

推薦論文

Range-key Skip Graph による範囲検索可能な 大規模分散キーバリューストアの実現

石 芳正^{1,a)} 寺西 裕一^{1,2} 吉田 幹³ 竹内 亨⁴ 下條 真司^{1,2}

受付日 2011年9月16日, 採録日 2012年4月2日

概要: オーバレイネットワーク Range-key Skip Graph を用いることにより大規模環境においてもスケラブルな範囲検索を実現可能とする分散キーバリューストアの構成法を提案する. Skip Graph 拡張の 1 つである Multi-key Skip Graph を用いれば範囲検索可能な分散キーバリューストアを構成できるが, この方法では 1 つのキーごとに個別の経路表を持つため, キー数に比例して経路表のサイズが増加し, ノードの参加離脱時にオーバーヘッドが大きくなるという問題がある. そこで本研究では, ノードが担当するキーの集合を範囲キーとして扱い, Range-key Skip Graph によりルーティングを行うことで経路表を簡素化する方法を提案する. シミュレーションと実機による評価を行い, 本提案により検索コストを Multi-key Skip Graph と同等に保ちつつ, 経路表を大幅に簡素化可能であること, ならびにその性能がスケールすることを示した.

キーワード: 大規模分散 Key-value ストア, クラウド, オーバレイネットワーク, Skip Graph, 範囲検索

An Implementation of Large Scale Distributed Key-value Store with Range Search Feature Based on Range-key Skip Graph

YOSHIMASA ISHI^{1,a)} YUUICHI TERANISHI^{1,2} MIKIO YOSHIDA³
SUSUMU TAKEUCHI⁴ SHINJI SHIMOJO^{1,2}

Received: September 16, 2011, Accepted: April 2, 2012

Abstract: In this paper, a distributed range search supported key-value store construction method that by utilizing Range-key Skip Graph (RKSG) is proposed for applying a large-scale environment. Using Multi-key Skip Graph (MKSG) is a straightforward way to construct distributed key-value store with range search support. However, this method need to have a routing table for each key, therefore size of the routing table increases in proportion to the number of keys. When the enormous number of data should be managed in the store, the management cost of routing table would be crucial. Therefore, we propose a method to simplify a routing table by handling set of keys in charge as a range key in RKSG. Our simulation results and evaluation in the large-scale environment show that our proposed method is scalable and the size of routing table can be greatly reduced, while the cost of search is almost the same with MKSG.

Keywords: large scale distributed key-value store, cloud computing, overlay network, Skip Graph, range search

¹ 大阪大学サイバーメディアセンター
Cybermedia Center, Osaka University, Ibaraki, Osaka 567-0047, Japan

² 独立行政法人情報通信研究機構
National Institute of Information and Communication Technology, Koganei, Tokyo 184-8795, Japan

³ 株式会社ビービーエール
BBR Inc., Osaka, 530-0002, Japan

⁴ 日本電信電話株式会社未来ねっと研究所
NTT Network Innovation Laboratories, NTT Corporation, Yokosuka, Kanagawa 239-0847, Japan

^{a)} ishi.yoshimasa@ais.cmc.osaka-u.ac.jp

1. はじめに

今日のインターネットの普及により, 世界中の人々を対象とした Amazon や Google, Facebook, Twitter といった事業者が現れ, 数千万人から数億人に及ぶユーザにサービスを提供している. これらの大規模サービスでは, サー

本論文の内容は 2010 年 9 月のマルチメディア通信と分散処理研究会で報告され, 同研究会主査により情報処理学会論文誌ジャーナルへの掲載が推薦された論文である.

ビスの成長にともない増加する保存データ量やトラフィックに追従するため、サービスを支える基幹プラットフォームには継続的な性能強化が必要であり、スケーラビリティが要求される。しかしながら、旧来の集中型システムでとられていた負荷の増加にあわせて高性能な機材に置き換えてゆくスケールアップ戦略では、コストに見合う性能が得られず、スケーラビリティの確保が困難になっている。このため、多数の安価な機材に処理を分散するスケールアウト戦略がとられる事例が増えてきている。

ストレージシステムに注目すると、データの基本要素をキーと値のペア（キーバリューストア）としてこれらの要素に対する操作を GET や PUT といった基本的な操作に限定する代わりに、スケールアウトさせやすい仕組みを持つキーバリューストアが様々なサービスで採用されている。

スケールアウトを前提としたキーバリューストアとして、Amazon Dynamo [1] や Apache Cassandra [2], ROMA [3], Flare [4], kumofs [5] があげられる。これらのシステムでは、応答時間を抑える、もしくは保証するため、Consistent-hashing [6] に基づきキーバリューストアを構成するサーバ群の中から 1-hop (no-hop, zero-hop と呼ばれる) で目的のデータを持つ担当サーバに到達できる仕組みとなっている。これらの 1-hop に基づくシステムは数百～千台程度の規模を前提としたものが多い。しかし、より多数のサーバから構成される大規模キーバリューストアではサーバの故障発生頻度も増えるため、サーバ故障による影響についても考慮しなければならなくなる。サーバ故障が生じた場合、一般には故障サーバへの経路情報を取り除き、迂回経路を設定するなどの回復処理が必要になる。1-hop に基づくキーバリューストアでは、クライアントもしくはゲートウェイサーバが直接担当サーバにアクセスできるため低遅延での応答が期待できるが、その一方でクライアントやゲートウェイサーバはキーバリューストアを構成している全サーバへの経路情報を持たなければならない。このため、あるサーバが故障した場合、すべてのクライアントやゲートウェイサーバの経路情報から故障サーバの情報を削除し、その内容を更新しなければならない。

Chord [7] や Kademlia [8] などの分散ハッシュテーブル (DHT; Distributed Hash Table) では、検索クエリがサーバ間で自律的に転送されることで目的のデータを持つ担当サーバに到達するマルチホップによる手法を採用している。Chord を改良した FRT-Chord [9] では、Chord が持つ経路表の制約を緩和することでマルチホップを基本としながらも、サーバ数が少ない状態では 1-hop で動作し、担当サーバまでのホップ数を削減している。また、DHT を基盤とし、実用に耐えうるキーバリューストア実装や評価も行われつつある。たとえば、Microsoft Azure [10] は、Chord を拡張し経路表を両方向とした DHT を基盤にしたストレージサービスを実現している。Scalaris [11] は

Chord [12], [13] を基盤にした Erlang によるキーバリューストア実装、SandStone [14] は KBR [15] の 1-hop 拡張に基づく C++ によるキーバリューストア実装であり、高いスケーラビリティなどが評価として示されている。マルチホップを採用するキーバリューストアでは担当サーバまでのホップ数に応じて応答時間が変化し、1-hop に基づくキーバリューストアと比較して応答に時間を要する場合があるが、クライアントやゲートウェイサーバはキーバリューストアを構成している一部のサーバへの経路情報を持つだけでよい。よって、故障が生じた場合、故障サーバへの経路情報を持つサーバは一部のサーバに限られ、回復処理はそれら一部のサーバが故障サーバを切り離すように経路情報を更新するだけでよく、回復処理に要するコストは 1-hop に基づくシステムより抑制されると期待できる。

一方、キーバリューストアの機能面に注目すると、Twitter では Twitter クライアントが Web API により新しいタイムラインを取得する際に「ある ID の発言より新しい発言の取得」という操作が頻繁に生じており、この操作には範囲検索が有効であると考えられる。しかしながら、既存のほとんどのキーバリューストアが前提とする Consistent-hashing では、与えられたキーに対してハッシュ処理を行ってからデータを格納するため、元のキーの順序関係が保存されず範囲検索をサポートしていない。逆に範囲検索が可能であれば、指定 ID 以降の発言データを 1 度の検索要求でまとめて取得することができ、キーバリューストアへの問合せ頻度を削減できる。このほかにも「ある期間のイベントログ情報」や「位置に紐付けられた写真情報」といった範囲検索が必要となる場面は多い。しかし、1-hop, マルチホップいずれに基づくキーバリューストアも、従来範囲検索をサポートするものはなく、スケールアウトが困難であった。

ここでオーバレイネットワーク技術に目を向けると、Skip Graph [16] やこれを拡張した Multi-key Skip Graph [17], Range-key Skip Graph [18] など、多数のサーバをマルチホップにより連携させ、範囲検索を実現している手法が存在する。本研究では、範囲検索をサポートし、スケールアウトにも対応できる Range-key Skip Graph による分散キーバリューストアを提案する。提案機構を実装したシステムを大規模テストベッド StarBED に適用して実機上で評価を行い、サーバ数に比例して性能を向上させることができることを確認した。

