprocessus fiable. La section suivante expose les fonctionnalités qui ont été ajoutées au module pyo pour pallier à ce problème et rendre l'utilisation des multiples cœurs simple et efficace.

3. ÉVOLUTION DU MODULE PYO

Pyo est un module offrant plus de 250 processeurs de signaux audionumériques permettant de créer des chaînes de calcul sonore sophistiquées. On y retrouve des outils allant des primitives telles que les opérateurs arithmétiques sur les signaux audios et la lecture de fichiers sonores, en passant par divers algorithmes de filtrage et de synthèse, jusqu'aux traitements audios les plus évolués (granulation, vocodeur de phase, traitement multibande, etc.). Pyo est aujourd'hui utilisé en composition musicale, pour le développement de traitements audios, dans la création de logiciels sonores ainsi qu'en recherche et en enseignement dans plusieurs universités de par le monde. Pour une introduction au fonctionnement de base du module, le lecteur est invité à parcourir la liste des exemples disponibles dans le manuel en ligne **[2]**.

Afin de permettre la gestion de processus audios multicœurs, certaines modifications ont dû être apportées à la bibliothèque. Cette section revient sur les principaux défis rencontrés lors de cette phase de recherche et développement ainsi que les solutions mises en place pour rendre le module compatible avec l'utilisation de plusieurs cœurs.

3.1 Modification du processus d'initialisation des connexions audios

Sous Linux, l'utilisateur de pyo a le choix entre **Portaudio [7]** ou **Jack [8]** comme pilote audio. Jack étant la norme pour la production audio professionnelle sous Linux, les exemples présents dans cet article utiliseront ce pilote.

Un processus audio multicœur doit créer et connecter un serveur audio indépendant pour chaque sous-processus utilisé. Un problème rencontré lors des premiers essais fût une latence considérable et indéterministe entre l'activation des différents sous-processus du programme. La cause était que la création de nouveaux ports de communication avec Jack ainsi que la routine d'autoconnexion des ports prennent un temps qui est variable (dépendamment des différentes opérations en cours dans le système) et assez long pour poser des problèmes de synchronisation audio. Avant la version 0.8.3 de pyo, la routine d'initialisation des connexions audios avait lieu dans la méthode **start** du serveur audio. Pour un serveur unique, cela ne pose pas de problème, mais si plusieurs serveurs doivent démarrer en même temps, il s'y passe beaucoup trop d'opérations sur lesquelles nous n'avons pas de contrôle. Voici la routine de démarrage avant la version 0.8.3 :

```
# Avant la version 0.8.3.
from pyo import *
s = Server(audio='jack')
s.boot() # Initialise le pilote audio
s.start() # Création des ports Jack,
autoconnexion et activation du serveur
```

Afin d'assurer un meilleur déterminisme temporel à l'activation du serveur, les opérations longues et non contrôlées ont été déplacées dans la méthode **boot**, ne laissant à la méthode **start** que la tâche d'activer la boucle de calcul audio (changement d'une variable booléenne de **false** à **true**).

```
# À partir de la version 0.8.3.
from pyo import *
s = Server(audio='jack')
s.boot() # Initialisation, création des ports Jack
et autoconnexion
s.start() # Activation du serveur (simple booléen)
```

3.2 Utilisation de la mémoire partagée pour la communication audio interprocessus

La communication audio interprocessus est sans aucun doute la principale nouveauté de la version 0.8.3 de pyo. Afin de partager des échantillons audios de façon simple et efficace, une nouvelle table (espace mémoire à une dimension), utilisant la mémoire partagée du système a été créée. C'est l'objet **SharedTable**. La mémoire partagée du système est un espace mémoire auquel tous les processus peuvent faire référence dans leur propre espace mémoire **[9, 10]**. Dans la situation qui nous intéresse ici, un premier processus peut créer un objet **SharedTable** et y écrire des échantillons tandis qu'un autre objet **SharedTable**, créé dans un second processus peut s'y connecter et en lire le contenu pour récupérer le signal audio. Voici la routine de création des deux objets pour partager un même espace mémoire :

```
# Création d'une table utilisant la mémoire partagée.
# Le processus qui crée la table est en mode écriture.
t = SharedTable(name="/shared", create=True, size=512)
```

```
# Second processus, connexion à la table précédemment créée.
# Ce processus est en mode lecture.
t = SharedTable(name="/shared", create=False, size=512)
```

3.3 Optimisation de l'écriture et de la lecture de table audio

Dans un contexte de signaux audios partagés entre différents processus, l'opération se résume simplement à copier les échantillons successifs dans un espace mémoire puis à les récupérer, un à un, par la suite. Les outils d'écriture et de lecture de table de pyo offrent plusieurs fonctionnalités supplémentaires (lecture et écriture avec interpolation, vitesse variable, changement de taille dynamique), nécessaires pour la grande majorité des techniques de manipulation du signal standards, mais qui cause ici une charge supplémentaire inutile dans la consommation du CPU. Deux nouveaux objets, aux fonctionnalités volontairement simplifiées, ont été ajoutés au module, **TableFill** et **TableScan**. Le premier écrit des échantillons audios de façon circulaire dans une table tandis que le second lit le contenu d'une table, en boucle, sans interpolation. Ces objets sont très simples à utiliser et leur consommation en CPU est minime. En voici un exemple :

```
# Écriture en continu dans une table
table1 = SharedTable(name="/shared", create=True, size=512)
signal = Sine(freq=440, mul=0.3)
fill = TableFill(signal, table1)

# Lecture en continu dans une table
table2 = SharedTable(name="/shared", create=False, size=512)
signal = TableScan(table2)
```

3.4 Multiple serveurs audio, désactivation des ports MIDI et simulation d'événements

Pyo permet de créer jusqu'à 64 objets **Server** simultanés sur une même machine (un objet **Server** doit absolument être présent dans tout processus qui implémente une chaîne de traitement audio). Ce qui veut dire qu'on peut créer 64 sous-processus contenant chacun leur propre séquence de processeurs sonores. Par contre, il est fortement déconseillé d'ouvrir un port MIDI vers une interface donnée dans plus d'un serveur à la fois. Le comportement standard du serveur de pyo étant justement d'ouvrir un port MIDI vers l'interface définie comme étant celle par défaut dans le système, créer plus d'un serveur audio simultané peut poser problème. Heureusement, il est maintenant possible de modifier le comportement du serveur et de lui indiquer qu'il ne doit pas tenter de se connecter à une interface MIDI. Pour cela, il suffit d'appeler sa méthode **deactivateMidi** avant d'appeler la méthode **boot**. Le programme peut toujours traiter des données MIDI, mais il faut les lui faire parvenir à l'aide de la méthode **addMidiEvent**. Cela permet d'avoir un seul processus en charge de la gestion des interfaces MIDI, qui fera suivre l'information aux autres processus, où la production sonore aura finalement lieu. La routine d'initialisation du serveur se lira donc comme suit :

```
s = Server(audio='jack')
s.deactivateMidi()
s.boot().start()
```

Pour ajouter un événement dans la pile d'événements MIDI :

```
s.addMidiEvent(status, data1, data2)
```

4. CONTEXTES D'UTILISATION AUDIO DU MULTICŒUR

Dans cette section, nous présentons quatre cas de figure typiques où l'utilisation du multicœur peut augmenter considérablement les possibilités des programmes audios.

4.1 Répartition de la charge de calcul sur plusieurs processeurs

Le premier exemple ne fait rien d'autre que de répartir la charge de calcul sur plusieurs processeurs. Chacun des processus calcule sa partie de signal, un chorus de 200 oscillateurs autour d'une fréquence donnée, sans rien connaître du travail des autres processus. Chacun envoie son signal directement au pilote audio.

```
01: import time, random, multiprocessing
02: from pyo import *
03:
04: class Proc(multiprocessing.Process):
05:     def __init__(self, pitch):
06:         super(Proc, self).__init__()
07:         self.daemon = True
08:         self.pitch = pitch
09:
10:     def run(self):
11:         self.server = Server(audio="jack")
12:         self.server.deactivateMidi()
13:         self.server.boot().start()
14:
```

Maker Faire® Paris

9 > 11 juin 2017

Cité des Sciences et de l'Industrie

Appel aux Makers

JUSQU'AU 15 AVRIL

Maker Faire est à la fois une fête de la science, une foire populaire et un événement de référence pour l'innovation. Ce concept regroupe stands de démonstration, ateliers de découverte, spectacles et conférences autour des thèmes de la créativité, de la fabrication, et des cultures Do It Yourself.
Aujourd'hui, plus de 200 éditions réunissent dans 38 pays **des communautés de passionnés, experts ou débutants, qui partagent l'envie de créer, fabriquer et apprendre les uns des autres.**
La 4ème édition de **Maker Faire Paris** se tiendra à la Cité des Sciences et de l'Industrie.

Vous souhaitez participer ?
Rejoignez la communauté des Makers en vous inscrivant à notre appel !

Clôture des inscriptions le 15 avril 2017 à minuit !

Un événement
Make: makezine.com

Présenté par
LEROY MERLIN ...et vos envies prennent Vie !

Partenaire
cité des sciences & de l'industrie

Inscription sur
paris.makerfaire.com

```
15:         # 200 oscillateurs à onde carrée limitée en
fréquence.
16:         lo, hi = midiToHz((self.pitch - 0.1, self.
pitch + 0.1))
17:         self.amp = Fader(fadein=5, mul=0.01).play()
18:         self.fr = Randi(lo, hi, [random.uniform(.2,
.4) for i in range(200)])
19:         self.sh = Randi(0.1, 0.9, [random.
uniform(.2, .4) for i in range(200)])
20:         self.osc = LFO(self.fr, sharp=self.sh,
type=2, mul=self.amp).out()
21:
22:         time.sleep(30) # Actif pour 30 secondes.
23:         self.server.stop()
24:
25: if __name__ == '__main__':
26:     # Accord de Do majeur (une note par processus).
27:     p1, p2, p3, p4 = Proc(48), Proc(52), Proc(55),
Proc(60)
28:     p1.start(); p2.start(); p3.start(); p4.start()
```

4.2 Partage de signaux audios interprocessus

Le deuxième exemple illustre le partage de signaux audios interprocessus. Le premier processus, *analysis*, sépare le signal en 50 bandes de fréquence (lignes 20 à 23) et écrit les 50 signaux dans des tables en mémoire partagée (lignes 24 et 25). Le second processus, *synthesis*, récupère les signaux par une lecture bouclée des 50 tables (lignes 27 et 28) et transpose indépendamment chaque tranche de fréquence (lignes 29 à 32) pour créer un chorus d'une grande qualité sonore.

```
01: import time, random, multiprocessing
02: from pyo import *
03:
04: class Proc(multiprocessing.Process):
05:     def __init__(self, create):
06:         super(Proc, self).__init__()
07:         self.daemon = True
08:         self.create = create
09:
10:     def run(self):
11:         self.server = Server(audio="jack")
12:         self.server.deactivateMidi()
13:         self.server.boot().start()
14:         bufsize = self.server.getBufferSize()
15:
16:         nbands = 50
17:         names = ["/f%02d" % i for i in
range(nbands)]
18:
19:         if self.create: # Séparation du
spectre en 50 bandes de fréquence.
20:             freq = [20 * 1.1487 ** i for i in
range(nbands)]
21:             amp = [pow(10, (i*-1)*0.05) * 8
for i in range(nbands)]
22:             self.input = SfPlayer(SNDS_
PATH+"/transparent.aif", loop=True)
23:             self.filts = IRWinSinc(self.
input, freq, freq, 3, 128, amp)
24:             self.table =
SharedTable(names, create=True, size=bufsize)
25:             self.recrd = TableFill(self.
filts, self.table)
26:         else: # Transposition indépendante
par bande.
27:             self.table =
SharedTable(names, create=False, size=bufsize)
28:             self.tscan = TableScan(self.
table)
29:             transpofac = [random.
uniform(0.98,1.02) for i in range(nbands)]
30:             self.pvana = PVAnal(self.
tscan, size=1024, overlaps=4)
31:             self.pvtra = PVTranspose(self.
pvana, transpo=transpofac)
32:             self.pvsyn = PVSynth(self.
pvtra).out()
33:
34:         time.sleep(30)
35:         self.server.stop()
36:
37: if __name__ == '__main__':
38:     analysis = Proc(create=True)
39:     synthesis = Proc(create=False)
40:     analysis.start()
41:     synthesis.start()
```

4.3 Répartition de la charge de calcul et synchronisation

Lorsque la charge de calcul doit être répartie sur plusieurs cœurs, mais que l'ensemble doit être musicalement synchrone, il est impératif de s'assurer que les voix démarrent bien au même moment. Deux éléments peuvent considérablement ralentir l'initialisation des processus, et ce, de manière non déterministe. D'une part, le temps de création d'un objet **Process** est variable et dépendant de l'activité courante du système. D'autre part, l'initialisation du serveur audio est aussi un procédé non déterministe du point de vue temporel. Tel que décrit à la section 3.1, tout le processus d'initialisation du serveur audio a lieu maintenant dans la méthode **boot**, ce qui ne laisse à la méthode **start** que l'action d'activer la boucle de calcul. La stratégie mise en place dans l'exemple suivant, où deux séquences rythmiques doivent arriver synchrones au processus **Main**, se déroule en deux phases. Dans un premier temps, tous les processus sont initialisés et activés (lignes 60 à 63) sauf les serveurs audios des générateurs (la classe **Proc**). L'attente d'un message dans le canal de communication (lignes 51 à 53) retarde l'activation des serveurs audios. La deuxième phase consiste à attendre que tous les éléments soient prêts puis à envoyer le signal de départ (lignes 64 et 65). Ainsi, les deux voix rythmiques seront toujours synchrones.

```
01: import time, random, multiprocessing
02: from pyo import *
03:
04: class Main(multiprocessing.Process):
05:     def __init__(self):
06:         super(Main, self).__init__()
07:         self.daemon = True
08:
09:     def run(self):
10:         self.server =
Server(audio="jack")
11:         self.server.deactivateMidi()
12:         self.server.boot().start()
13:         bufsize = self.server.
getBufferSize()
14:
15:         # Création d'un signal stéréo
avec les deux voix.
16:         self.tab1 = SharedTable(
"/audio-1", create=False, size=bufsize)
17:         self.tab2 = SharedTable(
"/audio-2", create=False, size=bufsize)
18:         self.out1 = TableScan(self.
tab1).out()
19:         self.out2 = TableScan(self.
tab2).out(1)
20:
21:         time.sleep(30)
22:         self.server.stop()
23:
24: class Proc(multiprocessing.Process):
25:     def __init__(self, voice, conn):
26:         super(Proc, self).__init__()
27:         self.voice = voice
28:         self.connection = conn
29:         self.daemon = True
30:
31:     def run(self):
32:         self.server =
Server(audio="jack")
33:         self.server.deactivateMidi()
34:         self.server.boot()
35:         bufsize = self.server.
getBufferSize()
36:
37:         name = "/audio-%d" % self.voice
38:         self.audiotable =
SharedTable(name, create=True,
size=bufsize)
39:
40:         # Génération d'une séquence
mélodico-rythmique.
41:         onsets = random.
sample([5,6,7,8,9], 2)
42:         self.tab = CosTable([(0,0),
(32,1), (512,0.5), (4096,0.5), (8191,0)])
43:         self.ryt = Euclide(time=.125,
taps=16, onsets=onsets, poly=1).play()
44:         self.mid = TrigXnoiseMidi(self.
ryt, dist=12, mrange=(60, 96))
45:         self.frq = Snap(self.mid,
choice=[0,2,3,5,7,8,10], scale=1)
46:         self.amp = TrigEnv(self.ryt,
table=self.tab, dur=self.ryt['dur'],
47:                            mul=self.ryt['amp'])
48:         self.sig = SineLoop(freq=self.frq,
feedback=0.08, mul=self.amp*0.3)
49:         self.fil = TableFill(self.sig, self.
audiotable)
50:
51:         # Attend la réception d'un signal avant
de démarrer le serveur.
52:         while not self.connection.poll():
53:             pass
54:         self.server.start()
55:
56:         time.sleep(30)
57:         self.server.stop()
58:
59: if __name__ == '__main__':
60:     signal, child = multiprocessing.Pipe()
61:     p1, p2 = Proc(1, child), Proc(2, child)
62:     main = Main()
63:     p1.start(); p2.start(); main.start()
64:     time.sleep(.05) # Attend que tous les
processus soient initialisés.
65:     signal.send(1)  # Envoie le signal de départ.
```

4.4 Communication interprocessus, contrôle d'un synthétiseur MIDI

Le dernier exemple concerne le contrôle des processus audios via des messages en provenance du programme principal. Un synthétiseur MIDI utilisant quatre cœurs, afin de générer quatre fois plus de voix de polyphonie, est illustré. Un canal de communication (objet **multiprocessing.Pipe**) est créé pour chaque sous-processus (lignes 37 à 43). Les processus audios « écoutent » continuellement l'activité à l'arrivée du canal et si des données sont présentes, une note MIDI est produite et redirigée vers le serveur audio (lignes 27 à 32). Le programme principal a pour tâche d'assigner les notes MIDI en entrée à un processus possédant au moins une voix de libre (lignes 58 à 64) et de s'assurer que les *noteoff* sont bien dirigés vers le processus ayant effectivement joué la note MIDI concernée (lignes 52 à 57). L'utilisation du multicœur nous permet ici de passer de quatre à seize voix de polyphonie sur la même machine.

```
01: import time, multiprocessing
02: from random import uniform
03: from pyo import *
04:
05: VOICES_PER_CORE = 4
06:
07: class Proc(multiprocessing.Process):
08:     def __init__(self, pipe):
09:         super(Proc, self).__init__()
10:         self.daemon = True
11:         self.pipe = pipe
12:
13:     def run(self):
15:         self.server.deactivateMidi()
```

```
16:         self.server.boot().start()
17:
18:         # Synthétiseur MIDI très coûteux.
19:         self.mid = Notein(poly=VOICES_PER_CORE,
scale=1, first=0, last=127)
20:         self.amp = MidiAdsr(self.mid['velocity'],
0.005, .1, .7, 0.5, mul=.1)
21:         self.pit = self.mid['pitch'] *
[uniform(.99, 1.01) for i in range(40)]
22:         self.rc1 = RCOsc(self.pit, sharp=0.8,
mul=self.amp).mix(1)
23:         self.rc2 = RCOsc(self.pit*0.99, sharp=0.8,
mul=self.amp).mix(1)
24:         self.mix = Mix([self.rc1, self.rc2],
voices=2)
25:         self.rev = STRev(Denorm(self.mix), [.1,
.9], 2, bal=0.30).out()
26:
27:         # Si un événement MIDI est reçu, il est
redirigé vers le serveur audio.
28:         while True:
29:             if self.pipe.poll():
30:                 data = self.pipe.recv()
31:                 self.server.addMidiEvent(*data)
32:             time.sleep(0.001)
33:
34:         self.server.stop()
35:
36: if __name__ == '__main__':
37:     # Un canal de communication par processus.
38:     main1, child1 = multiprocessing.Pipe()
39:     main2, child2 = multiprocessing.Pipe()
40:     main3, child3 = multiprocessing.Pipe()
41:     main4, child4 = multiprocessing.Pipe()
42:     mains = [main1, main2, main3, main4]
43:     p1, p2, p3, p4 = Proc(child1), Proc(child2),
Proc(child3), Proc(child4)
44:     p1.start(); p2.start(); p3.start(); p4.start()
45:
46:     # Dictionnaire pour garder en mémoire quel
processus joue quelle note MIDI.
47:     playing = {0: [], 1: [], 2: [], 3: []}
48:     currentcore = 0
49:     # Fonction appelée lors de la réception d'un
message MIDI.
50:     def callback(status, data1, data2):
51:         global currentcore
52:         if status == 0x80 or status == 0x90 and
data2 == 0: # noteoff
53:             for i in range(4):
54:                 if data1 in playing[i]:
55:                     playing[i].remove(data1)
56:                     mains[i].send([status, data1,
data2])
57:                     break
58:         elif status == 0x90:
# noteon
59:             for i in range(4):
60:                 currentcore = (currentcore + 1) % 4
61:                 if len(playing[currentcore]) <
VOICES_PER_CORE:
62:                     playing[currentcore].
append(data1)
63:                     mains[currentcore].
send([status, data1, data2])
64:                     break
65:
66:     s = Server()
67:     s.setMidiInputDevice(99) # Ouvre toutes
les interfaces MIDI connectées.
68:     s.boot().start()
69:     raw = RawMidi(callback)
```

CONCLUSION

Dans cet article, nous avons présenté les derniers développements du module Python « pyo » ainsi que diverses stratégies pour la mise en place de programmes audios multicœurs. L'utilisation combinée du module **multiprocessing** et des techniques de partage de mémoire, de synchronisation et de gestion des événements MIDI de pyo permet d'écrire des programmes audios profitant pleinement de la puissance des ordinateurs multicœurs, qui sont aujourd'hui la norme en matière d'ordinateur personnel. Il est donc tout à fait possible de contourner le GIL de Python et de faire « jouer » votre ordinateur à plein régime. ■

RÉFÉRENCES

[1] Site officiel de pyo : http://ajaxsoundstudio.com/software/pyo/

[2] Documentation en ligne de pyo : http://ajaxsoundstudio.com/pyodoc/index.html

[3] Quelques lois sur l'augmentation des performances des ordinateurs : https://zestedesavoir.com/articles/37/quelques-lois-sur-laugmentation-des-performances-des-ordinateurs/

[4] Inside the Python GIL : http://www.dabeaz.com/python/GIL.pdf

[5] Understanding the Python GIL : http://www.dabeaz.com/python/UnderstandingGIL.pdf

[6] Documentation du module multiprocessing : https://docs.python.org/3.5/library/multiprocessing.html#module-multiprocessing

[7] Site officiel de Portaudio : http://www.portaudio.com/

[8] Site officiel de Jack : http://www.jackaudio.org/

[9] Documentation unix de shm_open : http://pubs.opengroup.org/onlinepubs/007908799/xsh/shm_open.html

[10] Documentation unix de mmap : http://pubs.opengroup.org/onlinepubs/007908799/xsh/mmap.html

Shinken™

Nous recrutons sur Bordeaux des

Développeurs
Python / Javascript

Rejoignez-nous !

CV et Lettre de motivation :
recrutement@shinken-solutions.com

www.shinken-enterprise.com

WE WANT YOU in BORDEAUX

RECONNAISSANCE FACIALE

FRÉDÉRIC LE ROY
[Architecte, libriste, touche à tout]

MOTS-CLÉS : DÉTECTION DE VISAGE, RECONNAISSANCE FACIALE, PYTHON, OPENCV

La reconnaissance faciale est l'art d'identifier une ou plusieurs personnes de manière fiable depuis une photo. Nous allons voir comment mettre en pratique cet art.

La surveillance de masse et l'identification à outrance des individus font désormais (malheureusement) partie de notre quotidien, que l'on soit dans la rue ou derrière un écran (voire de plus en plus les deux en même temps !).

Derrière ces moyens de surveillance et d'identification, il y a trois grands acteurs : les états ou leurs représentants légaux, les grands acteurs du Web : **Facebook**, **Google**, etc. Et les derniers, plus discrets, mais tout aussi redoutables : les entreprises privées qui veulent vous connaître afin de mieux vous cerner.

Vous vous rappelez probablement tous de la scène du film **Minority Report** où les personnes ont leurs yeux scannés à la volée afin de les identifier, que ça soit d'un point de vue sécurité ou marketing. Sans aller jusqu'à l'analyse de nos yeux en temps réel, les technologies permettent maintenant de détecter et d'identifier à la volée des personnes via le flux d'une caméra. Pour réaliser cela, il faut peu de choses au final :

- de la puissance de calcul ;
- des photos de vous.

Passons sur la puissance de calcul qui n'est plus qu'un problème bassement financier de nos jours et concentrons-nous sur les données : les photos. Si on

excepte quelques cas particuliers comme moi-même, la majorité des personnes sont non seulement heureuses de partager leurs photos avec leurs amis ou leur famille sur les réseaux sociaux, mais sont aussi tout aussi ravies d'associer les photos avec l'identité (du réseau social) des personnes présentes dessus.

C'est ainsi que les réseaux sociaux se sont construit des banques de données énormes avec des multitudes de photos associées à des identités. C'est grâce à tout ceci que Facebook (entre autres) est désormais capable d'identifier de lui-même vos amis et les membres de votre famille sur la photo que vous venez de publier !

Nous allons voir dans cet article comment mettre en place à la fois de la détection de visage, mais aussi l'identification de ces visages, ce que l'on appelle communément la reconnaissance faciale.

1. OBJECTIF À ATTEINDRE

À la fin de cet article, nous devrons avoir à disposition une ou plusieurs méthodes pour réaliser l'identification et la reconnaissance de visages. Ces méthodes, implémentées sous forme de programmes **Python** devront pouvoir être réutilisées facilement pour être utilisées ultérieurement dans un autre projet sans devoir tout réécrire.

Afin d'atteindre cet objectif, nous utiliserons les éléments suivants :

- Python en version 2.7 ;
- la librairie **OpenCV** ;
- des photos.

2. UN PEU DE THÉORIE POUR COMMENCER

Voyons comment fonctionnent la détection et la reconnaissance de visages.

2.1 Le point de vue humain

En tant qu'humain, détecter un visage est simple : deux yeux, un nez, une bouche et deux oreilles inscrits dans une ellipse et nous avons trouvé un visage.

Pour identifier un visage, nous faisons appel à notre mémoire : la couleur des yeux, de la peau, la forme du visage, de la bouche, la taille du nez ou des oreilles, la pilosité du visage. Tous ces éléments nous permettent de rapprocher un visage du souvenir que nous avons d'une personne et d'en déduire son identité. Mais il peut nous arriver de nous tromper et de confondre deux personnes qui se ressemblent. Dans le cas où nous hésitons entre deux personnes (c'est Martin ou Henri ?), nous essayons alors de trouver qui a le plus de chance d'être la personne en face de nous par rapport aux souvenirs que nous avons de Martin et de Henri.

2.2 Le point de vue de la machine

La machine va procéder de manière semblable : elle va parcourir l'image pour trouver les visages. Ensuite pour chaque visage, elle va réaliser une comparaison avec ses « souvenirs » : un jeu de données de plusieurs photos du visage pour chaque personne qu'elle connaît. Elle va donner un score à chaque visage et le meilleur score sera l'identité la plus probable d'un visage détecté.

Simple, non ?

Là où ça se complique, c'est que la machine ne sait pas analyser en même temps la couleur des yeux, leur forme, la forme du nez, sa position par rapport à la bouche, comment gérer les moustaches qui vont et viennent au gré des mois, etc.

3. LES DIFFÉRENTES MÉTHODES POUR RÉALISER DE LA RECONNAISSANCE FACIALE

3.1 OpenCV

La première méthode va utiliser la librairie **OpenCV [1]** (pour *Open source Computer Vision*), sous licence BSD. OpenCV a des interfaces dans les langages les plus populaires : **C++**, **C**, Python et **Java** et tourne sur les principaux systèmes d'exploitation : **Linux**, **Windows**, **Mac OS**, **Android**, **iOS**.

OpenCV permet de réaliser des traitements directement depuis un flux vidéo! Mais nous n'aborderons dans cet article que le traitement des images. Ne soyez pas frustrés, ce que nous verrons pourra facilement s'appliquer à un flux vidéo, car ce sont les images du flux qui sont traitées.

Pour détecter les visages, OpenCV utilise **la méthode de Viola et Jones [2]** qui permet de détecter en temps réel (ou presque) des objets dans un flux d'images. L'objet qui nous intéresse étant un visage. Afin de détecter un objet, cette méthode nécessite la création d'un **classifier**. Pour cela, de grands nombres d'images qui représentent l'objet et qui ne représentent pas l'objet ont été utilisés afin d'entraîner ce fameux classifier.

Une fois entraîné, il est prêt à être utilisé pour détecter l'objet où qu'il soit dans une image et quelle que soit sa taille dans l'image. Je ne rentrerais pas dans la théorie mathématique des ondelettes de Haar qui est utilisée, car nous allons utiliser les classifiers déjà bien entraînés et fournis par le projet OpenCV !

Vous pouvez trouver ces classifiers sous forme de fichiers xml dans les sources du projet OpenCV **[3]**.

Pour l'identification des visages, OpenCV propose trois algorithmes différents (que je nomme en anglais pour simplifier vos futures recherches) :

- *Eigenfaces* ;
- *Fisherfaces* ;
- *Local Binary Patterns Histograms*.

Ces 3 algorithmes ont différents avantages et inconvénients que nous verrons plus tard, mais ils ont un point commun important : ce sont des algorithmes (donc, pour caricaturer, des formules mathématiques) !

À noter que l'algorithme des *Local Binary Patterns Histograms* n'est pas impacté par la luminosité de l'image alors que les deux premiers le sont.

3.2 Les réseaux de neurones

Les réseaux de neurones permettent aussi de réaliser de la reconnaissance faciale. Cet article n'abordera pas ces méthodes et se concentrera sur l'utilisation d'OpenCV.

4. DÉTECTION DE VISAGES AVEC OPENCV

4.1 Installation des outils

Nous allons devoir installer les packages suivants :

```
# apt-get install python-opencv
# apt-get install libopencv-dev
```

Le premier va installer la partie Python de OpenCV et par extension OpenCV. Le second n'est pas nécessaire pour utiliser OpenCV, mais contient des modèles au format XML. Nous en aurons besoin pour la détection des visages.

4.2 Vocabulaire

- *classifier* : un classifier est un fichier, résultat d'un entraînement, qui décrit un item à trouver dans une image ;
- *frame* : une *frame* dans OpenCV est un objet image chargé en mémoire. Il est plus aisé de manipuler directement des *frames* dans OpenCV que de passer son temps à charger des images. De plus, ceci permet de travailler sur des données temporaires sans pour autant devoir réaliser des lectures ou écritures sur le disque ;
- *item* : dans la suite, j'appellerai un item, l'élément que nous recherchons. Dans notre cas, il s'agira d'un visage, mais le code pourra facilement être étendu pour utiliser des classifiers qui permettent de détecter un corps ou un objet.

4.3 Vue macro du code

Notre code devra être réutilisable, mais OpenCV fournit, comme nous allons le voir plusieurs **classifiers** : afin de ne pas dupliquer inutilement du code, nous allons créer une classe générique qui sera déclinée pour chaque méthode de détection. Voici le squelette de cette classe :

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import cv2
# + d'autres imports

# la liste des classifiers d'OpenCV que nous utiliserons
TRAINSET_FACE_FRONTAL = "/usr/share/opencv/haarcascades/haarcascade_frontalface_default.xml"
TRAINSET_FACE_PROFILE = "/usr/share/opencv/haarcascades/haarcascade_profileface.xml"
TRAINSET_BODY_FULL = "/usr/share/opencv/haarcascades/haarcascade_fullbody.xml"

# des paramètres liés à la taille des images
DOWNSCALE = 1.1     # ratio appliqué à l'image. La valeur doit être > 1
MAX_SIZE = 800      # taille maximale de l'image en pixels

class OpenCVGenericDetection:

    def __init__(self, image_path, archive_folder = "/tmp/", debug = False):
        """ Constructeur de la classe
        """
        pass

    def set_classifier(self):
        """ Méthode à surcharger
        """
        pass
```

```
    def find_items(self):
        """ Trouver les items dans une frame
        """
        pass

    def extract_items_frames(self):
        """ Extraire les frames des items de
la frame complète
        """
        pass

    def get_items_frames(self, grayscale =
False):
        """ Retourne les frames des items et
leurs coordonnées dans une liste
        """
        pass

    def add_label(self, text, x, y):
        """ Ajout d'un label sur la frame
complète
        """
        pass

    def archive_items_frames(self):
    """ Ecrit dans le dossier d'archive
chaque frame de chaque item en tant qu'une
image
        """
        pass

    def archive_with_items(self):
        """ Ecrit dans le dossier d'archive
la frame complète avec des carrés dessinés
autour des visages détectés
        """
        pass
```

Notez l'import de **cv2**, qui est la librairie d'OpenCV (en version 2 donc). Cet import est suivi de la déclaration de 3 variables globales qui donnent l'emplacement de trois classifiers. Ces trois classifiers sont fournis par le *package* **libopencv-dev** précédemment installé.

En fonction des besoins, il sera judicieux d'inclure ces fichiers dans votre projet, car :

- le chemin est susceptible de varier d'une distribution à une autre ;
- vous pourriez créer vous-même vos propres classifiers.

Le constructeur de la classe prend en paramètres l'image sur laquelle nous allons appliquer la détection de visage, un dossier d'archive dans lequel seront écrites les images résultat et un *flag* debug qui permettra d'afficher pendant les phases de test les résultats dans une fenêtre, ce qui évitera d'aller regarder dans le dossier d'archive de manière systématique.

4.4 Le constructeur

Voyons le code complet du constructeur :

```
    def __init__(self, image_path, archive_folder =
"/tmp/", debug = False):

        """ init
            @image_path : le chemin d'une image sur
le disque
            @archive_folder : dossier d'archive
            @debug : si True, affichage des images
dans une fenêtre
        """
        logging.info("Image : {0}".format(image_path))
        self.image_path = image_path
        self.archive_folder = archive_folder
        self.debug = debug
        self.items = []
        self.items_frames = []

        # On initialise le classifier
        self.set_classifier()

        # Afin de grouper les archives, nous allons
utiliser un préfixe unique
        self.images_prefix = "{0}_{1}_".
format(datetime.datetime.now().strftime("%Y%m%d-
%H%M%S-%f"), self.__class__)

        # On charge l'image dans une frame
        self.frame = cv2.imread(image_path)
        logging.info("Résolution de l'image : '{0}
x{1}'".format(self.frame.shape[0], self.frame.
shape[1]))

        # On vérifie si l'image est trop grande et si
c'est le cas on calcule un ratio pour la réduire
        ratio = 1
        if self.frame.shape[1] > MAX_SIZE or self.
frame.shape[0] > MAX_SIZE:
            if self.frame.shape[1] / MAX_SIZE > self.
frame.shape[0] / MAX_SIZE:
                ratio = float(self.frame.shape[1])
/ MAX_SIZE
            else:
                ratio = float(self.frame.shape[0])
/ MAX_SIZE

        # Si l'image est trop grande, on la retaille
        if ratio != 1:
            newsize = (int(self.frame.shape[1]/ratio),
int(self.frame.shape[0]/ratio))
            logging.info("Redimensionnement de l'image
en : {0}".format(newsize))
            self.frame = cv2.resize(self.frame,
newsize)

        # Affichage de l'image originale
        if self.debug:
            cv2.imshow("preview", self.frame)
            cv2.waitKey()
```

Les points notables sont le chargement de l'image via la fonction **cv2.imread(<chemin>)**, l'analyse de la taille de l'image via un accès direct aux données dans l'objet **self.frame** : **self.frame.shape[0]** et **self.frame.shape[1]**. Pour finir, il faut noter le redimensionnement de l'image avec la fonction **cv2.resize(self.frame, newsize)**.

La fonction **cv2.imshow(<titre de la fenêtre>, <frame>)** d'affichage de l'image dans une fenêtre devra toujours être suivie de la fonction **cv2.waitKey()** qui permet d'attendre l'appui sur une touche avant de passer à la suite.

Comme vous le voyez, la manipulation des images est on ne peut plus simple grâce aux objets *frames* !

4.5 Le choix du classifier

La fonction suivante, **set_classifier** ne fait qu'initialiser le classifier à « rien » (rappelez-vous : on va la surcharger) :

```
    def set_classifier(self):
        """ Méthode à surcharger
        """
        self.classifier = None
```

4.6 La détection des visages

Passons à la fonction **find_items** :

```
    def find_items(self):
        """ Trouver les items dans une frame.
            Valorise self.items en tant que liste
contenant les coordonnées des visages au format (x,
y, h, w).
            Exemple :
                      [[ 483  137   47   47]
                       [ 357  152   46   46]
                       ...
                       [ 126  167   51   51]]

        """
        logging.info("Recherche des items...")

