Boot2Container : La flexibilité des conteneurs sur bare metal

Vincent Autefage
Université de Bordeaux
15 Rue Naudet
33175 Gradignan, France

Martin Roukala
MuPuf TMI - Valve Corporation
Helsinki, Uusimaa, Finlande

Résumé

Boot2Container, ou b2c, est un projet open source offrant la possibilité de déployer directement et surtout facilement des images de conteneurs sur machines physiques ou virtuelles sans avoir recours à un orchestrateur ni même avoir besoin d'un système d'exploitation hôte.

Né du besoin de Valve de tester des pilotes pour la Steam Deck de la façon la plus simple, native et reproductible possible, b2c permet de se soustraire des tâches d'installation et de maintenance du système d'exploitation en se reposant sur l'amorçage par le réseau et des images de conteneurs stockées sur un registry dans le but de démarrer un parc entier de machines. En outre, le contexte d'exécution reste rigoureusement identique à chaque nouveau démarrage, laissant l'administrateur seul décisionnaire des données devant être pérennisées.

Cet article présente les concepts principaux régissant l'architecture de b2c ainsi que des scénarios d'utilisation caractéristiques sur machines virtuelles mais aussi sur serveurs bare metal. Nous présentons enfin des cas d'usage pour lesquels une telle solution nous paraît pertinente.

Mots-clefs

Boot2Container, b2c, Conteneur, Linux, Netboot, QEMU, Podman, Virtualisation.

1 Introduction

Les conteneurs révolutionnent depuis plus de 10 ans la manière dont beaucoup d'applications et de services sont déployés [01]. Bien que cette technologie soit fortement associée aux architectures micro-services [02][03], son utilisation dépasse largement ce cadre. Nous pouvons par exemple citer leur présence massive dans les systèmes d'intégration continue [04] et plus généralement leur rôle dans l'émergence de la culture de l'automatisation ayant conduit à la grande popularité des mouvances *DevOps* [05] et *GitOps* [06].

Les conteneurs, outre l'isolation du contexte d'exécution qu'ils confèrent, apportent pléthore d'avantages en comparaison d'une application native [07]. Nous nous attachons particulièrement ici aux propriétés suivantes qui ont largement participé à l'adoption massive de la conteneurisation :

- la simplicité de construction et l'interopérabilité des images de part leur standardisation[1] ;
- la simplicité de distribution et de déploiement de part le recours aux *registries* [08] ;

1 https://opencontainers.org

- l'immuabilité des images qui garantit la reproductibilité du déploiement et donc l'homogénéité d'un parc de machines [09].

Ces trois propriétés sont difficilement réplicables sur un serveur de type bare metal et même sur une machine virtuelle. En effet, l'utilisation de conteneurs suppose l'installation et la configuration au préalable d'un système d'exploitation sur la machine hôte, et par la suite sa maintenance.

Il existe naturellement des outils qui permettent l'auto-configuration d'une machine selon un schéma précis de recettes tels que *Ansible*[2], *cloud-init*[3] ou encore *Salt*[4]. Ces solutions ne permettent néanmoins pas d'obtenir une installation rigoureusement identique et impliquent en outre un temps de convergence plus ou moins long. De plus, l'aspect immuable de l'installation sous-jacente n'est pas possible en l'état avec de tels outils.

Des outils d'installation automatisée tels que *Fog*[5] permettent de déployer une image strictement identique, appelée g*host*, au prix d'une réinstallation complète du système, ou plutôt un *flash*, sans pour autant fournir d'immuabilité de ce dernier à posteriori.

Une stratégie tierce repose quant à elle sur la fabrication d'images immuables spécifiques, stratégie proposée notamment par *Firecracker*[6], *rpm-ostree*[7] ou encore *casync* [10]. Cette approche nécessite néanmoins une fabrication manuelle de ces images (*i.e. rootfs*), opération laborieuse et potentiellement chronophage complexifiant ainsi grandement l'utilisation et l'adoption massive de ces outils.

Leurs pendants plus spécifiques au monde l'embarqué tels que *Buildroot*[8] ou encore *Yocto*[9] souffrent également d'une forte complexité de prise en main ainsi que d'un important temps de compilation. De plus, ces outils ne reposent pas sur des distributions Linux standards ; *i.e.* ils peuvent être considérés comme leur propre distribution. De ce fait, l'offre de paquets logiciels prêts à l'emploi y est fortement restreinte.

