Etate de l'art Kubernetes

https://www.researchgate.net/publication/345481618_KCSS_Kubernetes_container_scheduling_strategy

Dans l'article, l'approche d'ordonnancement repose sur l'utilisation de critères multiples, nommée « KCSS : Kubernetes Container Scheduling Strategy ». Cette stratégie vise à améliorer les performances en tenant compte des besoins des utilisateurs en termes de durée totale d'exécution (makespan) - le temps écoulé depuis la soumission du premier conteneur jusqu'à la fin de l'exécution du dernier conteneur - et des besoins des fournisseurs de cloud en termes de consommation d'énergie. Pour parvenir à cet ordonnancement optimal, six critères ont été sélectionnés :

- **Maximiser le taux d'utilisation des CPU** : L'objectif est de choisir, pour chaque conteneur nouvellement soumis, le nœud ayant le taux d'utilisation du CPU le plus élevé.
- **Maximiser le taux d'utilisation de la mémoire** : De manière similaire, il s'agit de sélectionner pour chaque nouveau conteneur le nœud affichant le taux d'utilisation de la mémoire le plus élevé.
- **Maximiser le taux d'utilisation du disque de stockage** : Pour chaque conteneur nouvellement soumis, le nœud choisi devrait avoir le taux d'utilisation de disque le plus élevé.
- **Minimiser la consommation d'énergie** : L'utilisation du framework CloudSim Plus permet de mesurer la consommation énergétique de chaque nœud. À chaque soumission d'un nouveau conteneur, CloudSim Plus est utilisé pour calculer la consommation moyenne d'énergie de chaque nœud. La valeur obtenue est ensuite affectée au critère de consommation d'énergie pris en compte par KCSS.
- **Minimiser le nombre de conteneurs en cours d'exécution** : Cette approche donne la priorité au nœud ayant le moins de conteneurs en cours d'exécution. Cette stratégie, nommée Spread, est celle utilisée par défaut dans Docker Swarmkit pour l'ordonnancement des conteneurs.
- **Minimiser le temps de transmission de l'image sélectionnée par l'utilisateur vers son conteneur** : Le choix se porte sur le nœud possédant déjà les images nécessaires pour le conteneur, afin de réduire le temps de transmission.

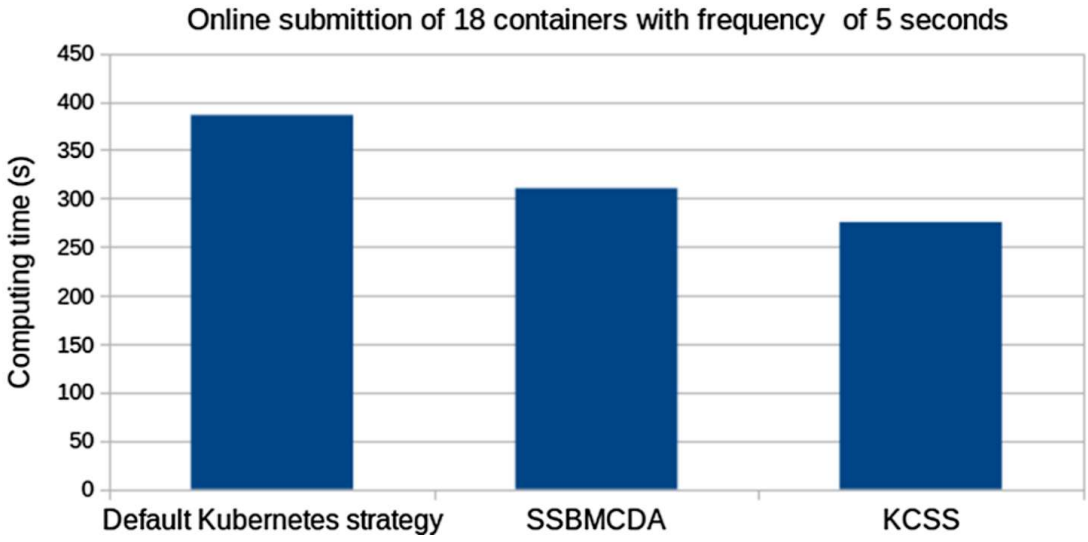
Fonctionnement de l'algorithme KCSS :

Pour un ensemble de conteneurs, chaque conteneur est évalué selon les six critères mentionnés ci-dessus. Ensuite, la technique de l'**Ordre de Priorité par Similarité à la Solution Idéale (TOPSIS),** une méthode d'agrégation multicritère, est appliquée pour combiner tous les critères en un seul rang. Cela permet de choisir le nœud idéal pour chaque conteneur en sélectionnant celui avec le rang le plus élevé.

