2007 年度 修士論文

DHT における
コンテンツの更新を考慮した
検索手法の提案

提出日： 2008 年 2 月 4 日

指導： 村岡洋一教授

早稲田大学大学院 理工学研究科 情報・ネットワーク専攻
学籍番号： 3606U087-4
一杉　孝之

目次

図目次

表目次

第 1 章

序論

本研究では, コンテンツの更新を考慮した, DHT の新しい検索手法を提案する.

1.1 研究の背景

今や, 情報の爆発的な増加やコンテンツの多様化などによって, 従来の集中管理型のクライアント・サーバモデルだけでなく自律分散型の P2P ネットワークも利用されるようになっている. P2P ネットワークのように下位ネットワークレイヤの上に重なって別のトポロジを形成するネットワークのことをオーバーレイネットワークという. とくに Gnutella[1] や Winny[2] などのようにオーバーレイネットワークのトポロジに数学的な制約を持たないネットワークは非構造化オーバーレイと呼ばれる. これらは探索を flooding で行うため, ノード数が膨大なネットワークにおいてトラフィックが溢れる可能性があり, さらに探索が成功する保証がない. そこで近年では, 構造化オーバーレイと呼ばれる, トポロジに数学的な制約を持つオーバーレイネットワークが注目されている. 構造化オーバーレイには分散ハッシュテーブル（DHT:Distributed Hash Table）や Skip Graph[3] があげられる. これらはノード数に対してスケーラブルかつ確実な探索を実現している. Skip Graph は, skip list[4] を P2P ネットワークに適応させた構造化オーバーレイであり, 標準で範囲検索が可能である. 一方, DHT は分散 Map 構造を形成する. DHT では, Chord[5], Kademlia[6], Pastry[7], Tapestry[8], CAN[9] などといった様々なルーティングアルゴリズムが考案されており, システム構築時に目的に合わせてトポロジとノードの探索方法の選定をできるという利点がある.

現在は, 分散環境のテストベッドとして利用できる PlanetLab[10] や, オーバーレイ

ネットワークの構築ツールキットである Overlay Weaver[11][12] が公開されており, 分散ネットワークシステムの構築の敷居が低くなってきている. そのため, 構造化オーバーレイの利用が今後増えていくことが期待される.

1.2 研究の目的

前置きとして, 本論文におけるコンテンツの更新とは古いコンテンツを残しておいたまま新しいコンテンツを追加することを指す, と述べておく.

従来の DHT の研究ではコンテンツの取得・公開・削除の動作にばかり着目し, 更新の動作への配慮が欠けている. 例えば, 更新前のコンテンツと更新後のコンテンツを同じ key に割り当てた場合, 取得者はコンテンツを取得するまでそれらを判別できない. 一方, 異なる key に割り当てた場合には, 元の key では更新後のコンテンツを取得できないため, 取得者はなんらかの手段で新しい key の知識を手に入れなくてはならない. そのため, 従来の DHT はコンテンツを頻繁に更新するサービスには適していない. したがって, 今後 DHT を実用化していくうえでこの問題が妨げとなることが考えられる. 本研究では, これを解決する手法を提案し, 実験による評価および考察を行うことが目的である. この提案手法は以下の 3 つの要件を満たす.

- コンテンツの更新時の問題を解決する.
- 属性検索が可能である.
- ルーティング・アルゴリズムに非依存である.

1.3 研究の意義

情報の爆発的な増加やコンテンツの多様化などに伴い, P2P ネットワークの需要が高まっている. 本研究の目標はすなわち, ノード数に対してスケーラブルかつコンテンツの頻繁な更新にも耐えうる P2P ネットワークである. そして, この実現こそが本研究の意義である.

1.4 本論文の構成

本論文は以下の 6 章からなる.

第 1 章　序章

本論文の概要, 目的, 構成について述べる.

第 2 章　関連研究

本研究と関連のある従来研究について述べる.

第 3 章　提案手法

従来研究の問題点について述べ, 解決するための手法を提案する.

第 4 章　実験環境の構築

実験環境の構築方法について述べる.

第 5 章　評価実験と考察

評価実験と, そこから得られる考察について述べる.

第 6 章　結論

本論文のまとめと今後の課題について述べる.

第 2 章

関連研究

本章では, 本研究と関連のある従来研究について述べる.

2.1 DHT の基本動作

2.1.1 はじめに

本節では DHT の基本的な動作を説明する. DHT の細かい動作はルーティングアルゴリズムによって異なるので, ここでは土台となる共通部分について述べる.

2.1.2 ID

DHT ではノード数に対して十分に広い ID 空間を使用する. ID 空間の決定方法は任意であるが, 160 bit の bit 列を使うことが多い.

DHT 上の各ノードはノードの識別子となる nodeID を持つ. nodeID は ID 空間に含まれる. とくに nodeID は ID 空間に散らばることが好ましく, 例えば一様分布でランダムに決定したり, IP アドレスとポート番号の組を元に決定したりする.

ID 空間には距離が存在し, ノード間の距離を計ることができる. 距離の定義はルーティングアルゴリズムによって異なり, 例えば ID の差や排他的論理和が使われている.

2.1.3 HashTable

DHT 上の各ノードは HashTable を持つ．この HashTable が key に使う hash 空間は ID 空間と共通である．すなわち, HashTable の要素となる ⟨key, value⟩ の key は ID 空間に含まれる値となる．ちなみに後で詳しく説明するが, value は DHT 上で共有するコンテンツを示す．本論文ではこの HashTable の要素のことを単に pair と表記する．

ノードは nodeID を元にして, そのノードが担当する ID 空間を割り当てられる．そして, 自分の担当する ID 空間に key が含まれるときに, pair を自分の HashTable に追加する（ただし, 障害耐性のために近傍ノードが担当する ID 空間も同様に扱う場合がある）．すべてのノードがすべての pair を HashTable に持つ必要はなく, ノード群が協調動作することで DHT 全体がすべての pair を持つ Map 構造を形成する（DHT は基本的には同期化されないため, ここではあえて Table でなく Map と表記している）．

2.1.4 join

DHT でコンテンツの公開および取得をするためには, ノードが DHT に参加（join）している必要がある．参加方法はルーティングアルゴリズムにより大きく異なるため, ここでは省略する．

2.1.5 put

DHT でノードがコンテンツを公開する手順を説明する．このノードは既に nodeID を持ち, DHT に参加中であるとする．

DHT で共有するコンテンツには, それを指し示す名前が必要である．便宜上, 本論文ではこの名前のことを name と表記する．name の形式は任意であり, 例えばコンテンツのファイル名やタイトルなどが使われる．

ただし name に合わせて, DHT で共通して使用するハッシュ関数も決定しなくてはならない．このハッシュ関数は引数に name をとり, それを ID 空間に写像する．言い換えれば, hash(name) はノードの HashTable の key になりうる．hash(name) は ID 空間にできるだけ一様に散らばることが望ましく, ハッシュ関数には SHA-1 がよく用いられる．

ノードは hash(name) を求めて, それに nodeID が最も近いノードを探索する（ルー

ティングアルゴリズムに探索方向が指定されている場合には, その方向において最も近いノードを探索する.）この探索方法については 2.1.8 節で後述する．探索で発見したノードの HashTable に ⟨hash(name), contents⟩ を追加する．ここで, contents は公開するコンテンツを意味している．つまり, ノードの HashTable の value はコンテンツを示すわけである．ここまででコンテンツの公開は完了である．

hash(name) が ID 空間に一様に散らばるならばコンテンツの追加先ノードも散らばることになる．よって, このとき DHT はネットワーク上のノードに負荷が分散されるという特性を持つ.

ここで説明した公開の動作は, DHT が Map 構造をしていることから put と呼ばれている．

2.1.6 get

DHT でノードがコンテンツを取得する手順を説明する．このノードは既に nodeID を持ち, DHT に参加中であるとする.

ノードは取得するコンテンツの name をあらかじめ知っている必要がある．put のときと同様に, hash(name) を求めて, それに nodeID が最も近いノードを探索する．すでにコンテンツが DHT で公開されているならば, 探索で発見したノードの HashTable には ⟨hash(name), contents⟩ が存在しているはずである．そこで, hash(name) を key として, このノードの HashTable からコンテンツを得る．これでコンテンツの取得は完了である．

一般的に引数が異なるときハッシュ値も異なるので, DHT は基本的に name の完全一致検索しか対応していない.

ここで説明した取得の動作は, put と同様の理由で get と呼ばれている．

2.1.7 remove

DHT でノードがコンテンツを削除する手順を説明する．このノードは既に nodeID を持ち, DHT に参加中であるとする.

get と同様に, HashTable に ⟨hash(name), contents⟩ を持つノードを探索する．ここで, hash(name) を key とすれば, このノードの HashTable からコンテンツを削除することができる．しかし, DHT では key が同じで value が異なる pair を許すので, そのようなコンテンツを間違えて削除してしまう可能性がある．その事態を防ぐために, HashTable か

ら value を削除をするときには key だけでなく hash(value) も指定する, という対策が考えられている. これにより, 同じ key に対応付けられた異なるコンテンツを間違えて削除することが十分な確率で防げるとともに, value を直接指定するよりも通信量が少なくて済む. これでコンテンツの削除は完了である.

ここで説明した削除の動作は, remove または delete と呼ばれている. 本論文では remove と統一して呼ぶ.

