RT-Cloud and real-time containers

Real-Time Industrial Systems

Marcello Cinque
Marco Barletta

Roadmap

- Real-time cloud, IoT, Industry 4.0
- Containers (ideas and technologies)
- Orchestration and DevOps and microservices
- Real-time containers: previous solutions
- RT-containers over Xenomai
- References:
  - The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers, Allison Randal
  - A time-predictable fog-integrated cloud framework: One step forward in the deployment of a smart factory, Faragardi et al.
  - Challenges in real-time virtualization and predictable cloud computing, Marisol García-Valls and Tommaso Cucinotta and Chenyang Lu
  - Real-time cloud computing, C. Lu , http://www.cse.wustl.edu/~lu/

Ok, let’s stop with the

Real-Time Cloud

Motivations

Real Time Cloud

- Brand new **requirements**: IoT and Industry 4.0
  - Timeliness
  - Availability
  - Security
- Example: IoT sensors giving feedbacks for industrial controllers:
  - A multi-layered MCS, real-time and reliability requirements
  - Edge devices hosting different loads
    - E.g., real-time supervision, control and predictive maintenance
- Overcome centralized cloud **drawbacks:**
  - unpredictability of the network
  - high response times **-> thoughput and utilization over latency**

	at-least-once	best-effort
milliseconds	e.g., Emergency response	e.g., Real-time monitoring
hours	e.g., Predictive maintenance	

Source: C. Lu, "Real-time cloud computing", http://www.cse.wustl.edu/~lu/

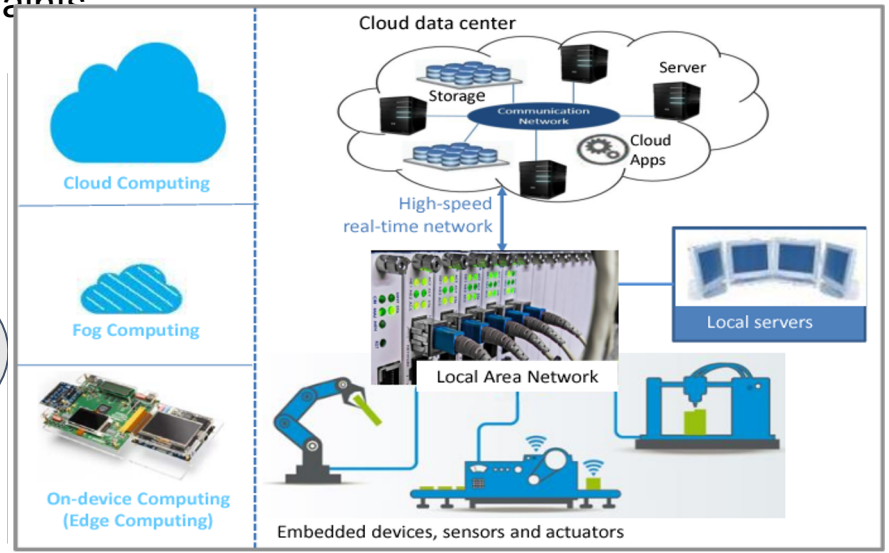
Real Time Cloud

- High number of **nodes**, exponential increase of **data volume**
- Retain part of the workload on **local resources**: fog and edge cloud

  Low to medium loads with timeliness constraints

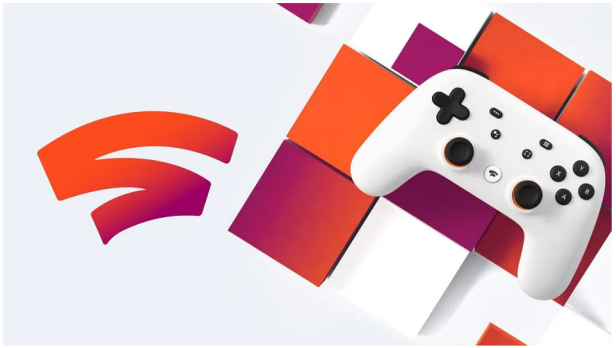
  Less data traveling “abroad”
- Keep cloud model **advantages**

- Flexible deployment
- Scalability
- Small footprint
- Isolation



Real Time Cloud

- Useful also for **centralized** cloud
  - Think about services with less narrow timing requirements
  - Services with **soft real time** requirements…
  - Hard real time systems are not the only real-time systems!
  - Think about Stadia!
- However the problem is in the **path** to the cloud!
- Not the scope of the course

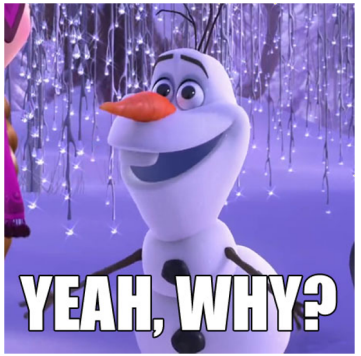


Containers

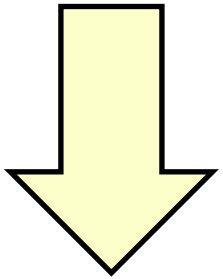
Motivations, advantages, technologies

Containers - intro

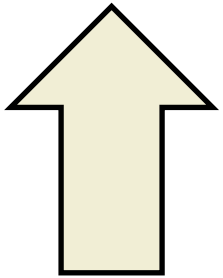
- Used in a lot of **cloud environments**
- More than **12,000 companies** that use Docker since its creation (2015)
- In the organizations with an approximate of **1,000 hosts**, nearly 50% have adopted Docker.
- The **top five** companies using Docker are JPMorgan Chase, ThoughtWorks, Inc., Docker, Inc., Neudesic, and SLALOM, LLC (https://medium.com/@tao_66792/interesting-facts-companies-and-the-use-of-docker-948baa8cf309)
- Netflix relies part of his greatness upon containers, since they help **microservices** architecture (https://www.slideshare.net/aspyker/reinvent-2016-container-scheduling-execution-and-aws-integration)



Containers vs VMs



No OS replication
Faster startup
Easier deploy
Less overhead and footprint
Greater scalability

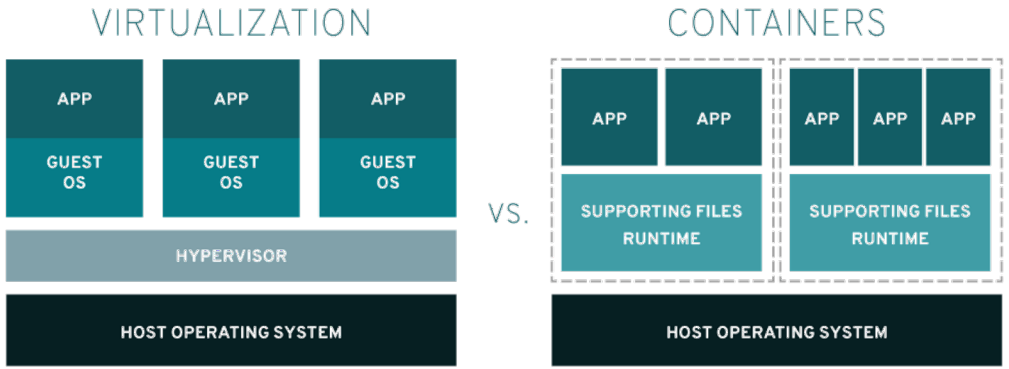


No chance to emulate hardware
Shared kernel
Security implications
No chance to run a different OS
Require a fully fledged kernel

What is a container?

- **Sandboxing** technology…
  … not quite “virtualization”
- **Kernel mechanisms** to enforce isolation

Naming Time
Isolation
Fault Space



(https://www.redhat.com/it/topics/containers/containers-vs-vms)

- **No hardware virtualization**
  some of you may be fooled by Windows, damn Windows…
- One **single underlying OS**!
- **Think** about consequences...

Ok, but still...what is a container?

