Oakestra:

A Lightweight Hierarchical Orchestration Framework for Edge Computing

USENIX ATC'23

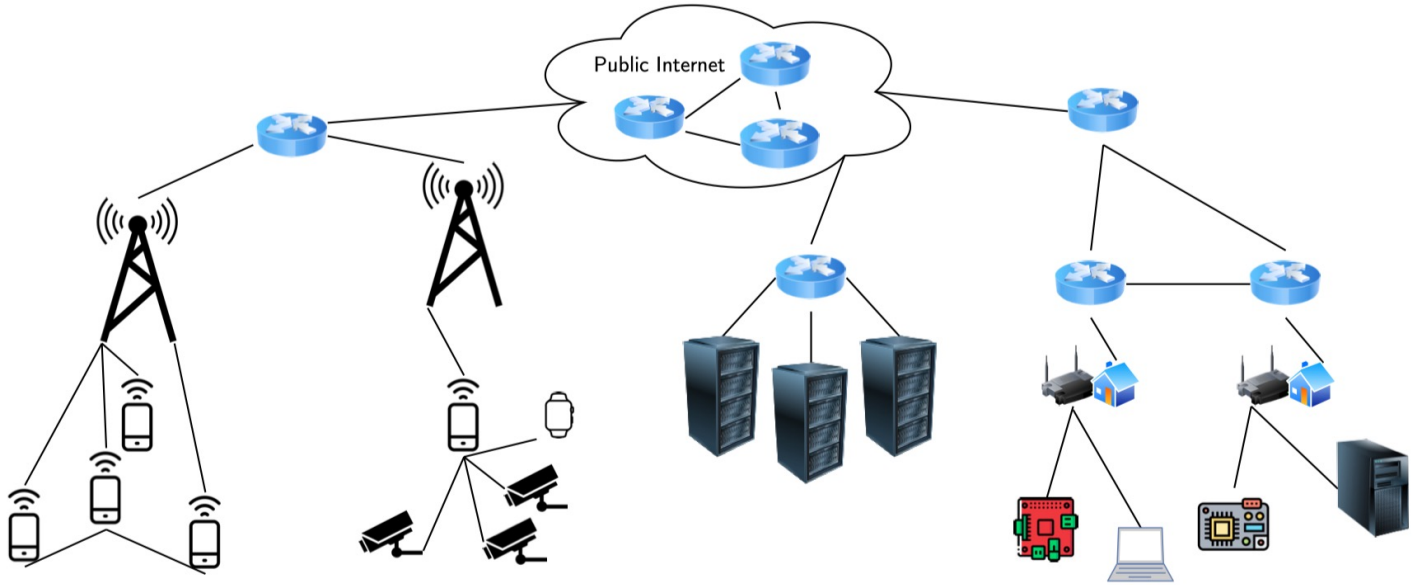
2023.09.27

Background

Edge Computing

Actual real-world implementations of edge computing remain limited

- Resources at the edge are far less capable and more heterogeneous than datacenters
- Existing orchestration frameworks perform poorly at the edge since they were designed for reliable, low latency, high bandwidth cloud environments



Background

Goal

Design a hierarchical orchestration framework for enabling running edge computing applications on heterogeneous resources

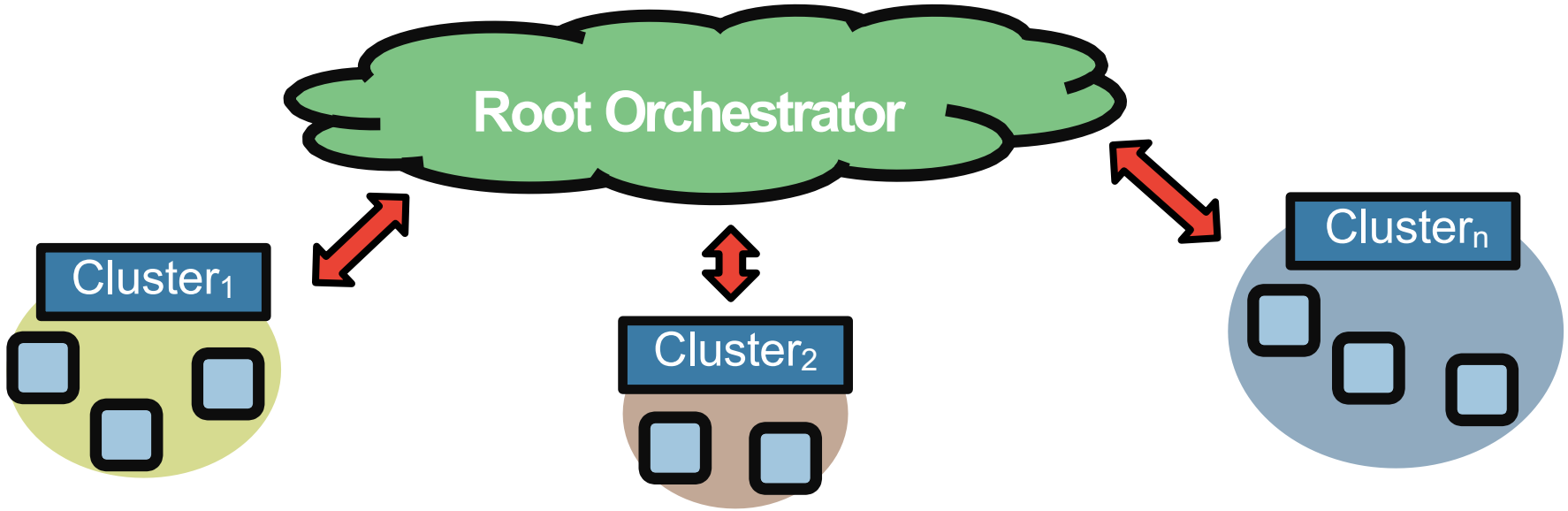
Challenges

- **Scalability**: support the infrastructure-at-scale – allowing scaling from thousands to millions of distributed nodes without management overheads
- **Deployment**: consider most up-to-date constraints of edge servers and autonomously find a compatible node for deployment
- **Communication**: ensure microservices communicate efficiently while meeting detailed SLA requirements of apps

