平成２８年度修士論文

インタークラウドを活用した
遠隔データ保存の自動化機構

大学院情報システム学研究科
情報ネットワークシステム学専攻

学 籍 番 号 :　1552022
氏 名:　溝田　敦也
主任指導教員: 吉永　努 教授
指 導 教 員 :　大坐畠　智 准教授
指 導 教 員 :　荻野　長生 客員教授
提出年月日 :　平成29年1月26日

(表紙裏)

目 次

図 目 次

表 目 次

第1章　序論

物理マシンが持つ計算機資源を複数の仮想マシン (VM) として論理的に分割し，情報システムに必要な物理サーバ数を削減できる仮想化が普及した．この仮想化機構の発展に加え，インターネットの急速な拡大により，情報システムを構成するマシン群を自前で用意せず，データセンタ上の仮想化された計算機資源にシステムを構築してインターネット越しにサービスを提供するクラウドが提唱された．Amazon Web Service(AWS)[1] や Google Cloud Platform[2], Microsoft Azure[3] 等の事業者がクラウドサービスを世界規模で展開しており，メールやストレージといったアプリケーションや，情報システムの運用基盤となる VM をユーザに貸与するサービスを提供している．サービスの利用量や時間に応じて料金を徴収する事業形態と，保守管理の手間を省き運用コストを低減するメリットにより，現在では EC サイトや官公庁などの重要な社会基盤を担うシステムもクラウドサービスに依存している．

様々な情報システムを運用するクラウドの技術課題として，継続的なサービス提供の実現が挙げられる．クラウドが普及する以前から，システムが甚大災害や人的ミスといった計画外のサービス停止による機会損失を抑える迅速なシステム復旧を支援する技術が数多く提案されているが，クラウド環境のシステムに対しても同技術が必要である．2012 年に AWS が管理するアメリカ・バージニア北部のデータセンタにおいて，激しい豪雨による停電が起き，当拠点上に構築された多数のサービスが数時間停止した事例がある [4]．世界的な動画配信サービスの Netflix や，写真共有ができる Instagram などの人気サービスが当拠点のクラウド環境を使用しており，データセンタの停電によるサービス停止が多くユーザに影響を与えた．長時間のサービス停止は利用者・管理者双方に機会損失を招くため，クラウド上に展開された情報システムに対しても継続的なサービス提供を実現する仕組みが求められている．

情報システムの停止に対する復旧の仕組みとして，システムの待機系を事前に用意しておき，主系側の停止時にサービス運用を待機系に切り替える方法がある [5]．待機系も常時稼働状態にしておけば，システムへのアクセス先の変更のみでサービスの継続的な運用は可能であるが，平常時から待機系の維持管理コストが少なからず発生する．

一方，待機系を用意しない復旧の仕組みとして，クラウドオーケストレーションが提供されている [6],[7]．複数クラウドの横断的な扱いと，システムを構成する仮想マシン群の一括構築機能により，甚大災害や大規模障害からの迅速なサービス復旧に有効に機能する．しかし，システムの運用データを平常時からバックアップし，サービス復旧後にデータを復元する取り組みは行われていなかった．また，複数のクラウド環境を活用したサービス保護において，システムとデータ両方を復旧復元する取り組みは少ない．

そこで本研究では，クラウドオーケストレーションのサービス復旧機能に，遠隔地へのデータ保存と復旧後にデータを復元する仕組みの導入を提案する．サービスが提供されているクラウド環境が停止してから，データの復元を含めて別の拠点で再稼働するまでの一連の動作を自動化し，複数のクラウド環境で有効性を示した．2 種類の小規模な情報システムを対象とした評価実験の結果，クラウド環境の移行に要する時間は約 10 分程度であることから，クラウドオーケストレー

ションが提供する待機系無しによる復旧機能であっても短時間でのサービス再稼働が可能であることを確認した．

以降，2 章では関連研究について述べ，3 章でデータを含めたサービス再稼働の仕組みの提案と実装を述べる．4 章でブログシステムと EC サイトのシステムを対象とした評価実験，5 章で議論を行い，6 章で結論を述べる．

第2章　関連研究

2.1　データ保護の取り組み

システムを再稼働してサービスを継続提供するためには，必要な運用データを保護する仕組みが必要不可欠である．地理的に異なる 3 箇所以上の拠点にバックアップデータを分散配置することで，拠点規模の停止が 2 箇所同時に発生してもデータ損失を防ぐ仕組みが提案されている [8],[9],[10].

データのバックアップには同期方式と非同期方式があり，採用する方式に応じて情報システムのパフォーマンスに影響を与える．同期方式は，遠隔地へのバックアップ処理完了後にアプリケーション側に応答を返す方式であるため，システムの応答性能が悪化する．このため，遠隔地へのバックアップはデータの更新とは独立して行われる非同期方式の場合が多く，上記文献でも採用されている．非同期方式では，システムのローカル上のデータ更新頻度とは独立してバックアップを実行するため，システム停止時に一部のデータが消失する可能性がある．このデータの消失量を示す指標があり，目標復旧地点 (Recovery-Point-Objective：RPO) と定義され，システム停止時から遡って復元可能な時刻までの差を表す．RPO とバックアップ品質に払うコストはトレードオフの関係にあるため，両者の関係を数式化する方式も提案されており，システム構築前の設計段階でバックアップに必要な資源コストを過不足無く見積もることが可能である [11].

一方で，上記全ての提案手法はデータ保護のみに焦点を当てており，サービス基盤となるシステム自体の冗長化に関しては検討されていない．システムを再稼働するためには，データの保護だけでなくサービス提供を行うシステム基盤も必要である．

2.2　システムインフラの復旧

AWS や Google Cloud Platform，Microsoft Azure など，クラウドサービスを提供する事業者は多数存在する．各クラウド事業者自身が自社インフラ上のシステム監視や管理機能を提供しており，継続的なシステム運用に活用可能であるが，事業者毎に制御命令が異なるため，停止要因の影響が事業者の管轄するインフラ全体に波及する場合には対応できない．特定のクラウド事業者が提供する制御命令に依存する問題はベンダロックインと呼ばれ，サービス停止に強いシステム構築の障壁となっている．

ベンダロックインの問題を克服するために，各事業者が提供する制御命令を統一する仕組みの開発が取り組まれている．文献 [12] では，事業者毎に提供されるプラットフォームを一つのリソースプールとして扱い，拠点間を跨いだ VM のスケールアウトを実現している．複数のクラウドを横断して制御し，情報システムを構成する複数の VM を一括構築できるクラウドオーケストレーションも多数登場した [6],[7]．クラウドオーケストレーションは，システムの状態を監視するツールと組み合わせることで，サービスの迅速な復旧が可能である．

複数のクラウドを統一して制御する機能は，システム停止の影響を受けない他拠点のクラウド環境を利用できるため，拠点間を跨いだサービス復旧には有効に働く．しかし，上記の仕組みや

ツールにはシステム上の運用データを含めた管理・保護に関しては十分に行われていない．このため，データも含めてサービスを再稼働するには，システム復旧後のデータ復元作業が別途必要になる．

2.3 システムとデータ両方を保護する取り組み

情報システムの運用には，システム復旧とデータ復元両方が必要であり，再稼働時は両者を統括して復旧・復元できることが望まれる．そのため，システム復旧に複数のクラウドを活用し，データの保護も同時に行う取り組みが提案されている．

待機系のシステムを地理的に離れた別の拠点に用意し，常に運用データを待機系に複製しておくことで，主系側の停止に対し待機系に迅速に切り替えるホットスタンバイの方式は従来から実現しており，クラウド環境でも活用されている [5]．また，待機系システムとの同期間隔を抑えてコスト削減を行うウォームスタンバイの方式により，待機系側のプラットフォームに依存せずにシステムを自動で再稼働できる仕組みがある [13]．ところが，これらの方式は待機系のシステムを平常稼働時から用意する方法のため，余剰な運用コストが少なからず常に発生する．

本研究では，平常時の運用コストを抑え，かつシステム復旧とデータ復元の両立を目的として，クラウドオーケストレーションに備わる待機系無しのサービス復旧機能を活用し，自動でのデータのバックアップと復元操作を新たに追加する．これにより，システムの復旧とバックアップデータの復元を両立した短時間でのサービス再稼働を実現する．

