From data centers to fog computing: the evaporating cloud

Guillaume Pierre

Journées Cloud 2018

Table of Contents

1 Introduction

2 Fog computing

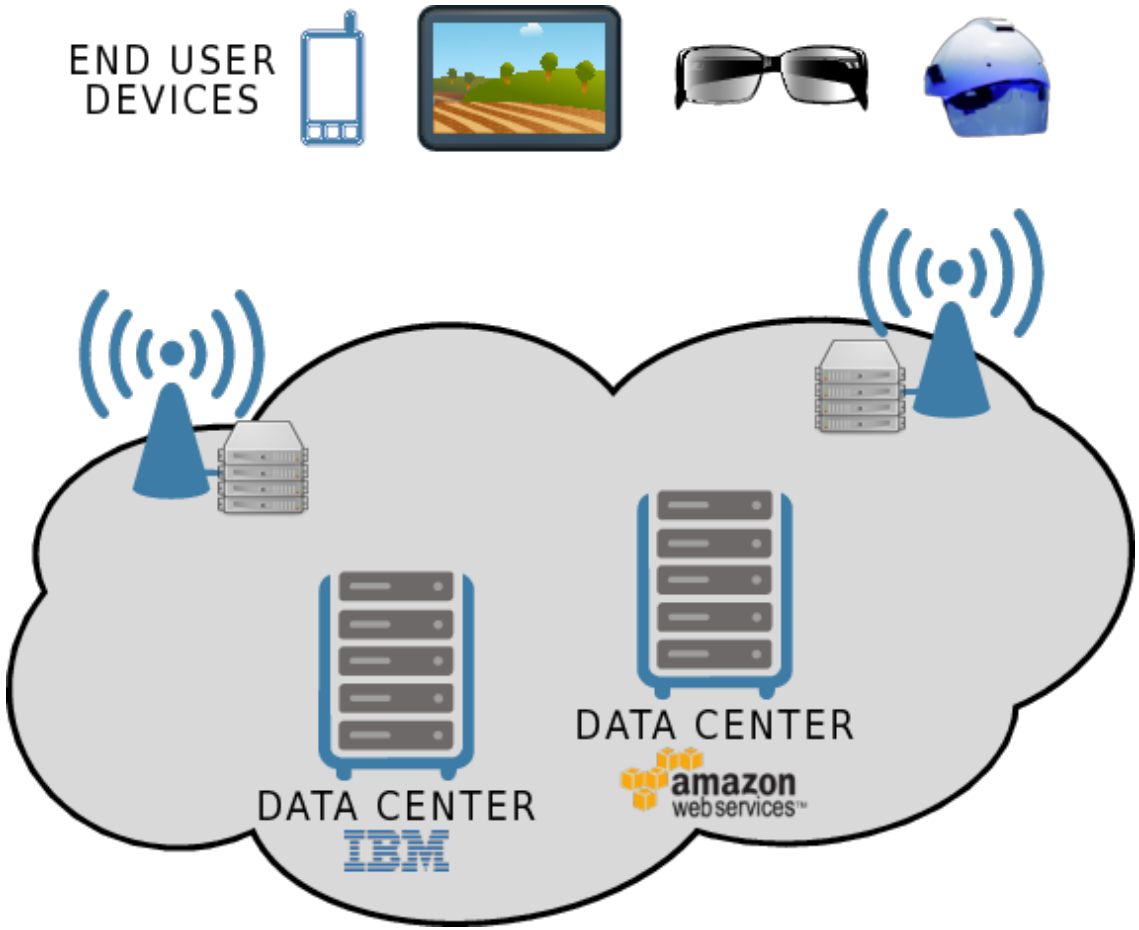
3 Making fog servers REALLY small

4 Toward proximity-aware Kubernetes

5 Conclusion

The new mobile computing landscape

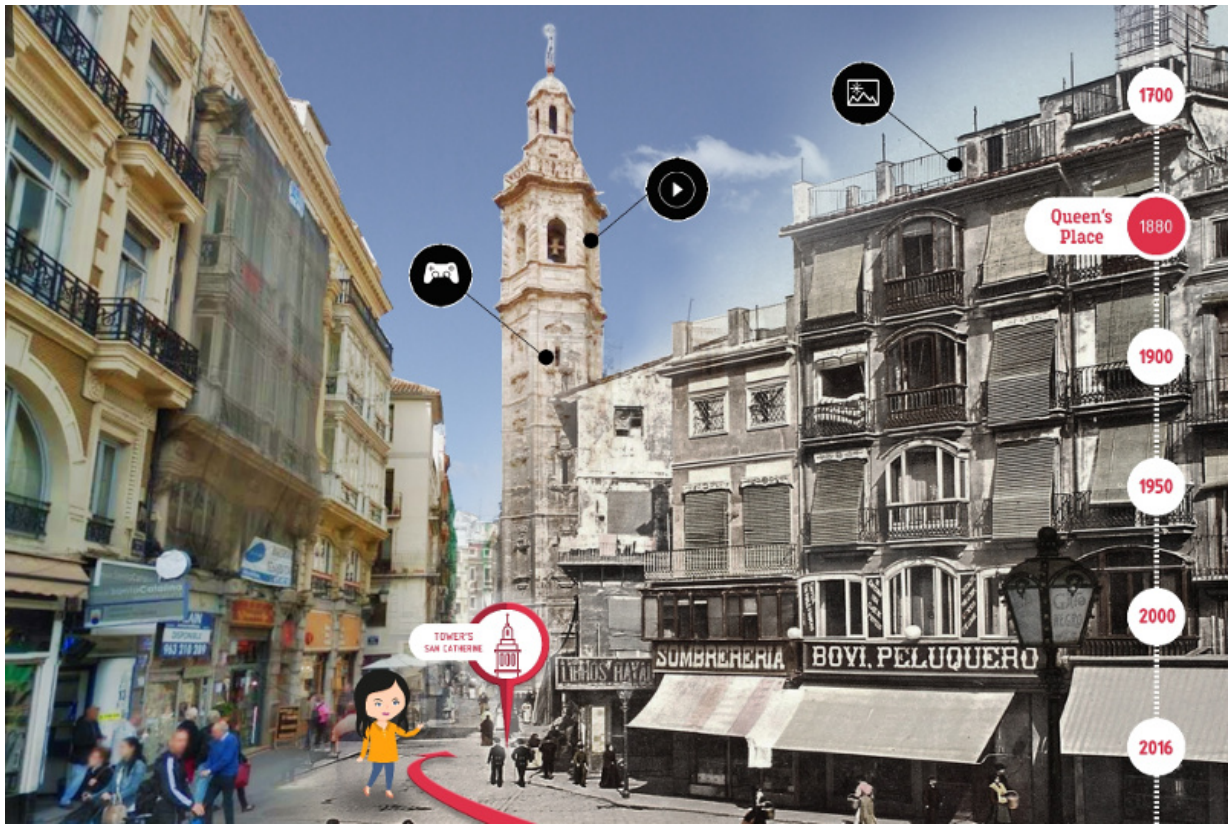
Since 2016: **mobile network traffic > fixed traffic**



New types of mobile applications

Interactive applications require ultra-low network latencies

- E.g., augmented reality require end-to-end delays under 20 ms
- But latencies to the closest data center are 20-30 ms using wired networks, up to 50-150 ms on 4G mobile networks!!!



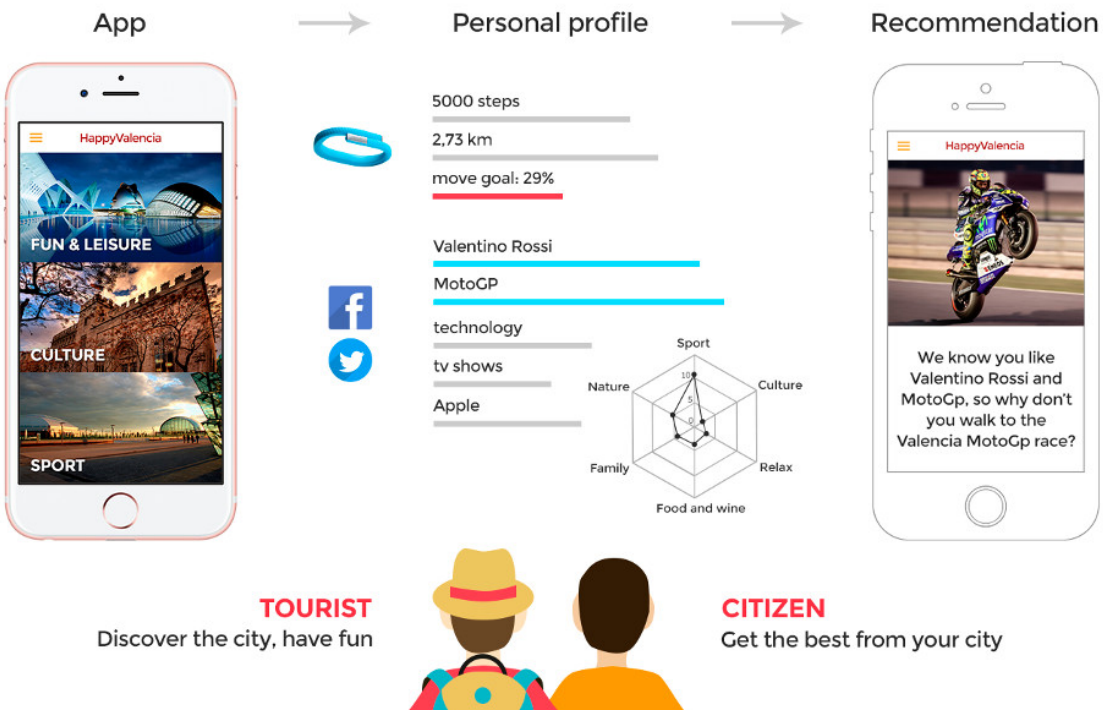
The new IoT landscape

Fact: IoT-generated traffic grows faster than the Internet backbone capacity

New types of mobile applications

Throughput-oriented applications require local computations

- E.g., distributed videosurveillance is relevant only close to the cameras
- Why waste long-distance network resources?



Question

Who owns computing resources located closest to the mobile end users and the IoT devices?

Who owns computing resources located closest to the mobile end users and the IoT devices?

