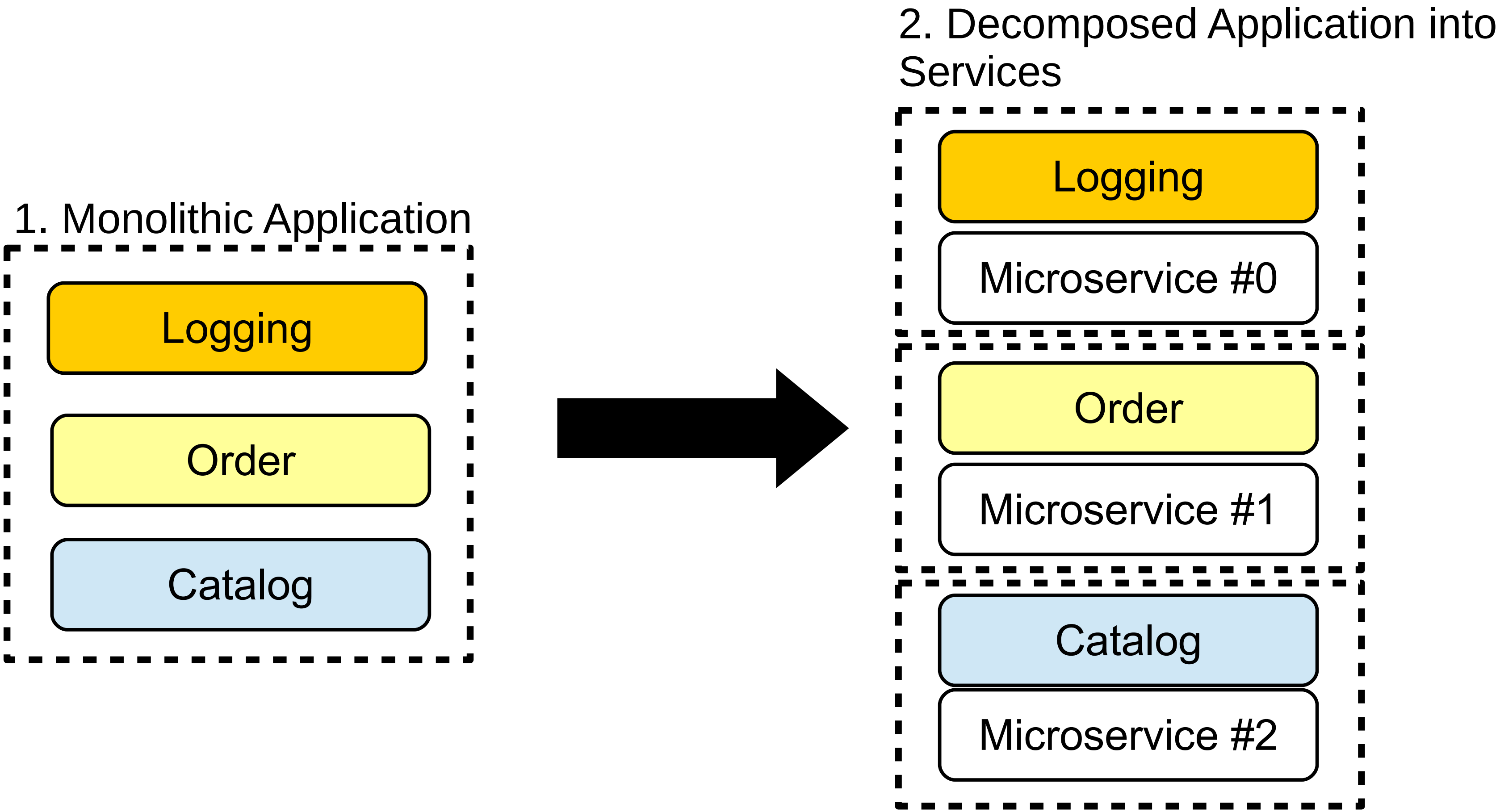
Toro, a Dedicated (Uni)kernel for Microservices

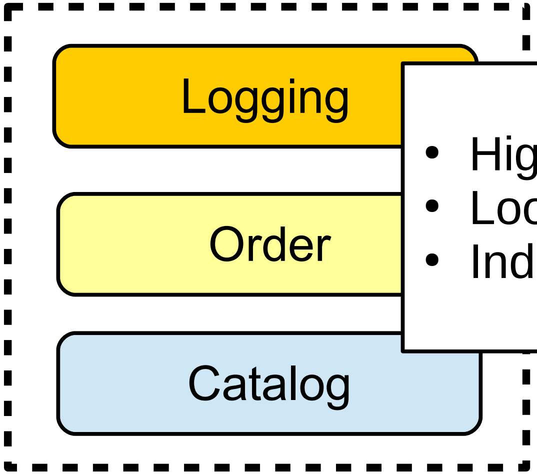
Dr. Matias E. Vara Larsen

What are microservices?

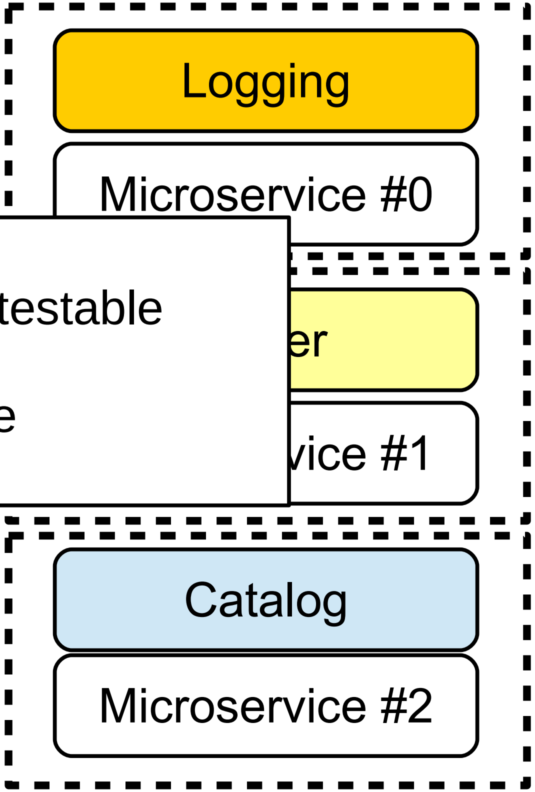


What are microservices?

1. Monolithic Application

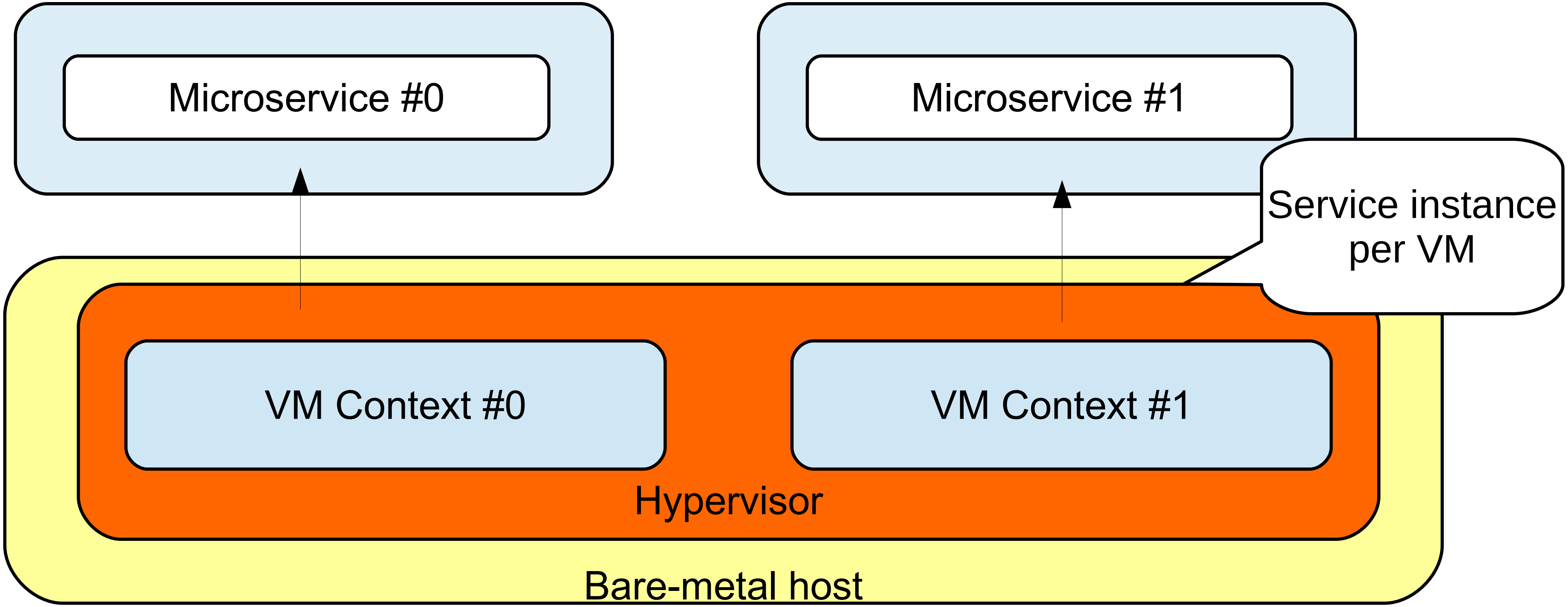


2. Decomposed Application into Services

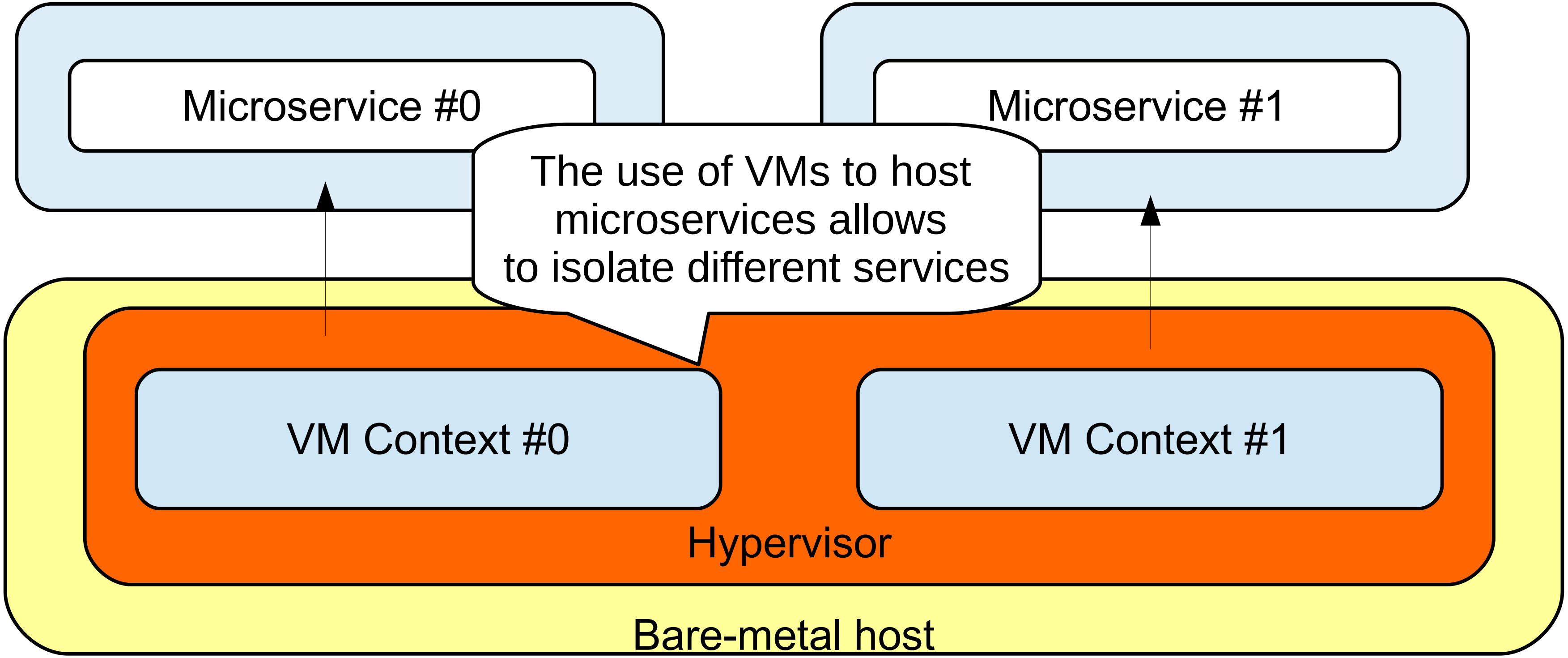


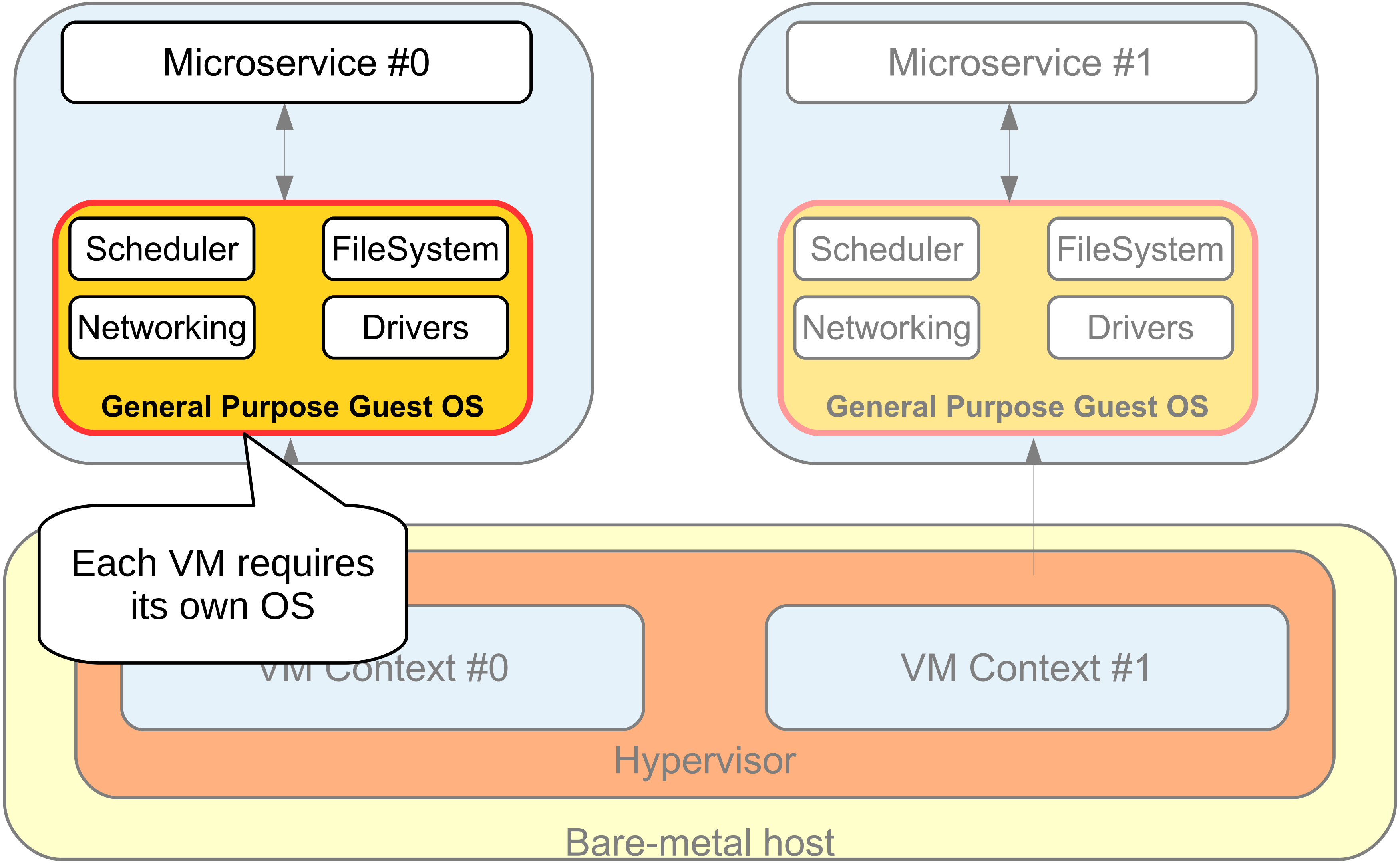
- Highly maintainable and testable
- Loosely coupled
- Independently deployable

How are microservices deployed?

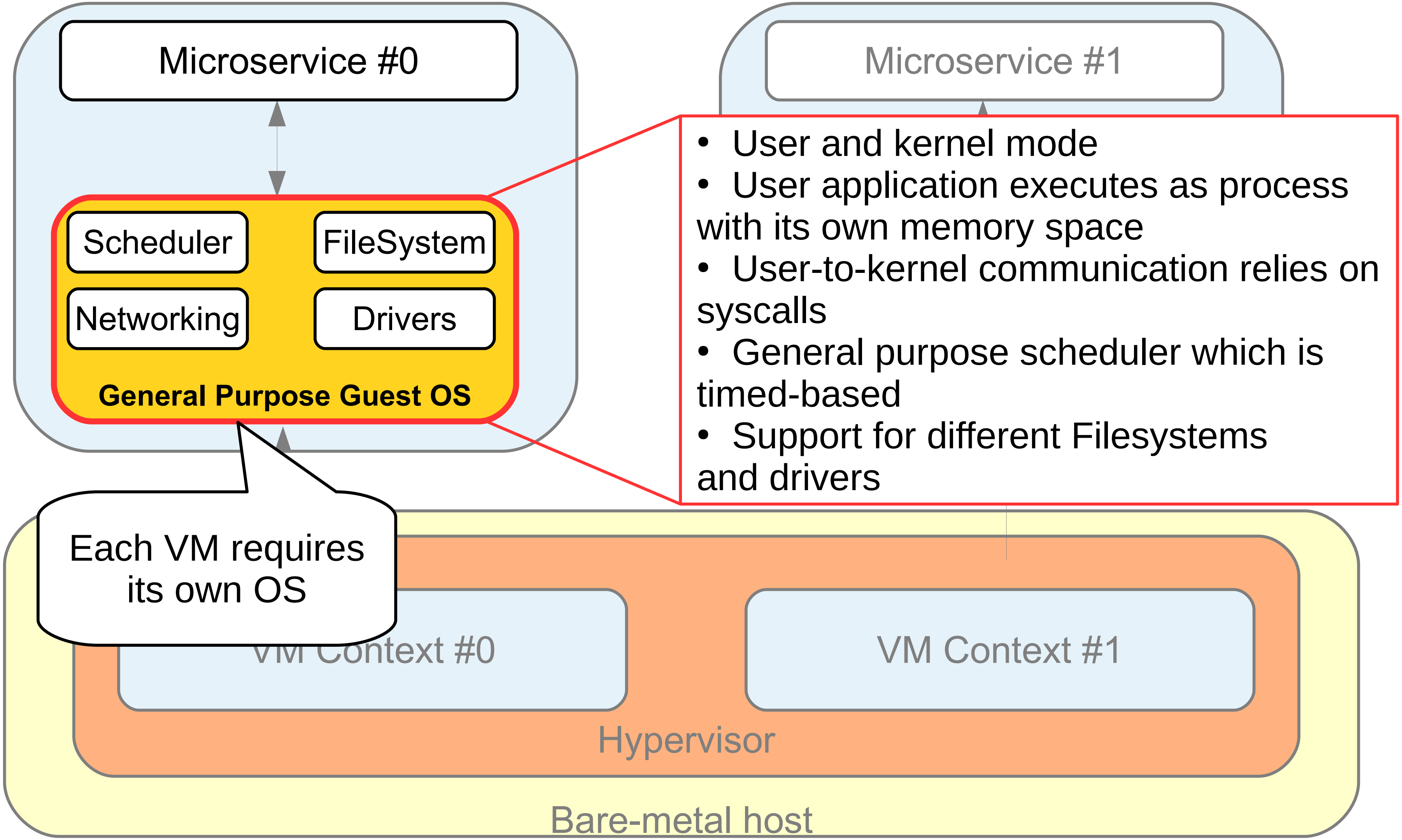


How are microservices deployed?