第3章　データを含めたサービス再稼働の提案

3.1　提案手法の概要

システムの復旧とデータの復元両方を実現するため，クラウドオーケストレーションに備わるシステム復旧機能に，定期的なバックアップと復元の操作を追加した．正常稼働時は，システムが運用中に生成したデータを非同期で遠隔地にバックアップすることで保護し，停止時のクラウドオーケストレータによるサービス復旧と同時に，遠隔地へと保存していたバックアップを取得しデータを復元することで，インタークラウド環境におけるシステム復旧とデータ復元の両立を実現する．バックアップ先は停止要因の影響を受けない遠隔地のストレージサービスとし，保存されたデータは複数の拠点に分散配置されるため，バックアップが完了されたデータに関しては消失しないとする．

3.2　クラウドオーケストレーションの活用

3.2.1　クラウドオーケストレーションの概要と特徴

本研究で利用するクラウドオーケストレーションは，複数のクラウド環境上の VM やネットワークの制御を行うツールであり，大きく分けると以下 3 点の機能を持つ．

(a) クラウド上に複数のサーバ群を一括で自動構築する機能

(b) 停止したシステムを自動で復旧する機能

(c) 異なる複数のクラウド環境を統一して制御する機能

クラウド上に複数 VM で構成されるシステムを用意する際，VM の起動からアプリケーションの導入までの複雑な作業を決められた順番で行わなければならない．クラウドオーケストレーションは複雑なシステム構築作業を一括で行い，簡単な操作でクラウドサービスを用意できる．構築の際は専用の構築用スクリプトを利用し，クラウド基盤毎の制御命令に変換され，異なる基盤上に同じ構成のシステムを構築可能である．また，システムの監視機能と連携することで，対象システムの停止を検知すると，別のクラウド環境にシステムを移行できる．

TIS 株式会社が開発している CloudConductor，SCSK 株式会社の Prime Cloud Controller 等，クラウドオーケストレーションと呼ばれるツールは多数提供されているが，本研究ではシステム復旧に加えてデータ保護の仕組みを導入する目的で CloudConductor を使用した．

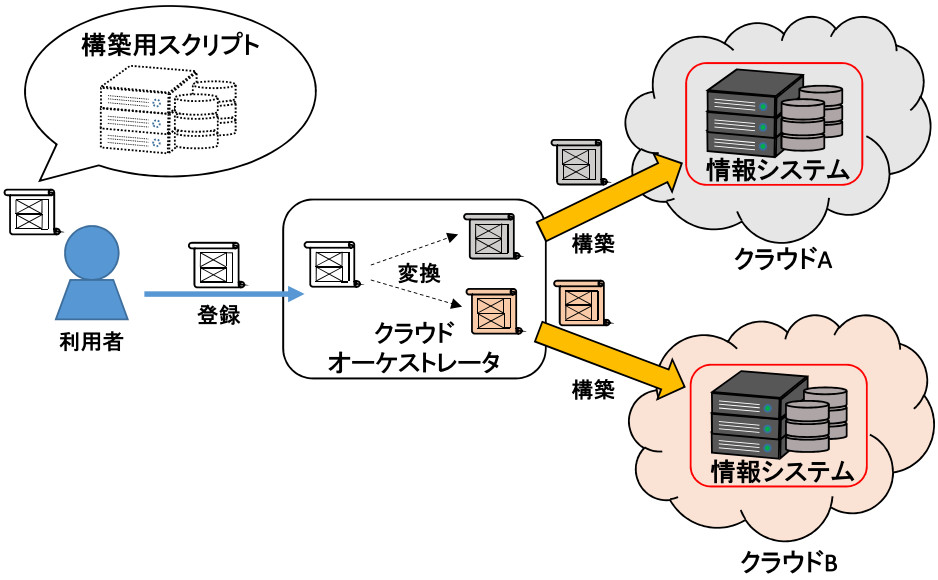

図 3.2.1: クラウドオーケストレータによるシステム構築

3.2.2 システムの構成が記述された構築用スクリプト

クラウドオーケストレーションは，複数のサーバ群からなる情報システムの構成が記述された構築用スクリプトと，スクリプトを読み込みシステムの一括構築を行うソフトウェアであるクラウドオーケストレータの 2 種類で構成される．構築用スクリプトには，システム全体の構成やサーバ間の依存関係が記述されており，クラウドオーケストレータが解釈してシステムを構築するために必要である．図 3.2.1 において，クラウドオーケストレータが構築用スクリプトを解釈すると，展開先のクラウド環境に合わせた制御命令に変換し，情報システムを一括で自動構築する．システム構築用スクリプトは，Chef や Puppet 等の既存の自動化ツールで構成されており，クラウドオーケストレータの利用者は各種ツール内にシステムの要件を記述できる．よって，システムの監視や運用データのレプリケーション，セキュリティの設定等を構築用スクリプト内に記述することで，システムの要件定義を満たす構成を自由に開発可能である．

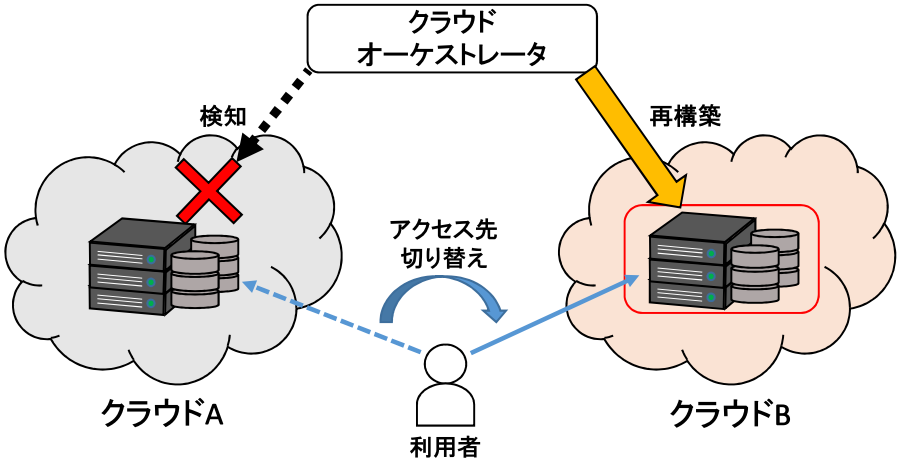

図 3.2.2: クラウドオーケストレータによるシステム復旧の動作

3.2.3 障害対応への応用

システムの停止検知機能と組み合わせることで，インタークラウド環境における迅速な障害対応に応用できる．検知機能は，対象システムのプロセスやポートの状態を恒常的に監視してサービスの稼働状況を確認する．CloudConductor に備わる検知機能では，システム監視ツールである Zabbix が活用されおり，対象システムの停止条件を構築前に設定する．対象システムの状態が事前に設定した停止条件を満たした場合，クラウドオーケストレータにシステムの復旧を依頼する．その後命令を受けたクラウドオーケストレータが，別の事業者が管理するクラウド環境にシステムを再構築し，DNS を更新することでアクセス先を旧システムから切り替える (図 3.2.2)．

クラウドオーケストレータによるサービス復旧は，平常時は待機系を用意せず，復旧時のみシステムを新規構築する方式であるため，平常稼働時に必要なコストが削減される．また，クラウドオーケストレータが一連の復旧作業を自動化するため，一瞬の停止も認められないミッションクリティカルなシステムで無い限り，停止時の迅速な対処に貢献する．一方，クラウドオーケストレータの復旧の仕組みには，平常稼働時の運用中に生成されたデータの移行は行われないため，特に運用データが根幹をなすシステムの利用は困難である．

3.3 遠隔データ保存とシステム復旧後のデータ復元

3.3.1 データ保護機能の導入

本研究では，クラウドオーケストレータが解釈するシステム構築用スクリプトに，平常時の遠隔データ保存と再構築後のデータ復元の操作を記述することで，システム復旧とデータ復元の両立を実現した．正常稼働時は，システムが生成するデータを定期的に外部にバックアップすることで保護し，サービス停止時のシステム再構築と同時に，バックアップしたデータを復元する．

3.3.2 遠隔データ保存の方法

データをバックアップするプログラムを事前に開発し，構築用スクリプトに組み込むことで，クラウドオーケストレータがシステム構築と同時に各サーバにプログラムを配置・展開する．バックアップ用のプログラムは定期的に，または任意のタイミングで実行され，生成データを遠隔地のストレージに保存する．