Évaluation de KCSS :

Pour l'évaluation, l'algorithme a été testé avec différents types de conteneurs et de nœuds :

- **Nœuds** :
    a. Un nœud Intel Xeon E5-2650 v2 avec 32 CPUs, 48 GB de mémoire, et 16 GB d'espace de stockage.
    b. Un nœud Intel Xeon E5-2650 v2 avec 32 CPUs, 128 GB de mémoire, et 16 GB d'espace de stockage.
    c. Deux nœuds Intel Xeon E5645, chaque nœud avec 24 CPUs, 24 GB de mémoire, et 16 GB d'espace de stockage.
- **Types de conteneurs** :
    a. Petit conteneur : 4 CPUs, 5 GB de mémoire, 2 GB de disque de stockage et image nginx.
    b. Conteneur moyen : 8 CPUs, 10 GB de mémoire, 4 GB de disque de stockage et image tomcat:8.0.
    c. Grand conteneur : 16 CPUs, 20 GB de mémoire, 8 GB de disque de stockage et image k8s.gcr.io/pause:2.0.



Comparison between the makespan obtained with the default Kubernetes strategy, the SSBMCDA and the KCSS to schedule 18 containers submitted online with frequency of 5s

L'idée proposée pour cette simulation est de modifier les poids utilisés pour les critères dans l'algorithme. Actuellement, les poids attribués à chaque critère sont tous égaux à 1/6. Cependant, l'objectif principal étant de minimiser la consommation énergétique, modifier le poids du critère de consommation d'énergie pourrait permettre d'observer son effet sur le temps de traitement.

https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

Ajustez automatiquement le nombre de pods dans un déploiement ou un ReplicaSet en fonction de l'utilisation actuelle du CPU ou d'autres métriques personnalisées.

https://tag-env-sustainability.cncf.io/blog/2023-sustainability-istio-kepler-smart-scheduling/

L'article de Peng Hui Jiang, publié le 12 octobre 2023, présente une approche optimisée pour améliorer les performances et la durabilité des microservices en utilisant Istio, Kepler et une planification intelligente dans Kubernetes. Cet état de l'art a été présenté lors de KubeCon + CloudNativeCon + Open Source Summit China 2023 et se concentre sur les défis et les solutions en matière de consommation de ressources et d'impact environnemental des microservices.

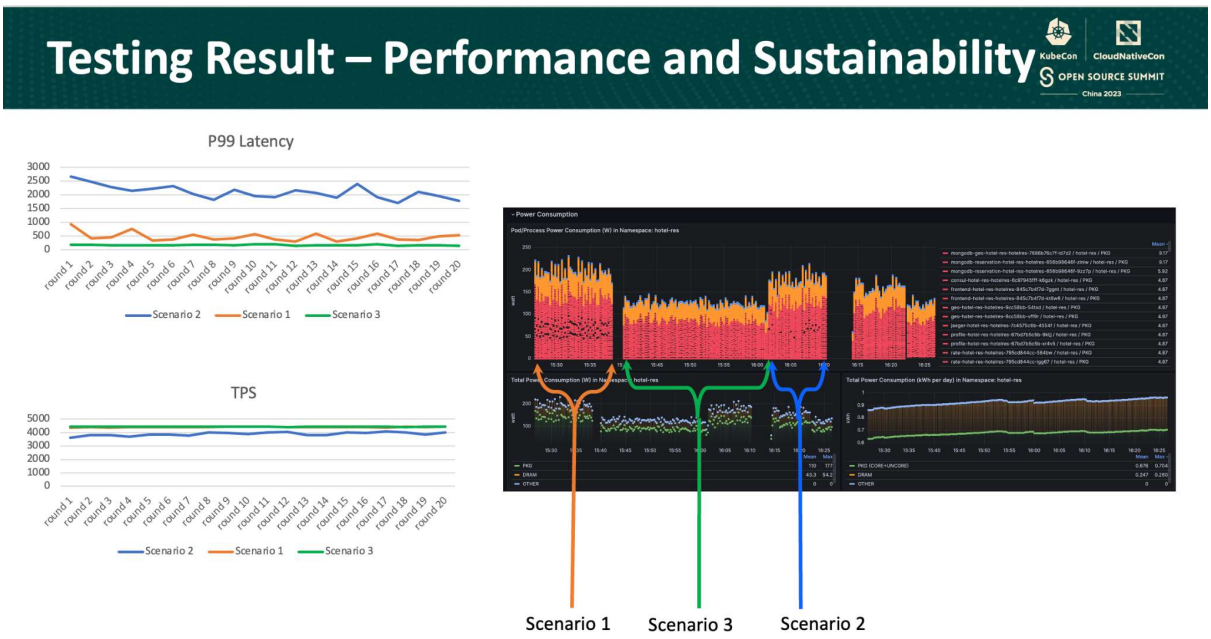
Problèmes et défis : Les microservices distribués peuvent entraîner une augmentation de la consommation des ressources et des coûts d'infrastructure. La latence dans les interactions de services peut affecter les performances. Il est donc nécessaire d'optimiser la communication au sein des applications microservices et de trouver un équilibre entre performance et responsabilité environnementale.

