200 YEARS GHENT UNIVERSITY

imec

IBM

Recent Advances in Container Orchestration: Network-Aware Scheduling in Container Clouds

José Santos, Chen Wang

FACULTY OF ENGINEERING AND ARCHITECTURE

IDLab INTERNET & DATA LAB

Who are we?

José Santos

Postdoctoral Researcher
at IDLab - Ghent University - imec

Research Interests:

- Cloud & Fog Computing
- SDN & NFV
- Service Function Chaining
- Machine Learning

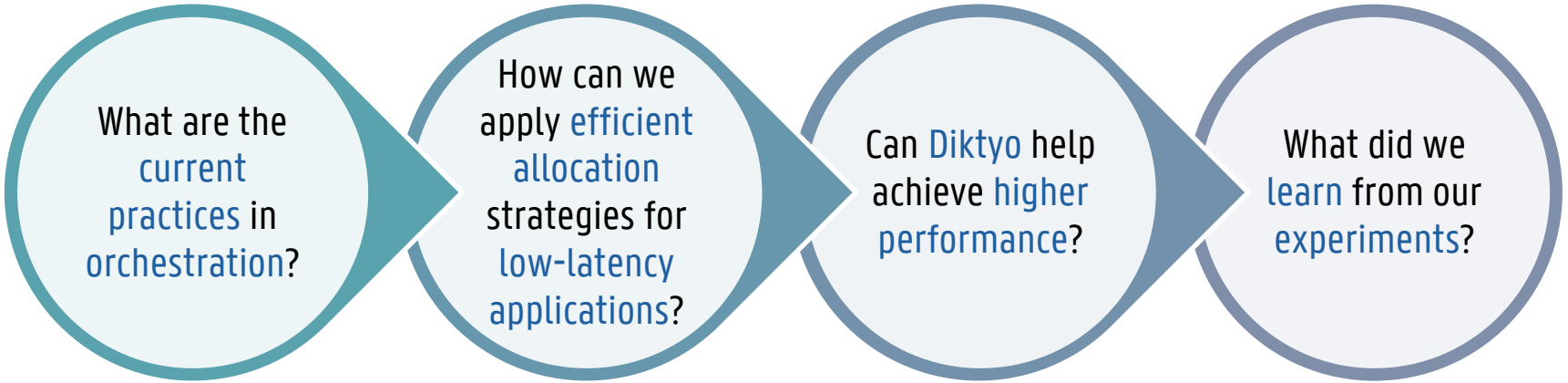
Chen Wang

Research Staff Member
at IBM Thomas J. Watson Research Center

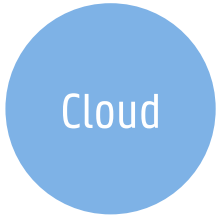
Research Interests:

- Cloud Orchestration
- Video Streaming systems
- Serverless Computing
- Machine Learning


CMA CGM
CMA CGM
tex


What are the current practices in orchestration?
How can we apply efficient allocation strategies for low-latency applications?
Can Diktyo help achieve higher performance?
What did we learn from our experiments?

Fog Computing vs Traditional Clouds



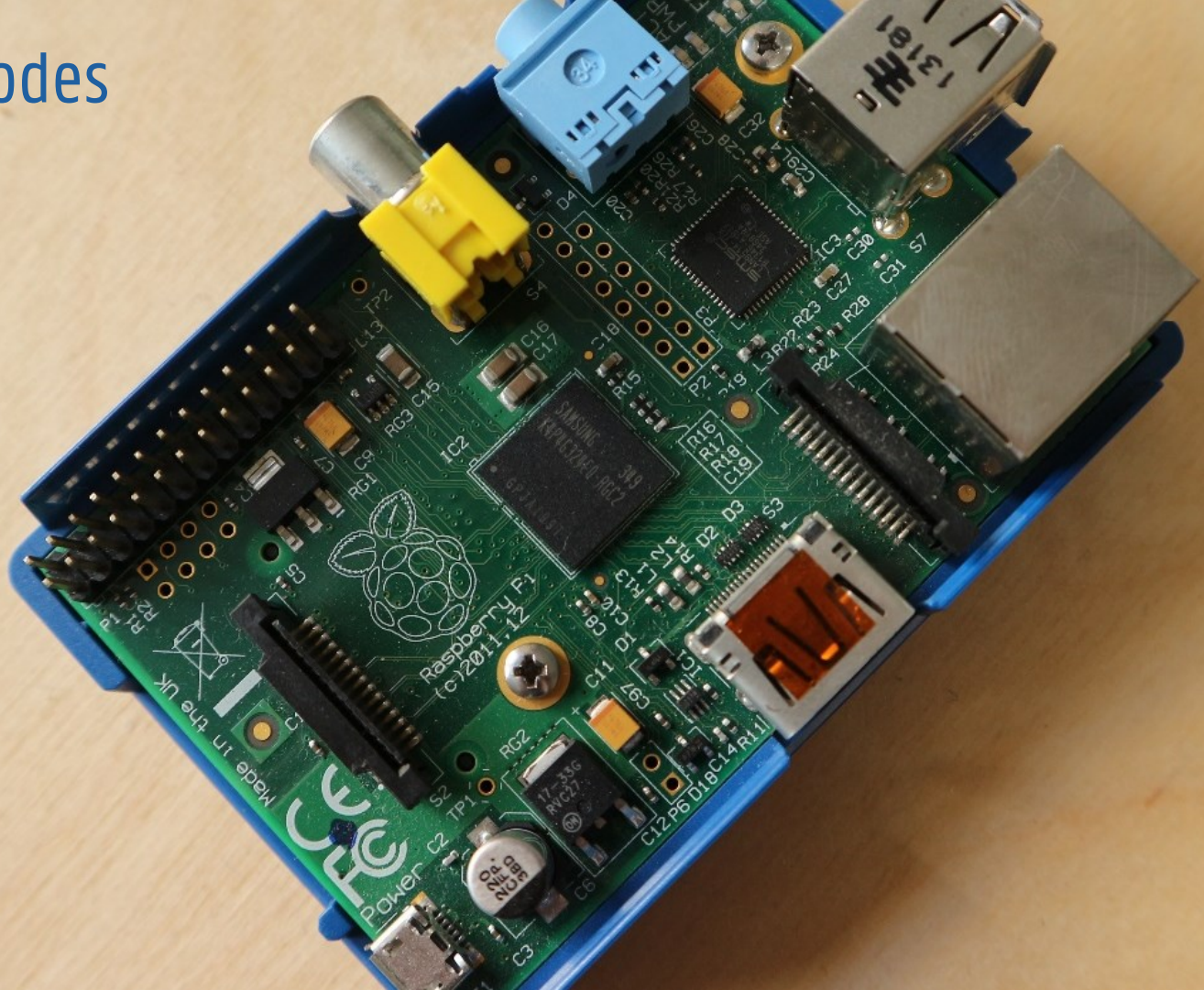
Heterogeneous resources: Edge and Fog nodes have lower capacity than cloud nodes

Bandwidth is scarce: Different types of connections among the edge and the fog.

Homogeneous resources: Typically, cluster nodes are similar in terms of resource capacity.

Similar bandwidth: Same type of network connections.

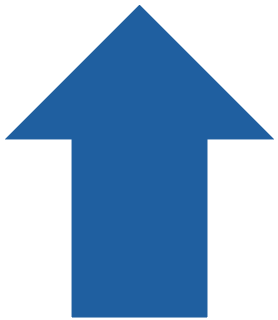
Edge Nodes



Fog Nodes

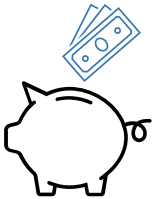
Cloud Nodes

Efficient Resource Allocation

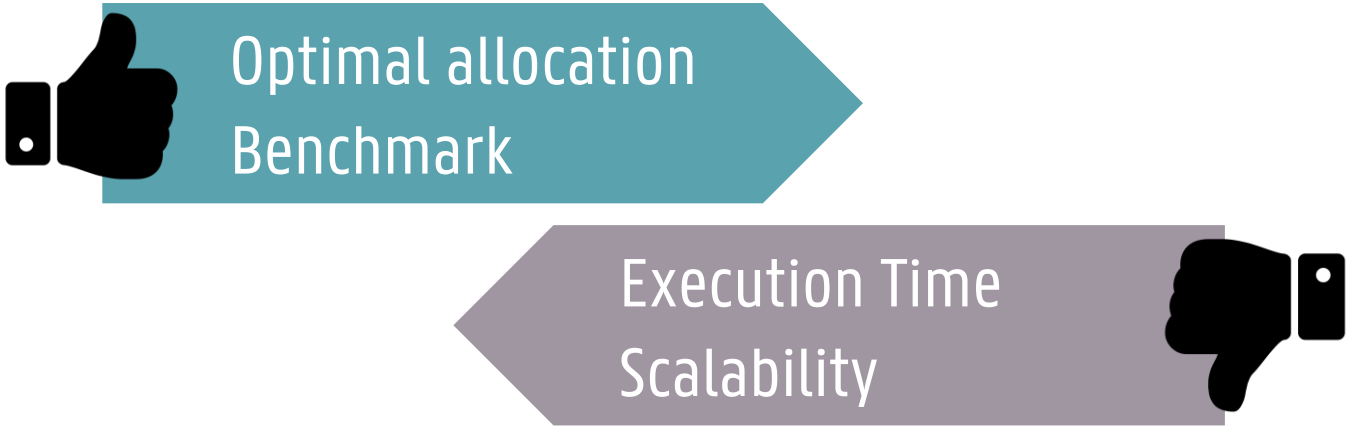


Quality of Experience

Costs



Previous work tackled resource allocation through theoretical modeling



Recent works focus mainly on…

	✓	✗
Priority-aware	• Meet deadlines. • Optimize resources.	• Low-priority tasks might suffer from starvation
Cost-aware	• Reduce deployment costs • Optimize resources	• Resource fragmentation • Performance trade-offs
Location-aware	• Reduce latency • Optimize resources	• Complexity grows with large-scale geo-distributed infrastructures
Network-aware	• Network efficiency • Optimize performance	• Complex algorithms • Overhead due to network monitoring
Topological-aware	• Enhance resource allocation • Improve data locality	• Dynamic topologies are challenging • Complex algorithms

Service Function Chaining (SFC)

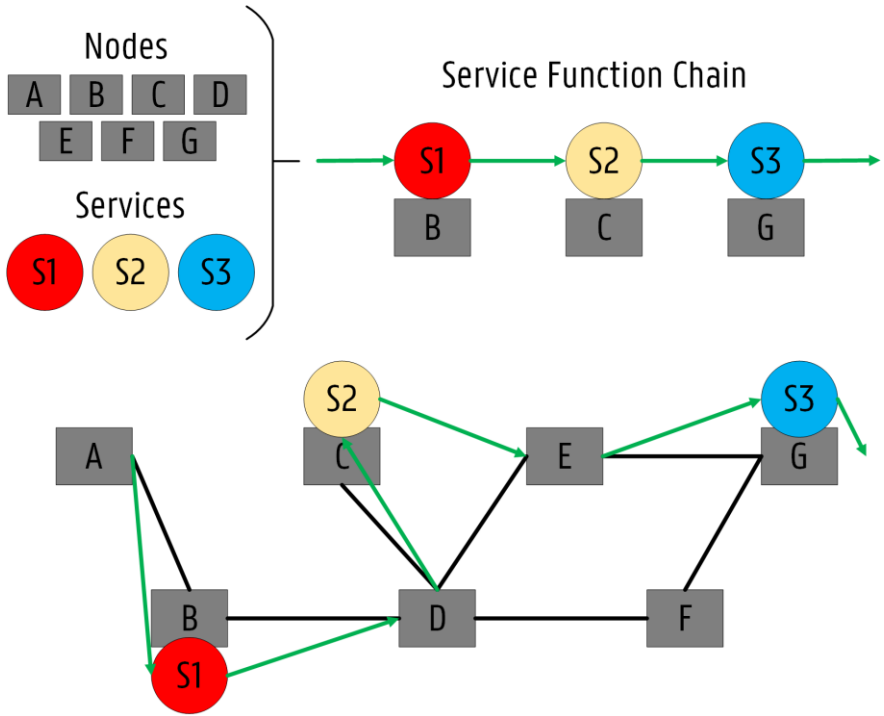
SFC

An ordered set of service functions and subsequent optimized traffic steering.

