Université de Strasbourg

Cursus Master Ingénierie
Informatique, Systèmes et Réseaux

Travail d'Étude et de Recherche

État de l'art des unikernels

Sujet

Réduire les systèmes pour mieux virtualiser ?

Encadrant
Pierre David

Auteur
Ludovic Muller

Rapporteur
Antoine Gallais

Mardi 7 mai 2019

Résumé

Dans le monde de la virtualisation, on utilise en grande partie des conteneurs pour déployer des services, notamment pour leurs performances et leur facilité de déploiement. Cependant les conteneurs n'offrent pas une solution d'isolation complète par design, alors que c'est le cas pour les machines virtuelles. Néanmoins, lancer une machine virtuelle avec un système d'exploitation traditionnel pour exécuter chacune des différentes applications entraîne un surcoût important. Un idéal serait d'avoir les performances d'un conteneurs et l'isolation fournie par les machines virtuelles. C'est ce à quoi essayent de répondre les unikernels.

Table des matières

1 Introduction

Aujourd'hui, de plus en plus de services sont proposés aux utilisateurs. Le nombre de ressources à traiter est également fortement croissant et c'est ainsi que de nouveaux centres de données sont construits au fur et à mesure partout dans le monde.

Au départ, une machine physique était attribuée à un service particulier. Lorsque la demande augmentait, on ajoutait simplement une nouvelle machine dédiée uniquement à ce service.

Cependant le nombre de services proposés ne cessant de croître, il n'est plus possible de se contenter de ne faire tourner qu'un seul service par machine, notamment pour des raisons budgétaires et pour un déploiement beaucoup plus rapide.

Les fournisseurs de services souhaitent gagner en qualité pour faire face à la concurrence. Cette qualité se traduit notamment par une démultiplication des données à travers le globe : cela rapproche le contenu des utilisateurs finaux, diminue les temps de latence, et permet d'être plus robuste en cas de panne. En effet, il y aura toujours une machine présente pour prendre le relais. On gagne donc en souplesse en étant capable d'adapter l'offre et la demande en temps réel.

Pour ces fournisseurs, pouvoir faire tourner un grand nombre de services sur une même machine permettrait d'économiser le nombre de machines et de mieux les rentabiliser, du fait que l'on utilisera pleinement les ressources d'une machine au lieu de n'utiliser qu'un faible pourcentage sur plusieurs machines.

La virtualisation répond parfaitement à ces attentes. En effet, virtualiser plusieurs systèmes sur une seule machine permettrait d'exécuter indépendamment un grand nombre de services.

La virtualisation répond à un besoin d'isolation. En effet, si l'on souhaite lancer un grand nombre de services différents sur une même machine physique, un service ne doit pas compromettre un des autres services tournant sur la machine. Les systèmes d'exploitation traditionnels répondent déjà en partie à ce besoin d'isolation, en virtualisant la mémoire et en isolant le matériel avec la séparation de l'espace utilisateur de l'espace système via les appels système. En virtualisation de système d'exploitation, le besoin d'isolation va encore plus loin puisqu'il faut isoler la mémoire de chacun des systèmes d'exploitation ainsi que les opérations de lecture et d'écriture. Virtualiser ainsi un système d'exploitation est un peu comme si l'on créait des machines virtuelles à l'intérieur d'une seule et même machine physique ; c'est la raison pour laquelle on parle justement de machines virtuelles, ou VM pour *virtual machines* en anglais.

Enfin, on peut dire que la virtualisation offre une certaine sécurité, du fait de cette isolation, et qu'il est plus facile de gérer une machine virtuelle compromise qu'une machine physique ; on peut en recréer une plus rapidement en clonant une VM de base par exemple. Virtualiser offre également une certaine tolérance aux pannes, puisqu'il est possible de dupliquer ou migrer très simplement des machines virtuelles qui tourneraient sur un matériel défaillant par exemple. Enfin, certaines entreprises ont besoin de faire tourner certaines applications obsolètes (applications *legacy*) : utiliser la virtualisation permettrait à ces entreprises de continuer à utiliser ces applications sans sacrifier la sécurité de leur parc informatique. On constate donc que la virtualisation offre une réelle souplesse dans la gestion des services.

Aujourd'hui, d'autres solutions que la virtualisation de systèmes d'exploitation traditionnels ont émergé [11, 17]. On utilise par exemple de plus en plus le mécanisme des conteneurs : au lieu de virtualiser complètement un système d'exploitation, on fait tourner directement l'application sur la machine hôte, en interceptant les appels systèmes, offrant ainsi une alternative beaucoup plus légère et rapide que la virtualisation classique.

2 Définition du problème

La virtualisation offre une réelle souplesse comme nous avons pu le voir. Cependant, virtualiser un système d'exploitation traditionnel peut s'avérer plutôt lourd. Pour éviter cette lourdeur, le mécanisme des conteneurs peut être particulièrement efficace. Puisqu'ils tournent directement sur

la machine hôte, on économise toute la couche de la virtualisation d'un système complet. Les ressources nécessaires en matière de stockage, de mémoire et de calcul sont moindres par rapport à l'utilisation massive de machines virtuelles : on peut donc faire tourner un nombre nettement plus élevé de services.

Cependant l'utilisation des conteneurs peut s'avérer problématique en termes de sécurité, notamment du fait qu'ils tournent directement sur l'hôte [11] et que le nombre d'appels système ne cesse de croître au fil des années [12] : aujourd'hui un noyau Linux en compte environ 400. L'API des appels système permet aux conteneurs d'interagir avec le système d'exploitation hôte et offre une gestion des processus, des *threads*, de la mémoire, du réseau, du système de fichiers, de la communication IPC, etc. Le fait que les conteneurs tournent directement sur l'hôte nécessite également une homogénéité entre le système d'exploitation hôte et invité, ce qui peut être limitant.

