分散コンピューティング基盤における宣言的構成管理の適用可能性と論点

真壁徹 [1, a)]　篠田 陽一 [1]

概要：IT 基盤の複雑化が，専門性を持つ技術者への依存を高めている．そこで手続きを逐一指示せずとも，あるべき姿を定義すれば基盤をその通りに設定，維持できる宣言的構成管理が注目されている．Kubernetes は宣言的構成管理が従来抱えていた課題を解決し，実用化した．STPA(System-Theoretic Accident Model and Processes)によりその構造を分析し，宣言的構成管理が広く分散コンピューティング基盤に適用できるコンセプトとなるか，その論点を導く．

キーワード：分散コンピューティング，宣言的構成管理，安全性分析，STPA，Kubernetes，コンテナ

Applicability and discussions of declarative configuration management in distributed computing infrastructure

TORU MAKABE [1, a)]　YOICHI SHINODA[1]

1. はじめに

インターネットは社会とビジネスの変化スピードを高めた. そして現在, COVID-19 が変化に拍車をかけている．基盤を構成するリソースは増加，分散し，その運用は専門性を持つ技術者へ依存している．

そこで提供するサービスのあるべき姿を宣言すれば，手順を明示せずとも基盤を構成，維持できる宣言的構成管理が注目されている．しかし，これまで Ansible[1]や Terraform[2]など宣言的な記法が可能なツールは存在したものの，計画的な作業支援に止まっていた．求められるのは, 故障など計画外変化においても機能する仕組みである．なぜなら大規模基盤では常にどこかが故障している. また，基盤は手作業や時間経過を通じ，計画時に意図した構成から離れていく．そこで宣言と状態を分離できれば，運用者は状態に追従する負担から解放される．

Kubernetes[3]は宣言的構成管理の実現を阻む課題をコンテナの特徴を活かし，結果整合を認めることで解決した．特に構成管理と回復機能を一体にした構造は特徴的である．

反面，普及と同時に課題も散見され，コミュニティサイトではサービス停止やアクシデント事例が数多く共有されている[4]．宣言的構成管理というアイデアが普及する過程で明らかになった課題は示唆に富む．

Kubernetes の回復性については，コンテナの回復時間に注目した先行研究[5]があるが，構造上の特徴は追究されていない。そこで本論文は Kubernetes の安全性，回復性の分析を通じ, 一体である構成管理の構造上の課題を考察する．本稿は Kubernetes に止まらず，宣言的構成管理を他の分散コンピューティング基盤へ適用する際に，検討や判断の助けになることを目的とする．

2. 背景

2.1 宣言的構成管理の普及を阻む要因

宣言的構成管理はネットワークやクラウドコンピューティング領域で先行研究があり[6][7][8]，その基礎には Prolog など宣言型の論理プログラミング，制約プログラミングがある．しかしこのコンセプトが一般化したとは言い難く，サービス提供中の基盤へ宣言，制約を継続的に適用するには至っていない．以下に理由を考察する．

管理対象が持つ自律性との齟齬　構成管理機能と管理対象自身が持つ構成維持，回復機能が衝突する，もしくは過敏に反応することがある．また，構成管理機能ではトランザクション処理が可能であっても，管理対象がその境界外で非同期に動作するケースもある．例えば，ルーティングプロトコルやルータが故障や輻輳を検知し経路を切り替え, 収束を待つ間に, 構成管理機能から変更を指示すると，順序やタイミングによっては一時的なループや経路振動に繋がる恐れがある[6].

リソース操作に要する時間　アプリケーション（以下，アプリ）とは異なり，基盤を構成するリソースの中には状態の変更に時間を要するものがある．例えばクラウドコンピューティングサービスで仮想マシンの作成には一般的に

1 北陸先端科学技術大学院大学
Japan Advanced Institute of Science and Technology,
Asahidai, Nomi, Ishikawa 923-1292 Japan
a) tomakabe@jaist.ac.jp

数十秒を要する[9]．仮想マシンがクラッシュした場合には宣言した数を維持するため再作成が必要だが，その間の数十秒は制約に反することになる．そのため予備リソースを常時待機させる解決策もあるが，コストと複雑度は増す．

宣言や制約の記述に必要な専門性　先行研究では宣言や制約を表現するために Prolog を元にした COPElog[7]や Network Datalog[8]が提案されている．論理，制約プログラミングの習得には専門性を必要とする．反面，宣言的な構成管理の目的の 1 つは，専門性を持つ技術者への依存を解決することである．

加えて基盤変更のリスクが大きいことも，一般化を阻む要因であろう．例えばクラウドコンピューティングサービスの主な停止原因は変更作業である[10]．多くのユーザが共用し，停止の影響が大きな基盤では変更作業を慎重に行わざるを得ない．

2.2 Kubernetes の解決アプローチ

しかし昨今，宣言的な構成管理を行う分散コンピューティング基盤として Kubernetes が注目され，導入例が増えている[11].

