Kubernetes PodのRTTを用いたリーダー候補の順位付けによる再選出時間の短縮

富田 啓太[1]　串田 高幸[1]

概要：Kubernetes のステートフルな Pod は運用する際に永続的なストレージを必要としているため, 独立したデータストアを持っている．また高いスケーラビリティを得るために Pod を複製できるが, ステートフルな Pod の場合アプリケーション Pod が使用するデータの整合性の維持が必要となる．整合性の維持としてリーダー選出アルゴリズムによって選出されたリーダー Pod が書き込み処理を行う手法がある．リーダー Pod は, 複数の Pod の中から選出された Pod である．すべての Pod はユーザーからの読み込みリクエストは受け取るが, 書き込みリクエストはリーダー Pod が受け取る．しかし, 多くの書き込みリクエストが受け取った場合リーダー Pod にアクセスが集中し, リーダー Pod があるノードの CPU 使用率が上がる．その場合リーダー Pod の処理が追いつかなくなり, フォロワーである Pod との整合性の維持が間に合わなくなる．またリーダーの維持方法であるリーダーレコード更新も間に合わずリーダー再選出が起きる．その間書き込みを行えるリーダー Pod がいないためサービスが停止する．本稿では事前に順位付けをすることでリーダー Pod の再選出時間を短縮するアルゴリズムを提案する．評価として提案の手法とデフォルトのリーダー選出アルゴリズムとの再選出時間の比較を行う．実験した結果, 選出時間は変わらない結果となった．

1. はじめに

背景

仮想化技術はクラウドコンピューティングを扱う上で重要な技術となっている．特に, コンテナベースの仮想化は軽量な手法である．ハイパーバイザー型仮想化に比べて, OS を必要とせず高速に動作することがメリットである [1]．しかし, 簡易的にコンテナを構築できるがゆえに, コンテナの数が増えてしまうと管理が難しくなってしまうデメリットが存在する．この問題を解決するために開発された技術がコンテナオーケストレーションシステムである．コンテナオーケストレーションシステムはコンテナを管理, 配置, 設定をおこなう概念であり, デメリットであったコンテナを管理することができる．代表的なコンテナオーケストレーションシステムとして Kubernetes が存在する [2]．Google が 2013 年に発表した Borg が基となっており, 2014 年にオープンソースプロジェクトとして発表され, 現在も開発が進められている．[3]

リーダー選出アルゴリズムによる解決

アプリケーションコンテナを Kubernetes にデプロイする際に, Kubernetes では Pod という単位で実行される．Pod は単一のアプリケーションとしても実行されるが, 同一の Pod を複製したレプリケーションにも対応しており, ReplicaSet というコントローラーで管理されている．Kubernetes クラスタで実行されている Pod は一般的にステートレス, ステートフルに分類される．ステートレスな Pod とは永続的なストレージを持たないのに対し, ステートフルな Pod は永続的なストレージを必要とする．そのため, ステートフルなレプリカの Pod はそれぞれ独立したデータストアを保持している．

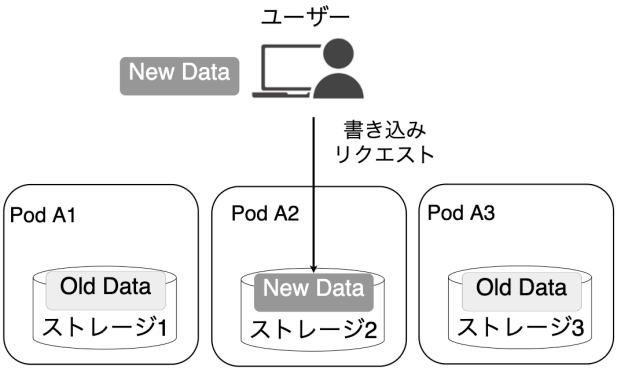

図 1　レプリカ間のストレージの問題点

図 1 では, ステートフルアプリケーションがデプロイ

[1] 東京工科大学コンピュータサイエンス学部〒 192–0982 東京都八王子市片倉町 1404-1

された同種の 3 つ Pod があり, Pod A1, Pod A2, Pod A3 と表している．New Data はストレージに新たに書き込むデータ, Old Data は New Data を書き込む前のデータを表している．ユーザーはアプリケーション Pod に対してストレージに New Data を書き込むための利用者を表している．ユーザーが Pod A2 のストレージ 2 へ書き込むと, Pod A1 のストレージ 1 と Pod A3 のストレージ 3 に New Data が存在せず整合性が保たれない問題がある．この問題を解決するのがリーダー選出アルゴリズムによる解決である．リーダー Pod を選出することにより, リーダー Pod がレプリカ間のデータストアとの整合性の維持を行う．これにより, レプリカ間のデータの不整合が起きることがなく, サービスを運用することが可能である．

