Building a Cloud Infrastructure to Deploy Microservices as MicroVM Toro Guests

www.torokernel.io

Matias Vara Larsen
matiasevara@gmail.com
What are microservices?

Monolithic Application

- Logging
- Order
- Catalog

Decomposed Application into Services

- Logging
- Order
- Catalog

e.g., Amazon website

Microservice #0

Microservice #1

Microservice #2
How are microservices deployed?

Microservice #0

Microservice #1

Service instance per VM

VM Context #0

VM Context #1

Hypervisor

Bare-metal host
How are microservices deployed?

The use of VMs to host microservices allows to isolate different services.

VMs are hosted on a Baremetal Host or as instances on a Cloud Provider, e.g., AWS or GCE.
How are microservices deployed?

- Device Model (Virtual Machine Monitor)
  - Microservice #0
  - Microservice #1

- Hypervisor
  - e.g., KVM, HyperV, Xen
  - Hypervisor

- Bare-metal host
  - e.g., Linux, Windows

- e.g., User application on a GP OS
  - e.g., QEMU, Firecracker
  - Microservice #1
• User and kernel mode
• User application executes as process with its own memory space
• User-to-kernel communication relies on syscalls
• General purpose scheduler which is timed-based
• Support for different Filesystems and drivers
Guests consume a lot of resources, e.g., memory, CPU, on-disk image.

VMs take long time to be up and running.

The creation and storage of VMs image are not simple [1]

A different set of drivers may be needed depending on the Cloud provider device model.

Guests consume a lot of resources, e.g., memory, CPU, on-disk image.

VMs take long time to be up and running.

The creation and storage of VMs image are not simple[1].

These issues end up by limiting the number of instances that a server can host and the cost of maintenance.

Bare-metal host

Hypervisor

VM Context #0

VM Context #1

Microservice #0

Microservice #1

General Purpose Guest OS

Scheduler

FileSystem

Networking

Drivers

Unikernel [2]

Unikernels: library operating systems for the cloud", Madhavapeddy et al., 2013

Unikernels: the next stage of Linux's dominance", Ali Raza et al., 2019

e.g., Osv, MirageOS, Unikraft, NanoVMs
Bare-metal host

Hypervisor

VM Context #0

VM Context #1

Microservice #0

Microservice #1

Single Address Space

Kernel + User’s application, e.g., NGIX, Python

Scheduler

FileSystem

Networking

Drivers

General Purpose Guest OS

Unikernel [2]
The port of an existing application takes a lot of work.
Toro is an application-oriented unikernel that allows microservices to run efficiently in VMs thus leveraging the strong isolation VMs provide.
Toro is an embedded kernel including five units:
- Process
- Memory
- Filesystem
- Networking
- Devices, e.g., Block Device, Network Device

Each unit provides minimalist APIs accessible from the embedded application.
Toro is an embedded kernel including five units:
- Process
- Memory
- Filesystem
- Networking
- Devices, e.g., Block Device, Network Device

Each unit provides minimalist APIs accessible from the embedded application.

Toro focus on VirtIO devices:
- virtio-block
- virtio-net

Toro focus on blocking and non-blocking AF_INET sockets

~ 18 KLOC
Application-oriented Kernel

- User application and kernel units are compiled in a single binary
- The application includes only the component required
Microservice

Application-oriented Kernel

- User application and kernel units are compiled in a single binary.
- The application includes only the component required.

```plaintext
program WebServerAppliance;
uses
Memory,
Filesystem,
Threads,
Networking,
Fat,
Virtio-blk,
Virtio-net;
Begin
//
// Your Code Goes Here
//
End.
```
Application-oriented Kernel

The generated binary is immutable[1], i.e., the generated image can be used across different hypervisors without the need to recompile it.

