Vol.2012-OS-122 No.7
2012/8/1

複数のデータセンタを跨ぐ伸縮性を備えたキーバリューストレージの実現手法

堀江 光[1]　浅原 理人[2]　山田 浩史[1]　河野 健二[1]

概要：データセンタ上で運用するサービスにおけるデータ管理基盤として分散型キーバリューストレージ（KVS）が広く用いられている．KVS は伸縮性及び可用性に優れているが，現状ではデータセンタ内での運用を想定して設計されており，その性能は単一のデータセンタの能力に限定されている．本論文では，複数のデータセンタ間を跨いだ伸縮性を備える KVS を実現するための手法を提案する．本手法では，複数の異なるデータセンタに属するノードを分散ハッシュ表を用いて管理し，単一の KVS を構築する．KVS を構成するノードの割合を制御することで各データセンタの負荷を制御することが可能である．また，多層化 DHT による無駄なデータセンタ間通信の排除，レプリカの利用による伸縮時の転送量削減により，データセンタ間を跨いだ通信はデータセンタ内の通信比べて高遅延かつ狭帯域であるという問題に対応する．評価実験として行ったシミュレーションでは，提案手法により構築した KVS が無駄なデータセンタ間通信を伴わずに運用可能であることを確認した．

Inter-Datacenter Elastic Key-value Storage

Hikaru Horie[1]　Masato Asahara[2]　Hiroshi Yamada[1]　Kenji Kono[1]

Abstract: Distributed key-value storages (KVSs) have been widely used as infrastructures for data management in datacenters. Although KVS is excellent in elasticity and availability, its performance is limited to the capacity of a single datacenter. This is because current KVSs have been designed for operation in a single datacenter. This paper presents a technique to make an elastic KVS which works on multiple datacenters. In the technique, each physical server in the datacenters works as a node of an overlay network and they are managed by a distributed hash table (DHT). We can control the load of each datacenter by controlling the ratio of running servers as the KVS's node. In addition, a multi-layered DHT eliminates unnecessary communication between datacenters and pre-replicated data objects reduce data amount which is transferred while the KVS extends or shrinks. These are especially effective to multiple datacenters environments because networks between datacenters are high-latency and narrow-band. In a simulation, we confirmed the technique eliminates unnecessary inter-datacenter communications.

1. 背景

データセンタ上で運用するサービスのデータ管理基盤として分散型キーバリューストレージ（KVS）が広く用いられている [2] [14] [4] [8] [1]．KVS は連想配列型データストレージの一種で，保存対象のデータオブジェクト (value) をキー (key) と呼ばれる識別子と対応づけて管理する．KVS は複数のストレージサーバ（ノード）で構成され，各ノードは担当するキー空間に含まれるキーに対応づけられたデータオブジェクトを保持する．

KVS は伸縮性と可用性に優れている．まず伸縮性という点において，KVS はノードの追加や削除が容易に行えるという特徴を持つ．この特徴は例えばコンシステントハッシングという手法で実現できる．コンシステントハッシングは，キー空間を連続した複数の部分空間に分割することで，キー空間とデータオブジェクトの委譲を隣接したキー空間を保持するノード間のみで完結させる．これにより，

[1] 慶應義塾大学
Keio University
[2] NEC クラウドシステム研究所
NEC Cloud System Research Laboratory

KVS 全体に影響を与えることなくノードの追加や削除が行える．また可用性という点において，KVS は一部のノードが障害で停止しても，他のノードのデータオブジェクトに影響を与えないという特徴を持つ．これは，KVS ではノードごとに独立したキー空間が対応づけられており，あるノードの障害が他のノードに影響を与えない構造になっているからである．

