Vol.2014-OS-128 No.11
2014/3/7

データセンタ間通信による性能低下を抑えたキーバリューストア構築手法

堀江 光[1]　浅原 理人[2]　山田 浩史[3]　河野 健二[1,4]

概要：近年，複数のデータセンタ間を跨いで提供されるクラウド環境が登場している．これは各データセンタで稼働する計算資源を集約することで，クラウド環境の特徴である伸縮性や可用性の向上を図っている．スケーラビリティの観点から，このような多数のサーバを用いた環境での利用に適したストレージのひとつとして分散型キーバリューストアがあるが，データセンタ間を結ぶ狭帯域・高遅延なネットワークにより性能が著しく低下する問題がある．この問題を解決するために，本研究では *Local-first Data Rebuilding* (*LDR*)，*Multi-Layered Distributed Hash Table* (*ML-DHT*) という 2 つの手法を用いてキーバリューストアを構築する方法を提案する．LDR は保存データに冗長性を与えた上で分割することで，ストレージ使用量の増加を抑えつつデータセンタ間通信の量を軽減する．ML-DHT はデータセンタ間通信を最小限に抑えたルーティングにより，通信遅延を抑えたデータ探索を実現する．実験により提案手法が，代表的な DHT である Chord を用いた手法を用いた場合と比較し，データ探索のための通信遅延を約 74 % 軽減し，ストレージ使用量とデータセンタ間通信量のバランスを柔軟に設定できることを示した．

キーワード：分散キーバリューストア，分散ハッシュ表，データセンタ間連携

1. 背景

インターネット上で展開されるサービスの基盤として複数のデータセンタを跨いだクラウド環境が登場している．これは単一のデータセンタではサービスの要求性能を満たさない場合があるためである [1]．このように単一のデータセンタで稼働している複数のクラウド環境の資源を集約することで，単一のクラウド環境と比較してより柔軟な資源管理が可能となる．例えば，クラウド環境上で運用されているサービスの利用状況等に応じて，使用するサーバインスタンスの数／場所／時機を適切に選択することが可能となる．

このようなデータセンタ間を跨ぐクラウド環境に適したストレージのひとつとして **分散キーバリューストア** [2–8] がある．分散キーバリューストアは高いスケーラビリティを提供すると共に，データの物理的な配置を抽象化して提供する．一般に，分散キーバリューストアは複数のストレージサーバ（ノード）から構成され，その台数を増加させることで容易に全体の容量や I/O スループットを向上させることが可能である．また，分散キーバリューストアに保存されたデータを利用するアプリケーションは，データが実際にどのノードに保存されているかといった情報を知る必要は無い．分散キーバリューストアはこのような特徴を，図 1 に示すように 4 層からなるソフトウェアスタックによって実現している．キーバリューストア API 層は，アプリケーションが put/get 命令を使用するための汎用的な API を提供する．データフェッチ層は，オリジナルのデータとそれを構成するチャンクと呼ばれるデータ断片とを相互に変換する．ルーティング層は，要求されたチャンクがローカルストレージに存在しない場合に実際にチャンクを保持しているノードの探索を行う．ストレージ層は，チャンクをローカルストレージに保存する．

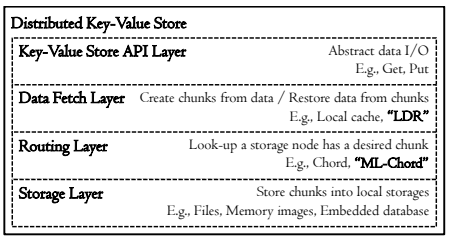


図 1　分散キーバリューストアを構成するソフトウェアスタック

従来の分散キーバリューストアは **データセンタ内** での利用を想定して設計されており，高遅延かつ狭帯域である

[1] 慶應義塾大学
[2] NEC グリーンプラットフォーム研究所
[3] 東京農工大学
[4] JST CREST

データセンタ間 の通信は考慮されていない．分散キーバリューストアを構成する各ノードは，ノードの状態の通知，リクエストされたデータの探索，データの送受信のため，相互に通信を行う．このような通信が複数のデータセンタ間を跨いで行われる場合，一般的にインターネットはデータセンタ内ネットワークと比較して高遅延／狭帯域であるため，分散キーバリューストアの応答遅延とスループットは著しく低下する．すなわち，データセンタ間を跨いで構築された分散キーバリューストアは，それを利用するアプリケーションの応答性を顕著に低下させる場合がある．このような性能低下は分散キーバリューストアの利点を阻害する．

本研究では，複数のデータセンタ間を跨ぐ環境における分散キーバリューストア構築手法の実現を目的とする．本論文では，複数のデータセンタ間を跨ぐ分散キーバリューストアを構築するための要素技術として *multi-layered DHT* (*ML-DHT*) 及び *local-first data rebuilding* (*LDR*) という 2 つの手法を提案する．図 1 で示したように，LDR 及び ML-DHT はそれぞれデータフェッチ層及びルーティング層で動作する．これらの手法はデータセンタ間通信の機会を軽減し，ストレージ容量とパフォーマンスの低下というトレードオフの関係をより柔軟に設定することを可能とする．

ML-DHT はデータセンタによらず全てのノードとデータを一律なキー空間上で取り扱うことで効率的なストレージ容量の拡張を実現する．一律のキー空間を用いることで，インデックス管理をする特別なサーバ等を用いずに，全てのデータセンタに属する全てのノードが目的のデータを探索することが可能となる．また，ML-DHT はデータセンタ間を跨ぐ通信を必要最低限に抑え探索を行うよう設計されており，データセンタ間通信による探索の応答性低下を避ける．ML-DHT を用いて構築された分散キーバリューストアは，データセンタ毎にキー空間を階層構造で分割するシンプルな方法 [9] と比較して効率的に容量を拡張することができる．このような階層型キー空間を用いた場合，各ノードが担当するキー空間を均等に保つにはノードが加入／離脱する度に各データセンタに割り当てたキー空間を再割当てする必要があり効率的な管理が難しい．また，各データセンタ内でのみキー空間管理を行う場合，あるデータセンタにおけるノードの加入が他のデータセンタの負荷分散には貢献しないため，分散キーバリューストア全体の性能向上に繋がりづらい．ML-DHT の一律なキー空間は階層型キー空間におけるこのような問題を避けることが可能である．ML-DHT については 3.2 章で詳しく述べる．