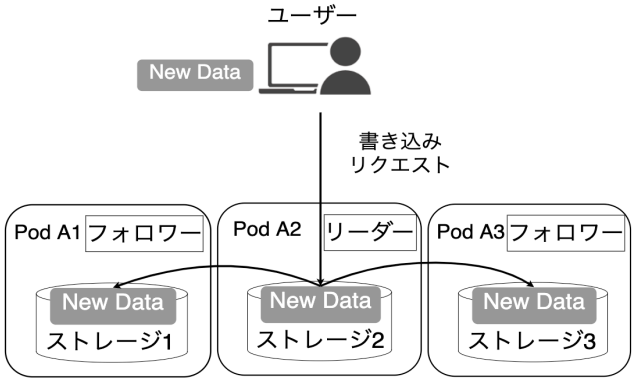

図 2　リーダー選出による整合性の保持

図 2 ではリーダー選出によるレプリカ間のストレージの書き込み処理を示している．図 2 は図 1 と同様に 3 つのステートフルアプリケーション Pod を表しており, リーダー選出によって Pod A2 がリーダー, Pod A1, Pod A3 がフォロワーとなっている．ユーザーは図 1 と同様にアプリケーションの利用者を表しており, Pod に対して New Data の書き込み処理が行われている．図 1 と違う点として, リーダー Pod がストレージへの New Data の書き込み処理を行う点である．データの読み込みリクエストはリーダー Pod 関係なくレプリカ Pod がすべて対応するが, 書き込み処理はすべてリーダー Pod が処理を行う．リーダー Pod の処理が終わった後にリーダー Pod がレプリカ Pod に対しての書き込みを行う．これをアトミックブロードキャストと呼ばれている．アトミックブロードキャストを行うことにより, リーダー Pod のデータストアに書き込んだ内容をレプリカのデータストアに書き込む [4]．これによりレプリカ間のデータストアとの整合性を維持することができる．

リーダー選出アルゴリズム

リーダー選出アルゴリズムとは, 複数に分散されているプロセスの中からまとめ役となるリーダーを選出し, 特別な権限を与える分散アルゴリズムである．リーダーを選出することで, 書き込みをリーダーのみに絞ることができる．それにより, 図 2 のようなレプリカ間の整合性を保つことができる．それ以外のフォロワーとの コンセンサスアルゴリズムが重要な役割を果たしている．Paxos, Raft, Zookeeper は, 実システムで利用されている代表的なコンセンサスアルゴリズムである [5–7]．複製されたデータストアの整合性を確保するために, トランザクションに関する合意を必要としている．

Kubernetes でのリーダー選出アルゴリズムの実装

Kubernetes でリーダー選出アルゴリズムの実装方法として既存のリーダー選出アルゴリズムを実装するか, 自身で新たなアルゴリズムを実装する必要がある．しかし, Kubernetes では, リーダー選出の導入コストを抑えつつ, 簡易的にリーダー選出アルゴリズムを実装できるようにコンテナイメージが用意されており, そのイメージを用いてリーダー選出アルゴリズムを実装することができる[*1]．用意されているリーダー選出アルゴリズムのイメージをデフォルトのリーダー選出と記述する．リーダー選出アルゴリズムの実装として, サイドカーを用いることで, 既存のアプリケーションに対してリーダー選出することができる．図 3 がサイドカーを用いたリーダー選出アルゴリズムの実装の概要図を示している [8]．Kubernetes で用意されているリーダー選出ではリーダー選出に関する情報を保持している Kubernetes のエンドポイントオブジェクト (以下 EP) に基づいて実装されてる．EP とは Pod の名前やリーダー選出が発生する Pod 間のネットワークアドレス, リーダー選出に必要なパラメータを保持しているオブジェクトである．

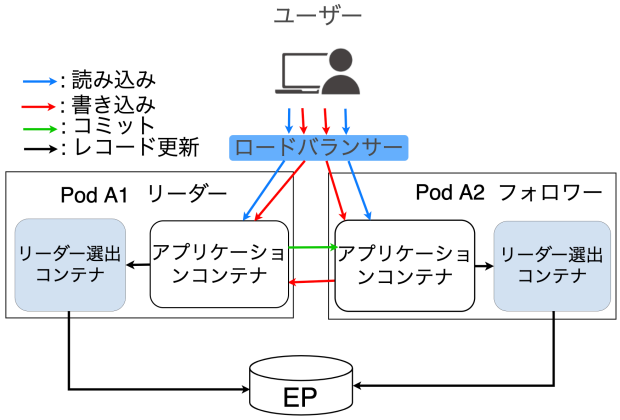

図 3　サイドカーを用いたリーダー選出

図 3 ではアプリケーションコンテナとリーダー選出コンテナで構成されたリーダーである Pod A1, フォロワーである Pod A2 がある．ユーザーはアプリケーションに書き