しかし，既存の KVS の伸縮性や可用性は，単一のデータセンタの能力に限定されるという制約がある．KVS は低遅延かつ広帯域で接続可能なノードで構成されることを前提に設計されることが多い．このため，KVS は単一のデータセンタで運用されることが一般的である．この運用では，データセンタの全てのノードを集めた以上の KVS は構築できないため，KVS の性能は単一のデータセンタの性能に制限される．また，あるデータセンタで構成されている KVS の負荷を他のデータセンタの KVS に分散することも，WAN を介したデータオブジェクトの移送が必要であるため容易ではない．さらに，データセンタ全体に影響するメンテナンスを実施する場合，既存の KVS では停止せざるを得ない．

本研究では，複数のデータセンタ間を跨いだ伸縮性を備える KVS の実現を目的とする．本研究で実現する KVS によって以下のことが可能となる．

- テナントサービスを停止せずに，KVS が受ける負荷を他のデータセンタに分散することができる．
- テナントサービスを停止せずに，KVS 全体を他のデータセンタへ移すことができる．

本論文では，複数のデータセンタ間を跨いだ伸縮性を備える KVS を実現するための手法を提案する．本手法では，複数の異なるデータセンタに属するノードを分散ハッシュ表（DHT）で管理し，単一の KVS を構築する．この KVS を構成するノードの割合を制御することで各データセンタの負荷を制御することが可能である．また，本手法ではデータセンタ間通信がデータセンタ内通信よりも高遅延かつ狭帯域であることを考慮し，DHT の多層化による無駄なデータセンタ間通信の排除，レプリカの事前配置による効率的な KVS の伸縮を実現した．

本論文の構成は以下の通りである．2 章では設計課題を整理する．3 章では，提案する KVS の設計について述べる．4 章では，提案する KVS の実装について述べる．5 章では，提案する KVS の有用性を確認するための評価実験について述べる．6 章で，本研究の関連研究について述べ，7 章で本論文をまとめる．

2. 設計上の課題

データセンタ間を跨いだ伸縮性を備える KVS を実現するために考慮すべき課題について整理する．データセンタ間を跨いだ伸縮性を備えるということは，構成ノードがそれぞれ異なるデータセンタに属する状況が，一時的または長期的に発生することを意味する．このような状況が発生する KVS の課題として，伸縮性，可用性，ノード検索の遅延，データ転送の負荷の 4 つを挙げる．

まず，伸縮性として，データセンタの負荷の状況やテナントサービス要求性能に応じて，複数のデータセンタに対してノードを増減できる必要がある．これにより，負荷の集中しているデータセンタから他のデータセンタへの負荷分散や，過剰な資源の停止やより安価な運用コストのデータセンタの利用による運用コストの削減が期待できる [6] [10].

2 点目に可用性として，KVS サービスを停止することなく継続的に運用できる必要がある．単一障害点が存在しない設計やノード故障に対応した設計を採ることにより，KVS の可用性を高めることができる．

3 点目として，データオブジェクトを保持するノードを検索する際に要する時間を短縮することが重要である．KVS ではデータオブジェクトの取得に当たり実際にそれを保持するノードを特定する必要があるが，データオブジェクトの検索に要する時間は，そのデータを利用するリクエストの応答時間に直結するためである．特に KVS を構成するノードが複数のデータセンタに存在する状況においては，データセンタ間の通信に注意しなければならない．なぜなら，データセンタ間の通信はデータセンタ内の通信に比べて遅延が大きいためである．

最後に，KVS の伸縮に伴うデータ転送が稼働中のテナントサービスの性能に悪影響を及ぼさないようにすることが重要である．KVS を伸縮させるということは各ノードが保持するデータオブジェクトの全部または一部が転送されることを意味するが，この転送のためのディスクアクセスやネットワーク帯域の消費がテナントサービスを逼迫してはならない．しかし一方で，時間あたりのデータ転送量を小さくすると転送に要する時間が長くなり，KVS の伸縮に長時間要してしまう．転送に要する時間と時間あたりのデータ転送量はトレードオフの関係にあるため，伸縮時以外にも一部の転送を行うこと等により，伸縮時の転送量を抑えるといった工夫が必要である．