        # On applique le classifier pour détecter les
visages
        items = self.classifier.
detectMultiScale(self.frame, scaleFactor = DOWNSCALE)

        # On valorise self.items et on affiche un peu
de log
        logging.info("Nombre d'items : '{0}'".
format(len(items)))
        logging.info("Items = {0}".format(items))
        self.items = items
```

C'est ici que tout se passe : c'est la fonction **self.classifier.detectMultiScale(<frame>, ...)** qui retourne la liste des visages détectés sous la forme d'une liste de coordonnées.

Cette fonction peut prendre plusieurs paramètres, mais pour les comprendre, voyons comment elle fonctionne !

Nous allons travailler sur une image libre de droits (figure 1).

Fig. 1 : Image originale.

Un classifier est entraîné pour une taille donnée, disons 30x30 pixels. Dans une image, les visages peuvent avoir différentes tailles : 100x100, 75x75, etc. Afin de pouvoir tester le classifier, une pyramide d'images sera générée par la fonction (figure 2).

Fig. 2 : La pyramide d'images.

Ensuite, pour chacune de ses images, la fonction va parcourir toute l'image avec une fenêtre mobile (*sliding window* en anglais) : pour chaque fenêtre, le classifier sera testé (figure 3).

Fig. 3 : La fenêtre mobile.

scaleFactor	Redimensionnement préalable	Temps total (redimensionnement + détection)	Nombre de visages détectés
1,05	non	6s	11
1,1	non	3,1s	7
1,5	non	1,1s	5
2	non	0,9s	2
3	non	0,7s	2
1,05	oui	1,8s	7
1,1	oui	1,1s	5
1,5	oui	0,6s	5
2	oui	0,5s	2
3	oui	0,4s	5

Maintenant que nous connaissons le principe, voyons les différents paramètres qui s'offrent à nous.

4.6.1 Le paramètre scaleFactor

Le **scaleFactor** définit le ratio qui sera appliqué pour générer la pyramide des images. Sa valeur par défaut est **1,1**.

Un ratio de **1,05** correspond à une diminution de taille de 5 %. Plus la valeur de ce paramètre sera élevée, plus le traitement sera rapide, mais plus vous risquez de louper des visages...

Avant même de choisir une valeur pour ce paramètre, vous avez déjà compris ici pourquoi j'ai choisi de retailler l'image originale avant tout traitement : nous allons économiser du temps !

L'image originale fait 1920x1280. Avec un **scaleFactor** à **1,05**, et le redimensionnement à 800 pixels maximum, le programme qui fait le redimensionnement de l'image originale puis la détection prend en moyenne 1,8 secondes.

Si on retire le redimensionnement de l'image avant la phase de détection, le programme prend en moyenne 6 secondes.

Voyons l'impact de différentes valeurs du **scaleFactor** sur notre image originale avec et sans redimensionnement (tableau ci-dessus et images résultat en figure 4 et figure 5).

Fig. 4 : Résultats sans redimensionner l'image originale.

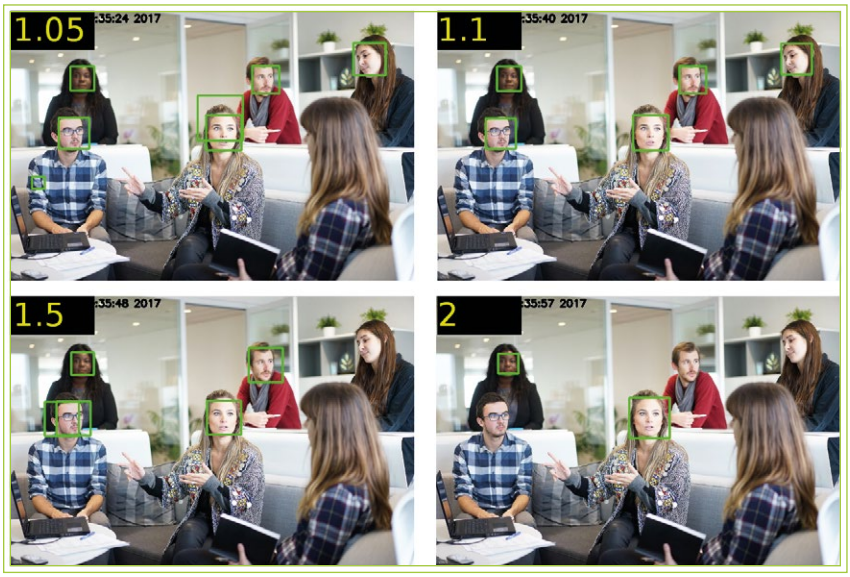


Fig. 5 : Résultats en redimensionnant l'image originale.

Nous voyons ici que le **scaleFactor**, combiné à la taille originale de l'image a vraiment une importance cruciale :

- si le **scaleFactor** est trop grand, des visages ne seront pas identifiés ;
- plus il est faible, plus le temps de traitement est long ;
- plus la taille de l'image originale est grande, plus le temps de traitement sera long et les faux positifs nombreux.

Il semble plus intéressant de combiner un redimensionnement préalable avec un petit scaleFactor.

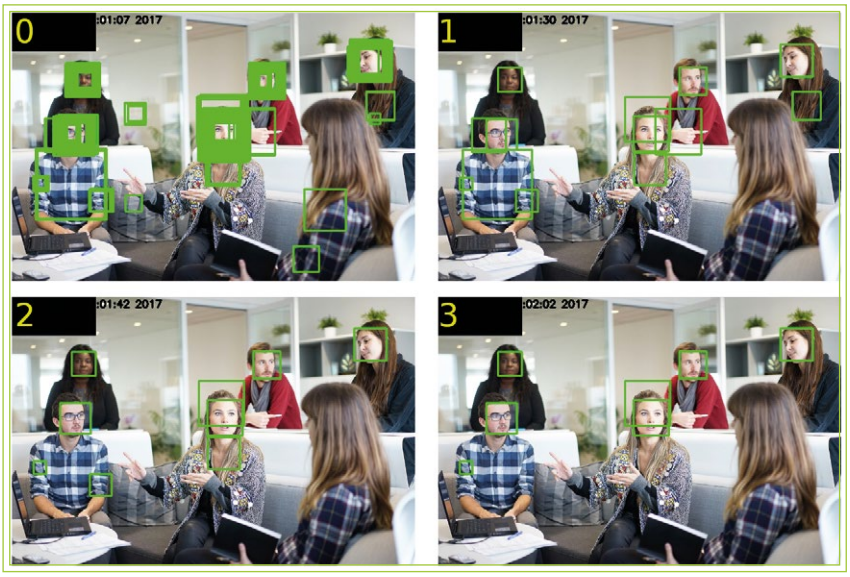


Fig. 6 : Résultats de la variation du paramètre minNeighbors (1/2).

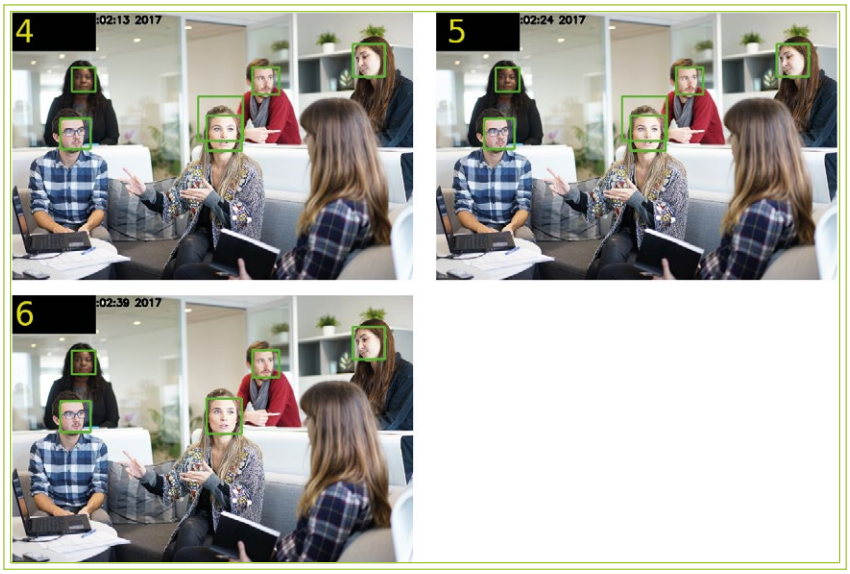


Fig. 7 : Résultats de la variation du paramètre minNeighbors (2/2).

4.6.2 Le paramètre minNeighbors

Ce paramètre permet de filtrer les visages détectés dans une même zone de l'image. Sa valeur par défaut est **3**. Plus la valeur est faible, plus il pourra y avoir des visages détectés dans une même zone.

Nous allons prendre pour les exemples qui suivent un **scaleFactor** à **1,05** et une image retaillée à 800px. Dans le test précédent (figure 5), nous avions 2 visages détectés dans la même région (la jolie blonde au milieu). Nous avions aussi un faux positif sur la manche de la chemise du personnage de gauche.

Nous adapterons ainsi l'appel de la fonction **detectMultiScale** :

```
items = self.classifier.detectMultiScale(self.
frame, scaleFactor = 1.05, minNeighbors = 3)
```

La modification de ce paramètre n'a pas d'impact significatif sur le temps de traitement.

Voyons le nombre de visages détectés pour différentes valeurs dans le tableau ci-contre.

En regardant les images (voir figures 6 et 7), nous comprenons tout de suite le fonctionnement de ce paramètre : la fenêtre glissante détecte en réalité énormément de visages, principalement autour des vrais visages, jusqu'à 627 au lieu de 5 !

Le fait de monter la valeur de ce paramètre (qui est un seuil au final) permet de filtrer tous les visages détectés dans une même zone pour ne garder que le ou les meilleurs candidats. Plus la valeur est importante, plus la précision sera bonne.

minNeighbors	Nombre de visages détectés
0	627
1	13
2	9
3 (valeur par défaut)	7
4	6
5	6
6	5

Attention toutefois, sur d'autres images, une valeur importante aura pour effet de retirer certains visages détectés du résultat final.

4.6.3 Les paramètres minSize et maxSize

Ces deux paramètres qui sont des tuples : **(valeur, valeur)** permettent de définir un intervalle de taille pour la recherche des visages. Ceci peut être utile si vous avez une idée des tailles minimales ou maximales que pourront avoir les visages dans votre image afin de limiter des faux positifs.

4.7 L'extraction des frames des visages

Afin de pouvoir réaliser des traitements ultérieurs sur les visages détectés (au hasard de la reconnaissance !), nous aurons besoin de récupérer ces données. La fonction **extract_items_frames** va stocker sous forme de liste la *frame* et les coordonnées du visage (qui pourront servir plus tard pour taguer l'image originale).

```
    def extract_items_frames(self):
        """ Extraire les frames des items de la frame complète
            Valorise self.items_frames en tant que liste des
frames et coordonnées.
            Exemple :
                    [
                      {  "frame" : ...,
                         "x" : ...,
                         "y" : ...,
                         "w" : ...,
                         "h" : ...
                      },
                      { ... },
                      ...
                    ]
        """
        logging.info("Extractions des frames des items ('{0}'
à extraire)...".format(len(self.items)))
        items_frames = []
        # pour chaque coordonnée d'items...
        for f in self.items:
            # On extrait le sous-ensemble de la frame complète
            x, y, w, h = f
            item_frame = self.frame[y:y+h,x:x+w]
            # Et on le stocke ainsi que ses coordonnées
            items_frames.append( {
                                   "frame" : item_frame,
                                   "x" : x,
                                   "y" : y,
                                   "w" : w,
                                   "h" : h,
                                 })

            # On affiche chaque visage extrait dans une
fenêtre
            if self.debug:
                cv2.imshow("preview", item_frame)
                cv2.waitKey()

        self.items_frames = items_frames
```

Cette fonction n'a rien de très complexe, vous noterez juste que l'extraction d'une *sous frame* est on ne peut plus simple...

4.8 Un getter pour récupérer les données des items

La fonction **get_items_frames** permet de récupérer les données extraites et stockées par la fonction précédente.

Un paramètre lui a été ajouté afin de pouvoir retourner les *frames* en niveau de gris à la place des couleurs : certains traitements pourront nécessiter l'usage d'images en niveaux de gris.

```
    def get_items_frames(self,
grayscale = False):
        """ Retourne les frames
des items et leurs coordonnées
dans une liste
            @grayscale : si True,
retourne les frames des items en
niveaux de gris
        """
        # Si on ne désire pas
un retour en niveaux de gris,
on retourne les données telles
quelles
        if not grayscale:
            return self.items_
frames

        # Dans le cas contraire,
on crée une liste temporaire et
pour chaque frame de la liste
        # originale, on insère
dans la nouvelle liste la frame
convertie en niveaux de gris
        items_frames = []
        for item_frame in self.
items_frames:
            item_frame["frame"] =
cv2.cvtColor(item_frame["frame"],
cv2.COLOR_BGR2GRAY)
            items_frames.
append(item_frame)
        return items_frames
```

C'est la fonction **cv2.cvtColor** qui permet de faire la conversion en niveaux de gris.

4.9 Ajouter un label

La fonction suivante permet d'ajouter un label sur la *frame*. Son usage sera, dans le cas de la reconnaissance faciale, de pouvoir afficher l'identité d'un visage détecté sur l'image.

```
    def add_label(self, text, x, y):
        """ Ajout d'un label sur la frame complète
            @text : texte à afficher
            @x, y : coordonnées du texte à afficher
        """
        # Comme cette fonction sera utilisée pour afficher un label sur un carré autour d'un visage,
        # on ajoute volontairement un peu d'espace entre le label et le carré, sauf si on est au bord de l'image
        if y > 11:
            y = y - 5
        cv2.putText(self.frame, text, (x, y),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0,255,0), 2)
```

L'ajout de texte se fait via la fonction **cv2.putText** qui prend en paramètres :

- la *frame*,
- le texte à afficher,
- les coordonnées d'affichage,
- la police,
- l'échelle de la taille de la police par rapport à la taille par défaut,
- la couleur au format (rouge, vers, bleu),
- l'épaisseur du texte.

4.10 Deux fonctions d'archivage

Pour finir, nous avons les deux fonctions d'archivage :

```
    def archive_items_frames(self):
        """ Ecrit dans le dossier d'archive chaque frame de chaque item en tant qu'une image
        """
        logging.info("Archive les items ('{0}' à archiver)...".format(len(self.items_frames)))
        idx = 0
        # Pour chaque item, on le sauve dans un fichier
        for item_frame in self.items_frames:
            a_frame = item_frame["frame"]
            image_name = "{0}_item_{1}.jpg".format(self.images_prefix, idx)
            logging.info("Archive un item dans le fichier : '{0}'".format(image_name))
            cv2.imwrite(os.path.join(self.archive_folder,  image_name), a_frame)
            idx += 1


    def archive_with_items(self):
        """ Ecrit dans le dossier d'archive la frame complète avec des carrés dessinés autour
            des items détectés
        """
        logging.info("Archive l'image avec les items trouvés...")
        # Dessine un carré autour de chaque item
        for f in self.items:
            x, y, w, h = f #[ v for v in f ]
            cv2.rectangle(self.frame, (x,y), (x+w,y+h), (0,255,0), 3)

        # Ajoute la date et l'heure à l'image
        cv2.putText(self.frame, datetime.datetime.now().strftime("%c"), (5, 25),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0,0,0), 3)

        # On affiche l'image qui va être archivée dans une fenêtre
        if self.debug:
            cv2.imshow("preview", self.frame)
            cv2.waitKey()

        # Ecriture du fichier
        archive_full_name = "{0}_full.jpg".format(self.images_prefix)
        logging.info("Archive file is : '{0}'".format(archive_full_name))
        cv2.imwrite(os.path.join(self.archive_folder,  archive_full_name), self.frame)
```

La première pourrait sembler avoir un intérêt limité, car elle stocke chaque visage détecté dans un fichier image. Nous verrons qu'elle sera bien pratique pour la suite afin de créer des collections de visages.

La seconde fonction, plus utile permet de garder une trace des images complètes avec en surimpression un carré dessiné autour de chaque visage détecté, ainsi que la date et l'heure de traitement.

Le dessin d'un rectangle se fait avec la fonction **cv2.rectangle** qui prend en paramètres :

- la *frame* sur laquelle on souhaite tracer le rectangle,
- les coordonnées d'un angle du rectangle,
- les coordonnées de l'angle opposé,
- la couleur au format (rouge, vert, bleu),
- l'épaisseur du trait.

4.11 On implémente !

Passons maintenant à l'utilisation (enfin !). Tout d'abord, créons la classe finale, qui va hériter de notre classe générique :

```
class OpenCVFaceFrontalDetection(OpenCVGe
nericDetection):

    def set_classifier(self):
        self.classifier = cv2.
CascadeClassifier(TRAINSET_FACE_FRONTAL)
```

Nous instancions ici un classifier depuis le fichier xml pointé par **TRAINSET_FACE_FRONTAL**. Ce fichier décrit un visage vu de face.

Pour tester cette classe sur une image, il suffit de faire :

```
test = OpenCVFaceFrontalDetection(
"./sample_01.jpg".format(i),
          archive_folder = "./archives/",
          debug = True)
test.find_items()
```

Voici la sortie du script Python :

```
INFO:root:Image : ./sample_01.jpg
INFO:root:Résolution de l'image :
'1920x1280'
INFO:root:Redimensionnement de l'image en :
(800, 533)
INFO:root:Recherche des items...
INFO:root:Nombre d'items : '6'
INFO:root:Items = [[117 109  49  49]
 [479 109  54  54]
 [154 352  40  40]
 [ 95 210  61  61]
 [680  64  63  63]
 [387 205  71  71]]
```

Tel quel, ce script nous indique juste qu'il y a des visages et leurs coordonnées. Pour récupérer les images des visages et l'image complète, nous pouvons ajouter ces lignes :

```
test.extract_items_frames()
test.archive_items_frames()
test.archive_with_items()
```

En lançant le script modifié, nous obtenons :

```
INFO:root:Image : ./sample_01.jpg
INFO:root:Résolution de l'image : '1920x1280'
INFO:root:Redimensionnement de l'image en : (800, 533)
INFO:root:Recherche des items...
INFO:root:Nombre d'items : '6'
INFO:root:Items = [[117 109  49  49]
 [479 109  54  54]
 [154 352  40  40]
 [ 95 210  61  61]
 [680  64  63  63]
 [387 205  71  71]]
INFO:root:Extractions des frames des items ('6' à
extraire)...
INFO:root:Archive les items ('6' à archiver)...
INFO:root:Archive un item dans le fichier :
'20170215-210358-112041__main__.
OpenCVFaceFrontalDetection__item_0.jpg'
INFO:root:Archive un item dans le fichier :
'20170215-210358-112041__main__.
OpenCVFaceFrontalDetection__item_1.jpg'
INFO:root:Archive un item dans le fichier :
'20170215-210358-112041__main__.
OpenCVFaceFrontalDetection__item_2.jpg'
INFO:root:Archive un item dans le fichier :
'20170215-210358-112041__main__.
OpenCVFaceFrontalDetection__item_3.jpg'
INFO:root:Archive un item dans le fichier :
'20170215-210358-112041__main__.
OpenCVFaceFrontalDetection__item_4.jpg'
INFO:root:Archive un item dans le fichier :
'20170215-210358-112041__main__.
OpenCVFaceFrontalDetection__item_5.jpg'
INFO:root:Archive l'image avec les items trouvés...
INFO:root:Archive file is : '20170215-210358-112041__
main__.OpenCVFaceFrontalDetection__full.jpg'
```

Nous obtenons un set de 7 images (voir figure 8) :

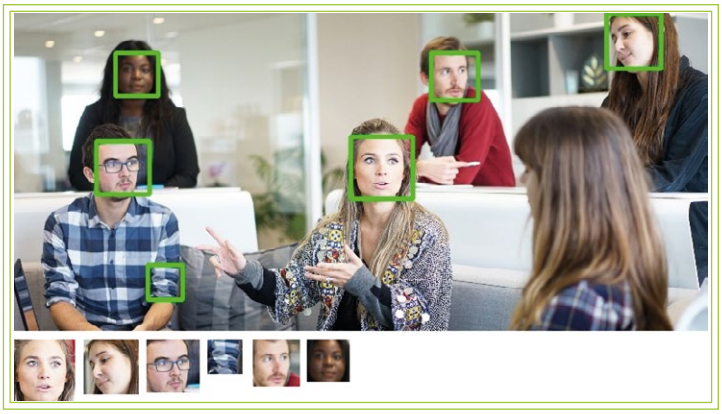


Fig. 8 : Images générées par la détection de visage.

4.12 Aller plus loin

Le code que nous avons créé ici est facilement réutilisable avec d'autres classifiers : si d'aventure vous trouviez un fichier de classifier qui détecte une tasse, vous n'auriez qu'à déclarer une classe qui l'utilise :

```
TRAINSET_FOR_A_CUP = "/chemin/vers/le/classifier.xml"

class OpenCVCupDetection(OpenCVGenericDetection):

    def set_classifier(self):
        self.classifier = cv2.
CascadeClassifier(TRAINSET_FOR_A_CUP)
```

Puis à l'utiliser ainsi :

```
test = OpenCVCupDetection("./des_tasses.jpg".format(i),
                          archive_folder = "./archives/",
                          debug = True)
test.find_items()
```

Il existe des classifiers qui détectent :

- les visages de profil,
- les corps humains,
- les yeux,
- les mains,
- etc.

5. BILAN SUR LA DÉTECTION DES VISAGES AVEC OPENCV

Nous avons créé une librairie fonctionnelle et réutilisable : la première mission est remplie ! J'attire votre attention sur le fait que même si la détection de visages est au final très simple (création d'un classifier, application du classifier), il y a un travail périphérique non négligeable en proportion pour que ça soit exploitable :

- redimensionnement des images,
- extraction des visages depuis les coordonnées fournies,
- modification des images pour encadrer les visages.

Cette librairie n'est pourtant pas parfaite : elle sera à compléter afin de pouvoir passer en paramètres à minima les valeurs de `scaleFactor` et `minNeighbors`, mais aussi la dimension maximale de l'image pour le redimensionnement.

En fonction du cas d'usage que vous aurez, il faudra porter une attention particulière à correctement tuner ces paramètres afin d'optimiser la détection et de limiter les faux positifs, tout en respectant vos contraintes de réactivité et de puissance.

6. LA RECONNAISSANCE DE VISAGES AVEC OPENCV

6.1 Installation des outils

Nous allons avoir besoin d'installer le package suivant :

```
# apt-get install python-numpy
```

6.2 Vocabulaire

Nous utiliserons les termes :

- modèle (ou *model*) : le modèle de reconnaissance faciale ;
- confiance (ou *confidence*) : une valeur numérique qui indique la confiance que l'on a dans le pronostic fourni par le modèle. Plus la confiance est proche de zéro, plus le pronostic est fiable ;
- jeu de données : il s'agit des données connues - un ensemble de visages par identité ;
- jeu de test : il s'agit d'une ou plusieurs images contenant un ou plusieurs visages. Ce sont sur ces images que nous voulons identifier des personnes.

6.3 Construction du jeu de données

Pour construire mon jeu de données, j'ai procédé de manière semi-manuelle. Dans un premier temps, j'ai pioché dans mes photos familiales pour trouver des photos avec les différents membres de la famille.

Puis, j'ai passé chacune de ces images à la moulinette de la librairie de détection des visages et j'ai récupéré dans le dossier d'archive les images de tous les visages détectés.

J'ai ensuite manuellement déplacé ces images de visages dans des dossiers qui correspondent aux membres de la famille.

Rappelez-vous que ce qui fait la force d'un jeu de données c'est sa taille ! Plus vous aurez de visages de chaque personne, meilleure sera (en théorie) l'identification !

6.4 Vue macro du code

De la même manière que pour la détection, nous allons créer une classe. Cette classe aura plusieurs parties :

- le constructeur qui va initialiser ce qu'il y a à initialiser,
- une fonction qui chargera en mémoire le jeu de données,
- une fonction qui entraînera le modèle avec un jeu de données,
- une dernière fonction qui prendra en paramètre la *frame* d'un visage et qui retournera l'identité correspondant au visage, ainsi qu'un niveau de confiance.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import cv2
import numpy
# ... autres imports

class OpenCVGenericRecognition:

    def __init__(self, trainset_path, archive_folder = "/tmp/"):
        """ Constructeur
            @trainset_path : le chemin vers le jeu de données
            @archive_folder : le dossier d'archive
        """
        pass

    def load_trainset(self):
        """ Chargement en mémoire du jeu de données
        """
        pass

    def train(self):
        """ Entraînement du jeu de données
            Méthode à surcharger
        """
        pass

    def recognize(self, frame):
        """ Applique la reconnaissance de visage à une frame
            @param frame : la frame
        """
        pass
```

Contrairement à la classe de détection où à la création nous lui fournissions l'image à traiter, ici pour des raisons de performances, nous devrons dans un premier temps instancier la classe, lancer le chargement du jeu de données et son entraînement.

Une fois ces actions effectuées, nous pourrons lancer la reconnaissance de visage pour chaque visage que nous aurons à disposition. Il ne sera nécessaire de relancer le chargement du jeu de données puis l'entraînement que si le jeu de données a été modifié.

6.5 Le constructeur

Le constructeur se contente d'initialiser des variables.

```
    def __init__(self, trainset_path, archive_folder = "/tmp/"):
        """ Constructeur
            @trainset_path : le chemin vers le jeu de données
            @archive_folder : le dossier d'archive
        """
        logging.info("Trainset : {0}".format(trainset_path))
        self.trainset_path = trainset_path
        self.archive_folder = archive_folder

        # Taille de conversion des images du jeu de données
        # Ceci permettra de mettre des images de tailles variables dans le jeu de données
        self.resize_faces = (170, 170)    # valeur prise un peu au hasard je dois l'avouer

        # Initialisations diverses
        self.model = None
        self.trainset_images = []
        self.trainset_index = []
        self.trainset_identities = []
```

Les trois dernières sont importantes et sont liées entre elles :

- **self.trainset_images** contiendra chaque image du jeu de données,
- **self.trainset_index** contiendra pour chaque image du jeu de données l'index de l'identité associée,
- **self.trainset_identities** contiendra la liste des identités du jeu de données.

6.6 Chargement en mémoire du jeu de données

Le chargement du jeu de données va consister à récupérer la liste des noms de dossiers du jeu de données (le nom du dossier correspondant à l'identité du visage), puis à charger en mémoire ces images (voir figure 9, page suivante).

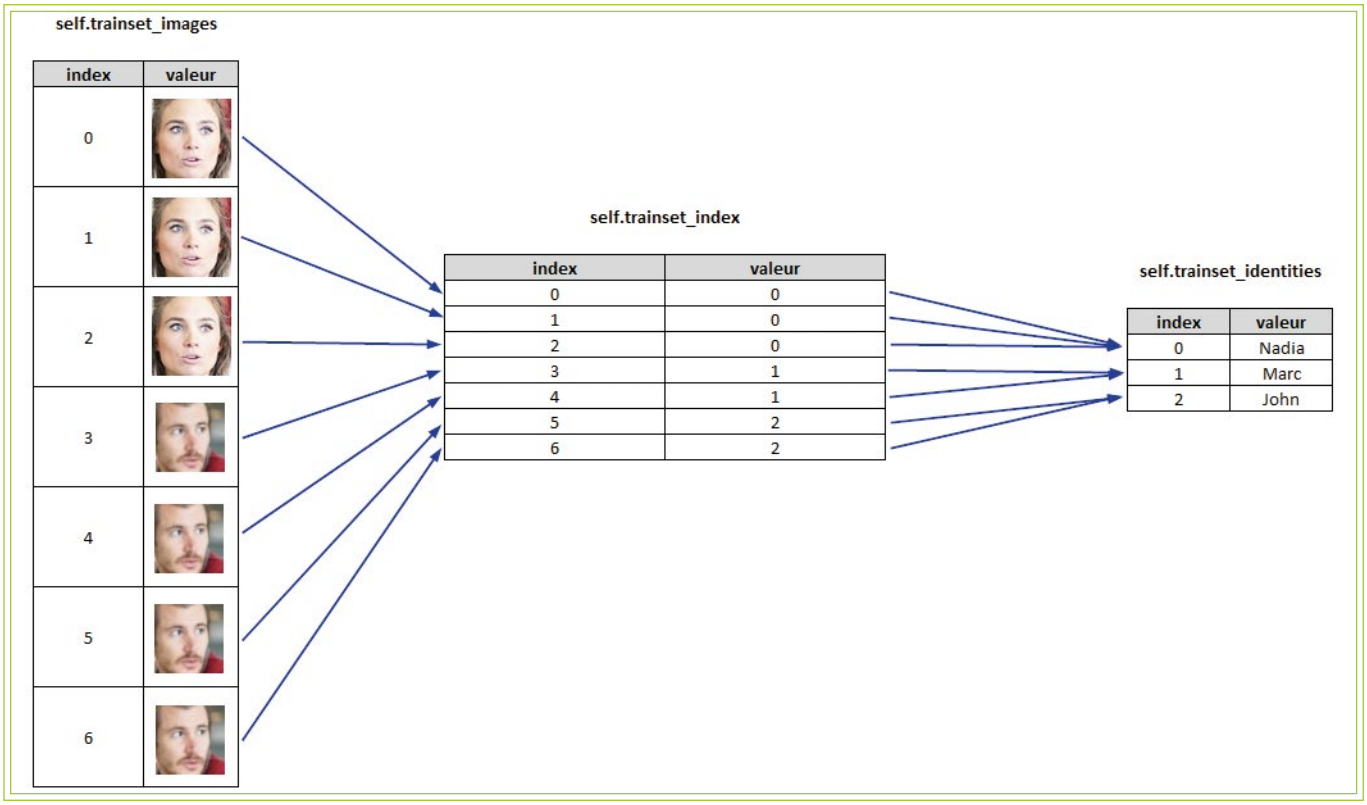


Fig. 10 : Représentation du jeu de données chargé en mémoire.

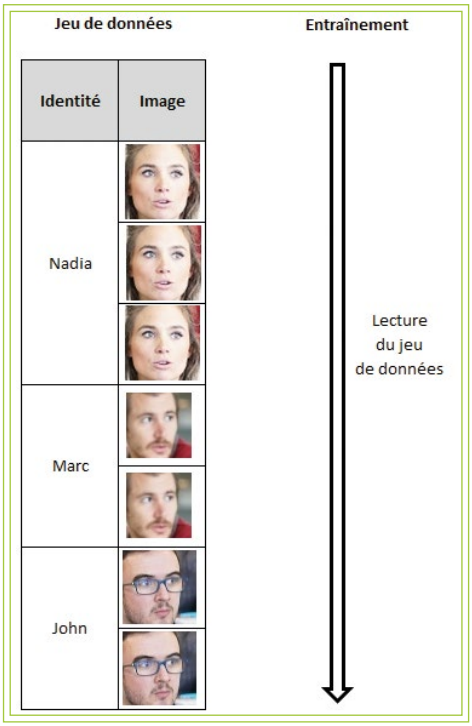


Fig. 9 : Lecture du jeu de données (dans la vraie vie, chaque image d'une identité est différente des autres : ici, il s'agit seulement d'une illustration de principe).

Afin de pouvoir avoir un lien entre les images chargées en mémoire et l'identité, les 3 variables **self.trainset_*** sont nécessaires. Regardez la figure 10, en page suivante, pour bien comprendre le principe.

```
    def load_trainset(self):
        """ Chargement en mémoire du jeu de données
        """
        logging.info("Chargement du trainset...")

        c = 0
        self.trainset_images = []       # images chargées
        self.trainset_index = []        # index (numéro) de
l'identité
        self.trainset_identities = []   # identités du jeu de
données

        # On parcourt le dossier contenant le jeu de données.
        # Il doit contenir des dossiers. Le nom du dossier sera
le nom de la personne, son identité.
        # Chaque dossier contiendra des images du visage de la
personne.
        for dirname, dirnames, filenames in os.walk(self.
trainset_path):
            for subdirname in dirnames:
                self.trainset_identities.append(subdirname)
                logging.info("- identité
'{0}'...".format(subdirname))
                subject_path = os.path.join(dirname, subdirname)
                for filename in os.listdir(subject_path):
```

```
                    try:
                        # On convertit en
niveaux de gris et on redimensionne dans une
dimension commune
                        im = cv2.imread(os.
path.join(subject_path, filename), cv2.
IMREAD_GRAYSCALE)
                        im = cv2.resize(im,
self.resize_faces)
                        # On convertit
l'image en 'array'
                        self.trainset_images.
append(numpy.asarray(im, dtype=numpy.uint8))
                        # Et on dit que cet
array correspond à l'identité n°c
                        self.trainset_index.
append(c)
                    except IOError, (errno,
strerror):
                        logging.error("I/O
error({0}): {1}".format(errno, strerror))
                    except:
                        logging.
error("Unexpected error:", sys.exc_info()[0])
                        raise

                c = c+1
```

6.7 Entraînement

Maintenant que nous avons les données, nous pouvons les entraîner...

Non, vous y avez cru ? Rappelez-vous, la fonction d'entraînement est à surcharger, on la verra un peu plus tard ;).

6.8 Reconnaissance

Quelle que soit la méthode de reconnaissance qui sera appliquée, la reconnaissance d'un visage se fera tout le temps de la même manière avec l'appel de la fonction **recognize** que voici :

```
    def recognize(self, frame):
        """ Applique la reconnaissance à une
frame d'un visage
            @param frame : la frame
            Retourne :
            - True si l'identity est estimée
valide par rapport à l'indice de confiance.
False sinon
            - l'identité
            - l'indice de confiance tronqué
(les décimales n'ont que peu d'intérêt ici)
        """
        # Redimensionnement de la frame du
visage aux dimensions du trainset
        frame = cv2.resize(frame, self.
resize_faces)
        # Reconnaissance
        # confidence = 0 ==> score parfait ! Plus la
confidence est faible, plus on est sûr de l'identité

        [idx, confidence] = self.model.predict(frame)
        found_identity = self.trainset_identities[idx]
        # En plus de définir le flag connu ou inconnu,
nous allons renommer l'identité
        # en la préfixant de n/a si le seuil de
confiance est dépassé
        if confidence < 120:
            identity = found_identity
            found = True
        else:
            identity = "n/a ({0})".format(found_
identity)
            found = False
        return found, identity, int(confidence)
```

Cette fonction prend en paramètre une *frame* qui contient seulement un visage. Comme vous le devinez déjà, cette *frame* sera le résultat de l'utilisation de la détection des visages.

Cette *frame* sera redimensionnée à la taille des images du modèle de données en mémoire puis passée en paramètre à la fonction **predict** du modèle qui sera créé dans la fonction d'entraînement.

Cette fonction renvoie deux informations :

- l'index de l'identité : il permet grâce au tableau **self.trainset_identities** de retrouver le label de l'identité ;
- un indice de confiance : plus il sera petit, meilleur le pronostic sera. C'est grâce à cet indice de confiance que l'on peut décider si l'identité renvoyée est la bonne ou si, au final il va s'agir d'un visage inconnu ;

Pour finir, notre fonction renvoie un *flag* pour indiquer si la personne est connue ou pas, l'identité supposée et l'indice de confiance.

6.9 Les trois méthodes disponibles

Comme nous l'avons vu au début de cet article, il existe trois méthodes de reconnaissance faciale dans OpenCV :

- *Eigenfaces* ;
- *Fisherfaces* ;
- *Local Binary Patterns Histograms*.

Expliquer comment elles fonctionnent serait trop long à aborder dans le cadre de cet article. Il faut surtout retenir deux points :

- la troisième méthode n'est pas sensible aux variations de lumières alors que les deux premières le sont. Si vous cherchez à faire de la reconnaissance faciale dans un environnement extérieur, privilégiez systématiquement cette troisième méthode !
- des essais seront nécessaires pour vérifier la méthode qui aura les résultats les plus probants sur votre jeu de test.

Pour l'article, j'ai choisi de vous montrer l'utilisation de la méthode *Local Binary Patterns Histograms*. Mais comme je suis d'humeur généreuse, j'ai ajouté en commentaire les lignes à utiliser pour utiliser les deux autres algorithmes.

```
class OpenCVFaceRecognitionLBPH(OpenCVGeneric
Recognition):

    def train(self):
        """ Entraînement du jeu de données
            Méthode à surcharger
        """
        logging.info("Entraînement du trainset...")

        # Les autres algorithmes :
        #self.model = cv2.createEigenFaceRecognizer()
        #self.model = cv2.createFisherFaceRecognizer()

        # Celui que l'on va utiliser :
        self.model = cv2.createLBPHFaceRecognizer()
        self.model.train(numpy.asarray(self.trainset_
images), numpy.asarray(self.trainset_index))
```

Vous noterez que le choix de l'un ou l'autre des algorithmes tient à une seule ligne. Attention toutefois, ici nous utilisons les valeurs par défaut des paramètres des fonctions (vu que nous n'en avons spécifié aucun) ! Si nous éprouvons le besoin de tuner ces paramètres, sachez qu'ils ne sont pas les mêmes pour les trois algorithmes !

Fig. 11 : Mon jeu de test des 7 familles anonymisé.

6.10 Mise en œuvre

Nous avons maintenant toutes les briques. Nous allons pouvoir les tester sur un jeu de test (voir figure 11).

Dans ce jeu de test où les photos sont de bonne qualité et les visages bien orientés (et sans grimaces), nous devrions obtenir de bon résultats !

Il y a 13 visages (si nous ne tenons pas compte du vieux barbu) à détecter, dont 2 qui sont inconnus de mon jeu de données.

Voici le code utilisé :

```
from ocv_detection import
OpenCVFaceFrontalDetection

# On instancie la classe de reconnaissance
avec en paramètre le jeu de données
reco = OpenCVFaceRecognitionLBPH("./samples/
faces/",
                                 archive_
folder = "../../archives/")
# On charge le jeu de données en mémoire
reco.load_trainset()
# On lance l'entraînement
reco.train()

# Utilisation de mon jeu de test n°6
dir = "./test/jeu6"
# On boucle sur chaque image du jeu de test
for filename in os.listdir(dir):
    filepath = os.path.join(dir, filename)
    # pour une image, je lance la détection
des visages
    detect = OpenCVFaceFrontalDetection(fil
epath,
                        archive_folder =
"../../archives/",
                        debug = True)
    detect.find_items()
    detect.extract_items_frames()
    # et on boucle sur la frame de chaque
visage (convertie en niveaux de gris)
    for item in detect.get_items_
frames(grayscale = True):
        # On récupère le statut (connu ou
non), l'identité théorique et l'indice de
confiance
        known, identity, confidence = reco.
recognize(item["frame"])
        # On génère un label qu'on va écrire
sur l'image
        label = "{0} ({1})".format(identity,
confidence)
```

```
        logging.info("Trouvé : {0}".
format(label))
        x = item["x"]
        y = item["y"]
        detect.add_label(label, x, y)
        # Et on archive à la fois les
visages et l'image complète
        detect.archive_items_frames()
        detect.archive_with_items()
```

Avec ce code, j'obtiens sur mon jeu de test (dont aucun visage n'a été intégré dans le jeu de données) 9 résultats ok et 3 résultats ko.

Pour être franc, je m'attendais à mieux !

6.11 Tuning

La fonction **cv2.createLBPHFaceRecognizer** a plusieurs paramètres optionnels. Voyons si nous pouvons améliorer un peu les résultats ! Les paramètres qui existent sont :

- **radius** : a priori augmenter ce paramètre augmente la précision... j'ai eu l'effet inverse... passons !
- **neighbors** : de la même manière, augmenter ce paramètre augmenterait la précision... Je n'ai pas vu de différence de résultat flagrante en passant de **8** (la valeur par défaut) à **16**. Par contre, il y a un réel impact sur le temps de traitement !
- **grid_x** et **grid_y** (valeur par défaut : **8**) : augmenter ces deux paramètres a eu pour effet de diminuer l'indice de confiance... les diminuer a augmenté les indices de confiance, mais a généré plus de mauvais résultats...
- **threshold** : ce paramètre permet en théorie de définir le seuil de notre indice de confiance. S'il est au-dessus, la fonction de prédiction est censée retourner **-1**. Dans la pratique, elle me renvoie **1.79769313486e+308** ! Il reste donc plus pratique de gérer nous-mêmes le seuil de l'indice de confiance.

Le tuning de ces fonctions nécessite malheureusement des connaissances approfondies dans les concepts mathématiques utilisés par les algorithmes afin de bien comprendre les impacts...

6.12 Test rapide des autres algorithmes

Afin de comparer les résultats des autres algorithmes, j'ai testé mon jeu de test avec les valeurs par défaut de chaque algorithme. J'obtiens ces résultats :

Algorithme	Nombre de visages identifiés correctement	Nombre de visages mal identifiés	Pourcentage de réussite
Eigenfaces	7	5	58 %
Fisherfaces	9	3	75 %
Local Binary Patterns Histograms	9	3	75 %

Comme vous le voyez, les résultats ne sont pas vraiment à la hauteur de mes attentes (j'espérais un modeste 90%).

CONCLUSION

Nous sommes arrivés au bout de l'exercice pour cet article. En tant qu'utilisateur de mon code, je suis un peu déçu par les résultats obtenus...

Les mauvais résultats pourraient-ils venir du jeu de données ? C'est une piste à ne pas négliger... Ou est-ce lié à des jeux de données et de tests très orientés famille ? C'est également possible. À ce jour, très peu de solutions commerciales de vidéosurveillance mettent en avant des capacités de reconnaissance faciale et ce n'est sûrement pas un hasard : entre réussir à avoir 75 % de réussite pour « jouer » et avoir un taux de réussite suffisant pour être un argument de vente, il y a un fossé énorme ! Quant aux mastodontes du Web, ils disposent de la puissance et des connaissances scientifiques qu'il manque aux humbles bidouilleurs que nous sommes.

Quoi qu'il en soit, si vous comptez à votre tour creuser le sujet, ne négligez surtout pas l'aspect tuning ! Nous avons vu qu'il est très important de paramétrer au mieux afin de faire le bon compromis entre vitesse et efficacité. ■

RÉFÉRENCES

[1] Site officiel d'*OpenCV* : http://opencv.org/

[2] La méthode de Vialo et Jones sur *Wikipédia* : https://fr.wikipedia.org/wiki/M%C3%A9thode_de_Viola_et_Jones

[3] Les classifiers d'*OpenCV* : https://github.com/opencv/opencv/tree/master/data/haarcascades

NIX ET NIXOS

DAMIEN CASSOU
[Enseignant chercheur et développeur professionel, auteur et contributeur sur de nombreux projets libres dont NixOS]

MOTS-CLÉS : PAQUETS, DISTRIBUTION, FONCTIONNEL, SYSTÈME

Nix est un gestionnaire de paquets purement fonctionnel pour GNU/Linux et OS X. NixOS est une distribution GNU/Linux basée sur Nix et entièrement configurable dans un fichier texte. Nous verrons comment les fonctionnalités de mise à jour atomique, d'annulation, ou encore d'installation en parallèle d'un même paquet font de l'écosystème Nix un superbe environnement de travail.

Dans cet article, nous présenterons tout d'abord **Nix**, un gestionnaire de paquets purement fonctionnel pour GNU/Linux et OS X. Nous présenterons ses avantages et décrirons des fonctionnalités que l'on ne trouve nulle part ailleurs. Nous donnerons deux exemples de paquets Nix.

Nous discuterons ensuite de **NixOS**, une distribution Linux basée sur Nix entièrement configurable au travers d'un fichier texte. Nous décrirons NixOS au travers de morceaux du fichier de configuration tiré de mes propres préférences système (description du démarrage, des services utiles et des utilisateurs). Nous décrirons quelques avantages, mais aussi un inconvénient de NixOS par rapport aux distributions plus classiques.

1. NIX, LE GESTIONNAIRE DE PAQUETS

Nix **[1]** est un gestionnaire de paquets au même titre que **Apt** pour **Debian** et **Homebrew** pour **OS X**. Il est le résultat des travaux de thèse d'Eelco Dolstra et de nombreux contributeurs depuis dix ans. Il fonctionne en parallèle du gestionnaire de paquets natif sans jamais entrer en conflit : tous les paquets installés le sont dans `/nix/store` et les dossiers `/usr`, `/lib`, etc. ne sont jamais modifiés.

Nix est construit autour des fonctionnalités suivantes qui le distinguent des autres gestionnaires de paquets :

- *Mise à jour atomique* : lorsque Nix doit *modifier l'environnement* (installer/supprimer/mettre à jour), il n'y a que deux possibilités : soit il réussit, soit aucun changement n'est fait. Par exemple, si vous demandez la suppression d'un paquet et l'installation d'un autre et que le deuxième paquet ne peut être installé, le premier ne sera pas supprimé.
- *Annulation* : lorsque Nix modifie l'environnement, si quelque chose ne vous plaît pas, vous pouvez toujours revenir en arrière aussi loin que vous le souhaitez.
- *Installation côte à côte de plusieurs versions* : Nix permet l'installation d'autant de versions d'un paquet qu'on le souhaite. Vous pouvez par exemple installer plusieurs versions de **Firefox** ou encore la même version d'**Emacs** compilée avec des options différentes (e.g., une **GTK2** et une **GTK3**).
- *Environnement utilisateur* : chaque utilisateur d'une même machine peut modifier son environnement sans impacter les autres et sans avoir besoin des droits `root`.

Et l'espace disque dans tout ça ? Effectivement, si chaque utilisateur se met à installer une version différente de tous les paquets et que l'on peut annuler et revenir arbitrairement loin en arrière, ça peut prendre beaucoup de place. Cependant, Nix a deux mécanismes pour limiter les problèmes. `nix-store --optimize` fait le tour de tous les fichiers de tous les paquets installés et remplace les fichiers similaires par des liens. Le deuxième mécanisme est `nix-collect-garbage` qui efface tous les paquets qui ne sont plus utiles à personne. Une version destructrice `nix-collect-garbage --delete-older-than 2d` permet aussi d'effacer toutes les versions plus vieilles que 2 jours (`2d`), interdisant alors d'annuler et de revenir plus de 2 jours en arrière. Aujourd'hui, sur mon système avec Firefox, Emacs, GNOME, **LibreOffice** etc., Nix utilise 34Gio d'espace disque (d'après `df -h`). Après un nettoyage non destructeur, l'espace occupé tombe à 13Gio seulement.

