情報処理学会論文誌　コンピューティングシステム　Vol.8　No.2　1–14 (June 2015)

データセンタ間通信による性能低下を抑えた広域分散型キーバリューストア構築手法

堀江 光[1,a)]　浅原 理人[2]　山田 浩史[3]　河野 健二[1]

受付日 2014年10月18日, 採録日 2015年3月25日

概要：近年，複数のデータセンタの計算資源を集約した新たな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の計算資源を用いる場合と比較して，クラウド環境の特徴である伸縮性や可用性を向上させることを目的としている．このように非常に多数のサーバを用いる環境で高いスケーラビリティを発揮するストレージの 1 つとして分散キーバリューストアがある．しかし分散キーバリューストアを複数のデータセンタ間をまたぐ環境で利用する場合には著しく性能が低下するという問題がある．この問題の原因は，分散キーバリューストアを構成する各サーバ間で行われる通信が，データセンタ内ネットワークに比べ高遅延/狭帯域なネットワークをまたぐためである．この問題を解決するために，本研究では *Multi-Layered Distributed Hash Table*（*ML-DHT*），*Local-first Data Rebuilding*（*LDR*）という 2 つの基礎手法と，それらを用いて複数のデータセンタをまたいで分散キーバリューストアを構築する方法を提案する．ML-DHT は，冗長なデータセンタ間通信の発生を抑制するルーティングを行うことにより，通信遅延を抑えたデータ探索を実現する．LDR は，保存データに冗長性を与えたうえで分割することで，各データセンタにレプリケーションを行う場合と比較して，ストレージ使用量とデータセンタ間通信の量をより柔軟に設定可能にする．実験により提案手法が，代表的な分散ハッシュ表（DHT）である Chord を用いて構築したキーバリューストアと比較し，データ探索に要する時間を約 74%軽減し，データの冗長度を調整することで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，クラウド間連携

Making Key-value Stores More Efficient for Inter-datacenter Environments

Hikaru Horie[1,a)]　Masato Asahara[2]　Hiroshi Yamada[3]　Kenji Kono[1]

Received: October 18, 2014, Accepted: March 25, 2015

Abstract: Cloud-federations, which aggregate resources running on multiple datacenters, become popular platforms for many Internet-scale services. This is because a cloud-federation has higher elasticity and availability than a cloud running on a single datacenter. In terms of scalability, distributed key-value store (DKVSs) are one of attractive databases in such large scale environments. However, the large latency and narrow bandwidth of inter-datacenter communications do not alllow DKVSs to realize the potencial of their performance. In this paper, we organize issues of DKVSs over multiple datacenters, and demonstrate how to reduce and hide the weakness of inter-datacenter communications for DKVSs. To solve the issues, we introduce two techniques called *Multi-Layered Distributed Hash Table* (*ML-DHT*) and *Local-first Data Rebuilding* (*LDR*). ML-DHT is a technique to extend an existing DHT and reduces inter-datacenter communications even over multiple datacenters. LDR enables DKVs administrators to characterize the capacity and inter-datacenter traffic of the storage by fine-grained redundancy. Experimental results demonstrate that a prototype system contains the techniques improve the latency up to 74% compared with a Chord-based system and enables us to balance the amount of storage usage and remote data transfer by changing the redundancy of each data fragment.

Keywords: distributed key-value stores, distributed hash tables, cloud federations

1. 背景

現在，インターネット上で展開されている各種ウェブサービスの運用基盤として，複数のデータセンタを用いたクラウド環境が利用され始めている．これは，ウェブサービスの社会的重要性の高まりを受け，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出てきたためである [1]．複数のデータセンタを用いたクラウド環境では，それぞれ単一のデータセンタで稼働している計算資源を集約し 1 つの巨大なクラウド環境を構築することで，単一のデータセンタで構築されたクラウド環境よりも柔軟な資源管理を実現する．たとえば，このようなクラウド環境上では，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台数/地理的な場所/稼働させる時機をより適切に選択することが可能となる．

このような膨大な計算資源を効率的に利用するために適したストレージの 1 つとして**分散キーバリューストア** [2], [3], [4], [5], [6], [7], [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実現するとともに，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を利用するアプリケーションは，データが物理的にどのノードに保存されているかという情報を知る必要はない．分散キーバリューストアはこれらの特徴を，**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．ただし，手法によっては特定の層の機能が必要最低限に最小化されている場合もある．また，このような構造の整理には OverlayWeaver [9] におけるアーキテクチャを参考にした．キーバリューストア API（Application Programming Interface）層は，アプリケーションが Put，Get 等のストレージを使用するための汎用的な API を提供する．MondoDB [6] 等のように SQL 文に似たクエリに対応したものも存在する．データフェッチ層は，オリジナルのデータおよびそれを構成するチャンクと呼ばれるデータ断片を相互に変換する．たとえば，データを細かく分割することによる負荷分散や，過去に取得したデータをキャッシュとして保持することによる高速化等を行う．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内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．ノードの探索には，分散ハッシュ表（DHT）を用いる完全分散型の手法やインデックスサーバを用いる集中管理型の手法等が存在する．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．データの保存には，ディスクのみでなくメモリを利用する場合もある．

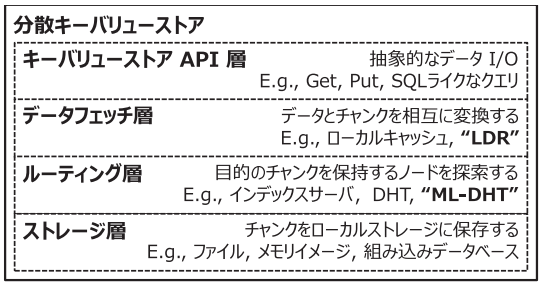


図 1　分散キーバリューストアを構成するソフトウェアスタック
Fig. 1　Software stack of distributed key-value stores.