Ce que l'on souhaiterait, c'est d'une part avoir la possibilité d'une isolation forte qui garantit une certaine sécurité, et d'autre part, d'être léger. Ceci permet un déploiement en masse de manière adaptative, une rapidité pour servir et s'adapter continuellement à la demande et ainsi faire face à la concurrence, tout en offrant une qualité de service sûre.

Cela fait quelques années que des équipes de chercheurs se posent la question et une solution semble émerger : les unikernels.

Que sont les unikernels ? En quoi diffèrent-ils des solutions actuellement utilisées ? Comment arrivent-ils à offrir des gains réels en performances sans sacrifier la sécurité, chose que l'on n'arrivait pas à satisfaire simultanément jusqu'à présent avec des conteneurs et des VM traditionnelles ?

3 Unikernels

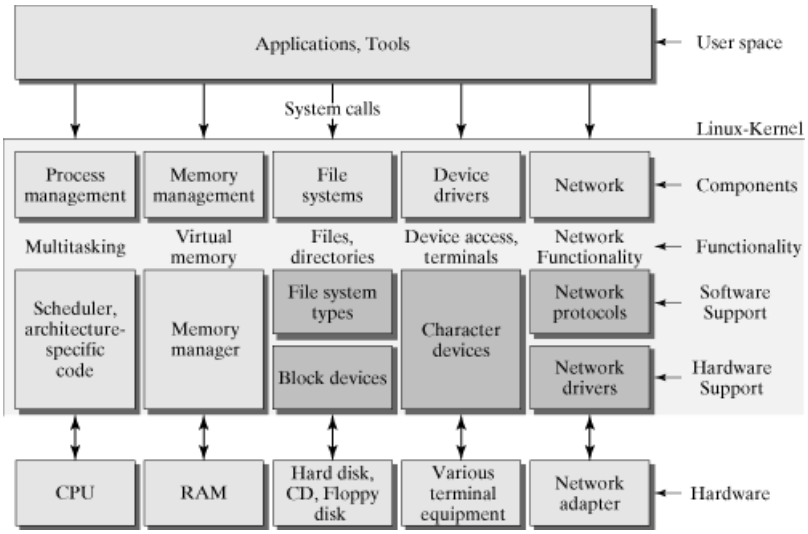


Fig. 1 : Architecture d'un système Linux classique[1]

La figure 1 nous montre l'architecture d'un système Linux classique. On y retrouve toute la partie matérielle (CPU, RAM, disques, réseau, …). Dessus, nous avons le noyau Linux, qui tourne dans l'espace noyau qui est un mode privilégié, qui va se charger de la gestion des processus, de la mémoire, du système de fichier, des différents pilotes de périphériques et de toute la couche protocolaire réseau. Les applications quant à elles, tournent dans l'espace utilisateur qui est un mode non privilégié. Pour accéder aux différentes ressources, l'application doit effectuer des appels système (`syscall`). Ici, le noyau Linux permet une couche d'abstraction entre les applications et le matériel.

1. Source : https://www.linux-india.org/characteristics-and-architecture-of-linux-oprating-system/

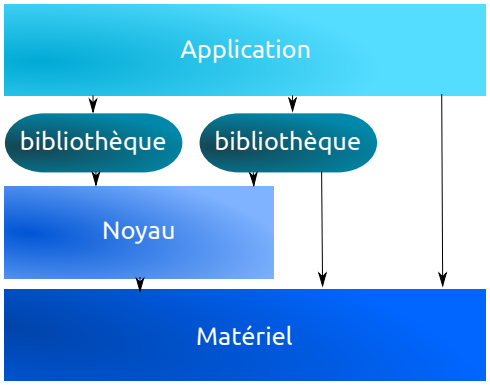


Fig. 2 : Architecture d'un système d'exploitation bibliothèque

Les unikernels quant à eux pourraient être décrits simplement comme étant un système d'exploitation créé à partir d'un assemblage de briques de LEGO, chacune de ces briques étant une bibliothèque élémentaire permettant une tâche de base, par exemple la partie réseau, ou bien la communication IPC, etc. Un système construit de cette manière à partir de bibliothèques de base s'appelle un système d'exploitation bibliothèque ou encore *library OS* ou *libOS* en anglais. Le but de cette architecture que l'on retrouve sur la figure 2 est d'exécuter un maximum de code dans l'espace utilisateur directement, et de n'avoir qu'un noyau minimaliste. Cette architecture permet également aux applications et aux librairies d'accéder directement au matériel si elles le souhaitent. Les premiers systèmes d'exploitation de ce type étaient Exokernel [6] et Nemesis[2] dans les années 1990s.

Le principe des unikernels est qu'au lieu de lancer un système d'exploitation classique composé d'un grand nombre d'applications, il va se charger de ne faire tourner que le binaire d'une application, et donc de ne faire tourner qu'un seul processus [17] au sein d'un seul espace mémoire. On va ainsi uniquement utiliser les briques dont l'application a réellement besoin pour fonctionner.