Boot2Container[10], plus communément dénommé *b2c*, est un projet libre sous licence *MIT* visant à proposer l'utilisation de conteneurs natifs sur des machines virtuelles ou des serveurs bare metal sans nécessiter l'installation d'un système au préalable. Il rend ainsi possible la capacité de *booter* des conteneurs, *i.e.* instancier une ou plusieurs images de conteneurs, sur une machine virtuelle ou physique dépourvue de système d'exploitation.

Des distributions Linux spécialisées dans l'instanciation de conteneurs existent et pourraient ainsi paraître similaires à *b2c* [11]. Nous pouvons par exemple citer *Fedora CoreOS*[11], *Flatcar*[12] ou encore *Bottlerocket*[13] qui ont pour raison d'être de faciliter et d'optimiser l'exploitation de conteneurs. Ces distributions nécessitent toutes néanmoins une installation ainsi qu'une configuration préalable sur la machine hôte, aspect dont *b2c* s'abstrait totalement. Cette singularité offre de nombreux avantages que nous tentons d'illustrer au travers de cet article.

2 https://www.ansible.com

3 https://cloud-init.io

4 https://saltproject.io

5 https://fogproject.org

6 https://firecracker-microvm.github.io

7 https://coreos.github.io/rpm-ostree

8 https://buildroot.org

9 https://www.yoctoproject.org

10 https://gitlab.freedesktop.org/gfx-ci/boot2container

11 https://fedoraproject.org/coreos

12 https://www.flatcar.org

13 https://bottlerocket.dev

Dans un premier temps, nous proposons une présentation rapide de l'historique du projet avant d'aborder ses concepts clés ainsi que ses fonctionnalités majeures. Par la suite nous abordons son utilisation pour une machine virtuelle ainsi que sur un serveur bare metal avec amorçage par le réseau. Enfin, nous donnons quelques cas d'utilisation caractéristiques mettant ainsi en lumière l'intérêt d'une telle solution.

2 Origine et architecture

2.1 Historique du projet

Boot2Container est né du besoin de *Valve*[14] de pouvoir automatiser l'exécution de suites de test sur une grande variété de machines physiques, dont la *Steam Deck*[15]. Ces tests portent sur de nombreux composants qui permettent de faire tourner les jeux *Steam*, développés initialement pour Windows, sur Linux (*e.g.* pilotes graphiques, émulation *DirectX*).

Le nombre de projets concernés par ces tests n'étant pas négligeable et surtout répartis à la fois sur *GitHub*[16] mais aussi sur *Gitlab*[17], il était difficilement concevable de déployer une ferme de machines de test par projet ; raison pour laquelle il a été décidé de privilégier une unique ferme commune. Il est alors devenu essentiel de disposer d'un outil permettant d'instancier ces tests sur le parc de machines ainsi déployé, outil devant être agnostique en termes de système d'intégration continue mais devant également être capable d'appliquer une configuration bas niveau à ces machines (*e.g.* amorçage, noyau, pilotes). Les orchestrateurs conventionnels d'intégration continue sont de fait inutilisables dans ce contexte.

C'est ainsi que les fondations du projet *b2c* virent le jour : un système d'exploitation minimal pouvant être téléchargé au démarrage d'une machine et amorcé en mémoire vive afin de garantir que toute erreur lors d'un test ne puisse avoir d'incidence sur les résultats des tests suivants. Ces derniers portant sur des éléments du noyau, toute erreur provoque un redémarrage de la machine physique. Le système d'exploitation fournit par *b2c* est totalement configurable via la ligne de commande du noyau et permet ainsi de configurer l'environnement de test *complet* de l'hôte, noyau inclus.

2.2 Concepts clés

Le démarrage d'un système implique de disposer des éléments suivants :

- un *noyau* (*i.e. kernel*) comportant l'ensemble des pilotes nécessaires au fonctionnement du matériel cible ;
- un *système de fichiers racine* (*i.e. rootfs* ou *initrd*) contenant l'ensemble des logiciels, bibliothèques et fichiers de configuration nécessaires à l'utilisation du système ;
- un *chargeur d'amorçage* (*i.e. bootloader*) permettant l'initialisation du matériel et le chargement du noyau.

b2c se matérialise par un *initrd* contenant du code écrit en *shell script* ainsi qu'en *Go*. Fabriqué avec *u-root*[18] et consciencieusement minifié pour arriver à une taille d'environ 20 Mo, il permet la