[1] 慶應義塾大学
Keio University, Yokohama, Kanagawa 223–8522, Japan
[2] NEC グリーンプラットフォーム研究所
NEC Green Platforms Research Laboratories, Kawasaki, Kanagawa 211–8666, Japan
[3] 東京農工大学
Tokyo University of Agriculture and Technology, Koganei, Tokyo 184–8588, Japan
[a)] hikaru.horie@sslab.ics.keio.ac.jp

従来の分散キーバリューストアの設計は**データセンタ内**での利用が想定されており，高遅延かつ狭帯域である**データセンタ間**の通信を介して各ノードが接続されることは考慮されていない．各ノードは一般的に，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，高頻度に通信を行う．これらの通信がデータセンタ間をまたいで行われる場合，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，それを利用する分散キーバリューストアの応答遅延の増大とスループットの低下も著しい．すなわち，データセンタ間をまたいで構築された分散キーバリューストアは十分な性能を発揮できず，それを利用するアプリケーションの応答性とスループットを顕著に低下させる場合がある．

本研究は，複数のデータセンタ間をまた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，このような環境における分散キーバリューストアの性能低下を軽減するための要素技術として *Multi-Layered DHT*（*ML-DHT*）および *Local-first Data Rebuilding*（*LDR*）という 2 つの手法を提案する．図 1 で示すように，ML-DHT および LDR はそれぞれルーティング層およびデータフェッチ層で動作する．これらの手法はデータセンタ間通信の量および頻度を軽減するとともに，トレードオフの関係にあるストレージ使用量とデータセンタ間通信量をより柔軟に設定することを可能とする．

ML-DHT はデータセンタやノードの物理的な構成によらず全ノードおよび全データを一律なキー空間上で管理する．キー空間を一律にすることでストレージ容量の効率的な拡張を実現するとともに，インデックス管理をする特別なサーバ等を用いず任意のノードから任意のデータを探索可能である．また，ML-DHT は冗長なデータセンタ間通信を行わずにデータを探索するよう設計されており，データセンタ間通信にともなう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，データセンタごとにキー空間を階層構造で分割するシンプルな方法 [10] 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の大きさを均等に保つにはノードの加入/離脱ごと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閉じたキー空間を管理する場合，あるデータセンタにノードを追加しても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つながりにくい．ML-DHT では一律なキー空間を採用することで階層型キー空間におけるこのような問題を回避している．ML-DHT については 3.2 節で詳しく述べる．

LDR は，オリジナルデータとチャンク（冗長性を持ったデータ断片）を相互に変換して扱うことで，データセンタ間をまたぐ通信の量を軽減する．この変換には，Erasure Coding [11], [12], [13] を用いる．データセンタ間通信を減らすため LDR では，一部であってもチャンクがローカルのデータセンタに存在する場合はそれを優先的に利用してオリジナルデータの復元を行う．LDR は，トレードオフの関係に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冗長度が高い場合，ローカルのデータセンタ内で必要な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，チャンクの冗長度が低い場合，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ものの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 [9] を用い ML-DHT，LDR を用いた分散キーバリューストアの実装および評価実験を行った．評価にあたり，ML-DHT により Chord [14] を拡張した ML-Chord を実装した．実験で，提案手法が，Chord を用いたシステムと比較してデータ探索時間を 74%減少させ，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とおりである．2 章では複数のデータセンタ間をまたぐ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 章では提案手法のプロトタイプを用いてそ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間をまたぐ分散キーバリューストアは，単一データセンタ内で運用される分散キーバリューストアに対して，拡張性や伸縮性が勝る可能性が高い．これは複数のデータセンタを統合した方が利用可能となる資源量の選択の幅が広がるためである．**図 2** に複数のデータセンタ間をまた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/離脱するたびに，すべてのノードの担当キー空間の再割当てをともなわずに済む．本論文では分散ハッシュ表（DHT）を用いる分散キーバリューストア [14], [16], [17], [18], [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すべてのノードが，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は単一のデータセンタのみを用いる場合より利用可能な資源も多くなるため，完全分散型の管理手法はスケーラビリティの観点で親和性が高い．

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設計について，3 つの観点から議論する．

2.1 ストレージ性能の均等な拡張性

分散キーバリューストアはノードの追加に応じてストレージ性能を均等に拡張できる必要がある．なぜなら，追加ノードの貢献が一部に偏っていると，ノードを追加してもキーバリューストア全体の性能があまり向上せず運用コストが割高になるといった問題につながるためである．つまり，各ノードの性能が同じ場合，担当するキー空間の大きさは確率的に均一である必要がある．

単一のデータセンタから構成される分散キーバリュー

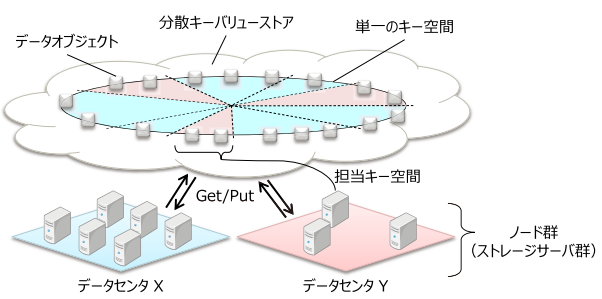


図 2　一律のキー空間を持つ分散キーバリューストア

Fig. 2　An inter-datacenter distributed key-value store with a flat key space.

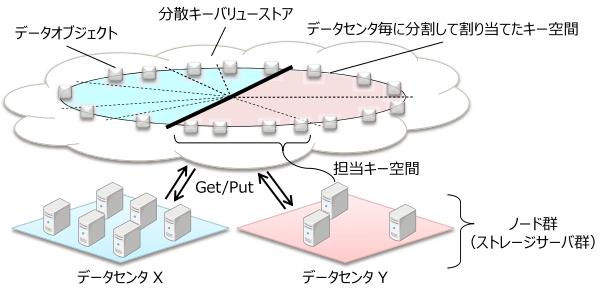


図 3　キー空間が分割された分散キーバリューストア
Fig. 3　An inter-datacenter distributed key-value store with a split key space.

ストアでは，追加されたノードは確率的に均等にキーバリューストア全体の負荷分散に貢献するため，効率的な性能向上が期待できる．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，ノードの追加は他のノードの負荷を均等に分散することにつながり，キーバリューストア全体の性能向上に貢献する．

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効率的な拡張性を得られない場合がある．たとえば，キー空間が分割された分散キーバリューストアでは，ノードを追加しても全体の性能向上にはつながらない場合がある．**図 3** に，キー空間がデータセンタごとに分割された分散キーバリューストアの例を示す．この例では，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，各キー空間はさらに各データセンタに属するノードに均一に割り当てられている．このとき，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は寄与することができない．このように分割されたキー空間は，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　データセンタ間通信の頻度の低減

複数のデータセンタ間をまたぐ分散キーバリューストアでは，データセンタ間通信の発生を少なく抑える必要がある．なぜなら，データセンタ間通信はデータセンタ内通信に対して遅延が大きく，データ探索時の応答遅延の増加につながるためである．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つながり，そのサービスの品質を低下させる．