Answer: Mobile network operators

Table of Contents

1 Introduction

2 Fog computing

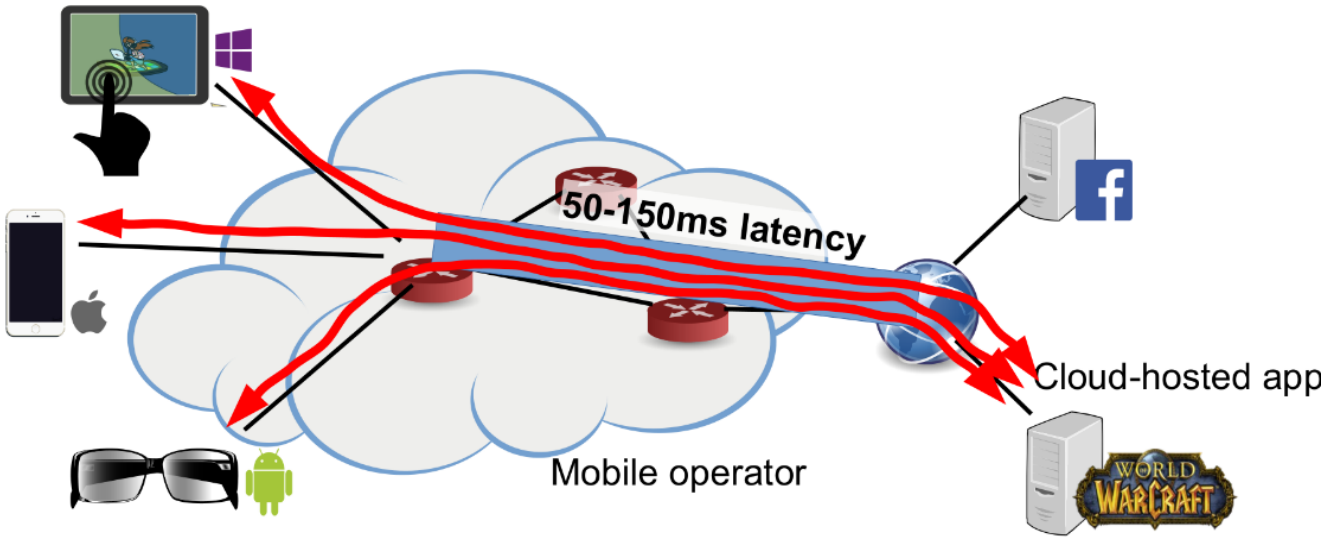
3 Making fog servers REALLY small

4 Toward proximity-aware Kubernetes

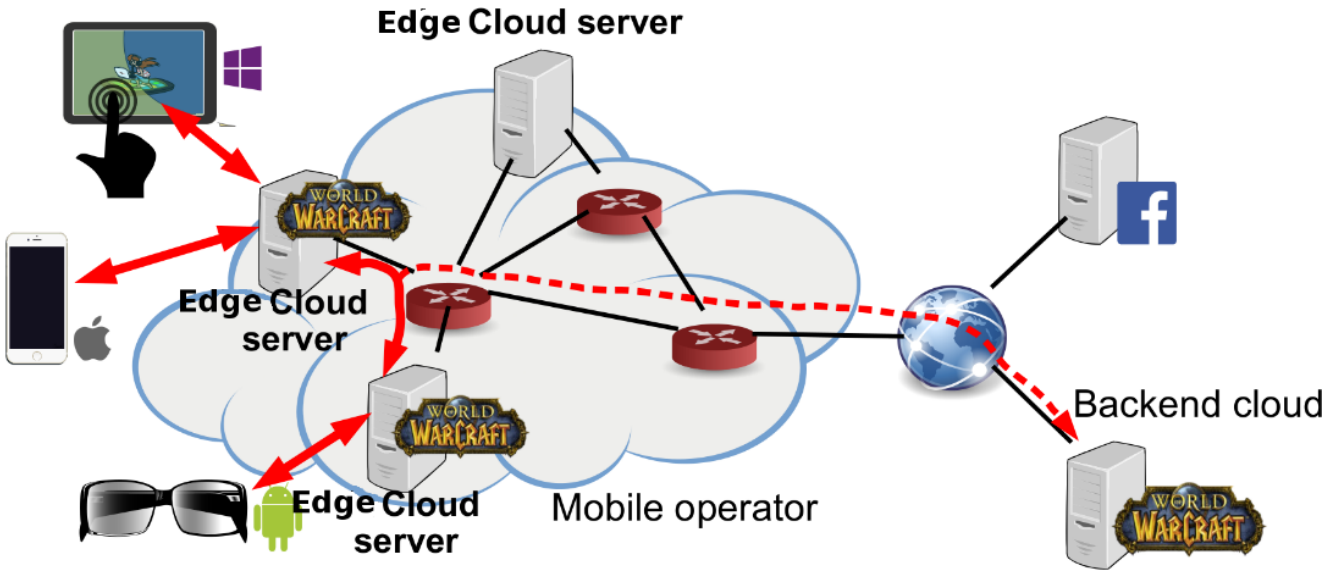
5 Conclusion

Edge computing

Before:



After:



What is fog computing?

Fog computing = cloud
+ edge
+ end-user devices
as a single execution platform

- Low latency
- Localized traffic
  - Less global traffic
  - Better reliability
  - Privacy
  - ...

Fog $\neq$ Cloud

- We need cloud servers close to the users, but the users are everywhere (and they are mobile)
  - Let's place one cloud server within short range of any end-user device

  $\Rightarrow$ Fog computing resources will need to be distributed in thousands of locations

Fog $\neq$ Cloud

- We need cloud servers close to the users, but the users are everywhere (and they are mobile)
  - Let's place one cloud server within short range of any end-user device
  - $\Rightarrow$ Fog computing resources will need to be distributed in thousands of locations
- Fog nodes will be connected with commodity networks
  - Forget your multi-Gbps data center networks!

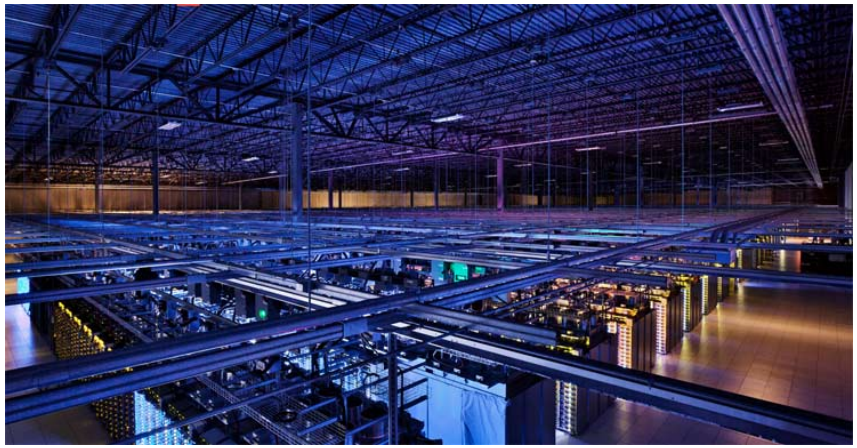
Fog $\neq$ Cloud

- We need cloud servers close to the users, but the users are everywhere (and they are mobile)
  - Let's place one cloud server within short range of any end-user device
  - $\Rightarrow$ Fog computing resources will need to be distributed in thousands of locations
- Fog nodes will be connected with commodity networks
  - Forget your multi-Gbps data center networks!
- Fog nodes must be small, cheap, easy to deploy and to replace

Fog ≠ Cloud

Typical Cloud platform

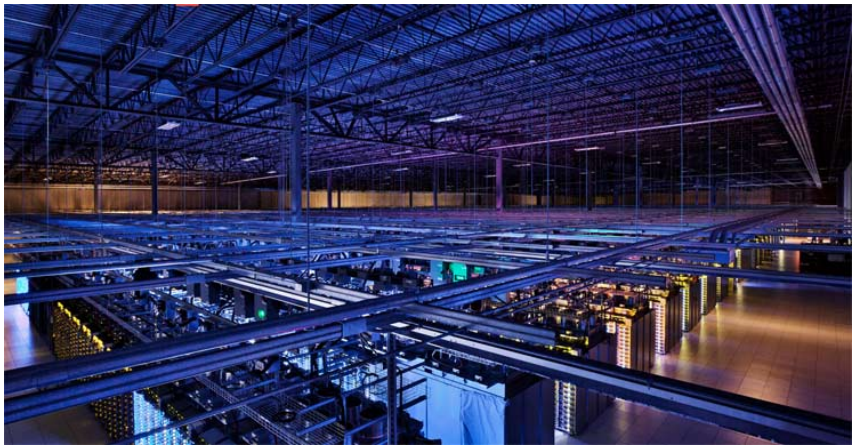
- Few data centers
- Lots of resources per data center
- High-performance networks
- Cloud resource location is (mostly) irrelevant



Fog ≠ Cloud

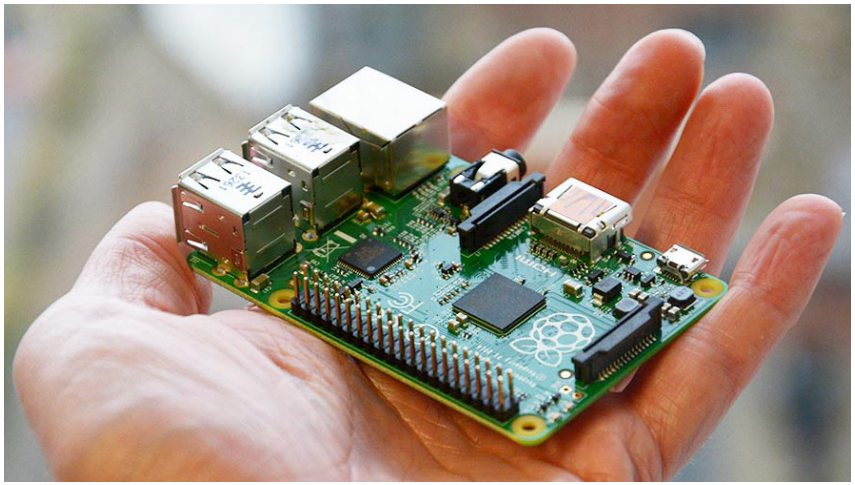
Typical Cloud platform

- Few data centers
- Lots of resources per data center
- High-performance networks
- Cloud resource location is (mostly) irrelevant



Typical Fog platform

- Huge number of points-of-presence
- Few resources per PoP
- Commodity networks
- Fog resource location is extremely important