14 https://www.valvesoftware.com

15 https://www.steamdeck.com

16 https://github.com

17 https://gitlab.com

18 https://u-root.org

récupération et l'instanciation automatique de conteneurs avec *Podman*[19]. Couplé à un noyau Linux générique, *b2c* permet de démarrer une machine, qu'elle soit physique ou virtuelle, en utilisant des conteneurs plutôt qu'un système Linux installé préalablement sur le disque. La configuration de ces conteneurs étant entièrement effectuée au travers des paramètres de la ligne de commande du noyau Linux, il devient possible de contrôler à distance les propriétés de démarrage de la machine et ce même lorsqu'elle est éteinte.

Ainsi, démarrer une machine virtuelle *QEMU*[20] sans système d'exploitation ni même de stockage de masse sur une distribution Linux standard telle qu'*Alpine*[21] avec un clavier configuré en français s'effectue en une simple ligne de commande :

```
> qemu-system-x86_64 -m 1G -accel kvm -kernel b2c-kernel -initrd b2c-initrd.xz \
                     -append 'b2c.run="-it docker.io/alpine:3.20" b2c.keymap=fr'
```

À l'heure à laquelle nous écrivons ces lignes, *b2c* est disponible pour les architectures *x86_64*, *ARMv6*, *ARM64* et *RISCV64*.

3 Fonctionnalités principales

3.1 Synopsis

Comme introduit dans la section précédente, la configuration de *b2c* est intégralement effectuée sous la forme de multiples commandes de type `b2c.<command>[=<value>]` directement insérées comme arguments du noyau Linux. Cette section présente les fonctionnalités principales offertes par *b2c*. Nous supposons ici que le lecteur est familier avec les commandes usuelles de manipulation des conteneurs que ce soit avec *Docker*[22] ou *Podman*, ces deux moteurs de conteneurs ayant une interface en ligne de commande quasi identique.

3.2 Conteneurs

La commande `b2c.run` permet de configurer la manière dont `podman run` est exécuté, `b2c.run=<args...>` se traduisant par `podman run <args...>`. Cette commande peut être répétée plusieurs fois, instanciant les conteneurs demandés de façon séquentielle ; *i.e.* le conteneur *n* est lancé après que l'exécution du conteneur *n-1* soit terminée. La commande `b2c.pipefail` peut être ajoutée afin d'indiquer qu'une erreur dans le code de retour d'un conteneur doit stopper le lancement des conteneurs suivants.

Il est possible d'instancier plusieurs conteneurs en parallèle et en arrière plan grâce à la commande `b2c.run_service` dont la forme est identique à celle de `b2c.run`.

Enfin, la commande `b2c.run_post` permet de lancer des conteneurs supplémentaires de façon séquentielle lorsque tous les conteneurs instanciés avec `b2c.run` ont terminé leur exécution et ce sans prendre en compte le code de retour des conteneurs précédents. Cette fonctionnalité est particulièrement utile pour des tâches de nettoyage ou de plus généralement de finalisation.

L'ensemble des conteneurs est lancé en mode *privilégié* [12] et sans isolation réseau (*i.e. host network*) donnant ainsi aux conteneurs un accès plus important aux ressources matérielles de l'hôte et autorisant tous les programmes tournant dans ces conteneurs à dialoguer entre eux comme s'ils étaient directement lancés sur la machine.

19 https://podman.io

20 https://www.qemu.org

21 https://www.alpinelinux.org

22 https://www.docker.com

Il est également possible d'utiliser des images de conteneurs présentes sur un *registry* soumis à authentification en utilisant la commande `b2c.login` dont nous illustrons l'utilisation ci-dessous.