2. 範囲検索可能なキーバリューストア

2.1 分散キーバリューストアモデル

分散キーバリューストアの扱うデータは、キー (Key) と値 (Value) であり、ユーザ (クライアント) がキーを指定して、対応する値の入出力を行う。値は、複数のノード (Node) に分散して保存される。

図 1 は分散キーバリューストアのモデルを示している。

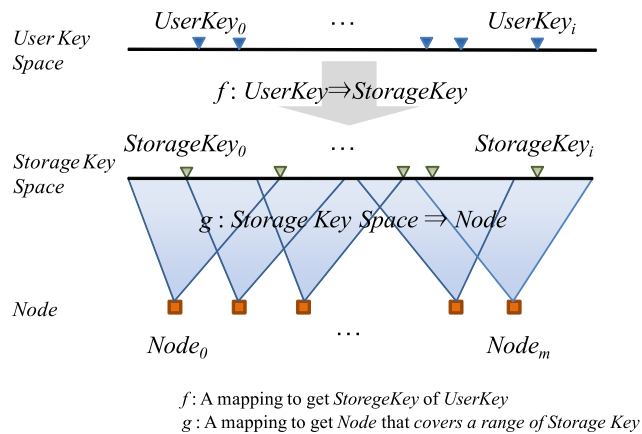


図 1 分散キーバリューストアモデル

Fig. 1 A model of distributed key-value store.

Storage Key Space は、分散キーバリューストアを構成するノード群が取り扱うキー (Storage Key) の値域を表している。分散キーバリューストアでは、各ノードが Storage Key Space 上の特定の領域を担当し、値の分散管理を行う。すなわち、Storage Key Space からノードへのマッピング g が必要となる。

User Key Space は、キーバリューストアにおいてユーザが指定するキー (User Key) の値域を表すキー空間である。User Key Space と Storage Key Space は必ずしも一致しない。すなわち、User Key Space から Storage Key Space へのマッピング f が必要となる。

通常、分散キーバリューストアでは、ノードの出入りがあっても値を返却できるよう、単一の User Key に対応する値を複数のノードに分散して持たせる。よって、 g は、通常 Storage Key から複数のノードへのマッピングとなる。

Chord, Kademlia などの代表的な分散ハッシュテーブルをこのモデルにあてはめると、User Key Space がキー、 f がハッシュ関数となり、Storage Key Space がハッシュ値の空間に対応する。 g には、ノードの ID に f と同一のハッシュ関数を適用して区切られた領域と当該 ID を持つノードのマッピングが対応する。各ノードにおいて、ノードの割当て領域外のハッシュ値を探し出すためにルーティングアルゴリズムが存在し、Chord では successor list, finger table によるマルチホップ検索が用いられる。

従来の分散キーバリューストアが持つ API (入出力インタフェース) は、User Key に対応する値を格納する PUT, 単一の User Key に対応する値を取得する GET が基本である。PUT(Ku, V) を実行した場合、 $g(f(Ku))$ により得られるノード集合に、 Ku, V を対応付けて格納する。GET(Ku, V) を実行すると、 $g(f(Ku))$ により得られるノード集合から、 Ku に対応付けられた V を取得する。

2.2 分散キーバリューストアにおける範囲検索

ここで、分散キーバリューストアの API として、指定範

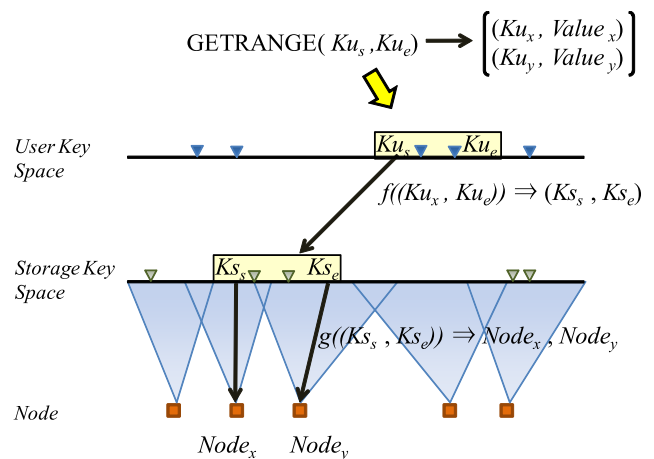


図 2 分散キーバリューストアにおける範囲検索

Fig. 2 Range search of distributed key-value store.

囲 $[Ku_s, Ku_e]$ 内にあるキーバリューストアをすべて取得する範囲検索を行う GETRANGE(Ku_s, Ku_e) を追加することを考える。

f がハッシュ関数のように順序が保存されないマッピングである場合、範囲 $[Ku_s, Ku_e]$ に対応する Storage Key Space 上の範囲を得ることができない。よって範囲検索を行うには、 f は順序が保存されるマッピングである必要がある。また、 g としては、 $[f(Ku_s), f(Ku_e)]$ に対応するノードをすべて取得する必要がある、その数が膨大とはならないマッピングであることが望ましい。

範囲検索 GETRANGE(Ku_s, Ku_e) を実行した場合、 $[f(Ku_s), f(Ku_e)]$ より得られる Storage Key Space 上のキー範囲 $[Ks_s, Ks_e]$ がマッピングされたノード集合 ($Node_x, Node_y$) から $[Ku_s, Ku_e]$ に含まれるキーバリューストアをすべて取得する (図 2)。

3. Multi-key Skip Graph による分散キーバリューストア

前章の g として、 $[f(Ku_s), f(Ku_e)]$ に対応するノードをすべて取得することができるルーティング方式には、Multi-key Skip Graph がある。以下では、Multi-key Skip Graph について概説する。

3.1 Multi-key Skip Graph

Multi-key Skip Graph は Skip Graph に拡張を行ったオーバレイネットワークであり、単一のノード上に複数のキーを保持可能としたオーバレイネットワークである。ここではまず、基本アルゴリズムである Skip Graph を用いて説明する。

Skip Graph は Skip List [19] を分散環境に適用し、構造化オーバレイネットワークとしたものであり、分散環境下で目的とするキーを持つノードを発見できる。分散ハッシュテーブルとは異なり、元のキーをハッシュ処理せずに

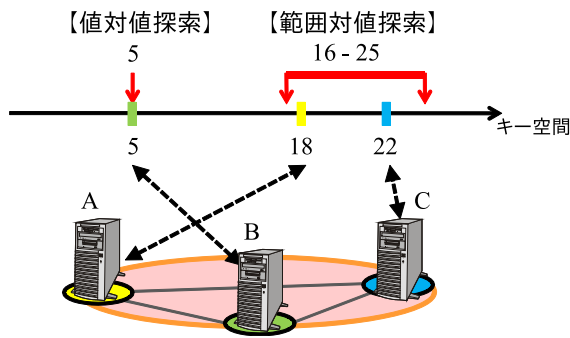


図 3 Skip Graph における検索例

Fig. 3 An example of searching keys on Skip Graph.

オーバーレイネットワーク上で扱い、範囲検索を可能としている。このキーは全順序関係を定義可能であればどのような要素であっても利用可能である。

Skip Graph では次の 2 種類の検索をサポートしている。前者は分散ハッシュテーブルと同様の完全一致検索、後者が範囲検索になる。

- ある値を指定して、その値を持つキーを検索する値対値検索
- ある範囲を指定して、その範囲に含まれるキーを検索する範囲対値検索

すなわち、検索クエリを Q としたとき、以下の 2 条件のいずれかを満たす Skip Graph 上のキーを K_i が発見される (Q_s は検索クエリ Q が持つ範囲の最小値, Q_e は最大値を表す)。

- $K_i = Q$
- $Q_s \leq K_i \leq Q_e$

図 3 に Skip Graph で実現される検索例を示す。図中下部は Skip Graph を構成する各サーバを示しており、それぞれのサーバがキーを持ちそれらがキー空間上に配置されている様子を示している。この例ではサーバ A がキー 18、サーバ B がキー 5、サーバ C がキー 22 を持ち、それぞれ Skip Graph の 1 ノードとして動作している。任意のサーバよりキー 5 を検索した場合、該当するキーを持つサーバ B が発見される。同様に 16 から 25 の範囲指定により検索した場合、範囲内に含まれるキー 18 とキー 22 を持つサーバ A とサーバ C が発見される。