Main Idea

Cloud-edge continuum is separated into fine- and coarse-grained management responsibilities across hierarchy

- Logical three-tier hierarchy → **Scalability**
- Delegated scheduling mechanism → **Deployment**
- Semantic overlay networking → **Communication**

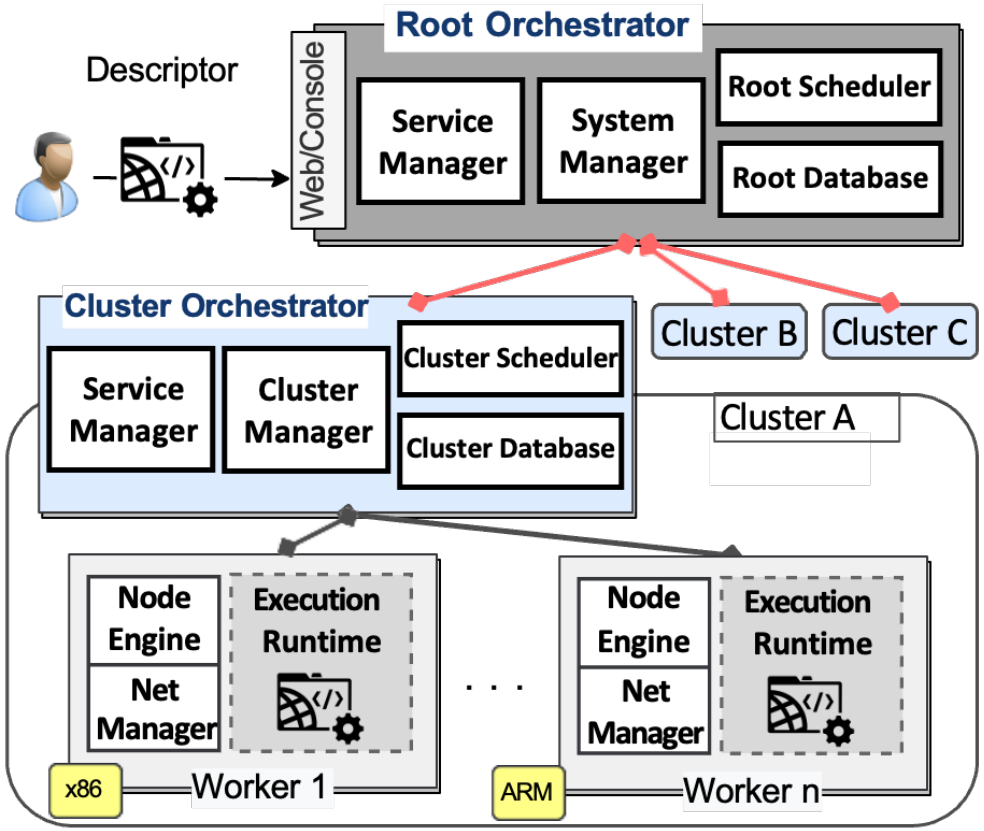


Architecture Overview

Oakestra

A flexible hierarchical orchestration framework designed for heterogeneous and constrained edge computing infrastructures

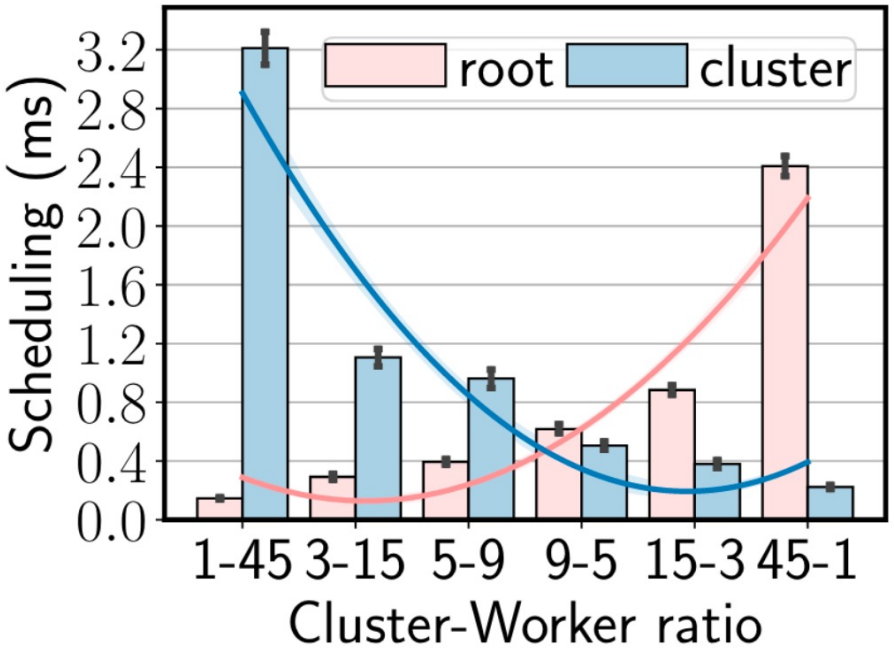
- **Root Orchestrator**: centralized control plane and is responsible for managing resource clusters
- **Cluster Orchestrator**: a logical twin of the root but with management responsibility restricted to resources within the local cluster
- **Worker Nodes**: edge servers in clusters responsible for executing services