Improves Operational Efficiency:

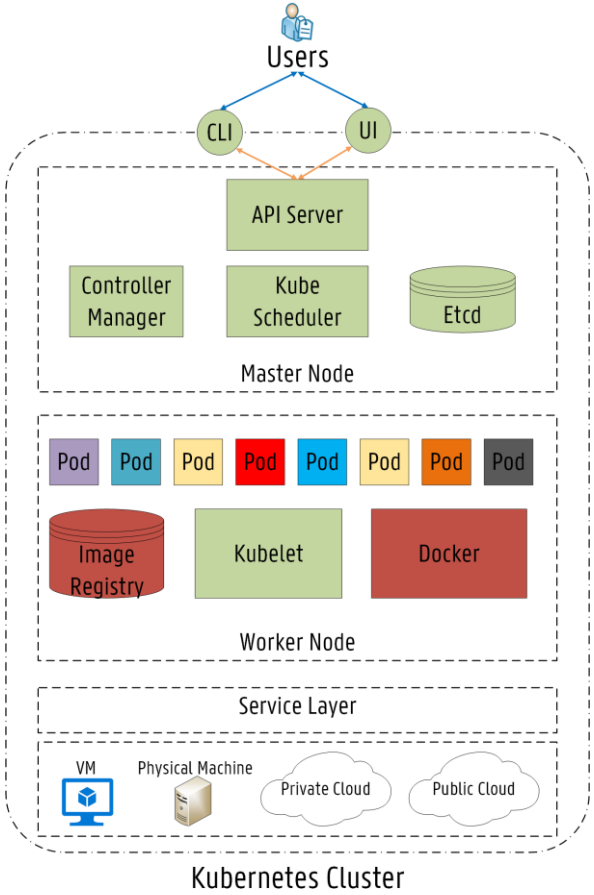
- Resource usage: applications are split into smaller services.
- Application performance.
- Traffic optimization.

Without proper resource allocation & traffic optimization, **performance can deteriorate**.

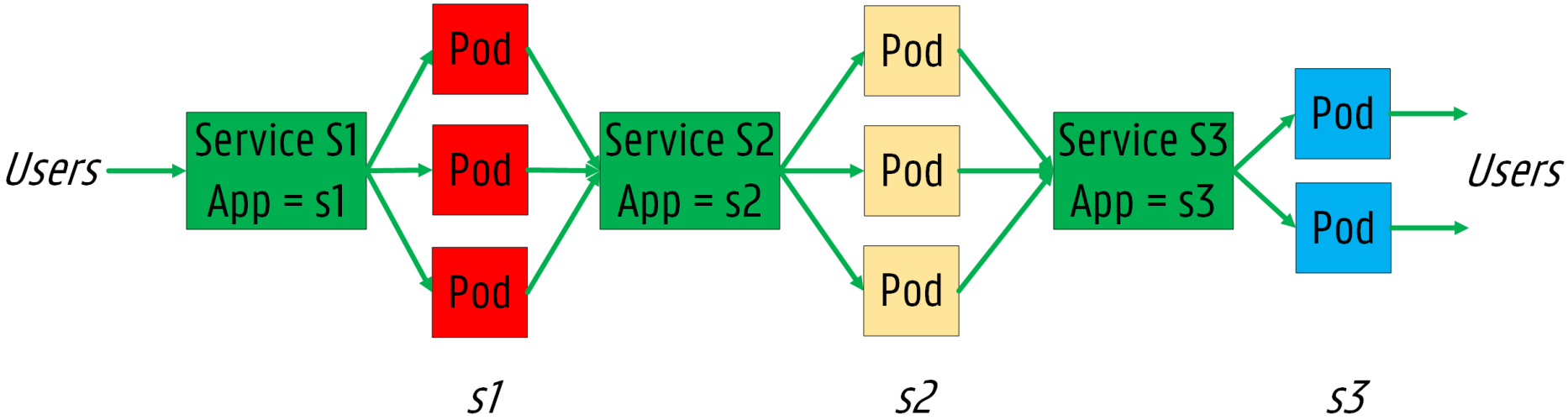


Kubernetes (K8s)

- Container Orchestration Platform
- Robust & Reliable
- Scaling up Containers (Docker)
- Massive Open-Source Community



SFC Concepts in K8s



Pod Lifecycle

Informers

Feed pods

Update (internal) cache

Queues

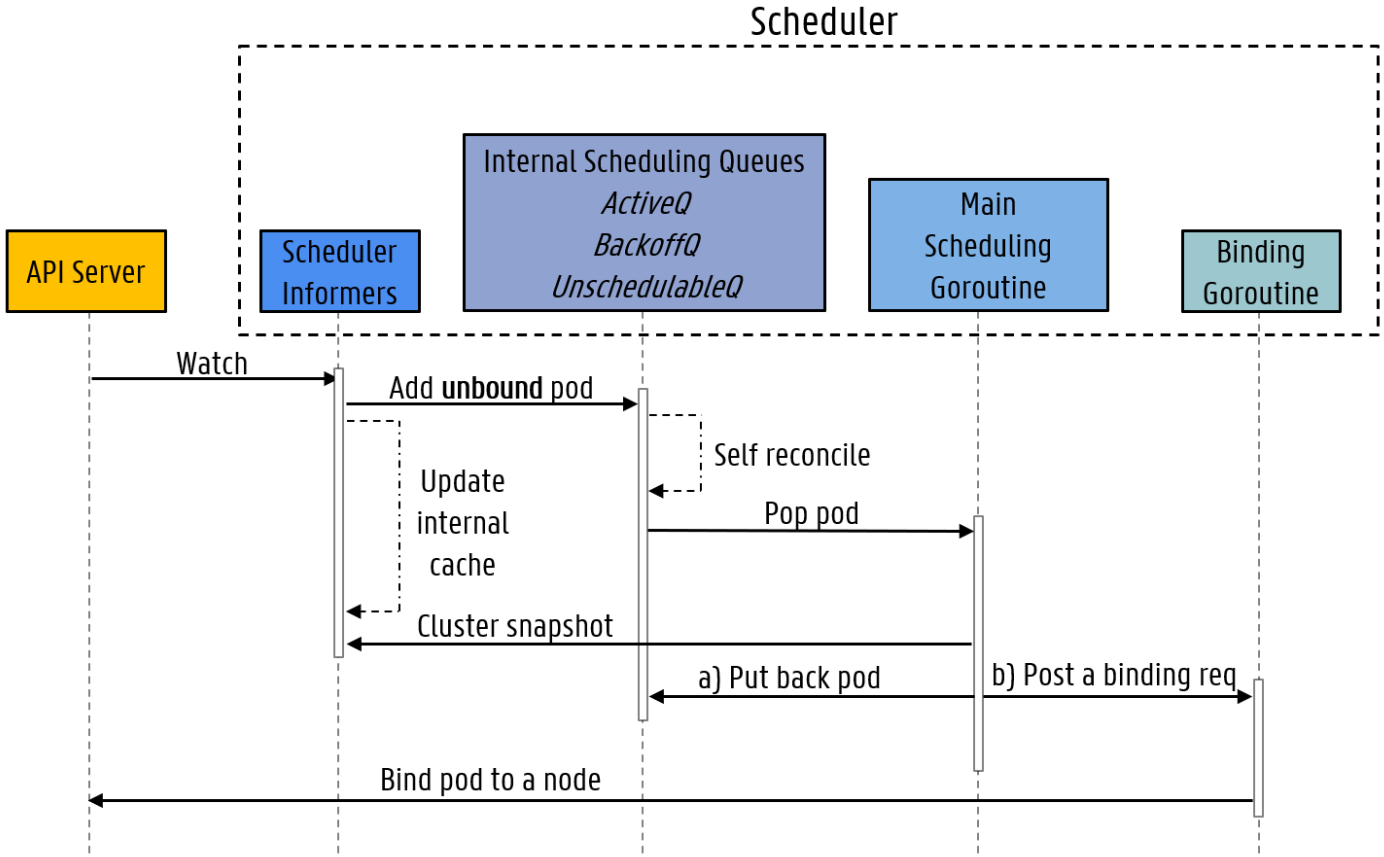
Pop pods

Fairness

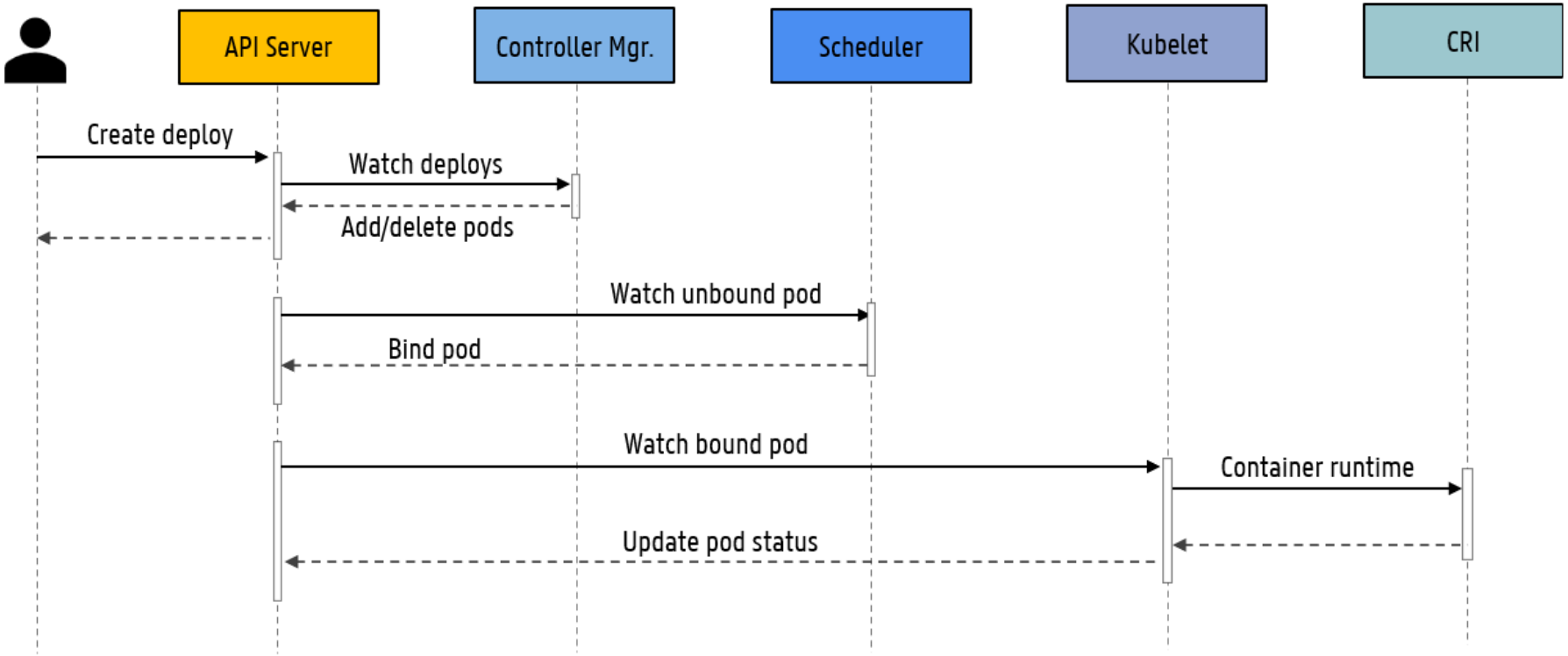
Core Scheduling

Queue pod back

Binding cycle



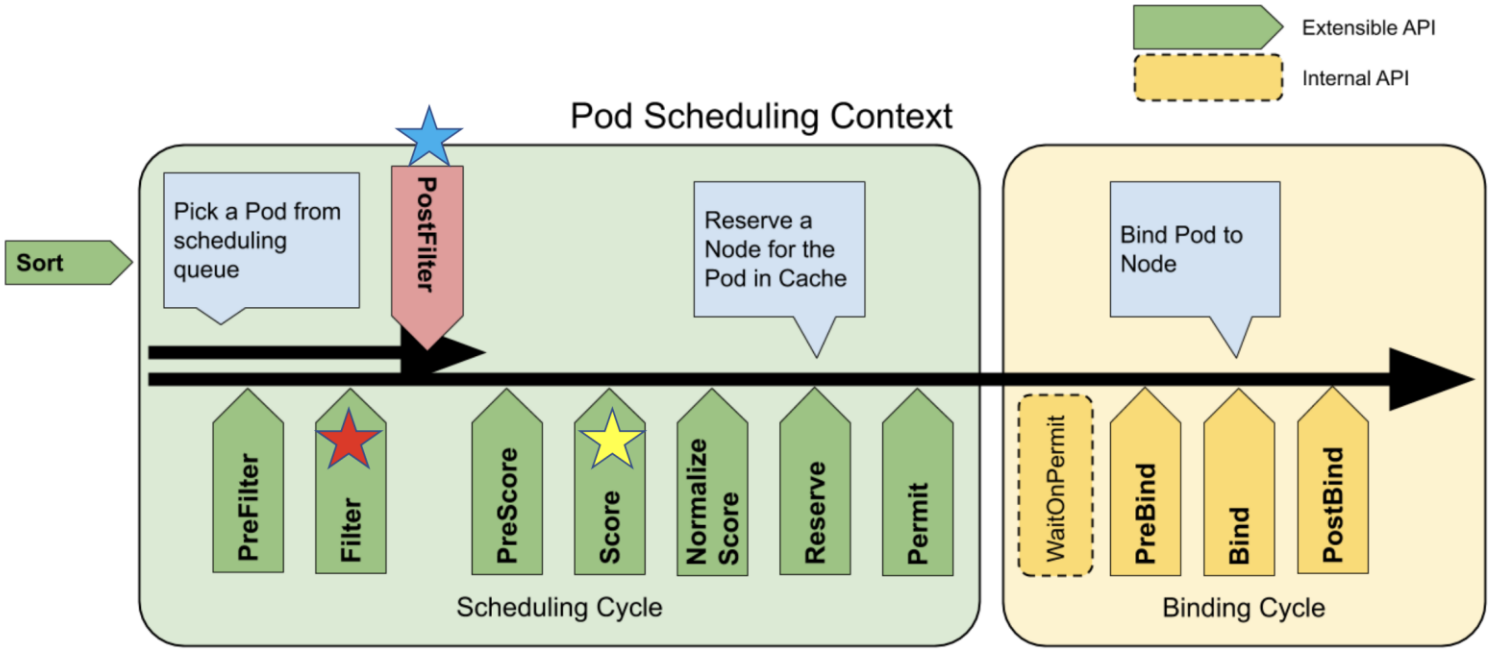
Scheduler Workflow Internals



Scheduling Framework

Hard constraints: filter, postfilter

Soft constraints: score (select the best node)