Solutions :
- **Istio** : Un service mesh open-source qui facilite la communication entre services et réduit la latence.
- **Kepler** : Un outil qui analyse les compteurs de performances CPU et les points de trace du noyau pour guider vers des pratiques économes en énergie.
- **Ordonnancement Kubernetes** : Permet une allocation optimale des ressources pour éviter le gaspillage.
- **Intelligence Artificielle (AI)** : Favorise la gestion intelligente des microservices avec une prise de décision en temps réel et une allocation de ressources adaptative.

Architecture de Kepler : Kepler utilise la technologie eBPF pour évaluer la consommation d'énergie des Pods Kubernetes, en se basant sur les performances CPU et les données du noyau Linux. Il fonctionne de manière ubiquitaire sur différentes plateformes et architectures, est léger et s'appuie sur des principes scientifiques.

Planification intelligente avec IA : Des tests ont été menés sur un cluster Kubernetes pour comparer différents scénarios de planification des ressources. Les scénarios incluent la politique de planification par défaut de Kubernetes et une allocation de service personnalisée pour optimiser les performances et la consommation d'énergie.

Résultats : Les tests ont montré que l'approche personnalisée (Scénario 3) où le service de base de données est programmé sur un seul nœud tout en utilisant la planification par défaut de Kubernetes pour le service de logique métier a surpassé les autres scénarios en termes de performance et de durabilité.