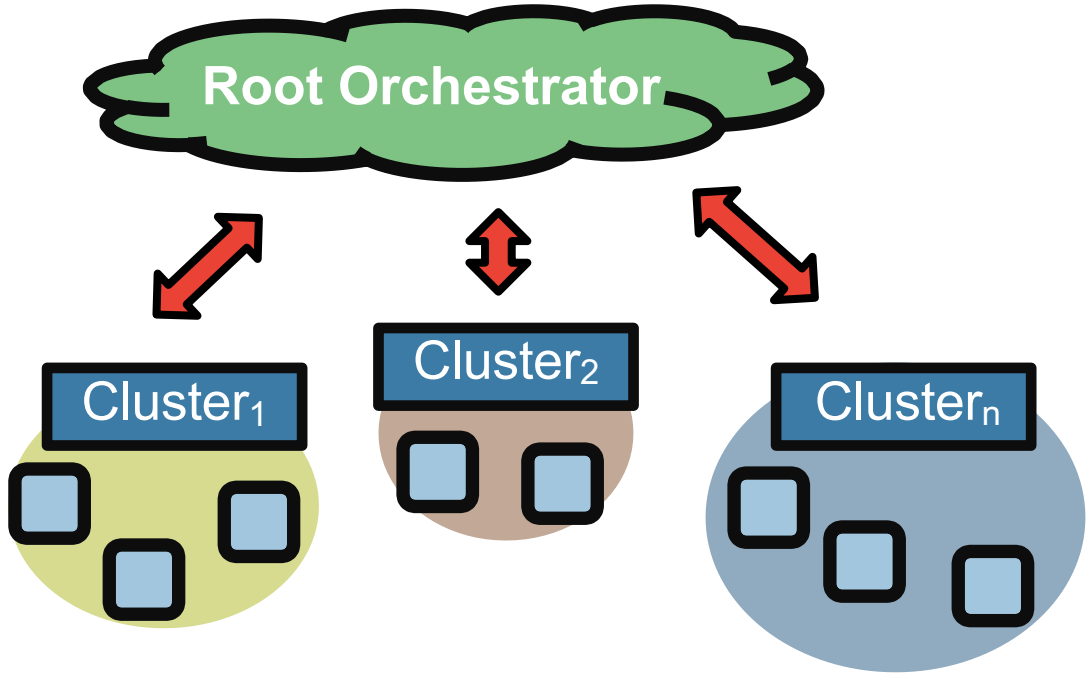


Design #1: Decoupled three-tier hierarchy

Problem

Flat management (inherent to most orchestration solutions) limits scalability.





A **hierarchical** management design is inherently better for **scalability** at the edge

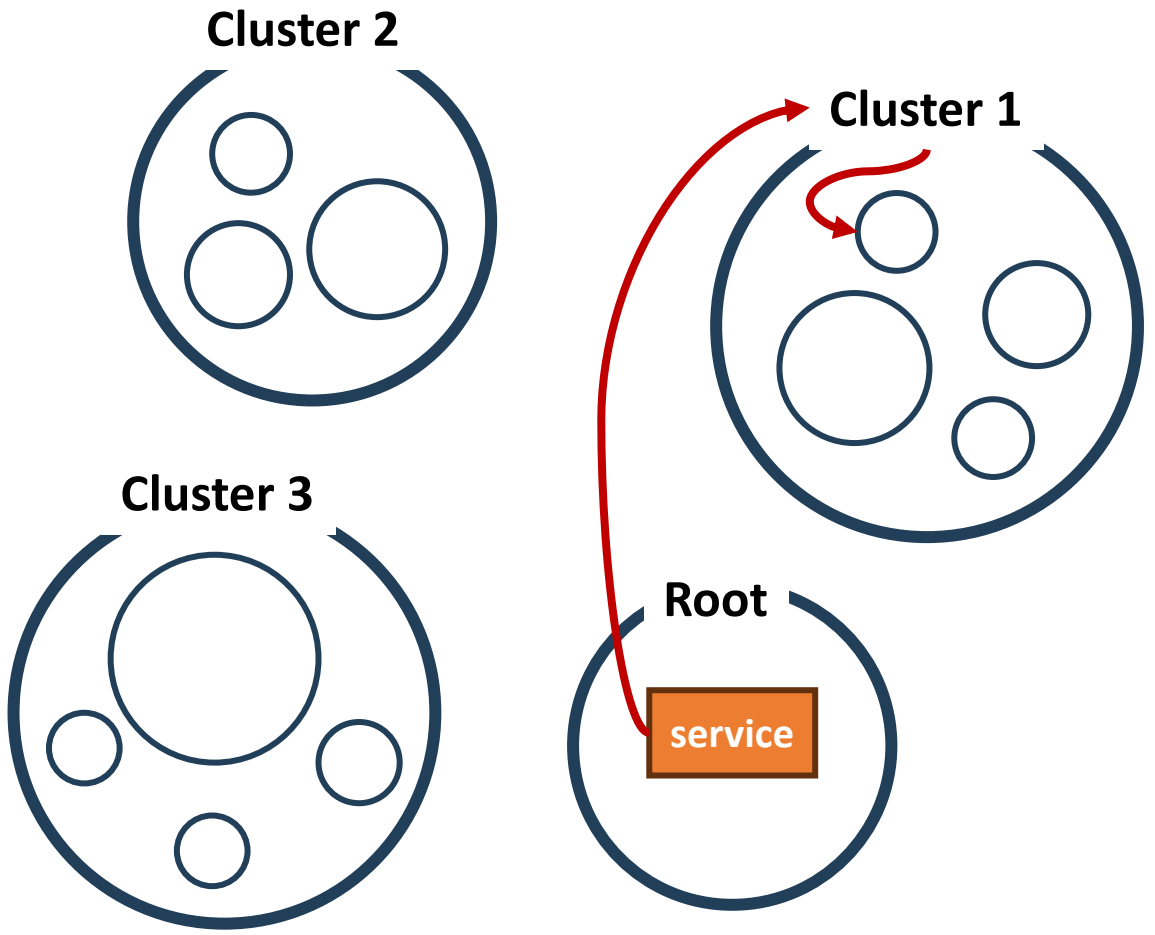
Design #2: Delegated scheduling mechanism

Problem

Existing edge orchestration framework only considers service scheduling and does not take into account resource management.

Key Idea

Upon receiving a service deployment request, the root scheduler finds a suitable cluster, and cluster scheduler finds a suitable worker node.