[1] "Immutable Infrastructure", Stella
CloudIt.sh Launches VM

Application-oriented Kernel

Toro Kernel
- Threads
- Memory
- Devices
- Filesystem
- Networking

Uses

VM Launches

CloudIt.sh Uses

MyMicroservice.elf Uses

Microservice
Application-oriented Kernel

CloudIt.sh Launches VM Application-oriented Kernel

- Threads
- Devices
- Networking
- Memory
- Filesystem

Uses Toro Kernel

Let’s see how is deployed a WebServer appliance

Uses MyMicroservice.elf

Uses CloudIt.sh

VM

Launched

User

Microservice
The WebServer Appliance

• Simple microservice that serves files by using the HTTP protocol
  - Find it at https://github.com/torokernel/torokernel among other examples
  - This appliance is used to host Toro's website (http://www.torokernel.io and click on “View on Toro”)
How the appliance is setup?

the code

Bare-metal host

KVM

VM Context #0

WebServer App

FileSystem (FAT)

Networking (TCP/IP Stack)

Drivers (virtio-blk, virtio-net)

Toro Guest
How the appliance is setup?

device model

- WebServer App
- FileSystem (FAT)
- Networking (TCP/IP Stack)
- Drivers (virtio-blk, virtio-net)
- Toro Guest

- Guest (Mode)
- Device Emulation
- BIOS

- In-kernel device emulation

- KVM Driver

- VM Context #0

- KVM

Bare-metal host
How the appliance is setup?

on-disk images

WebServer App

FileSystem
(FAT)

Networking
(TCP/IP Stack)

Drivers
(virtio-blk, virtio-net)

Toro Guest

Image/Binary for Guest

Image for Files

Host Filesystem

VM Context #0

KVM

Bare-metal host
How the appliance is setup?
ethernet configuration

WebServer App

FileSystem (FAT)
Networking (TCP/IP Stack)
Drivers (virtio-blk, virtio-net)

Toro Guest (192.100.200.101)

IP Forwarding to communicate to outside

Ethernet

BRIDGE

Host Application

KVM

Bare-metal host (192.100.200.100)
The Static WebServer drawbacks

- Disk images consume memory and on-disk space, e.g., each guest has its own image
- Disk images have to be distributed in all the nodes
- The use of a TCP/IP stack requires configurations, e.g., bridge, an IP per guest, guest drivers, devices
- The use of more devices increases the attack surface
- Sharing of files between guests and host is hard
- Relying on a specific FS is not good for immutable images
The Static WebServer

drawbacks

● Disk images consume memory and on-disk space, e.g., each guest has its own image

● Disk images have to be distributed in all the nodes

● The use of a TCP/IP stack requires configurations, e.g., bridge, an IP per guest, guest drivers, devices

● The use of more devices increases the attack surface

● Sharing of files between guests and host is hard

● Relying on a specific FS is not good for immutable images

Can we do it better?!
The Static WebServer drawbacks

- Disk images consume memory and on-disk space, e.g., each guest has its own image
- Disk images have to be distributed in all the nodes
- The use of a TCP/IP stack requires configurations, e.g., bridge, an IP per guest, guest drivers, devices
- The use of more devices increases the attack surface
- Sharing of files between guests and host is hard
- Relying on a specific FS is not good for immutable images

We propose to use of **virtio-fs** for filesystem, **virtio-vsocket** for networking and **microvm** as QEMU machine to simplify toro unikernel’s code, reduce attack surface and ease appliance configuration. Also, we propose to use **CephFS** to provide a distributed FS among VMs.
• **Virtio-fs** and **virtio-vsocket** are virtio-devices that are in QEMU since 5.0

• **Microvm** is a minimalist QEMU machine which provides a simplified device-model based on virtio. In QEMU since 5.x
An instance of virtiofsd is launched per guest.