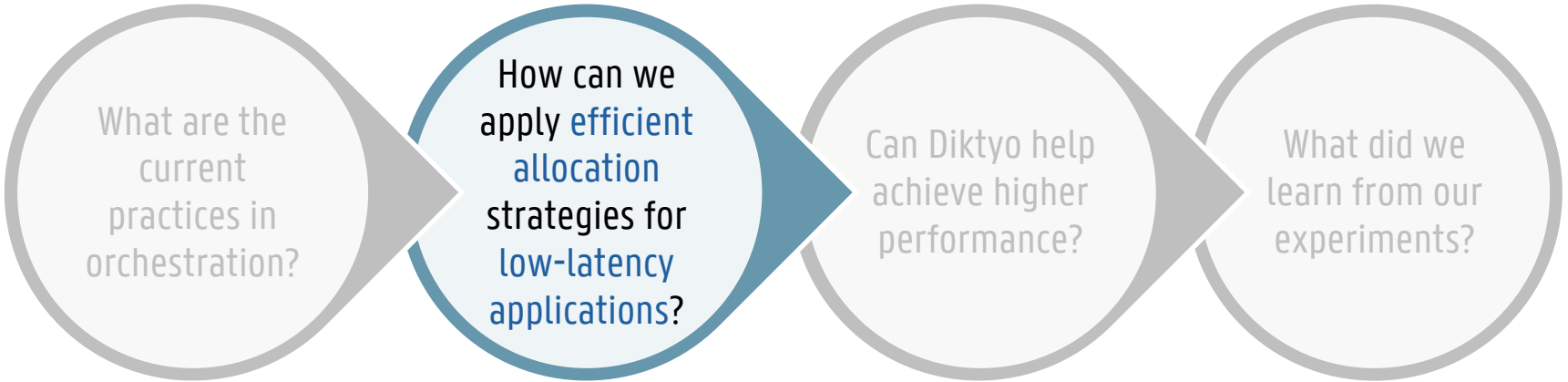
K8s scheduling

How good is the K8s scheduler?

Fast decision time

Best effort performance

Focus mainly on computing resources (e.g., CPU)


What are the current practices in orchestration?
How can we apply efficient allocation strategies for low-latency applications?
Can Diktyo help achieve higher performance?
What did we learn from our experiments?

Motivation

The K8s scheduler typically focuses on Resource Efficiency (e.g., CPU and Memory).

No contextual awareness about application dependencies or infrastructure topology.

Low latency plays a major role for several applications (e.g., IoT, video streaming).

How can we do better?

Consider latency and network bandwidth in the scheduling process. **But how?**

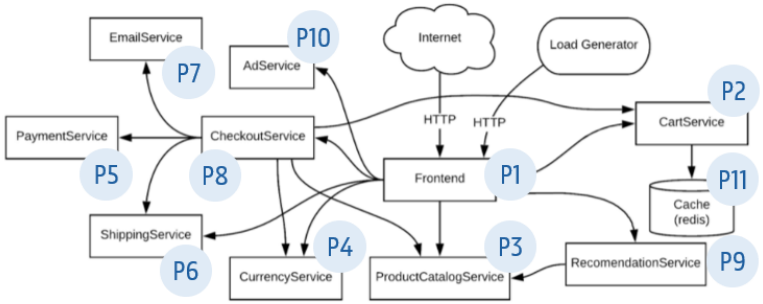
The Diktyo[1] Framework: Reducing latency in K8s → Demo today!

Diktyo supports additional scheduling plugins that address:

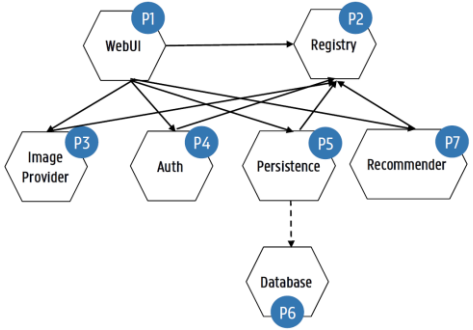
Application characteristics: microservice dependencies play a major role.

Infrastructure Topology: weights among cluster nodes based on different metrics.

Online Boutique



TeaStore



[1] J. Santos, C. Wang, T. Wauters, and F. De Turck, "Diktyo: Network-aware scheduling in container-based clouds," IEEE Transactions on Network and Service Management, 2023.

Use Cases / Topologies

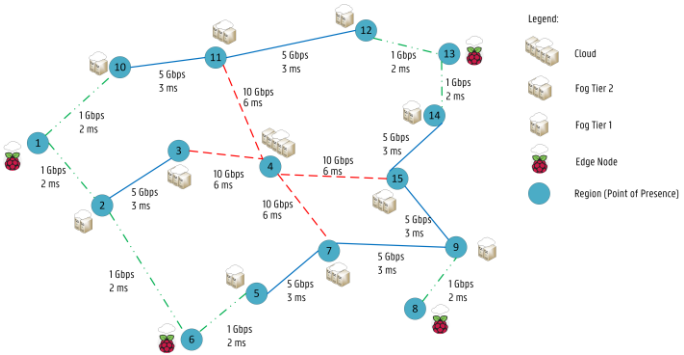
All kinds of topologies would benefit from our plugin.

High latency is a big concern in multi-region clusters.

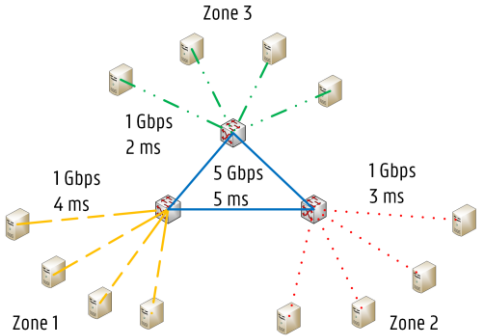
Even a small-scale cluster would benefit from latency-aware scheduling decisions.

Latency in Data centers is a critical requirement for certain applications (e.g., financial).

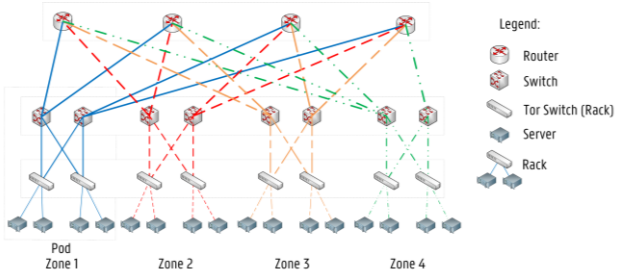
Multi-region



Cluster



Data center



Current plugins differ by not fully addressing microservice inter-dependencies and latency or bandwidth

TABLE I: Comparison among different scheduling plugins.

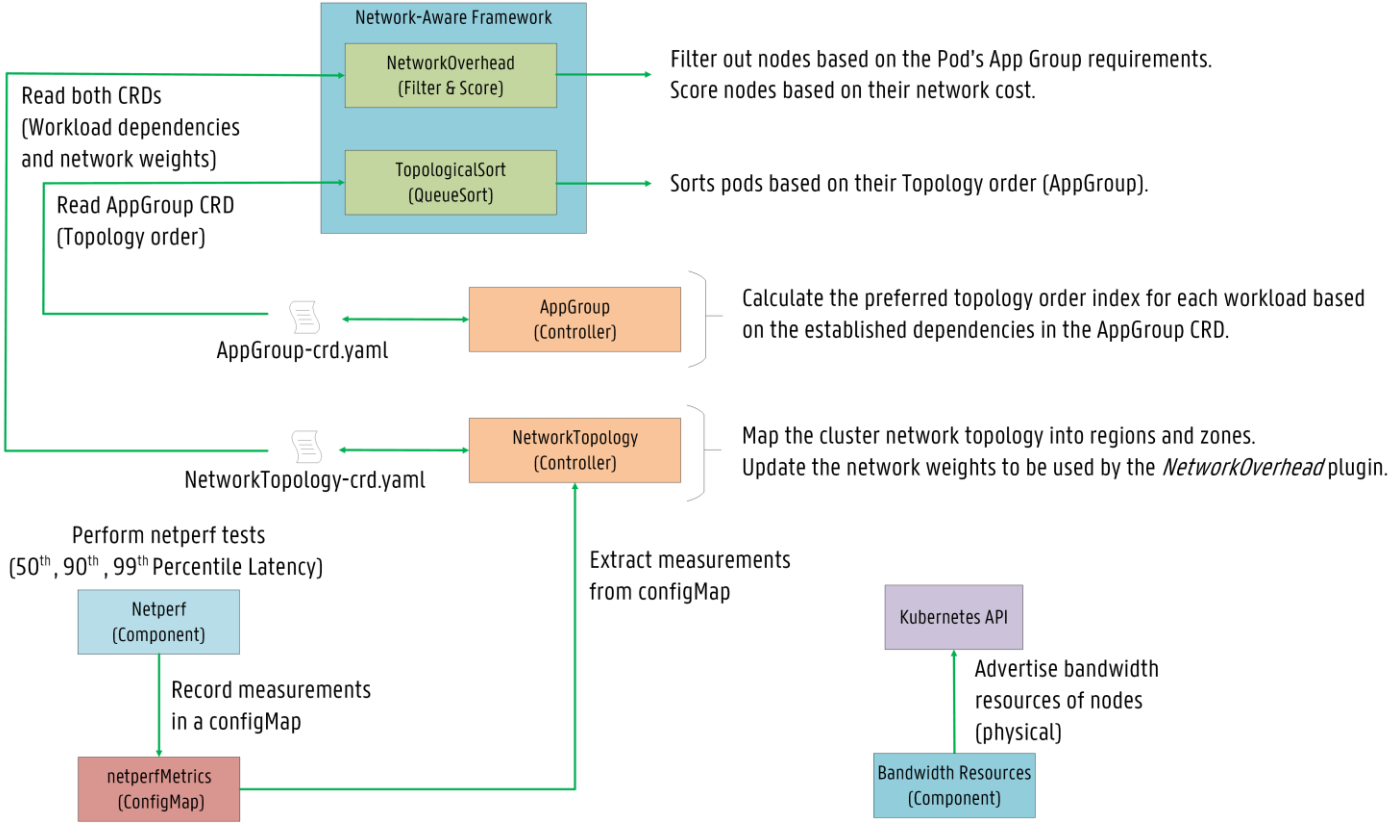
Plugin / Framework	Project	Extension Point	Scheduling Goal	Service Topology	Microservice Dependencies	Latency	Bandwidth
Gang	V	PF	AP & R	×	×	×	×
Task Topology	V	QS & PF & S	T & R	✓	×	×	×
Binpack	V	S	R	×	×	×	×
Dom. Res. Fairn. (DRF)	V	QS & PF	R	×	×	×	×
CoScheduling	S	QS & PF	AP	✓	×	×	×
Topology-aware	S	F	R & T	✓	×	×	×
Diktyo	D	QS & F & S	L & B	✓	✓	✓	✓

Project: V = Volcano, S = Scheduler Plugins, D = Diktyo framework.
Extension Point: QS= QueueSort, PF = PreFilter, F = Filter, S = Score.
Scheduling Goal: R = Resources, AP = App. Dependencies, T = Topology-aware, L = Latency, B = Bandwidth.
Service Topology, Microservice Dependencies, Latency, Bandwidth: ✓= addressed, ×= not considered.

Design & Implementation

Main ideas

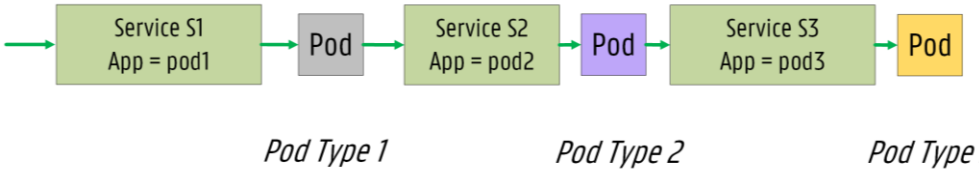
Diktyo framework – Design Overview

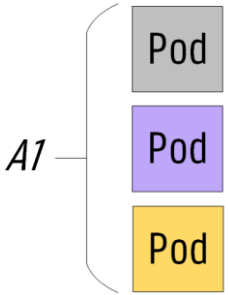


This design introduces an end-to-end solution to model/weight a cluster's network latency and topological information, and leverage that to optimize the scheduling of latency- and bandwidth-sensitive workloads.

