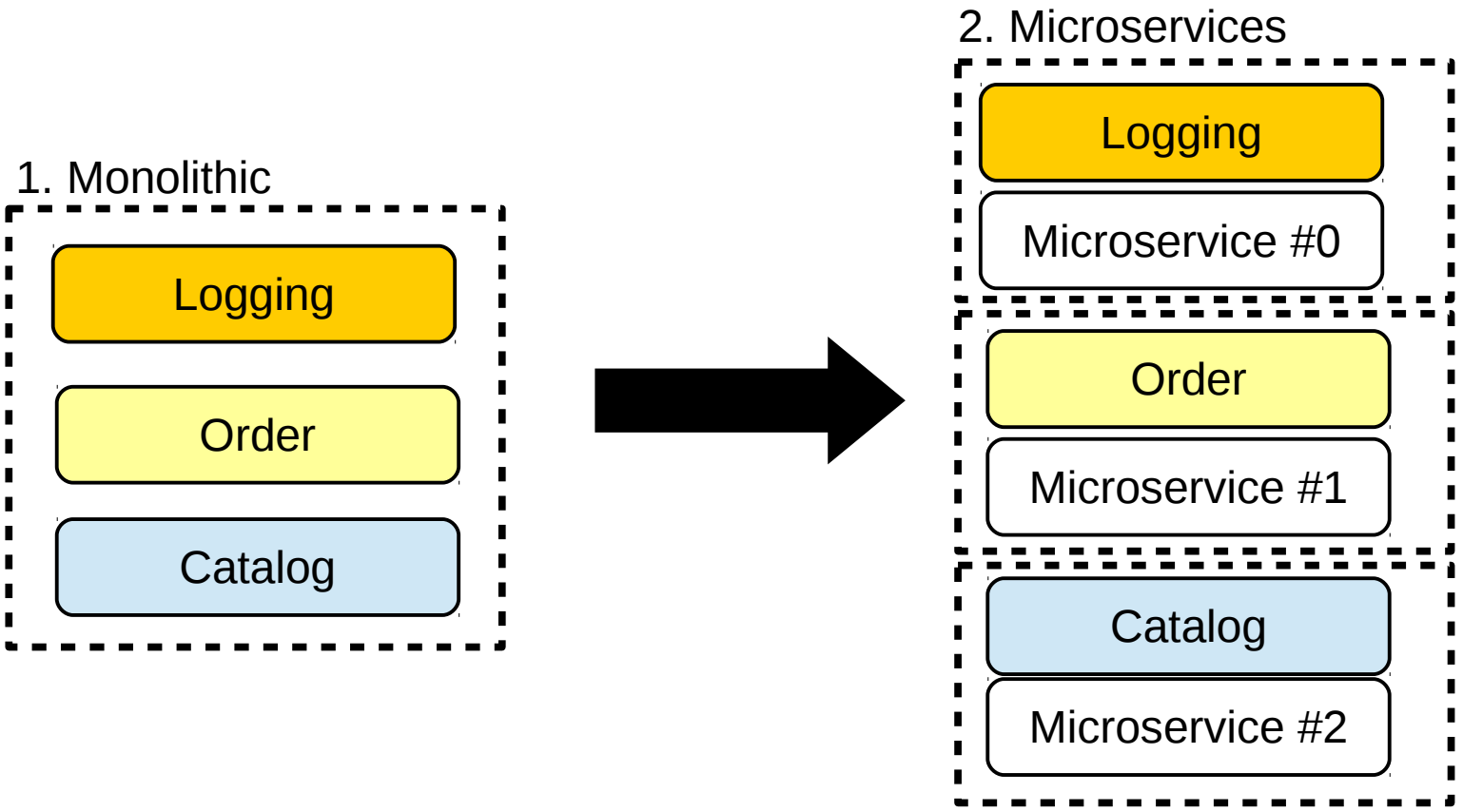
Toro, a Dedicated Kernel for Microservices