図 6 (a) に，実ネットワーク上におけるノードの分布を考慮しない一般的な DHT による非効率なデータ探索の例を示す．本例において，各ノードは目的のノードに近いノードを自身の保持する経路表より選択し，次のホップ先とする．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．この例では，探索経路に同じデータセンタ間をまたぐ通信を複数回含んでいるが，一度別のデータセンタにホップした後に再び元のデータセンタに戻ることは通信遅延が大きく無駄である．このような冗長なデータセンタ間通信の発生は，オーバレイネットワークが各ノードの属するデータセンタを考慮していないことに起因する．これは不必要にデータ探索の応答遅延の増大させるため，複数のデータセンタ間をまたぐ分散キーバリューストアには不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　データセンタ間通信の転送量の低減

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，必要なデータが別のデータセンタに配置されていた場合にスループットが低下する．したがって，このようなデータセンタ間通信路を介した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通信路を介した転送量を減少させる仕組みが必要である．

データ読み込み時におけるデータ転送量を減らすために効果的なアプローチの 1 つとして，別のデータセンタに存在するデータ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で，狭帯域なデータセンタ間通信の利用を避けることができる．しかし，このアプローチはすべ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 カ所のデータセンタで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．

一方，このようなデータ複製を行った場合，データ書き込み時に各レプリカを更新するためのデータ転送が必要になるという問題がある．したがって，更新頻度の高いワークロード下においては，更新にともなうデータ転送量を抑えるための対策が必要である．たとえば，配置するレプリカの数を減らすことや，更新データの一部のみを転送すること等によって，転送量を軽減することができる．

ストレージ使用量とデータ転送量のバランスをとるために，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データセンタ間通信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ともなって発生するデータセンタ間通信を減少させることを目的とはしていない．このようなデータセンタ間通信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が分散キーバリューストアの性能に及ぼす悪影響を低減させる．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ない．なぜなら，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，性能低下につながるデータセンタ間通信を少なく抑える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通信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きにともなうデータセンタ間通信を抑制する．3.2 節と 3.3 節で，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．分割されたキー空間と異なり，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ごとに分割されることなく，すべてのノードが同様の扱いを受ける．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つながる．たとえば，図 2 においてデータセンタ X 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 X 以外のデータセンタ Y であった場合，データセンタ X に追加されたこのノードによってデータセンタ Y のノードが負荷分散の恩恵を受けることとなる．この際，担当するキー空間の移譲のためにデータセンタ間通信が発生するが，これは頻繁に起こらないと想定する．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 等から構成される 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，一定の基準でまとめられたノードの集合に対し，集合間を往復することなく目的のノードへ到達可能とする，DHT 拡張手法である．本論文では，実ネットワーク上における距離に基づいた階層型クラスタをノードの集合として用いる．これにより，通信遅延の小さいノードを優先利用し目的のノードへ到達可能となる．たとえば，同じデータセンタに属するノードを集合とすることで，データセンタ間通信の頻度を低減させる．すなわち，ML-DHT を用いることで，2.2 節で述べたような非効率的なデータ探索を避けることができる．また，本手法はマルチホップ探索を行う DHT であれば一般的に適用可能であり，任意のノードへの到達性とデータの可用性をオリジナルの DHT と同等の水準で保証する．

本手法の説明に用いる各記号を**表 1** に，集合 $G_{n,l}$ の関係について具体例を**図 4** に示す．本手法において各ノードは，実ネットワーク上における距離が一定の基準より小さいノードの集合を管理対象とする L 個の経路表を保持し，L 層のオーバレイネットワークレイヤを構成する．本論文では，第 0 層を最下位レイヤと定義し，全体集合を管理対象とする．一方，l が大きいレイヤを上位レイヤと定義し，より小さな集合を管理対象とする．すなわち，上位層における集合は，実ネットワーク上でより近いノードから構成される．また，集合 $G_{n,l}$ は一般的に次の各性質を満たす．ただし，$n_1 \neq n_2$ とする．

$$G_{n,k+1} \subset G_{n,k} \quad \text{for} \quad \forall k \in \mathbb{N}, 0 \leq k < L-1 \qquad (1)$$

表 1　記号の定義
Table 1　Definition of symbols.

記号	定義
L	ML-DHT が構成するオーバレイネットワークレイヤの数．$L \in \mathbb{N}$
l	ML-DHT が構成するオーバレイネットワークレイヤの 1 つ．$l \in \mathbb{N}, 0 \leq l < L$
G	全ノードの集合
$G_{n,l}$	第 l 層でノード n が属する集合．
$n.tables[l]$	ノード n が保持する，第 l 層用の経路表．$G_{n,l}$ に含まれるノードを管理対象とする．
$n.intervals[l]$	第 l 層におけるノード n の担当キー空間
$n.responsible$	ノード n が実際にデータの保持を担当するキー空間．$n.intervals[0]$ と一致する．

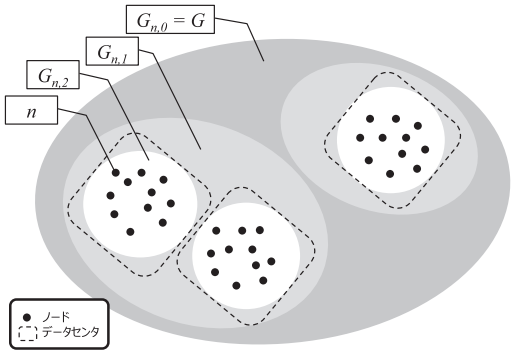


図 4　実ネットワーク上におけるノード分布と ML-DHT におけるレイヤ構成の例（$L = 3$）
Fig. 4　An example of node distribution in a physical network and logical layering with ML-DHT ($L = 3$).

$$G_{n,0} = G \quad \text{for} \quad \forall n \in G \tag{2}$$

$$G_{n_1,l} = G_{n_2,l} \quad \text{for} \quad n_1 \in G_{n_1,l} \wedge n_2 \in G_{n_1,l} \tag{3}$$

$$G_{n_1,l} \cap G_{n_2,l} = \emptyset \quad \text{for} \quad n_1 \in G_{n_1,l} \wedge n_2 \notin G_{n_1,l} \tag{4}$$

ML-DHT を構成する各ノードは各レイヤについて担当のキー空間 $n.intervals[l]$ を割り当てられるが，これはオーバレイネットワークを管理するためのものであり，実際に保存/管理を担当するデータは $n.responsible$（$= n.intervals[0]$）に含まれるものが対象となる．

また，各経路表の更新処理は，対象となるノードの集合が異なる点を除き，それぞれオリジナルの DHT と同様のアルゴリズムで行う．すなわち，すべてのノードを対象としている経路表 $n.table[0]$ はオリジナルの DHT と完全に同等であり，これにより全ノードへの到達性と可用性を同等の水準で保証する．

