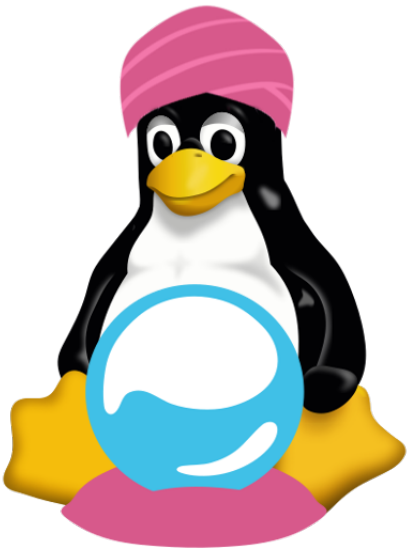
Projet TuxML

Corentin Chédotal Gwendal Didot Dorian Dumanget
Erwan Le Flem Pierre Le Luron Alexis Le Masle
Mickaël Lebreton Fahim Merzouk

Encadrés par Mathieu Acher

22 Décembre 2017



Résumé

Le système d'exploitation Linux est omniprésent (sur nos ordinateurs, téléphones, tablettes, ou objets connectés) et ce grâce aux contributions de milliers de développeurs. Cependant son impressionnante versatilité rend sa configuration difficile avec plus de 14000 options à choisir. Le but du projet TuxML est d'aider à la configuration du noyau Linux par l'emploi du *Machine Learning*. L'approche consiste à compiler un large ensemble de configurations et de déterminer, par apprentissage, quelles sont les options du noyau à absolument activer ou désactiver pour atteindre un certain objectif (e.g., une taille du noyau inférieure à 50Mo).

Dans ce but, une série de scripts en Python ont été réalisés et une infrastructure a été mise en place afin de collecter et organiser les données. La commande `tuxml` instrumente le tout, afin de compiler de très nombreuses configurations par le biais d'un conteneur Docker pour fournir une portabilité et ainsi pouvoir compiler le noyau Linux sur différentes machines. Nous détaillons dans ce document la commande `tuxml` ainsi que les différents scripts

utilisés (e.g., pour installer tout paquet manquant si un problème se présente à la compilation, pour générer une configuration aléatoire).

Des alternatives comme Autotest pour la compilation du noyau Linux ou JHipster pour la collecte de données ont été étudiées et leur non-emploi est explicité. Enfin nous discutons des travaux futurs et notamment comment notre travail actuel va alimenter les techniques de *Machine Learning*.

Remerciements

L'équipe du projet TuxML souhaiterait remercier monsieur Olivier Barais pour avoir pris le temps de présenter Docker et son utilisation ainsi que pour les pistes de travail concernant son emploi dans le cadre de notre projet.

Le logo du projet utilise des *emojis* modifiés du projet Noto mis à disposition gracieusement sous la licence Apache. Seul l'emoji `1f473 1f3fb` a été modifié avant son inclusion dans le logo afin d'en extraire et recolorer son turban.

Table des matières

Première partie
Enjeux et état de l'art

1 Linux

1.1 Un logiciel inégalé

Souvent présenté dans l'imaginaire commun comme le système d'exploitation "geek" par excellence, Linux est en réalité bien plus que ça.

1.1.1 Précision et présentation

Il apparaît important de noter que Linux ne constitue pas en lui-même un système d'exploitation mais seulement le noyau ou *kernel*. Les systèmes d'exploitation étant alors complétés par différents logiciels comme ceux du projet GNU dans les distributions *GNU/Linux* (celles que l'on appelle couramment Linux). Cette nuance est particulièrement importante puisqu'ici ce projet s'intéresse exclusivement à la compilation de ce noyau.

Ainsi par "Linux" nous évoquerons ici bien le noyau en tant que tel, ses sources et ses fichiers binaires post-compilation.

Le noyau Linux est développé à l'origine par Linus Torvalds en 1991 durant ses études universitaires. Se basant sur le système UNIX existant déjà à l'époque il souhaite pouvoir profiter pleinement des capacités de son propre ordinateur personnel et développe ainsi le futur noyau Linux. Torvalds décide très vite de publier sa création sous la licence GNU GPL, une licence logicielle libre qui va rendre le noyau accessible à tous tant dans l'emploi que l'édition ou l'adaptation.

1.1.2 Le fleuron du logiciel libre

Ainsi le noyau Linux n'est pas qu'un élément charnière du développement logiciel mais aussi et surtout le projet *open source* le plus actif et ayant le plus de contributions. On peut par exemple noter que depuis que son code source est disponible sur la plate-forme de développement collaboratif *GitHub* plus de 720000 *commits* (ou modifications du code) ont été faites sur sa branche de travail principale. Au delà de simplement Linus Torvalds, le nombre de contributeurs distincts sur le site est tel que le compteur de celui-ci à été bloqué à la valeur abstraite "∞". Enfin, *GitHub* affiche plus de 19000 *forks* (ou copies totales) éditées par d'autres acteurs et équipes. Or ceci ne tient compte que des *forks* mis en ligne sur GitHub et non pas sur des dépôts moins visibles.