The FogGuru project

- No general-purpose reference fog platform available
  - Mostly cloud systems which pretend to address fog concerns
  - But with very little solutions to the specific challenges of fog computing
  - FogGuru will investigate some of the most challenging management issues in fog platforms: resource scheduling, service migration, autonomous management, anomaly detection...
- No fog-specific middleware
  - FogGuru will investigate how to adapt Apache Flink for the fog
- No guidelines on how to develop new fog applications
  - FogGuru will develop blueprints for latency-sensitive and throughput-intensive applications
- No highly-trained specialist in fog computing
  - FogGuru will train 8 PhD students on various aspects of fog computing

fogguru

www.fogguru.eu

Table of Contents

1 Introduction

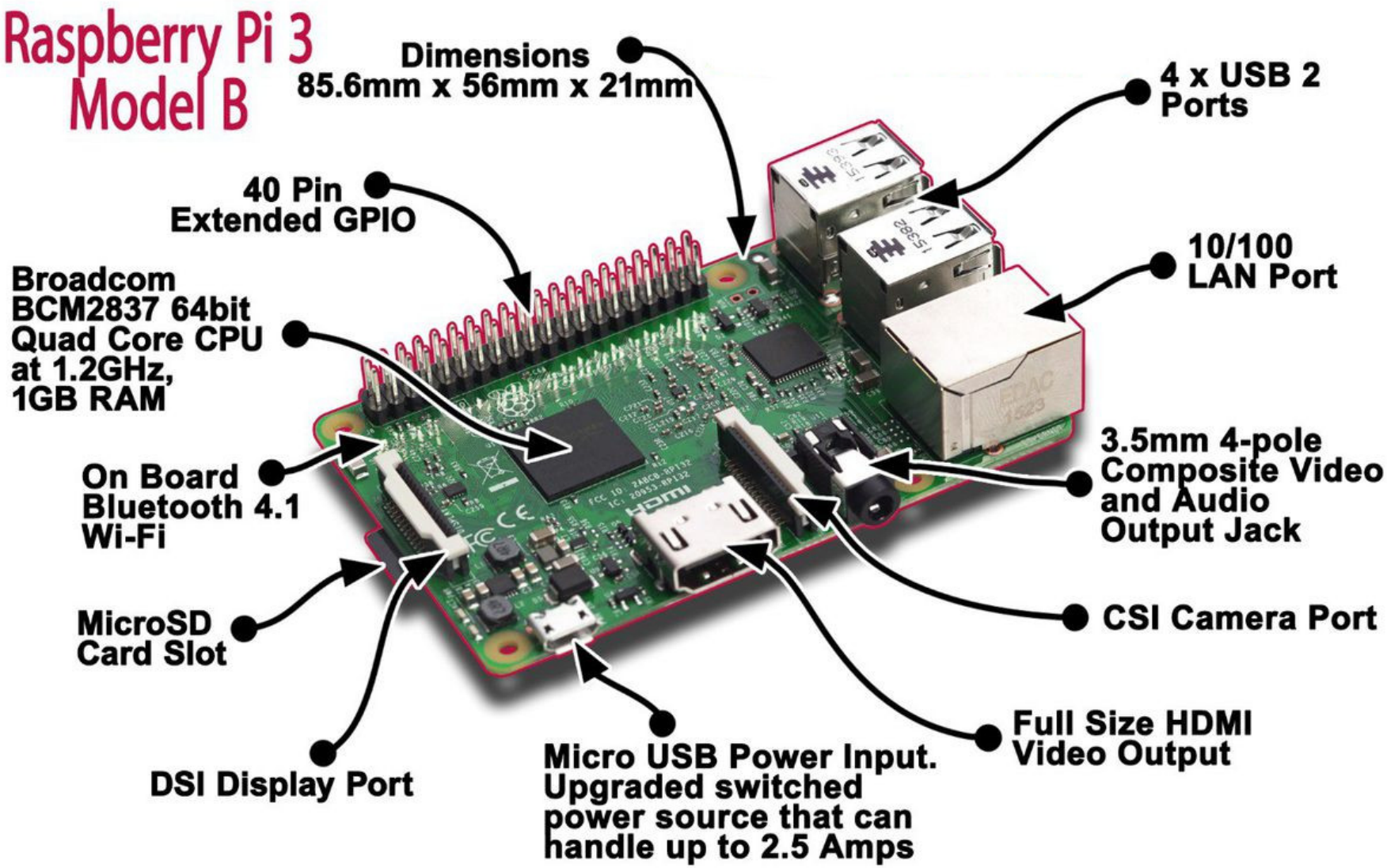
2 Fog computing

3 Making fog servers REALLY small

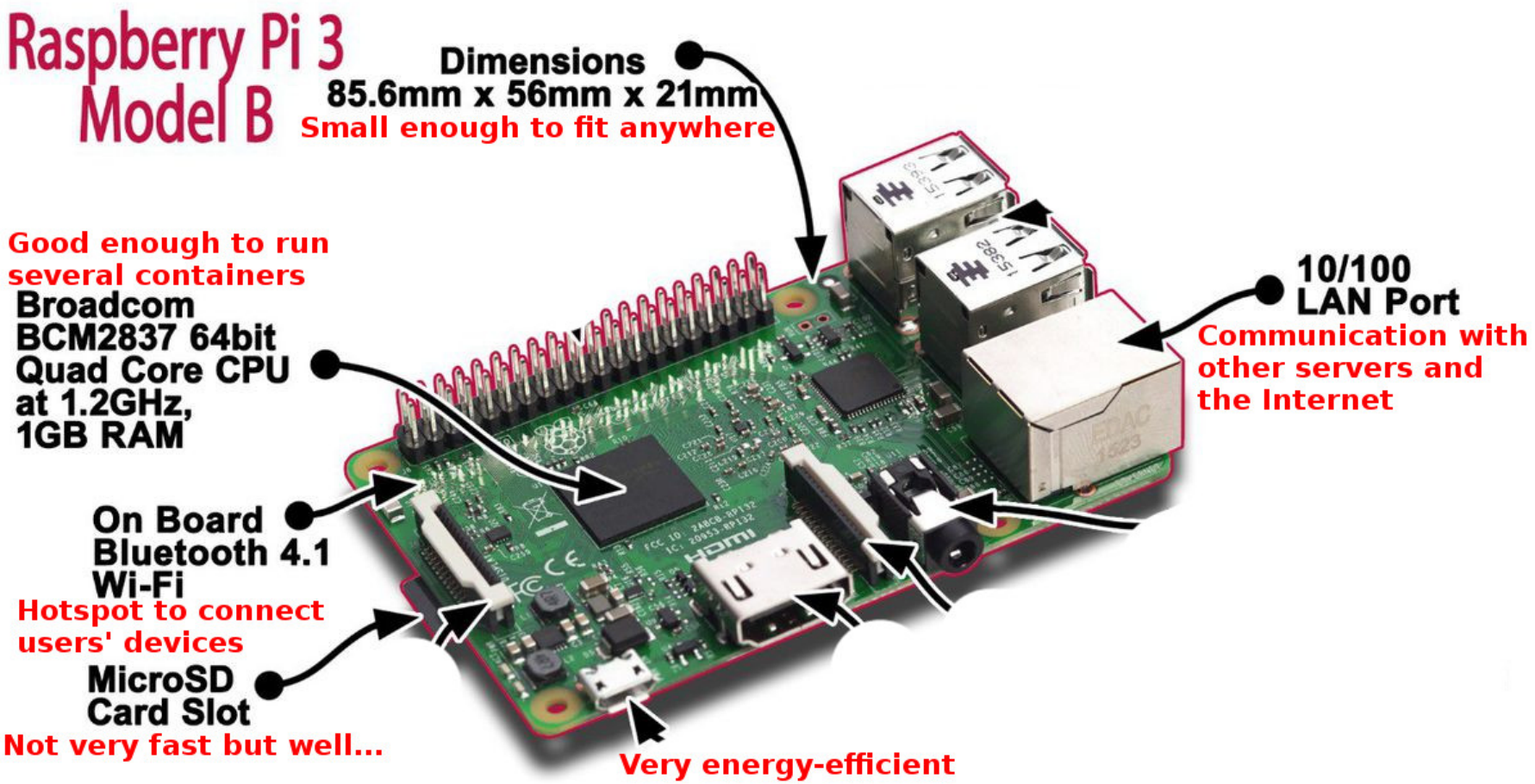
4 Toward proximity-aware Kubernetes

5 Conclusion

Our powerful server machines

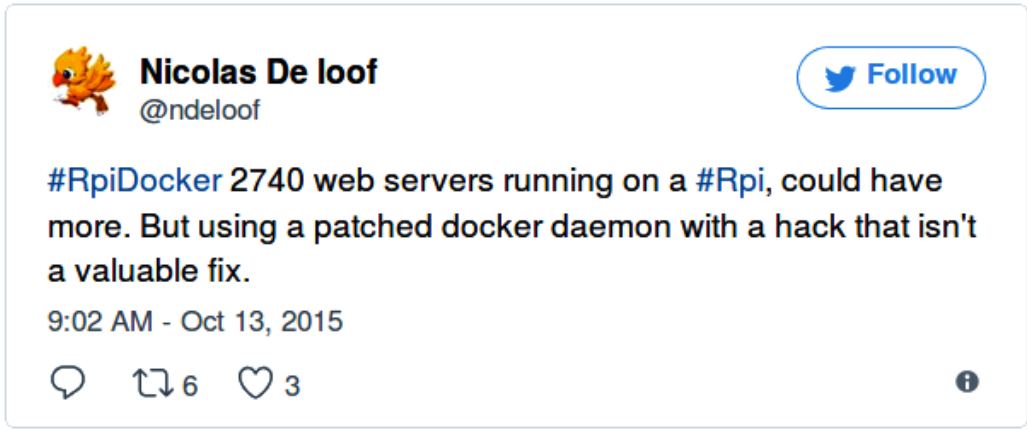


Our powerful server machines



A big congratulations to Damien Duportal, Nicolas de Loof and Yoann Dubreuil on running 2500 web servers in Docker containers on a single Raspberry Pi 2!