3. 提案

本論文では，オーバレイネットワークを用いることでデータセンタ間を跨いだ伸縮性を備えた KVS を実現する手法を提案する．

3.1 概要

本手法では，分散ハッシュ表（DHT）を用い，複数のデータセンタのストレージサーバをノードとするオーバレイネットワークを構築し，各ノードの担当するキー空間を一括管理する．オーバレイへの参加ノード数が増加するこ

とで各ノードの担当キー空間が小さくなり，各ノードの負荷が低下し全体の容量も増加する．すなわち，ノードの追加及び離脱によって KVS 全体の性能を伸縮できる．追加及び離脱の際には，影響するキー空間を担当するノード間でデータオブジェクトの授受を行う．

3.2 伸縮性と高可用性

本手法では，KVS の構築に DHT を用いることで伸縮性と高可用性を実現する．

DHT は一般的に各ノードが担当するキー空間やそこに到達するための情報（e.g., IP アドレス）を断片的に保持している．これらの情報を断片的に扱うことで，ノードの数が膨大な場合でも各ノードが保持する情報を少なくすることができ，どのノードを起点にしても目的のデータオブジェクトを保持するノードを探索することが可能である．このような特徴により，DHT には特定のノードへの負荷集中が起こりづらいという利点がある．また，ノードの追加及び離脱時にキー空間へ与える影響を抑える手法が取り入れられる等，ノードの追加及び離脱に対して柔軟な設計がなされているものが多い [7].

DHT を用いて KVS のノードを管理することは，負荷集中の回避及びデータセンタの負荷制御の点で有用である．まず，データオブジェクトの探索によって特定のサーバに負荷が集中する状況を避けることができる．これは，各ノードの担当するキー空間の大きさは確率的に均一であり，各ノードの負荷も概ね均一であるためである．ノード分布の偏りやキー毎（データオブジェクト毎）の人気の偏りについては本研究の対象外であるが，これらを考慮した DHT と本手法は併用可能であり，補完的な位置付けである．また，あるデータセンタの負荷は，KVS を構成するノード全体に占めるそのデータセンタのノードの割合で制御することができる．これは先に述べたように，各ノードの負荷が概ね均一になることを利用している．例えば，KVS がデータセンタ A, B, C に跨がって構築され各データセンタのオーバレイへの参加ノード数比を 1:2:3 とすると，各データセンタの負荷の割合もほぼ 1:2:3 となる．

KVS の伸縮性は，各データセンタからオーバレイへの参加ノード数を変化させることで実現できる．図 1 に，提案手法によるデータセンタ間を跨いだ KVS の伸縮の例を示す．図 1 左はデータセンタ A に属する 6 つのノードから KVS が構築されている状況を示す．このとき全てのデータアクセスはデータセンタ A に対して行われるため，データセンタ A, B の負荷比率は 6:0 である．ここでデータセンタ B に属する 3 つのノードが追加されると図 1 右の状況へと遷移する．このとき元々データセンタ A のノードが担当していたキー空間及びデータオブジェクトの一部は追加されたノードへ委譲されるため，データセンタ A, B の負荷比率は 6:3 となる．すなわち，データセンタ A の負荷は 2/3 に軽減される．その後，データセンタ B のノードを離脱させると再び全てのデータアクセスがデータセンタ A に対して行われる状況に戻る．このようにしてデータセンタ間を跨いだ KVS の伸縮が実現される．またこの際，データセンタ B の代わりにデータセンタ A のノードを離脱させることで，KVS をデータセンタ B へ完全に移行することも可能である．

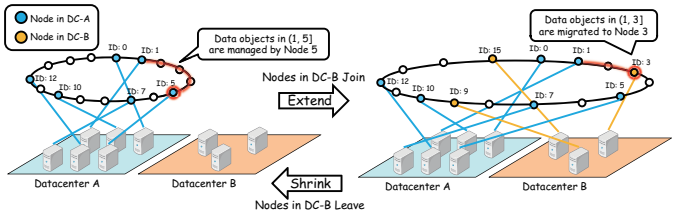

図 1 提案手法によるデータセンタ間を跨いだ KVS の伸縮

KVS の可用性は，DHT によるノード管理により向上することができる．ノード管理用サーバ等の単一障害点になり得る仕組みを導入せず， DHT によって完全分散管理することで，あるノードに障害が発生した際にも KVS 自体は利用可能な状態を継続することができる．これは，あるノードに障害が発生した場合でも目的のノードを探索できるように経路情報に冗長性を持たせていることや，各ノードの担当するキー空間が独立しており障害が他のノードに影響を及ぼさないことによって実現されている．