LDR はオリジナルデータとチャンクを変換する際に Erasure Coding [10–12] を用い冗長性を持たせることで，データセンタ間の通信量を低減させる．LDR では，オリジナルのデータ全てをリモートのデータセンタから取得する代わりに，ローカルに存在する一部のチャンクを優先利用しデータを復元する．また LDR は，トレードオフの関係であるストレージ容量の拡張性とデータセンタ間通信量の削減を，クラウド環境の管理者が自由に設定できるようにする．あるデータから生成するチャンクの数が多い場合，ローカルのデータセンタ内にてチャンクを取得しやすくなるためデータセンタ間通信の削減が期待できる．一方，チャンクの数が少ない場合，冗長なデータが減少するためストレージの利用効率が向上するが，データセンタ間通信が増加する可能性が高まる．このように LDR は，単純にデータレプリケーションを行う場合と比較して，ストレージの使用量とデータセンタ間の転送量を任意に設定することを可能とする．

提案手法の有用性を示すために，オーバレイ構築ツールキット OverlayWeaver [13] を用い実装及び評価実験を行った．実験にて，提案手法が，Chord [14] を用いたシステムと比較して応答遅延が 74 % 向上し，ストレージの使用量とデータ転送量を様々に設定できることを確認した．

本論文の構成は以下の通りである．2 章では分散キーバリューストアの設計において考慮すべき事項を整理する．3 章と 4 章ではそれぞれ提案手法の設計と実装について述べる．5 章ではプロトタイプを用いて提案手法の有用性について評価する．6 章で関連研究について整理し，7 章で本論文をまとめる．

2. 設計上の課題

複数のデータセンタのノードから構成される分散キーバリューストアは，単一のデータセンタ内で運用される分散キーバリューストアと比較してストレージ容量の拡張性が高い．単一のデータセンタ内で運用される分散キーバリューストアと異なり，複数のデータセンタからなる分散キーバリューストアはそれぞれのデータセンタについて利用する資源の量を選択できるためよりストレージの伸縮性が高い．図 2 に複数のデータセンタ間を跨ぐ分散キーバリューストアの概要を示す．複数のデータセンタから構成される分散キーバリューストアも単一のデータセンタ内のノードのみで構成される分散キーバリューストアと同様に，サービスを停止することなくノードの加入／離脱を行うことができる．これはキー空間が Consistent Hashing [15] を用いて構築されているためである．Consistent Hashing を用いると，新たにノードが加入／離脱する度に，全てのノードの担当キー空間の再割当てを伴わずに済む．本論文では分散ハッシュ表を用いる分散キーバリューストア [14,16–19] について扱う．このような分散キーバリューストアでは，属するデータセンタによらず全てのノードが，互いに通信を行うことで目的のデータを探索することが可能である．複数のデータセンタから構成されるキーバリューストアでは，使用できる資源も多くなるため，ス

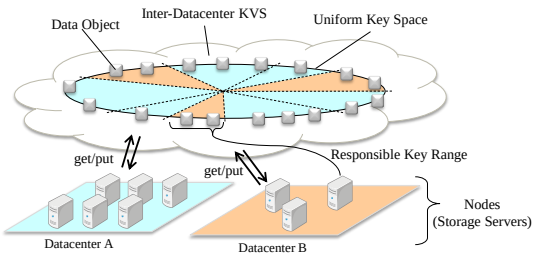


図 2　一律のキー空間を持つ分散キーバリューストア

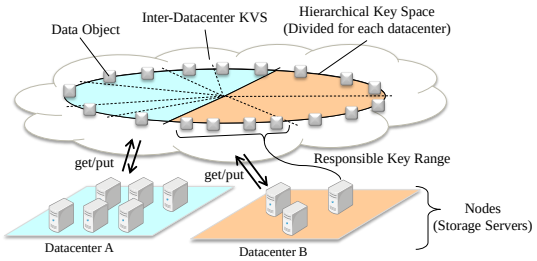


図 3　階層型のキー空間を持つ分散キーバリューストア

ケーラビリティが高い．

本章では複数のデータセンタから構成される分散キーバリューストアのデザインについて，3 つ観点から議論する．

2.1　ストレージ性能の均一な拡張性

複数のデータセンタから構成される分散キーバリューストアはストレージ性能を均一に拡張できる必要がある．なぜなら，ノード追加に応じてキーバリューストアの性能が向上しない場合，資源の利用効率を低減させスケーラビリティを妨げるためである．

単一のデータセンタかから構成される分散キーバリューストアでは，ノードの追加によりキーバリューストア全体の性能が均一に拡張できる．新たに追加されたノードにはキー空間の一部が割り当てられ，該当するキー空間に対応するデータを元々の担当ノードから移譲する．図 2 に示すようにキー空間が均一に割り当てられるため，各ノードはキーバリューストアの性能向上に均等に寄与する．

一方で複数のデータセンタから構成される分散キーバリューストアではこのような均一な拡張性を得られない場合がある．例えば，階層型のキー空間を用いた分散キーバリューストアは均一な拡張性を得ることができない．図 3 に，階層型のキー空間を用いた分散キーバリューストアの例を示す．この場合，各データセンタにキー空間の半分がそれぞれ割り当てられ，各キー空間は更に各データセンタに属するのノードに均一に割り当てられる．例えば，負荷分散のため新たにノードを追加しても，そのノードが属するデータセンタの負荷が軽減されるのみであり，分散キーバリューストア全体の性能向上に寄与することができない．このように階層型のキー空間は，単一のデータセンタから構成される分散キーバリューストアのような均一な拡張性を得ることができない．