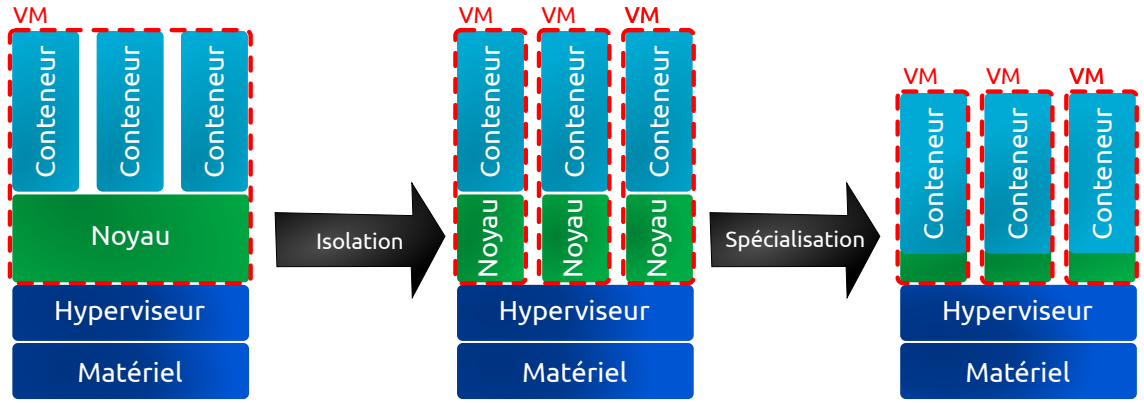
Pour éviter d'avoir à prendre en charge l'ensemble des périphériques possibles, dans les cas des unikernels on va partir du principe qu'il tournera dans une machine virtuelle, et que ce sera à l'hyperviseur de s'occuper du matériel et d'en faire l'abstraction. Un hyperviseur est un programme qui permet la gestion (création, lancement, …) des VM.

On se retrouve donc avec une image beaucoup plus légère. Il y a un lien de corrélation entre la taille des images et le temps de démarrage (*boot time*) suite au temps de chargement de l'ensemble en mémoire [12], ce qui fait que les unikernels peuvent démarrer beaucoup plus rapidement que les systèmes traditionnels. De plus, ils garantissent davantage de sécurité : il n'est même pas possible de se connecter sur la machine, puisqu'ils tournent directement au sein d'une machine virtuelle et n'incluent que le strict nécessaire pour faire tourner l'unique application. En outre, ils ne dépendent que d'un nombre très restreint de bibliothèques, limitant le nombre de failles et *bugs* possibles, ce qui limite fortement la surface d'attaque [12].

Cependant, obtenir ces gains de performances tout en garantissant une certaine sécurité nécessite beaucoup de temps de configuration, étant donné qu'il faut spécialiser le plus possible le système pour l'application souhaitée.

La figure 3[3] nous montre en (a) ce qui se fait de manière classique : lancer un gestionnaire de conteneurs dans une VM. L'ensemble des conteneurs se partagent le même noyau. Si l'on souhaite isoler les différents services, on peut lancer une VM par conteneur, comme en (b). Cela introduit hélas un surcoût en termes de ressources, dû au fait que l'on doive dupliquer à chaque fois le noyau. Le fait de travailler avec des noyaux spécialisés pour l'application, les unikernels (c), permet de limiter ce surcoût, tout en assurant une isolation forte.

2. https://www.cl.cam.ac.uk/research/srg/netos/projects/archive/nemesis/
3. Figure faite à partir d'un schéma publié par Docker



(a) **Conteneur Linux**
Noyau partagé

(b) **Un conteneur par VM**
Noyaux dupliqués

(c) **Unikernels**
Noyaux spécialisés

FIG. 3 : Isolation et spécialisation avec les unikernels

4 Différents problèmes

4.1 Composants

On souhaite donc voir comment l'on procède pour réaliser un unikernel. Nous avons dit dans la partie précédente que les unikernels étaient des systèmes légers où l'on incluait que le strict nécessaire. Comment est-ce que l'on fait pour choisir les composants absolument nécessaires ? En effet, garder un nombre trop important de composants ferait perdre l'avantage de passer par un unikernel, et en retirer trop peut empêcher l'application de tourner convenablement.

4.2 Plusieurs processus

Certaines applications ont besoin de lancer plusieurs processus, notamment en faisant des `fork` par exemple. Comment est-ce qu'il est possible d'envisager de les faire fonctionner au sein d'un unikernel, sachant que par *design* un unikernel n'est fait que pour faire tourner qu'un seul processus ? Est-ce qu'il y a besoin d'effectuer des modifications à son application pour espérer la faire tourner ?

4.3 Performances

La majorité des unikernels, pour éviter d'avoir à prendre en charge une liste importante de périphériques à gérer via une multitude de pilotes de périphériques, partent du principe qu'ils seront lancés au sein d'une machine virtuelle gérée par un hyperviseur, en l'occurrence Xen. Cependant, cela induit un surcoût du fait que la machine hôte doit émuler les différents périphériques. Comment est-ce que les diverses solutions proposées se débrouillent pour limiter ce surcoût ? Si les performances sont mauvaises, il sera difficile d'envisager d'utiliser une telle solution en production.

4.4 Spécialisation

Les unikernels sont, comme nous l'avons vu, spécialisés. Cela implique que l'on a dû faire un certain nombre de concessions, notamment en ne prennent qu'un nombre très restreint de langages en charge. Quels choix ont pu être faits ? Et pourquoi ?

4.5 Sécurité

Du point de vue de la sécurité, le fait de tourner au sein d'une machine virtuelle permet d'assurer une certaine étanchéité. Est-ce que les différents auteurs proposant des solutions d'unikernels portent attention à d'autres aspects pour assurer une sécurité encore plus importante ?

Nous verrons dans la partie suivante quelles sont les principales solutions d'unikernels et les différents outils associés à l'heure actuelle et comment ils répondent aux différents problèmes soulevés tout au long de cette partie.

5 Différentes solutions

La majorité des solutions étudiées dans le cadre de ce travail se basent toutes sur le même hyperviseur : Xen [2]. Il s'agit donc d'une référence essentielle, crédible et fiable dans le domaine de la virtualisation. Cependant un nombre important de solutions n'hésitent pas à réimplémenter certaines parties de Xen, comme le XenStore [12], dans le but de limiter le surcoût occasionné par le fait de passer par de la virtualisation.

5.1 Unikernels

Pour commencer, nous allons voir comment font les solutions principales d'unikernel pour répondre aux différentes problématiques soulevées précédemment.

