Orchestration-aware Workload Collocation Agent.

Architecture & reference setup.

Author: Paweł Pałucki (pawel.palucki@intel.com)

Co-authors:

Maciej Iwanowski (maciej.iwanowski@intel.com)
Szymon Bugaj (szymon.bugaj@intel.com)

Contents

Changelog

Date	Version	Author	Description
2018.04.10	0.1	Paweł Pałucki	Initial version with high level description and initial charts, requirements assumptions and constraints.
2018.04.26	0.2	Paweł Pałucki	User environment requirement chapter and HW and SW details.
2018.04.27	0.3	Paweł Pałucki	More details and description about overall architecture.
2018.05.17	0.4	Damian Darczuk, Roman Dobosz, Maciej Iwanowski, Mohit Malik, Marta Mucek, Paweł Pałucki	Team's review of the document.
2018.05.18	0.5	Paweł Pałucki, Maciej Iwanowski, Szymon Bugaj	Addressing team comments: simple API, more details about controller, fix direction of flow in charts more terminology and "performance" metrics formula
2018.10.01	1.0	Pawel Palucki	OWCA wrapper component description.
2018.05.23	1.1	Pawel Palucki, Szymon Bugaj, Maciej Iwanowski	Polishing and grammar fixes, additional terminology. Better description of controller.
2018.06.23	1.2	Pawel Palucki	Added sections: "Monitoring information flow" and "Mesos integration" Fixed typos architecture diagram
2018.09.30	1.3	Pawel Palucki, Maciej Iwanowski Szymon Bugaj	Changed document title, added OWCA.Wrapper section and updated system architecture.
2018.10.16	1.4	Pawel Palucki	Agent communication and sequence diagrams chapter added and page numbers.
2018.10.16	1.5	Pawel Palucki, Szymon Bugaj	High resolution images and grammar and spelling fixes.

Terminology

Term	Meaning	Example
Workload	Service and work to be done	Memcached with anonymized capture of production requests
Service Level Objective (SLO)	Agreed individual performance metrics that must be achieved to meet defined service level commitments.	30ms maximum latency in 99.9 percentile counted over the last 15 seconds under condition of maximum 10k RPS
Service Level Indicator (SLI)	Measurement of service level characteristic which enables systems and operators to deem whether the Service Level Objective has been violated or is within an acceptable range.	99% of responses are below 10ms under load of 1k RPS
Service Level Agreement (SLA)	Contains detailed description of SLOs. Describes how objectives and indicators are collected and validated.	
Peak load	Maximum load on a task while not violating the SLO	100k RPS
Resource Usage	Amount of resources currently used by a task	1 core, 10 MB RAM
Platform metrics	Platform metrics contains information about resources usage grouped by workloads or aggregated over whole platform or socket.	workload1:100%CPU,50 MB workload2:30% CPU, 30 MB
APM	Application Performance Metrics	100 QPS, 230ms (99th percentile)
Aggressor	Best effort task that interferes with high priority task	Machine Learning algorithm training
Victim	HP Task affected by Aggressor	Memcached, Redis, Cassandra

High Priority (HP) Task	High priority task which takes precedence over BE tasks in terms of dedicated resources, task preemption, etc. Project's objective is to protect this class of tasks to guarantee stable performance. Usually those tasks are services where short and predictable end-to-end response time is critical; they are also known as latency critical tasks.	Memcached, Redis, Cassandra
Best Effort (BE) Task	Best effort, low priority task - task we have right to cancel without consequences	Machine learning algorithm training
Cache Allocation Technology (**CAT)**	Hardware capability to control the allocation of processor last-level cache.	
Cache Monitoring Technology (CMT)	Hardware capability to determine the usage of last-level cache by applications running on the platform.	
Memory Bandwidth Monitoring (MBM)	Provides per-thread and per process memory bandwidth monitoring.	
IRQ affinity	IRQ are typically served by designated kernel threads. The latency for delivering I/O responses, like for networking, can be influenced by whether the kernel thread dealing with the IRQ is on the same hyper thread as the destination application.	
Hyper Threading (HT)	Technology of sharing one core between two logical cores to increase the number of independent instructions in the pipeline.	Enabled / Disabled
QPS/RPS	Queries Per Second a.k.a. Requests Per Second.	2000 queries per second
Tail latency	High percentile value of response latency.	latency of 100ms in 99th percentile
Kafka consumer	Application responsible for fetching messages from Kafka message queue and passing metrics to time-series database (e.g. Prometheus).	