2.1.8 lookup

DHT 上のノードは, ID 空間に含まれる任意の ID に nodeID が最も近いノードを探索（lookup）できる. この探索方法について説明する. 本論文では, 探索目標となる ID のことを targetID と表記することにする

各ノードは他のノードへのポインタを記録したルーティングテーブルを持つ. ポインタの形式は既定されていないが, 一般的には IP アドレスとポート番号の組が使われる. ルーティングテーブルは, nodeID とそのノードへのポインタの組を要素にとる. つまり, ポインタを pointer とおくとき, ⟨nodeID, pointer⟩ が一つのルートとなる.

各ノードは複数のノードのルートをルーティングテーブルに保持する. このとき, 近くのノードほど密に, 遠くのノードほど疎に, ルートを持っている. とくに, 隣接するノードのルートは確実に持っている.

探索では, targetID とルーティングテーブル内の nodeID を比較して, その中で targetID と nodeID が最も近いノードを調べ, ポインタを用いてそのノードと通信する, という作業を繰り返す. ルーティングテーブルには隣接するノードのルートが確実に存在するので, 自分のルーティングテーブルから開始して, nodeID が targetID により近いノードを辿っていくことで, 最も近いノードを確実に探索できる.

先に述べたように各ノードは, 疎らではあるが遠くのノードへのルートもルーティングテーブルに保持する. ここで注目すべきは, DHT では ID 空間の広さが有限かつ既知ということである. これを利用して適切なルートをルーティングテーブルに持つことで, DHT はノード数に対してスケーラブルな高速探索を可能とする. それと同時に, 近傍のノードすべてにクエリを伝播させていく flooding を探索で行わないため, DHT はトラフィック量を抑えることができる. 具体的なルートの選定方法はルーティングテーブルによって異なるので後述する.

2.1.9 iterative/recursive

探索では targetID に nodeID がより近いノードを辿っていく．そのときのルーティングの様式には反復（iterative）と再帰（recursive）の 2 種類が考えられている．

iterative では, 要求を受け取ったノードは次のノードのルートを探索元に返答する．探索元は次のノードに要求を送る．この処理を繰り返して目的のノードに要求が届いたとき, そのノードは要求された処理を実行する．

recursive では, 要求を受け取ったノードが次のノードに要求を転送する．iterative と同じく, 最終的に目的のノードに要求が届いたとき, そのノードは要求された処理を実行する．

2.1.10 まとめ

DHT では, ノード数に対して十分に広い ID 空間を使用して, コンテンツの公開・取得・削除を行う．この ID 空間が有限かつ既知であることを利用して, DHT はノード数に対してスケーラブルに高速かつトラフィックを抑えた探索を実現する．

2.2 Overlay Weaver

2.2.1 はじめに

本節では, Overlay Weaver[11][12] について説明する．Overlay Weaver はオーバーレイネットワークの構築ツールキットであり, Apache License Version 2.0 でオープンソースとして配布されている．

図 2.1 は Overlay Weaver のランタイムを構成するコンポーネント群を示している．各コンポーネントでどの実装を使うかを選択できるため, 研究者・設計者は既存の実装を再利用でき, 開発の負担を減らせる．また, エミュレータを用いた数万ノードの実験や, 実ネットワーク上での動作も行える．

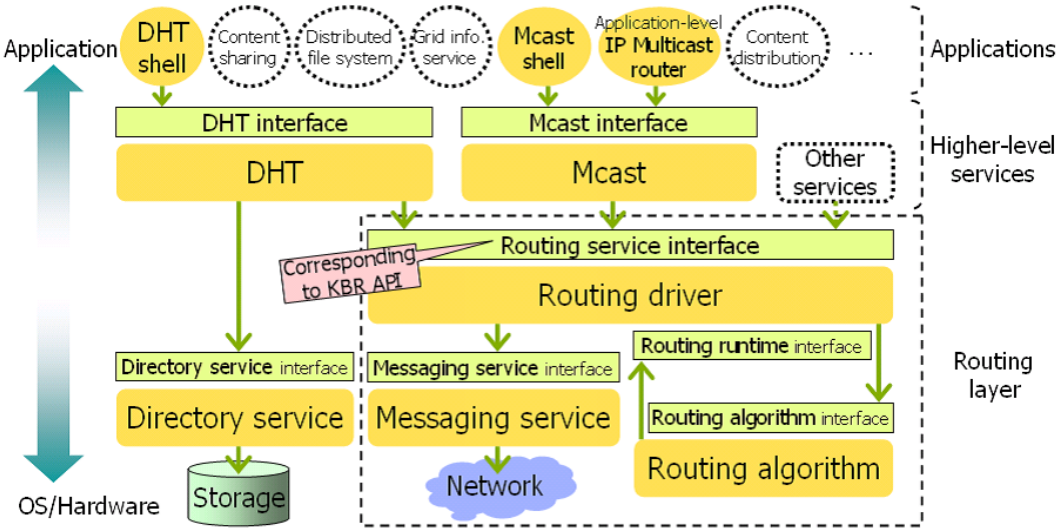

図 2.1　Overlay Weaver のランタイムを構成するコンポーネント群（[12] から引用）

2.2.2　DHT

図 2.1 の DHT は文字通り DHT 層を意味するが, ここでの DHT 層とはルーティングアルゴリズムや HashTable の仕様を含めず, それらを担当するコンポーネントへ処理を委譲するコンポーネントである．他のコンポーネント群を組み合わせて join や put などといったノードの振る舞いに関するアルゴリズムを記述するコンポーネントとも言える．

2.2.3　Mcast

Mcast はオーバーレイマルチキャストを意味する．本論文ではオーバーレイマルチキャストを扱わないので, この説明は割愛する．

2.2.4　ディレクトリサービス

ディレクトリサービスとはノードのデータベースを表す．DHT においては HashTable 実装と同義である．

2.2.5 ルーティングアルゴリズム

ルーティングアルゴリズムを記述するコンポーネントである．Chord[5], Kademlia[6], Pastry[7], Tapestry[8] などのアルゴリズムが既に実装されている．

2.2.6 ルーティングドライバ

ルーティングドライバとはルーティングの様式を記述するコンポーネントである．現在, 反復（iterative）と再帰（recursive）の 2 種類が実装されている．

2.2.7 メッセージングサービス

ノード間の通信方法を記述するコンポーネントである．エミュレータ環境や, TCP, UDP などの実装がある．

2.2.8 DHT shell

DHT shell とは文字通り DHT のシェルを表す．DHT のノードを起動（DHT 層のインスタンスを生成）して, コマンドを受け取り, その処理を行う．

2.2.9 まとめ

Overlay Weaver はオーバーレイネットワーク構築ツールキットのオープンソースである．Overlay Weaver のランタイムは複数のコンポーネントに分割されているため, 研究者・設計者は既存の実装を再利用でき, 開発の負担を減らせる．また, エミュレータを用いた数万ノードの実験や, 実ネットワーク上での動作も行える．

2.3 Chord

2.3.1 はじめに

本節では, DHT のルーティングアルゴリズムの一例として Chord[5] について説明する．Chord は比較的シンプルなアルゴリズムである．

2.3.2 ID

ハッシュ関数にはSHA-1を想定しており, 160 bitのうちm bit ($1 \leq m \leq 160$, $m \in \mathbb{N}$)をID空間に使用する. 距離の定義はIDの単純な差である. ただし例外として, IDの最小値 (0) と最大値 ($2^m - 1$) は隣接していることとする. したがって, 最小値と最大値の差は1となる. つまり, Chordのトポロジーは数直線の最小値と最大値を結んだ円形である.

2.3.3 lookup

Chordでの探索はIDが大きくなる方向で行われる（ただしIDの最小値と最大値が隣接している箇所ではIDが小さくなる方向である）. あるノードにこの方向において最も近いノードを, そのノードのsuccessorという. 逆方向の場合にはpredecessorという. Chordでは3種類のルーティングテーブルを用いる.

まず1つはsuccessorリストと呼ばれるルーティングテーブルである. successorリストは後続ノードのルートをr ($r \in \mathbb{N}$) 個だけ保持する. 後続ノードとはsuccessor, successorのsuccessor, $\cdots$ のことである.

次の1つはprecedessorのルートを持つルーティングテーブルである.

最後の1つはfingerテーブルと呼ばれるルーティングテーブルである. fingerテーブルは2^k ($0 \leq k \leq m-1$, $k \in \mathrm{N}$) 個離れたノードを保持する. これを使用することで, 1ホップ毎に探索範囲を少なくとも2つに分割しながらノードを絞っていくので, ノード数をnとおくと探索のホップ数は$\Theta(\log n)$になる.

2.3.4 put/get/remove

2.1.5~2.1.7節のput/get/removeではnodeIDがtargetIDに最も近いノードを探索したが, Chordの場合にはsuccessorの方向で最も近いノードを探索する. その後の処理は2.1.5~2.1.7節と同様である.

2.3.5 join

ノードがChordのネットワークに参加する手順を説明する. このとき, 最低1ノード, 既にネットワークに参加しているノードと通信できる状態であるとする. 本論文ではこの

ノードのことを初期ノードと呼ぶことにする.

まず, ノードは初期ノードと通信して, 自分の successor と predecessor を探索する. 次に, successor と predecessor のルーティングテーブルを利用して, 自分のルーティングテーブルを構築する. 合わせて, successor から自分が担当するべき HashTable の要素を受け取る. 最後に, 自分が参加した通知を Chord のネットワークに転送させることで, 他ノードのルーティングテーブルを更新する.