Si un utilisateur peut décider de la version qu'il veut de chaque paquet ainsi que des options de compilation, est-ce que cela signifie des heures de compilation pour avoir un système fonctionnel ? Non. À partir de la définition d'un paquet, Nix génère un hashage cryptographique (une chaîne de caractères résumant le paquet). Au moment de l'installation, Nix demande à une liste (configurable) de serveurs s'ils ne disposent pas déjà d'un paquet binaire pour ce hashage. S'il y en a un, le paquet binaire est téléchargé et installé. Si ce n'est pas le cas, le paquet source est téléchargé et compilé localement.

1.1 Avantages d'un gestionnaire de paquets comme Nix

Voyons maintenant quelques cas pratiques d'utilisation qui montrent l'intérêt de Nix.

Vous voulez synchroniser certains dossiers de votre ordinateur portable avec l'ordinateur fixe de votre conjoint. Vous avez l'habitude d'utiliser le très bon **Unison** pour faire ça, mais la version sur votre portable est bien plus récente que sur la machine de votre conjoint. Vous ne pouvez pas facilement mettre à jour la version d'Unison sur la machine de votre conjoint alors vous utilisez Nix pour forcer l'installation d'une version antérieure d'Unison sur la vôtre.

Vous êtes développeur et souhaitez installer une bibliothèque dont votre code dépend dans une version non encore officielle. Vous demandez à Nix d'installer cette bibliothèque à partir d'une branche particulière d'un dépôt Git.

Vous voulez basculer votre machine de test sur la dernière version de PHP et voir comment se comportent vos applications. Vous faites l'installation et vous vous rendez compte qu'il y a des problèmes, mais vous n'avez pas le temps de les résoudre. Vous décidez de revenir à votre système tel qu'il était avant la bascule.

Vous êtes administrateur système dans une université et les enseignants vous demandent d'installer **Eclipse** avec des ensembles de *plugins* différents. Vous demandez alors à Nix d'installer Eclipse 4.5.1 d'un côté avec un ensemble de *plugins* pour **Java** et d'un autre côté pour **C**. Pour que ces 2 installations ne prennent pas trop de place sur le disque dur, vous lancez `nix-store --optimize`.

1.2 Comment ça marche ?

Chaque paquet est installé dans un dossier dédié de `/nix/store`, par exemple `/nix/store/b6gvzjyb2pg0kjfwrjmg1vfhh54ad73z-firefox-33.1/` où `b6gvzjyb2pg0...` est un identifiant unique (un hachage cryptographique de la description du paquet). Quand on dit que Nix est un gestionnaire de paquets purement fonctionnel, ça signifie que le contenu d'un tel dossier ne change jamais. Nix distingue un paquet installé (globalement) d'un paquet ajouté (par un utilisateur). Un paquet installé est un paquet dont il existe un dossier dédié dans `/nix/store`. Un paquet ajouté est un paquet dont les binaires sont dans le `PATH` de l'utilisateur. L'environnement de l'utilisateur (liste des paquets qu'il demande) est considéré par Nix comme une sorte de paquet (appelé *user environment*) dont les dépendances sont les paquets souhaités. Mettre à jour son environnement revient donc à créer un nouveau dossier dans `/nix/store` ayant un sous-dossier `bin` regroupant des liens symboliques :

```
$ ls -l /nix/store/d5[...]2-user-environment/bin/
| head
ack -> /nix/store/gc[...]6-perl-ack-2.14/bin/ack
ag -> /nix/store/bi[...]9-silver-searcher-0.32.0/
bin/ag [...]
```

Pour pouvoir annuler un changement dans l'environnement d'un utilisateur (par exemple la mise à jour d'un paquet), Nix maintient une liste ordonnée de tous les environnements ainsi qu'un lien vers l'environnement actuel :

```
$ ls -l /nix/var/nix/profiles/per-user/cassou
profile -> profile-41-link
profile-37-link -> /nix/store/nn[...]x-user-environment
profile-38-link -> /nix/store/yr[...]j-user-environment
[...]
profile-41-link -> /nix/store/d5[...]2-user-environment
```

Ici, l'utilisateur **cassou** en est à son quarante et unième changement d'environnement. Le lien symbolique **profile** pointe toujours vers le profil en cours d'utilisation et c'est son sous-dossier **bin/** que chaque utilisateur doit ajouter dans sa variable **PATH**. Dans le résultat de la commande, on voit aussi que le trente-septième environnement est le plus ancien encore accessible : les autres ont été supprimés par un **nix-collect-garbage –delete-older-than...**.

Pour faire son travail de nettoyage des paquets inutiles, **nix-collect-garbage** a besoin de connaître la liste des dépendances à l'exécution d'un paquet installé. En effet, **nix-collect-garbage** doit garder tous les environnements de tous les utilisateurs, toutes les dépendances de ces environnements ainsi que toutes les dépendances des dépendances (récursivement) puis effacer tout le reste. Pour connaître la liste des dépendances à l'exécution d'un paquet, Nix analyse chaque fichier du dossier dédié dans **/nix/store**. Si le fichier est un lien symbolique vers un fichier d'un autre paquet dans **/nix/store**, Nix considère que c'est une dépendance. Si le fichier est un fichier réel (textuel ou binaire), Nix regarde à l'intérieur, cherche toutes les références à **/nix/store** et considère chacune comme une dépendance à l'exécution **[2]**. Par exemple, on voit ici que bash fait des références à des bibliothèques dynamiques dans **/nix/store** :

```
$ ldd /nix/store/65[...]z-bash-4.3-p42/bin/bash
libreadline.so.6 => /nix/store/q9[...]
q-readline-6.3p08/lib/libreadline.so.6
libc.so.6 => /nix/store/98[...]a-glibc-2.23/lib/
libc.so.6 [...]
```

Cette liste de dépendances est calculée une seule fois pour chaque dossier de **/nix/store** puis stockée dans une base : chaque dossier étant immuable, la liste des dépendances ne change jamais.

1.3 Définition de paquets Nix

Présentons rapidement comment les paquets sont définis sous Nix. Une définition de paquet simple est celle pour **GNU Hello**, un programme qui affiche simplement **Hello, world!** sur la sortie standard :

```
{ stdenv, fetchurl }:
stdenv.mkDerivation rec {
  name = "hello-2.10";
  src = fetchurl {
    url = "mirror://gnu/hello/${name}.tar.gz";
    sha256 = "0s[...]i";
  };
  meta = {
    description = "A program that produces a
familiar, friendly greeting";
    [...autres informations sur le paquet
comme la licence et les mainteneurs Nix...]
}; }
```

Sans rentrer dans les détails, on voit que la définition Nix d'un paquet peu complexe est extrêmement simple. On y voit le nom du paquet (**hello**) et son numéro de version (**2.10**). On voit aussi où Nix doit récupérer les sources du paquet (un miroir GNU) ainsi que quelques métadonnées. Lorsque rien n'est spécifié, Nix essaye le processus d'installation standard : **tar**, **./configure**, **make** et **make install**.

Une définition de paquet est une fonction dont les paramètres sont les outils et les dépendances nécessaires à la construction du paquet et dont le corps construit le paquet. Ici, la première ligne indique les paramètres de la fonction (**stdenv** et **fetchurl**), tout le reste étant le corps de la fonction.

stdenv.mkDerivation est une fonction qui prend en paramètre un dictionnaire (ensemble non ordonné de paires clé/valeur) et produit un paquet. Le dictionnaire possède ici 3 clés : **name**, **src** et **meta**.

Analysons une partie de la définition d'un paquet (Emacs 25) paramétrable par l'utilisateur **[3]** :

```
{ stdenv, lib, [...], withX ? !stdenv.
isDarwin, withGTK3 ? false, gtk3, withGTK2
? true, gtk2 }:
let toolkit = if withGTK3 then "gtk3"
              else if withGTK2 then "gtk2"
              else "lucid";
in stdenv.mkDerivation rec {
  name = "emacs-25.1-rc1";
  src = fetchurl {
    url = "ftp://alpha.gnu.org/gnu/emacs/
pretest/${name}.tar.xz";
    sha256 = "0c[...]0";
```

```
  };
  buildInputs = [ ncurses [...] autoconf
automake ]
    ++ stdenv.lib.optionals withX [
xlibsWrapper libpng [...] ]
    ++ stdenv.lib.optional (withX &&
withGTK2) gtk2
    ++ stdenv.lib.optional (withX &&
withGTK3) gtk3;
  configureFlags = if withX then [
"--with-x-toolkit=${toolkit}" [...] ]
                   else [ "--with-x=no"
"--with-jpeg=no" [...] ];
[...] }
```

Ici, la définition du paquet (toujours une fonction) déclare quelques outils et dépendances puis quelques paramètres configurables par l'utilisateur. Par exemple, le paramètre **withX** permet de spécifier si l'on souhaite compiler Emacs pour X. La valeur par défaut de ce paramètre est **true** pour tous les systèmes sauf **Darwin** (le cœur d'OS X). Certains paramètres sont des booléens (ici, ils commencent par **with**), d'autres sont des outils (*e.g.*, **stdenv**) ou encore des paquets (*e.g.*, **gtk3**). Le corps de la fonction commence par la définition d'une variable **toolkit** prenant la valeur **gtk3**, **gtk2** ou **lucid** en fonction des paramètres.

À la suite des deux clés **name** et **src**, la clé **buildInputs** liste toutes les dépendances du paquet. Les crochets permettent de délimiter une liste tandis que l'opérateur **++** concatène deux listes. Les 6 lignes spécifiant la valeur de la clé **buildInputs** forment une expression permettant de spécifier les dépendances en fonction des paramètres du paquet. Les dépendances spécifiées ici sont les dépendances nécessaires à toutes les étapes jusqu'à la création du dossier dédié dans **/nix/store**. Les dépendances nécessaires à l'exécution (par exemple les bibliothèques dynamiques) forment un sous-ensemble de ces dépendances et sont calculées automatiquement (voir plus haut).

En pratique, chaque utilisateur maintient un fichier Nix listant tous les paquets qu'il souhaite dans son environnement. Nous venons de faire un très rapide tour d'horizon de Nix. N'hésitez pas à continuer la découverte en lisant les documents référencés à la fin de cet article.

2. NIXOS, LE SYSTÈME D'EXPLOITATION

NixOS est une distribution GNU/Linux qui utilise Nix comme gestionnaire de paquets **[4]**. NixOS pousse le paradigme "fonctionnel pur" de Nix au niveau du système : sous NixOS, le système déployé est le résultat d'une construction de zéro (*from scratch*) à partir d'une description du résultat souhaité plutôt que le résultat de changements successifs **[5]**. En pratique, l'administrateur décrit son système dans un fichier **configuration.nix** contenant entre autres la liste des utilisateurs du système ainsi que la liste des services à installer (e.g., **nginx** et **openssh**) et leurs configurations.

2.1 Extraits du fichier configuration.nix

Dans la suite, je présente des bouts de mon fichier **configuration.nix**. Ce fichier commence en général par une définition du cœur du système (en partie générée automatiquement lors de l'installation de NixOS). Voici la configuration du chiffrement de mon disque, du noyau Linux et des points de montage :

```
 let wm = "gnome"; # définition d'une variable
in { [...]
boot = {
  cleanTmpDir = true; # delete all files in /tmp
  initrd.luks.devices = [ {
    device = "/dev/nvme0n1p5"; # encrypted partition
    name = "nixroot";
    preLVM = true;
  } ];
  kernelPackages = pkgs.linuxPackages_latest;
# default is LTS
};
fileSystems = { # the file systems to be mounted
  "/".device = "/dev/disk/by-label/nixroot";
  "/boot".device = "/dev/disk/by-label/SYSTEM";
  "/home".device = "/dev/disk/by-label/home"; };
```

configuration.nix permet aussi de configurer quelques paramètres généraux, comme le réseau, les consoles virtuelles et le fuseau horaire. On voit que je demande à NixOS de choisir le gestionnaire réseau en fonction du gestionnaire de fenêtres spécifié par une variable **wm** en début de fichier.

```
 networking = {
  hostName = "luz4";
  networkmanager.enable = (wm != "enlightenment");
# use networkmanager in all cases except with
  connman.enable = ! networking.networkmanager.
enable; # Enlightenment which prefers connman
};
i18n.consoleKeyMap = "us-acentos"; # keyboard
mapping table
time.timeZone = "Europe/Paris";
virtualisation.virtualbox.host.enable = true;
# install virtualbox
fonts.fonts = with pkgs; [ cantarell_fonts
corefonts dejavu_fonts [...] ];
```

Sous NixOS, il n'y a (quasiment) rien dans **/etc**. Les fichiers de configuration des différents services (par exemple **Apache**) sont générés automatiquement à partir de **configuration.nix**. Ils sont placés dans **/nix/store** et sont passés en paramètres aux services au démarrage.

```
 services =
  emacs = {
    enable = true; # automatically starts Emacs
daemon
    defaultEditor = true; # set the EDITOR
environment variable to emacsclient
  };
  locate = { # locate will update its database
everyday during lunch
    enable = true;
    interval = "11:59"; };
  openssh.enable = true; # ssh server
  xserver = {
    enable = true;
    layout     = "us"; # keyboard
    xkbVariant = "altgr-intl";
    libinput.enable = true; # use libinput
    config = '' # This part will be copied as is to
Xorg.conf
      Section "InputClass"
        Identifier     "Enable libinput for
TrackPoint" [...]
      EndSection'';
    displayManager.gdm.enable = wm == "gnome";
# only use GDM when Gnome is chosen
    desktopManager = {
      enlightenment.enable = wm == "enlightenment";
      gnome3.enable = wm == "gnome";
    }; }; };
```

Les utilisateurs et groupes peuvent aussi être gérés par NixOS :

```
 users.extraUsers.cassou = { # Add "cassou" as a
user
  home = "/home/cassou";
  description = "Damien Cassou";
  hashedPassword = "$6$O[...]kU0";
  extraGroups = ["audio" "vboxusers" "wheel"];
};
```

2.2 Avantages d'un système comme NixOS

Une fois que le fichier **configuration.nix** est prêt, l'administrateur doit lancer la commande **nixos-rebuild** ayant pour but la construction de l'environnement système. Alors que Nix maintient une liste ordonnée de tous les environnements utilisateurs pour pouvoir revenir en arrière, NixOS maintient une liste ordonnée de tous les environnements systèmes et fournit une commande pour revenir à l'état précédent du système. De plus, après une modification, si le système ne démarre plus, il est toujours possible de revenir à une version précédente de l'environnement grâce à un menu s'affichant au sein de grub (ou équivalent) et montrant toutes les versions (voir figure 1).

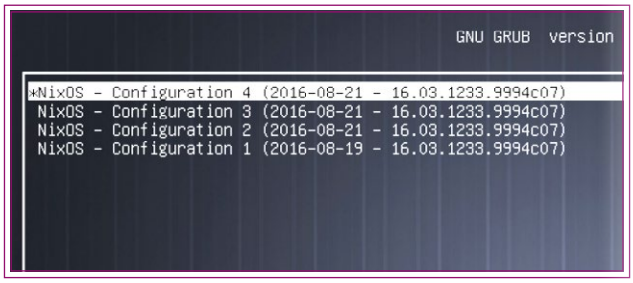


Fig. 1 : NixOS propose, au démarrage, de revenir à un état précédent du système.

Un autre avantage d'utiliser NixOS est que toute la configuration du système est explicitée dans un fichier écrit en Nix. Configurer une nouvelle machine (pour migration d'une machine ou pour l'établissement d'un parc dans le cloud) revient donc à copier ce fichier sur la nouvelle machine et à lancer **nixos-rebuild**. Comme Nix est un langage de programmation, il est possible de paramétrer le fichier de configuration de façon à avoir des variations entre machines (par exemple, le nom réseau de la machine).

NixOS contient une batterie de tests automatiques vérifiant que le système fonctionne. Chaque test est décrit dans un fichier Nix séparé. Il y a par exemple un test qui vérifie que la couche réseau fonctionne en lançant deux machines virtuelles sous NixOS et les faisant communiquer. Il y a aussi un test automatique, plus anecdotique, vérifiant qu'un serveur **Quake 3** sous NixOS peut être lancé avec des bots et deux joueurs. Le test vérifie que les joueurs ont bien rejoint la partie.

2.3 Inconvénient

Puisque le gestionnaire de paquets Nix ne touche à rien dans **/usr** ou **/lib** et que NixOS utilise uniquement Nix pour la gestion des paquets, le dossier **/usr** d'une machine sous NixOS contient uniquement **/usr/bin/env** et **/lib** n'existe même pas. Une des conséquences gênantes de cela est que les binaires téléchargés depuis Internet ne peuvent jamais être exécutés directement sous NixOS : ils font en général au moins référence à **/lib/ld-linux.so.2**. Heureusement, Nix fournit des outils pour réécrire automatiquement un binaire téléchargé (**patchelf**). En fait, c'est grâce à ces outils que Nix peut proposer des logiciels propriétaires (comme **Skype**).

CONCLUSION

Au-delà de Nix et NixOS, l'écosystème est vraiment riche. **nixos-hardware** est un dépôt Git en construction regroupant des bouts de fichiers `configuration.nix` pour différents matériels afin de faciliter la configuration de NixOS sur ces matériels. `nix-shell` est une commande shell qui place, à partir d'une description d'un paquet, l'utilisateur dans un shell dans lequel toutes les dépendances du paquet sont accessibles : cela permet d'avoir un environnement de développement différent pour chaque projet. **Hydra** est un serveur d'intégration continue basé sur Nix et dont les tâches du serveur sont décrites en Nix. NixOps et **Disnix** sont des outils facilitant la configuration et le déploiement de machines réelles et virtuelles (y compris dans le cloud) sous NixOS. Les deux outils se basent sur des fichiers Nix décrivant le déploiement souhaité.

J'utilise NixOS et donc Nix exclusivement depuis plus de 2 ans maintenant, sur deux ordinateurs différents. Configurer NixOS revient uniquement à modifier `configuration.nix` et c'est un vrai plaisir au quotidien. De plus, la communauté autour de Nix est très active : les paquets se mettent à jour rapidement (et si votre paquet préféré n'est pas à jour, une *pull request* de 2 lignes suffit en général) ; si vous avez un problème, il y a toujours quelqu'un sur IRC pour expliquer et aider ; il y a des dizaines de commits par jour, etc.

D'un autre côté, la communauté est encore petite et il faudra souvent mettre les mains dans le cambouis pour que vos applications préférées soient aussi bien empaquetées qu'ailleurs. Il arrive en effet souvent qu'une dépendance optionnelle d'un paquet n'ait pas été ajoutée à la description ou qu'il manque une option de compilation utile. Venez nous aider à transformer un excellent outil en un outil incontournable ! ■

RÉFÉRENCES

[1] Site officiel de Nix : http://nixos.org/nix/

[2] Calcul des dépendances : http://lethalman.blogspot.se/2014/08/nix-pill-9-automatic-runtime.html

[3] Définition d'Emacs 25 : https://github.com/NixOS/nixpkgs/blob/master/pkgs/applications/editors/emacs-25/

[4] Site officiel de NixOS http://nixos.org/nixos/

[5] Discussion sur les gestionnaires de systèmes : https://blog.flyingcircus.io/2016/05/06/thoughts-on-systems-management-methods/

DÉPLOIEMENT CONTINU À L'ÉCHELLE AVEC OPENSHIFT

NICOLAS DORDET
[Consultant OpenShift @Red Hat France]

ROMAIN PELISSE
[Sustain Developer @Red Hat Gmbh]

MOTS-CLÉS : DÉPLOIEMENT CONTINU, OPENSHIFT, CLOUD, DOCKER, KUBERNETES

Construire un environnement de livraison continu à la fois dynamique, optimisé et capable de tenir la charge (sans faire monter la facture plus que nécessaire) : c'est possible ! Et même facile à l'aide d'OpenShift ! Voyons rapidement comment mettre cela en place...

L'une des caractéristiques d'**OpenShift** est de permettre de réunir différents types d'environnements au sein de la même infrastructure, afin de conserver une séparation claire des usages, mais aussi de faciliter l'évolution des applications au sein de ces environnements. Ceci facilite grandement la gestion du système d'information et garantit l'homogénéité des systèmes.

Le cycle de vie des applications est ainsi couvert dans son intégralité et OpenShift se charge de gérer le réseau, les volumes, la sécurité, les mises à jour de l'application (sans arrêt de service), le passage à l'échelle ou encore le placement de nos applications pour optimiser les ressources des serveurs sous-jacents. Le tout, avec le minimum d'actions manuelles.

Avant de se lancer dans l'installation de ce produit, présentons-le en quelques mots.

1. FONDAMENTAUX D'OPENSHIFT (V3)

OpenShift est un projet relativement complexe, nous n'allons donc pas rentrer ici dans les détails de son fonctionnement interne, mais plutôt couvrir, en quelques lignes, les éléments d'architecture nécessaires à la bonne compréhension de cet article.

Essentiellement, OpenShift est un PaaS (*Platform-as-a-Service*) construit autour des produits **Docker [1]** et **Kubernetes [2]**. Il facilite donc l'utilisation de conteneurs et leur orchestration au sein de son « cluster ».

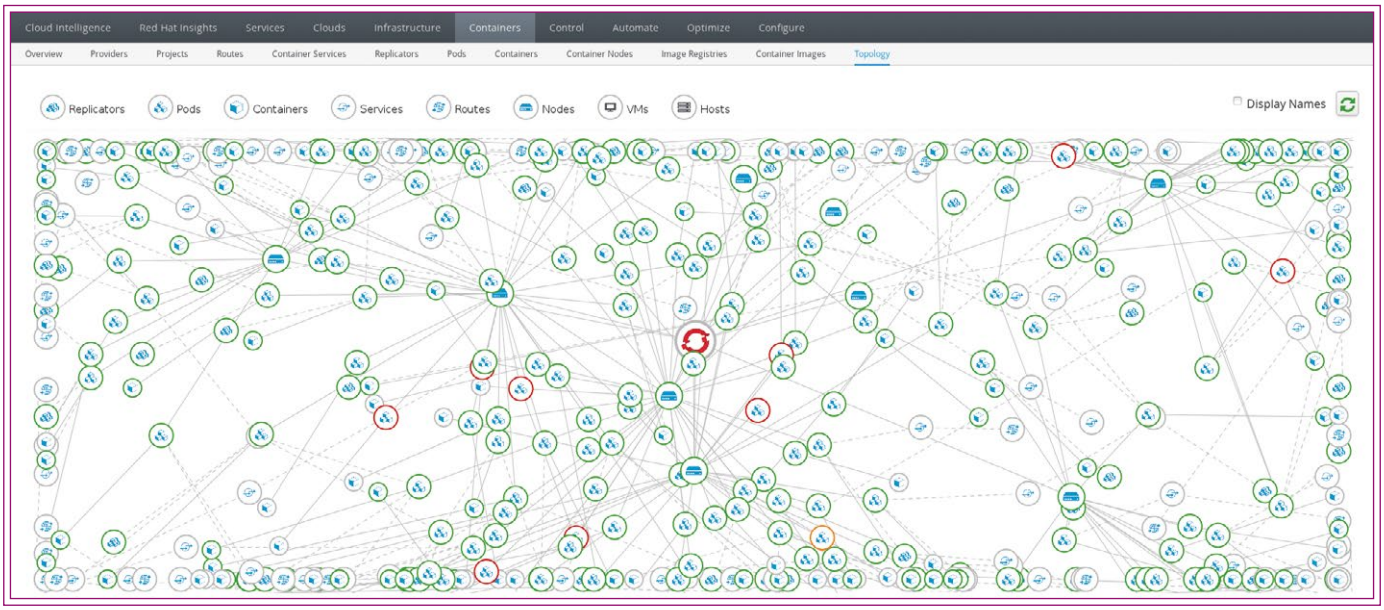


Fig. 1.

Tout ceci est loin d'être trivial, surtout que dans la plupart des environnements d'exécution, le nombre de conteneurs (et les dépendances entre eux) peut rapidement rendre la gestion difficile. La figure 1 montre par exemple la topologie d'un petit cloud.

Il est donc nécessaire de disposer d'outils capables de se charger de la gestion du stockage, des réseaux, etc. Et c'est justement le rôle de Kubernetes au sein de l'infrastructure fournie par OpenShift.

En effet, l'orchestration effectuée par Kubernetes s'occupe de toute la politique de création, de placement, et de destruction de nos conteneurs. OpenShift vient ajouter des éléments sur l'authentification, la construction d'image ou encore la stratégie de déploiement. Les opérationnels peuvent ainsi fournir un environnement de travail aux développeurs, testeurs, ou tout autre intervenant jusqu'à la production, tout en facilitant leur collaboration au travers d'un même outil et des mêmes technologies et sans avoir à gérer dans le détail la tuyauterie interne.

Afin d'avoir une vision assez globale du produit, ce schéma d'architecture, issu de la documentation officielle d'OpenShift **[3],** est souvent présenté comme en figure 2 (voir page suivante).

Dans cet article, l'objectif sera de créer une infrastructure permettant à notre serveur d'intégration continue, **Jenkins [4]**, d'effectuer ses tâches d'intégration, de construction et de test. Chaque lancement d'un job Jenkins (unité de tâches lancées et contrôlées par Jenkins) bénéficie de son propre environnement dédié, tout en exploitant autant que possible toutes les ressources à disposition ; sans pour autant surcharger le système et en libérant les ressources lorsqu'elles ne sont plus utilisées.

Notes sur l'installation de l'environnement

Les manières d'installer OpenShift sont multiples (installeur spécifique, Ansible, Puppet, installation entièrement manuelle, ou encore de manière éphémère dans un conteneur avec la commande `oc cluster up`). Notons donc au passage qu'OpenShift peut être installé en tant que service sur un environnement Linux classique via rpm, mais aussi sous forme de conteneur Docker.

Par souci de facilité, et surtout pour permettre au lecteur d'effectuer simplement l'installation sur son propre système, nous avons opté pour l'installation à l'aide de Vagrant [5] et de Virtualbox [6]. Cette approche a aussi l'avantage d'être clairement documentée et compatible avec de nombreux systèmes.

La première étape consiste donc en l'installation de l'image vagrant :

```
$ vagrant init openshift/origin-all-in-one
$ vagrant up --provider virtualbox
$ vagrant ssh
$ curl https://10.2.2.2:8443/console/ # pour vérifier qu'OpenShift est démarré
```

Néanmoins, en attendant le téléchargement de l'image, et avant de passer à la configuration de notre plateforme, revenons un peu sur la notion de chaîne de production au sein d'OpenShift, et voyons comment nous l'envisageons dans le cadre de cet article.

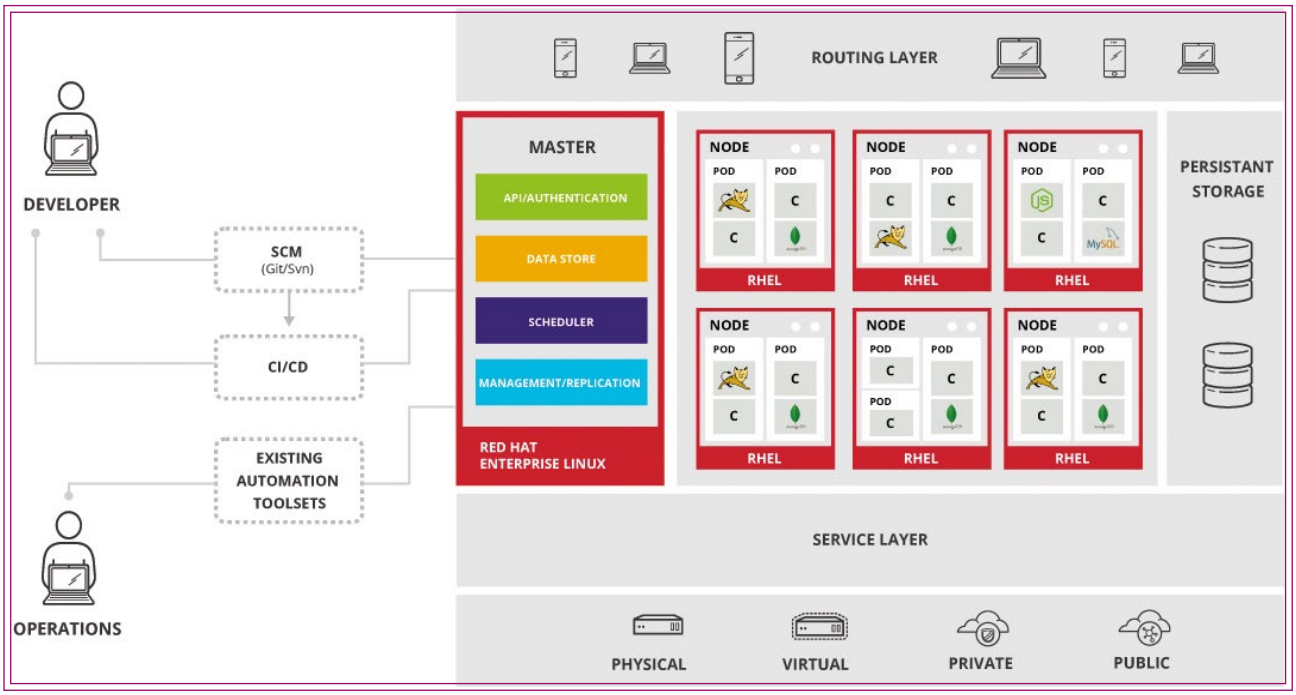


Fig. 2.

2. UN PEU DE THÉORIE...

2.1 Mise en place d'une chaîne de production

Dans OpenShift, il existe de nombreuses stratégies d'intégration et de manières de construire son *pipeline* au travers de différents environnements. Chacune d'elle peut être réalisée manuellement par l'utilisation de commandes OpenShift et Docker, de manière automatique en utilisant les capacités internes d'OpenShift ou bien par l'intermédiaire d'un système d'intégration continue comme Jenkins (qui peut être déployé au sein d'OpenShift ou être utilisé en tant que service externe si Jenkins est déjà installé sur un autre serveur).

Dans notre cas, nous mettrons tout d'abord en place la construction unique d'une image (code source et Docker) dans l'environnement de développement. Cette image sera ensuite automatiquement promue vers les autres environnements (test et production) - si la construction et les tests sont passés sans encombre.

On notera qu'ici, la promotion se fera à l'aide du système de tag offert par Docker et d'un déclencheur de déploiement d'images fourni par OpenShift. Enfin, Jenkins viendra orchestrer le tout avec la possibilité d'ajouter des séries de tests dans notre chaîne d'intégration et de déploiement continu.

2.2 Processus

2.2.1 Récupération du code source

Le code de notre application est stocké sur **GitHub [8]**. Il s'agit d'une application exemple très utilisée par la communauté **JBoss**, « kitchensink ». Cette application permet de faire la démonstration de la plupart des fonctionnalités Java/EE supportées par JBoss AS. Dans notre cas, l'application a été adaptée pour être déployée dans un conteneur.

2.2.2 Construction de l'image

Au sein de notre processus de livraison continue, la chaîne de publication sera déclenchée automatiquement lors de l'ajout de changement de code sur ce dépôt. Ainsi, à chaque publication de changements dans le projet sur GitHub, le code est recompilé, l'application est reconstruite en entier et une nouvelle image Docker est créée dans l'environnement de développement.

NOTE

On notera aussi au passage que le type de projet est analysé par OpenShift, celui-ci va donc pouvoir instancier une image Docker adaptée aux besoins de l'application. Dans notre cas, l'image doit disposer de Java et du serveur d'application JBoss, car les deux sont nécessaires à l'exécution de l'application. Un conteneur avec ces éléments sera donc instancié par OpenShift.

INNOROBO EVENT

16-18 MAI 2017

PARIS FRANCE

WHERE INNOVATION GROWS

Plongez au **cœur** de l'**innovation robotique**

Chercheurs, créateurs et utilisateurs vous dévoilent le monde de demain

UNE APPROCHE HUMAINE DE LA ROBOTIQUE

SMART HOMES

INDUSTRY 4.0 & SUPPLY CHAIN SERVICES

MEDICAL & HEALTH

TECHNOLOGIES & FORESIGHT

SMART CITIES

FIELD ROBOTICS

HOSPITALITY RETAIL & TOURISM

www.**innorobo**.com

YouTube theinnorobo

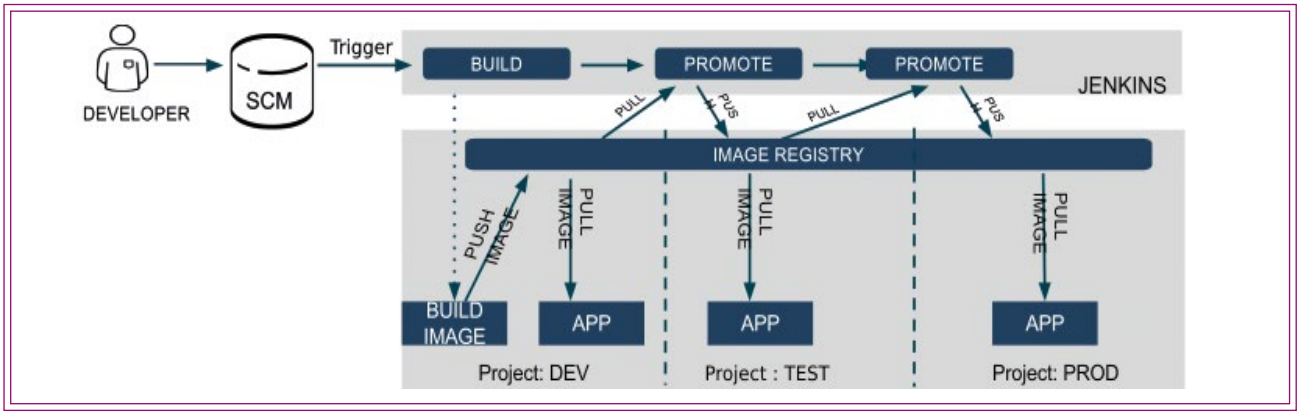


Fig. 3.

2.2.3 Déploiement de l'application et étiquetage de son image

Après une construction sans encombre - et d'éventuels tests unitaires d'intégration passés avec succès ou encore une analyse qualité du code - le déploiement est automatiquement déclenché sur un conteneur, et suivi de la validation du bon fonctionnement de l'application. Après cette étape, des tests supplémentaires peuvent être lancés (performance, fonctionnel, sécurité, ...), pour s'assurer en détail de la conformité de l'application. Enfin, on associe l'image à un label (« tag ») afin de la promouvoir (c'est-à-dire de la mettre à disposition pour l'environnement suivant).

2.2.4 Passage à l'environnement suivant

Grâce à la présence d'une nouvelle image associée au label précédemment cité, OpenShift va déclencher la récupération automatique de cette image dans le second environnement de notre application. Une série de tests pourra aussi être effectuée avant que l'image soit à nouveau promue, si tout se passe bien, et donc mise à disposition pour l'environnement suivant.

Le flux de travail de notre système est défini en figure 3.

Dans ce processus, nous faisons en sorte que l'image du conteneur créée en phase de développement puisse être délivrée en production si et seulement si tous les tests et vérifications sont passés avec succès. Il garantit aussi que les systèmes testés et en production sont strictement les mêmes, et ce sans intervention manuelle.

Du point de vue de notre serveur d'intégration continue Jenkins, notre chaîne de build est présentée en figure 4.

2.3 Construction d'image avec S2I et déploiement OpenShift

Pour parfaire notre approche, nous allons utiliser l'outil : **Source-to-Image** (**S2I**) **[9].** Ce dernier permet de construire des images Docker, et sa particularité réside dans la séparation qu'il fait entre l'**image de base** du conteneur et le **code source** des applications. Lors de la construction, il va en effet injecter le code source de l'application et les dépendances nécessaires à son exécution, à l'image de base – dans notre cas, une image contenant le serveur d'application Java/JEE Wildfly – afin de créer une nouvelle image Docker, prête à être déployée.

Pour construire l'application, nous utilisons l'outil de construction de logiciel, **Maven [11]**. Par défaut, voici comment celui-ci est invoqué par S2I :

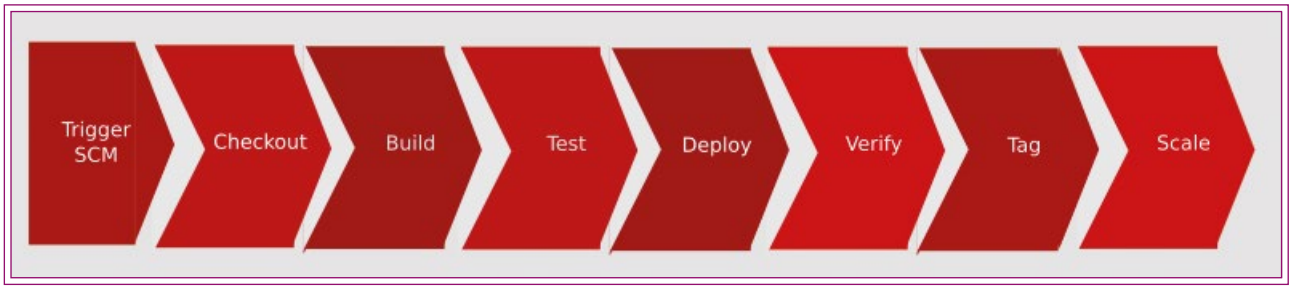


Fig. 4.

```
$ mvn package dependency:copy-dependencies -Popenshift
-DskipTests -e
```

Dans notre application, nous avons simplement ajouté au fichier **pom** le bloc suivant :

```
     <profile>
         <!-- When built in OpenShift the 'openshift'
profile will be used when invoking mvn. -->
         <!-- Use this profile for any OpenShift specific
customization your app will need. -->
         <!-- By default that is to put the resulting
archive into the 'deployments' folder. -->
         <!-- http://maven.apache.org/guides/mini/guide-
building-for-different-environments.html -->
         <id>openshift</id>
         <build>
            <plugins>
               <plugin>
                  <artifactId>maven-war-plugin</artifactId>
                  <version>2.1.1</version>
                  <configuration>
                     <outputDirectory>deployments
</outputDirectory>
                     <warName>ROOT</warName>
                  </configuration>
               </plugin>
            </plugins>
         </build>
      </profile>
```

Une fois la construction de l'image terminée, celle-ci sera poussée dans la « registry » interne d'OpenShift. Ce dernier **va détecter que l'image de notre application a changé et va lancer un nouveau déploiement.**

3. MISE EN PRATIQUE DANS OPENSHIFT

3.1 Espace de nommage et projets

Comme souvent dans notre industrie, chaque groupe d'utilisateur et les technologies qu'il emploie développent leur propre sémantique - chacune d'entre elles étant valide de leur point de vue, entraînant souvent et malheureusement une certaine confusion. La notion d'espace de nommage (*namespace*) et de *project* au sein d'OpenShift en est certainement l'une des victimes.

Il faut donc bien comprendre qu'un *project* dans OpenShift est en fait basé (avec quelques éléments supplémentaires) sur le concept de namespace de Linux. D'autre part, les projets que vous déployez sont appelés application. On notera donc aussi que l'on peut avoir la même *application* dans plusieurs *projects*.

On va créer ici trois *projects*, c'est-à-dire trois environnements d'exécution pour notre application (ou encore trois *namespaces*) :

```
$ oadm new-project dev --display-name="Development"
$ oadm new-project test --display-name="Testing"
$ oadm new-project prod --display-name="Production"
```

3.2 Mise en place de l'application dans le project dev

Nous allons créer une nouvelle application à partir de notre dépôt Git. OpenShift va récupérer le code source du dépôt distant, puis il va déterminer la stratégie de construction à adopter (« source », « docker » ou « custom »).

Dans notre cas, ce sera la stratégie « source » qui sera invoquée, puisqu'il s'agit de cloner le projet et de le construire depuis ses sources sur GitHub avant de les déployer sur son conteneur grâce à S2I.

Comme vu plus haut, OpenShift va déterminer l'image Docker la plus appropriée à l'exécution de notre application à l'aide d'une analyse du fichier `pom.xml` du projet. Néanmoins, nous lui précisons tout de même le nom de l'image référence (« stream ») que nous allons utiliser.

```
$ oc project dev
$ oc new-app https://github.com/
ndox/kitchensink-example.git -n
dev --image-stream="openshift/
wildfly"
```

NOTE

Avant d'aborder la partie réseau et le déploiement du service, il est important de noter qu'OpenShift utilise le concept, de Pod propre à Kubernetes. Un Pod est défini comme objet contenant un ou plusieurs conteneurs déployés sur le même hôte. C'est grâce à cet objet que Kubernetes peut contrôler le cycle de vie des conteneurs, assigner une adresse IP, un volume ou encore gérer les autorisations d'actions sur les conteneurs.

Au passage, on remarquera que c'est la plus petite entité qui peut être manipulée au sein du « cluster ». Dans la suite de l'article, on utilisera donc le terme de Pod pour désigner notre conteneur.

À l'issue de l'exécution de ces commandes, l'application est donc construite une première fois et déployée dans l'espace de développement. Nous avons donc notre conteneur Docker WildFly contenant notre application Kitchensink déployée dans un « Pod ».

3.3 Exposition du service

Kubernetes s'assure que chaque Pod reçoive une adresse IP et soit capable de communiquer avec le cluster. OpenShift déploie quant à lui un SDN (*Software Defined Networking*) pour connecter les Pods. Ceci donne un réseau virtuel, mais commun, à l'ensemble des Pods, comme s'ils s'exécutaient tous sur le même système.