2.2　データ探索時の応答遅延の低減

複数のデータセンタからなる分散キーバリューストアでは，データセンタ間通信が高遅延であるために，データ探索時の応答遅延の増加が問題となる．なぜなら，このようなデータ探索時間の増大は，この分散キーバリューストアを利用するアプリケーションの応答時間の増大に繋がり，サービスの品質を低下させるためである．

図 5 (a) は非効率的なデータ探索の例を示す．探索を開始するノードと目的のノードはそれぞれ異なるデータセンタに属している．ルーティング経路は，通信遅延の大きなデータセンタ間ホップを 3 回含んでいる．このようなデータ探索の応答遅延の増大を避けるために，複数のデータセンタから構成される分散キーバリューストアには不必要なデータセンタ間通信を避けるための仕組みが必要である．

2.3　狭帯域通信を介したデータ転送量の低減

一般的にデータセンタ間の通信路はデータセンタ内の通信路と比較して狭帯域であるため，データをリモートのデータセンタから取得する場合アプリケーションの動作が遅くなる．したがって，このようなデータセンタ間の遅いデータ転送を避けるため，複数のデータセンタから構成される分散キーバリューストアにはデータセンタ間のデータ転送量を減少させる仕組みが必要である．

データ転送量を減らすためのアプローチのひとつとして，別のデータセンタに存在するデータオブジェクトをローカルに複製して保持する方法がある．これにより，そのデータを必要とするノードは，リモートのデータセンタから取得するのではなく，同じデータセンタ内に保持されているレプリカを取得することができる．しかし，このアプローチは全てのデータオブジェクトを各データセンタに複製するため大量のストレージ資源を消費する．n 箇所のデータセンタにて分散キーバリューストアが稼働する場合，本来保存されるデータのサイズの n 倍のストレージを消費する．

ストレージ使用量とデータ転送量のバランスを取るために，分散キーバリューストアにはより少ないストレージ使用量でデータ転送量も軽減する仕組みが必要である．

本研究では，一般的な分散キーバリューストアの操作における WAN 通信の量を減少させることに焦点を合わせている．すなわち，ノードの障害や復旧に伴って発生する WAN 通信の量を減少させることを目的とはしていない．このような WAN 通信の量を減少させることも興味深い事案であるが，本論文では対象としない．

3.　提案

データセンタ間を跨ぐ通信による問題を解決するために，

本論文では multi-layered DHT (ML-DHT) と local-first data rebuilding (LDR) の 2 つの要素技術を提案する．

3.1 概要

ML-DHT と LDR は高遅延／狭帯域であるデータセンタ間通信の影響を低減する．本手法を適用した分散キーバリューストアを利用するアプリケーションは，それが単一のデータセンタのノードのみで構成されているのか複数のデータセンタのノードから構成されているのかを意識する必要は無い．アプリケーションが利用するデータの保存場所は抽象化されており，また，性能低下に繋がるリモートのデータセンタに属するノードとの通信ができるだけ発生しない仕組みを備えているためである．ML-DHT はデータ探索時の応答遅延の増大を抑制する仕組みを備える．これはデータ探索時に冗長なデータセンタ間ホップが発生することを許さない．LDR は狭帯域であるデータセンタ間通信路を用いたデータ転送の量を低減させる仕組みを備える．これはキーバリューストアに対する読み書に伴うデータセンタ間の通信を抑制する．3.2 章と 3.3 章にて，ML-DHT と LDR についてそれぞれ説明する．

また，ML-DHT はストレージの均一な拡張性の実現にも貢献する．階層型のキー空間と異なり，ML-DHT は一律のキー空間を提供する（図 2）．すなわち，キー空間はデータセンタ毎に分割されることなく，全てのノードが同一の扱いを受ける．ML-DHT における新たなノードの追加は，分散キーバリューストア全体の性能向上に繋がる．例えば，図 2 においてデータセンタ A にノードが追加された場合，そのノードは一部のキー空間の割り当てを受け，隣接ノードから対象のデータを受け取る．もし隣接ノードがデータセンタ A 以外のデータセンタ X であった場合，データセンタ A に追加されたこのノードによってデータセンタ X のノードが負荷分散の恩恵を受けることとなる．この際，データセンタ間の通信が発生しているが，これは頻繁に発生することは無いと想定する．なぜなら，データセンタで運用される分散キーバリューストアにおいては，一般ユーザの PC から構成される Peer-to-Peer 型システムと異なり，ノードの加入／離脱の頻度が低いと考えられるためである．

3.2 Multi-Layered DHT

Multi-layered DHT (ML-DHT) はデータセンタ間通信の頻度を減らす仕組みを備えた分散ハッシュ表の仕様拡張である．通常の分散ハッシュ表と異なり，ML-DHT では各ノードが **グローバル経路表** と **ローカル経路表** の 2 つの経路表を持つ．グローバル経路表は元となる分散ハッシュ表と同様のアルゴリズムで構築される．また，グローバル経路表は，全てのデータについて到達性と可用性を元となる分散ハッシュ表と完全に同等の水準で保証する．一方，ローカル経路表はグローバル経路表と異なり，同じデータセンタに属するノードのみを対象として構築される．ローカル経路表はデータセンタ間ホップが冗長に発生することを避けるために利用される．ML-DHT へのノードの加入，及び，データ探索は以下の手順で行う．