Agent	Main OWCA executable responsible for: collecting node state, communication with resource allocation or anomaly detection algorithm and allocating resources.	
Wrapper	An application responsible for: launching analyzed application or respective load generator and parsing its output to provide APMs.	
OWCA	Acronym for Orchestration-aware Workload Collocation Agent.	

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119[1].

Introduction

Rationale

Reduced infrastructure cost is an important aspect of cloud services. One approach is to raise server utilization by providing effective use of server resources. Efficiency can be improved by (a) launching best effort tasks next to latency-critical services and exploiting resources that are underutilized by latency-critical workloads, and (b) by co-locating workloads that do not have contention for the same shared resource on the system

Goal and Scope

One of the goals of the project is to address the concern of high TCO caused by low server utilization in private cloud environments and democratize tier-1 co-location performance capabilities. Project aims to reduce interference between co-located tasks and increase tasks density while ensuring the quality of service for high priority tasks. One possible solution is to dynamically manage platform isolation mechanisms to ensure that high priority jobs meet their service level objective (SLO) and best-effort jobs effectively utilize as many idle resources as possible.

The initial resources that project will monitor and control are CPU cores and processor cache. More specifically, project will address the above by:

- Controllers for CPU cores and processor cache
- Measuring application performance metrics
- Measuring IA resources through telemetry
- Incorporating IA platform quality of service capabilities such as CMT, CAT, CDP & MBM
- Integrating control loop controller and anomaly detection with orchestration platform

[1] https://www.ietf.org/rfc/rfc2119.txt

Overview

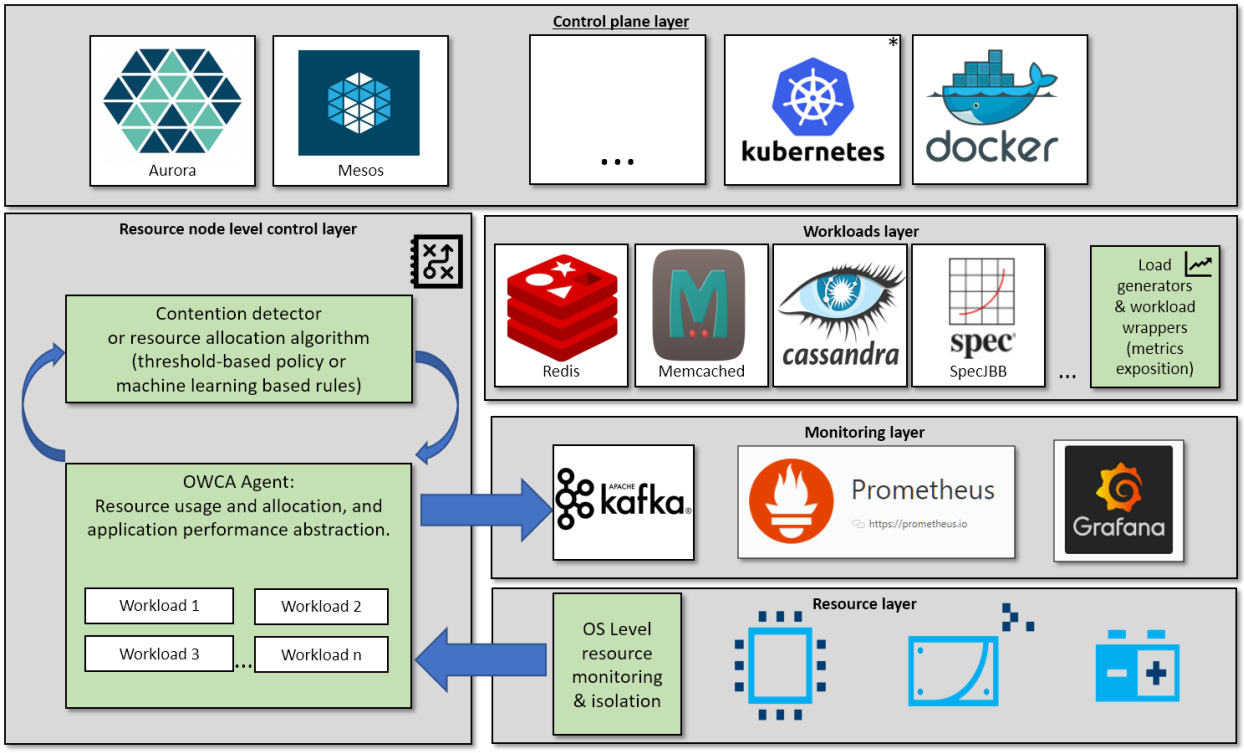

Figure 1 Solution overview

Findings from our previous work and publicly available research have led to few insights:

- Relying on the isolation mechanisms available in current open source scheduler runtimes such as Apache Mesos and Kubernetes, have not been enough to protect HP tasks from noisy neighbors.
- Systems like Google Borg dynamically change the CFS (completely fair scheduling) settings of a process, which may be required to protect high priority tasks from scheduling hazards.
- Performance counters such as IPC and low-level metrics such as L3 cache usage can be used to predict possible interference.

It is important to note that it is easier to oversubscribe compressible (throttleable) resources such as CPU and memory bandwidth because tasks can run with lower than required amount of them. It is more complex to oversubscribe non-compressible resources (e.g. memory or disk). Due to complexity, project will firstly support oversubscription of compressible resources.

To ensure that the user-defined SLO is met for high priority workloads, controller will create an abstraction over workloads and then based on a policy it will:

- tune isolation of resources for running applications or
- detect anomaly (e.g. resource contention)

When running with resource allocation capabilities OWCA will use externally provided closed control loop to mitigate interference by re-allocating resources to prevent performance of applications from degrading.

To facilitate operations, we plan to integrate with existing orchestration software to discover scheduled and running tasks and their respective performance requirements. Support for Mesos/Aurora cluster is the first one implemented.

User stories

1. As a Data Center Operator, I want to maximize the work done on my infrastructure and thus lower TCO.
2. As a Data Center Operator, I want to co-locate low priority jobs with all kinds of high priority jobs to exploit unused resources.
3. As a Service Owner, I want my HP jobs to always meet their SLO.
4. As a Service Owner, I want ways to use idle resources to run my existing services and batch jobs on.
5. As a Service Owner, I want to be able to control which services get oversubscribed. In other words, I don't want to try my luck for HP and BE combinations I haven't tested in an isolated setting.
6. As a Data Center Operator, I want ways to determine an oversubscription quality in terms of achieved compaction and stability
7. As a Data Center Operator, I want to reduce the number of best effort tasks from being killed, evicted or paused.
8. As a Data Center Operator, I want controls to adjust the policy aggressiveness and thus the ratio of failures/degraded performance to utilization ratio.
9. As a Data Center Operator, I want ways to debug oversubscription policies and parameters