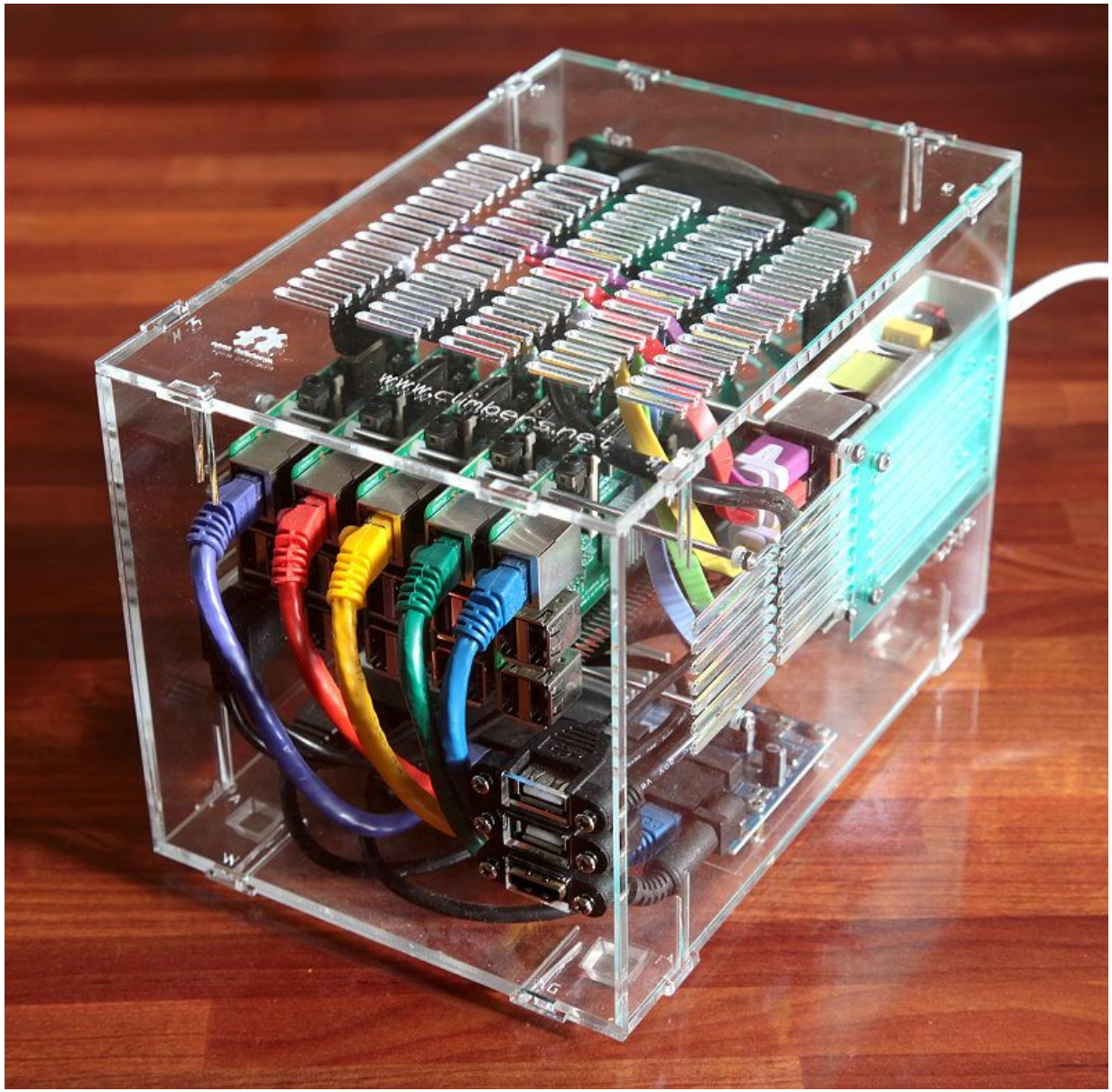
Damien, Nicolas and Yoann each win a complimentary pass to DockerCon EU 2015 and speaking slot during the conference to demo how they accomplished this.



Nicolas de Loof explains more:

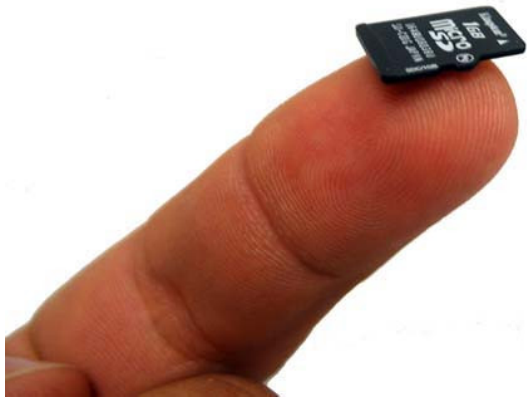
> *To be honest this 2740 web servers score is not strictly eligible for this challenge. We built a custom Docker daemon with some hack-ish `debug.setMaxThread` to bypass Go 1.4 limitation, which didn't allow us to run more than 10000 threads for 2500 containers. A valuable fix would require understanding why the daemon needs so many threads (4 per container – maybe using Non Blocking IO multiplexing?). This is definitely not something I can contribute with my "hello-world" level Golang skills* 🙂
>
> *It was really interesting for us to find such a limit, understand it and get a bit further to discover the next one! Adhering to the challenge rules, we ran 2499 containers. Nobody could run more with the current Docker design, so this opens up an interesting discussion of future Docker improvements!*
>
> *Regardless, 2500 web servers on a RPi, is such a crazy metric!*

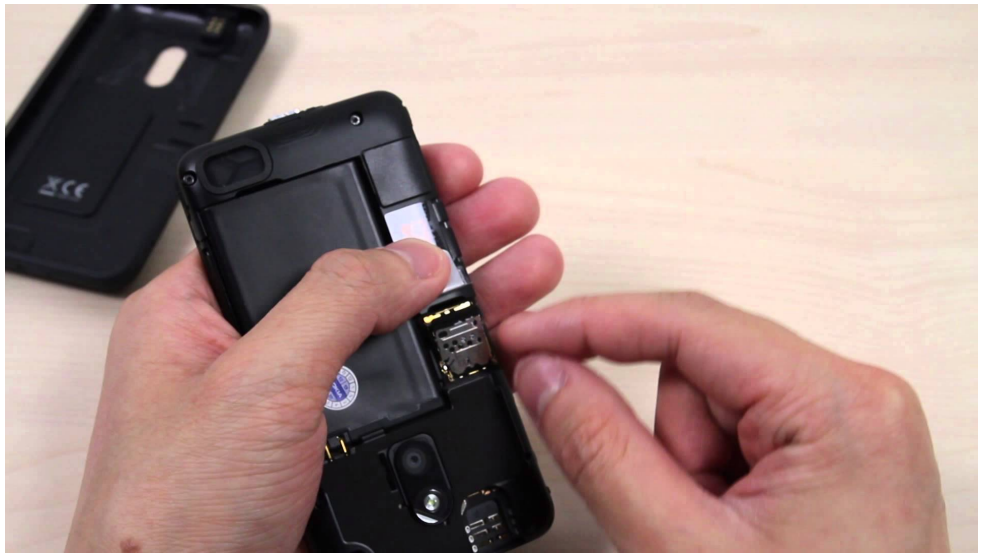
https://blog.docker.com/2015/10/raspberry-pi-dockercon-challenge-winner/



https://climbers.net/sbc/diy-raspberry-pi-3-cluster-2017/

Getting the best out of our RPIs: file system tuning





- SD Cards were designed to store photos, videos, etc.
  - Small number of large files

  $\Rightarrow$ Large physical block sizes

- But Linux file systems expect small blocks by default
  - ☞ Tune the file system for larger blocks
  - ☞ Align partitions on the erase block boundaries (multiples of 4 MB)

http://3gfp.com/wp/2014/07/formatting-sd-cards-for-speed-and-lifetime/

Getting the best out of our RPIs: swappiness tuning

Swappiness is the kernel parameter that defines how much (and how often) your Linux kernel will copy RAM contents to swap. This parameter's default value is "60" and it can take anything from "0" to "100". The higher the value of the swappiness parameter, the more aggressively your kernel will swap.

https://www.howtoforge.com/tutorial/linux-swappiness/

What's the right swappiness value for us?

Getting the best out of our RPIs: swappiness tuning

Swappiness is the kernel parameter that defines how much (and how often) your Linux kernel will copy RAM contents to swap. This parameter's default value is "60" and it can take anything from "0" to "100". The higher the value of the swappiness parameter, the more aggressively your kernel will swap.

https://www.howtoforge.com/tutorial/linux-swappiness/

What's the right swappiness value for us?

- RPI-specific linux distributions set it to a very low value (e.g., 1)
- We changed it to the maximum value: 100

Getting the best out of our RPIs: swappiness tuning

Swappiness is the kernel parameter that defines how much (and how often) your Linux kernel will copy RAM contents to swap. This parameter's default value is "60" and it can take anything from "0" to "100". The higher the value of the swappiness parameter, the more aggressively your kernel will swap.

https://www.howtoforge.com/tutorial/linux-swappiness/

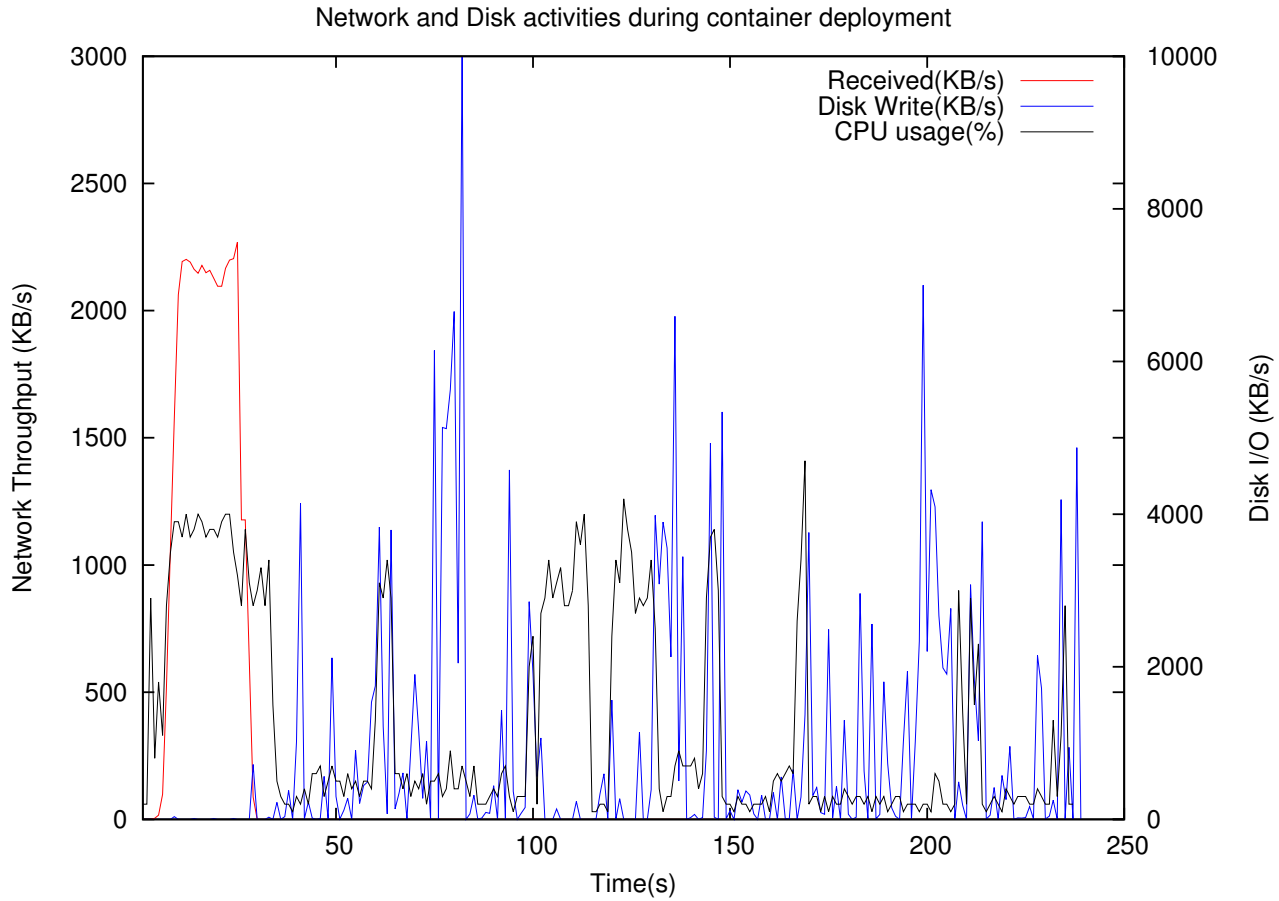
What's the right swappiness value for us?