Netperf Component: https://github.com/jpedro1992/pushing-netperf-metrics-to-prometheus
Development repo: https://github.com/jpedro1992/scheduler-plugins/tree/KepDevWithNTController

AppGroup CRD





Spec

```yaml
# Example App Group CRD spec
apiVersion: scheduling.sigs.k8s.io/v1alpha1
kind: AppGroup
metadata:
  name: a1
spec:
  numMembers: 3
  topologySortingAlgorithm: KahnSort
  workloads:
    - workload:
        kind: Deployment
        apiVersion: apps/v1
        namespace: default
        name: P1
      dependencies:
        - workload:
            kind: Deployment
            apiVersion: apps/v1
            namespace: default
            name: P2
          minBandwidth: "100Mi"
          maxNetworkCost: 30
    - workload:
        kind: Deployment
        apiVersion: apps/v1
        namespace: default
        name: P2
      dependencies:
        - workload:
            kind: Deployment
            apiVersion: apps/v1
            namespace: default
            name: P3
          minBandwidth: "250Mi"
          maxNetworkCost: 20
    - workload:
        kind: Deployment
        apiVersion: apps/v1
        namespace: default
        name: P3
```

Bandwidth and network requirements

Workload dependencies with specific affinities

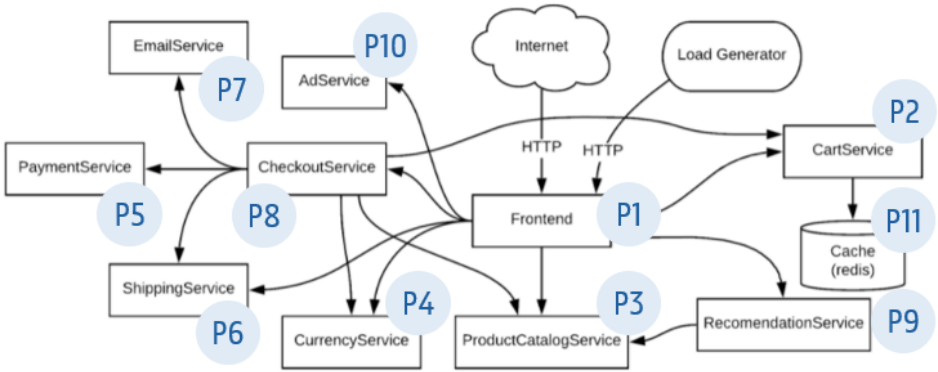
Status

Records preferred Topology order

The Impact of Queue Sorting

If applications have several microservices, how to select the first microservice to be allocated?

Consider OnlineBoutique (P1 – P11) shown below. Which microservice should be deployed first?



Diktyo Solution

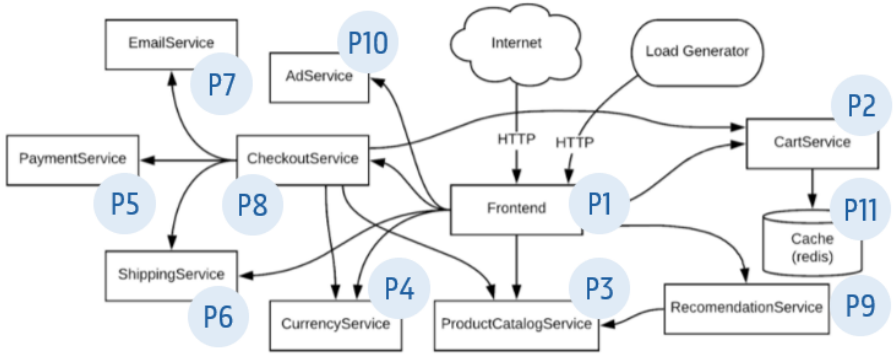
Develop a Queue Sort plugin that sorts pods based on Topological Sorting.

Pods belonging to an AppGroup should be sorted based on their topology information.

Significant differences are obtained depending on which topology algorithm is selected.

AppGroup Controller

- Updates the AppGroup CR regarding the preferred topology order for scheduling workloads.
- Six algorithms are currently supported.
- Records workload dependencies to be applied in the scheduling workflow.



Supported algorithms	
Sorting Algorithm	**Topological order**
Kahn	P1, P10, P9, P8, P7, P6, P5, P4, P3, P2, P11.
Alternate Kahn	P1, P11, P10, P2, P9, P3, P8, P4, P7, P5, P6.
Reverve Kahn	P11, P2, P3, P4, P5, P6, P7, P8, P9, P10, P1.
Tarjan	P1, P8, P7, P5, P4, P2, P11, P9, P10, P6, P3.
Alternate Tarjan	P1, P3, P8, P6, P7, P10, P5, P9, P4, P11, P2.
Reverve Tarjan	P3, P6, P10, P9, P11, P2, P4, P5, P7, P8, P1.

The inclusion of bandwidth in the scheduling process

Advertise bandwidth resources on all nodes (physical)

```
PATCH /api/v1/nodes/<your-node-name>/status HTTP/1.1
Accept: application/json
Content-Type: application/json-patch+json
Host: k8s-master:8080

[
  {
    "op": "add",
    "path": "/status/capacity/network.aware.com~1bandwidth",
    "value": "10000000000"
  }
]
```

The bandwidth CNI plugin supports pod ingress and egress traffic shaping

```
# Example Pod deployment with bandwidth limitations
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/ingress-bandwidth: 1M
    kubernetes.io/egress-bandwidth: 1M
```

```
# Example Pod deployment:
# bandwidth requests (Extended Resources)
# bandwidth limitations (bandwidth CNI plugin)
apiVersion: v1
kind: Pod
metadata:
  name: network-aware-bandwidth-example
  annotations:
      kubernetes.io/ingress-bandwidth: 10M
      kubernetes.io/egress-bandwidth: 10M
spec:
  containers:
  - name: network-aware-bandwidth-example
    image: example
    resources:
      requests:
        network.aware.com/bandwidth: 10000000 # 10M
      limits:
        network.aware.com/bandwidth: 10000000 # 10M
```

https://kubernetes.io/docs/tasks/administer-cluster/extended-resource-node/
https://kubernetes.io/docs/tasks/configure-pod-container/extended-resource

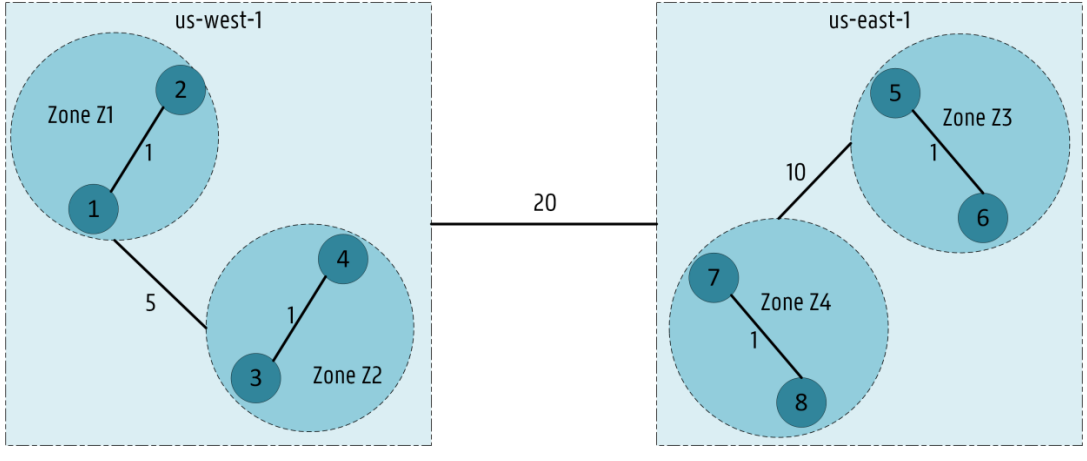
NetworkTopology CRD

Spec

```yaml
# Example Network CRD
apiVersion: scheduling.sigs.k8s.io/v1alpha1
kind: NetworkTopology
metadata:
  name: net-topology-test
  namespace: default
spec:
  configMapName: "netperfMetrics"
  weights:
    # Region label: "topology.kubernetes.io/region"
    # Zone Label:   "topology.kubernetes.io/zone"
    # 2 Regions:  us-west-1
    #             us-east-1
    # 4 Zones:    us-west-1: z1, z2
    #             us-east-1: z3, z4
    - name: "UserDefined"
      costList: # Define weights between regions or between zones
        - topologyKey: "topology.kubernetes.io/region" # region costs
          originCosts:
            - origin: "us-west-1"
              costs:
                - destination: "us-east-1"
                  bandwidthCapacity: "10Gi"
                  networkCost: 20
            - origin: "us-east-1"
              costs:
                - destination: "us-west-1"
                  bandwidthCapacity: "10Gi"
                  networkCost: 20
        - topologyKey: "topology.kubernetes.io/zone" # zone costs
          originCosts:
            - origin: "z1"
              costs:
                - destination: "z2"
                  bandwidthCapacity: "1Gi"
                  networkCost: 5
            - origin: "z2"
              costs:
                - destination: "z1"
                  bandwidthCapacity: "1Gi"
                  networkCost: 5
            - origin: "z3"
              costs:
                - destination: "z4"
                  bandwidthCapacity: "1Gi"
                  networkCost: 10
            - origin: "z4"
              costs:
                - destination: "z3"
                  bandwidthCapacity: "1Gi"
                  networkCost: 10
```

Region costs

Zone costs



Monitor latency through Netperf

90th Percentile Latency

Throughput Units	Throughput	Mean Latency Microseconds	Minimum Latency Microseconds	Maximum Latency Microseconds	50th Percentile Latency Microseconds	90th Percentile Latency Microseconds	99th Percentile Latency Microseconds	Stddev Latency Microseconds	Local CPU Util %
Trans/s	8389.90	118.32	109	582	116	128	170	20.90	5.56
Trans/s	7785.43	127.56	117	418	125	129	148	10.33	5.56
Trans/s	7960.96	124.64	107	1700	120	129	168	61.37	17.48
Trans/s	8351.08	118.89	110	587	115	124	166	23.60	14.17
Trans/s	7749.65	128.08	117	414	125	131	173	11.68	5.00
Trans/s	8082.83	122.80	109	1493	118	129	180	51.49	14.17
Trans/s	7150.57	138.76	111	4553	128	146	190	170.64	17.69
Trans/s	8351.22	118.93	103	1714	114	123	170	63.30	13.64

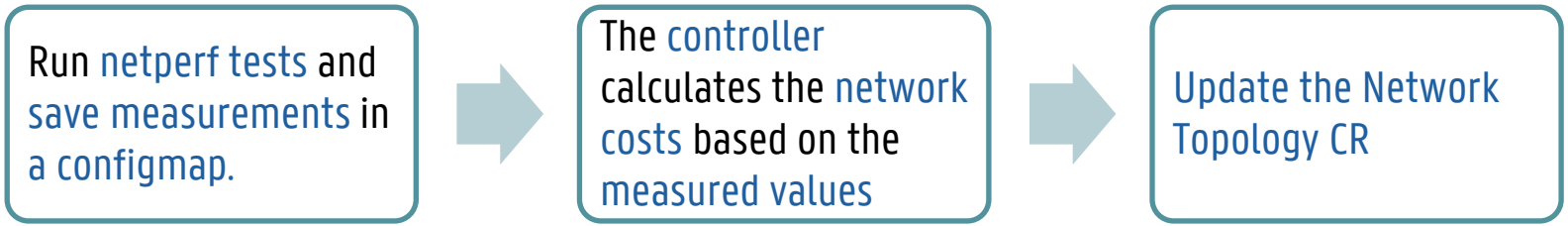
The netperf component is still lightweight for small to medium-sized clusters.

TABLE V: The traffic generated by the netperf component in a 16-node K8s cluster.

Mode	Number of measurements	Avg. received traffic (in Mbps)	Avg. transmitted traffic (in Mbps)
Basic	16	12.96 Mbps	13.44 Mbps
Full	240	33.52 Mbps	35.20 Mbps

NetworkTopology Controller

- Accesses the configmap to calculate network costs between regions and zones in the cluster.
- Keeps a record of the allocated bandwidth between zones and regions based on the deployed pods



Network Aware Scheduling Plugins

Design Details & Examples

TopologicalSort

Extension Point: QueueSort

- It compares pods based on their AppGroup CR.
- Get the pod index of both pods.
- Lower order is better: should be allocated first.
- If Pods do not belong to the same AppGroup, follow the strategy of the priority (QueueSort) plugin.