続いて Skip Graph オーバレイの構造について概説する。オーバレイの構造は図 4 となっており、図中の縦に並んだ正方形の組がノード上に保持されるキーを表し、中の数値がキーの値を表している。個々の正方形はノードが持つ経路表の 1 エントリを表し、正方形間を横に結んだ矢印がキー間の経路を表している。経路表の各エントリはそれぞれ図中左側に示したレベルに属し、レベルごとに隣接するキーの値とそのキーを保持しているノードへの経路情報を持つ。各々のノードは、オーバレイへの参加時にキーの順序関係に従い、その順序を崩さない場所に参加する。

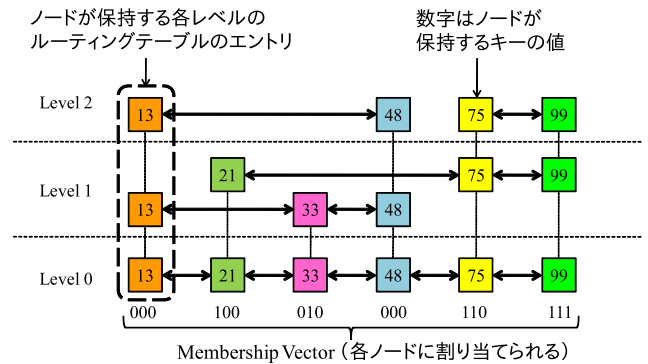


図 4 Skip Graph の構造

Fig. 4 The structure of Skip Graph.

各キーの下部に示した 2 進数値は membership vector であり、各レベルにおいて隣接とするキーを決定するために使用される。具体的には、レベル i において membership vector の接頭 i 桁が等しいキーが隣接するキーとなり、各レベルにおいて接頭 i 桁が等しいキーのエントリ群により構成されたリストが接頭 i 桁の種類だけ存在することになる。

この Skip Graph の 1 ノードが持つ経路情報数を考えると、Skip Graph を構成するノード数が十分に多くその数を n とした場合、レベルの高さは $\log(n)$ となる。この各レベルにおいて両隣のキーへの経路情報を持つため、1 ノードあたりの経路情報数は $2 \times \log(n)$ となる。

Multi-key Skip Graph は、この Skip Graph を拡張し、1 ノード上に複数のキーを保持可能としたうえで、検索時におけるメッセージの多重処理を防ぐマルチレンジフォワードリングにより効率の良い検索を実現している。

Multi-key Skip Graph では、キーごとに経路表を持つため、ノードが持つキー数を m とすると 1 ノードが持つ経路情報数は $2 \times \log(n) \times m$ となる。キーバリューストアでは 1 ノード上に複数のキーを持つことから $\log(n) < m$ となるため、空間計算量は $O(m)$ となる。

3.2 分散キーバリューストアの実現方法と問題点

Multi-key Skip Graph では、キーの値域に制限がないため、User Key Space と Storage Key Space は同一キー空間とすることができる。したがって、 f は同一のキーへのマッピングとする。また、Multi-key Skip Graph においては、ノードが持つキーに対応してルーティングが行われるため、 g にも制約がなく、毎回対応する値が変わるランダムなマッピングであってもかまわない。ただし、ランダムなマッピングではキーに対応する値を格納するノード数が多くなってしまうと考えられるため、ある程度順序が保存され、ノードあたりの担当キー数が分散するマッピングを行うことが望ましいといえる。

すなわち、Multi-key Skip Graph による分散キーバリューストアは、PUT, GET, GETRANGE の各 API において以下の動作をすることとなる。

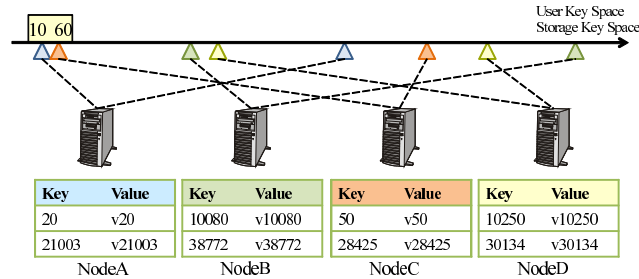


図 5 Multi-key Skip Graph によるキーバリューストア

Fig. 5 Multi-key Skip Graph based key-value store.

PUT(Ku, V)

マッピング g により得られるノードすべてが Multi-key Skip Graph 上に Ku を加え, Ku と V を保存する. g は, 必要数の担当ノードが得られるマッピングであればよく, 制限はない.

GET(Ku)

Ku のキーを持つノードへのルーティングを行い, 当該ノードから Ku に対応する V を取得する.

GETRANGE(Ku_s, Ku_e)

$[Ku_s, Ku_e]$ の範囲のキーを持つノードへのルーティングを行い, 当該ノードから $[Ku_s, Ku_e]$ にあるキーバリューストアを取得する.

図 5 は Multi-key Skip Graph によるキーバリューストアの構成例を示している. この状態で GETRANGE(10, 60) を実行すると, 範囲内のキーを持つ NodeA, NodeC が検索され, [10, 60] の範囲内にある (20, v20), (50, v50) が取得できる.

3.1 節で述べたとおり, Multi-key Skip Graph では, 各ノードが保持するキーごとに経路表が必要となる. したがって, 格納されたキー数に比例して各ノードが維持管理しなければならない経路数が多くなる. Multi-key Skip Graph では, ノードの出入りに際して関係するノードの経路表を更新する必要があるため, 経路表のサイズが大きくなると, ノードの出入り時のオーバーヘッドが大きくなってしまう.

4. Range-key Skip Graph による分散キーバリューストア

本研究では, 格納されたキー数に比例して各ノードが管理すべき経路表のサイズが大きくなるという問題に対処するため, Range-key Skip Graph により分散キーバリューストアを実現する方法を提案する. Range-key Skip Graph は, Multi-key Skip Graph を拡張し, 範囲対範囲の検索を可能とする構造化オーバーレイネットワークである. 以下では, まず Range-key Skip Graph について概説する.

4.1 Range-key Skip Graph

Range-key Skip Graph は Multi-key Skip Graph を拡張

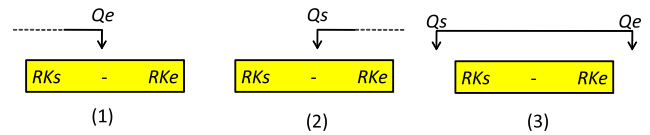


図 6 範囲検索に合致するレンジキー

Fig. 6 Matchable range query and range key.

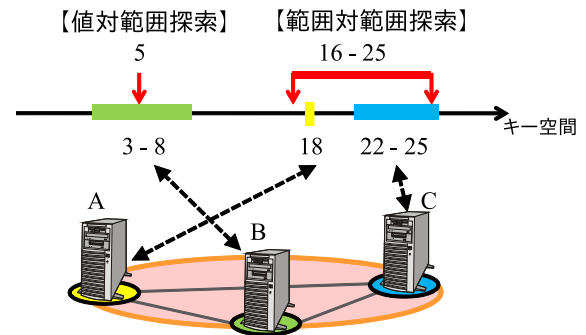


図 7 Range-key Skip Graph における検索例

Fig. 7 An example of searching keys on Range-key Skip Graph.

し, 上限値と下限値で指定される範囲を持つキー, レンジキーを登録可能としたオーバーレイネットワークである. Range-key Skip Graph ではレンジキーを登録可能となったことにより Skip Graph がサポートしていた「値対値」「範囲対値」の 2 種類の検索に加えて範囲も対象となるため, 新たに次の 2 種類の検索が可能となっている.

- ある値を指定して, その値を含むレンジキーを検索する値対範囲検索
- ある範囲を指定して, その範囲と重複部分を持つレンジキーを検索する範囲対範囲検索

検索クエリを Q としたとき, 以下の 3 条件のいずれかを満たすレンジキー RK_i が発見される (Q_s は検索クエリ Q が持つ範囲の最小値, Q_e は最大値を表す. RK_{is} はレンジキー RK_i の下限値, RK_{ie} は上限値を表す).

- $RK_{is} \leq Q_e \leq RK_{ie}$ (図 6 (1))
- $RK_{is} \leq Q_s \leq RK_{ie}$ (図 6 (2))
- $Q_s \leq RK_{is}$ かつ $RK_{ie} \leq Q_e$ (図 6 (3))

図 7 は Range-key Skip Graph で実現される検索例を示している. サーバ A がキー 18, サーバ B がレンジキー 3~8, サーバ C がレンジキー 22~23 を持ち, Range-key Skip Graph の 1 ノードとして動作している. キー 5 を検索した場合, 5 を含むレンジキーを持つサーバ B が発見される. 16 から 25 の範囲検索を行った場合, 検索範囲と重複部分を持つキー 18 とレンジキー 22~23 を持つサーバ A とサーバ C が発見される.