Kubernetes の主な管理対象はコンテナであり，複数のコンテナを包含する Pod を管理の単位とする．そして利用者は Pod と関連リソースの仕様を宣言し，Kubernetes はそれを作成，維持する．なお Kubernetes クラスタの土台となるサーバ，ネットワーク，ストレージなどインフラストラクチャの管理は Kubernetes から分離される．インフラストラクチャが提供する API を通じて操作は可能であるが，拡張機能であり選択は任意である．

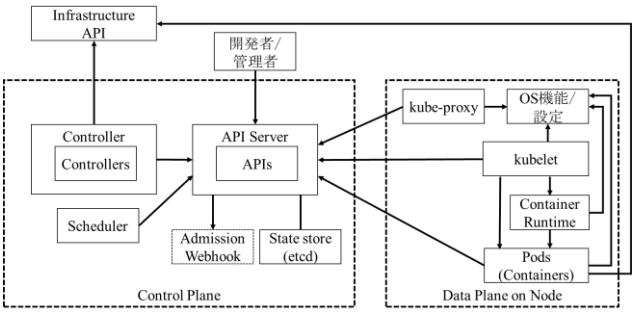

図 1 Kubernetes の構成要素

図 1 が Kubernetes を構成する要素の概観である．API Server と状態ストア(etcd)が，Kubernetes リソースの宣言と状態を一元管理する．そしてコントローラ群(Controller Manager)と Pod の配置先 Node を決定する Scheduler が，API Server を通じて構成情報を操作する．

また, Node はベアメタルサーバや仮想マシンで構成され，Node 上の kubelet が Pod の定義に従い，コンテナランタイムを通じてコンテナを作成する．kube-proxy は Node のネットワーク設定やパケット転送を行う．

これらのコンポーネントが常に宣言と状態の差分を確認し，差分があれば埋めるようループする(Reconciliation loop)．なお操作の方向は API Server からのプッシュではなく，各コンポーネントからのプルである．つまり API Server へ問い合わせ，構成情報を更新するのは各コンポーネントの責務である．API Server は反応的で，各コンポーネントへデータを提供する順序やタイミングを制御しない．

このような特徴を持つ Kubernetes であるが，支持を得た理由を宣言的構成管理の文脈で考察する．

構成管理と回復機能が一体　Kubernetes において，宣言を満たすリソースの割り当てや設定と，故障や不具合からの回復は同じ機能，ループで実現される．よって構成管理と回復機能の齟齬が生じにくい．

リソース操作に要する時間が短い　主な管理対象はコンテナとそれを包含する Pod であり，イメージのサイズや事前配備など条件を整えれば 1 秒以内で作成できる[12]．また，MicroVM など他の軽量な選択肢も増えている[13]．これらの手段により，構成変更や回復が必要なタイミングで Pod を再作成するというシンプルな戦略が可能となった．

結果整合の許容　Kubernetes は宣言を受け付ける際，プラグイン可能なアドミッションコントロールなど一部の例外を除き，入力内容の妥当性検証を行わない．例えば要求されたリソースに対し割り当て可能なものがなくとも，API Server は拒否せず受け付け，利用可能なリソースが用意されるか空くのを待つ．よって複雑な検証アルゴリズムやその計算にかかる時間を不要とした．

また，ネットワークを介して接続される分散システムでは，管理対象の正確な状態把握は困難である．そこで Kubernetes の各コンポーネントは API Server から得られる状態(Current State)を正とし，仮に実際の状態(Actual State)と異なっていても，一時的な不整合を受け入れ，Reconciliation loop を通じて解決する[14]．各コンポーネントに構成維持機能を委譲することで API Server の肥大化を避け，加えて，一時的なネットワーク分断による制御やフィードバックの失敗に対しての耐性を持たせている．

論理ではなく状態を記述　Kubernetes の利用者は，リソースのあるべき状態を DSL(Domain Specific Language)で宣言する．リソースの属性を理解する必要はあるが，制約やルールなど論理をプログラミングする必要はない．かつ DSL の抽象度は高くなく，Pod に必要なリソースの量や複製の数，起動や配置条件など属性を直接的に記述できる．

3. STPA による安全性分析

Kubernetes の構成要素は多様で，相互に作用する．よって各構成要素が期待通りに動作しても，構成要素の内部状

態や受け渡されるデータによっては，全体として問題が生じる可能性がある．つまり Kubernetes の安全性分析では，構成要素単体の故障や不具合に注目すると特徴を捉えられない．そこで本論文では，事故因果関係モデル STAMP(System-Theoretic Accident Model and Processes)を基礎とするハザード分析手法 STPA(System-Theoretic Process Analysis)[15][16][17]を選択する．

STPA において損失は構成要素の相互作用の結果と考えるため，Kubernetes の安全性分析に適する．また，STPA では分析過程で制御構造図（コントロールストラクチャ）を成果物とするが，構造上の課題の考察に有用である．

3.1 分析目的の定義

STPA の本来の目的は人命，設備，金銭などの損失を防ぐことである．損失の重要度や影響の大きさはシステムの位置づけと利害関係者に依存するため，まず目的を定義する．はじめに受け入れられない損失を定義し，損失に繋がるシステムの状態や条件であるハザードを識別する．本分析では一般的なビジネスアプリを想定し，顧客満足の喪失を受け入れられない損失とする[表 1]．そしてハザードは顧客満足を得て維持するために必要なサービスレベルを満たせていない状態とし，高遅延，タイムアウト，エラー応答の発生とした．具体的な数値は割愛する．