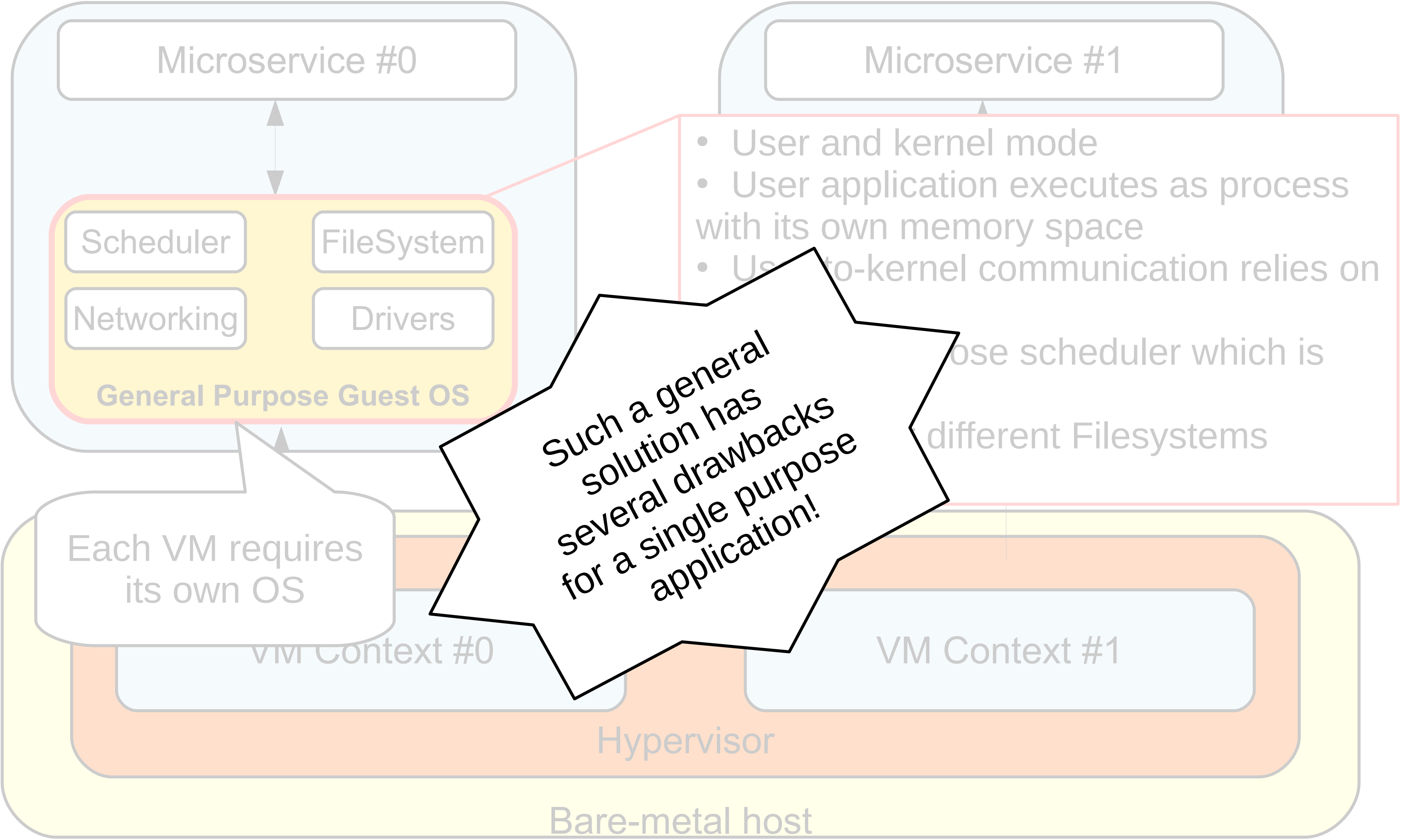


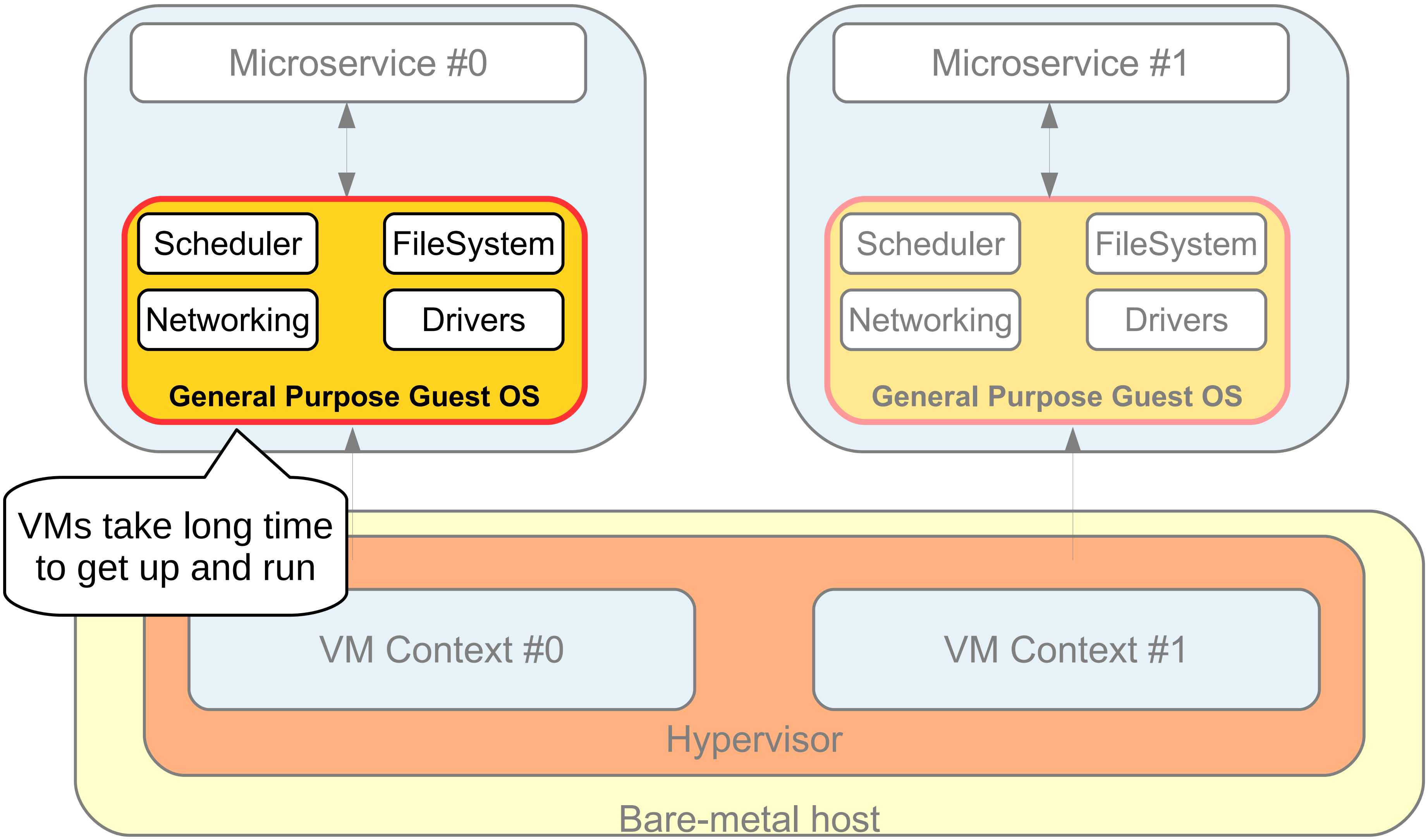


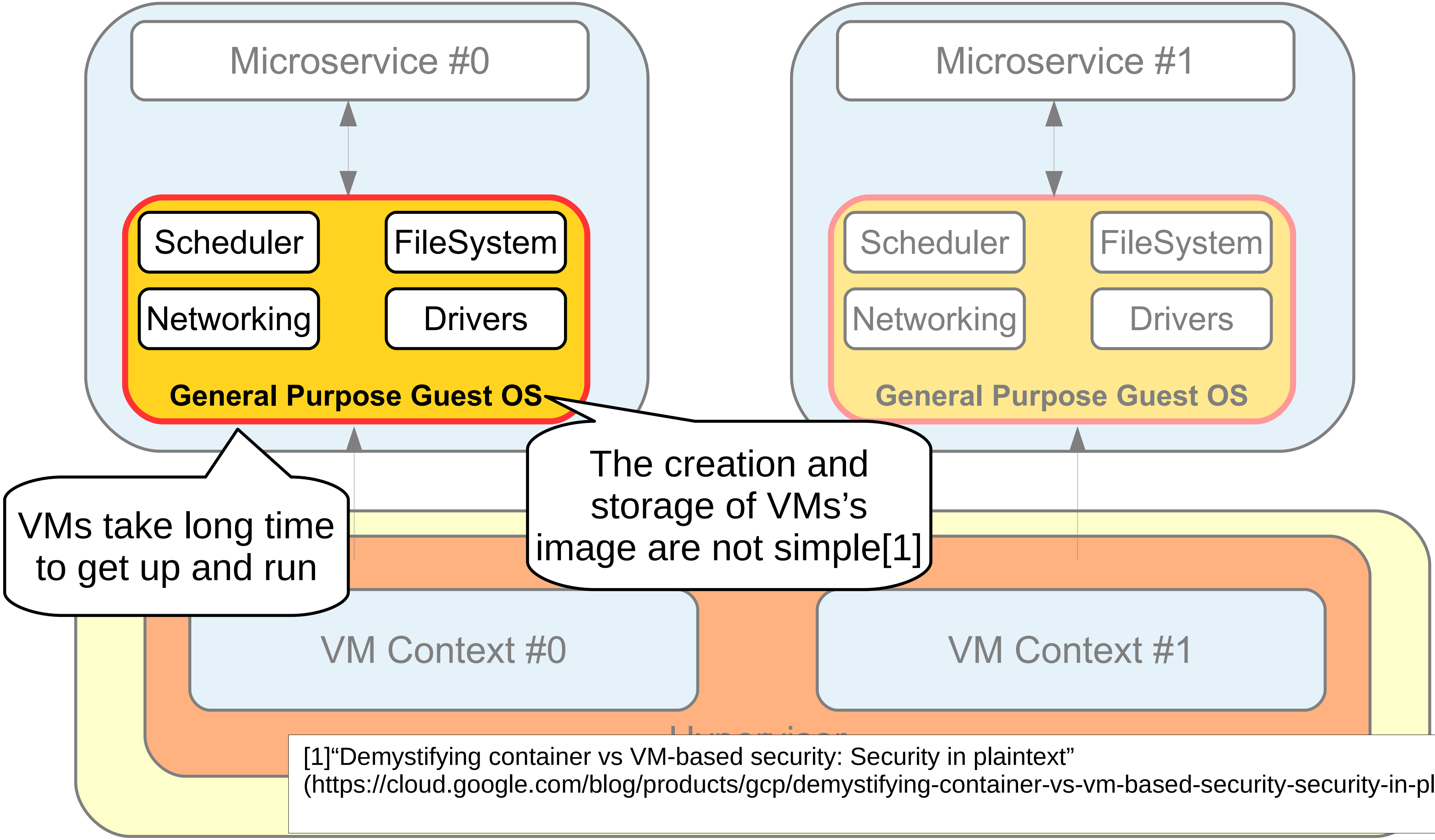
Microservice #0
Scheduler
FileSystem
Networking
Drivers
General Purpose Guest OS
Microservice #1
Scheduler
FileSystem
Networking
Drivers
General Purpose Guest OS
Each VM requires its own OS
VM Context #1
Hypervisor
Bare-metal host



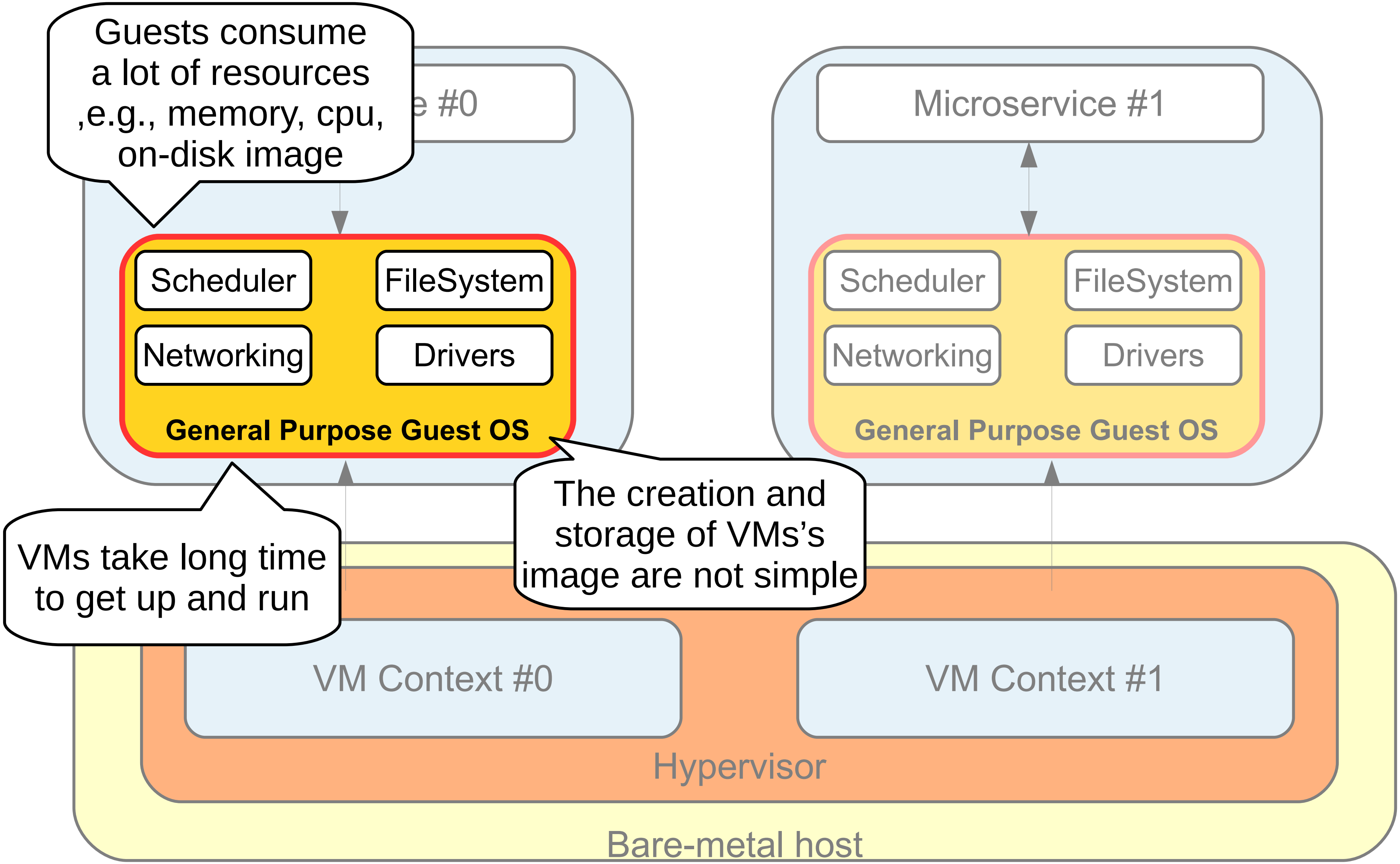
Microservice #0
Microservice #1
Scheduler
FileSystem
Networking
Drivers
General Purpose Guest OS
• User and kernel mode
• User application executes as process with its own memory space
• User-to-kernel communication relies on syscalls
• General purpose scheduler which is timed-based
• Support for different Filesystems and drivers
Each VM requires its own OS
VM Context #0
VM Context #1
Hypervisor
Bare-metal host

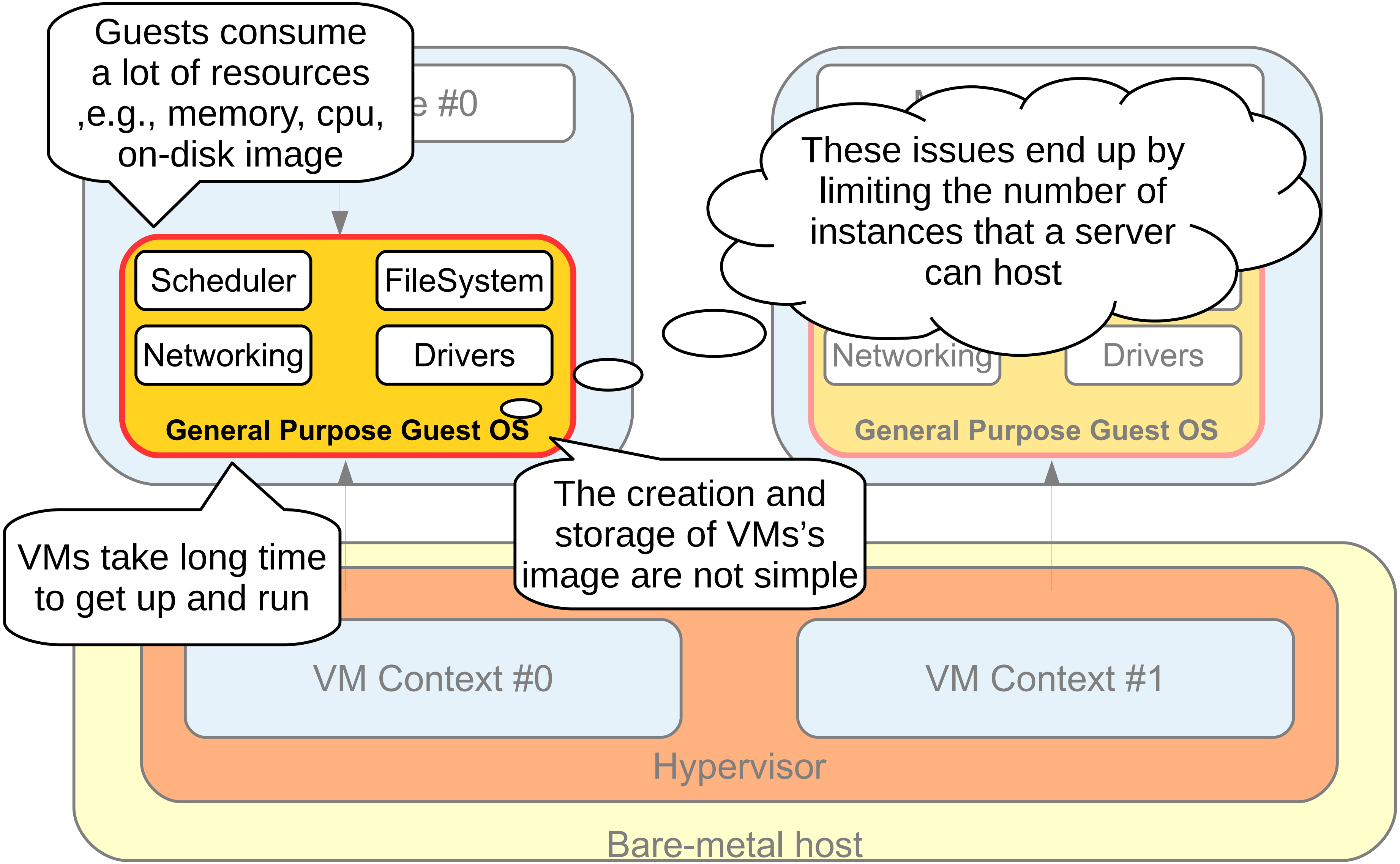

Microservice #0
Microservice #1
Scheduler
FileSystem
Networking
Drivers
General Purpose Guest OS
User and kernel mode
User application executes as process with its own memory space
different Filesystems
Each VM requires its own OS
Such a general solution has several drawbacks for a single purpose application!
VM Context #0
VM Context #1
Hypervisor
Bare-metal host