L'avantage de cette approche est qu'elle permet de traiter chaque Pod comme un hôte physique ou une machine virtuelle en terme d'allocation des ports, de réseau, de nommage, de la balance de charge... Tout en permettant une communication entre eux facile à mettre en place au besoin.

Finalement, pour pouvoir accéder à notre application et étant donné que la gestion du réseau est gérée par Openshift, il suffit simplement d'exposer le service via la commande :

```
$ oc expose service kitchensink-example -n dev
```

Puis de vérifier sa disponibilité avec la commande (ou bien ouvrir L'URL dans un butineur) :

```
$ curl http://kitchensink-example-dev.apps.10.2.2.2.xip.io/index.xhtml
```

3.4 Permission inter-environnement

OpenShift utilise une politique d'autorisation à base de règles et de rôles. Ils permettent d'autoriser différents niveaux d'accès, que ce soit à l'échelle d'un projet ou du cluster. OpenShift utilise également le concept de compte de service pour effectuer certaines opérations.

Pour passer une image d'un environnement à un autre, il est donc nécessaire d'accorder au compte de service de l'environnement cible le droit de de "puller" (c'est-à-dire récupérer) des images sur l'environnement de départ :

```
$ oc policy add-role-to-group system:image-puller system:serviceaccounts:prod -n dev
$ oc policy add-role-to-group system:image-puller system:serviceaccounts:test -n dev
```

3.5 Association d'une étiquette

La politique d'intégration que nous mettons en place est la suivante : l'application et l'image Docker ne seront assemblées qu'une seule fois dans l'environnement de développement. Dans les environnements test et production, ce sera cette image avec un tag spécifique qui sera utilisée.

Pour cela, nous allons d'ores et déjà appliquer un premier « tag » afin de pouvoir déployer une première fois l'image dans les environnements de test et production :

```
$ oc tag kitchensink-example:latest kitchensink-example:promote-qa -n dev;
$ oc tag kitchensink-example:latest kitchensink-example:promote-prod -n dev;
```

3.6 Mise en place de l'application dans les environnements de test et production

L'instanciation de l'application exemple, kitchensink, dans les environnements test et production se fait à partir des images taguées précédemment. On va donc créer une nouvelle application en lui précisant que la source est l'image contenant un tag particulier :

```
$ oc new-app dev/kitchensink-example:promote-qa -n test
$ oc new-app dev/kitchensink-example:promote-prod -n prod
```

Pour se rapprocher d'un modèle plus réel, on va mettre à l'échelle l'application à l'aide de 5 « Pods » dans l'environnement de production. Ceci va assurer la disponibilité de l'application (puisqu'il y aura au final 5 instances disponibles de notre application) :

```
$ oc scale dc/kitchensink-example --replicas=5  -n prod
```

Et on finit par exposer les services :

```
$ oc expose service kitchensink-example -n test
$ oc expose service kitchensink-example -n prod
```

Pour vérifier la bonne installation des environnements, il est possible de lancer les commandes suivantes :

```
$ oc get pods -n dev
$ oc get pods -n test
$ oc get pods -n prod
```

Ou pour surveiller tous les « Pods » du cluster :

```
$ oc get pods --all-
namespaces -w
```

3.7 Désactivation du déclencheur « ImageChange »

Par défaut, lors de la création d'une nouvelle application dans OpenShift, un déclencheur de déploiement est activé. À chaque modification de l'image source, celui-ci déclenche le déploiement de la nouvelle image. Pour utiliser le déploiement uniquement via Jenkins, il faut le désactiver.

Dans notre exemple, nous allons donc le désactiver uniquement pour les projets test et prod (car le *project* dev utilise lui cette fonctionnalité pour se mettre à jour automatiquement depuis le dépôt Git) :

```
$ oc get dc kitchensink-
example -o yaml -n test |
sed 's/automatic: true/
automatic: false/g' | oc
replace -f - ;
$ oc get dc kitchensink-
example -o yaml -n prod |
sed 's/automatic: true/
automatic: false/g' | oc
replace -f - ;
```

4. MISE EN PLACE DU DÉPLOIEMENT CONTINU

4.1 Installation de jenkins

Nous allons créer un projet « ci » puis instancier une image de Jenkins. Pour nous faciliter la tâche, il existe déjà un patron (*template*) Jenkins disponible dans OpenShift.

Nous allons ensuite donner le droit d'édition à l'utilisateur **serviceaccount** du projet « ci », sur les projets dev, test et prod :

```
$ oc new-project ci
$ oc new-app jenkins-persistent
$ oc policy add-role-to-user edit system:serviceaccount:ci:default
-n dev;
$ oc policy add-role-to-user edit system:serviceaccount:ci:default
-n test;
$ oc policy add-role-to-user edit system:serviceaccount:ci:default
-n prod;
```

4.2 Création du job

Jenkins, vient avec un « job » exemple configuré. Il est possible de le copier pour avoir un flux de travail cohérent avec OpenShift, puis l'adapter à ses besoins. Nous avons mis à disposition la configuration du « job » au sein de notre projet exemple. Pour le mettre en place, il suffit d'exécuter les commandes suivantes :

```
$ wget https://raw.githubusercontent.com/ndox/kitchensink-example/
master/settings/Kitchen-JobConfig.xml
$ curl -X POST -H "Content-Type:application/xml" -d @Kitchen-
JobConfig.xml https://jenkins-ci.apps.10.2.2.2.xip.io/
createItem?name=kitchensink-example --insecure --user admin:password
```

4.3 Lancement du pipeline

Vous pouvez faire une copie (*fork*) du projet GitHub et effectuer la configuration avec votre dépôt d'un webhook, afin de déclencher le lancement de la construction en effectuant une modification d'un des fichiers de votre dépôt. Cette procédure est clairement décrite dans la documentation officielle **[16]**.

Mais si vous préférez une approche plus « graphique », vous pouvez simplement cliquer sur le bouton ***Build Now*** dans l'interface Jenkins :)

Une fois le processus lancé, puis lorsque le conteneur est finalement déployé en « production », lancez la commande :

```
$ oc describe is kitchensink-example -n dev
Name:      kitchensink-example
Created:     44 hours ago
Labels:                  app=kitchensink-example
Annotations:    OpenShift.io/generated-by=OpenShiftNewApp
Docker Pull Spec:    172.30.118.161:5000/dev/kitchensink-example

Tag     Spec      Created           PullSpec          Image
latest  <pushed>  43 seconds ago    172.30.118.161:5000/dev/
kitchensink-example@sha256:d5b46d4965c56d...    <same>
promote-prod    kitchensink-example:latest    43 seconds ago
172.30.118.161:5000/dev/kitchensink-example@sha256:d5b46d4965c56d...
<same>
promote-qa    kitchensink-example:latest    43 seconds ago
172.30.118.161:5000/dev/kitchensink-example@sha256:d5b46d4965c56d...
<same>
```

Vous pouvez remarquer que les « hashtags » des images étiquetées « promote-qa » et « promote-prod » sont les mêmes que ceux de la dernière image « latest ».

On a donc bien eu la construction de l'image dans l'espace de développement qui a été promue puis déployée, en test puis en prod.

5. OPTIMISATION DES RESSOURCES

5.1 Service Jenkins dynamique

Notre système d'intégration et de promotion inter-environnements est maintenant fonctionnel. Cependant, si le nombre de constructions de notre application augmente ou si d'autres applications sont ajoutées, notre serveur Jenkins risque de ne pas tenir la charge.

Il est possible de configurer OpenShift pour que notre nombre d'instances de Jenkins augmente en fonction des besoins. Jenkins offre aussi la possibilité de créer des esclaves afin de répartir la charge et de ne pas congestionner l'instance maître. Nous allons donc mettre en œuvre ce « cluster » dont les membres esclaves seront automatiquement créés puis détruits au besoin. Il sera piloté par notre instance maître, ce qui facilitera notamment la gestion.

5.2 Installation et configuration du plugin kubernetes pour jenkins

Le plugin Kubernetes va nous permettre de rendre un peu plus dynamique notre système d'intégration continue et tirer pleinement parti des possibilités d'OpenShift. En effet, à chaque lancement d'un « build » sur notre projet, Kubernetes va créer un Pod contenant un esclave Jenkins, sur lequel s'exécutera le « build » du projet, puis le détruira à la fin de la construction.

L'utilisation des ressources est donc optimisée, sans risquer de surcharger l'instance maître de Jenkins. On aura donc une solution d'intégration continue dynamique et automatiquement scalable au besoin !

Sur la page de configuration générale de Jenkins dans ***cloud provider***, configurer comme suit :

```
Kuberntes URL : https://172.30.0.1:443 <= oc get svc --all-namespaces |
grep kube
Disable https certificate check coché
Credentials : Ajouter un login / mot de passe d'un administrateur
OpenShift
Jenkins URL : http://172.30.248.4:8080 <= oc get svc -n ci | grep jenkins
```

Dans le job de notre projet exemple **kitchensink-example**, nous rajoutons les éléments suivants afin de n'autoriser les builds que sur notre Pod labellisé maven (donc pas d'exécution sur le maître) :

```
Label Expression : maven
```

Finalement, dans OpenShift, on va ajouter un port à notre service Jenkins, afin qu'il puisse communiquer avec ses esclaves :

```
oc edit service jenkins
```

Ou via la console web d'OpenShift : https://10.2.2.2:8443/console/project/ci/browse/services/jenkins - cliquer sur ***edit yaml***.

Dans la sections des « ports », on va rajouter ceci :

```
-
  name: jnlp
  protocol: TCP
  port: 50000
  targetPort: 50000
```

Il suffit maintenant de relancer un « build » dans Jenkins (soit en modifiant le code source dans GitHub, soit en cliquant sur le bouton ***build*** du projet dans Jenkins) et de regarder dans la console d'OpenShift. Par exemple, on peut y voir la création du Pod esclave, l'exécution du « build », le déploiement de la nouvelle version de notre application et enfin la destruction du Pod esclave.

Via l'utilitaire **Cockpit [12]** qui est préinstallé dans la VM All-in-one **[13]** : on peut voir graphiquement et en temps réel la création du Pod esclave « maven » qui nous sert à effectuer le build de notre application, puis sa disparition (voir figures 5 et 6).

CONCLUSION

Comme cet article l'a illustré, du moins nous l'espérons, OpenShift prend en charge une « plomberie » assez conséquente (gestion de la sécurité, réseaux, ressources, déploiement...), à l'aide de technologies ouvertes et open source (Docker, Kubernetes, Jenkins...) et ce de manière automatique. Une fois l'infrastructure mise en place, il ne reste donc qu'à fournir les applicatifs sous format de conteneur Docker, et à configurer leur processus d'intégration et leur stratégie de déploiement. Les efforts fournis par les équipes de développement d'OpenShift vont en grandissant sur ces aspects de construction, de gestion de cycle de vie et d'intégration. Dans les

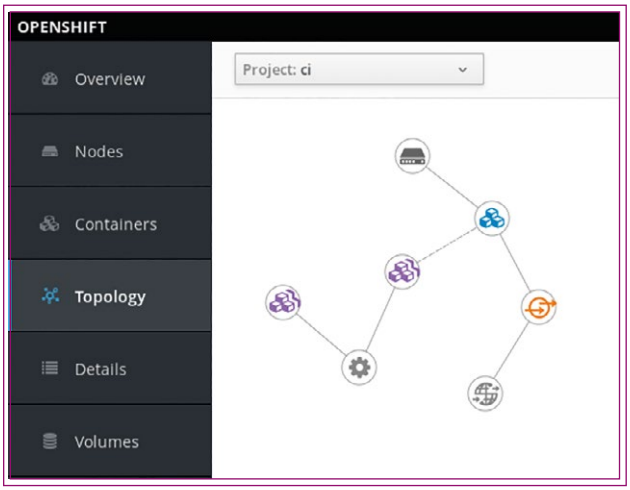


Fig. 5.

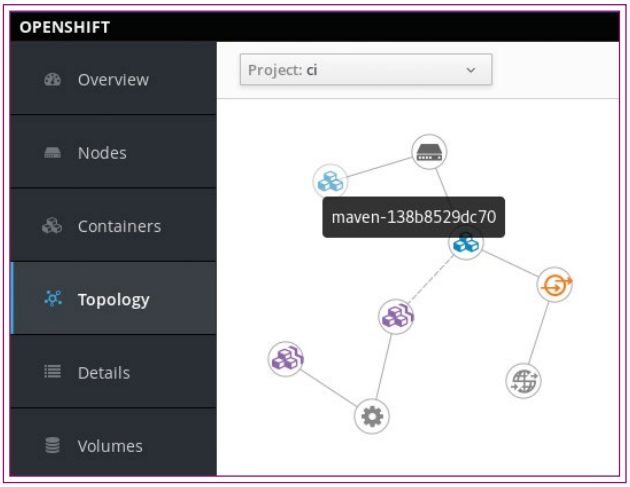


Fig. 6.

prochaines versions, le pilotage de Jenkins via OpenShift va être amélioré ainsi que l'utilisation des esclaves ou encore la configuration automatique des jobs Jenkins.

En outre, en tant qu'offre PaaS, OpenShift permet facilement de mettre en place une approche DevOps. En effet, les développeurs auront la mainmise sur le conteneur de l'applicatif, et auront la garantie que c'est ce dernier qui a été promu jusqu'à la production, sans changement non suivi. De leur côté, les opérationnels disposent de toute l'infrastructure mise en place par OpenShift pour gérer les systèmes. Les deux équipes travaillant sur la même plateforme avec les mêmes technologies, la cohésion et la collaboration sont facilitées.

Pour finir, nous conclurons sur le fait qu'OpenShift est aussi disponible sous forme de produit Red Hat, « OpenShift Enterprise »**[14]** (comme Red Hat Enterprise Linux) et en est déjà à sa troisième version, mais aussi sous forme de service disponible à la demande avec « OpenShift Online »**[15]** (pratique pour essayer OpenShift sans infrastructure à sa disposition).

Docker est encore une technologie récente, et donc prompte à des changements soudains et parfois radicaux. La stabilité du projet ou sa rétrocompatibilité avec des versions précédentes sont donc des points importants à surveiller. Surtout qu'il s'agit, au final, d'une partie conséquente de l'infrastructure de production qui restera longtemps en place et nécessitera une mise à niveau régulière, si ce n'est que pour des raisons de sécurité. Red Hat et la communauté OpenShift font beaucoup d'efforts pour fournir des versions stables et sécurisées des composants embarqués dans OpenShift. Ce dernier est ainsi l'outil idéal pour faciliter l'adoption des conteneurs en production. ■

RÉFÉRENCES

[1] Site officiel de Docker : https://www.docker.com/

[2] Site officiel de Kubernetes : http://kubernetes.io/

[3] Schéma d'architecture d'OpenShift : https://docs.openshift.com/enterprise/3.2/architecture/index.html

[4] Site officiel de Jenkins : https://jenkins.io/

[5] Vagrant : https://www.vagrantup.com/

[6] VirtualBox : https://www.virtualbox.org/

[7] All-In-One Virtual Machine : https://www.openshift.org/vm/

[8] Notre application « Kitchensink » : https://github.com/ndox/kitchensink-example/

[9] Source-to-Image (S2I) : https://github.com/openshift/source-to-image

[10] All-In-One Virtual Machine : https://www.openshift.org/vm/

[11] Maven : https://maven.apache.org/

[12] Cockpit : http://cockpit-project.org/

[13] Cockpit avec All-In-One VM : http://cockpit.apps.10.2.2.2.xip.io/kubernetes#/topology/ci

[14] OpenShift Enterprise : https://www.openshift.com

[15] OpenShift Enterprise « Online » : https://www.openshift.com

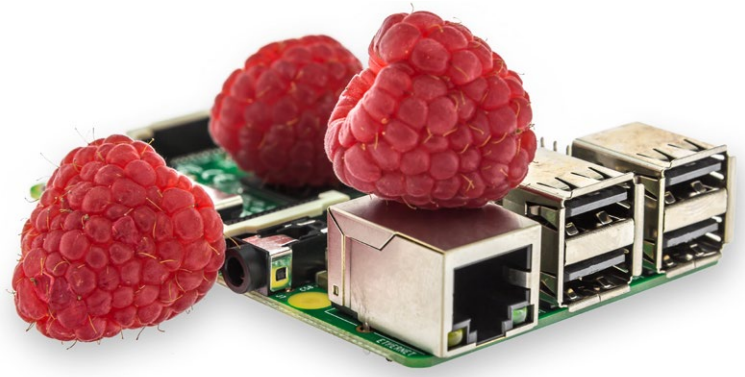
[16] OpenShift « Web Hooks » : https://docs.openshift.org/latest/dev_guide/migrating_applications/web_hooks_action_hooks.html#dev-guide-migrating-applications-webhooks

PROGRAMMATION EMBARQUÉE SUR RASPBERRY PI
SANS SONDE JTAG

CHRISTOPHE PLÉ
[CEO de Farjump et Expert en développement logiciel embarqué]

JULIO GUERRA
[CTO de Farjump]

MOTS-CLÉS : RASPBERRY PI, BARE-METAL, GCC, GDB



Le standard JTAG, au succès indéniable, est aujourd'hui ancré dans la majorité des processeurs et proposé comme moyen privilégié de programmation embarquée et de debug. Toutefois, l'utilisation d'une sonde JTAG n'est en rien triviale. L'utilisation du débuggeur GNU s'impose alors, car il est possible de l'employer en bare-metal, c'est-à-dire sans aucun système d'exploitation embarqué pour gérer le processeur et sa carte. À titre d'illustration, nous nous intéressons dans cet article à la programmation bare-metal d'une Raspberry Pi à l'aide de GDB uniquement.

En situation de développement embarqué, la **Raspberry Pi** (RPi) est appelée « système cible » (*target*), tandis que le poste de travail est appelé « système hôte » (*host*). L'hôte doit donc programmer le système cible à distance (*remote*) via un moyen de communication dédié. À noter également que nous sommes dans une situation de développement croisé : la RPi, *bare-metal* et architecture ARM, ne correspond pas au système hôte, couramment muni d'une architecture Intel avec un système d'exploitation (OS). Il convient donc d'utiliser une chaîne d'outils croisés pour produire et manipuler un programme cible depuis un système hôte.

Pour ce faire, nous déploierons d'abord sur le poste de travail la chaîne d'outils GNU croisés, `arm-none-eabi`, incluant le compilateur **GCC** et le débuggeur **GDB**, puis embarquerons le serveur **GDB freemium Alpha** directement sur la RPi, et l'utiliserons comme

moyen de développement afin de développer et d'embarquer, avec la seule aide de GDB, une série d'exemples de programmes **C** standards allant du simple *Hello World* sur la sortie standard, au plus avancé *raytracer* utilisant le processeur graphique (GPU) pour produire des images sur la sortie HDMI de la RPi. Nous finirons enfin par embarquer le *raytracer* directement sur la RPi, pour qu'il soit démarré par son *bootloader*, comme si nous passions en production une fois le développement terminé.

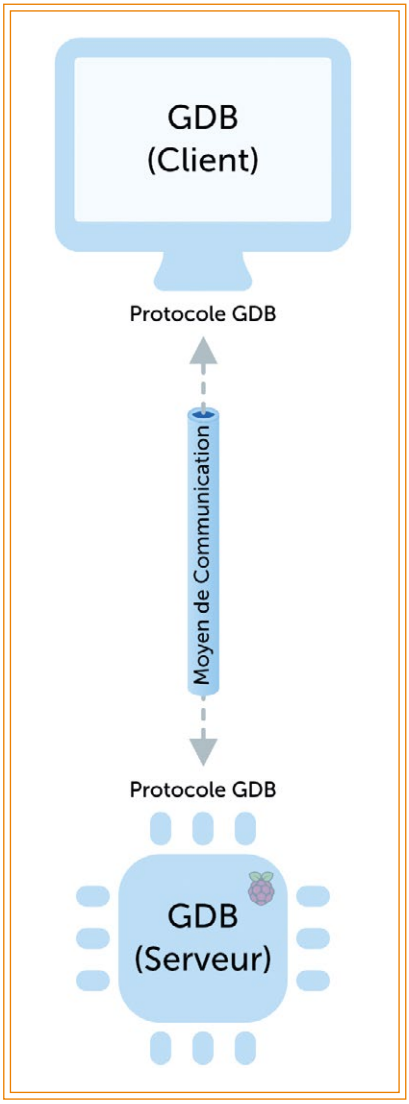


Fig. 1 : Vue générale du développement embarqué à l'aide de GDB.

1. INSTALLATION

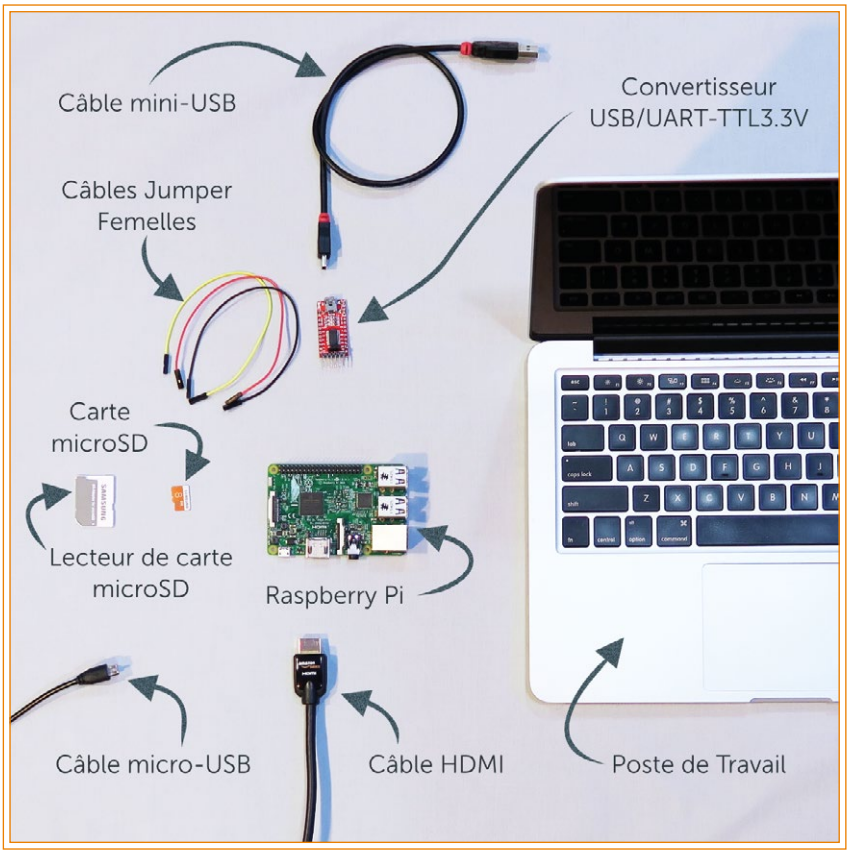


Fig. 2 : Matériel complet nécessaire : une RPi, un poste de travail, une carte microSD pour la RPi, un lecteur de carte microSD pour le poste de travail, un câble HDMI pour la RPi, un câble micro-USB pour alimenter la RPi, un convertisseur USB/UART-TTL3.3V, trois câbles jumper femelle pour le branchement convertisseur/RPi, un câble mini-USB pour le branchement convertisseur/poste de travail.

1.1 Poste de travail

Un poste de travail POSIX (**GNU/Linux**, **Cygwin**, **OS X**...) est nécessaire. Il exécutera la compilation croisée ARM pour RPi, ainsi que le client GDB. Les droits de lecture et écriture sont également nécessaires sur l'interface de communication employée et décrite plus bas.

Un dépôt git est mis à disposition : il contient les éléments nécessaires pour la suite (et pour échanger sur https://github.com/farjump/raspberry-pi en cas de problème). Les commandes ci-dessous permettent de le cloner :

```
$ git clone https://github.com/farjump/raspberry-pi.git glmf-rpi
$ cd glmf-rpi
glmf-rpi/ $ git checkout v1.0.0
glmf-rpi/ $ ls
LICENSE  Makefile  boot/  run.gdb  scripts/  sdk/  src/
```

Optionnellement, les plus modernes d'entre nous apprécieront certainement la commande **make shell** qui produit directement le poste de travail attendu dans un container docker basé sur **Debian 9**, puis y lance un terminal dans lequel peut s'exécuter de manière garantie la suite de l'article.

1.1.1 Chaîne de compilation croisée ARM

ARM distribue la chaîne de compilation croisée pour les principaux systèmes d'exploitation au format 64 bits uniquement **[1]**. Le script **scripts/install-toolchain.sh**, fourni dans le dépôt, télécharge et décompresse la version Linux 64 bits de la chaîne :

```
glmf-rpi/ $ sudo ./scripts/
install-toolchain.sh --prefix /
opt/glmf-arm-none-eabi
[+] Installing ARM cross-compiler
into '/opt/glmf-arm-none-eabi'
[+] The toolchain has been
successfully installed
glmf-rpi/ $ PATH=/opt/glmf-arm-
none-eabi/bin:$PATH
glmf-rpi/ $ export PATH
glmf-rpi/ $ arm-none-eabi-gcc -v
&& arm-none-eabi-gdb -v && echo
mini test ok
...
mini test ok
```

Compte tenu du succès de l'architecture ARM, les distributions les plus courantes (Debian, **Archlinux**, **Fedora**, etc.) la mettent aussi directement à disposition dans leurs gestionnaires de paquets, toujours en partie nommée avec le triplet **arm-none-eabi**. Pour OS X et Windows, il convient d'adapter les étapes précédentes à la distribution officielle ARM **[1]** en la déployant dans votre environnement.

1.2 Raspberry Pi

1.2.1 Carte microSD

La RPi démarre le programme qu'elle trouve sur sa carte microSD suivant les directives contenues dans le fichier de configuration **config.txt**. Le serveur GDB Alpha doit donc être copié sur la carte microSD pour être démarré à l'allumage de la RPi.

L'étape décrite ici simplifie au maximum la préparation de la carte microSD pour la RPi et son bootloader. Elle suppose une carte microSD vierge compatible avec la RPi. Un script d'aide à son installation est fourni, mais l'étape de formatage en FAT32, attendu par le firmware et le bootloader de la RPi, est laissée explicite afin d'insister sur le caractère irréversible de cette opération destructrice pour la carte microSD. Enfin, le script **scripts/install-rpi-boot.sh** installe le firmware, le bootloader et Alpha sur la carte microSD :

```
glmf-rpi/ $ # 1. Insérer la carte microSD dans le lecteur de carte du
poste de travail
glmf-rpi/ $ # 2. Trouver son point de montage
glmf-rpi/ $ dmesg -H | tail
[Feb14 13:37] sd 3:0:0:0: [sdc] 30318592 512-byte logical blocks:
(15.5 GB/14.5 GiB)
[  +0.026020]  sdc:
glmf-rpi/ $ # La carte microSD est ici apparue sur le nœud /dev/sdc
glmf-rpi/ $ # 3. Formater la carte microSD en FAT32
glmf-rpi/ $ # Attention, cette étape supprime entièrement le contenu
de la carte microSD
glmf-rpi/ $ sudo mkfs.vfat -F 32 -n RPI -I /dev/sdc
glmf-rpi/ $ # 4. Utiliser le script fourni, installant les fichiers
sur la carte formatée
glmf-rpi/ $ ./scripts/install-rpi-boot.sh /dev/sdc
[+] Downloading the Raspberry Pi's firmware version 1.20161215
############################################################ 100.0%
############################################################ 100.0%
[+] Temporarily mounting '/dev/sdc' into '/tmp/rpi-sdcard-mountpoint'
[+] Installing the RPi firmware and the Alpha debugger
'boot/bootcode.bin' -> '/tmp/rpi-sdcard-mountpoint/bootcode.bin'
'boot/start.elf' -> '/tmp/rpi-sdcard-mountpoint/start.elf'
'boot/Alpha.bin' -> '/tmp/rpi-sdcard-mountpoint/Alpha.bin'
'boot/config.txt' -> '/tmp/rpi-sdcard-mountpoint/config.txt'
[+] Checking the integrity
/tmp/rpi-sdcard-mountpoint/bootcode.bin: OK
/tmp/rpi-sdcard-mountpoint/start.elf: OK
/tmp/rpi-sdcard-mountpoint/Alpha.bin: OK
[+] Un-mounting '/tmp/rpi-sdcard-mountpoint'
[+] Your SD card is ready!
[+] You can now insert it into the RPi and use Alpha through the
RPI's Mini-UART
```

La carte microSD est alors prête à l'emploi et la RPi y trouvera tout le nécessaire pour démarrer le serveur GDB. À son prochain démarrage, le firmware, ayant pour tâche d'initialiser le processeur et sa carte, lira les directives contenues dans le fichier **config.txt** indiquant que le programme **Alpha.bin** doit être copié à l'adresse d'exécution **0x7F0_8000**, point d'entrée du serveur.

1.2.2 Interface de communication GDB

Une fois le serveur GDB en cours d'exécution, celui-ci attend la connexion d'un client GDB et ses commandes, en écoutant une interface de communication. Le serveur GDB Alpha utilise l'interface série Mini-UART, car commune à toutes les versions de la RPi. Le port de cette interface n'étant pas disponible tel quel sur un poste de travail standard, il est nécessaire d'interposer un convertisseur entre les deux.

Du coté poste de travail, les ports séries sont les ports USB (et RS232 pour les plus anciens). Du côté RPi, l'interface Mini-UART, présente sur le connecteur GPIO (figure 3), se décompose en trois broches TTL 3.3 Volts. Il est donc nécessaire d'utiliser un convertisseur UART TTL 3.3V vers USB. Attention à bien utiliser un modèle 3.3V.

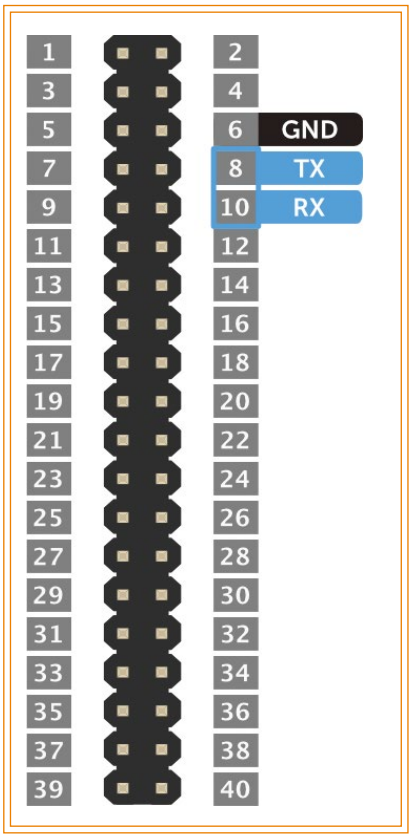


Fig. 3 : Les connecteurs TTL 3.3V de l'interface Mini-UART sont toujours les mêmes pour toutes les RPi.

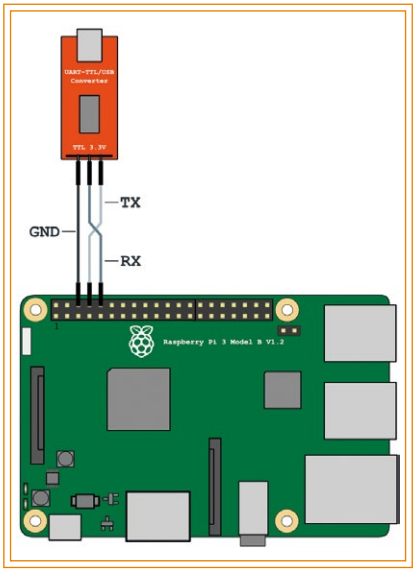


Fig. 4 : Schéma de câblage entre la RPi 3 et un convertisseur : connexion de la masse et croisement entre les fils d'émission et réception.

Une fois relié au poste de travail par un câble USB, le convertisseur est géré par l'OS et son driver de périphériques série, dont les points de montage possibles sont **`/dev/ttyUSBx`** ou bien **`/dev/ttyACMx`** pour Linux, **`COMx`** pour Windows, ou encore **`/dev/cu.usbserial-xxxxxx`** pour OS X. Pour la suite de cet article, **`/dev/ttyUSB0`** désignera notre port série de communication GDB.

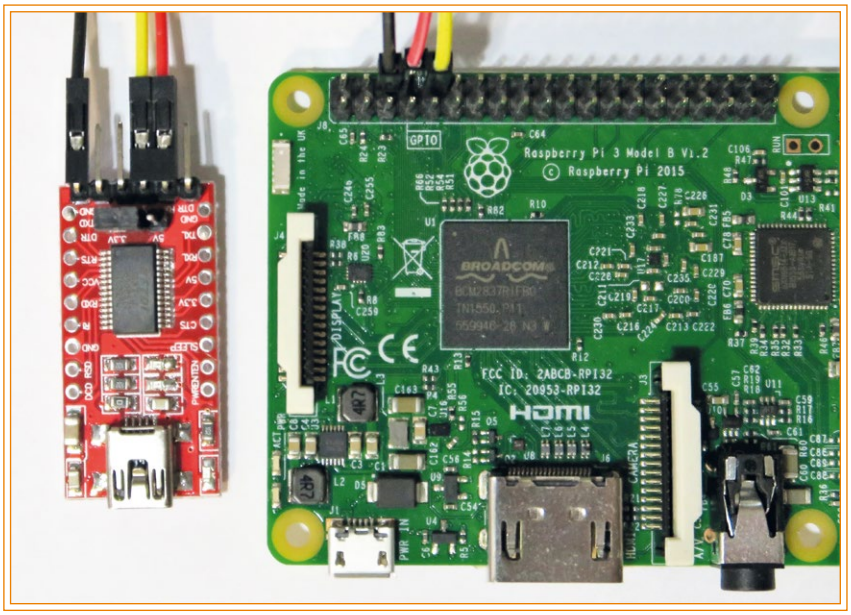


Fig. 5 : Exemple de câblage d'une RPi 3 avec un convertisseur série UART TTL-3.3V vers USB.

Le coût total du montage avoisine les 9€ et nécessite l'achat d'un convertisseur **[2]**, de trois câbles jupper femelle **[3]** ainsi que d'un câble USB **[4]**. Des montages complets, couramment nommés « câbles TTL », plus compacts et incluant les câbles, sont aussi disponibles, mais à des prix beaucoup plus aléatoires **[5]**.

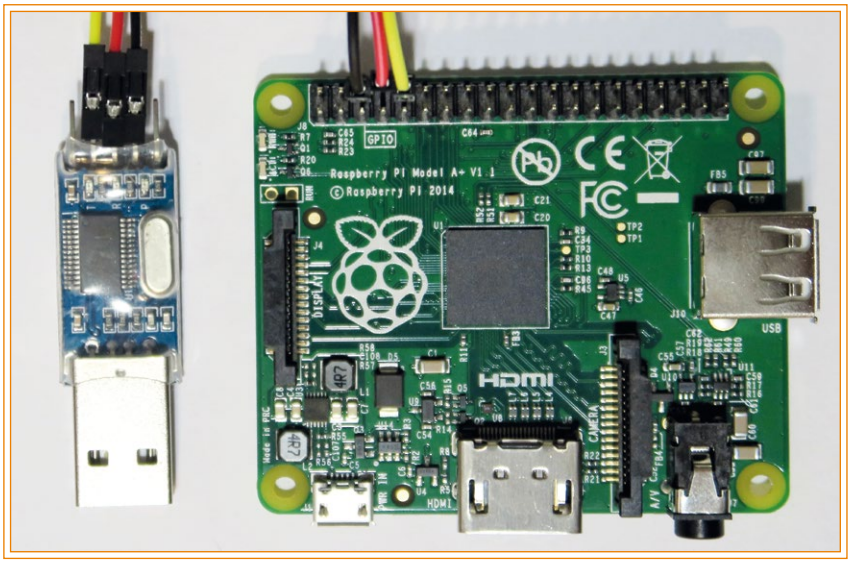


Fig. 6 : Exemple de câblage d'une RPi 1 A+ avec un convertisseur série UART TTL3.3V vers USB.

2. ÉDITER, COMPILER & DÉBUGGER

Nous sommes désormais prêts à utiliser GDB comme moyen de développement embarqué. Mais contrairement à une utilisation classique de GDB, dite « native », où le programme à exécuter et débugger est au préalable préparé par le système d'exploitation, il est nécessaire ici d'effectuer explicitement ces mêmes étapes préliminaires : téléchargement du programme sur la cible une fois la connexion distante établie. Le mode client/serveur de GDB offre en effet au serveur la possibilité de supporter la commande **load** qui prend alors directement en charge le format d'exécutable ELF et en télécharge les sections de code et de données sur la cible, puis place finalement le pointeur sur instruction sur son point d'entrée. Les sections de debug sont quant à elles lues et chargées par le client GDB et lui permettent d'apporter les fonctionnalités de debug de code et de données depuis le code source (*source-level debugging*).

Une session de debug d'un logiciel embarqué commence donc souvent par :

```
$ arm-none-eabi-gdb <fichier ELF avec informations de debug>
(gdb) # 1. Configuration de l'interface de communication et
connexion à la cible
(gdb) target remote <interface>
(gdb) # 2. Téléchargement du programme sur la cible
(gdb) load
(gdb) # Prêt!
```

Il est ensuite possible de profiter des fonctionnalités élémentaires de debug de code et de données. Les fonctionnalités plus avancées dépendent quant à elles du serveur. Par exemple, la version freemium d'Alpha, le serveur GDB utilisé, n'implémente pas les *tracepoints* **[6]**, mais implémente l'extension File I/O **[7]** (dont l'utilité sera révélée par la suite). GDB signalera au final son incapacité à exécuter une commande si le serveur n'en est pas capable.

NOTE

GDB est une solution d'instrumentation dynamique, par opposition à l'instrumentation statique qui se fait à la compilation du code source en le modifiant. GDB n'altère donc pas le programme et utilise les ressources de debug du processeur afin d'arrêter et d'observer l'exécution du programme.

De plus, Alpha met en place un environnement d'exécution permettant un développement *bare-metal* plus simple et dont le programme embarqué via GDB peut alors profiter. Alpha initialise en effet le processeur plus vastement que le bootloader de la RPi en activant, par exemple, l'unité flottante ou encore en programmant un espace mémoire (figure 7). Ceci doit donc être pris en considération lorsque GDB n'est plus utilisé pour charger le programme, qui doit alors lui-même effectuer les initialisations dont il a besoin. Il en est de même pour toutes les fonctionnalités nécessaires et en dehors du périmètre de cet environnement d'exécution.

Enfin, il ne sera pas nécessaire de redémarrer manuellement la RPi entre chaque session de debug puisqu'Alpha effectue un reset du processeur lorsque GDB lui transmet la commande **kill**. Cette commande est aussi induite par toutes celles impliquant, en debug natif, l'arrêt du processus, et notamment celle de connexion distante **target remote <interface>**, ainsi que la commande **quit** lorsque GDB est quitté. Relancer l'un des scripts GDB qui suit ou bien

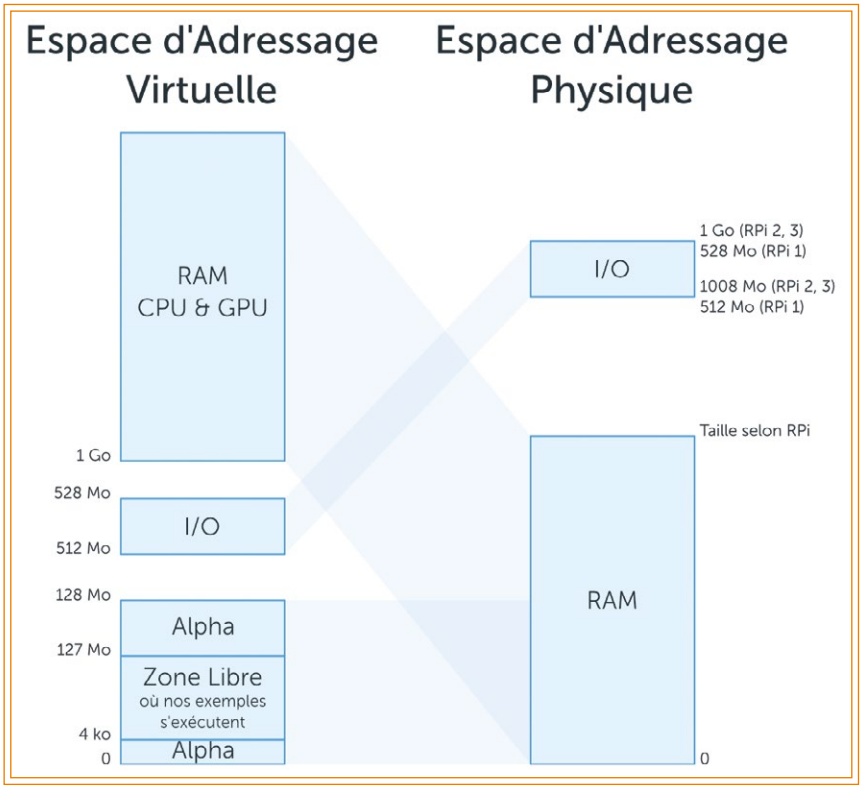


Fig. 7 : Mapping mémoire initialisé par Alpha dont le programme hérite.

quitter GDB (correctement) implique donc un reset de la RPi, permettant ainsi de repartir depuis l'état zéro du processeur et d'éviter les effets de bord indésirables d'une session à une autre. À noter que le reset est observable grâce aux LED situées sur la RPi.