5.1.1 ClickOS

ClickOS [13], un système d'exploitation sur mesure pour du traitement réseau. Les machines virtuelles sont légères, démarrent rapidement et n'utilisent que très peu de mémoire.

5.1.2 Drawbridge

Drawbridge[4] [14] qui est un prototype de recherche conçu par l'équipe de recherche de Microsoft, qui permet de lancer des applications de manière isolées au sein d'un Windows 7 modifié sous forme de *libOS*, ce qui peut être intéressant d'un point de vue sécurité.

5.1.3 HaLVM

HaLVM, pour *Ha*skell *L*ightweight *V*irtual *M*achine, permet de lancer du code Haskell dans un environnement minimal directement sur un hyperviseur Xen. On retrouve grandement l'aspect système d'exploitation bibliothèque (libOS), par le fait que l'on peut choisir ou non les différents composants dont on a besoin, par exemple un système de fichier avec Halfs (*Ha*skell *F*ile *S*ystem) et la stack TCP/IP complète avec HaNS (*Ha*skell *N*etwork *S*tack).

5.1.4 IncludeOS

IncludeOS [5] est un *libOS* minimal, à l'état de projet de recherche, fait pour faire tourner du code C++ sur du matériel virtualisé.

5.1.5 KylinX

KylinX [17] offre un mécanisme de pVM, pour *process-like VM*. Ce procédé permet de réaliser des `fork` en instanciant une nouvelle machine virtuelle. La machine ayant fait l'appel à `fork` sera mise en pause le temps de la création de la pVM et recevra ensuite un moyen de la contacter, ceci permet de répondre au besoin de prendre en charge plusieurs processus.

5.1.6 LING

LING[5] est un unikernel basé sur Erlang/OTP. Erlang est un langage de programmation qui permet de construire des applications temps réel évolutives tout en assurant de la haute disponibilité. OTP (=*Open Telecom Platform*) est composé d'un système d'exécution Erlang ainsi que d'un ensemble de bibliothèques et composants pour Erlang.

4. https://www.microsoft.com/en-us/research/project/drawbridge/#!publications
5. http://erlangonxen.org/

L'unikernel est capable d'interpréter les fichiers `.beam`, qui sont le résultat de la compilation d'un programme Erlang en code objet. Il est donc aisé d'intégrer une application à LING, pour la faire tourner dans une machine virtuelle légère sur Xen, et donc de la déployer en production.

5.1.7 MirageOS

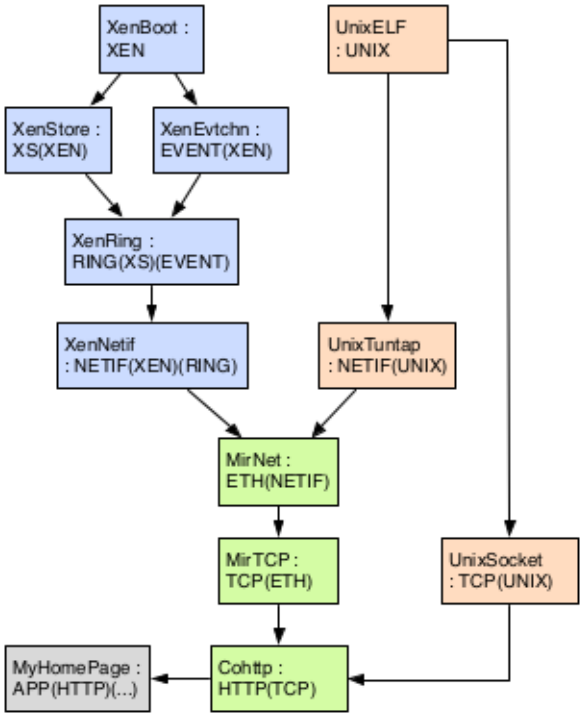


FIG. 4 : Exemple d'une application HTTP sur MirageOS (extrait de la doc. de MirageOS[6])

L'avantage de MirageOS [10], une solution d'unikernel utilisant le langage OCaml, est qu'il offre de bonnes couches d'abstractions. L'exemple que l'on a en figure 4 nous montre les différentes dépendances dont on peut avoir besoin lors du développement ainsi que du déploiement d'une application MirageOS.

En effet, lors de la phase de développement, les développeurs sont généralement sur un système de style Unix. La bibliothèque Cohttp a besoin d'une implémentation TCP, qui est fournie par la bibliothèque UnixSocket. Une fois que le développement est terminé, toute la partie Unix est retirée et on recompile l'application en utilisant cette fois-ci le module MirNet, afin d'établir une liaison directe avec le pilote réseau Xen.

5.1.8 OSv

OSv [7], un OS conçu pour le *cloud* qui peut uniquement être lancée depuis un hyperviseur, capable de faire tourner une unique application avec des gains en performances principalement sur la partie réseau. Les auteurs ont cherché à réutiliser un maximum de composants déjà existants. Par exemple, le système de fichier utilisé est ZFS [18], récupéré depuis FreeBSD, qui permet de s'assurer de l'intégrité des données et propose un mécanisme de *snapshots* et de gestion de volumes. Sont aussi supportés ramfs, dans le cas où l'on souhaiterait démarrer sans disque, ainsi que devfs, un système de fichier simple pour visualiser les périphériques. Ils ont également récupéré les fichiers de *header* C depuis le projet `musl libc`[7], le VFS (=Virtual File System) depuis le projet Prex[8],

6. https://mirage.io/wiki/technical-background
7. https://www.musl-libc.org/
8. https://github.com/tworaz/prex

et les drivers ACPI depuis le projet ACPICA[9]. Concernant la partie réseau, elle a également été importé au départ de FreeBSD, mais elle a été longuement réécrite.