1.1.3 Un noyau utilisé par tous

Pour finir de tordre le cou à cette idée reçue que le noyau Linux n'est que peu utilisé ou seulement par des informaticiens barbus dans leurs caves, il suffit de souligner le déploiement effectif du noyau Linux.

En effet on peut d'abord noter qu'une très grande majorité des serveurs utilisent ou sont basés sur le noyau Linux. Ces mêmes serveurs qui sont utilisés pour Internet, les messageries mais aussi plus simplement pour les intranets d'entreprises ou universitaires. À ce fait s'ajoute aussi la présence du *kernel* Linux voire de systèmes GNU/Linux complets dans les box internet utilisées par tous. Ainsi on peut citer la *Freebox*, la *Neufbox* de SFR ou encore la *Livebox* d'Orange juste pour la France à titre d'exemple. Ensuite les terminaux mobiles employant *Android* représentaient 85,0% des parts du marché au premier trimestre 2017. Or ce système d'exploitation mobile est basé sur Linux bien qu'il en emploie une version très modifiée. Enfin la croissance presque exponentielle de l'*Internet of Things* et des objets connectés rapproche une fois de plus Linux de l'emploi de tous les jours puisque pour réduire les coûts de développement bon nombre de ces objets emploient, de manière parfois brute, le noyau Linux.

Ce faisant, avoir un impact sur la compilation du noyau, sur les choix quant à celle-ci ou pouvoir impacter des décisions pouvant amener à une amélioration du *kernel* aurait potentiellement des conséquences très étendues et visibles à de très nombreuses échelles.

1.2 Généralités sur la compilation du noyau

Avant toute compilation d'un noyau Linux, il faut commencer par spécifier quelles options seront présentes dans le noyau une fois celui-ci compilé. Afin de spécifier ces options, le *kernel* se base un fichier particulier : le `.config`. Ce fichier regroupe l'ensemble des options activées (en dur ou en module) dans le noyau à générer.

Pour générer ce fichier il existe plusieurs méthodes. La première méthode, et la plus utile pour ceux voulant réaliser leur propre Linux, est la commande `make menuconfig`. En effet, cette commande permet de choisir les options du *kernel* dans un menu assez simple. Bien que cette méthode soit fastidieuse, elle permet d'être sûr des options présentes dans son noyau. Il existe d'autre commandes permettant de réaliser la même chose avec une interface plus graphique, mais ces commandes ne seront pas présentées ici. La deuxième méthode quant à elle permet de générer un `.config` sans aucune garantie sur les options activées. La commande utilisée est `make randconfig`. C'est cette dernière méthode que nous utilisons dans ce projet pour la génération de nos noyaux Linux. Nous présenterons plus en détail cette commande plus bas dans le rapport.

Une fois le `.config` prêt, il ne reste plus qu'à le compiler pour produire le noyau Linux avec les options que vous avez choisi. Il suffit pour cela d'utiliser la commande `make`.

Extrait de `.config` :

```
CONFIG_64BIT=y # Booleen (y,n)
CONFIG_OUTPUT_FORMAT="elf64-x86-64" # Chaine de caracteres
CONFIG_ARCH_MMAP_RND_BITS_MIN=28 # Nombre entier
CONFIG_X86_MSR=m # Tristate (y,m,n)
```

2 *Machine Learning*

2.1 Présentation générale

Le *Machine Learning* que l'on pourrait traduire par "Apprentissage automatique" est un domaine d'étude de l'intelligence artificielle. Il intervient lorsque le nombre de données à analyser est trop important à traiter, et ce, même avec des algorithmes classiques. On parle alors d'explosion combinatoire.

Les algorithmes de *Machine Learning* permettent d'analyser ces données et d'effectuer des corrélations entre les différents paramètres. Puis l'algorithme va fournir une réponse en fonction de ses analyses.

Actuellement le *Machine Learning* est très utile dans la reconnaissance d'objets (visages, véhicules, écriture manuscrite) ou les moteurs de recherches par exemple.

2.2 Intérêt du *Machine Learning*

La configuration du noyau Linux est très compliquée. Certaines options sont dépendantes les unes des autres ou alors incompatibles entre elles, qui plus est la documentation n'est pas très explicite. Certaines fois le noyau compile mais ne démarre pas. Ainsi, comment choisir les bonnes options, parmi les 14 000 disponibles, pour avoir un noyau qui réponde à nos critères (démarrage rapide ou noyau léger par exemple) ?

Il est impossible de tester les 2^{14000} options, cela consomme du temps et des ressources. C'est donc dans ce contexte qu'intervient le *Machine Learning*. Il est nécessaire de prédire les comportements des options à partir d'un échantillon restreint de configurations et d'apprendre sur cet échantillon.

Le processus se déroule en quatre étapes :
- sampling : il s'agit de générer un échantillon de configurations et des les stocker dans une base de données
- testing : on réalise un ensemble de tests sur l'échantillon généré à l'étape précédente, tels que : est-ce que le noyau démarre ? En combien de temps ? Quel taille fait-il ? etc...
- learning : on détermine des corrélations entre les options et les test réalisés
- on recommence l'étape de sampling avec nos nouvelles données

Aujourd'hui nous sommes à la première étape, nous avons un programme capable de compiler un noyau et de gérer les erreurs durant la compilation. Il s'agit maintenant de le déployer à très large échelle pour avoir un échantillon suffisamment grand sur lequel lancer des tests.