- RPI-specific linux distributions set it to a very low value (e.g., 1)
- We changed it to the maximum value: 100
  - If your machine is going to swap anyway (because of small memory), then better do it out of the critical path
  - Also the file system cache makes use of all the unused RAM
  - Actually this results in less I/O...

https://hal.inria.fr/hal-01446483v1

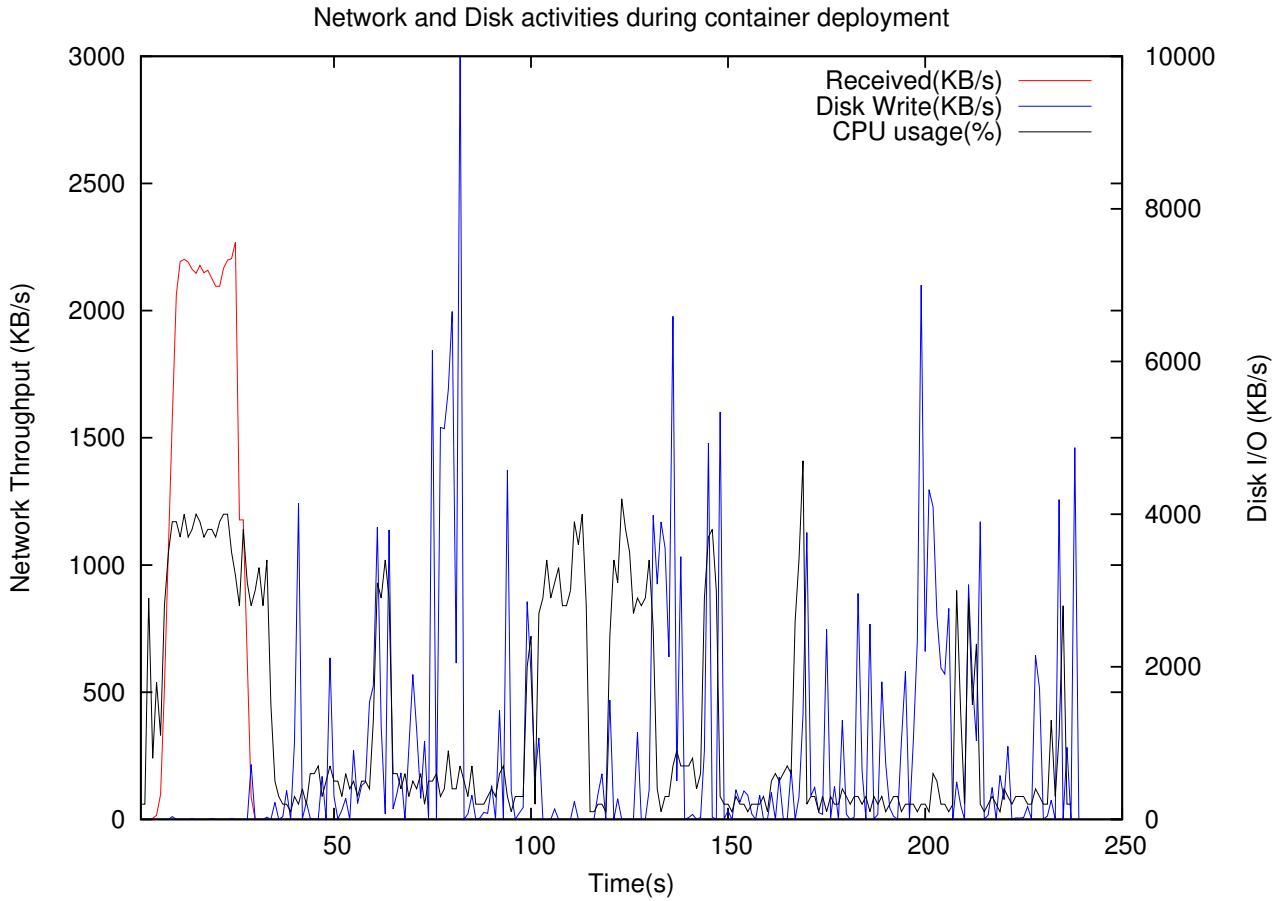
Container deployment in a RPI

- Let's deploy a very simple Docker container on the RPI3
  - Standard ubuntu container (∼45 MB) + one extra 51-MB layer



Container deployment in a RPI

- Let's deploy a very simple Docker container on the RPI3
  - Standard ubuntu container (∼45 MB) + one extra 51-MB layer



- Total deployment time: **about 4 minutes!!!**
  - ☞ Can we make that faster?

How Docker deploys a new container image

- A container image is composed of multiple layers
  - Each layer complements or modifies the previous one
  - Layer 0 with the base OS, layer 1 with extra config files, layer 2 with the necessary middleware, layer 3 with the application, layer 4 with a quick-and-dirty fix for some wrong config file, etc.
- What takes a lot of time is bringing the image from the external repository to the raspberry pi
  - Starting the container itself is much faster
- Deployment process:
  1. Download all layers simultaneously
  2. Decompress and extract each layer to disk sequentially
  3. Start the container

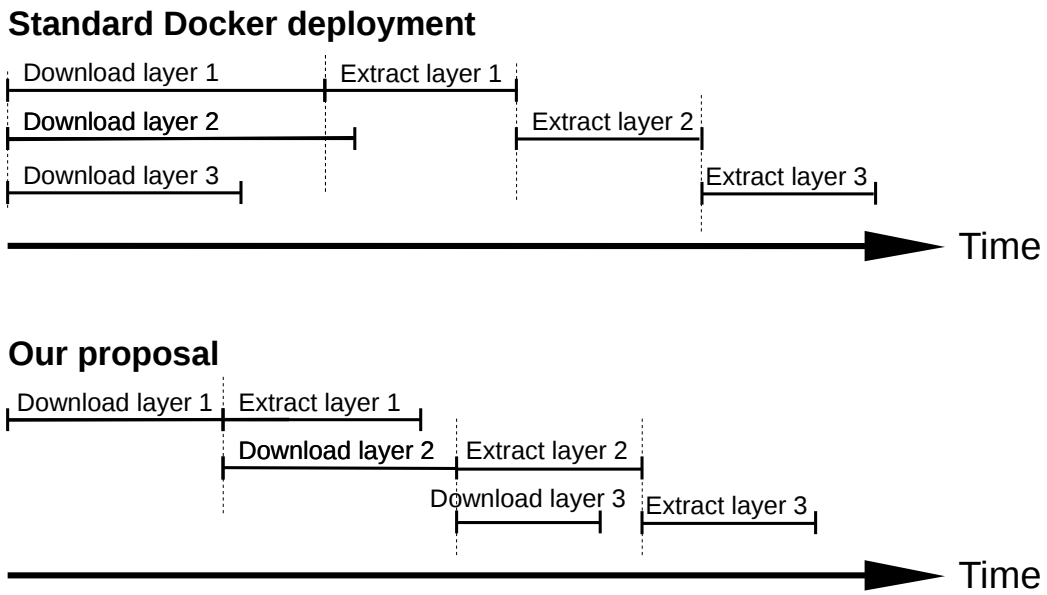
How Docker deploys a new container image

- A container image is composed of multiple layers
  - Each layer complements or modifies the previous one
  - Layer 0 with the base OS, layer 1 with extra config files, layer 2 with the necessary middleware, layer 3 with the application, layer 4 with a quick-and-dirty fix for some wrong config file, etc.
- What takes a lot of time is bringing the image from the external repository to the raspberry pi
  - Starting the container itself is much faster
- Deployment process:
  1. Download all layers simultaneously
  2. Decompress and extract each layer to disk sequentially
  3. Start the container

Question: What's wrong with this?

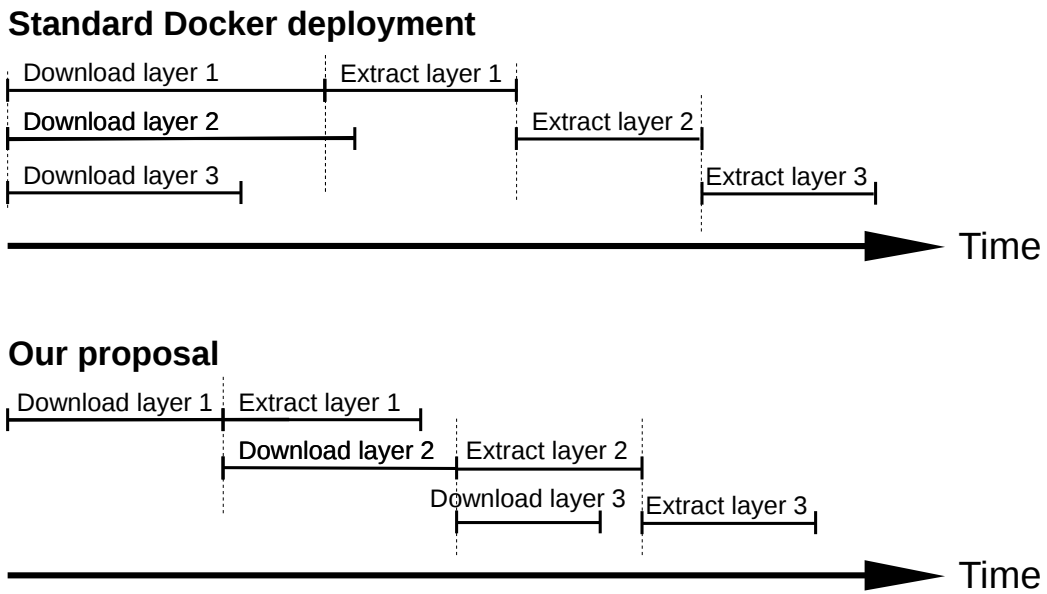
Sequential layer download

- Observation: parallel downloading delays the time when the first download has completed
- Idea: let's download layers sequentially. This should allow the deployment process to use the bandwidth and disk I/O simultaneously