ML-DHT へのノードの加入およびデータの探索は以下に示す手順で行う．

加入処理（*Join*）：

ML-DHT において，新たに加入するノード n_{new} は L 個の経路表すべてについて加入処理を行う．加入処理にあたり，n_{new} は $G_{n_{\text{new}},L-1}$ から任意のノードをブートストラップノード n_{bs} として選出する．加入処理にともなう経路表 $n.tables[l]$ の更新手順はオリジナルの DHT と同様であるが，各層の独立性を保証するため，管理対象層の異なる経路表を混同しないよう行う．経路表 $n.tables[l]$ は次の手順で更新される．

(1) n_{new} が n_{bs} に対し $n_{\text{new}}.tables[l]$ について加入要求を送信する．
(2) n_{bs} は $n_{\text{bs}}.tables[l]$ を用いて，$n_{\text{new}}.intervals[l]$ を現在担当しているノード n_{cur} を特定する．
(3) n_{bs} は n_{new} に n_{cur} の情報を送信する．
(4) n_{cur} は n_{new} に対し，オリジナルの DHT と同様の方法で $n_{\text{cur}}.intervals[l]$ から $n_{\text{new}}.intervals[l]$ を移譲，加入処理を完了する．ただし，移譲するキー空間に該当する保存データの移送は $l = 0$ についてのみ行う．

もし第 l 層に n_{bs} となり得るノードが存在しない場合は，n_{new} を最初のノードとして $n_{\text{new}}.tables[l]$ の構築を行う．

探索処理（*Look-up*）：

ML-DHT は，経路表 $n.tables[l]$ を用いて到達可能なノードが集合 $G_{n,l}$ のノードであることを利用し，同じ集合間を往復せずに目的のキーを探索する．

図 5 に ML-DHT におけるキー id 探索処理の擬似コードを示す．各ノードはキー id を探索する際に，より上位の（l が大きい）層の経路表を優先的に利用し，その層で担当のノードが見つからなかった場合に下位の経路表を利用する．第 l 層における探索処理はオリジナルの DHT と同様に行い $id \in n.intervals[l]$ となるノード n が見つかるまで実行するが，この間経路に $G_{n,l}$ 以外のノードを含むことはない．

```
// node n has routing tables for each layer (L is # of layers)
// each table is dealt with following an original DHT algorithm
n.tables = {table_0, table_1, … , table_L-1};

// ask node n to find responsible node for specified id
n.find_responsible_node(id)
    while (id not in n.intervals[0])
        for layer = L-1 downto 0
            n' = get_next_candidate(id, layer);
            if (n' exists)
                return n'.find_responsible_node(id);
    return n; // n is a responsible node for the id

// get next node from layer^th routing table of local node
n.get_next_candidate(id, layer)
    // choose next node n' from n.tables[layer]
    // following the original DHT algorithm
    n' = tables[layer].get_next_candidate_from_table(id);
    if (n' exists and n' != n)
        return n'; // node n' is more suitable than node n
    return; // node n is responsible for the id in this layer
```

図 5　ML-DHT におけるキー id 探索処理の擬似コード
Fig. 5　Pseudo-code to find identifier id in ML-DHT.

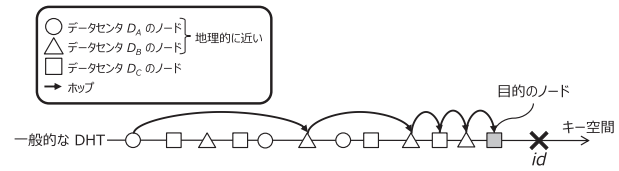
$id \in n.intervals[l]$ を発見した場合，$id \in n.responsible$ であるかを確認する．この確認処理は，ノード n が自身のローカル情報を確認するのみで通信はともなわない．ここで $id \in n.responsible$ の場合，ノード n が担当のノードであるため探索を終了する．一方 $id \notin n.responsible$ の場合，ノード n は担当ノードではないが第 l 層にはそれ以上探索対象となるノードが残っていないため，下位層で次のホップ先を探す．下位層でホップした後は，同様に探索を行う．

また，ML-DHT における探索の経路長はオリジナルの DHT と同じかそれより短い．これは，$G_{n,l} \subset G$ より $|G_{n,l}| < |G|$ であり，上位層では 1 ホップで目的のノードにより近づけるためである．一方で，ML-DHT では各層に対して経路表を保持する必要があるため，オリジナルの DHT に比べ L 倍の領域を経路表のために確保する必要がある．

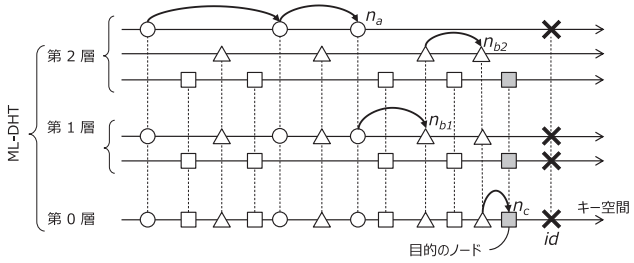
ML-DHT は反復型/再帰型（Iterative/Recursive）の探索いずれにも適用可能であり経路長を短縮できるが，本論文ではより利点の多い再帰型探索を用いる．反復型の場合，探索が一度でも別の集合にホップするとそれ以降は集合内でのホップであってもつねに集合をまたいだ通信をともなう．一方，再帰型の場合，つねにホップ元とホップ先のノード間で通信するため，集合内でのホップは集合をまたいだ通信をともなわない．すなわち，再帰型の探索の方がより ML-DHT によるデータセンタ間通信頻度の削減効果が得られる．

図 6 (b) に ML-DHT によりデータセンタ間通信の頻度を削減する例を示す．本例において，各ノードは目的のノードに近いノードを自身の保持する経路表より選択し，次のホップ先とする．また，各層 0，1，2 はそれぞれすべてのノード，近隣地域のノード，データセンタ内のノードを管理対象とする．探索処理は次のように進行する．

(1) $id \in n_a.intervals[2]$ となるノード n_a まで第 2 層（D_A 内）を探索する．
(2) $id \notin n_a.responsible$ であるため，$n_a.tables[1]$ を用いて n_{b1} へホップする．



(a) 一般的な DHT における探索処理例
(a) An example of routing process in a general DHT.



(b) ML-DHT に拡張した場合における探索処理例（$L = 3$）
(b) An example of routing process in ML-DHT ($L = 3$).

図 6　データセンタ間をまたぐ環境における探索処理の比較
Fig. 6　Comparison of routing process in an inter-datacenter environment.