Dans l'exemple fictif suivant, nous instancions un conteneur *Pingvin Share*[23], projet libre de partage et de transfert de fichiers, pour une durée de 2 heures. Pour cela, ce conteneur est instancié en tant que *service* tandis qu'un conteneur *Alpine* est quant à lui instancié en tant que conteneur principal avec pour commande `sleep 7200`. Un conteneur d'analyse antivirale *ClamAV*[24] de type *post* est enfin ajouté en fin d'exécution afin de vérifier la légitimité des fichiers partagés. Cet exemple est volontairement simpliste, nous pourrions tout à fait remplacer le conteneur d'attente par un conteneur contenant une *API REST*. Cela nous permettrait de contrôler l'arrêt du service et donc de la machine en lieu et place d'une simple minuterie. Nous pourrions également instancier le conteneur de partage de fichiers comme conteneur principal et ainsi lui allouer une durée de vie illimitée jusqu'à l'arrêt forcé de l'hôte.

```
b2c.login=docker.io,username="$LOGIN",password="$PASSWORD" \
b2c.run_service="-t -v data:/opt/app/backend/data docker.io/stonith404/pingvin-share" \
b2c.run="-t docker.io/alpine:3.20 sleep 7200" \
b2c.run_post="-t -v data:/scan docker.io/clamav/clamav:latest clamscan \
                              --recursive=yes --remove=yes /scan"
```

3.3 Réseau

Le téléchargement d'images de conteneurs tout comme certains programmes ou services embarqués nécessitent la mise en réseau de l'hôte. Pour ce faire, *b2c* initie par défaut un client *DHCP* sur toutes les interfaces réseaux filaires de l'hôte au démarrage du système.

Ce comportement peut être finement adapté grâce à la commande `b2c.iface`. Cette dernière, en plus de définir la méthode de connexion (*i.e. DHCP* ou statique) d'une ou plusieurs interfaces, permet également de régler différents paramètres essentiels tels que les serveurs de nom, la passerelle par défaut, des routes spécifiques ou bien encore l'*ip forwarding*.

Les commandes connexes `b2c.hostname` et `b2c.ntp_peer` permettent respectivement de définir le nom de l'hôte et de préciser le serveur de temps à utiliser.

L'exemple ci-dessous illustre l'instanciation d'une sonde réseau *NtopNG*[25] permettant de surveiller le trafic en provenance d'un réseau local. Nous considérons ici que la machine sur laquelle nous instancions le conteneur, qu'elle soit virtuelle ou physique, possède deux interfaces réseaux : la première connectée en *DHCP* sur le réseau externe, la seconde connectée en statique sur le réseau local `192.168.42.0/24`. La machine fait ici office de passerelle entre le réseau local et le monde extérieur.

```
b2c.hostname=ntop b2c.ntp_peer=0.fr.pool.ntp.org \
b2c.iface=eth0,dhcp,forward,nameserver=9.9.9.9 \
b2c.iface=eth1,address=192.168.42.1/24 \
b2c.run="-t docker.io/ntop/ntopng:stable -i eth1"
```

3.4 Cache

Le système *b2c* s'exécutant intégralement depuis la mémoire vive, chaque image de conteneurs téléchargée devra l'être à nouveau à chaque redémarrage de la machine.

23 https://github.com/stonith404/pingvin-share

24 https://www.clamav.net

25 https://www.ntop.org/products/traffic-analysis/ntop

De surcroît, il est par essence impossible d'instancier un conteneur dont la taille de l'image dépasserait la quantité de RAM disponible, freinant ainsi fortement les possibilités d'intégration de *b2c* sur les machines disposant de ressources limitées.

La commande `b2c.cache_device` a donc été introduite dans le but de surmonter ces limitations. En effet, elle rend possible l'utilisation d'un périphérique de stockage de masse (*e.g. NVME*, *SATA*) ou bien d'un système de fichiers réseau (*e.g. NFS*, *SMB*) afin d'y enregistrer et y conserver les images de conteneurs téléchargées, libérant ainsi la mémoire vive de la machine de cette tâche. Cette commande possède de nombreux paramètres permettant de régler finement son comportement tels que le formatage, le *trim* automatique, la spécification du périphérique ou encore la réinitialisation complète du disque. Nous recommandons néanmoins sa forme la plus simple pour le cas d'un stockage de masse. La commande `b2c.cache_device=auto` laisse en effet *b2c* libre de créer une partition dédiée à cet usage sur l'espace non partitionné le plus pertinent ; *i.e.* celui disposant de la plus grande capacité et du débit le plus rapide.

3.5 Volumes

Comme nous l'avons vu dans la section 3.2, il est possible d'utiliser des volumes afin de partager des données entre conteneurs. Ces volumes sont néanmoins stockés en mémoire vive et ne sont donc pas persistants. La commande `b2c.volume` permet de créer un volume persistant, stocké dans la partition de cache configurée via la commande `b2c.cache_device` présentée dans la section précédente.