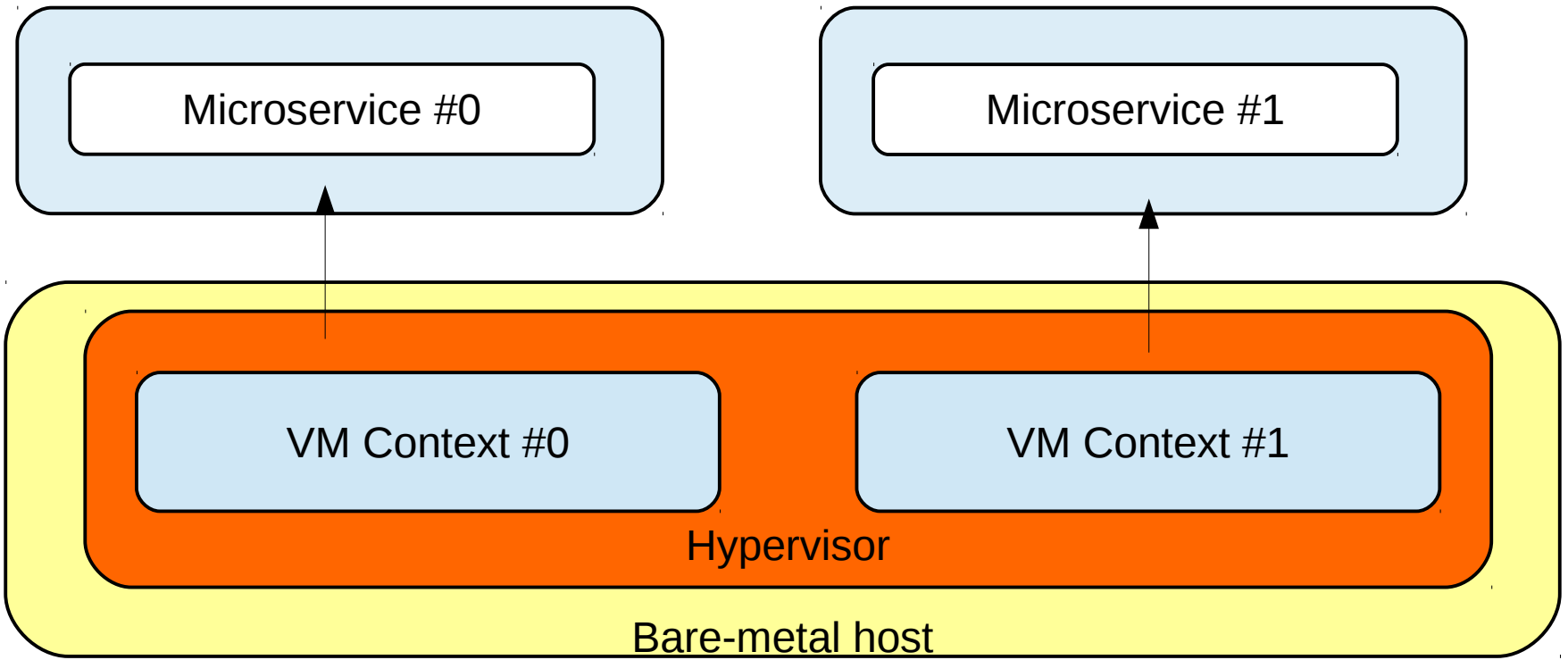
Matias E. Vara Larsen
Silicon-Gears

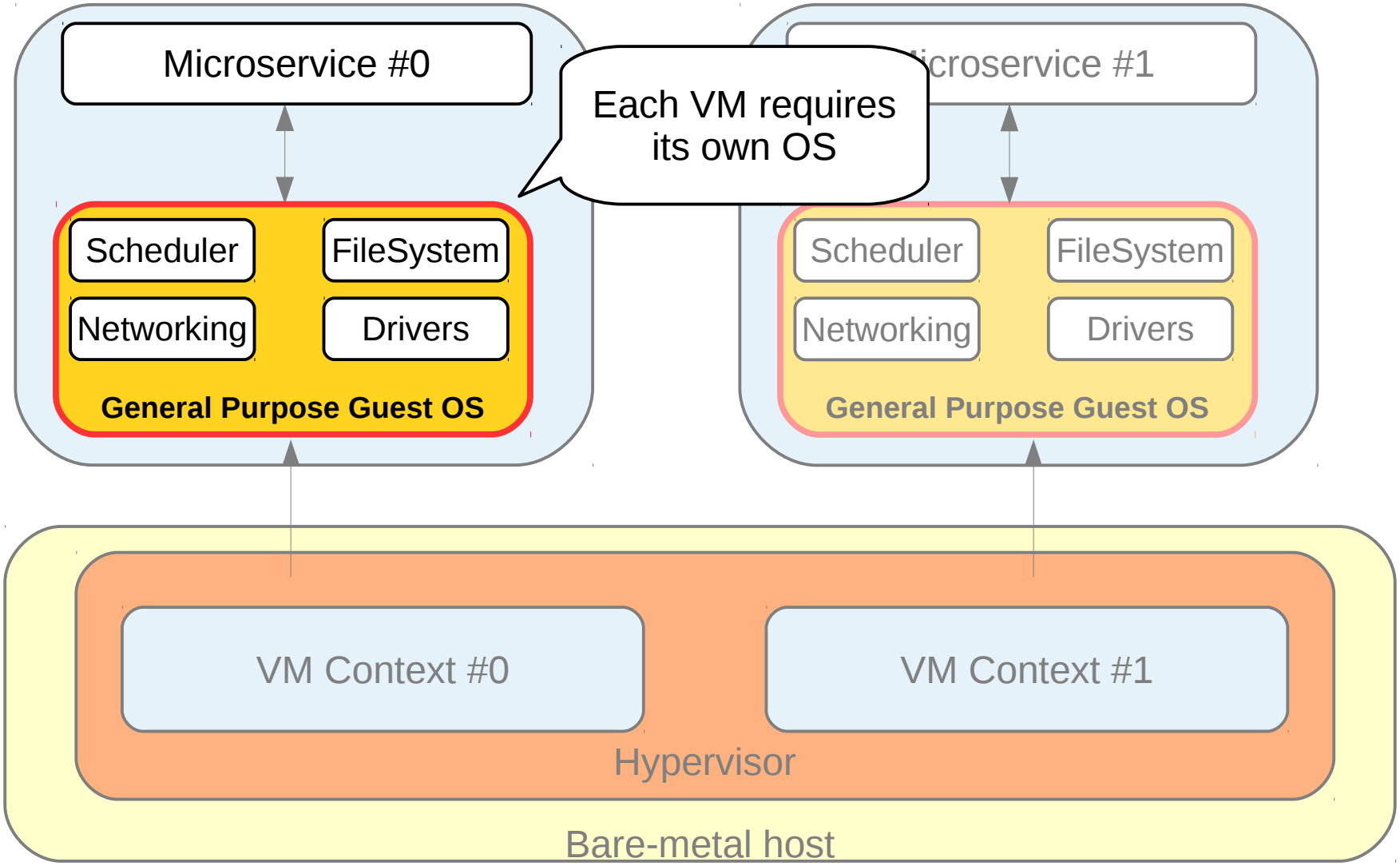
Cesar Bernardini
Barracuda Networks

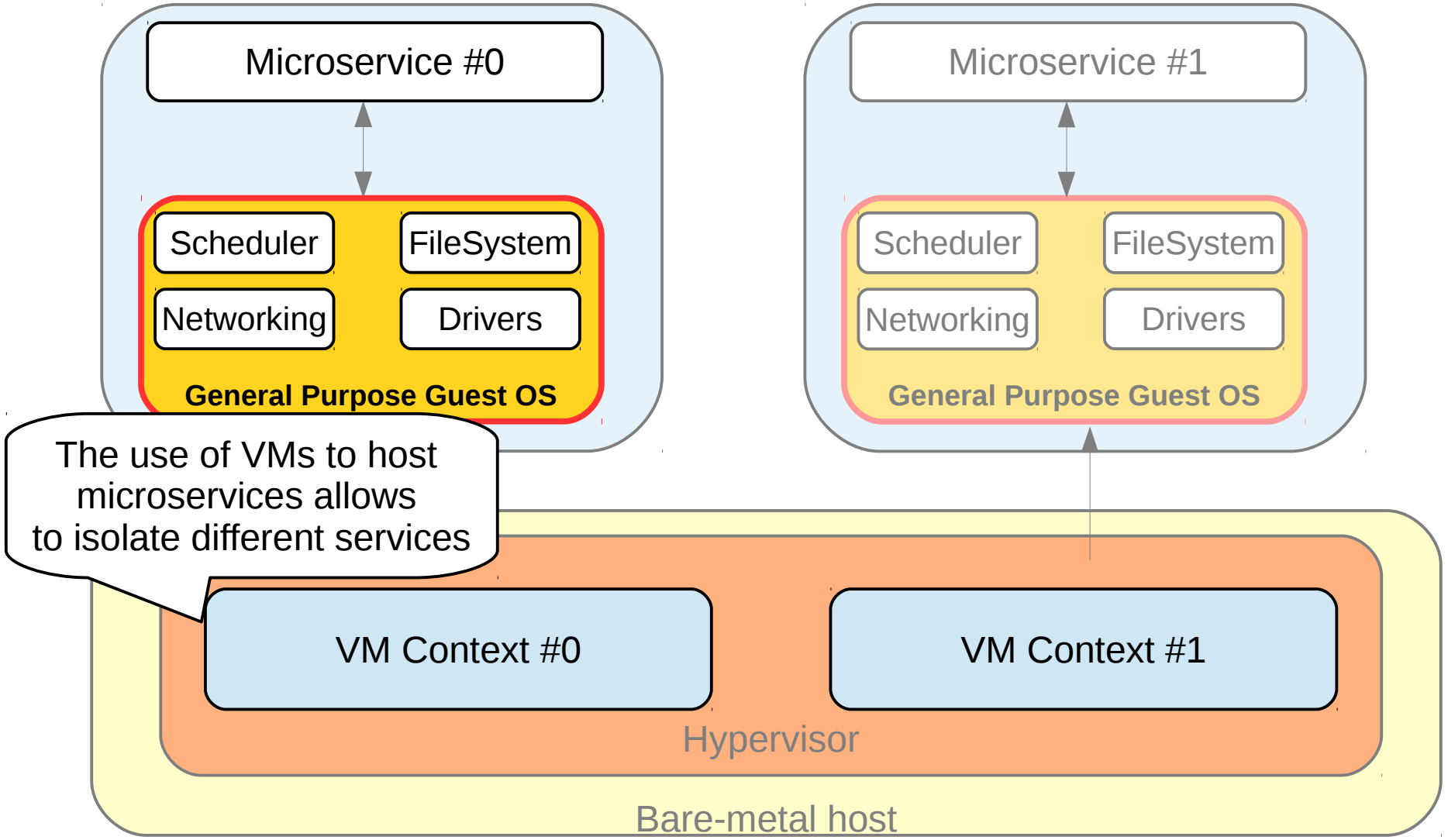
What is a microservice?

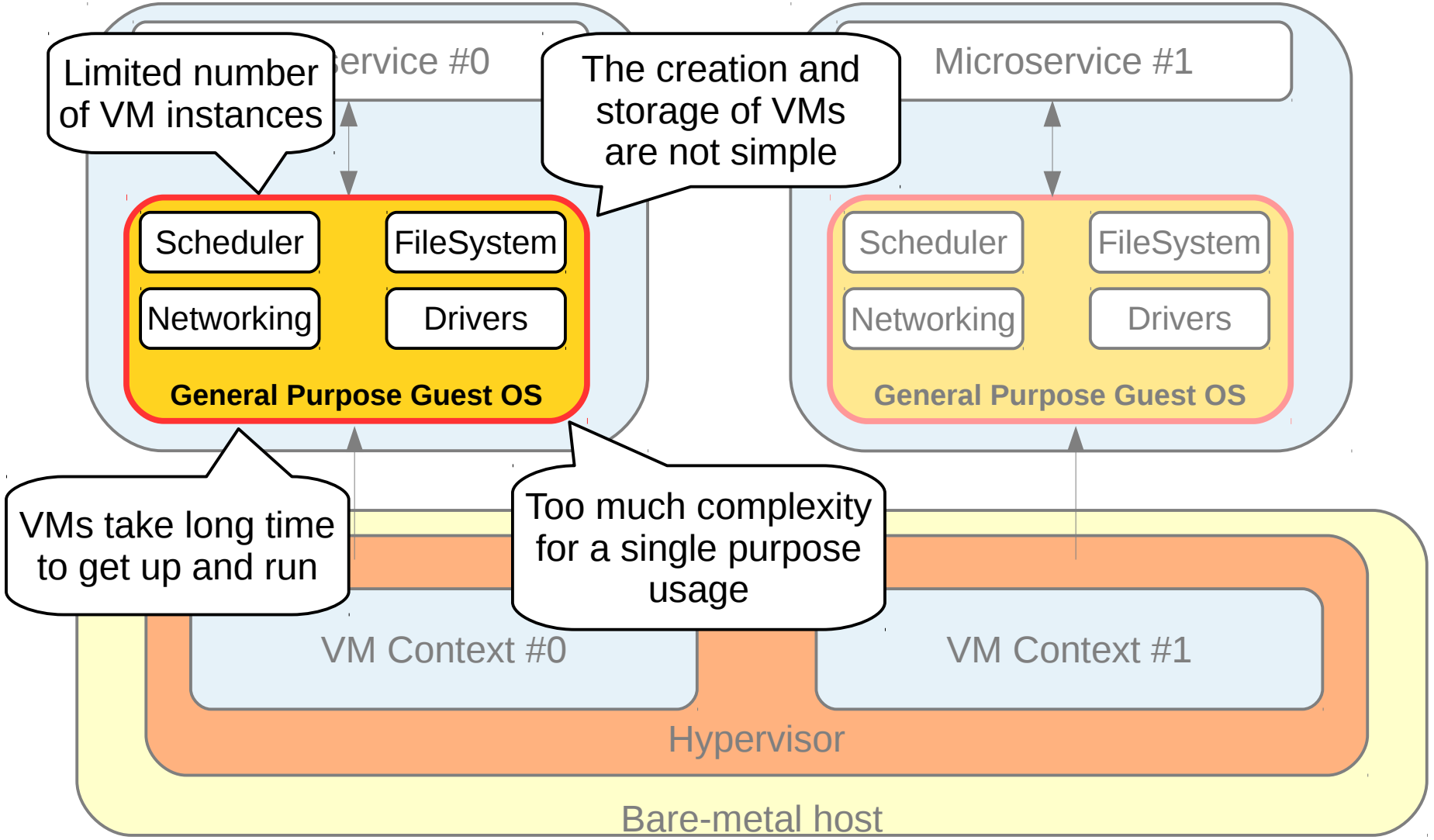


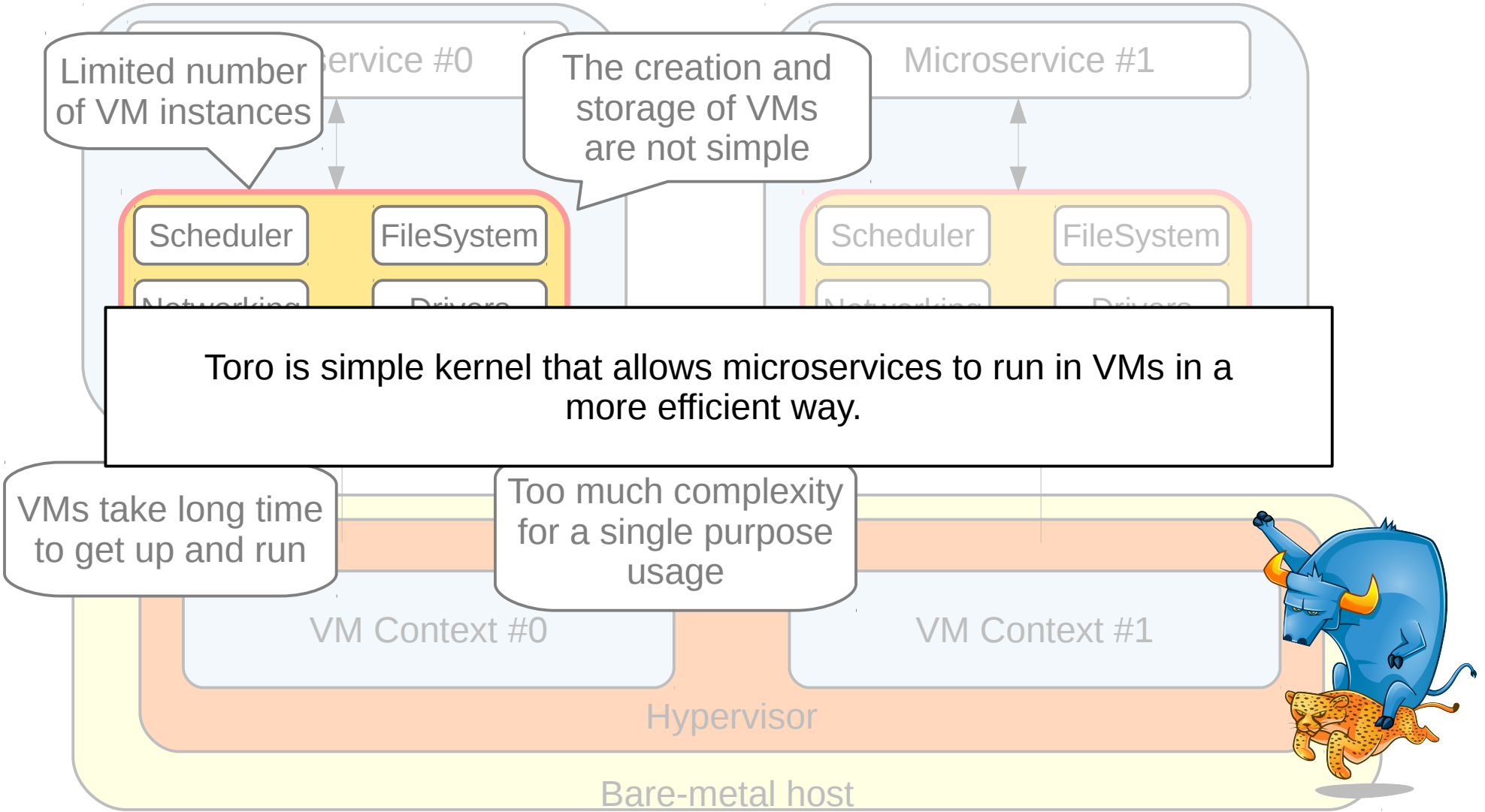
Service Instance per Virtual Machine




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


Microservice #0
Microservice #1
Scheduler
FileSystem
Networking
Drivers
General Purpose Guest OS
Scheduler
FileSystem
Networking
Drivers
General Purpose Guest OS
The use of VMs to host microservices allows to isolate different services
VM Context #0
VM Context #1
Hypervisor
Bare-metal host