3.3 遅延を抑えたノード検索

本手法では，DHT を多層化して用いることでデータオブジェクトの探索の際に発生する無駄なデータセンタ間通信を無くす．

まず，単純な DHT を用いた場合，ノードの参加状況によって無駄なデータセンタ間通信が発生することについて説明する．単純な DHT が無駄なデータ通信間通信を発生させるのは，オーバレイが物理的なネットワークトポロジを隠蔽しているためである．すなわち，オーバレイのトポロジは物理的なネットワークトポロジと独立して構成されており，オーバレイ上では隣接しているノード間の距離が地理的には極めて離れているという状況が生じてしまう．図 2 に，Chord [13] を用いた場合に，ノード探索時に無駄なデータセンタ間通信が発生する例を示す．この例は，データセンタ A, B に属するノードによって構成されている KVS において，ID: 1 のノードが ID: 14 に属するデータオブジェクトを探索する際のルーティングの様子を示している．Chord は DHT を実現する代表的なアルゴリズムの一種であり，一次元リング状のキー空間を持つ，ノード数 n に対し $O(\log n)$ 回のホップ数で目的のノードに辿り着くことが可能である，といった特徴を持つ．このときわざわざデータセンタ A, B のノードを交互にホップして，より遅延の大きなデータセンタ間通信を繰り返しながら目的のノードまでのルーティングが行われている．

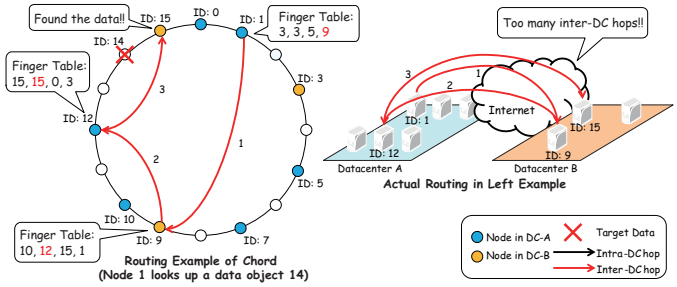

図 2 単純な DHT にて無駄なデータセンタ間通信が発生する例

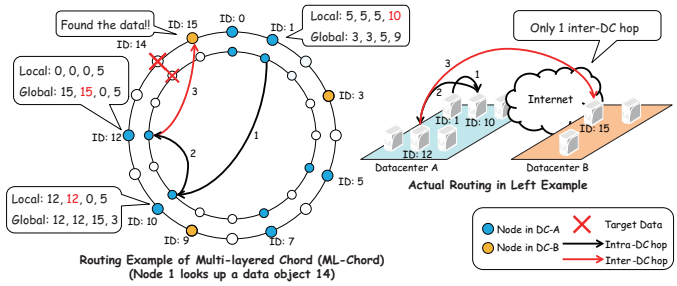

図 3 多層化 DHT にて無駄なデータセンタ間通信を排除した例

次に，本手法で用いる多層化 DHT によって先の例に挙げたような無駄なデータセンタ間通信が発生しないことについて説明する．多層化 DHT では，ルーティングの際のデータセンタ間ホップは可能な限り先送りにされ，同じデータセンタ間を往復するような無駄は一切発生しなくなる．このようなルーティングを実現するために，多層化 DHT ではオーバレイを構築する際に属するデータセンタを考慮し経路表を作成する．そして，この経路情報を用いて同一データセンタノードを優先的に選択してホップする．ノードの所属データセンタを考慮しないルーティングのために無駄なデータセンタ間通信が生じる通常の DHT とはこの点で大きく異なる．