最後に，ハザードを防ぐために満たすべきシステムの条件や動作である安全制約を定義する．本分析は Kubernetes が Pod を安全に動かすために守るべき制約に焦点を絞る．

表 1 分析目的の定義

損失	ハザード	安全制約
顧客満足の喪失	H-1: 高遅延 H-2: タイムアウト H-3: エラー応答	SC-1: Pod が必要とする機能を提供，維持しなければならない (ConfigMap や名前解決など) [H-1, H-2, H-3]
		SC-2: Pod に適切な量の資源を提供, 維持しなければならない(CPU, メモリなど) [H-1, H-2, H-3]
		SC-3: Pod がサービス提供するのに必要なクラスタ内ネットワーク経路を提供，維持しなければならない [H-1, H-2, H-3]

なお分析の対象範囲は Kubernetes クラスタとその上で動作するアプリとする．例えばユーザからクラスタに至るネットワーク経路などは範囲外である．

3.2 コントロールストラクチャの図式化

分析目的の定義に続き，コンポーネント間の制御とフィードバックの流れを可視化するため，コントロールストラクチャを作成する．まずシステム全体を概観し特徴を得る[図 2]．

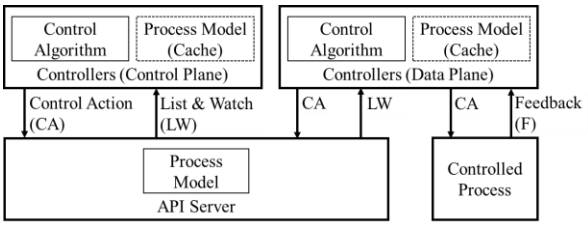

図 2 コントロールストラクチャ(システムレベル)

STPA では被コントロールプロセスの状態など，コントローラが信じていることをプロセスモデルと呼ぶが，Kubernetes では API Server がプロセスモデルを一元的に管理している．

各コントローラはプロセスモデルのキャッシュを持ち，API Server の変化イベントを監視する(List & Watch)．そして宣言と状態に差分が生じた場合に，それぞれの責務に従って処理を行う．なお本分析では構成要素を役割で 2 つのサブシステムに分類し，Node や Pod などアプリが利用するリソースとそれを操作する要素をデータプレーン，その他をコントロールプレーンとする．

プロセスモデルが一元管理され，継続的に問い合わせと更新を繰り返すため，分散システムで課題となりがちな，不適切なプロセスモデルやフィードバックを原因とする問題が起きにくい．仮に更新が一時的に滞ったとしても，いずれ API Server 上のプロセスモデルと同期されるからである．

次に，各コントローラの制御動作であるコントロールアクションを識別するため，サブシステムレベルのコントロールストラクチャを作成する[図 3][図 4]. ただしコンポーネント間の相互作用と構造上の安全性分析が目的であるため，各コンポーネントは冗長化され，単一故障に耐えると仮定する．同様に，各コンポーネントのコントロールアルゴリズムは適切とする．なお図式化によりトレーサビリティがあるため，複数のコントローラの作用の結果生じるハザードは，ハザードの直接的な原因となったコントロールアクションのみを識別する．そして簡単のため，コンポーネント間に複数のコントロールアクションがある場合，1 つの識別子にまとめる．

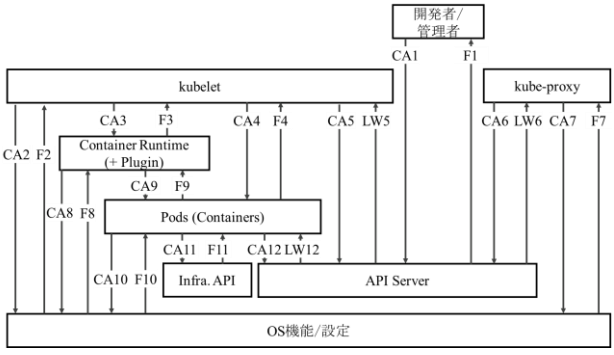

図 3 コントロールストラクチャ(データプレーン)

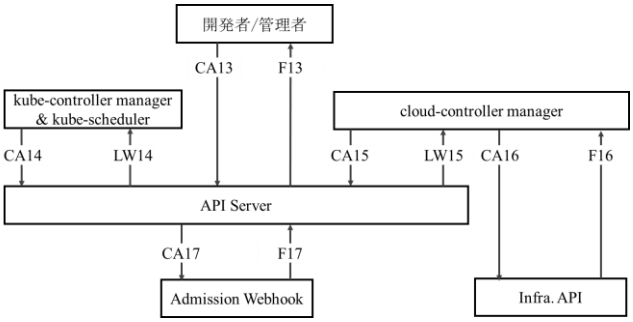

図 4 コントロールストラクチャ(コントロールプレーン)

STPA のコントロールストラクチャでは権限の大きい，制御の起点となるコントローラを図の上部に配置する．そして作成したコントロールストラクチャにおいて，API Server は下部に位置付けられる．従って，API Server は Kubernetes の中核機能でありながら，反応的な被コントロールプロセスであることが確認できる．