L'exemple ci-dessous permet ainsi de conserver les données présentes dans le volume `vdata` et ce même lors du redémarrage de l'hôte ou bien lors d'un changement de conteneur.

```
b2c.cache_device=auto b2c.volume=vdata \
b2c.run="-it -v vdata:/data docker.io/alpine:3.20" \
b2c.run="-it -v vdata:/data docker.io/debian:12-slim"
```

Cette fonctionnalité, bien qu'intéressante, restreint la persistance des données au périphérique de cache. Afin d'outrepasser cette limitation, *b2c* permet de monter un volume persistant sur un système de fichiers spécifique grâce à la commande `b2c.filesystem`.

L'exemple ci-dessous illustre l'utilisation de deux volumes, le premier utilisant une partition locale pré-existante formatée en *EXT4*, tandis que le deuxième repose sur un stockage *NFS* en lecture seule exposée sur `192.168.42.254`.

```
b2c.filesystem=fs-disk,type=ext4,src=/dev/sdb1 \
b2c.filesystem=fs-nfs,type=nfs,src=192.168.42.254:/,opts=ro \
b2c.volume=vol-disk,filesystem=fs-disk \
b2c.volume=vol-nfs,filesystem=fs-nfs \
b2c.run="-it -v vol-disk:/mnt/disk -v vol-nfs:/mnt/nfs docker.io/alpine:3.20"
```

La commande `b2c.minio` permet de synchroniser un volume avec un *bucket S3*[26], avec ou sans chiffrement coté serveur (*i.e. SSE*) et selon une politique prédéfinie ; *i.e.* au démarrage, à l'arrêt de l'hôte ou bien encore à la création ou à la terminaison de chaque conteneur.

L'exemple ci-dessous permet à un conteneur d'effectuer une synchronisation descendante (*i.e. pull*) au démarrage de l'hôte ainsi qu'une synchronisation ascendante (*i.e. push*) lors de la terminaison de chaque conteneur. Nous supposons ici qu'un serveur *S3* est accessible à l'adresse `192.168.42.254.`

```
b2c.minio=srv-s3,https://192.168.42.254,$access_key,$secret_key
b2c.volume=vol-s3,mirror=srv-s3/$bucket_name,pull_on=pipeline_start,push_on=container_end
b2c.run="-it -v vol-s3:/mnt/s3 docker.io/alpine:3.20"
```

26 https://aws.amazon.com/fr/s3

Enfin, il est possible d'appliquer un chiffrement symétrique aux volumes locaux de façon transparente par l'intermédiaire de la fonctionnalité *fscrypt*[27] offerte par le noyau Linux. Pour cela une clé de *512 bits* encodée en *base 64* est nécessaire comme l'illustre l'exemple ci-dessous.

```
# Génération de la clé
> FSCRYPT_KEY=$(head -c 64 /dev/urandom | base64 -w 0)

# Configuration de b2c
b2c.filesystem=fs-enc,type=ext4,src=/dev/sdb1 \
b2c.volume=vol-enc,filesystem=fs-enc,fscrypt_key=$FSCRYPT_KEY \
b2c.run="-it -v vol-enc:/mnt/enc docker.io/alpine:3.20"
```

4 Déploiement

Comme énoncé dans l'introduction de cet article, *b2c* peut être déployé sur une machine virtuelle ou bare metal. Dans ces deux cas, il est nécessaire de récupérer un noyau Linux compilé pour *b2c* ainsi que son système de fichiers racine, tous deux disponibles sur la page de téléchargement du projet[28]. Dans cette section, nous utiliserons la dernière version stable officielle, soit la version `9.12.3`.

4.1 Machine virtuelle

Nous prenons ici le cas d'une machine virtuelle *QEMU* dotée d'une architecture *x86_64*. La première étape consiste à télécharger le noyau ainsi que le système de fichiers racine de *b2c*.

```
> wget https://gitlab.freedesktop.org/gfx-ci/boot2container/-/releases/v0.9.12.3/downloads/linux-x86_64-qemu
> wget https://gitlab.freedesktop.org/gfx-ci/boot2container/-/releases/v0.9.12.3/downloads/initramfs.linux_amd64.cpio.xz
```

Nous créons ensuite une image disque pour notre machine virtuelle afin d'héberger un cache pour nos images de conteneurs et nos volumes. Cette étape n'est nécessaire que dans le cas où nous souhaitons assurer une persistance des données.

```
> qemu-img create -f qcow2 b2c.img 10G
```