- Breaking down a container is a **set of processes** and **libraries isolated** from the rest of the machine
- So… why are compared to VMs?
- The underlying **idea** is the same!
  - Applications can behave as they were **alone** on the machine
  - No problem on dependencies, no problems (or at least we try) about resource contention
- How are isolated? It depends… generally thanks to
  - Linux **cgroups**
  - Linux **namespaces**

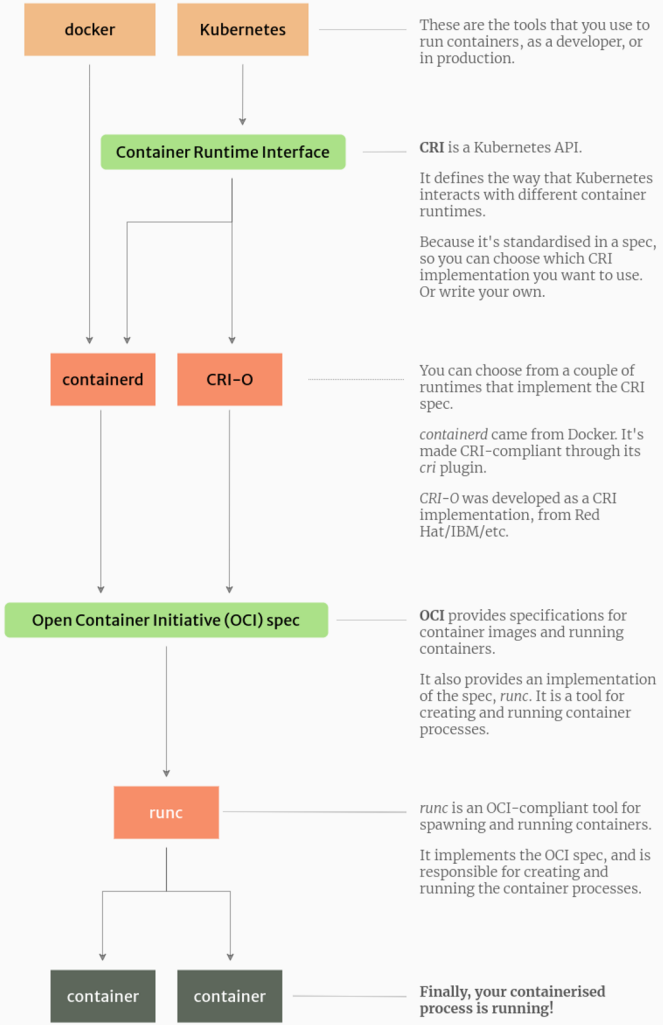
Container technologies

- Container **does not mean** 
- Container is an **idea**, implemented in different ways
- E.g. LXC/LXD, OpenVZ, Docker, Solaris containers
  - LXD is like fully fledged system snapshot
  - Docker is more like sandboxed stateless applications
- But also similar concepts such as **microVMs**
  - RunX, Firecracker, NEMU …
    - ...can be seen as containers, but are VMs
- Docker **features**:
  - Swarm, stack: deployment and node clustering/scheduling
  - Compose: easy deploy with a single description file
  - Volumes: mapping between underlying fs and containers, for persistence
  - Software defined networking: overlay, IPVLAN, MACVLAN, bridge (default), host
  - Services: monitoring and reaction policy

Container technologies - neighbourhood

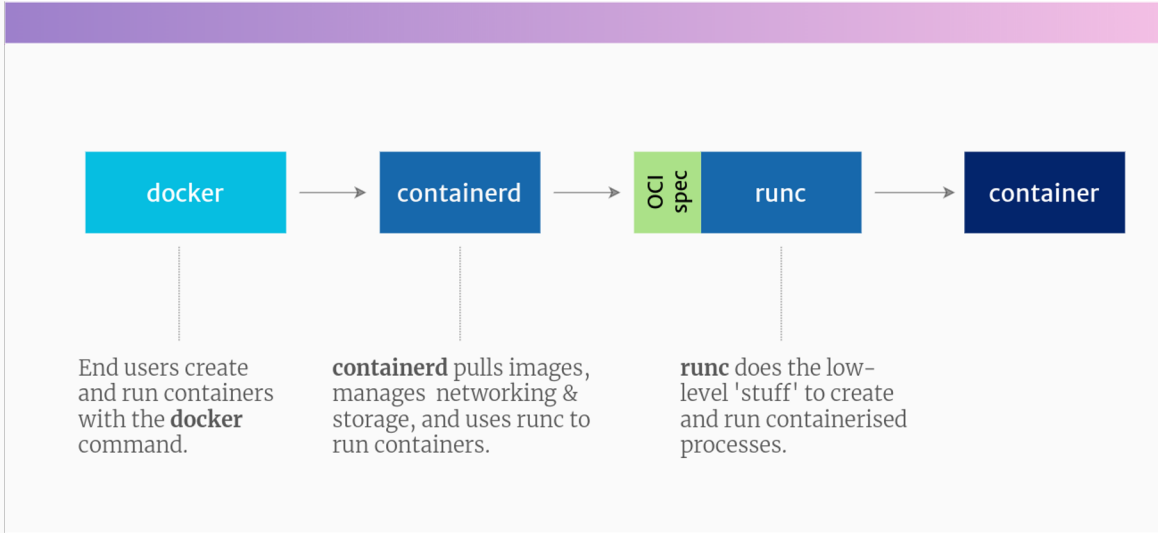
- Still, key technologies **around** containers
- Kubernetes, CoreOS, BalenaOS, Google Container Engine, AmazonECS, Mesos and so on...
- They allow an **easy deploy** with a lot of features:
  - Migration, replication, fault tolerance, clustering, load balancing, monitoring and the list could continue…
- Kubernetes (K8s) reigns widely....
  - ... versions created tailored to run on **embedded** devices
  - e.g. K3s, microK8s
- The need for orchestration is raising in **IoT**…
- **Unified** embedded deployment with cloud for CI/CD

(https://catalog.us-east-1.prod.workshops.aws/v2/workshops/12f31c93-5926-4477-996c-d47f4524905d/en-US)



- When things become spread out, they need to be **standardized…**
- Several technologies share the API

(https://www.tutorialworks.com/difference-docker-containerd-runc-crio-oci/)



Containers - nerd details

Acknowledgement: Luigi De Simone

- Let's go back to cgroups and namespaces:
  - Namespaces create a **virtual environment**
  - Cgroups limit **resource usage**

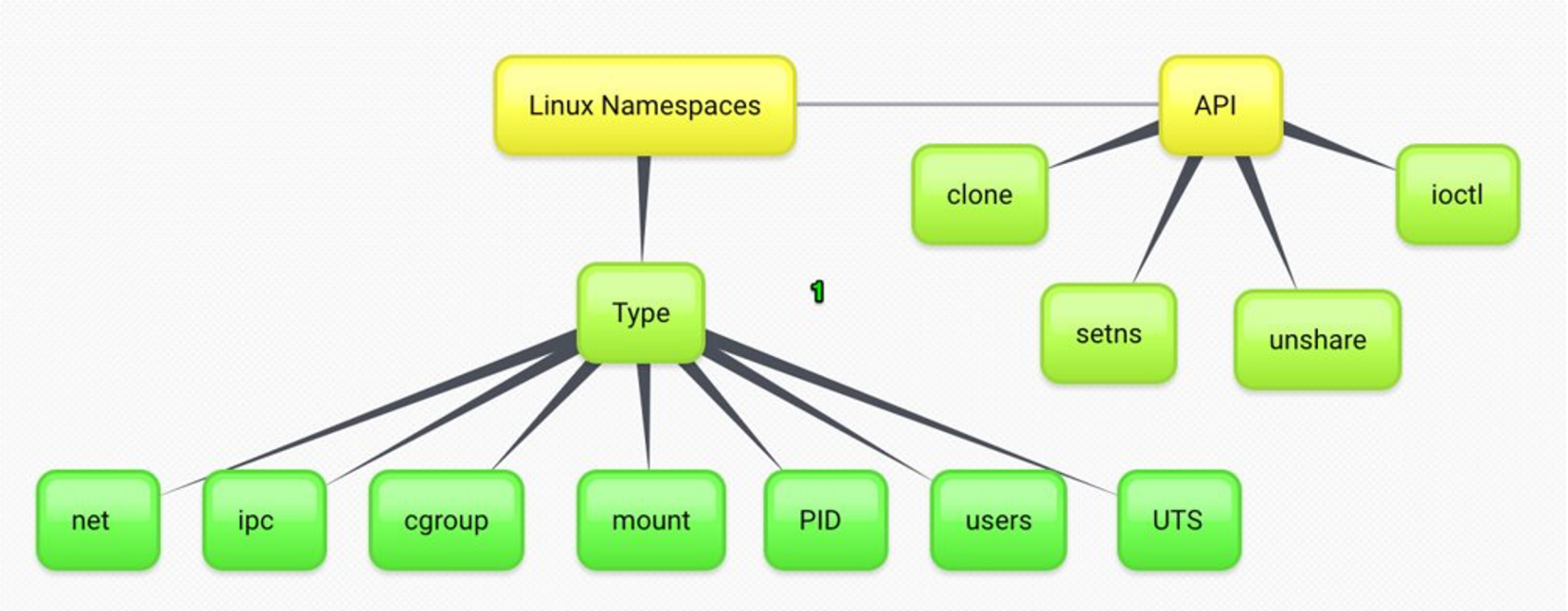
- Namespace: naming domains for various resources:
  - mnt (mount points, filesystems)
  - pid (processes)
  - net (network stack)
  - ipc (System V IPC)
  - uts (hostname)
  - user (UIDs)