Architecture

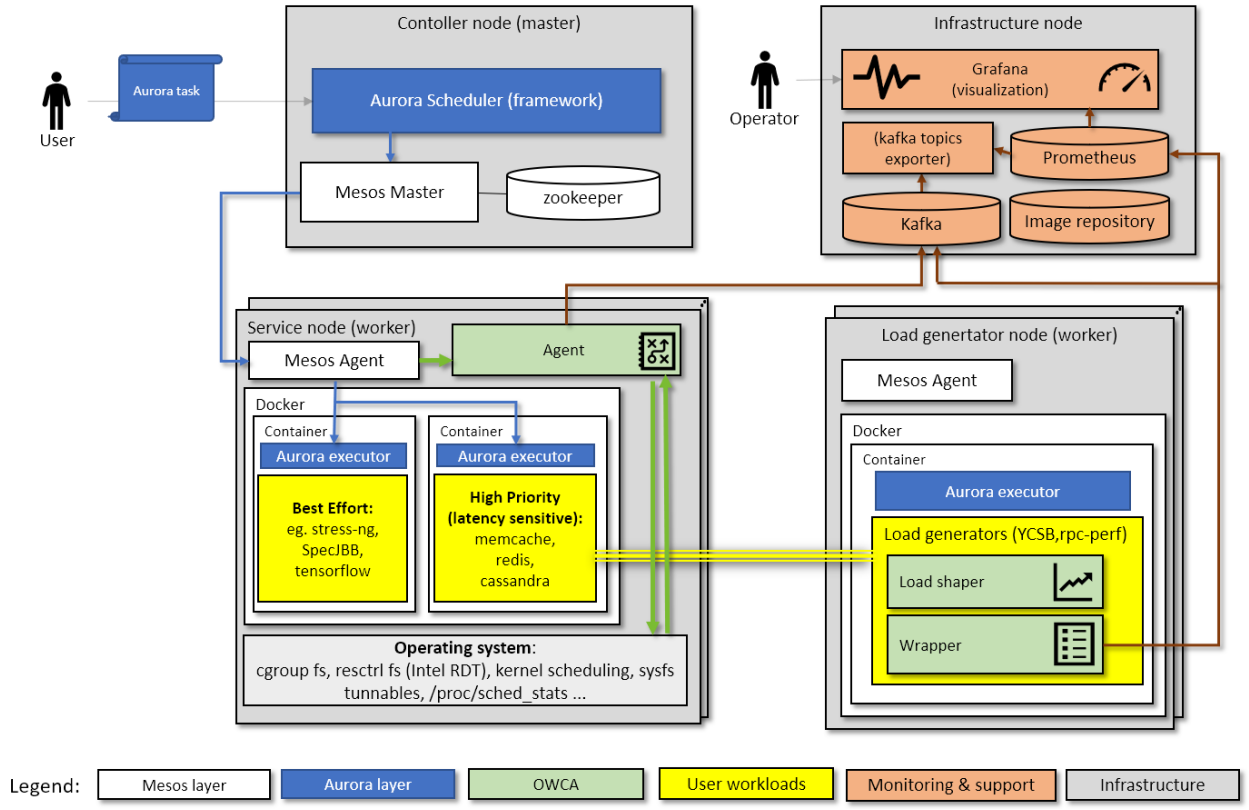

Figure 2 Detailed system architecture

Description

Figure 2 describes how OWCA components interact with each other. Direction of arrows describes information flow between components (it doesn't indicate the initiator of communication nor push-pull model).

System will consist following layers of components:

Layer name	Description and purpose
Mesos	Orchestration and cluster level resource management.
Aurora	Mesos compatible framework for task scheduling and task definition.
OWCA	Feedback based node level resource anomaly detection and resource isolation agent installed as side-kick application on each worker under control of system.
Load shaper patches & Wrapper	Extensions to load generators to generate traffic and expose metrics in compatible Prometheus format.
User workloads	User defined application run on cluster.
Monitoring & support services	Auxiliary components to visualize and facilitate workload deployment.
Infrastructure	Hardware and operating system.

Roles of nodes in the cluster

The setup shows Mesos cluster with 10 nodes and consists four roles:

Roles	Responsibility
Controller	Nodes dedicated to running orchestration software control components for Mesos & Aurora.
Infrastructure	Runs components required for visualization, monitoring and logging infrastructure, and docker image repository.
Worker	Mesos cluster node running customer workloads and OWCA.
Load generator	Mesos cluster node running load generators that mimic customer client side of workload e.g. rpc-perf, YCSB, mutilate.