3.3 非安全なコントロールアクションの識別

次に，識別したコントロールアクションから，ハザードにつながる非安全なコントロールアクション(UCA: Unsafe Control Action)を識別する．識別には 3 つのガイドワード「与えられないとハザード」「与えられるとハザード」「早すぎ，遅過ぎ，誤順序」を活用する．なお「早すぎる停止，長すぎる適用」も STPA のガイドワードであるが，自動車のブレーキを踏み続けるような連続的なコントロールアクションを対象とするため，離散的な API 呼び出しがコントロールアクションの大半である Kubernetes の分析では適用しない．

識別した非安全なコントロールアクションの一覧は[付録 1]を参照のこと．

3.4 ハザードにつながるシナリオ(要因)の識別

分析の最後に，非安全なコントロールアクションに至る要因，シナリオを識別する．STPA におけるシナリオにはタイプがあり，2 つのグループに大別できる．

非安全なコントロールアクションが起こる要因　このグループにタイプは 4 つあり，「コントローラの故障」「不適切なコントロールアルゴリズムと実装」「非安全なコントロール入力」「不十分または不適切なプロセスモデル/フィードバック」である．

なお本分析では構造に注目するため，「コントローラの故障」と「不適切なコントロールアルゴリズムと実装」を除く．

コントロールアクションが不適切に実行される，または実行されない要因　もう 1 つのグループには 2 つのタイプがあり，「コントロール経路の問題」「被コントロールプロセスの問題」である．

非安全なコントロールアクションをタイプごとに整理し，シナリオを識別した[表 2].

表 2 ハザードシナリオ一覧

タイプ	識別子	シナリオ	UCA
非安全なコントロール入力	HS-1	開発者や管理者が，Pod に適切なリソース要求(Request)と制限(Limit)，優先度設定(Priority Class)，分離や Node 特性に応じた設定(Taint/Toleration，Affinity/Anti Affinity)を行っていない。また，それを検証する仕組みがない	10, 12, 23
	HS-2	開発者や管理者が，(HS-1 に含まれない)Kubernetes コンポーネントの仕様や挙動，制約，状態，サイズを加味した設定や入力，実装を行っていない。また，それを検証する仕組みがない	2, 3, 4, 6 ,9, 11, 13, 14, 16, 17, 20, 23, 29, 30
	HS-3	開発者や管理者が，(HS-1, 2 に含まれない)Node やインフラリソース，外部依存先の仕様や挙動，制約，状態，サイズを加味した設定や入力を行っていない。また，それを検証する仕組みがない	2, 11, 13, 16, 27, 30
	HS-4	開発者や管理者が，サービス提供に必要なリソースを削除してしまう。また，それを検証する仕組みがない	26
不十分，不適切なプロセスモデル/フィードバック	-	-	-
コントロール経路の問題	HS-5	Data Plane から API Server への経路が失われる，疎通できない	5, 8, 18
	HS-6	Control Plane から API Server への経路が失われる，疎通できない	21, 22, 24
	HS-7	Data Plane から Infra. API への経路が失われる，疎通できない	15
	HS-8	Control Plane から Infra. API への経路が失われる，疎通できない	25
	HS-9	Control Plane から Pod や Admission Webhook への経路が失われる，疎通できない	28
被コントロールプロセスの問題	HS-10	被コントロールプロセスが長時間応答しない（過負荷によるフリーズやインフラ変更の待ち時間など）	1
	HS-11	Data Plane から API Server へ過剰な数の要求が行われる(API Server の能力と Node 数が不均衡)	7, 19

注目すべきは，非安全なコントロール入力を要因とするシナリオの数である．アプリ開発者やシステム管理者の入力したリソース量などの宣言がハザード要因になっており，それを防ぐ仕組みが不十分であることが窺える．一方で不十分，不適切なプロセスモデル/フィードバックを原因とするシナリオが無い．これは Reconciliation loop によるプロセスモデルの継続的な更新と結果整合の許容という特徴から説明できる．

4. 障害事例による評価と考察

コミュニティサイトで共有されている障害事例[4]を参考に，識別したハザードシナリオを評価する．

原因を特定し，根拠が説明されている事例から，原因が本分析の対象範囲にある事例を抽出した．結果，本分析で STPA により識別したシナリオで全ての事例を説明できた[表 3]．よって網羅性の観点で妥当と判断できる．

また，非安全なコントロール入力(HS-1，HS-2，HS-3)が支配的な要因であることが事例の数からも分かる．Kubernetes が宣言を，直接的で自由度が高く，入力時検証を任意とした負の側面を認める．