Sequential layer download

- Observation: parallel downloading delays the time when the first download has completed
- Idea: let's download layers sequentially. This should allow the deployment process to use the bandwidth and disk I/O simultaneously



- Performance gains:
  - $\sim 3-6\%$ on fast networks
  - $\sim 6-12\%$ on slow (256 kbps) networks
  - Best gains with several big layers + slow networks

Let's speed up the decompression part

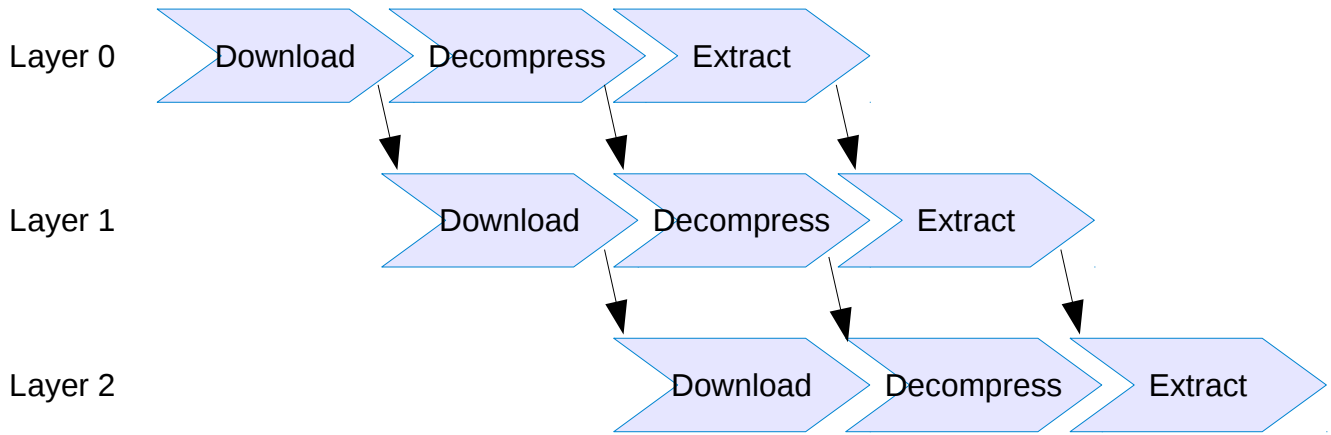
- Image layers are always delivered in compressed form, usually with gzip
  - Docker calls gunzip to decompress the images
  - But gunzip is single-threaded while RPis have 4 CPU cores!
- Let's use a multithreaded gunzip implementation instead

Let's speed up the decompression part

- Image layers are always delivered in compressed form, usually with gzip
  - Docker calls gunzip to decompress the images
  - But gunzip is single-threaded while RPis have 4 CPU cores!
- Let's use a multithreaded gunzip implementation instead
- Performance improvement: $\sim$15 – 18% on the entire deployment process
  - Much more if we look just at the decompression part of the process...

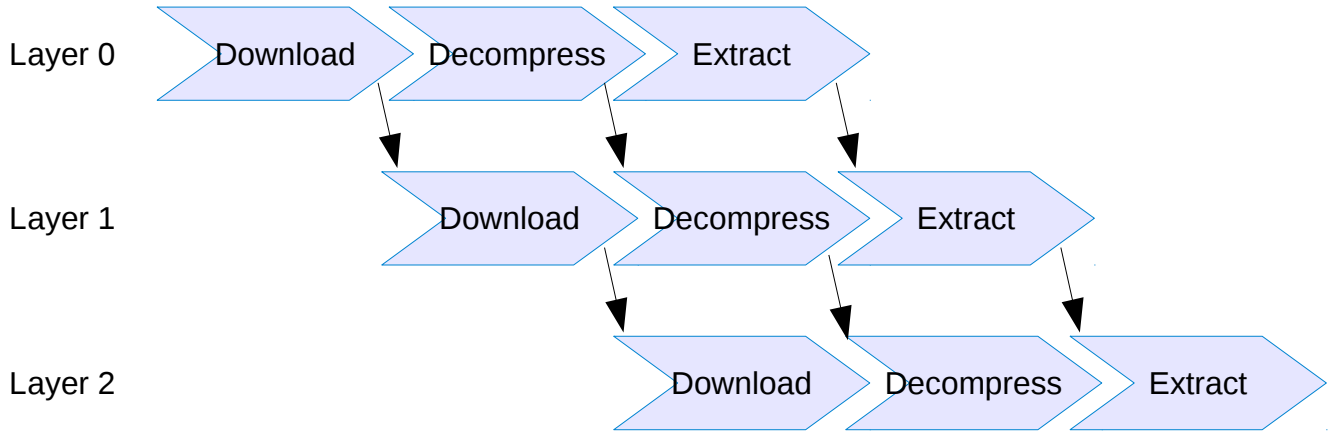
Let's pipeline the download, decompression and extraction processes

- Idea: let's split the download/decompress/extract process into three threads per layer
  - And pipeline the three parts: start the decompression and extract processes as soon as the first bytes are downloaded
  - ☞ Intersting thread synchronization exercise... ☺



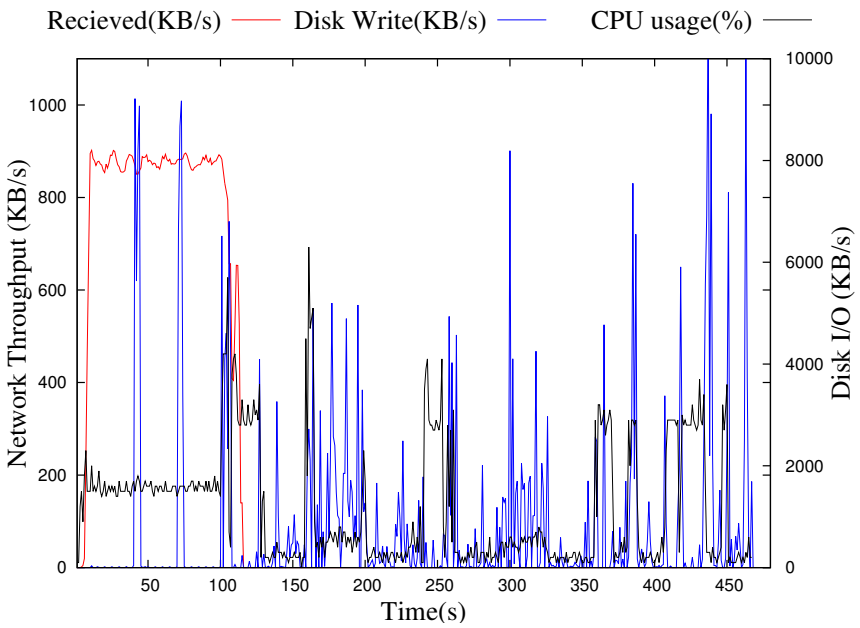
Let's pipeline the download, decompression and extraction processes

- Idea: let's split the download/decompress/extract process into three threads per layer
  - And pipeline the three parts: start the decompression and extract processes as soon as the first bytes are downloaded
  - ☞ Intersting thread synchronization exercise... ☺

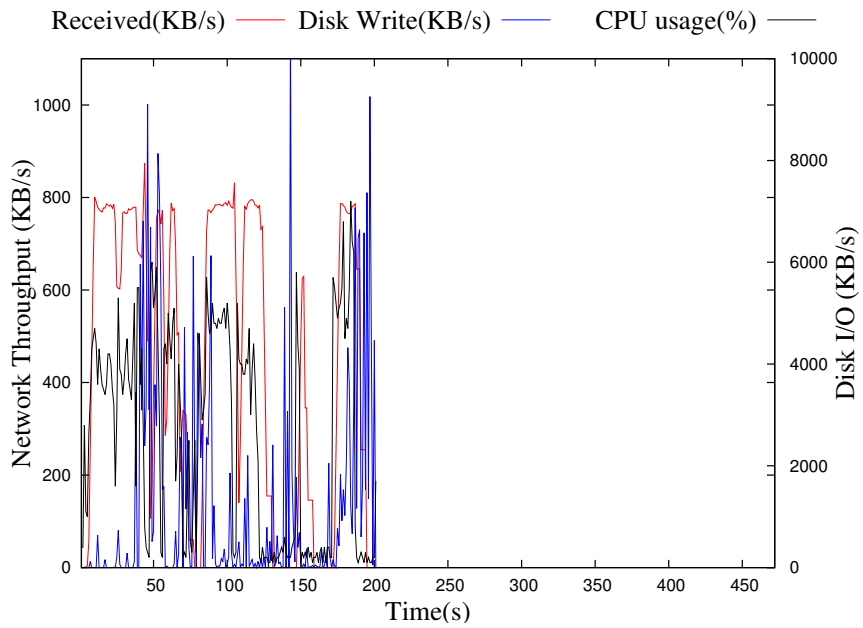


- Performance improvement: $\sim 20-55\%$

Let's combine all three techniques



Before



After

- Combined performance gain: 17 – 73%
  - Lowest gains obtained with low (256 kbps) networks. The network is the bottleneck, not many local optimizations are possible
  - When using a "decent" network connection: $\sim$ **50 – 73% gains**

What about "real" server machines?

- We reproduced the same experiments on a "real" server machine
  - 2x 10-core CPU (Intel Xeon E5-2660v2)
  - 128 GB RAM
  - 2x 10 Gbps network
- Performance gains: 29-36%
  - Most effective for images with multiple large layers

These optimizations are also relevant
in cloud-like environments

Docker Container Deployment in Fog Computing Infrastructures. A. Ahmed and G. Pierre. In Proc. IEEE EDGE conference, July 2018.

