Toro, a Dedicated Kernel for Microservices

Matias E. Vara Larsen
Silicon-Gears

Cesar Bernardini
Barracuda Networks
What is a microservice?

1. Monolithic
   - Logging
   - Order
   - Catalog

2. Microservices
   - Logging
   - Microservice #0
   - Order
   - Microservice #1
   - Catalog
   - Microservice #2
Service Instance per Virtual Machine (VM)

- Bare-metal host
- Hypervisor
- VM Context #0
- Microservice #0
- Service Instance per Virtual Machine (VM)
- VM Context #1
- Microservice #1
Each VM requires its own OS.
The use of VMs to host microservices allows to isolate different services.
VMs take long time to get up and run

The creation and storage of VMs are not simple

Too much complexity for a single purpose usage

Limited number of VM instances

VM Context #0

VM Context #1

Hypervisor

Bare-metal host
Toro is simple kernel that allows microservices to run efficiently in VMs thus leveraging the strong isolation VMs provide.
Ingredients for ToroKernel

- User application within the kernel
- Cooperative Scheduler
- Dedicated Resources
- Single Thread Event Loop Networking
User application within the Kernel

- In general purpose OS, the user application executes as a process in the less privileged mode, e.g., ring3 in x86.
- The communication from the user application to the kernel relies on syscalls.
- Context switch needs to switch from kernel mode to user mode, e.g., interruption and scheduling.
User application within the Kernel

- In general purpose OS, the user application executes as a process in the less privileged mode, e.g., ring3 in x86.
- The communication from the user application to the kernel relies on syscalls.
- Context switch needs to switch from kernel mode to user mode, e.g., interruption and scheduling.

This is too expensive! Can we do it better for a single purpose kernel?
User application within the Kernel Proposal

1. Run kernel and user application in the most privileged level.
2. Rely on the hypervisor to isolate the context of each VM.
3. Use a flat memory model that is shared by the kernel and the user application.
4. Use only threads.
5. Provide a simple kernel API dedicated to microservices.
Toro Kernel

- Filesystem
- Memory
- Devices
- Networking

Only the needed components are included

Microservice

Toro.elf
The generated image is **immutable**, i.e., the generated image can be used across different hypervisors without the need of recompile it.
Ingredients for ToroKernel

• User application within the kernel
• Cooperative Scheduler
• Dedicated Resources
• Single Thread Event Loop Networking
Cooperative Scheduler

- In a General Purpose OS, the scheduler is in charge to distribute the CPU time for each process.
- When a task has consumed its time, the scheduler switches to the next ready task.
- The scheduler relies on a timer.
Cooperative Scheduler

- In a General Purpose OS, the scheduler is in charge to distribute the CPU time for each process.
- When a task has consumed its time, the scheduler switches to the next ready task.
- The scheduler relies on a timer.

Can we do it better for a single purpose kernel?
Cooperative Scheduler Proposal

1. Cooperative scheduler, i.e., each thread decides when to yield the CPU

2. Simple scheduler, i.e., the scheduler chooses the first thread in ready state

2. One scheduler per core

3. Remote creation of threads by relying on a lock-free algorithm
Order

DataBase  Microservice

Threads

BeginThread(DataBase, Thread1, Core1)
BeginThread(Microservice, Thread2, Core2)
Better cache performance!

One core one task!

Order

DataBase

Thread 1

Core 1

Microservice

Thread 2

Core 2
Ingredients for ToroKernel

- User application within the kernel
- Cooperative Scheduler
- Dedicated Resources
- Single Thread Event Loop Networking
Dedicated Resources

• In a multicore system, the problematic resource is the shared memory. The use of shared memory causes:
  – Overhead in the memory bus
  – Overhead in the cache to keep it coherent
  – Overhead to guaranty mutual exclusion, e.g., use of spin-locks
Dedicated Resources Proposal

• Toro improves memory access by keeping the resources locals:
  − The memory is dedicated per core
  − The kernel data structures are dedicated per core
  − The access to kernel data structures is lock free
Dedicated Resources

Memory space in Toro

Memory Region 1

TORO Memory allocator

Memory Region 2

ToroGetMem() Thread 1 Core 1

ToroGetMem() Thread 2 Core 2

Access to this region can be improved by using Hypertransport or Intel Quick Path.
Dedicated Resources

DedicateBlockDriver(Disk1, Core1)
DedicateNetworkDriver(Network1, Core2)
Dedicated Resources

- Core 1
- Core 2
- Disk1
- Network1
- Memory 1
- Memory 2
- DedicatedResources

- DedicateBlockDriver(Disk1, Core1)
- DedicateNetworkDriver(Network1, Core2)