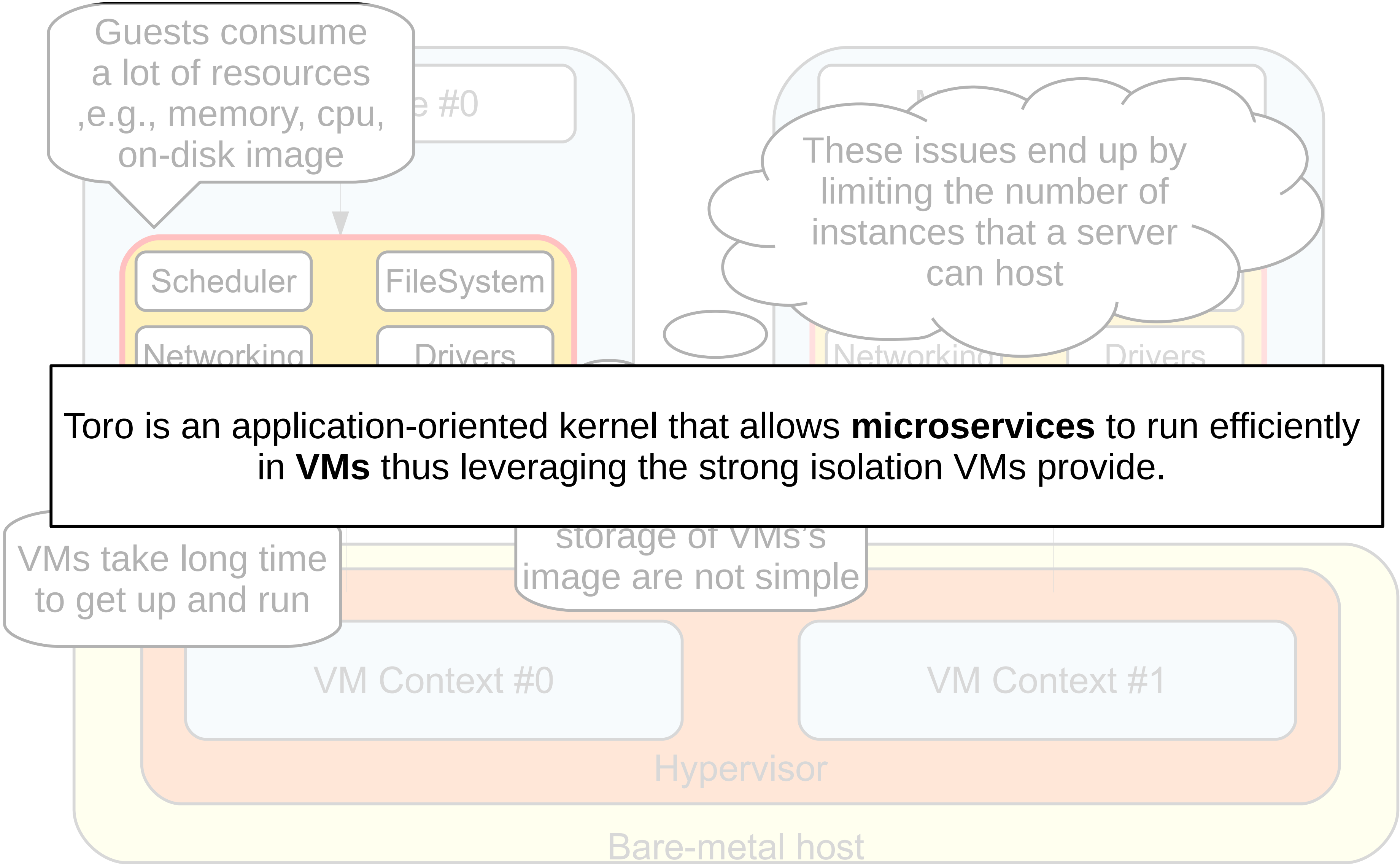

Microservice #0
Scheduler
FileSystem
Networking
Drivers
General Purpose Guest OS
Microservice #1
Scheduler
FileSystem
Networking
Drivers
General Purpose Guest OS
VMs take long time to get up and run
VM Context #0
VM Context #1
Hypervisor
Bare-metal host



[1]"Demystifying container vs VM-based security: Security in plaintext" (https://cloud.google.com/blog/products/gcp/demystifying-container-vs-vm-based-security-security-in-plaintext)


Guests consume a lot of resources ,e.g., memory, cpu, on-disk image
e #0
Microservice #1
Scheduler
FileSystem
Networking
Drivers
General Purpose Guest OS
Scheduler
FileSystem
Networking
Drivers
General Purpose Guest OS
VMs take long time to get up and run
The creation and storage of VMs's image are not simple
VM Context #0
VM Context #1
Hypervisor
Bare-metal host


Guests consume a lot of resources ,e.g., memory, cpu, on-disk image
e #0
These issues end up by limiting the number of instances that a server can host
Scheduler
FileSystem
Networking
Drivers
General Purpose Guest OS
Networking
Drivers
General Purpose Guest OS
VMs take long time to get up and run
The creation and storage of VMs's image are not simple
VM Context #0
VM Context #1
Hypervisor
Bare-metal host

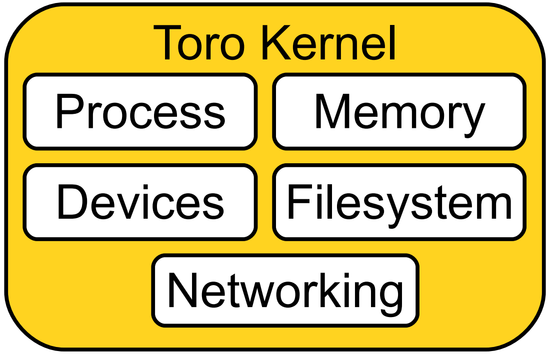


Toro is an application-oriented kernel that allows **microservices** to run efficiently in **VMs** thus leveraging the strong isolation VMs provide.

Ingredients for ToroKernel

- Application-oriented Kernel
- Simple Scheduler
- Dedicated Resources
- Microservice-oriented Networking

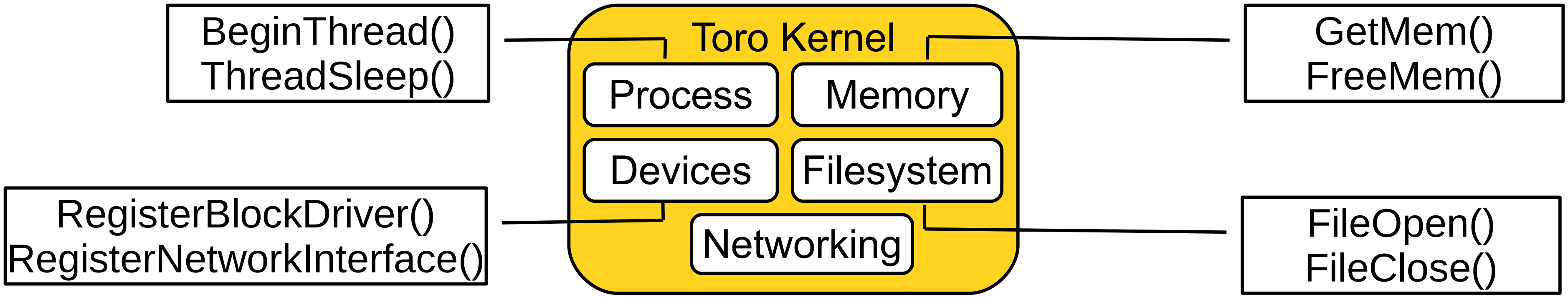
Application-oriented Kernel



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

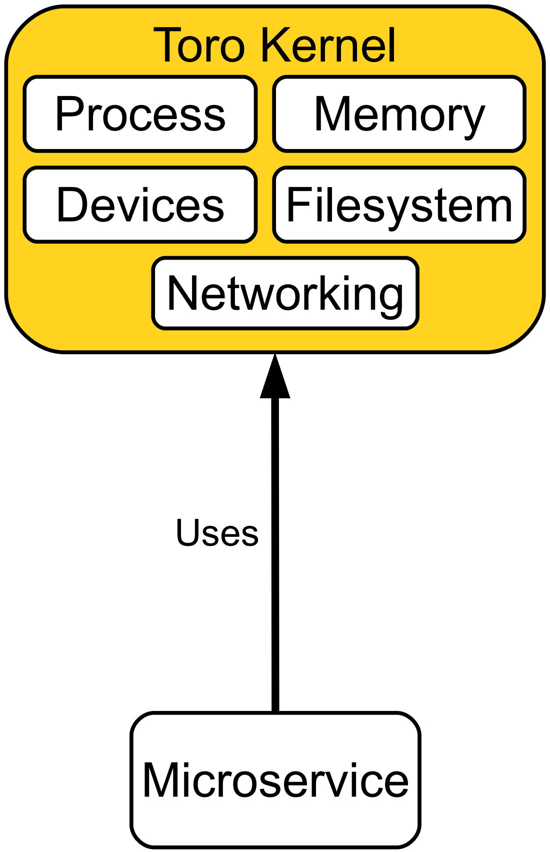
Application-oriented Kernel



Toro is an embedded kernel including five units:
- Process
- Memory
- Filesystem
- Networking
- Devices, e.g., Block Device, Network Device

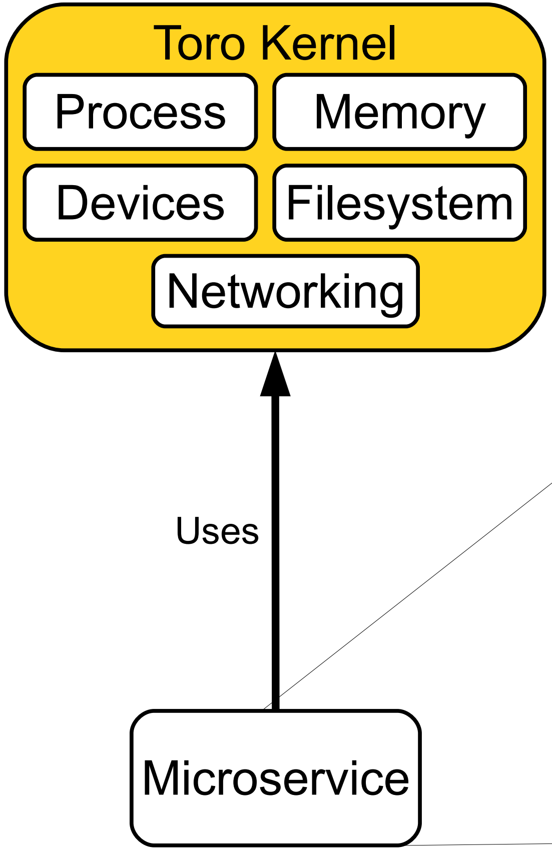
Each unit provides minimalist APIs accessible from the embedded application

Application-oriented Kernel



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

Application-oriented Kernel



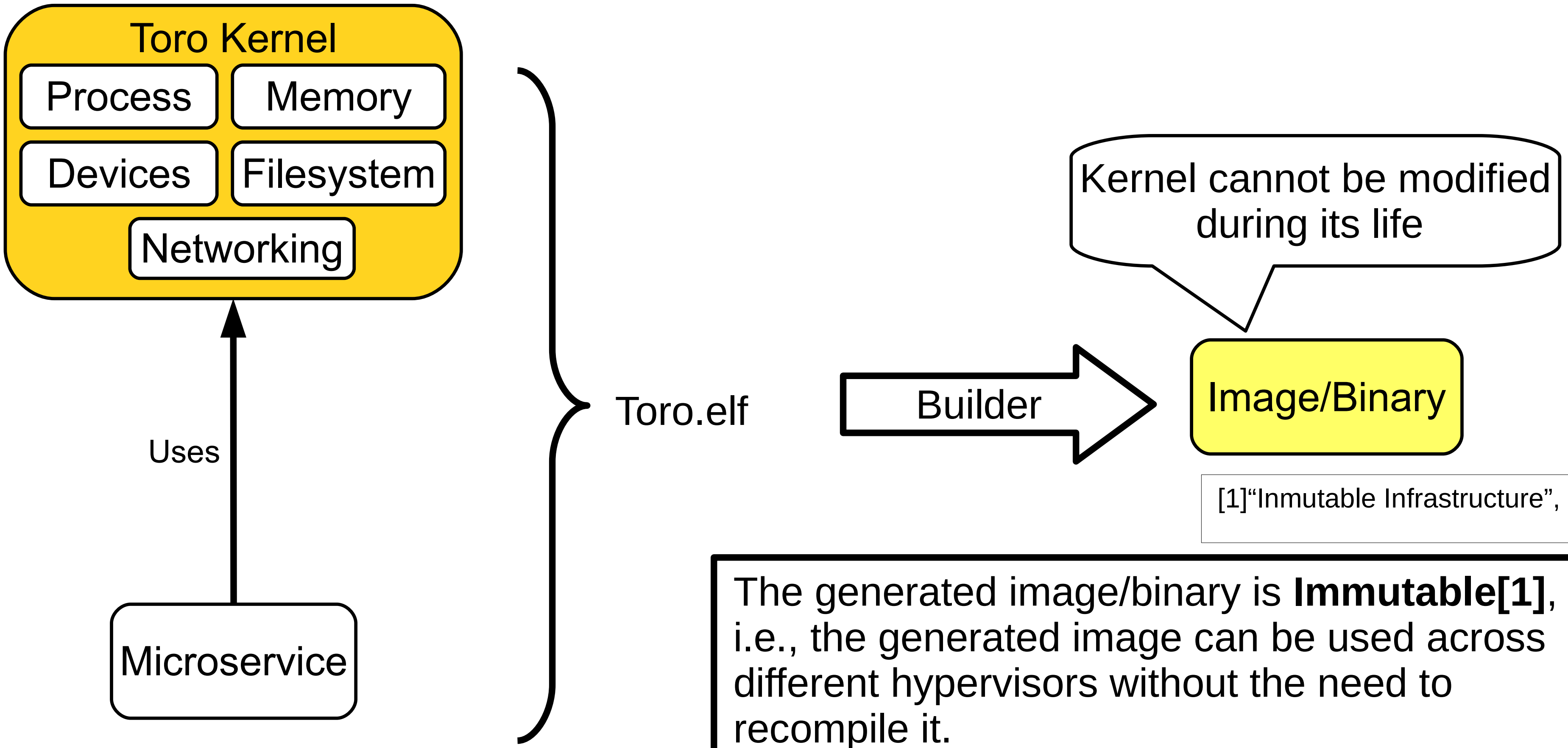
- User application and kernel units are compiled in a single binary
- The
com

```
program HelloWorldMicroservice;

uses
  Memory,
  Filesystem,
  Ext3,
  E1000;

Begin
//
// Your Code
//
End.
```