表 3 事例の原因とシナリオとの対応，数

原因	シナリオ	事例数
Pod のリソース利用量に Limit を設定しておらず OOM Kill やクラッシュが発生した	HS-1	6
Priority Class の設定ミスで重要度の高い Pod の Priority が低く再作成対象となった		2
認証情報取得アプリのメモリ利用量過多により OOM Kill と Pod 再作成が多発し，API Server への問い合わせが急増した		1
大量の DNS 問い合わせによる名前解決の遅延，DNS 稼働 Node の過負荷，問い合わせ元 Node のアウトバウンド通信過負荷	HS-2	6
CronJob，Job の設定ミスで大量の Job を実行，大量の Pending Pod によるスケジューリング過負荷，再実行ループの発生		5
Endpoint の削除に時間を要し，Ingress が削除済みの Pod にトラフィックを送信した		2
kubelet の API Server 問い合わせレート制限が低過ぎ，認証情報の取得ができなかった		2
アップグレード時に OPA が API Server の起動を拒否するポリシを適用し，API Server が起動できなかった		1
API Server が利用できない状態で Node の自動修復が機能し，Node 再作成を繰り返した		1
監査ログが Disk I/O を圧迫した		1
イメージプル過多でレジストリが要求を拒否した(ImagePullPolicy=Always を設定)		1
HPA と Deployment のレプリカ数設定が合っていない		1
ConfigMap/Secret の変更後に Pod を明示的に再作成せず，複数ある Pod が異なる設定で動作した		1
Node Pool 移行時に旧 Node Pool に Endpoint が残存した(Drain 漏れ)		1
Pod Disruption Budget を設定せず Pod が一斉に再作成した		1
敏感すぎる liveness prove で Pod が頻繁に再作成した		1
Cluster Autoscaler のスケールイン猶予時間が短過ぎ，急激に Node 数が減少した		1
アウトバウンド通信過多で conntrack テーブルが飽和，競合した	HS-3	3
CFS スケジューラの特性を理解せず CPU Limit を設定し，想定以上のスロットリングが発生した		1
Node，Pod ともにオートスケール設定をしたが，仮想ネットワークの Pod IP 仕様を誤解し，IP アドレスを割り当てられず，スケールに失敗した		1
CNI の SNAT 設定が不適切であり，割り当てられるポートが不足した		1
PV が存在しない/別データセンタにあり Pod を起動できなかった		1
起動コンテナ数が過多でファイルディスクリプタが枯渇した		1
Pod Lifecycle Event Generator relist などコンテナランタイムの処理がタイムアウトし Node が NotReady 状態となった（過負荷など)	HS-10	2
多 Node 環境において DaemonSet からの API 呼び出しが API Server の過負荷につながった	HS-11	1

5. 適用可能性の論点

5.1 入力時検証

識別したハザードシナリオと事例が示す通り，宣言の入力時検証は明らかな論点である．なお Kubernetes コミュニティは必要性の高まりから，Open Policy Agent[18]によるポリシベースの入力値検証機能を開発している．

ただし現状の厳密な把握を前提に，受け入れ可能な入力値を検証することには議論がある．その実現には低遅延なテレメトリが求められる．また，過度に現状を求めることは結果整合を許容して得た利点と相反するため，トレードオフを踏まえた判断が必要である．

5.2 アプリのエラー耐性

Kubernetes のように，構成管理を行うコンポーネントが非同期に並列動作する構造では，依存関係のある管理対象の間で設定に一時的な不整合が生じ得る．例えば Kubernetes では Service の Endpoint 追加，削除と，その転送先である Pod の作成，削除は並列して行われる．従って Pod を削除する際，Pod が Endpoint より先に削除される可能性がある．この間にクライアントが Service にアクセスすると，Endpoint リストの中から削除済み Pod の IP アドレスが選択され，コネクションが拒否，リセットされる恐れがある[付録 1: UCA-9]．Pod の停止と再作成は構成変更の基本戦略であり，軽視できない挙動である．

とはいえ，これは構造上の制約である．よって，エラーが発生し得ることを前提にアプリで緩和，対応すべきであろう．アプリの安全な停止と，呼び出し元での再試行がその例である．具体的には，プロセスや接続の終了と閉塞処理に加え，依存先の処理が完了するのに十分と思われる待機時間を確保するのが一般的である．

しかし待機時間の確保は環境の影響を受けるため確実ではない．例えば Endpoint 削除に伴う iptables(netfilter)の転送ルールを更新する時間は，各 Node の更新量や排他制御の影響を受ける．よって，緩和手段として用いるのが良い．そのため，エラー耐性を高めるためには，呼び出し元での再試行を合わせて実装することが望ましい．

そこで，再試行の実装によりエラー耐性を向上できることを検証した．検証は Kubernetes における一般的なゲートウェイ/フロントエンド/バックエンド構成の Web アプリで

行う[図 5]. Kubernetes のバージョンは 1.18.4 である．クラスタ外部とのゲートウェイに NGINX Ingress Controller[19]を，フロントエンドとバックエンドの Web アプリには podinfo[20]を採用した. クライアントが Ingress Controller の指定パスへ HTTP POST を行うと Ingress Controller がリクエストをフロントエンドへ転送し，さらにフロントエンドがバックエンドへ POST する．なお，各 Pod は複製を持ち全現用で動作する構成とした．つまり Pod の 1 つが停止しても縮退して回復を待つ.