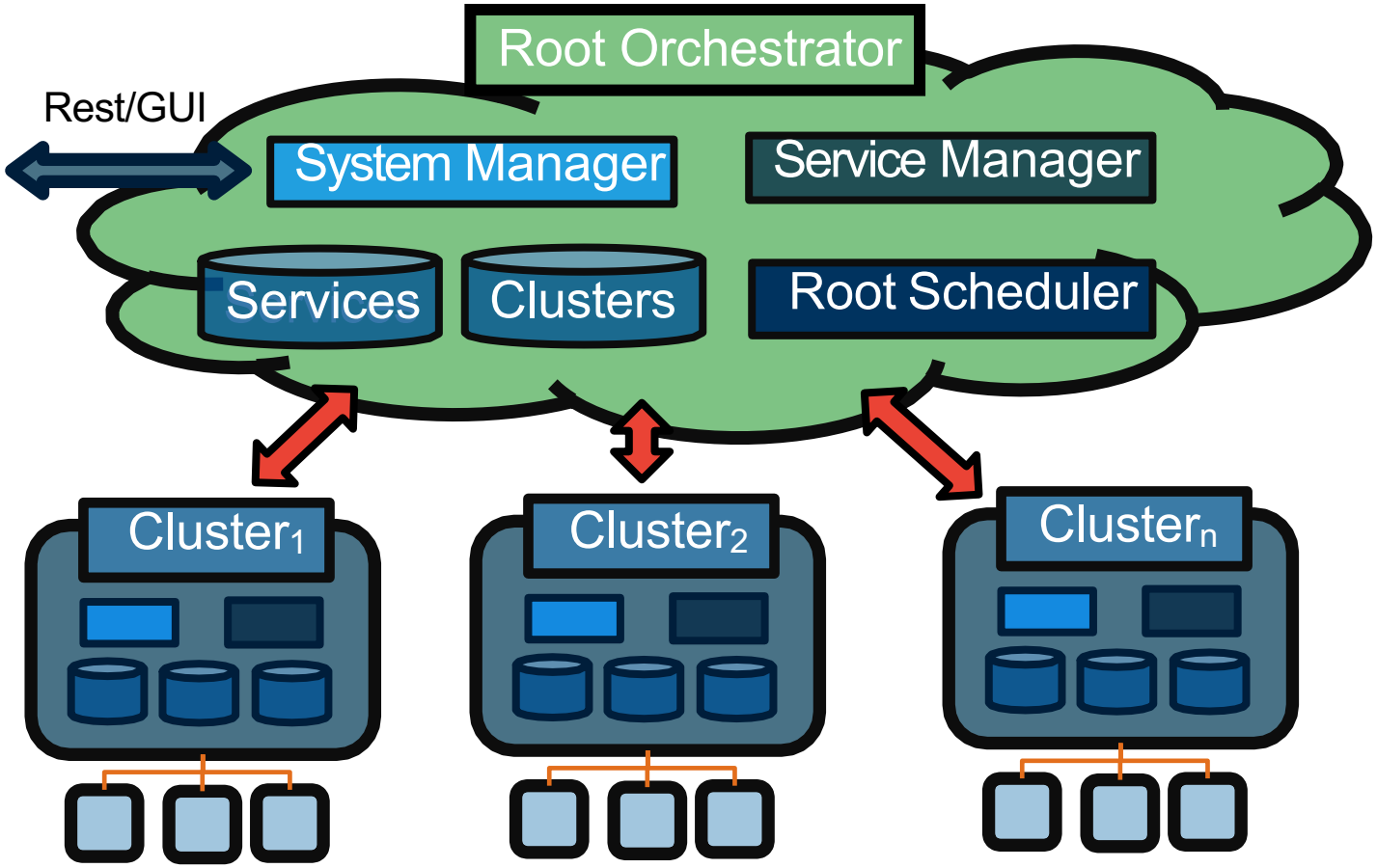
Design #2: Delegated scheduling mechanism

Workflow

Step #1: Developers submit the code along with an SLA descriptor to the system manager.

```
...
"application_name" : "ArPipeline",
"application_namespace" : "production",
"application_desc" : "AR object detection",
"microservices" : [
    {
    "microservice_name": "object-detection",
    "microservice_namespace": "production",
    "virtualization": "container",
    "memory": 100,
    "vcpus": 1,
    "vgpus": 1,
    "code": "demo-pipeline:detection",
    "port": "5001:5001/udp",
    "addresses": { "rr_ip": "10.30.30.30" },
    "constraints":[...]
      ...
    }...]
...
```