Deuxième partie
Mise en place

3 La configuration

3.1 Architecture du `.config`

Comme expliqué précédemment, le `.config` est la base de toute compilation. Ce fichier contient la liste des options qui seront activées, en dur ou en module, lors de la compilation. Ce fichier a toujours la même architecture de la forme `NOM_DE_L'OPTION=VALEUR`.

3.2 Fonctionnement de `make randconfig`

Afin de générer un grand nombre de `.config` et de façon à ce que ceux-ci soit le plus distincts possibles, nous utilisons une fonctionnalité du *kernel Linux*, la commande `make randconfig`.
Cette commande permet de générer une configuration du *kernel* dont les options sont choisis aléatoirement : avec 50% de chance d'être activée pour les options dont uniquement 2 choix sont possibles (OUI ou NON) et 33% de chance d'être activée en dur et 33% de chance d'être activée en module pour les options qui possèdent 3 choix (OUI, NON ou MODULE). Cette commande se base sur le temps Unix de la machine pour générer un nombre en hexadécimal afin de construire le `.config` ce qui limite les doublons.
Il existe pour cette commandes différentes variables permettant de "limiter" l'aléatoire :

- KCONFIG_ALLCONFIG : permettant de spécifier un chemin vers un `.config` déjà existant afin de récupérer les valeurs déjà présentes dans ce dernier puis de relancer un choix aléatoire sur les autres options.
  La syntaxe devient alors : `KCONFIG_ALLCONFIG=path/to/my.config make randconfig`
- KCONFIG_SEED : permettant de spécifier un nombre en hexadécimal afin de retrouver un `.config` donné. Le choix des options ne sera plus basé sur le temps Unix actuel mais sur l'hexadécimal fourni.
  La syntaxe devient alors : `KCONFIG_SEED=0x1234abcd make randconfig`
- KCONFIG_PROBABILITY : permettant de modifier la probabilité d'activer les options (en dur ou en module). Permet par exemple de changer la probabilité d'obtenir un OUI pour les choix doubles de 50% à 80% sans modifier la probabilité des choix triples.
  La syntaxe devient alors : `KCONFIG_PROBABILITY=80 make randconfig` (80% OUI pour les doubles choix et 40% OUI DUR et 40% OUI MODULE pour les choix triples). Pour cette syntaxe la valeur doit être compris entre 0 et 100
  Ou alors : `KCONFIG_PROBABILITY=75:5 make randconfig` (80% OUI pour les doubles choix et 75% OUI DUR et 5% OUI MODULE pour les choix triples). Pour cette syntaxe la somme des valeurs doit être compris entre 0 et 100

Ou enfin : `KCONFIG_PROBABILITY=80:33:33 make randconfig` (80% OUI pour les doubles choix et 33% OUI DUR et 33% OUI MODULE pour les choix triples). Pour cette syntaxe la 1ère valeur doit être compris entre 0 et 100 et la somme des 2 et 3e valeurs doit être compris entre 0 et 100.

4 Compilation du noyau

4.1 Gestion des dépendances

Le noyau Linux requiert un nombre non négligeable de dépendances pour pouvoir être compilé. Puisque nous voulons que le processus soit le plus automatisé possible, on doit gérer l'installation automatique de ces dépendances.

Les difficultés ici sont :

1. Le nombre important de distributions différentes avec leurs propres gestionnaires de paquets.
2. Plusieurs dépendances n'ont pas le même nom de paquet en fonction des distributions.
3. Les paquets requis peuvent dépendre des options sélectionnées. Il est difficile de connaître à l'avance une liste parfaitement exhaustive de dépendances suffisantes dans tout les cas. (voire partie analyse de logs).

La gestion des dépendances consiste d'abord à détecter le gestionnaire de paquet installé. C'est plus simple que de passer par une détection d'une distribution en particulier.

Une fois le gestionnaire de paquet détecté, on utilise les commandes liées à ce gestionnaire (pour la synchronisation avec les dépôts par exemple). On installe la liste des dépendances minimales, qui contient les outils qui seront forcément requis pour la compilation quelque soient les options activées (make, gcc, etc...)

Pour faciliter la maintenance, on utilise un système basé sur des tables de hachage (Map) : à partir du nom d'un gestionnaire de paquet, on peut récupérer la liste des commandes à invoquer et un tableau des paquets à installer.

Dans un premier temps on considère que le gestionnaire de paquets nous donne la distribution, par exemple si le gestionnaire apt-get est détecté on considère la liste des paquets de Debian. C'est une approche potentiellement limitée car il est possible que les paquets d'une sous-distribution ne soient pas forcément les mêmes que ceux de la distribution mère. Ceci dit le but n'est pas de faire quelque chose de compatible avec les centaines de distributions existantes mais surtout avec les distributions phares qui ne sont pas si nombreuses que ça.

Pendant le projet, la gestion des dépendances est simplifiée par l'utilisation de container Docker : puisque le container est un système Debian, on se concentre surtout sur la compatibilité avec cette distribution pour l'instant.