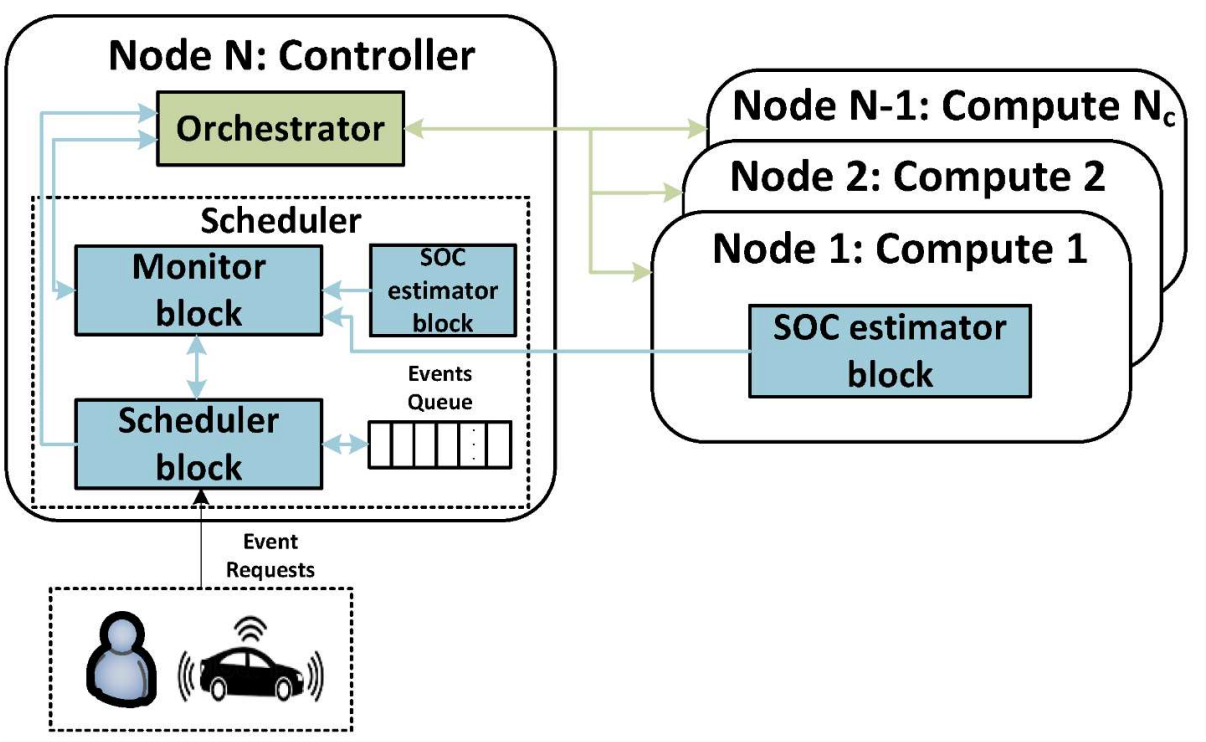


Conclusion : Les métriques réseau collectées par Istio sont essentielles pour optimiser la planification dans Kubernetes. Kepler est crucial pour mesurer la consommation d'énergie au niveau du conteneur, ce qui permet d'optimiser les microservices. L'intelligence artificielle, en synergie avec Istio, Kepler et la planification intelligente, améliore la gestion des microservices grâce à une prise de décision intelligente et une automatisation en temps réel. Les données de consommation d'énergie de Kepler et les tests de performance valident l'efficacité de ces optimisations.

https://www.mdpi.com/1424-8220/21/21/7151#sec4dot3-sensors-21-07151

Proposed Scheduling Solution

SOC and capacity-based scheduler (SOCCS). It processes event requests and determines the best node in a cluster to run them based on the remaining battery estimations and CPU usage in the nodes. Three main elements: the SOC estimator, the monitor and the scheduler.



1 - SOC *Estimator Block*

Its main functions are to receive the necessary information from the measurement equipment and calculate the *SOC* using the coulomb counting method.

d'après l'article suivant : SOC = (Q_releasable / Q_rated) * 100%

https://www.sciencedirect.com/science/article/pii/S0306261908003061?casa_token=uy-yRaTrJusAAAAA:x01rXz4d_bxTgGxcNJY5n4Ik8jJFpR8VtROurmzQyA3Qdw1BLgJZopXDAiRfiGwfiwt8brxHFqQ

Q(releasable) : l'énergie restante que vous pouvez utiliser avant que la batterie n'atteigne son seuil de coupure et soit considérée comme déchargée.

Finally, the SOC estimator sends the estimated value to the monitor block.

2 - The Monitor Block

the monitoring block receives the SOC battery information sent by the SOC estimator block in a parallel process to Algorithm 1. The received SOC information is saved in nusageSOC. As a result, the scheduling block is able to obtain the utilization of a node in terms of CPU usage, memory usage and SOC by reading the stored values in n(usage).

Procedure 1: Update Nodes.

```
forall n ∈ N do
    Update node usage by adding up the CPU and memory utilization of all the virtual
     nodes placed in this node
    if n_usage_CPU ≥ Usage_CPU_max and n_status is scheduled then
        n_status ← unscheduled
    if n_usage_CPU < Usage_CPU_max and n_status is unscheduled then
        n_status ← scheduled
```

Algorithm 1: Monitor Process.

```
Event_rejected ← 0 (Amount of rejected events)
Event_violations ← 0 (Amount of deadline violation in events)
while True do
    forall p ∈ P do
        if p_status is Running then
            Get CPU and Memory usage from metrics server
            Save CPU and Memory values in p_usage
        else
            if p_status is Succeeded and p_tc > p_d then
                Event_violations = Event_violations + 1
            if p_status is Failed then
                Event_rejected = Event_rejected + 1
            Delete the virtual node running the event to release it resources
            Remove p from P
            Remove S or T from l_S or l_T accordingly
    Procedure 1: Update Nodes
```

3 - *Scheduler Block*

This module determines the best node where an event can run according to the *SOC* prediction. The *SOC* prediction is determined through a regression model.