[*1] https://kubernetes.io/blog/2016/01/simple-leader-election-with-kubernetes/

込みや読み込みを行う利用者を表している．ロードバランサーはユーザーがアプリケーション Pod に外部からアクセスするための機能や ユーザーのリクエストを分散する役割を持つ．リーダーに選出されている Pod は書き込みリクエストを処理し, 読み込みリクエストはリーダー Pod 関係なく処理を行う．アプリケーションコンテナと横にあるリーダー選出コンテナで実装されており, アプリケーションコンテナはクライアントからのリクエスト処理する役割を果たしている．リーダー選出コンテナではアプリケーションコンテナ間のリーダー選出プロセスを実行する．

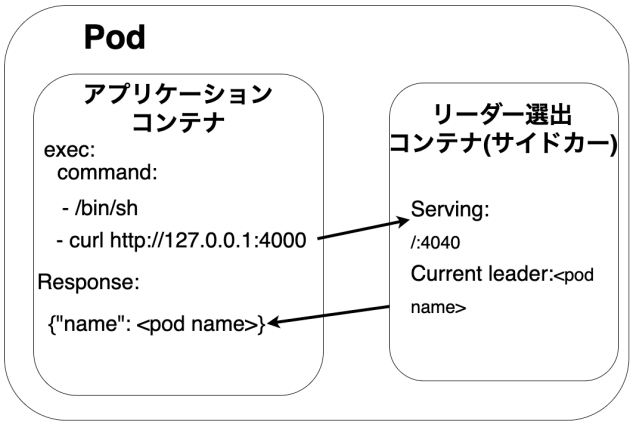

図 4　サイドカーでのリーダー選出の動き

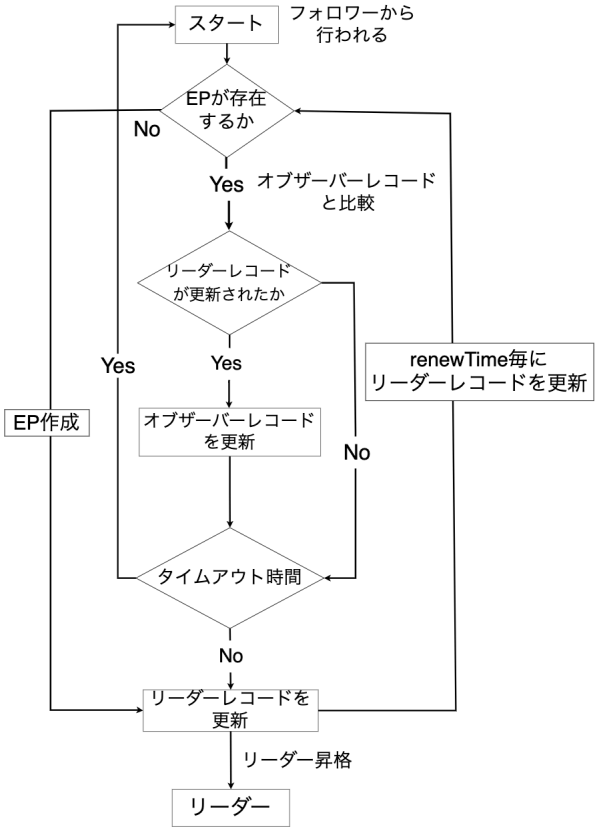

図 5　デフォルトのリーダー選出のフローチャート

図 4 の Pod はアプリケーションコンテナとリーダー選出コンテナから構成されている．アプリケーションコンテナはループバックアドレスにリーダー選出コンテナにアクセスし, 現在のリーダーである Pod の名前を検索している．リーダー選出コンテナは現在のリーダー Pod の名前を含む JSON オブジェクトを返す Web サーバーを提供している．アプリケーションコンテナは指定されたアドレスを検索するだけで, リーダー選出コンテナから選出されたリーダー Pod の名前を得る事ができる．

デフォルトのリーダー選出アルゴリズムのプロセス

EP ではリーダー Pod の情報であるリーダーレコードを定期的に記録することで, リーダーの位置を登録している．以下がリーダーの記録に関する特に重要なパラメータがある．

- holderIdentity
- leaseDurationSeconds
- renewTime

holderIdentity とはリーダーの名前, leaseDurationSeconds はフォロワーが強制的にリーダーの座を獲得するまでのタイムアウト時間, renewTime はリーダーが EP のリーダーレコードを更新する時間である．以上のパラメータを用いたリーダー選出アルゴリズムのフローチャートが図 5 である．