5.1.9 Rumprun

Rumprun[10], un unikernel dans lequel on intègre un binaire d'application C, C++, Erlang, Go, Java, JavaScript, Python, Ruby ou Rust par exemple à un kernel appelé `rump`[11]. Cela permet de créer des applications *bootables*, légères et portables.

5.1.10 runtime.js

Runtime.js[12] est un unikernel open source[13], sur lequel on peut greffer une application JavaScript, pour déployer cette dernière de manière légère et immuable. Il utilise un modèle d'entrée/sortie non bloquant basé sur des évènements inspiré de NodeJS, ce qui permet de gérer plusieurs tâches simultanément. KVM est le seul hyperviseur pris en charge à l'heure actuelle.

Il y a deux composants principaux. Le premier est le noyau du système d'exploitation, qui est écrit en C++, et qui permet la gestion des ressources telles que le CPU et la mémoire ainsi que de faire tourner le moteur JavaScript V8. Le second composant est une bibliothèque JavaScript qui gère l'ensemble du système ainsi que les différents périphériques virtualisés.

À l'heure actuelle, ce projet n'est désormais plus maintenu et n'est pas fait pour tourner en production.

5.2 Outils

Dans cette partie, nous allons étudier quelques outils intéressants faisant partie de l'écosystème des unikernels.

5.2.1 Jitsu

Jitsu [9] est un serveur DNS qui utilise les unikernels pour servir des applications à la demande. Les auteurs ont fait quelques optimisations sur Xen, notamment pour le faire fonctionner sous ARM.

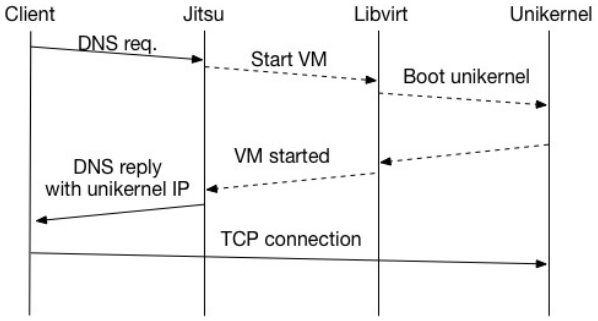


Fig. 5 : Principe de fonctionnement de Jitsu (extrait du dépôt GitHub du projet[14])

Cette solution permet de *booter* des unikernels à la demande suite à une simple requête DNS, comme on peut le voir sur la figure 5. Par exemple cela permet de démarrer une VM lorsque l'on souhaite accéder à une page web.

Concernant l'aspect sécurité, ils ont pris la liste des dernières vulnérabilités et ont pu constater que nombreuses d'entre elles affectent les systèmes réseaux embarqués, le noyau Linux et Xen.

9. https://www.acpica.org/
10. https://github.com/rumpkernel/rumprun
11. http://rumpkernel.org/
12. http://runtimejs.org
13. https://github.com/runtimejs/runtime
14. Source : https://github.com/mirage/jitsu

Les applications déployées avec Jitsu ne sont que sensibles aux vulnérabilités de Xen ainsi que de quelques unes de Linux, le reste étant complètement éliminé grâce au mécanisme d'isolation.

5.2.2 LightVM

LightVM [12], une nouvelle solution de virtualisation, permet de démarrer une machine virtuelle presque aussi rapidement qu'un `fork` ou un `exec` sur Linux et serait deux fois plus rapide que Docker [1], une solution permettant de lancer des conteneurs. Il propose une refonte complète du plan de contrôle de l'hyperviseur Xen, grâce à des opérations distribuées ayant des interactions réduites au minimum, dans le but de gagner en performances.

5.2.3 Tynix

Tinyx [12], une solution de *build* automatisée permet de créer des images de machines virtuelles Linux minimalistes en utilisant une approche particulièrement originale. En effet, il va tenter de retirer une à une les différentes options de *boot* en testant si l'application continue toujours de fonctionner comme souhaité avec l'aide d'un jeu de test. Si un des tests ne passe plus, l'option est réactivée, car essentielle. Cette manière de procéder permet de rajouter les options de *boot* uniquement nécessaires à l'application.

5.2.4 UniK

UniK[15] est un outil open source écrit en Go qui permet de compiler des applications sous forme d'image *bootable* légère plutôt que sous forme de binaire. UniK permet de construire des images pour AWS Firecracker, Virtualbox, AWS, Google Cloud, vSphere, QEMU, UKVM, Xen, OpenStack, Photon Controller et en lancer des instances.

UniK se charge de compiler l'application et de greffer le binaire à une base d'unikernel existante, tel que rump, OSv, IncludeOS ou MirageOS par exemple, que l'on peut préférer en fonction du langage dans lequel est écrit l'application.

Le fait de prendre en charge un grand nombre de types d'unikernels permet de générer des unikernels pouvant faire tourner des applications écrites dans un nombre varié de langages.

5.3 Conclusion

Concernant le choix des composants à inclure, certaines solutions offrent la possibilité de tester pour nous ce qu'il faut inclure ou non, tel que Tinyx par exemple pour les options de *boot*.

Pour la gestion de plusieurs processus, il y a plusieurs manières de faire. Par exemple KylinX implémente un mécanisme de *pVM*, alors que runtime.js se base sur le fait que JavaScript soit un langage évènementiel avec des entrées/sorties non bloquantes pour supporter le multi tâches, ce qui évite d'avoir à modifier son application.

Certains n'hésitent pas à réécrire certaines parties de Xen, comme LightVM qui est une réécriture du plan de contrôle de l'hyperviseur, afin de gagner en performances.