4.1.1 レンジキーの構造と経路数

レンジキーは, 下限値 RK_{is} ・上限値 RK_{ie} で表される範囲と, 他のレンジキーへの経路情報を持つ. 下限値は Skip Graph のキーとしてオーバーレイ上に登録され, 経路情報は下限値を範囲に含む他のレンジキー (包含キーと呼ぶ) を

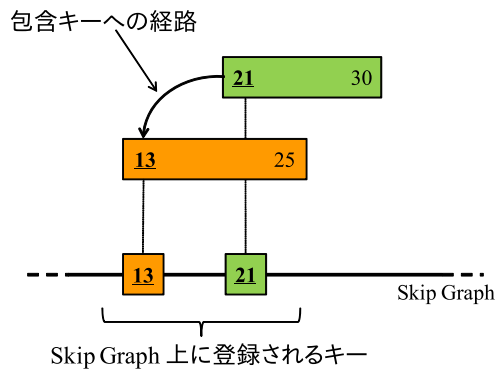


図 8 レンジキーの構造

Fig. 8 The routing structure of range key.

指している (図 8)。図中では、レンジキー 21~30 は、その下限値 21 を含む範囲を持つレンジキー 13~25 への経路情報を持っている。

ここで Range-key Skip Graph が保持すべき経路表を考えると、Multi-key Skip Graph の経路表に包含キーへの経路情報を加えたものになる。ノードが持つレンジキー数を r 、各レンジキーが持つ包含キーのリンク数の平均を k とすると Range-key Skip Graph 1 ノードが持つ経路情報数は $(2 \times \log(n) + k) \times r$ となる。ここで、キーバリューストアにおいてノードが担当するレンジ数、包含キーの数は小さく抑えることができ、 $k, r < \log(n)$ となると考えられ、空間計算量は $O(\log(n))$ となる。

4.1.2 レンジキーの検索・追加・除去

レンジキーの検索は、Range-key Skip Graph 上において次の手順で行う。

- (1) クエリで指定された範囲により Skip Graph を検索する。
- (2) クエリの範囲の最小値を超えない最大のキーを Skip Graph 上で検索する。
- (3) 手順 (2) でクエリを受け取ったレンジキーは自身が持つ経路情報が指すレンジキーにクエリを転送する。

手順 (1) により、図 6 (1), (3) が示すクエリの範囲内に下限値を持つレンジキーが発見される。手順 (2) は Range-key Skip Graph の実現にあたり、Skip Graph の検索手法を拡充したもので、これにより図 6 (2) に相当するレンジキーを発見できる。また、たとえば図 8 において、22~28 の範囲を検索した場合、手順 (1), (2) だけでは、レンジキー 21~30 のみが発見され、発見されるべきレンジキー 13~25 が発見できないが、レンジキー 21~30 が持つ経路情報に従ってクエリを転送することで発見できる (手順 (3))。

レンジキーの追加は以下の手順で行う。

- (1) 追加するレンジキー RK の下限値 RKs で検索し、包含キーとなるレンジキーを発見する。
- (2) 手順 (1) により得られた包含キー情報をレンジキー

RK に保持させる。

- (3) 下限値 RKs をキーとして、Skip Graph 上に登録する。

図 8 において、レンジキー 13~25 が追加済みの状態にレンジキー 21~30 を追加する場合、まず下限値の 21 で検索を行う (手順 (1))。これによりレンジキー 13~25 が発見されるので、これを包含キーとしてレンジキー 21~30 に経路情報を持たせる (手順 (2))。最後に下限値の 13 を Skip Graph に登録し、オーバレイネットワークへの追加を終える (手順 (3))。

また、レンジキーの除去は以下の手順で行う。

- (1) 除去しようとするレンジキー RK の範囲で Skip Graph 上を検索し、レンジキー RK を包含キーとして保持しているレンジキーを発見する。
- (2) 発見されたレンジキーの包含キー情報から、レンジキー RK を削除する。
- (3) レンジキー RK の下限値 RKs を Skip Graph から除去する。

図 8 の状態からレンジキー 13~25 を除去するには、まず 13~25 の範囲で検索する (手順 (1))。これにより、13~25 の範囲に下限値を持つレンジキー 21~30 が発見できる。レンジキー 21~30 はレンジキー 13~25 を包含キーとして経路情報を持っているためこれを削除する (手順 (2))。最後に登録されていた下限値 13 を Skip Graph 上から除去し、オーバレイネットワークからの除去を終える (手順 (3))。

4.2 Range-key Skip Graph による分散キーバリューストア

Range-key Skip Graph の検索動作は見方を変えるとレンジキーを持つノードにその範囲に重複する範囲条件を持つ検索クエリを送り届ける仕組みと見る事ができる。すなわち、ノードがレンジキーを持つことはそのノードにレンジキーに相当する Storage Key Space の範囲をマッピングすることに相当する。Range-key Skip Graph においては、ノードが持つレンジキーに応じてルーティングが行われるため、 g には制約はなく、Storage Key Space を任意に分割すればよい。

また、Range-key Skip Graph ではレンジキーの値域に制限がないため、User Key Space の値域を対応した Storage Key Space とすることで f は同一キーへのマッピングとすることができる。

すなわち、Range-key Skip Graph による分散キーバリューストアでは、PUT, GET, GETRANGE の各 API は以下の動作で実現される。

PUT(Ku, V)

保持するレンジキーの範囲に Ku が含まれるノードまでルーティングし、 V を Ku に対応づけて当該ノードのローカルストアに保存する。

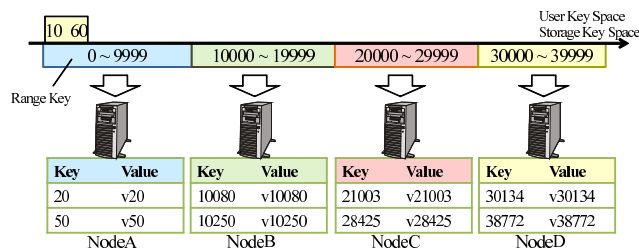


図 9 Range-key Skip Graph によるキーバリューストア
Fig. 9 Range-key Skip Graph based key-value store.

GET(K_u)

保持するレンジキーの範囲に K_u が含まれるノードまでルーティングし、当該ノードのローカルストアから K_u に対応する V を取得する。

GETRANGE(K_{u_s}, K_{u_e})

$[K_{u_s}, K_{u_e}]$ を検索クエリの条件とすることで $[K_{u_s}, K_{u_e}]$ に重複するレンジキーを持つノードまでルーティングし、当該ノードから $[K_{u_s}, K_{u_e}]$ に含まれるキーバリューストアを取得する。

図 9 は Range-key Skip Graph によるキーバリューストアの構成例を示している。図では、0 以上の整数空間を User Key Space, Storage Key Space として幅 10,000 ずつに区切り、その範囲に相当するレンジキーを各ノードに対応づけている。この状態で GETRANGE(10, 60) を実行すると、範囲に重複するレンジキー $[0, 99999]$ が検索され、レンジキーを持つ NodeA より $[10, 60]$ の範囲内にある $(20, v20)$, $(50, v50)$ が取得できる。

このように Range-key Skip Graph をキーバリューストアに用いる場合、各ノードには Storage Key Space を割り当てるために事前にレンジキーを持たせることができる。この場合、各レンジキーの順序は既知となるため、レンジキーを持つノードの順序も既知となる。この順序情報を利用し、ノードの順序に即した membership vector を各ノードに与えることでランダム値を membership vector に用いた場合に比較して少ないホップ数で検索クエリを転送できる経路表を構築できる。具体的には、最小のレンジキーを持つノードを 0 番とし、以降レンジキーの順に従い 1 番、2 番と連番を割り振る。この番号をビット反転させた値を各ノードの membership vector とする。3 bit の membership vector での例をあげると、順に 000, 100, 010, 110, 001, 101, 011, 111 という membership vector を持つことになる。経路表は membership vector の上位ビットの合致長に応じて構築されるため、各レベルでバランスする経路表となる。

5. キーバリューストアの実装と評価

5.1 キーバリューストアの実装

図 10 にサーバマシン上で動作させるキーバリューストアサーバの実装を示した。Key-value RPC Interface は、PUT, GET, GETRANGE の各 API を実装した RPC モジュール