4.2 Compilation et analyse de logs

Après avoir généré un .config et installé les dépendances minimales le programme TuxML doit lancer la compilation. Nous avons développé un ensemble de scripts

pour automatiser ce processus.

- tuxml.py : le fichier principal, contenant la fonction `main` et effectue l'analyse de logs
- tuxml_common.py : contient diverses fonctions communes aux fichiers tuxml_*, telles que les fonctions d'installations de paquets et de mise à jour des dépôts linux
- tuxml_settings.py : initialise les paramètres par défaut de TuxML (dossier de logs, niveau de verbosité, nombre de coeurs utilisés...)
- tuxml_depman.py : gère les dépendances manquantes
- tuxml_sendDB.py : envoie les résultats de la compilation à la base de donnée
- tuxml_environment.py : permet de récupérer des informations sur le système (distribution, processeur, version de la libc...)

Par défaut le programme TuxML génère un fichier de configuration aléatoire, mais il est aussi possible, suivant les paramètres donnés à la commande `./tuxml.py`, de générer un fichier de configuration spécifique à partir du `KCONFIG_SEED`, voir même de donner le chemin vers le .config. Il s'ensuit la phase d'installation des dépendances, expliquée ci-dessus. Puis la compilation à proprement parlé peut commencer.

Il arrive que la compilation échoue, souvent dû à l'absence d'un paquet ou d'un fichier spécifique. En effet, même après la phase initiale d'installation des dépendances il est toujours possible que certains paquets soient manquants. Cela dépend des options qui ont été activées pendant la génération du fichier de configuration.

Dans ce cas le programme analyse les logs d'erreur pour déterminer la cause de l'échec. Cette étape nous a posé quelques problèmes, il s'agissait de parser le contenu du fichier `err.log` dans lequel était redirigée la sortie d'erreur de la compilation. Les erreurs n'était pas toutes formatées de la même façon et il était donc difficile de faire une liste exhaustive des formats d'erreurs. Par exemple on a rencontré les formats suivants :

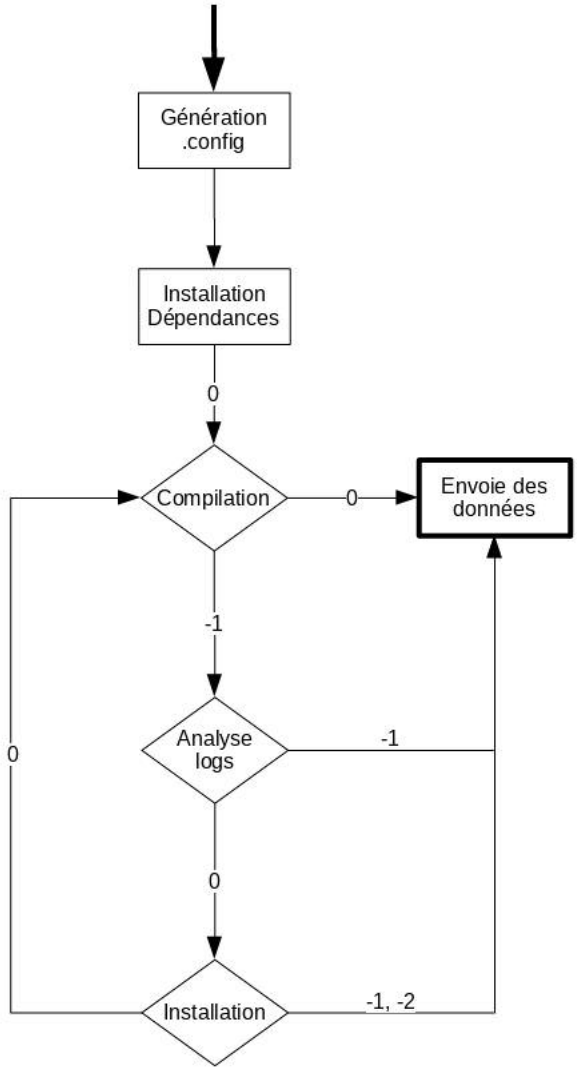
- `<file.c>:<nombre>:<nombre>: fatal error: <file.h>: No such file or directory`
- `make[<number>]: <command>: command not found`
- `/bin/sh: <number>: <command>: not found`
- `./scripts/gcc-plugin.sh: <number>: ./scripts/gcc-plugin.sh <package>: not found`

Si l'échec est dû à un paquet ou un fichier manquant tuxml doit trouver et installer les dépendances requises puis reprendre la compilation. Sinon on considère que l'échec ne peut pas être corrigé et on arrête la compilation.

Par exemple, sur Debian, on a utilisé la commande `apt-file search <nom_du_fichier>` qui renvoie une liste de paquets contenant le fichier manquant. Le programme vérifie si le premier paquet de la liste est installé : si oui il passe au paquet suivant, si non il l'installe et reprend la compilation. S'il arrive en fin de liste sans avoir installé de paquets alors tuxml génère une erreur et la compilation s'arrête.

Quelque soit le résultat de l'analyse de logs ou de l'installation de paquets, on envoie le résultat à la base de données.