Limited number of VM instances
The creation and storage of VMs are not simple
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
Too much complexity for a single purpose usage
VM Context #0
VM Context #1
Hypervisor
Bare-metal host


Limited number of VM instances
The creation and storage of VMs are not simple
Microservice #1
Scheduler
FileSystem
Scheduler
FileSystem
Toro is simple kernel that allows microservices to run in VMs in a more efficient way.
VMs take long time to get up and run
Too much complexity for a single purpose usage
VM Context #0
VM Context #1
Hypervisor
Bare-metal host

Ingredients for ToroKernel

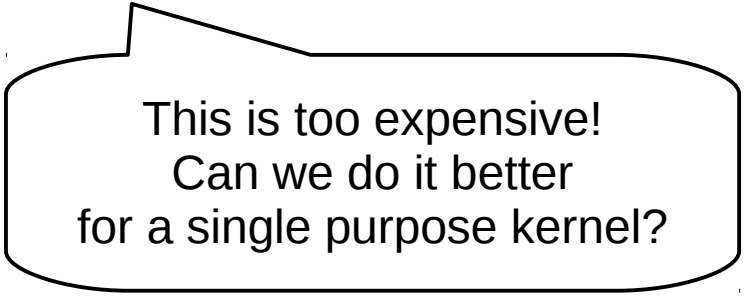
- User application within the kernel
- Cooperative Scheduler
- Dedicated Resources
- Single Thread Event Loop Networking

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 model 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 model to user mode, e.g., interruption and scheduling



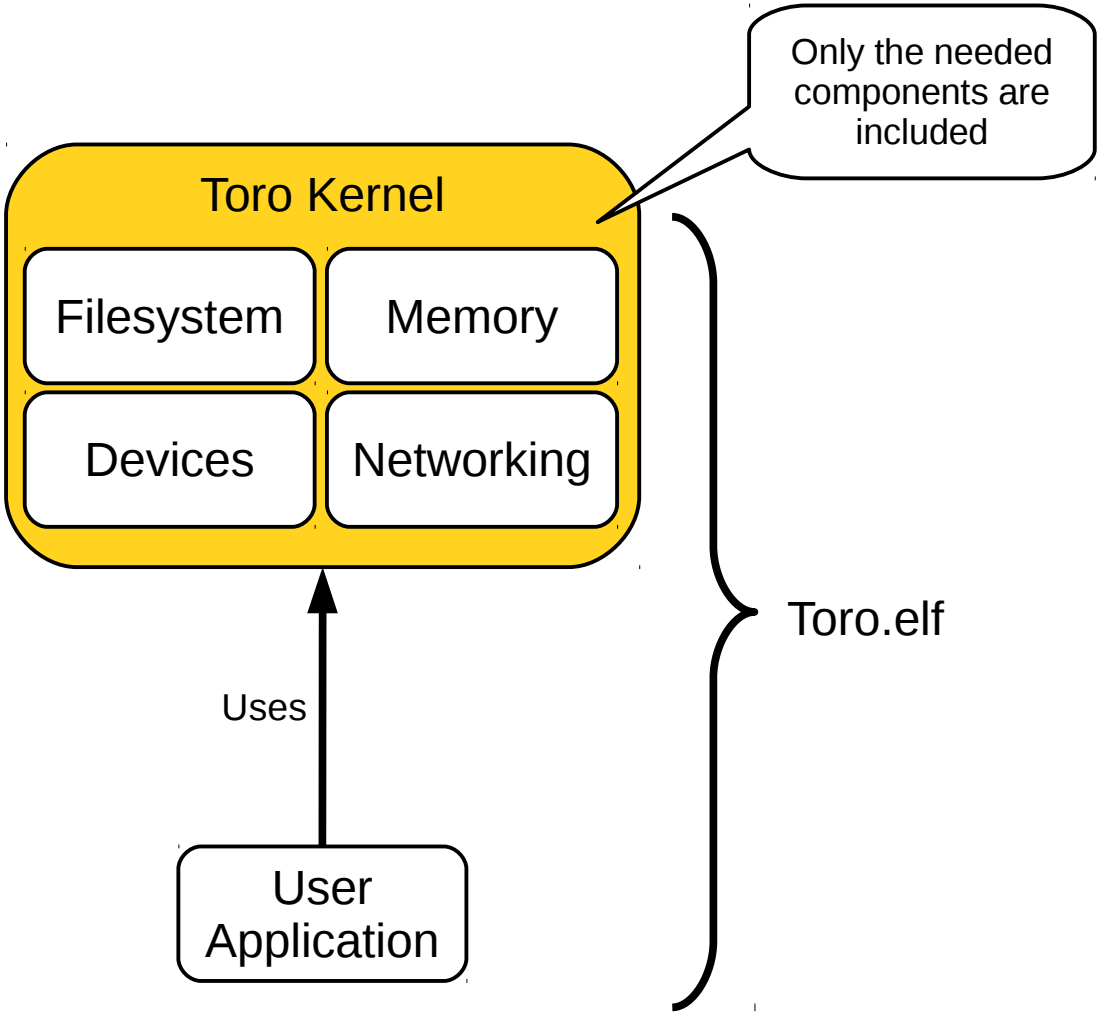
User application within the Kernel Proposal

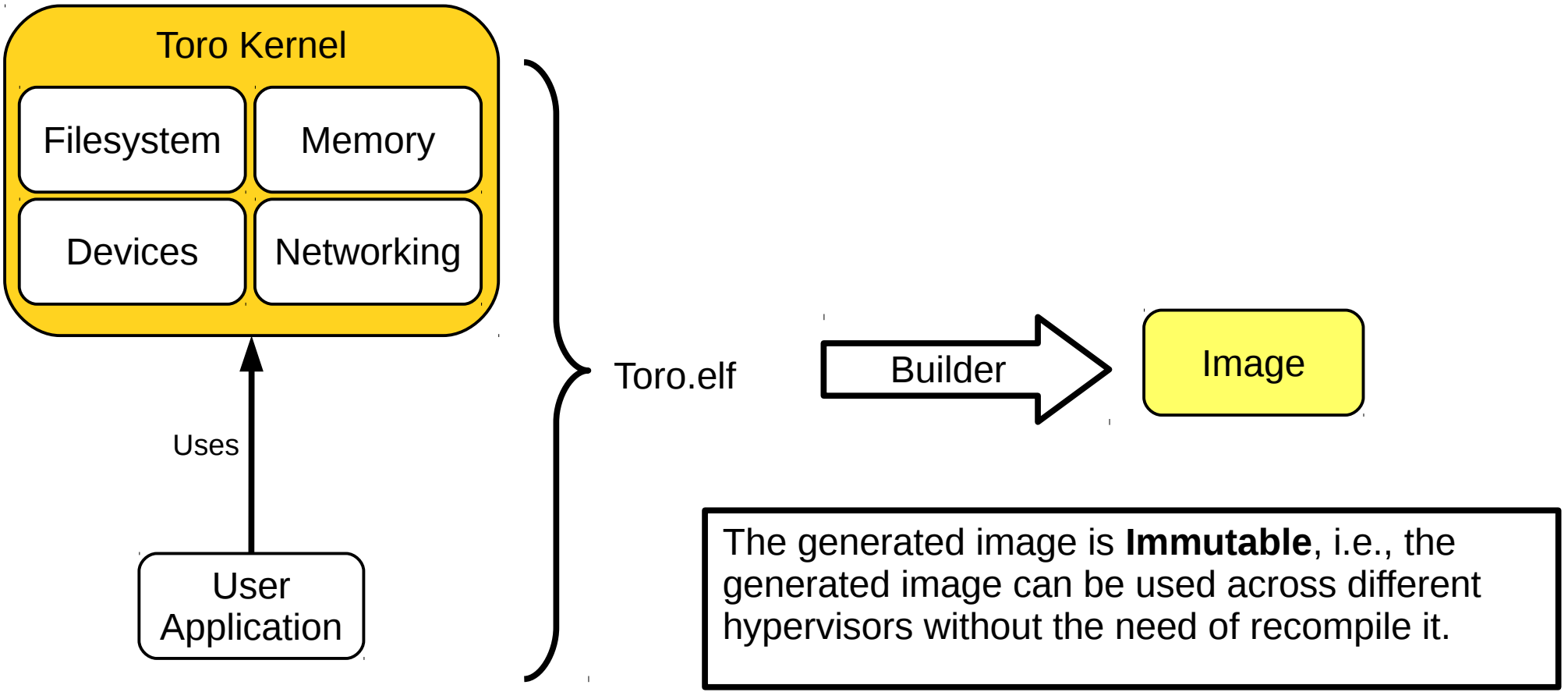
1. Run kernel and user application in the most privileged level. The protection is provided by the hypervisor that isolates the context of each VMs

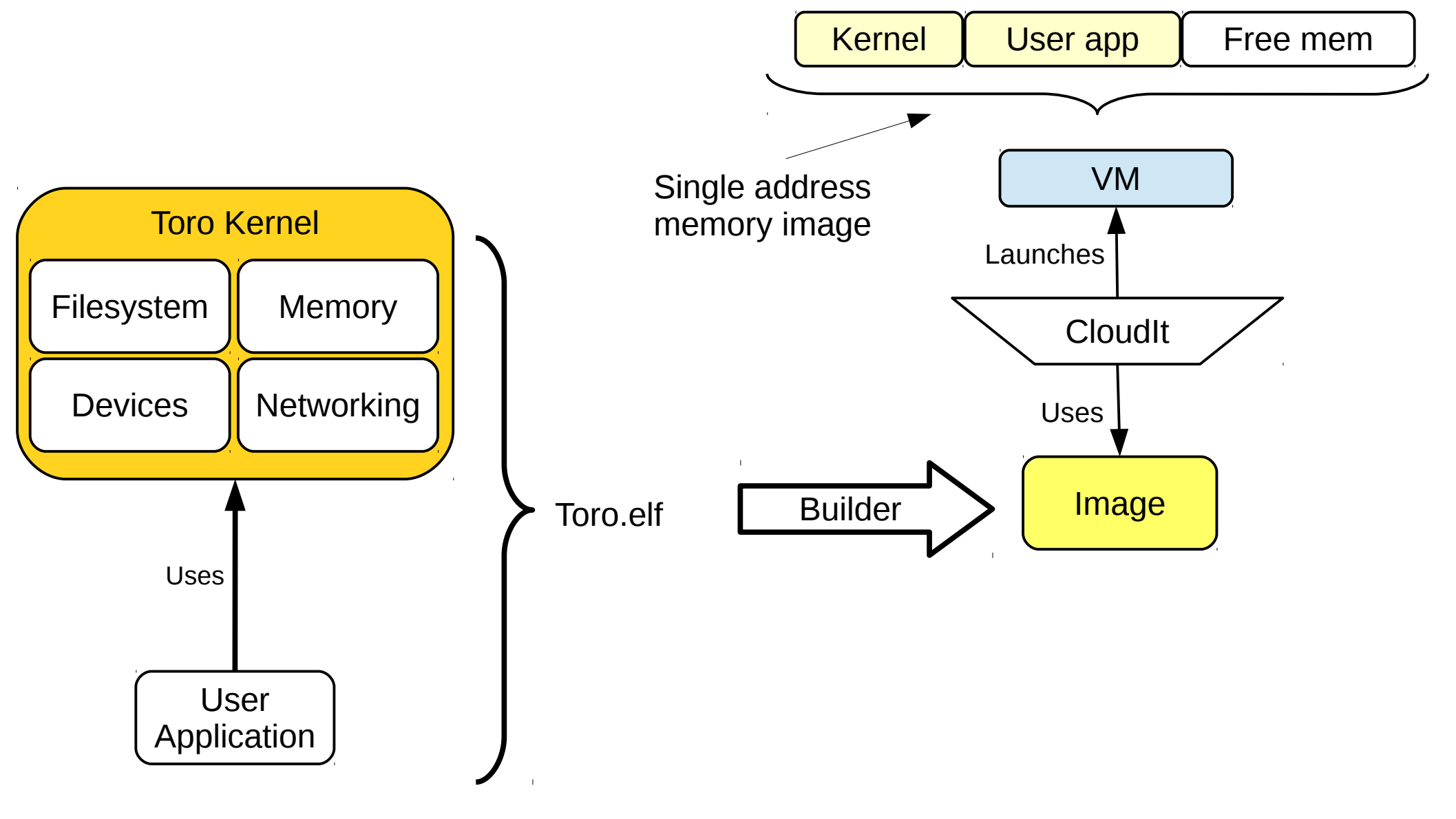
2. Use a flat memory model that is shared by the kernel and the user application