で、これを介してクライアントと Key-value Connector を結び付ける。Local Storage はサーバが持つレンジキーの範囲内のキーバリューストアを保持する。R-key SG Overlay Interface は Range-key Skip Graph を介した別のサーバとの通信を司る。Key-value Connector は Local Storage, R-key SG Overlay Interface, Key-value RPC Interface を結び付け、各モジュールからの要求や応答の処理を行う。Range-key Skip Graph は PIAX 2.2 [20], [21] の実装を利用した。

クライアントが発した PUT や GET, GETRANGE 要求は RPC, すなわち Key-value RPC Interface を介して Key-value Connector が受け付ける。この要求が自サーバの担当範囲内への要求であった場合は、Local Storage に対して入出力処理を行い、その結果をクライアントに返す。自サーバの担当範囲外であった場合は、R-key SG Overlay Interface を介して Range-key Skip Graph で結ばれた担当サーバを検索し、要求を通知する。担当サーバより応答が得られるとその結果をクライアントに返す。他サーバより R-key SG Overlay Interface を介して要求が届いた場合、その要求は Key-value Connector に受け渡され、要求に応じて Local Storage に入出力処理を行う。Local Storage から得られた結果を Range-key Skip Graph を介して要求元に送り返す。

5.2 Range-key Skip Graph による実装と Multi-key Skip Graph による実装の比較

5.1 節で記した Range-key Skip Graph による分散キーバリューストア実装と、オーバーレイ部を Multi-key Skip Graph に差し替えた実装との間で比較を行った。ここでは、各ノード間の通信に PIAX の実験支援機能である仮想的な通信路 (EmuTransport) を使用し、単一 PC 上で計測評価を行っている。評価に用いた機材環境は表 1 のとおりである。

キーバリューストアは 100 ノードで構成し、登録するキーには 64 bit 長の整数値、値には 512 バイトの固定長バイト列を用いる。各ノードが持つローカルストレージにはオンメモリストレージを使用する。このキーバリューストアに対して、キーを 0 から順に 1 ずつ増加させ 100 万個のキーバリューストアを PUT し、初期登録を行う。その際、10 万ペア登録するごとに、登録されたキーの範囲に対して GET, GETRANGE, PUT をそれぞれ 1,000 回実行し、評価値を取得する。ここでの PUT はキーに対応する値の更新処理に相当する。これらの要求発行にはキーバリューストアを構成する 100 ノードの中から 1 ノードをランダムに選択し、そのノードより要求を発行する。API に与えるキーはキーが登録されている範囲からランダムに選択し、さらに GETRANGE では 300 を上限としたランダムな取得範囲幅を与える。以上の手順で 10 回計測を実施し、各々の評価値の平均値を求めた。

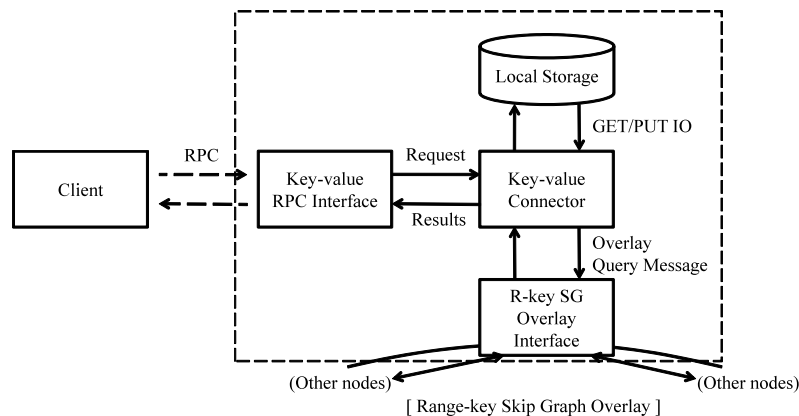


図 10 キーバリューストアサーバの構成

Fig. 10 The composition of key-value store server.

表 1 評価環境

Table 1 Evaluation environment.

CPU	Intel E8500 (2core 3.16 GHz)
Memory	4 GB (実効 3.5 GB)
OS	CentOS 5.7 (32 bit)
JVM	Oracle Java 1.6.0_30 (Client VM)

Range-key Skip Graph による実装（以下 Range-key 手法とする）では、1 万幅のレンジキーを各ノードに順に持たせ、レンジキーの範囲に応じたキー空間を各ノードに担当させている。Multi-key Skip Graph による実装では、最もシンプルな方法としてクライアントから PUT 要求を受け付けたノードがそのキーを保持する仕様とした。PUT 時のノードの選択法として 1 万ずつ PUT 先ノードを順に切り替える Multi-key(Ordered) 手法と、PUT ごとにランダムに選択する Multi-key(Random) 手法の 2 手法を用いた。以下 Multi-key(Ordered) 手法と Multi-key(Random) 手法をまとめて扱う場合は Multi-key 手法と表記する。Range-key 手法における PUT 時のノード選択は Multi-key(Ordered) 手法と同様とした。Range-key 手法、Multi-key 手法ともに各ノードの membership vector は 4.2 節で示した手法で割り当てている。

GET, GETRANGE, PUT の実行により生じる負荷の評価として、平均最大ホップ長とメッセージ数、所要時間を計測するとともに、登録キー数によるメモリの使用量を計測した。また、スケーラビリティにつながる評価として 1 ノードが管理する平均経路数を計測した。

まず、キーバリューストアの初回 PUT 時の 1 要求あたりの平均メッセージ数を図 11 に示す。初回 PUT では Multi-key(Ordered) 手法、Multi-key(Random) 手法ともに、Range-key 手法と比較して数万～数十万倍のメッセージが生じている。これは Multi-key Skip Graph を用いた場合、キーバリューストアとオーバレイ上のキーが 1 対 1 対応であるため、キーバリューストアの登録ごとに経路表の追加と更新が生じるためである。これに対して、Range-key

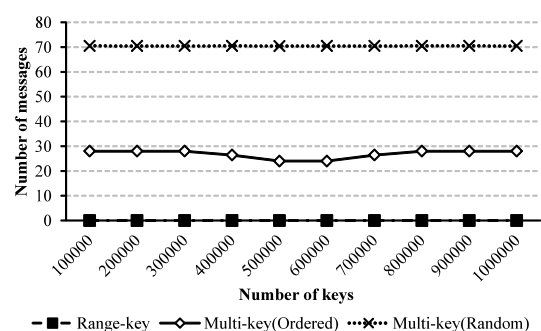


図 11 初回 PUT 1 要求あたりの平均メッセージ数

Fig. 11 The average number of messages per request at first time of PUT.

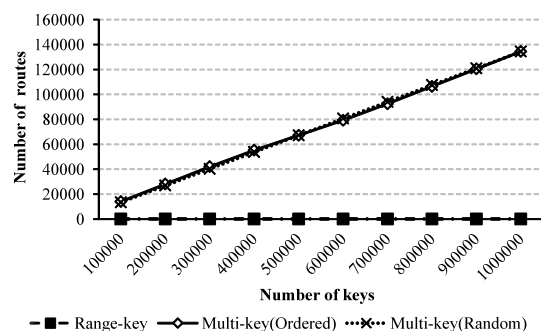


図 12 ノードあたりの平均経路数

Fig. 12 The average value of routes per node.

手法では、レンジキーの境界部に相当するキーバリューストアを登録する場合以外は他のノードとの通信が生じないため、キーバリューストアあたりのメッセージ数は非常に少なくなる。また、Multi-key(Ordered) 手法において 40 万キーから 70 万キーの範囲ではそれ以外の範囲と比較してメッセージ数が減少しているが、これはこの範囲を担当するノードにおいて Skip Graph の特性上、経路表のサイズが 1 レベル少なくなるため、経路表の更新に必要なメッセージ数が少なくなることによる。

続いて図 12 に初回 PUT にともなうキーバリューストア数の変化とノードあたりの平均経路数の関係を示す。先に

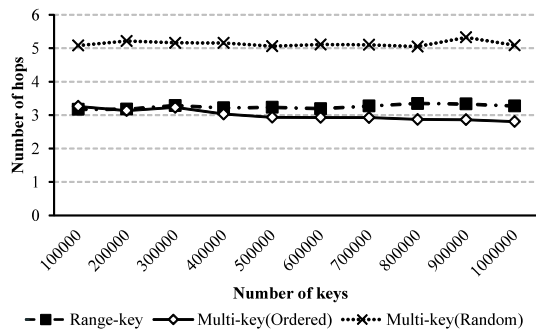


図 13 PUT 時の平均最大ホップ長

Fig. 13 The average value of maximum hops at PUT.