本研究の提案手法は定期的なバックアップを行うため，バックアップしたデータは常にシステム上のデータより古い．よって，バックアップとシステム上のデータの差がデータ消失量となる．システムに応じて生成されるデータの重要度は異なるため，管理者が許容可能な損失データ量を自由に設定できるのが望ましい．そこで，損失データ量を示す指標である RPO を活用し，目標とする RPO 値をシステム構築時に設定できるようにすることで，設定値に応じてバックアップ頻度を決定する．

RPO とは，サービス停止時刻から遡りデータ復元が可能な時刻の差を表す．例えば図 3.3.1 に示すように，RPO が 5 分と設定された場合，システム停止時から 5 分以前までに生成されたデータは全てバックアップが完了され復元できることを保証し，5 分以降のデータは損失を許す．RPO を短く設定すれば損失するデータ量が少なくなる一方，その分短い頻度でバックアップ処理を行うことになり回線コストが増大する．

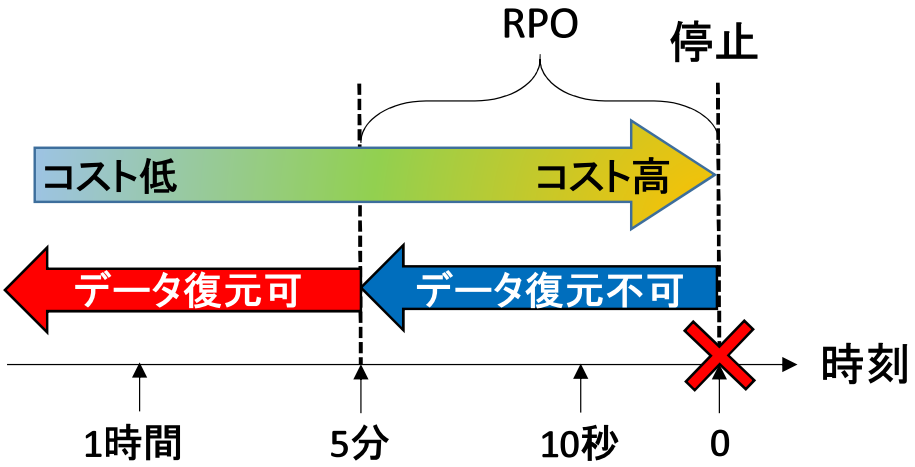

図 3.3.1: RPO の説明

3.3.3 サービス復旧時のデータ復元

データ復元を行うプログラムも同様に開発し，クラウドオーケストレータがシステム構築に利用する構築用スクリプトに組み込む．クラウドオーケストレータによるシステム再構築後に同プログラムを実行し，遠隔地に保存していた複数のバックアップファイルを取得してシステムに正しい順番で書き戻すことで，停止前の最後にバックアップした状態で再びサービスが再稼働する．

3.4 データのバックアップと復元の実装

3.4.1 バックアップ間隔の設定

平常稼働時のバックアップにおいて，頻度は必要最低限に抑えつつデータ消失量を少なくし，なおかつデータ復元時の所要時間が短い事が望ましい．よって，バックアップ処理を行うプログラムに，管理者が自由に RPO を設定する変数を用意し，それに応じて最低限の頻度を計算しバックアップ処理を行うように拡張した．

図 3.4.1 に示すように，バックアップ処理を時間 t 毎に絶え間なく続ける場合，バックアップするデータが届き終わる直前にシステムが停止すると，2t 分のデータが消失し最大の消失量となる．一方，クラウド事業者によってはバックアップの為に十分に広い帯域を持つ専用線を提供しているサービスもあるため，バックアップ用の回線速度は一定とし，かつ秒間で生成されるデータ量よりも大きい場合を仮定する．これにより，任意のバックアップ間隔で必ずデータ転送が完了するため，クラウドオーケストレータの利用者が事前に設定した RPO を満足するために，設定値の 1/2 の間隔でバックアップ処理を自動実行するように実装した．

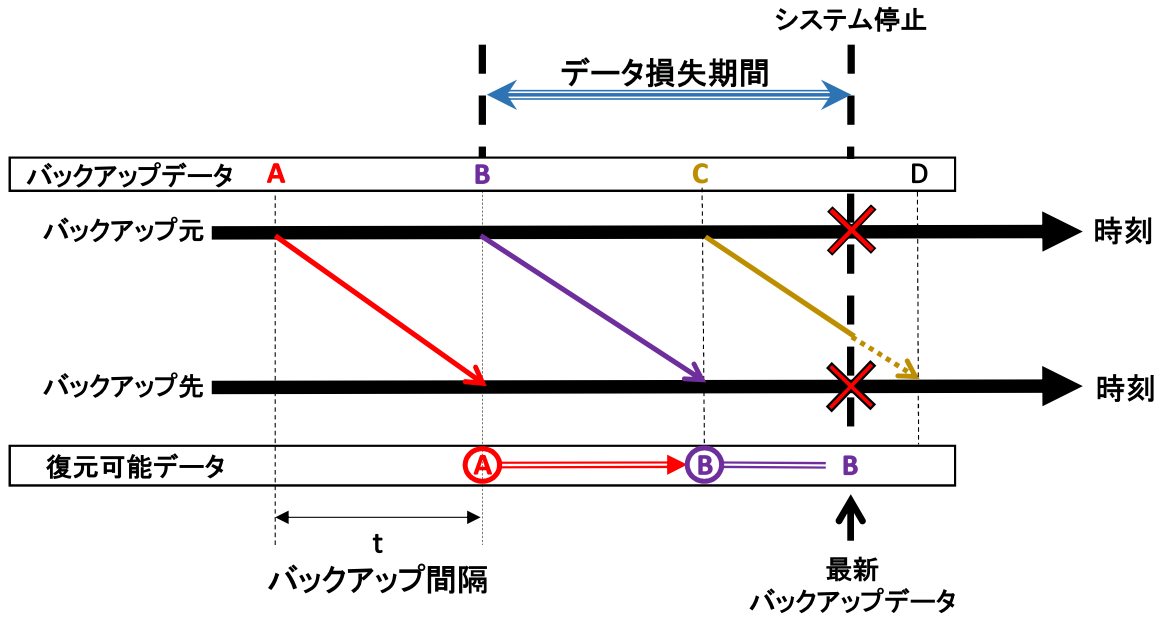

図 3.4.1: システム停止時のデータ損失時間範囲

3.4.2 平常時の定期的なバックアップ処理の実装

定期的なバックアップ処理において，損失データ量を抑えつつ復元時間を短くするのが望ましい．そこで，フルバックアップと増分バックアップの 2 種類の処理を併用する．

システム停止時に損失するデータ量と復元に掛かる時間は，トレードオフの関係にある．一般的な方法であるフルバックアップは，データベース上の全データを取得する方法であり，最新のバックアップのみを復元時に必要とするため，復元時間が短い特徴がある．しかし，全データを取得する特性上バックアップファイルのデータ量が大きく，転送時にネットワークに高負荷を掛けてしまう．このため，頻繁にバックアップを実行できず，システム停止時に損失するデータ量が大きくなる場合がある．一方，増分バックアップは前回のバックアップからの差分のみを取得する方法であるため，取得するデータ量が少ない分頻繁に実行でき，損失するデータ量を抑えられる．ところが，差分を取得する仕組み上復元時には取得した全てのバックアップファイルが必要となり，作業が煩雑になる．

そこで図 3.4.2 に示すように，長い頻度でフルバックアップを取得・更新し，合間の短い間隔で増分バックアップを取得する方法を採用した．フルバックアップの更新条件である増分バックアップの総データ量も，プログラム内で任意に設定できるようにしておく．これにより，復元時は最新のフルバックアップと以降の増分バックアップのみを取得すれば良く，損失データ量を抑えつつ短い復元時間を実現できる．

