修士論文

分散可能なコンテナ間動的スケジューリング最適化

Yao Yudie

主指導教員 宇多 仁

北陸先端科学技術大学院大学
先端科学技術専攻
（情報科学）

令和 7 年 3 月

Abstract

In recent years, the container orchestration system Kubernetes is widely used to implement services in microservice architectures, but important control functions, including the scheduler, are concentrated on the master node. While this design simplifies operation and management, it also brings challenges to the scalability and fault tolerance of the system. If the scheduler stops due to failure, it can affect the scheduling of new pods, cluster monitoring, and other critical operations. The Kube-scheduler performs scheduling by mainly considering CPU and memory resources, which may cause load imbalance between nodes. In addition, because the Kube-scheduler operates as a centralized component in the cluster, its failure may have a significant impact on the scheduling function of the entire cluster.

The purpose of this research is to develop a distributed scheduling system based on Deep Reinforcement(DRS), which optimizes the utilization rate and load balancing of the environment. By distributing the scheduling components, the entire cluster scheduling function is not affected even if one component fails.

In this research, I propose a distributed scheduling method using MARL, where the scheduler learns the resource usage of the nodes, the resource requirements of pods, and the network delay between the nodes, and learns how to make scheduling decisions based on the needs and priorities of different pods. This allows the scheduler to learn from its own experience and autonomously adapt to changes in the environment. This reduces the risk of single points of failure and minimizes the impact of node or component failures on the entire system, while enabling real-time decision-making by leveraging local information on each node, achieving optimal resource utilization and even load distribution.

The MARL model design in this proposed method is defined as follows: The model uses schedulers located on each node as agents. The state space is defined in three types: local state, global state, and waiting pod information. Local state includes directly observable information by each agent (CPU utilization, memory utilization, disk I/O information, node health status and number of currently running pods), while global state represents cluster information (CPU utilization, memory utilization and disk I/O information). Waiting pod information contains the CPU and memory resources requirements of pods. The reward function is designed to consider both local and global rewards, evaluated based on resource utilization. The action space is defined as a discrete space where each agent makes binary decisions on whether to accept waiting pods for their assigned node. The Deep Q-Network algorithm is implemented for the agent's policy.

The proposed distributed scheduler follows a stepped design similar to the Kubernetes default scheduler. First, in the pre-filter step, each scheduler examines its node's resource information and conditions. Next, during the filter step, the

MARL model is implemented where each scheduler determines whether to accept or reject the pod. If a pod is accepted, the node undergoes evaluation and receives a score. In the scoring step, agents share their actions and scores, with the best node selected based on sorted results. Finally, once the decision is made, the model updates in response to the outcome.

The proposed schedulers share decisions through a distributed leadership model. When the scheduler initializes, one instance is elected as the leader, which then establishes heartbeat monitoring and launches a socket server. The remaining scheduler instances transition into a monitoring role. During normal operation, all scheduler instances actively monitor pods and generate scheduling decisions. However, only the leader scheduler has the authority to receive these decisions, determine the optimal node placement, and execute pod binding operations. To ensure high availability, non-leader instances continuously monitor the leader's health through heartbeat checks. If the leader becomes unresponsive, the non-leader instances trigger a new leader election process. The newly elected leader assumes all leadership responsibilities, ensuring uninterrupted system operation.

The verification of this research consists of operational validation and performance evaluation. For operational validation, I confirmed the startup, scheduling and failure handling capabilities of the proposed distributed scheduler and I also measured computation, communication, and resource overheads. The performance evaluation involved deploying 105 pods across three different types of microservices, testing them under both uniform and random workload distributions for the resource utilization compared to the DRS.

As a result of my experiments, regarding system stability, the distributed scheduler successfully maintained cluster stability and scalability during node additions and failures, effectively addressing the single point of failure issue. The distributed architecture proved effective, with individual schedulers making independent decisions while the leader scheduler successfully aggregated these for final scheduling. About the performance evaluation results, under uniform workload distribution, the system achieved superior memory utilization and disk I/O stability, while maintaining stable network bandwidth and CPU utilization. With random workloads, the distributed scheduler demonstrated better performance through stabilized memory utilization, reduced network load, and more consistent disk write speeds compared to DRS. The scheduler also showed efficient resource management by appropriately reducing resource utilization as tasks completed. Overall, the proposed distributed scheduler showed marked improvements over previous research, particularly in stabilizing memory utilization, reducing network load, and optimizing disk I/O control.

However, due to the short experimental period, I could not comprehensively

evaluate the algorithm's performance and stability during long-term operation. While my evaluation was conducted on a Kubernetes cluster with four worker nodes and one master node, real-world environments typically consist of tens to hundreds of nodes. Therefore, evaluation of scalability and resource utilization in practical-scale cluster environments is necessary. Also, the current Python implementation has limitations where only a single scheduler can receive new pod information when deployed as a DaemonSet in the Kubernetes cluster environment. A potential solution is migrating to Go language, which would enable the proposed scheduler to be deployed as a DaemonSet with guaranteed pod instances on each node through the Informer feature. Additionally, long-term experiments are needed to verify algorithm convergence performance and system stability, particularly against various complex workloads. Finally, the implementation of an automatic deployment mechanism remains a crucial area for future development.

目 次

図 目 次

表 目 次

第1章　はじめに

本章では、本研究の背景と目的、本論文の構成を述べる。

1.1　背景

近年、クラウドコンピューティングの進展に伴い、クラウドプラットフォームは強力なリソースの柔軟性を提供するだけでなく、企業の多様なコンピューティングニーズにも対応可能となっている。そのため、より多くの企業や組織が業務をクラウドプラットフォームへ移行している [1]。このような背景から、軽量かつ迅速な展開が可能であり、管理や移行も容易なコンテナ技術は、様々なクラウド環境において幅広く活用されている。特に、Kubernetes[2] は成熟したコンテナオーケストレーションシステムとして、コンテナ化されたアプリケーションの管理とスケジューリングを効率的に行うために広く利用されており、クラウドプラットフォームを支える中核的要素となっている。

しかし、Kubernetes はスケジューラーを含む重要なコントロール機能がマスターノード上に集中しているため、システムの拡張性や耐障害性を低下させるという課題がある。スケジューラーが障害により停止した場合、新しい Pod のスケジューリングやクラスターの監視といったシステムにおける重要な操作が影響を受ける可能性がある。

Kubernetes 標準のスケジューラー実装である、Kube-scheduler は主に CPU とメモリの使用率のみを考慮してスケジューリングを行うため、ノード間での負荷不均衡が生じるという課題がある。

Deep Reinforcement Learning を用いた Kubernetes スケジューラー (以下 DRS) [3] は、リソース利用率の向上と負荷分散の最適化を実現することができたが、集中型サービスとしての動作が単一障害点を生むリスクを持っている。そのため、このシステムで障害が発生すると、新しい Pod の割り当てや管理に大きな影響が及び、クラスター全体の機能性が損なわれる可能性がある。

1.2　目的

本研究の目的として、リソースの利用率と負荷分散の最適化を実現した DRS に基づき、分散型のスケジューリングとし、スケジューリングコンポーネントを分

散させることで、一つのコンポーネントに障害が発生しても全体のクラスタースケジューリング機能に影響が及ばないようにする。

この目的を達成するために、分散型のスケジューリングを提案し、スケジューラーがノードのリソース使用状況、Pod のリソース要件、ノード間のネットワーク遅延を学習し、異なる Pod のニーズと優先度に基づいてスケジューリングの決定を行う。

スケジューラーが独自の経験から学習し、Kubernetes クラスター環境の変化に自律的に適応できるようになる。この研究によって、システムの耐障害性が向上し、信頼性と効率が高まり、継続的なサービス提供が可能になるだけでなく、動的なリソース管理によりコスト効率の良い運用が実現し、高いサービス品質を維持しながら、クラウドインフラストラクチャの持続可能性と効率性が大幅に向上する。

1.3 本論文の構成

本論文の構成を以下に示す。

第 1 章では、本研究の背景と目的を述べる。

第 2 章では、本研究において利用する関連技術および Kubernetes スケジューリングに関する既存研究について説明する。

第 3 章では、分散可能なコンテナ間動的スケジューリングの最適化を実現するための提案手法について述べる。

第 4 章では、提案手法に基づく分散型スケジューラーの設計と実装について述べる。

第 5 章では、本提案手法の有効性を確認するため行った実験と評価について述べる。

第 6 章では、本研究を総括とともに今後の展望について述べる。

第2章　関連技術と関連研究

本章で本研究の関連技術および関連研究について述べる。関連技術として、本研究で対象としている Kubernetes、分散的な意思決定のスケジューリングを可能にする MARL フレームワークについて述べる。

2.1　関連技術

2.1.1　Kubernetes

Kubernetes は、宣言的な構成管理と自動化を促進し、コンテナ化されたワークロードやサービスを管理するための、ポータブルで拡張性のあるオープンソースのプラットフォームである。本質として、クラスター内の各ノードで特定のプログラムを実行してコンテナを管理、リソース管理を自動化することである。一つのクラスターはマスターノードとワーカーノードで構成される。クラスターのアーキテクチャは図 2.1[4] を示す。

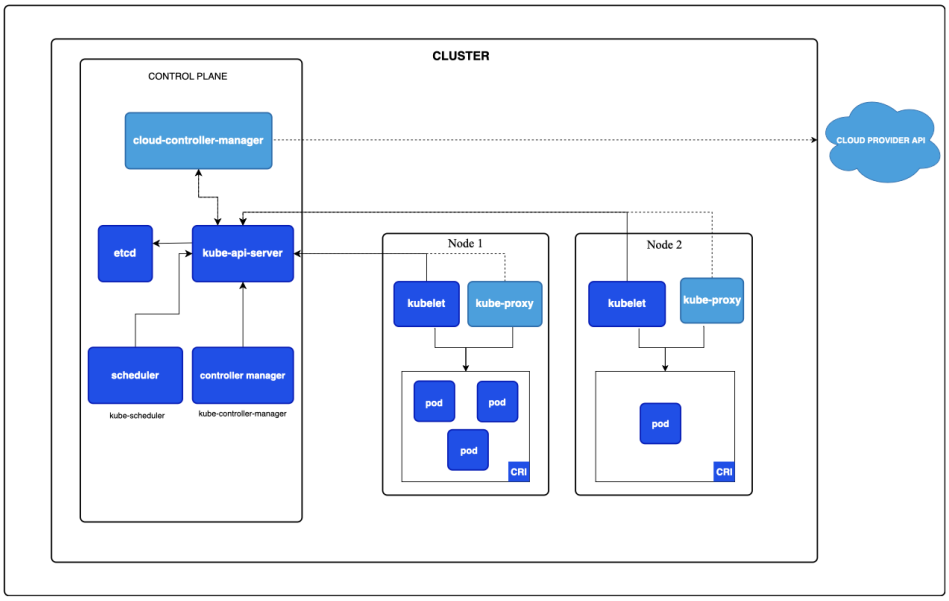

図 2.1: Kubernetes クラスターのアーキテクチャ

マスターノードはクラスターのコントロールプレーンとしてクラスターの意思決定を担当する。以下のコンポーネントを含む。

- **API サーバー** クラスター管理の統一されたエントリーポイント。Kubernetes コンポーネントと外部クライアントがクラスターリソースとやり取りして管理するための REST API インターフェイスを提供する。
- **Etcd** 分散キーバリューストアデータベース。クラスター全体の設定情報と状態データを保存、データ変更のリアルタイム通知を実現するウォッチ機能をサポートしている。
- **コントローラーマネジャー** クラスターのステータス維持担当。複数のコントローラーを管理して、リソースが変更された場合、必要な調整を自動的に実行する。
- **スケジューラー** 新しく作成されたポッドを監視し、実行するノードを選択する。