Application Group

Pod
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
P11

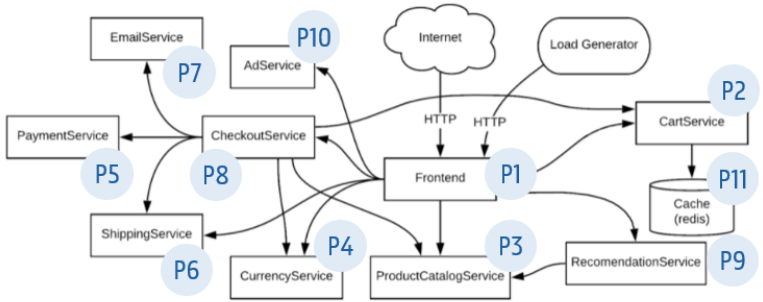
Topology Order

name	index
P1	1
P10	2
P9	3
P8	4
P7	5
P6	6
P5	7
P4	8
P3	9
P2	10
P11	11

pInfo1.Pod	pInfo2.Pod	orderP1 > orderP2	Result
P1	P9	FALSE	TRUE
P6	P2	FALSE	TRUE
P8	P10	TRUE	FALSE
P4	P6	TRUE	FALSE
P2	P3	TRUE	FALSE

Diktyo's Topological Sorting: Example

Online Boutique



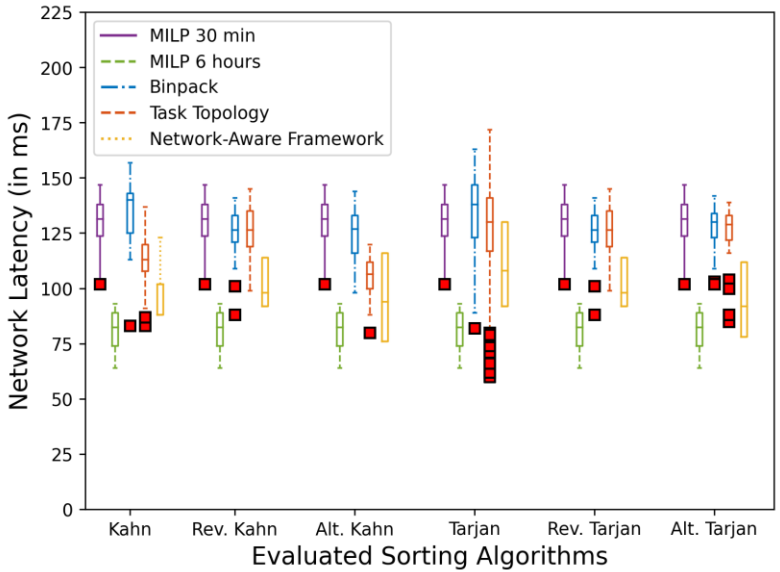
Online Boutique (OB) application (11-deployments)	
Kahn	[P1, P10, P9, P8, P7, P6, P5, P4, P3, P2, P11]
Alt. Kahn	[P1, P11, P10, P2, P9, P3, P8, P4, P7, P5, P6]
Rev. Kahn	[P11, P2, P3, P4, P5, P6, P7, P8, P9, P10, P1]
Tarjan	[P1, P8, P7, P5, P4, P2, P11, P9, P10, P6, P3]
Alt. Tarj.	[P1, P3, P8, P6, P7, P10, P5, P9, P4, P11, P2]
Rev. Tarj.	[P3, P6, P10, P9, P11, P2, P4, P5, P7, P8, P1]
Default (Cycle)	[P1, P2, P3, P4, P5, P6, P7, P8, P9, P10, P11]
K8s *Priority & QoS*	[P10, P2, P8, P4, P7, P1, P5, P3, P9, P11, P6]

8 different deployment orders!

Diktyo's Topological Sorting: Performance[1] (Simulations)

Supported Algorithms	
Kahn	P1, P10, P9, P8, P7, P6, P5, P4, P3, P2, P11
Al. Kahn	P1, P11, P10, P2, P9, P3, P8, P4, P7, P5, P6
Rev. Kahn	P11, P2, P3, P4, P5, P6, P7, P8, P9, P10, P1
Tarjan	P1, P8, P7, P5, P4, P2, P11, P9, P10, P6, P3
Alt. Tarjan	P1, P3, P8, P6, P7, P10, P5, P9, P4, P11, P2
Rev. Tarjan	P3, P6, P10, P9, P11, P2, P4, P5, P7, P8, P1

We achieved lower latency with Kahn, Alt. Kahn and Alt. Tarjan for Online Boutique.

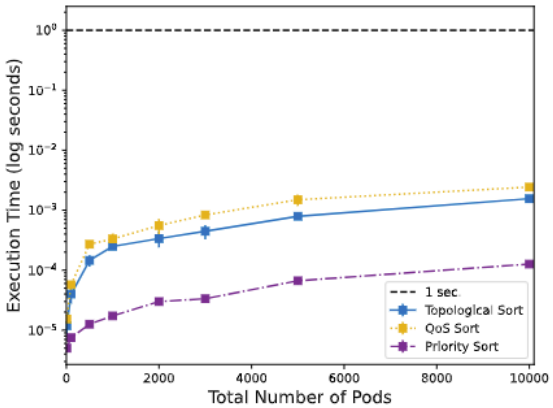


[1] J. Santos, C. Wang, T. Wauters, and F. De Turck, "Diktyo: Network-aware scheduling in container-based clouds," IEEE Transactions on Network and Service Management, 2023.

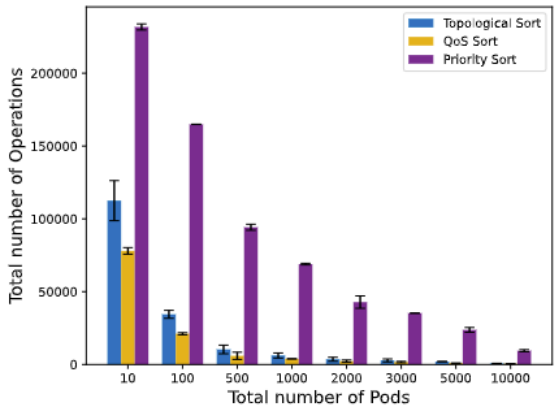
PrioritySort[1] is significantly faster than the QoSSort and TopologicalSort.

Time Complexity

Execution Time



Number of Operations

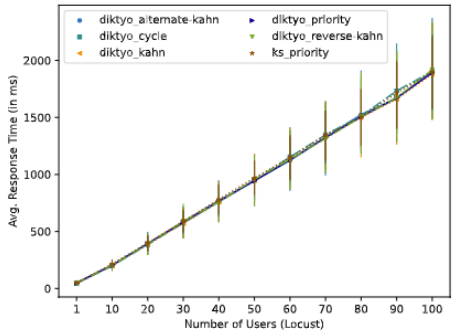


TopologicalSort can handle 10000 pods in about 1.57 ms while QoSSort and PrioritySort need 2.5 ms and 0.13 ms, respectively.

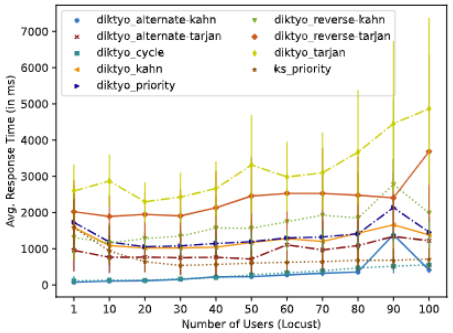
[1] Santos, J., Verkerken, M., D'hooge, L., Wauters, T., Volckaert, B., & De Turck, F. (2023, October). Performance Impact of Queue Sorting in Container-Based Application Scheduling. In 2023 19th International Conference on Network and Service Management (CNSM) (pp. 1-9). IEEE.

The higher the number of K8s deployments, the higher the impact of queue sorting.

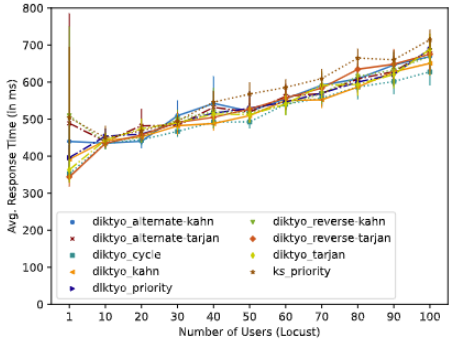
Latency



Cluster - IDS



Cluster - TS



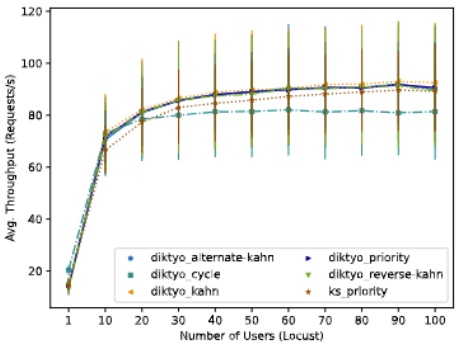
Cluster - OB

Small differences are observed for the Multi-Stage IDS, while discrepancies are more noticeable for the TS and OB.

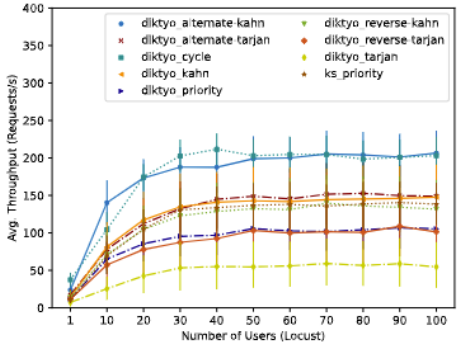
Throughput is significantly affected by queue sorting

Throughput

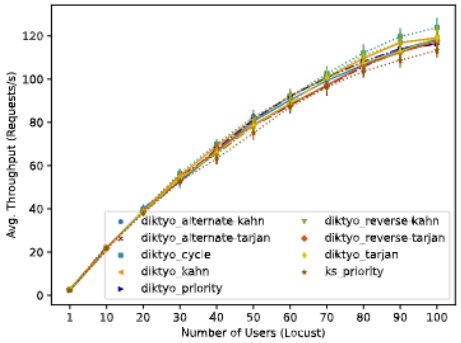
Substantial variations in attained throughput are observed across all evaluated applications, particularly when the number of emulated users exceeds 30.



Cluster - IDS



Cluster - TS



Cluster - OB

NetworkOverhead (Filter & Score)

Workload dependencies established in the AppGroup CR must be respected.
Several dependencies may exist since multiple pods can be already deployed on the network.

Extension Point: Filter → nodes are filtered out if they cannot support the network requirements of the Pod's AppGroup.

Extension Point: Score → scored based on network weights ensuring network latency is minimized for pods belonging to the same AppGroup.

NetworkOverhead (Filter & Score)

Extension Point: Filter

- It filters out nodes with high maxNetworkCosts requirements.
- Network weights are available in the Network Topology CR.
- Nodes that unmet a higher number of dependencies are filtered out to reduce the number of nodes being scored.

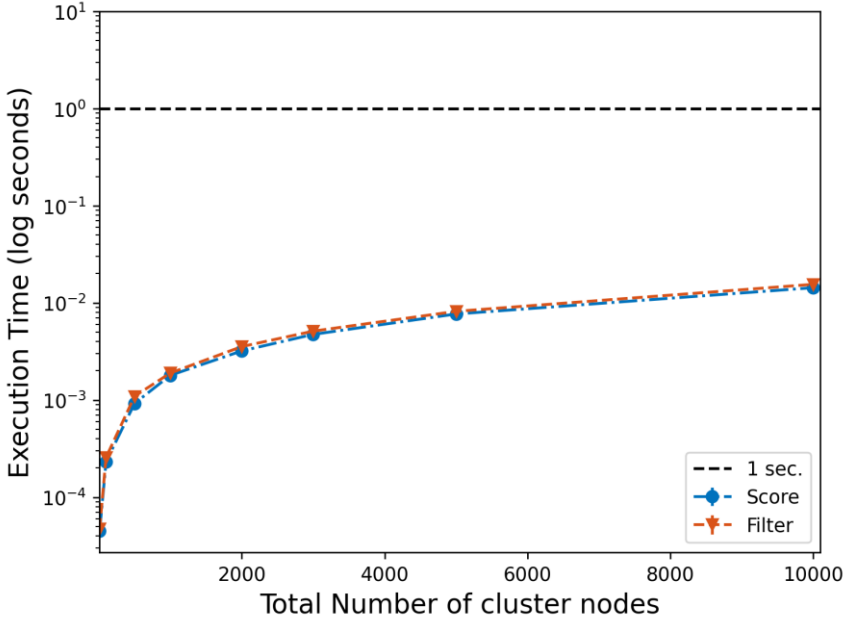
Extension Point: Score

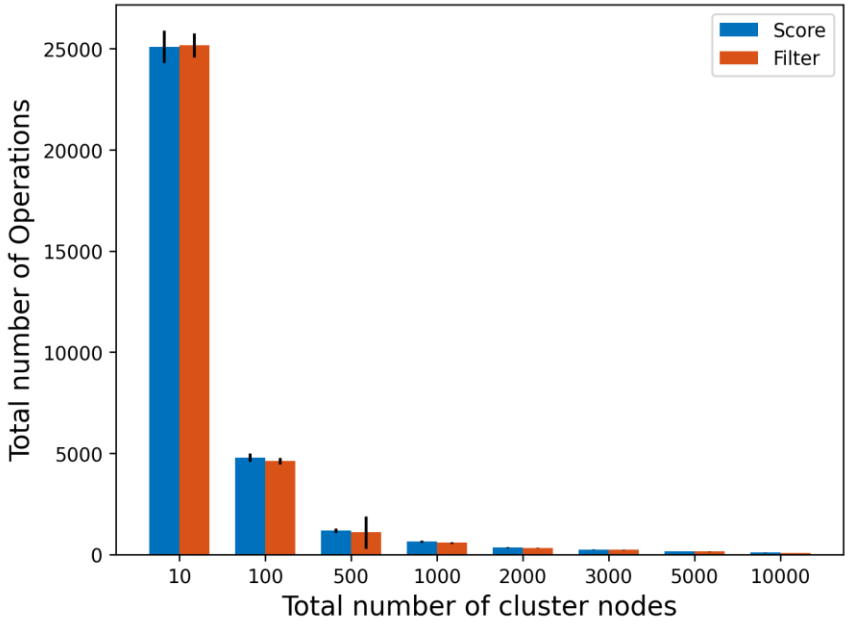
- It favors nodes with the lowest combined cost based on its AppGroup.
- Pod allocations are retrieved from the Scheduler Cache.
- Network weights are available in the Network Topology CRD. The default algorithm is the Dijkstra shortest path.