3. Use only threads

4. Provide a simple kernel API dedicated to microservices


Only the needed components are included
Toro Kernel
Filesystem
Memory
Devices
Networking
Toro.elf
Uses
User Application


Toro Kernel
Filesystem
Memory
Devices
Networking
Uses
User Application
Toro.elf
Builder
Image
The generated image is **Immutable**, i.e., the generated image can be used across different hypervisors without the need of recompile it.

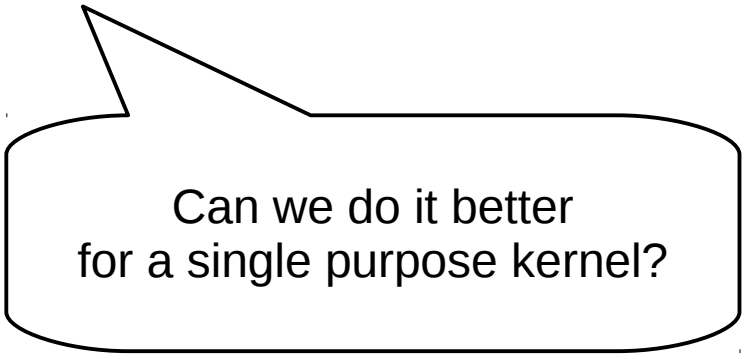

Kernel
User app
Free mem
Single address
memory image
VM
Launches
CloudIt
Uses
Image
Builder
Toro.elf
Toro Kernel
Filesystem
Memory
Devices
Networking
Uses
User
Application

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



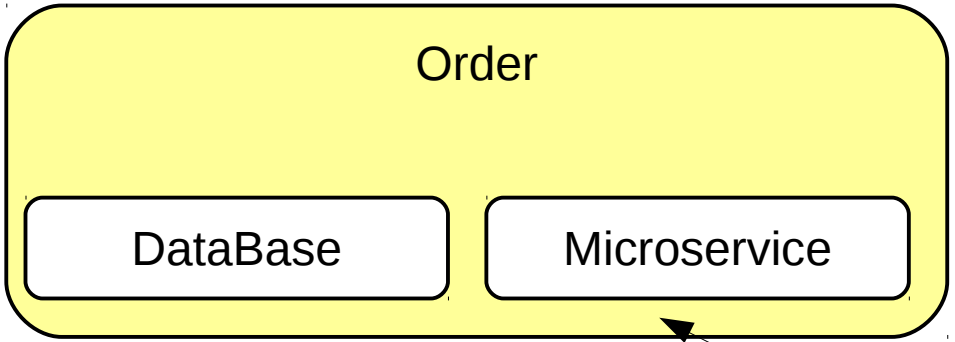
Cooperative Scheduler Proposal

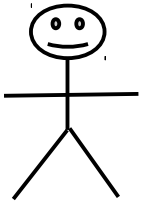
1. Cooperative scheduler, i.e., each thread decides when yield the CPU

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

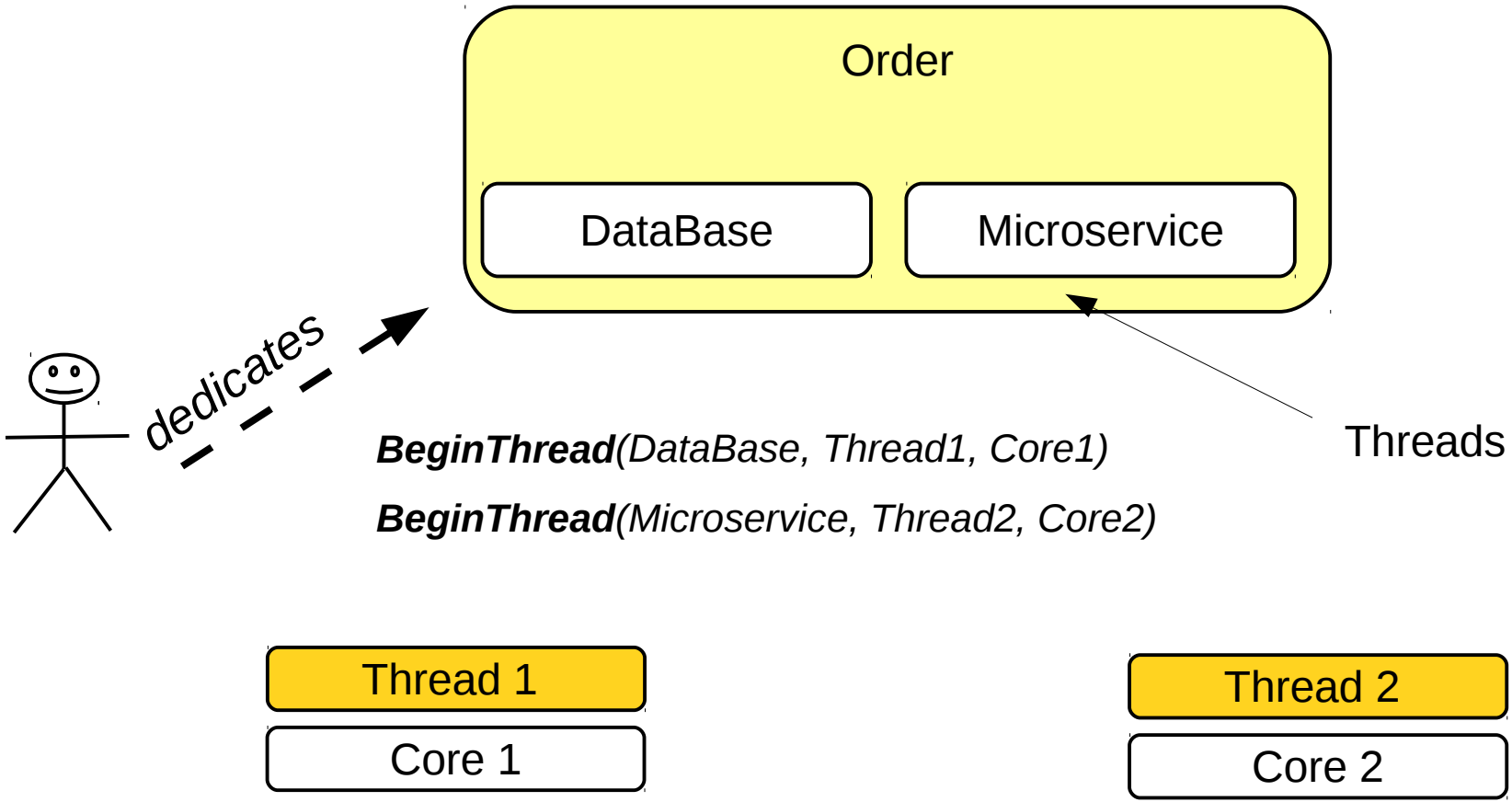
2. One scheduler per core

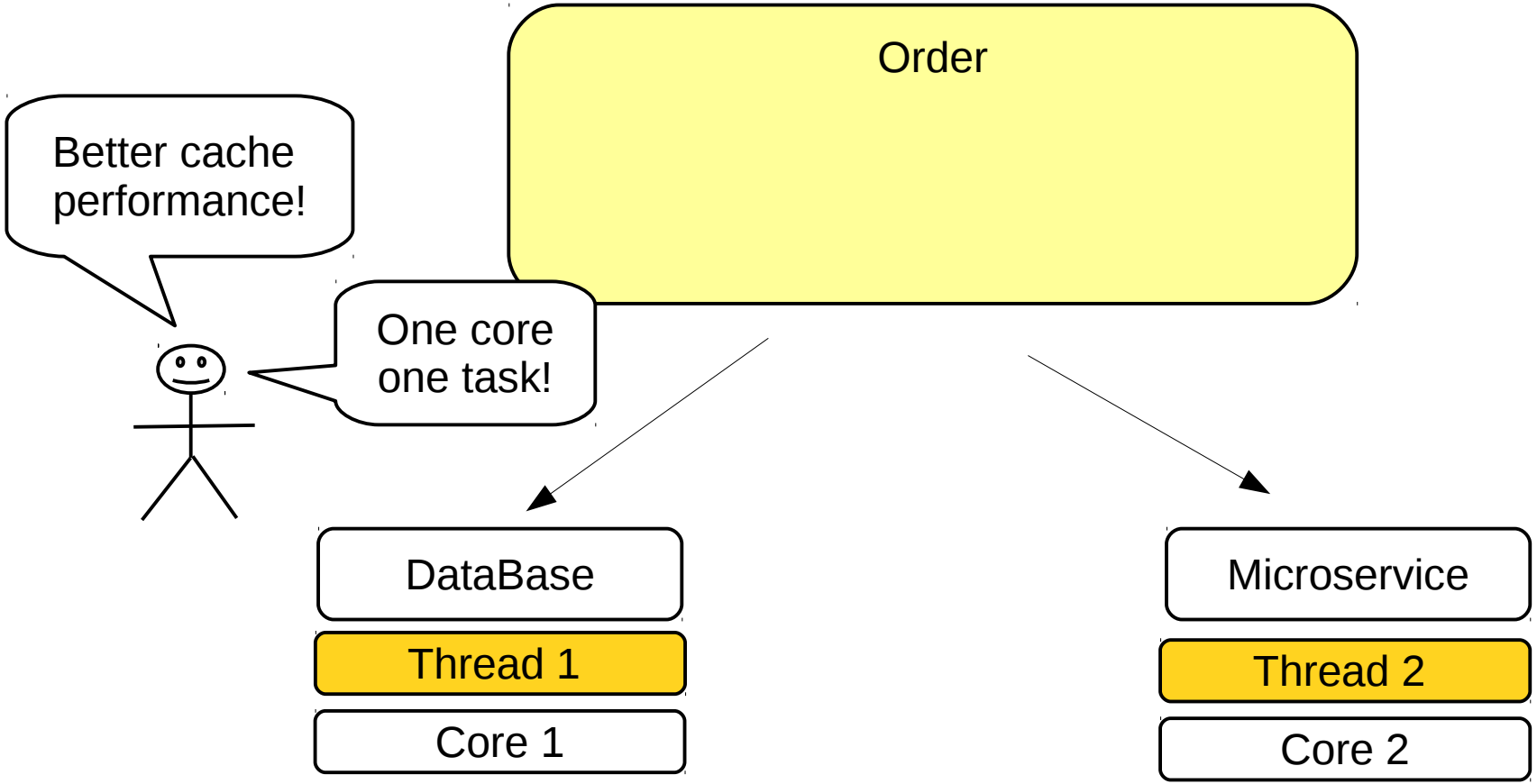
3. Remote creation of threads by relying on a lock-free algorithm