Table of Contents

1 Introduction

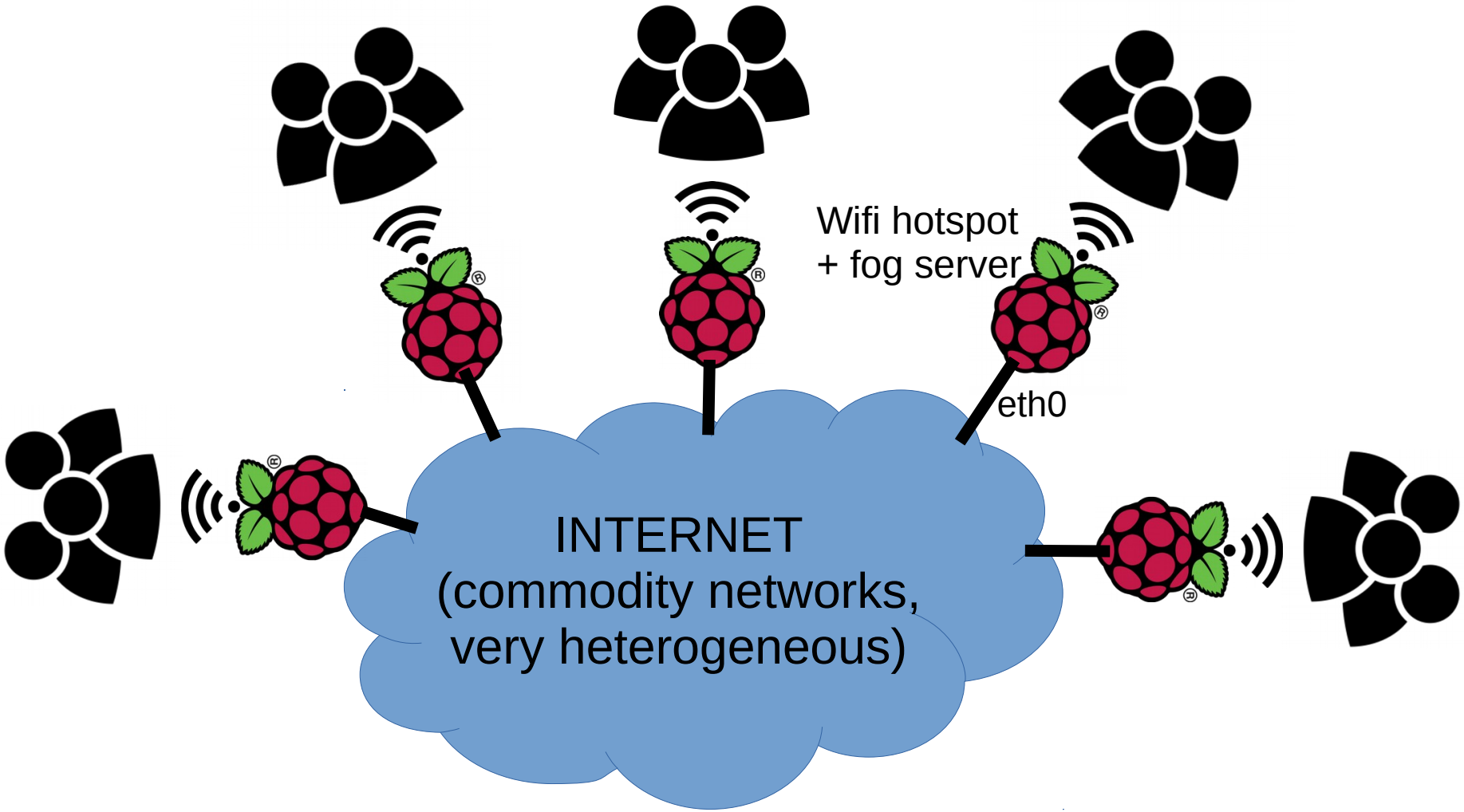
2 Fog computing

3 Making fog servers REALLY small

4 Toward proximity-aware Kubernetes

5 Conclusion

The fog according to us

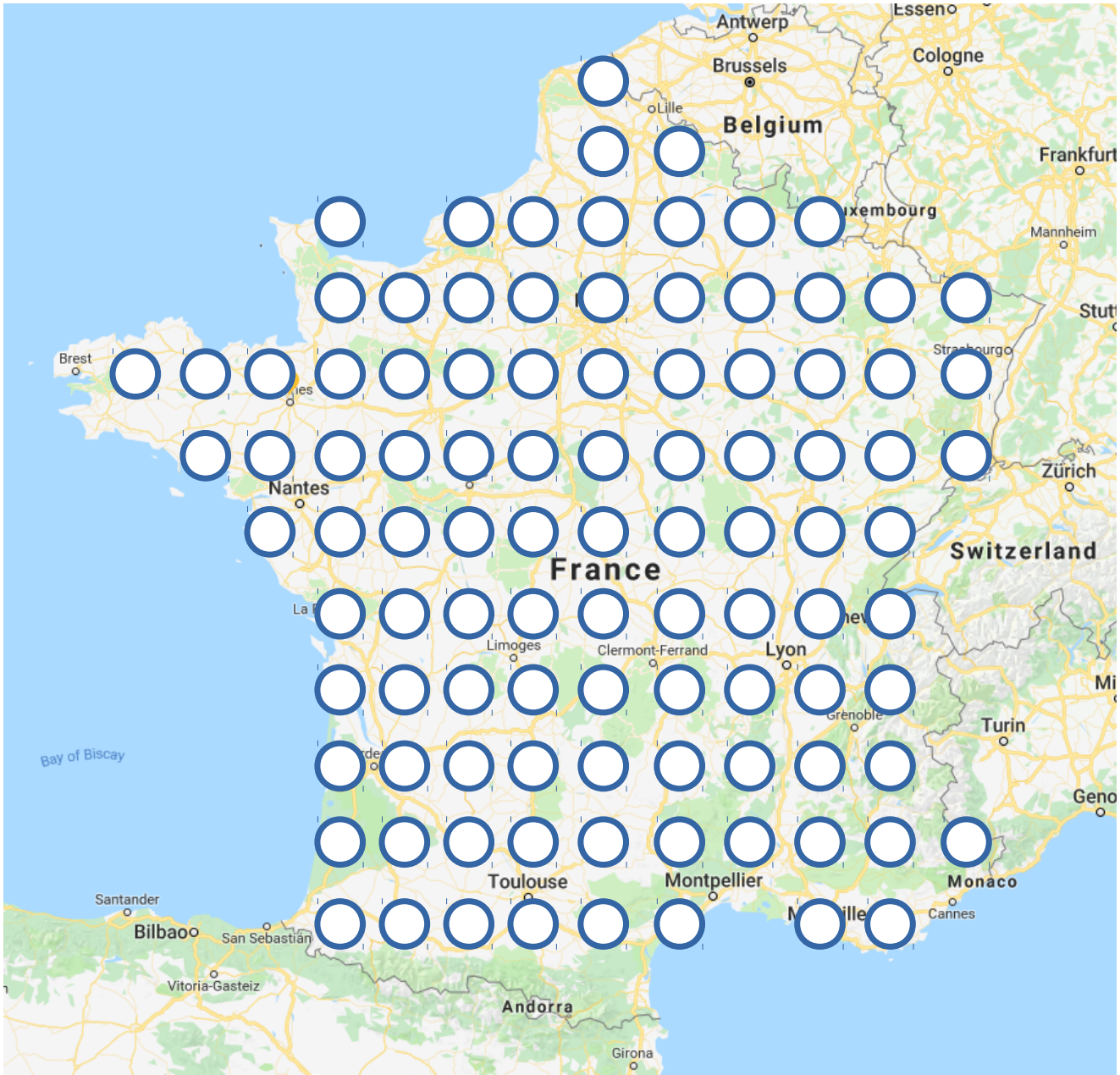


Kubernetes is a great platform...

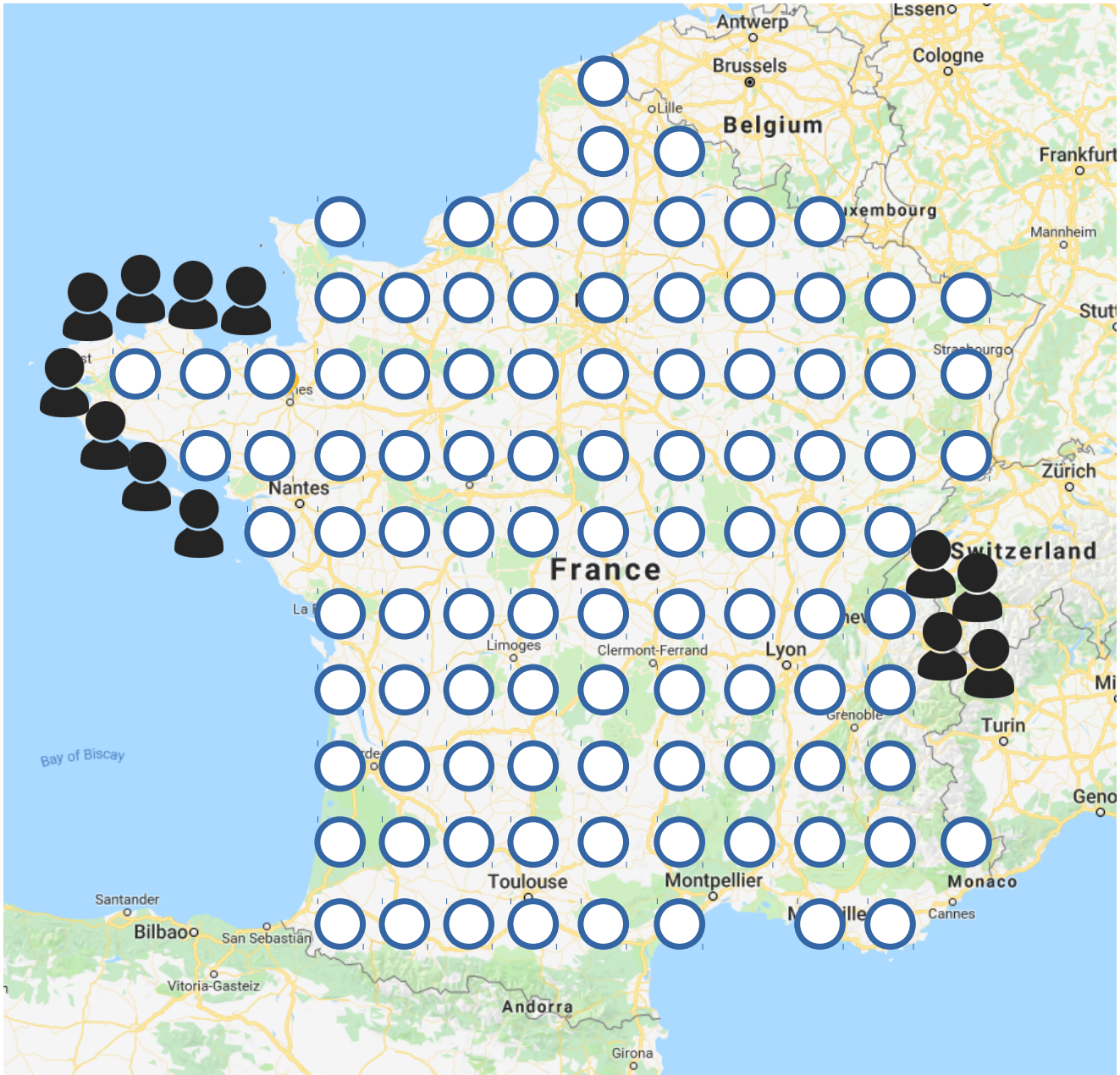
- Container-based ⇒ lightweight, can run on a Raspberry Pi
- Pods and services ⇒ easy to deploy and (re-)scale applications
- Integrated autoscaling ⇒ ready for fluctuating workloads
- Designed around feedback control loops ⇒ simple and robust
- Large community of users and developers ⇒ we can find help whenever necessary ☺