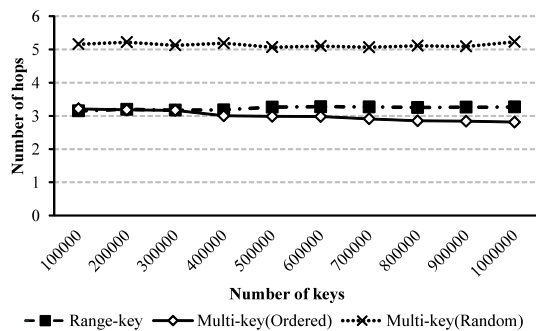


図 14 GET 時の平均最大ホップ長

Fig. 14 The average value of maximum hops at GET.

述べたように Multi-key 手法では、キーバリュースペアごとに経路表が必要になることから、キー数に比例して経路数が増えている。Range-key 手法ではノードが保持しているキーバリュースペア数の変化は経路表に影響をあたえないことから、Multi-key 手法と比較して非常に少ない一定値 (13.5) となっている。

次に初回 PUT 後の PUT, GET 実行時の各 1 要求に要した平均最大ホップ長を図 13 と図 14 に、平均メッセージ数を図 15 と図 16 に示す。最大ホップ長とは、1 要求で生じた 1 個以上のメッセージによる転送経路のうち、最大のホップ長のものを示す。

PUT, GET は、オーバーレイ上では「指定されたキーを持つノードの検索」という同じ処理であるため、API の違いにかかわらず平均最大ホップ長は Range-key 手法と Multi-key(Ordered) 手法では約 3、Multi-key(Random) 手法では約 5 となっている。同様に、平均メッセージ数は Range-key 手法では約 18、Multi-key(Ordered) 手法では約 12、Multi-key(Random) 手法では約 20 と、ほぼ同じメッセージ数となっている。

PUT, GET とともに、Multi-key(Random) 手法でのホップ数・メッセージ数が明らかに大きくなっている。これは、4.2 節による順序を考慮した membership vector を設定したにもかかわらず PUT を要求するノードをランダムに選択することから、キーがその順序に関係なくキーバリューストア構成ノード全体に分散することとなり、キー検索に要

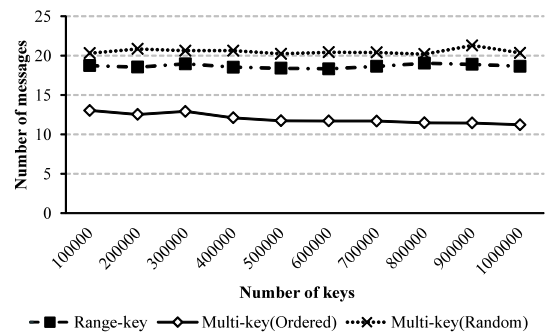


図 15 PUT 1 要求あたりの平均メッセージ数

Fig. 15 The average number of messages per request at PUT.

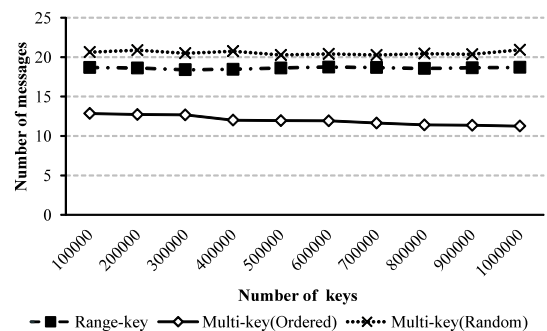


図 16 GET 1 要求あたりの平均メッセージ数

Fig. 16 The average number of messages per request at GET.

するホップが増え、それに従いメッセージ数も増えたことによる。

これに対して Range-key 手法と Multi-key(Ordered) 手法では、キーが特定のノードに集約されるとともに、その順序に即した membership vector がノードに設定されていることから Multi-key(Random) 手法と比較して少ないホップ数・メッセージ数となっている。Multi-key(Ordered) 手法では、キーが集約されていることから Multi-key Skip Graph でのノード内メッセージ転送機構が有効に働き Range-key 手法とほぼ同等のホップ数となっている。また、キー数が少ない時点では Range-key 手法より多くのホップ数を要しているが、これはキーが登録されていないノードからのクエリ発行が多いためであり、キーの登録が進むに従いキーが登録されているノードからのクエリ発行が増え、その分ホップ数が減少することによる。最終的に Range-key 手法よりも Multi-key(Ordered) 手法のホップ数が少なくなるが、これは Range-key Skip Graph と Multi-key Skip Graph のルーティング方法の差異によるものと考えられる。メッセージ数を見ると、Multi-key(Ordered) 手法が最も少ない値となっているが、これは Range-key 手法では、Multi-key(Ordered) 手法と同じ検索処理に加えて「指定のキーを超えない最大のキー」を検索する処理が必要になる分、メッセージ数が増えることによる。

図 17 は GETRANGE での平均最大ホップ長、図 18 は平均メッセージ数を表している。平均最大ホップ長は

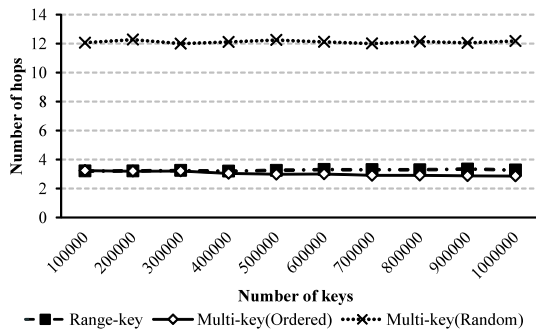


図 17 GETRANGE 時の平均最大ホップ長

Fig. 17 The average value of maximum hops at GETRANGE.

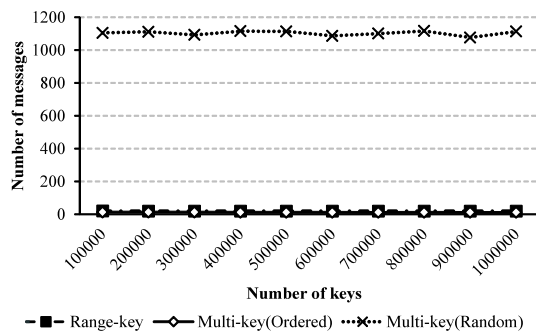


図 18 GETRANGE 1 要求あたりの平均メッセージ数

Fig. 18 The average number of messages per request at GETRANGE.

Range-key 手法と Multi-key(Ordered) 手法では約 3, Multi-key(Random) 手法では約 12 となっている。平均メッセージ数では Range-key 手法では約 18, Multi-key(Ordered) 手法では約 12, Multi-key(Random) 手法では約 1,000 となっている。

Range-key 手法と Multi-key(Ordered) 手法では、キーバリュペアがその順序に応じて集約されているため、範囲検索により応答するノードは 1~2 ノードであるのに対し、Multi-key(Random) 手法では、キーがその順序に関係なくキーバリュストア構成ノード全体に分散しているため、指定された範囲に含まれるキーは多数のノードに分散することとなる。このため、ホップ数では複数生じるメッセージの転送経路のうち、最もホップ数を要したものを最大ホップ長としていることから他の手法と比較して約 4 倍のホップ数を要している。メッセージ数においても、複数のノードに転送され、その応答を得ていることから大きな差が生じている。

次に、キーバリュペアの初回 PUT および登録後の GET, PUT, GETRANGE の各 1 要求に要した平均所要時間を 図 19, 図 20, 図 21, 図 22 に示す。

初回 PUT では、キーバリュペアの登録において経路表の操作が必要ない Range-key 手法が最も短時間で処理を終えている。対して Multi-key 手法では、経路表の更新による通信で生じるメッセージ数 (図 11) に準ずる処理時間を要していることが分かる。GET, PUT, GETRANGE において

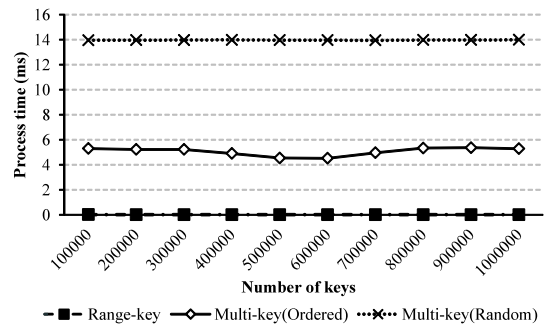


図 19 初回 PUT 1 要求あたりの平均所要時間

Fig. 19 The average number of process time per request at first time of PUT.