L'exemple ci-dessous démarre un serveur web *nginx*[29] sur une machine virtuelle *QEMU*, accélérée par *KVM*, dotée de 2 Go de mémoire vive, de 2 cœurs de processeur, d'une interface réseau et d'un clavier en français ; le port `8080` de notre hôte physique redirigeant le trafic sur le port `80` du serveur web ainsi instancié. Un volume monté vers un répertoire de notre hôte physique est configuré au travers de l'interface *virtio*[30]. Ce répertoire contient les fichiers devant être distribués par notre serveur web. Un volume secondaire qui sera sauvegardé dans l'image disque de cache est ajouté afin de conserver les fichiers de logs générés par notre conteneur.

```
> qemu-system-x86_64 -accel kvm -k fr -m 2G -cpu host -smp cores=2 \
     -nic user,hostfwd=tcp::8080-:80 \
     -drive file=b2c.img,format=qcow2 \
     -virtfs local,path=./www,mount_tag=www,security_model=passthrough \
     -kernel linux-x86_64-qemu -initrd initramfs.linux_amd64.cpio.xz \
     -append 'b2c.keymap=fr b2c.hostname=web b2c.cache_device=auto \
       b2c.filesystem=fs-www,type=9p,src=www \
       b2c.volume=vol-www,filesystem=fs-www \
       b2c.volume=vol-log \
       b2c.run="-it -v vol-log:/var/log -v vol-www:/usr/share/nginx/html docker.io/nginx:latest"'
```

27 https://www.kernel.org/doc/html/latest/filesystems/fscrypt.html

28 https://gitlab.freedesktop.org/gfx-ci/boot2container/-/releases

29 https://nginx.org

30 https://wiki.qemu.org/Documentation/9psetup

4.2 Machine bare metal

À la différence d'une utilisation sur machine virtuelle, une machine physique n'a pas de système hôte pour charger dans sa mémoire le noyau ainsi que le système de fichiers racine et configurer la ligne de commande *b2c*. Dès lors, il est nécessaire de recourir à un chargeur d'amorçage dont la mission principale sera de récupérer et configurer les éléments nécessaires au démarrage de la machine via *b2c*.

Pour ce faire, nous proposons d'utiliser le chargeur d'amorçage libre *iPXE*[31] qui permet le téléchargement d'un script d'amorçage ainsi que les ressources nécessaires à l'initialisation de *b2c* via *TFTP* ou encore *HTTP(s)*. Disposer d'un tel script d'amorçage hébergé sur un serveur distant offre la possibilité de configurer les paramètres de *b2c* de façon asynchrone et ce même lorsque la machine est éteinte.

Nous proposons dans l'exemple ci-dessous un script *iPXE* permettant de télécharger le noyau ainsi que le système de fichiers racine de *b2c* afin d'amorcer un conteneur *Alpine*.

```
#!ipxe
kernel https://gitlab.freedesktop.org/.../linux-x86_64 initrd=b2c b2c.keymap=fr b2c.run="-it docker.io/alpine:latest"
initrd --name b2c https://gitlab.freedesktop.org/.../initramfs.linux_amd64.cpio.xz
boot
```

5 Exemples d'utilisation concrète

5.1 Intégration continue

Comme nous l'avons évoqué dans la section 2.1, le recours à *b2c* dans le cadre d'un pipeline d'intégration continue est le cas d'usage emblématique et historique du projet : permettre d'effectuer des tests d'intégration bas niveau sans avoir à installer, ré-installer ou configurer au préalable de système d'exploitation sur une ferme de machines.

Couplé à un orchestrateur responsable de réserver temporairement des machines[32], *Valve* met à profit *b2c* afin d'effectuer de nombreux tests sur divers projets bas niveau ; tout cela sans ce soucier de l'état préalable des machines ainsi utilisées. Les résultats de ces tests sont ensuite partagés au travers de *buckets S3*.

D'autre part, *b2c* est utilisé dans le cadre du développement du noyau Linux afin de tester le support des périphériques *HID*[33]. Ces tests sont réalisés sur machines virtuelles *QEMU*[34].

5.2 Systèmes d'affichage embarqué

De par ses propriétés, *b2c* est particulièrement adapté à la configuration de systèmes embarqués comme des écrans d'information (*e.g.* emploi du temps, transport, publicité). En effet, son aspect immuable, couplé à sa grande facilité de mise à jour et de retour en arrière de version, en fait une solution simple et surtout fiable pour cet usage.