ここでは，同一データセンタに属するノードのみを対象としたローカル層，所在によらず全てのノードを対象とするグローバル層からなる二層構造の場合について説明するが「データセンタ」「地域」「全て」というようにより多くの層からなる構成も実現可能である．ノードを追加する際，ブートストラップノードが同一データセンタのノードであるならローカル層とグローバル層で，別のデータセンタのノードであるならグローバル層でのみ追加処理を行う．このようにすることで，各ノードは自分と同じデータセンタに属するノードのみを含む経路表（ローカル層）と全体の経路表（グローバル層）を持つことができる．このとき，グローバル層は通常の DHT と全く同じものとなる．ルーティングを行う際には，各ノードはローカル層の情報を優先的に利用し，より目的のノードに接近できる場合のみグローバル層の情報を用いたホップを行う．こうすることで，ルーティングの際のデータセンタ間ホップは可能な限り先送りにされ，同じデータセンタに戻るホップは一切発生しない．

尚，各層におけるホップ先の選択は通常のルーティングアルゴリズムに準拠する．図 3 に，Chord を多層化して用いた場合のルーティングの例を示す．この例でも，データセンタ A, B に属するノードによって構成されている KVS において，ID: 1 のノードが ID: 14 に属するデータオブジェクトを探索する際のルーティングの様子を示している．ノード 1, 10 はローカル層にてより目的の ID 14 に近いノードを発見できているためこちらを用いている．一方ノード 12 はローカル層での候補としてノード 0 を挙げているが，グローバル層にてより近いノード 15 を発見しているため，こちらをホップ先として確定している，今回のルーティングではこの時点で初めてかつ唯一のデータセンタ間通信を伴うホップを行っており，無駄なデータ通信間ホップを行っていないことが確認できる．このように，本手法では DHT を多層化して属するデータセンタの情報を持たせることで，無駄なデータセンタ間ホップを排除し，データセンタ間通信の大きな遅延によるノード探索の遅延増加を防いでいる．

3.4 サービスへの悪影響を抑えたデータ転送

本手法では，KVS の伸縮を行っていない段階から各データオブジェクトのレプリカを配置しておくことで，伸縮時のディスクアクセスやネットワーク帯域の消費を抑え，同時に伸縮に要する時間も削減することを可能とする．2 章で述べたように，KVS の伸縮はテナントサービスへ悪影響を及ぼさず行われなければならないが，影響を抑えるために転送速度を遅くすると KVS の伸縮に時間がかかり伸縮性を大きく損なってしまう．これは伸縮時に全てのデータオブジェクトを転送しようとすると直面する問題である．そこで本手法では，伸縮を行っていない時点からレプリカを配置しておき，伸縮時はレプリカに更新が反映されていないデータオブジェクトのみを転送することでデータ転送量を抑える．これによりディスクアクセス及びネットワーク帯域消費を抑えながら同時に迅速な伸縮を実現する．

図 4 にノード追加時の動作の様子を示す．ノード追加時，新しく追加されたノードはまだ実体を持っていないデータオブジェクトのリクエストを受けることがある．この際そのノードはクライアントに代わってレプリカを取得し，それを自身のデータオブジェクトとして保持することで必要になったデータオブジェクトから移送を進めることができる．リクエストを受けていないデータオブジェクトは，追加ノード及びレプリカ保持ノードの負荷に注意しながら移送を進めることで対応する．

また，図 5 にノード離脱時の動作の様子を示す．ノード離脱時，各ノードは担当する範囲のデータオブジェクトのうち，変更が加えられかつレプリカに反映されていないものについてだけ反映されるのを待てば良い．担当キー空間に属するデータオブジェクトを一斉に移送する必要が無く