図 5 ではデフォルトのリーダー選出ではすべてはフォロワーからスタートし, EP がなかったらリーダーレコードを書き込み, リーダーに選出される．その後, リーダーは renewTime 毎にリーダーレコードを更新し, リーダー Pod であることを証明し続ける必要がある．フォロワーは常にリーダーがリーダーレコードを更新しているかを監視している．リーダーレコードが更新されなかったら, 猶予として leaseDurationSeconds の時間までリーダーレコードが更新されているかの監視を続ける．そこまでリーダーレコードが更新されなかったら, フォロワーは新たなリーダーを選出する．

課題

リーダー選出アルゴリズムで選出されたリーダー Pod には多くの機能が与えられ, フォロワーであるレプリカの管理を行う．しかし多くのリクエストがリーダー Pod に集中してしまう問題がある．図 6 が概要の図を示している．

図 6 はアプリケーションがデプロイされた 3 つの Pod があり, Pod A1, Pod A2, Pod A3 を表し, Pod A2 はリーダー, Pod A1, Pod A3 はフォロワーである．ユーザーはアプリケーションに書き込みリクエストを送る利用者を表している．ロードバランサーは Pod に外部からアクセスするための機能を表し, リクエストを分散する役割を持つ．ユーザーが Pod に対して書き込みリクエストをロードバラン

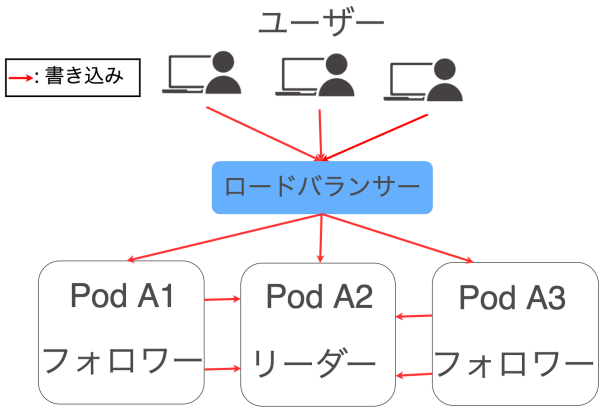

図 6 書き込みリクエストがリーダー Pod に集中する問題

サーで 3 つの Pod に分散しているが, リーダー Pod に対して書き込みリクエストが集中している. Kubernetes ではステートフル Pod との整合性を保つためにリーダーを選出するリーダー選出アルゴリズムが用意されている. これにより, データ更新要求をリーダー Pod が受け付け, フォロワーである Pod に対してブロードキャストを行うことで, レプリカが持つデータストアとの整合性を保っている. しかし, 書き込みリクエストが増加するとリーダー Pod に対してリクエストが集中してしまい, リーダー Pod があるノードの CPU 使用率が上昇してしまう [9]. CPU 使用率が高い場合, フォロワーである Pod との整合性の維持が間に合わなくなる. さらにリーダーレコード更新も遅れてしまうため, リーダーの交代が発生する. その場合, 残りの Pod でリーダーを再選出するためにリーダー選出アルゴリズムが行われる. しかしリーダー選出に時間がかかってしまう場合, その間のデータの更新要求を受け付けるリーダーがいない. データの更新が行えなくなり, サービスのダウンタイムが発生してしまう [9]. リーダー選出に時間をかける ことなく, リーダーの再選出時間を短くする必要がある.

2. 関連研究

この章では本研究と関連した関連研究について取り上げていく. Kubernetes を用いたリーダー選出の実装に関する論文として, 複数のノードにリーダー Pod を分散することでリーダー Pod に対するリクエストを分散している研究がある [10]. これはノードの状態を把握していないデフォルトのリーダー選出アルゴリズムを改良したアルゴリズムであり, デフォルトのリーダー選出との CPU 使用率の比較を行っているが, リーダー選出時間が伸びてしまっている問題がある.

Kubernetes のコントローラーでリーダー Pod を管理することで分散データストアのスループットを最大化するリーダー選出アルゴリズムを提案している [11]. この研究では複数オープンソースである OpenDaylight が提供しているリーダー選出アルゴリズムが単一のリーダー Pod に集中してしまい, 単一のリーダー Pod に集中することを課題としている. リーダー Pod を管理するコントローラを用いてノードに配置されている一つのリーダー Pod へのリクエスト最大数を制限し, 複数のノードにリーダー Pod を分散することでデータ更新要求を分散している. 分散データストアのスループットは評価しているが, リーダー Pod 削除後の再選出による評価は行っていない.

3. 提案

事前に Pod の順位付けを行うことで, リーダー Pod 削除後のリーダーを再選出する. 図 7 に提案の概要図を示す.