2.3.6 stabilization

ノードの離脱や通知の失敗などが原因で successor リストが狂う場合も考えられる. successor を間違えると探索が失敗してしまう. そこで Chord は, successor リストを修復する, stabilization という機構を持つ.

stabilization では, 各ノードは自分の successor に predecessor を定期的に問い合わせる. 本来ならばその返答は自分自身となるはずである. しかし異なる返答が来た場合, 隣接ノードが参加もしくは離脱したことがわかる. この返答を元にして successor リストを最新の状態に修復する.

2.3.7 まとめ

Chord は, 円形トポロジーを形成する, DHT のルーティングアルゴリズムである. ルーティングテーブルには

- successor リスト
- precedessor
- finger テーブル

の 3 つを使用する. 比較的シンプルなアルゴリズムながら, ノード数を n とおくと, 探索のホップ数は $\Theta(\log n)$ になる.

2.4 Kademlia

2.4.1 はじめに

本節では, DHT のルーティングアルゴリズムのもう一つの例として Kademlia[6] について説明する.

2.4.2 ID

Chord と同じく, ハッシュ関数には SHA-1 を想定しており, 160 bit のうち m bit ($1 \leq m \leq 160,\ m \in \mathbb{N}$) を ID 空間に使用する. 距離の定義は ID の排他的論理和である. したがって, 2 つの ID 間の距離に対称性を持つことになる.

2.4.3 lookup

Kademlia では k-bucket というルーティングテーブルを使用する. ノードは, 自分から $2^i \sim 2^{i+1}$ ($0 \leq i < 160,\ i \in \mathbb{N}$) だけ離れたノードを k-bucket に保持する. このとき, それぞれの i に対して最大 k 個のルートを保持する. k-bucket を使用することで, 1 ホップ毎に探索範囲を少なくとも 1 bit ずつ絞りながらノードを探索していくので, ノード数を n とおくと探索のホップ数は $\Theta(\log n)$ になる.

2.4.4 put/get/remove

k-bucket を用いて, 2.1.5~2.1.7 節と同様の処理を行う.

2.4.5 join

ノードは初期ノードと通信して, 自分自身の nodeID を探索する. その経路情報から自分の k-bucket を構築する.

2.4.6 k-bucket の更新

先に述べたように, Kademlia では 2 つの ID 間の距離に対称性を持っている．よって, ある距離で自分がメッセージを送るノードは, 同じ距離で自分へメッセージを送るノードになる．これを利用して, Kademlia では自分に来たメッセージを元に k-bucket の更新を行う．このとき, その i に対する k-bucket の欄が埋まっていた場合, 古くからネットワーク上に存在し今も生存し続けているノードを優先して, そうでないほうを破棄する．これは, そのようなノードのほうがこの先も生存し続ける可能性が高い, という Gnutella[1] の解析結果に基づいている．

2.4.7 まとめ

Kademlia は, k-bucket というルーティングテーブルを持つ, DHT のルーティングアルゴリズムである．このルーティングテーブルは自ノードに来たメッセージを元に更新される．ノード数を n とおくと, 探索のホップ数は $\Theta(\log n)$ になる．

第 3 章

提案手法

本章では, 従来研究の問題点を述べ, それを解決するための手法を提案する.

3.1 従来研究の問題点

3.1.1 はじめに

本節では, 従来の DHT でコンテンツを更新した場合の動作を説明し, その問題点について述べる. 更新の動作のことを本論文では update と呼ぶことにする.

説明にあたって, DHT で公開済みの任意のコンテンツを t ($t \in \mathbb{N}$) 回更新することを考える. このコンテンツを含む pair を $\langle key_0, contents_0 \rangle$, 更新後のコンテンツを $contents_1, contents_2, \cdots, contents_t$ とおく.

3.1.2 same keys

更新後の全コンテンツに対して同じ key を割り当てるアプローチについて述べる. まず, 一意の key である key_c を用意する. $key_0 = key_c$ かどうかはここでは問わない. $1 \leq i \leq t$ ($i \in \mathbb{N}$) となるすべての i に対して $\langle key_c, contents_i \rangle$ を put すれば, key_c の知識を持つノードは更新後の全コンテンツを取得できる. もちろん, key_0 の知識を持つノードは初期のコンテンツ $contents_0$ を取得できる. しかし, 更新後のコンテンツを取得するとき, get 要求先のノードは key_c に対応付けられている複数のコンテンツのうち要求元のノードが欲しているものを判断できないので, 要求元のノードは一度それらすべてをダウ

key	value
key_0	$contents_0$
key_c	$contents_1$
	$contents_2$
	$contents_3$
	$\cdots$
	$contents_t$

表 3.1　same keys における DHT の Map の例

ンロードしなくてはならない. そのため, 余分なコンテンツも同時にダウンロードすることで無駄なトラフィックが生じてしまう.

remove では $hash(contents)$ も指定することで目的のコンテンツのみを削除できたが, この方法は remove を実行するノードがそのコンテンツの公開者であり $hash(contents)$ を知っていることが前提となっている. したがって, ここでの問題に対してこの方法をそのまま用いることはできない.

便宜上, この節で述べた手法を same keys と呼ぶことにする. same keys における DHT の Map を表 3.1 に例示する.

3.1.3　different keys

same keys では key として key_0 と key_c のみを使用したが, 今度は更新後の各コンテンツに対して key を新たに用意してみる. これらの key を $key_1, key_2, \cdots, key_t$ とおく. 例えば, 更新後のコンテンツ $contents_{t+1}$ を追加する場合には key_{t+1} を新たに用意することになる. ここで, $1 \leq i \leq t$ $(i \in \mathbb{N})$ となるすべての i に対して $\langle key_i, contents_i \rangle$ を put すれば, get 要求で使う key を使い分けることで, 取得時に欲しいコンテンツのみのダウンロードが可能となる. よって, 先に述べた無駄なトラフィックが生じる問題を防げる. しかしながら, この手法では取得元のノードは取得したいコンテンツに対応する key の知識を持つ必要があり, 更新後のコンテンツが増える度に新しく生成する key の知識をどのようにオーバーレイネットワーク上に広めていくかが課題となる.

この節で述べた手法を different keys と呼ぶことにする. different keys における DHT

key	value
key_0	$contents_0$
key_1	$contents_1$
key_2	$contents_2$
key_3	$contents_3$
...	...
key_t	$contents_t$

表 3.2　different keys における DHT の Map の例

の Map を表 3.2 に例示する.

3.1.4　group keys

same keys と different keys では, t 個の更新後のコンテンツに対して key を 1 個または t 個用意したが, いくつか用意するという選択肢も存在する. 例えば, 更新後のコンテンツを任意の方法でグループ分けしてそのグループごとに新しい key を割り当てるという案が考えられる. しかし, この方法でも結局は上記と同様の問題が発生する.

この節で述べた手法を group keys と呼ぶことにする. group keys における DHT の Map を表 3.3 に例示する.

key	value
key_0	$contents_0$
key_1	$contents_1$
key_2	$contents_2$
	$contents_3$
...	...
key_t	$contents_t$

表 3.3　group keys における DHT の Map の例

3.1.5 まとめ

従来の DHT でコンテンツを更新する場合には, コンテンツの取得時に余分なコンテンツも同時にダウンロードしてしまいトラフィック量が増える, もしくは新しい key の知識をオーバーレイネットワーク上に広めなくてはならない, という問題が生じてしまう. この問題はコンテンツを頻繁に更新する環境で顕著に顕れることが予想される.

3.2 解決手法

3.2.1 はじめに

本節では, 前節で述べた問題点を解決する手法を提案する.

3.2.2 検索フレーム

本提案手法では検索システムに下記の 3 つのフレームを設ける.

- 各バージョンのコンテンツを扱う version フレーム
- バージョンの管理を行う history フレーム
- 検索の入り口となる attribute フレーム

各ノードはこれらのフレームごとに専用の HashTable を持つ. 例えば, version フレームで pair の put 要求があった場合には, 要求先ノードは version フレーム用の HashTable にこのペアを加える. そして, version フレームで get 要求があった場合には, 要求先ノードは version フレーム用の HashTable から指定された key に対応付けられている value を返す.

以降の節で, 本提案手法における put, update, remove, get を説明する. その際, 従来の DHT における当該動作と区別するために, 従来の DHT の場合には version-, history-, attribute-というフレーム別の接頭辞をつけて表記する. 例えば, history-remove と表記したときには history フレームで従来の DHT における remove を実行することを意味する. また, $hash()$ は引数を ID 空間に写像するハッシュ関数, {} は中の要素を区別して取り出せる集合とする. 説明の補助として, 提案手法における DHT の Map を表 3.4 に例示する.

frame	key	value
version	$hash(contents_0)$	$contents_0$
	$hash(contents_1)$	$contents_1$
	$hash(contents_2)$	$contents_2$
	$\cdots$	$\cdots$
history	$hash(contents_0)$	$\{hash(contents_0), hash(contents_1), hash(contents_2), \cdots\}$
attribute	$hash(name_0)$	$hash(contents_0)$
	$hash(attr_a)$	$hash(contents_0)$
	$hash(attr_b)$	$hash(contents_0)$
	$\cdots$	$\cdots$

表 3.4　提案手法における DHT の Map の例

3.2.3　put

コンテンツを公開する手順について述べる．ある初期コンテンツ $contents_0$ を DHT で公開することを考える．$contents_0$ の name を $name_0$ とおく．

はじめに $\langle hash(contents_0), contents_0\rangle$ を version-put する．次に $\langle hash(contents_0), \{hash(contents_0)\}\rangle$ を history-put する．最後に $\langle hash(name_0), hash(contents_0)\rangle$ を attribute-put する．これでコンテンツの公開は完了である．