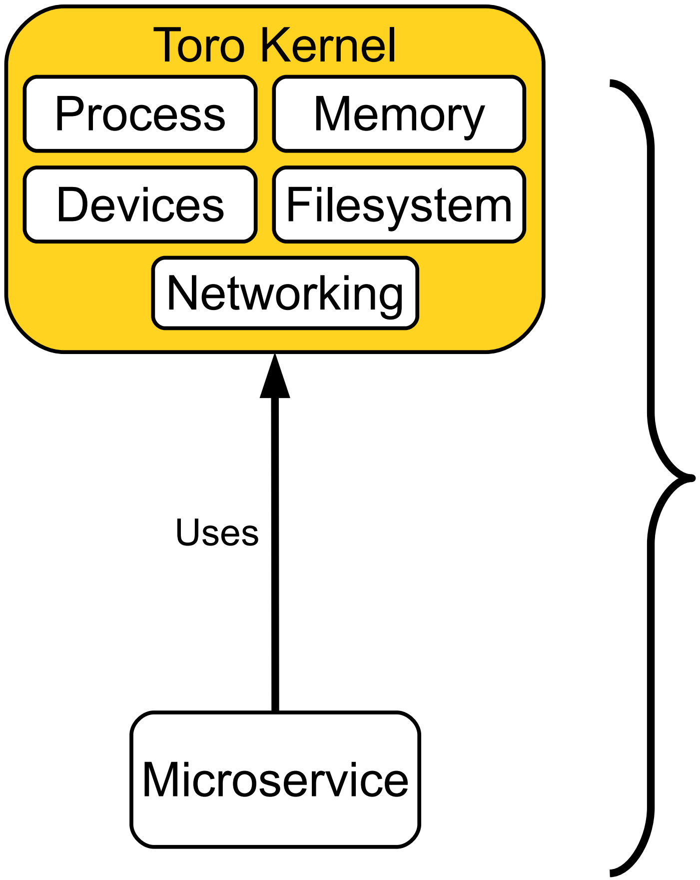
Application-oriented Kernel

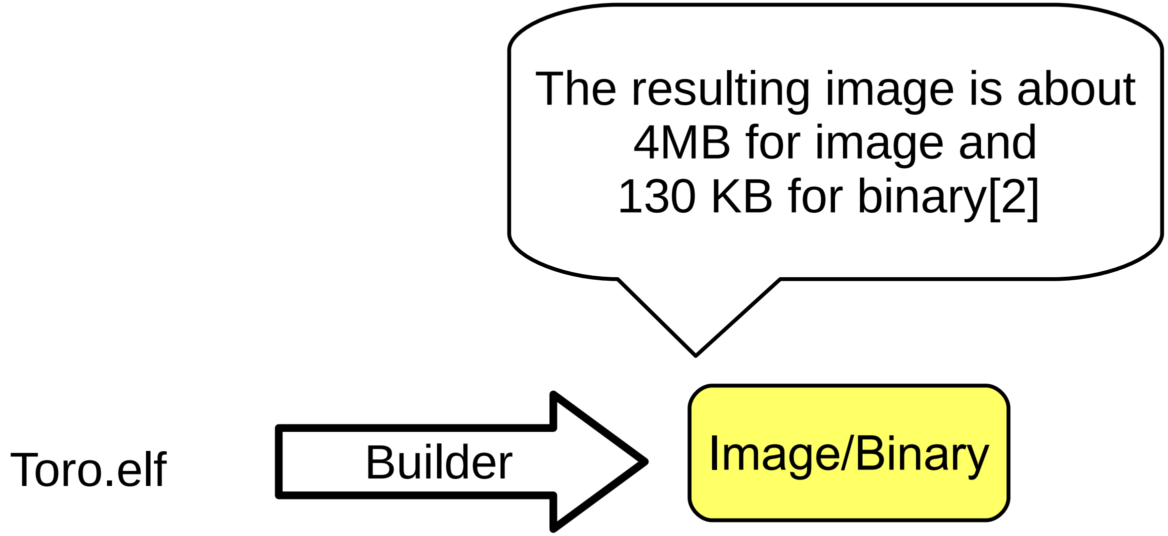


[1]“Inmutable Infrastructure”, Stella

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

Application-oriented Kernel

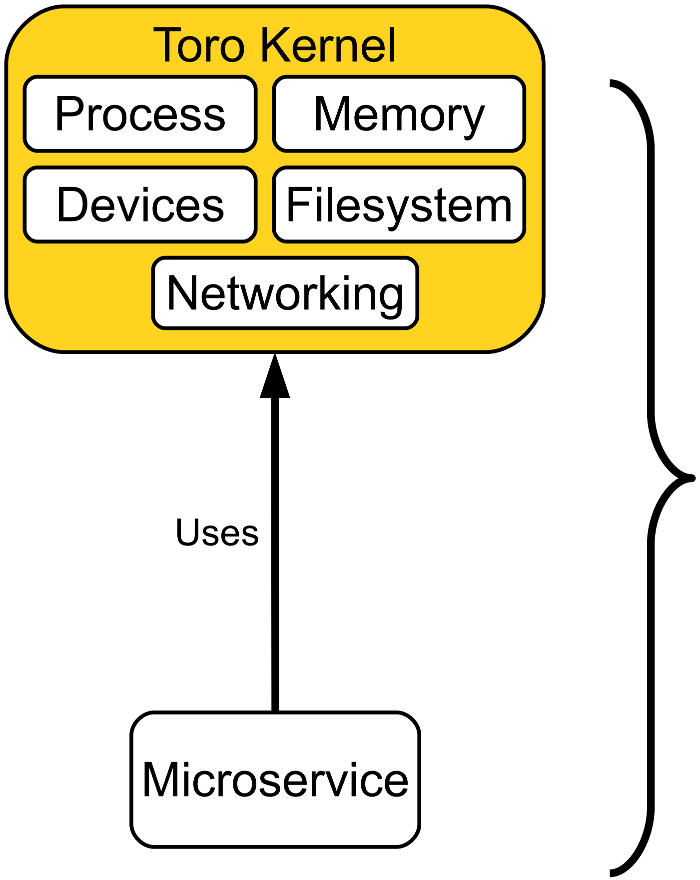


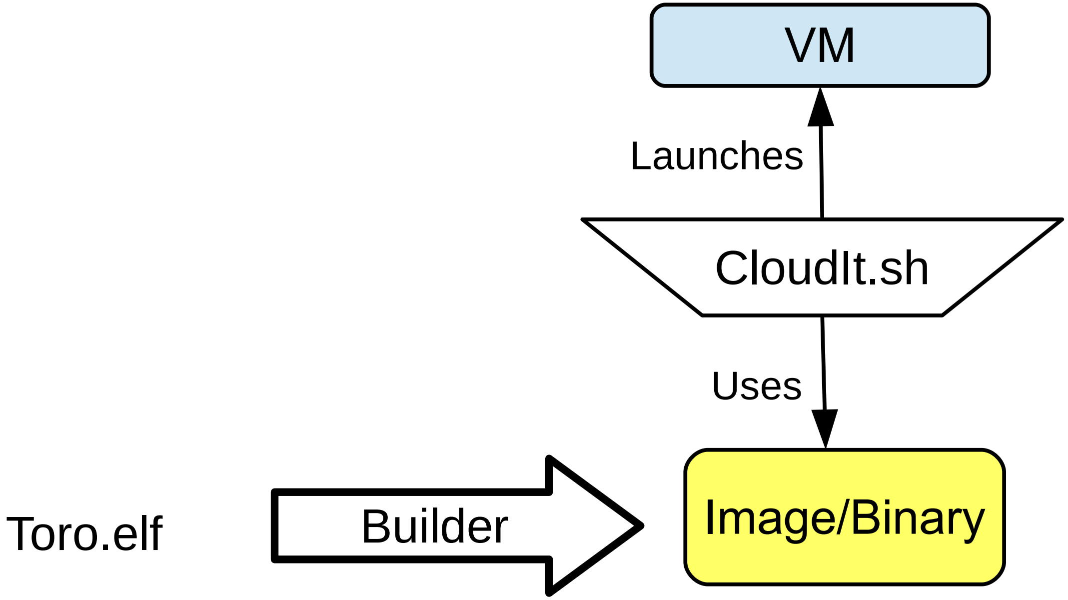


[2] Speed up booting time of a Toro appliance, FOSDEM’19

The generated image/binary is **Immutable**, i.e., the generated image can be used across different hypervisors without the need to recompile it.

Application-oriented Kernel

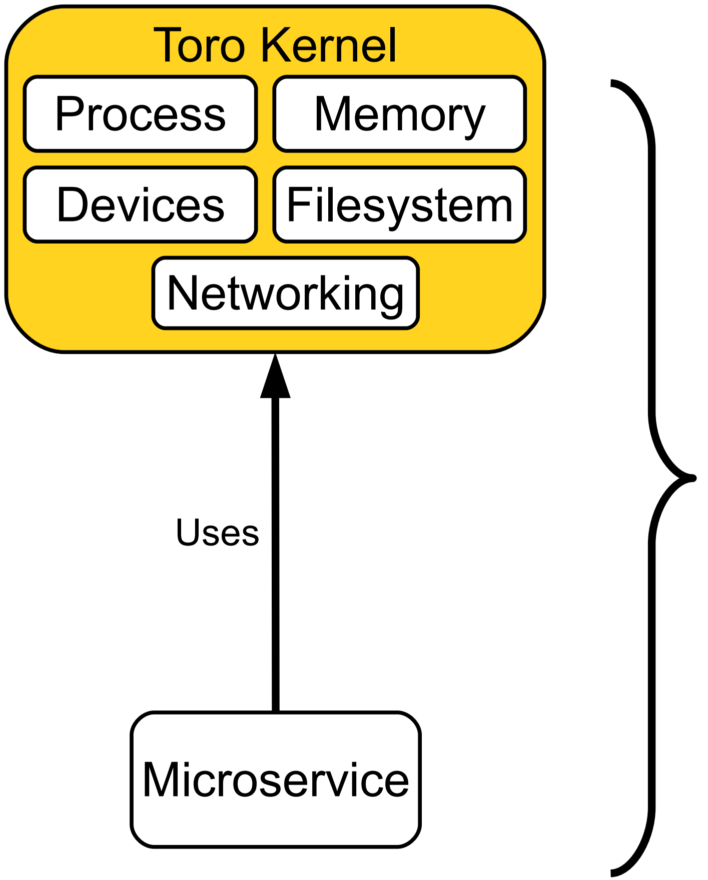


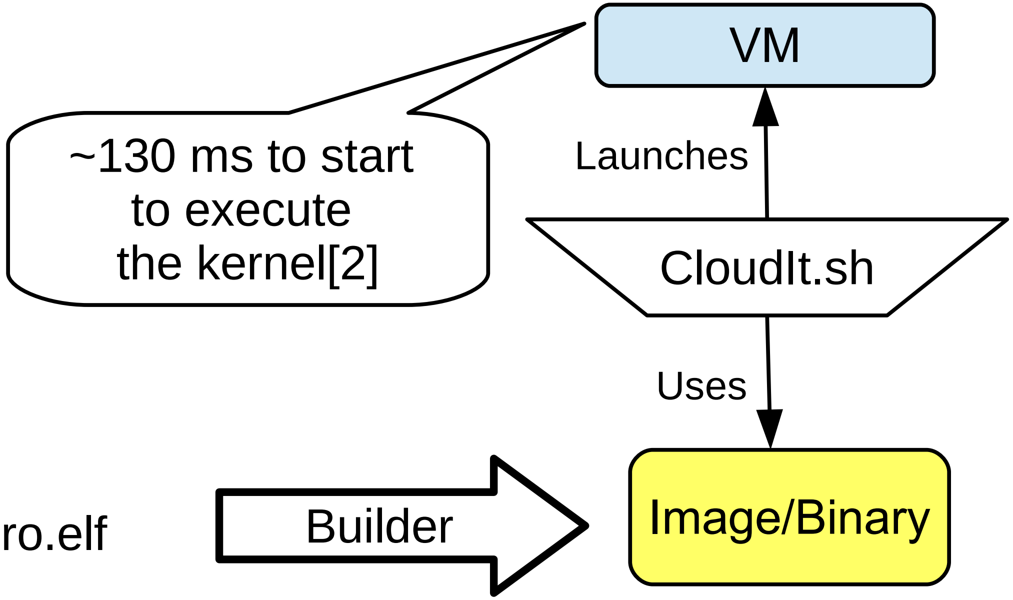


[2] Speed up booting time of a Toro appliance, FOSDEM’19

The generated image/binary is **Immutable**, i.e., the generated image can be used across different hypervisors without the need to recompile it.

Application-oriented Kernel





[2] Speed up booting time of a Toro appliance, FOSDEM'19

The generated image/binary is **Immutable**, i.e., the generated image can be used across different hypervisors without the need to recompile it.

Benefits of Application-oriented Kernel

- Security is based on the hypervisor which ensures a reduced attack surface
- The kernel size is smaller since it only includes the units required for the application
- The smaller kernel size reduces the booting time and eases image manipulation, i.e., your kernel is your program!
- No communication overhead since application is using the kernel APIs

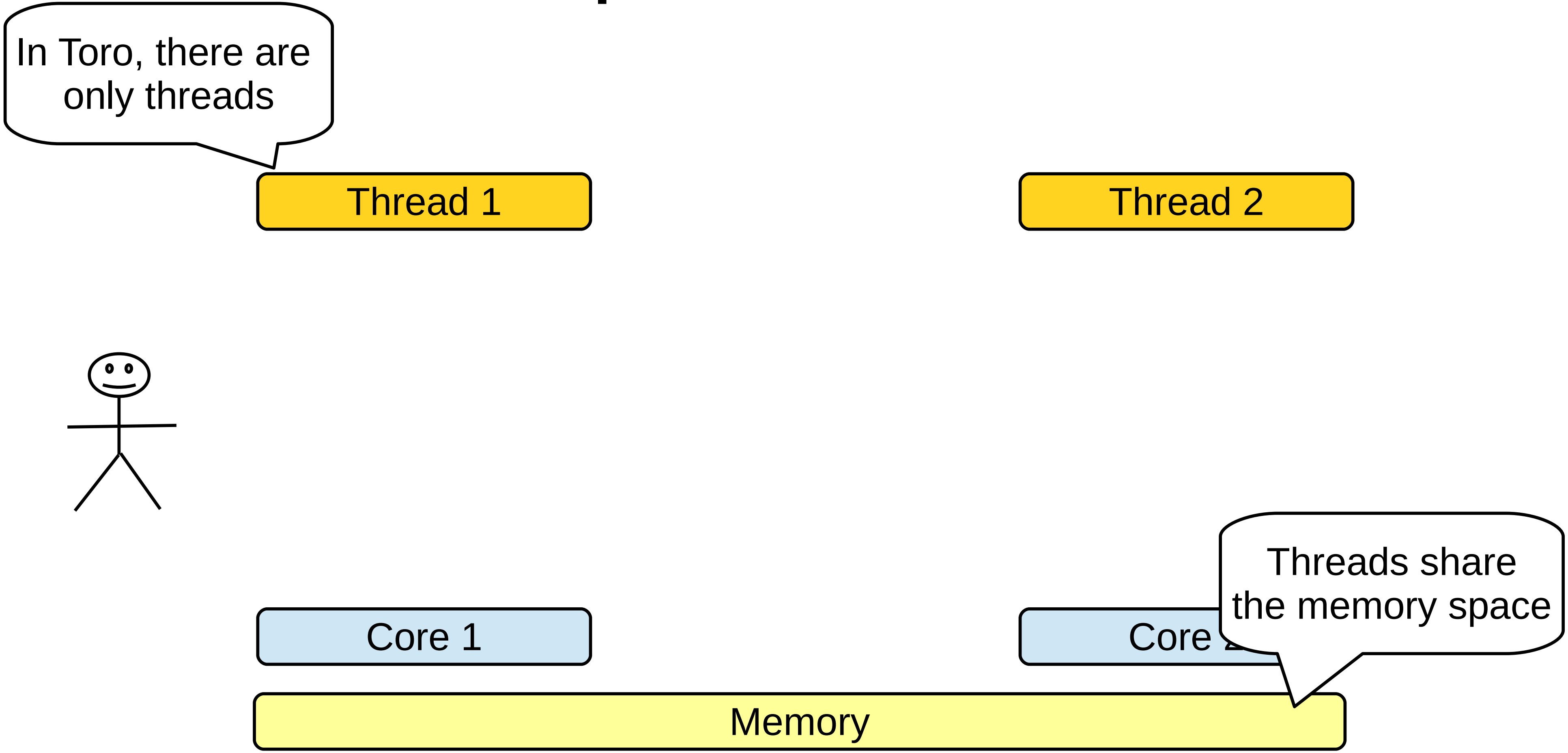
Ingredients for ToroKernel

- Application-oriented Kernel
- Simple Scheduler
- Dedicated Resources
- Microservice-oriented Networking

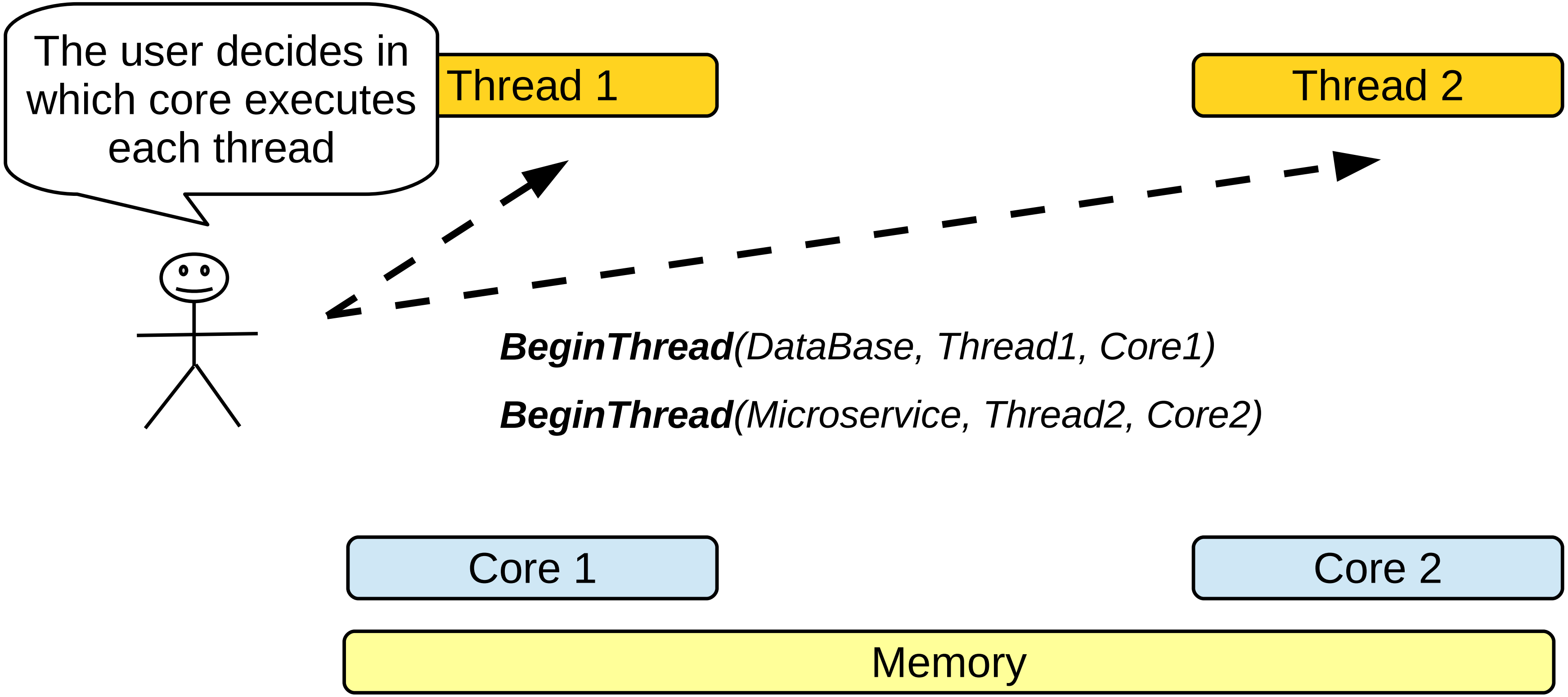
Simple Scheduler Requirements

- Scale with the number of cores
- Simple scheduler’s algorithm, i.e., Cooperative threading model
- Minimalist context switches