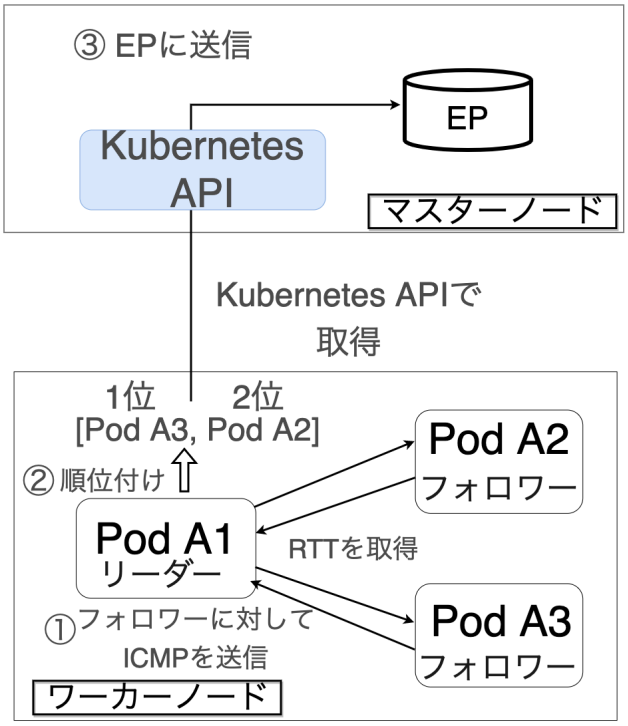

図 7 提案の概要図

図 7 は, ワーカーノード内に 3 つのアプリケーションである Pod をデプロイしており, Pod A1, Pod A2, Pod A3 としている. Pod A1 は最初にリーダーに選出されており, Pod A2, Pod A3 はフォロワーとなっている. リーダー Pod はフォロワーである Pod に対して ICMP を送信し, Round Trip Time(RTT) を取得している. RTT からフォロワーである Pod の順位付けを行い, Kubernetes API にアクセスしてマスターノード内にある EP に保存している. RTT とはパケットを送信して受信した側が送信側に ACK パケットを送り、送信側でそれを受取るまでの時間です。図 7 ではリーダー Pod がフォロワーである Pod に対してパケットを送信し, フォロワーである Pod がリーダー Pod に ACK パケットを送信している. この一連の流れは Internet Control Message Protocol (ICMP) が用いられている. RTT を使用する理由として, フォロワーとの平均

RTT が短いほうがフォロワーが持つデータストアとのコミット完了速度が速いことがわかっているためである [12]. フォロワーの RTT を取得後, フォロワーの Pod の順位付けを行う. その後, EP オブジェクトにデータを保存する.

はじめに選出されたリーダーはフォロワーである Pod に対して ICMP を送信し, RTT によって順位付けを行う. また, 図 8 が提案のリーダー選出方法である.

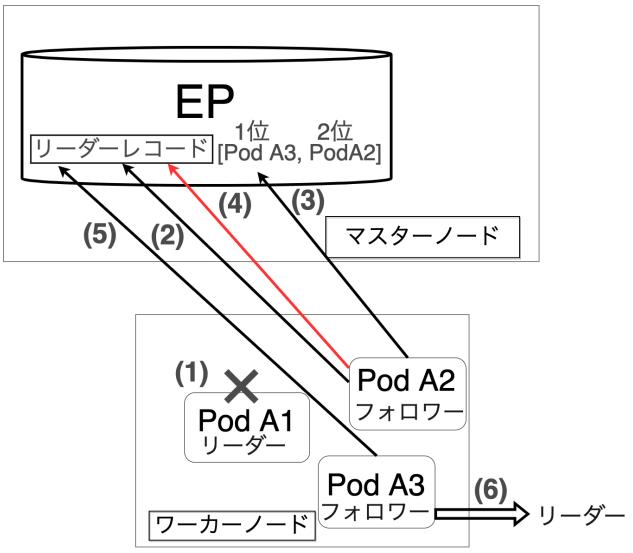

図 8 提案のリーダー選出方法

図 8 はワーカーノードにアプリケーションがデプロイされた Pod A が 3 つあり, Pod A1, Pod A2, Pod A3 と表している. マスターノードにある EP はリーダー Pod が更新するリーダーレコード, 順位付けされたフォロワーである Pod のリストがある. 以下が図 8 の処理の流れである.

(1) リーダー Pod が削除される
(2) フォロワーが EP にあるリーダーレコードが更新されているかを確認. 更新されていなかった場合, (3) に進む, 更新されていた場合, リーダー Pod は変わらない
(3) EP にある順位付けされた Pod リストの先頭の Pod を取得
(4) 取得した Pod 情報をリーダーレコードを更新
(5) リーダーレコードを確認
(6) リーダーに昇格

提案でもデフォルトのリーダー選出同様, リーダーがリーダーレコードを更新しているかどうかを確認する. リーダーレコードが更新されていなかったら事前に順位付けをおこなったリストを EP にアクセスして取得する. 1 位の Pod 情報をリーダーレコードを更新し, 1 位の Pod をリーダーに昇格させる.