Toutes les sessions de debug présentées par la suite utilisent exclusivement l'interface ligne de commandes de GDB (CLI). Il est toutefois conseillé de profiter de son interface texte, **TUI [8]**, et ce à tout moment, grâce à la commande `tui enable` (figure 8).

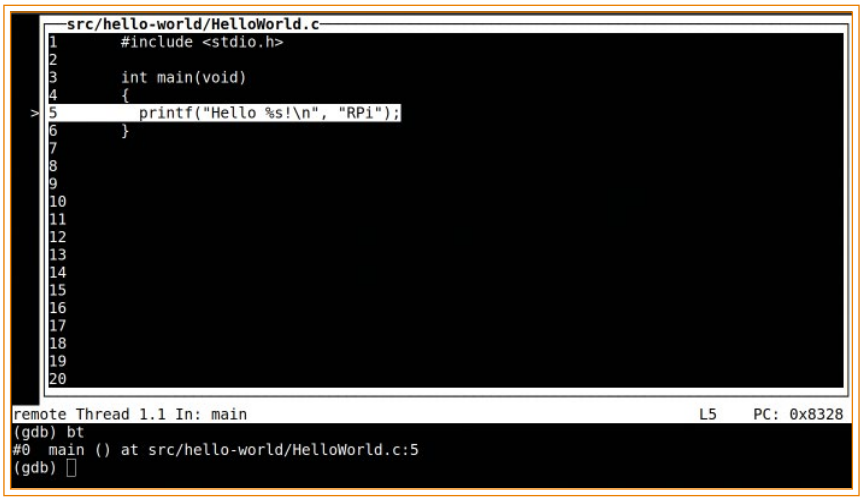


Fig. 8 : Interface utilisateur texte de GDB.

2.1 Hello World

2.1.1 Compilation

Soit le programme :

```
#include <stdio.h>
void main(void)
{
  printf("Hello RPi!\n");
}
```

Malgré les apparences, ce programme n'est en rien anodin. Il fait en effet appel à des fonctions de la bibliothèque C standard alors qu'elles nécessitent normalement un système d'exploitation, absent dans le cas présent. Il y a d'une part l'initialisation de l'environnement d'exécution (*runtime*) C préalable à l'appel de la fonction `main()` (initialisation de la pile et des données), et d'autre part l'écriture sur la sortie standard par `printf()`.

Pour ce faire, les fonctions et appels systèmes nécessaires sont implémentés dans la mesure du possible dans le contexte bare-metal. GDB bénéficie d'une fonctionnalité peu connue que le serveur GDB peut optionnellement implémenter pour transmettre et déléguer au client les appels système, alors exécutés sur le poste de travail sur lequel le client GDB s'exécute. Ainsi, `printf()` utilise l'appel système `write()` qui est communiqué et délégué au client tel quel, pour écrire donc finalement sur la sortie standard du client GDB, sur le poste de travail.

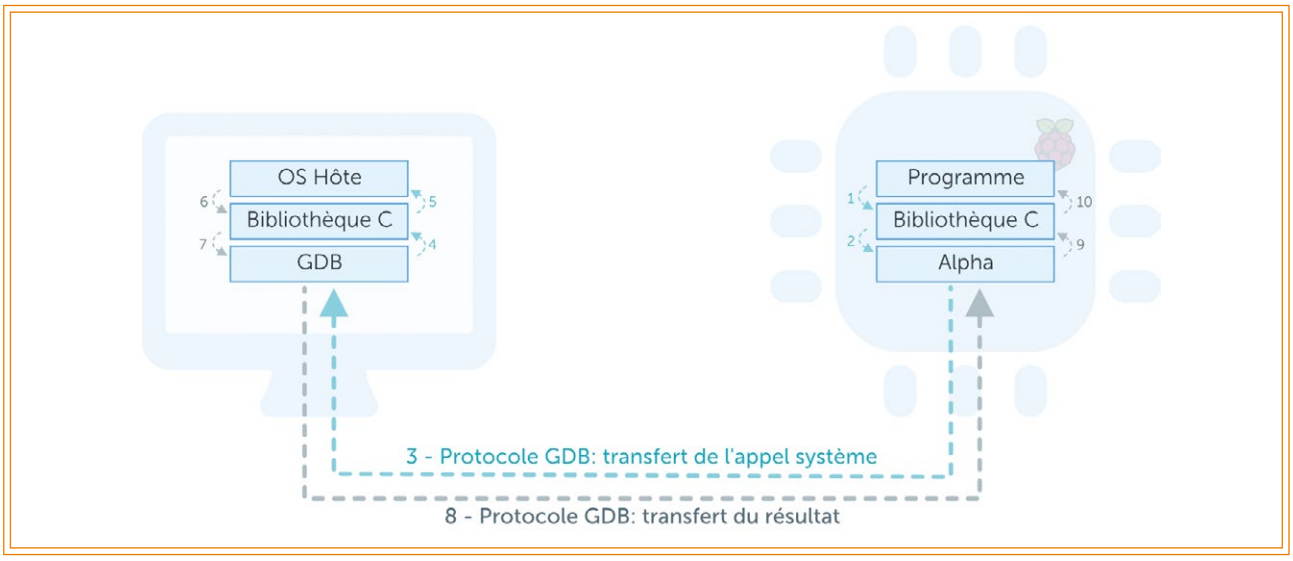


Fig. 9 : Un appel système effectué depuis la bibliothèque C sur la RPi est délégué au client GDB sur le système hôte.

Cette astuce n'est possible qu'avec un nombre restreint d'appels systèmes **[9]** dont les capacités sont parfois limitées par GDB (ex. : ouvrir un fichier spécial), tandis que d'autres pourront être implémentés très simplement sans aucune aide d'un système d'exploitation ou de GDB (ex. : **brk()** pour **malloc()**).

Cette fonctionnalité de GDB s'appelle *File I/O* **[7]** et est implémentée par Alpha. Nous l'interfaçons avec la bibliothèque C **newlib** en remplaçant ses appels systèmes par des appels à Alpha (figure 9). Cette bibliothèque C est la seule, avec la bibliothèque C GNU **glibc**, à s'intégrer officiellement dans la chaîne de compilation GCC, ce qui en rend l'utilisation aussi aisée qu'en compilation native : compiler un programme est aussi simple et direct que **arm-eabi-none-gcc <cppflags> <cflags> <ldflags> -o main.elf main.c <libs>**. Le résultat final est la possibilité d'embarquer n'importe quel programme C standard.

Pour compiler le programme :

```
glmf-rpi/ $ make hello.elf
arm-none-eabi-gcc -specs=sdk/Alpha.specs
-mfloat-abi=hard -mfpu=vfp -march=armv6zk
-mtune=arm1176jzf-s -g3 -ggdb -Wl,-Tsdk/link.
ld -Lsdk -Wl,-Map,hello.map -o hello.elf  src/
hello-world/HelloWorld.c
```

NOTE

Les options de compilation gcc présentes dans le Makefile sont précisément adaptées au processeur ARM11 de la première RPi : `-mfloat-abi=hard -mfpu=vfp -march=armv6zk -mtune=arm1176jzf-s`.

L'option **-gN** permet d'activer la génération des informations de debug du programme, où **N** est le niveau d'informations souhaité allant de **0** (désactivation) à **3** (maximum). L'option **-ggdb** permet quant à elle d'activer les extensions spécifiques à GDB.

2.1.2 GDB

```
glmf-rpi/ $ arm-none-eabi-gdb hello.elf
GNU gdb (GNU Tools for ARM Embedded Processors)
7.12.0.20161204-git
...
For help, type "help".
Type "apropos word" to search for commands
related to "word".
(gdb) # 1. Configuration GDB dépendante de Alpha
(gdb) source sdk/alpha.gdb
(gdb)
(gdb) # 2. Connexion a la RPi
(gdb) # Nécessite les droits de lecture/
écriture sur le TTY du lien série,
(gdb) # ici le nœud '/dev/ttyUSB0'
(gdb) set serial baud 115200
(gdb) target remote /dev/ttyUSB0
(gdb)
(gdb) # 3. Téléchargement sur la RPi du
programme passé en argument à GDB
(gdb) load
(gdb) # 'load' place aussi le pointeur
sur instruction sur son point
(gdb) # d'entrée '_start'
(gdb)
(gdb) # 4. Lancer alors l'exécution
jusqu'à atteindre la fonction main
(gdb) tbreak main
(gdb) continue
```

Il est conseillé d'écrire ces commandes, préalables à toute les prochaines sessions de debug, dans un fichier que GDB pourra alors lire et exécuter via l'option de lancement **-x <fichier>**, ou la commande **source <fichier>**. Le fichier **run.gdb**, dont le TTY doit être ajusté selon le cas, est à ce titre fourni à la racine du dépôt.

```
glmf-rpi/ $ arm-none-eabi-gdb -x run.gdb hello.
elf
...
Temporary breakpoint 1, main () at src/hello-
world/HelloWorld.c:5
5         printf("Hello %s!\n", "RPi");
(gdb) # Nous avons atteint la fonction main()
(gdb) list
1       #include <stdio.h>
2
3       int main(void)
4       {
5         printf("Hello RPi!\n");
6       }
(gdb) # Prêt pour une nouvelle session de debug
```

Ou encore :

```
glmf-rpi/ $ arm-none-eabi-gdb hello.elf
(gdb) source -v run.gdb
...
Temporary breakpoint 1, main () at src/hello-
world/HelloWorld.c:5
5         printf("Hello %s!\n", "RPi");
(gdb) # Nous avons atteint la fonction main()
(gdb) list
1       #include <stdio.h>
2
3       int main(void)
4       {
5         printf("Hello RPi!\n");
6       }
(gdb) # Prêt pour une nouvelle session de debug
```

Nous pouvons alors le laisser s'exécuter et observer la sortie standard sur la console GDB :

```
(gdb) continue
Continuing.
Hello RPi!

Program received signal SIGTRAP, Trace/
breakpoint trap.
_exit (rc=0) at SYSFILEIO/MAKEFILE/../
SOURCE/SYSFILEIO_EXIT.c:11
11      SYSFILEIO/MAKEFILE/../SOURCE/
SYSFILEIO_EXIT.c: No such file or directory.
(gdb)
```

Ou encore l'exécuter au pas-à-pas :

```
(gdb) source run.gdb
...
Temporary breakpoint 1, main () at src/hello-
world/HelloWorld.c:5
5         printf("Hello %s!\n", "RPi");
(gdb) break printf
Breakpoint 2 at 0x854c
(gdb) continue
Continuing.

Breakpoint 2, 0x0000854c in printf ()
(gdb) bt
#0  0x0000854c in printf ()
#1  0x00008334 in main () at src/hello-world/
HelloWorld.c:5
(gdb) finish
Run till exit from #0  0x0000854c in printf ()
Hello RPi!
0x00008334 in main () at src/hello-world/
HelloWorld.c:5
5         printf("Hello %s!\n", "RPi");
(gdb) next
6       }
(gdb) delete 2
(gdb) call printf("appel dynamique!\n")
appel dynamique!
$1 = 17
```

Les fonctionnalités de GDB se classent en deux grandes catégories : le debug de code et le debug de données. Entre autres, il est possible de visualiser la pile d'appel, de lire/écrire les données, les registres et la mémoire. Mais aussi d'insérer des *breakpoints*, d'intercepter des appels de fonctions à l'aide de *breakpoints* conditionnels...

Il est également possible d'envoyer des commandes spécifiques au serveur GDB via la commande **`monitor <commande>`**, lui permettant d'implémenter ses propres commandes. Par exemple, ici, la capture d'exceptions :

```
(gdb) monitor help
...
(gdb) monitor gdb/catch
   RST : no   : Reset Exception
   UND : no   : Undefined Instruction Exception
   SWI : no   : Software Interrupt Exception
 PABRT : no   : Prefetch Abort Exception
 DABRT : no   : Data Abort Exception
   IRQ : no   : IRQ (interrupt) Exception
  FIQR : no   : FIQ (fast interrupt) Exception
```

Cet exemple de programme constitue donc un point de départ idéal pour une exploration plus approfondie des possibilités de GDB et/ou du développement embarqué sur RPi. Le lecteur peut laisser libre cours à son imagination pour exploiter d'autres fonctions de la bibliothèque C (ex. : **`scanf()`**, **`fopen()`**...) ou en utilisant d'autres exemples de programmes C standards à embarquer sur la RPi en se reposant sur la bibliothèque C fournie (ex. : la runtime C++).

NOTE

Un programme se reposant sur l'extension File I/O ne peut pas s'exécuter sans client ou serveur GDB puisque ce sont eux qui échangent et exécutent les appels systèmes (figure 9). Cette fonctionnalité doit donc être considérée comme un moyen de développement supplémentaire dont les utilisateurs peuvent bénéficier ou non selon leur bon vouloir. À bon entendeur.

2.2 Raytracer

Cet exemple est une nouvelle illustration des concepts vus jusqu'à présent et appliqués cette fois-ci à un programme exploitant de manière plus avancée la RPi. Il utilise en effet son unité flottante ainsi que son GPU. Nous finirons avec sa « mise en production » sur la carte microSD de la RPi pour qu'il démarre de manière autonome, sans aucune intervention de GDB.

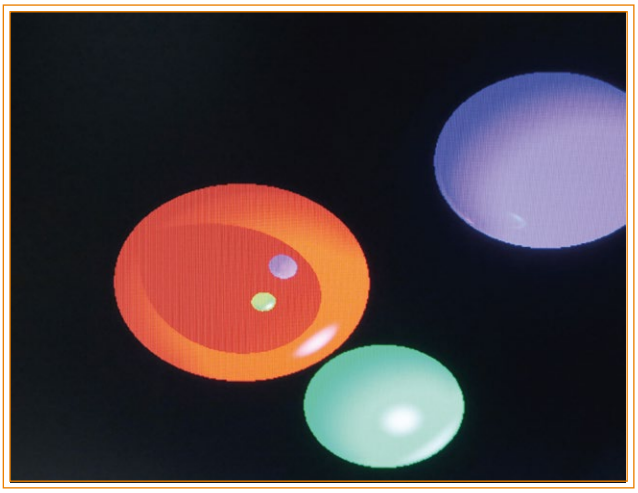


Fig. 10 : Image générée par le raytracer en bare-metal sur la sortie HDMI de la RPi.

2.2.1 Compilation

Ce raytracer affiche sur la sortie HDMI les images calculées par l'algorithme de *raytracing* (figure 10), contrôlé par le GPU, lui-même commandé par le processeur ARM à travers

des requêtes échangées par *mailbox* **[10]**. Le calcul se fait pixel par pixel et les résultats sont écrits dans le *framebuffer* alloué par le GPU. Le fichier **VC.c** contient le code source des fonctions d'affichage graphiques : une fonction pour demander au GPU de préparer un *framebuffer*, et une seconde fonction pour récupérer son adresse. Le framebuffer consiste alors en une matrice de pixels à écrire au format 32 bits ARGB.

Pour le compiler :

```
glmf-rpi/ $ make raytracer.elf
arm-none-eabi-gcc -specs=sdk/Alpha.specs
-mfloat-abi=hard -mfpu=vfp -march=armv6zk
-mtune=arm1176jzf-s -g3 -ggdb -Wl,-Tsdk/link.
ld -Lsdk -Wl,-umalloc -Wl,-Map,raytracer.map
-o raytracer.elf -Og src/raytracer/main.c src/
raytracer/Raytracing.c src/raytracer/VC.c src/
raytracer/VC_aligned_buffer.S -lm
```

Cet exemple introduit aussi le niveau d'optimisation **g** avec l'option **-Og** qui permet d'optimiser le programme tout en conservant une qualité de debug du programme acceptable. Elle évite donc les optimisations les plus agressives qui entraînent trop de perte de traçabilité entre le code objet et son code source. L'expérience de debug est quoi qu'il en soit dégradée (par exemple, impossibilité de lire certaines variables, exécution au pas-à-pas du code réordonnancé, etc.) par rapport à un programme compilé sans aucune optimisation.

2.2.2 GDB

Nous allons utiliser GDB afin d'observer le fonctionnement de l'algorithme de *raytracing* en employant les moyens de modification dynamique des données du programme.

Démarrer d'abord la session de debug :

```
glmf-rpi/ $ arm-none-eabi-gdb -x run.gdb raytracer.elf
...
Temporary breakpoint 1, main () at src/raytracer/
main.c:221
221     {
(gdb)
```

Les temps de traitement longs du *raytracer* et sa boucle infinie de rendu vidéo sont de parfaits candidats à la fonctionnalité d'interruption asynchrone de l'exécution de la cible, c'est-à-dire tandis qu'elle exécute le *raytracer*, lorsque le client reçoit le signal **<Ctrl>** + **<c>** (**SIGINT**). Cette fonctionnalité repose sur une interruption matérielle qui nécessite donc de démasquer les interruptions externes du processeur. Pour ce faire, nous utilisons les fonctionnalités de modification des registres ainsi que de scripting GDB, dont la syntaxe des expressions est identique à celle du langage C. Le registre en question est en plus l'un des registres superviseurs, absents du mode GDB « natif », mais ici communiqué par Alpha :

```
(gdb) # Démasquer les interruptions externes
pour rendre possible
(gdb) # l'interruption du programme via ctrl-c.
(gdb) print /x $cpsr &= ~(1 << 7)
$1 = 0x6000015f
```

Nous sommes désormais en mesure d'interrompre à tout moment l'exécution du *raytracer* sur la RPi :

```
(gdb) continue
Continuing.
^C
Program received signal SIGSTOP, Stopped
(signal).
0x0000921c in __ieee754_sqrt ()
```

NOTE

Les interruptions externes sont masquées par le point d'entrée `_start` tel qu'implémenté par la bibliothèque C `newlib`. Le démasquage doit donc intervenir une fois cette fonction terminée, lorsque nous atteignons la fonction `main()`.

Nous reprenons alors la main avec GDB où le programme a été interrompu. Voici donc un exemple modifiant successivement la couleur d'une sphère :

```
(gdb) set A_SPHERE[0].S_PROPERTY.S_A.F_
GREEN = 0.9
(gdb) continue
Continuing.
^C
Program received signal SIGSTOP, Stopped
(signal).
0x00008aa0 in RAYT_TRACE (...) at src/
raytracer/Raytracing.c:306
306         S_RGB.F_BLUE = (P_PROPERTY-
>S_A.F_BLUE * P_RAYT_WORLD->S_AMBIANT_
LIGHT.F_BLUE);

(gdb) bt
#0  0x00008aa0 in RAYT_TRACE (...) at
src/raytracer/Raytracing.c:306
#1  0x00008e24 in RAYT_RENDER (...) at
src/raytracer/Raytracing.c:421
#2  0x000083d0 in main () at src/
raytracer/main.c:239
(gdb) set A_SPHERE[0].S_PROPERTY.S_A.F_
GREEN = 0.3
(gdb) continue
Continuing.
^C
Program received signal SIGSTOP, Stopped
(signal).
0x00009080 in sqrt ()
```

```
(gdb) set A_SPHERE[0].S_PROPERTY.S_A.F_
GREEN = 0.6
(gdb) # Execution background (gdb) #
Permet de garder la main sur le client
tandis que le serveur la rend au
programme.
(gdb) continue &
Continuing.
(gdb) # Commande GDB équivalente au signal
ctrl-c
(gdb) interrupt
Program received signal SIGSTOP, Stopped
(signal).
0x00009240 in __ieee754_sqrt ()
(gdb) # Les variables telles que I_
RESOLUTION_FACTOR sont lues une seule fois
(gdb) # lors de l'initialisation du
programme. Il est donc nécessaire de
reprendre
(gdb) # depuis le début l'exécution du
programme pour pouvoir la modifier.
(gdb) print I_RESOLUTION_FACTOR
$2 = 2
(gdb) # Alpha reset le processeur
lorsqu'il reçoit la commande kill, aussi
induite
(gdb) # par d'autres commandes GDB. C'est
le cas notamment de la commande de
(gdb) # connexion contenue dans le script
run.gdb. Il suffit donc de le relancer
(gdb) # pour recommencer une session.
(gdb) source run.gdb
...
Temporary breakpoint 1, main () at src/
raytracer/main.c:221
221     {
(gdb) # Nous pouvons ici modifier la
résolution de l'image avant sa lecture
(gdb) # par la fonction d'initialisation
du GPU.
(gdb) set I_RESOLUTION_FACTOR = 4
(gdb) continue
(gdb) # Observer la nouvelle résolution.
```

NOTE

Le *raytracer* est statistiquement le plus souvent interrompu durant l'exécution de la fonction `sqrt()` (racine carré), car longue à exécuter et appelée très fréquemment.

2.2.3 Embarquement

Nous souhaitons finalement embarquer le programme et qu'il s'exécute automatiquement au démarrage de la RPi sans avoir besoin d'utiliser GDB pour le charger. Nous allons donc l'interfacer avec le bootloader de la RPi qui charge puis exécute le contenu de la carte microSD, selon les directives contenues dans le fichier **`config.txt`**. En l'absence de ce fichier, le comportement par défaut copie le fichier **`kernel.img`** dans la RAM à l'adresse **`0x8000`** et lance l'exécution à cette adresse.

Il est avant tout nécessaire de prendre en compte les dépendances avec l'utilisation de GDB (ex. : File I/O) ou du serveur Alpha (ex. : *mapping* mémoire). Si nécessaire, il convient alors de les implémenter. Ainsi, nous avons fait le choix arbitraire de profiter du comportement par défaut du bootloader pour placer à l'adresse **`0x8000`** un second point d'entrée contenant le code d'initialisation, puis invoquant le point d'entrée de la *runtime* C **`_start`**. D'autres solutions sont parfaitement envisageables et le code source de cette implémentation est fourni à titre d'exemple dans **`sdk/CPU_start.S`**.

Pour générer **`kernel.img`** et le copier sur la carte microSD :

```
glmf-rpi/ $ make kernel.img
arm-none-eabi-objcopy -O binary raytracer.elf
kernel.img
arm-none-eabi-objcopy --only-keep-debug raytracer.
elf kernel.dbg
glmf-rpi/ $ # Insérer la carte microSD dans le
poste de travail
glmf-rpi/ $ sudo mount -t vfat -o rw,umask=0000
/dev/sdc /mnt
glmf-rpi/ $ mv /mnt/config.txt /mnt/config.txt.gdb
glmf-rpi/ $ cp -v kernel.img /mnt
'kernel.img' -> '/mnt/hdd/kernel.img'
glmf-rpi/ $ sudo umount /mnt
```

Insérez à nouveau la carte microSD dans la RPi. Celle-ci démarre directement avec le *raytracer* sans plus aucune intervention de GDB.

Enfin, le fichier **`kernel.img`**, forme purement binaire de l'exécutable **`raytracer.elf`**, ne contient plus aucune information de debug. Il n'est donc plus possible de le débugger depuis le code source en tant que tel, mais uniquement sous forme binaire désassemblée par GDB (**`layout asm`**). Il est toutefois possible d'obtenir les informations de debug dans un fichier séparé. Le fichier **`kernel.dbg`** généré à ses côtés par la commande de compilation précédente contient toutes les informations de debug de **`kernel.img`**. Pour finir, voici comment les utiliser :

```
glmf-rpi/ $ # Une fois le fichier config.txt
restauré...
glmf-rpi/ $ arm-none-eabi-gdb
(gdb) source sdk/alpha.gdb
(gdb) set serial baud 115200
(gdb) target remote /dev/ttyUSB0
Remote debugging using /dev/ttyUSB0
```

```
warning: No executable has been specified and target
does not support
determining executable automatically.  Try using the
"file" command.
0x07f10570 in ?? ()
(gdb) # 1. Lecture des informations de debug
(gdb) file kernel.dbg
 program is being debugged already.
Are you sure you want to change the file? (y or n) y
Reading symbols from kernel.dbg...done.
(gdb) # 2. Téléchargement binaire de kernel.img à
l'adresse 0x8000
(gdb) restore kernel.img binary 0x8000
Restoring binary file kernel.img into memory (0x8000
to 0x1b130)
(gdb) # 3. Placer le registre pointeur sur
instruction à l'adresse
(gdb) # de démarrage avec Alpha
(gdb) set $pc = &_start
(gdb) break main
Breakpoint 1 at 0x8320: file src/raytracer/main.c,
line 221.
(gdb) continue
Continuing.

Breakpoint 1, main () at src/raytracer/main.c:221
221     {
(gdb) # Prêt pour débugger kernel.img
```

Dans cet exemple, nous faisons manuellement ce que la commande **load** effectuait pour nous depuis le fichier ELF **raytracer.elf**. Les deux méthodes sont d'ailleurs strictement équivalentes dans le cas présent qui ne comporte aucune autre distinction que les formats des fichiers **raytracer.elf** et **kernel.img** : tous deux contiennent le même programme binaire.

Ce dernier exemple est un moyen de distribuer le programme tout en conservant les informations de debug. Il constitue le point de départ d'une possible stratégie de support et maintenance d'une distribution binaire d'un programme embarqué.

CONCLUSION

Nous nous sommes donc totalement affranchis du coût et de la complexité que peut représenter une sonde JTAG. Maîtriser un tel équipement est une tâche complexe, souvent attribuée aux professionnels les plus aguerris. L'utilisation de GDB, couplée à une implémentation du serveur entièrement logicielle, facilite la démocratisation et la simplification du développement embarqué au quotidien. L'effort nécessaire pour mettre en œuvre le serveur relève strictement des mêmes compétences que celles du développement embarqué. GDB est en plus le débugger bénéficiant de la plus large communauté en ligne, et tout étudiant en informatique sait par exemple l'utiliser après avoir débuggé son premier programme C ou C++. Depuis les choix d'architecture en phases amont de R&D jusqu'à la vérification et la validation d'un logiciel embarqué complet en phase de production, chacun peut trouver son compte parmi les fonctionnalités de GDB. Les sondes restent néanmoins nécessaires dans certains cas bien précis, comme le debug des firmwares, le debug spécialisé de certaines unités processeur (ex. : lire/écrire des caches internes) ou encore le debug temps réel de bus rapides (ex. : debug PCIe).

Les opportunités sont désormais nombreuses sur ce terrain de jeu RPi. Comprendre le mode superviseur ARM, développer du logiciel bare-metal haute performance, ou encore découvrir les interfaces SPI ou USB, sont tout autant de sujets bas niveaux et universels à explorer. ■

RÉFÉRENCES

[1] Distribution ARM officielle de la toolchain GNU sur : https://developer.arm.com/open-source/gnu-toolchain/gnu-rm

[2] Exemple de convertisseur seul sur : http://amzn.eu/3H5FIx9

[3] Exemple de câbles jumper sur : http://amzn.eu/jiuRBbi

[4] Exemple de câble USB pour le convertisseur **[1]** sur http://amzn.eu/ihFNmFf

[5] Exemple de convertisseur format « câble » sur : http://amzn.eu/flbbyHF

[6] Documentation des tracepoints sur : https://sourceware.org/gdb/onlinedocs/gdb/Tracepoints.html

[7] Documentation de l'extension File I/O de GDB sur : https://sourceware.org/gdb/onlinedocs/gdb/File_002dI_002fO-Remote-Protocol-Extension.html

[8] Documentation de l'interface texte sur : https://sourceware.org/gdb/onlinedocs/gdb/TUI.html

[9] Liste des appels système supportés sur : https://sourceware.org/gdb/onlinedocs/gdb/List-of-Supported-Calls.html

[10] Documentation du framebuffer du SoC BCM2835 sur : http://elinux.org/RPi_Framebuffer

JDEV 2017

Journées **Développement Logiciel**

Science des données et apprentissage automatique
Systèmes embarqués et internet des objets
Infrastructures logicielles et science ouverte
Parallélisme itinérant et virtualisation
Ingénierie et web des données
Programmation de la matière
Big data et Sécurité
Usines logicielles
Génie logiciel

Webcast

4, 5, 6, 7 juillet 2017

Aix-Marseille Université, la Canebière

JDEV2017

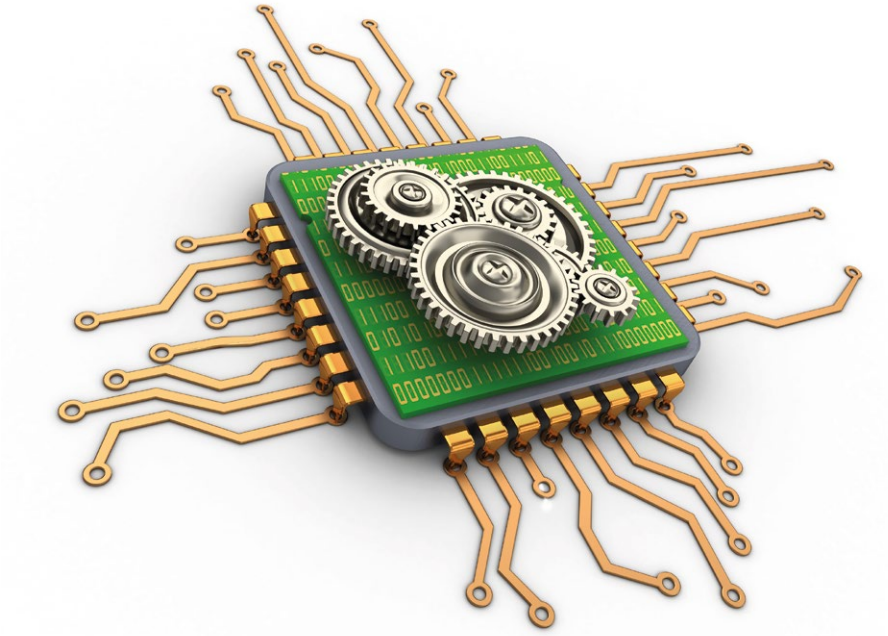
Information, programme, réservation et inscription :

http://devlog.cnrs.fr/jdev2017

PROGRAMMER DANS LE MONDE UEFI

CÉDRIC PONTOIS
[Tombé dans la marmite de l'électronique tout petit]

MOTS-CLÉS : UEFI, BIOS, GRUB, DEBUG



L'UEFI (Unified Extensible Firmware Interface) est un standard qui définit une interface uniformisée afin que les systèmes d'exploitation puissent démarrer sans se soucier des spécificités du matériel utilisé. Dans la plupart des besoins, le passage de relais entre le BIOS UEFI, le bootloader et le système d'exploitation est invisible et ne nécessite pas d'attention. Cependant, il peut être nécessaire (dans l'industrie) ou intéressant (pour les curieux) d'aller mettre les mains et quelques lignes de code dans cet univers discret...

L'UEFI (*Unified Extensible Firmware Interface*) est le successeur de l'EFI (*Extensible Firmware Interface*). À l'origine, l'EFI a été spécifié par **INTEL** afin de proposer un BIOS amélioré pour les machines avec des *hardwares* de type PC. En effet, le BIOS avait atteint ses limites depuis quelque temps : pas de prise en charge native des modes 32 et 64 bits, architecture *hardware* PC AT seulement et espace d'adressage mémoire limité à 1 Mio.

L'UEFI est spécifié et orienté par l'**Unified EFI forum**, mais dans les faits l'essentiel est guidé et fourni par INTEL du fait de sa position historique de leader dans le monde PC. Par conséquent, hors les processeurs INTEL et ses clones (x86 32/64), très peu d'architectures matérielles avec d'autres types de processeurs (ARM, PPC, etc.) utilisent l'UEFI. Il existe toutefois la possibilité de choisir une solution UEFI pour ARM depuis qu'Intel supporte les ARM64 dans **TianoCore EDK2**. Ainsi, **Microsoft** peut démarrer **WindowsRT** sur ARM. En dehors des plateformes destinées à Windows ou tournant avec processeur x86, l'UEFI n'est pas beaucoup utilisé. En effet, le développement d'un *firmware* qui est conforme aux spécifications UEFI est coûteux et rarement fait de zéro.

Si d'aventure vous concevez une carte et que vous souhaitez fournir un firmware avec une interface UEFI, plusieurs possibilités s'offrent à vous. La première, comme évoquée précédemment, est de développer depuis une feuille blanche si vous êtes courageux (et ambitieux). La seconde, si vous avez les moyens, est de sous-traiter le travail à un spécialiste comme AMI (*American Megatrends Inc.*) par exemple. Mais attention, le ticket d'entrée est entre 50 000 et 100 000 $, sans parler des royalties pour chaque processeur de chaque carte produite. Une autre solution intéressante est de partir du *package* Tianocore EDK2.

1. QUE PROPOSE UN BIOS UEFI ?

Malgré sa simplicité apparente, un BIOS UEFI (ou *firmware* UEFI) fournit beaucoup de services.

Tel un système d'exploitation, il offre une abstraction pour communiquer avec les périphériques réseaux, disques, USB et d'affichage. Il propose également des services, telle la sécurité (signature, chiffrement), souvent utilisés pour démarrer un système d'exploitation. Cela permet d'avoir un environnement « constant » d'une machine à l'autre sans avoir à se soucier des drivers de périphériques ou de la configuration de la machine. Les applications utilisant cet univers UEFI sont donc portables et facilement réutilisables.

2. POURQUOI PROGRAMMER DANS CET UNIVERS ?

2.1 Le cas de l'embarqué

Dans l'industrie et d'autres secteurs, des machines dédiées avec des architectures souvent éloignées des PC sont utilisées. Cela ne les empêche pas de tourner de plus en plus souvent sous Linux. Ces architectures matérielles requièrent souvent des initialisations de périphériques spécifiques avant le démarrage de Linux. On peut également rencontrer des besoins de vérification de code (signature, authentification) local ou réseau avec occasionnellement des protocoles propriétaires. Ou bien encore une nécessité de valider l'intégrité du matériel, ou de s'assurer d'un démarrage dans tous les cas avec de la redondance matérielle et logicielle. Dans ce contexte embarqué, si vous utilisez une architecture x86 (Intel ou autres), l'UEFI est l'étage de la fusée propice pour mettre en place ces comportements spécifiques.

2.2 Prendre le contrôle avant le système d'exploitation

Dans la même veine que pour l'embarqué, vous pouvez souhaiter faire quelques personnalisations ou vérifications même sur un PC ou un serveur (par exemple, du contrôle d'accès réseau ou de la gestion de parc). Un seul programme UEFI permet de le faire et cela quel que soit le système d'exploitation qui démarre ensuite (Windows, Linux ou autre).

2.3 Chargement spécifique

Plus simplement, un programme UEFI peut vous permettre de choisir (plus finement qu'avec un *menuentry* **GRUB**) le système d'exploitation ou la distribution à démarrer. Par exemple, en choisissant les options du noyau, le *mapping* mémoire, le niveau de verbosité des traces ou l'activation de métriques. Ce n'est qu'une question d'imagination !

3. LE CHOIX DE L'ENVIRONNEMENT DE DÉVELOPPEMENT (FRAMEWORK)

Plusieurs solutions existent si vous souhaitez écrire un programme UEFI. Trois sont exposées ci-après, mais cette liste n'est pas exhaustive.

3.1 EDK2 (Intel)

EDK2 est un projet open source provenant essentiellement d'Intel. L'idée d'Intel est de favoriser la diffusion de solutions UEFI, mais également de faciliter le démarrage de ses processeurs, qui n'est pas une sinécure. C'est un des projets du regroupement TianoCore (Tiano signifiant *Intel® Platform Innovation Framework for UEFI*).

Un des points le démarquant des autres est sa licence BSD qui le rend très souple dans son utilisation commerciale. Il est plutôt complet au niveau des fonctionnalités et des drivers. Malheureusement, c'est un plat de spaghettis au niveau des sources. Malgré des sous-projets distincts, on se retrouve rapidement avec des dépendances en série qui obligent à *linker* beaucoup d'éléments. De plus, les fonctionnalités ne sont pas bien séparées et parfois dupliquées.

3.2 La page blanche

Partir de zéro est très formateur et permet de produire des binaires minimalistes. Cependant, les outils des librairies classiques nous manquent rapidement (manipulation de chaînes, etc.). Dans ce cas, réinventer la roue est chronophage et pas forcément utile. Partir de la page blanche est donc à réserver à la formation ou pour des projets avec des contraintes spécifiques. Par exemple, un binaire inférieur à 100 kio, pas de source extérieure, pas de licence open source, etc.

Dans cette optique, un exemple basique affichant « Hello world ! » est décrit un peu plus loin.

3.3 GRUB (au-delà du bootloader)

Normalement GRUB (*GRand Unified Bootloader*) est uniquement un programme d'amorçage (un *loader* d'OS pour faire simple). Au fil du temps, ce projet est devenu très complet. Ceci tant au niveau des services, des bibliothèques, des drivers, que des systèmes d'exploitation qu'il est capable de démarrer. Il supporte nativement plusieurs systèmes de fichiers, dont les incontournables VFAT, NTFS et EXT3/4.

Pour en revenir à notre sujet, GRUB est compatible UEFI (mais pas uniquement). De plus, il se compile facilement au format EFI.

Enfin, les sources GRUB sont structurées de façon modulaire et permettent de constituer un exécutable à la carte. Le projet est constitué d'un cœur obligatoire appelé *core* et d'une multitude de modules optionnels qui peuvent être compilés statiquement avec le *core* ou sous forme de modules. La philosophie est assez proche du noyau Linux et de ses modules noyaux qui peuvent être « built-in », en module à charger ou non sélectionnés. Les 250 modules vont du simple **sleep.mod** qui propose la commande **sleep** du shell GRUB à toutes sortes de services tels que tftp, ntfs, USB, lspci, etc.

Tous ces points font du projet GRUB une très bonne boîte à outils pour travailler dans l'univers UEFI. D'autant plus qu'il propose un shell bienvenu pour charger et tracer les programmes à tester.

Pour créer un programme en utilisant GRUB, il suffira donc d'ajouter un module GRUB dans les sources. Ensuite, vous aurez le choix entre le laisser seul (et dans ce cas, il faudra le charger pour l'exécuter), ou compiler un cœur GRUB minimaliste avec votre programme/module *linké* statiquement dans ce même binaire. Ces solutions sont utilisées dans l'exemple « Bonjour» ci-dessous.

Une dernière remarque concernant GRUB : il est sous licence GPL v3. Cela ne pose pas de problème en général, mais c'est à prendre en compte si vous souhaitez vendre ou distribuer vos travaux basés sur GRUB. Tant au niveau de la mise à disposition des sources que de la gestion des clés si vous chiffrez et/ou signez vos programmes.

3.4 Les drivers

En fonction de votre projet, vous aurez peut-être besoin de communiquer avec des périphériques moins communs. Pour les classiques USB, PCI et consorts, les drivers et interfaces correspondants sont disponibles. Pour les autres, non mis à disposition par votre *firmware* UEFI, certains sont implémentés dans GRUB. Malheureusement, il n'y a pas autant de choix que sous Linux. Par exemple, si vous utilisez un bus SPI ou I2C, il faudra soit écrire le driver vous-même, soit faire un portage depuis un autre projet (EDK, Linux, fabricant du composant, uboot ou autre).

Pour respecter la tradition, nous allons décrire la création et la compilation du fameux « hello world » avec son exécution dans un shell EFI. Afin de continuer à comparer les différents *frameworks*, ce programme sera développé à partir de zéro d'un côté et dans l'environnement GRUB de l'autre.

4. COMPILER UN BINAIRE UEFI

4.1 La feuille blanche

Étape préliminaire : installer le paquet **gnu-efi** sur votre machine hôte. Attention, si vous compilez en architecture croisée, le paquet est spécifique à l'architecture.

Le source **hello.c** minimaliste qui utilise les API EFI dont le point d'entrée est **efi_main** :

```
#include <efi.h>
#include <efilib.h>

EFI_STATUS
EFIAPI
efi_main (EFI_HANDLE ImageHandle, EFI_SYSTEM_
TABLE *SystemTable)
{
   // on initialise la bibliothèque qui sert
entre autres au print:
   InitializeLib(ImageHandle, SystemTable);
   Print(L"Hello, world!\n");
   // Le return SUCCESS est important car
sinon le firmware BIOS
   // va redémarrer la machine en cas d'échec:
   return EFI_SUCCESS;
}
```

Abonnez-vous !

C'est simple... c'est possible sur :

http://www.ed-diamond.com

... OU SÉLECTIONNEZ VOTRE OFFRE DANS LA GRILLE AU VERSO 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 :

.....

.....

Bon d'abonnement

CHOISISSEZ VOTRE OFFRE !