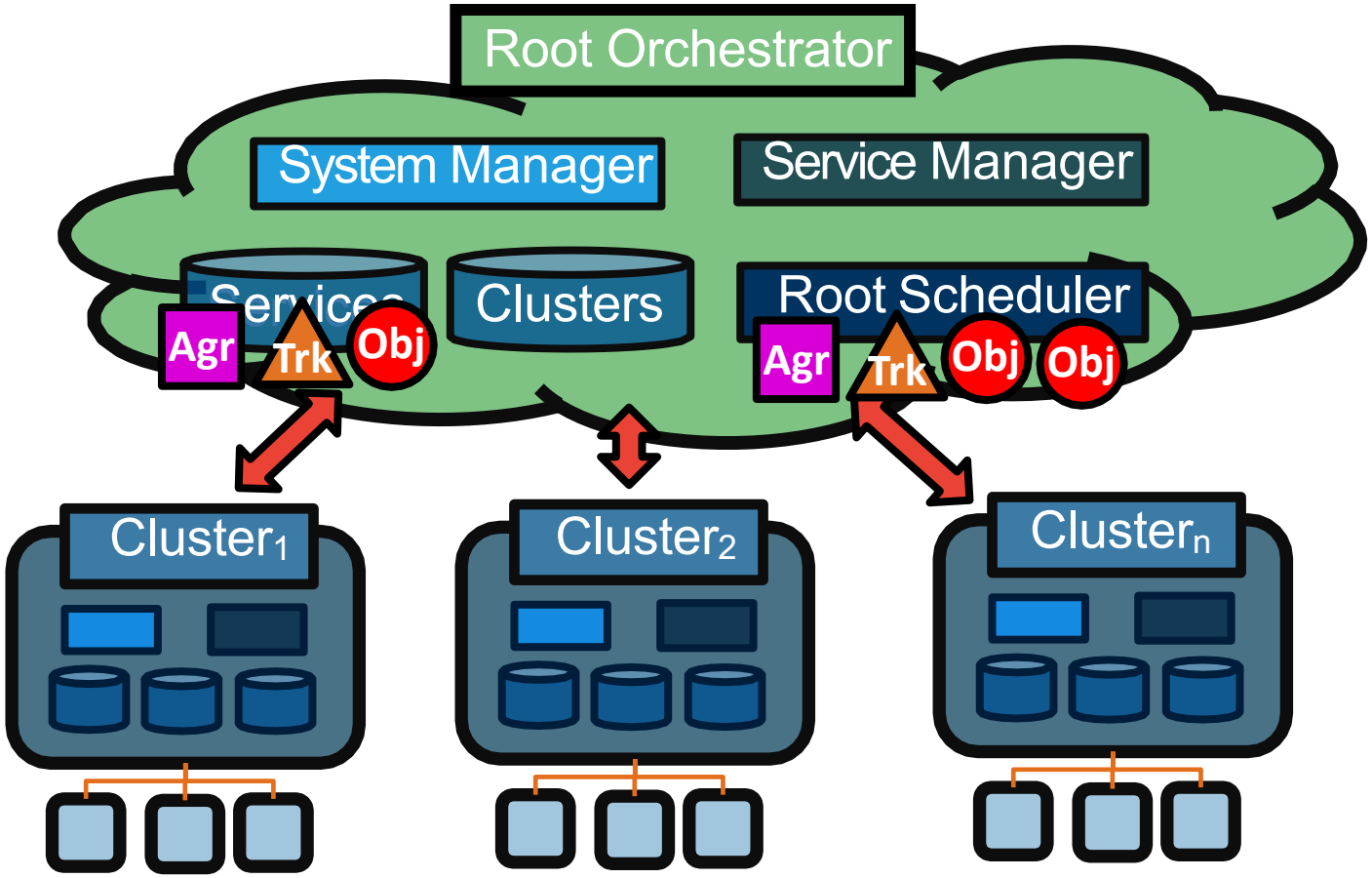
SLA descriptor



Design #2: Delegated scheduling mechanism

Workflow

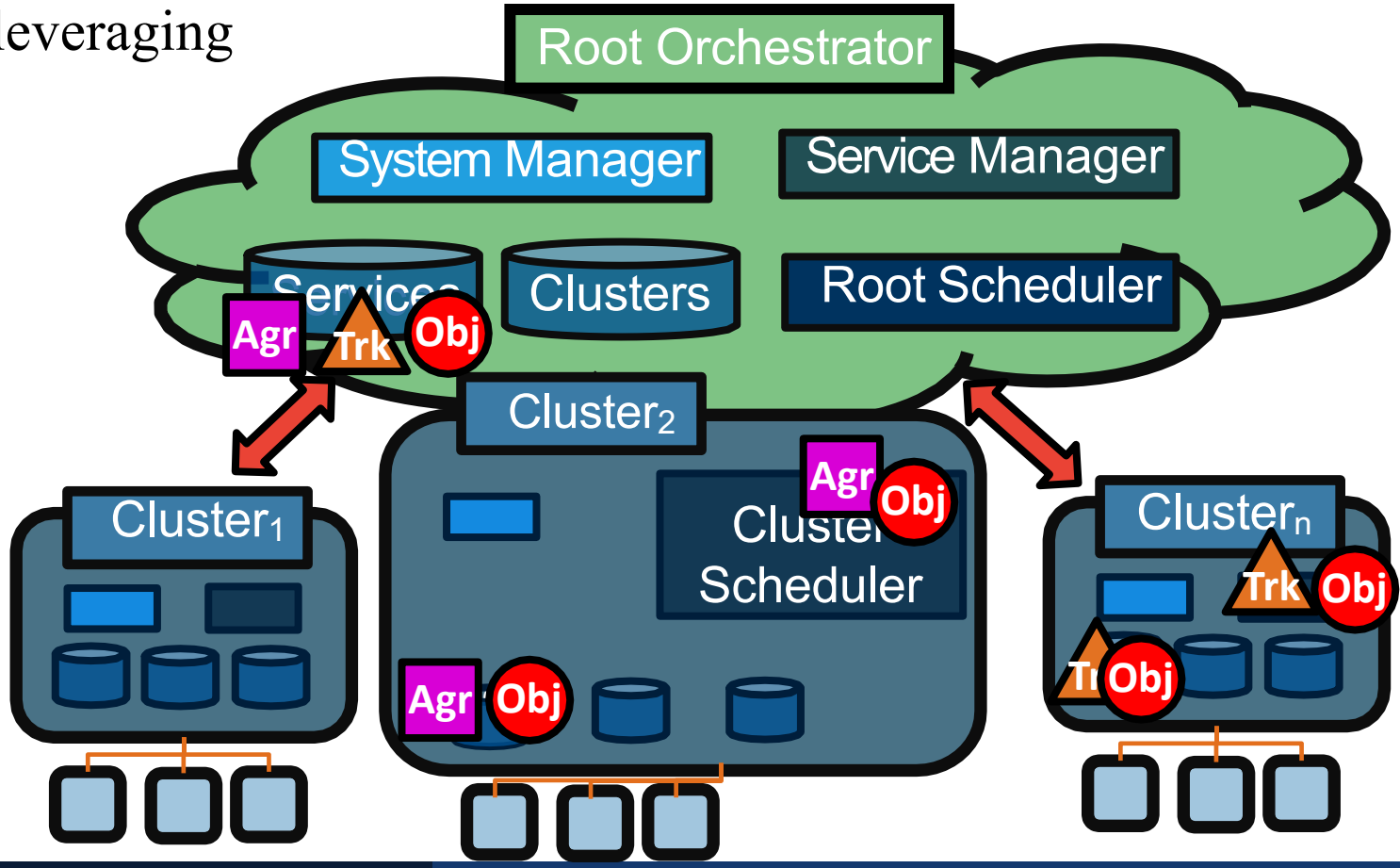
Step #2: Root scheduler matches SLA constraints to current capacity of each cluster and finds a suitable cluster to offloads the deployment request.



