Multi-Level Tracing of Containerized Application Orchestrator

Benjamin Saint-Cyr

Polytechnique Montréal
DORSAL

Introduction

Cloud Computing

1. aggregation and distribution of computing resources in a homogeneous way to clients
2. Allows customers to scale up operations instantly
3. Also facilitate the restitution of resources as needed
4. Flexible billing based on usage

Introduction

Containers

1. Containers have emerged as a more efficient method for resource allocation and isolation
2. Share the same Kernel
3. Containers simplify packaging and distribution services
4. Provide a consistent runtime environment in both development and production

Introduction

1. Orchestrators are an easy way to deploy containers to the cloud
2. They enable the deployment of services using descriptive configurations
3. Orchestrators facilitate achieving and maintaining the desired state
4. Also offers utilities for automatic scaling, implementing rollout strategies, and managing configurations and secrets

Introduction

1. Performance is an important factor
2. Reduced latency enhances the quality of service provided
3. Quicker boot times improves elastic scaling
4. More efficient resource utilization directly leads to reduced hosting costs

Research Objective

To observe and analyze the behaviour of a distributed system of orchestrated containers, with the aim of facilitating the diagnosis of performance issues

Methodology

Performance Aspects

1. Quicker boot times enhance elasticity in scaling
2. Better utilization of resources improves Quality of Service (QoS) and reduces costs

Methodology

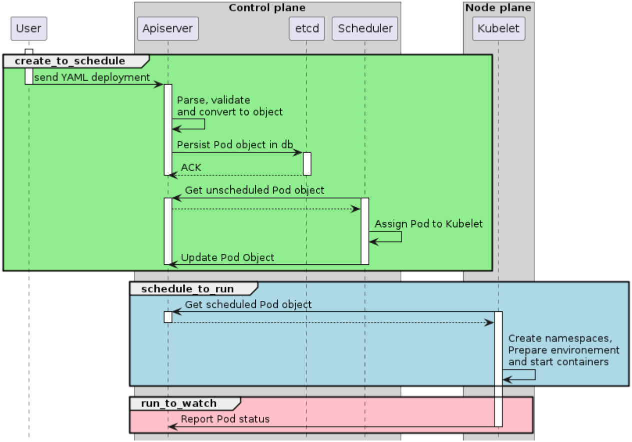


Figure: Components and phases of a pod start up

Methodology

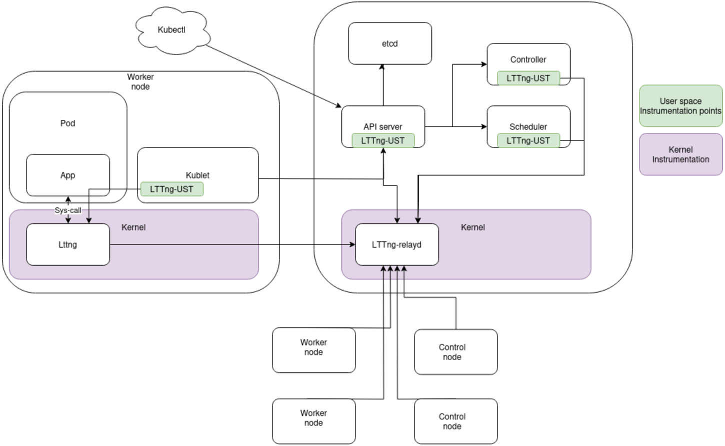


Figure: Tracing Architecture

Methodology

Common Issue During Execution : Managing CPU Limits

1. Limits are specified in cores, but enforced as quotas

2. A container can use more cores than allocated

$$Quota = \text{Cores Limit} \times Period$$

$$Throttled\ \ Time = Period \times (Cores\ \text{Used} - Cores\ \text{Limit})$$

Methodology

Expectation

Running	Running
Running	Running
Iddle	Iddle
Iddle	Iddle
Iddle	Iddle
Iddle	Iddle
Iddle	Iddle
Iddle	Iddle

Reality

Running	PREEMPTED	Running	PREEMPTED
Running	PREEMPTED	Running	PREEMPTED
Running	PREEMPTED	Running	PREEMPTED
Running	PREEMPTED	Running	PREEMPTED
Running	PREEMPTED	Running	PREEMPTED
Running	PREEMPTED	Running	PREEMPTED
Running	PREEMPTED	Running	PREEMPTED
Running	PREEMPTED	Running	PREEMPTED