ユースケース・シナリオ

図 9 に, リーダー選出アルゴリズムを用いた分散データベースクラスタのユースケースを示す.

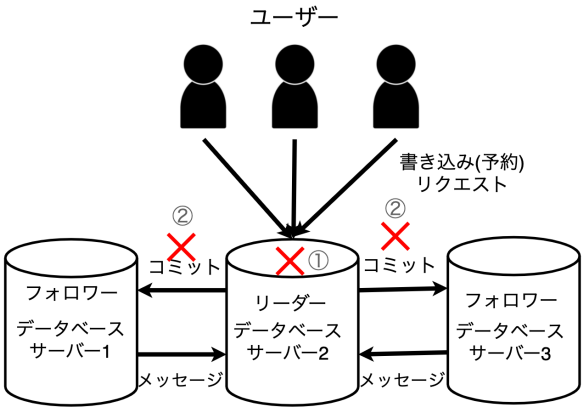

図 9 リーダー選出による分散データベースの書き込み負荷

図 9 では, デプロイされた 3 つのデータベースサーバーがあり, リーダー選出によってリーダーに選出されたデータベースサーバー 2, フォロワーがデータベースサーバー 1, データベースサーバー 3 である. ユーザーは予約を行うためのシステム利用者である. リーダーであるデータベースシステムに予約リクエストである書き込みリクエストが集中している. 書き込みリクエストによってリーダーにアクセス負荷が生じ, サーバーがダウンする. その場合, 書き込みリクエストを処理するリーダーがいなくなるため, その間のシステムのダウンタイムが発生する. そのためリーダーの再選出時間を短くすることで, リーダーが存在しない時間を短くすることを可能とする.

4. 実装と実験方法

実装

Pod を Python の Kubernetes モジュールで取得し, リーダー選出アルゴリズムの実装を行った. 実装の概要を図 10 に示す.

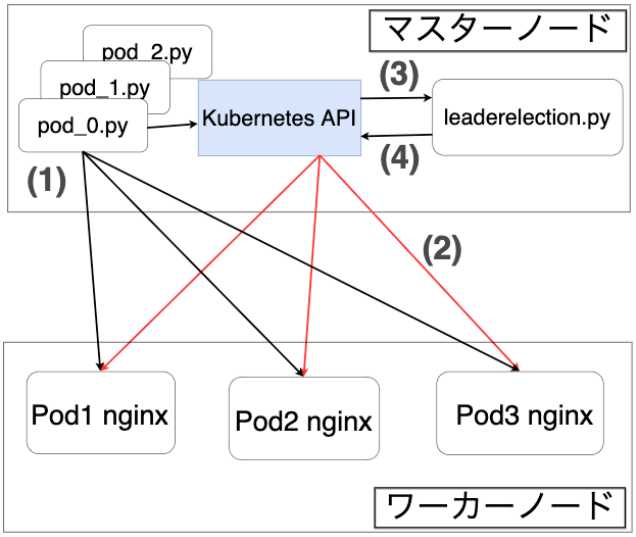

図 10 実装の概要図

図 10 はワーカーノードに nginx をデプロイした 3 つの Pod で構成されており, Pod1 nginx, Pod2 nginx, Pod3 nginx と表している. マスターノードでは提案のリーダー選出を行うための Python ファイルで構成されている. 以下で図 10 の流れを説明する.

(1) default の namespace で稼働している Pod を取得する.
(2) Pod の状態を比較するために, old_pod_list として保存する.
(3) 取得した Pod はフォロワーとして, リーダーを取得するためのリーダー選出を行う.
(4) リーダー選出終了後, 現在の Pod の状態を確認するために新たな Pod リストを取得し, 前に取得した Pod リストとの差分があるかを確認する. 差分がない場合, (3) の処理に戻りリーダーはリーダーレコードを更新し, フォロワーはリーダーの座を獲得するために EP にアクセスする.

leaderelection.py でリーダーに選出された Pod を削除することで再度行われるリーダー再選出の時間を取得し, デフォルトのリーダー選出アルゴリズムと提案のリーダー選出アルゴリズム再選出時間の比較を行う. 常に Pod の差分の確認を行う理由として, 一度 leaderelection.py で行われる処理は実際に Pod を削除しても, 削除されたことが処理に反映されず永遠にリーダーレコードを更新し続けてしまう. それにより, 再選出時間のデータが取れない問題があった. Pod が削除されたことを処理に反映させるために, リーダー選出での処理終了後には新しく取得した Pod リストとその前に取得した Pod リストの差分の比較を行う.

提案である RTT を取得する処理として図 11 を示す.