The diagram illustrates the following components:

- **Bare-metal host**: The physical hardware on which the virtualization and networking components run.
- **KVM (Kernel-based Virtual Machine)**: The virtualization layer that allows multiple guest operating systems to run on the bare-metal host.
- **VM Context #0**: The first virtual machine context managed by KVM.
- **ToroKernel**: The virtual kernel that manages the virtio-vsocket and virtio-fs devices.
- **VMM (Virtual Machine Monitor)**: The software layer that manages the QEMU microvm and provides virtualization services.
- **QEMU microvm**: The virtual machine where the Microservice runs.
- **Host Filesystem**: The filesystem on which virtiofsd is launched.
- **Mount Point /mnt/cephfs**: The directory where the filesystem is mounted.
- **virtiofsd**: The device driver that mounts the filesystem on the guest's filesystem.

The diagram shows the flow of communication between the components, highlighting how virtiofsd is launched per guest, providing access to the host's filesystem within the guest environment.
Guest is not based on a specific FS

Only need to setup the tag and path of the mount-point

Simpler than managing files inside disk image

54% LoC less In Filesystem!

Mount Point
/mnt/cephfs

Host Filesystem

virtio vfs
virtio fs

virtiofsd

virtio-vsocket
virtio-fs

ToroKernel

VM Context #0

KVM

Bare-metal host

Host Filesystem

Mount Point
/mnt/cephfs

Simpler than managing files inside disk image

54% LoC less In Filesystem!
TCP/IP stack not needed

Simpler communication based on POSIX Socket API

No need of virtio-net

Only need to setup the CID of each guest

40% LoC less in Networking!
CephFS is a highly available and performant file store for a variety of applications.
Ceph 3-node cluster

Toro Cloud

Ceph cluster

Node1
MON
OSD
(Files)

Node2
OSD
(Files)

Node3
OSD
(Files)

Each OSD host has a /dev/sdb of 10Gb of disk

OVH Cloud nodes at ~ $16 per month per host (2 vCores, 8GB RAM)
Ceph 3-node cluster

Toro Cloud

Ceph cluster

Node1
MON
OSD (Files)

Node2
OSD (Files)

Node3
OSD (Files)

Host Client1
Mount Point
/mnt/cephfs

Host Client2
Mount Point
/mnt/cephfs

Files and binaries

OVH Cloud nodes at ~ $3 per month per host (1vCore, 2GB RAM)

ToroVMM (orchestration)
Ceph 3-node cluster

Toro Cloud

Ceph cluster

Node1
MON
OSD
(Files)

Node2
OSD
(Files)

Node3
OSD
(Files)

Host Client1
Mount Point
/mnt/cephfs

Host Client2
Mount Point
/mnt/cephfs

Microservice1

Microservice2

ToroKernel

virio-fs

virtio-vsocket

ToroVMM
(orchestration)

Outside the host

QEMU microvm
Results

- Binary Size: 235 Kb that includes kernel and user’s application
- Time to rebuild the microservice: ~ 500ms
- Boot cycle: ~ 80ms
  - `$echo “Hello World”` is ~2.6 ms
- CPU Usage: 90% at high and 10% sleep
- Memory footprint per VM: 2.9% (~ 60Mb) or 35 VMs per hosts
  - QEMU compiled with all enabled
- Price: 58 euros/month ~ 0.85 euros/month per VM
- See https://github.com/torokernel/torocloudscripts
Results

- Binary Size: 235 Kb that includes kernel and user’s application
- Time to rebuild the microservice: ~ 500ms
- Boot cycle: ~ 80ms
  - `$echo "Hello World"` is ~2.6 ms
- CPU Usage: 90% at high and 10% sleep
- Memory footprint per VM: 2.9% (~ 60Mb) or 35 VMs per hosts
  - QEMU compiled with all the configuration
- Price: 58 euros/month ~ 0.85 euros/month per VM
- See https://github.com/torokernel/torocloudscripts

“It is all talk until code runs.” - Ward Cunningham
Challenges

- Support live-migration which is not currently supported by microvm machine
- Improve bottleneck at vsocket forwarding
- Improve overall performance by using zero copy in virtio-fs and virtio-vsocket
- Improve evaluation by comparing with unikernels/containers/gpos