Typically, Dijkstra:
O (V + E log V)

V: Vertices
E: Edges

NetworkOverhead – Performance (Time Complexity)



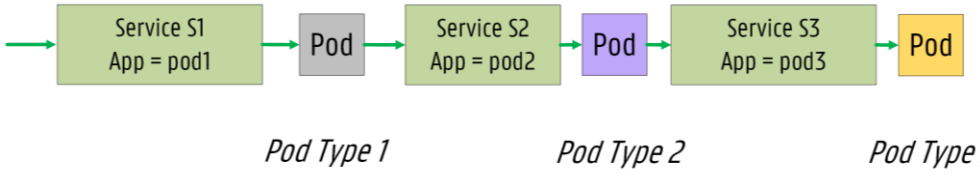


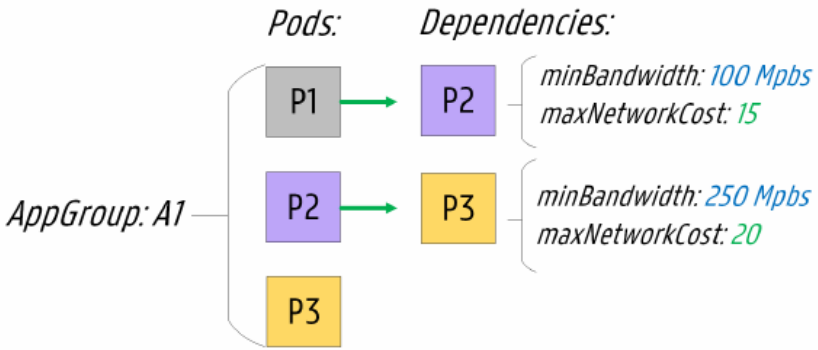
Logarithmic performance over the number of nodes
well < 1 sec for 10K cluster nodes.

Scheduling Workflow with Filter and Score plugins

Examples

NetworkOverhead – Example (AppGroup)





AppGroup CRD

A1: P1, P2, P3

Dependencies

P1: P2

P2: P3

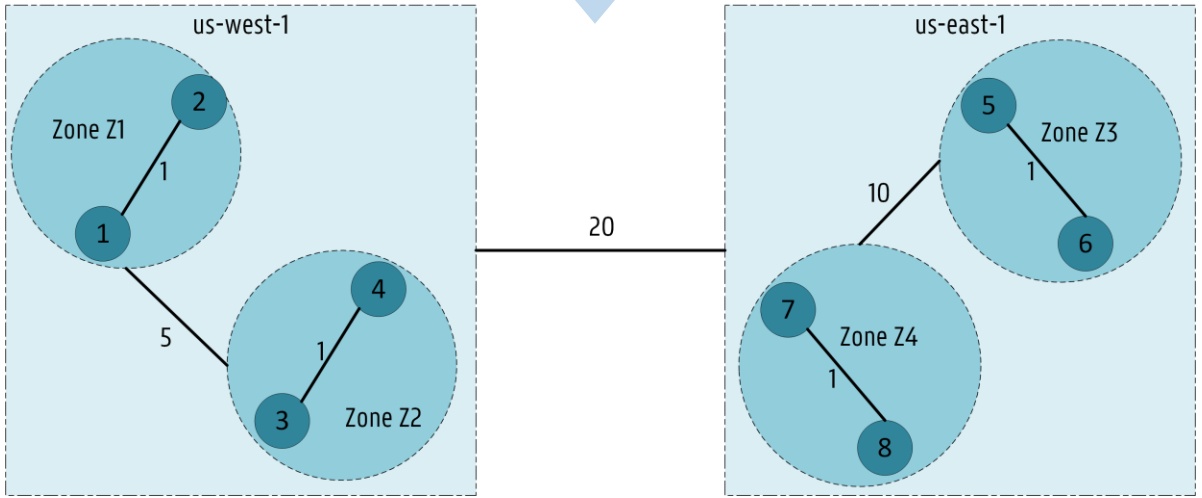
```yaml
# Example App Group CRD spec
apiVersion: scheduling.sigs.k8s.io/v1alpha1
kind: AppGroup
metadata:
  name: a1
spec:
  numMembers: 3
  topologySortingAlgorithm: KahnSort
  workloads:
    - workload:
        kind: Deployment
        name: P1-deployment
        selector: P1
        apiVersion: apps/v1
        namespace: default
      dependencies:
        - workload:
            kind: Deployment
            name: P2-deployment
            selector: P2
            apiVersion: apps/v1
            namespace: default
          minBandwidth: "100Mi"
          maxNetworkCost: 30
    - workload:
        kind: Deployment
        name: P2-deployment
        selector: P2
        apiVersion: apps/v1
        namespace: default
      dependencies:
        - workload:
            kind: Deployment
            name: P3-deployment
            selector: P3
            apiVersion: apps/v1
            namespace: default
          minBandwidth: "250Mi"
          maxNetworkCost: 20
    - workload:
        kind: Deployment
        name: P3-deployment
        selector: P3
        apiVersion: apps/v1
        namespace: default
```

NetworkOverhead – Example (NetworkTopology)

```yaml
# Example Network CRD
apiVersion: scheduling.sigs.k8s.io/v1alpha1
kind: NetworkTopology
metadata:
  name: net-topology-test
  namespace: default
spec:
  configmapName: "netperfMetrics"
  weights:
    # Region label: "topology.kubernetes.io/region"
    # Zone Label:   "topology.kubernetes.io/zone"
    # 2 Regions:  us-west-1
    #             us-east-1
    # 4 Zones:    us-west-1: z1, z2
    #             us-east-1: z3, z4
    - name: "UserDefined"
      topologyList: # Define weights between regions or between zones
        - topologyKey: "topology.kubernetes.io/region" # region costs
          originList:
            - origin: "us-west-1"
              costList:
                - destination: "us-east-1"
                  bandwidthCapacity: "10Gi"
                  networkCost: 20
            - origin: "us-east-1"
              costList:
                - destination: "us-west-1"
                  bandwidthCapacity: "10Gi"
                  networkCost: 20
        - topologyKey: "topology.kubernetes.io/zone" # zone costs
          originList:
            - origin: "z1"
              costList:
                - destination: "z2"
                  bandwidthCapacity: "1Gi"
                  networkCost: 5
            - origin: "z2"
              costList:
                - destination: "z1"
                  bandwidthCapacity: "1Gi"
                  networkCost: 5
            - origin: "z3"
              costList:
                - destination: "z4"
                  bandwidthCapacity: "1Gi"
                  networkCost: 10
            - origin: "z4"
              costList:
                - destination: "z3"
                  bandwidthCapacity: "1Gi"
                  networkCost: 10
```

NetworkTopology CRD
net-topology-test

Topology Info:
2x Regions: us-west-1, us-east-1
4x Zones: Z1 – Z4
8x Nodes: N1 – N8



NetworkOverhead – Example (Filter)

A1: P1, P2, P3
Candidate Nodes: N1 – N8

Nodes that unmet a higher number of dependencies are filtered out to reduce the number of nodes being scored.

Workload allocations:

P2 deployed on N1!

For this example, nodes that do not respect the network cost requirement between P1 and P2 (i.e., maxNetworkCost: 15) are filtered out.

Pod to schedule

Pod
P1

Workload to schedule: P1

Application Group

Workload	Kind
P1	Deployment
P2	Deployment
P3	Deployment

P2 is a dependency!

Pod dependencies

Workload	Pod	maxNetworkCost	Hostname
P2	P2-one	15	N1

Filter Operation

Nodes	Region	Zone	Network Cost to N1	Result
N1	us-west-1	Z1	0	PASS
N2	us-west-1	Z1	1	PASS
N3	us-west-1	Z2	5	PASS
N4	us-west-1	Z2	5	PASS
N5	us-east-1	Z3	20	FILTER
N6	us-east-1	Z3	20	FILTER
N7	us-east-1	Z4	20	FILTER
N8	us-east-1	Z4	20	FILTER

NetworkOverhead – Example (Score)

Candidate Nodes: N1 – N4

Calculate the accumulated shortest path cost for all candidate nodes.

Normalize (between 0 and 100) the accumulated cost for all nodes.

Nodes with lower costs are favored since lower costs correspond to lower latency

Pod to schedule

Pod
P1

Network Costs

SameHostname	SameZone
0	1

Application Group

Workload	Kind
P1	Deployment
P2	Deployment
P3	Deployment

Maximum Score

MAX Score
100

Pod dependencies

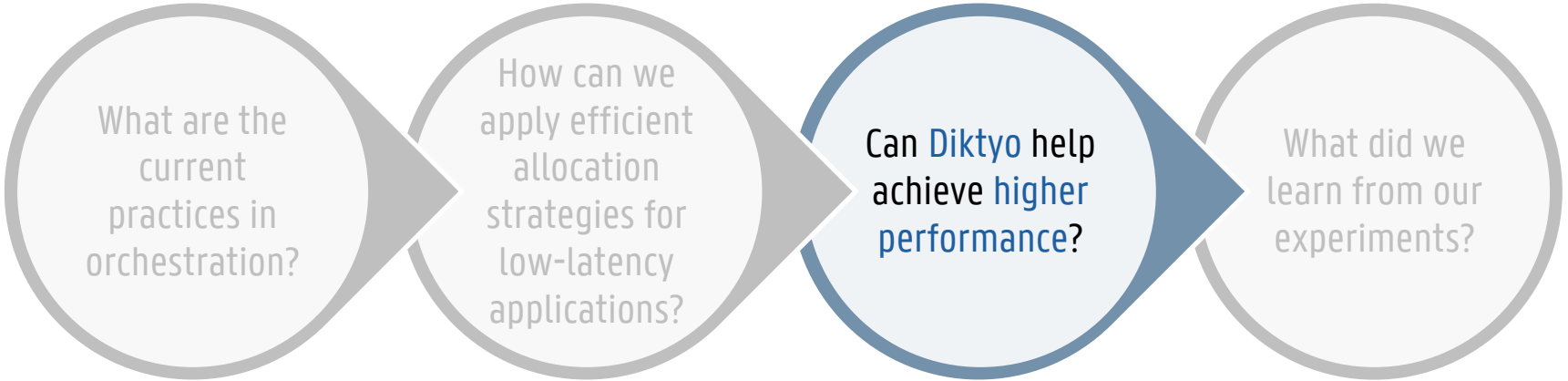
Workload	Hostname
P2	N1

Allocations

Workload	Pod	Hostname
P2	P2-one	N1

Scoring Operation

Nodes	Region	Zone	Accumulated cost	MAX Cost	MIN Cost	Normalized Cost	Node Score
N1	us-west-1	z1	0	5	0	0.00	100
N2	us-west-1	z1	1	5	0	20.00	80
N3	us-west-1	z2	5	5	0	100.00	0
N4	us-west-1	z2	5	5	0	100.00	0

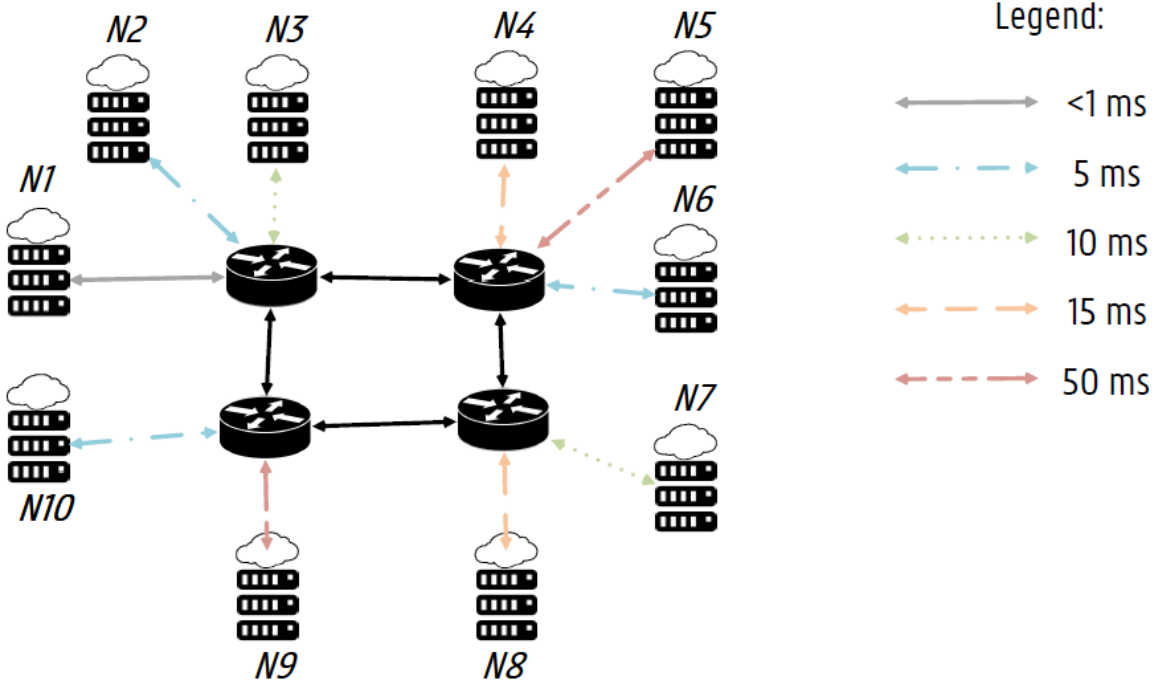

What are the current practices in orchestration?
How can we apply efficient allocation strategies for low-latency applications?
Can Diktyo help achieve higher performance?
What did we learn from our experiments?