Cette approche permet d'éviter les risques induits par la mise à jour du système d'exploitation comme cela a été le cas pour l'incident expérimenté par *CrowdStrike* en juillet 2024, paralysant 8.5 millions de machines et nécessitant une intervention manuelle de récupération sur chacun de ces systèmes [13].

31 https://ipxe.org

32 https://gitlab.freedesktop.org/gfx-ci/ci-tron

33 https://docs.kernel.org/hid/hiddev.html

34 https://github.com/torvalds/linux/blob/master/tools/testing/selftests/hid/vmtest.sh

5.3 Mode examen

Basculer une machine de TP dans un mode propice à un examen est une opération complexe à laquelle de nombreuses équipes techniques se sont confrontées. Le recours à *b2c* peut encore ici être une stratégie pertinente, permettant de disposer d'un environnement dédié, éphémère, immuable et configuré à distance et en amont. La remise éventuelle de données peut être effectuée par l'intermédiaire d'un système de fichiers distant ou bien par la synchronisation d'un *bucket S3*. L'environnement pourra être remis à zéro lors du prochain redémarrage de la machine, permettant éventuellement un retour sur un système d'exploitation régulier installé au préalable et dédié aux travaux pratiques.

6 Conclusion

Dans cet article, nous avons présenté les fondements ainsi que les fonctionnalités principales de *Boot2Container*, aussi dénommé *b2c*. Ce dernier permet de remplacer le système d'exploitation Linux de la machine hôte, qu'elle soit virtuelle ou physique, en utilisant des conteneurs. De part sa flexibilité et son architecture, *b2c* est adapté à de nombreux scénarios d'usage.

Plusieurs fonctionnalités restent encore à développer telles que le support des fichiers de déploiement *Kubernetes*[35], la configuration d'interfaces réseaux virtuelles *Wireguard*[36], ou encore l'ajout d'une *API* de contrôle permettant ainsi d'interrompre ou de modifier l'exécution des conteneurs à chaud.

35 https://docs.podman.io/en/latest/markdown/podman-kube-play.1.html

36 https://www.wireguard.com

Bibliographie

[01] T. Siddiqui, S. A. Siddiqui and N. A. Khan, « Comprehensive Analysis of Container Technology », *4th International Conference on Information Systems and Computer Networks (ISCON)*, Mathura, India, 2019.

[02] V. Singh and S. K. Peddoju, « Container-based microservice architecture for cloud applications », *International Conference on Computing, Communication and Automation (ICCCA)*, Greater Noida, India, 2017.

[03] G. Liu, B. Huang, Z. Liang, M. Qin, H. Zhou and Z. Li, « Microservices: architecture, container, and challenges », *IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Macau, China, 2020.

[04] S. Garg and S. Garg, « Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security », *IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*, San Jose, CA, USA, 2019.

[05] Red Hat, « Understanding DevOps », 2022. https://www.redhat.com/en/topics/devops

[06] Gitlab, « What is GitOps? ». https://about.gitlab.com/topics/gitops

[07] Kim Clark, Callum Jackson, « The True Benefits of Moving to Containers », *IBM Developer*, 2020. https://developer.ibm.com/articles/true-benefits-of-moving-to-containers-1

[08] Red Hat, « What is a Container Registry? », 2022. https://www.redhat.com/en/topics/cloud-native-apps/what-is-a-container-registry

[09] Tak, Byungchul, et al. « Understanding Security Implications of Using Containers in the Cloud», *USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, USA, 2017.

[10] Lennart Poettering, « casync - A Tool for Distributing File System Images », 2017. https://0pointer.net/blog/casync-a-tool-for-distributing-file-system-images.html

[11] Nicolas Massé, « À la découverte de CoreOS ! », 2021. https://www.itix.fr/fr/blog/decouverte-de-coreos

[12] Dan Walsh, « How to Use the --privileged Flag with Container Engines », *Red Hat Enable Sysadmin*, 2020. https://www.redhat.com/sysadmin/privileged-flag-container-engines

[13] Sean Michael Kerner, « CrowdStrike Outage Explained: What Caused it and What's Next », *TechTarget*, 2024. https://www.techtarget.com/whatis/feature/Explaining-the-largest-IT-outage-in-history-and-whats-next