ワーカーノードはクラスターのデータプレーンとしてコンテナの実行環境を提供する。以下のコンポーネントを含む。

- **Kubelet** クラスター内の各ノードで実行されるエージェント。コンテナが Pod 内で実行されていることを確認する。
- **Kube-proxy** クラスター内の各ノードで実行されるネットワークプロキシ。ノード上のネットワークルールを維持する。
- **Container Runtime** Kubernetes 環境内でのコンテナの実行とライフサイクルの管理を担当する。

2.1.2 MARL

Multi-Agent Reinforcement Learning(以下は MARL)[5] は、共有環境で共存する複数の学習エージェントの行動の学習に焦点を当てている強化学習の手法である。MARL における各エージェントは自らの報酬に励まされ、自らの利益を促進するために行動する。

2.2 関連研究

Lorido-Botran ら [6] が提案した RLSched は、Deep Reinforcement Learning(DRL) に基づくコンテナスケジューラーであり、クラウドデータセンターのリソース利用率の最適化に焦点を当てている。複数のエージェントがそれぞれ異なるコピー

環境で学習を行う A3C(Asynchronous Advantage Actor-Critic) アルゴリズムとマルチエージェントアーキテクチャを通じて、RLSched は動的なリクエスト割り当てにおいて迅速に収束し、アクションシェーピングによってスケジューリング効率を向上させた。この研究における状態空間は、各物理サーバーのリソース使用状況と現在のスケジューリングリクエストを主に含み、単一のリクエストに対して適切な物理マシンを見つけることをスケジューリング最適化の重点とし、できるだけ多くのコンテナが物理マシン上で正常にスケジュールされることを保証する。

Gor Mack Diouf ら [7] は KmMR プラットフォームを提案した。このプラットフォームは、複数のマスターノードアーキテクチャとビザンチンフォールトトレランス機構を実現することで、単一のマスターノードの障害が引き起こす可能性のあるシステム全体の中断の問題を解決し、Kubernetes クラスターの信頼性とシステム全体の弾力性を大幅に向上させる。複数のマスターノードと複数のノードが分散環境で合意形成を行う BFT-SMaRt(Byzantine Fault-Tolerant State Machine Replication Tool) プロトコルを導入することで、クラスターは性能を低下させることなく、動的にリソース配分を調整および最適化し、より広範なアプリケーションシナリオとより高いサービス需要をサポートする。

Espinosa ら [8] は、ポリシー更新幅を制限するクリッピング手法を導入した PPO(Proximal Policy Optimization) に基づく深層強化学習手法を提案し、Kubernetes クラスターの消費電力の最適化を図った。この研究では、Kubernetes 環境のマスターノード上のスケジューラーを単一のエージェントとして扱い、カスタマイズされた Kubernetes オペレーターを介して意思決定を実行する。このスケジューラーは、新規ポッドの割り当てだけでなく、クラスター全体のエネルギー消費を最適化するために実行中のポッドの再割り当ても考慮している。実験結果によると、本手法によりクラスターの消費電力を 19〜24%削減することを確認できた。しかしながら、本研究では単一エージェントのシナリオのみを考慮しており、マルチクラスター環境におけるスケーラビリティやデータプライバシーに関する課題が残されている。

Ajay Kattepur ら [9] は Madelyn フレームワークを提案した。このフレームワークは、Kubernetes とデータセンターネットワークを統合したマルチエージェント強化学習手法であり、従来の Kubernetes スケジューラがネットワークトラフィックを考慮せずにリソース使用率が偏るという課題を解決することを目的としている。本フレームワークでは、リーフ、スパイン、スーパースパインの三種類のネットワークエージェントが、データセンター内のトラフィックスケジューリングを最適化し、負荷分散を向上させる。また、ポッドエージェントがタスクの移行、自動スケーリング、スケジュール最適化を行い、Kubernetes のリソース管理を強化する。実験結果から、Madelyn は従来の負荷分散手法と比較してスループットを 50%向上させ、コンピューティングリソースとネットワークリソースの無駄を削減することが示されており、5G およびクラウドコンピューティング環境におけるより優れたスケジューリングソリューションを提供する。

第3章　提案手法

本章では、MARL を用いて分散型のスケジューリングを実現するための提案手法について述べる。

3.1　提案手法の概要

Kubernetes クラスターの運用においてマスターノードは全てのワーカーノードのリソース状態を監視し、スケジューリングの意思決定を一括して行う。たとえば、特定のワーカーノードに障害が発生した場合は他のノードへワークロードを移行したり、新規の Pod デプロイメント要求に対して最適なノードを選択したりする。その結果、マスターノードに障害が発生した場合、クラスター全体のスケジューリング機能が停止し、システムの可用性が著しく低下する。

本研究では、Kubernetes クラスターにおける分散型スケジューリングを実現するため、分散型 MARL に基づく新しい手法を提案する。本手法は、従来の集中型アーキテクチャから分散型アーキテクチャへの移行且つ、リソース管理および負荷分散の最適化を目指すものである。

従来のアーキテクチャでは図 3.1 に示すように、DRS Decision Maker と DRS スケジューラーがマスターノードに集中配置され、TCP コネクションを介して Pod スケジューリング情報やノードの状態情報を収集する集中型の意思決定システムを採用しているため、単一障害点のリスクが存在する。

図 3.2 は提案するアーキテクチャを示している。分散配置として、各ワーカーノードにスケジューラーと Decision Maker を配置し、自律的なスケジューリング決定を可能にするとともに、ノード間での直接的な状態共有と決定の調整を実現する。また、分散型意思決定として、Decision Maker 間の相互通信を通じた協調的な判断により、ローカルでの即時的な判断と全体最適の両立を図り、分散合意プロトコルによってスケジューリングの整合性を確保する。さらに、高可用性の観点から、単一ノードの障害が全体に影響を与えにくい構造を採用することで、システム全体の耐障害性を向上させ、継続的なサービス提供を実現する設計となっている。

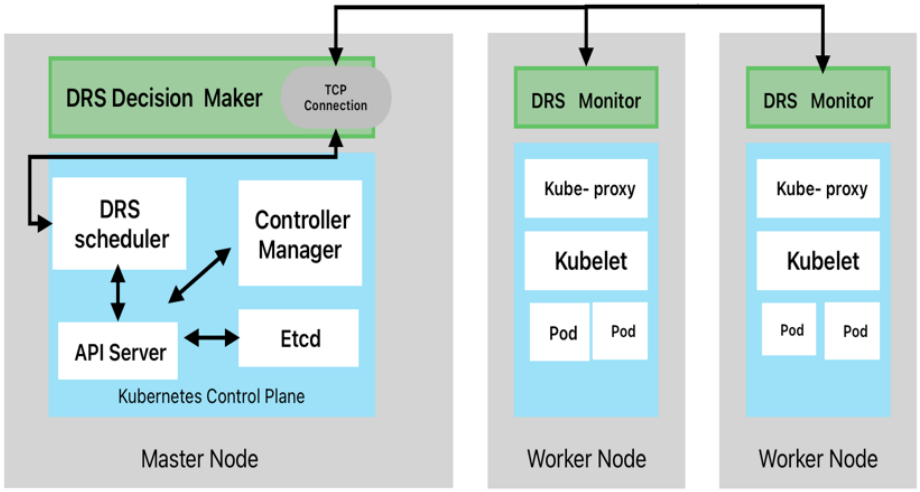

図 3.1: 従来のアーキテクチャ

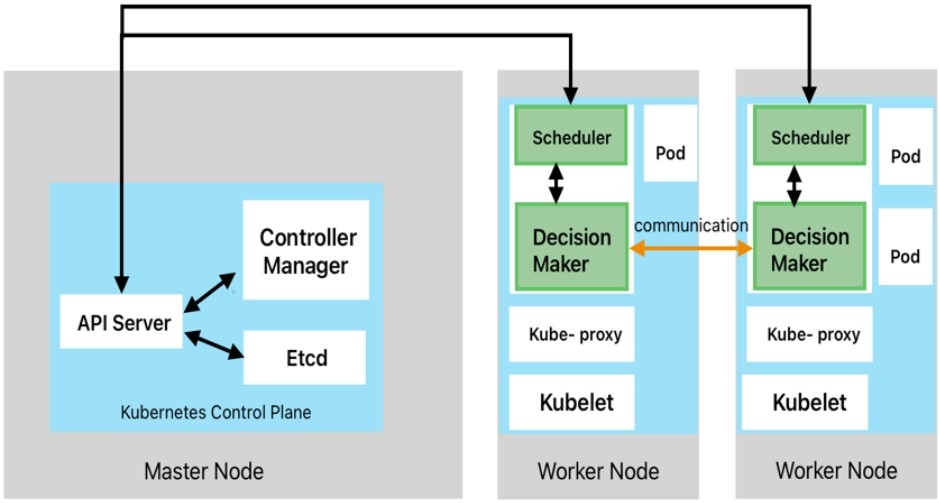

図 3.2: 提案するアーキテクチャ

3.2 本提案方式における MARL モデルの定義

本節では、本研究で提案する MARL モデルについて説明する。このモデルは、分散環境における各ノードが独立して意思決定を行いつつ、協調的な行動を通じて全体的なシステム最適化を目指す構造を持つ。また、深層強化学習の手法を活

用し、複雑かつ動的なタスクスケジューリング問題に柔軟に対応できる。

3.2.1 MARL モデルの概要

MARL は、複数の自律的なエージェントが同時に学習・行動する環境において、システム全体のパフォーマンスを最適化するための手法である。単一エージェントの強化学習と異なり、MARL では各エージェントの行動が他のエージェントの状態と報酬に影響を与える。各エージェントは、環境からフィードバックされる報酬信号に基づいて行動を学習し、報酬を最大化することを目指している。図 3.3[10] にはエージェントと環境の相互作用モデルを示す。エージェントははじめに現在の状態を観測し、その情報を基に最適と思われる行動を選択する。この行動が環境に影響を与えると、環境は新しい状態とともに、行動の結果に応じた報酬をエージェントに返す。以下に、MARL モデルの構成要素であるエージェント、状態空間、報酬関数、ポリシー、そして行動空間について詳しく説明する。

- **エージェント**　環境と相互作用しながら学習を通じて意思決定を行う主体である。各エージェントは、局所的な観測情報に基づいて自律的に行動しながら、他のエージェントの行動を考慮した意思決定を行う。

- **状態空間**　環境全体または各エージェントが観測可能な情報を表し、システムの現在の状況を示す。これには、エージェント自身の状態情報に加え、他のエージェントとの関係性を示す情報も含まれる。

- **報酬関数**　エージェントの行動を評価するための指標である。システム全体の最適化を目指し、複数の評価指標を組み合わせて定義する。個々のエージェントの局所的な目標達成度と、システム全体のパフォーマンスの両方を考慮する。