Les unikernels étant spécialisés, ils ne peuvent pas forcément faire tourner n'importe quelle application. Certains par exemple ne font que tourner des programmes en OCaml, comme MirageOS, d'autres permettent de faire tourner quelques programmes en provenance de langages différents, comme c'est le cas par exemple avec rumprun, qui permet de faire tourner des applications écrites en C, C++, Erlang, Go, Java, JavaScript, Python, Ruby ou Rust. Le choix de MirageOS de ne supporter que OCaml est surtout motivé par le fait qu'il s'agisse d'un langage de haut niveau, il y a moins de risques de soucis avec la gestion de la mémoire et on limite le nombre de lignes de code.

Quant à la sécurité, contraindre l'utilisation d'un langage de haut niveau comme le fait MirageOS permet de limiter les attaques dues à une mauvaise gestion de la mémoire par exemple. Les auteurs de certaines solutions, par exemple Jitsu, ont pris la liste des dernières vulnérabilités connues et ont

15. https://github.com/solo-io/unik

pu voir toutes celles qui pourraient impacter leur solution. Finalement l'ensemble des unikernels, par leur spécialisation, limitent drastiquement la surface d'attaque.

6 Évaluation des performances

Les différents auteurs utilisent diverses métriques afin de comparer leurs solutions, comme on peut le constater dans la table 1.

La majorité des solutions étudiées se comparent avec des conteneurs Docker [4] et des machines virtuelles faisant tourner un système d'exploitation classique.

Tab. 1 : Différentes métriques utilisées pour évaluer les performances de différents unikernels

Unikernel	Métriques utilisées
OSv	performances de l'application, taille du tas, réseau, changements de contexte
KylinX	temps de boot, utilisation mémoire, communication inter-pVM, mise à jour de l'environnement d'exécution (bibliothèques), performances d'applications
MirageOS	temps de boot, threads, réseau, stockage, performances d'application
Jitsu	débits, latences réseau, latence démarrage du service, puissance utilisée / autonomie, sécurité
LightVM	temps d'instanciation, temps de migration, utilisation CPU, empreinte mémoire

La table 1 est une vision d'ensemble des différentes métriques utilisées par chacune des différentes solutions que nous avons mentionnées dans ce rapport. Nous constatons que les métriques les plus utilisées concernent les performances de l'application (temps de lancement, temps de réponse), le temps de boot et l'utilisation de ressources telles que la consommation mémoire ou CPU. Nous allons détailler ces métriques dans la suite de cette partie.

Cependant reprendre les chiffres obtenus dans les différents papiers de recherche n'a pas spécialement de sens, étant donné qu'ils utilisent chacun une configuration différente pour effectuer les différents tests. Le but de cette partie est donc véritablement de voir ce qui est mesuré et comment, en fonction de ce qui est précisé par les différents auteurs.

6.1 OSv

Ils utilisent `Memaslap` pour mesurer les performances de Memcached, un serveur de stockage clé-valeur et `SPECjvm2008` pour mesurer les performances d'application dans la JVM.

Les performances réseau ont quant à elles été mesurées avec `Netperf`, et la taille du tas avec `JVM balloon`.

6.2 KylinX

[17] ne décrit pas spécifiquement quels outils ont été utilisés pour faire les différentes mesures. Néanmoins, la mesure de la communication inter-pVM a été faite en mesurant la latence de communication entre la pVM mère et la pVM fille, possiblement avec un `ping`.

6.3 MirageOS

La mesure des latences est faite avec `ping`. Les mesures des débits a été fait avec `iperf` [16]. Pour mesurer les performances d'application, ils ont mis en place un serveur web et mesuré les

performances avec l'aide de `httperf`.

6.4 Jitsu

Tout comme MirageOS, la mesure des débits est faite avec `iperf`, les latences réseau avec `ping`, la latence de démarrage de service en regardant combien de temps cela prend avant que ça ne réponde aux différentes requêtes. Un point intéressant est qu'ils utilisent une Cubieboard, qui est un nano-ordinateur, pour mesurer la puissance utilisée ainsi que pour tester l'autonomie. Pour mesurer la sécurité de leur solution, ils ont pris la liste des dernières vulnérabilités connues et ont pu regarder à chaque fois si leur solution était ou non vulnérable.

6.5 LightVM

Il n'est pas précisé la manière dont sont évalués le temps d'instanciation ainsi que l'empreinte mémoire. Ils utilisent cependant `iostat` pour mesurer l'utilisation du CPU.

7 UniK

Il serait bien de pouvoir tester différentes solutions d'unikernels afin de comparer leurs performances entre elles, ainsi qu'avec les solutions traditionnelles telles que les conteneurs et les machines virtuelles. Ceci nous permettra de nous donner un ordre de grandeur sur plusieurs aspects à propos de ces solutions.

La solution qui nous semble la plus intéressante à tester est UniK, d'une part le fait qu'elle soit relativement bien documentée, et d'autre part par le fait qu'elle permet de compiler et d'inclure des applications à un nombre relativement correct d'unikernels, ce qui est parfait pour expérimenter.

Pour commencer, nous avons installé UniK sur notre machine en suivant la documentation, et nous avons pris la décision de cibler initialement QEMU du fait de sa présence sur la machine. Cependant, nous n'étions pas en mesure de construire des images et nous avons pu constater par la suite que l'intégration n'était pas complète à l'heure actuelle. Nous avons donc finalement fait le choix de cibler VirtualBox.

7.1 Un exemple concret

Nous trouvons qu'il est intéressant de pouvoir illustrer nos propos avec un exemple concret. Nous avons donc décidé d'écrire un serveur web avec le langage Go, qui nous permet de le faire de manière assez rapide. Ce choix d'exemple offre l'avantage de montrer la communication réseau entre l'hôte et l'application et nous permet de détailler la manière dont fonctionne UniK.

Nous compilons notre code en local sur ma machine et testons si la page s'affiche bien dans le navigateur pour tester si notre serveur est fonctionnel, ce qui est le cas.