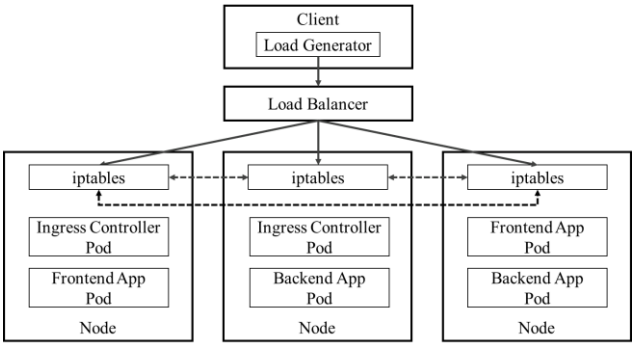

図 5 アプリ停止時挙動の検証環境

Pod を 1 つ停止してもクライアントへのエラー応答無く処理継続可能かを確認する．そこで，クライアントから HTTP POST を並列数 500 で実行し, 複製された Pod の 1 つをその間に停止し，応答を記録する．なお停止には Pod の delete API を用いる．この API によりコンテナには SIGTERM が送られ，Pod が終了する．合わせて，Pod の削除とは同期せず，並列して Service から Endpoint が削除される.

なお SIGTERM 受信時に NGINX Ingress Controller は 10 秒間，また，podinfo は 3 秒間，コネクションのドレインと Endpoint の削除を待つよう実装されている．加えて，NGINX Ingress Controller は再試行と接続先の切り替え機能を持つ．つまり，この構成ではフロントエンドが一時的に利用できない場合でも，再試行と切り替えを期待できる.

停止対象ごとに 5 回試行し，クライアントに対するエラー応答を含む試行数を表 4 に示す.

表 4 アプリ停止時挙動の検証結果

停止対象	エラー応答を含む試行数 (試行数: 5)
Ingress Controller	2
フロントエンド	0
バックエンド	2

フロントエンドの停止ではエラー応答が無い．つまり呼び出し元である Ingress Controller の再試行，切り替え機能が寄与している．一方で他の停止対象では，持続時間は短いが[表 5]エラー応答が発生した．エラーの発生しない試行もあり，Pod 停止時の各 Node やコンポーネントの状態に依存することが分かる.

表 5 アプリ停止時挙動のエラー内容

停止対象	TCP/IP エラー数	HTTP エラー数	アプリエラー数	エラー持続時間(秒)
Ingress Controller	5	0	0	0.018
	60	0-	0	0.279
バックエンド	0	0	1	0.002
	0	0	1	0.002

Ingress Controller 停止時の TCP/IP エラーはコード 104(ECONNRESET)と 111(ECONNREFUSED)であり，クライアントが削除済み Pod に対して接続を試み，RST パケットが返された結果である．また，バックエンドでのアプリエラーでは，Ingress Controller はクライアントへ正常な HTTP 応答を返す一方で，ペイロードにアプリからのエラーメッセージが記録されている．その内容は，フロントエンドがバックエンドに接続できないというものである.

仮定の通り，全てのエラーの原因は Pod の削除後に，iptables ルールに残った, 存在しない Pod へ転送を試みたことである．そして呼び出し元が再試行や切り替えなどエラー対処を行っていないケースでは，クライアントにエラー応答が返された.

高精度な時刻同期などで構成管理コンポーネント間のタイミングを合わせ，設定の非同期を原因とするエラーを回避することは可能であろう．しかし複雑化は否めない．よってシンプルさと並列性を重視するならば，アプリ層での対処が望ましい. それはアプリ自身での実装に限らない. 例えばサービスメッシュでプロキシとして使われる Envoy は再試行機能を有する[21]. アプリ層のエラー耐性は，宣言的構成管理の適用可否に影響する重要な論点である.

5.3 実行空間の分離と優先制御

Kubernetes では DNS サーバなど，重要度の高いコンポーネントがアプリと同じ Node で動作する構成が可能である. 従って入力時検証に加え, 実行時の優先制御が求められる. 例えば Kubernetes の OOM(Out Of Memory) Kill は，要求(Request)と制限(Limit)の宣言値と実際の使用量によって kill 対象の Pod を決定する．Node 全体の使用メモリ量が閾値(Eviction Policy)を超えた場合，使用量が Request を超えたコンテナを持つ Pod が候補となり，Priority Class と使用量を考慮したランク付けで停止対象を決定する．優先度の低い Pod を停止し，重要度の高いものを保護するポリシである．なお，不十分な優先度設定がアクシデントにつながる事例は多い[4].

このポリシを許容できず，アプリの強制停止を回避したいユースケースもあるであろう．よって他の分離方式や優

先度制御も議論されるべきである．ただし管理対象の保護が行き過ぎて利点を損なわないよう，全体最適の視点が必要である．例えば，Kubernetes は Node をいわゆる Pets ではなく，容易に取り替えられる Cattle[22]として扱うことで，運用者の負担を軽減した．

5.4 プルかプッシュか

Kubernetes は各コンポーネントが API Server をプルする構造を選択し，API Server の単純化に寄与している．一方，プロセスモデル操作の起点は各コンポーネントとなるため，その数に応じた API Server の処理能力が必要となる．DaemonSet など全 Node で設定を同じにするコンポーネントで考慮が不足し，API 呼び出しが自クラスタに対する DDoS 攻撃となった事例もある[4]．入力時検証などで解決すべき課題であるが，他領域への適用においては，規模や制御のパターンによってプッシュ型が適するケースもあるだろう．例えば，コントローラが中央集権的に順序やタイミングを制御する必要があればプッシュ型が向く．