- **行動空間**　各エージェントが実行可能な行動を表現する空間である。エージェントはこの行動空間から、現在の状態に基づいて最適な行動を選択する。

- **ポリシー**　エージェントが現在の状態に基づいてどの行動を選択するかを決定するルールまたは戦略である。学習を通じて最適なポリシーを獲得することで、エージェントは与えられたタスクを効率的に実行できるようになる。

3.2.2 深層強化学習アルゴリズム

本研究では、複数のノードおよび異なるリソースタイプを持つクラスター環境におけるスケジューリング問題を効果的に解決するために Q 学習 (Q-Learning) と

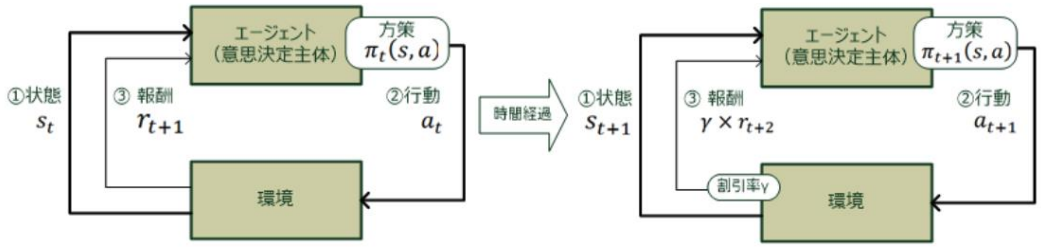

図 3.3: エージェントと環境の相互作用モデル

深層学習 (Deep Learning) を組み合わせた DQN(Deep Q-Network)[11] アルゴリズムを採用した。

Q 学習は、行動価値関数 (Q 関数) を通じて最適な行動を選択する手法であり、環境からのフィードバックに基づいて Q 値表を構築し、最適なポリシーを学習する。しかし、コンテナスケジューリングのような複雑な問題では、状態空間が広大であり、従来の Q テーブルを使用してすべての状態と行動を記録、索引付けすることが困難である。これに対して DQN は、図 3.4 に示すようニューラルネットワークを用いて Q 関数を近似することで、高次元の状態空間にも対応可能な手法として提案された。ニューラルネットワークの学習を安定化させるため、DQN は二つの重要なメカニズムを導入している。

- **エクスペリエンスリプレイ**：エージェントの状態、行動、報酬、次の状態といった経験をメモリに保存し、学習時にはこれらの経験からランダムにサンプリングしてミニバッチ学習を実施する。このランダム化によってデータ間の相関が減少し、より安定した学習が可能となる。
- **ターゲットネットワーク**：DQN は二つの独立したネットワークを使用する。一つは Q ネットワークと呼ばれ、現在の状態から各行動の Q 値を予測する役割を担う。もう一つはターゲットネットワークと呼ばれ、学習時のターゲットとなる Q 値の計算に使用される。学習の収束を助けるため、定期的に更新されるターゲット Q ネットワークを使用して、推定される価値関数の目標値を生成。

DQN は以下のような学習手順で学習する [12]。

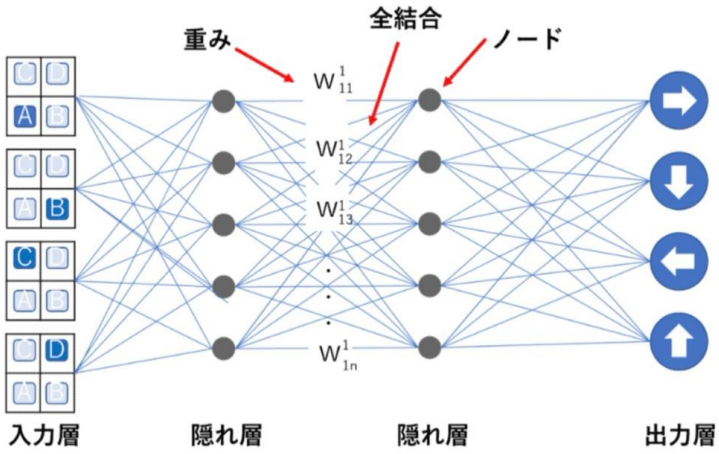

図 3.4: DQN のニューラルネットワーク構造

ステップ１：Q-network に状態 s_t を入力し、$Q(s_t, a_t)$ を求め。

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max Q(s',a') - Q(s,a)\right] \tag{3.1}$$

ステップ２：ε -greedy 法に基づいて行動 a_t を選択し、環境から報酬 R_t と次の状態 S_{t+1} を取得し、それらを保存。

ステップ３：誤差関数を求め、Q ネットワークの重みを更新。

$$L = \left(r + \gamma \cdot \max Q(s',a';\theta^-) - Q(s,a;\theta)\right)^2 \tag{3.2}$$

ステップ４：設定した試行ステップごとにターゲットネットワークの重みを更新。

ステップ５：過去に蓄積された経験 $(s_t,\ a_t,\ R_t, S_{t+1},\ Q(s_t,a_t))$ からランダムにサンプルを取得。

ステップ６：そのデータを利用して、設定したミニバッチ数でターゲットネットワークの重みの学習を行う。

ステップ７：ターゲットネットワーク k の重みを Q ネットワークと同期。

ステップ８：更新された Q ネットワークを使用し、ステップ１からステップ３のプロセスを繰り返して学習を進める。

3.2.3　エージェントの協力と通信メカニズム

複数のワーカーノードが独立して動作する分散環境では、各エージェントが持つ情報は必然的に局所的なものとなり、これが非効率な資源割り当てやワークロー

ドの不均衡といった問題を引き起こす可能性がある。このような課題に対応するため、エージェント間の協力と情報共有により、グローバルな環境に対するより包括的な理解が形成され、より最適な意思決定戦略を共同で策定することが可能となる。

情報共有　本提案手法では、情報共有において間接的な協力メカニズムを採用している。Kubernetes クラスターではすでに Etcd という分散キーバリューストアが実装されており、クラスター全体の状態情報が集約されている。このため、各ノードは Etcd を介してクラスターの状態情報に容易にアクセスすることが可能となっている。この既存のインフラストラクチャを活用することで、システムに新たな複雑性を追加することなく、効率的な情報共有基盤を実現することができる。さらに、各ノードが互いに直接通信を行う場合、ノード数の増加に伴い通信量が指数関数的に増大する可能性があり、Etcd を介した間接的な情報共有では、各ノードは Etcd とのみ通信を行い、ノード間の直接的な通信量を大幅に削減できる。

最適な意思決定　本提案手法では、効率的な意思決定を実現するために集中型の協力メカニズムを採用している。この方式では、リーダー選出機能を通じて、リーダーエージェントが他のエージェントからの情報を集約し、クラスター全体における最適化なスケジューリングを行う。このリーダーエージェントは、集められたデータに基づき、最適的ケジューリングの決定を行う。N 個のノードが存在し、各通信データが A バイトである場合を考える。完全共有型メカニズムの場合、各ノードは他のすべてのノードとデータを共有する必要がある。各ノードは (N-1) 個の他のノードにデータを送信するため、N 個のノードそれぞれについてこの送信が発生する。したがって、1 回のスケジューリングにおける総通信量 T は $T = A * N * (N - 1)$ である。一方で、集中型メカニズムでは、各ノードがリーダーノードにのみデータを送信するため、リーダーノードを除く（N-1）個のノードがデータを送信する。その結果、1 回のスケジューリングにおける総通信量 T は $T = A * (N - 1)$ に削減され、これにより通信の効率が大幅に向上し、スケーラビリティとシステムの効率が高まる。

第4章　設計と実装

本章では、前章の提案手法を実現するための分散型スケジューラーのアーキテクチャの設計や実装について述べる。

4.1　本提案方式における MARL モデル設計

4.1.1　MARL モデルのアーキテクチャ

本研究においては、本提案方式における MARL モデルを実装した。そのアーキテクチャを図 4.1 に示す。

図 4.1 に示す通り、各ワーカーノードに独立した分散型スケジューラーが配置されている。各エージェント内に組み込まれた MARL モデルは強化学習に基づくポリシーを学習し、効率的なスケジューリングを実現する。エージェントは、Kubernetes API を介してノードのリソース状態やコンテナの実行状況を取得し、これらの情報に基づいて最適なスケジューリング決定を行う。

各エージェントは独立して動作しながらも、状態共有コンセンサスメカニズムを通じて他のエージェントと学習結果を共有する。この共有メカニズムにより、各エージェントは他のノードの状態や決定を考慮しながら、システム全体として最適な資源割り当てを実現することが可能となる。

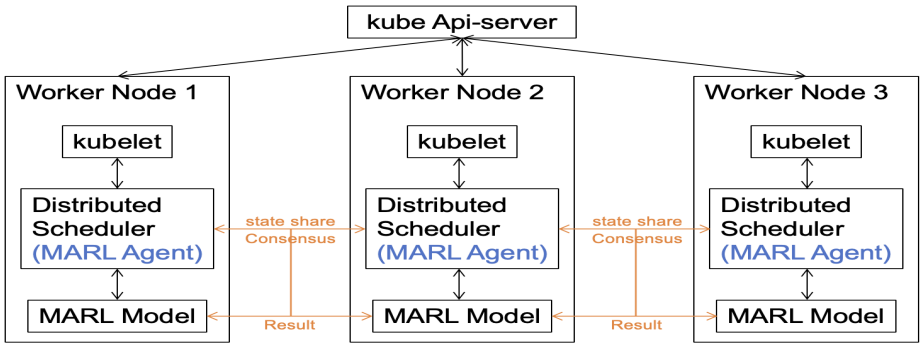

図 4.1: Kubernetes 環境で MARL モデル設計

4.1.2 Kubernetes 環境で MARL モデルの実装

提案する MARL モデルに含まれる要件を以下に示す。

エージェント：各ノードに配置されているスケジューラー

状態空間：本モデルでは、ローカル状態、グローバル状態、および待機中の Pod 情報の 3 種類の状態を定義する。ローカル状態は各エージェントが直接観測可能な情報を指し、グローバル状態はクラスター全体に関する情報を表す。また、待機中の Pod 情報はスケジューリングが必要な Pod に関連する情報を含む。具体的な状態空間として、ローカル状態には CPU 使用率、メモリ使用率、ディスク I/O 情報、ノードの健康状態、現在ノード上で稼働中の Pod の数を含み、グローバル状態にはクラスター全体の CPU 使用率、メモリ使用率、ディスク I/O 情報を含む。待機中の Pod 情報には、Pod が要求する CPU 使用率やメモリ使用率が含まれている。ローカル状態の取得方法を表 4.1 に示す。

表 4.1: システム状態の取得方法

項目	取得コマンド	動作の仕組み
CPU 使用率	`top -b -n 3 -d 0.01`	3 回のサンプリングを 0.01 秒間隔で実行し、各サンプリングの CPU アイドル率を取得。
メモリ使用率	`head -n 3 /proc/meminfo`	/proc/meminfo ファイルから MemTotal、MemFree、MemAvailable を抽出。
ディスク I/O 情報	`iotop -k -P -o -n 2 -b`	I/O アクティビティが発生しているプロセスの情報を 2 回サンプリングし、その中から合計の読み書き速度を計算。
ノードの健康状態	Kubernetes API	API サーバーから HTTP リクエストを介してノードの健康状態を収集。
稼働中の Pod の数	Kubernetes API	API サーバーから HTTP リクエストを介して現在ノード上で稼働中の Pod の数を収集。