Threads


Order
DataBase
Microservice
dedicates
Threads
BeginThread(DataBase, Thread1, Core1)
BeginThread(Microservice, Thread2, Core2)
Thread 1
Core 1
Thread 2
Core 2


Order
Better cache performance!
One core one task!
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

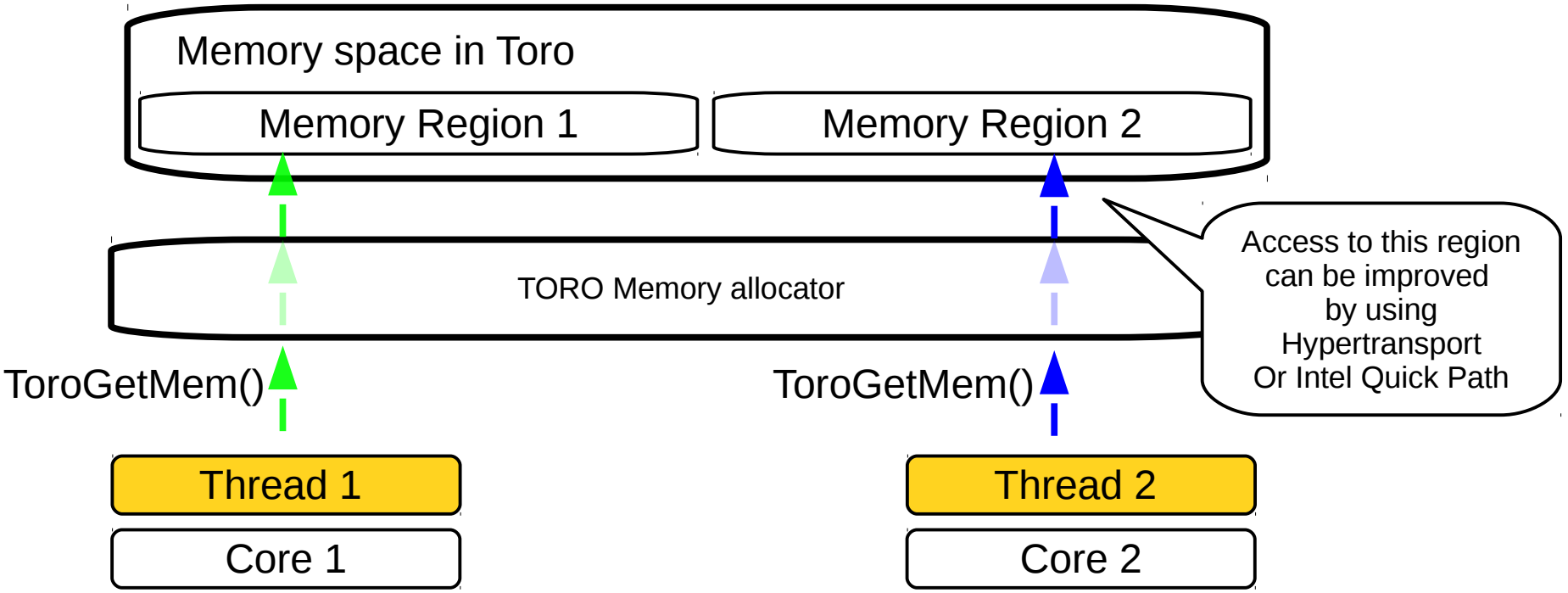
Dedicated Resources

- 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

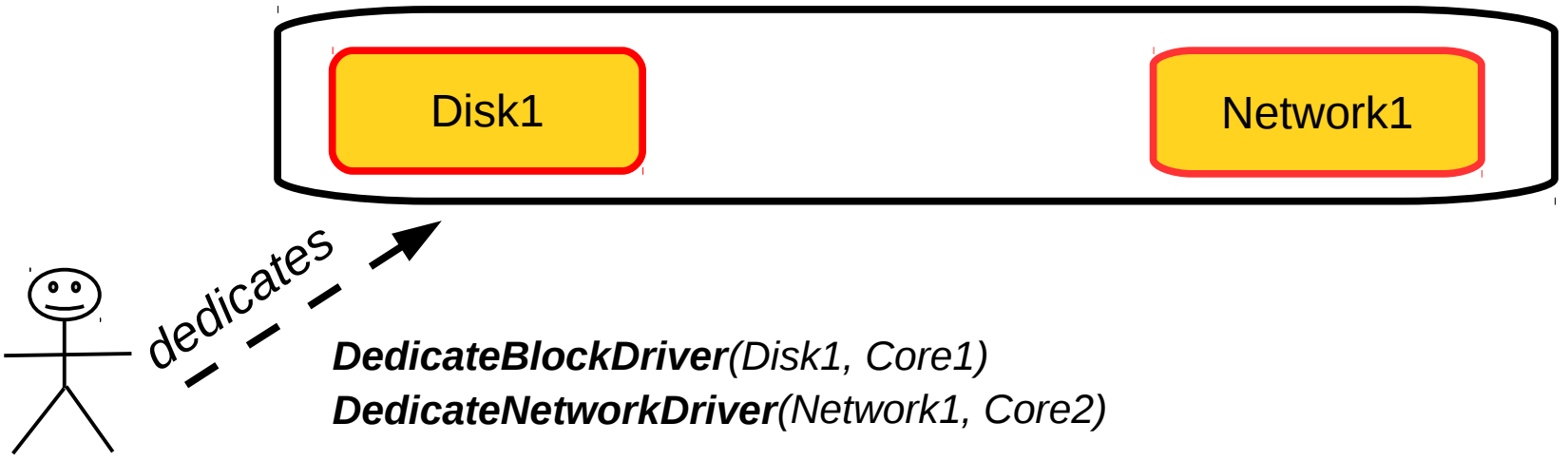
Dedicated Resources Proposal

- Toro improves memory access by keeping the resources locals:
  - The memory is dedicated per core
  - The kernel data structure is dedicated per core
  - The access to kernel data structures is lock free