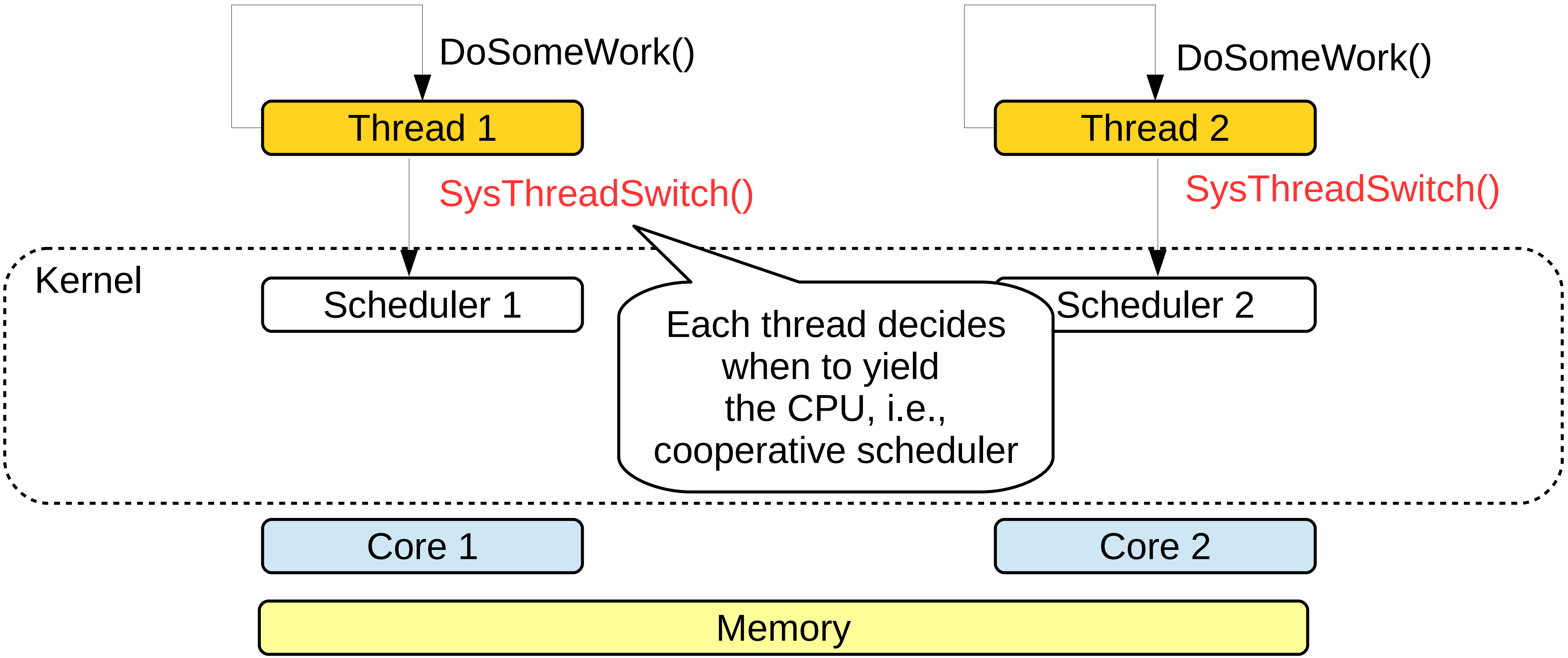
Simple Scheduler



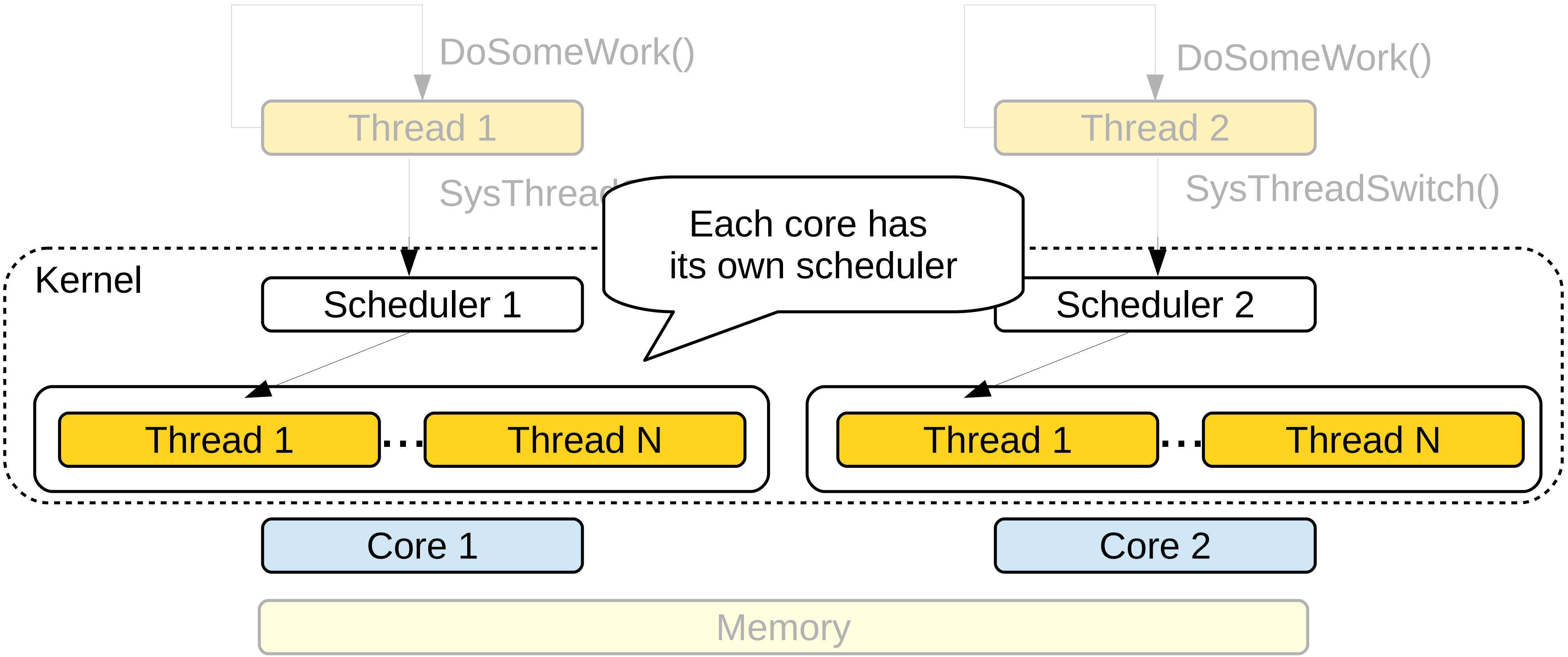
Simple Scheduler



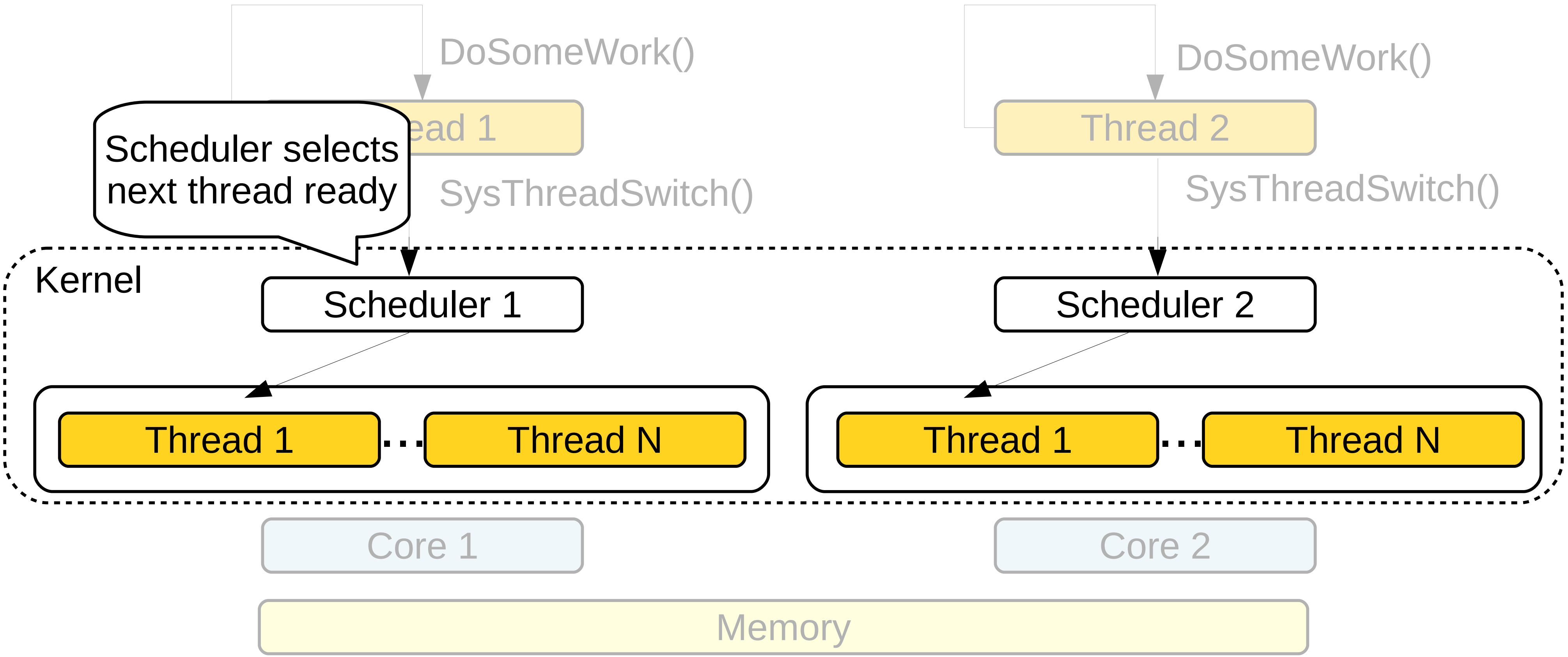
Simple Scheduler



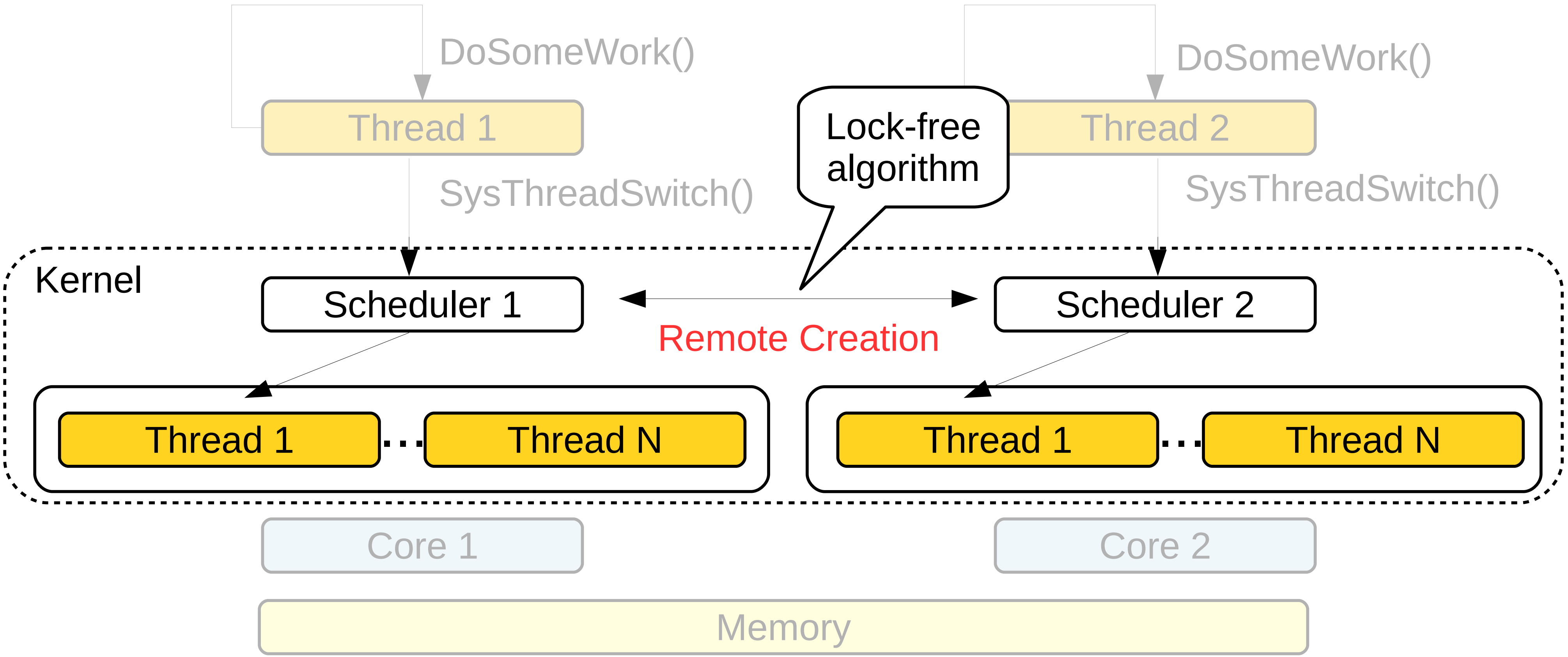
Simple Scheduler



Simple Scheduler



Simple Scheduler



Benefits of Simple Scheduler

- The use of threads makes context switching cheaper
- The use of a cooperative scheduler:
  - simplifies user’s and kernel’s code by avoiding to implement protection inside the scheduler
  - reduces number of context switches since it does not relies on interruptions

Ingredients for ToroKernel

- Application-oriented Kernel
- Simple Scheduler
- Dedicated Resources
- Microservice-oriented Networking

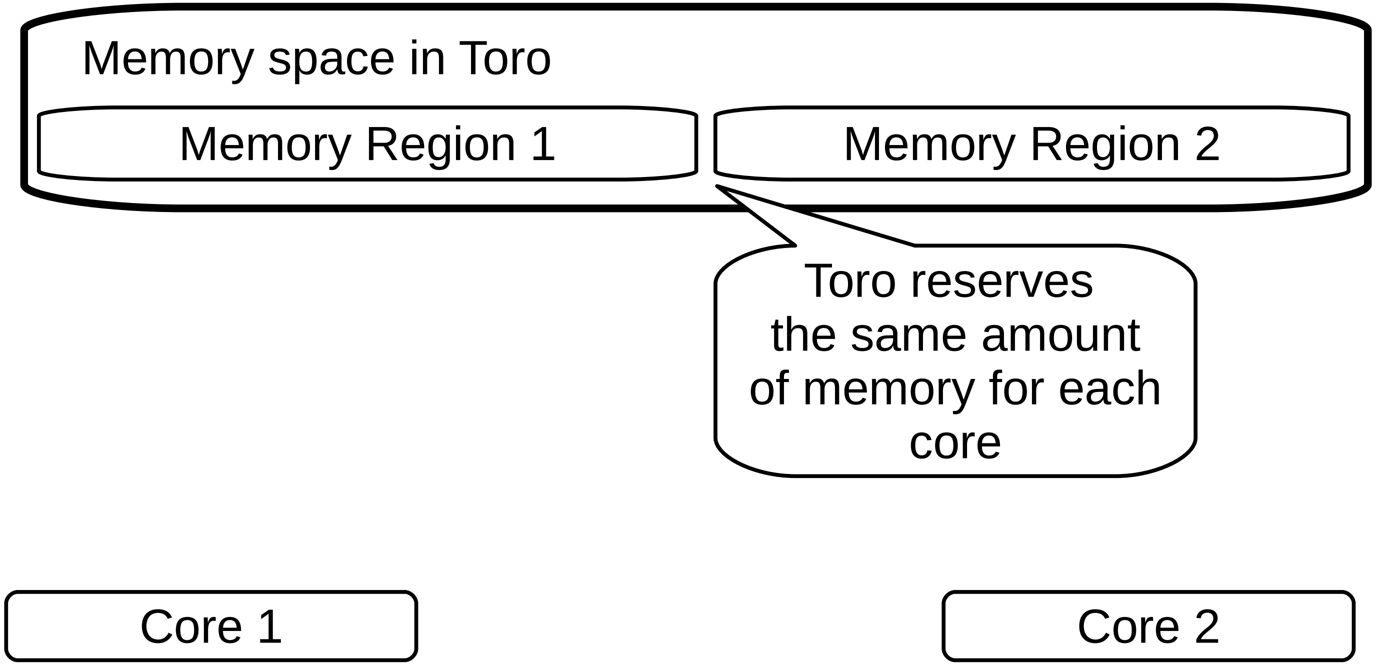
Dedicated Resources

- 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 when spin-locks are used

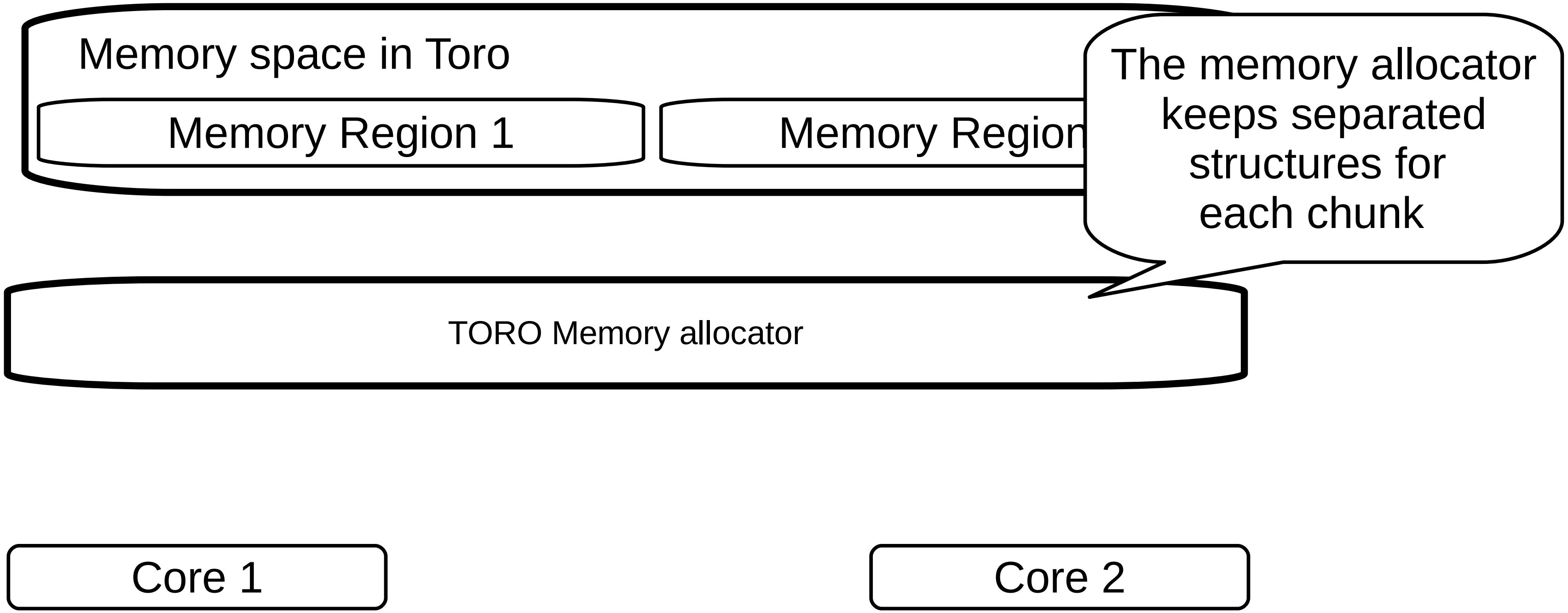
Dedicated Resources Proposal

- 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
  - leveraging NUMA technologies, e.g., Hypertransport, Intel QuickPath