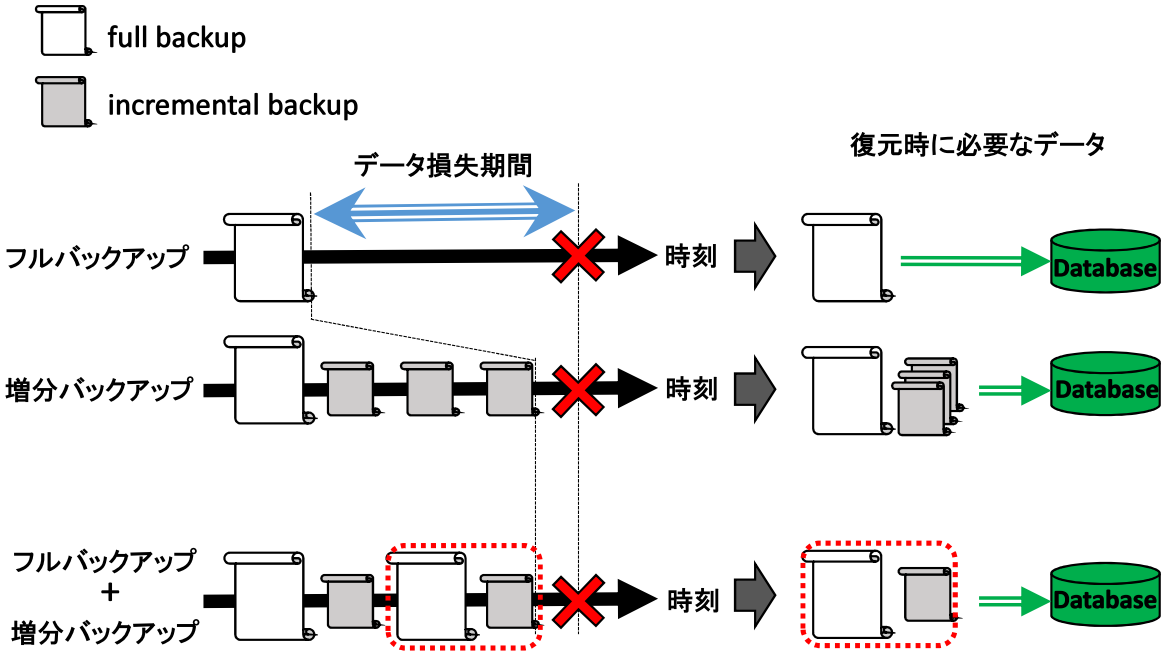

図 3.4.2: プログラムに組み込むバックアップの方法

3.4.3 RPO を満足するためのバックアップの圧縮

システムに生成される秒間のデータ生成量が回線速度を上回る場合は，任意に設定したバックアップ間隔以内に処理が完了しないため，設定 RPO を満足するのは困難となる．よりデータ生成量が大きいシステムでも RPO を設定可能とするためには，バックアップするデータ量を削減する工夫が必要である．そこで，将来的なサービス負荷の上昇に対処するため，取得した増分バックアップファイルは転送前に圧縮を掛けるようにプログラムを拡張した．増分バックアップはテキスト形式で記述された部分が多く，圧縮を掛けることによりデータ量を大幅に削減できる．zip,gzip,tar 等の複数の圧縮形式で削減量を比較したところ，lzma 形式が元のデータ量から約 1/7 に削減可能で最も圧縮効率が良いため，同方式を採用した．

3.4.4 データ復元の実装

データの復元は，外部から複数のバックアップファイルを取得して順番に書き戻す．複数のファイルがバックアップされているが，データ復元時は最新のフルバックアップと以降の増分バックアップのみを取得する必要がある．開発したプログラムは，遠隔地に保存していた最新のフルバックアップと以降の増分バックアップのみを取得する．

また，データ復元時に取得した増分バックアップデータを書き戻す際は，圧縮されたデータを展開するようにプログラムに書き加えた．バックアップしたデータを遠隔地からダウンロードする際は圧縮された状態で行い，データ復元処理を行う直前で展開する．

第4章　評価

4.1　ブログサービスによる停止時間の評価

4.1.1　実験環境

3 章で述べた提案手法の有効性を検証するために，図 4.1.1 に示すような実験環境を用意し，対象システムの停止に対するクラウドオーケストレータの復旧及びデータの復元が完了するまでの時間 (停止時間) を計測した．評価には地理的に離れた 2 拠点のクラウド基盤を用意した．片方の拠点のクラウド基盤上にシステムを構築し，もう片方の拠点ではクラウドオーケストレータを設置してシステムを監視する．システム上にはブログサービスが稼働しており，生成されたデータは外部のストレージサービスに保存するように設定する．

本研究でのサービス停止は，クラウドオーケストレータが導入された計算ホストと，対象システムが稼働するクラウド基盤間の通信路を遮断することで実現する．したがって，停止から別拠点でサービスが再稼働するまでに，(a) 停止の検知，(b) 別拠点へのシステムの再構築，(c) バックアップしたデータの復元の 3 ステップが自動で行われ，実験環境の構成が図 4.1.2 に変化する．評価では上記 3 ステップの経過時間を計測し，合計した時間を停止時間とする．

各拠点のクラウド基盤にはプライベートクラウド環境である OpenStack を導入し，クラウドオーケストレータとして TIS 株式会社が提供する CloudConductor[6]，バックアップ先のストレージとして Amazon Web Service が提供する Amazon S3[14] を使用する．対象システムは，データベースである MySQL が導入された VM1 台構成とし，ブログサービスである WordPress が稼働している．この実験では RPO 値を設定せず，任意のタイミングでバックアップを実行する．システム構築直後にフルバックアップ処理を行い，1 件の投稿後に増分バックアップ処理を行った後にサービスを停止させる．この実験で取得するデータ量は小さいため，取得した増分バックアップファイルは非圧縮状態で転送した．

停止させるシステムに使用した VM の構成及び CloudConductor を導入した計算ホストの構成を表 4.1 に．OpenStack を導入した計算ホストの構成を表 4.2 に示す．

表 4.1: 実験で使用した VM とクラウドオーケストレータを導入した計算ホストの構成

	対象システムの VM	クラウドオーケストレータ用の計算ホスト
CPU	1 Core @ 3.0GHz	4 Cores @ 3.4GHz
RAM	2GB	8GB
HDD	20GB	80GB
OS	CentOS 7.2 (64bit)	CentOS 6.7 (64bit)

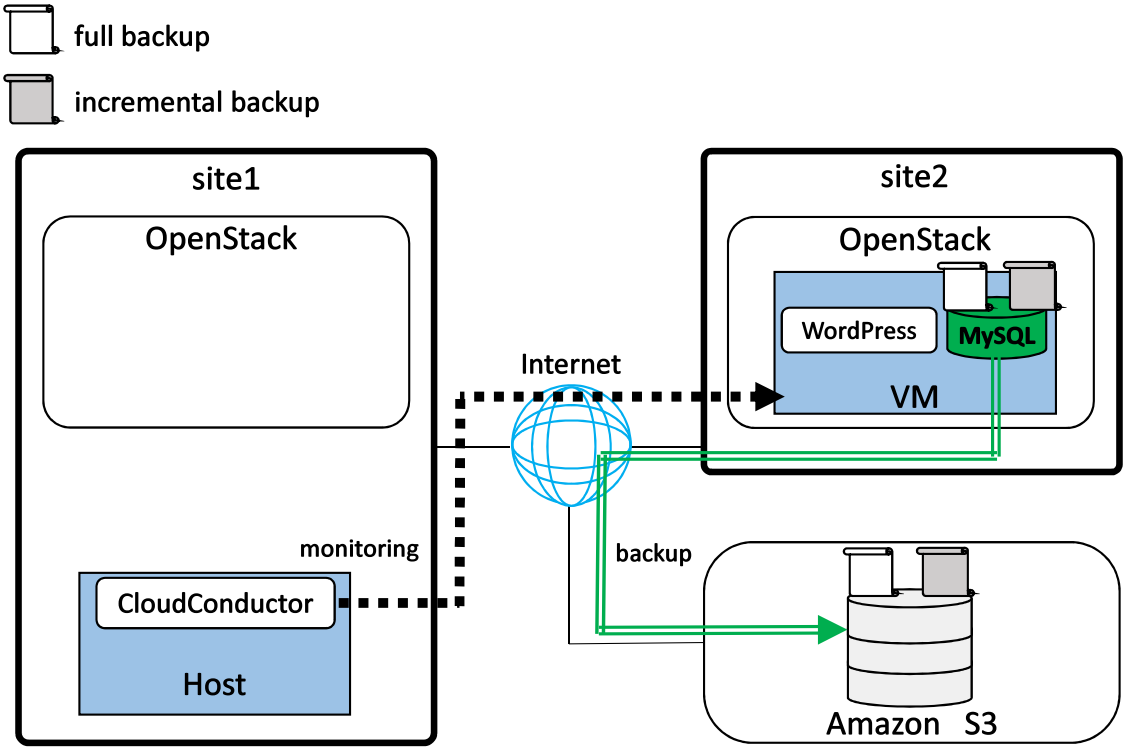

図 4.1.1: 評価に使用したクラウド環境

表 4.2: クラウド環境を導入した計算ホストの構成

拠点 1	CPU	Intel Core i7-4770 @ 3.40GHz
	RAM	32GB
	HDD	2.0TB
拠点 2	CPU	Intel Xeon E5-4770 @ 2.60GHz
	RAM	64GB
	HDD	500GB
各拠点共通	OS	CentOS 7.2 (64bit)
	Hypervisor	Linux KVM
	Cloud 基盤	OpenStack Liberty