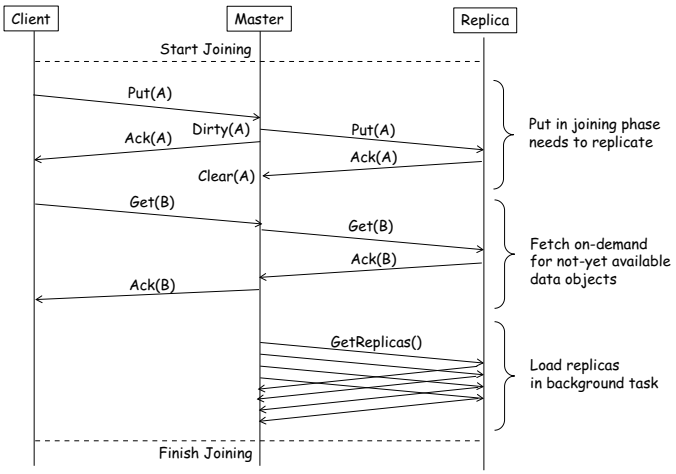

図 4 ノード追加時のデータオブジェクト移送の様子

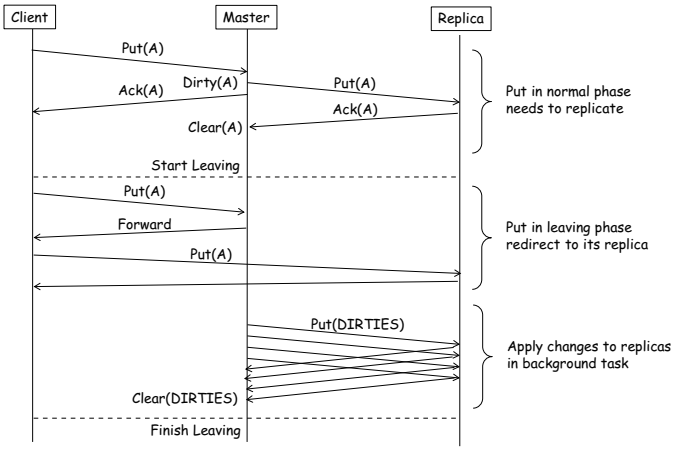

図 5 ノード離脱時のデータオブジェクト移送の様子

なるため，ノード離脱に伴う突発的な大量のデータ転送の発生を防ぐことができる．

このように，本手法では事前のレプリカ配置により，KVS の伸縮時に伴うデータ転送量や転送に要する時間を制御を可能としている．

4. 実装

Chord を多層化し（Multi-layered Chord, ML-Chord），OverlayWeaver [11] 上に実装した．図 6 にアーキテクチャの概要を示す．OverlayWeaver はオーバレイ構築ツールキットで，作成したルーティングアルゴリズムの動作検証等を容易に行うことができる．本実装において層の構成は 3 章の説明と同様，ローカル層及びグローバル層の 2 つとした．Chord では経路表として，Successor，Finger Table を保持する．Successor はキー空間上で隣接するノードを指し，Finger Table はより少ないホップ数で目的のノードに到達するために複数のノードの情報を含む表である．ML-Chord では各層について Successor, Finger Table を保持する．尚，グローバル層の Successor, Finger Table は通常の Chord と全く同様である．

提案手法では，無駄なデータセンタ間通信が発生しな

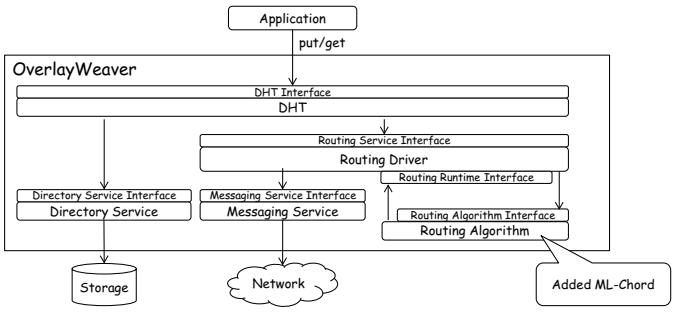

図 6 アーキテクチャ

```
// node n has successor and finger for each layer
n.successors = {global_successor, local_successor};
n.fingers = {global_finger, local_finger};

// ask node n to find id's successor
n.find_successor(id)
    n' = find_predecessor(id);
    return n'.successors[0]; // 0 means the global layer

// 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 = k downto 0  // k is # of layers (0 means the global layer)
        for i = m - 1 downto 0 // m is # of finger's entries
            if (fingers[l][i].node in (n, id))
                return fingers[l][i].node;
    return n;
```

