Advanced Cloud Infrastructures

From Data Centers to Fog Computing (part 2)
Guillaume Pierre

Master 2 CCS & SIF, 2017
Fog computing is...

A system-level architecture to extend

Compute

Network

Storage

Capability of Cloud to the edge of the IoT network

Source: Cisco
Table of Contents

1  Fog applications taxonomy
2  From centralized clouds to decentralized fogs
3  Container deployment in an edge-cloud server
4  Live container migration
5  Data storage
6  Stream processing for IoT fog applications
7  Conclusion
Computation offloading

- Parts of the application is offloaded out of the mobile device
Low-latency mediation between multiple users

(example application)
IoT analytics

Data processing is intelligently partitioned between fog nodes co-located with IoT devices and the cloud.

This enables real-time tracking, anomaly detection, and collection of insights from data captured over long intervals of time.

which-iot-applications-work-best-with-fog-computing.html
We can derive system-level requirements for a fog computing platform from the reference applications:

1. A complete cloud-like service offering
2. Context-aware application/data placement and resource scheduling
3. Fog computing infrastructure management and control
4. Efficient software deployment
5. Efficient container live-migration
6. Efficient data storage
7. Software engineering methodologies and middlewares suitable for fog computing scenarios
8. And many other open research topics...
Table of Contents

1. Fog applications taxonomy

2. From centralized clouds to decentralized fogs

3. Container deployment in an edge-cloud server

4. Live container migration

5. Data storage

6. Stream processing for IoT fog applications

7. Conclusion
OpenStack overview

https://hal.inria.fr/hal-01273427
Core services of OpenStack

- **KeyStone**: Administrative tools, Information manager, Accounting/Auditing
- **Nova**: Compute manager
- **Glance**: Storage manager
- **Horizon**: Network manager
- **Neutron**: Network manager
- **Swift**: Network manager
OpenStack Nova’s architecture

 Diagram:

- Nova Network
- Nova Compute
- Nova Conductor
- Nova Scheduler

Interconnections:

- Non-Relational
  - Key/Value DB
  - db.api
  - ROME

- Relational
  - db.api
  - SQLAlchemy
  - MySQL DB
Replacing MySQL with Redis is not as easy as it sounds:
- Distributed locking system
- Support for atomic updates
- Create secondary index to support join queries
- ...
Remaining challenges

- The internal OpenStack message bus is based on RabbitMQ
  - Simple, robust, perfect in a single data center environment
  - Centralized architecture based on a single broker node
  - Every message queue is mirrored in every node

- Problem: RabbitMQ is very sensitive to network latency!
  ⇒ Replace RabbitMQ with a P2P message bus (ActiveMQ, ZeroMQ)
Decentralized clouds ≠ fogs

- OpenStack is suitable for highly distributed clouds but not for fogs
  - Lots of heavyweight processes
  - VMs are not suitable for very small machines
  ⇒ Let’s move to container-based technologies
Table of Contents

1 Fog applications taxonomy
2 From centralized clouds to decentralized fogs
3 Container deployment in an edge-cloud server
4 Live container migration
5 Data storage
6 Stream processing for IoT fog applications
7 Conclusion
Using a RPI as a cloud server

- The RPI is quite a powerful machine
  - 4-core ARM CPU
  - GPU
  - Excellent performance/price/energy ratio

But it also faces major performance challenges
- Limited memory (1 GB)
- Slow disk I/O
- Slow network throughput

Traditional cloud systems were not designed for such environments
- The RPI is an extreme case, but researchers using other small machines are facing exactly the same challenges
Using a RPI as a cloud server

- The RPI is quite a powerful machine
  - 4-core ARM CPU
  - GPU
  - Excellent performance/price/energy ratio

- But it also faces major performance challenges
  - Limited memory (1 GB)
  - Slow disk I/O
  - Slow network throughput
Using a RPI as a cloud server

- The RPI is quite a powerful machine
  - 4-core ARM CPU
  - GPU
  - Excellent performance/price/energy ratio

- But it also faces major performance challenges
  - Limited memory (1 GB)
  - Slow disk I/O
  - Slow network throughput

- Traditional cloud systems were not designed for such environments
  - The RPI is an extreme case, but researchers using other “small” machines are facing exactly the same challenges
SD Cards were designed to store photos, videos, etc.
- Small number of large files
  - 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)

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.

What’s the right swappiness value for us?
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.

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
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
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

![Network and Disk activities during container deployment](attachment:image.png)

- 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?
**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.

---

**Standard Docker deployment**

- Download layer 1
- Extract layer 1
- Download layer 2
- Extract layer 2
- Download layer 3
- Extract layer 3

**Our proposal**

- Download layer 1
- Extract layer 1
- Download layer 2
- Extract layer 2
- Download layer 3
- Extract layer 3

---

**Performance gains:**

- \( \sim 36\% \) on fast networks
- \( \sim 612\% \) on slow (256 kbps) networks

---

**Advanced Cloud Infrastructures**

**Container deployment in an edge-cloud server**

---
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:
- ~3–6% on fast networks
- ~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:** ~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
  - Interesting thread synchronization exercise... 😊

![Diagram showing the pipeline of download, decompression, and extraction processes across three layers. Each layer has a download, decompress, and extract 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
  - Interesting thread synchronization exercise... 😊

- Performance improvement: ~20 – 55%
Let’s combine all three techniques

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
1 Fog applications taxonomy
2 From centralized clouds to decentralized fogs
3 Container deployment in an edge-cloud server
4 Live container migration
5 Data storage
6 Stream processing for IoT fog applications
7 Conclusion
Why do we need migration?