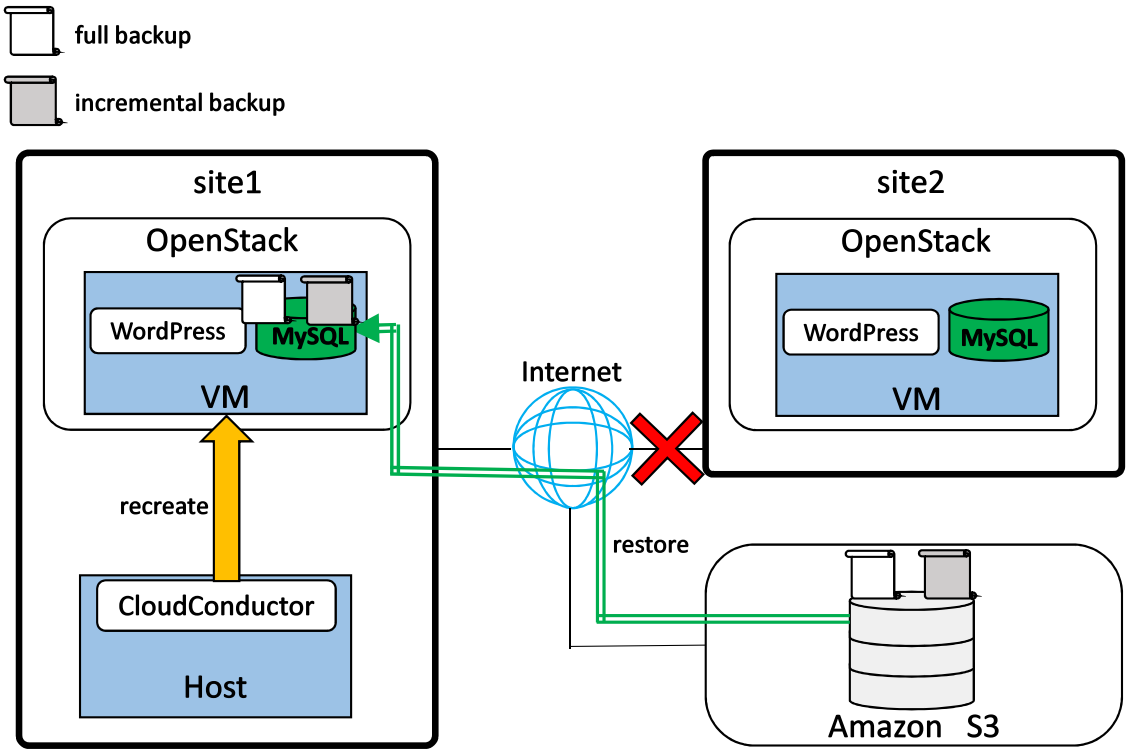

図 4.1.2: システム再稼働時の評価環境

4.1.2 停止時間の結果及び内訳

CloudConductor が生成したログデータのタイムスタンプを元にし，サービス停止から別拠点にシステム復旧とデータ復元が完了するまでの時間を計測した．各ステップで経過した時間とその内訳を表 4.3 に示す．検知・システム復旧・データ復元の 3 ステップ全ての実行時間は全体として 9 分程度となった．

図 4.1.3 に示すように，停止前と再稼働後のブログシステムの状態をブラウザから確認したところ，再稼働後のブログシステムに停止前の投稿が復元されていた．また，再稼働したシステムに

表 4.3: 停止時間とその内訳

検知時間	6 分 26 秒
再構築時間	2 分 28 秒
データ復元時間	0 分 30 秒
停止時間	9 分 24 秒

対して再び 1 件の投稿を行ったところ，サービスが正しく動いていることも確認できた．

図 4.1.3: 評価前後でのブログシステムの状態

システム停止の検知が完了するのに約 6 分と長い時間が経過した．これは，CloudConductor がクラウド環境の停止を検知する際の仕様によるものである．CloudConductor は，システム停止の検知後に同一プラットフォームに再構築命令を発行し，タイムアウトが発生することで初めてクラウド環境の停止を検知できる．したがって検知にかかった時間は，システム停止の検知時間に加え再構築のタイムアウト時間も含まれている．

再構築は 2 分の時間を要しており，VM の起動時間と，システム上で稼働するアプリケーションの展開時間の両方が含まれる．一方，CloudConductor で設計図にしたがってシステムを構築・再構築する際，事前にシステムを構成する VM のイメージを各拠点に配置する仕組みになっている．VM のイメージに，稼働させるアプリケーションである WordPress と MySQL をあらかじめインストールしているため，再構築時は VM の構築時間とアプリケーション・ネットワークの設定のみ

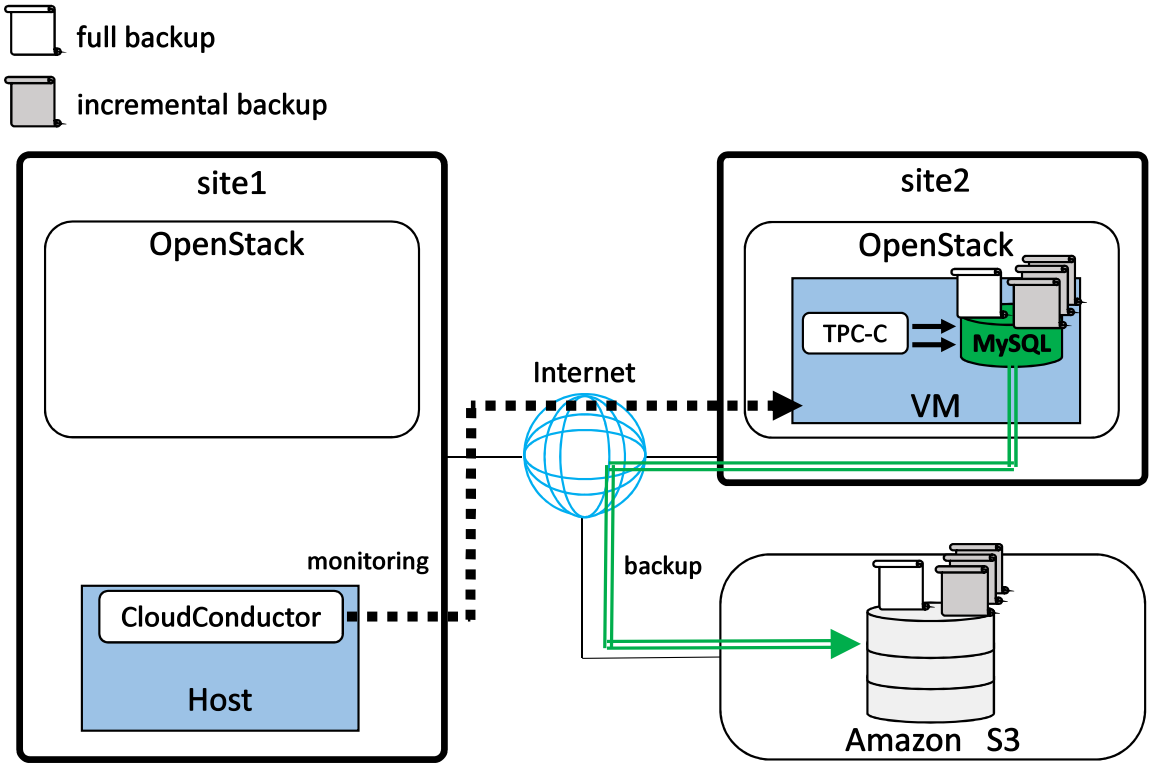

図 4.1.4: EC サイトを模したシステムでの評価環境

となり，時間短縮に寄与した．

一方，データの復元においては，バックアップデータの取得及び復元作業も含めて 30 秒と非常に短い時間になった．これは，実験で取得したフルバックアップデータ量が 532KB，増分バックアップのデータ量が 238KB とバックアップデータ総量が少なかったことに起因する．