. . . but it is not ready for the fog

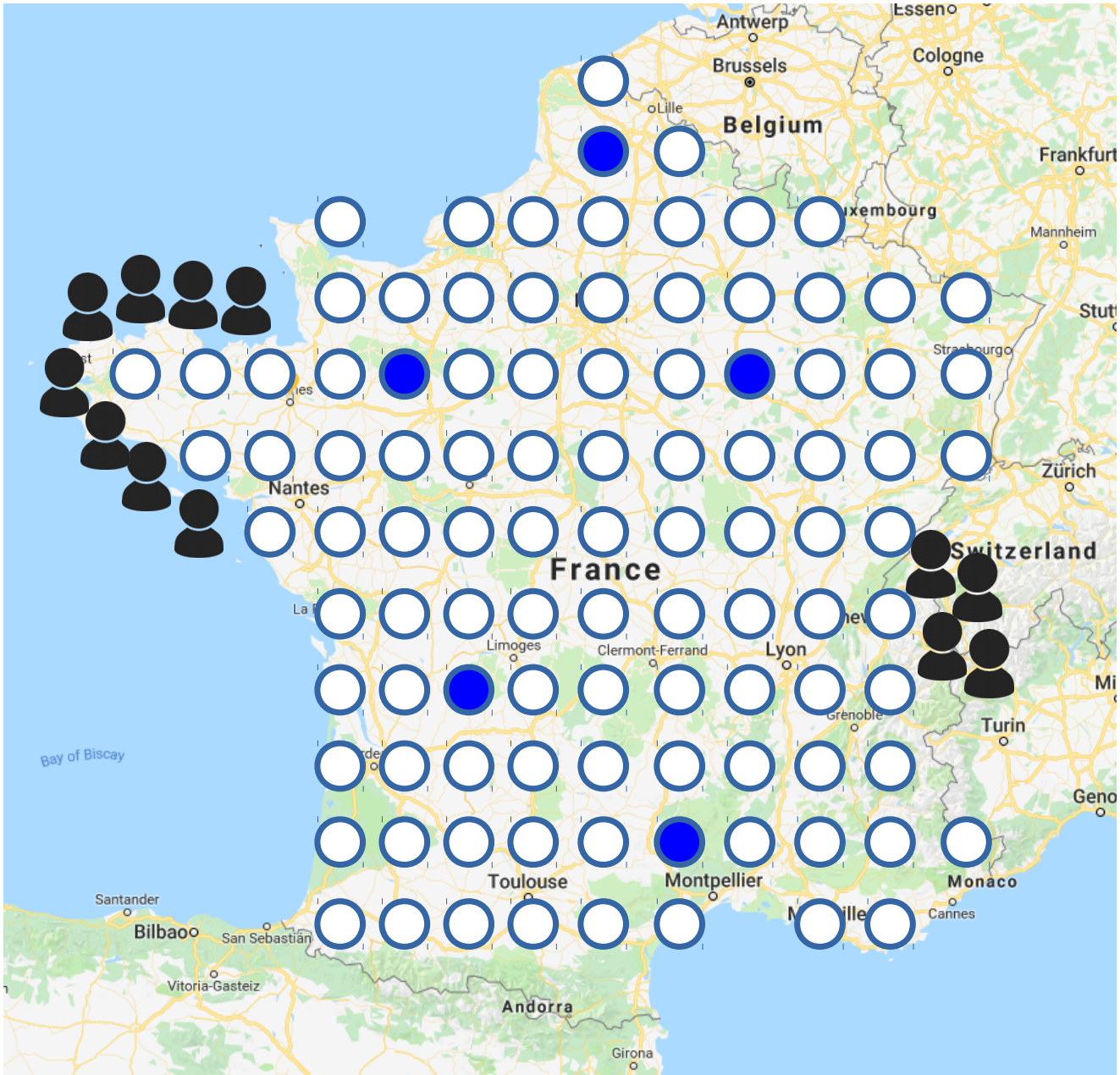
. . . but it is not ready for the fog



. . . but it is not ready for the fog



. . . but it is not ready for the fog

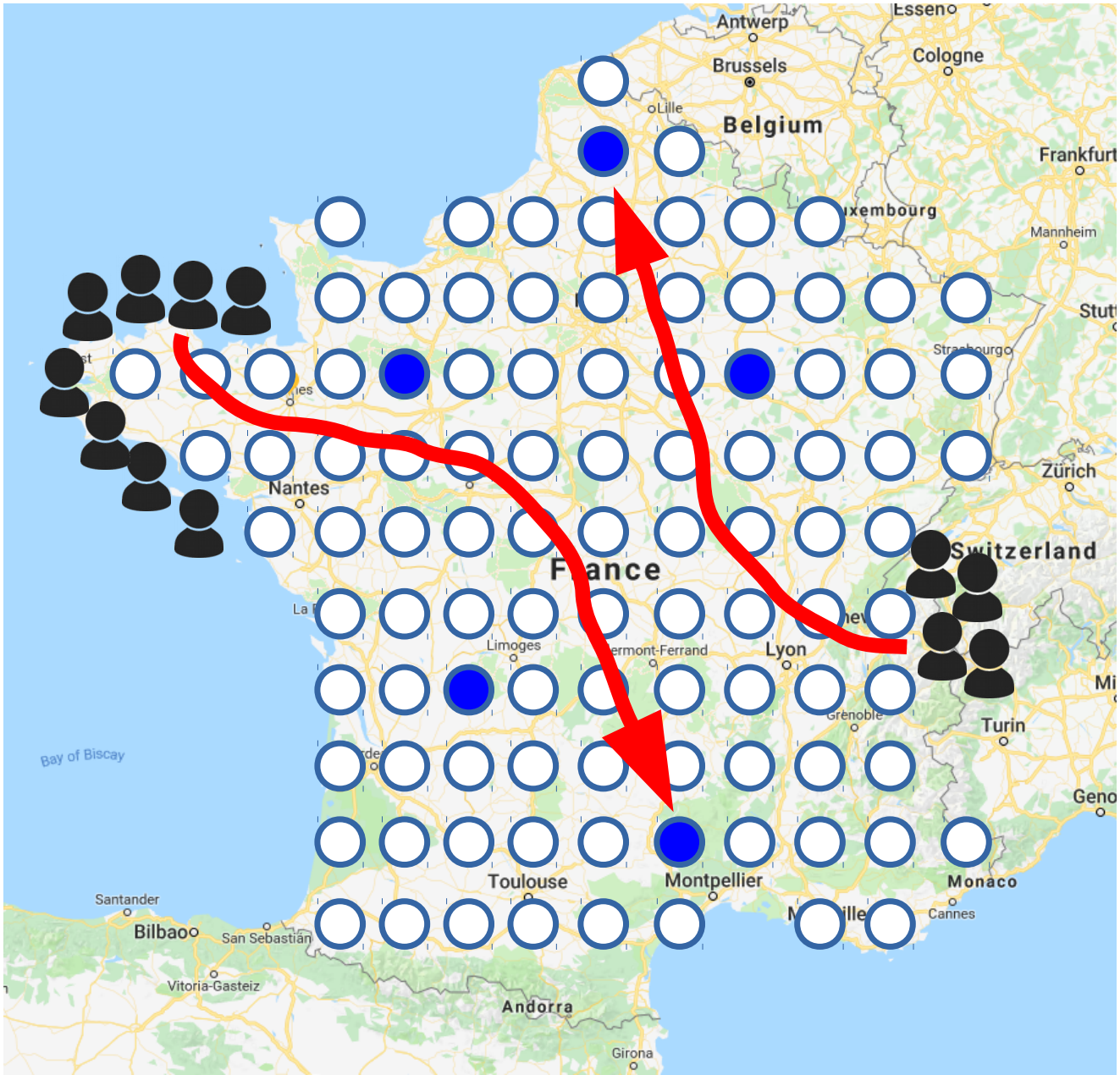


. . . but it is not ready for the fog

Scheduling policies:

- Random
- Round-Robin
- Least loaded node
- Hand-picked
- . . . but no location-aware policy! ☹

. . . but it is not ready for the fog



. . . but it is not ready for the fog

Load-balancing policies:

- Random
- Round-Robin
- Least loaded node
- Hand-picked
- . . . but no location-aware policy! ☹

How Kubernetes routes user requests

- (not necessary in our use cases): expose a public IP address to external users
- Route traffic from any Kubernetes cluster node to one of the pods
  1. The central `kube-proxy` controller detects a change in the set of pods
  2. It requests all cluster nodes to update their routes
  3. Routes are implemented using `iptables`
- By default: requests are randomly distributed according to weights (which also seem random)

Making request redirection location-aware

Let's change kube-proxy:

- Detect a change in the set of pods
- Request each cluster node to update its routes to nearby pods

Where should each node route its traffic?

- ~~To the closest pod according to GPS locations~~
- To the closest pod according to Vivaldi coordinates
- Load-balance to the *n* closest pods according to their load
- ...

We need to deploy additional monitoring functionality

- Vivaldi...


WORK IN PROGRESS

Table of Contents

1 Introduction

2 Fog computing

3 Making fog servers REALLY small

4 Toward proximity-aware Kubernetes

5 Conclusion

Conclusion

- Cloud data centers are very powerful and flexible
  - But not all applications can use them (latency, traffic locality)
- If we evaporate a cloud, then we get a fog
  - Extremely distributed infrastructure: there must be a server node close to every end user
    - Server nodes must be small, cheap, easy to add and replace
    - Server nodes are very far from each other
- This is only the beginning
  - No satisfactory edge/fog platforms are available today (we are not even close)
  - There remains thousands of potential PhD research topics in this domain ☺

www.fogguru.eu

Shameless announcement

http://2018.middleware-conference.org/