グローバル状態は Kubernetes API を利用してクラスター全体のリソース状態を取得する。具体的には、CoreV1API を通じて各ノードから CPU、メモリ、ストレージの容量と割り当て可能量を取得し、数値に変換して全ノードの合計値を計算、総割り当て可能量と総容量を比較して、CPU、メモリ、ストレージの利用率比率を計算する。

待機中の Pod 情報はグローバル状態と同じく Kubernetes API を利用して Pod に含まれる各コンテナのリソース要求を取得する。リソース要求が設定されている場合、CPU とメモリの要求量を抽出し、抽出した CPU 要求量はミリコア単位からコア単位に変換され、メモリ要求量は MiB 単位で取得される。これらの値をノードの総 CPU コア数や総メモリ容量で正規化し、相対的なリソース要求率を計算する。

報酬関数：本モデルの報酬関数はローカル報酬とグローバル報酬の両方を設計されている。同様の計算ロジックに基づいて評価され、リソース使用率に応じて 4 段階の報酬値が定義される。

ソースの効率的な利用を促進するため、使用率が 50%未満の場合、使用率と最適下限値 (50%) の差分に基づいた負の報酬が与えられる。50%から 80%の範囲では使用率が最適であるため、正の報酬が付与される。報酬は使用率の正規化によって計算され、この範囲を維持することが奨励される。システムの安定性を確保するため、使用率が 80%を超えた場合、最適上限値 (80%) との差分に基づいた負の報酬が与えられる。特に使用率が 90%以上に達した場合は、システムの安定性に重大なリスクが生じる可能性があるため、大きな負の報酬 (-60) が付与される。ローカル報酬とグローバル報酬は、CPU、メモリ、ストレージの使用率に基づいた報酬の合計であり、最終的な報酬は、ローカル報酬とグローバル報酬の合計から要素数で割った平均値で算出される。

行動空間：本モデルの行動空間は、Pod のスケジューリング決定を表現する離散的な空間として定義される。具体的には、各エージェントは担当するノードに対して、待機中の Pod を受け入れるか否かの 2 値的な決定を行う。

ポリシー：Deep Q-Network アルゴリズムを採用してエージェントのポリシーを実装している。詳細については 4.1.3 で説明する。

4.1.3 学習アルゴリズムの実装

提案するモデルの中心機能である意思決定者は、3.2.2 で説明した DQN アルゴリズムを基に実装されている。その具体的学習プロセスをアルゴリズム 1 に示した。

この意思決定者は、環境に関する履歴情報を保存するために、容量 N の経験プール D を利用している。初期段階では、意思決定者は行動選択に使用される Q 関数をランダムな値で初期化する。学習プロセスが開始されると、意思決定者は継続的にスケジューラーからのメッセージを待ち受ける。スケジューリング要求を受信すると、意思決定者は Pod 情報を処理する。次に、意思決定者は現在のノード状態 $Node_t^i$ とクラスター状態 $Cluster_t^i$ を取得し、クラスター状態ベクトル $State_t$ を生成する。ベクトル内の値は、事前に設定された使用率範囲に従って正規化される。意思決定者は、すべての可能な行動について Q を計算し、ε-グリーディー戦略を用いて最大の Q 値を選択する。また、1-ε の確率でランダムに行動を選択する。スケジューラーに行動 Actiont を返した後、意思決定者は T_i 秒待機し、その

後再度ノードの状態を取得する。T_i は 15 秒に設定されており、これは Pod が対応するノードにバインドされ、実行が開始されることを確実にするためである。このスケジューリングプロセスの $(State_t、\ Action_t、\ Reward_t、\ State_{t+1})$ は経験プールに格納される。経験プールが満杯になると、意思決定者は経験プールからランダムな遷移をサンプリングし、ニューラルネットワークの学習を開始する。評価ネットワークは損失を評価し、勾配降下ステップを実行する。ターゲットネットワークは、100 回のイテレーションごとに評価ネットワークのパラメータをコピーして重みを更新する。このプロセスは学習が終了するまで繰り返される。

Algorithm 1 意思決定者の訓練アルゴリズム

Initialize experience pool D to capacity N

Initialize action-value function Q with random weights

Initialize socket connections with leader scheduler

for $episode = 1, M$ **do**
  if get scheduling request at time t then **then**
    Get state of worker node $[Node_n]$ and cluster $[Cluster]$
    Combine the node states and the cluster state with the pod resource requirements as the state vector $State_t$
    With probability ϵ select a random action $Action_t$, otherwise select $Action_t = \max_a Q(\phi_t, a)$
    Return the result $Action_t$ to scheduler
    Get state of worker node $[Node'_n]$ and cluster$[Cluster']$ after time T_i
    Calculate the reward $Reward_t$
    Store transition $(State_t, Action_t, Reward_t, State_{t+1})$ in D
    if $D.counter > capacity\ N$ then **then**
      Sample random minibatch of transitions from D
      Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a') & \text{for non-terminal } \phi_{j+1} \end{cases}$
      Perform a gradient descent step on $(y_j - Q(\phi_j, a_j))^2$
    end if
    Copy parameters of eval Network to target Network per K iterations
  end if
end for=0

4.2 Kubernetesとの統合

4.2.1 Kubernetes APIとの連携

本節では、提案するモデルにおけるKubernetes APIとの統合について、主要な連携機能とその実装方法を説明する。モデルの効率的な運用を実現するため、Kubernetes APIの各機能を効果的に活用している。

Podの監視　Kubernetes Watch APIを用いて新規Podの作成イベントを監視している。Watch APIの実装では、イベントハンドラーを設定し、ADDED、MODIFIED、DELETEDなどのイベントタイプに応じて適切な処理を行う。特に新規Pod作成時のADDEDイベントに対しては、Podのリソース要求、制約条件などの情報を解析し、迅速なスケジューリング処理を開始することが可能となっている。

リソース情報の取得　Etcdを介して、Kubernetesクラスターのリソース状態に関する詳細な情報を取得する。これには、各ノードのリソース容量や利用可能量、全クラスターのリソース容量、使用中のリソースおよびPodのリソース要求などが含まれている。この情報は意思決定の基礎として利用され、リソース割り当ての効率化と最適化に寄与する。

Podのバインディングと状態管理　スケジューリング決定後の処理を確実に実行するため、Kubernetes Binding APIを用いてPodを特定のノードにバインドする機能を実装している。さらに、Pod Status APIを活用して実行状態の更新も行っている。

リーダー選出と情報更新 Kubernetesの設定管理を行うための仕組みであるConfigMapを利用してリーダー選出プロセスを実装している。ConfigMapにリーダーノードの情報を保存し、リーダーノードの情報をクラスター全体に共有する。リーダーの情報が変更された場合には、このConfigMapを更新する。リーダー選出プロセスの詳細はアルゴリズム2に示す通りであり、これによりエージェント間の協調を維持しながら、一貫性のある意思決定プロセスを実現している。

4.2.2 分散型スケジューリングプロセス

Kubernetes環境で新しいポッドをデポロイする場合、標準のスケジューラーは次に挙げる5つの段階に従ってスケジューリングを行う。はじめにPrefilter段階でノードに対してリソース情報や条件を確認し、次のfilter段階では基本的な要件を満たすノードをさらに評価を行う。これからのPreScore段階ではポッドが配置できるノードに対してScoreに必要なデータを準備する、この後のScore段階でポリシーに基づいてスコアを計算、ノードに対してスコアを付ける。これを基づいてNormalizeScore段階で各ノードのスコアを標準化、ソートの結果による最適的

Algorithm 2 リーダー選挙アルゴリズム

Start System

Perform Leader Election

```
if node is elected as Leader then
    Update Leader information in configmap
    Start Heartbeat to maintain active status
    Start Socket Server to communicate with other nodes
else
    Operate as Non-Leader
    Before sharing decisions, check if Leader's heartbeat is present
    if Leader is not responding then
        Leader is considered as "Not Alive"
        Attempt to Become the new Leader
        if successful then
            Update Leader information in configmap
            Start Heartbeat and Socket Server
        end if
    end if
end if
Continue normal operations
End =0
```

なノードを決定する。今回提案した分散型スケジューラーは、表 4.2 に示すように３段階に従って設計を行う。

表 4.2: 分散型スケジューリングプロセス

段階	**内容**
Prefilter 段階	各スケジューラーは自分の Node リソースをチェック 自分の Node Anti-Affinity/Affinity をチェック Node の Pod Anti-Affinity/Affinity をチェック Custom Plugin を呼び出し
Filter 段階	MARL モデルを導入、各スケジューラーは自分の行動を決定 行動は受け入れる場合、ノードを評価
Score 段階	エージェントの間に行動と評価を共有され、ソート結果から最適的なノードを選択する。 最後の結果が出たらモデルに反応してモデル更新を行う

4.2.3　カスタムスケジューラーの実装

本節では、提案手法に基づいて実装したカスタムスケジューラーについて説明する。はじめに、図 4.2 に示した、MARL モデルに用いた Kubernetes のカスタムスケジューラーにおける、Pod の作成プロセスについて説明する。ユーザーからのポッド作成リクエストを APIServer が受け取り、その情報を Etcd に保存する。次に、Master-Slave 構成で実装されたスケジューラーシステムにおいて、各ノードに配置されるスケジューラーがポッド情報を監視し、MARL モデルを用いてスケジューリングの決定を行い、その結果は Master スケジューラーに共有され、最適なノードが選定される。その後、Master スケジューラーがバインディングオブジェクトを作成して APIServer に登録し、最終的に Kubelet がポッド情報を監視して Docker にコンテナの実行を指示してポッド状態を更新する。

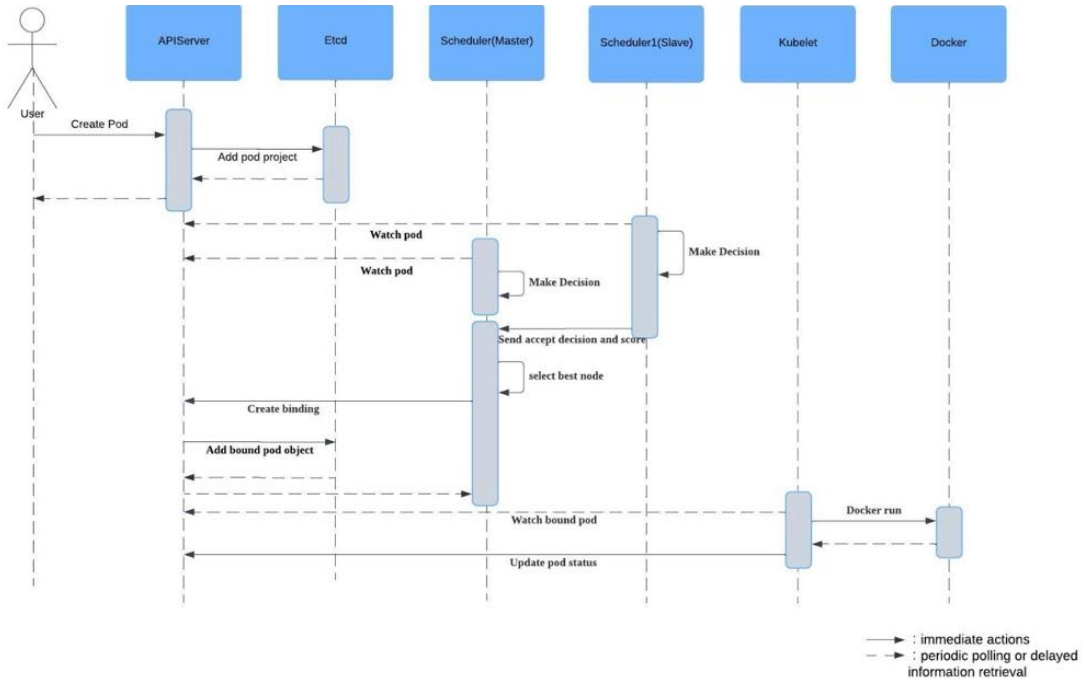

図 4.2: ポッド作成プロセス

次に、図 4.3 に示したカスタムスケジューラーの内部処理フローをについて説明する。スケジューラー起動時にリーダー選出を行い、選出されたリーダーが Heartbeat と socket サーバーを起動し、他のスケジューラーは監視状態に移行する。スケジューリングプロセスにおいて、全てのスケジューラーはポッドの監視と決定作成を担当し、リーダースケジューラーは決定を受け取り、最適ノードの選択とポッドのバインドを行う。一方、非リーダーはリーダーの生存確認、必要に応じた新リーダー選出を行い、決定をリーダーに送信する。フェイルオーバー機能として、非