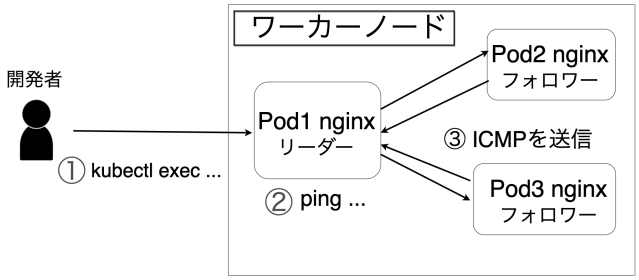

図 11 ping コマンドを使った RTT の取得

図 11 ではワーカーノード内に nginx をデプロイした 3 つの Pod で構成されており, Pod1 nginx をリーダーとし, Pod2 nginx, Pod3 nginx をフォロワーとしている. 開発者がリーダー Pod にログインし, フォロワーである Pod に対して ICMP を送信する ping コマンドを行う. ICMP を送信し, 取得した RTT 値をもとに順位付けを行った Pod リストを作成し, leaderelection.py のインスタンス変数としてリーダー選出を行う.

Pod のレプリカとして 4 つ作成し, 1 つの Pod を削除した際の 3 つの Pod で行うリーダー再選出時間を取得する. そのため, リーダー Pod 削除後に再作成される Pod は考慮に入れないとする.

実験環境

今回の研究では, 研究室内で構築した k8s クラスタを使用する. システムアーキテクチャを図 12 に示す. 図 12 は OS が Ubuntu20.04 の VM で構成されているマスターノードが 1 台, ワーカーノードが 1 台の計 2 台で構成されている. 使用している VM のスペックは以下に記述する.

- vCPU × 2
- RAM 4GB
- HDD 40GB
- RKE2

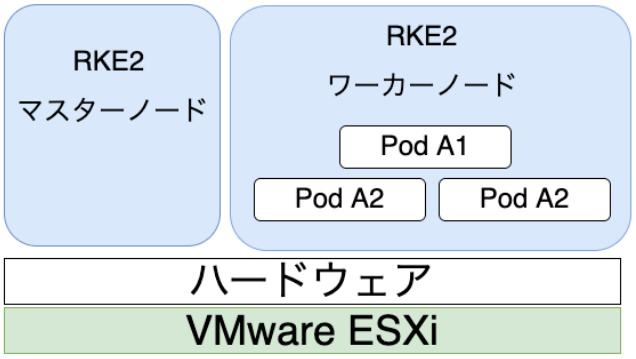

図 12 システムアーキテクチャ

5. 評価

評価項目としては図 5 のリーダー選出と提案のリーダーの選出時間の比較を行う. 図 5 のリーダー選出をデフォルトとする. 結果を図 13 に示す.

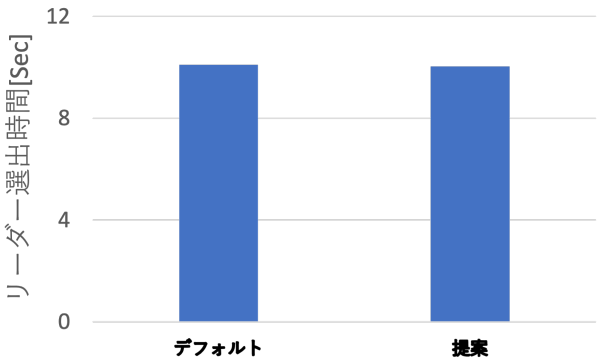

図 13 リーダー選出時間の比較

図 13 は, デフォルトのリーダー選出と提案のリーダー選出の再選出時間の計測を 10 回行い, 平均を表す. X 軸は比較を行うリーダー選出アルゴリズム, Y 軸はリーダーの

再選出時間を表し. 単位に Sec(Seconds) をとる. デフォルトのリーダー選出アルゴリズムの再選出時間は約 10.10s, 提案のアルゴリズムでは約 10.04s となり, 再選出時間が変わらかった. 原因として両方のアルゴリズムでは EP にアクセスする際にリーダーレコードを確認しているためである. リーダーレコードが更新されていない場合にフォロワーはリーダーに選出される. しかしリーダーレコードはリーダー Pod が削除されても即座に削除されず, フォロワーが強制的にリーダー選出を開始する leaseDurationSeconds に時間が達するまでで, リーダーレコードが削除されないことがわかった. 今回の実験でのの設定値である leaseDurationSeconds は 10 と設定したため, 再選出時間も leaseDurationSeconds に近い値となった. したがって, リーダーレコードを即座に削除しない限り, リーダーの交代が起きないことが実験からわかった.

6. 議論

本研究の議論点として, 再選出時間に差がでない実験となった. リーダーレコードでの確認より, Raft のような分散合意でのリーダー選出を用いた順位付けを検討する必要がある.