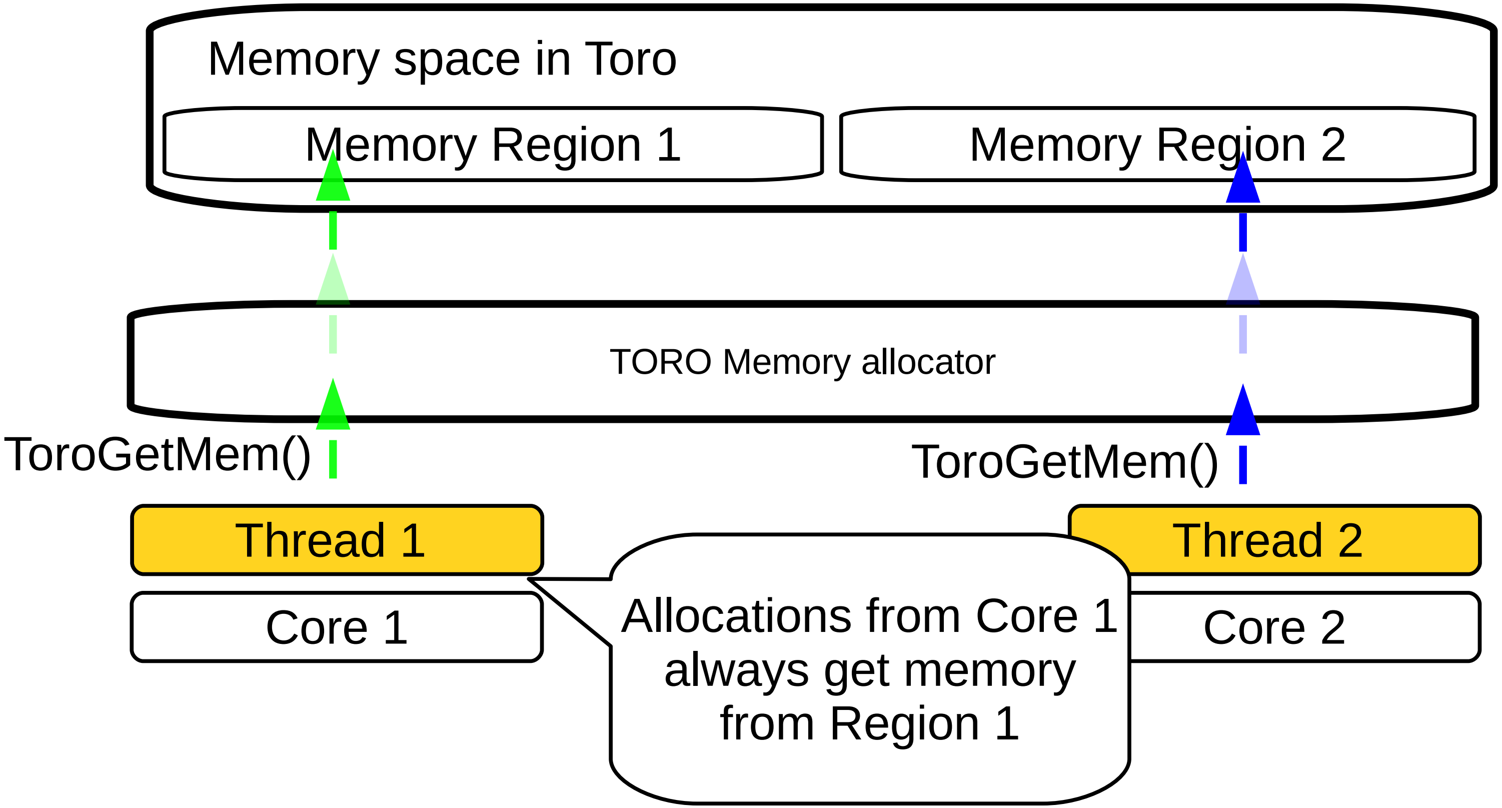
Dedicated Resources Memory



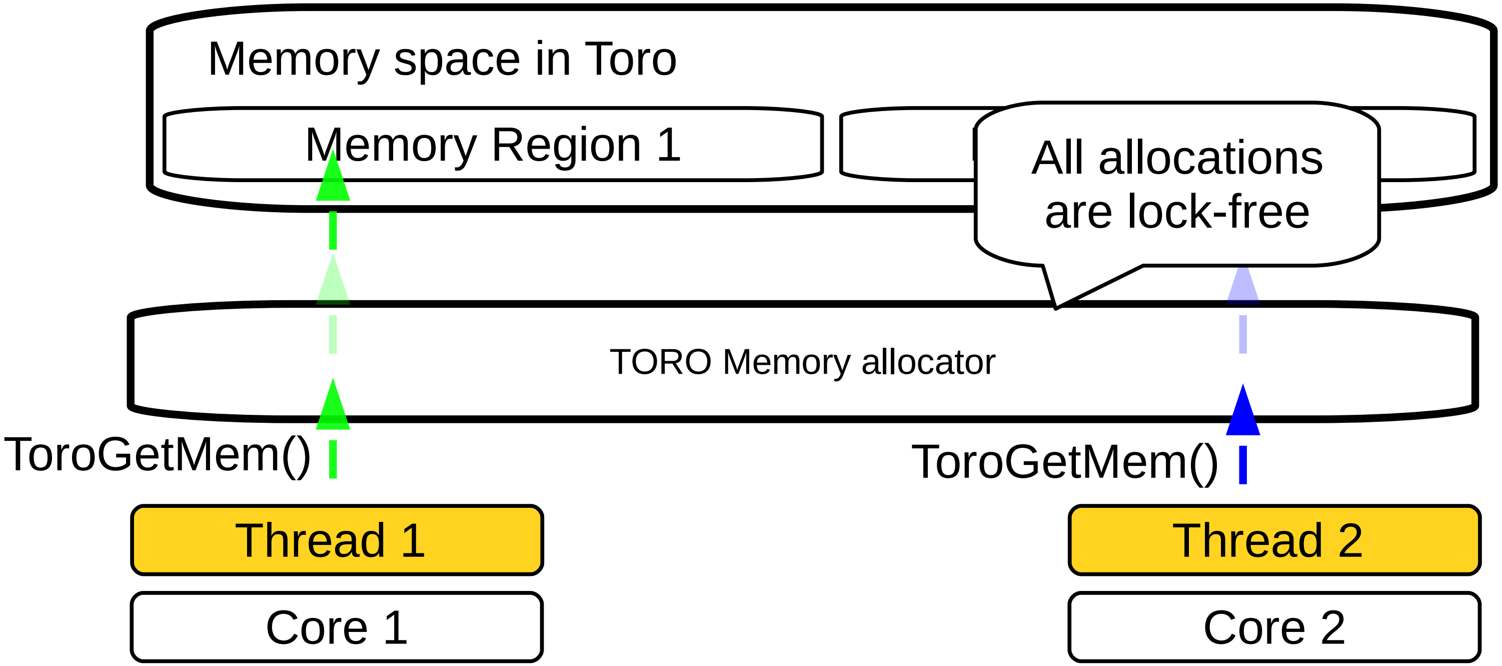
Dedicated Resources
Memory



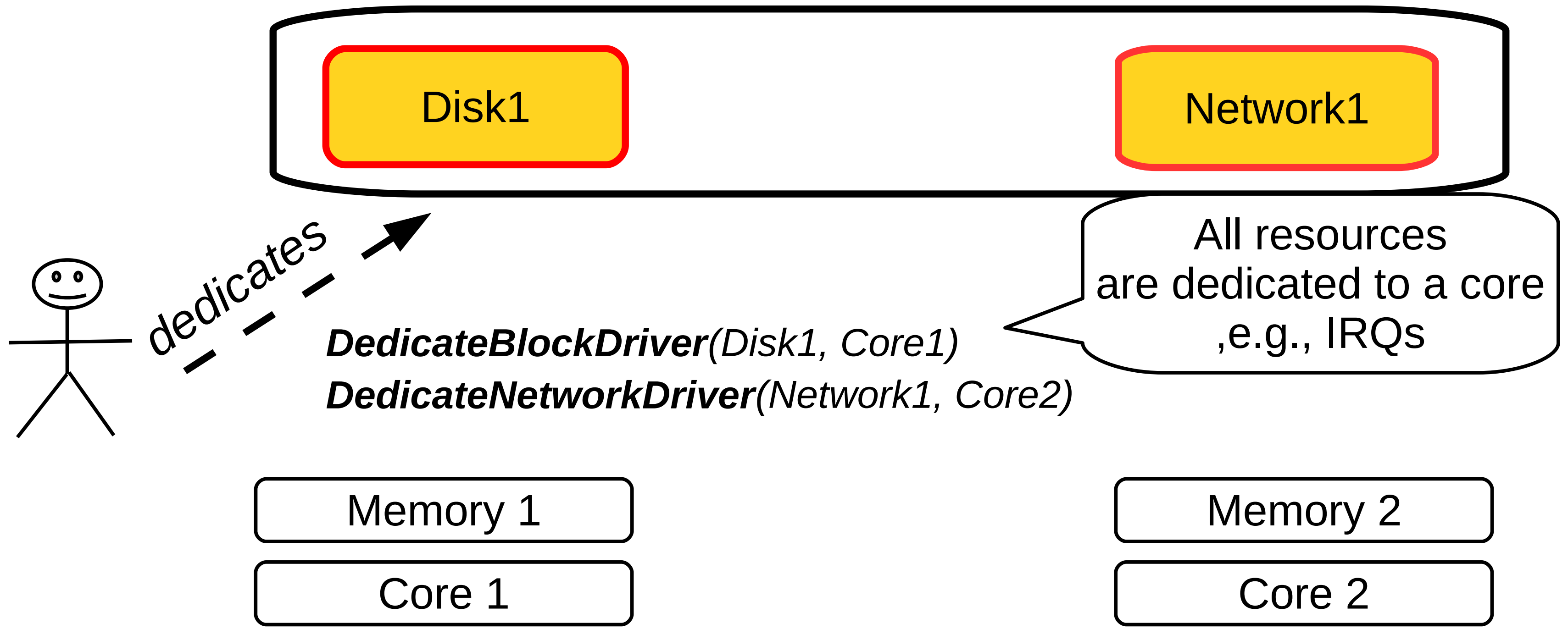
Dedicated Resources Memory



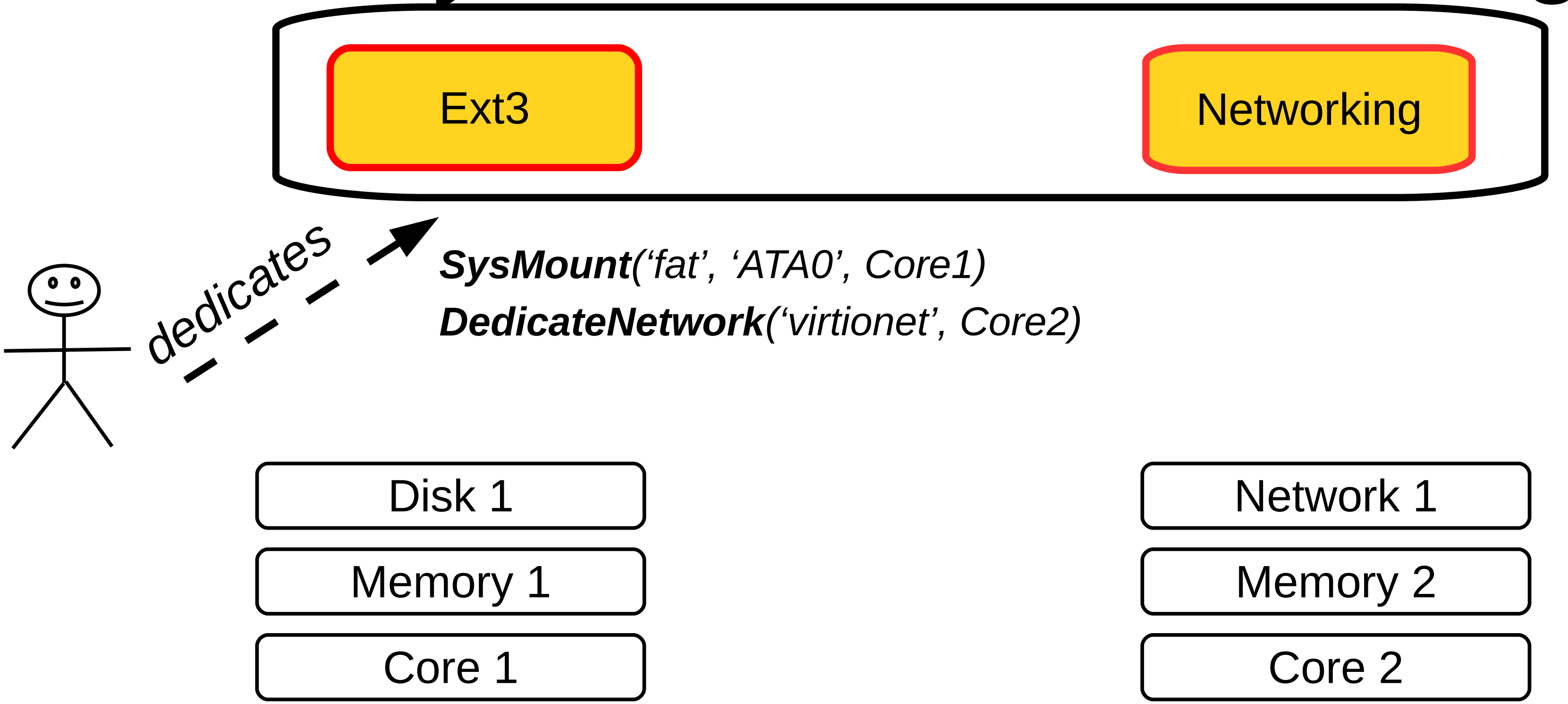
Dedicated Resources
Memory



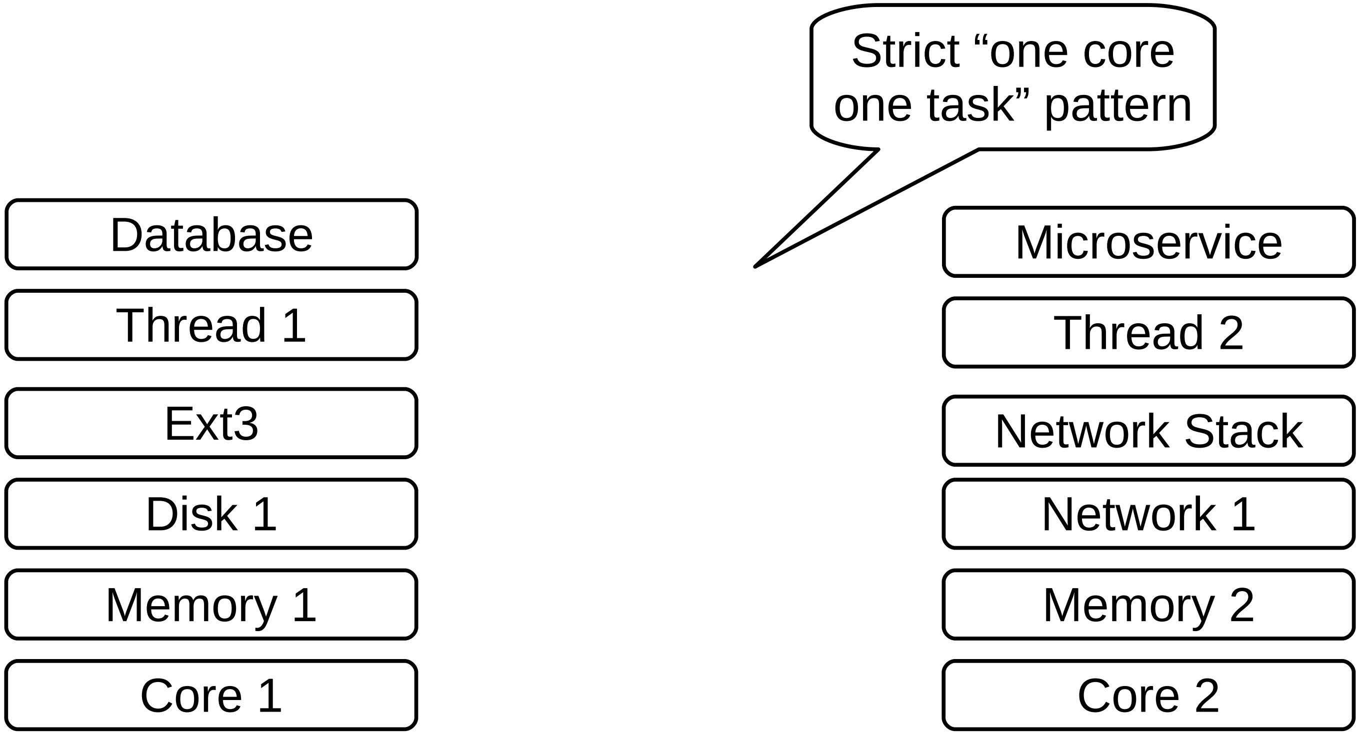
Dedicated Resources
Devices



Dedicated Resources
Filesystems and Networking



Dedicated Resources
Filesystems and Networking



Ingredients for ToroKernel

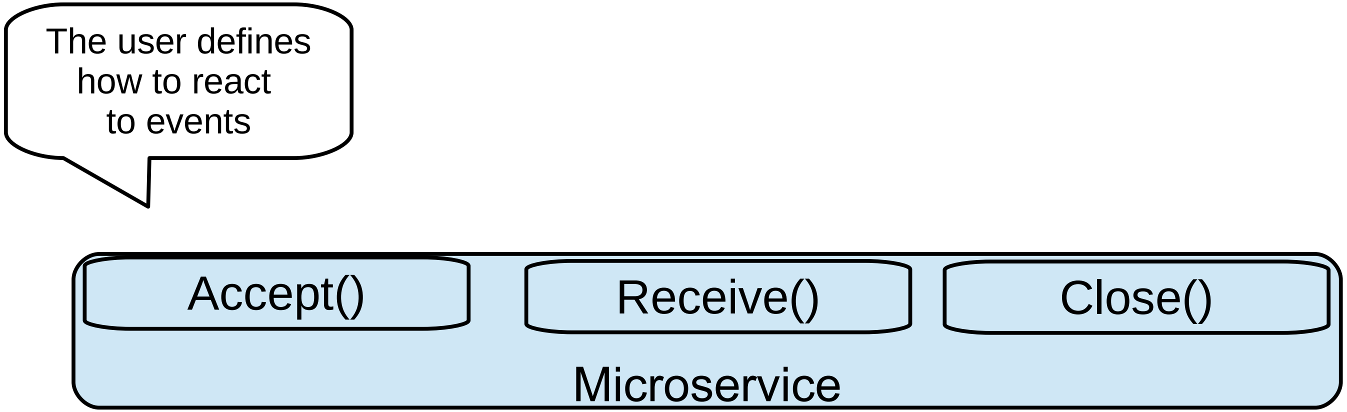
- Application-oriented Kernel
- Simple Scheduler
- Dedicated of Resources
- Microservice-oriented Networking

Microservice-oriented Networking

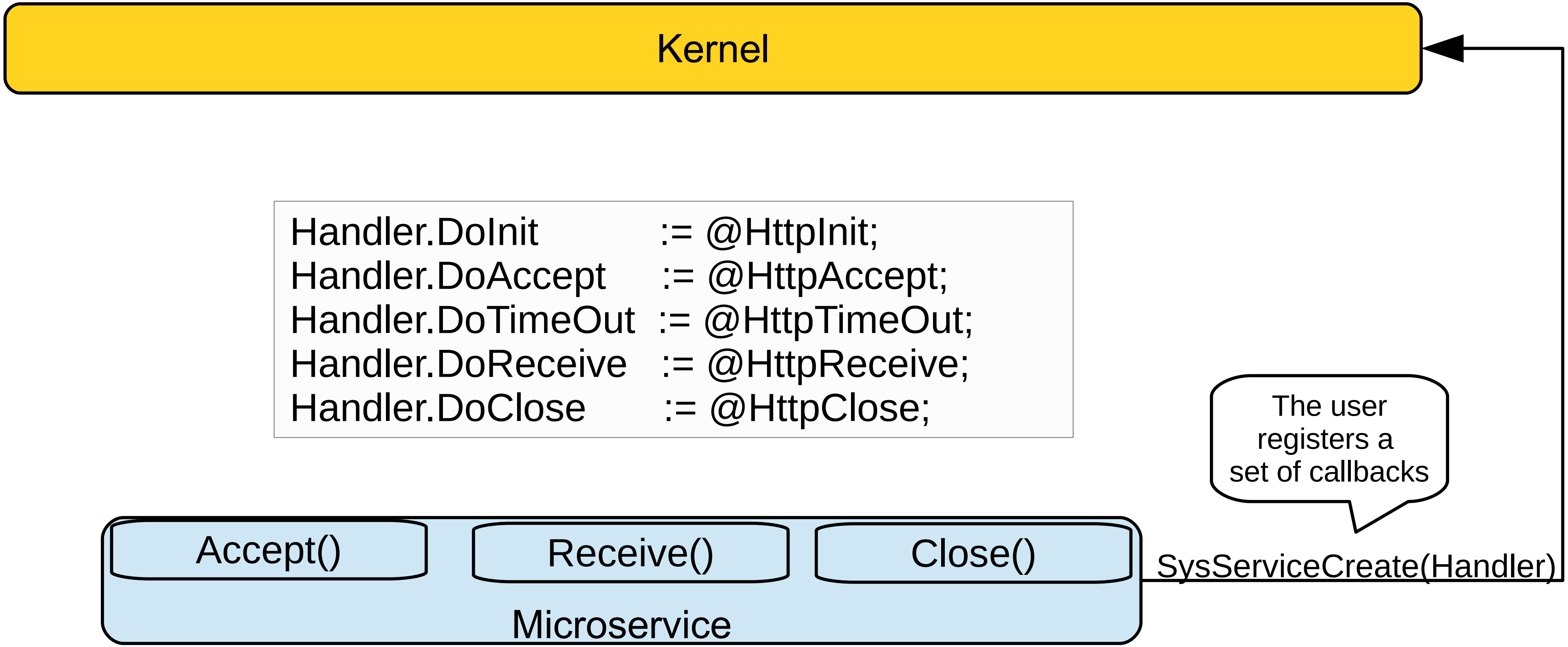
- The kernel provides a dedicated API to create microservices
- Toro proposes two ways to develop microservices:
  - Single thread event loop model[2] for non-blocking microservices
  - Classical Berkeley sockets for those microservices that do IO
- Event loop has been optimized to reduce CPU's usage[3]