リーダーが定期的なリーダー生存確認、リーダーダウン検知時の新リーダー選出、新リーダーによるシステム継続性の確保を実装している。

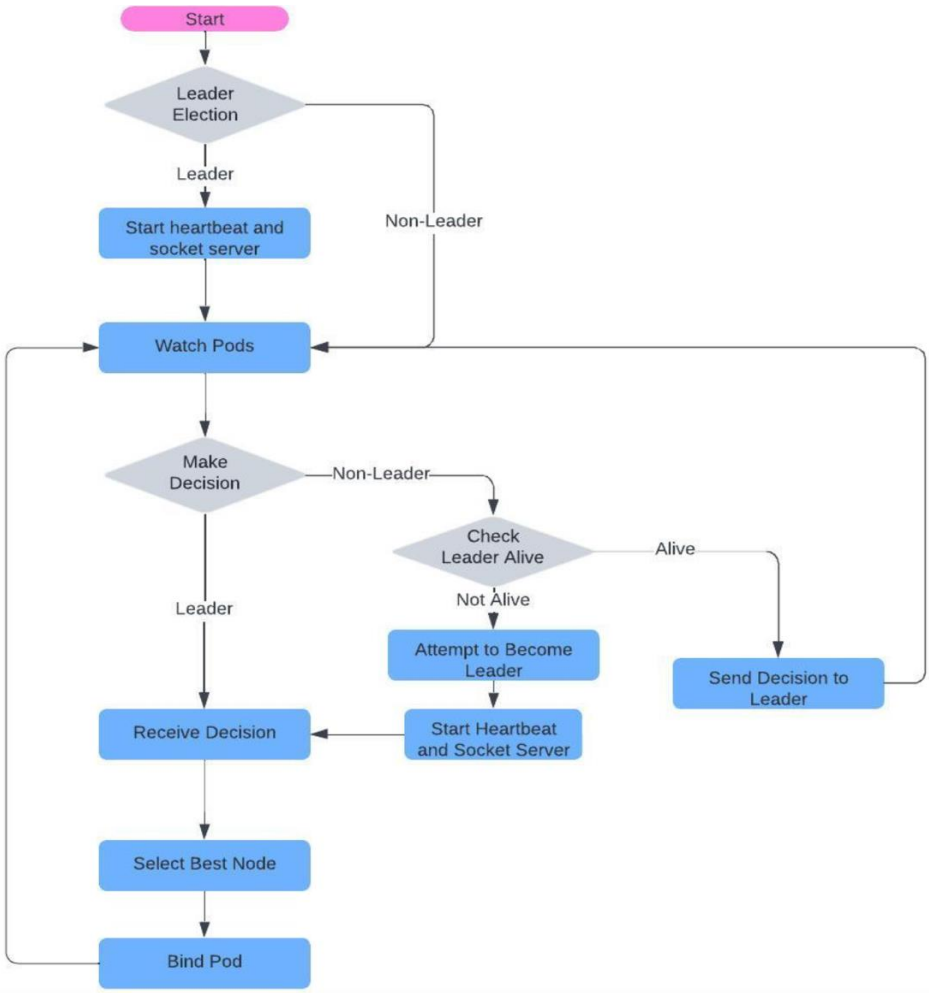

図 4.3: 内部処理フロー

このような実装により、単一障害点を排除した高可用性システムを実現し、リーダー選出とフェイルオーバーの仕組みによってシステムの安定性と信頼性を向上させている。また、MARL モデルを各スケジューラーに分散配置することで、クラスター全体の性能向上と負荷分散を実現している。

第5章　実験と評価

前章までに、提案手法の設計と実装を行った。本章では、実験によって提案手法の効果を検証した。検証は動作確認と性能評価の二つの観点から行う。実装環境で本研究提案する分散型スケジューラーの動作確認を行い、評価環境で既存研究 DRS 比較して性能評価を実施した。

5.1　実現に用いた環境

実現環境に使用した仮想マシン環境と、利用したソフトウェアを表 5.1 に示す。実装環境と評価環境は同じく表 5.1 に示した環境を使用した。コンテナ環境として Kubernetes 利用した。提案手法に基づいて分散型スケジューラーは Python 言語で実装した。

表 5.1: 実現環境

項目	内容
OS	ubuntu 24.04 server
CPU	QEMU vCPUx 4
メモリ	8192MB
コンテナオーケストレーションシステム	Kubernetes (Version 1.31.0)

5.2　実装環境での検証

5.2.1　実装環境

実装環境では、仮想マシンを 3 台使用した。Kubernetes クラスター環境として、1 台のマスターノードと 2 台のワーカーノードで構成される。

5.2.2 動作確認

スケジューラー起動時

図 5.1 と図 5.2 は、それぞれリーダースケジューラーと非リーダースケジューラーが起動する際のプロセスと実行結果を示している。これにより、リーダースケジューラーがモデルの初期化後にリーダーとして選出され、ソケットサーバーの起動及びハートビートの定期更新を行っている様子が確認できる。一方、非リーダースケジューラー側では、モデルの初期化が行われた後、他のスケジューラーをリーダーとして認識、非リーダーとしての役割を適切に受け入れていることが示されている。

```
root@workernode3:~/DMS# python3 main.py
Initialize Model OK
[INFO] Current leader is expired, trying to become the leader...
...
[INFO] I am the new leader: workernode3, term: 20
[INFO] Socket server started on port 1234
[INFO] Heartbeat updated for leader workernode3, term: 20
[INFO] Heartbeat updated for leader workernode3, term: 20
```

図 5.1: リーダースケジューラー起動時

```
root@workernode2:~/test# python3 main.py
Initialize Model OK
[INFO] workernode3 is the leader, term: 20
```

図 5.2: 非リーダースケジューラー起動時

スケジューリング時

図 5.3 と図 5.4 はそれぞれ非リーダースケジューラーとリーダースケジューラーのスケジューリング実行時における動作を示している。図 5.3 の非リーダースケジューラーでは、ポッドのスケジューリング要求を受信し、現在のノード状態を取得後、MARL モデルによるスケジューリング判断を行い、その決定をリーダーノードへ送信。その後、Deep Q-Learning ステップの実行と報酬の計算を行っていることが確認した。一方、図 5.4 のリーダースケジューラーでは、同様にポッドの

スケジューリング要求を受信した後、各ワーカーノードからのスケジューリング決定を収集し、それらの決定に基づいて最適なノード（workernode2）を選択してポッドのバインド処理を実行している。また、ハートビートによるリーダーシップの維持やスケジューリング後の状態更新と報酬計算も行っていることが確認した。

```
2025-01-19 06:15:05.037 [INFO] Get pod: stress-disk
2025-01-19 06:15:05.038 [INFO] get state
2025-01-19 06:15:05.187 [INFO] Action for Pod stress-disk is: accept,
    Node score: 0.58
[INFO] Start the Step thread...
[INFO] Current leader is: workernode3
Decision sent to leader successfully
[INFO] Start  step 1/100
[INFO] Wait for pod execution...
[INFO] State after action: [0.358, 0.15805022447199768,
   0.0006466959635416667, 1, 1, 1.0, 0.9874077246919368,
   0.8999999985340964, 0.0, 0.0, None, None]
[INFO] Reward of this step: -0.395118467131749
[INFO] End dql step 1/100
[INFO] Exit the Step thread!
```

図 5.3: 非リーダーススケジューリング実行時

```
2025-01-19 06:15:05.036 [INFO] Get pod: stress-disk
2025-01-19 06:15:05.037 [INFO] get state
2025-01-19 06:15:05.191 [INFO] Action for Pod stress-disk is: accept,
    Node score: 0.58
[INFO] Start the Step thread...
[INFO] Current leader is: workernode3
2025-01-19 06:15:05.249 [INFO] Received decision for pod stress-disk
   from node workernode2
[INFO] Current leader is: workernode3
Decision sent to leader successfully
2025-01-19 06:15:05.303 [INFO] Received decision for pod stress-disk
   from node workernode3
2025-01-19 06:15:06.249 [INFO] Decisions stress-disk collected  for
   [('workernode2', 0.58), ('workernode3', 0.58)]
2025-01-19 06:15:06.250 [INFO] Best node selected: workernode2 for
   stress-disk
[INFO] Pod stress-disk bound to workernode2
[INFO] Start  step 1/100
[INFO] Wait for pod execution...
[INFO] Heartbeat updated for leader workernode3, term: 20
[INFO] Heartbeat updated for leader workernode3, term: 20
[INFO] Heartbeat updated for leader workernode3, term: 20
[INFO] State after action: [0.054000000000000006, 0.130051239774303,
   0.0, 1, 0, 1.0, 0.9874077246919368, 0.8999999985340964, 0.0, 0.0,
   None, None]
[INFO] Reward of this step: -0.45055941390862175
[INFO] End dql step 1/100
[INFO] Exit the Step thread!
```

図 5.4: リーダーススケジューリング実行時

以上から、各スケジューラーで独立してスケジューリング判断を行い、リーダースケジューラーがそれらの決定を集約して最終的なスケジューリングを実行する分散型アーキテクチャが正常に機能していることが確認した。

スケーラビリティ

リーダースケジューラー障害の場合、フェイルオーバー機能の検証を行った。検証結果は図 5.5 に示すよう、workernode3 がリーダーとして動作している状態から、リーダーノードの障害を検知し、新しいリーダー選出プロセスが実行される様子が確認した。

```
2025-01-19 07:12:36.205 [INFO] Get pod: stress-video-1
2025-01-19 07:12:36.216 [INFO] get state
2025-01-19 07:12:36.571 [INFO] Action for Pod stress-video-1 is:
   accept, Node score: 0.7
[INFO] Start the Step thread...
[INFO] Current leader is: workernode3
[ERROR] Leader is not alive. Trying to become the leader.
[INFO] I am the new leader: workernode2, term: 1
[INFO] Socket server started on port 1234
[INFO] Heartbeat updated for leader workernode2, term: 1
```