- Linux Control Groups: collection of processes:
  - Limits resource usages at group level
    - E.g., memory, CPU, device
  - Fair sharing of resources
  - Track resource utilization
    - could be used for billing/management
  - Control processes
    - pause/resume, checkpoint/restore, etc.

Containers - nerd details

Acknowledgement: Luigi De Simone

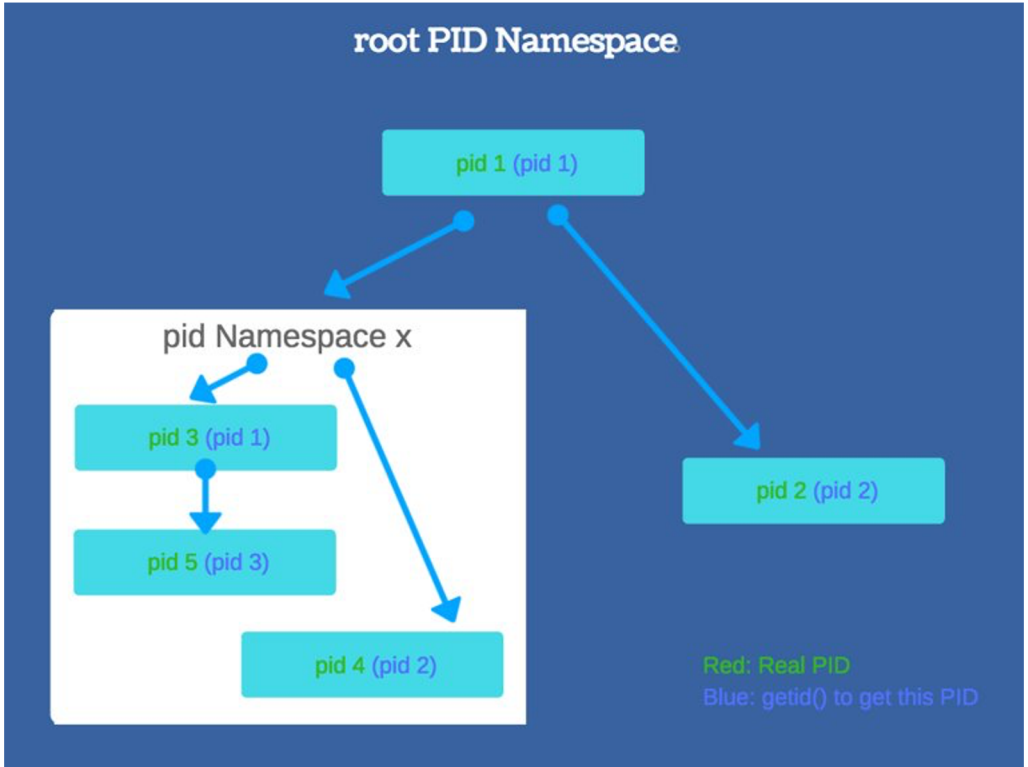
- Namespaces system calls



Containers - nerd details

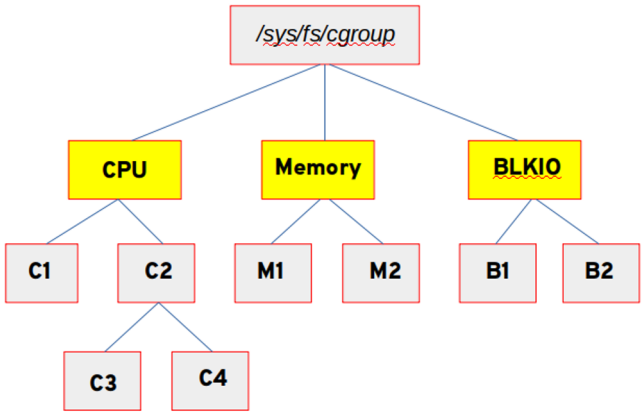
Acknowledgement: Luigi De Simone

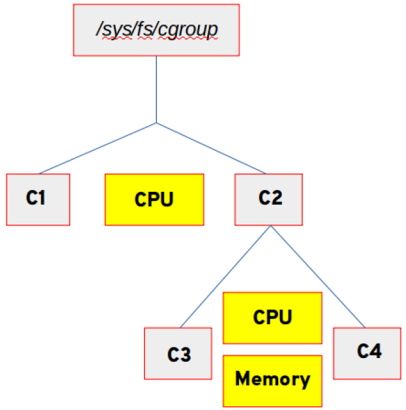
- PID namespace example



Containers - nerd details

- CGroups: nested hierarchy, inheritance
- Independent from containers!
- CGroups v1 vs v2, by default cgroups v2 since Linux 4.5



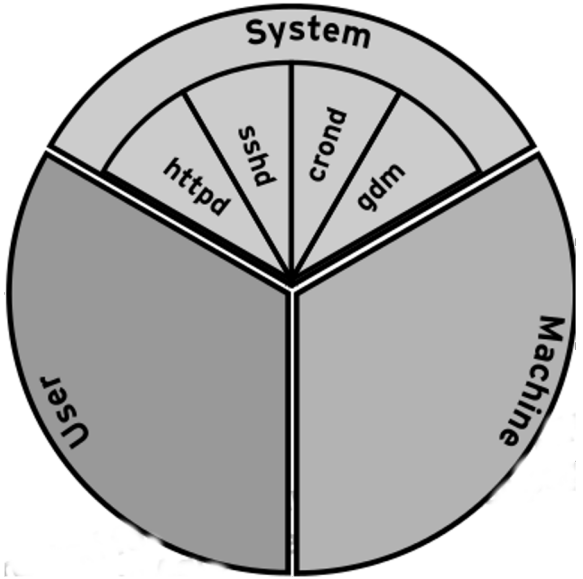


(https://www.cse.iitb.ac.in/~puru/courses/spring19/cs695/refs/containers-manual-prashanth.pdf
https://www.redhat.com/en/blog/world-domination-cgroups-rhel-8-welcome-cgroups-v2
https://wiki.archlinux.org/title/cgroups