Procedure 2: SOC prediction.

Input: p^0, n
Output: SOC_{value}

$SOC_{value} \leftarrow 0, data \leftarrow \varnothing$
if n *is controller* **then**
  $cpu \leftarrow n_{usage_{CPU}} + p^0_{CPU_{req}}$
  $pkt_{in} \leftarrow n_{pkt_{in}} + \frac{n_{pkt_{in}}}{\sum_{p \in P}}$
  $pkt_{out} \leftarrow n_{pkt_{out}} + \frac{n_{pkt_{out}}}{\sum_{p \in P}}$
  $data \leftarrow cpu, pkt_{in}, pkt_{out}, n$
else
  $cpu \leftarrow n_{usage_{CPU}} + p^0_{CPU_{req}}$
  $data \leftarrow cpu, n$
$SOC_{value} \leftarrow SOC_{pred_{model}}(data)$
return SOC_{value}

Algorithm 2: Event Scheduler.

```
while len(l_priority) > 0 do
    Get nodes' capacity information
    p^0 ← First element of l_priority
    Update l_priority
    l_candidate ← ∅
    forall n ∈ N do
        if n_usage_SOC > SOC_min_threshold and n_status is scheduled then
            l_candidate ← l_candidate + n
    if l_candidate > 1 and n^controller ∈ l_candidate then
        Remove n^controller from l_candidate
    else if l_candidate == 0 or (l_candidate == 1 and n^controller ∈ l_candidate and p^0_{t_r} == 0) then
        if p^0_event is Service then
            forall f ∈ F do
                if p_{f_i} is deployed then
                    Delete p_{f_i} to release its resources
                if p_{f_i} ∈ l_priority then
                    Remove p_{f_i} from l_priority
        else
            Delete p_T to release its resources
        Remove p from P
        Remove S or T from l_S or l_T accordingly
        Event_rejected = Event_rejected + 1
    else
        n^best ← First element in l_candidate
        forall n ∈ l_candidate do
            if SOC_pred_model ≠ ∅ then
                n_SOC_pred ← Procedure 2: Predict SOC(p^0, n)
                Calculate n_score using Equation (9) with n_SOC_pred
            else
                Calculate n_score using Equation (9) with n_usage_SOC
            if n_score > n^best_score then
                n^best ← n
        Bind p^0 to n^best
```

$$n_{score} = \alpha_1 \cdot (SOC/100) + (1 - \alpha_1) \cdot (1 - \mathbb{E}_{\mathrm{CPU}}/Usage_{\mathrm{CPU}_{max}}) \tag{9}$$

Algorithm 3: Main process.

```
P ← 0
l_S ← 0 (List of running services)
l_T ← 0 (List of running tasks)
SOC_pred_model ← ∅, data_training ← ∅
Algorithm 1: Monitor Process
Algorithm 2: Event Scheduler
Algorithm 4: Regression Model Handler
while True do
    Add S or T to l_S or l_T accordingly to the event request
    Create p for S or T and add it to P
    Determine maximum delay to process p through Equation (10)
    Determine the time before putting p into priority queue through Equation (11)
    Calculate p_rank using Equation (12)
    Add p to l_priority
    Sort l_priority by virtual node ranking
```

5. Evaluation and Results

The main elements of the test to evaluate the proposed scheduling algorithm are:

1. **Testbed Composition**: A cluster of four Raspberry Pi 4 Model B units.
2. **Devices Role**: The Raspberry Pis serve as regular IoT devices capable of connecting to various sensors and providing edge processing capabilities.
3. **Power Consumption Measurement**: The UM24C module is used to measure the power consumption of each node and connects to the Raspberry Pis via Bluetooth.
4. **Energy Source**: Batteries with a capacity of 10,000 mAh power the Raspberry Pi devices.
5. **Management Framework**: Kubernetes 20.04 is deployed to manage virtualized services and tasks.
6. **Service and Task Execution**: Services and tasks are run as Docker containers within Kubernetes pods.
7. **Scheduling Algorithm Implementation**: The proposed scheduling algorithm is implemented using Python 3.6.8.
8. **Integration with Kubernetes**: The algorithm is deployed within Kubernetes, where it replaces the baseline scheduler.

Figure 15. Battery consumption for each node while running different scheduling algorithms.