Design #2: Delegated scheduling mechanism

Workflow

Step #3: Cluster scheduler calculates the optimal service placement within its cluster, leveraging the available schedulers



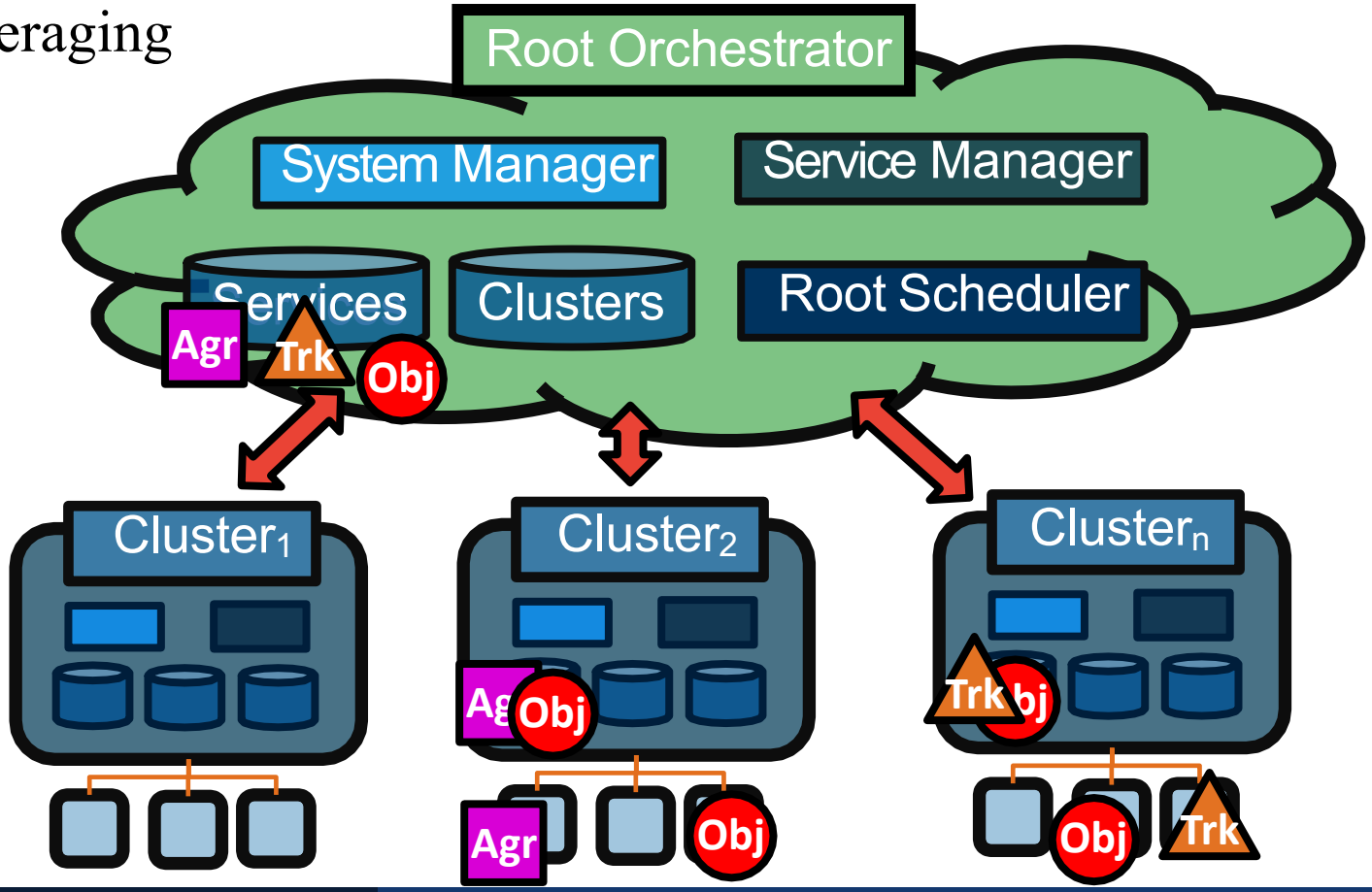
Design #2: Delegated scheduling mechanism

Workflow

Step #3: Cluster scheduler calculates the optimal service placement within its cluster, leveraging the available schedulers

- **Resource-Only Manager Match (ROM)**
  Maximize hardware utilization

- **Latency & Distance Aware Placement (LDP)**
  Service placement closer to user's location



Design #3: Semantic overlay networking

Problem

Majority of existing solutions make an assumption that edge servers from multiple participating infrastructure operators can interact over a common/public network, which is impractical.

Key Idea

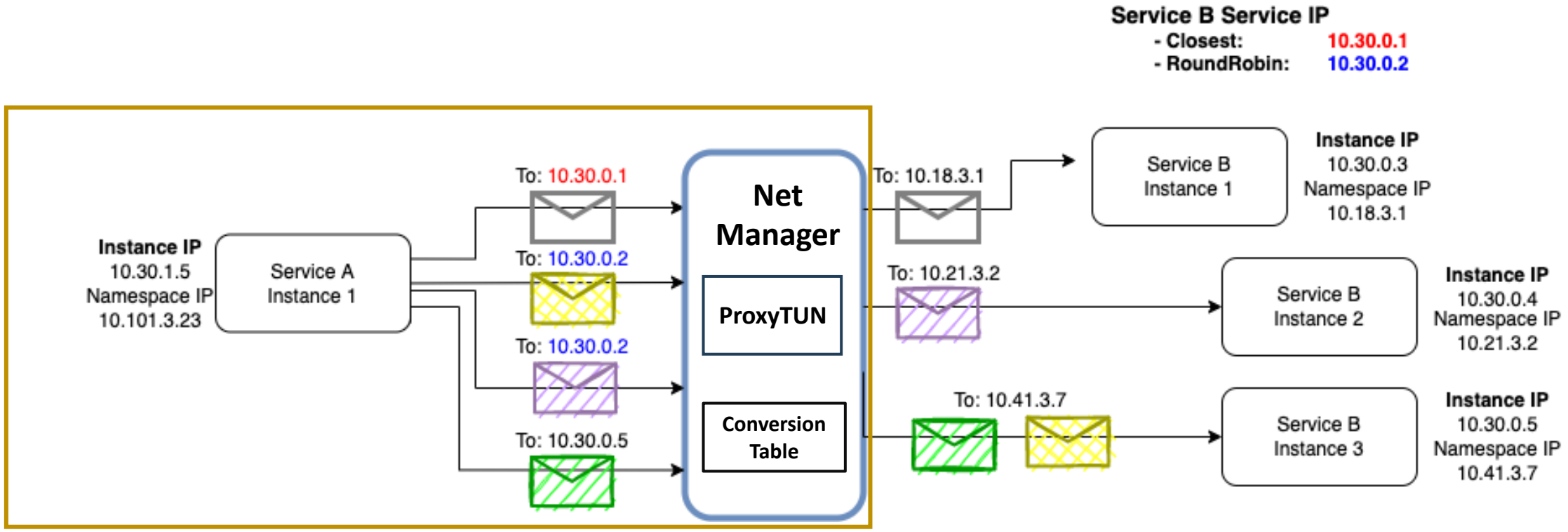
- **Transport layer packet tunneling** to interconnect services operating on resources with limited accessibility
- Dynamic routing policies transparently enforced via **semantic service addressing** to support load balancing catering to edge environments