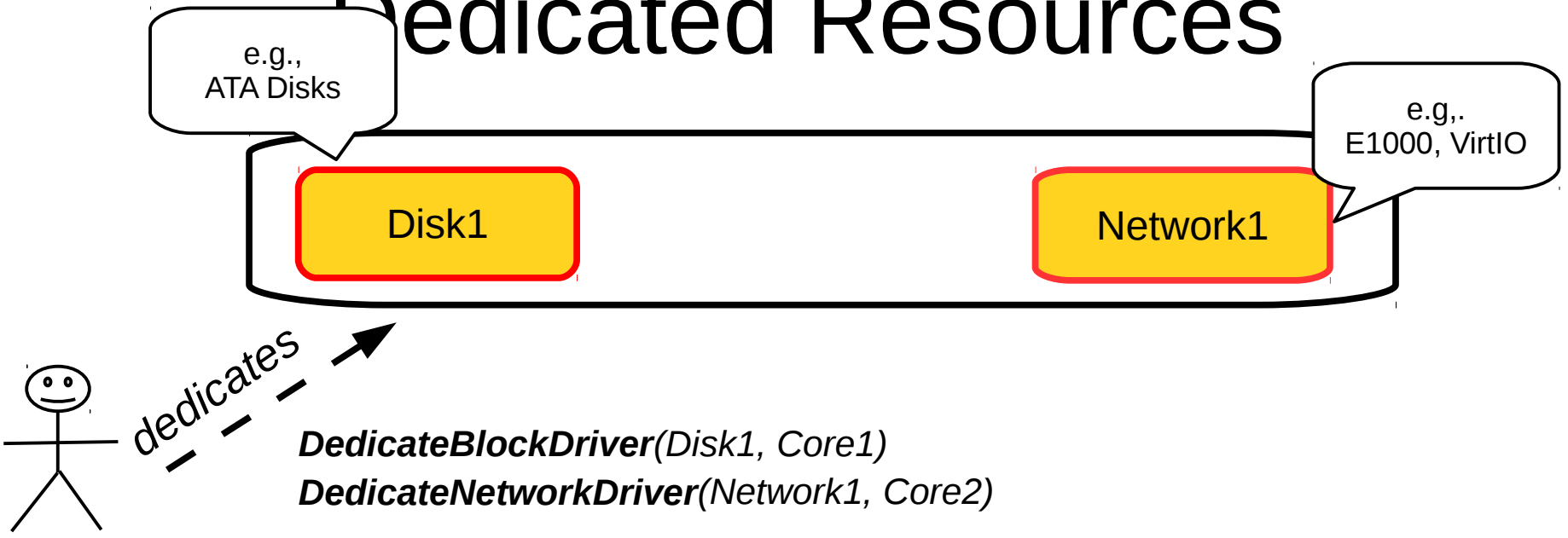
Dedicated Resources



Dedicated Resources

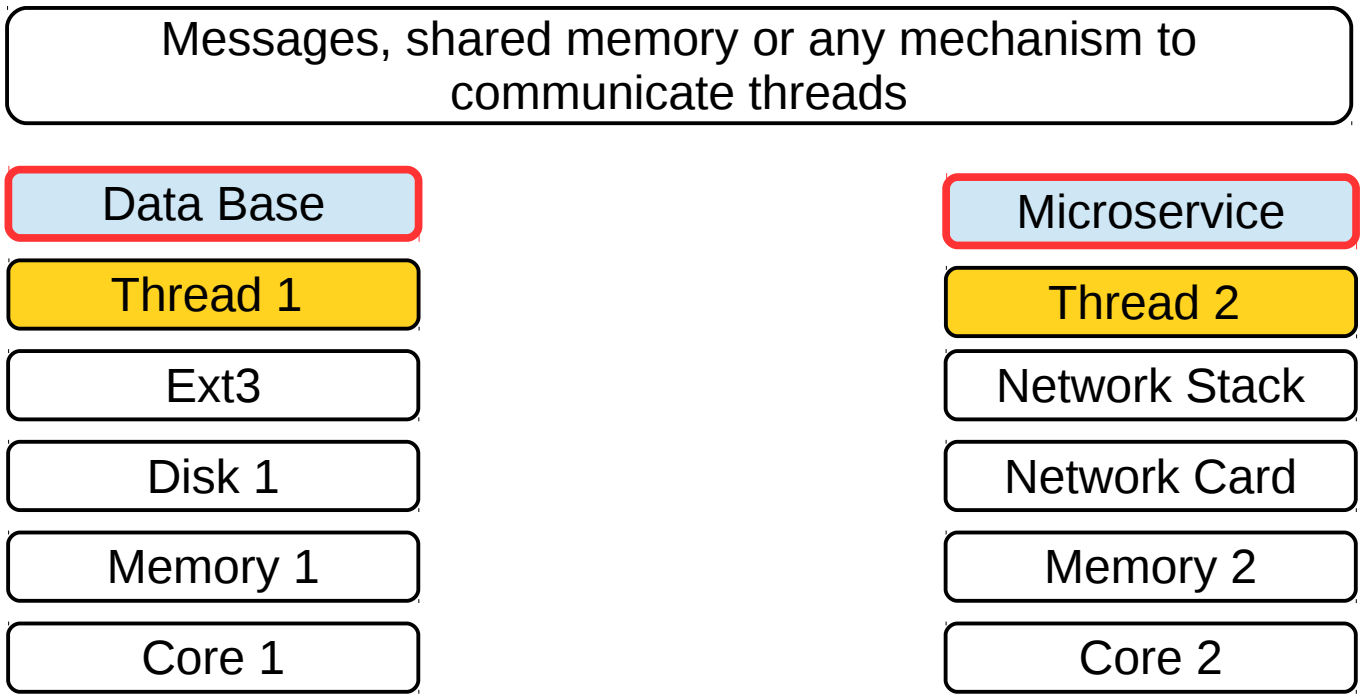


Dedicated Resources

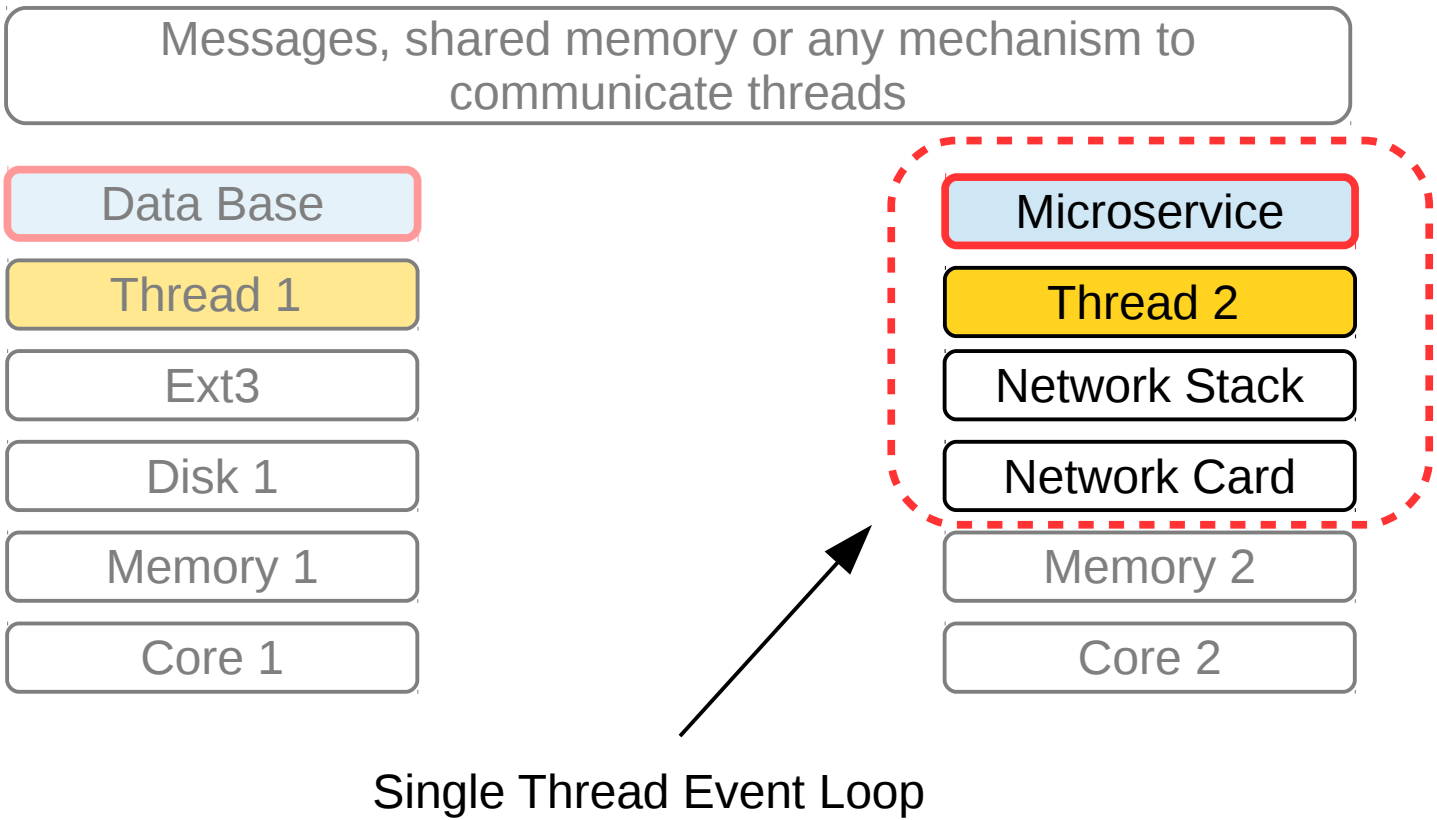


Dedicated Resources

Dedicated Resources



Dedicated Resources



Ingredients for ToroKernel

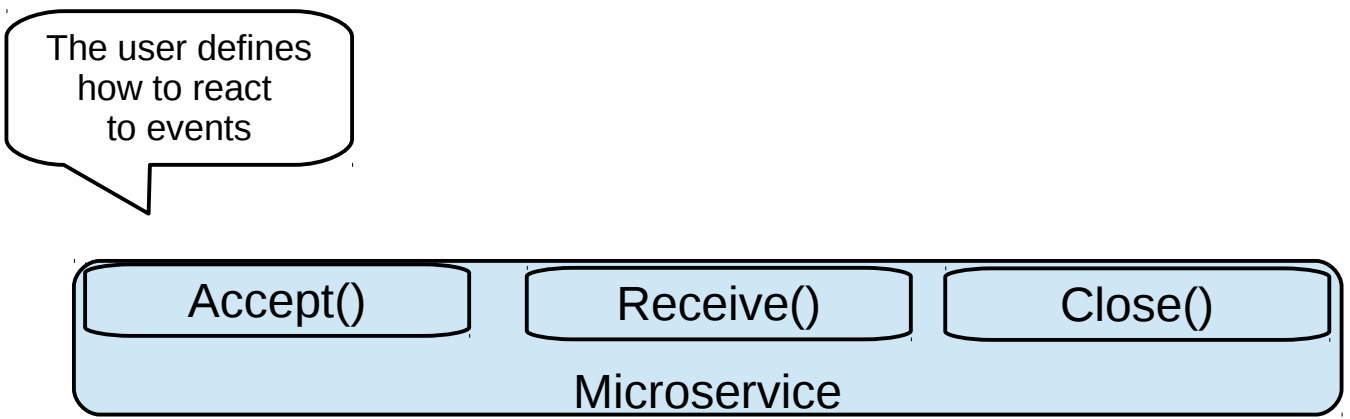
- User application within the kernel
- Cooperative Scheduler
- Dedicated Resources
- Single Thread Event Loop Networking

Single Thread Event Loop

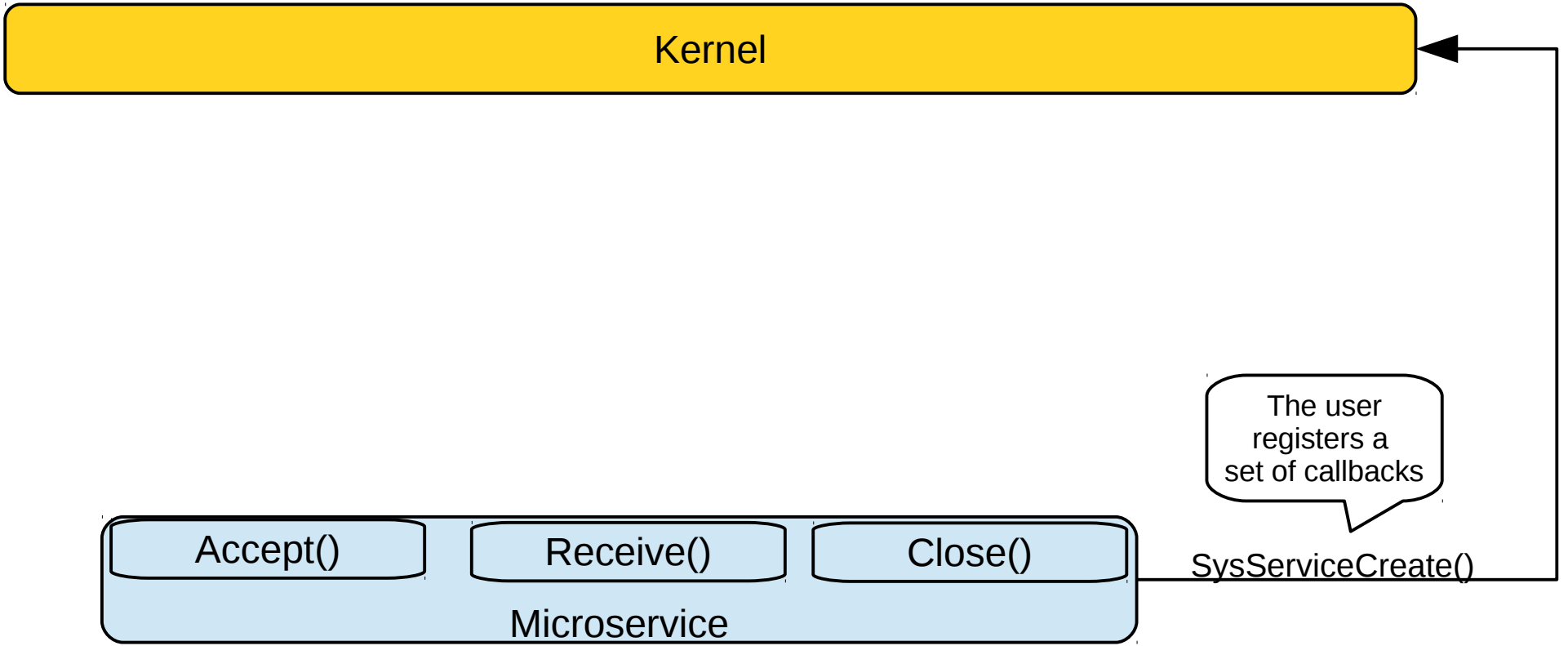
- Toro networking is based on the single thread event loop model [1], i.e., one thread per microservice
- The kernel provides a dedicated API to create microservices
- The kernel implements the microservice and improves the CPU usage[2]