Agent

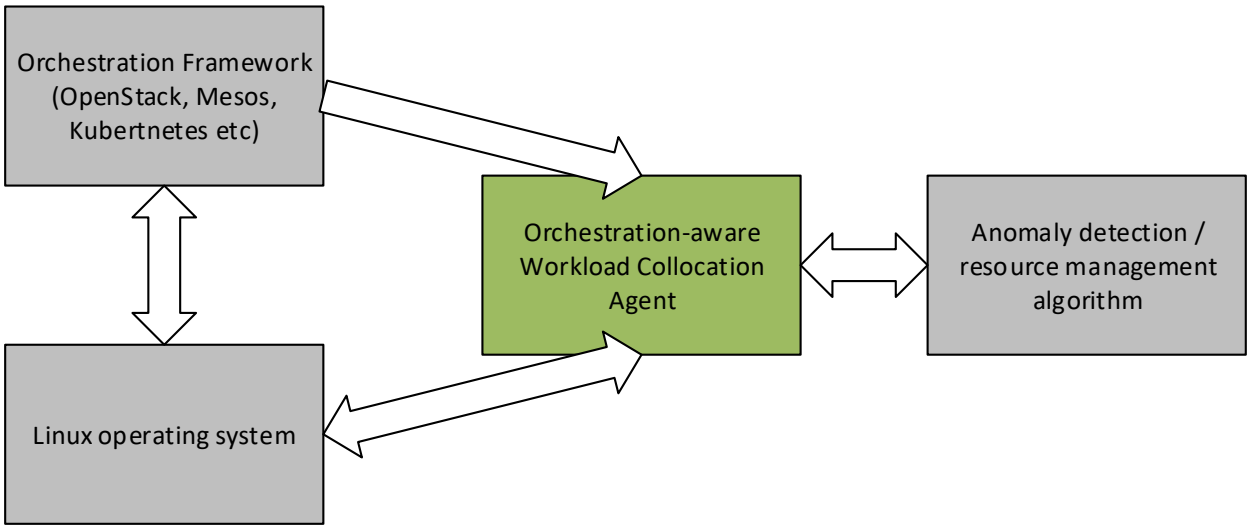

Figure 3 Software modules

The OWCA abstracts compute node and running workloads as well as monitoring and resource allocation. An externally provided algorithm is responsible for allocating resources or anomaly detection logic. OWCA and the algorithm exchange information about current resource usage, isolation actuations and detected anomalies. OWCA can save all collected metrics to Kafka message queue.

Wrapper

Wrapper parses standard output or standard error of launched application and exposes APMs in Prometheus format using HTTP or by storing them in Kafka. Output of the application can be parsed using regular expression rules that can be provided using command line argument or by implementing custom parser.

If custom wrapper needs to be implemented, then `owca.wrapper` module should be used to integrate the implementation with process launching and metrics exposition features.

Wrapper can generate, beside `sli` and `slo`, normalized metric `sli_normalized` that equals to quotient of SLI and SLO.

Monitoring information flow

If configured OWCA periodically stores internal state in Kafka for validation, debugging and alerting purposes. Data is stored in Kafka using Prometheus text exposition format[2]. Messages from Kafka can be consumed and inserted to Prometheus using custom made kafka consumer (not included). Kafka is introduced as durable storage that protects data from being lost accidentally and can be replaced with any application that can serve such purpose.

[2] https://github.com/prometheus/docs/blob/master/content/docs/instrumenting/exposition_formats.md#text-based-format

Example Kafka message:

```
# HELP instructions Instructions per second.
# TYPE instructions gauage
instructions{application="cassandra",aurora_tier="preemptible",cores="44",cpus="88",env_uniq_id="16",host=
"igk-0104",instance="127.0.0.1:9099",job="owca_metrics",load_generator="ycsb",name="cassandra--9042",owca_
version="0.1.dev157+ga46948f",own_ip="100.64.176.16",sockets="2",task_id="root-staging16-cassandra--9042-0
-2c980994-2aea-4ddc-a033-d71f6dd448b8",workload_uniq_id="9042"} 48587678829397 1530024506211

# HELP cache_misses
# TYPE cache_misses counter
cache_misses{application="cassandra",aurora_tier="preemptible",cores="44",cpus="88",env_uniq_id="16",host=
"igk-0104",instance="127.0.0.1:9099",job="owca_metrics",load_generator="ycsb",name="cassandra--9042",owca_
version="0.1.dev157+ga46948f",own_ip="100.64.176.16",sockets="2",task_id="root-staging16-cassandra--9042-0
-2c980994-2aea-4ddc-a033-d71f6dd448b8",workload_uniq_id="9042"} 64593346768 1530024506211

…
```

Message can contain following metrics:

- platform (resource metrics) labeled with workloads names:
  - cache usage (Intel RDT),
  - memory bandwidth usage (Intel RDT),
  - CPU usage (both scaled and absolute values in seconds, collected cgroups cpuacct controller),
  - cache misses, instruction, cycles (collected using Linux perf counters),
  - number of CPU quota throttles (collected using cgroup CPU controller from Linux bandwidth control),
  - scheduling policy assigned to all processes of workload
- actuations for resource isolations labeled with workloads names:
  - number of CPUs assignment,
  - number of CPU shares,
  - amount of cache ways allowed (both scaled and absolute),
  - assigned scheduling policy for workload,
  - quota for CPU time (cpu-period / cpu-quota normalized to number of CPUs assigned)
- alerts
  - resource contention
  - anomaly detection

Mesos integration description

Agent communicates with *Mesos Agent* to discover scheduled workloads and create instances of objects representing node state. The objects are passed to resource allocation or anomaly detection algorithms. Agent connects to Mesos Agent http endpoint at **http(s)://localhost:5051/api/v1** periodically and sends `GET_STATE` request. Collected information is used to recreate *cgroup* path that processes run in.