(3) $id \in n_{b2}.intervals[2]$ となるノード n_{b2} まで第 2 層（D_B 内）を探索する．
(4) $id \notin n_{b2}.responsible$ であるが，$n_{b2}.tables[1]$ からは次のホップ先を得られないため，$n_{b2}.tables[0]$ を用いて n_c へホップする．
(5) $id \in n_c.responsible$ であるため，探索を終了する．

このように ML-DHT では集合間を往復するホップは発生しないため，データセンタの拠点数以上のデータセンタ間ホップが発生することはない．また，探索処理は一般的な DHT 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．データセンタの拠点数が増えると，データセンタ間ホップの最大値も増加するため，下位層で複数のデータセンタを含む集合を設ける等の対応をとることで，データセンタ間ホップが生じても比較的近距離のデータセンタが選択されやすくなる．

3.2.1　ML-Chord

本項では，DHT を ML-DHT に拡張する例として，代表的な DHT アルゴリズムである Chord [14] を多層化した *ML-Chord* について述べる．ここでは各ノードが持つ経路表の数 L を 2 とし，各経路表をグローバル経路表（$l = 0$）とローカル経路表（$l = 1$）と呼ぶ．また，第 1 層におけるノードの集合はデータセンタ単位とする．

ML-Chord に加入するには，n_{new} は n_{bs} に加入要求を送信する．$n_{\mathrm{new}}/n_{\mathrm{bs}}$ は，通常の Chord と同様のアルゴリズムでグローバル経路表をそれぞれ初期化/更新する．双方が同じデータセンタに属する場合，ローカル経路表に含まれる情報を用いて，それぞれのローカル経路表を初期化/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，n_{new} はローカル経路表を自身のみ存在するものとして初期化し，n_{bs} はローカル経路表に対して何も処理を行わない．