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

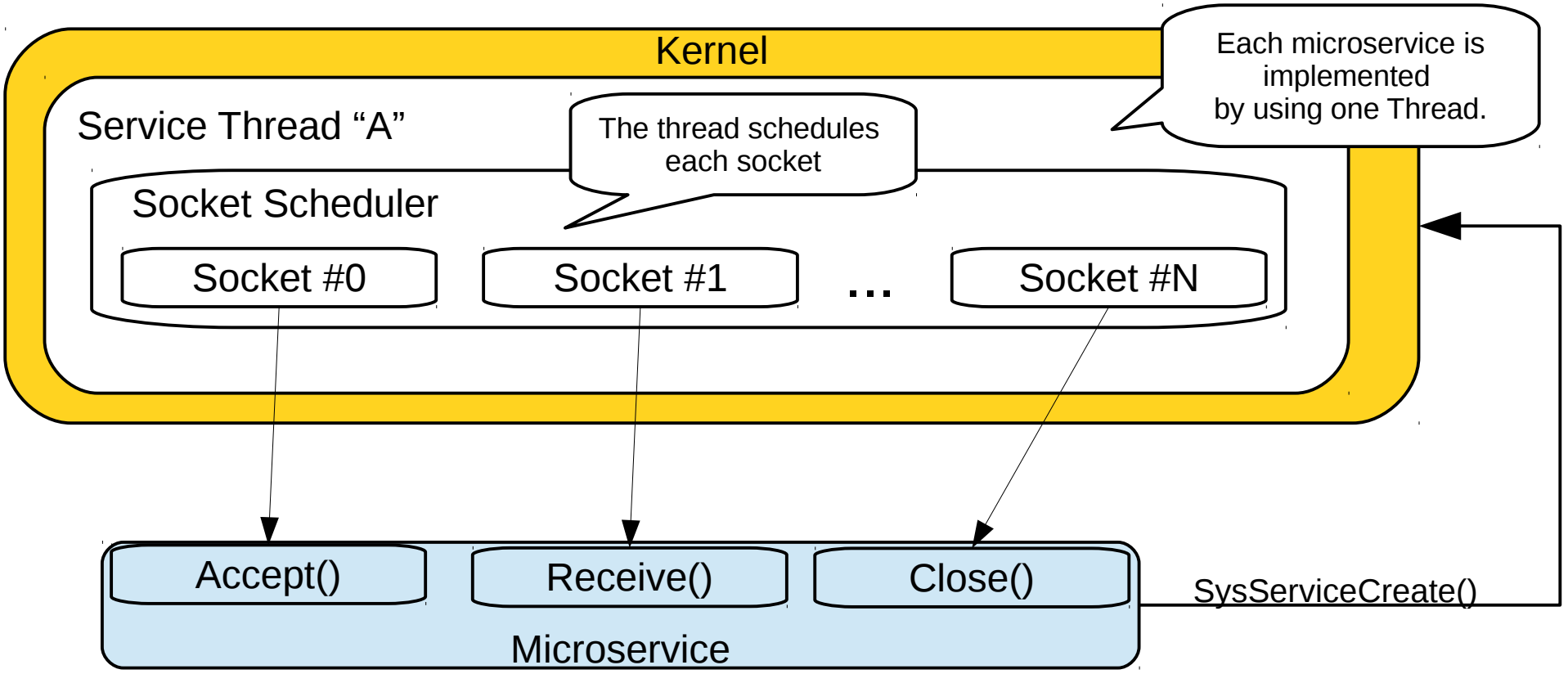
Single Thread Event Loop



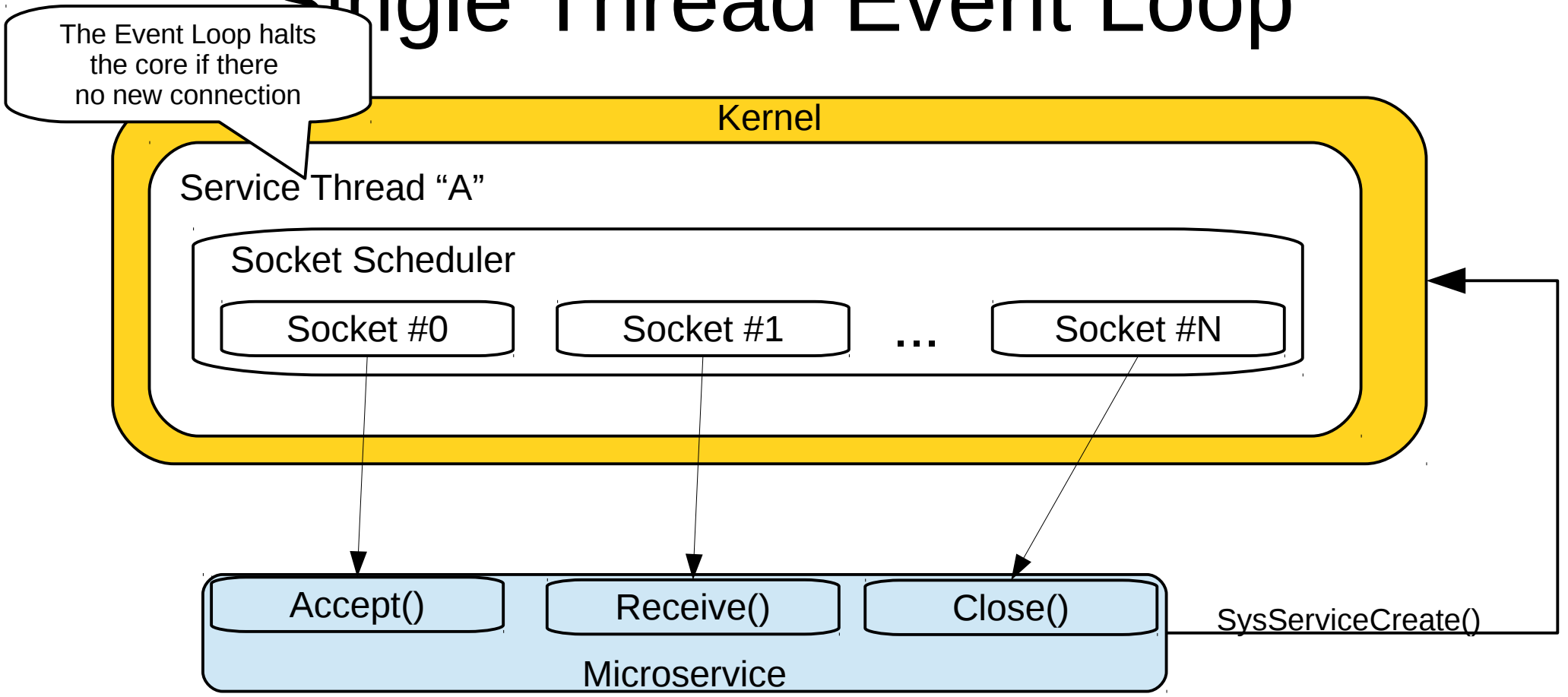
Single Thread Event Loop



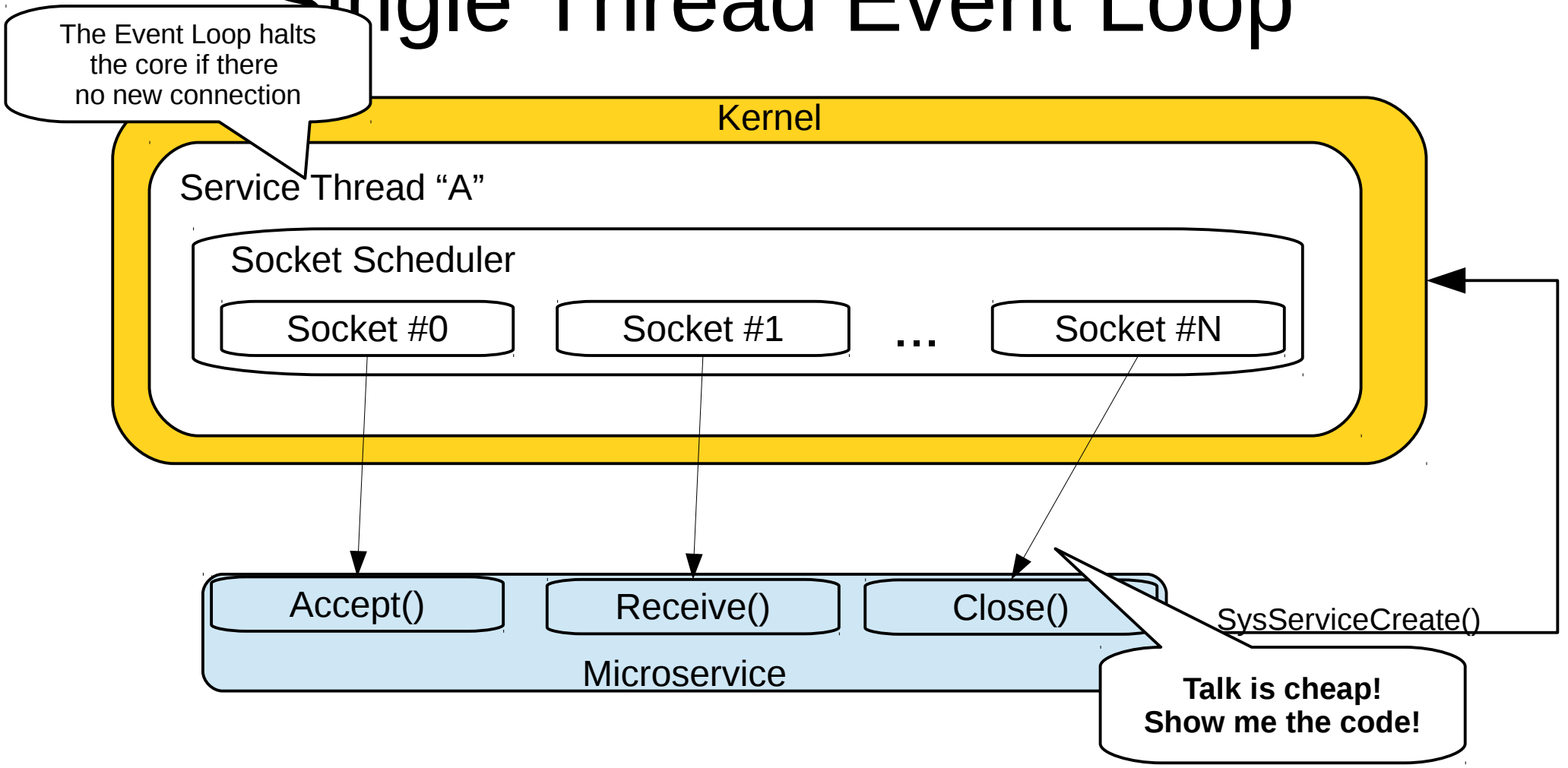
Single Thread Event Loop



Single Thread Event Loop



Single Thread Event Loop



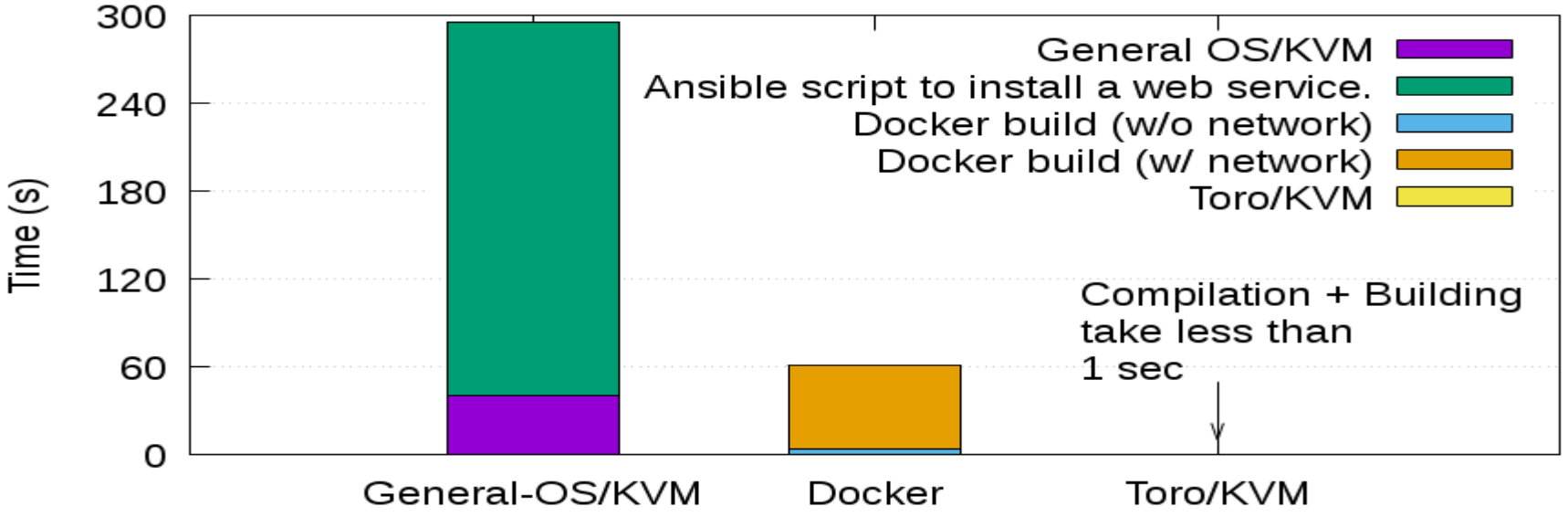
HelloWorld Microservice Example

- Simple microservice that responds “Hello World”
- Setup:

Docker	General OS/KVM	Toro/KVM
• Cpu limit • Memory limit • NGINX • UWSGI (4 processes) • Flask	• Cpu limit • Memory limit • NGINX • UWSGI (4 processes) • Flask	• CPU limit • Memory limit • Toro WWW server

Deploying Time

- Time required to build an image within the microservice



Bootstrap Time

- Time to boot and to answer the first request

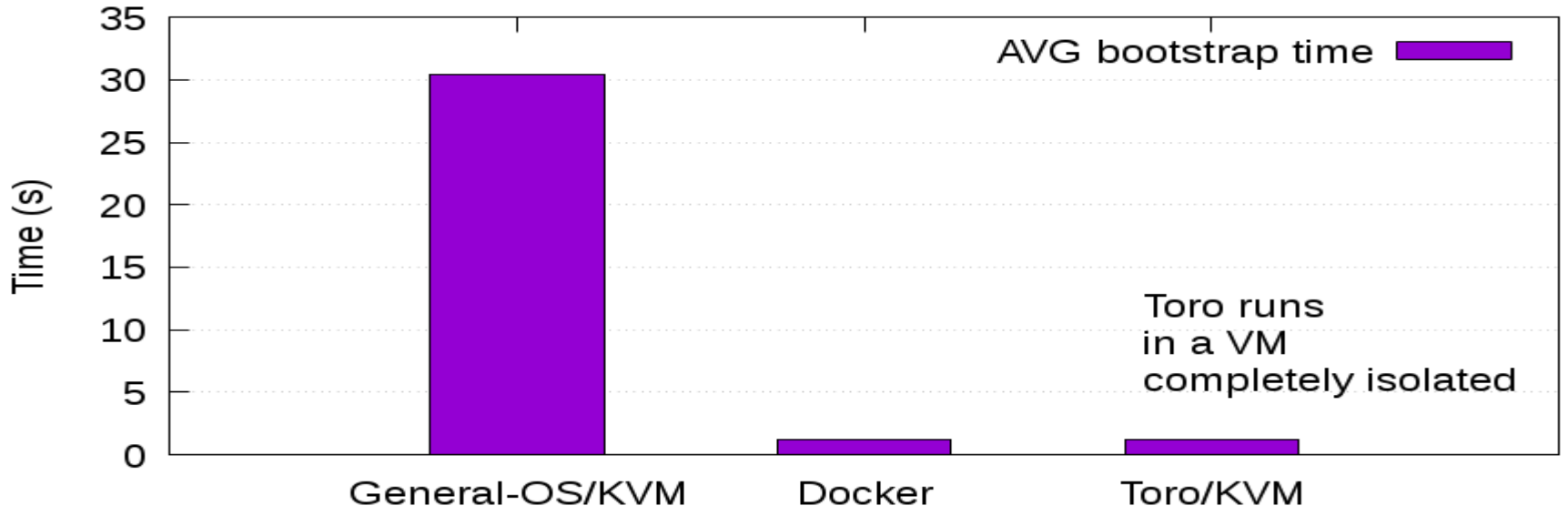
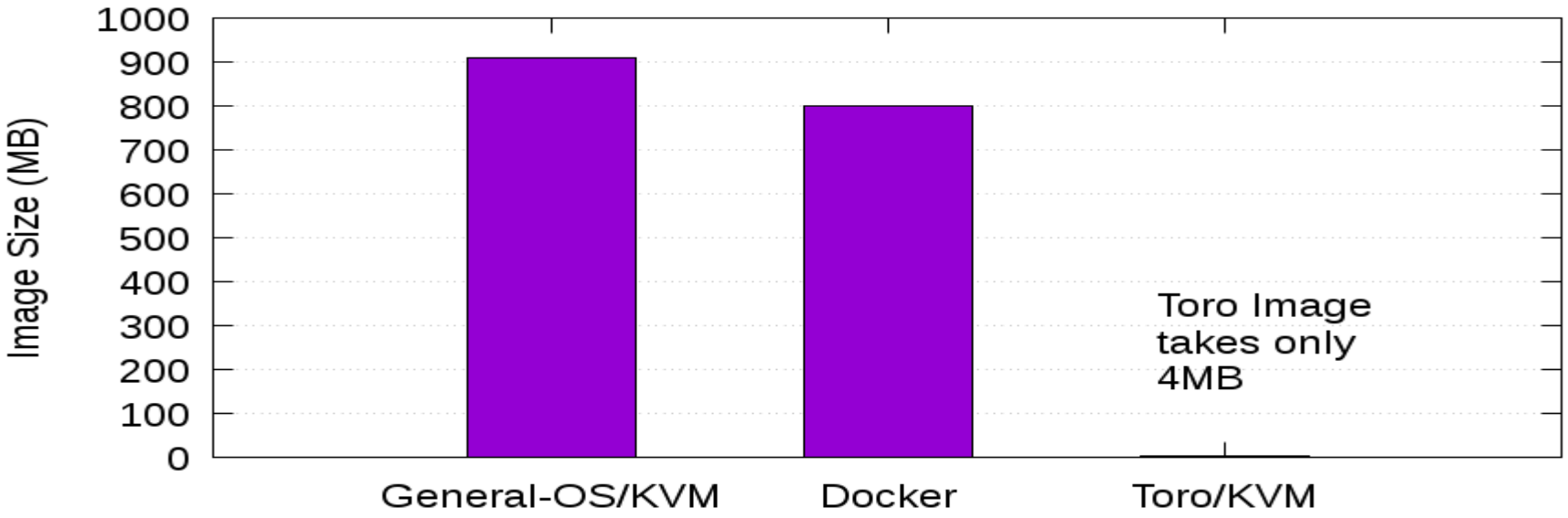
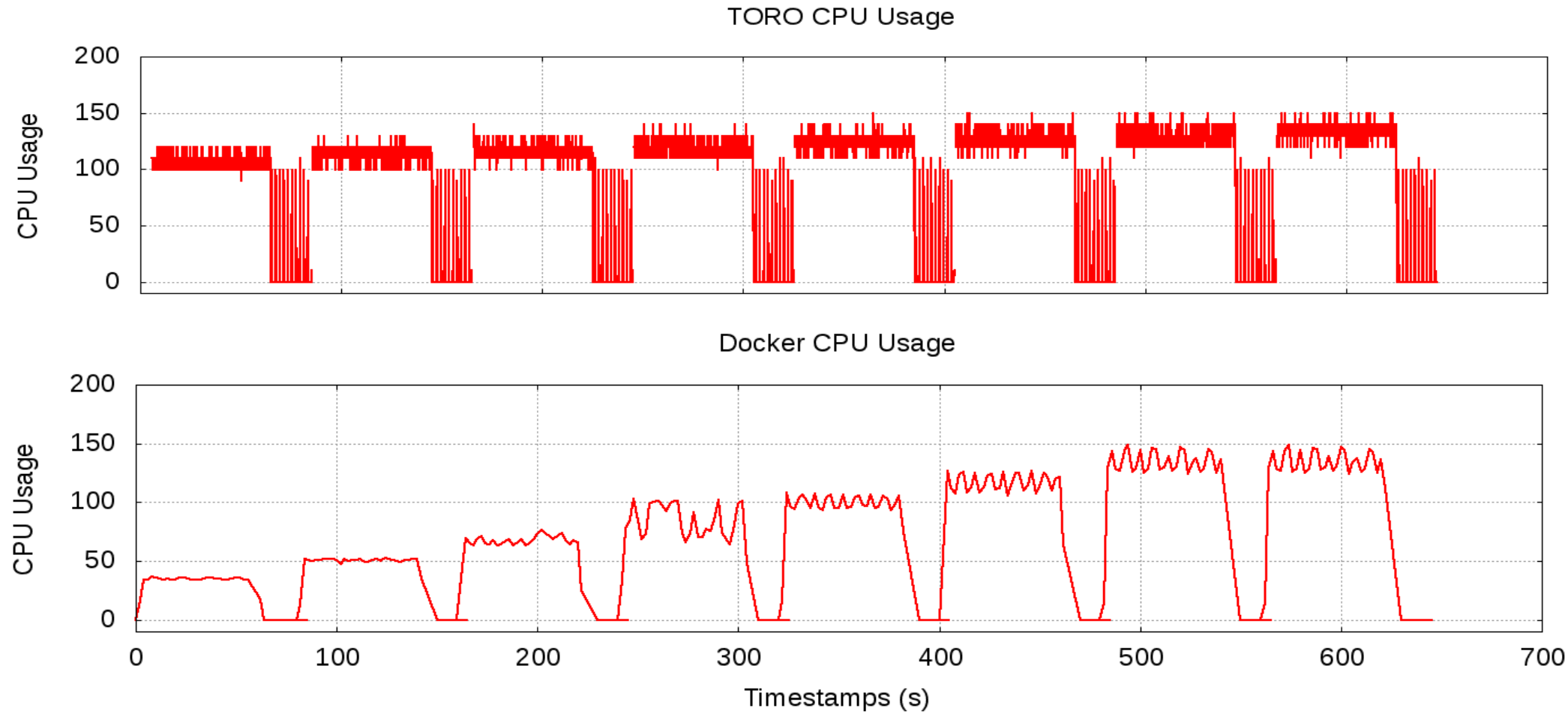


Image Size

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



CPU Usage


TORO CPU Usage
CPU Usage
200
150
100
50
0
Docker CPU Usage
CPU Usage
200
150
100
50
0
0
100
200
300
400
500
600
700
Timestamps (s)

End-User Delay

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

		Number of Concurrent Request		
Approach	CPUs	200	500	1000
Docker	4	102 ms	243,295 ms	474,487
OS/KVM	4	74 ms	176 ms	364 ms
TORO/KVM	1	143,456 ms	362,881 ms	720,961 ms

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!