Nous lançons donc la commande pour construire l'image pour VirtualBox. En cas d'erreur lors du build, il est possible de récupérer les logs depuis de démon UniK. Dans notre exemple, nous avons choisi de partir sur la base de l'unikernel rump pour y greffer notre serveur web. Finalement, l'image ne fait que 39 Mo.

Une fois l'image créée, nous pouvons créer autant d'instances que nous souhaitons de cette image.

UniK permet de récupérer la liste des différentes images créées ainsi que des différentes instances lancées, avec leurs adresses IP. Pour récupérer l'adresse IP, lors de la création d'une instance pour VirtualBox, le démon UniK va envoyer en UDP une requête à un *listener*, qui va lui répondre.

Nous avons donc plusieurs instances de notre serveur web qui fonctionnent parfaitement. On pourrait imaginer la possibilité d'y intégrer un *load-balancer* pour répartir équitablement la charge sur les différentes instances lancées.

Nous avons pu effectuer quelques tests sur notre application en Go avec une base avec l'unikernel rump. Nous avons pu découvrir que l'application continuait à répondre aux requêtes même si

l'application paniquait manuellement ou tentait de faire un `Exit`, du fait que les différents `syscall` n'y sont pas implémentés. Cependant, lorsque nous souhaitions effectuer des tests de performances, l'ensemble crashait subitement de manière non déterministe. En outre, le fait qu'il soit relativement aisé de récupérer les logs propres à une instance avec UniK peut être pratique pour *debug* une application.

Nous souhaitions également tester la présence d'un système de fichier. Pour cela, nous avons ajouté un dossier dans lequel se trouve une image, et nous l'avons servi depuis le serveur web. Nous avons pu constater que l'image est belle est bien accessible depuis le navigateur web : un système de fichier est donc bien intégré dans notre cas. L'analyse des logs de *debug* du démon UniK lors de la création de l'image nous confirme l'inclusion du système de fichier.

Il aurait pu être intéressant de tester la découverte de services : est-ce qu'il serait possible de lancer une instance d'unikernel avec un serveur d'API et une autre instance d'unikernel avec une application qui y effectue des requêtes ? Autrement dit, est-ce qu'il y aurait la possibilité de faire de la découverte de service et récupérer dynamiquement l'adresse IP du serveur d'API depuis une autre instance ? Nous n'avons malheureusement pas eu le temps de tester cela lors de ce travail.

7.2 Évaluation des performances

Pour évaluer les performances de différentes solutions, nous avons effectué une démarche similaire à ce que l'on a fait précédemment pour avoir un serveur web écrit en Go sur une base rump, afin d'obtenir d'autres images d'unikernels. Nous sommes parvenus à créer en plus de l'image précédente, une image avec un serveur web écrit en Java sur une base rump et sur une base OSv. Cependant nous n'étions pas en mesure de pouvoir accéder au serveur web depuis l'hôte dans le cas du serveur web en Java sur une base rump.

Nous ne sommes pas parvenus à trouver un moyen propre pour évaluer la consommation mémoire de chacune des solutions testées, mais nous avons réussi à obtenir des résultats pertinents concernant la taille des images ainsi que le temps de réponse d'un serveur web.

7.2.1 Taille des images

TAB. 2 : Taille des images des différentes solutions testées

Langage	Type	Base	Taille de l'image (en Mo)
Go	unikernel	rump	39
Java	unikernel	rump	241
Java	unikernel	OSv	90
Go	conteneur Docker	`golang :1.12`	788
Java	conteneur Docker	`openjdk :7`	880
Go	VM	Debian 9	1 600
Java	VM	Debian 9	1 900

La table 2 nous montre la taille des différentes images que nous avons pu créer avec UniK. Nous pouvons constater que la taille varie en fonction de la base d'unikernel utilisée, par exemple un serveur web en Java nécessitera 241 Mo sur une base rump alors qu'il nécessitera que 90 Mo sur une base OSv. De même, la taille dépend également du langage utilisé, par exemple sur une base rump, un serveur web ne nécessite que 39 Mo s'il est écrit en Go, alors qu'il pèse 241 Mo s'il est écrit en Java.

En conteneurisant l'application pour Docker, nous obtenons une taille plus importante, de l'ordre de 788 Mo pour le serveur web en Go et 880 Mo pour celui en Java, ce qui revient à la même conclusion que nous avons pu faire lors de la comparaison sur une même base d'unikernel.

Nous avons également installé deux Debian 9 avec les outils nécessaires pour créer et lancer le même serveur web en Java et en Go respectivement sur ces VM. On peut observer qu'un serveur web en

Java génère une plus grande taille d'image qu'en Go. Les images sont nettement plus lourdes dans le cas d'une VM sous Debian 9 qu'avec les unikernels.

7.2.2 Temps de réponse

Étant donné que nous avons créé un serveur web pour nos différents tests, nous allons comparer avec l'aide de `curl` le temps de réponse d'une simple page web affichant une ligne de texte. Dans le but d'avoir des résultats significatifs, nous avons lancé à chaque fois 10 000 simulations.

Tab. 3 : Temps de réponse (en ms) aux requêtes

Solution	Temps moyen	Temps minimal	Temps maximal	Temps médian
Go sur base rump	3.128	0.470	15.453	2.385
Java sur base OSv	1.172	0.488	21.485	0.771
Go sur conteneur Docker	0.349	0.296	3.888	0.341
Java sur conteneur Docker	0.383	0.260	10.235	0.359
Go sur VM Debian 9	0.715	0.559	7.155	0.705
Java sur VM Debian 9	1.102	0.676	20.548	0.909

On constate sur la table 3 que l'on a des temps de réponse similaires pour le serveur web écrit en Java, qu'on soit sur une base d'unikernel OSv ou dans une VM Debian 9. Le serveur web écrit en Go quant à lui répond plus rapidement que celui en Java lorsqu'il est sur Debian 9, et met plus de temps à répondre que toutes les autres solutions lorsqu'il est greffé sur une base rump.