Containers - nerd details

- Divided in slices: by default -.slice, system.slice, user.slice, machine.slice

```
[root@rhel7lab ~]# systemd-cgls --no-page
├─1 /usr/lib/systemd/systemd --switched-root --system --deserialize 20
├─user.slice
│ ├─user-1000.slice
│ │ └─session-331.scope
│ │   ├─2584 sshd: mrichter [priv]
│ │   ├─2588 sshd: mrichter@pts/1
│ │   └─2589 -bash
│ └─user-0.slice
│   └─session-330.scope
│     ├─2306 sshd: root@pts/0
│     ├─2309 -bash
│     └─3224 systemd-cgls --no-page
└─system.slice
  ├─httpd.service
  │ ├─1243 /usr/sbin/httpd -DFOREGROUND
  │ ├─1244 /usr/sbin/httpd -DFOREGROUND
  │ ├─1245 /usr/sbin/httpd -DFOREGROUND
  │ ├─1246 /usr/sbin/httpd -DFOREGROUND
  │ ├─1247 /usr/sbin/httpd -DFOREGROUND
  │ └─1248 /usr/sbin/httpd -DFOREGROUND
  ├─rhnsd.service
  │ └─1212 rhnsd
  ├─rhsmcertd.service
  │ └─1206 /usr/bin/rhsmcertd
  ├─sshd.service
```



Container demo

- Useful commands:
  - For namespaces:
    - Man unshare : description of the command to create all ns
    - unshare -Urpf --mount-proc … important flags
    - nsenter -t [pid] -a , to enter all ns of pid
    - lsns to list namespaces
    - cat /proc/$$/status | grep NSpid
    - ps -ef … well you should know (check difference inside and outside ns)
    - whoami
    - cat /proc/$$/uid_map

https://www.redhat.com/sysadmin/building-container-namespaces https://www.youtube.com/watch?v=J17rXQ5XkDE
https://www.redhat.com/sysadmin/pid-namespace

Container demo

- Useful commands:
  - For chroot:
    - systemd-machine-id-setup --root=/path/to/your/rootfs
    - debootstrap --arch i386 stretch PATHOFROOT http://deb.debian.org/debian to download a minimal rootfs
    - chroot PATHOFROOT /bin/bash , to chroot
  - For cgroups:
    - cgcreate -t uid:gid -a uid:gid -g subsystems:path
    - mkdir /sys/fs/cgroup/cg1 to create cgroup
    - echo '+cpu -memory' > /sys/fs/cgroup/cg1/cgroup.subtree_control
    - echo $$ > /sys/fs/cgroup/cg1/cgroup.procs for moving process to cgroup (cgroup v1)

https://man7.org/linux/man-pages/man7/cgroups.7.html https://facebookmicrosites.github.io/cgroup2/docs/create-cgroups.html
https://www.howtogeek.com/441534/how-to-use-the-chroot-command-on-linux/ https://wiki.debian.org/it/chroot

Real-Time Containers

Motivations, previous solutions

Why real-time containers?

- Real-time Containers:
  realize **MCSs** with the same **benefits** already experienced by large integrators in cloud environments

- Modularity, scalability, **efficient usage** of hardware resources, naming and fault isolation

- **Compared** to traditional virtualization methods:
  - No replication of the OS for every system
  - Less overhead
  - Container orchestrators can help to create dynamic real-time clouds

- Also: When talking about embedded systems, **scalability** is the key!

Why real-time containers?

Final aim

- Provide isolation guarantees to real-time tasks running within critical containers as if they were running alone on the machine

while

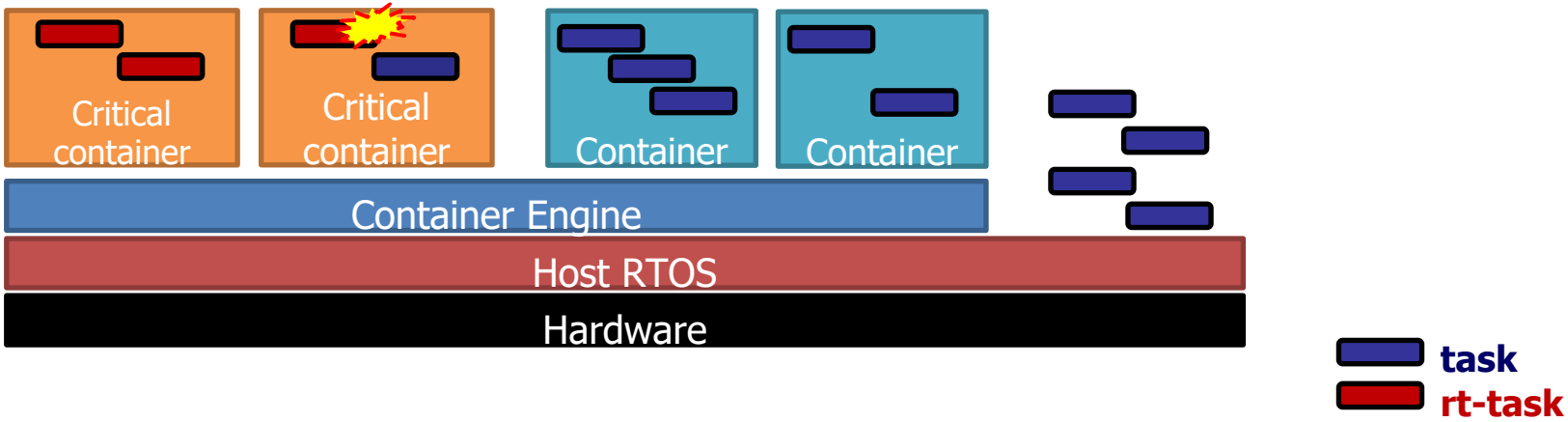
- Allowing other critical or non-critical containers and tasks to be admitted and scheduled on the machine in a controlled way

What about trying Docker over a RTOS?
Feasible! But keep in mind non real-time paths!

Real-time containers: objective

Acknowledgement: Raffaele Della Corte

- **Avoid** late **timing failures** to rt-tasks running in critical containers, despite:
  - **External** disturbances: non-real time CPU/IO intensive workload on the same machine
  - **Internal** disturbances: faulty hard real-time task consuming more CPU than declared



State of art - works

- Use of **preemptive fixed-priority** (PFP) scheduling and temporal protection with **on-line** execution time **monitoring**
- Use of **dynamic scheduling** (EDF) coupled with bandwidth reservation (CBS), implemented through **active monitoring** on RTAI
- Use of **SCHED_DEADLINE** policy integrating cgroups, creating **rt-cgroups** in Linux
- Use of **FIFO** threads over a Xenomai and **PREEMPT_RT** patched kernel for **control applications** talking through ZeroMQ
- Other feasibility studies to discover measures

However

State of art - downsides

- A global PFP scheduler may **not reflect** designer priorities, **unpractical** for migration
- Monitoring **overhead**, **single point of failure**
- Vanilla Linux presents higher **switching times** and scheduling **latencies**
- PREEMPT_RT patch **clashes** with rt-cgroups
- FIFO threads don’t allow **bandwidth limitation**

Our contribution

A new solution based on temporal protection by design, avoiding the use of active monitoring and dealing with network isolation as well

Containers and real-time - Linux

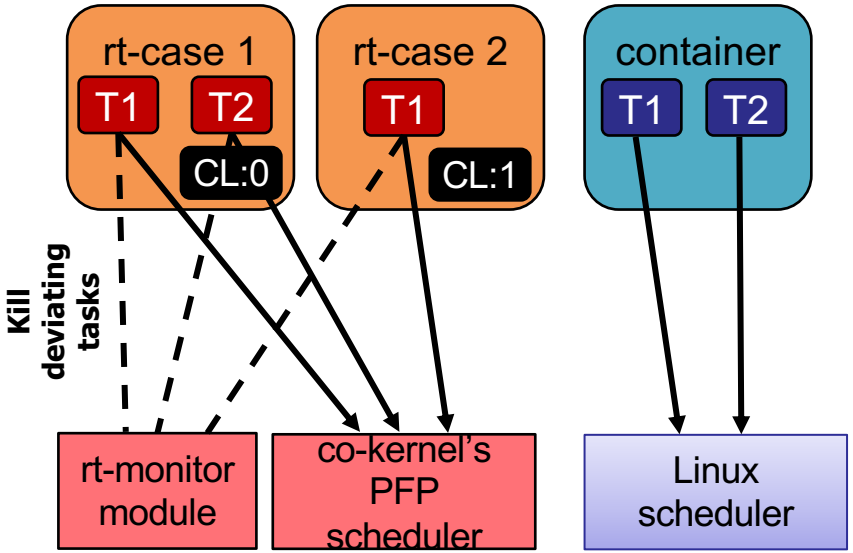
- Recent studies have explored the use of Docker containers on top of Linux using:
  - **PREEMPT-RT**, which improves the latency of dockerized apps, with no sensible overhead introduced by containers themselves
  - Linux offers real-time cgroups out of the box, but its behaviour is **unclear**
  - Research solutions use the **SCHED_DEADLINE** policy (Earliest Deadline First - EDF) with group scheduling
    - In Linux each core has a real-time runqueue, a deadline runqueue and non-rt runqueue. Each task has **descriptors** inserted in queues
    - **CGroups** real-time **runqueues** are inserted in the SCHED_DEADLINE queues, using it at root level, with FIFO or RR at second level
    - **Flattening** of nested CGroups