図 7 ノード探索の擬似コード

いようにローカル層の探索結果を優先的に利用する．図 7 に，ML-Chord におけるノード探索の擬似コードを示す．通常の Chord と異なるのは複数の経路表を管理すること及びローカル層を優先する点で，それ以外は通常のアルゴリズムを流用することが可能である．DHT を実現するアルゴリズムは多々存在するが，ホップしながら対象との論理距離を詰めていくタイプのものであれば Chord と同様に少ない改変で多層化でき，多層化によりデータセンタ間通信を避けることが可能である．

提案手法では，レプリカの事前配置により伸縮時のデータ転送量を抑える．ML-Chord の実装では，レプリカを Successor に配置することでこれを実現している．あるノードがオーバレイから離脱するとその担当キー空間は Successor に引き継がれるため，レプリカの配置先を Successor にしておくことで，ノード離脱時にレプリカの所在が不明になってしまう自体を避けられる．

5. 評価

データセンタ間を跨いだ KVS において，提案手法が無駄なデータセンタ間通信の発生を防ぐことを確認するために OverlayWeaver を用いてシミュレーションによる評価を行った．シミュレーションでは，2 つのデータセンタを想定し，各データセンタに 500 台ずつのノードが存在する環境とした．この環境で，ランダムに生成したキーによる GET/PUT を各ノードから実行し，目的のノードに到達す

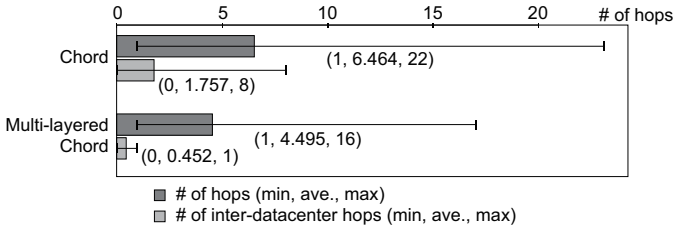

図 8 ノード探索におけるホップ数の比較

るまでにデータセンタ間を跨ぐホップが何回発生するかを調べた．本実験では 2 つのデータセンタしか存在しないため，データセンタ間を跨いだホップは多くとも 1 回で十分であり，これ以上の場合は無駄にデータセンタ間を往復していることになる．

本実験の結果を図 8 に示す．ML-Chord におけるデータセンタ間ホップ数の平均値は Chord に比べ約 75 % 少なかった．また，Chord では一度のルーティングにおいて最大 4 往復ものデータセンタ間通信が発生するケースもあったのに対し，データセンタ間を往復するような無駄な通信は一切発生していなかった．すなわち，提案手法により無駄なデータセンタ間通信を無くすことができると示された．

6. 関連研究

Zephyr [5], Albatross [3] はストレージを対象としたライブマイグレーション手法である．Zephyr は Shared Nothing 型ストレージを対象としており，データオブジェクトの実体の転送を想定した手法である．移送に際してはどのようなデータが存在するかという情報のみを最初に転送し，その後は移送先のノードで必要になったデータオブジェクトから順次移送する．これによりノード切替時のデータ転送量は少なく済むため，サービスの停止時間もわずかで済む．Albatross は Shared Storage 型ストレージを対象としており，データオブジェクトそのものではなく，それらを扱うデータベースインスタンスの移送を想定としている．この手法では，データベースインスタンスの状態やキャッシュを移送することで，移送に伴うトランザクション処理の失敗や移送後の性能低下等を避けることができる．これらはいずれも，ストレージに伸縮性を持たせるための手法であるが，複数のデータセンタ間を跨いだ環境での利用は想定しておらず，本研究とは目的が異なる．