4.2 EC サイトを模したシステムでの評価

4.2.1 実験環境

構築用スクリプトに導入した RPO の設定機能の有用性を検証する目的で，図 4.1.4 に示すように，前節で利用した同様の実験環境で停止時間の評価を行った．停止時間の計測に加え，損失データ量も評価して RPO の達成可否も確認する．前節とは異なり，対象のシステムには EC サイトを模したデータ書き込みを常に行う．事前に RPO を設定し，設定値に従った頻度でデータが Amazon S3 にバックアップされる．

対象システムは，データベースが導入された VM1 台構成とし，データの書き込みを行うアプリケーションとして，EC サイトを模した TPC-C ベンチマークを利用した．実験前に TPC-C でシステム上に 1GB 分のデータを書き込み，フルバックアップの処理時間を測定した．測定の結果，2 分以内に処理が完了したため，RPO を 4 分間と設定することでバックアップ間隔以内に処理が完

了するようにした．また，取得した増分バックアップは転送前に lzma 形式で圧縮を掛け，圧縮した増分バックアップのデータ量が 5MB を超えた場合はフルバックアップを再取得し，過去に取得したバックアップは削除する．TPC-C とバックアップの処理を同時に開始してから 20 分後にシステムを停止させ，停止時間の計測と損失データ量を評価した．

対象システムに使用した VM の構成を表 4.4 に示す．対象システムが保有するデータは前節に比べて大きいため，異なる構成を使用した．他のホストの構成は WordPress が導入されたシステムでの実験と同様である．

4.2.2 停止時間の結果及び内訳

WordPress が導入されたシステムと同様に，サービスの停止時間を計測した．表 4.5 に評価結果を示す．

検知時間は，前節の実験と同様にクラウド環境に対する停止時のプロセスを含め 6 分台となった．扱われる運用データ量は，システム停止時の検知操作に影響を与えない．

再構築時間も同様，アプリケーションのインストールを事前に済ませた状態での VM テンプレートを各拠点に配置しているため，評価時の作業が VM の起動とアプリケーションやネットワークの設定作業に掛かる時間となる．

データの復元時間においては, 表 4.6 に示したファイルとデータ量を復元時に取得した．表内の増分バックアップファイルは，lzma 形式で圧縮されたものを展開した時のデータ量である．最新のバックアップと以降の増分バックアップのみを取得し，復元時間を 3 分台に抑えられ，再稼働までの時間短縮に寄与した．

表 4.4: 対象システムの VM の構成

CPU	2 Core @ 3.0GHz
RAM	4GB
HDD	40GB
OS	CentOS 7.2 (64bit)

表 4.5: 停止時間とその内訳

検知時間	6 分 15 秒
再構築時間	3 分 39 秒
データ復元時間	3 分 54 秒
停止時間	13 分 48 秒

表 4.6: 取得したバックアップとデータ量

ファイル名	データ量
フルバックアップ	1GB
増分バックアップ 1	0.9MB
増分バックアップ 2	1.8MB

4.2.3 データ損失量の結果

データの損失量を評価する尺度として，設定した RPO 値を活用する．評価実験で取得した最新のバックアップデータのタイムスタンプから，設定した RPO を満足しているかを確認した．評価時は RPO を 4 分間と設定していたため，システム停止時の時間から遡り，4 分以内のデータがバックアップされていれば RPO は満足される．表 4.6 の増分バックアップ 2 におけるタイムスタンプには，約 2 分前までのデータがバックアップされていることが確認され，評価実験で設定した RPO が達成された．

第5章　議論

5.1　複数サーバで構成されるシステムへの対応

本研究での評価は，VM1 台構成の情報システムを対象とした．一方，システムを構成する VM の台数が増えても，同様の仕組みで各サーバのデータを保護できる．また，クラウドオーケストレータで VM 複数台構成のシステムを構築する場合は，サーバ間の相互干渉がない操作は並列で実行するため，VM の台数やデータ量が増大するほど，自動化によるメリットは大きくなる．

5.2　3 拠点以上のクラウド環境の活用

システムを再構築する基盤の選択肢も重要である．評価実験では 2 拠点での評価を行ったが，クラウドオーケストレータによるサービス復旧では複数の異なるプラットフォームから再構築先を選択可能である．複数存在する拠点から管理者にとって最適な再構築先を自動で決定するために，性能や料金から算出されるコスト値の概念を導入する．これにより，拠点毎のコストと管理者が設定した許容コストの比較結果を，再構築先を選ぶ指標にできる．コストを勘案したクラウドプラットフォームの自動選択機能が導入されれば，意図しないサービス停止の復旧だけでなく，システムの成長に合わせて適切なクラウド基盤に乗り移る作業を自動化するといった応用も可能である．

5.3　より厳密に RPO を満足する仕組み

バックアップ処理の頻度とデータの生成速度の関係にもさらなる議論が必要である．本研究では，バックアップ回線速度がシステムのデータ生成量を上回る前提を置いているため，設定 RPO 値の 1/2 の間隔でバックアップを取得するように実装した．しかし，多くのアプリケーションは，時間帯に応じてデータ生成量が変動するため，サービスの繁忙期はデータ生成速度が回線速度を上回り，RPO を満たせなくなる恐れがある．また，提案した仕組みはバックアップファイル自体の取得時間や圧縮操作によるオーバーヘッドを勘案していない．

厳密に設定 RPO 値を満足するには，データ生成量とバックアップ回線速度の変化に応じてバックアップ間隔を自動調整する仕組みがあると良い．回線速度とシステムのローカルストレージに蓄積されたバックアップデータ量を比較し，回線速度が大きい場合には設定した頻度よりも早くバックアップを開始し，逆に蓄積されたデータ量が大きい場合は頻度を遅らせることで，生成されるデータ量が時時刻刻と変動する場合でも RPO を満足できる．

5.4　検知精度の向上による応用

評価実験では，システムの停止を通信路の断線で実現したが，本研究の提案手法はシステム停止と判断する閾値や停止を確認する手段を複数から柔軟に選択できる．停止と判断できる手段を

複数用意すれば，定常的なメンテナンスから人為的なミス，大規模災害時の復旧作業まで様々な停止要因に応じて同仕組みは利用可能である．

第6章　結論

本研究では，複数のクラウドを横断制御でき，システムを一括構築できるクラウドオーケストレーションの復旧機能に，データのバックアップと復元の仕組みの導入を提案した．２つのケーススタディで評価を行ったところ，WordPress を導入したシステムの場合は約 9 分，TPC-C による EC サイトを模したシステムの場合は約 13 分でデータの復元も含めた迅速な再稼働を確認した．また EC サイトを模した評価実験においては，事前に設定した RPO を満足し，復元可能なデータを保証できることを確認した．E コマーズからブログシステムまで，データの重要度は情報システムの性質により異なるため，自由に RPO を設定できる本研究の仕組みの利用範囲は広い．

今後は時間帯に応じてバックアップすべきデータ量が時間変動する場合に着目し，蓄積されたデータ量と回線速度からバックアップ間隔を動的に設定する仕組みを導入することで，より厳密に RPO を満足する機構を開発する．

謝辞

本研究を進めるにあたり，熱心なご指導と的確なご助言を頂きました，吉永努教授に感謝の意を表します．また，日常の議論を通じて研究面・技術面ともに多大なる知識や示唆を頂いた吉見真聡助教に，感謝致します．最後に，研究生活を通じて様々な指摘，協力を下さいました吉永・吉見研究室，策力木格研究室の皆様に，厚く御礼申し上げます．

参考文献

[1] Amazon Web Service. https://aws.amazon.com.

[2] Google Cloud Platform. https://cloud.google.com.

[3] Microsoft Azure. https://azure.microsoft.com.