Containers and real-time - Linux

- However:
  - Time isolation is **not** fully **guaranteed** in Linux
  - It is known that real-time co-kernels (RTAI, Xenomai) **outperform** PREEMPT-RT in terms of latencies and task switch times
    - Co-kernels make Linux **fully preemptable** in favor of real-time tasks
    - They also add core real-time **support** to user level real-time tasks, such as fixed-priority or EDF scheduling, priority inheritance, and task communication
    - Real-time cgroups (vanilla) **clash** with PREEMPT-RT patch
- Summing up: good for soft real-time, unsuitable for hard real-time

Containers and real-time - RTAI

- Use of preemptive fixed-priority (PFP) scheduling and temporal protection with on-line execution time monitoring



- **Pros:**
  - Transparency and simplicity
  - A proper global priority assignment can guarantee critical rt-tasks timeliness depending on rt-case's criticality (CL)
- **Cons:**
  - The priority assignment may not reflect the priorities planned by design
  - impractical if application dependent constraints on priorities must be met
  - and if rt-cases need to be migrated, as in dynamic cloud environments
  - Monitoring overhead

M. Cinque, R. Della Corte, A. Eliso, and A. Pecchia. «RT-CASEs: Container-Based Virtualization for Temporally Separated Mixed-Criticality Task Sets»

Real-Time Containers

My solution over Xenomai

Our proposal

- Both earlier co-kernel based proposals require dedicated active monitoring to enforce temporal protection
  - **Non-negligible overhead**
  - **Single point of failure**

temporal protection by design is a proactive way to isolate task like CBS or Deferrable Server

- A new solution allowing t**emporal protection by design**, avoiding the use of active monitoring and dealing with **network isolation** as well

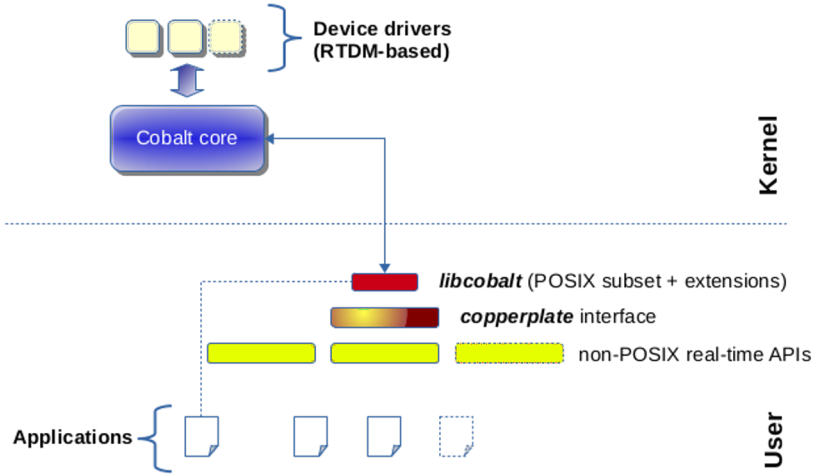
- The proposal is based on:
  - **Xenomai** co-kernel
  - Modification to **SCHED_QUOTA** policy provided by Xenomai
  - **Hierarchical** group scheduling
  - **RTnet** stack

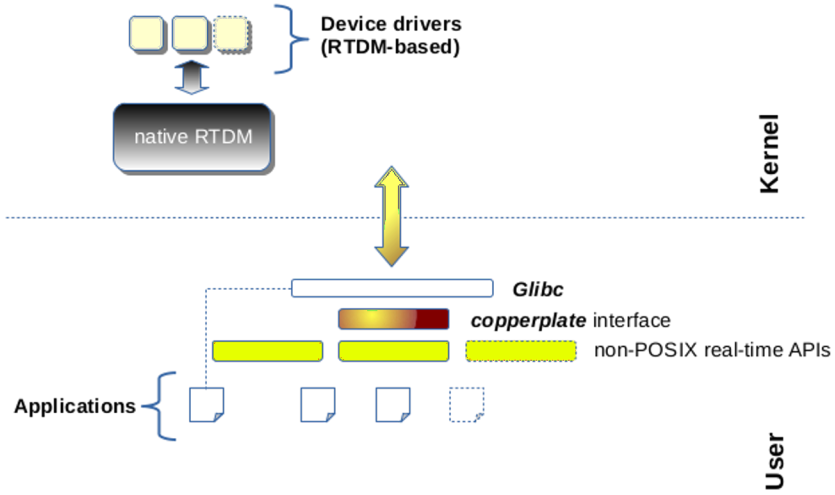
Why Xenomai?

- Greater **extensibility** (RTOS skins), portability, and **maintainability**
- **POSIX** 1003.1c compliance for rt-threads in user space
- Both **dual kernel** and **single kernel** configurations
- **Support** to group scheduling in dual kernel mode:
  - **SCHED_TP** provides cyclic temporal partitioning for thread groups execution
  - **SCHED_QUOTA** enforces a limitation on the CPU consumption of group of threads over a globally defined period, a.k.a the quota interval.

Xenomai details

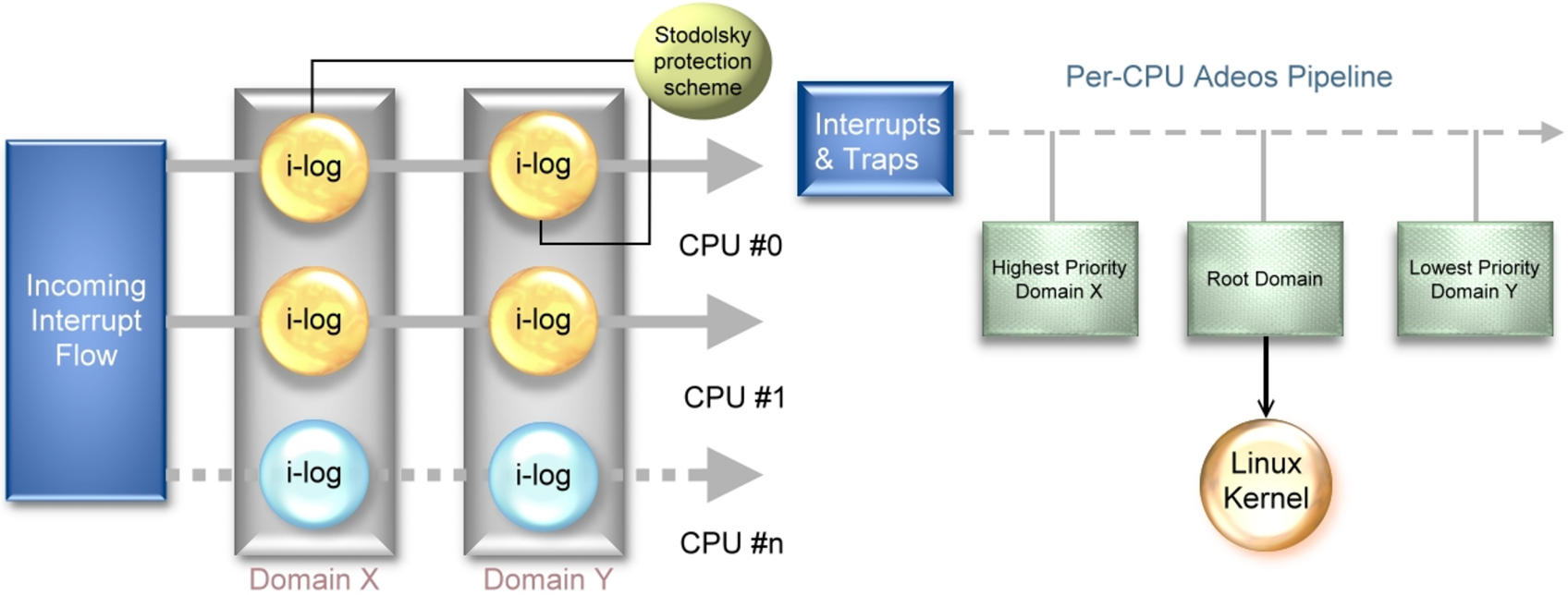
- Dual kernel vs Single kernel structure