この動作が本提案手法における put である．history-put した {} 区切りの集合を以降，HistoryInfo と呼ぶことにする．公開した pair の使用方法についてはおいおい説明していく．

3.2.4　update

コンテンツを更新する手順について述べる．初期コンテンツ $contents_0$ をコンテンツ $contents_c$ で更新することを考える．

まず，$\langle hash(contents_c), contents_c\rangle$ を version-put する．次に，put 時に history-put した $\langle hash(contents_0), \{hash(contents_0)\}\rangle$ の value を $\{hash(contents_0), hash(contents_c)\}$ に変更する．コンテンツの更新操作はここまでである．

この動作が本提案手法における update である．さらに $contents_0$ の更新を続ける場合には, 同様の処理を繰り返せばよい．このとき, history フレームでは $\{hash(contents_0), hash(contents_1), hash(contents_2), \cdots\}$ のように HistoryInfo の要素を増やしていく．

3.2.5 remove

コンテンツを削除する手順について述べる．初期バージョンが $contents_0$ であるコンテンツ $contents_c$ を削除することを考える．

まず, history-put してある $\langle hash(contents_0), \{hash(contents_0), \cdots, hash(contents_c), \cdots\}\rangle$ の HistoryInfo の要素から $hash(contents_c)$ を削除する．次に $\langle hash(contents_c), contents_c\rangle$ を version-remove する．これでコンテンツの削除操作は完了である．

ただし HistoryInfo が空, つまり $\{\}$ になるときには, history フレームの $\langle hash(contents_0), \{\}\rangle$ および attribute フレームの $\langle hash(name_0), hash(contents_0)\rangle$ を history/attribute-remove するか, その削除を DHT の TTL（pair の有効期間）機構による自然消滅に委ねる．

この動作が本提案手法における remove である．

3.2.6 get

コンテンツを取得する手順について述べる．初期バージョンが $contents_0$ であるコンテンツ $contents_c$ を取得することを考える．

まず, $hash(name_0)$ を key として $hash(contents_0)$ を attribute-get する． 次に, $hash(contents_0)$ を key として $\{hash(contents_0), \cdots, hash(contents_c), \cdots\}$ を history-get する．あとはこの HistoryInfo から $hash(contents_c)$ を取り出して, これを key とすれば目的の $contents_c$ を version-get できる．

この動作が本提案手法における get である．

3.2.7 属性検索

属性検索とは name 以外の情報で検索することである．例えば, コンテンツの作者・公開者・作成日時・公開日時などによる検索が挙げられる．

本提案手法では, コンテンツの name の代わりに属性を ID 空間へ写像しても, 同様に検

索を行うことができる．例えば, 属性 $attr$ で検索を行えるようにするには, put するときに $hash(name)$ の代わりに $hash(attr)$ で attribute-put しておけばよい．get するときには $attr$ の知識があるノードは, $hash(name)$ ではなく $hash(attr)$ で attribute-get すれば目的のコンテンツまで辿り着ける．つまり, 本研究手法においては ID 空間に写像できさえすれば name も属性も違いはない．別の言い方をすれば, 本研究手法では name も 1 つの属性として考えられる．ゆえに, 検索の入り口となるフレームを attribute フレームと名付けたのである．

3.2.8 分散特性

本提案手法で初期コンテンツおよび更新後のコンテンツを公開するとき, value の $contents$ に対応する key はすべて $hash(contents)$ である．このハッシュ関数が異なる引数を十分な確率で異なる値に写像するならば, key と value は十分な確率で一対一対応となるので, $hash(contents_c)$ を key とした get 要求に $contents_c$ 以外のコンテンツがヒットする事態は, 十分な確率で防げる．これにより, 3.1 節で述べた, 従来の DHT でコンテンツを更新するときの問題点を解決できた．また, $contents_c$ に対して $hash(contents_c)$ が ID 空間に十分に散らばるならば, コンテンツの追加先ノードを散らばらせるという DHT の分散特性を保つ.

3.2.9 HistoryInfo

HistoryInfo の形式は本研究では限定しないが表 3.5 の XML フォーマットを例示しておく.

```
<history>
  <version>
    <id>xxxxxxxxxxxxxxx</id>
    <id>xxxxxxxxxxxxxxx</id>
  </version>
</history>
```

表 3.5　HistoryInfo の例

get において, $hash(contents_0) = hash(contents_c)$ ならば, $contents_c$ に関する history-get 先ノードと version-get 先ノードは一致するので, ノード探索を 1 回短縮できる. $\langle hash(contents_0), \{hash(contents_0), hash(contents_1), hash(contents_2), \cdots\}\rangle$ を history-put していることにこれは起因する. ここで, この pair の key に使われるコンテンツ（上の例では $contents_0$）を primary version と呼ぶことにする. もし, update などのときに primary version を $contents_0$ から $contents_1$ に変更して, $\langle hash(contents_1), \{hash(contents_0), hash(contents_1), hash(contents_2), \cdots\}\rangle$ を history-put し直したならば, $hash(contents_1) = hash(contents_c)$ のときにノード探索を 1 回短縮することができるようになる. したがって, あらかじめ get 要求の多いコンテンツがわかるならば, 全体の探索回数を減らせる可能性がある. ただしこの方法を使う場合には, 先の例で言えば attribute-put する pair も合わせて $\langle key_c, hash(contents_0)\rangle$ から $\langle key_c, hash(contents_1)\rangle$ へと変更する必要がある.

また, HistoryInfo を閲覧したうえで取得するコンテンツのバージョンを決定するという環境では, HistoryInfo に各バージョンの要約や Bloom Filter[13] を用いた縮約などを含めておくことで, バージョン決定の補助とすることができる.

3.2.10 まとめ

本小節では提案手法についてまとめる.

3.1 節で, 従来の DHT でコンテンツを更新した場合には次のどちらかもしくは両方の問題点があることを述べた.

- コンテンツ取得時に目的以外のコンテンツも同時にダウンロードしてしまうため, 無駄なトラフィックが生じる.
- 各バージョンのコンテンツに対応する key の知識をどのようにネットワーク上に広めていくかが課題となる.

これらの問題はコンテンツを頻繁に更新する環境で顕著に顕れることが予想される.

その解決策として本研究では DHT の新しい検索方式を提案した. この手法は name での検索と同様に属性検索も可能である. また, 提案手法はルーティングアルゴリズムに依存していないので, システム構築時に目的に合わせてトポロジとノードの探索方法の選定をできるという DHT の利点を維持している.

第 4 章

実験環境の構築

4.1 はじめに

本研究では, 提案手法を評価するための実験を行った. 本章では, その実験環境の構築方法について述べる.

本実験は, 1 台のマシンで複数のノードをエミュレートして DHT ネットワークを形成して行う. エミュレーション環境には Overlay Weaver[11][12] を利用した.

4.2 Multi-Directory DHT

本節では 3.2.2 節で述べた検索フレームの実装について述べる.

検索システムに検索フレームの設置するためには, ノードにそれぞれのフレームごとの HashTable を用意する必要がある. だが, Overlay Weaver で提供されている DHT 層の実装では, ノードは 1 つのディレクトリ（ここでは HashTable と同義）しか持たない. そこで, 本実験システムでは複数のディレクトリを持つことが可能な DHT 層, Multi-Directory DHT（MD DHT と略記）を実装した. MD DHT ではディレクトリを任意の数だけ持つことができ, この個数は外部の設定コンポーネントで設定できる仕組みになっている. 本システムでは version, history, attribute フレーム用の 3 つのディレクトリを持つように設定している.

ノードが複数のディレクトリを持つだけで各フレームにおいて登録や検索などの処理が可能になる, というわけではない. 処理の要求を受け取ったノードがどのフレームでその処理をすべきかを判断できなくてはならない. 本実験システムでは Overlay Weaver の

メッセージングサービスを書き換えてこの課題に対処した．書き換えた箇所はメッセージのタグである．Overlay Weaver ではメッセージのタグを元にそのメッセージが何の要求や返答であるかを判断している．しかし, Overlay Weaver でのタグの標準実装には当然フレームを示す情報が載っていないために, 要求先ノードはどのフレームで処理をすべきかを判断できなくなる．そこで, 本実験システムでは各フレーム用の要求のタグを追加した．あるフレームにおいて要求を行う場合には, メッセージのタグをそのフレームに対応したタグに決定することで, 要求先ノードは届いたメッセージのタグからどのフレームで処理をすべきかを判断できる．これを踏まえて, MD DHT では対象ディレクトリのフレームをパラメータで与えて送信メッセージのタグを決定する仕組みや, 受信メッセージのタグから対象ディレクトリを決定する仕組みを備えている．

4.3 オペレーションの実装

get, put, remove, update のオペレーションは MD DHT の上層の独自コンポーネントで実装した．アプリケーション層の DHT シェルなどからこのコンポーネントを介して MD DHT を利用することで各オペレーションは実現される．ただし今回の実装では実験のために, 4.5 節で述べるデータサイズのシミュレーション処理を加えている．

4.4 メッセージ集計器の実装

Overlay Weaver にはメッセージ集計の機能が存在するが, 現在完全に実装されているメッセージ集計器は TCP/UDP 用のみであり, エミュレーション環境用は通信量計測の部分が実装されていない．そのため, 本研究では実験で用いるメッセージ集計器を実装した．

このメッセージ集計器は Overlay Weaver の TCP/UDP 用の実装に倣って, メッセージを構成する Java オブジェクトをシリアライズしたときの容量を通信量として計測する．ただし, DHT の value については 4.5 節で述べるデータサイズを通信量としている．