Les conteneurs offrent le meilleur temps de réponse, du fait qu'il y a moins de couches d'abstraction, contrairement au unikernels ou les systèmes d'exploitation, qui eux, tournent dans un environnement virtualisé.

7.3 Maturité

Le projet est relativement fonctionnel dans son ensemble et propose un certain choix en termes de cibles de déploiements et en base d'unikernel pour construire les images. Cependant certaines de ces cibles semblent être moins prises en charge que d'autres, comme QEMU par exemple.

Lors de certains de nos tests, le *listener* déployé par UniK au sein d'une nouvelle VM dans VirtualBox plantait de temps en temps, nous obligeant soit à redémarrer la machine virtuelle en question manuellement, soit à la détruire pour que le démon la crée à nouveau, ce qui ne serait pas envisageable dans un environnement de production.

Nous avons pu constater dans une conférence donnée par Idit Levine en 2016[16], que l'API de Docker est en mesure de communiquer avec celle d'UniK et qu'il était possible d'intégrer des unikernels créées par UniK à Kubernetes [3][17], qui est un orchestrateur de conteneurs qui est de plus en plus utilisé.

7.4 Conclusion

Bien que le projet ne soit pas encore tout à fait mature pour être lancé en production, il est prometteur et semble s'adapter aux besoins actuels, qui sont notamment le fait de pouvoir s'intégrer à des solutions d'orchestrations tels que Kubernetes.

Il est relativement simple à prendre en main et rend le monde des unikernels beaucoup plus accessible qu'auparavant.

Concernant les performances obtenues, nous pouvons voir que là où l'on gagne le plus à utiliser des unikernels est sur la taille des images. Les temps de réponses sont cependant moins élevés

16. https://www.youtube.com/watch?v=wcZWg3YtvnY
17. https://kubernetes.io/

que ceux des conteneurs, mais il s'agit d'un compromis à faire pour avoir une sécurité renforcée. Ainsi, utiliser des unikernels permettrait de faire des économies sur l'espace disque. La mesure de la consommation mémoire ainsi que les ressources CPU utilisées auraient pu être d'excellents indicateurs, mais nous ne sommes pas parvenus à effectuer des mesures de manière fiable.

8 Conclusion

Les unikernels offrent des avantages certains, tels que de très bonnes performances avec une sécurité plus importante que lors de l'utilisation de conteneurs comme Docker[18] par exemple, tout en ayant une faible empreinte mémoire. En effet, les mesures que nous avons pu faire avec UniK nous montrent que les unikernels sont certes légèrement moins performants que les conteneurs, mais offrent un gain non négligeable en termes de sécurité ainsi que des images bien plus légères.

Cependant, construire une image spécifique pour chaque application est très coûteux en termes de temps, du fait que chaque application a ses propres besoins, mais cela tend à devenir plus abordable notamment avec des projets comme UniK. Un autre problème est que l'on est amené à recompiler tout l'ensemble lorsque l'on souhaite effectuer des changements, étant donné que l'on intègre que le strict minimum pour lancer le binaire de l'application, et non ceux pour effectuer les changements.

Aujourd'hui il existe d'autres systèmes concurrents aux unikernels, tels que les lambdas proposés par Amazon (AWS Lambda [8, 15][19]) par exemple, qui permettent d'exécuter des fonctions de manière indépendante, et de pouvoir adapter les ressources nécessaires à la demande en temps réel.

Références

[1] Charles Anderson. Docker [software engineering]. *IEEE Software*, 32(3) :102–c3, 2015.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[3] David Bernstein. Containers and cloud : From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3) :81–84, 2014.

[4] Carl Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1) :71–79, 2015.

[5] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. Includeos : A minimal, resource efficient unikernel for cloud services. In *2015 ieee 7th international conference on cloud computing technology and science (cloudcom)*, pages 250–257. IEEE, 2015.

[6] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel : An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[7] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. Osv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, 2014. USENIX Association.

[8] MichałKról and Ioannis Psaras. Nfaas : Named function as a service. In *Proceedings of the 4th ACM Conference on Information-Centric Networking*, ICN '17, pages 134–144, New York, NY, USA, 2017. ACM.

18. https://www.docker.com/
19. https://aws.amazon.com/fr/lambda/

[9] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, David J Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. Jitsu : Just-in-time summoning of unikernels. In *NSDI*, pages 559–573, 2015.

[10] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels : Library operating systems for the cloud. *Acm Sigplan Notices*, 48(4) :461–472, 2013.

[11] Anil Madhavapeddy and David J. Scott. Unikernels : Rise of the virtual library operating system. *Queue*, 11(11) :30 :30–30 :44, December 2013.

[12] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 218–233, New York, NY, USA, 2017. ACM.

[13] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, 2014.

[14] Donald E. Porter, Silas Boyd-Wickizer, , Reuben Olinsky, and Galen Hunt. Rethinking the library os from the top down. Association for Computing Machinery, Inc., March 2011.

[15] Josef Spillner. Snafu : Function-as-a-service (faas) runtime design and implementation. *CoRR*, abs/1703.07562, 2017.

[16] Ajay Tirumala, Tom Dunigan, and Les Cottrell. Measuring end-to-end bandwidth with iperf using web100. In *Presented at*, number SLAC-PUB-9733, 2003.

[17] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. Kylinx : A dynamic library operating system for simplified and efficient cloud virtualization. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 173–186, Boston, MA, 2018. USENIX Association.

[18] Yupu Zhang, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. End-to-end data integrity for file systems : A zfs case study. In *FAST*, pages 29–42, 2010.