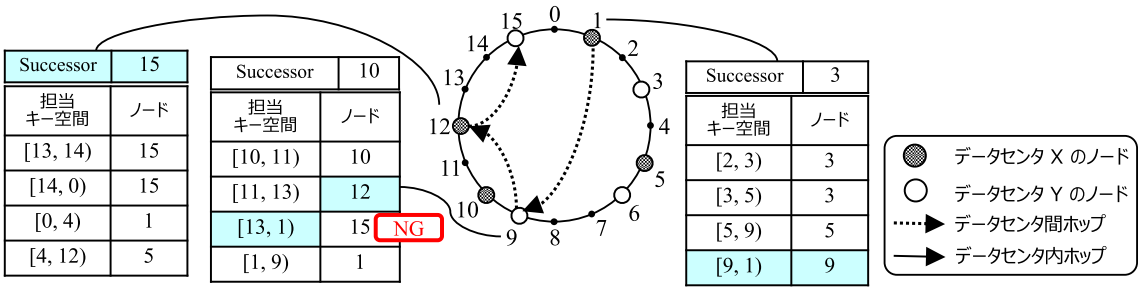
```
#define GLOBAL_LAYER 0
#define LOCAL_LAYER 1

// 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'.successor;

// 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 7　ML-Chord におけるキー *id* 探索処理の擬似コード
Fig. 7　Pseudo-code to find a successor node of an identifier *id* in ML-Chord.

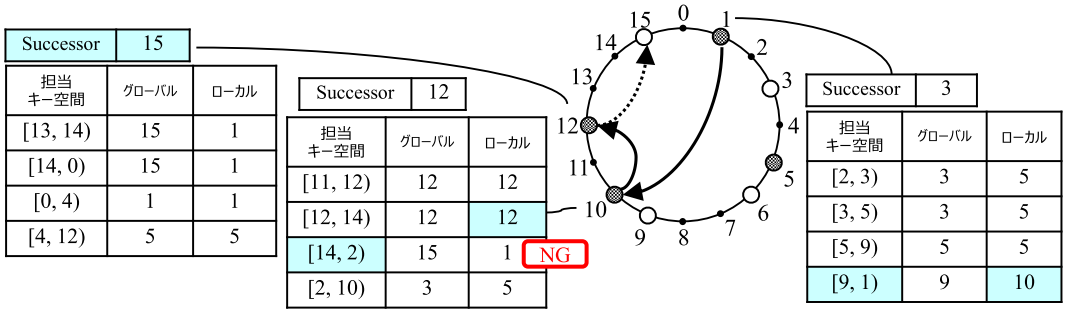
図 7 に ML-Chord における探索処理の擬似コードを示す．各ノードはクエリの転送先，すなわち次のホップ先を選択する際にローカル経路表を優先的に利用する．各ノードは，次のホップ先として適切なノードをローカル経路表から発見できない場合のみ，グローバル経路表を用いる．

図 8 は Chord 型の DHT と ML-Chord 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 および 12 はデータセンタ X，残りはデータセンタ Y で稼働している．また，各例とも，データセンタ Y 所属ノード 15 の管理担当データ 14 をデータセンタ X 所属ノード 1 が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またぐホップが必要である．Chord 型の DHT では，3 つのホップすべてがデータセンタ間をまたいでおり，探索にともな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属するデータセンタ（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またいでいる．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 8 (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



(a) Chord 型 DHT における探索処理例
(a) An example of routing process in a Chord-like DHT.



(b) ML-Chord における探索処理例
(b) An example of routing process in ML-DHT.

図 8　Chord 型 DHT と ML-Chord における探索の比較
Fig. 8　Comparison of routing process between a Chord-like DHT and Chord-based ML-DHT.

分かる．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．

3.3　Local-first Data Rebuilding

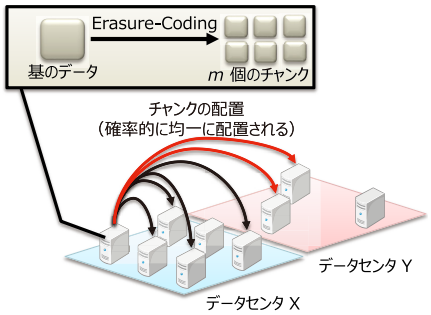
Local-first Data Rebuilding（LDR）はデータセンタ間のデータ転送量を小さく抑えつつ，目的のデータを取得するための手法である．LDR は Erasure Coding を用いることで，データセンタ間のデータ転送量とストレージの使用量を任意に調整可能にする．この手法は，Weatherspoon ら [20] の研究に影響を受けている．彼らは Erasure Coding とレプリケーションについて，耐障害性とストレージ使用量の観点で評価を行った．

LDR は 3 つのステップからなる．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 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．

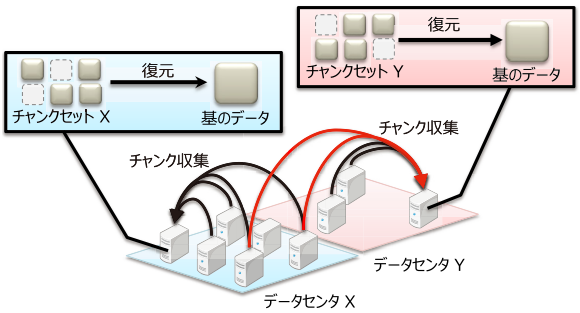
Erasure-coding step において，保存対象のデータは冗長性を持った m 個のチャンクに分割される．Erasure Coding を用いることで，元のデータは m 個のチャンクのうち，どの k 個のチャンクを用いても復元が可能となっている（$k < m$）．

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



(a) 元のデータからチャンクを生成し，担当ノードに配置する．
(a) Generate chunks from the original data and put the chunks on responsible nodes.



(b) ローカル優先でチャンクを収集し，元のデータを復元する．
(b) Collect chunks necessary to rebuild the original data, giving priority to local chunks.

図 9　LDR を用いたデータの配置と取得の例
Fig. 9　An example to Put/Get data with LDR.

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 個以上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．このときデータセンタ間をまたぐ通信はいっさい発生しないため，処理は短時間で終了する．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．

データセンタ間をまた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 9 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 X および Y に属するノード数の比率は 2:1 とする．この場合，各データは次のように処理される．

(1) 元のデータを Erasure Coding を用いて 6 個のチャンクに分割し，生成したチャンクを担当ノードに配置する．この際，チャンクは確率的に均一にキー空間上に配置されるため，2:1 の比率でデータセンタ X，Y に配置される．すなわち，6 個のチャンクのうち 4 個はデータセンタ X へ，残りの 2 個はデータセンタ Y に配置される可能性が高い（図 9 (a)）．
(2) データセンタ X に属するノードは元のデータ復元に必要な 4 個のチャンクすべ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 Y に属するノードはローカルのデータセンタ内の収集では 2 個のチャンクが不足するため，これらをリモートのデータセンタ X から取得する必要がある．しかし，2 個のチャンクのみをリモートのデータセンタ X から取得すれば十分であり，元のデータすべてをリモートから取得する場合よりも少ない転送量で目的を達成できる（図 9 (b)）．

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．また，データセンタの拠点数が増加すると，他のデータセンタに配置されるチャンクの数が増加するため，データセンタ間のデータ転送量が増加する．LDR によるデータセンタ間の転送量の削減効果については 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 [9] 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，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の数とした．各チャンクのキーには元のデータを put する際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．たと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による分散キーバリューストアの性能低下を軽減させることを示す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には，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 および 2 GB の主記憶を備えた計算機を用い，Linux 3.2 および 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を暗号化した．東京リージョンには 2 つのマイクロインスタンス（t1.micro）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（t1.micro）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，東京リージョン内およびリージョン間で PING メッセージと iperf-2.0.5 を用いて計測した．iPerf は TCP および 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．東京リージョン内および東京–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，東京リージョン内および東京–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分かった．

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．

また，個別に指定する場合を除き，ML-Chord のレイヤ数は $L = 2$，LDR のパラメータ設定は $(m, k) = (10, 5)$ とした．

5.1 データ探索の所要時間

ML-DHT がデータ探索に要する時間を削減することを示すために，実環境で Chord と ML-Chord それぞれの探索時間を比較した．ここでは，あるデータの担当ノードの探索処理を開始してから目的ノードを発見するまでに要した時間を探索時間と定義する．ML-Chord は ML-DHT の実装例である．**図 10** に探索時間の平均値の比較を示す．ML-Chord の探索時間は Chord より約 74%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させていたためである．

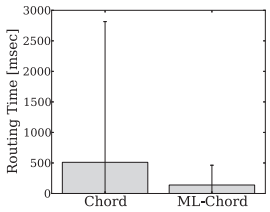


図 10　実環境における探索時間
Fig. 10　Routing time for each request in the real environment.

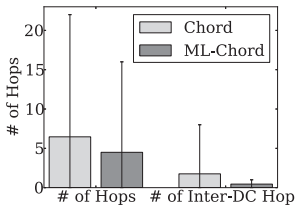


図 11　各探索処理におけるデータセンタ間ホップ数と総ホップ数