5.5 管理対象の異種混在

例えば Kubernetes は Linux エコシステムで開発されたため，iptables など Linux カーネルの機能に強く依存しており，非 Linux 環境を管理対象にし難い．例えば Microsoft Windows[b]を Node として利用するために，管理される OS 側で Kubernetes 向けの機能追加が必要となっている[23]．しかし Windows コンテナのイメージサイズが大きくコンテナ作成時間，ひいては回復時間に影響するなど，機能追加だけでは解決できない根本的な課題が残っている．

管理対象が多様であれば，そのインタフェイスを抽象化し，透過性を確保するのが従来の定石であった．しかし Kubernetes は管理対象を絞り，その機能や特徴を前提に宣言的構成管理を実現した．よって適用領域の検討に際し，管理対象の多様さと抽象化は論点である．特にネットワークやストレージ領域においては管理対象が異種混在するケースが想像され，慎重な議論が必要であろう．

6. おわりに

本論文は Kubernetes の構造と事例の分析を通じ，宣言的構成管理の適用可能性と論点を導いた．Kubernetes の宣言的構成管理の実用化には，トレードオフを受け入れ，また，管理対象を絞り，対象が持つ機能を活用したことが寄与している．レイヤ化による責任分解と透過性の確保は一般的なアプローチであるが，全体の視点を欠くと各レイヤでの部分最適に陥る．従って，宣言的構成管理の適用には，基盤の構成要素やレイヤに閉じない，アプリやユーザを含めた全体最適の視点が重要である．

b) Microsoft，Microsoft Windows は，米国 Microsoft Corporation の米国およびその他の国における登録商標です．

参考文献

[1] Ansible, https://docs.ansible.com (参照 2020-08-13).
[2] Terraform, https://www.terraform.io (参照 2020-08-13).
[3] Burns, B., Grant, B., Oppenheimer, D., et al: Borg, omega, and Kubernetes, *Comm. ACM*, Vol.59, No.5, pp.50-57(2016).
[4] Kubernetes Failure Stories, https://github.com/hjacobs/kubernetes-failure-stories (参照 2020-08-13).
[5] Bozóki, S., Szalontai, J., Pethő, D., et al: Application of extreme value analysis for characterizing the execution time of resilience supporting mechanisms in Kubernetes, Proc. *EDCC '20,* pp.185-199, Springer(2020).
[6] Chen, X., Mao, Y., Mao, Z.M., et al: Declarative configuration management for complex and dynamic networks, Proc. *Co-NEXT '10*, No.6, pp.1-12, ACM(2010).
[7] Liu, C., Loo, B.T., Mao, Y.: Declarative automated cloud resource orchestration, Proc. *SoCC '11*, No.26, pp.1-8, ACM(2011).
[8] Loo, B.T., Condie, T., Garofalakis, M., et al: Declarative networking: language, execution and optimization. Proc. *SIGMOD '06*, pp.97-108, ACM(2006).
[9] Mao, M. and Humphrey, M.: A performance study on the VM startup time in the cloud, Proc. *CLOUD '12*, pp.423-430, IEEE(2012)
[10] Gunawi, H.S.,Hao, M., Suminto, R.O., et al: Why does the cloud stop computing? Lessons from hundreds of service outages. Proc. *SoCC '16*, pp.1-16, ACM(2016).
[11] CNCF: CNCF SURVEY 2019, https://www.cncf.io/wp-content/uploads/2020/03/CNCF_Survey_Report.pdf (参照 2020-08-13).
[12] Xavier, B., Ferreto, T. and Jersak, L.: Time provisioning evaluation of kvm docker and unikernels in a cloud platform. Proc. CCGrid '16, pp.277-280, IEEE(2016).
[13] Agache, A., Brooker, M., Florescu, A., et al: Firecracker: Lightweight Virtualization for Serverless Applications. Proc. NSDI '20, Usenix(2020).
[14] Burns, B: How Kubernetes Changes Operations. login Usenix Mag., Vol.40, Usenix(2015)
[15] Leveson, N.G.: Engineering A Safer World: Systems Thinking Applied to Safety, MIT Press(2011).
[16] Leveson, N.G. and Thomas, J P.: STPA HANDBOOK(2018).
[17] Leveson, N.G. and Thomas, J P., Shirasaka, S. (Translators into Japanese), et al.: STPA HANDBOOK 日本語版 Ver.0.2(2018).
[18] Open Policy Agent, https://github.com/open-policy-agent/opa (参照 2020-08-13).
[19] NGINX Ingress Controller, https://github.com/kubernetes/ingress-nginx (参照 2020-08-13).
[20] podinfo, https://github.com/stefanprodan/podinfo (参照 2020-08-13).
[21] envoy: How do I handle transient failures?, https://www.envoyproxy.io/docs/envoy/latest/faq/load_balancing/transient_failures (参照 2020-08-13).
[22] S. Tilkov. The modern cloud-based platform. *IEEE Software*, Vol. 32, No.2, pp. 112-116, IEEE(2015).
[23] Microsoft: Built-in Support for Kubernetes – What's new in Windows Server 2019, https://docs.microsoft.com/en-us/windows-server/get-started-19/whats-new-19#built-in-support-for-kubernetes (参照 2020-08-13).