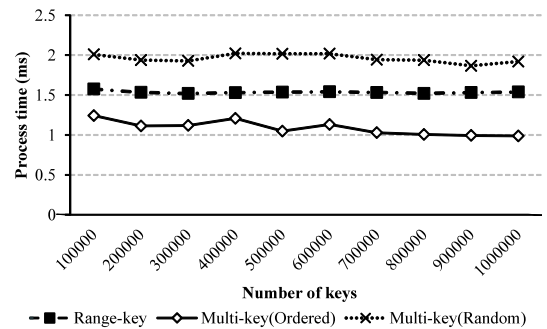


図 20 GET 1 要求あたりの平均所要時間

Fig. 20 The average number of process time per request at GET.

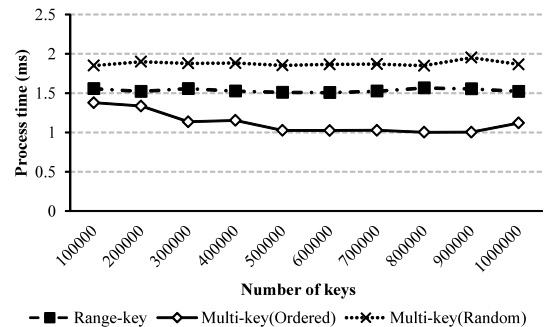


図 21 PUT 1 要求あたりの平均所要時間

Fig. 21 The average number of process time per request at PUT.

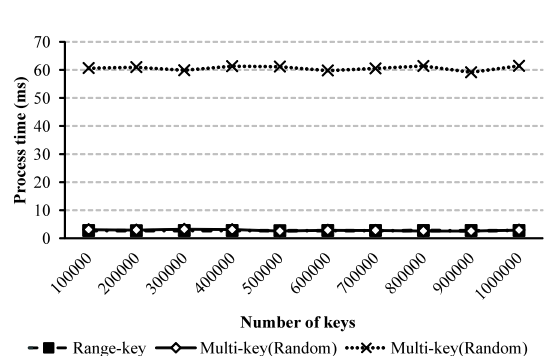


図 22 GETRANGE 1 要求あたりの平均所要時間

Fig. 22 The average number of process time per request at GETRANGE.

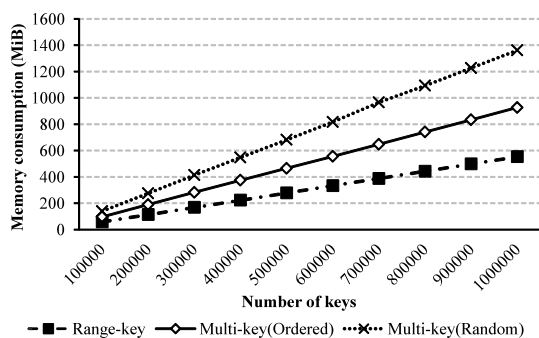


図 23 メモリ使用量

Fig. 23 Memory consumption.

も、同様の傾向であることから、通信オーバーヘッドが所要時間の大部分を占めていると考えられる。

図 23 に登録されたキーバリュースタア数によるメモリ使用量を示す。メモリ使用量は Java ヒープの使用量より得た。

各手法ともキーバリュースタアの登録数に比例してメモリ使用量が増えているが、各グラフの傾きから分かるように、1 キーバリュースタアのメモリ使用量は Range-key 手法が最も少なく、Multi-key(Ordered) 手法、Multi-key(Random) 手法とメモリ使用量が増えていることが分かる。Range-key 手法では、キーバリュースタアの登録は各ノードが持つローカルストレージへの格納のみであることに対し、Multi-key 手法ではローカルストレージへの格納に加えて経路表への登録も必要となる。さらに Multi-key(Random) 手法では、Multi-key(Ordered) 手法より 1 キーあたりの経路数が多くなることから、より多くのメモリが必要となっている。

以上の評価より Range-key 手法では、キーバリュースタアを集約して登録する Multi-key(Ordered) 手法との比較では PUT, GET, GETRANGE とともに約 1.5 倍のメッセージ数・所要時間を要することが分かる。しかしながら、Multi-key(Ordered) 手法ではキーバリュースタアを登録するノードを適切に選択することでキーを集約を実現しており、実環境において同様の動作を実現するには、キーバリュースタアの登録要求を適切なノードに割り振るゲートウェイサーバが必要になり、1 章で述べた 1-hop によるキーバリュースタアと同じ問題を持つことになる。対して、任意のキーバリュースタア構成ノードからキーバリュースタアを登録する場合、Multi-key(Random) 手法相当となり、Range-key 手法が優位となる。

加えて、Range-key 手法では、ノードが維持すべき経路情報数が Multi-key 手法より大幅に少なくなるため、ノードの出入りによる経路情報の修正コストや動作中の経路維持コストを大幅に抑制できると考えられる。

5.3 クラウド環境を想定した性能評価

次に、情報通信研究機構北陸リサーチセンターが運用す

表 2 StarBED Group F サーバスペック
Table 2 StarBED Group F server specification.

CPU	Pentium4 3.2 GHz; Hyper Threading
Memory	2 GB
Network	GigabitEthernet LAN
OS	Fedora 10 (32 bit)
JVM	Sun Java 1.6.0_13 (Server VM)

る大規模ネットワーク実験用 PC クラスタである StarBED のサーバ群を用いて、クラウド環境を想定したリクエスト処理性能の評価を行った。StarBED では、様々なスペックのサーバが利用可能であるが、本研究では Group F と呼ばれるサーバ群を使用した (表 2)。

評価では Group F サーバを 108 台使用し、そのうち 100 台をキーバリュースタアに割り当て、残り 8 台をリクエスト発行クライアントとした。キーバリュースタアを構成するサーバには、1 台あたり 10 万個のキーバリュースタアを保持させ、全体で $10 \text{ 万} \times 100 \text{ 台} = 1,000 \text{ 万}$ のキー空間を持つ分散キーバリュースタアとする。リクエスト発行クライアントは、キーバリュースタアに GET または GETRANGE リクエストを発行し、その発行時刻と応答時刻を記録する。GET では、有効なキー空間内からランダムに 1 つキーを選出し、そのキーをリクエストとして使用する。GETRANGE では、GET と同様に有効なキー空間内を対象とし、その取得範囲幅が 300 未満のランダムな範囲をリクエストに用いる。リクエストの発行は、各リクエスト発行クライアントから複数のスレッドを用いて行う。各スレッドでは、リクエストの発行、応答待ち、次のリクエストの発行、応答待ちというサイクルを繰り返すことで、キーバリュースタアに対して連続的、同時並列にリクエストを送り続ける。リクエストを依頼するキーバリュースタアサーバは、リクエスト発行時にランダムに選択する。

以上の手法で 5 分間リクエストを発行させ続けて得られた計測ログを 10 秒ごとに区切り、その区間内にリクエスト発行時刻と応答時刻が収まっている記録をその区間内で実行されたリクエストとする。各区間に含まれるリクエスト数を求め、さらに区間の間で平均をとり、1 秒間の平均リクエスト処理数を得る。ただし、JIT コンパイルを行う Java の性質を考慮し、JVM プロセスの状態が安定していない計測開始から 30 秒間の記録は使用しない。

以上の計測法を用いて、キーバリュースタアサーバ数を 10 台追加するごとに計測した結果を図 24 に示す。GET, GETRANGE とともに、サーバ数が増えるに従ってリクエスト処理数が増えていることが分かる。計測時、サーバの CPU 使用率はほぼ 100% となっており、本評価の環境における各サーバ数でのリクエスト処理の上限性能を示している。GET に対して GETRANGE のリクエスト処理数が少なくなっているが、GETRANGE では、1 リクエストに対して複

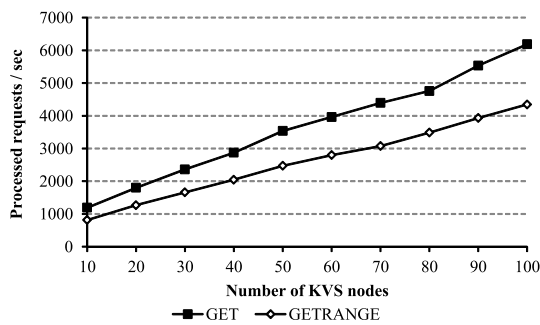


図 24 秒間平均リクエスト処理数

Fig. 24 The average value of request per second.