Xenomai details

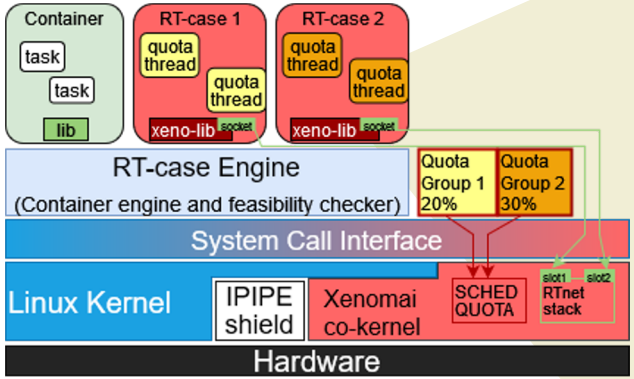
- Similar to RTAI in the use of interrupt pipeline

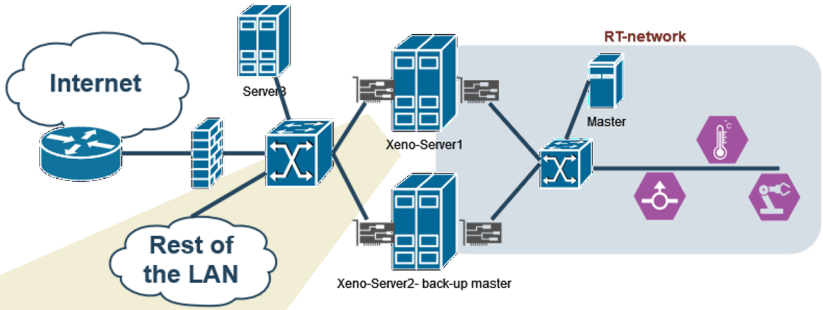


*These images are out of date, current structure is a little different

System architecture

- Elements of the architecture:
  - Xenomai
  - A hierarchical scheduler
  - A feasibility checker
  - Docker
  - The xeno-lib
  - The RTnet stack
  - Mechanisms for other resources

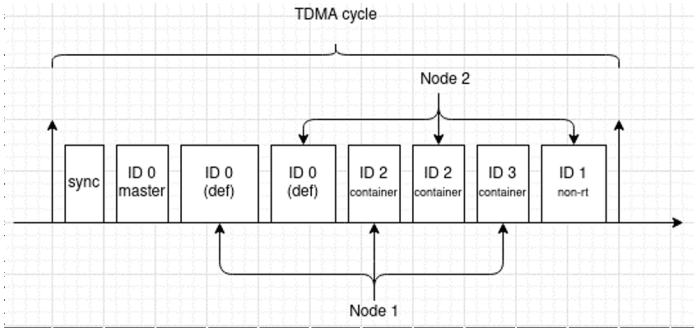




- The orchestration system creates a **quota group** and sets up **RTnet**
- It opens a container passing the TGID as **environment variable**
- The xeno-lib layer **wraps** the POSIX API imposing SCHED_QUOTA policy and binding sockets to timeslots

Xeno-lib wrapper

- Replaces normal POSIX calls with extended ones, masking the complexity
- Uses a linked list made of nodes that couple a standard pthread_attr_t with a pthread_attr_ex_t
- Uses input parameters as a key to find extended structures
- Replaces FIFO scheduling policy with QUOTA, using the correct group ID
- Wraps the socket call to add an ioctl to bind to the correct timeslot



- The orchestration system has the burden to handle at runtime and it keeps track of timeslots of every node

TDMA= Temporal Division Multiple Access

SCHED_QUOTA modification

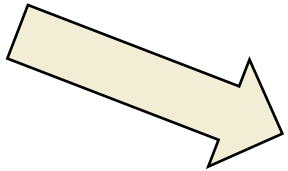
- SCHED_QUOTA enforces a limitation on the CPU consumption of group of threads over a globally defined period

Algorithm 1 original xnsched_quota_pick pseudocode

```
1:  procedure XNSCHED_QUOTA_PICK
2:      if outgoing thread is a quota thread then
3:          subtract used runtime from group budget
4:      pick highest priority ready thread from runqueue
5:      if thread is null then
6:          stop limit timer and idle
7:      if thread is not quota thread then
8:          return thread
9:      start_time of thread group ← now
10:     if thread group runtime budget is 0 then
11:         enqueue thread in expired queue
12:         pick another thread (4)
13:     arm limit timer to expire at now+group_budget
14:     decrease active threads in group
15:     return thread
```

Issues:

- Budget evaluation made at thread level ⇒ **O(N)** complexity
- No notion of **group priority**
- **Useless** checks due to the fixed-prio runqueue shared between policies



Algorithm 2 modified xnsched_quota_pick pseudocode

```
1:  procedure XNSCHED_QUOTA_PICK
2:      if outgoing thread is a quota thread then
3:          subtract used runtime from group budget
4:      pick highest priority ready group from runqueue
5:      if group is null then
6:          stop limit timer and idle
7:      if thread group runtime budget is 0 then
8:          enqueue group in expired queue
9:          pick another group (4)
10:     start_time of thread group ← now
11:     pick highest priority ready thread from runqueue
12:     arm limit timer to expire at now+group_budget
13:     decrease active threads in group
14:     return thread
```

SCHED_QUOTA modification

- Modifications include:
  - Group priority notion
  - A new fixed-prio runqueue for each group
  - The removal of the queue of expired threads
  - A new per-core expired and runnable group queue
- Modified groups can be modeled as deferrable servers
  - Response time analysis from Davis and Burns can be simplified
  - Do you remember her? :)

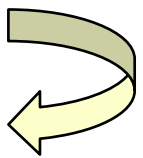
- **Group priority** prevents mixing threads of different groups
- **O(1)** complexity due to the bounded number of groups
- Useless checks and credit mechanism **removed**