加入処理 *(Join):* ML-DHT において，加入するノードはグローバル経路表とローカル経路表双方に加入処理を行う．加入処理に伴うグローバル経路表の更新手順は元の分散ハッシュ表と完全に同様であるが，ローカル経路表は以下の特別な手順で更新される．まず，加入するノードはブートストラップノードをローカルのデータセンタから選定し，加入要求を送信する．次に，そのブートストラップノードは，加入ノードが新たに担当となるキー空間を現在担当しているノードをローカル経路表のみを用いて特定する．続いて，加入ノードに現在の担当ノードの情報を送信する．最後に，加入ノードは現在の担当ノードより，元の分散ハッシュ表と同様の方法で，新たに担当することになるキー空間とそこに含まれるデータを受け取り，加入処理が完了する．もし，ローカルのデータセンタにブートストラップノードが存在しない場合，加入ノードは自身のみが存在するものとしてローカル経路表を構築する．

ML-Chord は Chord [14] を拡張した ML-DHT で，グローバルフィンガーテーブルとローカルフィンガーテーブルを持つ．ML-Chord に加入するには，加入ノードはブートストラップノードに加入要求を送信する．加入ノード／ブートストラップノードは，通常の Chord と同様のアルゴリズムでグローバルフィンガーテーブルをそれぞれ初期化／更新する．双方が同じデータセンタに属する場合，ローカルフィンガーテーブルに含まれる情報を用いて，それぞれのローカルフィンガーテーブルを初期化／更新する．一方，双方がそれぞれ異なるデータセンタに属する場合，加入ノードはローカルフィンガーテーブルを自身のみ存在するものとして初期化し，ブートストラップノードはローカルフィンガーテーブルに対して何も処理を行わない．

探索処理*:* 各ノードはデータを探索する際にローカル経路表を優先的に利用する．探索処理を開始する前に，起点ノードは自身が対象データを保持していないかを確認する．もし保持していない場合，探索処理が実行される．ローカル経路表にて対象データの担当と想定されるノードを発見した場合，グローバル経路表を参照し実際に該当するかを確認する．ML-DHT ではローカル経路表から他のデータセンタのノードを除外しているためこのような確認が必要であるが，各ノードが保持しているグローバル経路表を用いるのみで通信は発生しないため処理のオーバヘッドは小さい．ローカル経路表より発見したノートが実際の担当ノードであった場合，そのノードにクエリを転送する．

ローカル経路表より発見したノードが担当ではなかった場合は，不必要なデータセンタ間通信の発生を抑えるた

```
#define GLOBAL_LAYER 0
#define LOCAL_LAYER 1

// node n has finger table for each layer
n.finger_tables = {global_finger_table, local_finger_table};

// ask node n to find id's successor
n.find_successor(id)
    n' = find_predecessor(id);
    return n'.successors;

// ask node n to find id's predecessor
n.find_predecessor(id)
    n' = n;
    while (id not in (n', n'.successor])
        n' = n'.closest_preceding_finger(id);
    return n';

// return closest finger preceding id
n.closest_preceding_finger(id)
    for l = LOCAL_LAYER downto GLOBAL_LAYER
        for i = m - 1 downto 0 // m is # of finger table's entries
            if (finger_tables[l][i].node in (n, id))
                return finger_tables[l][i].node;
    return n;
```

図 4　ML-Chord におけるキー *id* 探索処理の擬似コード

め，可能な限りローカル経路表を用いローカルのデータセンタ内でクエリの転送を行う．担当ノードをローカルのデータセンタ内におけるルーティングで極力発見するために，クエリが対象とするキーに漸近するようにローカル経路表のみを用い，クエリの転送を行う．もし担当ノードを発見できないまま，それ以上対象のキーに近いノードが存在しない場合は，グローバル経路表を参照して次に対象のキーに近いノードへクエリの転送を行う．何故ならば，対象のキーに近いノードが他に存在しないということは，そのデータセンタ内に対象のキーを担当するノードが存在しないことを意味するためである．クエリを受け取ったノードは，可能な限りローカル経路表を用いた同様の手順で担当ノードを探索する．

図 4 に ML-Chord における探索処理の擬似コードを示す．各ノードはクエリの転送先，すなわち次のホップ先を選択する際にローカルフィンガーテーブルを優先的に利用する．各ノードは，次のホップ先として適切なノードをローカルフィンガーテーブルから発見できない場合のみ，グローバルフィンガーテーブルを用いる．

図 5(a), 5(b) は Chord 型の分散ハッシュ表と ML-DHT におけるルーティングの例の比較である．各例とも，ノードの構成は同様で，ノード 1, 5, 10 及び 12 はデータセンタ A，残りはデータセンタ B で稼働している．また，各例とも，データセンタ A に属する ノード 1 がデータセンタ B に属するノード 15 が保持するデータ 14 を取得する場合を図示している．すなわち，最低でも 1 度はデータセンタ間を跨ぐホップが必要である．Chord 型の分散ハッシュ表では，3 つのホップ全てがデータセンタ間を跨いでおり，探索に伴う応答遅延が大きくなっている．このような無駄なホップが発生しているのは，経路表が各ノードの位置（ローカルまたはリモート）を考慮せずに次のノードを示すためである．一方，ML-DHT では 3 つのホップのうち最後の 1 つのみがデータセンタ間を跨いでいる．これは，ローカル経路表がデータセンタ間ホップの実行を可能な限り先送りする役割を果たしているためである．図 5(b) において各ノードは以下の手順で次のノードを選定している．