数の要素が得られるため、サーバ内での要素集約と結果の転送処理に GET より時間を要するためである。この評価より、Range-key Skip Graph を用いたキーバリューストアでは、それを構成するサーバ数に応じてリクエスト処理数がスケールすることが確認できた。

6. まとめ

本研究では、構造化オーバーレイネットワークである Range-key Skip Graph による分散キーバリューストアを提案した。従来の分散キーバリューストアでは一般的ではなかった範囲指定によるデータ取得において、Range-key Skip Graph を用いることで各サーバが持つ経路情報数を抑制し、数十万～数百万台へのスケールアウトの目処をたてた。また、StarBED を用いた実機評価において、100 台までサーバ数を増やすことで、その台数に応じてリクエスト処理性能がリニアに向上することを確認した。

今後の課題として、現時点ではレプリカを考慮しておらず、サーバ障害時には担当範囲への入出力が行えなくなるため、レプリカの配置について検討する必要がある。また、負荷分散についても考慮していないため、アクセス負荷とストレージ負荷の両面において負荷の集中が生じた際の分散手法の検討が必要である。

謝辞 大規模環境下での動作検証に独立行政法人情報通信研究機構北陸リサーチセンターが運用する StarBED、および同機構運用の JGN2plus を利用した (JGN2P-A20089「PIAX に基づく広域オーバーレイネットワークを用いたユビキタスサービスプラットフォームの検証」)。利用に際し各々の運用チームの皆様には多大なるご協力をいただいた。ここに記して謝意を表す。

参考文献

- [1] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Voshall, P. and Vogels, W.: Dynamo: Amazon's highly available key-value store, *SIGOPS Operating Systems Review*, Vol.41, No.6, pp.205-220 (2007).
- [2] The Apache Cassandra Project, available from <http://cassandra.apache.org/> (accessed 2012-04-04).
- [3] Rakuten ROMA, available from <http://code.google.com/p/roma-prj/> (accessed 2012-04-04).

- [4] Flare - GREE Labs, available from <http://labs.gree.jp/Top/OpenSource/Flare.html>, (accessed 2012-04-04).
- [5] The Kumofs Project, available from <http://kumofs.sourceforge.net/> (accessed 2012-04-04).
- [6] 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, *Proc. 29th Annual ACM Symposium on Theory of Computing (STOC '97)*, pp.654-663 (1997).
- [7] Stoica, I., Morris, R., Karger, D., Kaashoek, F. and Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications, *ACM SIGCOMM Computer Communication Review*, Vol.31, No.4, pp.149-160, ACM (2001).
- [8] Maymounkov, P. and Mazieres, D.: Kademlia: A Peer-to-Peer Information System Based on the XOR Metric, *Proc. 1st International Workshop on Peer-to-Peer System (IPTPS '02)*, p.263 (2002).
- [9] 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 International Conference on Peer-to-Peer Computing (IEEE P2P '11)* (2011).
- [10] Windows Azure Platform, available from <http://www.microsoft.com/windowsazure/> (accessed 2012-04-04).
- [11] Schütt, T., Schintke, F. and Reinefeld, A.: Scalaris: Reliable Transactional P2P Key/Value Store, *Proc. 7th ACM SIGPLAN Workshop on ERLANG (ERLANG '08)*, pp.41-47 (2008).
- [12] Schütt, T., Schintke, F. and Reinefeld, A.: Structured Overlay without Consistent Hashing: Empirical Results, *Proc. 6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID '06)* (2006).
- [13] Schütt, T., Schintke, F. and Reinefeld, A.: A Structured Overlay for Multi-dimensional Range Queries, *Proc. 13th International Conference on Parallel Computing (Euro-Par '07)*, LNCS 4641, pp.503-513, Springer (2007).
- [14] Shi, G., Chen, J., Gong, H., Fan, L., Xue, H., Lu, Q. and Liang, L.: SandStone: A DHT Based Carrier Grade Distributed Storage System, *Proc. 2009 International Conference on Parallel Processing (ICPP '09)*, pp.420-428 (2009).
- [15] Zhao, F.D.B., Druschel, P., Kubiawicz, J. and Stoica, I.: Towards a common API for Structured Peer-to-Peer Overlays, *Proc. 2nd International Workshop on Peer-to-peer Systems (IPTPS '03)* (2003).
- [16] Aspnes, J. and Shah, G.: Skip Graphs, *ACM Trans. Algorithms*, Vol.3, No.4, p.37 (2007).
- [17] 小西佑治, 吉田 幹, 竹内 亨, 寺西裕一, 春本 要, 下條真司: 単一ノードに複数キーを保持可能とする Skip Graph 拡張, *情報処理学会論文誌*, Vol.49, No.9, pp.3223-3233 (2008).
- [18] Ishi, Y., Teranishi, Y., Yoshida, M., Takeuchi, S., Shimojo, S. and Nishio, S.: Range-Key Extension of the Skip Graph, *Proc. IEEE 2010 Global Communications Conference (IEEE GLOBECOM 2010)* (2010).
- [19] Pugh, W.: Skip Lists: A Probabilistic Alternative to Balanced Trees, *Workshop on Algorithms and Data Structures*, pp.437-449 (1989).
- [20] 吉田 幹, 奥田 剛, 寺西裕一, 春本 要, 下條真司: マルチオーバーレイと分散エージェントの機構を統合した P2P プラットフォーム PIAX, *情報処理学会論文誌*, Vol.49,

No.1, pp.402–413 (2008).

- [21] PIAX – P2P Interactive Agent eXtensions, available from <http://www.piax.org/> (accessed 2012-04-04).

推薦文

現在の大規模ネットワーク環境において、Twitterなどの情報の範囲検索を必要とするアプリケーションが利用されている。本論文では、オーバーレイネットワーク Range-key Skip Graph を用いて、このような状況に対応するための手法を提案している。StarBED 上での実装による大規模環境での動作確認により提案手法の有効性が確かめられており、有用性の高さが認められ、推薦に値する。

(マルチメディア通信と分散処理研究会主査 勝本道哲)



石 芳正 (正会員)

2004 年京都工芸大学工芸学部電子情報工学科卒業。2006 年大阪大学大学院情報科学研究科博士前期課程修了。同年同大学サイバーメディアセンター特任研究員。2008 年同大学院情報科学研究科特任研究員。2012 年同大学

サイバーメディアセンター特任研究員となり、現在に至る。修士 (情報科学)。ユビキタス応用システム等の研究開発に従事。



寺西 裕一 (正会員)

1993 年大阪大学基礎工学部情報工学科卒業。1995 年同大学院基礎工学研究科博士前期課程修了。同年日本電信電話株式会社入社。2005 年大阪大学サイバーメディアセンター講師。2007 年同大学院情報科学研究科准教授。2008

年より情報通信研究機構専攻研究員、招へい専門員を兼任、現在に至る。博士 (工学) (2004 年 3 月、大阪大学)。マルチメディア情報システム、ユビキタス応用システム等の研究開発に従事。本会論文賞を受賞。IEEE 会員。



吉田 幹 (正会員)

1981 年京都大学工学部情報工学科卒業。1986 年同大学院工学研究科博士後期課程修了。同年日本アイ・ビー・エム株式会社入社。東京基礎研究所勤務。1994 年新日鉄ソリューションズ株式会社入社。2002 年株式会社ビービーアール設立、現在に至る。人工知能学会会員。



竹内 亨 (正会員)

2001 年大阪大学基礎工学部情報科学科卒業。2003 年同大学院基礎工学研究科博士前期課程修了。2006 年同大学院情報科学研究科博士後期課程修了。博士 (情報科学)。同年同研究科マルチメディア工学専攻助手。2007 年同助教。2009 年情報通信研究機構専攻研究員を経て、2011 年日本電信電話株式会社未来ねっと研究所研究主任となり、現在に至る。ソーシャルネットワークおよびオーバーレイネットワークを活用した情報システムの研究開発・展開活動に従事。本会論文賞を受賞。IEEE 会員。



下條 真司 (正会員)

1981 年大阪大学基礎工学部情報工学科卒業。1986 年同大学院基礎工学研究科博士後期課程修了。博士 (工学)。同年同大学基礎工学部情報工学科助手。1989 年同大学大型計算機センター講師。1991 年同助教授。1998 年同教授。2000 年同大学サイバーメディアセンター教授。2008 年 4 月独立行政法人情報通信研究機構上席研究員。LAN のアクセス方式の性能評価、分散システムの性能評価、分散型オペレーションシステムの研究に従事。電子情報通信学会、ACM、IEEE、ソフトウェア科学会各会員。