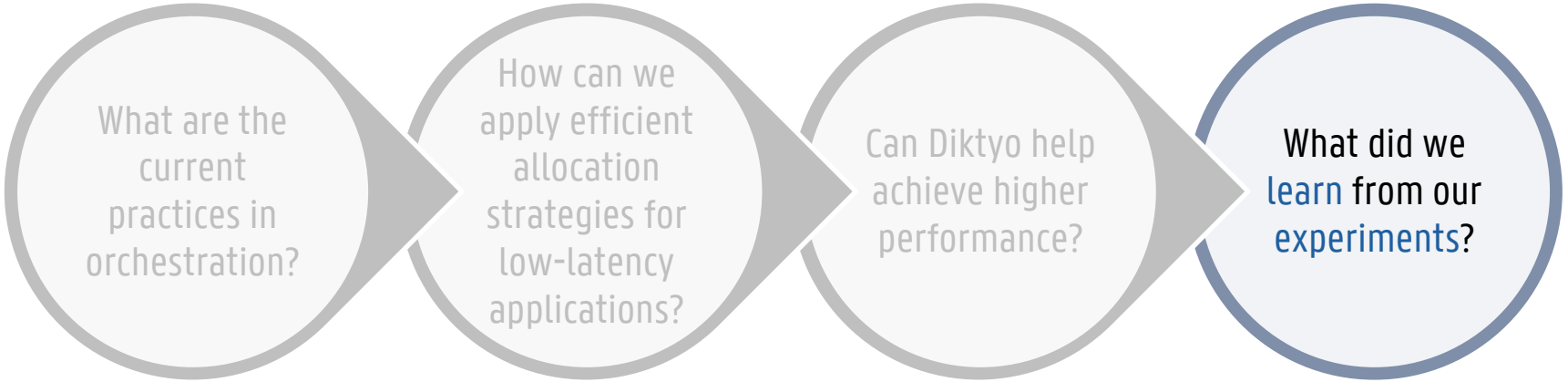
Live Demo!

Setup

Demo Setup

- A K8s cluster with 10 nodes (1 master and 9 workers).
- All nodes belong to the same region (*cloud*), but each node is in a different zone (i.e.,*z1, .., z10*).
- Varying delays are emulated on network connections via Traffic Control (TC).

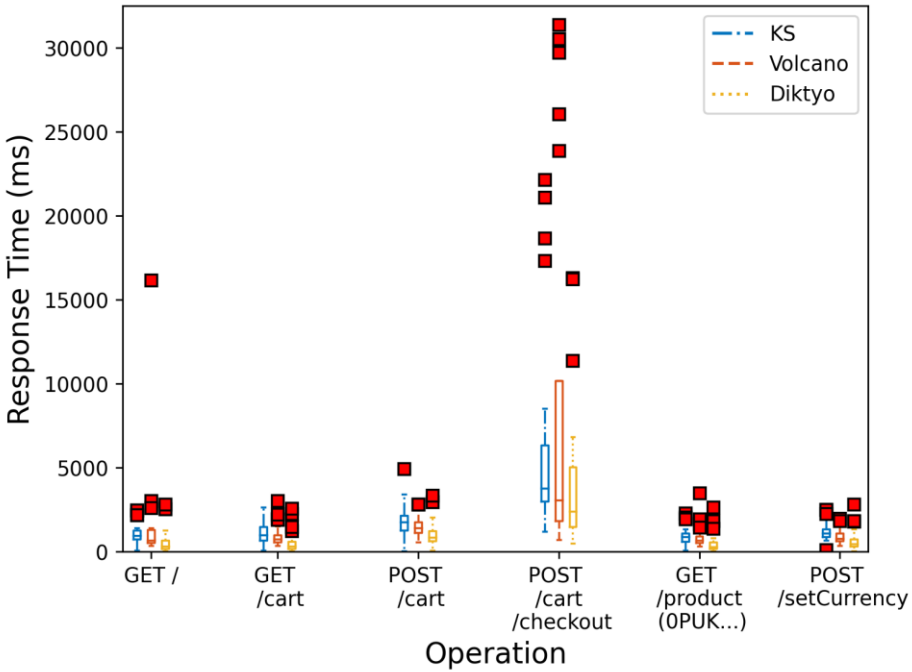


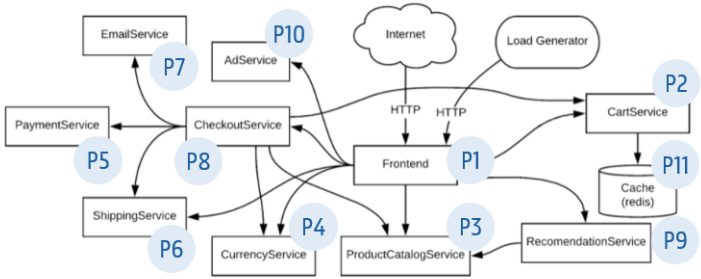

What are the current practices in orchestration?
How can we apply efficient allocation strategies for low-latency applications?
Can Diktyo help achieve higher performance?
What did we learn from our experiments?

Experiments with Microservice Benchmarks

Results

Experiments – Online Boutique[1]

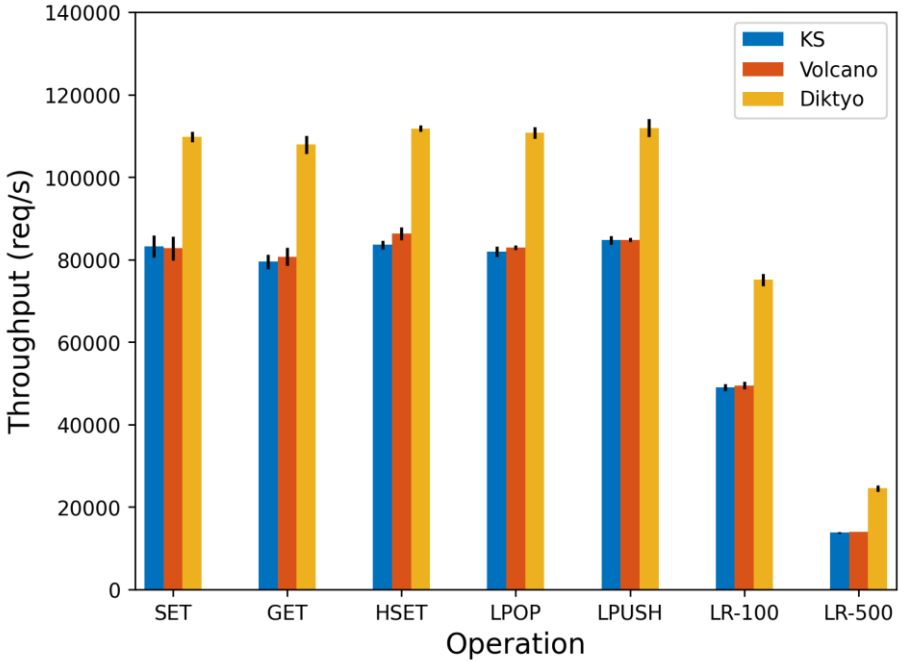


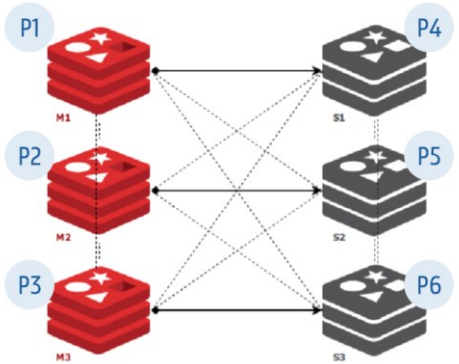


Diktyo reduces application response time by 45%,
compared to default plugins.

[1] Santos, J., Wang, C., Wauters, T., & De Turck, F. (2023). Diktyo: Network-Aware Scheduling in Container-based Clouds. IEEE Transactions on Network and Service Management.

Experiments – Redis cluster[1]



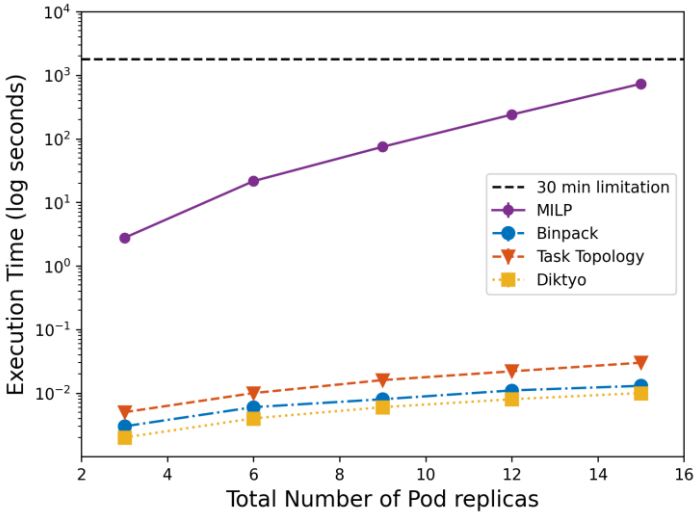


Diktyo increases database throughput on average by 22% compared to default plugins.

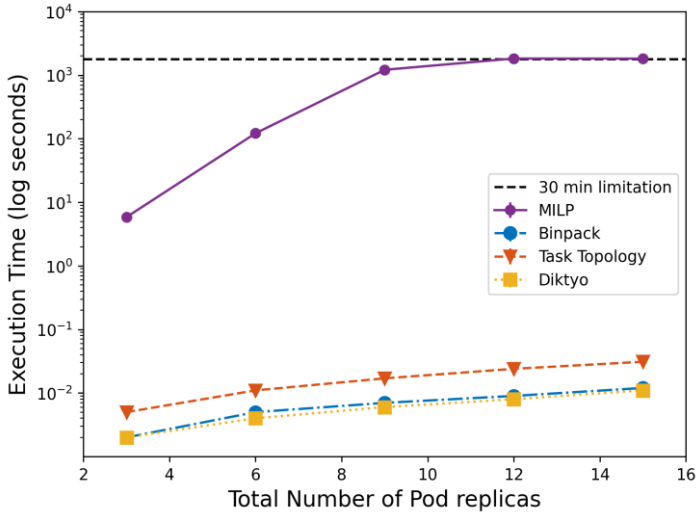
[1] Santos, J., Wang, C., Wauters, T., & De Turck, F. (2023). Diktyo: Network-Aware Scheduling in Container-based Clouds. IEEE Transactions on Network and Service Management.

Diktyo is significantly faster than a MILP-based model[1]

Cluster



Multi-region



The MILP model requires 12 and 30 minutes for 15 pods in the cluster and multi-region infrastructures.

[1] Santos, J., Wang, C., Wauters, T., & De Turck, F. (2023). Diktyo: Network-Aware Scheduling in Container-based Clouds. IEEE Transactions on Network and Service Management.

Diktyo's resource consumption is similar to KS[1]

TABLE IX: The resource consumption of the different scheduling mechanisms.

Scheduler	CPU usage (in millicpu)	Memory usage (in MiB)
KS	9.41m	83.5 MiB
Volcano	76.6m	72.0 MiB
Diktyo	6.19m	82.7 MiB

[1] Santos, J., Wang, C., Wauters, T., & De Turck, F. (2023). Diktyo: Network-Aware Scheduling in Container-based Clouds. IEEE Transactions on Network and Service Management.

NAS[1] (Earlier version) vs Diktyo

NAS

Extender process : Relies on an external call for the final node selection (time consuming)

Node Labels : RTT values and bandwidth reservations are made via Node Labels.

Recursive Function : The final selection is made based on a recursive function that finds the node with the lowest RTT to the target location and enough bandwidth.

Diktyo

Scheduling plugins project : Integrated on the scheduler plugins framework. All Diktyo plugins run alongside the plugins already available.