Methodology

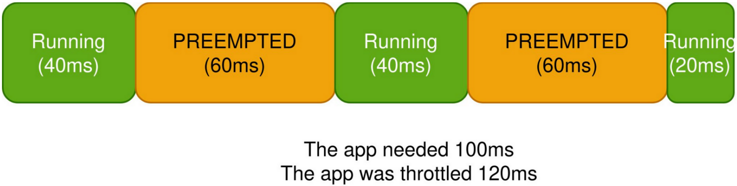


Figure: Example of Latency Introduced by CPU Limit

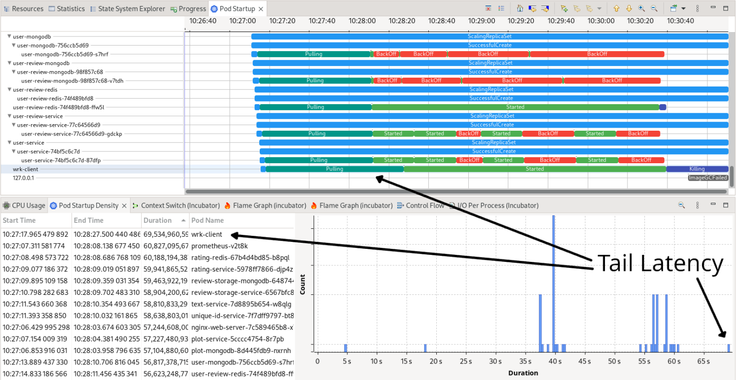
Algorithm to characterize CPU usage per pod

Algorithm 1 CPU usage calculation per Cgroup

Define the function `normalize(prevTime, time, cpuTime)` as $\frac{\texttt{cpuTime}}{\texttt{time}-\texttt{prevTime}} \times 100$
procedure ANALYZECPUUSAGE(fetchParameters, stateSystem)
    Initialize $totalCpu \leftarrow 0$
    Initialize $prevTime \leftarrow Get\ the\ start\ time\ from\ fetchParameters$
    Initialize $xAxis \leftarrow Initialize\ xAxis\ based\ on\ the\ time\ range\ and\ resolution$
    Initialize map $threadUsage \leftarrow Initialize\ the\ cpu\ usage\ of\ each\ thread\ based\ on\ stateSystem$
    Initialize map $CgroupUsage \leftarrow Initialize\ the\ cpu\ usage\ of\ each\ cgroup\ based\ on\ stateSystem$
    for $i \leftarrow 0$ to $xAxis.length - 1$ **do**
        $time \leftarrow xAxis[i]$
        for all $(threadName, cpuTime) \in$ `fetchThreadsCpuUsage`$(time)$ **do**
            $totalCpu \leftarrow totalCpu + cpuTime$
            $normalizedThreadCpuUsage \leftarrow$ `normalize`$(prevTime, time, cpuTime)$
            $threadUsage[threadName][i] \leftarrow normalizedThreadCpuUsage$
            if $TidToCgroup.contains(threadName)$ **then**
                $key \leftarrow TidToCgroup.get(threadName)$
                $CgroupUsage[key][i] \leftarrow CgroupUsage[key][i] + normalizedThreadCpuUsage$
            end if
        end for
        $prevTime \leftarrow time$
    end for
    return CgroupUsage, threadUsage
end procedure