FIGURE 1 – Représentation du fonctionnement de tuxml. Zéro correspond à un succès et un chiffre négatif à un échec



5 Déploiement

5.1 Docker

Dans l'optique de remplir notre base de données, il est nécessaire de pouvoir exécuter notre commande principale de compilation sur toutes les machines linux indépendamment de leur architecture.
C'est donc ainsi que, sur les conseils de Monsieur Olivier BARAIS, nous avons choisi d'utiliser le logiciel Docker et donc de créer une image Docker servant de base à toutes nos compilations.

5.1.1 Image intermédiaire

Pour des facilités d'utilisation, nous avons décidé de mettre en place une image docker intermédiaire. Cette image, créée à partir de l'image `debian:latest`, se trouve sur notre repository distant https://hub.docker.com/r/tuxml/debiantuxml/. Elle contient le dossier du *kernel Linux* que l'on utilise, et installe des paquets essentiels utilisés par `MLFood.py`. Cela nous permet de réduire la taille de l'image `tuxmldebian`, et ainsi de réduire le temps pris par le build et le push de cette image.

5.1.2 Debian TuxML

L'utilisation de Docker nous permet d'écrire et d'exécuter nos commandes et scripts sur tous les types d'architecture car nous simulons l'utilisation d'un système Debian. Cette image est construite à partir de l'image intermédiaire Cette image Debian modifiée contiendra aussi tous les scripts python écrit par les membres du projet et nous servira d'image de référence qui sera utilisée à chaque lancement d'un nouveau conteneur docker par le script MLfood. Elle est située sur le repository distant suivant : https://hub.docker.com/r/tuxml/tuxmldebian/

5.1.3 MLfood

Le script `MLfood.py` est à la base de la chaîne d'exécution permettant de remplir la base de données, d'où le nom *MLfood – Machine Learning Food –*.

Il s'agit d'un script python permettant de lancer un nombre donné de compilation à faire à la suite les unes des autres par la création d'un nouveau conteneur docker pour y exécuter le script `tuxml.py`, l'image docker utilisée est la dernière version de l'image Debian modifiée pour le projet TuxML et sera donc mise à jour à chaque lancement du script `MLfood.py`. Étant donné que *MLfood* télécharge la dernière image Debian, ce script est le seul à avoir besoin d'être téléchargé afin de lancer des compilations qui entreront dans la base de données.

`tuxLogs.py` est un script intermédiaire qui se situe dans l'image debiantuxml et qui exécute `tuxml.py` tout en sauvegardant les logs, c'est ce script qui est lancé au démarrage d'un nouveau conteneur durant l'exécution de `MLfood.py`.

Le script `MLfood.py` a été créé afin de permettre le déploiement sur plusieurs machines de toute l'architecture des fichiers et dépendances du projets pour l'exécution. De cette manière lors d'un déploiement potentiel de *MLfood* sur une grille de calcul, chaque machine exécutant `MLfood.py` téléchargera la dernière image personnalisée de Debian spéciale TuxML et lancera un nouveau conteneur Docker qui exécutera `tuxml.py`, le script de compilation général, et donc provoquera l'envoie des données à la base de données.

Pour résumer cette section, `tuxml.py` permet de lancer une compilation tandis que `MLfood.py` permet de lancer `tuxml.py` un nombre de fois donné grâce au script `tuxLogs.py` qui lance TuxML en sauvegardant les logs.

5.2 Autotest

En parallèle de l'étude de faisabilité de l'emploi de Docker l'utilisation d'autres outils très communs dans la communauté du développement du noyau Linux étaient envisagés. Plusieurs ne furent pas utilisés de par l'absence de documentation ou, étant propriétaires, étaient inaccessibles. Cependant ce ne fut pas le cas d'Autotest que l'on pu tester et mettre en œuvre dans un environnement contrôlé.

5.2.1 Présentation

Autotest est une suite logicielle open-source tout-en-un de tests de *kernel Linux* expérimentaux. Elle permet de récupérer des sources spécifiques du noyau Linux, de récupérer un `.config` donné, de compiler les sources selon la configuration donnée, de déployer le nouveau noyau sur une machine et d'exécuter une batterie de test sur celle-ci.

5.2.2 Étude de documentation et de fonctionnement

Toutes ses fonctionnalités impliquent donc un système assez lourd avec de très nombreux scripts et commandes à employer. Ainsi une grande partie du temps consacré à Autotest l'a d'abord été à sa documentation. Celle-ci se présentait cependant très verbeuse et il fut nécessaire de réaliser de nombreux tests sur une machine virtuelle pour commencer à pouvoir appréhender les détails de son fonctionnement. Autotest permet à l'utilisateur de définir par le biais d'un fichier de configuration particulier la conduite à tenir pour une série d'action en particulier. C'est à dire que l'on va pouvoir indiquer justement quelle source du noyau Linux employer et où les récupérer, de même pour le `.config`. Mais on va indiquer aussi quels tests réaliser et dans quel ordre.

Cependant le lecteur pourra noter qu'il n'est pas précisé que l'on peut choisir la machine sur laquelle le déploiement du noyau et les tests ont lieu. En effet dans son utilisation la plus simple Autotest va remplacer le noyau de la machine sur lequel il tourne par celui à tester. Or si celui-ci ne fonctionne pas la machine devient bien inutilisable. Heureusement nos tests ont eu lieu sur machine virtuelle mais ce comportement était inacceptable puisque complètement incompatible avec une mise à l'échelle. D'autant plus qu'il était impossible de récupérer une indication que les tests avait échoué et que le noyau ne fonctionnait pas puisque la machine ne démarrait plus.