E.g., ATA Disks
E.g., E1000, VirtIO
Dedicated Resources

- Ext3
- Disk 1
- Memory 1
- Core 1

- Network Stack
- Network Card
- Memory 2
- Core 2
Dedicated Resources

Messages, shared memory or any mechanism to communicate between threads

Data Base
- Thread 1
  - Ext3
  - Disk 1
  - Memory 1
  - Core 1

Microservice
- Thread 2
  - Network Stack
  - Network Card
  - Memory 2
  - Core 2
Messages, shared memory or any mechanism to communicate between threads

In Toro, microservices are first-class objects
Ingredients for ToroKernel

• User application within the kernel
• Cooperative Scheduler
• Dedicated Resources
• Single Thread Event Loop Networking
Single Thread Event Loop

• Toro networking is based on the single thread event loop model [1], i.e., one thread per microservice

• The kernel provides a dedicated API to create microservices

• The kernel implements the microservice and improves the CPU usage[2]

Single Thread Event Loop

The user defines how to react to events

Accept()  Receive()  Close()

Microservice
Single Thread Event Loop

The user registers a set of callbacks

Kernel

Accept() | Receive() | Close()
Single Thread Event Loop

The thread schedules each socket

Each microservice is implemented by using one thread.
Single Thread Event Loop

Kernel

Service Thread “A”

Socket Scheduler

- Socket #0
- Socket #1
- ... Socket #N

Microservice

- Accept()
- Receive()
- Close()

The Event Loop halts the core if no connection arrives

SysServiceCreate()
“It’s all talk until the code runs.” - Ward Cunningham
HelloWorld Microservice Example

• We implement a simple microservice that responds “Hello World”
• We implement it by using three approaches: Docker, Ubuntu guest (KVM) and Toro guest (KVM)
• We compare these approaches in term of:
  - Deploying Time
  - Bootstrap Time
  - Image Size
  - CPU usage
  - Time per Request

Microservice footprint
## HelloWorld Microservice Example setup

<table>
<thead>
<tr>
<th>Docker</th>
<th>General OS/KVM</th>
<th>Toro/KVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Cpu limit</td>
<td>• Cpu limit</td>
<td>• CPU limit</td>
</tr>
<tr>
<td>• Memory limit</td>
<td>• Memory limit</td>
<td>• Memory limit</td>
</tr>
<tr>
<td>• NGINX</td>
<td>• NGINX</td>
<td>• Toro WWW server</td>
</tr>
<tr>
<td>• UWSGI (4 processes)</td>
<td>• UWSGI (4 processes)</td>
<td></td>
</tr>
<tr>
<td>• Flask</td>
<td>• Flask</td>
<td></td>
</tr>
</tbody>
</table>
Deploying Time

- Time required to build an image within the microservice
Bootstrap Time

- Time to boot and to answer the first request
Image Size

- Size of the image that contains the microservice and its dependencies.
End-User Delay

- Benchmarking with ab and measuring the Time per Request (mean) [ms]

<table>
<thead>
<tr>
<th>Approach</th>
<th>CPUs</th>
<th>200</th>
<th>500</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Docker</td>
<td>4</td>
<td>139.980 ms</td>
<td>333.937 ms</td>
<td>801.422 ms</td>
</tr>
<tr>
<td>Ubuntu/KVM</td>
<td>4</td>
<td>94.507 ms</td>
<td>238.149 ms</td>
<td>560.513 ms</td>
</tr>
<tr>
<td>TORO/KVM</td>
<td>1</td>
<td>120.065 ms</td>
<td>301.736 ms</td>
<td>596.792 ms</td>
</tr>
</tbody>
</table>
Take away lessons

- Minimal image size (< 4MB)
  - NGINX docker image is 15-times the size of a Toro image
- Continuous Integration: 1 sec to re-deploy a microservice
  - Deploy an OS w/similar configuration takes 300 sec, with docker ~50 sec
- Time per Requests
  - Comparable level with cutting edge technology (NGINX)
- CPU Usage
  - Comparable with Docker
  - Toro is 100% isolated from the host OS, Docker is not.
Summary

- Toro is a kernel dedicated to run microservices
- Toro provides a dedicated API to specify microservices
- Toro design is improved in four main points:
  - Booting time and building time
  - communication to kernel
  - memory access
  - networking
Future Work

- Ease tooling to develop, test and debug microservices
- Investigate new use-cases
- Investigate the porting of applications
- Investigate new ideas to improve the network stack for microservices, e.g., improve socket scheduling for http, resource allocation algorithm
QA

- http://www.torokernel.io
- torokernel@gmail.com
- Twitter @torokernel
- Torokernel wiki at github
  - My first Three examples with Toro
- Test Toro in 5 minutes (or less...)
  - torokernel-docker-qemu-webservices at Github
Thanks!