[2] Node JS Architecture – Single Threaded Event Loop
[3] Reducing CPU usage of a Toro Appliance, FOSDEM'18

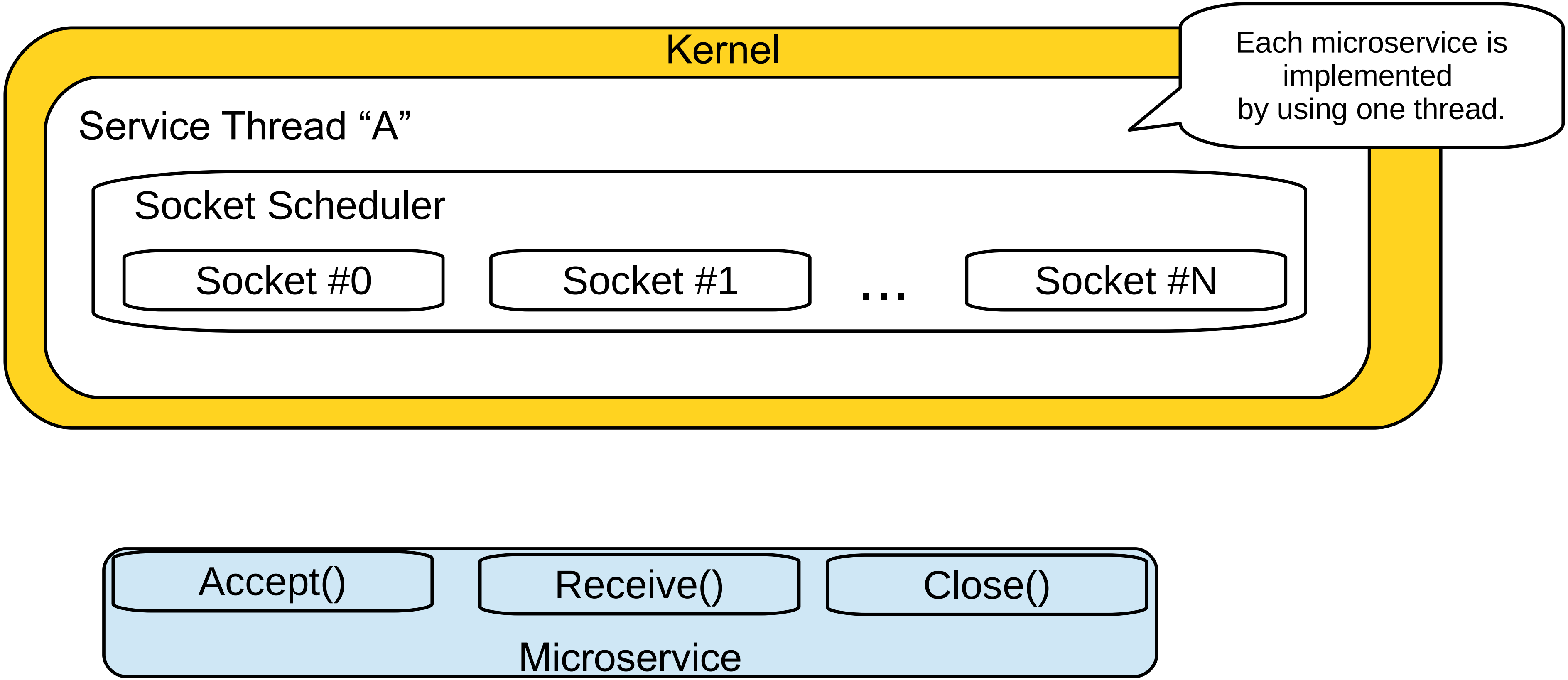
Non-Blocking Microservices



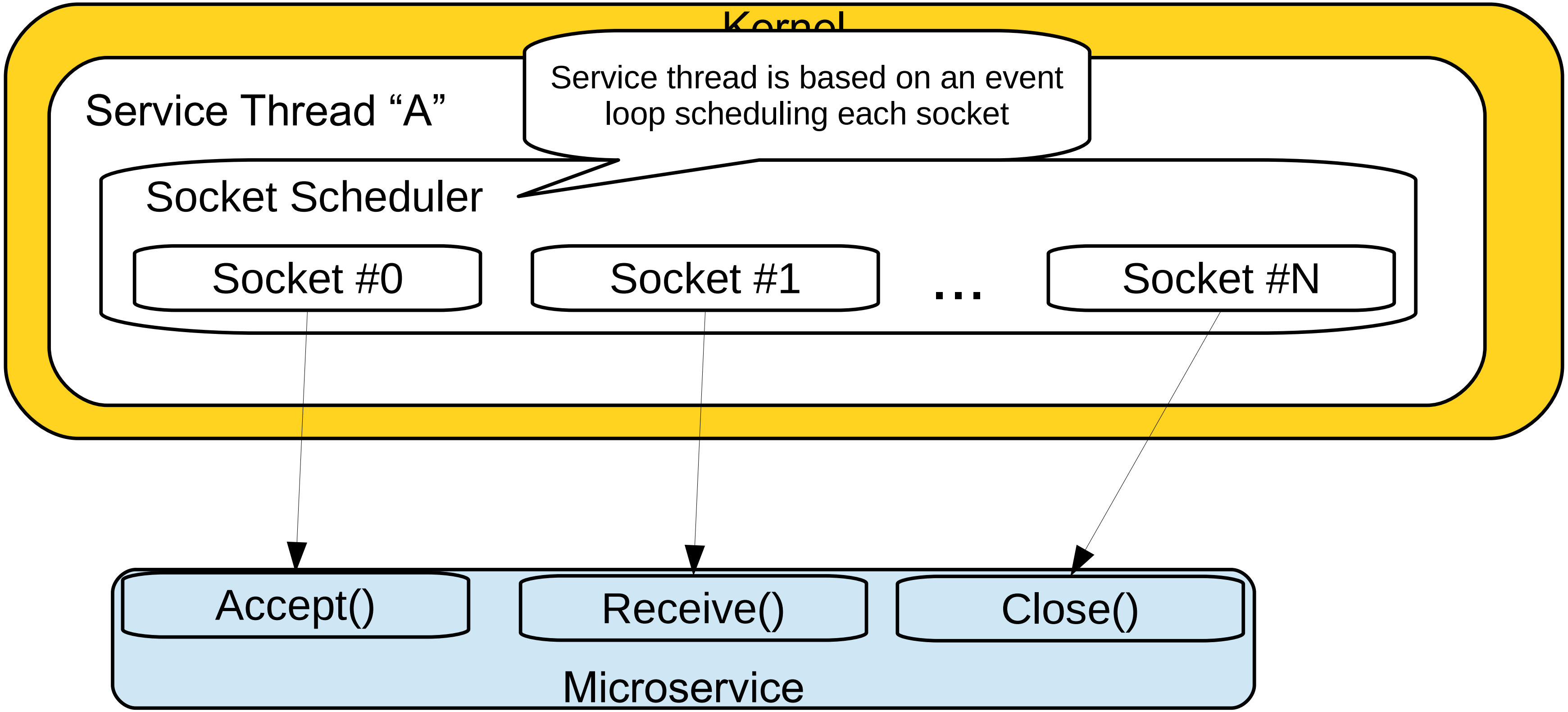
Non-Blocking Microservices



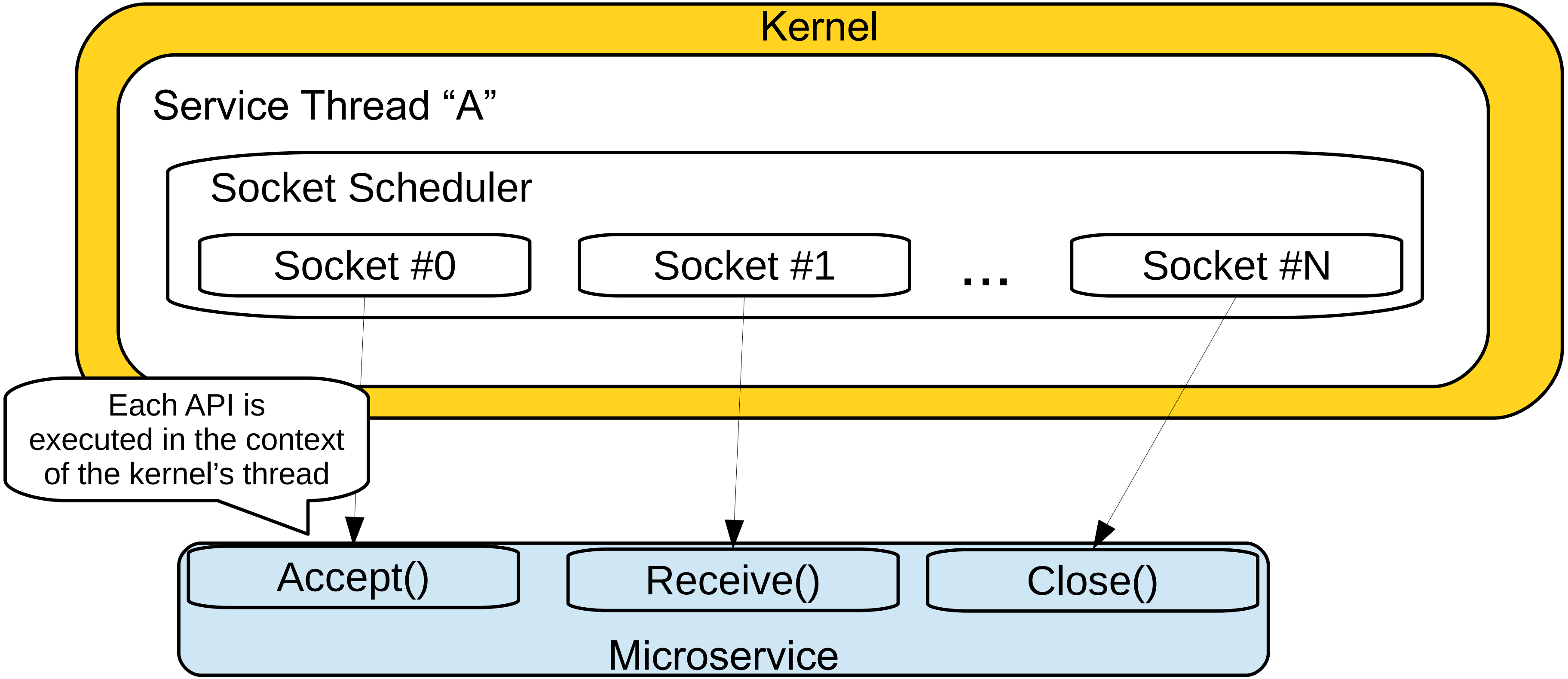
Non-Blocking Microservices



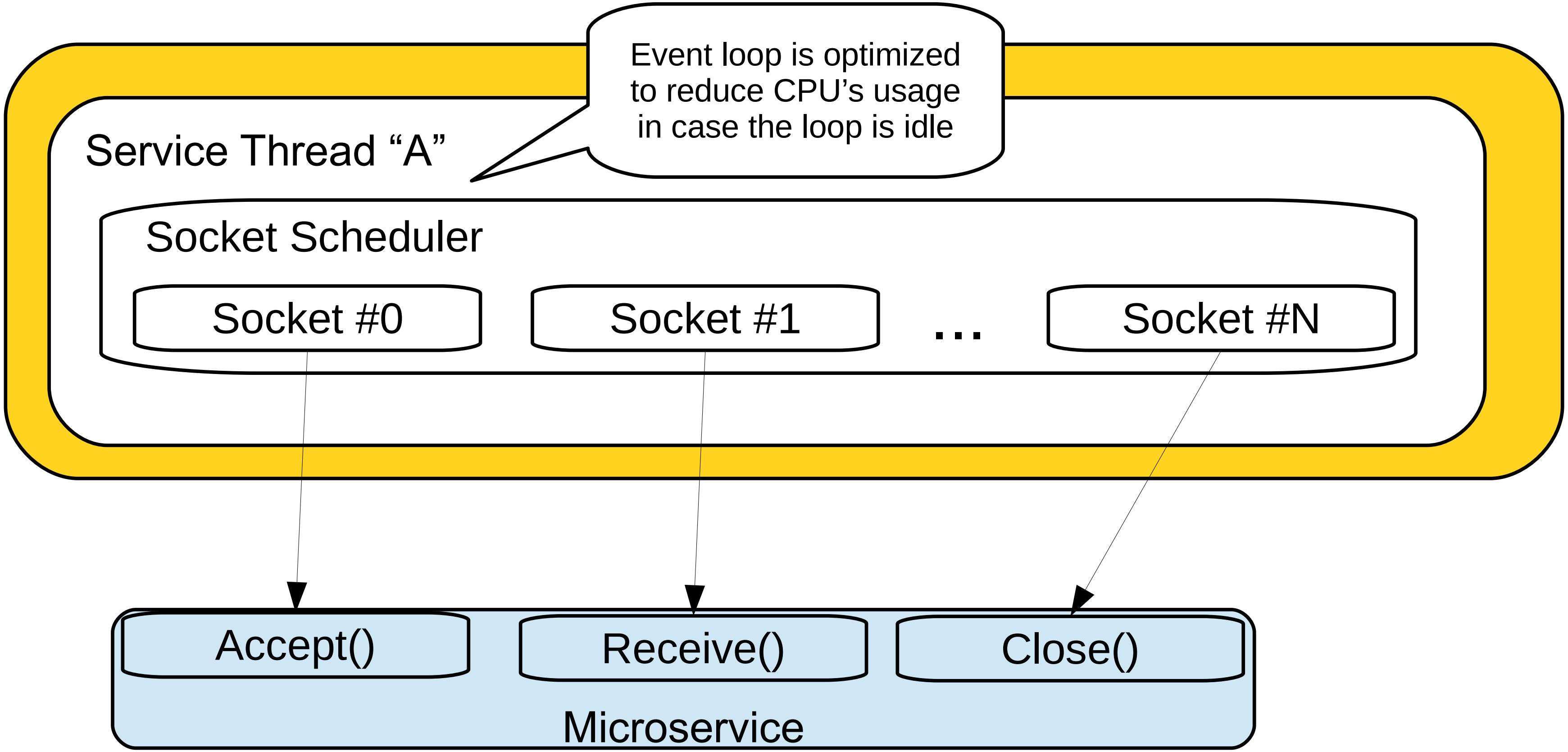
Non-Blocking Microservices



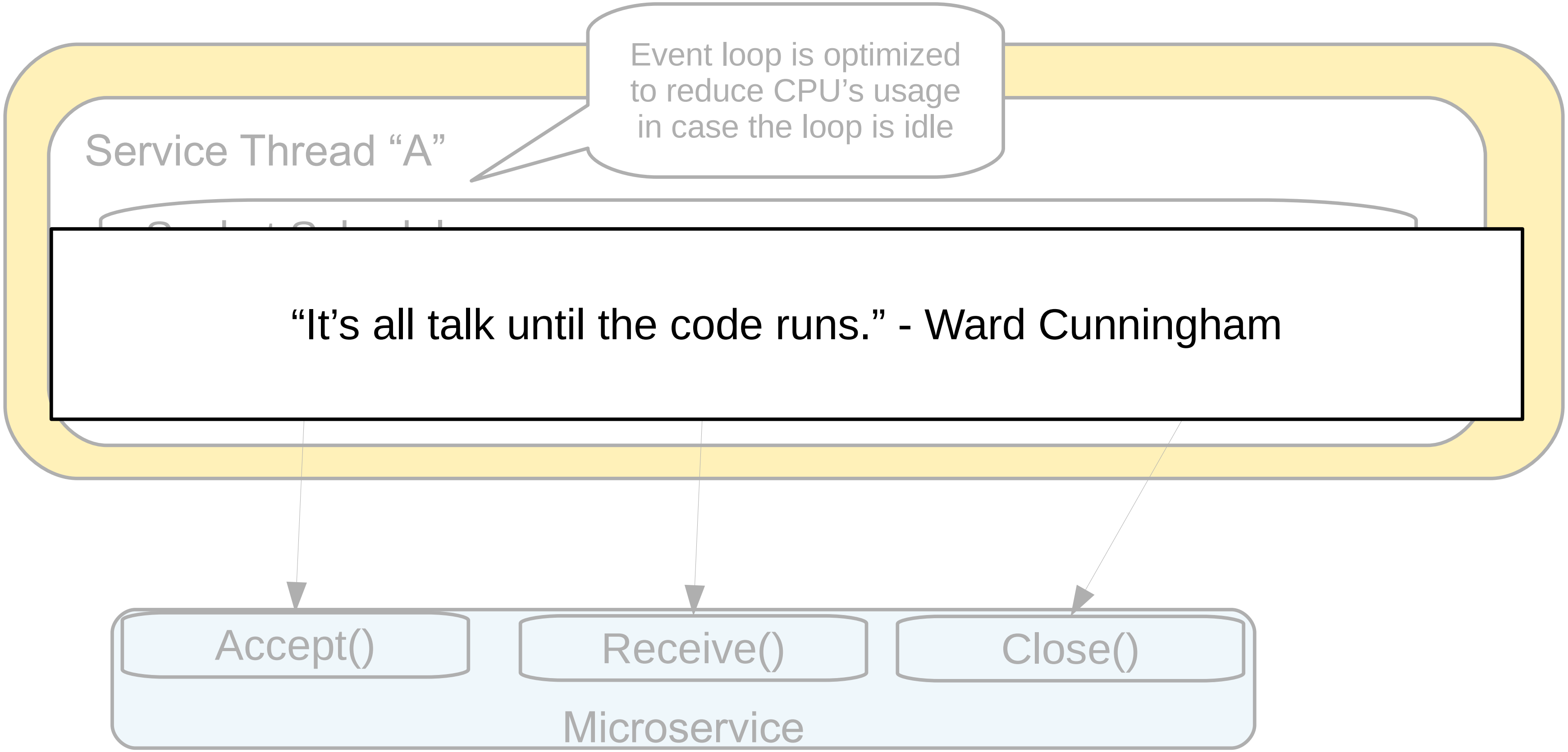
Non-Blocking Microservices



Non-Blocking Microservices



Non-Blocking Microservices



"It's all talk until the code runs." - Ward Cunningham

Evaluation

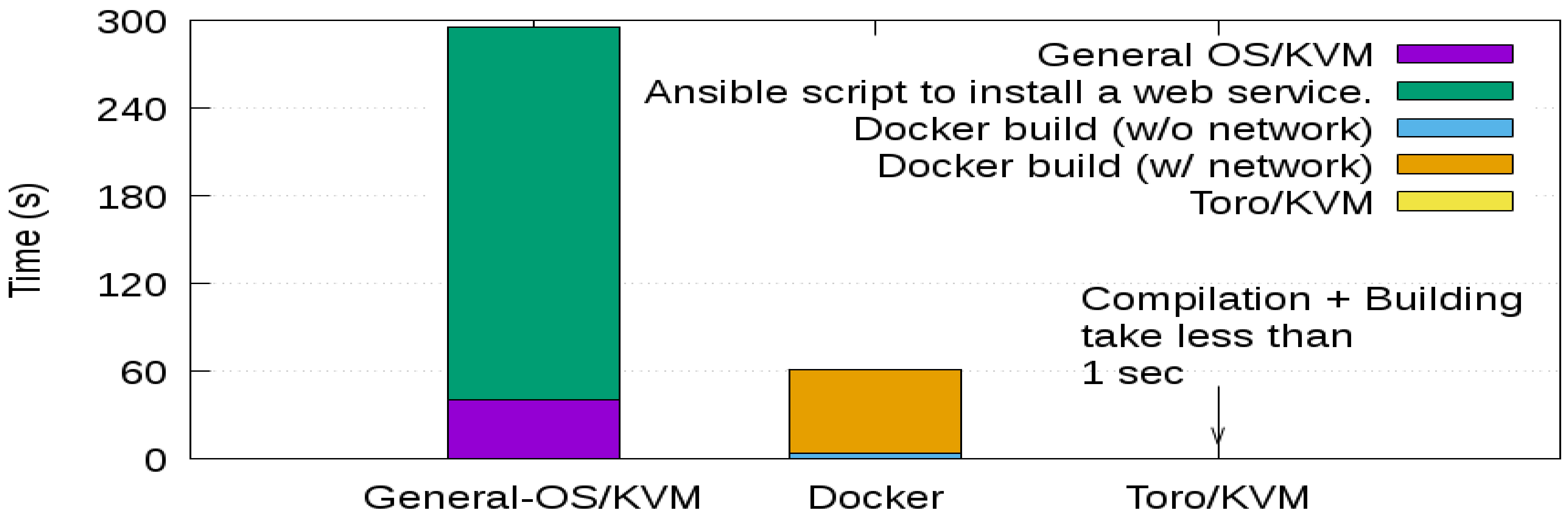
- We implement a “Hello World” microservice[4] (see https://github.com/MatiasVara/torokernel/tree/master/examples/HelloWorldMicroservice)
- We deploy it in Docker, Ubuntu (KVM Guest) and Toro (KVM Guest)
- We chose the following technology:
  - NGINX, UWSGI (4 processes) and Flask
- Memory is limited in all the platforms
- We compare these approaches in term of:
  - Deploying Time
  - Bootstrap Time
  - Image Size
  - CPU usage
  - Time per Request

Microservice’s footprint

[4] “Toro, a Dedicated Kernel for Microservices”, OSSummit’18

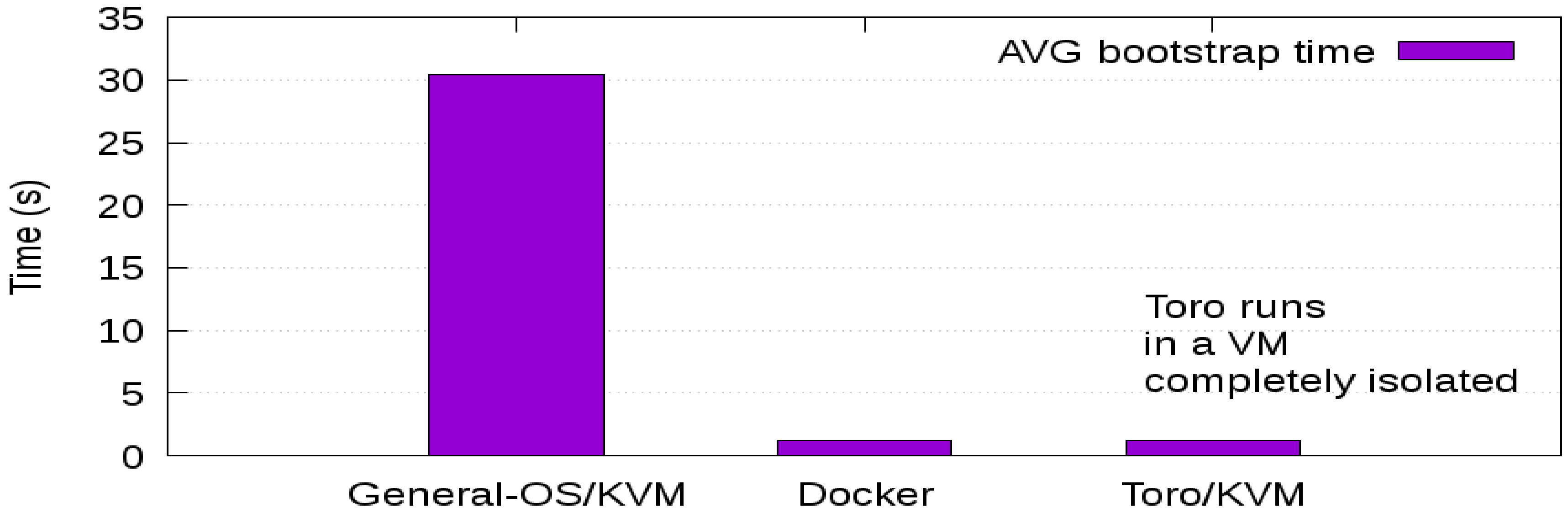
Deploying Time

- Time required to build an image within the microservice



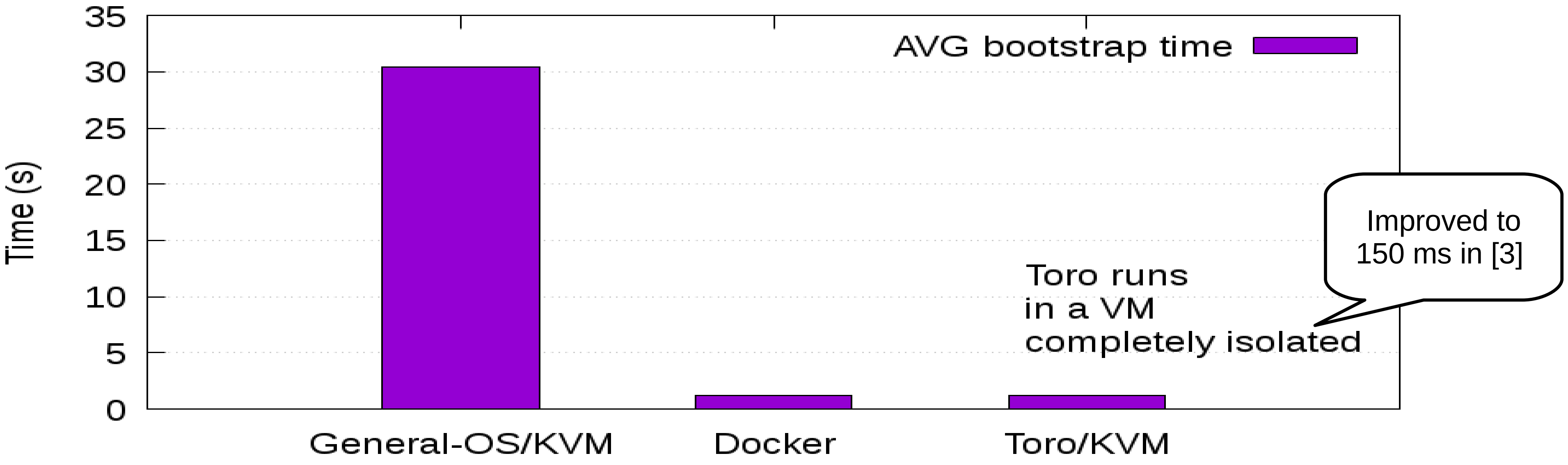
Bootstrap Time

- Time to boot and to serve the first request



Bootstrap Time

- Time to boot and to serve the first request



[3] Speed up Booting time of a Toro Appliance, FOSDEM’19

Image Size

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

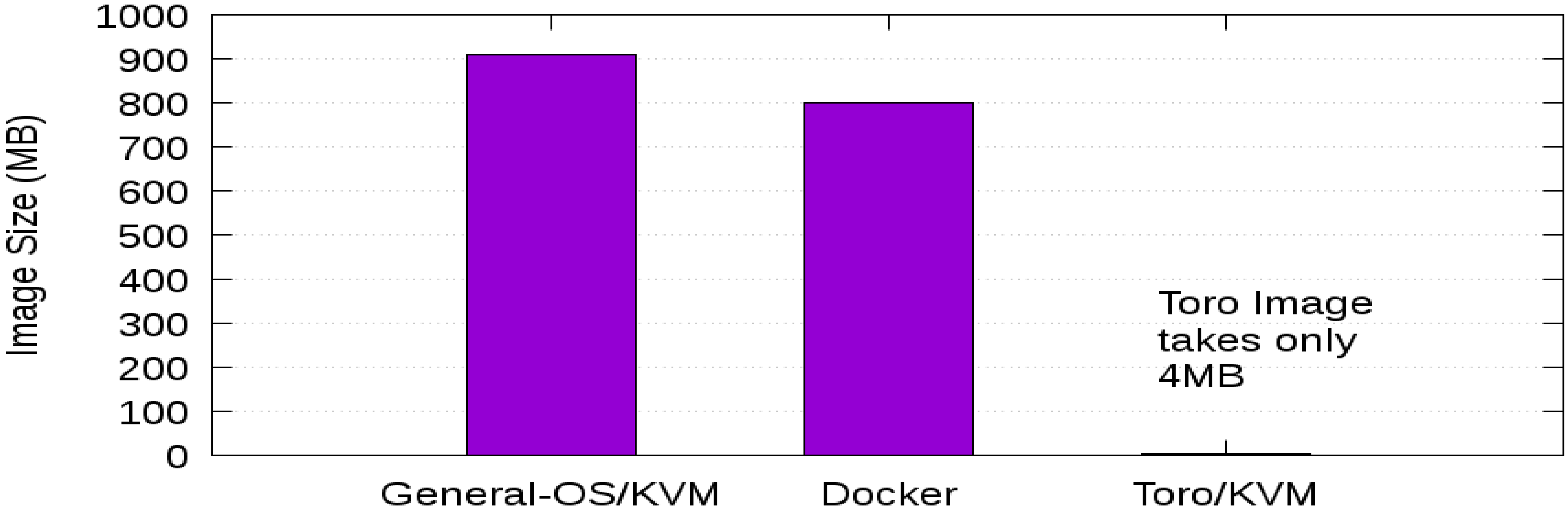
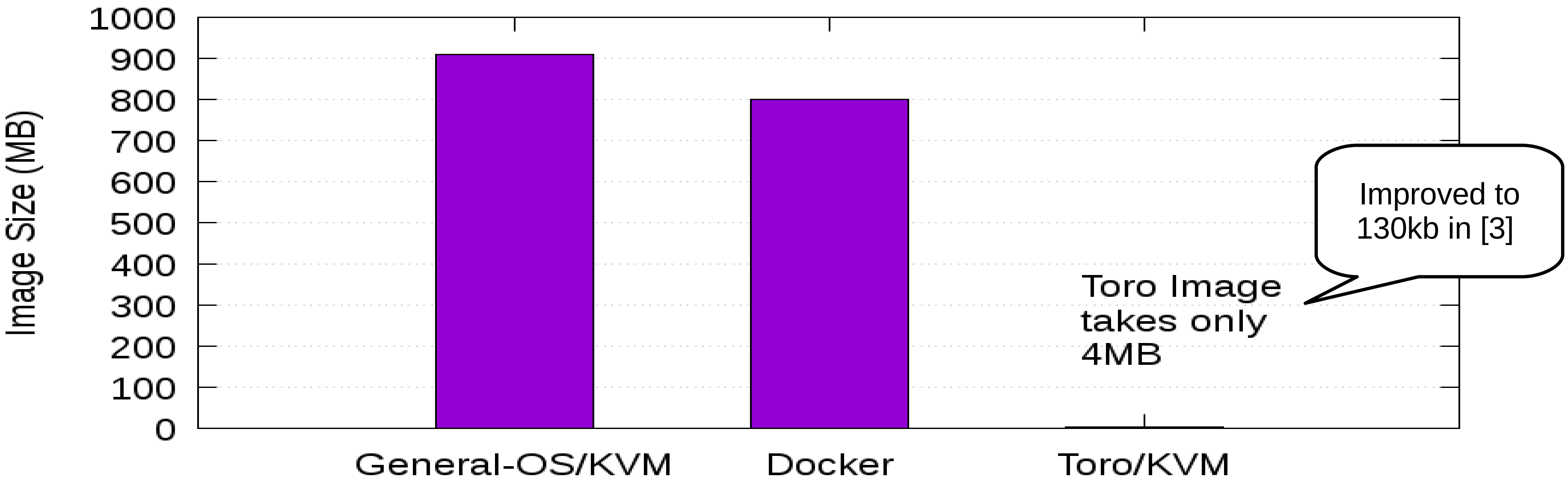
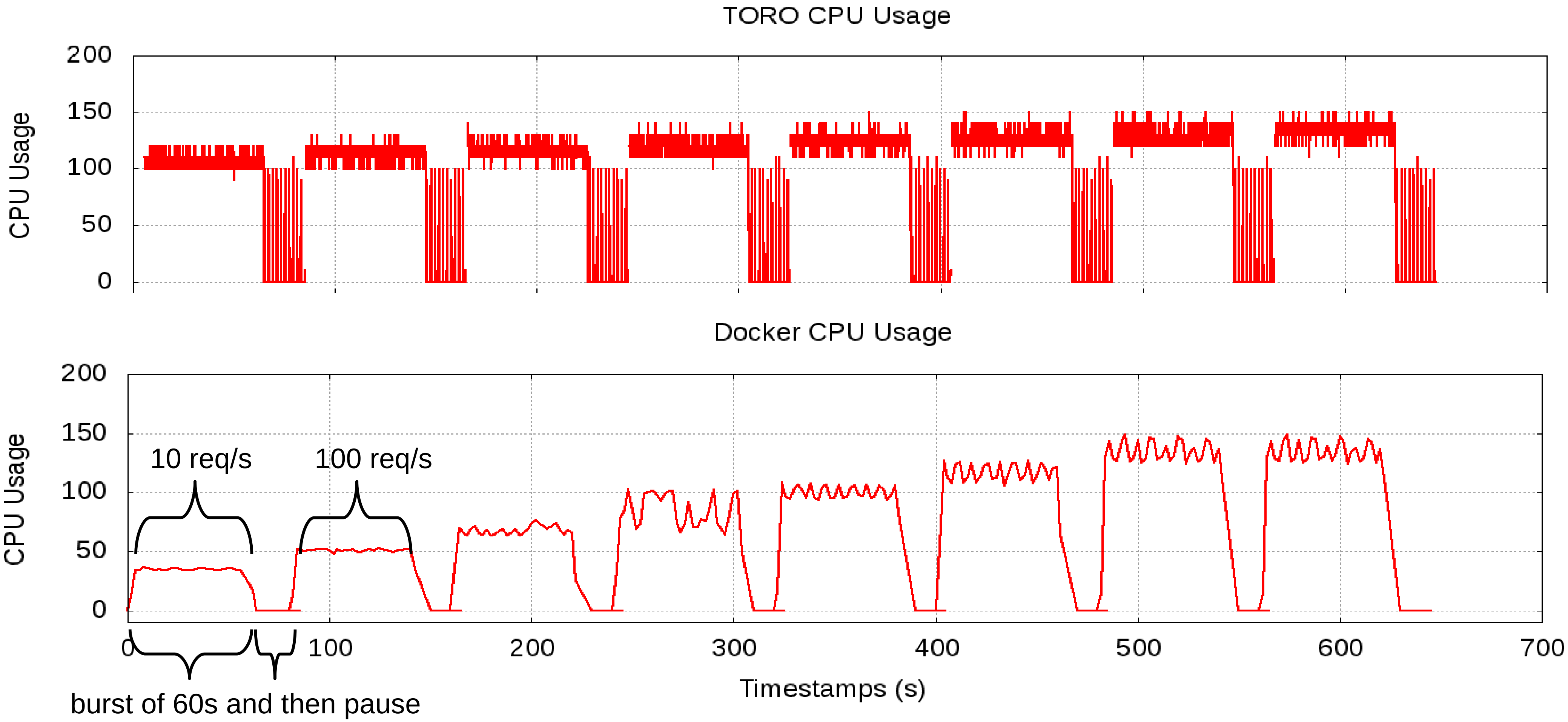


Image Size