また，COPS [9], Walter [12] はデータセンタ間を跨いだストレージ複製手法である．各データセンタにおけるトランザクション処理を非同期的に他のデータセンタへ反映させることで，性能と可用性のバランスを取る．通常，ストレージのレプリカを作成する場合，完全な一致を求めるには全ての処理がレプリカへ反映されたことを確認するまでその後の処理を待機する必要がある．しかしレプリカが遠方に存在する場合，この確認処理は大きな時間をを要するためサービスの著しい性能低下に繋がってしまう，これらの手法は，データセンタ内で済む処理とデータセンタ間での同期が必要な処理を切り分けることで，可用性と一貫性を確保しつつ性能低下も防ぐことを目的としているが，負荷や要求性能に応じてノードを増減させるような伸縮性の実現を目指したものではなく，本研究とは目的が異なる．

7. まとめと今後の予定

データセンタ間における伸縮性を持つ KVS を実現する手法を提案した．本手法では各ノードを DHT により管理し，ノードの追加及び離脱により各データセンタの負荷を制御することが可能である．また，DHT を多層化することでより遅延の大きなデータセンタ間通信が無駄に発生することを防いでおり，効率的なデータの取得が可能となっている．

KVS の伸縮に伴うデータ転送量や伸縮に要する時間については現在評価を進めている．また，シミュレーションではなく実環境で利用可能な形で提案手法を実装し．KVS の伸縮がテナントサービスに及ぼす影響等について評価をする予定である．

参考文献

[1] 10gen, Inc.: MongoDB. `http://www.mongodb.org/`.
[2] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R. E.: Bigtable: A Distributed Storage System for Structured Data, *Proc. of 7th USENIX Symposium on Operating Systems Design and Implementation* (2006).
[3] Das, S., Nishimura, S., Agrawal, D. and El Abbadi, A.: Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration, *Proc. VLDB Endow.*, Vol. 4, No. 8, pp. 494–505 (2011).
[4] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W.: Dynamo: amazon's highly available key-value store, *Proc. of twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (2007).
[5] Elmore, A. J., Das, S., Agrawal, D. and Abbadi, A. E.: Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms, *Proc. of ACM's Special Interest Group on Management Of Data* (2011).
[6] Heller, B., Seetharaman, S., Mahadevan, P., Yiakoumis, Y., Sharma, P., Banerjee, S. and McKeown, N.: ElasticTree: saving energy in data center networks, *Proc. of the 7th USENIX Conference on Networked Systems Design and Implementation* (2010).
[7] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M. and Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web, *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (1997).
[8] Lakshman, A. and Malik, P.: Cassandra: A Decentralized Structured Storage System, *Proc. of 3rd ACM SIGOPS Int'l Workshop on Large Scale Distributed*

Systems and Middleware (2009).

[9] Lloyd, W., Freedman, M. J., Kaminskyy, M. and Andersen, D. G.: Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS, *Proc. of 23th ACM Symposium on Operating System Principles* (2011).

[10] Qureshi, A., Weber, R., Balakrishnan, H., Guttag, J. and Maggs, B.: Cutting the electric bill for internet-scale systems, *Proc. of the ACM SIGCOMM 2009 Conference on Data Communication* (2009).

[11] Shudo, K., Tanaka, Y. and Sekiguchi, S.: Overlay Weaver: An overlay construction toolkit, *Computer Communications*, Vol. 31, No. 2, pp. 402–412 (2008).

[12] Sovran, Y., Power, R., Aguilera, M. K. and Li, J.: Transactional storage for geo-replicated systems, *Proc. of 23th ACM Symposium on Operating System Principles* (2011).

[13] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F. and Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, *Proc. of ACM Special Interest Group on Data Communications Conference* (2001).

[14] The Apache Software Foundation: Apache Hadoop. `http://hadoop.apache.org/`.