4.5 データの実装

本実験では DHT の value に, 実際のファイルではなくそれをシミュレートしたデータを使用する．このデータは, どのフレームの value であるかを表す情報や, 識別子などを記

```
<?xml version=”1.0” encoding=”UTF-8”?>
<history>
  <primary>
    <id>xxxxxxxxxxxxxxx</id>
  </primary>
  <version>
    <id>xxxxxxxxxxxxxxx</id>
    <id>xxxxxxxxxxxxxxx</id>
  </version>
</history>
```

表 4.1　本実験における HistoryInfo の形式

録している．4.4 節のメッセージ集計器はこれを元にデータサイズを計測する．本節では，本実験における各フレームの value のサイズについて述べる．

attribute フレームの value, つまり ID 形式の value のサイズはこの ID を 16 進出力したときの容量とする．したがって, ID 空間が 160 bit のときにはこの value のサイズは 40 byte になる．

history フレームの value, すなわち HistoryInfo のサイズは表 4.1 に示す XML フォーマットに従うファイルの容量とする．primary タグは 3.2.9 節で述べた primary version を ID 空間にマップしたハッシュ値を示している．ただし, 本実験では primary version の変更は行わないので primary version は必ず初期データとなる．

version フレームの value, つまりコンテンツのサイズは起動時のパラメータで与える．

4.6　シナリオ自動生成器の実装

Overlay Weaver のエミュレーション環境には, シナリオファイルを入力して DHT シェルなどのアプリケーションにコマンド（get や update などのオペレーションの命令）を送る機能が備わっている．Overlay Weaver のエミュレータの構造を図 4.1 に示す．このシナリオファイルは手作業で作成することも可能であるが, ノード数やコマンド数などが膨大になるにつれ手作業での作成には手間がかかるようになる．したがって, シナリオファ

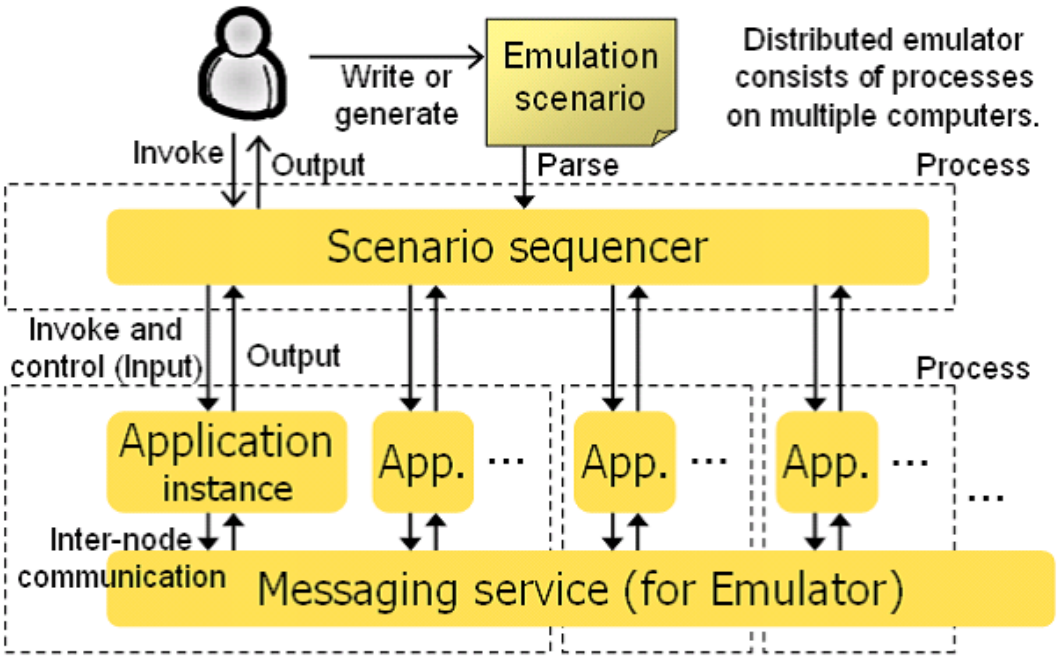

図 4.1　Overlay Weaver のエミュレータの構造（[12] から引用）

イルの自動生成器があると好ましい.

シナリオは闇雲に作ればいいわけではない. 例えば, 初期コンテンツがなければアップデートはできないので, update のコマンドは該当コンテンツの put のコマンドより後に来なければならない. また同様に, 存在しないコンテンツに対して削除はできないので, remove のコマンドは該当コンテンツの put もしくは update のコマンドより後に来なければならない.

本研究では, このような制約に従いつつシナリオファイルを生成する, シナリオ自動生成器を実装した. 以下にその生成手順を説明する.

まずはじめに, コマンドの回数の比を決定する. 例えば, $get : put : remove : update = 1 : 2 : 1 : 1$ のように決める.

次に, コマンドの回数のスケールを決定する. これが全コマンドの合計数になる. 例えば, コマンドの比率が $get : put : remove : update = 1 : 2 : 1 : 1$ で, コマンドのスケールが 500 ならば, put を 200 回, get, remove, update を 100 回ずつ行う. ただし, 端数があるとスケールと合計数が異なる場合もある. 端数の扱い方は Java の java.math.RoundingMode の丸めモードで指定できる.

コマンドの回数と比とスケールから算出したら, 指定した方法でソートもしくはシャッフルを行い, コマンドを並べる.

初期ノード数をパラメータで受け取って, ノードプールに初期ノードを生成する. このプールはオーバーレイネットワークに参加中のノードを格納している. これとは別に, オーバーレイネットワーク上で共有しているコンテンツを格納するコンテンツプールも用意している. コンテンツプール内のコンテンツは公開元ノードや取得済みノードなどを記憶している.

並べたコマンドを順番に読み取ってノードプールとコンテンツプールから適切な引数を割り当てていく. 例えば put コマンドでは, ノードプールに格納されているノードから 1 つだけ選び, put を実行するノードとする. 次に, 新たなコンテンツを生成してコンテンツプールに入れ, そのコンテンツを put の対象とする. このとき, コンテンツは公開元ノードを記憶する. また remove コマンドでは, コンテンツプールに格納されているコンテンツから 1 つだけ選んで削除し, そのコンテンツを remove の対象とする. 同時に, コンテンツの公開元ノードを調べて, remove を実行するノードとする（本実験ではコンテンツの公開元ノードのみがそのコンテンツの更新・削除を実行できるという環境を想定している）.

コマンド実行の時間間隔やルーティングアルゴリズムといったパラメータ群を設定ファイルやテンプレートファイルなどから指定して, シナリオファイルを出力する. これでシナリオの生成は完了である.

このシナリオ自動生成器は本研究の実験シナリオだけでなく, 他の環境の実験シナリオにも拡張できるように設計している. 先に述べた各動作はそれぞれコンポーネントに分かれており, コンポーネント毎に差し替えることが可能である. 例えば, コマンドに引数を割り当てるコンポーネントを差し替えて, 引数の選定方法を変更したり新しいコマンドを追加したりできる.

4.7 まとめ

エミュレーション環境には Overlay Weaver を用いた. 提案手法を実装するために, 複数のディレクトリを持つことが可能な DHT 層, Multi-Directory DHT（MD DHT）を実装した. そして合わせて, Overlay Weaver のメッセージングサービスを書き換えた. アプリケーション層から, MD DHT の上層に実装したオペレーション用のコンポーネントを介して, MD DHT を利用することで get, put, remove, update のオペレーションは実

現される. さらに, シナリオ自動生成器と本実験用のメッセージ集計器を実装した.

第 5 章

評価実験と考察

本章では, 評価実験と, そこから得られる考察について述べる.

本研究の提案手法の優位性は 3.2.10 節で述べたとおりである. このうち, コンテンツ取得時に目的以外のコンテンツも同時にダウンロードしなくなることにより無駄なトラフィック量を軽減できる, という利点を実験によって確かめる. 合わせて, 提案手法によってオーバーレイ・ネットワーク上のトラフィック量がどのように増減するのかを調べる.

5.1 計測対象

本節では実験での計測対象について述べる.

実験では以下の 4 つを計測する. すべて DHT 上でのみ計測し, 下位ネットワークは考慮しない. また, タグが Overlay Weaver の標準実装で計測対象となっているメッセージのみを本実験での計測対象とする.

- アップロードサイズ
- ダウンロードサイズ
- メッセージサイズ
- メッセージ数

アップロードサイズはアップロードした value の通信量を示す. Overlay Weaver の DHT 実装では put, remove メソッドでその引数の key にそれまで割り当てられていた value をダウンロードしているが, 現実のアプリケーションでは必ずしもそうする必要はなく, その value の用途がなければ通信コストの無駄となるだけである. 今回の実験では,

put, remove メソッドでその成否のみを応答として受け取る環境を想定し, そのときにダウンロードする value のサイズは計測対象としない. ただし, value 以外にメッセージに含まれるシグネチャやタグなどのヘッダについては, 成否を示す応答としてメッセージサイズの項目で計測する.

ダウンロードサイズはダウンロードした value の通信量を示す.

メッセージサイズはアップデートサイズとダウンロードサイズを除いた通信量を示す. つまり, value を除いた通信量を示すことになる.

メッセージ数は通信回数を示す.

5.2 比較対象

5.2.1 はじめに

本節では実験の比較対象について述べる. 本実験では 3 つのアルゴリズムを比較する.