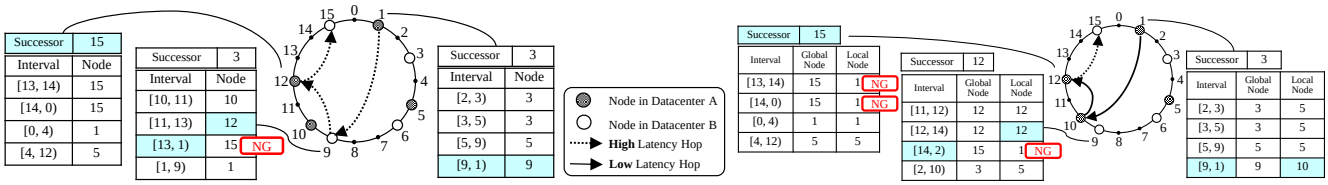
(1) 次のノード候補として，ノード 1 のローカル経路表はノード 10 を，グローバル経路表はノード 9 を示している．ノード 10 の方がより目的のノードに近いため，ノード 1 はノード 10 を選択する．
(2) 次のノード候補として，ノード 10 のローカル経路表はノード 1 を，グローバル経路表はノード 15 を示している．ノード 10 は目的のデータ 14 とノード 1 の間にノード 15 が存在することを知っているため，ノード 1 を選択すると目的のキーを通過してしまうこともわかる．したがって，ノード 10 はローカル経路表においてキーに近い候補としてノード 12 を選択する．
(3) 次のノード候補として，ノード 12 のローカル経路表はノード 1 を示しているが，ノード 15 がノード 1 の手前にあることを知っている．直前の手順と同様に，ノード 12 はローカル経路表から次の候補の選定を試みるが，適切なノードが含まれないため選択できない．このような場合に初めて，ノード 12 はグローバル経路表における候補であるノード 15 を次のノードとして選択する．本例における，データセンタ間ホップはこの一度のみである．

このように ML-DHT における探索処理は通常の分散ハッシュ表よりも少ない数のデータセンタ間ホップで完了することが可能である．ML-DHT は反復型／再帰型ルーティング (Iterative/Recursive Routing) のどちらにも適用可能であるが，本論文では再帰的ルーティングを用いる．再帰型ルーティングにおいては，次のノードをグローバル経路表から選んだ時のみデータセンタ間通信が発生する．また，反復型ルーティングにおいても，通常の分散ハッシュ表よりは少ない数のデータセンタ間通信で済む．

3.3 Local-First Data Rebuilding

Local-first data rebuilding (LDR) はデータセンタ間のデータ転送量を小さく抑えつつ，目的のデータを取得するための手法である．LDR は Erasure Coding を用いることで，データセンタ間のデータ転送量とストレージの使用量を任意に調整可能にする．この手法は，Weatherspoon と Kubiatowicz [20] の研究に影響を受けている．彼らは Erasure Coding とレプリケーションについて，耐障害性とストレージ使用量の観点で評価を行った．

LDR は 3 つのステップから成る．まず *erasure-coding step* では，Erasure Coding を用い保存対象のデータを複数のチャンクと呼ばれるデータ断片に分割する．これは分散キーバリューストアにデータを put する際に，put を行うノードが実行する処理である．次に *uniform data putting*



(a) Chord 型分散ハッシュ表における探索処理例　　(b) ML-DHT における探索処理例

図 5　通常の Chord 型分散ハッシュ表と ML-DHT におけるルーティング

step では，キー空間へ均一に分散するようにして各チャンクを保存する．本研究では，この均一性は既存のハッシュアルゴリズムによって実現されるものとする．最後に *local-first data fetching step* では，ローカルのデータセンタに属するノードが保持するチャンクを優先的に利用して元のデータの復元を行う．Erasure-coding step と uniform data putting step はデータを保存 (put) する際に実行され， local-first data fetching step はデータを取得 (get) する際に実行される．

Erasure-coding step において，保存対象のデータは冗長性を持った m 個のチャンクに分割される．Erasure Coding を用いることで，元のデータは m 個のチャンクのうち，どの k 個のチャンクを用いても復元が可能となっている．$(k < m)$

Uniform data putting step において，m 個のチャンクはそれぞれ分散キーバリューストアに保存される．データセンタ間通信の機会を削減させるため，各チャンクはキー空間へ均一に分散して保存されなければならない．これを実現するために，LDR では各チャンクはハッシュ関数を用いて生成した値をキーとすることで確率的に均一に分散して保存を行う．各チャンクの保存処理自体は通常のキーバリューストアと同様の手順で行う．

Local-first data fetching step では，データを要求するノードは k 個のチャンクを収集する．この際，**任意の** k 個のチャンクから元のデータを復元できることと，各ノードが ML-DHT のローカル経路表を用いて同じデータセンタに属するノードを優先的に探索できることが重要となる．データを要求するノードはまず，k 個のチャンクの収集を，ローカル経路表のみを用いて取得することを試みる．k 個より多くのチャンクを収集できた場合は，それらを用いて元のデータの復元処理を行い処理を終了する．この時データセンタ間を跨ぐ通信は一切発生しないため，処理は短時間に終了する．一方，k 個未満のチャンクしか収集できなかった場合は，グローバル経路表も用いてチャンクを収集する．この際リモートのデータセンタからの取得が必要なチャンクの数は最大で k 個であり，1 個以上ローカルのデータセンタから取得できていた場合データセンタ間のデータ転送量はその分削減される．

データセンタ間を跨いで転送されるチャンクの数は，ローカルのデータセンタに属するノードの比率に比例して減少する．これは各チャンクのキーがハッシュアルゴリズムによってほぼ均一に分散されているためである．

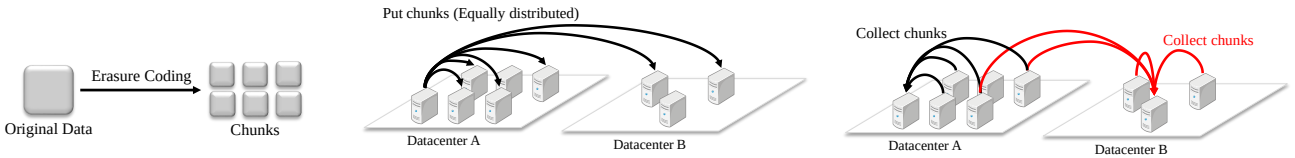
図 6 に LDR がデータセンタ間のデータ転送量を削減する例を示す．本例において，Erasure Coding のパラメータ (m, k) は $(6, 4)$ とし，また，データセンタ A 及び B に属するノード数の比率は 2:1．この場合，各データは以下の 3 つのステップで処理される．