Diagrams

New workload synchronization loop sequence diagram

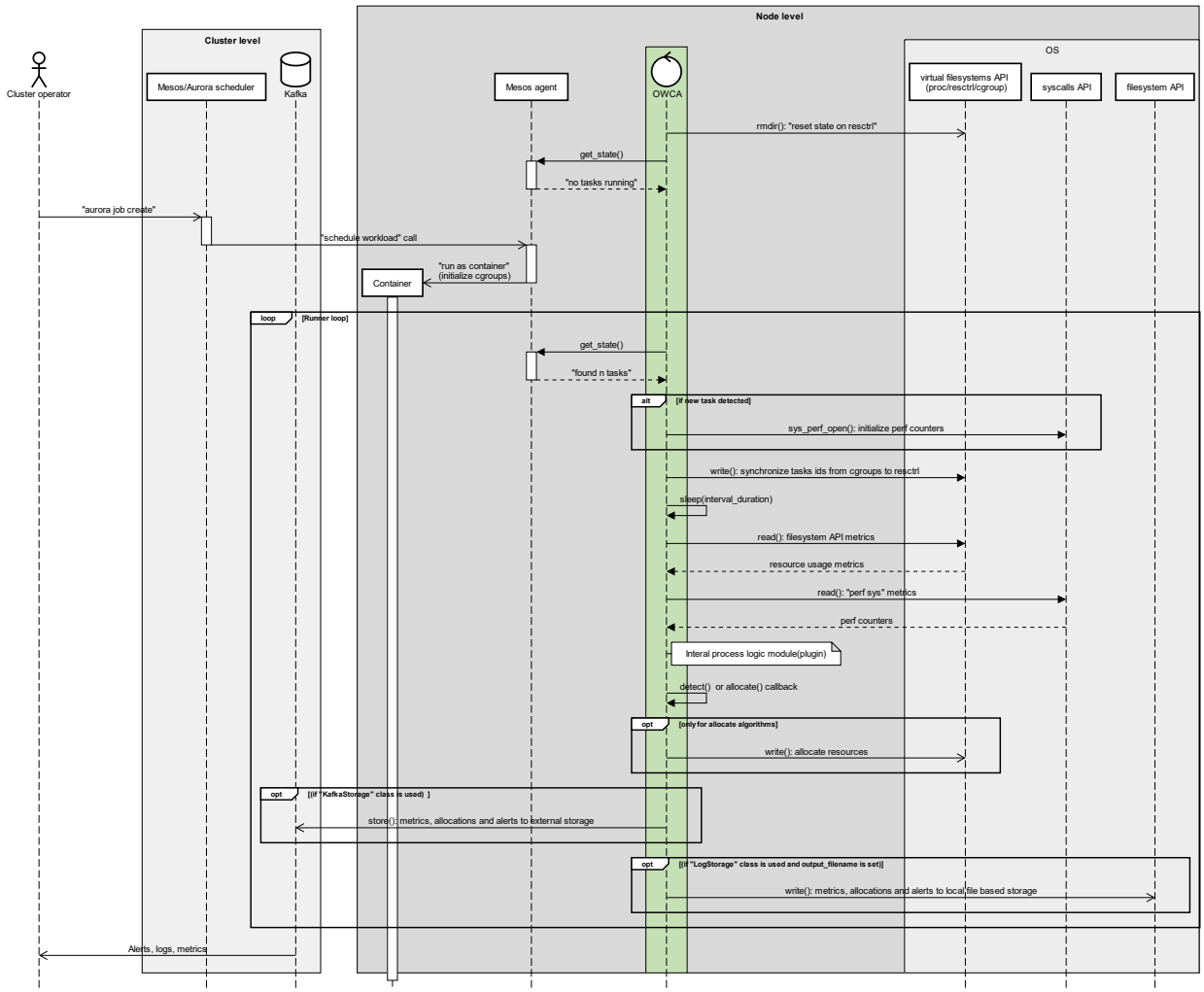

Figure 4 "New workload detected" case - sequence diagram

The sequence diagram above describes how OWCA communicates with external entities when a new workload is detected. The agent synchronizes the state of workloads running on the node with OS subsystems responsible for collection of resources and platform metrics. The agent analyses metrics and may generate alerts. Results of this analysis are sent to storage (to Kafka or to a local file). Optionally allocation and isolation of resources is performed.