[4] Amazon Web Service. Summary of the AWS Service Event in the US East Region. https://aws.amazon.com/message/67457.

[5] Mohammad Ali Khoshkholghi, Azizol Abdullah, Rohaya Latip, Shamala Subramaniam, and Mohamed Othman. Disaster recovery in cloud computing: A survey. *Computer and Information Science*, Vol. 7, No. 4, p. 39, 2014.

[6] CloudConductor. デザイン指向クラウドオーケストレータ. http://cloudconductor.org.

[7] PrimeCloud Controller. オープンソース版 primecloud controller - ハイブリッドクラウド対応クラウドコントローラ. http://www.primecloud-controller.org.

[8] Shubhashis Sengupta and KM Annervaz. Multi-site data distribution for disaster recovery—A planning framewrk. *Future Generation Computer Systems*, Vol. 41, pp. 53–64, 2014.

[9] Yu Gu, Dongsheng Wang, and Chuanyi Liu. DR-Cloud: Multi-cloud based disaster recovery service. *Tsinghua Science and Technology*, Vol. 19, No. 1, pp. 13–23, 2014.

[10] 柏崎礼生, 北口善明, 近堂徹, 楠田友彦, 大沼善朗, 中川郁夫, 阿部俊二, 横山重俊, 下條真司. 広域分散仮想化環境のための分散ストレージシステムの提案と評価. 情報処理学会論文誌, Vol. 55, No. 3, pp. 1140–1150, 2014.

[11] 田口雄一, 山本政行. ディザスタリカバリに向けた非同期リモートコピー構成資源算出方式. 情報処理学会論文誌コンピューティングシステム（ACS）, Vol. 7, No. 2, pp. 1–10, 5 2014.

[12] 波戸邦夫, 上水流由香, 岡本隆史, 横山大作ほか. 複数の異種クラウド間におけるスケールアウトおよびディザスタリカバリ機構の実装とその評価. デジタルプラクティス, Vol. 4, No. 4, pp. 314–322, 2013.

[13] Alexander Lenk. Cloud Standby Deployment: A Model-Driven Deployment Method for Disaster Recovery in the Cloud. In *Proceedings of 2015 IEEE 8th International Conference on Cloud Computing*, pp. 933–940, 2015.

[14] Amazon S3 (スケーラブルなクラウドストレージサービス) — AWS. https://aws.amazon.com/s3.

付 録A　CloudConductorによるシステム構築

A.1　CloudConductorの概要

CloudConductorは，TIS株式会社が独自で開発を行っているデザイン指向クラウドオーケストレーションソフトウェアである．Chef，Zabbix等多数のOSS技術を取り込み，システム構成の記述を「パターン」として抽象化することで，複数のサーバ群からなる情報システムを一括構築できる．現在ではAmazon Web Service，OpenStackの2種類のクラウド環境に対応し，各プラットフォームを同一の操作で制御できる．また，CloudConductorをコマンドラインで操作するためのCLIやWebUIで操作するGUI環境も合わせて用意されており，OSSとしてgithubで公開されている．

A.2　CloudConductorによるシステム構築の流れ

本付録では，図A.2.1に示すように計算ホスト1台にOpenStack環境を導入し，その上でCloudConductorをインストールしたVMを用意してシステム構築までを行う．OpenStackは，UbuntuとCentOS両方に対応していが，ここではCentOS 7.2に自動構築ツールであるPackStackを使用してOpenStackの導入を行う．また，CloudConductorが推奨する環境はCentOSであるため，CloudConductorのインストール作業を行うVMのOSはCentOS 6.7を使用する．構築するシステムはgithubで公開されている，Webサーバ，アプリケーションサーバ，データベースサーバのVM3台構成であるTomcatパターンを使用する．

A.3　PackStackによるOpenStackのインストール

PackStackとは，OpenStackを構成する多数のコンポーネント各々の導入を実施するか否かを1つのAnswerファイルに集約することで，OpenStackのインストール作業を簡略化できるツールである．本付録で用意するクラウド環境は1台の計算ホストに必要なコンポーネント全てを導入するため，All-in-One構成によるインストールを行う．

OpenStackのインストール時は，事前にOSを最新の状態に更新し，SELinuxとNetworkManagerを無効にする操作を行っておく．CloudConductorのバージョン2.0が対応しているOpenStackのリリース名はLibertyであるため，計算ホスト上で以下のコマンドを実行することでLiberty導入用のPackStackをインストールし，OpenStackのインストール作業に必要なAnswerファイルであるanswer.txtを生成する．

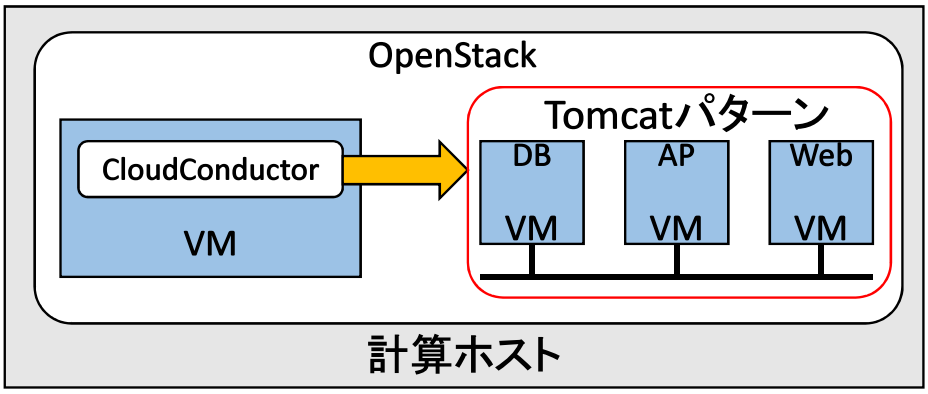

図 A.2.1: 構築するクラウド環境

```
$ yum install -y https://rdoproject.org/repos/openstack/\
openstack-liberty/rdo-release-liberty-5.noarch.rpm
$ yum install -y openstack-packstack
$ packstack --gen-answer-file answer.txt
```

CloudConductor でシステムを構築するためには，システムの自動構築コンポーネントである Heat が必要になる．しかし，PackStack によるデフォルトのインストールでは Heat は導入されないため，OpenStack のインストール作業前に answer.txt 内を以下のように書き換えておく．

```
CONFIG_HEAT_INSTALL=y
CONFIG_HEAT_CFN_INSTALL=y
```

その後以下のコマンドを実行することで，計算ホストに OpenStack が導入される．

```
$ packstack --answer-file=answer.txt
```

A.4　CloudConductor 及び CLI ツールのインストール

OpenStack 上に構築した VM に CloudConductor と専用の CLI を導入する．事前に CentOS のベースイメージを OpenStack に登録し，登録したベースイメージから CloudConductor 用の VM を 1 台構築する．また OpenStack 上では，VM が外部と通信できるようにネットワークの設定を行っておく．

以下のコマンドで CloudConductor を導入する VM の OS を最新の状態に更新し，必要なパッケーズをインストールする．下記で上げているのパッケージ以外にもバージョン 2.1.2 以上の ruby が必要になるため，適宜導入する．

```
$ sudo yum -y update
$ sudo yum -y install git wget unzip gcc gcc-c++ make \
patch openssl-devel libxslt-devel libxml2-devel sqlite-devel
```

CloudConductor は多数の自動化ツールで構成されており，インストール時はいくつかのツールも同時に必要となる．システム構築に必要な VM テンプレートとインフラに必要な Packer，Terraform を以下のコマンドでそれぞれ導入する．

```
$ sudo mkdir /opt/packer
$ wget https://releases.hashicorp.com/packer/ \
0.9.0/packer_0.9.0_linux_amd64.zip
$ sudo unzip packer_0.9.0_linux_amd64.zip -d /opt/packer

$ sudo mkdir /opt/terraform
$ wget https://releases.hashicorp.com/terraform/ \
0.6.14/terraform_0.6.14_linux_amd64.zip
$ sudo unzip terraform_0.6.14_linux_amd64.zip -d /opt/terraform
```