(1) 元のデータを Erasure Coding を用いて 6 個のチャンクに分割する．(図 6(a))
(2) 生成されたチャンクは確率的に 2:1 の比率でデータセンタ A と B へ保存される．すなわち，生成したチャンクのうち 4 個はデータセンタ A へ，残りの 2 個はデータセンタ B へ保存される可能性が高い．(Fig. 6(b))
(3) データセンタ A に属するノードは元のデータ復元に必要な 4 個のチャンク全てをローカルのデータセンタ内で収集することができる．一方，データセンタ B に属するノードは 2 個のチャンクのみローカルのデータセンタ内で収集できない．しかし，2 個のチャンクのみをリモートのデータセンタ A から取得すれば十分なため，全てのデータをリモートから取得する場合よりも少ない転送量で済む．(Fig. 6(c))

もし単純分割（ストライピング）を用いた場合，データセンタ間のデータ転送量は LDR よりも大きくなる．これは単純分割で生成されたチャンクから元のデータを復元する場合は 6 個すべてが必要となるためである．LDR におけるデータセンタ間のデータ転送量の削減については 5 章で評価する．

4. 実装

オーバレイ構築ツールキット OverlayWeaver 0.10.3 [13] を用いて提案手法のプロトタイプを実装した．ML-DHT の実装例として，代表的な分散ハッシュ表アルゴリズムである Chord をベースとする，*ML-Chord* を OverlayWeaver 上に実装した．ML-Chord は OverlayWeaver に含まれる Chord の実装を拡張することで実装した．また，LDR の実装には Erasure Coding の実装である Zfec-1.4.24 [21] を用いた．LDR におけるエンコード関数は m, k 2 つのパラメータを引数とし，m は元のデータから生成するチャンクの数，k は元のデータを復元するために必要なチャンクを数とした．各チャンクのキーには元のデータを put する際



(a) 元のデータから 6 個のチャンクを生成する．(4 個あれば復元が可能)

(b) 各チャンクは，確率的に 2:1 の比率でデータセンタ A と B へ保存される．

(c) ローカルを優先して必要なチャンクを収集し，元のデータを復元する．

図 6 An example to put/get data with LDR.

に用いられたキーに添字として通し番号を付加したものを用いた．例えば元のリクエスト put(“foo”, *value*) はチャンクを生成した後に， put(“foo_0”, $chunk_0$), put(“foo_1”, $chunk_1$), . . . , put(“foo_m-1”, $chunk_{m-1}$) という形で処理される．

5. 評価

本章では，本論文の提案手法 ML-DHT と LDR がデータセンタ間通信が原因となる分散キーバリューストアの性能低下を低減するための評価実験について述べる．提案手法の有用性を確認するために，シミュレーション環境と実環境の 2 つを用いた．

シミュレーション環境では，500 台のノードが稼働するデータセンタを 2 つ用意した．シミュレーションには 1 コア 3.00 GHz Intel Xeon CPU 及び 2 GB の主記憶を備えた計算機を用い，Linux 3.2 及び Java 1.6 を使用した．

実環境には，Amazon の Elastic Computing Cloud (EC2) [22] と Virtual Private Cloud (VPC) [23] を用いた．東京リージョンとサンパウロリージョンに VPC を用意し，OpenVPN-2.3.2 を用いてデータセンタ間通信は暗号化した．東京リージョンには 2 つのマイクロインスタンスを用意し，サンパウロリージョンには 1 つのマイクロインスタンスを用意した．通信遅延と帯域幅の参考にするため，ラウンドトリップ時間 (RTT) とネットワーク帯域幅を，東京リージョン内及びリージョン間にて PING メッセージと iperf-2.0.5 を用いて計測した．iPerf は TCP 及び UDP での帯域幅を計測するためのツールである．各計測は 2013 年 10 月 10 日に行った．東京リージョン内及び東京–サンパウロ間の RTT はそれぞれ 0.391 msec と 384 msec であった．すなわち，この環境におけるデータセンタ間通信はデータセンタ内通信よりも通信遅延が約 1,000 倍大きかった．また，東京リージョン内及び東京–サンパウロ間のネットワーク帯域幅はそれぞれ 147 Mbps と 3.29 Mbps であった．すなわち，この環境におけるデータセンタ間通信はデータセンタ内通信の約 2.2 % の帯域幅であることがわかった．

本章で示す実験には，データセットとして一様分布となるようランダム生成した 10,000 個のキー／データのペアを用いた．各キーは 160 ビットのハッシュ値で，データにはランダム生成した 10 KB のバイト列を用いた．このデータセットの生成のために，キーバリューストアへ put/get 要求を発行するワークロード生成器を実装した．

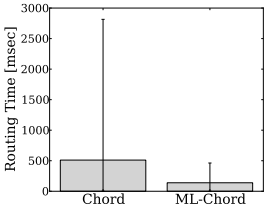


図 7 実環境における探索処理の応答時間

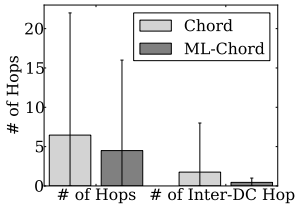


図 8 各探索処理におけるデータセンタ間ホップ数と総ホップ数

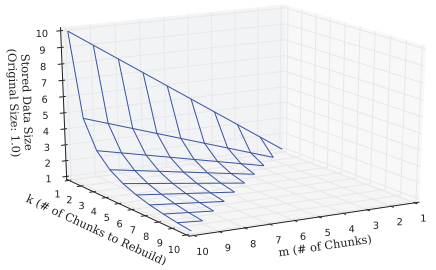
5.1 探索の応答遅延

ML-DHT がデータ探索時の応答遅延を削減することを示すために，実環境にて Chord と ML-Chord それぞれが探索に要する時間を比較した．ML-Chord は ML-DHT の実装例である．図 7 に平均応答時間の比較を示す．ML-Chord の応答遅延は約 74 % Chord より小さかった．これは ML-DHT のグローバル経路表とローカル経路表が，データセンタ間ホップの数を低減していたためである．