Results

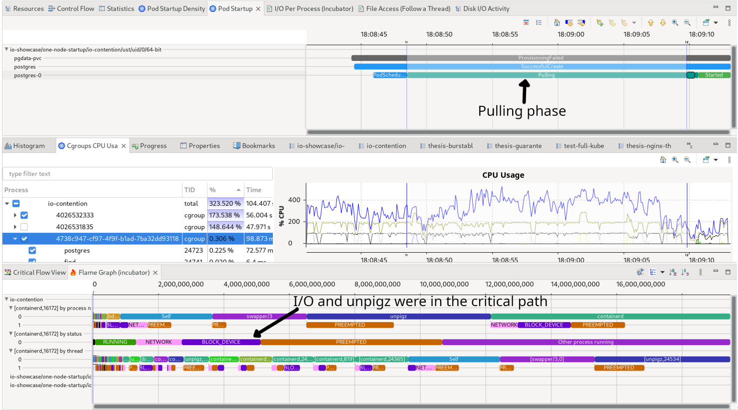
Pod Lifecycle



Start Time	End Time	Duration	Pod Name
10:27:17.965 479 892	10:28:27.500 440 486	69,534,960,59	wrk-client
10:27:07.311 581 774	10:28:08.138 677 450	60,827,095,67	prometheus-v2t8k
10:27:08.498 573 722	10:28:08.686 768 109	60,188,194,38	rating-redis-67b4d4bd85-b8pql
10:27:09.077 186 372	10:28:09.019 051 897	59,941,865,52	rating-service-5978ff7866-djp4z
10:27:09.895 109 158	10:28:09.359 031 354	59,463,922,19	review-storage-mongodb-64874
10:27:10.798 282 683	10:28:09.702 483 310	58,904,200,62	review-storage-service-6567bfc8
10:27:11.543 660 368	10:28:10.354 493 667	58,810,833,29	text-service-7d8895b654-w8qlg
10:27:11.393 358 850	10:28:10.032 161 865	58,638,803,01	unique-id-service-7f7dff9797-bt8
10:27:06.429 995 298	10:28:03.674 603 305	57,244,608,00	nginx-web-server-7c589465b8-x
10:27:07.154 009 319	10:28:04.381 490 255	57,227,480,93	plot-service-5cccc4754-8r7pb
10:27:06.853 916 031	10:28:03.958 796 635	57,104,880,60	plot-mongodb-8d445fdb9-nxrnh
10:27:13.889 437 330	10:28:10.706 816 045	56,817,378,71	user-mongodb-756ccb5d69-s7hr
10:27:14.833 186 566	10:28:11.456 435 341	56,623,248,77	user-review-redis-74f489bfd8-ff

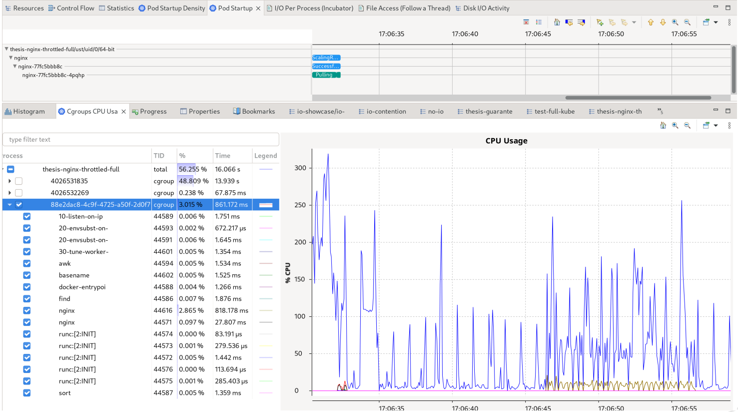
Results

Diagnosing Slow Pod Startup Issues Attributable to I/O Contention



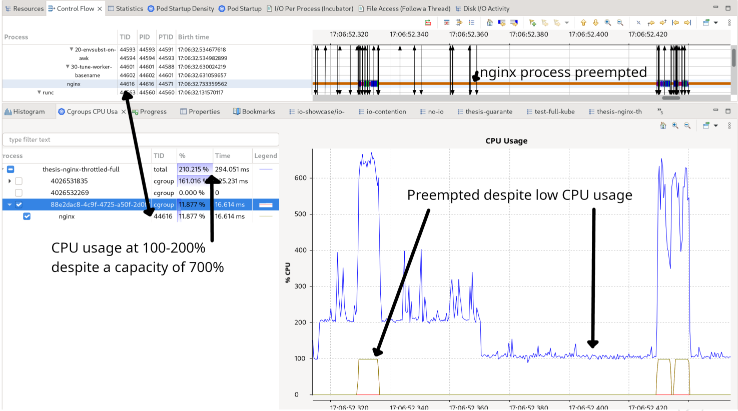
Results

CPU usage per cgroup and pod



Results

Diagnose latency due to limits



Conclusion

1. Demonstrates how tracing aids in observing the internal states of the orchestrator
2. Diagnosing issues at both the orchestrator and kernel levels
3. Highlights the utility of multi-level tracing for diagnostic purposes

Conclusion

Future work

1. Reuse methodology for other resources: Network, memory, disk, etc.

2. Use the trace data to characterize workload and improve scheduling

Conclusion

Thank you!
Questions?