図 5.5: リーダースケジューラー障害の場合

以上から、リーダースケジューラーに障害が発生した場合でも、自動的に新しいリーダーを選出し、スケジューリング機能を継続できることが確認できた。

新しいスケジューラー追加する場合、スケジューラーの拡張性の検証を行った。結果は図 5.6 に示すように、新しいスケジューラー (workernode3) を追加した際、モデルの初期化が正常に完了し、現在のリーダー (workernode2) を正しく認識できていることが確認できた。

```
root@workernode3:~/test# python3 main.py
Initialize Model OK
[INFO] workernode2 is the leader, term: 1
```

図 5.6: 新しいスケジューラー追加の場合

以上から、新しいスケジューラーを追加した場合でも、既存のリーダー選出メカニズムに従って適切に統合され、分散システムの一部として機能することが確認できた。

5.2.3　オーバヘッド計測

計算オーバヘッド

計算オーバヘッドの測定では、スケジューリング実行時におけるリソースの使用状況を 5 秒間隔で約 4 分 30 秒測定した。測定期間中、3 秒間隔で新規ポッドを継続てきにデプロイした。具体的には、スケジューラーが使用する CPU 使用率、メモリ使用量、およびディスク I/O を計測した。図 5.7 にスケジューラーの実負荷下での測定結果を示す。

CPU 使用率として、ポッドデプロイ開始と同時に CPU 使用率が約 11%に上昇、3 秒ごとのポッドスケジューリング時に基本的な CPU 使用率は 8-10%の範囲で推移した。

メモリ使用量について、実メモリサイズ (RSS) は 482 MB から約 493 MB へと徐々に増加しており、これは新しいポッドが生成され、それが物理メモリを消費しているということである。仮想メモリの使用量は 3528 MB から約 3984 MB へと増加しており、ポッド数の増加に対しても急激なメモリ拡張は発生しなかった。

ディスク I/O について、ディスク読み込みは一貫して 0 MB で、これは観測期間中にディスクからのデータ読み取りが発生していないことを示す。ディスク書き込みの方は定期的な小規模書き込みがある。

通信オーバヘッド

通信オーバヘッドでは、リーダースケジューラーに対して非リーダースケジューラーからの新規ポッドに関する情報共有する時の一次的な通信量を測定した。合計 50 個のポッドがデプロイされる結果は表 5.2 に示す。測定結果から、リーダースケジューラーが集中的に情報を受信する一方向の通信パターンを確認できた。新規ポッド一回の受信されたデータ量は 78 バイトから 95 バイトの範囲で、非リーダーノードからリーダーノードへの通信が最小化されていることが確認できた。

表 5.2: 通信量オーバヘッド測定結果

項目	内容
最小受信データ量	78 バイト
最大受信データ量	95 バイト
平均受信データ量リ	83.1 バイト

遅延オーバヘッド

遅延オーバヘッドでは、スケジューラーが新規ポッドに対してスケジューリングの意志決定の所要時間を計測した。図 5.8 では合計 45 個のポッドがデプロイされる実験結果を示す。

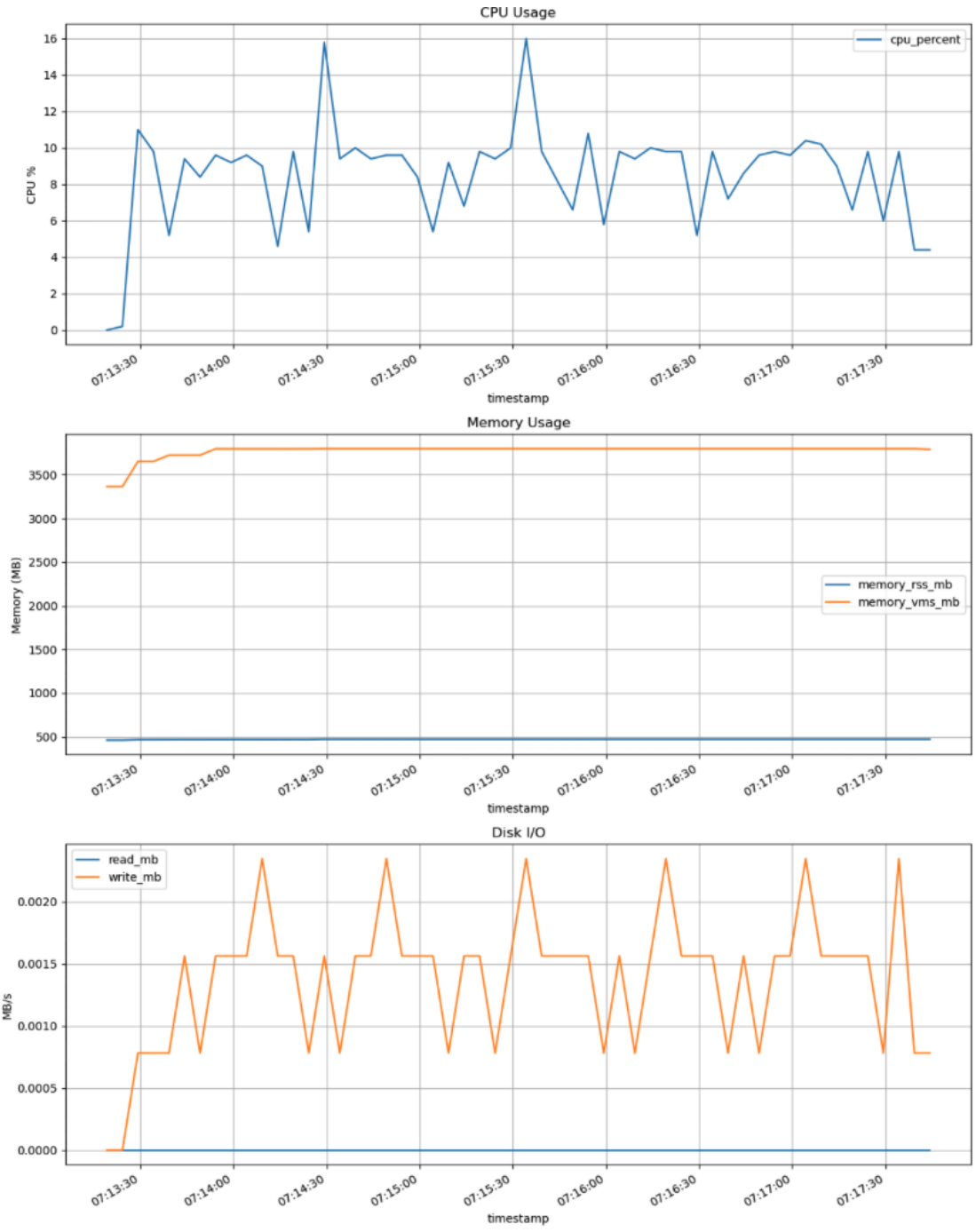

図 5.7: 計算オーバヘッド測定結果

スケジューリング意思決定の時間は 0.001 秒から 0.008 秒の範囲内であり、平均遅延時間は 0.003 秒である。異なるタイプのポッドに対してスケジューリングが均

等に迅速に処理されており、意志決定の時間には、時間が経過しても顕著な増減の傾向が見られない。この結果から、スケユーラーが安定して効率的に動作していることとスケジューラーのパフォーマンスが時間とともに低下することはないことがないことが確認した。

```
Pod Action Timing Analysis
--------------------------------------------------------------------------------
Pod Name             Start Time                Action Time               Duration (s)
--------------------------------------------------------------------------------
stress-video-1       2024-11-25 07:43:28.779   2024-11-25 07:43:28.787   0.008
stress-disk-1        2024-11-25 07:43:34.304   2024-11-25 07:43:34.309   0.005
stress-net-1         2024-11-25 07:43:39.798   2024-11-25 07:43:39.800   0.002
stress-video-2       2024-11-25 07:43:45.382   2024-11-25 07:43:45.383   0.001
stress-disk-2        2024-11-25 07:43:50.951   2024-11-25 07:43:50.954   0.003
stress-net-2         2024-11-25 07:43:56.499   2024-11-25 07:43:56.502   0.003
stress-video-3       2024-11-25 07:44:01.919   2024-11-25 07:44:01.920   0.001
stress-disk-3        2024-11-25 07:44:07.512   2024-11-25 07:44:07.515   0.003
stress-net-3         2024-11-25 07:44:12.819   2024-11-25 07:44:12.821   0.002
stress-video-4       2024-11-25 07:44:18.139   2024-11-25 07:44:18.141   0.002
stress-disk-4        2024-11-25 07:44:23.827   2024-11-25 07:44:23.835   0.008
stress-net-4         2024-11-25 07:44:29.447   2024-11-25 07:44:29.451   0.004
stress-video-5       2024-11-25 07:44:35.073   2024-11-25 07:44:35.074   0.001
stress-disk-5        2024-11-25 07:44:40.759   2024-11-25 07:44:40.760   0.001
stress-net-5         2024-11-25 07:44:46.358   2024-11-25 07:44:46.363   0.005
stress-video-6       2024-11-25 07:44:51.924   2024-11-25 07:44:51.928   0.004
stress-disk-6        2024-11-25 07:44:57.568   2024-11-25 07:44:57.570   0.002
stress-net-6         2024-11-25 07:45:03.169   2024-11-25 07:45:03.171   0.002
stress-video-7       2024-11-25 07:45:08.776   2024-11-25 07:45:08.779   0.003
stress-disk-7        2024-11-25 07:45:14.341   2024-11-25 07:45:14.345   0.004
stress-net-7         2024-11-25 07:45:19.916   2024-11-25 07:45:19.919   0.003
stress-video-8       2024-11-25 07:45:25.470   2024-11-25 07:45:25.471   0.001
stress-disk-8        2024-11-25 07:45:31.062   2024-11-25 07:45:31.064   0.002
stress-net-8         2024-11-25 07:45:36.604   2024-11-25 07:45:36.605   0.001
stress-video-9       2024-11-25 07:45:42.179   2024-11-25 07:45:42.181   0.002
stress-disk-9        2024-11-25 07:45:47.727   2024-11-25 07:45:47.729   0.002
stress-net-9         2024-11-25 07:45:53.256   2024-11-25 07:45:53.258   0.002
stress-video-10      2024-11-25 07:45:58.778   2024-11-25 07:45:58.781   0.003
stress-disk-10       2024-11-25 07:46:04.305   2024-11-25 07:46:04.309   0.004
stress-net-10        2024-11-25 07:46:09.592   2024-11-25 07:46:09.595   0.003
stress-video-11      2024-11-25 07:46:15.158   2024-11-25 07:46:15.159   0.001
stress-disk-11       2024-11-25 07:46:20.785   2024-11-25 07:46:20.786   0.001
stress-net-11        2024-11-25 07:46:26.231   2024-11-25 07:46:26.232   0.001
stress-video-12      2024-11-25 07:46:31.822   2024-11-25 07:46:31.823   0.001
stress-disk-12       2024-11-25 07:46:37.480   2024-11-25 07:46:37.482   0.002
stress-net-12        2024-11-25 07:46:43.103   2024-11-25 07:46:43.105   0.002
stress-video-13      2024-11-25 07:46:48.665   2024-11-25 07:46:48.672   0.007
stress-disk-13       2024-11-25 07:46:54.158   2024-11-25 07:46:54.162   0.004
stress-net-13        2024-11-25 07:46:59.715   2024-11-25 07:46:59.716   0.001
stress-video-14      2024-11-25 07:47:05.316   2024-11-25 07:47:05.320   0.004
stress-disk-14       2024-11-25 07:47:10.884   2024-11-25 07:47:10.885   0.001
stress-net-14        2024-11-25 07:47:16.481   2024-11-25 07:47:16.482   0.001
stress-video-15      2024-11-25 07:47:22.072   2024-11-25 07:47:22.073   0.001
stress-disk-15       2024-11-25 07:47:27.626   2024-11-25 07:47:27.629   0.003
stress-net-15        2024-11-25 07:47:33.160   2024-11-25 07:47:33.164   0.004

Summary Statistics
----------------------------------------
Total Pods Processed: 45
Average Processing Time: 0.003 seconds
Minimum Processing Time: 0.001 seconds
Maximum Processing Time: 0.008 seconds
```