5.2.2 提案手法

比較対象の 1 つは本研究の提案手法である.

5.2.3 従来手法 ①

次の 1 つは従来の DHT である. これを従来手法 ① と呼ぶことにする. コンテンツの更新は, 3.1.2 節で述べた same keys で行う. 本実験では, 更新コンテンツ用の key に初期コンテンツの key を使用することとした. same keys を選んだ理由は, 3.1.3 節の different keys, 3.1.4 節の group keys では, key の知識をネットワーク上に広めることが課題となり, それを解決しない限り検索が成立しないためである.

5.2.4 従来手法 ②

最後の 1 つも従来の DHT である. これを従来手法 ② と呼ぶことにする. コンテンツの更新も同様に same keys で行う. 従来手法 ① との違いは次節で述べる.

5.2.5 従来手法 ① と ② における属性検索

従来手法 ① と ② の違いは属性検索の方法である.

まず, どちらの手法でもコンテンツを公開するノードは $\langle hash(name), contents \rangle$ を put する. これで通常通り, name からコンテンツの取得が可能になる. ここで, 属性を ID 空間にマップした値を $hash(attr)$ とおく.

次に, 従来手法 ① では, コンテンツの公開元ノードは $\langle hash(attr), contents \rangle$ を put する. これにより, attr からもコンテンツの取得が可能になる.

それに対して, 従来手法 ② では, コンテンツの公開元ノードは $\langle hash(attr), hash(name) \rangle$ を put する. この場合には, attr から hash(name) を取得した後, これを key とすることでコンテンツの取得が可能になる.

従来手法 ② ではノード探索が 1 回増えることになるが, 従来手法 ① にはコンテンツを更新する度に属性の数だけそのコンテンツを put しなおすという欠点がある.

key	value
$hash(name)$	$contents$
$hash(attr_a)$	$contents$
$hash(attr_b)$	$contents$
$\cdots$	$\cdots$

表 5.1 従来手法 ① における DHT の Map の例

key	value
$hash(name)$	$contents$
$hash(attr_a)$	$hash(name)$
$hash(attr_b)$	$hash(name)$
$\cdots$	$\cdots$

表 5.2 従来手法 ② における DHT の Map の例

5.2.6 まとめ

本実験では 3 つのアルゴリズムを比較する．比較対象の 1 つは本研究の提案手法であり, 残り 2 つは従来手法である．従来手法の 2 つは属性検索の方法が異なる．

5.3 実験シナリオ

本節では実験で使用するシナリオについて述べる．実験シナリオは第 4.6 節で述べたシナリオ自動生成器を用いて作成した．

初期ノード数を 10000 とする．ノードは途中参加／離脱をしない．要するにノード数は常に 10000 である．

get を 3000 個, put, remove, update を 1000 個ずつの計 6000 個のコマンドを用意する．コマンドの並び替えでは一様分布に従いランダムにシャッフルした．ただし, put コマンドを先に実行しなければ DHT 上に共有コンテンツが存在しないためにその他のコマンドを実行できないので, 例外的に 500 個の put コマンドを先頭に並べる．コマンドへの引数の割り当てでも, ノードプールとコンテンツプールから一様分布に従いランダムに要素を抽出する．

ID 空間は 160 bit, ルーティングアルゴリズムは Kademlia[6], メッセージングサービスは Iterative である．耐 churn 手法 [14] は導入していない．

属性数とコンテンツのサイズを変更したときの各比較対象の様子を計測・考察する．

5.4 実験結果

5.4.1 はじめに

本節では実験結果を示す．この実験結果において属性数は name を含めて数えている．

5.4.2 コンテンツサイズに関する実験結果

図 5.1, 5.2, 5.3 は順に, コンテンツサイズを変更したときのアップロードサイズ, ダウンロードサイズ, その合計を示している．属性数は 3 個である．

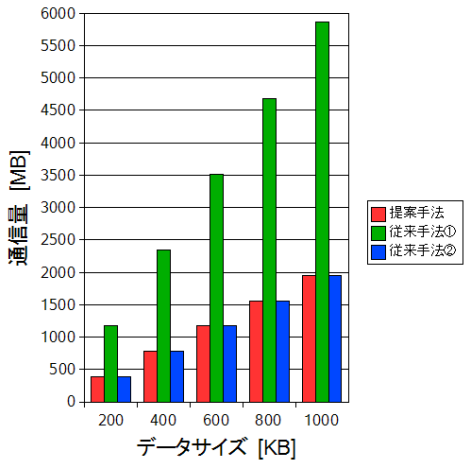

図 5.1　コンテンツサイズを変更したときのアップロードサイズ

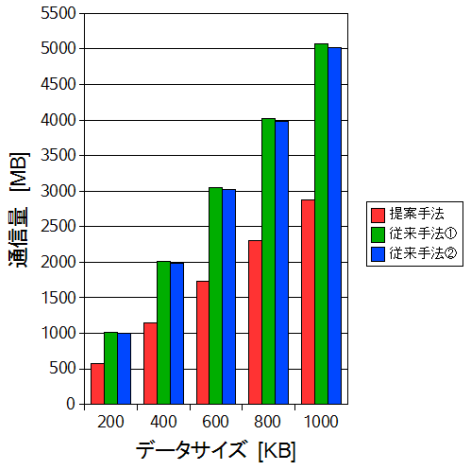

図 5.2　コンテンツサイズを変更したときのダウンロードサイズ

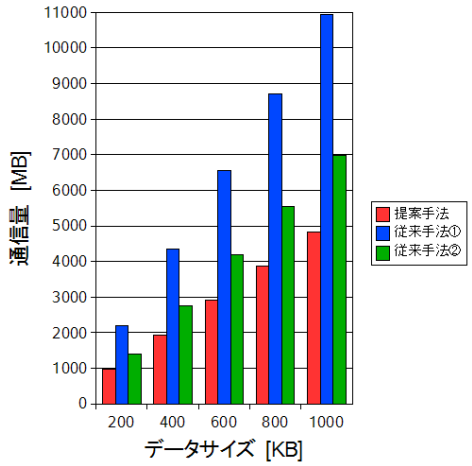

図 5.3　コンテンツサイズを変更したときのアップ／ダウンロードサイズ

5.4.3　属性数に関する実験結果

図 5.4, 5.5, 5.6 は順に, 属性数を変更したときのアップロードサイズ, ダウンロードサイズ, その合計を示している. コンテンツサイズは 500 KB である.

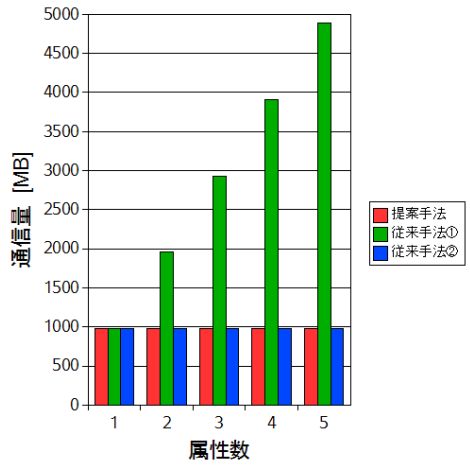

図 5.4　属性数を変更したときのアップロードサイズ

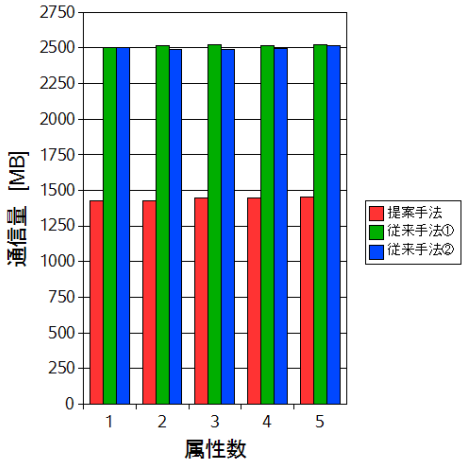

図 5.5　属性数を変更したときのダウンロードサイズ

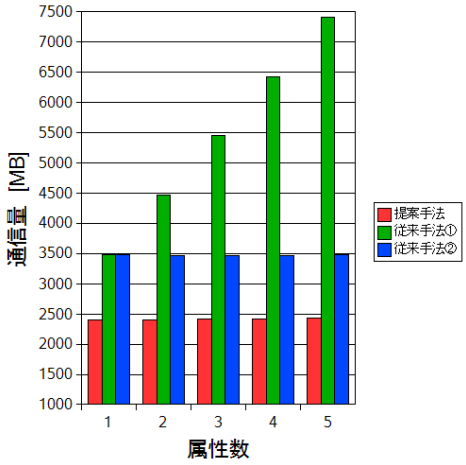

図 5.6　属性数を変更したときのアップ／ダウンロードサイズ

5.4.4　メッセージ数／サイズに関する実験結果

図 5.7, 5.8 は順に, 属性数を変更したときメッセージ数, メッセージサイズを示している. 図 5.8 の通信量に図 5.6 の通信量を加算したものが図 5.9 である.

メッセージ数とメッセージサイズの計測ではコンテンツサイズは含まれないので, コンテンツサイズを変更させたときのメッセージ数とメッセージサイズは計測していない.