$$L_i(w_i) = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j$$

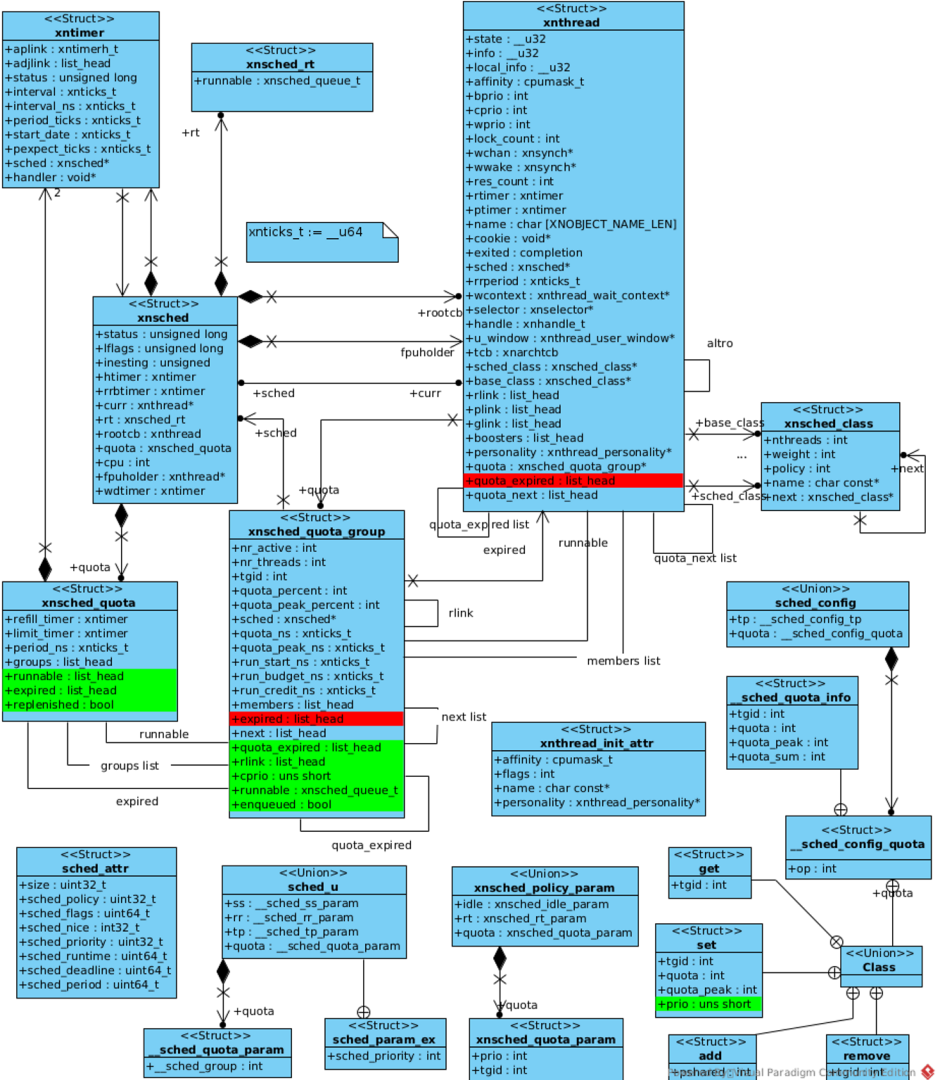
$$w_i^{n+1} = L(w_i^n) + \left(\left\lceil \frac{L(w_i^n)}{C_s} \right\rceil - 1 \right) (T_s - C_s) + \sum_{\substack{\forall X \in hp(S) \\ servers}} \left\lceil \frac{max\left(0, w_i^n - \left(\left\lceil \frac{L(w_i^n)}{C_s} \right\rceil - 1 \right) T_s \right) + J_X}{T_X} \right\rceil C_X$$

$$w_i^{n+1} = L(w_i^n) + \left(\left\lceil \frac{L(w_i^n)}{C_s} \right\rceil - 1 \right) (T_s - C_s) + \sum_{\substack{\forall X \in hp(S) \\ servers}} C_X$$

non possiamo avere back to back interferenze. Abbiamo un lock step period: the timere is the same, the period and refil is the same, so within in one period the higher priority Defferable Server can execute just for one computational budjet. Quindi possiamo semplificare



R.I. Davis, A. Burns “Hierarchical Fixed Priority Pre-emptive Scheduling” In proceedings of the IEEE Real-Time Systems Symposium, December 2005.

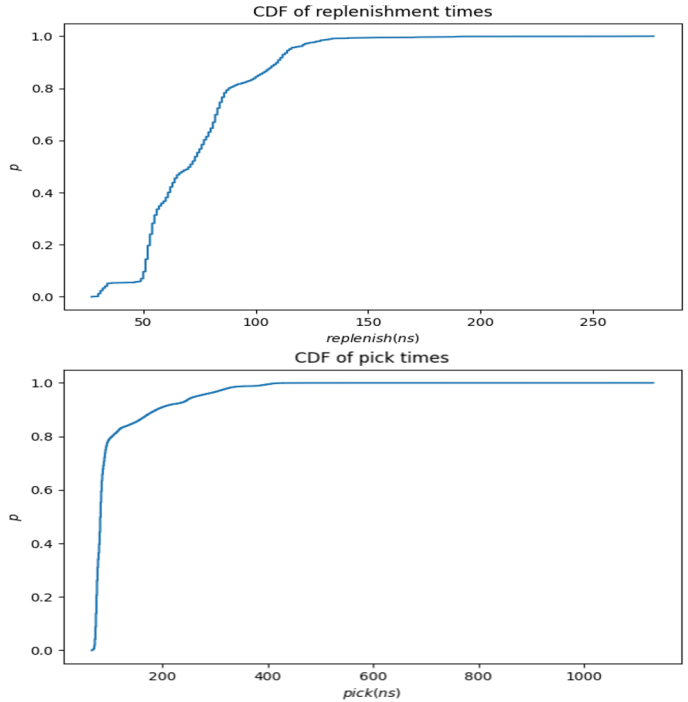


SCHED_QUOTA modification

- Simplification exploits the refill_timer in xnsched_quota that is shared between groups
- Period elapse in a lockstep fashion! No back to back hits

Testing - 1

- Execution times



- 4 groups with 6 quota threads each
- Duration of 10 seconds
- Permanently executing task to exhaust budget
- Comparison with RTDS

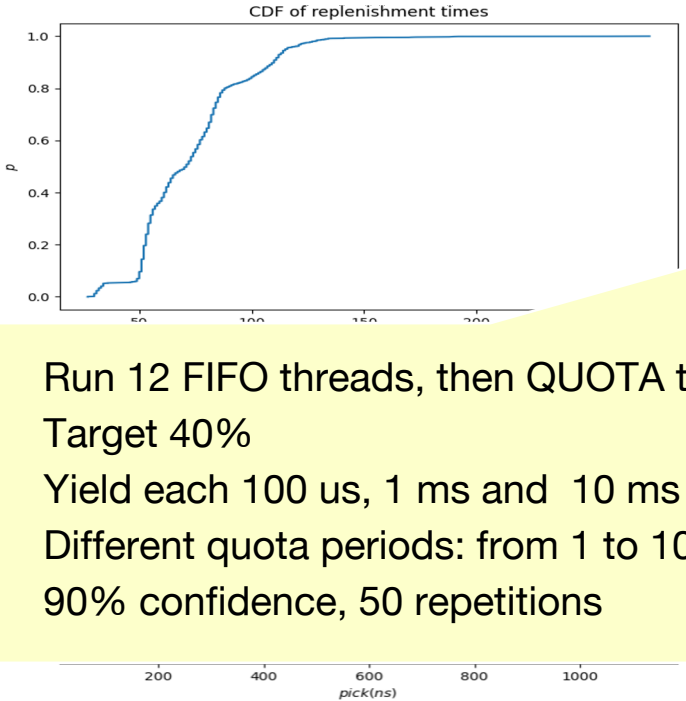
Testing Platform

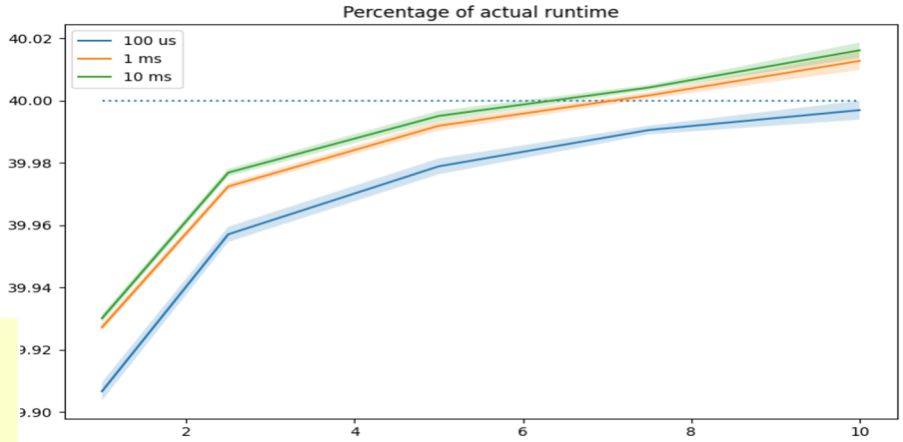
Intel core i5-6500 3.2 GHz
16 GB DDR4 RAM 2666 MHz
Docker 20.10.7