図 5.8: 遅延オーバヘッド測定結果

リソースオーバヘッド

リソースオーバヘッドでは、スケジューラーが稼働する際に必要とされる必要な CPU 使用率、メモリ使用量およびディスク I/O を計測した。測定は 5 秒間隔で約 4 分 30 秒に実施し、その測定結果を図 5.9 に示す。

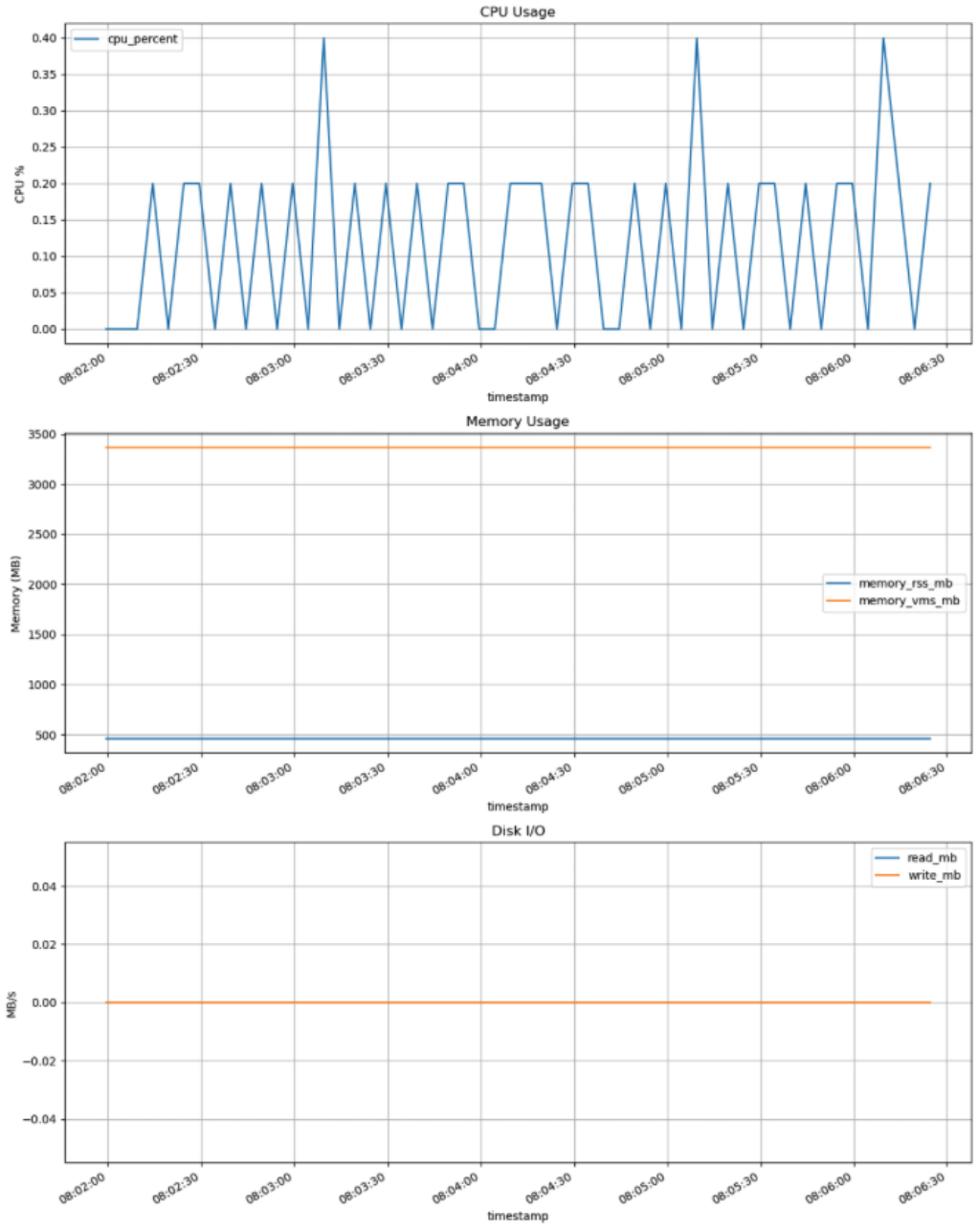

図 5.9: リソースオーバヘッド測定結果

計測データによると CPU 使用率は 0-0.4%の範囲で推移しており、実メモリサイズは常に約 460MB で、仮想メモリサイズは 3365MB ぐらいで安定であり、ディスク I/O については、読み取りと書き込みのバイト数がゼロを示しており、ディスクアクセスがほとんど発生していないことが確認できた。

この結果から、リソースオーバヘッドについては、CPU 負荷が低く、メモリ使用量も安定しており、またディスク I/O もほとんど発生させないことが実証された。

5.3　評価環境での検証

本節では、提案手法に対する評価を行う。提案手法は環境の利用率最適化を目的とした。この目的を実現するために、環境の利用率最適化を実現した既存 DRS を基準として、提案する分散型スケジューラーと比較して評価。本研究では、カスタムシミュレーションツールを用いて分散型スケジューラーの評価を行った。図 5.10 に示すように、シミュレーターは指定されたポッド数、デプロイ間隔、ポッドの種類をパラメータとして読み込み、事前に用意したポッドの Yaml ファイルを基にワークロードを生成する。ポッドのデプロイを順次実行しながら、メトリックツールを用いて CPU 用率、メモリ用率、ネットワーク帯域、ディスク I/O の書き込み速度を測定する。ワークロードテストでは 5 分間合計 105 個ポッド連続でシミュレーションを実施し、リソース使用率の変化を観測する。

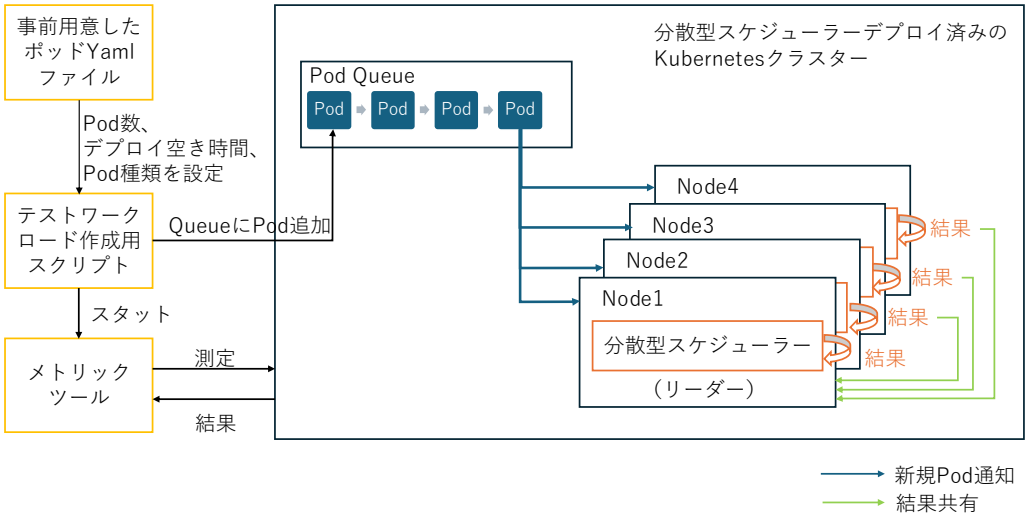

図 5.10: 評価用シミュレーター

5.3.1　評価環境

評価環境では、仮想マシンを 5 台使用した。評価用クラスター環境として、1 台のマスターノードと 4 台のワーカーノードで構成される。

5.3.2　ワークロードテスト

表 5.3 に示す三つのマイクロサービスを用いて均等分布ワークロードテストとランダムワークロードテストを行った。既存研究の DRS と提案する分散型スケ

ジューラーを CPU、メモリ、ネットワーク帯域使用率とディスク I/O について比較した。

表 5.3: マイクロサービス情報

アプリケーション	種類	説明	イメージ
stress-video	CPU 集約型	2 つの CPU スレッドで高負荷を生成、各スレッドが CPU 計算を実行し、プロセッサリソースを最大限に消費する	polinux/stress
Stress-net	ネット集約型	270 秒間、指定したホストに対して 50 バイトの UDP パケットを 1 ミリ秒間隔で連続送信する	utkudarilmaz/hping3
stress-disk	I/O 集約型	1 つのスレッドが 256KB のデータを繰り返しディスクに書き込むことで、ストレージに負荷をかける	polinux/stress

均等分布ワークロード

均等分布ワークロードでは、三種類のマイクロサービスは 3 秒ごとに 1:1:1 の比率で合計 105 個ポッドデプロイした。評価結果は図 5.11 に示す。

環境の CPU 使用率では、安定した負荷の下で分散型スケジューラーと DRS の両方がほぼ同じ 100%に達している。メモリ使用率においては、DRS は時間の経過とともに使用率が増加していくのに対し、分散型スケジューラーは約 15%の低い使用率を安定して維持している。ネットワーク帯域使用率では、分散型スケジューラーと DRS の両者ともに安定した負荷の下で約 175Kb/s を維持しており、二つのスケジューラー間で大きな差は見られない。ディスク I/O においては、DRS の書き込み速度が平均 778Kb/s、最大 1058.02Kb/s の範囲で大きく変動しているのに対し、分散型スケジューラーでは平均 501Kb/s、最大 1680.33Kb/s の範囲で比較的小さな変動が確認した。

分散型スケジューラーは、均等分布ワークロードにおいてはメモリ使用率の効率化とディスク I/O の安定性の面で、DRS よりも優れた最適化を実現できた。ネットワーク帯域使用率や CPU 使用率でも安定したパフォーマンスを維持しており、全体的にリソース管理の効率が向上していることを確認した。