Il est évident que ce système n'est pas celui courant d'Autotest et utilisé par divers groupes et entreprises. En réalité dans son fonctionnement classique la suite logicielle fonctionne sur un modèle machine maître–esclaves. Une machine recevant les instructions avec le fameux fichiers et les redistribuant à ses machines esclaves pour qu'elles soient exécutées. Une fois les résultats obtenus les machines esclaves les renvoient à la machine maître qui les agrègent et éventuellement effectue des moyennes avant de les mettre à disposition de l'utilisateur. De plus ce système permet de détecter les noyaux ne fonctionnant pas puisqu'en cas de non-réponse d'un nombre statistiquement signifiant de machines esclaves le maître peut déterminer que le noyau ne fonctionne pas et forcer la remise en place d'un noyau fonctionnel

sur les machines esclaves.

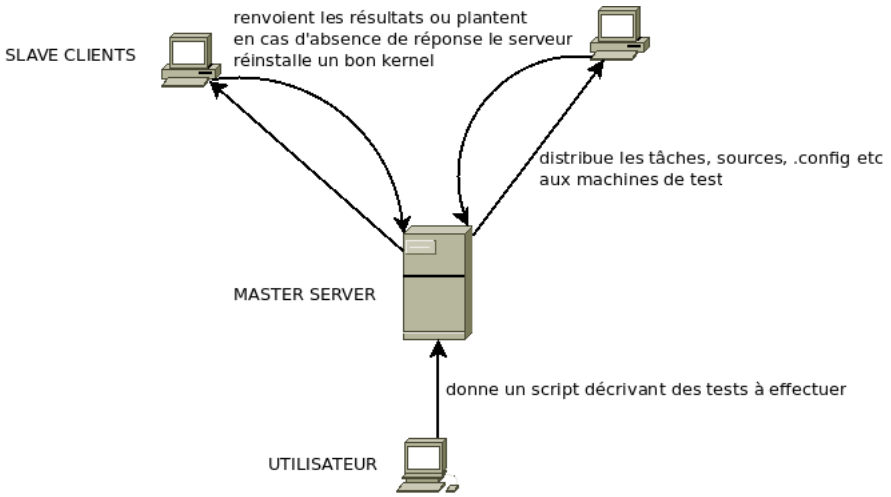


FIGURE 2 – L'architecture maître–esclave d'Autotest

5.2.3 À propos de son non-utilisation

Malgré les avantages réels ainsi que la puissance de cet outil nos expérimentations ne furent pas concluantes. Les raisons les plus importantes de ce choix seront explicitées ci-dessous.

La dépendance des paquets était encore un problème Tout d'abord un premier point gênant qui a pu être réglé en passant par notre propre système est celui de la dépendance et du téléchargement des paquets manquant. La compilation automatique par Autotest semble partir du principe que la machine sur laquelle elle est exécutée possède bien tous les paquets nécessaires à la compilation. Or de par l'aspect aléatoire nécessaire à notre approche de résolution par le *Machine Learning* il nous est difficile de prévoir les paquets à installer en avance. Ainsi avec Autotest des compilations seront couramment des échecs nécessitant l'intervention d'un humain pour installer les paquets demandé par le compilateur. Ceci réduisant une fois de plus la possibilité de mise à l'échelle sans au préalable installer une immense quantité de paquets, ce qui nécessite beaucoup de préparation en amont lors du déploiement et qui ne garantit en rien que toutes les compilations se réalisent.

Une architecture logicielle trop lourde Une autre particularité très contraignante d'Autotest est que malgré l'accessibilité de son code directement sur *GitHub*, il n'est que très rarement commenté. De plus quand il l'est, la documentation n'est pas toujours très claire ou semble partir de certains pré-requis que nous n'avions pas. Enfin Autotest est une suite logicielle, de très nombreux scripts et fichiers tous imbriqués les uns dans les autres et nécessitant des interactions très particulières entre eux. Ainsi tous ces facteurs rendaient caduque nos efforts de personnalisation de la suite pour l'adapter à nos besoins. D'autant plus que la documentation très fournie était principalement axée pour un utilisateur plutôt qu'un développeur/contributeur.

Un modèle maître–esclave pas forcément adapté Le modèle nécessaire au bon fonctionnement d'Autotest requiert de très longues et nombreuses étapes en amont de tout déploiement d'Autotest. Son installation sur les machines peut ne pas forcément être aisée et finalement son emploi le rendait très restrictif puisque tout était réalisé suivant des étapes très précises qu'il était difficile d'éditer comme indiqué plus haut. Ainsi Autotest force un type d'architecture particulier, architecture avec laquelle nous n'étions pas confortable et surtout pour laquelle nous ne pouvions garantir la mise en œuvre dans le cas de mise à l'échelle sur des grille de calcul. Enfin, cette infrastructure et ce fonctionnement avec plusieurs machines commençaient à s'éloigner de l'idée de simplicité d'utilisation que nous voulions pour le projet.