Prix TTC en Euros / France Métropolitaine*

Offre	SUPPORT / ABONNEMENT	PAPIER Réf	PAPIER Tarif TTC	PAPIER + BASE DOCUMENTAIRE (1 connexion BD) Réf	PAPIER + BASE DOCUMENTAIRE (1 connexion BD) Tarif TTC
LM	11n° GNU/Linux Magazine France	LM1	69,-	LM13	245,-
LM+	11n° GNU/Linux Magazine France + 6n° Hors-Série	LM+1	125,-	LM+13	299,-
LES COUPLAGES AVEC NOS AUTRES MAGAZINES					
A	11n° GNU/Linux Magazine France + 6n° Linux Pratique	A1	105,-	A13	399,-
A+	11n° GNU/Linux Magazine France + 6n° Hors-Série + 6n° Linux Pratique + 3n° Hors-Série	A+1	189,-	A+13	489,-
B	11n° GNU/Linux Magazine France + 6n° MISC	B1	109,-	B13	499,-
B+	11n° GNU/Linux Magazine France + 6n° Hors-Série + 6n° MISC + 2n° Hors-Série	B+1	185,-	B+13	629,-
C	11n° GNU/Linux Magazine France + 6n° Linux Pratique + 6n° MISC	C1	149,-	C13	669,-
C+	11n° GNU/Linux Magazine France + 6n° Hors-Série + 6n° Linux Pratique + 3n° Hors-Série + 6n° MISC + 2n° Hors-Série	C+1	249,-	C+13	769,-
J	11n° GNU/Linux Magazine France + 6n° Hackable	J1	105,-	J13	399,-
J+	11n° GNU/Linux Magazine France + 6n° Hors-Série + 6n° Hackable	J+1	159,-	J+13	459,-
LA TOTALE DIAMOND !					
L	11n° GLMF + 6n° HK* + 6n° LP + 6n° MISC	L1	189,-	L13	839,-
L+	11n° GLMF + 6n° HS + 6n° HK* + 6n° LP + 3n° HS + 6n° MISC + 2n° HS	L+1	289,-	L+13	939,-

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

Particuliers = CONNECTEZ-VOUS SUR :

http://www.ed-diamond.com

pour consulter toutes les offres !

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

Professionnels = CONNECTEZ-VOUS SUR :

http://proboutique.ed-diamond.com

pour consulter toutes les offres !

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

Le fichier nommé **Makefile** permettant de compiler ce source en binaire EFI avec quelques explications au fil du fichier :

```
#sélection de l'architecture: si architecture cible différente de
l'hôte, à définir en dur.
ARCH            = $(shell uname -m | sed s,i[3456789]86,ia32,)

# Liste à étoffer selon votre projet ....
OBJS            = hello.o
TARGET          = hello.efi

# Attention les paths suivants sont à ajuster en fonction de
votre distribution:
EFI_INC_BASE    = /usr/include/efi
EFI_INCLUDE     = -I$(EFI_INC_BASE) -I$(EFI_INC_BASE)/$(ARCH)
-I$(EFI_INC_BASE)/protocol
LIB             = /usr/lib
EFILIB          = /usr/lib
EFI_CRT_OBJS    = $(EFILIB)/crt0-efi-$(ARCH).o
EFI_LDS         = $(EFILIB)/elf_$(ARCH)_efi.lds

# Les options C importantes:
# - fno-stack-protector: il n'y pas de protection de stack dans
le monde EFI
# - fpic: "Position Independant code" -> nécessaire car
l'environnement EFI peut placer notre code n'importe où.
# - fshort-wchar: Indique au compilo d'utiliser par défaut 16
bits par caractère (standard EFI). Ainsi, la chaîne "Hello World"
utilisera 22 octets.
# - mno-red-zone: Indique au compilo de ne pas utiliser de "red-
zone" comme il a l'habitude d'en avoir pour les binaires ABI.
CFLAGS          = $(EFI_INCLUDE) -fno-stack-protector -fpic \
     -fshort-wchar -mno-red-zone -Wall

#attention: en x86, les appels de fonctions doivent être faits à
l'ancienne, on utilise donc un wrapper:
ifeq ($(ARCH),x86_64)
  CFLAGS += -DEFI_FUNCTION_WRAPPER
endif

# quelques explications:
#  - nostdlib: bien sûr, on ne peut pas se servir de la libC
#  - znocombreloc: empêche la re-allocation et le tri des
sections
#  - -T /usr/lib/elf_x86_64_efi.lds: script pour spécifier au
linker le format des objets EFI
LDFLAGS         = -nostdlib -znocombreloc -T $(EFI_LDS) -shared \
     -Bsymbolic -L $(EFILIB) -L $(LIB) $(EFI_CRT_OBJS)

all: $(TARGET)

# on n'oublie pas de linker les librairies EFI et GNUEFI
hello.so: $(OBJS)
 ld $(LDFLAGS) $(OBJS) -o $@ -lefi -lgnuefi

# On convertit le fichier objet en binaire EFI en sélectionnant
les sections utiles.
%.efi: %.so
 objcopy -j .text -j .sdata -j .data -j .dynamic \
  -j .dynsym  -j .rel -j .rela -j .reloc \
  --target=efi-app-$(ARCH) $^ $@
```

Ensuite, un simple appel à la commande **make** dans le répertoire où se trouvent ces deux fichiers (**hello.c** et **Makefile**), vous créera les fichiers objet, bibliothèque et exécutable EFI. Vous pouvez mettre dès maintenant le fichier **hello.efi** sur une clé USB. Il sera testé bientôt.

4.2 Compiler avec GRUB

Tout d'abord, récupérez une archive de GRUB ou clonez son dépôt Git :

```
$ git clone git://git.savannah.
gnu.org/grub.git
```

GRUB est tellement fourni qu'il contient déjà le module **hello**. Par esprit de contradiction, nous allons donc créer un module « Bonjour ! » en complément afin de comprendre la création de modules.

Le source **bonjour.c** est à placer dans un nouveau répertoire **bonjour** créé dans le répertoire **grub-core** existant:

```
#include <grub/types.h>
#include <grub/dl.h>
#include <grub/extcmd.h>

GRUB_MOD_LICENSE ("GPLv3+");

static grub_err_t
grub_cmd_bonjour (grub_extcmd_
context_t ctxt __attribute__
((unused)),
   int argc __attribute__ ((unused)),
   char **args __attribute__
((unused)))
{
  grub_printf ("Bonjour toto!\n");
  return GRUB_ERR_NONE;
}

static grub_extcmd_t cmd;

GRUB_MOD_INIT(bonjour)
{
  cmd = grub_register_extcmd
("bonjour", grub_cmd_bonjour, 0, 0,
       N_("Affiche un bonjour"), 0);
  grub_printf ("Module bonjour en
place.\n" );
}

GRUB_MOD_FINI(bonjour)
{
  grub_unregister_extcmd (cmd);
}
```

Ensuite, on ajoute un paragraphe dans le makefile **grub-core/Makefile.core.def** pour le module « Bonjour », par exemple juste après le module « hello » :

```
module = {
  name = bonjour;
  common = bonjour/bonjour.c;
};
```

Tout est maintenant en place pour compiler. Il suffit de lancer les commandes suivantes (la *target* est à adapter en fonction de votre processeur) :

```
$ ./autogen.sh
$ ./configure --target=x86_64 --with-platform=efi
$ make
```

À cet instant, le module bonjour **grub-core/bonjour.mod** est créé et prêt à être utilisé en tant que module insérable.

Cependant, tant que le compilo est chaud, on en profite pour générer un binaire exécutable en EFI contenant notre module « bonjour » :

```
$ cd grub-core/
$ ../grub-mkimage -d . -o bonjour.efi -O x86_64-efi -p '/' bonjour normal help
```

Le deuxième ligne spécifie la liste des modules à compiler dedans (*built-in*). N'hésitez pas à ajouter tout l'outillage dont vous aurez besoin. Cela évite de les insérer manuellement ensuite.

Vous pouvez enfin transférer **grub-core/bonjour.mod** et **grub-core/bonjour.efi** sur la clé USB.

5. LA MISE À FEU

5.1 Sur machine virtuelle

Différentes machines virtuelles existent sur le marché dont certaines très abouties au niveau de l'émulation du BIOS EFI. Par exemple, le célèbre **QEMU** augmenté de **OVMF** (*Open Virtual Machine Firmware*). Il y a également **VMware** (gratuit seulement pour une utilisation non commerciale). Ce dernier offre la possibilité d'activer un BIOS UEFI en changeant dans le fichier de config de la machine virtuelle **<votre_vm>.vmx** la ligne :

```
firmware= ...
```

En :

```
firmware = "efi"
```

Si on utilise VMware pour ce test, il suffit de démarrer la machine virtuelle, de penser à connecter la clé USB à cette VM (clic droit sur l'icône du lecteur en haut à droite) et de presser la touche **<F2>** afin d'entrer dans l'émulation du BIOS.

Nous allons utiliser le shell EFI fourni avec ce BIOS pour exécuter nos programmes de tests. Sélectionnez donc l'entrée ***EFI Internal Shell*** :

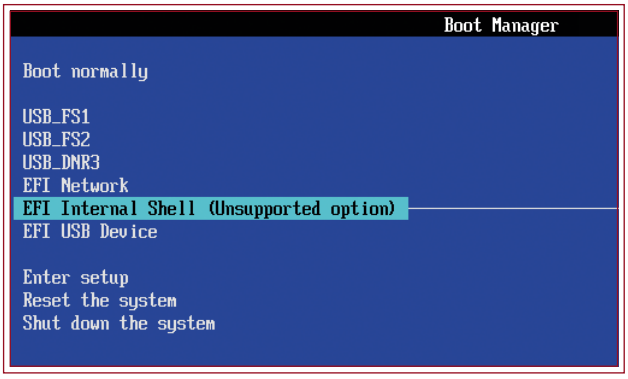


Fig. 1.

Automatiquement, le shell EFI liste les *devices* (disques) qu'il a trouvé. Les stockages USB et autres partitions de disque seront listés sous les noms **fs0**, ..., **fs<n>**.

Dans le cas où le programme à tester est placé sur la première partition du premier disque, on part à sa rencontre en tapant le nom du volume **fs<x> :** et éventuellement quelques **cd**.

Testons tout d'abord le programme réalisé depuis la feuille blanche **hello.efi** comme sur la copie d'écran suivante :

```
Press ESC in 4 seconds to skip startup.nsh, any other key to cont
Shell> fs0:

fs0:\> cd article

fs0:\article> dir
Directory of: fs0:\article

  02/21/17  04:03p <DIR>          2,048  .
  02/21/17  04:03p <DIR>              0  ..
  02/21/17  04:02p               45,023  hello.efi
  02/21/17  04:00p              114,688  bonjour.efi
  02/21/17  04:03p                1,944  bonjour.mod
  02/21/17  04:00p              313,856  bonjourn.efi
         4 File(s)     475,511 bytes
         2 Dir(s)


fs0:\article> hello.efi
Hello, world!

fs0:\article>
```

Fig. 2.

Et voilà !

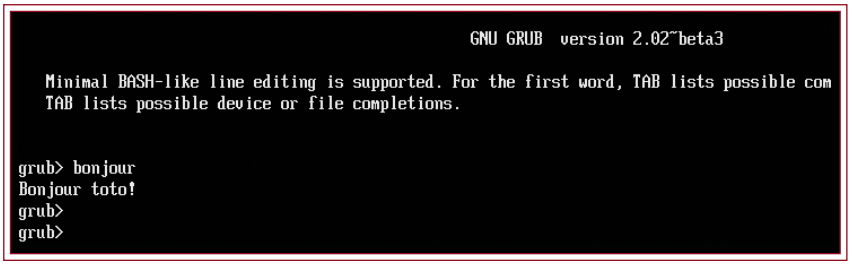


Fig. 3.

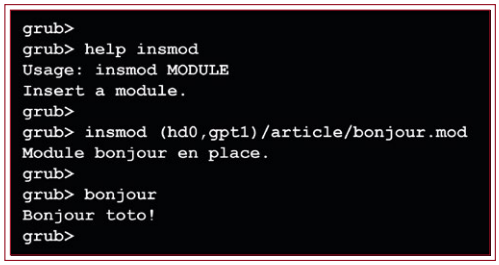


Fig. 5.

À noter que si vous effectuez de multiples essais, il est possible d'ajouter une entrée dans le BIOS VMware qui exécute directement votre programme **bonjour.efi** sans perdre de temps dans le BIOS et le shell du BIOS.

Ensuite, vient le tour du programme EFI basé sur une ossature GRUB qui se nomme **bonjour.efi**. Étant donné que **bonjour.c** se contente d'enregistrer une commande dans le shell lors de son initialisation, il faudra explicitement appeler la commande **bonjour** depuis le shell GRUB (voir figure 3).

Si vous souhaitez un comportement automatique, c'est-à-dire sans interaction avec le shell, vous pouvez définir un fichier de configuration **grub.cfg** qui contiendra ces deux lignes :

```
bonjour
exit
```

Ensuite, recompilez le programme **bonjour.efi** avec ce fichier de configuration intégré afin qu'il s'exécute automatiquement :

```
$ ../grub-mkimage -c grub.cfg -d . -o
bonjour.efi -O x86_64-efi -p '/' normal
bonjour minicmd
```

Le module **minicmd** contient la commande **exit** qui permet de sortir de GRUB sans lancer le Shell.

Et voici le résultat, de **bonjour.efi** sans shell GRUB :

Fig. 4

Le dernier élément à tester est le module **bonjour.mod** qui nécessite un GRUB en place et d'avoir la main sur le Shell. Le résultat est en image sur la figure 5.

La première étape est l'insertion du module qui nous donne un retour grâce au premier **print** mis dans la fonction **GRUB_MOD_INIT(bonjour)**. La seconde est l'appel qui affiche le message tant attendu !

5.2 Sur PC

Pour faire simple, c'est la même chose que sur machine virtuelle. On peut également se servir d'une clé USB pour transférer les binaires. Vous pouvez utiliser la même machine qui sert au développement, mais les reboot nécessaires seront chronophages. On peut avantageusement utiliser un vieux PC qui contient a minima un BIOS UEFI. Attention malgré tout à l'état des connecteurs USB après quelques centaines d'essais ...

6. LES PLUS

6.1 Les traces

En fonction des périphériques de votre machine cible, il sera plus ou moins aisé de suivre le cheminement de votre programme. En effet, si vous disposez d'un PC classique avec écran et clavier, c'est Byzance. Il suffira de laisser la configuration par défaut des entrées/sorties. En revanche, si vous avez affaire à une machine industrielle ou embarquée, la carte vidéo ne sera probablement pas du voyage. Dans ce cas, la solution historique, mais toujours vaillante en raison de sa simplicité est le port série. Dans le contexte EFI, cette interface fait très bien l'affaire, car de toute façon nous utilisons uniquement l'affichage de textes (mode console). De plus, certains *firmwares* BIOS sont capables nativement de gérer la double interface clavier/écran et port série. Ainsi, sans bricolage, il est possible de naviguer dans le BIOS et de continuer à suivre votre programme EFI sur le port série. Dans le cas contraire, il sera nécessaire de configurer et utiliser le port série via une API EFI. Ensuite, si vous utilisez GRUB, n'oubliez pas de configurer les entrées/sorties vers le port série dans le fichier de configuration (**grub.cfg**) :

```
terminal_input serial # port série uniquement
terminal_output serial # port série uniquement
```

Ou :

```
 terminal_input --append  serial # en addition de " console " si déjà présent.
 terminal_output --append serial # en addition de " console " si déjà présent.
```

La configuration de port série s'effectue ainsi :

```
serial --unit=0 --speed=115200 --parity=no --stop=1
```

À défaut de console (écran en mode texte) ou de port série, vous pouvez envoyer vos traces dans une zone mémoire que vous aurez pris le soin de réserver et de configurer en « reset safe », c'est-à-dire non initialisé au redémarrage. Ainsi, en cas d'erreur fatale, vous pourrez accéder à ces informations afin de « refaire le match ».

Une astuce pour GRUB : si vous utilisez la console comme écran, les traces dépassent très vite la taille de l'écran sans possibilité de revenir en arrière. L'option `set pager=1`, permet de mettre en pause l'exécution de GRUB, à chaque écran « plein ».

6.2 Le debug

Le debug n'est pas évident dans le monde EFI, car on ne peut pas utiliser **Eclipse** avec débogueur ni même un simple débogueur en ligne de commandes tel GDB. Cependant quelques armes peuvent être utilisées pour aller à la chasse aux bugs et gagner un peu de temps dans la mise au point.

La première arme historique et la plus empirique reste le `print`. On peut le décliner à toutes les sauces : traces d'investigations qui ne seront pas compilées dans la version nominale (livrable). Les traces conditionnelles qu'on pourra activer dynamiquement avec une option de debug. Dans ce cas, si vous avez des contraintes de performance, il faut veiller à ce que ces traces de debug ne soient pas trop gourmandes en temps CPU, mais surtout en temps d'affichage. Par exemple, un *dump* de *buffer* avec conversion en ASCII sera chronophage dès les premières dizaines d'octets. En particulier, si les sorties textes sont redirigées sur un port série.

La seconde arme est la machine virtuelle. En fonction de la complexité de votre projet EFI et de son adhérence à un matériel spécifique, les premières étapes pourront être effectuées sur machine virtuelle. Différentes machines virtuelles existent sur le marché dont certaines très abouties au niveau de l'émulation du BIOS EFI. Par exemple, VMware (comme dans les exemples précédents) ou QEMU avec OVMF.

L'intérêt de la machine virtuelle réside dans le temps de test, mais elle permet également de se passer du matériel. En effet, sur le premier point (le temps), il est très rapide de mettre le binaire à tester sur un disque partagé entre la machine hôte et la machine virtuelle. On peut également utiliser une clé USB si on veut être sûr de ne pas faire de mauvaise manipulation sur la machine hôte. Sur le deuxième point (le matériel), c'est sympa, car on peut commencer à développer et même à déboguer sans avoir accès à la machine cible en étant tranquillement sur un PC standard dans le canapé, les transports ou en vacances (euh non en fait, il y a mieux à faire en vacances :)). Plus sérieusement, cet aspect se révèle crucial si vous devez partager la machine cible entre plusieurs personnes. Par exemple, une équipe de développeurs avec seulement quelques cartes à disposition. Pire encore, il arrive que les développements logiciels doivent commencer avant même que la carte cible n'existe réellement (développement du *hardware* en parallèle par exemple). Dans ce dernier cas, vous serez content de pouvoir préparer votre code et de le déverminer au maximum afin d'avoir une version quasi bêta lors de l'arrivée des premiers prototypes. Cela peut éviter des semaines tendues sur le planning.

D'ailleurs, il ne faut pas hésiter à profiter de cette solution le plus longtemps possible quitte à bouchonner vos drivers et à adapter votre algorithme suivant la machine. Cela peut passer par une détection dynamique (chaîne « manufacturer » de la structure DMI par exemple - La structure DMI est assez complexe à décoder). Pour vous faciliter la tâche, le projet « DMI decode » vous aidera **[1]**) pour détecter sur quelle machine le binaire s'exécute ou des compilations différenciées avec des options de compilation différentes pour les cibles virtuelles et physiques.

La dernière arme (ultime) est le débogueur sur port JTAG. Avec lui, les bugs ne résistent pas...

6.3 Le système de fichiers

L'UEFI définit des API pour accéder en lecture et écriture au système de fichiers. En fonction de votre *firmware* BIOS, différents types sont supportés. A minima, le VFAT est de la partie. En revanche, l'écriture en EXT3/4 est rarement disponible.

Pour s'affranchir de cela, GRUB fournit ses propres accès au système de fichiers sans passer par l'abstraction EFI. C'est un autre avantage du *framework*

GRUB, il permet d'accéder aisément aux systèmes de fichiers classiques (FAT, EXT2/3/4, HFS, HFS+, ISO9660, JFS, minix fs, XFS, UFS), compressés ou archivés (squash FS, tar, CPIO) et virtuels (CBFS, NTFS). Le système de fichiers est donc à prendre en compte lors du choix de votre *framework*.

6.4 Le mapping mémoire

Un mot rapide sur les espaces mémoires. Ils sont définis dans le *firmware* BIOS. Certains BIOS permettent de les consulter (voire d'en créer) depuis le setup BIOS.

Afin de savoir comment ont été attribuées les zones mémoires et en particulier de connaître les blocs persistants (*reset safe*), les zones réservées et les autres, vous pouvez consulter une table UEFI des espaces mémoires. Depuis votre programme EFI, il suffira d'un appel à ***efi_system_table > boot_services > get_memory_map***.

Dans GRUB, c'est possible avec la fonction `grub_efi_get_memory_map` ou avec les commandes `lsmmap` et `lsefimmap`. La seconde commande est plus détaillée et permet d'afficher les *flags*. Par exemple, la politique de cache pour les I/O (PCI et autres).

CONCLUSION

Voici la fin de ce tour rapide de l'UEFI. Plusieurs thèmes ont été abordés a minima afin de ne pas devenir soporifiques. Si vous souhaitez aller plus en avant, des documents plus approfondis sont consultables sur le forum UEFI **[2]**, mais pas très agréables à lire. Ceux d'Intel, dans le projet TianoCore/EDK, sont un peu plus concrets **[3]**.

Enfin, pour le codage, rien ne vaut une immersion dans les sources de projets UEFI (EDK2, GRUB, BIOS Implementation Test Suite - BITS, etc.). ■

RÉFÉRENCES

[1] Projet Dmidecode :
http://www.nongnu.org/dmidecode/

[2] Documentation UEFI :
http://www.uefi.org/specifications

[3] Documentation TianoCore :
http://www.tianocore.org/docs/

CRÉATION D'UN SYSTÈME DE SAUVEGARDE « MAISON »

TRISTAN COLOMBO

MOTS-CLÉS : SAUVEGARDE, AUTOMATIQUE, RSYNC, CRON

Les services de sauvegarde dans le « cloud » sont très pratiques, car il suffit de payer et d'installer un logiciel pour les mettre en place. Pourtant, à moins de vérifier régulièrement que les sauvegardes se déroulent correctement, il est possible d'avoir des surprises... pas avec un système « fait main ».

En 2010, j'avais écrit un article intitulé « Je dois sauvegarder des fichiers », publié dans *Linux Pratique Hors-Série n°18*. Je commençais cet article de la manière suivante :

« On s'aperçoit généralement trop tard que l'on n'a pas fait de sauvegarde... et c'est à ce moment-là que l'on regrette l'heure que l'on pensait avoir gagné à ne pas mettre en place de système de sauvegarde. Comme dans la plupart des cas en informatique, il faut accepter de « perdre » un peu de temps en vue d'en gagner beaucoup plus par la suite. »

Au vu de ma récente expérience, force est de constater qu'il faut ajouter qu'il est impératif de disposer d'un système de logs avec émission de notifications en cas de problème. Et également que payer pour un système décentralisé n'est pas forcément toujours la bonne solution : un petit système « maison » peut parfois être bien plus efficace et, surtout, permet de savoir précisément ce qui est fait (ou pas).

Je vous propose donc de mettre en place rapidement à l'aide d'un simple script bash une solution adaptée à une sauvegarde automatique qui répondra précisément à nos besoins.

1. LE CAHIER DES CHARGES

Nous n'allons pas réaliser ici un véritable cahier des charges, mais plutôt une liste de besoins :

- au démarrage du système, affichage d'une popup indiquant si la sauvegarde a été correctement effectuée ou pas ;
- en cas d'erreur, sauvegarde des logs dans **/var/log/backup.log** ;
- la liste des répertoires à sauvegarder se trouvera dans un fichier **backup.list** ;
- script de sauvegarde lancé avec cron :
  - → à 12:00 sur le disque dur externe n°1 ;
  - → à 18:00 sur le disque dur externe n°2.

2. LES OUTILS

Nous allons utiliser des outils très simples : le shell (ici bash), la commande **rsync** qui est installée par défaut, une planification des tâches avec **cron**, et des notifications à l'aide de **notify-send**. Pour cette dernière commande, il faudra procéder à une installation, donnée ici pour un système basé sur **GNU/Debian** :

```
$ sudo apt install libnotify-bin
```

3. DÉFINITION DE LA COMMANDE DE SAUVEGARDE

rsync est un outil de sauvegarde incrémentale : il ne sauvegarde que les nouvelles données ou les données qui ont été modifiées depuis la dernière sauvegarde. Son utilisation basique est très simple : **rsync source destination** où **source** est le répertoire contenant les données à sauvegarder et **destination** est le répertoire contenant les fichiers de sauvegarde. Ce qui est plus complexe, c'est de déterminer quelles options utiliser ! En effet, **rsync** dispose d'un très grand nombre d'options (voir **man rsync**). Voici les options que j'ai utilisées, accompagnées de leur définition :

- **-a** : mode « archive », équivalent à l'utilisation des options **-rlptgoD**.
  - → **-r** : activation du parcours récursif des répertoires et sous-répertoires ;
  - → **-l** : copie des liens symboliques ;
  - → **-p** : copie des permissions ;
  - → **-t** : copie des dates de modification ;
  - → **-g** : copie du groupe ;
  - → **-o** : copie du propriétaire ;
  - → **-D** : copie des fichiers mode bloc et caractère ainsi que des fichiers spéciaux (sockets et fifos).
- **-H** : copie des liens en dur ;
- **--delete** : supprime les fichiers de la sauvegarde qui n'existent plus sur la source ;
- **--force** : supprime les répertoires de la sauvegarde qui ne sont plus présents sur la source, même si ces répertoires ne sont pas vides ;
- **--stats** : affiche les statistiques sur la sauvegarde (nombre de fichiers sauvegardés, créés, supprimés, etc.).

Sachant que mon premier disque externe se trouve sur **/media/tristan/485E7D9A5E7D820C**, testons la commande en sauvegardant un petit répertoire sans importance (ici **/home/tristan/Telechargements**) :

```
$ rsync -aH --delete --force --stats /home/tristan/
Telechargements /media/tristan/485E7D9A5E7D820C

Number of files: 2 (reg: 1, dir: 1)
Number of created files: 2 (reg: 1, dir: 1)
Number of deleted files: 0
Number of regular files transferred: 1
Total file size: 2,074,175,934 bytes
Total transferred file size: 2,074,175,934 bytes
Literal data: 2,074,175,934 bytes
Matched data: 0 bytes
File list size: 0
File list generation time: 0.001 seconds
File list transfer time: 0.000 seconds
Total bytes sent: 2,074,682,549
Total bytes received: 46

sent 2,074,682,549 bytes  received 46 bytes
133,850,490.00 bytes/sec
total size is 2,074,175,934  speedup is 1.00
```

Nous pouvons voir que deux fichiers ont été créés : un répertoire et un fichier. Occupons-nous maintenant du fichier de log.

4. AJOUT DU FICHIER DE LOG

Nous allons utiliser la commande **logger** qui va nous permettre de passer par **syslog** pour écrire dans un fichier **/var/log/backup.log** personnalisé. Sans paramètre, **logger** permet d'écrire dans **/var/log/syslog**, mais pour spécifier un nom de fichier de log, il va falloir configurer **rsyslog**.

À la fin du fichier **/etc/rsyslog.conf**, ajoutez la règle suivante :

```
local0.*                                -/var/log/backup.log
```

NOTE

Vous pouvez choisir un nom entre local0 et local7. Ce sont les seuls *facility names* autorisés (donc inutile de tenter un backup.* ou autre...).

Nous indiquons ici que tous les logs (**notice**, **info**, **warn**, etc.) de **local0** seront écrits dans le fichier **/var/log/backup.log**. Pour activer cette règle, il faut redémarrer rsyslog :

```
$ sudo service rsyslog restart
```

Nous pouvons tester que notre règle est correcte par :

```
$ sudo rsyslogd -N1
rsyslogd: version 8.4.2, config validation run
(level 1), master config /etc/rsyslog.conf
rsyslogd: End of config validation run. Bye.
```

Et maintenant pour un vrai test nous pouvons écrire un message dans le fichier de log :

```
$ logger -p local0.notice "Ceci est un test"
$ sudo tail /var/log/backup.log
Feb 21 14:09:03 servername tristan: Ceci est un test
```

C'est déjà bien, mais dans notre cas, c'est la commande **rsync** qui nous renverra des informations sur *stdout* (**1**) et sur *stderr* (**2**). Il faudra donc utiliser un *pipe* pour transmettre les informations à **logger** :

```
$ echo "Deuxieme test" | logger -p local0.notice
```

Problème : avec **rsync** et l'option **--stats**, nous voulons effectuer deux traitements différents pour *stdout* et pour *stderr* ! Nous allons utiliser des redirections : *stdout* sera « loggé » et *stderr* sera envoyé dans un fichier. Voici un exemple avec la commande **ls** au lieu de **rsync** pour simplifier les choses :

```
$ ls toto 2> error.log | cat
```

Ici, **ls toto** va renvoyer un message d'erreur qui sera stocké dans **error.log**. Si on utilise à la place la commande **ls** (on efface le **toto**), le fichier **error.log** sera vide et le résultat de la commande sera affiché par **cat**. Nous utiliserons le fait que le fichier d'erreur est vide ou non pour signaler une erreur lors de la sauvegarde. La commande de sauvegarde sera donc du type :

```
$ rsync -aH --delete --force --stats /
home/tristan/Telechargements /media/
tristan/485E7D9A5E7D820C 2> /home/tristan/bin/
data/backup/error.log | logger -p local0.notice
```

5. LECTURE DE LA LISTE DES RÉPERTOIRES À SAUVEGARDER

La liste des répertoires à sauvegarder se trouve dans **/home/tristan/bin/data/backupSys/backup.list**. Nous allons donc créer ce fichier en lui indiquant quelques noms de répertoires :

```
01: /home/tristan/bin
02: /home/tristan/Diamond_Editions
```

Nous devons parcourir ce fichier pour lancer les commandes **rsync**. Pour cela, créons un fichier **backupSys** qui deviendra progressivement notre script de sauvegarde. Pour l'instant, ce script va seulement lire le fichier **backup.list** et afficher son contenu :

```
01: #!/bin/bash
02:
03: while read -r directory || [[ -n
"${directory}" ]]; do
04:     echo "${directory}"
05: done < "/home/tristan/bin/data/
backupSys/backup.list"
```

Ce script permet de lire le fichier injecté en ligne 5. En ligne 3, **-r** empêche l'interprétation des anti-slashes comme des caractères de protection (ce sont des caractères comme les autres) et **[[-n "${directory}"]]** empêche la dernière ligne d'être ignorée si elle ne se termine pas par un **\n**.

6. STRUCTURATION DU SCRIPT

Notre script aura deux fonctions : lancer les sauvegardes et afficher des notifications. Nous allons créer deux fonctions pour cela ainsi qu'une fonction **usage()** indiquant comment appeler le script :

```
01: #!/bin/bash
02:
03: readonly DIR_LIST='/home/tristan/bin/
data/backupSys/backup.list'
04: readonly HDD=('/media/
tristan/485E7D9A5E7D820C' '/media/tristan/
EAA47D39A47D08F9')
05:
06: usage() {
07:     if [ "${1}" == "help" ]; then
08:         local hdd_number=0
09:         echo 'Usage: backupSys <action>
[hdd_number]'
10:         echo '       <action> :'
11:         echo '           - save :
Sauvegarde les répertoires DIR_LIST dans
HDD[hdd_number]'
12:         echo '           - notify : pour
envoyer les fichiers'
```

```
13:         echo '       [hdd_number] : numéro
du disque à utiliser'
14:         for hdd in ${HDD[@]}; do
15:             echo "           - ${hdd_
number} : ${hdd}"
16:             hdd_number=$((${hdd_number}+1))
17:         done
18:         exit 0
19:     fi
20: }
21:
22: # Sauvegarde des répertoires de DIR_LIST
sur le disque passé en paramètre
23: # ${1} : numéro du disque à utiliser dans
HDD
24: save() {
25:     if [ -z ${1} ] || [ -z ${HDD[${1}]} ];
then
26:         echo "Disque dur ${1} inexistant"
27:         exit 1
28:     fi
29:     while read -r directory || [[ -n
"${directory}" ]]; do
30:         echo "${directory}"
31:     done < ${DIR_LIST}
32: }
33:
34: # Notification d'erreur lors de la
sauvegarde
35: notify() {
36:     echo "A écrire..."
37: }
38:
39: if [ "${1}" == 'save' ]; then
40:     save ${2}
41: elif [ "${1}" == 'notify' ]; then
42:     notify
43: else
44:     usage 'help'
45: fi
```

Dans les lignes 3 et 4, nous définissons les constantes du script : **DIR_LIST** qui est le fichier contenant la liste des répertoires à sauvegarder et **HDD** qui est un tableau contenant les chemins vers les différents disques durs (la déclaration se fait en utilisant des parenthèses et en séparant chaque élément du tableau par un espace). La fonction **usage()** des lignes 6 à 20 attend un paramètre : s'il contient la chaîne de caractères **"help"** on affiche un message indiquant comment utiliser le script et sinon on ne fait rien. Notez que pour parcourir **HDD** et afficher tous ces éléments on utilise la syntaxe **for hdd in ${HDD[@]};** : la variable **hdd** prend ses valeurs dans **HDD** utilisé en tant que tableau (le **[@]**).

Dans les lignes 22 à 32, on trouve un embryon de fonction **save()** qui attend en paramètre le numéro du disque dur externe à utiliser (numéro de l'élément dans la liste **HDD**). Si ce numéro n'est pas passé en paramètre ou qu'il ne correspond à aucun élément (ligne 25), alors on sort du script avec un message (ligne 26) et un code (ligne 27) d'erreur. Sinon, pour l'instant, on affiche le contenu du fichier **backup.list**.

Dans les lignes 34 à 46 se trouve la fonction **notify()** que nous écrirons par la suite et dans les lignes 39 à 45 se trouve le « programme principal » où, en fonction des paramètres utilisés, on appelle la fonction correspondante. Ainsi, si l'on appelle **backupSys save 1**, **${1}** contient **"save"** et **${2}** contient **1** : **save ${2}** revient donc à appeler **save 1** et **1** devient la valeur du paramètre **${1}** de la fonction **save()**.

Maintenant que nous disposons de la structure générale de notre script, finalisons les différentes fonctions.

7. SAUVEGARDE

Pour la sauvegarde, nous disposons déjà de toutes les briques nécessaires, il suffit de les assembler :

```
01: #!/bin/bash
...
05: readonly ERROR_LOG='/home/tristan/bin/data/
backupSys/error.log'
06: readonly ERROR_SAVE='/home/tristan/bin/data/
backupSys/error.log.bak'
07:
08: usage() {
09:     if [ "${1}" == "help" ]; then
10:         local hdd_number=0
11:         echo 'Usage: backupSys <action> [hdd_
number] [init]'
...
16:         echo '       [init]       : si spécifié,
initialisation du fichier error.log'
...
23: }
24:
25: # Sauvegarde des répertoires de DIR_LIST sur le
disque passé en paramètre
26: # ${1} : numéro du disque à utiliser dans HDD
27: # ${2} : initialisation du fichier error.log
(optionnel)
28: save() {
29:     if [ -z ${1} ] || [ -z ${HDD[${1}]} ]; then
30:         echo "Disque dur ${1} inexistant"
31:         exit 1
32:     fi
33:     if [ ! -z ${2} ]; then
34:         mv ${ERROR_LOG} ${ERROR_SAVE}
35:     fi
36:     while read -r directory || [[ -n
"${directory}" ]]; do
37:         rsync -aH --delete --force --stats
${directory} ${HDD[${1}]} 2>> ${ERROR_LOG} | logger -p
local0.notice
38:     done < ${DIR_LIST}
39: }
...
```

J'ai ajouté la possibilité d'effacer le fichier **error.log** en passant un second paramètre à la fonction **save()** : le fichier **ERROR_LOG** est alors déplacé dans **ERROR_SAVE** comme sauvegarde. Ainsi avant la première des sauvegardes journalières, nous pourrons effacer le fichier **error.log** et y stocker seulement les erreurs de la journée. La fonction de notification ira lire ce fichier et s'il est vide n'indiquera pas d'erreur.

NOTE

Dans le cas de sauvegardes sur disque externe ou disque réseau, vous pouvez rajouter les lignes suivantes après la ligne 32 de manière à vous assurer que le disque dur est monté :

```
...
33:     if [ ! -e ${HDD[${1}]} ]; then
34:         echo "Disque du ${HDD[${1}]}
absent"
35:         echo "Disque du ${HDD[${1}]}
absent" >> ${ERROR_LOG}
36:         exit 2
37:     fi
...
```

NOTE

Il peut arriver que l'on souhaite exclure certains sous-répertoires de la sauvegarde. Pour cela, `rsync` dispose de l'option `--exclude`... encore faut-il l'intégrer à notre script. En cas de besoin, modifiez la fonction `save()` de la manière suivante :

```
...
03: readonly EXCLUDE_LIST='/home/tristan/bin/
data/backupSys/backup_exclude.list'
...
28: save() {
...
36:     # Exclusion des fichiers de EXCLUDE_LIST
37:     local exclude=""
38:     while read -r directory || [[ -n
"${directory}" ]]; do
39:         exclude="--exclude ${directory}
${exclude}"
40:     done < ${EXCLUDE_LIST}
41:     # Lancement des sauvegardes
42:     while read -r directory || [[ -n
"${directory}" ]]; do
43:         rsync -aH --delete --force --stats
${exclude} ${directory} ${HDD[${1}]} 2>>
${ERROR_LOG} | logger -p local0.notice
44:     done < ${DIR_LIST}
...
```

On parcourt simplement le fichier `backup_exclude.list` (voir la variable `EXCLUDE_LIST`) et on concatène les noms de répertoires qu'il contient en les préfixant par `--exclude` (dans la variable `exclude`). Enfin, on utilise la variable `exclude` lors de l'appel à `rsync`.

8. NOTIFICATIONS

Venons-en à la fonction de notification. Nous allons nous débrouiller pour ne pas manquer un message d'erreur ! Pour cela, j'ai téléchargé un fichier son **alarm.wav** que j'ai placé dans **bin/data/backupSys** et qui sera joué en cas d'erreur (en ayant mis le volume au maximum... on est jamais trop prudent :-)) :

```
01: #!/bin/bash
...
07: readonly ALARM_SOUND='/home/tristan/bin/
data/backupSys/alarm.wav'
...
42: # Notification d'erreur lors de la
sauvegarde
43: notify() {
44:     if [ -s ${ERROR_LOG} ]; then
45:         notify-send -i /usr/share/
icons/mate/32x32/status/dialog-warning.png
-u critical -t 15000 "backupSys" "Erreur
de sauvegarde\nConsultez les fichiers :\n-
error.log\n- /var/log/backup.log"
46:         amixer -D pulse sset Master 100%
47:         aplay ${ALARM_SOUND}
48:     else
49:         notify-send -i /usr/share/icons/
mate/32x32/emblems/emblem-default.png -t
1500 "backupSys" "Sauvegarde OK"
50:     fi
51: }
...
```

La commande **notify-send** qui permet d'afficher des notifications en haut et à droite de votre bureau accepte les paramètres suivants qui sont employés en ligne 45 :

- **-i** pour indiquer une icône. Ici, j'ai utilisé une icône du système ;
- **-u** pour donner un niveau de « gravité » qui peut être **low**, **normal** ou **critical**. Ces paramètres vont se traduire par une couleur différente de la bordure de la fenêtre de notification ;
- **-t** pour spécifier la durée d'affichage en millisecondes.

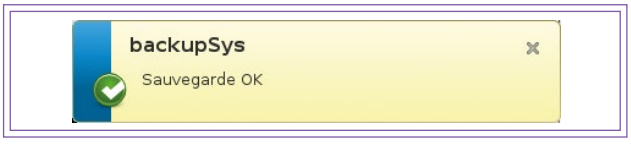


Fig. 1 : Notification rapide lorsque la sauvegarde s'est déroulée correctement.

Fig. 2 : Notification longue, la sauvegarde a échoué : ALARM !

Il est tout à fait possible de tester ces notifications en tapant la commande **`backupSys notify`**. S'il n'y a pas de fichier **`error.log`** ou que celui-ci est vide, vous obtiendrez la notification de la figure 1 et sinon vous obtiendrez la notification de la figure 2 (en plus de l'alarme tonitruante...).

9. MODE PARANOÏAQUE

Vous pouvez ajouter des contrôles un petit peu tout au long du processus de sauvegarde. Par exemple, on peut décider de recevoir un e-mail de contrôle une fois par semaine. Pour cela, j'ai employé Gmail, ce qui nécessite quelques installations :

```
$ sudo apt install msmtp
openssl ca-certificates
```

Vous devez ensuite accorder à **`msmtp`** l'autorisation d'utiliser votre compte Gmail. C'est très simple... quand on sait où cliquer ! Pour vous éviter de perdre du temps, voici les actions à effectuer, accompagnées de captures d'écrans :

1. Rendez-vous dans les paramètres de votre compte Gmail et, dans l'onglet ***Comptes et importation***, cliquez sur ***Autres paramètres de votre compte Google*** (figure 3).
2. Cliquez sur le bouton ***Connexion et Sécurité*** (figure 4).
3. Recherchez sur la page le bouton ***Mots de passe d'application*** (et cliquez dessus, ça aide... :-)) (figure 5).
4. Cliquez ensuite sur la liste déroulante ***Sélectionner une application***, choisissez ***Autre (nom personnalisé)***, saisissez « msmtp » et cliquez sur le bouton ***Générer*** (figure 6).
5. Recopiez le code (sans les espaces) (figure 7).

Fig. 3.

Fig. 4.