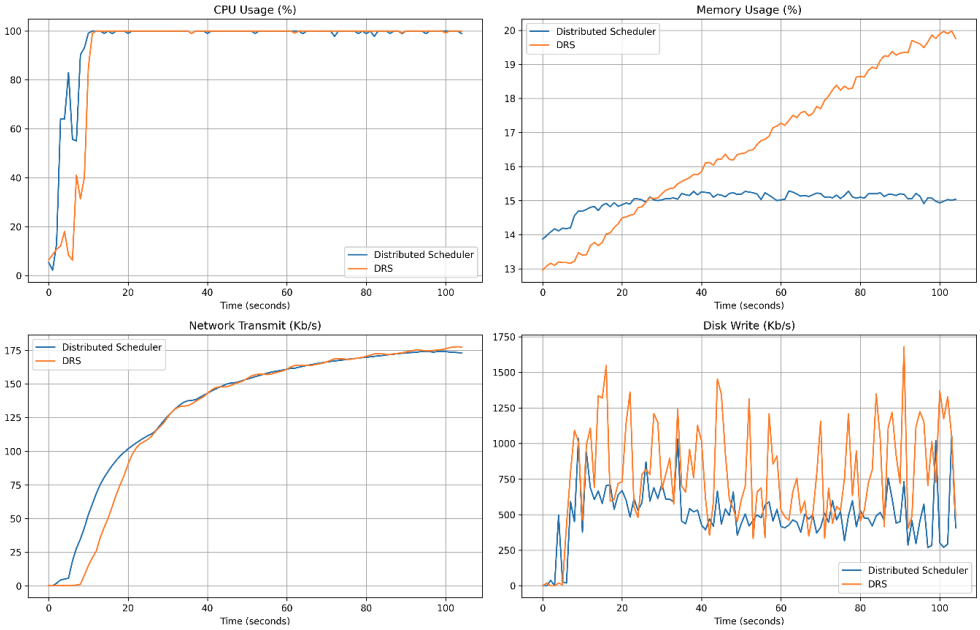

図 5.11: 均等分布評価結果

ランダムワークロード

ランダムワークロードでは、三種類のマイクロサービスはランダムな間隔に 1:1:1 の比率で合計 105 個ポッドデプロイされた。デプロイ時間は平均値 5、標準偏差 2 の正規分布に従うように設定されている。評価結果を図 5.12 に示す。

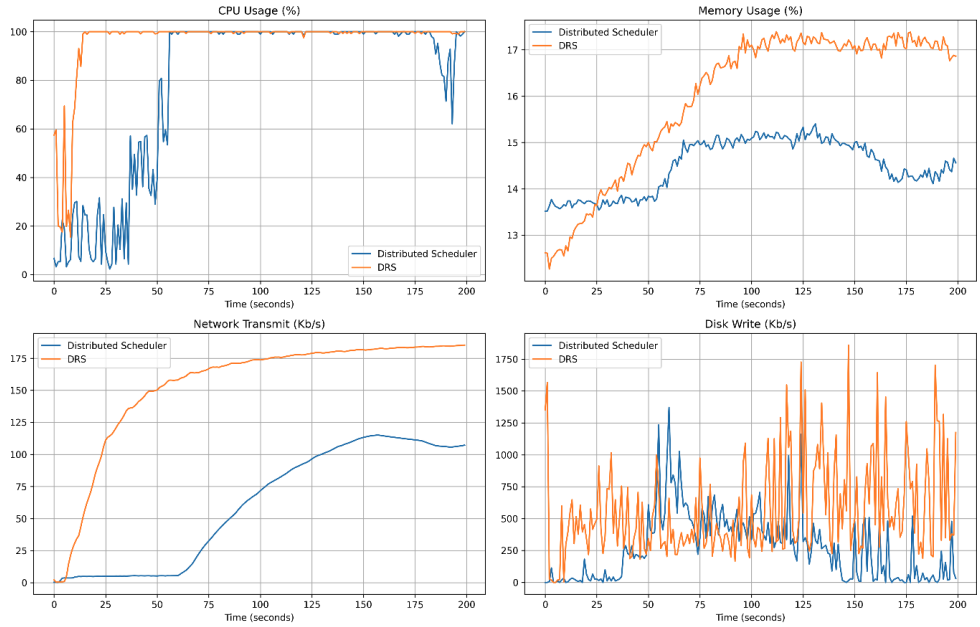

図 5.12: ランダム評価結果

結果によると、環境の CPU 使用率では、安定した負荷の下で分散型スケジューラーと DRS の両方がほぼ同じ 100%に達している。メモリ使用率については、DRS は時間の経過とともに使用率が増加していくのに対し、分散型スケジューラーは約 14%でほとんど変化しない。ネットワーク帯域使用率では、分散型スケジューラーは 150Kb/s に維持しており、DRS は 1751Kb/s に維持している。ディスク I/O においては、DRS の書き込み速度が 0 から 1700Kb/s 広い範囲で大きく変動しているのに対し、分散型スケジューラーは 0 から 500Kb/s 範囲で比較的小さな変動が確認した。また、実験後半でタスク完了によるリソース使用率の減少が確認した。

この結果から、ランダムワークロードにおいて、分散型スケジューラーは DRS と比較してメモリ使用率が安定し、ネットワーク負荷が削減され、ディスク書き込み速度もより小さな範囲で安定していることがわかる。特に、分散型スケジューラーのリソース使用率がタスク完了に応じて適切に低下しており、柔軟で効率的なリソース管理が可能であることを確認できた。

5.4 考察

本研究では、MARL モデルを用いた分散スケジューリングを実現し、各スケジューラーが自律的に最適なスケジューリング決定を行う仕組みにより、リソース利用効率の向上と、より柔軟かつ適応的なワークロード管理の実現を目的としている。評価により、提案手法で MARL を用いた分散型スケジューラーを実現し、ノード障害時の自動切り替え機能を検証し、スケジューラの単一障害点の問題を解決できたことを確認できた。リソース利用率については、メモリ使用率の安定化、ネットワーク負荷の削減、およびディスク I/O の効率的な制御が実現されたことを確認できた。しかし、実験時間が短かったため、長期運用時のアルゴリズムの性能と安定性を総合的に評価することができなかった。また、分散型スケジューラーの自動デプロイメント機能が未実装であり、手動でのデプロイメントが必要となっている。そのため、アルゴリズムの収束性能とシステムの長期安定性、特に様々な複雑なワークロードに対する性能を検証するための長期実験の実施と、自動デプロイメントメカニズムの実装が必要である。

第6章　おわりに

本章では、本研究における総括と今後の展望や実験を通じて明らかとなった課題について述べる。

6.1　本研究の総括

本研究では、Kubernetes 環境で MARL 用いて分散型のスケジューリングモデルについて検討した。具体的には、各ノードに配置された自律的なスケジューラーが強化学習を通じてスケジューリング戦略を最適化する分散型アーキテクチャを採用した。動作確認より、スケジューラーの単一障害点の問題に対し、ノードの追加や障害発生時においても、クラスター全体の安定性とスケーラビリティが維持されることが確認できた。リソースの使用率について、メモリ使用率の安定化、ネットワーク負荷の大幅な削減、およびディスク I/O の効率的な制御において、先行研究と比較して改善が実証された。また、タスクの完了に応じた適切なリソース解放など、動的な負荷変動に対する柔軟な対応能力も確認された。

6.2　今後の展望

6.2.1　実現規模

本研究では、ワーカーノード 4 台とマスターノード 1 台で構成される Kubernetes クラスター環境で提案手法の評価を行った。しかしながら、実際の運用環境における Kubernetes クラスター環境は数十から数百台規模のノードで構成されることが一般的である。そのため、実用的な規模のクラスター環境でも効果的に動作するかどうか、スケーラビリティと環境のリソース使用率などについて評価を行う必要がある。

6.2.2　スケジューラー実装における課題

本研究では、提案するスケジューラーの実装は Python 言語を使用した。現在の Python 実装では、Kubernetes クラスター環境に DaemonSet としてデプロイした場合、単一のスケジューラーしか新規ポッドの情報を受信できないという問題で

ある。そのため、全てのスケジューラーを各ノードのサービスとして実行する必要があり、これによって初めて各スケジューラーが watch 機能を通じて新規ポッドの情報を取得することが可能となる。そのため、Informer 機能を備え Go 言語への移行を考えられる。Go 言語を使うと、提案スケジューラーが DaemonSet としてデプロイでき、各ノードに一つのポッドのインスタンスが存在することが保証できる。

6.2.3 学習アルゴリズムのパフォーマンス

本研究では、分散型スケジューラーの学習アルゴリズムとして DQN を採用した。評価結果から、メモリ使用率、ネットワーク帯域使用率とディスク I/O の書き込み速度において効率的なリソース管理が確認できた。また、DQN アルゴリズムは環境が複雑化するにつれて学習速度が低下し、収束に時間がかかる可能性がある。この課題に対処するためには、より少ない計算リソースで高精度のスケジューリングが可能なアルゴリズムやマルチエージェント環境向けのアルゴリズムを導入する必要があると考える。

謝辞

本研究を進めるにあたり、多大なるご指導とご支援を賜りました主指導教員である宇多 仁准教授に、深く感謝いたします。研究テーマの選定から結果の考察に至るまで、ご指導をいただき、本研究を完成させることができました。また、副指導教員としてご協力いただいた篠田 陽一教授にも深く感謝申し上げます。研究における具体的な課題に対して有益なアドバイスをいただき、研究の方向性を明確にする上で大変助けとなりました。さらに、インターンシップ指導教授であるベウラン ラズバン准教授には、中間発表や修士論文審査会において貴重なご助言をいただきましたことに感謝申し上げます。

また、本研究室の博士前期課程の中川颯馬 氏、高橋亮真 氏、吉川健太 氏、高田敦生 氏、金田昂大 氏、中村一貴 氏、西村優典 氏、Lu Xingchen 氏には、研究計画提案書や論文の添削にご支援をいただいただけでなく、研究に関する多角的なご意見を共有していただき、さらには研究生活全般において多大なるご助力を賜りましたことに深く感謝いたします。

最後に、これまでの学生生活を支えてくださった家族と友人たちに、心より感謝申し上げます。修士論文の提出に至るまで皆様に多大なご支援にいただき、ありがとうございました。

参考文献

[1] Microsoft. The 2022 State of the Cloud Report, (2022)

[2] Kubernetes. "Kubernetes Documentation". https://kubernetes.io.(2022)

[3] Jian, Z., Xie, X., Fang, Y., Jiang, Y., Lu, Y., Dash, A., ... Wang, G. DRS: A deep reinforcement learning enhanced Kubernetes scheduler for microservice - based system. Software: Practice and Experience(2023)

[4] Kubernetes. "Kubernetes Architecture". https://kubernetes.io/docs/concepts /architecture/ (2024)

[5] ALBRECHT, Stefano V.; CHRISTIANOS, Filippos; SCHÄFER, Lukas. Multi-agent reinforcement learning: Foundations and modern approaches. MIT Press, (2024)

[6] Lorido-Botran, T., Bhatti, M. K. Adaptive container scheduling in cloud data centers: a deep reinforcement learning approach. In International conference on advanced information networking and applications (pp. 572-581). Cham: Springer International Publishing.(2021)

[7] Diouf, G. M., Elbiaze, H., Jaafar, W. On Byzantine fault tolerance in multi-master Kubernetes clusters. Future Generation Computer Systems, 109, 407-419.(2020)

[8] ESPINOSA, Alejandro, et al. Optimizing Energy Consumption of Kubernetes Clusters with Deep Reinforcement Learning. In: Artificial Intelligence Research and Development. IOS Press(pp.26-35).(2024)

[9] Kattepur, A., David, S. Madelyn: multi-domain multi-agent reinforcement learning for data-center networks. In 2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC) (pp. 624-629). IEEE. (2022)

[10] AI 研究所.「Deep Q-Network（DQN）とは?仕組みや強化学習との関係を解説」『AI 研究所』.https://ai-kenkyujo.com/programming/kyoukagakusyu/deep-q-network-toha/

[11] SEWAK, Mohit. Deep reinforcement learning. Singapore: Springer Singapore.(2019)

[12] Yagami, Y. ”【強化学習】Deep Q-Learning (DQN) の理論と実装.”. https://yagami12.hatenablog.com/entry/2019/02/22/210608.(2019)

[13] 森村哲郎.MLP 機械学習プロフェッショナルシリーズ，講談社（2019）