付録 1 非安全なコントロールアクション(UCA: Unsafe Control Action)一覧

コントロールアクション	与えられないとハザード	与えられるとハザード	早すぎ，遅過ぎ，誤順序
CA-3 (kubelet から Container Runtime)	UCA-1: Pod は正常に動作しているが，relist 処理の遅延などで，そう判断されない。結果，Node が NotReady 状態と判断され，サービス提供に必要なリソースが減少する[H-1]	UCA-2: ImagePullPolicy=Always 設定で短時間に大量のコンテナを作成する。ネットワーク帯域の圧迫だけでなく，レジストリに拒否され Pod が作成できない[H-1]	
CA-4 (kubelet から Pod)		UCA-3: 敏感過ぎる liveness Probe 設定により Pod 再作成が頻発し，サービス提供能力が低下する[H-1]	UCA-4: Readiness Probe の不備で準備不十分な Pod へトラフィックが転送される[H-3]
CA-5 (kubelet から API Server)	UCA-5: Node が正常な状態にも関わらず，API Server に伝わらない。経路が問題の場合はクラスタ全体に影響が及ぶ。例えば，Node 自動修復機能が Node 作成と削除をクラスタ全体で繰り返す [H-1, H-2, H-3] UCA-6: 十分な API Server 呼び出しレートが設定されていない[H-1, H-2, H-3]	UCA-7: API Server が過剰に呼び出され，API Server が過負荷状態に陥る[H-1, H-2, H-3]	
CA-7 (kube-proxy から OS 機能/設定)	UCA-8: iptables，conntrack など IP 転送に関する更新が行われない[H-3]		UCA-9: IP 転送に関する更新がコンポーネントや Node 間で同期しない (Ingress と Service の Endpoint など)[H-3]
CA-8 (Container Runtime から OS 機能/設定)	UCA-10: サービス提供に必要な Pod に十分な Node のリソースや優先度が割り当てられない。リソース利用率が高まると強制終了される[H-1, H-2, H-3] UCA-11: 適切な SNAT アルゴリズムなどサービスレベル遵守に必要な設定が行われない[H-1]	UCA-12: サービス提供に貢献度の低い Pod に過剰な Node のリソースや高い優先度が割り当てられる。リソース利用率が高まるとサービス提供に必要な Pod や Node のプロセスが強制終了される[H-1, H-2, H-3] UCA-13: カーネルの監査ログ有効化など Node 単位で影響する負荷の高い設定が行われる。Pod に十分なリソースが割り当てられない[H-1]	
CA-9 (Container Runtime から Pod)		UCA-14: 過剰な DNS 問い合わせを行うよう設定される。例えば既定の ndots: 5 設定を見直していない[H-1, H-2, H-3]	
CA-11 (Pod から Infrastructure API)	UCA-15: Cluster Autoscaler による Node やボリューム追加などにおいて，サービスレベル遵守に必要なインフラリソースの作成指示が行われない[H-1]	UCA-16: Pod に割り当てられない，別データセンタでのボリューム作成など，不整合のあるリソースの作成指示が行われる[H-1, H-2, H-3]	UCA-17: Cluster Autoscaler による Node 削除などにおいて，リソースの削除や割当解除の指示が早すぎる。設定した猶予時間が短すぎる[H-1, H-3]
CA-12 (Pod から API Server)	UCA-18: サービスメッシュの Operator などシステム全体に影響の大きな Pod からのコントロールアクションが行われない。関連する変更操作が停止する[H-1, H-2, H-3]	UCA-19: API Server が過剰に呼び出され，API Server が過負荷状態に陥る[H-1, H-2, H-3]	
CA-13 (開発者や管理者から API Server)			UCA-20: ConfigMap や Secret は Pod 作成のタイミングで読み込まれるため，変更後に意図的に再作成しないと Pod 間で設定の不一致が生じる[H-3]
CA-14 (kube-controller manager/kube-scheduler から API Server)	UCA-21: リソース作成のイベント(チェーン)が発生しない，もしくは途切れ，新規 Pod が作成されない[H-1, H-2, H-3] UCA-22: Pod のスケジューリング結果が登録されず，新規 Pod が作成されない[H-1]	UCA-23: Toleration の指定漏れなどスケジュールできない Pod や CronJob が大量に要求される。Pending 状態の大量の Pod がスケジューリング負荷の増大を招く[H-1]	
CA-15 (cloud-controller manager から API Server)	UCA-24: 作成したインフラリソースの情報が登録されず，利用可能と認識されない。処理量の増加に対応できない[H-1]		
CA-16 (cloud-controller manager から infra. API)	UCA-25: 要求したインフラリソースが作成されない[H-1, H-2, H-3]	UCA-26: 必要なインフラリソースが削除される[H-1, H-2, H-3] UCA-27: 利用可能量や制約，権限を超えたインフラリソースの作成が指示され，失敗する[H-1]	
CA-17 (API Server から Admission Webhook)	UCA-28: Webhook が実行されず，Sidecar コンテナなどアプリの依存するリソースが Pod に注入されない[H-2, H-3]	UCA-29: Admission Control の定義やポリシに適合しないリソースが要求され，失敗する[H-2, H-3]	UCA-30: Webhook 呼び出し先が準備できていない状態で実行される[H-2, H-3]