Fig. 11　The number of total/inter-datacenter hops for each routing process.

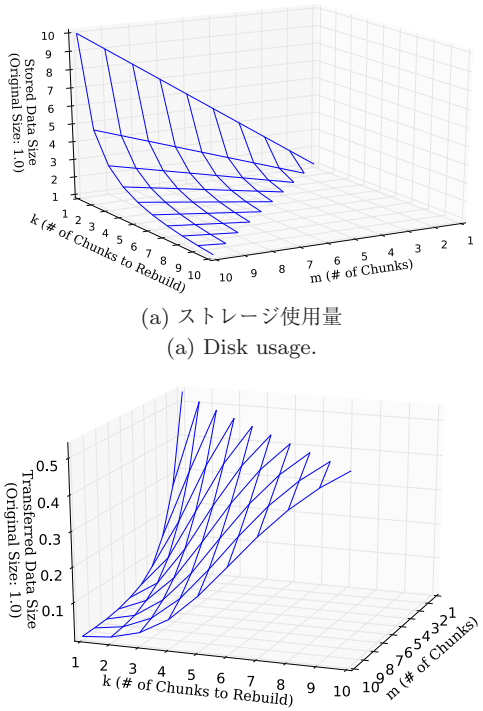
ML-Chord が実際にデータセンタ間ホップの回数を低減させていたことを示すために，同じワークロードを用いシミュレーション環境で Chord と ML-Chord での探索経路を解析した．**図 11** にデータセンタ間ホップ数および総ホップ数を示す．ML-Chord は Chord より約 74%少ないデータセンタ間ホップ数で目的のノードに到達していたことが分かった．そして，ML-Chord は Chord より約 30%少ない総ホップ数で目的のノードに到達していたことも分かった．総ホップ数も低減したのは，ローカル経路表を用いることで，スキップリスト [24] のようにグローバル経路表にのみ含まれるノードを飛ばして目的のキーへ近づく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をまたぐ分散キーバリューストアにおける探索時間は，データセンタ間の通信遅延の影響をより大きく受けることが分かる．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いっさい行っていないことが分かる．

さらに，本実験の条件設定と異なる場合における提案手法の効果を評価するため，ML-Chord と Chord におけるデータセンタ間ホップ数の期待値 $E_{\mathrm{ML-Chord}}$，E_{Chord} を算出した（式 (5)，(6)）．ここで，d はデータセンタの数，h は 1 回の探索処理に含まれるデータセンタ間ホップ数，r は探索経路長と定義し，簡単のために，各データセンタに属するノードの数は同じ，キー空間上におけるノードの分布は均一であるとする．E_{mlchord} は d にのみ応じて増加する．すなわち，ノード数が増加してもデータセンタの数が

増加しなければデータセンタ間ホップの機会が増加することはない．一方，E_{Chord} は d および r に応じて増加する．Chord において r はノード数 N に対し $\mathcal{O}(\log N)$ で増加するため，E_{Chord} はノード数とデータセンタ数に応じて増加すると言い換えることができる．したがって，データセンタ数に対しノード数が増加するほど，ML-Chord の Chord に対するデータセンタ間ホップ数削減効果が大きくなる．本実験の条件は $d = 2$，$N = 1000$ であったが，より多くのノードを利用する環境においては d に対し N がより大きく異なることが想定され，本手法による性能向上を見込むことができる．

$$E_{\mathrm{ML-Chord}} = \frac{\log d}{2} \tag{5}$$

$$E_{\mathrm{Chord}} = \sum_{h=1}^{r} {}_rC_h \left(\frac{1}{d}\right)^{r-h} \left(1 - \frac{1}{d}\right)^{h} h \tag{6}$$



(a) ストレージ使用量
(a) Disk usage.

(b) Get 操作にともなうデータセンタ間の平均データ転送量
(b) The average remote data transfer for each Get request.

図 12　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

Fig. 12　Disk usage and remote data transfer for parameters m and k.

5.2　データ転送量

LDR によって，キーバリューストアの管理者がストレージ使用量とデータセンタ間のデータ転送量を調整可能とな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 節および 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センタ間の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 12** (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，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転送量を細粒度に調整できることが分かった．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を

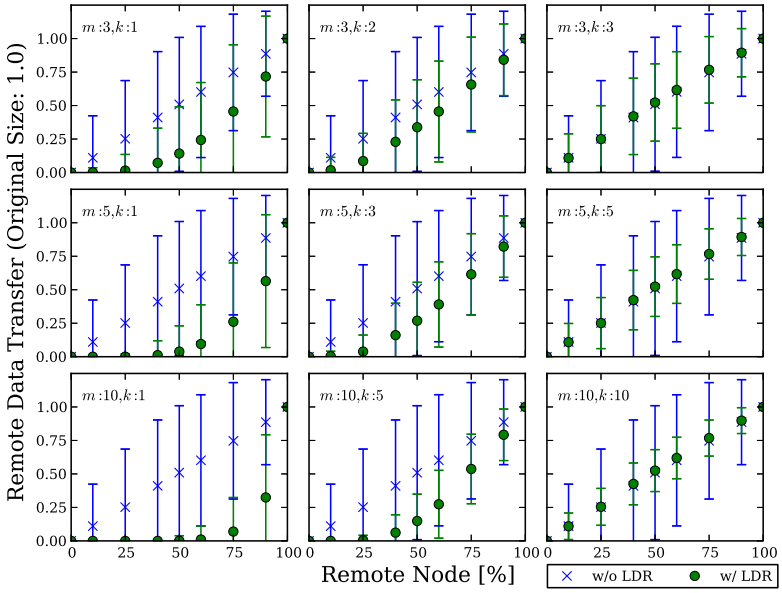


図 13　リモートデータセンタに属するノードの割合とリモートからのデータ転送量の関係

Fig. 13　Relationship between the proportion of nodes in remote datacenter and the amount of data transfer among datacenters for each Get request.

変えることで，管理ポリシ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，他方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転送量がどのように変化するかを比較した．**図 13** に結果を示す．他方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した．これは LDR が同じ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利用することで，他方のデータセンタからチャンクを取得する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した．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 らは Erasure Coding を用いた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データの冗長化による耐障害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．

クライアントユーザの地理的な位置を考慮したデータ移送手法 [25], [26] は，あるユーザが利用するデータを，そのユーザに対し地理的に近いデータセンタへ再配置することで，データ配信時の応答遅延を抑制する．これらの手法では，応答遅延を抑えてデータ配信を行う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の変化に応じて最寄りのデータセンタへユーザデータを動的に移送する．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，ユーザ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], [28], [29], [30] は，複数のデータセンタ間でストレージの状態を複製/同期しておくことで，リクエスト処理時の応答遅延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，これらの手法ではすべ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すべてのレプリカに対して適用する．すべ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．

拡張性の高い DHT アルゴリズムを構築する手法 Flexible Routing Tables（FRT）[31] は，各ノードの保持する経路表エントリを動的に整理することで，経路表エントリの増加を抑えつつ効率的な探索処理を実現する．FRT 自体には冗長なデータセンタ間通信を防ぐ仕組みは備わっていない．ML-DHT では基本 DHT に FRT を用い ML-FRT とすることが可能であるため，補完的である．ML-DHT において各ノードは各レイヤについて経路表を保持するため，FRT を用いることで各ノードが保持する経路表をより削減することができる．

FRT にグループの概念を導入した GFRT [31] では，各ノードにグループを割り当て，同グループのノードが経路表に多く存在するよう整理することで，冗長なグループ間通信の発生を防ぐ．すなわち，各データセンタをグループとして扱うことで，冗長なデータセンタ間通信の発生を防ぐことができる．一方 ML-DHT は，各ノードがそれぞれ 1 つのグループに属する GFRT と異なり，複数の包含関係にあるグループを同時に扱うことで，冗長なデータセンタ間通信の発生を防ぐだけでなく探索状況に応じ段階的に探索範囲を拡大できる．たとえば，図 6 (b) に示したように，ML-DHT ではデータセンタ内の探索のみでは目的のノードに到達できない場合に，段階的に地域，全体と探索範囲を拡大していくことができる．このような探索は，図 4 のようにグループ自体の分布にも偏りがある場合，特に有効である．反対に，グループが均一に分布している状況下では，複数のレイヤ（$L > 2$）を用いた探索手法は特に効果的ではない．また，ML-DHT では各レイヤについて経路表を持つため，不必要に L を大きくすることは無用な経路表サイズの増大につながる．したがって，階層的な探索が不要で経路表のサイズを最小限にしたい場合は GFRT が適しており，階層的な探索が効果的な場合は ML-DHT が適している．

7. まとめ

本論文では，データセンタ間通信を減らす要素技術として，*Multi-Layered DHT*（ML-DHT）と *Local-first Data Rebuilding*（LDR）およびこれらを用いて複数データセンタをまたぐ分散キーバリューストアを構築する手法を提案した．これらの手法はともに，データセンタ間をまたぐ分散キーバリューストアにおいてインターネット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またぐ通信が冗長に発生することを防ぐためローカル優先探索を行い，データ探索時における通信遅延の増大を抑制する．LDR は Erasure Coding を用いて保存するデータを冗長性を持ったチャンクに分割し，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データ探索に要する時間が 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] Rochwerger, B. et al.: Reservoir — when one cloud is not enough, *Computer*, Vol.44, No.3, pp.44–51 (2011).