Custom Resources & Controllers : Diktyo proposes two new Custom Resources (AppGroup & NetworkTopology) to deal with microservice dependencies and the underlying network topology.

Multiple plugins : Diktyo proposes three additional plugins (QueueSort, Filter, Score) to find low-latency allocation schemes in Kubernetes.

[1]Santos, José, et al. "Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing applications." 2019 IEEE Conference on Network Softwarization (NetSoft). IEEE, 2019.

Diktyo: Roadmap Status

2021	Sep 9th	Presentation to the Kubernetes sig-scheduling community. The framework raised interest from the community. Received feedback on the design.
	Nov 4th	Initial Kubernetes Enhancement Proposal (KEP) submitted for revision.
2022	Feb 3rd	KEP v0.1 merged in the Kubernetes sig-scheduling repository.
	Feb 28th	First commit regarding Diktyo submitted. Waiting for revision.
	Mar 8th	KubeCon Europe 2022 Talk accepted to showcase Diktyo to the CNCF ecosystem.
	May 18th	KubeCon Talk. Raised a lot of interest from the attendees.
	Dec 8th	PR approved in the Kubernetes sig-scheduling repository.
2023	March 1st	Tutorial proposal approved at Netsoft 2023 (Madrid, June)

Kubernetes sig-scheduling: https://github.com/kubernetes-sigs/scheduler-plugins
KEP accepted: https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/kep/260-network-aware-scheduling
PR accepted: https://github.com/kubernetes-sigs/scheduler-plugins/pull/432

Diktyo-io community

Supporting the creation of network-aware components for the Kubernetes platform

README.md

Diktyo-io project

This project introduces several components for the Kubernetes platform to model/weight a cluster's network latency and topological information, and leverage that to better schedule latency- and bandwidth-sensitive workloads.

Introduction

Many applications are latency-sensitive, demanding lower latency between microservices in the application. Scheduling policies that aim to reduce costs or increase resource efficiency are not enough for applications where end-to-end latency becomes a primary objective.

Applications such as the Internet of Things (IoT), multi-tier web services, and video streaming services would benefit the most from network-aware scheduling policies, which consider latency and bandwidth in addition to the default resources (e.g., CPU and memory) used by the scheduler.

Users encounter latency issues frequently when using multi-tier applications. These applications usually include tens to hundreds of microservices with complex interdependencies. Distance from servers is usually the primary culprit. The best strategy is to reduce the latency between chained microservices in the same application, according to the prior work about Service Function Chaining (SFC).

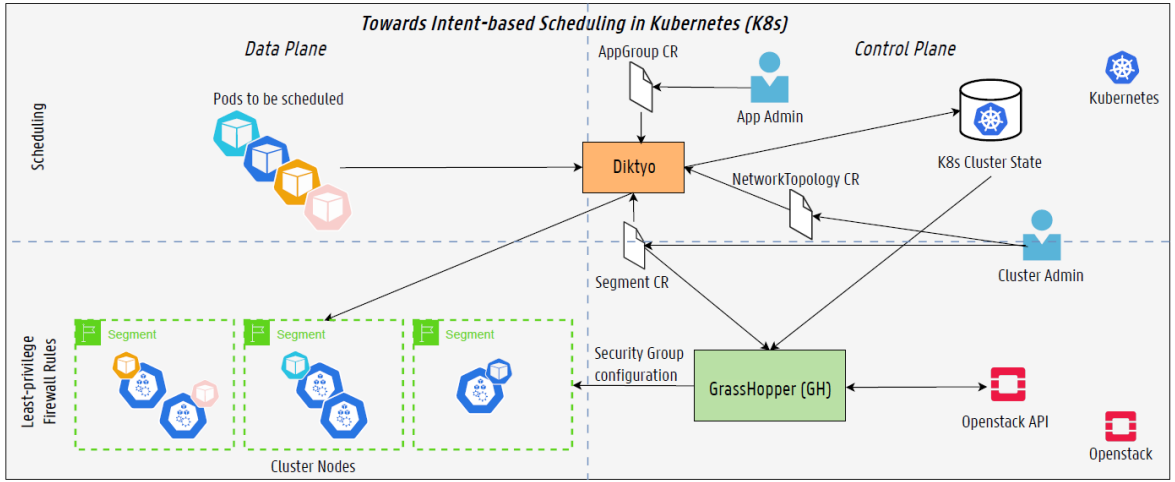
Besides, bandwidth plays an essential role for those applications with high volumes of data transfers among microservices. For example, multiple replicas in a database application may require frequent copies to ensure data consistency. Spark jobs may have frequent data transfers among map and reduce nodes. Insufficient network capacity in nodes would lead to increasing delay or packet drops, which will degrade the Quality of Service (QoS) for applications.

We propose several network-aware components for the Kubernetes platform that focus on delivering low latency to end-users and ensuring bandwidth reservations in pod scheduling.

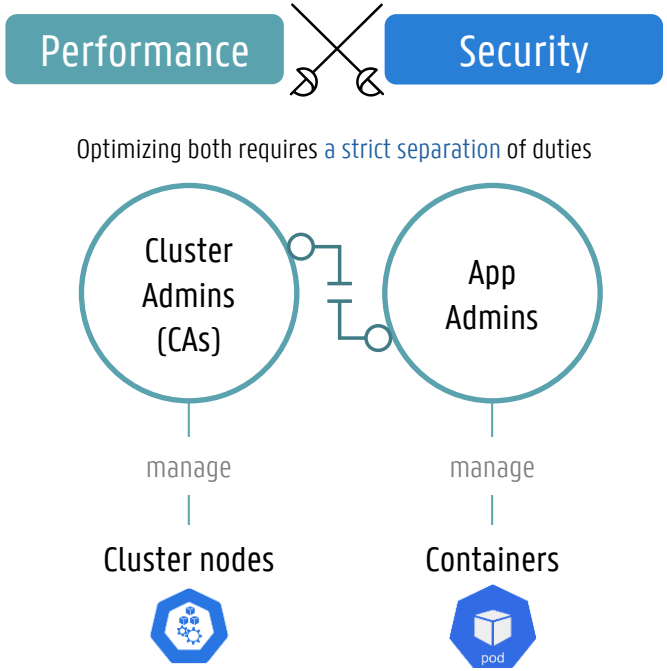
Looking for contributors and engagement from the community!

☺

Extending Diktyo Towards Intent-based Scheduling



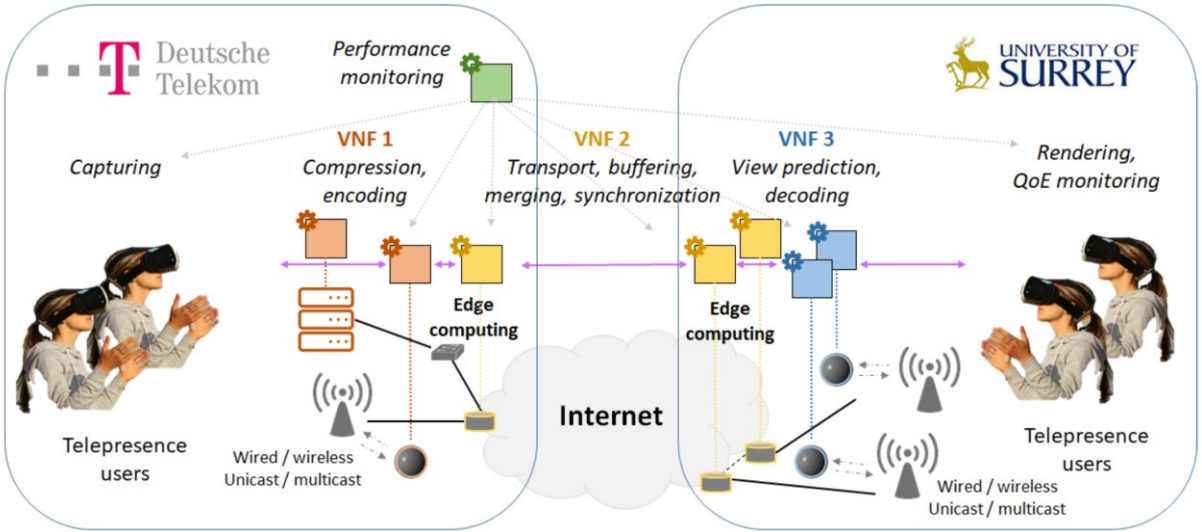
Adapting to the dynamic demands of multi-tenancy often leads to performance and security conflicts between Cluster Admins and App Admins.



Diktyo in the SPIRIT Project

Use Cases

Human-to-Human Live Telepresence.



Challenges & Lessons Learned

The developed plugins significantly reduce the expected network latency in K8s clusters.

Several pod “grouping” definitions exist, such as PodGroup, AppGroup, …

The developed plugins do not add significant overhead (Exec. Time: < 1 sec for 10000 nodes).

Getting a PR approved in the community takes time.

Acknowledgements

Chen Wang
IBM Research

Asser Tantawi
IBM Research

Tim Wauters
Ghent University, imec

Prof. Filip De Turck
Ghent University, imec

Kubernetes
Sig-scheduling

Scheduler Plugins

Repository for out-of-tree scheduler plugins based on the scheduler framework.

This repo provides scheduler plugins that are exercised in large companies. These plugins can be vendored as Golang SDK libraries or used out-of-box via the pre-built images or Helm charts. Additionally, this repo incorporates best practices and utilities to compose a high-quality scheduler plugin.

Q&A

José Santos

Postdoctoral Researcher at IDLab - Ghent University - imec

josepedro.pereiradossantos@UGent.be

jpedro1992@github

Chen Wang

Research Staff Member – IBM TJ Watson

chen.wang1@ibm.com

wangchen615@github

200 YEARS GHENT UNIVERSITY

imec

IDLab INTERNET & DATA LAB

IBM

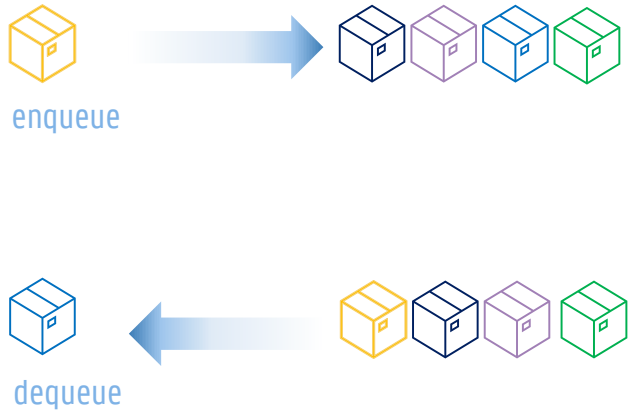
What is Queue Sorting?

Mechanisms or algorithms that determine the order in which containers are allocated in the infrastructure.

Priorities

Resources

These algorithms typically optimize resource allocation and reduce the overall execution time for scheduling, focused on container priorities and computing resources.

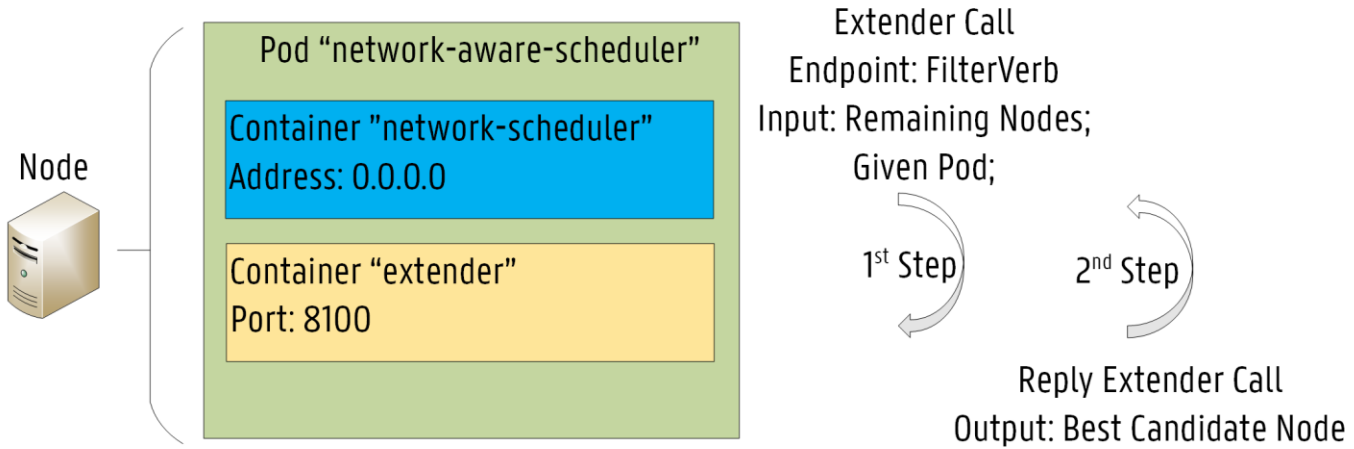


Network-Aware Scheduler (NAS)[1]

Implemented by extending KS:

1 Node Filtering

2 Minimization of RTT
Bandwidth Conservation



[1]Santos, José, et al. "Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing applications." 2019 IEEE Conference on Network Softwarization (NetSoft). IEEE, 2019.