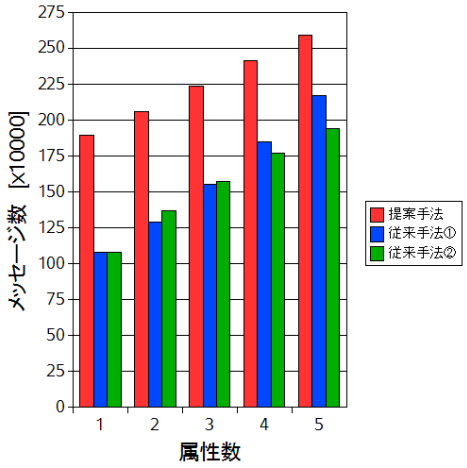

図 5.7　属性数を変更したときのメッセージ数

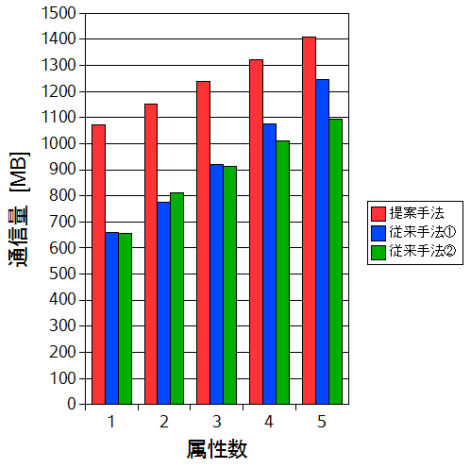

図 5.8　属性数を変更したときのメッセージサイズ

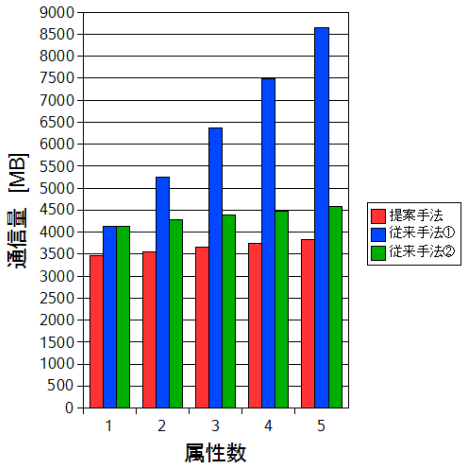

図 5.9　属性数を変更したときの全通信量

5.4.5　まとめ

本節では実験結果を示した．これらの結果から得られる考察を次の節で述べる．

5.5　考察

5.5.1　はじめに

本節では実験結果についての考察を行う．

5.5.2　コンテンツサイズに関する考察

まず図 5.1, 5.2, 5.3 の結果について考察する．従来手法 ① では更新の都度すべての属性に対して更新後のコンテンツを put するので, コンテンツサイズが大きくなるにつれてアップロードサイズが増大している．提案手法ではコンテンツ取得時に目的以外のコンテンツも同時にダウンロードすることがないので, ダウンロードサイズがどちらの従来手法よりも小さくなっている．さらに, コンテンツサイズが大きくなるにつれ, 余分なコンテンツをダウンロードするか否かの違いが顕著になり, ダウンロードサイズの差が広がっていく．このダウンロードサイズの差により, アップ／ダウンロードの合計サイズは, コンテン

ツサイズが大きいほど提案手法のほうが従来手法よりも小さくなった．以上の結果から提案手法の利点を確認できたことになる．

5.5.3 属性数に関する考察

次に図 5.4, 5.5, 5.6 の結果について考察する．先ほどと同様に, 従来手法 ① では属性数が増えるにつれてアップロードサイズが増大している．同じく, 提案手法ではダウンロードサイズがどちらの従来手法よりも小さくなっている．これにより, アップ／ダウンロードの合計サイズは, 提案手法のほうが従来手法よりも小さくなった．ただし, 提案と従来手法 ② の差は属性数を増加させても開いていかない．これらの結果からも提案手法の利点を確認できたことになる．

5.5.4 メッセージ数／サイズに関する考察

最後に図 5.7, 5.8, 5.9 の結果について考察する．ここでも同様に, 従来手法 ① では属性数が増えるにつれてメッセージ数とメッセージサイズが増大している．そして, 提案手法でもこれらが増大していると同時に, 提案手法では従来手法と比較して高い数値となっている．この原因は検索フレームを導入したことでノード探索が増えたためと思われる．メッセージサイズにアップ／ダウンロードの合計サイズを加えた結果においては, 提案手法が従来手法より少ない通信量を維持している．これは, アップ／ダウンロードの合計サイズの差がメッセージサイズの差よりも大きいためである．したがって, 本実験のようにコンテンツサイズがメッセージ一つあたりのサイズに対して十分大きい環境でなければ, 提案手法は従来手法と比べて通信量が多くなってしまう可能性がある．ゆえに, 本研究手法はコンテンツサイズが十分に大きい環境で真価を発揮する手法であると言える．

5.5.5 まとめ

実験結果についての考察をまとめる．コンテンツ取得時に目的以外のコンテンツも同時にダウンロードしなくなることにより無駄なトラフィック量を軽減できる, という提案手法の利点を確認できた．しかしその一方, 提案手法ではメッセージ数とメッセージサイズが増大するという弱点も示された．この理由として, 検索フレームの導入でノード探索が増えたことが考えられる．本実験のようにコンテンツサイズがメッセージ一つあたりのサ

イズに対して十分大きい環境でなければ, 提案手法は従来手法と比べて通信量が多くなってしまうかもしれない. つまり, 本研究手法はコンテンツサイズが十分に大きい環境において利用価値がある.

第 6 章

結論

本章では, 本論文のまとめと今後の課題を述べる.

6.1 本論文のまとめ

本論文では, コンテンツの更新を考慮した, DHT の新しい検索方式の提案を行った.

第 1 章では, 本論文の概要, 目的, 構成について述べた.
第 2 章では, 本研究と関連のある従来研究について述べた.
第 3 章では, 従来研究の問題点について述べ, それを解決する手法を提案した.
第 4 章では, 実験環境の構築方法について述べた.
第 5 章では, 評価実験と, そこから得られる考察について述べた.

従来の DHT でコンテンツを更新した場合には次のどちらかもしくは両方の問題点が存在する.

- コンテンツ取得時に目的以外のコンテンツも同時にダウンロードしてしまうため, 無駄なトラフィックが生じる.
- 各バージョンのコンテンツに対応する key の知識をどのようにネットワーク上に広めていくかが課題となる.

本研究の提案手法は以下の 3 つの要件を満たす.

- 上記のコンテンツの更新に関する問題を解決する.
- 属性検索が可能である.
- ルーティング・アルゴリズムに非依存である.

実験結果から, コンテンツ取得時に目的以外のコンテンツも同時にダウンロードしなくなることにより無駄なトラフィック量を軽減できる, という本提案手法の利点を確認できた. しかしその一方, 提案手法ではメッセージ数とメッセージサイズが増大するという弱点も実験結果で示された. この理由として, 提案手法においてノード探索が増えたことが考えられる. コンテンツサイズがメッセージ一つあたりのサイズに対して十分大きい環境でなければ, 提案手法は従来手法と比べて通信量が多くなってしまう可能性も考えられる. したがって, 本研究手法はコンテンツサイズが十分に大きい環境において高い効果を得られる手法である.

6.2 今後の課題

関根らの研究 [15] も構造化オーバーレイでのコンテンツの更新を扱っている. [15] ではコンテンツとして, リアルタイムで変動するセンサデータを想定している. そして, DHT に PHT（Prefix Hash Tree）[16][17] を適用することで範囲検索に対応させた構造化オーバーレイにおいて, 負荷の分散やメッセージ数／サイズの削減などを目指している. 本研究とは想定している環境が異なるが, コンテンツの更新時にノード探索の履歴を利用してメッセージ数を削減するといった, 本研究の提案手法と併用できる手法も考えられている. とくに, 本研究の提案手法では 5.5 節で述べたようにメッセージ数／サイズがネックとなっているので, 今後このような手法の導入を検討していく必要がある.

また, 松田の研究 [18] では複数属性の範囲を指定して範囲検索を行える構造化オーバーレイを提案している. 本研究の提案手法も PHT を適用すれば範囲検索を行えるようになる. しかし, [18] のように同時に複数属性を指定することはできない. したがって, このようなより柔軟な検索に対応するためには本提案手法をさらに改良していく必要がある.

参考文献

[1] “Gnutella.com”, http://www.gnutella.com/.

[2] “Winny - Wikipedia”, http://ja.wikipedia.org/wiki/Winny.

[3] J. Aspnes and G. Shah: “Skip graphs”, In Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 384-393 (Jan. 2003).

[4] W. Pugh: “Skip lists: a probabilistic alternative to balanced trees”, Communications of the ACM 33, pp. 668-676, (Jun. 1990).

[5] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan: “Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications”, In Proc. of ACM SIGCOMM 2001, pp. 149-160 (Aug. 2001).

[6] P. Maymounkov and D. Mazieres: “Kademlia: A Peer-to-peer Information System Based on the XOR Metric”, In Proc. of IPTPS 2002, pp. 53-65 (Mar. 2002).

[7] A. Rowstron and P. Druschel: “Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems”, In Proc. of IFIP/ACM Middleware 2001, pp. 329-350 (Nov. 2001).

[8] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph: “Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing”, U. C. Berkeley Technical Report, UCB/CSD-01-1141 (Apr. 2001).

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker: “A Scalable Content-Addressable Network”, In Proc. of ACM SIGCOMM 2001, pp. 161-172 (Aug. 2001).

[10] “PlanetLab — An open platform for developing, deploying, and accessing planetary-scale services”, http://www.planet-lab.org/.

[11] 首藤一幸, 田中良夫, 関口智嗣: “オーバレイ構築ツールキット Overlay Weaver”, 先

進的計算基盤システムシンポジウム (SACSIS 2006), pp.183-191 (May 2006).