Implemented in **NetManager**

Design #3: Semantic overlay networking

Semantic service addressing

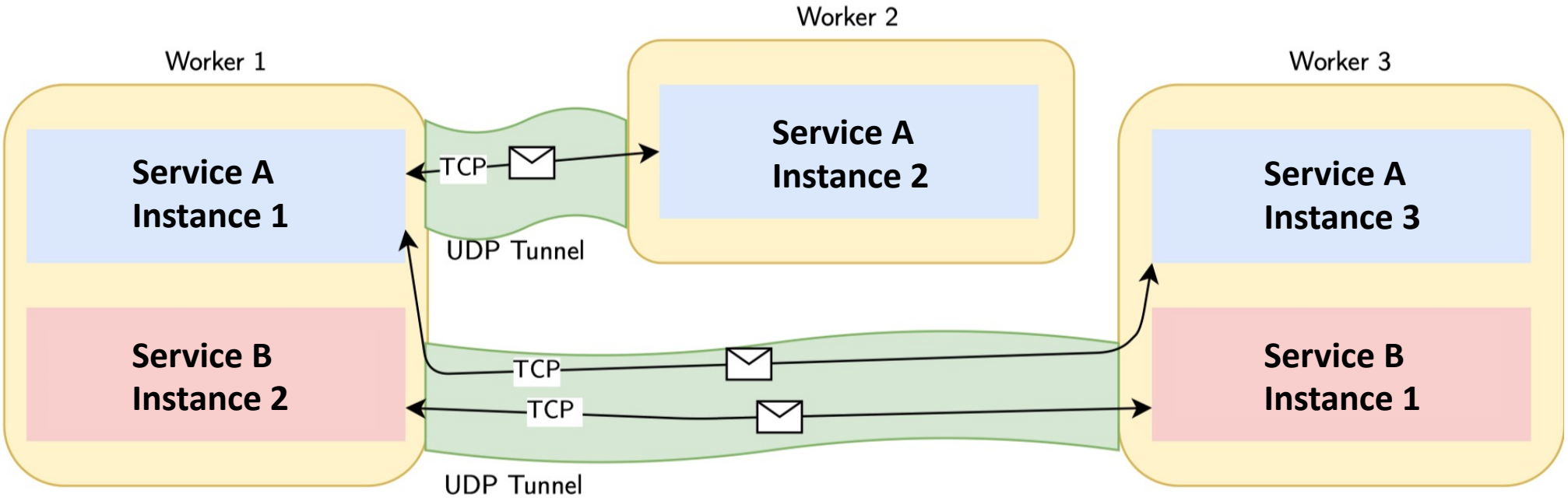
- **Namespace IP**: real address of the instance, the one provisioned at deployment time
- **Service IP**: references all the instances(replicas) of a microservice with a single address
- **Instance IP**: balances the traffic only to a specific service instance within the system



Design #3: Semantic overlay networking

Transport layer packet tunneling

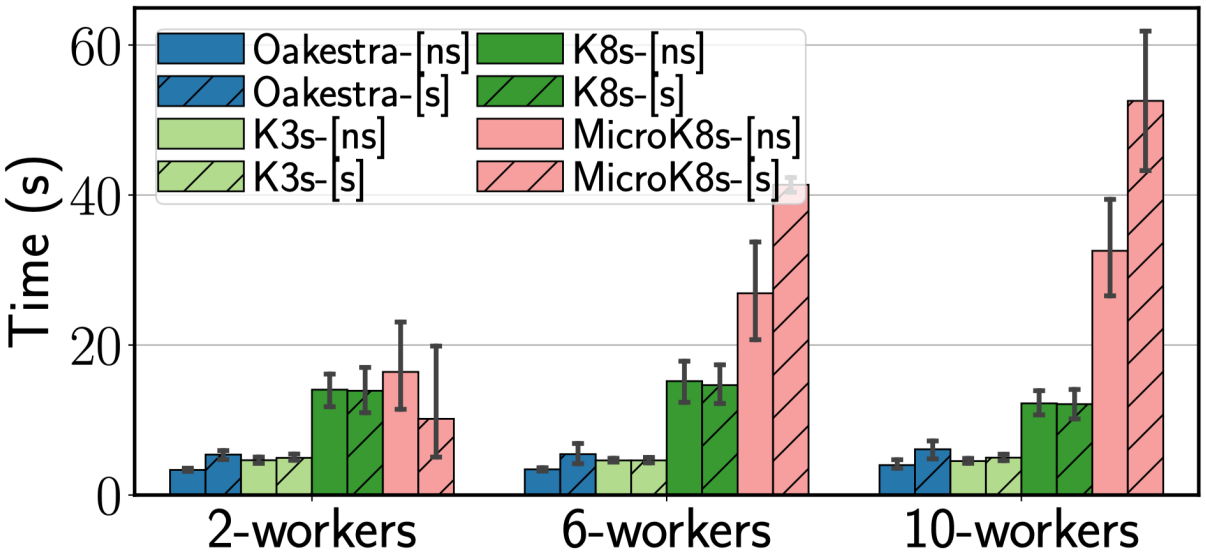
Oakestra enables inter-service communication across workers in different clusters with limited available ports through UDP tunneling