[2] Chang, F. et al.: Bigtable: A Distributed Storage System for Structured Data, *Proc. USENIX Symposium on Operating Systems Design and Implementation* (2006).

[3] DeCandia, G. et al.: Dynamo: amazon's highly available key-value store, *Proc. ACM SIGOPS Symposium on Operating Systems Principles* (2007).

[4] Lakshman, A. and Malik, P.: Cassandra: A Decentralized Structured Storage System, *Proc. ACM SIGOPS Int'l Workshop on Large Scale Distributed Systems and Middleware* (2009).

[5] The Apache Software Foundation: Apache HBase, available from ⟨http://hbase.apache.org/⟩.

[6] 10gen, Inc.: MongoDB, available from ⟨http://www.mongodb.org/⟩.

[7] Oracle Corporation: Oracle NoSQL Database, available from ⟨http://www.oracle.com/technetwork/products/nosqldb/⟩.

[8] Microsoft Corporation: Windows Azure Table Storage Services, available from ⟨http://azure.microsoft.com/⟩.

[9] Shudo, K. et al.: Overlay Weaver: An overlay construction toolkit, *Computer Communications*, Vol.31, No.2, pp.402–412 (2008).

[10] Locher, T. et al.: eQuus: A Provably Robust and Locality-Aware Peer-to-Peer System, *Proc. IEEE Int'l Conference on Peer-to-Peer Computing* (2006).

[11] Lin, W.K. et al.: Erasure Code Replication Revisited, *Proc. IEEE Int'l Conference on Peer-to-Peer Computing* (2004).

[12] Dimakis, R.G. et al.: Network coding for distributed storage systems, *Proc. IEEE Int'l Conference on Computer Communications* (2007).

[13] Plank, J.S. et al.: A performance evaluation and examination of open-source erasure coding libraries for storage, *Proc. USENIX Conference on File and Storage Technologies* (2009).

[14] Stoica, I. et al.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, *Proc. ACM Special Interest Group on Data Communications Conference* (2001).

[15] Karger, D. et al.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web, *Proc. annual ACM symposium on Theory of computing* (1997).

[16] Ratnasamy, S. et al.: A scalable content-addressable network, *Proc. ACM SIGCOMM 2001 Conference on applications, technologies, architectures, and protocols for computer communications* (2001).

[17] Rowstron, A.I.T. and Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems, *Proc. IFIP/ACM Int'l Conference on Distributed Systems Platforms* (2001).

[18] Maymounkov, P. and Mazières, D.: Kademlia: A peer-to-peer information system based on the xor metric, *Proc. Int'l Workshop on Peer-to-Peer Systems* (2002).

[19] Zhao, B.Y. 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, *Proc. First Int'l Workshop on Peer-to-Peer Systems* (2002).

[21] Zooko Wilcox-O'Hearn: Zfec, available from ⟨http://pypi.python.org/pypi/zfec/⟩.

[22] Amazon Web Services LLC: Amazon Elastic Compute Cloud, available from ⟨http://aws.amazon.com/ec2/⟩.

[23] Amazon Web Services LLC: Amazon Virtual Private Cloud, available from ⟨http://aws.amazon.com/vpc/⟩.

[24] Pugh, W.: Skip lists: a probabilistic alternative to balanced trees, *Comm. ACM*, Vol.33, No.6, pp.668–676 (1990).

[25] Agarwal, S. et al.: Volley: automated data placement for geo-distributed cloud services, *Proc. USENIX Conference on Networked Systems Design and Implementation* (2010).

[26] Tran, N. et al.: Online migration for geo-distributed storage systems, *Proc. USENIX Annual Technical Conference* (2011).

[27] Corbett, J.C. et al.: Spanner: Google's Globally-Distributed Database, *Proc. USENIX Symposium on Operating Systems Design and Implementation* (2012).

[28] Li, C. et al.: Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary, *Proc. USENIX Symposium on Operating Systems Design and Implementation* (2012).

[29] Lloyd, W. et al.: Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS, *Proc. ACM Symposium on Operating System Principles* (2011).

[30] Sovran, Y. et al.: Transactional storage for geo-replicated systems, *Proc. ACM Symposium on Operating System Principles* (2011).

[31] Nagao, H. and Shudo, K.: Flexible Routing Tables: Designing Routing Algorithms for Overlays Based on a Total Order on a Routing Table Set, *Proc. 11th IEEE Int'l Conference on Peer-to-Peer Computing* (2011).

堀江 光　（学生会員）

2009 年慶應義塾大学理工学部情報工学科卒業．2011 年同大学大学院理工学研究科開放環境科学専攻修士課程修了．同年同後期博士課程に進学，分散システムおよびデータセンタの省電力化手法の研究開発に従事．同年株式会社ライトマークス代表取締役社長就任．IEEE/CS，ACM 各会員．

浅原 理人　（正会員）

2005 年電気通信大学電気通信学部情報工学科卒業．2007 年慶應義塾大学大学院理工学研究科開放環境科学専攻修士課程修了．2010 年同大学院理工学研究科開放環境科学専攻後期博士課程所定単位取得満期退学．日本電気株式会社グリーンプラットフォーム研究所を経て，現在 NEC Laboratories America, Inc. 勤務．博士（工学）．クラウドコンピューティング基盤における分散並列処理技術等，ミドルウェア領域の研究開発に従事．ACM，IEEE/CS，USENIX 各会員．

山田 浩史　（正会員）

2004 年電気通信大学電気通信学部情報工学科卒業．2009 年慶應義塾大学大学院理工学研究科開放環境科学専攻後期博士課程修了．同大学特任助教を経て，現在，東京農工大学先端情報科学部門准教授．博士（工学）．平成 20 年，21 年度情報処理学会論文賞，平成 20 年度山下記念研究賞受賞．仮想マシン技術，オペレーティングシステム等のシステムソフトウェアの研究に従事．ACM，USENIX，IEEE/CS 各会員．

河野 健二　（正会員）

1993 年東京大学理学部情報科学科卒業．1997 年同大学大学院理学系研究科情報科学専攻博士課程中退，同専攻助手に就任．博士（理学）．電気通信大学情報工学科講師等を経て，現在，慶應義塾大学理工学部情報工学科教授．2000 年度情報処理学会山下記念研究賞，1999，2008，2009，2012 年度情報処理学会論文賞，2014 年日本ソフトウェア科学会ソフトウェア論文賞．オペレーティングシステム，システムソフトウェア，ディペンダブルシステム等に興味を持つ．IEEE，ACM，USENIX 各会員．