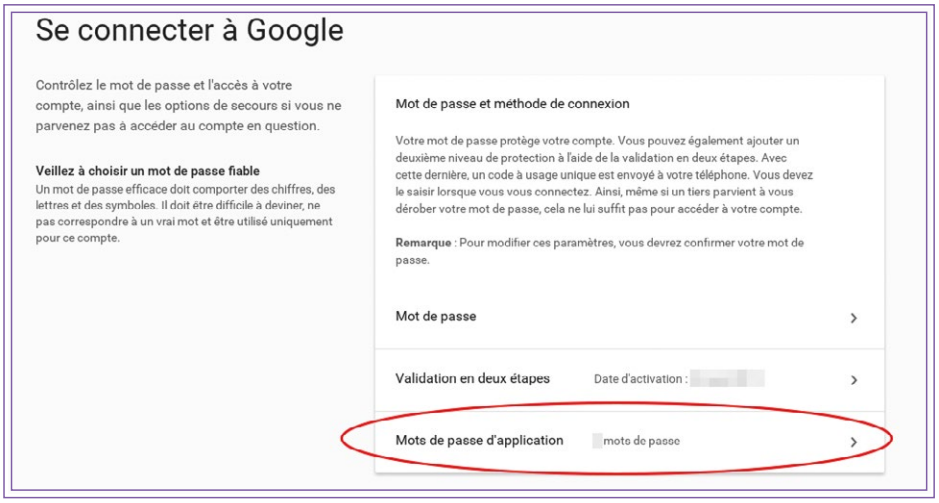


Fig. 5.

Fig. 6.

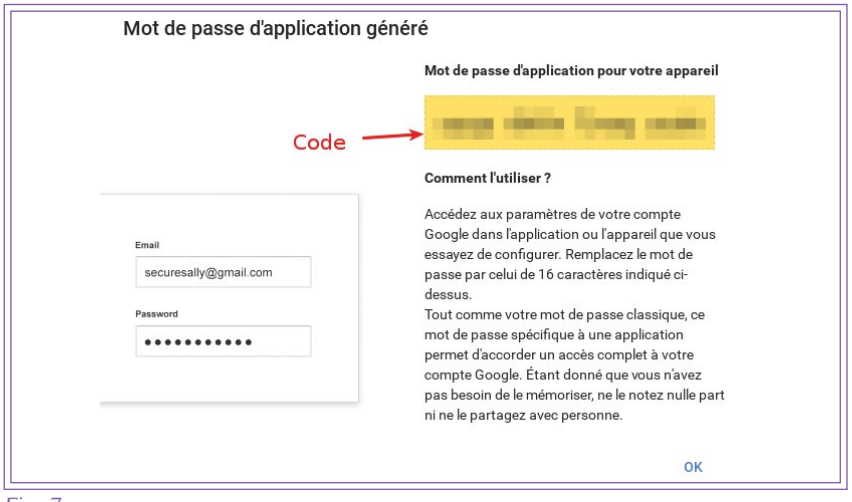


Fig. 7.

Créez ensuite le fichier de configuration **~/.msmtprc** :

```
01: account default
02: host smtp.gmail.com
03: port 587
04: from mon_adresse@gmail.com
05: tls on
06: tls_starttls on
07: tls_trust_file /etc/ssl/certs/ca-
certificates.crt
08: auth on
09: user mon_adresse@gmail.com
10: password code_obtenu_précédemment
11: logfile ~/.msmtp
```

Restreignez les droits sur ce fichier :

```
$ chmod 600 ~/.msmtprc
```

Créez ensuite un fichier **test_msg.txt** :

```
01: To:mon_adresse@gmail.com
02: Subject:Message de test
03:
04: Salut, ceci est un message de test
```

Et vous pouvez tester l'envoi d'un message :

```
$ msmtp -t < test_msg.txt
```

Il suffit ensuite d'ajouter une fonction d'envoi de mail à notre script en listant les fichiers modifiés depuis moins d'un jour grâce à **find -mtime -1** :

```
01: #!/bin/bash
...
09: readonly MAIL='mon_mail@chezmoi.com'
10:
11: usage() {
12:     if [ "${1}" == "help" ]; then
13:         local hdd_number=0
14:         echo 'Usage: backupSys <action> [hdd_
number] [init]'
15:         echo '       <action> :'
...
19:         echo '           - mail    : envoi d un
rapport par mail'
...
28: }
...
73: # Vérification des sauvegardes et rapport par mail
74: mail() {
75:     local content=""
76:     for hdd in ${HDD[@]}; do
77:         content=${content}"Sauvegarde sur
${hdd}\n"
78:         content=${content}$(find ${hdd} -mtime -1)
79:         content=${content}"\n\n"
80:     done
81:     printf "To:${MAIL}\nSubject:Rapport de
sauvegarde du $(date)\n\n${content}" > /tmp/mail.txt
82:     msmtp -t < /tmp/mail.txt
83:     rm /tmp/mail.txt
84: }
85:
86: if [ "${1}" == 'save' ]; then
87:     save ${2}
...
92: elif [ "${1}" == 'mail' ]; then
93:     mail
94: else
95:     usage 'help'
96: fi
```

Notez l'utilisation de **printf** en ligne 81 à la place d'**echo** de manière à interpréter les **\n**.

10. PLANIFICATION DES TÂCHES

Notre script est fonctionnel, il n'y a plus qu'à planifier les différents appels.

10.1 Cron

Comme indiqué en début d'article, j'ai décidé de lancer les sauvegardes sur le disque n°1 à 12:00 et sur le disque n°2 à 18:00. Pour cela, nous allons simplement ajouter des tâches planifiées :

```
$ crontab -e
```

Les lignes ajoutées seront :

```
0 12 * * * export DISPLAY=:0 && /home/
tristan/bin/backupSys save 0 init
0 18 * * * export DISPLAY=:0 && /home/
tristan/bin/backupSys save 1
```

Remarquez que pour la première sauvegarde de la journée, j'ai utilisé un paramètre **init** supplémentaire lors de l'appel de **backupSys save 0** (comme le paramètre n'est pas testé, vous pouvez utiliser absolument n'importe quoi). Cela permet d'effacer le fichier **error.log** (en fait de le déplacer pour être plus précis).

En mode paranoïaque, on peut ajouter l'envoi d'un rapport par mail le mardi à 13:00 (le lundi ne fonctionnera pas à cause du week-end... les fichiers auront été modifiés depuis plus de 24h... dans la théorie, bien sûr :-)) :

```
0 13 * * tue /home/tristan/bin/backupSys
mail
```

NOTE

Essayez de faire en sorte qu'il y ait eu au moins deux sauvegardes avant l'exécution de `backupSys mail`... sinon vous recevrez un mail listant l'ensemble des fichiers sauvegardés !

10.2 Notification au lancement de la session

Pour déclencher la notification au lancement de la session, la procédure dépendra de votre environnement graphique. Sous **Mate**, il faudra exécuter :

```
$ mate-session-properties
```

Cliquez ensuite sur le bouton ***Ajouter***, donnez un nom à la tâche puis indiquez comme commande **`backupSys notify`**.

Si vous préférez la méthode non graphique, créez directement un fichier **`~/.config/autostart/backupSys.desktop`** contenant :

```
01: [Desktop Entry]
02: Type=Application
03: Exec=backupSys notify
04: Hidden=false
05: X-MATE-Autostart-enabled=true
06: Name[fr_FR]=BackupSys Notification
07: Name=BackupSys Notification
08: Comment[fr_FR]=
09: Comment=
```

NOTE

Si vous utilisez GNOME, la commande pour le mode graphique est `gnome-session-properties` et dans le fichier `backupSys.desktop` il faut remplacer `X-MATE-Autostart-enabled` de la ligne 5 par `X-GNOME-Autostart-enabled`.

Désormais, à chaque fois que vous démarrerez une session, une notification apparaîtra pour vous tenir informé de l'état des sauvegardes.

CONCLUSION

Notre solution de sauvegarde « maison » est en place. Désormais, si jamais quelque chose ne se déroule pas correctement, nous ne pourrons nous en prendre qu'à nous-mêmes. Par contre, le temps passé à écrire ce script nous offre une liberté que nous ne pourrions pas trouver dans une solution toute prête. Rien ne nous empêche, en fonction des besoins, de modifier les paramètres de **`rsync`**, de déclarer de multiples disques de sauvegarde (locaux ou réseaux), de modifier simplement les répertoires concernés par la sauvegarde, de conserver des sauvegardes tournantes sur une semaine, etc. Les bases sont posées, libre à vous de les adapter selon vos besoins ! ■

ÉTENDRE UN SERVEUR MYSQL/MARIADB AVEC DES FONCTIONS COMPILÉES

GABRIEL ZERBIB
[Développeur de micro-services dans de gros nuages]

MOTS-CLÉS : MYSQL, MARIADB, UDF, FONCTION COMPILÉE

MySQL et MariaDB offrent la possibilité de développer des fonctions compilées, à utiliser comme toute autre fonction native dans vos requêtes SQL. Étudions la puissance et les dangers d'ajouter du code au serveur.

Les *User Defined Functions* sont un moyen d'étendre les possibilités du serveur MySQL en ajoutant des fonctions écrites en **C** ou **C++**, qui peuvent être utilisées comme d'autres fonctions SQL dans des requêtes (c'est-à-dire, qui se comportent comme les fonctions natives telles que `REPLACE`, `SUBSTR` ou `SUM`).

Elles sont entièrement compatibles entre **MySQL** et **MariaDB**, y compris au niveau binaire, car ces deux systèmes s'appuient sur la même interface pour l'écriture des UDF. Par souci de clarté, dans la suite de cet article les informations sont mentionnées pour MySQL, mais elles sont équivalentes pour MariaDB.

1. QU'EST-CE QU'UNE USER DEFINED FUNCTION ?

1.1 Les fonctions en SQL

Tout d'abord, une fonction stockée est, à l'instar d'une procédure stockée, une fonction écrite comme une série d'instructions SQL, et enregistrée au niveau de la base de données. Elle porte un nom, et accepte un nombre arbitraire d'arguments. En plus des arguments d'appel, la fonction peut travailler avec d'autres données de la base, telles que le résultat d'une requête `SELECT`.

Le code SQL d'une fonction stockée est lisible en clair, et modifiable (par les personnes autorisées). Un export de la base peut contenir le code des fonctions et procédures stockées, si l'option `--routines` est précisée à l'outil de *dump* :

```
$ mysqldump --routines [autres options] nom_de_la_base
```

1.2 Cas des fonctions UDF

Une fonction User-Defined, quant à elle, est du code compilé sous la forme d'un fichier binaire partagé (*shared object*), et installé puis déclaré au niveau du serveur de bases de données. La fonction peut accepter par valeur un nombre prédéfini d'arguments, mais elle ne peut pas travailler sur d'autres données de la base.

Le code, puisque compilé, n'est pas lisible. Il n'est pas non plus exporté dans un *dump* de la base.

Une fonction UDF est généralement considérée plus véloce qu'une fonction SQL. Mais surtout, il n'est pas toujours possible, ou optimal, d'obtenir un résultat souhaité à l'aide des seules fonctions SQL. Par exemple, pour transformer une chaîne de caractères en mettant une majuscule à chaque mot, il peut s'avérer très complexe, voire impossible, de n'utiliser que les fonctions natives de MySQL (telles que `LOCATE` et `SUBSTR`). Et, même avec une boucle `WHILE` écrite en SQL, la complexité d'exécution et l'efficacité de l'algorithme sont considérablement moins contrôlables qu'avec du code C. La fonction compilée peut également utiliser des ressources (système, fichiers, structures de données, etc.) auxquelles les fonctions SQL natives n'ont pas accès (voir plus loin la section Sécurité).

En outre, UDF est actuellement (v5.7) la seule voie pour créer une fonction d'agrégat (à utiliser en conjonction avec la clause `GROUP BY`), fournissant un résultat à partir de plusieurs enregistrements (ayant un comportement similaire à `SUM`, `AVG`, etc.). Par exemple, MySQL ne fournit pas nativement de fonction pour calculer la valeur médiane d'une population de nombres : certains projets pourraient avoir besoin de cet outil.

Dans leur forme actuelle, les UDF ont été introduites dans MySQL à la version 3.23 (janvier 2001). Auparavant, qui voulait enrichir les possibilités de MySQL avec des fonctions écrites en langage C, devait recompiler **[1]** le serveur avec son code personnalisé (cette option demeure possible pour ajouter des fonctions natives **[2]**).

Les fonctions UDF peuvent servir à des fins très diverses, de l'implémentation de formules mathématiques ou statistiques, jusqu'à l'émission de requêtes HTTP, en passant par l'interprétation de balises XML ou l'extraction d'éléments HTML par exemple.

2. MISE EN ŒUVRE

Supposons que l'on souhaite réaliser une bibliothèque qui contiendra une fonction UDF que nous nommerons `XXX` (les noms de fonctions SQL sont indifférents à la casse). Le code source de chaque UDF se compose d'un ensemble de fonctions C que le moteur exécute selon un certain cycle. Notez qu'il est possible d'écrire plusieurs fonctions UDF dans un même *shared object*.

Il faut écrire le code en respectant certaines règles, puis le compiler, installer le binaire sur l'hôte qui exécute le service de bases de données, et déclarer la bibliothèque au serveur MySQL.

2.1 Prérequis

Pour pouvoir compiler le code d'une fonction UDF, le système doit disposer des paquets suivants (si basé sur **Debian**) :

```
$ sudo apt-get install gcc libmysqlclient-dev
```

2.2 Écriture d'une UDF

Une fonction UDF peut retourner un des trois types SQL suivants : `INTEGER`, `REAL`, `STRING`, qui correspondent respectivement aux types C : `long long`, `double` et `char *`. Les arguments qu'elle accepte sont de ces mêmes types, et par valeur (c'est-à-dire qu'une fonction ne peut pas modifier la valeur d'un champ de la base ou d'une variable MySQL).

Le code source d'une fonction UDF doit respecter certaines conventions et signatures **[3]**. Un exemple complet et instructif est fourni **[4]** avec le code source du serveur MySQL.

2.2.1 Fonctions simples

Une UDF simple (sans agrégat) se compose de trois fonctions C : `xxx_init()`, `xxx()`, et `xxx_deinit()`. Elles sont invoquées selon le cycle suivant :

NOTE

Dans toute la suite, les `xxx` sont à comprendre comme un préfixe aux différentes fonctions C. Vous devrez le remplacer par le vrai nom de votre fonction UDF, tel qu'utilisé dans vos requêtes SQL.

1 : Le moteur identifie qu'une requête utilise la fonction `XXX( )`, et charge la bibliothèque correspondante si ce n'est pas déjà fait.

2 : Le moteur invoque le point d'entrée d'initialisation de la fonction : par convention, il s'agit obligatoirement du symbole `xxx_init()`, où le nom de la fonction UDF est en minuscules (ici `xxx`). La présence de cette fonction dans votre bibliothèque est optionnelle ; le moteur ne l'invoque que si elle s'y trouve, sans lever d'erreur dans le cas contraire. L'appel à `_init` a lieu une fois avant l'exécution de la requête (mais de nouveau pour chaque requête utilisant votre fonction, et autant de fois que la fonction y apparaît). Elle vous offre la possibilité d'allouer les ressources nécessaires (mémoire, etc.), de mettre à zéro vos compteurs, et autres opérations préliminaires. C'est également dans cette fonction que vous pouvez indiquer si la valeur `NULL` est une valeur de retour possible pour votre UDF.

3 : Une fois pour chaque ligne, le serveur appelle votre fonction `xxx()`, qui est la fonction principale (et qui porte obligatoirement le même nom que la fonction SQL). C'est elle qui retourne le résultat, ou qui indique si une erreur s'est produite (par exemple, en cas de division par zéro).

4 : Une fois la requête terminée, et toutes les lignes traitées, votre fonction `xxx_deinit()` (optionnelle) est appelée. C'est ici que vous libérez toutes les ressources réservées par `_init` ou dans votre fonction principale.

NOTE

Si le code est en C++, penser à exporter les points d'entrée avec la directive `extern "C"` afin que les noms soient conservés.

2.2.2 Agrégats

Pour une fonction d'agrégat, deux points d'entrée supplémentaires doivent être fournis : `xxx_add()` et `xxx_clear()`. Le cycle devient le suivant :

1 : Identification et chargement.

2 : Appel à `xxx_init()`.

3 : Le moteur groupe et trie les lignes selon les prescriptions de la clause `GROUP BY`.

4 : Au début de chaque nouveau groupe de lignes, appel de la fonction `xxx_clear()`. Vous y réinitialisez tous les compteurs et ressources communs à un même groupe de lignes.

5 : Pour chaque ligne du groupe, invocation de `xxx_add()`. Dans cette fonction, vous incrémentez vos compteurs et autres valeurs cumulatives dans vos propres structures de données (voir plus loin).

6 : À la fin de chaque groupe (une fois que `_add` a été appelé sur toutes les lignes du groupe), le moteur exécute votre fonction principale `xxx()` sur l'ensemble des données cumulées au cours de chaque appel `_add` dans votre structure personnelle.

7 : Reprise de l'étape 3, tant qu'il reste des groupes à traiter.

8 : Appel final à `xxx_deinit()`.

NOTE

Le nom `_add()` pourrait semer la confusion, mais il s'agit bien sûr d'ajouter une valeur à son groupe, afin d'y appliquer le calcul de votre choix. Il ne s'agit pas d'effectuer obligatoirement une simple addition.

2.2.3 Arguments et structure de travail

Chacune de ces fonctions reçoit en paramètre un pointeur vers une structure C `UDF_INIT`, toujours la même entre tous les appels au sein d'un même cycle.

Par exemple :

```
void xxx_add(UDF_INIT *initid, UDF_ARGS
*args, char *is_null, char *error);
```

Un des membres de cette *struct* est `char *ptr`, réservé aux besoins du développeur : c'est par lui que vous pouvez maintenir vos compteurs et autres variables partagées via votre propre structure de données (particulièrement utile pour le cas des fonctions d'agrégat), en ayant pris soin d'allouer la mémoire nécessaire dans `_init`, et de la libérer dans `_deinit`.

Les arguments SQL passés à la fonction (les vraies valeurs, pour chaque ligne) se récupèrent grâce à la structure `UDF_ARGS`. Cette *struct* contient (entre autres) :

- le membre `unsigned int arg_count` qui indique le nombre d'arguments ;
- le membre `enum Item_result *arg_type` qui est un tableau indiquant le type SQL de chaque argument ;
- le membre `char **args` qui fournit un tableau des valeurs effectives des arguments.

L'argument `char *error` vous permet d'indiquer (en lui donnant la valeur `1`) si les paramètres d'appel à votre fonction provoquent une erreur. Dans ce cas, la valeur de retour de votre UDF pour cette ligne sera `NULL`.

2.2.4 Compilation

Une fois votre code rédigé (par exemple, dans un fichier `glmf-udf.c`), la compilation se fait par la commande :

```
$ gcc -fPIC -shared  -I/usr/include/mysql
-o glmf-udf.so glmf-udf.c
```

Ceci produit le fichier `glmf-udf.so`. L'argument `-I` (i majuscule) précise le chemin des en-têtes MySQL, qui peut se trouver à un autre emplacement sur votre distribution. C'est le rôle du paquet `libmysqlclient-dev` installé en prérequis.

NOTE

Tout votre code doit être *thread-safe* : plusieurs requêtes peuvent s'exécuter en même temps, et une même requête peut faire appel plusieurs fois à votre fonction en différents endroits. Il faut donc notamment proscrire les variables globales et statiques.

2.3 Installation

Une fois programmée, puis compilée, la fonction UDF se trouve sous la forme d'un binaire partagé (`.so`) qu'il est nécessaire de déposer à un endroit du système de fichiers où le serveur MySQL saura le trouver.

Ce chemin peut être différent selon les distributions, et surtout il peut être modifié par l'administrateur, dans la configuration de lancement du service.

La variable MySQL globale `@@plugin_dir` a pour rôle d'indiquer le chemin où le serveur obtient les binaires des UDF :

```
mysql> select @@plugin_dir;
+-------------------------+
| @@plugin_dir            |
+-------------------------+
| /usr/lib/mysql/plugin/  |
+-------------------------+
```

C'est à cet emplacement que vous déposerez votre binaire. Le fichier doit appartenir au compte sous lequel tourne le service, lui être accessible en lecture et en exécution. En outre pour des raisons évidentes de sécurité vous devrez veiller à ce qu'il ne soit pas modifiable par d'autres utilisateurs.

```
$ sudo cp glmf-udf.so /usr/lib/mysql/plugin/
$ cd /usr/lib/mysql/plugin/
$ sudo chown mysql glmf-udf.so
$ sudo chmod 0500 glmf-udf.so
```

2.4 Déclaration

Une fois que le serveur MySQL est en mesure de trouver et d'exécuter votre binaire, il faut lui indiquer la ou les fonctions UDF qu'il fournit.

```
mysql> CREATE AGGREGATE FUNCTION xxx RETURNS INTEGER
SONAME "glmf-udf.so";
```

Le mot **AGGREGATE** n'est à indiquer que s'il s'agit d'une fonction d'agrégat. L'argument fourni après le mot **SONAME** est le nom du *shared object* relatif au chemin des plugins du serveur.

3. PIÈGES COURANTS

3.1 Calculs et arrondis

À la lumière de la démonstration **[5]** faite dans *GNU/Linux Magazine n°194* sur les difficultés liées au calcul à virgule flottante ou avec des très grands ou très petits nombres, la prudence s'impose dans les formules arithmétiques des fonctions UDF. Bien que les types **double** et **long long** laissent une bonne latitude concernant la précision ou la taille des nombres, la manipulation de valeurs potentiellement hétérogènes peut causer des inexactitudes difficiles à détecter.

Par exemple :

```
mysql> select 1e16 + 1 - 1e16;
+-----------------+
| 1e16 + 1 - 1e16 |
+-----------------+
|               0 |
+-----------------+
```

3.2 Mémoire

Une fois chargée en mémoire lors de la première utilisation d'une fonction UDF, la bibliothèque reste montée. Les fuites de mémoire occasionnées par du code UDF accompagnent toute la durée d'exécution du service MySQL, jusqu'à son redémarrage.

3.3 Encodage des chaînes de caractères

Les arguments textuels sont passés aux UDF sous forme de **char ***. L'inconvénient direct est que le contenu de ces chaînes varie selon l'encodage (*charset*) en vigueur lors de l'appel à la fonction UDF. Il n'existe pas de moyen simple d'identifier automatiquement (dès l'appel) l'encodage des paramètres. Des hypothèses doivent être faites (sous forme de convention d'appel à votre fonction, ou par un argument supplémentaire qui contiendrait le *charset* si c'est important pour votre traitement).

3.4 Mise au point

Il n'est pas rare de devoir effectuer plusieurs essais lors de la création d'une fonction UDF, or la mise au point n'est pas aisée, car le code s'exécute dans le contexte du service MySQL lui-même. Aussi, il est conseillé d'effectuer les tests dans un serveur de développement, voire un conteneur Docker **[6]** :

```
$ docker run -d --name
mysqlsrv -h mysqlsrv -e
MYSQL_ROOT_PASSWORD=glmf
mysql:5.7
$ docker exec -it
mysqlsrv bash
root@mysqlsrv:/#
```

3.5 Sécurité

Le code présent dans une bibliothèque UDF peut exécuter tout ce qui est autorisé au compte du service MySQL, et peut être déclenché par un simple utilisateur de la base de données sur simple appel de la fonction. Il est donc important de veiller à ce que les droits du compte du service soient convenablement configurés. En outre, il convient de ne pas faire confiance d'emblée à un binaire UDF, qui pourrait effectuer des actions sensibles ou malveillantes (ou mal codées !) provoquant des dégâts même sous le compte limité du service MySQL (qui possède tout de même tous les fichiers de données de la base...).

Avant de la déployer, il est préférable de toujours compiler le source d'une fonction UDF, à partir d'un dépôt de confiance.

CONCLUSION

La criticité de l'aspect sécuritaire des UDF oblige généralement les hébergeurs de serveurs de bases de données dans le nuage à les proscrire, à juste titre.

À l'heure où les directions informatiques tendent à s'orienter vers l'infrastructure locative, de façon à s'affranchir des tâches de maintenance et de supervision des serveurs de bases de données, il est à prévoir que les UDF, qui nécessitent un accès administratif à l'hôte, disparaîtront progressivement des architectures applicatives. Par ailleurs, l'augmentation des puissances de calcul permet le plus souvent d'envisager une solution programmatique au niveau de la couche de persistance des applications, en lieu et place d'une fonction compilée dans le SGBD, même pour ce qui est des agrégats.

Néanmoins, la connaissance de ces possibilités d'extension de MySQL/MariaDB et leurs points faibles, demeure recommandée pour savoir sécuriser un système ou faire les bons choix de conception d'un applicatif.

Notez qu'il existe un projet d'inventaire et de contributions open source de fonctions UDF utiles et populaires **[7]**. Vos besoins particuliers sont peut-être déjà couverts, et vous pourrez souhaiter contribuer à cette plateforme avec vos prochaines créations. ■

RÉFÉRENCES

[1] http://www.mathematik.uni-ulm.de/help/mysql-3.22.25/manual.html

[2] http://dev.mysql.com/doc/refman/5.7/en/adding-native-function.html

[3] http://dev.mysql.com/doc/refman/5.7/en/adding-udf.html

[4] https://github.com/mysql/mysql-server/blob/5.7/sql/udf_example.cc

[5] LANGRONET F., « Peut-on vraiment calculer avec un ordinateur », *GNU/Linux Magazine n°194,* pp 22-31

[6] https://hub.docker.com/_/mysql/

[7] http://www.mysqludf.org/

LIBÉREZ LES DONNÉES DE VOS UTILISATEURS AVEC REMOTESTORAGE

STÉPHANE MOUREY
[Mousse sur le Seeraiwer]

MOTS-CLÉS : APPLICATION UNHOSTED, JAVASCRIPT, STOCKAGE DISTANT, LITEWRITE

Vos utilisateurs sont inquiets : qui a accès à leurs données ? Sont-ils espionnés ? Permettez-leur de garder le contrôle ! Autorisez un stockage distant !

Dans un numéro précédent **[1]**, nous avons découvert une application *unhosted* et étudié comment installer un serveur **RemoteStorage** pour stocker ses données. Il est temps maintenant d'examiner comme produire notre propre application *unhosted*.

Pour ceux qui auraient manqué l'épisode précédent, rappelons ce qu'est le concept d'application *unhosted*. De quoi s'agit-il ? L'idée est de séparer le stockage des données d'une part de l'application, d'autre part : chacun de ces deux aspects s'appuie sur un service en ligne différent. L'application en ligne est alors écrite pour fonctionner entièrement dans le navigateur, sous la forme d'une unique page HTML lestée de tout le **JavaScript** nécessaire. Ce JavaScript inclut une bibliothèque, **RemoteStorage.js**, dont la tâche est de synchroniser le stockage local dans le navigateur avec le stockage distant, mais aussi de permettre à l'utilisateur d'utiliser ses données sans stockage distant, qu'il ne souhaite pas en utiliser ou que le service soit indisponible. L'un dans l'autre, RemoteStorage.js résout un problème épineux pour les applications web devant fonctionner hors ligne : celui du stockage et de la synchronisation des données.

1. POUR COMMENCER

1.1 Notre application

Pour notre première approche de RemoteStorage.js, nous allons nous contenter d'une application relativement simple :un éditeur de texte. Les différents documents existants seront listés dans une barre latérale, où sera également placé un bouton permettant la création d'un nouveau document. L'essentiel de la fenêtre sera occupé par une zone de saisie

de texte pour le contenu du document, surmontée d'un simple champ texte pour le titre. Cette zone restera vide tant qu'aucun document ne sera sélectionné ou créé. Vous pourrez éventuellement la masquer à l'aide d'une règle CSS.

1.2 Préliminaires : choix d'un serveur

Vous l'aurez compris, pour exploiter au mieux les capacités de RemoteStorage.js, il est nécessaire d'avoir recours à un service de stockage de données distant, que nous appellerons un serveur RemoteStorage. Plusieurs possibilités s'offrent à vous pour disposer d'un tel service : vous pouvez installer votre propre serveur en utilisant par exemple, **PHP Remote Storage [2]** ; vous pouvez utiliser un service en ligne tel que **5Apps [3]**, qui est gratuit. Toutes les solutions sont listées sur le wiki du projet **[4]**.

Si vous voulez simplement découvrir l'API sans perdre trop de temps sur l'installation d'un serveur, 5Apps constitue la solution la plus rapide à mettre en œuvre. Après y avoir créé un compte, vous disposerez de 1 Gio d'espace pour vos données, ainsi que la possibilité d'héberger autant d'applications *unhosted* libres que vous le souhaitez (seulement trois si vous souhaitez héberger des solutions propriétaires). Pour rappel, vous n'avez pas besoin d'héberger vos applications sur 5Apps pour pouvoir y déposer vos données, et inversement.

1.3 Installation

RemoteStorage.js adopte une architecture souple comportant un noyau toujours nécessaire et des modules, à charger selon vos besoins. Les modules permettent de manipuler des types d'objets. L'idée est que des applications différentes peuvent manipuler les mêmes objets, se partager ainsi les données et, du coup, permettre à un utilisateur de les utiliser de différentes façons. Prenons un exemple : une application de liste de tâches à réaliser (une « todo » liste) et une autre de suivi de temps (« timetracking ») pourront travailler toutes deux sur des objets **`todo`** de la catégorie **`tasks`**.

Vous trouverez la liste des modules actuellement disponibles sur le site officiel **[5]**. Si vous n'en trouvez aucun qui vous convient, vous pourrez toujours développer le vôtre, et même le proposer à la communauté, mais cela dépasse le cadre de cet article. Vous trouverez un nombre intéressant de modules, parmi lesquels nous citerons **`Bookmarks`** pour gérer les marques-pages, **`Contacts`**, **`Feeds`** pour les flux RSS ou Atom, **`Pictures`** pour les images, **`Shares`** pour les partages. Malheureusement, certains d'entre eux manquent de documentation. Cela tient peut-être à ce que le site officiel a été entièrement revu durant la rédaction de cet article : avec un peu de chance, il en sera autrement lorsque vous lirez ces lignes.

Pour chaque module que vous souhaiterez utiliser dans votre application, vous devrez installer la bibliothèque correspondante. En ce qui concerne la nôtre, nous allons nous appuyer sur le module **`Documents`**.

1.3.1 Le noyau

Trois paquets vous sont proposés parmi lesquels il vous faut choisir : avec ou sans cache local, et un permettant l'utilisation de l'API AMD (*Asynchronious Module Definition*) **[6]**. Cette dernière étant aujourd'hui dépréciée, ignorez-la tout simplement. La version avec cache est généralement celle à préférer, elle dispose d'ailleurs de certains paramètres permettant de désactiver partiellement le cache. Mais seule la version sans cache permet de le désactiver entièrement : vous pourrez y avoir recours par exemple si, cherchant à déboguer votre application, vous soupçonneriez que son comportement anormal est lié au cache.

Une fois votre choix fait, plusieurs méthodes d'installation sont possibles, selon vos préférences. Vous pouvez utiliser **`bower`** :

```
$ bower install -S remotestorage
```

Ou **`git`** :

```
$ git clone https://github.com/remotestorage/remotestorage.js.git
```

Ou encore un téléchargement direct :

```
$ wget https://raw.githubusercontent.com/remotestorage/
remotestorage.js/master/release/stable/remotestorage.min.js
```

Enfin, pour les plus pressés d'entre vous, pourquoi ne pas avoir recours au CDN en ajoutant cette simple ligne de code dans le **`head`** de votre document HTML ?

```
<script type="text/javascript" src="https://cdnjs.cloudflare.com/
ajax/libs/remoteStorage/0.12.1/remotestorage.min.js"></script>
```

Naturellement, si vous n'avez pas opté pour le CDN, une ligne de code équivalente, mais avec un chemin corrigé, sera nécessaire pour pourvoir utiliser RemoteStorage.js.

Pour ceux qui seraient tentés par le développement d'une application plus complexe, ou simplement utilisant une autre architecture que la page web, mais s'appuyant tout de même sur les capacités de stockage distant de RemoteStorage.js, il est possible d'installer le module pour NodeJs :

```
$ npm install --save
remotestoragejs
```

1.3.2 Le module

Tous les modules officiels se trouvent stockés dans le même dépôt **GitHub [7]**. Il vous est donc possible de les récupérer tous à la fois avec la commande suivante :

```
$ git clone https://github.com/
remotestorage/modules.git
```

Vous aurez ainsi la possibilité de piocher les modules dont vous aurez besoin. Cela facilitera également les mises à jour. Si vous optez toutefois pour un simple téléchargement, il vous faudra vous rendre sur la page https://github.com/remotestorage/modules/tree/master/dist pour trouver votre module avant de le télécharger en mode « raw », c'est-à-dire sans embellissement, ce qui nous donnerait, dans notre cas, la commande suivante :

```
$ wget https://raw.
githubusercontent.com/remotestorage/
modules/master/dist/documents.js
```

À supposer que vous adoptiez l'approche basée sur Git, il faudra donc ajouter la ligne suivante dans le **head** de votre document HTML :

```
<script type="text/javascript"
src="modules/dist/documents.
js"></script>
```

Dans le cas où vous auriez opté pour le téléchargement, le chemin est naturellement à adapter en fonction de l'emplacement réel du fichier.

1.3.3 JQuery

Bien que j'ai pour principe de minimiser l'utilisation de bibliothèques qui ne sont pas au cœur du sujet, je ferai ici exception en recourant à **JQuery** qui a une vertu parmi d'autres, celle de rendre le code JavaScript bien plus lisible. Ajoutons donc une simple ligne pour le charger depuis un CDN **[8]** :

```
<script src="https://code.jquery.com/jquery-3.0.0.min.js"
integrity="sha256-JmvOoLtYsmqlsWxa7mDSLMwa6dZ9rrIdtrrVYRn
DRH0=" crossorigin="anonymous"></script>
```

1.3.4 Placer notre code JavaScript et en terminer avec le HTML

Enfin, dans le but de faire les choses proprement, nous allons placer tout le code JavaScript de notre application dans un fichier séparé que nous appellerons **app.js**. Il nous faut également l'inclure depuis le **head** de notre document.

Pour en terminer avec la mise en place, j'indique ci-dessous le HTML complet de l'application. Vous y trouverez des éléments destinés aux CSS, mais qui ne rentrent pas dans le cadre de cet article, je n'en parlerai donc pas. Ce qui nous donne maintenant ceci :

```
<!DOCTYPE html>
<html>
<head>
 <meta charset="utf-8">
 <title>Application de test pour GNU/Linux Magazine
France</title>
 <link rel="stylesheet" type="text/css" href="style.css">
 <script type="text/javascript" src="https://cdnjs.
cloudflare.com/ajax/libs/remoteStorage/0.12.1/
remotestorage.min.js"></script>
 <script src="https://code.jquery.com/jquery-3.0.0.min.
js" integrity="sha256-JmvOoLtYsmqlsWxa7mDSLMwa6dZ9rrIdtrr
VYRnDRH0=" crossorigin="anonymous"></script>
 <script type="text/javascript" src="modules/dist/
documents.js"></script>
 <script type="text/javascript" src="app.js"></script>
</head>
<body>
 <div id="controls"><a class="control button" id="newDoc"
href="#">New</a></div>
 <div id="list"></div>
 <div id="work">
  <input id="title" placeholder="Title"><br>
  <textarea id="content" placeholder="Your text"></
textarea>
 </div>
</body>
</html>
```

Cela paraît très sommaire, mais c'est suffisant pour l'étude qui nous intéresse ici. Passons maintenant à l'aspect JavaScript de notre application.

2. RÉALISATION DE NOTRE APPLICATION

2.1 Afficher le widget

Bien que, à croire la documentation officielle, cette étape ne doive être réalisée qu'après que nous ayons fait un minimum de déclarations quant à notre application, je ne résiste pas à afficher tout de suite le widget qui va permettre à

l'utilisateur de se connecter à son stockage de données distant. Pour ce faire, une simple ligne de JavaScript suffit, à placer dans notre fichier **app.js**. Toutefois, pour éviter que celle-ci ne s'exécute trop tôt, nous allons différer son appel jusqu'à ce que la page soit entièrement chargée en la liant à l'évènement **window.load** :

```
window.onload = function(){
 remoteStorage.displayWidget();
}
```

De fait, toute l'initialisation de notre application sera à placer dans cette fonction anonyme déclenchée à la fin du chargement de la page.

Vous pouvez d'ores et déjà recharger la page pour voir le widget et même tester : vous obtiendrez une erreur, car notre application ne demande encore aucune autorisation, ce qui n'est pas cohérent avec l'utilisation de RemoteStorage.

2.2 Demander une autorisation d'accès

Les données enregistrées dans votre stockage distant sont structurées en dossiers. Pour autant, l'arborescence n'est pas arbitraire et il est recommandé que votre application respecte certaines conventions pour mieux interagir avec les autres applications. On retrouve ici une logique similaire à celle que nous avons vue pour les modules : les noms à utiliser pour les dossiers sont le plus souvent homonymes des modules. Il existe deux niveaux d'autorisation aux dossiers : soit l'application demande un simple accès en lecture, soit elle réclame un accès total. Dans notre cas, notre application requiert un accès total au dossier **/documents**. Cette demande est effectuée avec une seule ligne de code, à placer avant l'affichage du widget :

```
window.onload = function(){
 remoteStorage.access.claim('documents', 'rw');
 remoteStorage.displayWidget();
}
```

Nous aurions obtenu un accès en lecture seule en passant le second paramètre de **access.claim()** de **'rw'** à **'r'**.

À partir de ce moment, après avoir rechargé la page, il suffira de cliquer sur le widget pour que la connexion avec le service d'hébergement de données puisse s'établir. Il faudra alors que l'utilisateur valide les autorisations demandées pour que l'application puisse l'utiliser. Il pourra par la suite révoquer ces autorisations à tout moment à partir du site du service d'hébergement.

2.3 Architecture

À partir d'ici, le développement proprement dit de l'application commence. Il nous faut encore définir une architecture. Nous allons pour notre exemple rester sur une conception simple : le cœur de notre application sera un objet appelé **app** qui disposera de toutes les méthodes nécessaires pour le fonctionnement de l'application, mais sans considération de l'interface ; un autre objet, appelé **UI**, contiendra les méthodes faisant le lien entre l'interface et l'objet **app**. Lors du chargement initial de la page, une méthode **init** de l'objet **UI** sera appelée, qui effectuera tous les appels nécessaires pour alimenter l'interface graphique en données. Voici donc le squelette du code JavaScript de notre application :

```
app = {
};

UI = {
 init: function() {
 }
}

window.onload = function(){
 remoteStorage.access.
claim('documents', 'rw');
 remoteStorage.displayWidget();
 UI.init();
}
```

2.4 Lister les documents existants

Passons maintenant au développement de notre première fonctionnalité : l'affichage de la liste des documents. Oui, mais, comment vérifier le bon fonctionnement de notre code puisqu'il n'y a rien à afficher pour le moment ? Comme pour une application classique, il nous est possible de créer manuellement des fichiers de données. Pour cela, il nous aurait fallu utiliser un explorateur de fichiers, mais il n'en existe pas de fonctionnel à ma connaissance au moment où j'écris ces lignes **[9]**. Nos outils habituels ne sont pas appropriés **[10]**. Nous allons donc avoir recours à une autre application *unhosted* : **LiteWrite [1]**, accessible sur https://litewrite.net.

LiteWrite va utiliser le dossier **/documents/notes** dès que vous aurez écrit quelques lignes pour les stocker. Ce sont ces fichiers que nous allons lister dans un premier temps.

Pour ce faire, nous allons écrire la méthode **list** de l'objet **app**. Celle-ci sera appelée par la méthode **updateList** de l'objet **UI** pour mettre à jour l'affichage de notre liste de documents. Cette mise à jour doit avoir lieu pour commencer lors du chargement de la page : la méthode **init** de **UI** fera donc un appel à **updateList** :

```
app = {
 list: function(){
 }
};

UI = {
 init: function() {
 this.updateList();
},
updateList: function(){
 app.list();
 }
}
```

Cela fait, nous pouvons demander à RemoteStorage.js de nous fournir la liste des fichiers. Le module **documents** nous propose deux méthodes qui ont l'air très similaires : **privateList** et **publicList**. Celles-ci nous retournent chacune un objet nous permettant d'accéder aux objets stockés par RemoteStorage à l'intérieur de l'espace sur lequel nous avons obtenu une autorisation (**/documents**). Elles diffèrent en ceci que **privateList()** nous renverra tous les objets, alors que **publicList** ne nous reverra que les identifiants des documents partagés : le serveur RemoteStorage permet en effet de partager des objets accessibles ensuite à partir d'une URL. Ces deux méthodes acceptent en paramètre un chemin qui permet éventuellement de n'afficher que le contenu d'un sous-dossier de **/documents**, paramètre que nous n'allons pas utiliser ici. Nous ne nous occuperons pas du partage pour le moment, nous allons donc nous concentrer sur **privateList**.

En réalité, **privateList()** et **publicList()** ne nous renvoient pas la liste des documents, mais chacune un objet de classe respectivement **privateClient** et **publicClient**, héritant tous deux de **baseClient**. Et ce n'est que cette dernière classe qui nous propose les méthodes **getAll**, permettant de recevoir l'ensemble des documents et **getListing** permettant d'en recevoir une liste. Vous préfèrerez l'une ou l'autre selon les exigences de votre application. Dans notre cas, les documents créés par LiteWrite étant stockés par un UUID **[11]** et non par un nom compréhensible, nous allons préférer **getAll()** de façon à pouvoir accéder à leurs titres.