The depicted flow is:

a. The agent is run, and a cleanup procedure is performed (in case of unexpected program termination).
b. The agent gets state of currently running tasks from a Mesos Agent; the OWCA agent initializes internal structs.
c. An operator submits a new Aurora job; the action results in a new Mesos task being run on the node where the agent is run as a container.
d. The agent starts to execute the "runner loop":

- Gather information about tasks from Mesos Agent.
- If a new task is detected, initialize Linux perf event subsystem is performed to enable PMU events collections.
- For each task the agent synchronizes a list of processes identifiers between *cgroups* and *resctrl* filesystem.
- The agent sleeps for configured period to allow OS to collect enough information about requested metrics.
- *resctrl*, *cgroups, proc fs* based metrics are read from OS and passed to detection or allocation module.
- Based on response from detection or allocation module new metrics are created.
- If allocation is enabled, resources configuration is performed.
- All metrics collected (including detection and allocation metrics) are stored in configured destination.

Internal and external communication flow

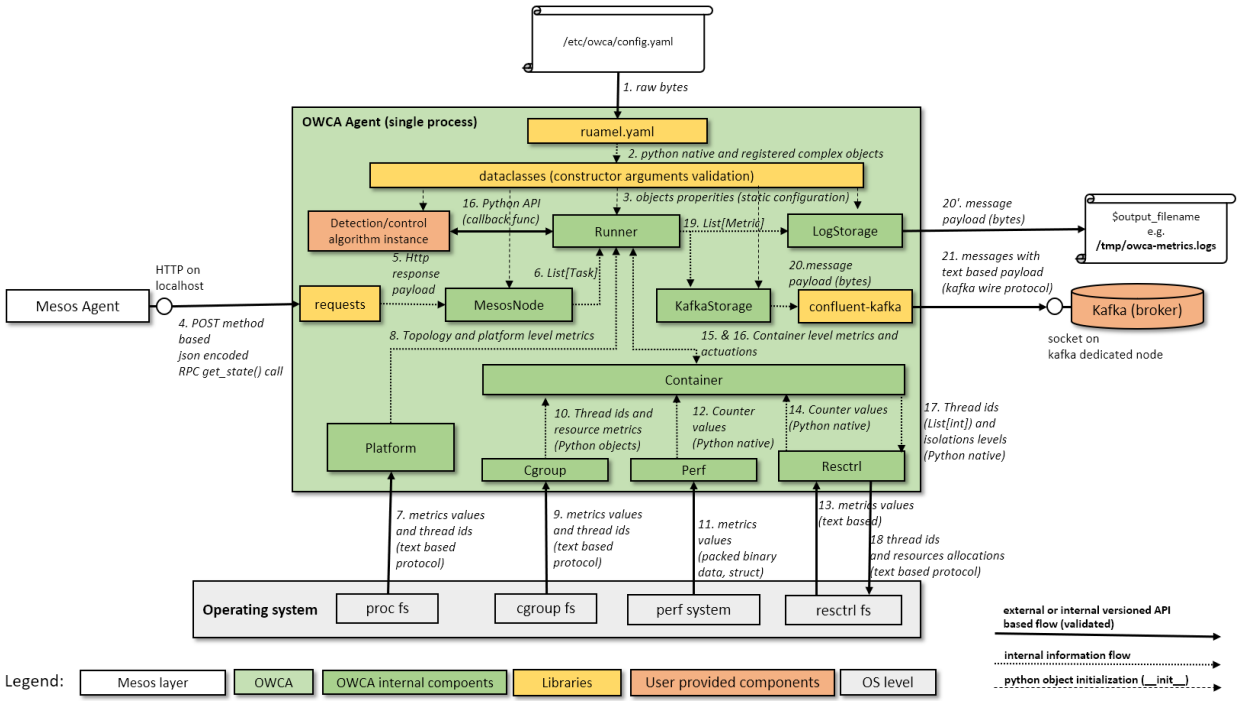

Figure 5 Internal & external communication flow

"Internal and external communication" flow diagram depicts communication that is performed with 1) external entities during initialization and 2) between internal components during execution of the "runner loop". The following information is passed between components:

1. Configuration (e.g. *"/etc/owca/owca.yaml"*) as raw bytes is read from configuration file and parsed by *ruamel.yaml* library.
2. Python raw objects, representing configuration, are used to build instances of registered python classes (*dataclasses* library generates python classes that perform validation of parameters).