1. Resources are fixed but users are mobile
2. Churn
3. Workload fluctuations
4. Consolidation
5. Competing applications
Live migration was initially designed for VMs.

1. **Stage 0: Pre-Migration**
   - Active VM on Host A
   - Alternate physical host may be preselected for migration
   - Block devices mirrored and free resources maintained

2. **Stage 1: Reservation**
   - Initialize a container on the target host

3. **Stage 2: Iterative Pre-copy**
   - Enable shadow paging
   - Copy dirty pages in successive rounds.

4. **Stage 3: Stop and Copy**
   - Suspend VM on host A
   - Generate ARP to redirect traffic to Host B
   - Synchronize all remaining VM state to Host B

5. **Stage 4: Commitment**
   - VM state on Host A is released

6. **Stage 5: Activation**
   - VM starts on Host B
   - Connects to local devices
   - Resumes normal operation

---

https://www.slideshare.net/mhajibaba/

cloud-computing-principles-and-paradigms-5-virtual-machines-provisioning-and-migration-services
Live migration effect (on a running web server)

Effect of Migration on Web Server Transmission Rate

- 1st precopy, 62 secs
- Further iterations
- 9.8 secs
- 694 Mbit/sec
- 165 ms total downtime

512 kb files
100 concurrent clients

Sample over 100 ms
Sample over 500 ms

Presented by Majid Hajibaba
The new LXD container system supports live migration

- Checkpoint/restart based on CRIU-2.0
- Requires a very recent Linux kernel
- Still announced as very experimental...
- ...and the mixed 64-bit (hardware) + 32-bit (OS) configuration of Raspberry PIs is very confusing for CRIU
A possible alternative: DMTCP

- DMTCP (Distributed MultiThreaded CheckPointing) checkpoints processes in user space
  - It also checkpoints open file descriptors, PID, sockets, etc.
  - And it replaces them with a “dummy” at the destination node to avoid name clashes

- No kernel support required!
  - And a container is nothing more than a group of processes...
  - We “just” need support for statically-linked executables and for Linux namespaces

http://dmtcp.sourceforge.net/
Migrating the application’s data

In a cloud environment we can rely on network-attached storage.

But in a fog it is pointless to migrate an application without migrating the data as well.

Presented by Majid Hajibaba
Another alternative: elasticity-based migration

- We expect many fog applications to be elastic
  - Easy support for dynamically adding/removing nodes

- Can migration be just a special case of elasticity?
  - Start with $X$ pods
  - Scale up to $X + 1$ pods, make sure the new pod is created at the right location
  - Scale down to $X$ pods, make sure to choose which pod gets stopped

- The hard parts:
  - When can we issue the scale-down operation? Liveness probes
  - What about the application state?
Data storage in fog platforms

- Where should we store fog user’s data?
  - In the fog node where data were produced?
  - Replicate in nearby nodes as well?
  - Replicate in all nodes? In all sites?
  ✏️ At least we need to make data available from everywhere (with decent performance)

- IPFS: the InterPlanetary File System
  - Actually this is a data store rather than a file system: objects are immutable
  - Objects are stored locally and registered in a DHT to allow remote access
  - But IPFS is not aware of the fog topology
    ⭐ If an object is available in another node of the same site, IPFS will not prefer this location rather than another site
Each node first checks if an object is present in the local NAS. Otherwise, it locates it using the DHT, and copies it locally for further usage.

https://hal.archives-ouvertes.fr/hal-01483702
Accessing local objects

(a) – Write

(b) – Read
Accessing remote objects

(a) – First read

(b) – Second read
# Fog applications taxonomy

# From centralized clouds to decentralized fogs

# Container deployment in an edge-cloud server

# Live container migration

# Data storage

# Stream processing for IoT fog applications

# Conclusion
Fig. 1. Examples of typical Fog data streaming Applications. (a) IoT stream query and analytics, (b) Real-time event monitoring, (c) Networked Control Systems (NCS) for Industrial automation, (d) Real-time Mobile Crowdsensing (MCS).

https://arxiv.org/abs/1705.05988
Can we use stream processing middlewares in the fog?

**The data analytics community builds interesting stream processing platforms**

- Storm, Spark, Flink etc.
- **Idea:**
  - Define processing as a workflow of operators
  - Let data “flow” through the operators
  - Each operator keeps a window of data items
Stream platforms were designed for big data processing

Problem: stream platforms were designed for computer clusters
- Lots of data to be processed
- Lots of resources
- Single location

Challenges for adapting stream processing to fogs

- **Data source/sink placement**
  - According to the location(s) where data are produced and consumed
  - Some stream processing systems (e.g., Spark) require a shared file system between all nodes

- **Intermediate operator placement**
  - To optimize metrics such as processing latency, requested resources, long-distance data volume, etc.

- **Operator elasticity**
  - Can we rescale stream processing operators dynamically?
  - Easy if we accept to drop data, harder otherwise

- **Resource management**
  - Can operators use less than 1 full core each?
1. Fog applications taxonomy
2. From centralized clouds to decentralized fogs
3. Container deployment in an edge-cloud server
4. Live container migration
5. Data storage
6. Stream processing for IoT fog applications
7. 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 😊
Shameless announcement

The FogGuru European project is starting soon on similar issues: University of Rennes 1, TU-Berlin, Elastisys (Umeå, Sweden), U-Hopper (Trento, Italy), Las Naves (Valencia, Spain)

We are looking for 8 ambitious and talented PhD students:

- All positions co-supervised by one academic + one industrial supervisor
- Extensive doctoral training program (scientific, technical, soft-skills, innovation & entrepreneurship)
- International mobility
- 7 months at Las Naves to deploy technologies in the city center of Valencia & get feedback from real end users

www.fogguru.eu