Réalisons un premier affichage en console des valeurs des retours faits par **getAll()** :

```
app = {
list: function(){
 console.log(remoteStorage.documents.
privateList('').getAll());
}
};
[...]
```

Ô surprise, nous n'obtenons pas une liste, mais un objet de type **Promise**. De quoi s'agit-il ? Les promesses **[12]** sont LA nouvelle méthode recommandée pour faire des traitements asynchrones en JavaScript aujourd'hui au lieu de fonctions de rappel associées à un évènement. Une promesse est un objet qui doit fournir une valeur de retour, qu'il s'agisse du résultat d'un traitement ou de l'erreur qu'il a provoquée, mais qui n'est pas en mesure de produire ce résultat immédiatement. Une promesse peut se trouver dans trois états différents : en attente, accompli (le résultat a été obtenu) ou rejeté (une erreur s'est produite). Une fonction peut être appelée lorsque la promesse atteint l'état accompli ou rejeté, associée à la promesse par la méthode **then**.

Bref, nous ne pouvons rafraîchir l'affichage de notre liste que lorsque la promesse aura été tenue. Commençons par afficher dans la console le résultat de cette promesse :

```
app = {
 list: function(){
  remoteStorage.documents.privateList('').
getAll().then(function(res){
  console.log(res);
  });
 }
};
[...]
```

getAll() nous renvoie un objet, vide si vous n'avez pas pris le temps d'ajouter des données depuis LiteWrite. Il est au format JSON et contient un nœud pour chaque dossier et pour chaque document, arborescence JSON respectant celle des dossiers, ce qui rend notre objet facile à exploiter. Pour simplifier à partir d'ici, et éviter d'avoir à parcourir cette arborescence tout en assurant la compatibilité avec LiteWrite, je lierai nos clients au chemin **documents/notes** en leur passant simplement **'notes'** en paramètre (le premier niveau étant déjà établi par le fait que nous utilisons **remoteStorage.documents**). Je ne considèrerai donc pas ici d'hypothétiques sous-dossiers. C'est d'ailleurs un choix pleinement assumé par LiteWrite.

À partir de là, nous connaissons suffisamment nos objets pour en faire le traitement nécessaire. Quelques minutes de travail plus tard, notre code devient :

```
app = {
 init: function(path){
  this.client = remoteStorage.documents.
privateList(path);
  this.currentDoc = null;
  this.path = path;
 },
 list: function(){
  return this.client.getAll();
 }
}
```

```
UI = {
 init: function() {
  this.updateList();
 },
 updateList: function(){
  app.list().then(UI.updateUIList);
 },
 updateUIList: function(list){
  newListDivHtml = 'Docs';
  newListDivHtml+= UI.listItems2UL(list);
  $('#list').html(newListDivHtml);
 },
 listItems2UL: function(items){
  ul = '';
  $.each(items,function(index, value){
  ul+= '<li><span class="load control"
onclick="UI.loadDoc(\''+index+'\');">';
  ul+=value.title+'</span>';
  });
  return '<ul>'+ul+'</ul>';
 },
 loadDoc: function(id){
  console.log(id);
 }
}

window.onload = function(){
 remoteStorage.access.claim('documents',
'rw');
 remoteStorage.displayWidget();
 app.init('/notes');
 UI.init();
}
```

Lorsque vous rechargez la page, vous devriez voir apparaître la liste des titres de vos documents dans la **div** ayant pour **id** la valeur **list**. Lorsque vous cliquez sur l'un de ces documents, la console JavaScript devrait afficher son identifiant. En effet, nous avons préparé la fonction **UI.loadDoc** qui aura pour tâche de charger le document cliqué dans l'interface.

2.5 Charger un document

Pour charger un document, nous allons ajouter deux fonctions, l'une, **loadDoc** à **UI**, l'autre, **getDoc** à **app**, en suivant toujours notre logique de séparation des tâches : **getDoc** va obtenir l'objet JSON correspondant au document demandé depuis RemoteStorage et UI va se charger de mettre à jour l'affichage dans notre interface en fonction de cet objet. Ce qui nous donne :

```
app = {
[...]
 getDoc: function(id){
  return this.client.getObject(id);
 }
}
```

```
[...]
UI = {
[...]
 loadDoc: function(id){
  app.getDoc(id).then(
   function(doc){
    app.currentDoc = doc;
    $('#work').css('display','block');
    $('#title').val(doc.title);
    $('#content').val(doc.content);
   }
  );
 }
}
```

Commençons par **getDoc()**. À vrai dire, cette fonction ne fait qu'une chose : elle retourne la promesse que lui renvoie l'objet client, promesse de fournir l'objet JSON correspondant à l'**id** passé en paramètre.

loadDoc() est un peu plus complexe : d'abord, elle appelle **getDoc()** dont elle reçoit la promesse susdite et, lorsque cette promesse est remplie, un traitement lui est associé. Ce traitement définit le document actuellement traité par l'application **app.currentDoc**, procède à l'affichage de la zone de travail contenue dans la **div** ayant **work** pour **id** et remplit les différents champs de formulaire avec les valeurs reçues depuis l'objet JSON. Rien de très sorcier en fait une fois qu'on a saisi la logique d'ensemble.

2.6 Enregistrer les modifications

Le principe habituel des applications *unhosted*, comme pour beaucoup d'autres applications en ligne, est d'enregistrer toutes les modifications à mesure qu'elles se produisent, sans attendre de l'utilisateur qu'il clique sur un quelconque bouton d'enregistrement. Ce sera également le cas ici.

Encore une fois, nous ajoutons à nouveau deux fonctions, une sur **app**, l'autre sur **UI**, toutes deux appelées **saveDoc**. Par ailleurs, il nous faut lier la fonction **UI.saveDoc** aux modifications ayant lieu sur nos champs de contrôle, ce qui est à faire dans **UI.init()** et que nous ferons à l'aide de JQuery :

```
app = {
[...]
 saveDoc: function(){
  return this.client.set(this.currentDoc.id,this.
currentDoc);
 }
}
[...]
UI = {
 init: function() {
  $("#work").bind('input propertychange',
function(){UI.saveDoc();});
[...]
 saveDoc: function(){
```

```
    app.currentDoc.title = $('#title').val();
    app.currentDoc.content = $('#content').val();
    app.saveDoc().then(UI.updateList);
  }
}
```

UI.saveDoc() sera donc appelée dès qu'une modification se produira soit sur le titre, soit sur le contenu de notre document. Que va-t-elle faire ensuite ? Elle va modifier les valeurs correspondantes du document courant de l'application **app.currentDoc** puis elle va demander à **app** de procéder à son enregistrement. En retour est reçue une promesse d'effectuer cet enregistrement : lorsqu'elle sera remplie, une mise à jour de la liste des documents aura lieu. Une optimisation possible ici est de n'effectuer cette mise à jour que si le titre a été modifié.

2.7 Créer un nouveau document

Mais notre application n'aurait pas beaucoup de sens s'il fallait toujours s'en remettre à LiteWrite pour créer nos documents. Voyons comment procéder. Suivant toujours notre logique, ajoutons deux fonctions **newDoc** à **app** et **UI**, cette dernière devant être liée au bouton ayant **newDoc** pour **id** afin de la déclencher :

```
app = {
[...]
 newDoc: function(){
  newId = this.client.uuid();
  d = new Date();
  title = 'New document - '+d.
toLocaleDateString()+'-'+d.toLocaleTimeString();
  cdate = d.getTime();
  return this.client.set(newId,{id:newId,content:''
,title:title,lastEdited:cdate,created:cdate});
 }
}
[...]
UI = {
 init: function() {
  $("#newDoc").click(function(){UI.newDoc();});
[...]
 newDoc: function(){
  app.newDoc().then(UI.loadDoc).then(UI.updateList);
 }
}
```

Ici, les choses sont un peu plus élaborées. Commençons par **app.newDoc()**. Nous commençons par obtenir un nouvel identifiant pour notre objet à l'aide de la fonction **uuid** du client RemoteStorage. Puis, étant donné que le document n'a pas encore de titre et que nous l'utilisons pourtant dans notre interface pour que l'utilisateur puisse l'identifier, il m'a paru bon de lui en donner un par défaut. Pour éviter les doublons, dans le cas où l'utilisateur créerait plusieurs documents sans prendre le soin de changer leurs titres, plutôt que de suivre l'usage qui veut que chaque nouveau document ait le même titre suivi d'un numéro entre parenthèses indiquant son ordre de création, j'ai trouvé plus simple et plus significatif d'utiliser la date et l'heure de création du document en suivant les conventions locales configurées dans le navigateur. D'où le petit travail pour la création du nouveau titre. Tant qu'à faire, j'utilise également cette date pour ajouter deux propriétés à notre document : **lastEdited** et **created** qui reçoivent respectivement le *timestamp* de dernière modification et de création. Je peux ajouter à ma *TODO list* de mettre à jour la propriété **lastEdited** lors de l'enregistrement d'une modification sur notre document. Enfin, la fonction **set** du client RemoteStorage est appelée, avec pour premier paramètre l'identifiant de l'objet, et pour second, l'objet lui-même. Cette fonction renvoie une promesse qui retournera l'objet lorsque celui-ci aura été enregistré.

Passons maintenant à **UI.newDoc()**. Cette fonction enchaîne les promesses. La promesse de création appelle celle du chargement, qui elle-même appelle la mise à jour de la liste des documents. Tout cela fonctionnerait très bien si la promesse de création renvoyait un identifiant, or ce n'est pas le cas, elle renvoie l'objet lui-même, contrairement à ce qu'attend **UI.loadDoc()**. Une solution rapide consiste à modifier **UI.loadDoc()** de façon à ce que, si elle ne reçoit pas un identifiant, elle suppose qu'elle reçoit un objet et donc en extrait l'identifiant pour poursuivre son fonctionnement précédent :

```
UI = {
[...]
 loadDoc: function(id){
  if (typeof myVar !== 'string'){
  id = id.id;
 }
[...]
```

Une solution plus propre serait de séparer les deux fonctionnalités assumées par **UI.loadDoc()** actuellement, et de déplacer le contenu de la fonction anonyme utilisée dans **UI.loadDoc** pour faire la mise à jour de l'interface dans une nouvelle fonction **UI.updateDoc** par exemple. Cela se révèlera plus performant également.

2.8 Supprimer un document

Nous pourrons considérer que nous aurons achevé une première version de notre application dès que nous aurons permis à l'utilisateur de supprimer un document. Après ce que nous venons de traverser, son développement paraît trivial.

Commençons par ajouter une lettre **X** à droite du nom du document dans la liste. Un peu de CSS vous permettra aisément de lui donner l'apparence d'une croix rouge. Au clic sur cette croix, nous associons une fonction **UI.eraseDoc** qui déclenchera l'effacement proprement dit. Pour cela, il nous faut modifier légèrement **UI.listItems2UL()** :

```
[...]
$.each(items,function(index,
value){
 ul+= '<li><span class="load
control" onclick="UI.
loadDoc(\''+index+'\');">';
 ul+=value.title+'</span>';
 ul+='<span class="erase
control" onclick="UI.
eraseDoc(\''+index+'\');">X</
span></li>';
});
[...]
```

Ensuite, vous l'aurez deviné, nous allons développer les fonctions **eraseDoc** sur **UI** et **app**.

```
app = {
[...]
 eraseDoc: function(id){
  if (this.currentDoc &&
id == this.currentDoc.id) {
   this.currentDoc = null;
  }
  return this.client.
remove(id);
 }
}
UI = {
[...]
 eraseDoc: function(id){
  sure = window.
confirm('Êtes-vous sûr ?');
  if (sure){
   if (app.currentDoc &&
id == app.currentDoc.id){
    $('#work').
css('display','none');
    $('#title').val('');
    $('#content').val('');
   }
   app.eraseDoc(id).
then(UI.updateList);
  }
 }
}
```

3. LE CODE

À l'issue de cette lecture, vous ressentirez peut-être le besoin de parcourir notre code dans son ensemble. Le voici ci-dessous, amélioré toutefois en suivant les recommandations que je vous ai indiquées ici ou là :

```
app = {
  init: function(path){
    this.client = remoteStorage.documents.privateList(path);
    this.currentDoc = null;
    this.path = path;
  },
  list: function(){
    return this.client.getAll();
  },
  getDoc: function(id){
    return this.client.getObject(id);
  },
  saveDoc: function(){
    d = new Date();
    app.currentDoc.lastEdited = d.getTime();
    return this.client.set(this.currentDoc.id,this.currentDoc);
  },
  newDoc: function(){
    newId = this.client.uuid();
    d = new Date();
    title = 'New document - '+d.toLocaleDateString()+'-'+d.
toLocaleTimeString();
    cdate = d.getTime();
    return this.client.set(newId,{id:newId,content:'',title:
title,lastEdited:cdate,created:cdate});
  },
  eraseDoc: function(id){
    if (this.currentDoc && id == this.currentDoc.id) {
      this.currentDoc = null;
    }
    return this.client.remove(id);
  }
}

UI = {
  init: function() {
    $("#newDoc").click(function(){UI.newDoc();});
    $("#work").bind('input propertychange', function(){UI.saveDoc();});
    this.updateList();
    remoteStorage.on('disconnected',function(){
      UI.updateList();
    });
  },
  updateList: function(){
    app.list().then(UI.updateUIList);
  },
  updateUIList: function(list){
    newListDivHtml = 'Docs';
    newListDivHtml+= UI.listItems2UL(list);
    $('#list').html(newListDivHtml);
  },
  listItems2UL: function(items){
    ul = '';
    $.each(items,function(index,value){
      ul+= '<li><span class="load control" onclick="UI.
loadDoc(\''+index+'\');">';
      ul+=value.title+'</span>';
      ul+='<span class="erase control" onclick="UI.
eraseDoc(\''+index+'\');">X</span></li>';
    });
    return '<ul>'+ul+'</ul>';
  },
  loadDoc: function(id){
    app.getDoc(id).then(UI.updateDoc);  },
  updateDoc: function(doc){
    app.currentDoc = doc;
```

```
    $('#work').css('display','block');
    $('#title').val(doc.title);
    $('#content').val(doc.content);
  },
  saveDoc: function(){
    oldTitle = app.currentDoc.title;
    app.currentDoc.title = $('#title').val();
    app.currentDoc.content = $('#content').val();
    oldTitle == app.currentDoc.title
      ? app.saveDoc()
      : app.saveDoc().then(UI.updateList);
  },
  eraseDoc: function(id){
    sure = window.confirm('Êtes-vous sûr ?');
    if (sure){
      if (app.currentDoc && id == app.currentDoc.id){
        $('#work').css('display','none');
        $('#title').val('');
        $('#content').val('');
      }
      app.eraseDoc(id).then(UI.updateList);
    }
  },
  newDoc: function(){
   app.newDoc().then(UI.updateDoc).then(UI.
updateList);
  }
}

window.onload = function(){
  remoteStorage.access.claim('documents', 'rw');
  remoteStorage.displayWidget();
  app.init('/notes');
  UI.init();
}
```

CONCLUSION

Voilà, nous avons réalisé notre première application utilisant RemoteStorage.js. Pour autant, certaines améliorations peuvent encore être apportées en suivant la philosophie *Unhosted*. En particulier, il pourra être utile d'ajouter un fichier manifeste **Appcache** pour permettre à notre application de fonctionner même sans réseau... Mais c'est une autre histoire.

POUR ALLER PLUS LOIN

Sachez que RemoteStorage.js peut aussi fonctionner avec des services d'hébergement de données propriétaires tels que **GoogleDrive [13]** ou **Dropbox [14]**, moyennant un peu de configuration. Ces fonctionnalités sont, au moment où j'écris ces lignes, en version bêta et ne sont donc pas recommandées pour une utilisation en production. Pourtant, certaines applications les proposent déjà aux utilisateurs. Par ailleurs, on peut y voir un abandon volontaire de l'utilisateur de la confidentialité de ses données. Toutefois, il y a là un moyen d'initier facilement et rapidement des utilisateurs aux applications *unhosted* à partir de comptes déjà existants. ■

RÉFÉRENCES

[1] MOUREY S., *« À la découverte d'une application « unhosted » : Litewrite et PHP RemoteStorage »*, *GNU/Linux Magazine n°196*, septembre 2016

[2] Voir https://github.com/fkooman/php-remote-storage. Par ailleurs, nous avons déjà détaillé l'installation de PHP Remote Storage dans **[1]**

[3] https://5apps.com/storage/beta

[4] https://wiki.remotestorage.io/Servers

[5] https://remotestorage.github.io/modules/files/overview-txt.html

[6] https://github.com/amdjs/amdjs-api/blob/master/AMD.md

[7] https://github.com/remotestorage/modules

[8] Ce code d'insertion vous paraîtra plus complexe qu'à l'accoutumée : c'est que le CDN officiel de JQuery implémente une nouvelle technique qui permet au navigateur de vérifier l'intégrité du code distant appelé *Subresource Integrity* (*SRI*) en cours de déploiement sur les navigateurs. Pour en savoir plus, consultez http://jquery.com/download/#jquery-39-s-cdn-provided-by-maxcdn

[9] J'avais l'intention de vous proposer remoteStorage-Browser, accessible depuis l'adresse https://remotestorage-browser.5apps.com, qui, lors de mes premiers tests, s'était révélé l'outil approprié, mais il ne fonctionne plus actuellement, me semble-t-il à cause de changements récents dans le noyau de RemoteStorage.js, peut-être le problème aura-t-il été corrigé lorsque vous me lirez

[10] À moins d'utiliser le pilote dédié pour *FUSE*, *remoteStorage-fuse* (https://github.com/remotestorage/fuse) qui vous permet de parcourir l'arborescence distante en la montant sur votre système de fichiers local... mais c'est une autre histoire qui dépasse le cadre de cet article.

[11] MOUREY S., *« Bonne pratique : préférer un UID à une clé incrémentale »*, *Linux Pratique n°74*, novembre/décembre 2012, p. 80.

[12] MOUREY S., *« Apprenez à tenir vos promesses avec JavaScript »*, *GNU/Linux Magazine n°195*, juillet-août 2016

[13] https://remotestorage.github.io/remotestorage.js/files/googledrive-js.html

[14] https://remotestorage.github.io/remotestorage.js/files/dropbox-js.html

ACTUELLEMENT DISPONIBLE

GNU/LINUX MAGAZINE HORS-SÉRIE N°89 !

MAÎTRISEZ LA PROGRAMMATION DE SCRIPTS SHELL

NE LE MANQUEZ PAS

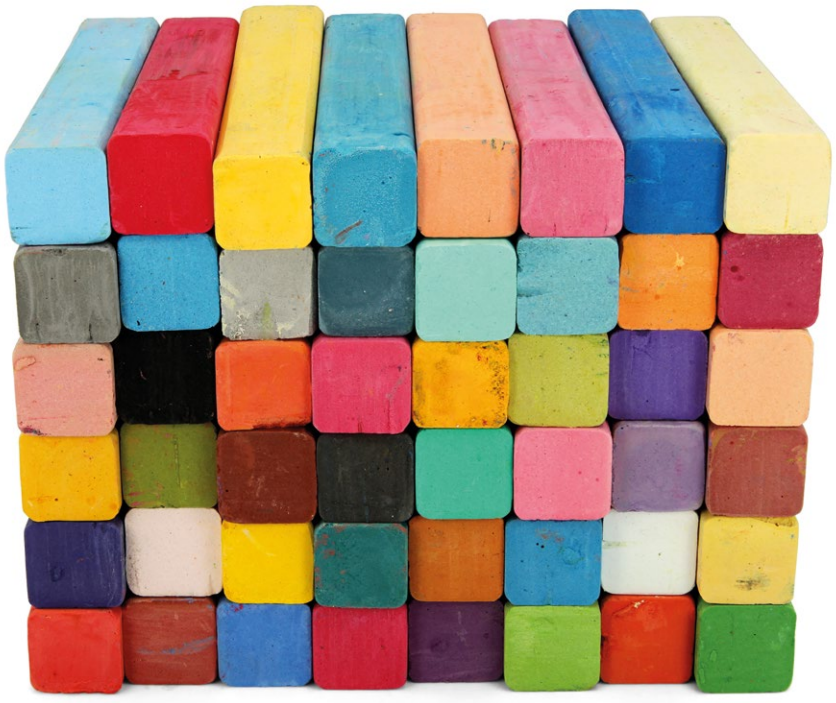
CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

http://www.ed-diamond.com

RETURN ORIENTED PROGRAMMING

SYLVAIN NAYROLLES
[Security Software Engineer]

MOTS-CLÉS : EXPLOITATION, BAS NIVEAU, ASSEMBLEUR, CANARI DE PILE



Les techniques d'exploitation de failles applicatives ont énormément évolué avec l'avènement de moyens de protection toujours plus sophistiqués. Le Return Oriented Programming ou ROP ne permet pas d'injecter du code, mais bien d'exploiter l'existant pour détourner le comportement nominal d'un logiciel via une classe de failles bien particulière.

Le *Return Oriented Programming*, ou ROP, est une technique d'exploitation sophistiquée, reposant sur l'exploitation d'un fonctionnement légitime, d'un « défaut » dit de conception, de la plupart des processeurs, ce qui la rend pérenne. C'est une technique qui met en jeu des notions de bas niveau qu'il faut appréhender pour pouvoir la comprendre et pouvoir ainsi s'en protéger de manière efficace. Nous verrons au travers de cet article que le ROP développe un formalisme avancé, un écosystème riche ainsi que de nombreux outils plus ou moins évolués. Enfin, nous étudierons les contres-mesures efficaces.

1. ASSEMBLEUR

L'exploitation via le ROP est très adhérente au processeur, et donc au jeu d'instructions sur lequel le logiciel ciblé est exécuté. Pour des raisons de tests, nous nous concentrerons sur des programmes s'exécutant sur la famille des processeurs x86, mais sachez que tout ce qui sera dit dans cet article reste valable pour des binaires ARM, PowerPC, SPARC et MIPS.

Pour comprendre le ROP, il faut au préalable comprendre comment une fonction est forgée et appelée au niveau assembleur. Un compilateur **C**, respectant la convention d'appel `ccall`, génère le code x86 suivant :

```
function_name :
 push ebp
 mov ebp, esp
 ...
 mov esp, ebp
 pop ebp
 ret
```

Pour appeler cette dernière, nous allons utiliser l'instruction **call** :

```
push param2
push param1
call function_name
```

Ceci nous permet d'introduire la notion de pile. La pile est une zone mémoire pointée par le registre **esp** et qui est gérée comme une pile par les instructions **push** et **pop**. Mais pas seulement ; les instructions **call** et **ret** vont, elles aussi, manipuler cette dernière. **call** est en fait la composition d'un push de l'adresse contenue dans le pointeur d'instruction (adresse de l'instruction courante : **eip**) et d'un **jmp** vers l'adresse de la fonction. L'instruction **ret** réalise l'opération inverse ; elle permet d'extraire depuis le sommet de la pile une valeur qu'elle va ensuite placer dans le registre de pointeur d'instruction, et ainsi continuer son exécution vers cette adresse.

Mais que se passerait-il si nous arrivions à surcharger la valeur de l'adresse de retour, ici symbolisée par la valeur **@ret**, dans la pile ?

2. STACK OVERFLOW

La *stack overflow* est une classe de failles de sécurité sévère. C'est une classe très recherchée car, comme nous le verrons dans la suite de cet article, elle possède une grande surface d'exploitation.

Le principe repose sur l'écrasement de l'adresse de retour d'une fonction. Dans un cas simple, c'est-à-dire sans aucun moyen de protection moderne, il suffit de surcharger l'adresse de retour par une adresse connue d'une fonction intéressante, telle **system** de la **libc**. Ce type d'exploitation est plus connu sous le nom de **return to libc**.

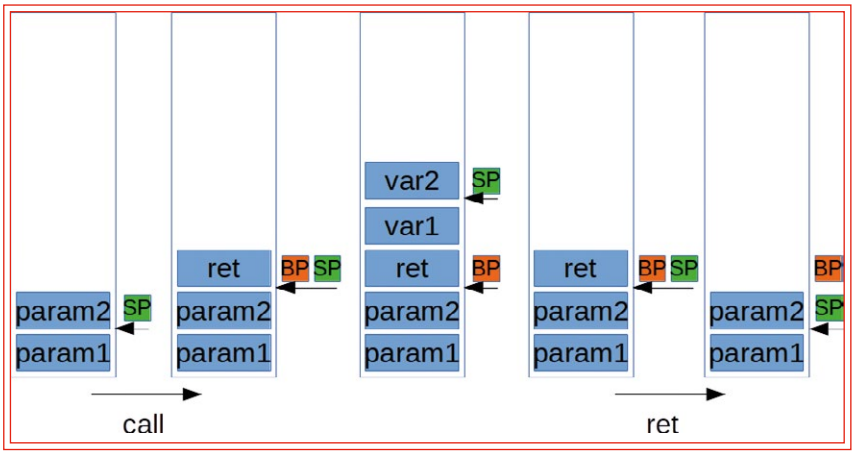


Fig. 1 : Impact des instructions call et ret sur l'état de la pile, ainsi que les prologues et les épilogues d'une fonction.

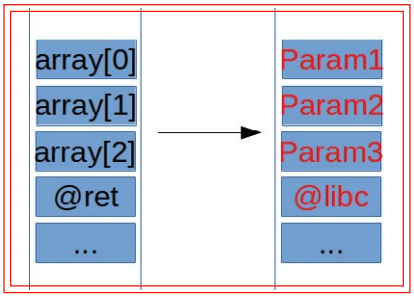


Fig. 2 : Modification de l'état de la pile dans le cadre de l'exploitation d'un stack overflow pour réaliser une attaque de type return to libc.

Mais ce type d'exploitation était envisageable dans la mesure où l'adresse de la fonction de la **libc** était prédictible en amont de l'exécution de l'application vulnérable. Ce type d'exploitation n'est donc plus possible depuis bientôt 10 ans. En effet, des moyens de protections tels que le **DEP** ou bien l'**ASLR** ont permis de considérablement augmenter le niveau de protection du socle d'exécution de cesdites applications. Ceci a rendu complexe l'exploitation de vulnérabilités, et a donc fait émerger des techniques complexes telles que le ROP.

NOTE

Le DEP (*Date Execution Protection*) est une protection active au niveau *hardware* permettant de protéger l'exécution de code depuis des pages mémoires ne contenant que de la donnée, telle que la pile ou le tas, rendant donc complexe l'exploitation de vulnérabilités par injection de code via des failles de type *heap overflow* ou encore *stack overflow*.

L'ASLR est une technique implémentée au niveau OS permettant de charger des segments exécutables à des adresses non prédictibles, rendant donc complexe l'exploitation de failles de type *stack overflow*.

3. ROP

Le principe du ROP repose sur la recherche de *patterns* assembleurs dans les segments exécutables,ainsi que sur l'assemblage de ces derniers, afin de constituer un flux d'exécution, cette fois-ci maîtrisé par l'attaquant. Il faut donc au préalable constituer une *payload* permettant par exemple d'exécuter une commande système (un **syscall**). Il existe de nombreux exemples de *payload* sur Internet pour les architectures x86.

3.1 Gadget

Une fois notre shellcode choisi, nous allons identifier chaque instruction assembleur nécessaire à son exécution. Puis nous allons *parser* notre segment exécutable à la recherche de ces dernières, mais avec la contrainte qu'elles soient immédiatement suivies par une instruction **RET**.

Le ROP va exploiter le fonctionnement de l'instruction **RET** afin de poursuivre son exécution suivant la valeur de l'adresse reposant sur le sommet de la pile à l'instant où cette dernière est appelée. L'adresse de ce bloc est appelée, dans la nomenclature ROP, un gadget. Si nous arrivons à chaîner l'exécution de ces derniers, nous pouvons reconstituer un programme entier, compatible avec une machine de Turing.

NOTE

Un gadget peut contenir plus d'une instruction. Ceci est particulièrement vrai dans le cadre du x86 et de ses dérivées (x86_64) influencés par une architecture de type CISC, car les instructions de ce dernier ne sont pas alignées, au contraire de jeux d'instructions de type RISC, tels que l'ARM, qui possède des tailles d'instructions alignées.

3.2 Chaîne

Une fois que tous nos gadgets sont trouvés, nous allons les chaîner pour rendre notre shellcode actif. Il faut bien comprendre qu'il est possible de ne pas tous les trouver ; plus la base de code est importante, plus il y a de chances d'obtenir nos gadgets.

Nous allons ensuite utiliser notre vulnérabilité pour pousser sur la pile l'ensemble des adresses de nos gadgets pour lancer l'exploitation.

On peut voir le ROP comme une succession d'exploitations récursives d'un *stack overflow*.

Le début de l'exploitation consiste à écraser l'adresse de retour de notre fonction courante par l'adresse du premier gadget. L'instruction **RET** de notre fonction vulnérable va permettre de modifier le flux d'exécution vers le premier gadget. Ensuite, l'instruction associée au gadget sera exécutée et là encore, l'instruction **RET** de notre gadget va permettre de sauter vers l'instruction en sommet de pile, où se trouve l'adresse de notre second gadget, etc.

Ceci constitue une chaîne ROP.

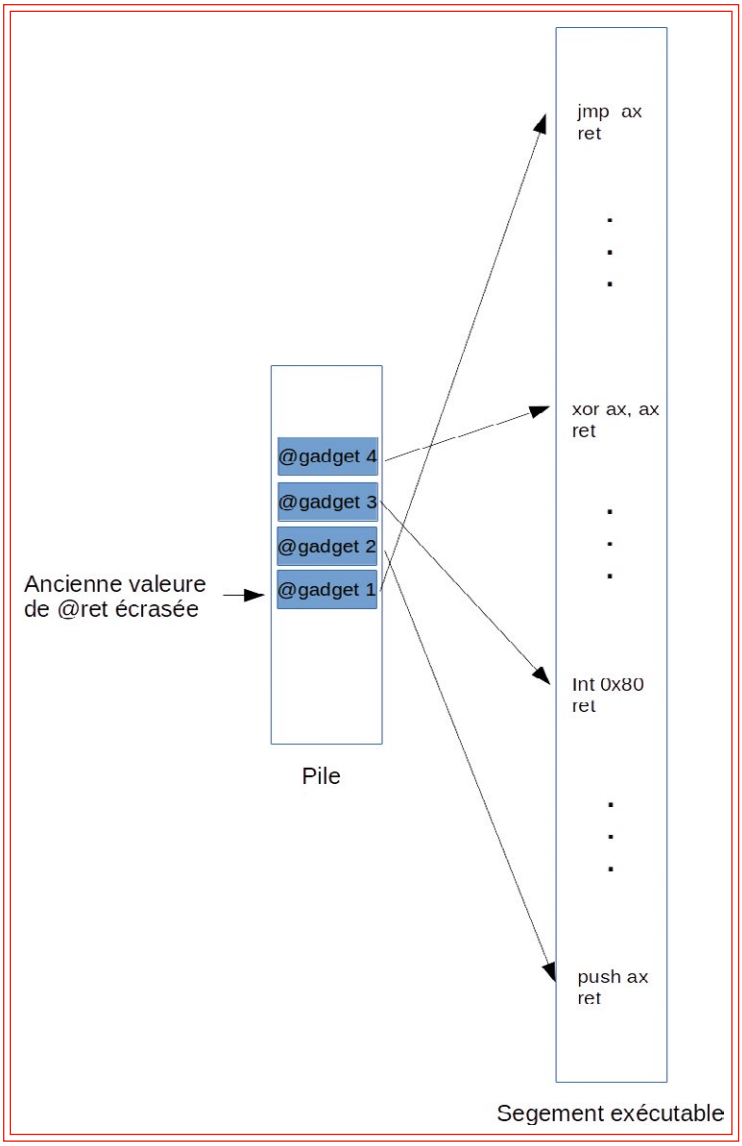


Fig. 3 : État de la pile et du segment de code après exploitation de la vulnérabilité, et donc injection de notre ROP chaîne.

4. EXPLOITATION

L'écosystème ROP possède de nombreux outils et techniques, permettant de faciliter l'exploitation, mais pas de la rendre triviale. Munissez-vous de votre livre d'assembleur, ou du récent article coécrit avec Tristan Colombo dans le numéro 201 de *GNU/Linux Magazine*.

4.1 Outils

Il existe de nombreux outils sur Internet permettant de constituer une chaîne ROP à partir d'un fichier exécutable comme du ELF :

- **Rp++** ;
- **Nrop** ;
- **Ropc** ;
- **ROPgadget**.

Celui de Jonathan Salwan, ROPgadget, est très intéressant, car il est écrit en Python et repose sur le moteur de désassemblage **capstone**. Il va nous permettre de trouver tous les gadgets présents dans un binaire. Si nous installons ROPGadget au sein d'un virtualenv Python, nous pouvons lister tous les gadgets présents dans le binaire python par exemple :

```
> mkvirtualenv ropgadget
(ropgadget)> pip install ropgadget
(ropgadget)> ROPgadget --binary $(which
python)
...
Unique gadgets found: 31212
```

À partir de ce moment, il faut pouvoir trouver une chaîne ROP exploitable dans le binaire, et c'est ici que cela devient difficile.

4.2 Shellcode

`ROPChain` inclut une option `--ropchain` qui permet de chercher une chaîne de type exécution de bash via des gadgets autour de l'interruption `0x80`. Sous Linux, l'interruption `0x80` redonne la main au noyau. Il va utiliser un gadget de type `xor eax, eax` pour initialiser la valeur du registre `eax` à `0`. Puis, il va chaîner un gadget `inc eax` 11 fois pour donner la valeur `11` à `eax`. Enfin, il va utiliser un gadget de type `int 0x80` pour appeler le noyau. Si nous allons voir dans le fichier `arch/x86/include/asm/unistd_32.h` du code source du noyau Linux, le code `11` correspond à un `execve` du `/bin/sh`.

NOTE

Dans Linux, avant l'avènement des instructions de type *fast syscall*, permettant de réaliser des appels aux noyaux, ces derniers étaient réalisés via l'interruption `0x80`. Cette technique n'était pas efficace, car cela impliquait une sauvegarde du contexte obligatoire à chaque appel.

La création de chaîne ROP est un exercice difficile, mais il en existe encore de nombreux exemples sur Internet.

5. CONTRE-MESURES

Il existe bien évidemment des moyens de se protéger de ce type d'exploitation, outre le fait de n'utiliser que des logiciels sans bug...

5.1 Canari de pile

Le canari de pile est une technique permettant de protéger notre application de ce type d'exploitation durant la phase de compilation. Le compilateur va insérer un **canari**, c'est-à-dire une valeur non prédictible, juste avant l'adresse de retour sur la pile, au tout début donc de la fonction. Il va ensuite vérifier cette dernière juste avant l'appel à l'instruction `RET`. Si les valeurs ne correspondent pas, c'est qu'il y a eu un dépassement de pile, et donc que nous devons stopper immédiatement l'exécution du programme. L'instruction `RET` déclenchant l'exploitation n'étant jamais atteinte, le programme est donc protégé de ce type de vulnérabilité.

Cette protection est disponible dans **GCC** via les options `--fstack-protector`. Malheureusement, ce type de protection possède un coût à l'exécution, car il faut pouvoir générer le canari à la volée pour chaque fonction, et le tester. Il faudrait donc pouvoir affiner notre protection qu'aux fonctions possédant une surface d'attaque. GCC propose deux méthodes :

- `--fstack-protector-all` qui va appliquer un canari à toutes les fonctions de votre programme ;
- `--fstack-protector` qui ne va appliquer cette protection qu'aux fonctions possédant un tableau de caractères sur la pile ayant une taille supérieure à 8. Une heuristique simple, mais efficace. La taille du tableau est paramétrable via l'option `--param=ssp-buffer-size=N`.

Plus récemment, un nouveau mode fut introduit dans la version 4.9 de GCC : `--fstack-protection-strong`. Cette dernière change l'heuristique, en l'étendant aux fonctions comportant des tableaux sur la pile autres que ce type de caractère, ainsi qu'aux fonctions faisant référence à une adresse d'une variable locale, c'est-à-dire une adresse sur la pile. Cette nouvelle heuristique fut introduite par l'équipe sécurité de **Chrome OS** qui utilisait anciennement l'option `--fstack-protector-all`. Afin d'évaluer leur heuristique, ils l'ont comparé à l'ancienne en l'appliquant sur le noyau Linux 3.14 :

- `--fstack-protector` provoque une augmentation de la taille du code de 0.33 % pour une couverture de 2.81 % des fonctions ;
- `--fstack-protection-strong` provoque une augmentation du volume de code de 2.4 % pour une couverture de code de 20.5 % des fonctions.

Ceci constitue une alternative très intéressante entre une couverture totale et minimaliste.

5.2 ASLR

Nous avons déjà eu l'occasion de parler de l'ASLR dans cet article. L'ASLR, pour *Adress Space Layout Randomization*, permet de charger les segments exécutables à des adresses non prédictibles, et donc dans notre cas d'usage, nous empêche de générer notre ROP chaîne. Par défaut, sous Linux, l'ASLR est activée, mais souvent n'est utilisée que dans l'espace mémoire des bibliothèques dynamiques, souvent symbolisées par l'option `-fPIC` de GCC. Mais les segments exécutables principaux sont rarement *randomisés*. Il faut pour cela utiliser l'option `-pie` de GCC. L'ajout de cette option n'est pas sans impact sur les performances de chargement de l'application ; c'est pour cela que seulement certaines applications critiques sont compilées avec cette option, telle que `sshd`. Mais ceci constitue l'une des protections les plus efficaces contre ce type d'attaque.

CONCLUSION

Le *Return Oriented Programming* est une technique avancée d'exploitation ; elle constitue une technique élégante. Mais elle nécessite des notions avancées en assembleur, et elle ne garantit pas le fait d'être exploitable à 100 % si la constitution d'une chaîne n'est pas possible. De plus, les OS et compilateurs modernes mettent en place des contre-mesures efficaces annihilant totalement la surface d'attaque de ce type d'exploitation. Malheureusement, ces options ne sont pas encore des options par défaut et nécessitent une expertise pour les comprendre et les appliquer. La maîtrise des canaris de pile ou de l'ASLR doit devenir la connaissance de tout bon développeur. ■

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 et privé 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

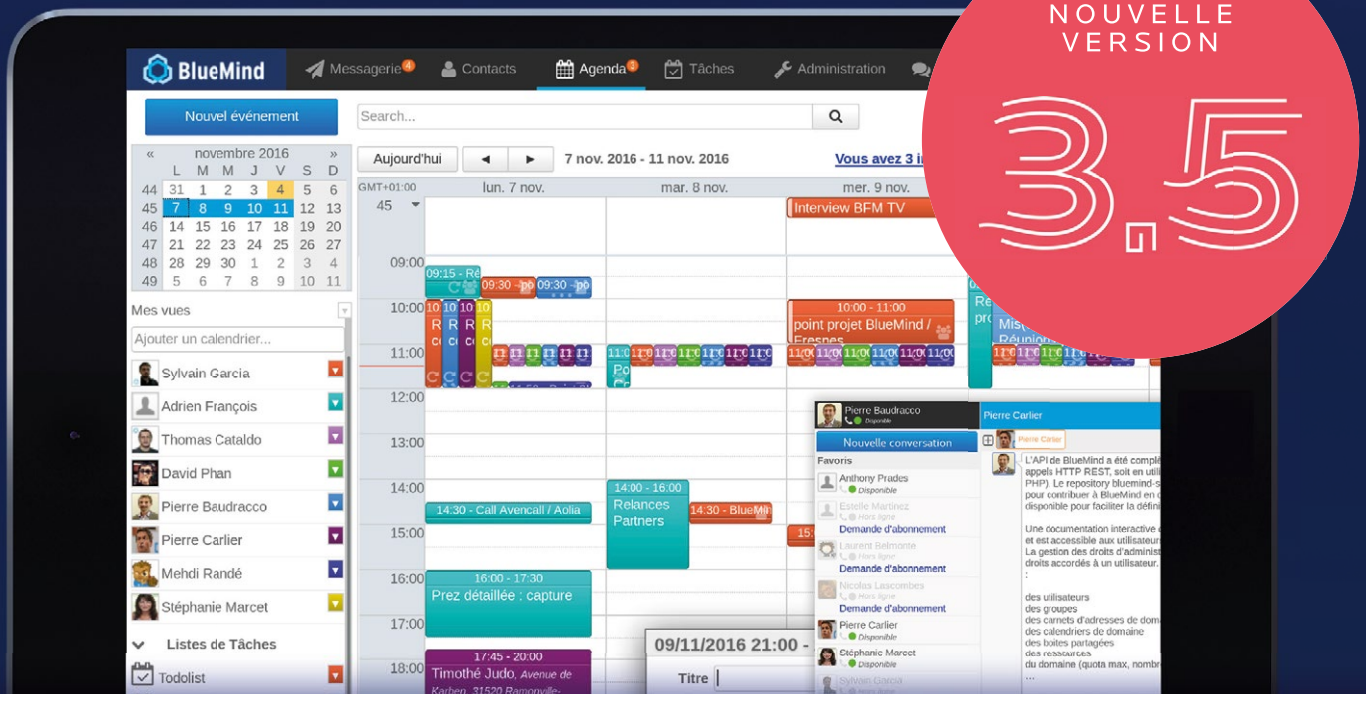
ikoula HÉBERGEUR CLOUD

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