その後，CloudConductor 用のデータベースをインストールし，初期化を行う．また，サービスを起動する前に認証方式を md5 に設定しておく．

```
$ sudo yum install -y http://yum.postgresql.org/9.4/redhat/ \
rhel-7-x86_64/pgdg-redhat94-9.4-1.noarch.rpm
$ sudo yum install -y postgresql94-server \
postgresql94-contrib postgresql94-devel
$ sudo /usr/pgsql-9.4/bin/postgresql94-setup initdb
$ sudo vi /var/lib/pgsql/9.4/data/pg_hba.conf
host all all 127.0.0.1/32 md5
host all all ::1/128 md5
```

初期化が完了すれば，データベースを起動してアカウントを発行する．

```
$ sudo systemctl start postgresql-9.4.service
$ sudo systemctl enable postgresql-9.4.service
$ export PATH=$PATH:/usr/pgsql-9.4/bin
$ sudo -u postgres createuser --createdb --encrypted\
--pwprompt [database_username]
Enter password for new role: [database_password]
Enter it again: [database_password]
```

インストールに必要なツールはすべて導入したため，以下の操作により CloudConductor をインストールする．

```
$ git clone https://github.com/cloudconductor/cloud_conductor.git
$ cd cloud_conductor
$ bundle install
$ git submodule update --init
```

CloudConductor を起動する前に secret_key_base の登録を行う．また，データベースに専用のテーブルを登録し，関連付けも行う．構築したシステムにドメイン名を割り当てる DNS サーバの指定など，他のオプション設定は起動前に適宜行っておく．

```
$ secret_key_base=$(bundle exec rake secret)
$ sed -i -e "s/secret_key_base: .*/secret_key_base: ${secret_key_base}/g" \
config/secrets.yml
$ sed -i -e
"s/# config.secret_key = '.*'/config.secret_key = '${secret_key_base}'/" \
config/initializers/devise.rb
$ bundle exec rake db:create RAILS_ENV=production
$ bundle exec rake db:migrate RAILS_ENV=production
$ cp config/database.yml.smp config/database.yml
$ vi config/database.yml
# username: <%= ENV['USER'] %>
# password: <%= ENV['USER'] %>
username: [database_username]
password: [database_password]
```

設定が完了したら以下のコマンドで CloudConductor の管理者アカウントを発行し，プロセスを起動する．

```
$ bundle exec rake register:admin RAILS_ENV=production
  Input administrator account information.
    Email: [your_email_address]
    Name: [user_name]
    Password: [password]
    Password Confirmation: [password]
  Administrator account registered to development environment successfully
$ bundle exec unicorn -c config/unicorn.rb -E production -D
```

これにより，CloudConductor に備わる API を実行することでクラウド上にシステム構築を行える．しかし，API から CloudConductor を制御できるものの，システムを構築するまでは複雑な入力パラメータを与えなければならないため，操作を簡略化するために CLI ツールの導入も行う．

```
$ cd ~/
$ git clone https://github.com/cloudconductor/cloud_conductor_cli
$ cd cloud_conductor_cli
$ bundle install
$ bundle exec rake install
```

CloudConductor 本体と関連付けるために複数の環境変数の設定し，CLI を使えるようにする．

```
$ vi ~/.bashrc
export CC_HOST=localhost
export CC_PORT=8080
export CC_AUTH_ID=[your_email_address]
export CC_AUTH_PASSWORD=[password]
$ source ~/.bashrc
```

A.5 CloudConductor によるシステム構築

前節で導入した CLI を使って CloudConductor を操作し，VM3 台構成のシステムをクラウド上に構築する．以下の 5 つの手順を行うことでシステム構築を行う．

- (1) 使用するクラウド環境を設定
- (2)VM テンプレート構築に使用するベースイメージを指定
- (3) システム構成が記述されたパターンの登録
- (4) アプリケーションのインストールが行われた VM テンプレートを配置
- (5)VM のテンプレートからのシステム構築

上記 5 ステップを行う前に CloudConductor の資源管理の単位であるプロジェクトを登録する．CloudConductor はプロジェクト単位でユーザの権限や使用するクラウド環境を設定する．以下の CLI が提供するコマンドを実行し，プロジェクトを生成する．

```
$ cc-cli project create --name sample_project --description \
"sample_project_description"
```

次に (1) の操作である，システム構築先のクラウド環境の登録を行う．登録を行う際は，OpenStack を導入した計算ホストの IP アドレス，OpenStack のユーザ名とパスワード，ユーザが所属するテナント名を指定する．

```
$ cc-cli cloud create --project sample_project --name openstack \
--type openstack --entry-point "[your-openstack-entry-point]" \
--tenant-name "[your-openstack-project-name]" --key "[your-tenant-user]" \
--secret "[your-tenant-password]"
```

クラウド環境が OpenStack の場合，事前に構築するシステム用のベースイメージを各クラウド環境に登録する必要がある．登録後，下記のコマンドにより CloudConductor がシステム構築を行う時に使用するベースイメージを指定する．

```
$ cc-cli base_image create --cloud "[cloud-name]" \
--source_image [registerd-image-id] --ssh_username "centos" \
--platform centos
```

その後，システムの設計図であるパターンを CloudConductor に登録する．CloudConductor のパターンは platform と optional の 2 種類があり，各々に役割がある．platform パターンはシステムを構成するインフラの骨格が記述されたものであり，単体でシステム構築を行える．一方，optional パターンはシステムの非機能要件であるサービスの監視などが記述されており，単体では用いられず必要な時に platform パターンに後付して使用する．よって，システム構築時は管理者の要求に応じて，1 つの platform パターンと複数の optional パターンを組み合わせる．本付録では，github で公開されている Tomcat パターンをシステムの骨格として使用する．以下のコマンドで CloudConductor に Tomcat パターン登録する．

```
$ cc-cli pattern create --project sample_project \
--url https://github.com/cloudconductor-patterns/tomcat_pattern \
--revision master
```

また，Terraform でシステムを構築する場合は，同じく github で公開されている optional パターンである common_network パターンを登録する．

```
$ cc-cli pattern create --project sample_project \
--url https://github.com/cloudconductor-patterns/common_network_pattern \
 --revision master
```

CloudConductor に登録した複数のパターンから blueprint(青写真) を生成する．この blueprint はシステムを構築する際の VM のテンプレートであり，事前に指定しておいたベースイメージを使って生成を行う．以下のコマンドを実行して blueprint を生成し，使用する Tomcat パターンと common_network パターンを割り当てる．

```
$ cc-cli blueprint create --project sample_project \
--name tomcat_blueprint --description "tomcat_pattern_description"
$ cc-cli blueprint pattern-add tomcat_blueprint \
--pattern tomcat_pattern --revision master --platform centos
$ cc-cli blueprint pattern-add tomcat_blueprint \
--pattern _pattern --revision master --platform centos
```

使用するパターンをすべて割り当てたら，以下のコマンドで事前に登録していたクラウド環境に VM のテンプレートを生成する．生成が完了するまでに 10 分程度の時間を要し．生成の成功可否は CloudConductor のログデータから確認できる．

```
$ cc-cli blueprint build tomcat_blueprint
```

その後，システムに動作させるサービスを割り当てるために，system の登録を行う．また，構築したシステムにドメイン名を割り当てる場合は，オプションで指定する．

```
$ cc-cli system create --project sample_project --name tomcat_system \
--description ‘‘tomcat_system_description’’ \
--domain [your-domain-name (optional)] \
```

以上の操作を踏まえた上で，下記のコマンドを実行することにより VM のテンプレートからシステムを構築する．パラメータには使用する blueprint 及び system を割り当てる．さらに Terraform を導入している場合は，構築するネットワークの名前やセキュリティグループ，各 VM のスペックなどをコマンド実行時に設定する．システムのインフラ構築も数分程度の時間が必要になる．

```
$ cc-cli environment create --system tomcat_system \
--name tomcat_environment --description "tomcat_environment_description" \
--blueprint tomcat_blueprint --clouds [your use cloud name] \
--version [blueprint version number]
```

構築完了後は，ブラウザで web サーバに割り当てられたアドレスにアクセスすることで，システムを操作できる．構築したシステムに運用させるアプリケーションは，適宜パターンに導入スクリプトを追加することで追加可能である．

発表論文

[1] 溝田敦也，城間 隆行, 中島 拓真, 吉見 真聡, 入江 英嗣, 吉永 努 “インタークラウドを活用した自動災害復旧システム” 信学技報, vol. 116, no. 177, pp. 229-234, Aug. 2016.