3. Based on configuration, registered classes are filled with properties during initialization – instances of *MesosNode*, *Runner, Storage and Detector/Allocator* classes are created.
4. MesosNode sends "GET_STATE" HTTP(s) POST request to Mesos Agent.
5. "GET_STATE" method returns information about currently running tasks.
6. List of tasks as python objects are passed to *Runner* loop; *Container* instances are created.
7. Based on metrics read from OS *proc* fs *Platform* object is created.
8. *Platform* object containing topology and platform level metrics is passed to *Runner*.
9. *cgroup fs* metrics are read and abstracted by *Cgroup object.*
10. *Cgroup object* abstracts access to *cgroup* fs and allows to gather metrics and identifiers of *threads and processes*.
11. Data is read from Linux perf event system using *read()* on file descriptors provided by *sys_perf_open()* syscalls.
12. Parsed data, in form of metrics values (counters), is exposed by *Container* abstraction.
13. Data read from *resctrl fs* and parsed by *Perf* class is exposed by *Container* abstraction in a consistent way.
14. Metrics are returned to Runner and send to "Detection" or "Control" module.
15. Depending on mode of operation (*allocate()* or *detect())* callback is executed; returned results are stored by a *Storage* class instance.
16. Allocations per individual task are executed over *Container* abstraction.
17. Thread's and processes identifiers, as python objects, are passed to *Resctrl* class.
18. Thread and process identifiers are written in required text-based form to *resctrl* fs to enable metrics collection
19. List of *Metric* instances are send to *Storage*.
20. If OWCA is configured to use *LogStorage then metrics are formatted in* "Prometheus exposition format" and written to standard logs (using built-in logging functionality) or appended to a file (a path to the file might be chosen in the config).
21. If OWCA is configured to use *KafkaStorage* then encoded metrics as payloads are sent to configured Kafka broker (internal Kafka wire protocol is used to transport Kafka messages).

BOM for reference architecture.

Software components

Role	Name	Version	Description/Patch/Extra features/Source
Operating system	**Centos**	**7.5**	Latest available version
Orchestration	**Mesos**	**1.4.x**	Official release supported by Aurora
Framework	**Aurora**	**0.20.x**	
Containerizer	**Docker**	**1.13.x**	
Load generator	**YCSB**	**0.13.x**	- dynamic load generation - performance metrics exposed in Prometheus format
	rpc-perf	**2.0.3**	https://github.com/twitter/rpc-perf
Workload	**Memcached**	**1.5.7**	http://memcached.org/
Workload	**twemcache**		https://github.com/twitter/twemcache
Workload	**Cassandra**	**3.11.x**	
Workload	**SpecJBB®2015**	**1.0.1**	
Workload	**stress-ng**	**0.09.x**	http://kernel.ubuntu.com/~cking/stress-ng/
Runtime	**Python**	**3.6.x**	Source: http://fedoraproject.org/wiki/EPEL

BIOS settings

Option	Value
Hyperthreading	**On**
Power CAP	**On**
Sidestep Speed	**Manual**

Hardware configuration

Roles	Hardware components		
Controller, Infrastructure	Platform		**Broadwell**
	CPU	CPU	**Intel® Xeon® CPU E5-2699 v4**
		Speed	**2200 MHz**
		Number of CPUs	**2 CPUs, 22 Cores per CPU**
		Last level cache	**55 Mb**
		Example product	**Dell Inc. PowerEdge R630**
	Memory	Type	**DDR-4**
		Speed	**1866Mhz**
		Quantity	**24 (384 GB)**
	NIC	Speed	**10Gb**
		Example product	**Intel 82599ES** or **Ethernet 10G 2P X520**
Service worker	Platform		**Broadwell**
	CPU	CPU	**Intel® Xeon® CPU E5-2620 v4**
		Speed	**2100 MHz**
		Number of CPUs	**2 CPUs, 8 Cores/CPU**
		Last level cache	**20 Mb**
		Example product	**Dell Inc. PowerEdge R630**
	Memory	Type	**DDR-4**
		Speed	**1866Mhz**
		Quantity	**24 (384 GB)**
	NIC	Speed	**10Gb**
		Example product	**Intel 82599ES** or **Ethernet 10G 2P X520**
Load generator	Platform		**Broadwell**
	CPU	CPU	**Intel(R) Xeon(R) CPU E5-2620 v4**
		Speed	**2100 MHz**
		Number of CPUs	**2 CPUs, 8 Cores/CPU**

		Last level cache	**20 Mb**
		Example product	**Dell Inc. PowerEdge R630**
	Memory	Type	**DDR-4**
		Speed	**1866Mhz**
		Quantity	**24 (384 GB)**
	NIC	Speed	**10Gb**
		Example product	**Intel 82599ES** or **Ethernet 10G 2P X520**