Un système particulier pour réaliser un test Enfin bien qu'Autotest vienne déjà avec de très nombreux tests sur beaucoup de métriques et que soit mis à la disposition des tests réalisés par la communauté, l'écriture de ses propres tests semblait particulièrement ardue. Ceci étant d'autant plus dû à un manque assez flagrant de documentation à ce sujet.

6 Collecte des données

6.1 Objectif

Une fois que le script a fini la compilation, nous voulions garder les résultats obtenus pour pouvoir les utiliser plus tard, lors de l'étape de *Machine Learning*. Pour cela, il nous fallait une base de données ainsi qu'un moyen de l'alimenter depuis nos scripts.

6.2 Une première tentative : JHipster

JHipster est un système permettant la mise en place de sites internet de façon rapide et efficace. Ce programme génère de façon automatique le front-end et le back-end. Il permet aussi l'utilisation d'une API pour interagir avec la base de données qu'utilise l'application. Pour ce faire, on utilise la commande suivante : `jhipster`. Il suffit ensuite de suivre les directives du programme pour pouvoir générer l'application. Puis on importe une *Entity* (soit une table de base de données) pour pouvoir utiliser la base de données de l'application. On utilise la commande suivante pour le faire : `jhipster import-jdl entity.jh`

Certaines difficultés concernant JHipster ont été constatées ; JHipster est conçu pour une utilisation avec Java, alors que nous utilisons Python. Beaucoup de types de la base de données sont des objets Java sérialisables et sont difficilement manipulables dans nos scripts. L'API HTTP est aussi très limitée, et ne permet pas la création de requêtes complexes, ou de routines.

6.3 État actuel

Un serveur LAMP a donc été mis en place pour pallier aux problèmes avec la base de données de JHipster. Le script d'envoi des données a donc été changé pour utiliser MySQL, et chaque compilation remplit notre table avec les données suivantes :

— la date d'envoi ;
— le fichier de configuration pour la compilation ;
— le temps de compilation ;
— la taille du noyau compilé ;
— un log d'erreur si la compilation a échoué ;
— une éventuelle liste de paquets à installer pour effectuer la compilation.

On peut donc maintenant lancer TuxML de façon automatique pour remplir la base de données. Les données seront utilisées pour générer un fichier `.csv` nécessaire pour l'étape de *Machine Learning*.