Samsung 970EVO SSD
Xenomai 3.1 on Linux 5.4.77
Linux Mint 20.1

Testing - 1

- Execution times





- Run 12 FIFO threads, then QUOTA threads
- Target 40%
- Yield each 100 us, 1 ms and 10 ms
- Different quota periods: from 1 to 10 ms
- 90% confidence, 50 repetitions

Testing Platform

Intel core i5-6500 3.2 GHz
16 GB DDR4 RAM 2666 MHz
Docker 20.10.7

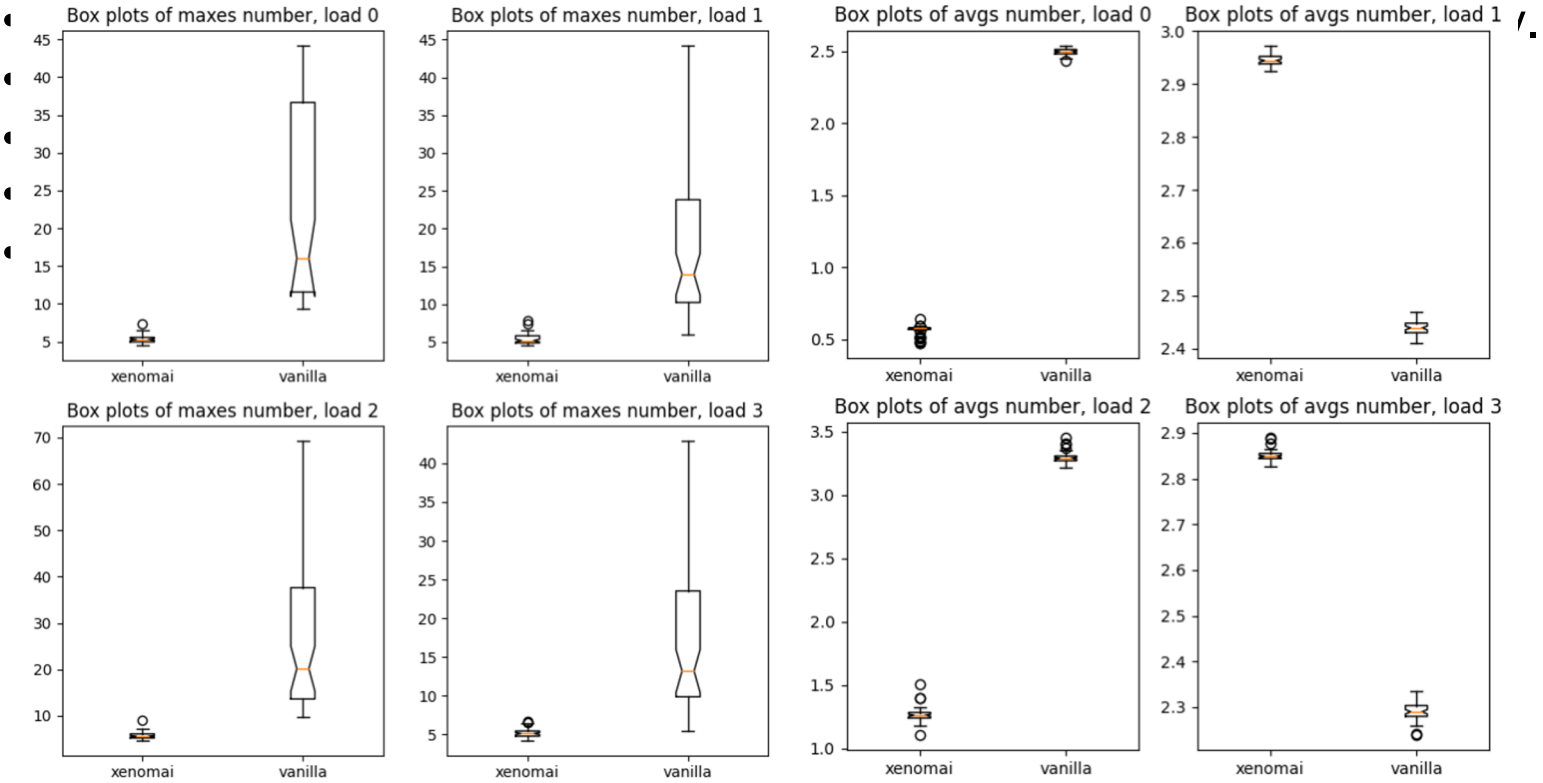
Samsung 970EVO SSD
Xenomai 3.1 on Linux 5.4.77
Linux Mint 20.1

Testing - 2

- Activation latency test adapted to run with SCHED_QUOTA policy.
- The test is executed within a docker container
- Overruns, medium and worst latency saved for each experiment
- Sampling period of 100 µsec
- 60 repetitions of 1 minute for each of the 4 loads:
  - no load
  - CPU stress (stress-ng --cpu-load 100)
  - I/O stress (stress-ng --io)
  - Net stress (stress-ng --netdev)

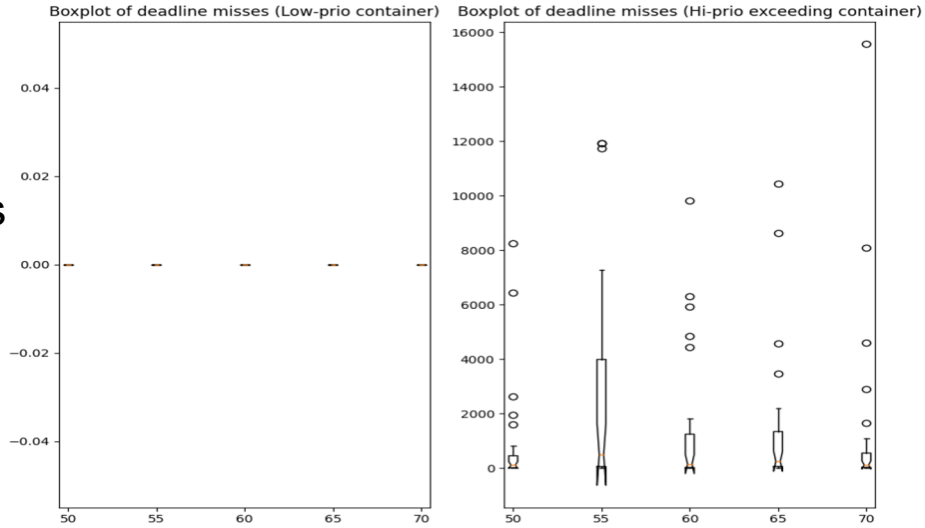
	Mean of avg	Std of avg	Mean of worst	Std of worst
No load	0.565	0.031	5.332	0.509
CPU load	2.945	0.011	5.367	0.665
IO load	1.267	0.054	5.695	0.726
Net load	2.860	0.014	5.231	0.600

Testing - 2


Box plots of maxes number, load 0
Box plots of maxes number, load 1
Box plots of avgs number, load 0
Box plots of avgs number, load 1
Box plots of maxes number, load 2
Box plots of maxes number, load 3
Box plots of avgs number, load 2
Box plots of avgs number, load 3
xenomai
vanilla

Testing - 3

- Temporal isolation test
- Two rt-containers
- Stafford algorithm to generate tasksets
- The high prio container runs for 1.8 times the declared
- 30 repetition, 4 stress condition, 4 total bandwidths
- No deadline miss occurred



- RT-net measurements
  - Both real-time and tunneled UDP traffic
  - Two containers on two slots on the slave
  - OS-level stress

- 112 usec of transmission delay
- Jitter of master: 385 ns mean, 103 ns std
- Jitter of slave: 211 ns mean, 1006 ns std

Thank you!

Questions?