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



CPU Usage


TORO CPU Usage
CPU Usage
200
150
100
50
0
Docker CPU Usage
CPU Usage
200
150
100
50
0
10 req/s
100 req/s
0
100
200
300
400
500
600
700
Timestamps (s)
burst of 60s and then pause

End-User Delay

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

		Number of Concurrent Request		
Approach	CPUs	200	500	1000
Docker	4	139.980 ms	333.937 ms	801.422
Ubuntu/KVM	1	94.507 ms	238.149 ms	560.513 ms
TORO/KVM	1	120.065 ms	301.736 ms	596.792 ms

Take away lessons

- Toro image is about 15-times less than NGINX docker image
- 1 sec to re-deploy a microservice
  - Deploy an OS w/similar configuration takes 300 sec, with docker ~50 sec
- Comparable time per request with cutting edge technology (NGINX)
- CPU Usage comparable with Docker
- Safer than Docker

Demo time

Summary

- Toro design is improved in five main points:
  - Booting time and building time
  - Kernel API with zero overhead function call
  - Access to shared memory
  - Networking
  - CPU usage

Future Work

- Improve tooling to build, test and debug microservices
- Investigate new use-cases
- Port networking applications to provide starter kits: SMTP relay, HTTP proxy, Web tracking
- Investigate new ideas to improve the network stack for microservices, e.g., improve socket scheduling for http, resource allocation algorithm
- Investigate the use of microVM technologies, e.g., Firecraker, NEMU.
- Investigate the use of OpenStack to ease the deployment of Toro appliances

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

QA

- http://www.torokernel.io
- torokernel@gmail.com
- T... torokernel
- ...ki at github
- ...e examples with Toro
- ...5 minutes (or less...)
- ...nel-docker-qemu-webservices at Github

That's all folks!