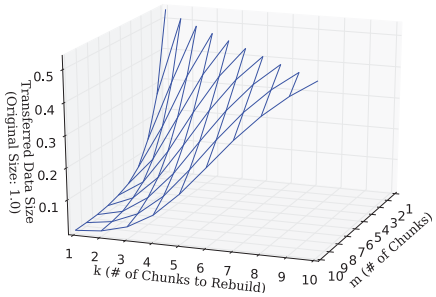
ML-DHT が実際にデータセンタ間ホップの数を低減していたことを示すために，シミュレーション環境にて Chord と ML-Chord でのルーティングを解析した．図 8 にデータセンタ間ホップ数及び総ホップ数を示す．ML-Chord は Chord より約 74 % 少ないデータセンタ間ホップ数で目的のノードに到達していたことがわかった．そして，ML-Chord は Chord より約 30 % 少ない総ホップ数で目的のノードに到達していたこともわかった．総ホップ数も低減したのは，ローカル経路表を用いることで，スキップリスト [24] のようにグローバル経路表にのみ含まれるノードを飛ばしてルーティングすることが可能なためである．これらの結果から，複数のデータセンタから構成される分散キーバリューストアの応答時間は，データセンタ間通信の応答時間の影響をより大きく受けることがわかる．また，ML-Chord は Chord と異なり，データセンタ間ホップが最大 1 回であったことから，データセンタ間を無駄に往復するホップを一切行っていないことがわかる．

5.2 データ転送量

LDR が，キーバリューストアの管理者がストレージ使



(a) ストレージ使用量



(b) Get 操作に伴うデータセンタ間の平均データ転送量

図 9 パラメータ設定を (m, k) とした場合のストレージ使用量とデータセンタ間転送量

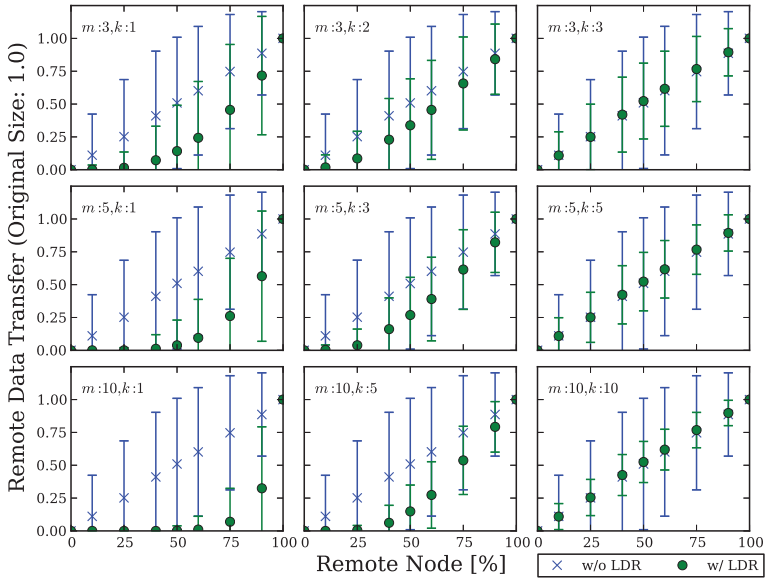


図 10 リモートのデータセンタに属するノードの比率と，データセンタ間のデータ転送量の比較

用量とデータセンタ間のデータ転送量を調整可能にすることを示すために 2 つの評価実験を行った．これらの実験におけるパラメータ設定 (m, k) は 3.3 章及び 4 章における定義と同様である．

第 1 の実験では，ストレージ使用量とデータ転送量がパラメータ設定によってどのように変化するかを比較した．図 9 (a), (b) はそれぞれストレージ使用量とデータ転送量を示す．各図における各格子点はは，パラメータ設定 (m, k) に対応する．各図より，LDR のパラメータ設定を変えることで，ストレージ使用量とデータセンタ間のデータ転送量を最粒度に調整できることがわかった．m または k を増加させると，データ転送量は減少するがストレージ使用量は増加した．反対に，m または k を減少させると，ストレージ使用量は減少したがデータ転送量は増加した．すなわち，キーバリューストアの管理者はパラメータ変えることで，必要に応じた性能設定を行うことが可能である．

第 2 の実験では，リモートのデータセンタに属するノード数の比率によってデータセンタ間のデータ転送量がどのように変化するかを比較した．図 10 に結果を示す．リモートのデータセンタに属するノードの比率が小さい場合は特に，データ転送量の平均値と分散が低減することがわかった．これは LDR がローカルのチャンクを優先利用することでリモートのデータセンタからのデータ転送の機会を削減するためである．また，パラメータ m, k がキーバリューストアの特徴に大きく影響することもわかった．パラメータ m はデータ転送量のばらつきに影響し，パラメータ k はデータ転送量の平均値に影響する．

6. 関連研究

Weatherspoon と Kubiatowicz は Erasure Coding をデータレプリケーションについて，ストレージ使用量と耐障害性の観点から比較評価した [20]．彼らは Erasure Coding によってストレージ使用量と冗長性を柔軟に設定できることを示した．LDR の設計はこの研究の影響を受けているが，本研究では Erasure Coding をストレージ使用量と **データセンタ間のデータ転送量** を柔軟に設定するために用いている．本論文では，LDR に Erasure Coding を用いることで，分散キーバリューストアにおけるデータセンタ間のデータ転送量を削減できることを示している．