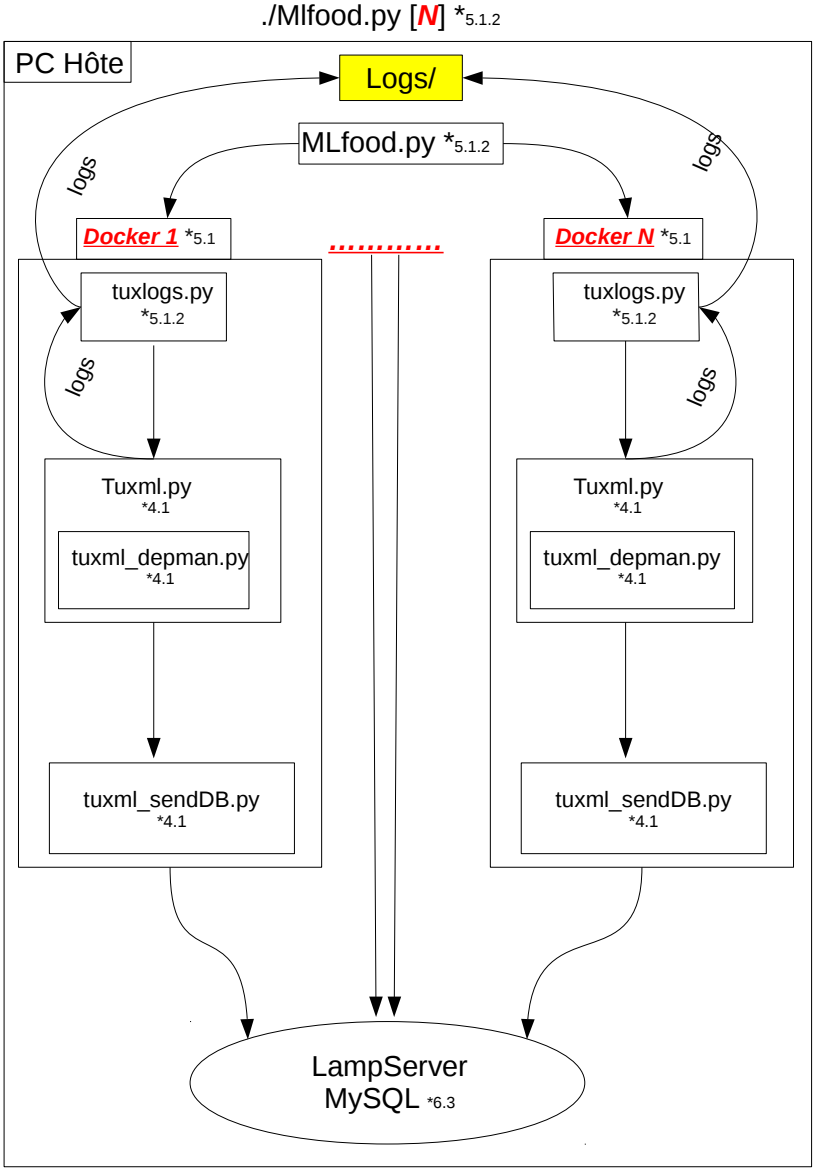


FIGURE 3 – Architecture globale du projet TuxML

Troisième partie
Résultats et réalisations futures

Ce projet se déroule en plusieurs phases, nous avons terminé la première qui consiste à remplir une base de données servant de matière première pour le futur algorithme de *Machine Learning*. Nous sommes capables de lancer un nombre donné de compilations et de les traiter afin d'en extraire les informations nécessaires au remplissage de la base. Ce qui nous amène à la phase suivante qui consiste en le déploiement de nos scripts de manière à avoir un panel conséquent de données utilisables pour l'algorithme de *Machine Learning*.

Il faut pour cela avoir accès à une grille de calcul et la mise en place de l'algorithme de *Machine Learning* qui nous permettra de "prédire" grâce à cet échantillon, le comportement d'un noyau selon les options choisies.

7 Résultats

À l'heure où nous écrivons ces lignes, les résultats de 1200 compilations ont ainsi été sauvegardés en base de données. Quelques statistiques ont aussi été calculées. Ainsi le noyau Linux pèse en moyenne 75.77Mo, le plus léger ne pesait que 12.44Mo tandis que le plus lourd 1,793Go. Le premier quartile est à 29,11Mo et le troisième à 76,73Mo.

En moyenne une compilation dure 16 minutes. Les compilations les plus rapides ont été effectuées en 2 minutes et les plus longues en 2 heures ! Le premier quartile est à 8 minutes et le troisième à 18 minutes.

D'autre part, TuxML a été testé par une dizaine de personnes au sein de l'équipe DiverSE et sur des machines aux architectures et distributions variées.

8 Maturation du projet

8.1 Passage à l'échelle

Pour pouvoir utiliser le *Machine Learning*, nous avons besoin de réaliser plusieurs milliers de compilations. Pour faire cela nous avons décidé, après proposition par notre professeur encadrant, d'utiliser un système de grille de calcul. Une grille de calcul consiste en un nombre conséquent d'ordinateurs relié à un ordinateur maître qui permet l'exécution d'un travail par toutes les machines de la grille, ici `MLFood.py` . Pour le projet nous avons décidés d'utiliser les grilles de calcul suivantes :

- https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home
- http://igrida.gforge.inria.fr/

8.2 Une métrique importante mais difficile à vérifier : le *boot*

Un des éléments les plus importants à vérifier et à quantifier dans le cadre de la compilation de tout programme est son fonctionnement ou non. Dans notre cas

il s'agit de la capacité ou non d'un noyau Linux à "booter" ou non. En effet empiriquement il semblerait que la majorité des configurations que l'on peut obtenir aléatoirement ne fonctionnent pas. Il est donc très important de pouvoir identifier si le noyau fonctionne ou pas pour les configurations qui déjà compilent correctement.

Or il s'agit d'une problématique particulièrement ardue que l'on a pu commencé à étudier mais encore sans résultats probants. Cependant notre piste actuelle est d'utiliser *QEMU*, un émulateur libre très puissant et fonctionnant par ligne de commande et d'en extraire l'affichage de la machine virtuelle qui booterait sur le noyau testé. L'affichage serait alors parsé, en temps réel si possible, à la recherche des signes indicateurs d'un *kernel panic* signifiant le non fonctionnement du noyau. Dans le cas contraire si le noyau fonctionnait le *parser* trouverait l'invité de *login* classique de Debian en ligne de commande. Enfin un système d'horodatage ou un chronomètre pourrait permettre d'obtenir le temps de *boot* le cas échéant, ce qui représente une autre métrique très intéressante dans le cadre de l'étude de l'impact des configurations sur le noyau Linux.

9 Lien avec l'apprentissage automatique

Le but final du projet est bien évidemment de faire appel au *Machine Learning* présenté plus haut afin d'obtenir un système capable de prédire l'influence des options de configuration choisies sur certaines métriques comme la taille du *kernel* ou son temps de compilation. Avec comme but à très long terme d'avoir pourquoi pas un système nous donnant plusieurs configurations jugées "optimales" pour une métrique donnée (le problème se rapprocherait alors du domaine de la Recherche Opérationnelle).

Cependant avant d'en arriver là il faut faire le lien avec les technologies d'apprentissage automatique existant déjà à ce jour. En effet nous n'avons pas l'intention ni les capacités pour recréer ce que des équipes de chercheurs et d'ingénieurs ont déjà fait dans le domaine. Or il se trouve que la majorité des technologies disponibles semble utiliser les fichiers `.csv` comme vecteurs de données pour les algorithmes. Ce faisant nous sommes déjà bien parti pour lier nos données collectées aux algorithmes. Cependant il nous reste encore à étudier justement ce qui est disponible dans l'état de l'art et voir les spécificités d'implémentation qu'auront les différentes alternatives. Mais ces problématiques n'ont pas encore pu être explorées au cours de ce semestre.