リーダーに書き込みリクエストが集中する問題を解決できていない. 本研究の提案では再選出時間短縮を目的としていたため, N.Nguyen ら研究より, 書き込みリクエストはリーダーに集中することが前提とした実験を行った. [9] リーダーに対して書き込みリクエストの負荷が増えることはリーダーが単一障害点となってしまう. その場合, リーダーがボトルネックとなる可能性があるため, 実際のサービスで使うものとして望ましくない. そのため, リーダーの負荷を分散するためにも, 複数のリーダーの配置を検討する必要がある.

もう一つの議論点として, 想定せずに複数のリーダーが存在してしまう問題がある. リーダーの生存の確認方法として, リーダーがフォロワーに対してハートビートを送ることで, 生存を確認する. その際に ハートビートを送る間隔が重要となってくる. リーダーがハートビートが送信されなかったらフォロワー間でリーダー選出が行われるが, 実際にはリーダー Pod が削除されずに生きている可能性が存在する. その場合, リーダーが複数存在することになってしまうことにより, リーダー間でデータの不整合が起こる. そのため, リーダーへの監視を行う必要がある.

7. おわりに

本研究の課題はリーダー Pod に大量の書き込みリクエストが集中し, リーダー Pod があるノードの CPU 使用率上昇し, 利用できる CPU 値節約のためにリーダー Pod が削除される リーダー Pod 削除後のリーダーの不在によるサービスのダウンタイムが発生するため, リーダーの再選出時間の短縮する必要があることである. 提案として RTT を用いてフォロワーである Pod の順位付けを行うことで再選出時間の短縮を行うことである. デフォルトのリーダー選出アルゴリズムと提案のリーダー選出アルゴリズムの再選出時間の比較を行った結果, デフォルトのリーダー選出時間と提案のリーダー選出時間は変わらない. 今後の課題としてリーダーの確認をリーダーレコードの更新の確認を行うのは, 実システムで使う際には非効率であるため, 分散合意を用いたリーダー選出方法を検討する必要がある.

参考文献

[1] Radchenko, G. I., Alaasam, A. B. A. and Tchernykh, A. N.: Comparative Analysis of Virtualization Methods in Big Data Processing, *Supercomput. Front. Innov.: Int. J.*, Vol. 6, No. 1, p. 48–79 (online), DOI: 10.14529/jsfi190107 (2019).

[2] El Haj Ahmed, G., Gil-Castiñeira, F. and Costa-Montenegro, E.: KubCG: A dynamic Kubernetes scheduler for heterogeneous clusters, *Software: Practice and Experience*, Vol. 51, No. 2, pp. 213–234 (2021).

[3] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E. and Wilkes, J.: Large-Scale Cluster Management at Google with Borg, *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/2741948.2741964 (2015).

[4] Défago, X., Schiper, A. and Urbán, P.: Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey, *ACM Comput. Surv.*, Vol. 36, No. 4, p. 372–421 (online), DOI: 10.1145/1041680.1041682 (2004).

[5] Lamport, L.: The Part-Time Parliament, *ACM Trans. Comput. Syst.*, Vol. 16, No. 2, p. 133–169 (online), DOI: 10.1145/279227.279229 (1998).

[6] Ongaro, D. and Ousterhout, J.: In Search of an Understandable Consensus Algorithm, *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, USENIX Association, pp. 305–319 (online), available from ⟨https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro⟩ (2014).

[7] Hunt, P., Konar, M., Junqueira, F. P. and Reed, B.: ZooKeeper: Wait-free Coordination for Internet-scale Systems, *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, USENIX Association, (online), available from ⟨https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems⟩ (2010).

[8] Burns, B. and Oppenheimer, D.: Design Patterns for Container-based Distributed Systems, *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, USENIX Association, (online), available from ⟨https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns⟩ (2016).

[9] Nguyen, N. and Kim, T.: Toward Highly Scalable Load Balancing in Kubernetes Clusters, *IEEE Communications Magazine*, Vol. 58, No. 7, pp. 78–83 (online), DOI: 10.1109/MCOM.001.1900660 (2020).

[10] Nguyen, N. D. and Kim, T.: Balanced Leader Distribution Algorithm in Kubernetes Clusters, *Sensors*, Vol. 21,

No. 3 (online), DOI: 10.3390/s21030869 (2021).

[11] Kim, T., Myung, J. and Yoo, S.-E.: Load Balancing of Distributed Datastore in OpenDaylight Controller Cluster, *IEEE Transactions on Network and Service Management*, Vol. 16, No. 1, pp. 72–83 (online), DOI: 10.1109/TNSM.2019.2891592 (2019).

[12] Choumas, K. and Korakis, T.: When Raft Meets SDN: How to Elect a Leader over a Network, *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, IEEE, pp. 140–144 (2020).