クライアントの地理的な位置にあわせたデータ移送手法 [25, 26] は，データを利用するクライアントに近いデータセンタに再配置することで，データ配信の遅延を抑制する．これらの手法では，応答遅延を抑制することでユーザ体験の向上を目的としている．応答遅延を抑制するため，これらの手法では，ユーザの地理的位置に対して最適なデータセンタにデータオブジェクトを動的に移送する．本研究でも複数のデータセンタにデータオブジェクトを配置するが，クライアント粒度での最適化は行わない．本研究とこれらの手法は補完的な関係にある．

ストレージの Geo-replication （地理的冗長化） 手法 [27–30] は，複数のデータセンタ間でストレージの状態を複製／同期しつつ，応答遅延の増加を抑制する手法である．これらの手法では，地理的に離れたデータセンタ間であっても一定レベルのトランザクション分離を保証しつつ，応答遅延の低減を実現している．これらの手法は，災

害復旧やデータセンタ間通信を防ぐことを主な目的としている．目的を達成するために，これらの手法では全てのデータセンタにレプリカを配置し，トランザクションを全てのレプリカに対して適用する．全てのデータセンタにレプリカを保持する必要があるため，これらの手法では本研究の目的であるストレージ資源の効率的な使用は実現できない．

7. まとめ

本論文では，データセンタ間を跨ぐ分散キーバリューストアを構築するための要素技術として，*Multi-layered DHT* (ML-DHT) と *Local-first data rebuilding* (LDR) を提案した．これらの手法は共に，データセンタ間を跨ぐ分散キーバリューストアにおいて WAN を介したデータセンタ間通信を減少させる働きをする．ML-DHT はデータセンタ間を跨ぐようなホップが無駄に発生することを防ぐ特殊なルーティングを行うことで，目的のデータを探索する際の通信遅延の増大を抑制する．LDR は Erasure Coding を用いて冗長性を持ったチャンクを生成することで，データセンタ間のデータ転送量の増大を抑制する．シミュレーション環境とインターネットを介した実環境を用いて行った提案手法の評価実験では，Chord を用いたシステムと比較して応答遅延が 74 % 減少し，ストレージ使用量とデータ転送量を自由に調整できることを確認した．

参考文献

[1] Benny Rochwerger et al. Reservoir - when one cloud is not enough. *Computer*, 44(3):44–51, 2011.
[2] Fay Chang et al. Bigtable: A Distributed Storage System for Structured Data. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation*, 2006.
[3] Giuseppe DeCandia et al. Dynamo: amazon's highly available key-value store. In *Proc. of ACM SIGOPS Symposium on Operating Systems Principles*, 2007.
[4] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. In *Proc. of ACM SIGOPS Int'l Workshop on Large Scale Distributed Systems and Middleware*, 2009.
[5] The Apache Software Foundation. Apache HBase. `http://hbase.apache.org/`.
[6] 10gen, Inc. MongoDB. `http://www.mongodb.org/`.
[7] Oracle Corporation. Oracle NoSQL Database. `http://www.oracle.com/technetwork/products/nosqldb/`.
[8] Microsoft Corporation. Windows Azure Table Storage Services. `http://www.windowsazure.com/en-us/develop/net/how-to-guides/table-services/`.
[9] Thomas Locher et al. eQuus: A Provably Robust and Locality-Aware Peer-to-Peer System. In *Proc. of IEEE International Conference on Peer-to-Peer Computing*, 2006.
[10] W. K. Lin et al. Erasure Code Replication Revisited. In *Proc. of IEEE International Conference on Peer-to-Peer Computing*, 2004.
[11] Ros G. Dimakis et al. Network coding for distributed storage systems. In *Proc. of IEEE International Conference on Computer Communications*, 2007.
[12] James S. Plank et al. A performance evaluation and examination of open-source erasure coding libraries for storage. In *Proc. of USENIX Conference on File and Storage Technologies*, 2009.
[13] Kazuyuki Shudo et al. Overlay Weaver: An overlay construction toolkit. *Computer Communications*, 31(2):402–412, 2008.
[14] Ion Stoica et al. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of ACM Special Interest Group on Data Communications Conference*, 2001.
[15] David Karger et al. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of annual ACM symposium on Theory of computing*, 1997.
[16] Sylvia Ratnasamy et al. A scalable content-addressable network. In *Proc. of ACM SIGCOMM 2001 Conference on applications, technologies, architectures, and protocols for computer communications*, 2001.
[17] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proc. of the IFIP/ACM Int'l Conference on Distributed Systems Platforms*, 2001.
[18] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proc. of International Workshop on Peer-to-Peer Systems*, 2002.
[19] B.Y. Zhao et al. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 2004.
[20] Weatherspoon and Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Proc. of the First International Workshop on Peer-to-Peer Systems*, 2002.
[21] Zooko Wilcox-O'Hearn. Zfec. `http://pypi.python.org/pypi/zfec/`.
[22] Amazon Web Services LLC. Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2/`.
[23] Amazon Web Services LLC. Amazon Virtual Private Cloud. `http://aws.amazon.com/vpc/`.
[24] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.
[25] Sharad Agarwal et al. Volley: automated data placement for geo-distributed cloud services. In *Proc. of USENIX Conference on Networked Systems Design and Implementation*, 2010.
[26] N. Tran et al. Online migration for geo-distributed storage systems. In *Proc. of USENIX Annual Technical Conference*, 2011.
[27] James C. Corbett et al. Spanner: Google' s Globally-Distributed Database. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation*, 2012.
[28] Cheng Li et al. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation*, 2012.
[29] Wyatt Lloyd et al. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proc. of ACM Symposium on Operating System Principles*, 2011.
[30] Yair Sovran et al. Transactional storage for geo-replicated systems. In *Proc. of ACM Symposium on Op-*

erating System Principles, 2011.