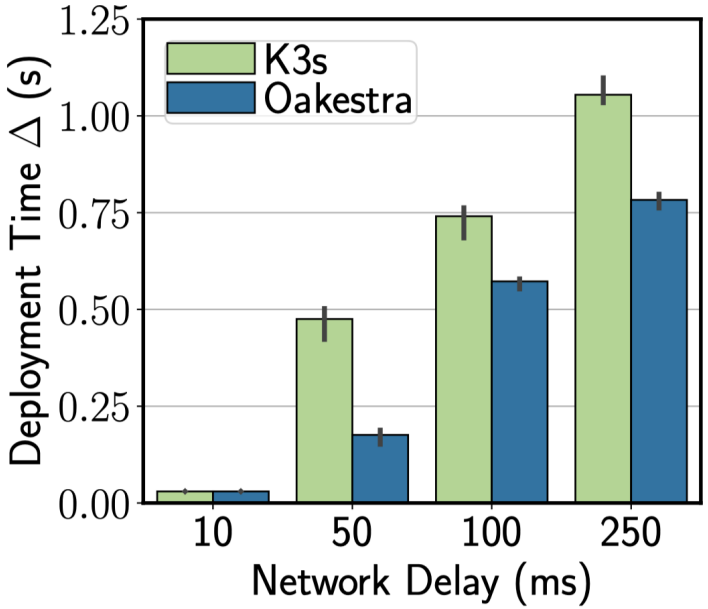
Evaluation

Service Deployment

Deployment time for different infrastructure sizes



Deployment time with network delay

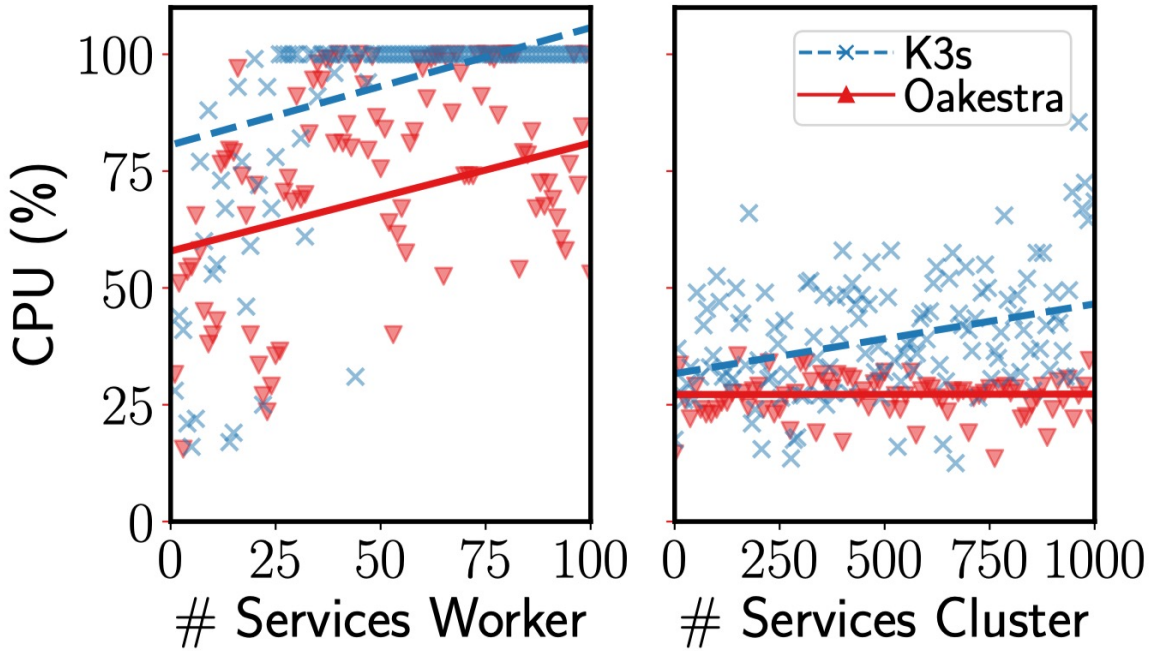


With low overhead scheduling, Oakestra has a short deployment time

Evaluation

Scalability

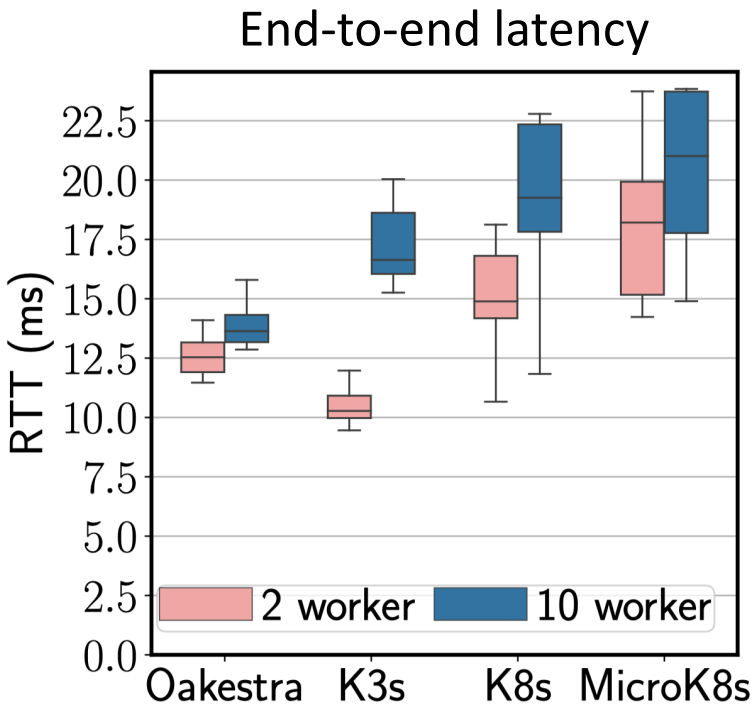
CPU usage of worker & cluster orchestrator in stress



Negligible overhead in Oakestra demonstrates its efficacy to support large service volumes

Evaluation

Networking



Proxying and site-to-site tunneling introducing minimal additional overhead while balancing with more replicas, especially at the edge

Paper Summary