[12] “Overlay Weaver: An Overlay Construction Toolkit”, http://overlayweaver.sourceforge.net/index-j.html.

[13] B. Bloom: “Space/Time Trade-Offs in Hash Coding with Allowable Errors”, Communications of ACM, vol. 13, no. 7, pp. 422-426 (Jul. 1970).

[14] 首藤一幸: “下位アルゴリズム中立な DHT 実装への耐 churn 手法の実装”, 第 19 回コンピュータシステム・シンポジウム (ComSys 2007), pp. 191-198 (Nov. 2007).

[15] 関根理敏, 瀬崎薫: “P2P ネットワークにおける高頻度なデータ更新を考慮した適応的コンテンツ配置法”, 電子情報通信学会技術研究報告 情報ネットワーク (IN 2006), pp. 197-202 (Mar. 2007).

[16] S.Ramabhadran, S.Ratnasamy, J.M.Hellerstein, and S.Shenker: “Prefix Hash Tree: An Indexing Data Structure over Distributed Hash Tables”, IRB Technical Report (Feb. 2004).

[17] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, J. Hellerstein, and S. Shenker: “A Case Study in Building Layered DHT Applications”, In Proc. of ACM SIGCOMM 2005, pp. 97-108 (Aug. 2005).

[18] 松田哲史: “複数属性を持つデータの属性値範囲指定検索用分散ネットワーク構成方式の提案”, 情報処理学会研究報告 分散システム/インターネット運用技術 (DSM-44), pp. 19-23 (Mar. 2007).

謝辞

本修士論文は, 私が早稲田大学大学院 理工学研究科 村岡洋一研究室に在籍する期間に行った研究をまとめたものです.

本研究を行う機会を与えて頂き, 日頃より御指導, 御助言を賜った村岡洋一教授に心より感謝の意を表します.

また, Overlay Weaver の開発者でありますウタゴエ株式会社 首藤一幸博士に深謝致します.

公私に渡り相談相手となって頂いた村岡研究室の先輩方, 同期の友人, 後輩たちに厚く御礼申し上げます.

最後に, 研究生活を暖かく見守ってくれ, 精神的に支えてくれた家族, 友人に深く感謝致します.

付録

以下の発表で本研究成果を報告することが決定しています.

一杉孝之, 村岡洋一: “DHT におけるデータの更新操作を考慮した検索方式の提案”, 第 19 回 データ工学ワークショップ (DEWS 2008) (Mar. 2008).

この発表原稿を巻末に添えます.

社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

DHTにおけるデータの更新操作を考慮した検索方式の提案

一杉　孝之†　　村岡　洋一†

† 早稲田大学大学院理工学研究科　〒 169–8555 東京都新宿区大久保 3–4–1
E-mail: †hitosugi@muraoka.info.waseda.ac.jp, ††muraoka@waseda.jp

あらまし　近年, 構造化オーバーレイネットワークの研究が盛んに行われている. その中の一つである分散ハッシュテーブル（DHT:Distributed Hash Table）は分散 Map 構造をなす. 従来の DHT ではデータの取得／追加／削除の操作に着目しているが, 更新の操作への配慮に欠けている. 例えば, 更新前のデータと更新後のデータを同じ key に割り当てた場合, 取得者はデータを取得するまでそれらを判別できない. 一方, 異なる key に割り当てた場合には, 元の key では更新後のデータを取得できなくなり, 取得者はなんらかの手段で新しい key についての知識を手に入れる必要がある. 本研究では以下の 3 つの要件を満たす, DHT の新しい検索方式を提案する. 1) 上記の更新操作の問題を解決. 2) 属性検索が可能. 3) ルーティング・アルゴリズムに非依存.

キーワード　P2P ネットワーク, 構造化オーバーレイネットワーク, 分散ハッシュテーブル, 更新操作, 情報検索

Takayuki HITOSUGI† and Yoichi MURAOKA†

† Graduate School of Science and Engineering, Waseda University　Okubo 3–4–1, Shinjuku-ku, Tokyo, 169–8555 Japan
E-mail: †hitosugi@muraoka.info.waseda.ac.jp, ††muraoka@waseda.jp

Abstract　In late years the research of the structured overlay network is often performed. Distributed Hash Table(DHT) that is one of these network models makes distributed map structure. We suggest a new search method of DHT satisfying three following matters in this study. One) It solves a problem of the update operation in DHT. Two) An attribute search is possible. Three) It non-depends on routing algorithm.

Key words　P2P Network, Structured Overlay Network, DHT, Update Operaion, Informaion Retrieval

1. はじめに

今や, 情報の爆発的な増加やコンテンツの多様化などによって, 従来の集中管理型のクライアント・サーバモデルだけでなく自律分散型の P2P ネットワークも利用されるようになっている. P2P ネットワークのように下位ネットワークレイヤの上に重なって別のトポロジを形成するネットワークのことをオーバーレイネットワークという. とくに Gnutella [1] や Winny [2] などのようにオーバーレイネットワークのトポロジに数学的な制約を持たないネットワークは非構造化オーバーレイと呼ばれる. これらは探索を flooding で行うため, ノード数が膨大なネットワークにおいてトラフィックが溢れる可能性があり, さらに探索が成功する保証がない. そこで近年では, 構造化オーバーレイと呼ばれる, トポロジに数学的な制約を持つオーバーレイネットワークが注目されている. 構造化オーバーレイには分散ハッシュテーブル（DHT:Distributed Hash Table）や Skip Graph [3] があげられる. これらはノード数に対してスケーラブルかつ確実な探索を実現している. Skip Graph は, skip list [4] を P2P ネットワークに適応させた構造化オーバーレイであり, 標準で範囲検索が可能である. 一方, DHT は分散 Map 構造を形成する. DHT では, Chord [5], Kademlia [6], Pastry [7], Tapestry [8], CAN [9] などといった様々なルーティングアルゴリズムが考案されており, システム構築時に目的に合わせてトポロジとノードの探索方法の選定をできるという利点がある.

現在は, 分散環境のテストベッドとして利用できる PlanetLab [10] や, オーバーレイネットワークの構築ツールキットである Overlay Weaver [11] [12] が公開されており, 分散ネットワークシステムの構築の敷居が低くなってきている. そのため, 構造化オーバーレイの利用が今後増えていくことが期待される.

本論文におけるデータの更新とは古いデータを残しておいたまま新しいデータを追加することを指す. 従来の DHT の研究ではデータの取得・公開・削除の動作にばかり着目し, 更新の動作への配慮が欠けている. 例えば, 更新前のデータと更新後のデータを同じ key に割り当てた場合, 取得者はデータを取得するまでそれらを判別できない. 一方, 異なる key に割り当てた場合には, 元の key では更新後のデータを取得できないため,

取得者はなんらかの手段で新しい key の知識を手に入れなくてはならない．そのため, 従来の DHT はデータを頻繁に更新するサービスには適していない．したがって, 今後 DHT を実用化していくうえでこの問題が妨げとなることが考えられる．本研究では, これを解決する手法を提案し, 実験による評価および考察を行うことが目的である．この提案手法は以下の 3 つの要件を満たす．

- データの更新時の問題を解決する．
- 属性検索が可能である．
- ルーティング・アルゴリズムに非依存である．

本稿では, まず 2. 節で従来の DHT における更新操作の具体的な問題点を述べ, 3. 節でそれを解決する手法を提案する．そして, 4. 節で提案手法の評価を行い, その後の節で今後の課題と本研究のまとめについて述べる．

2. 従来研究

本節では, 従来の DHT でデータを更新した場合の動作を説明し, その問題点について述べる．更新の動作のことを本論文では update と呼ぶことにする．

説明にあたって, DHT で公開済みの任意のデータを t ($t \in \mathbb{N}$) 回更新することを考える． このデータを含む pair を $\langle key_0, data_0 \rangle$, 更新後のデータを $data_1, data_2, \cdots, data_t$ とおく．

2.1 same keys

更新後の全データに対して同じ key を割り当てるアプローチについて述べる．まず, 一意の key である key_c を用意する．$key_0 = key_c$ かどうかはここでは問わない．$1 \leqq i \leqq t$ ($i \in \mathbb{N}$) となるすべての i に対して $\langle key_c, data_i \rangle$ を put すれば, key_c の知識を持つノードは更新後の全データを取得できる．もちろん, key_0 の知識を持つノードは初期のデータ $data_0$ を取得できる．しかし, 更新後のデータを取得するとき, get 要求先のノードは key_c に対応付けられている複数のデータのうち要求元のノードが欲しているものを判断できないので, 要求元のノードは一度それらすべてをダウンロードしなくてはならない．そのため, 余分なデータも同時にダウンロードすることで無駄なトラフィックが生じてしまう．

2.2 different keys

same keys では key として key_0 と key_c のみを使用したが, 今度は更新後の各データに対して key を新たに用意してみる．これらの key を $key_1, key_2, \cdots, key_t$ とおく．例えば, 更新後のデータ $data_{t+1}$ を追加する場合には key_{t+1} を新たに用意することになる．ここで, $1 \leqq i \leqq t$ ($i \in \mathbb{N}$) となるすべての i に対して $\langle key_i, data_i \rangle$ を put すれば, get 要求で使う key を使い分けることで, 取得時に欲しいデータのみのダウンロードが可能となる．よって, 先に述べた無駄なトラフィックが生じる問題を防げる．しかしながら, この手法では取得元のノードは取得したいデータに対応する key の知識を持つ必要があり, 更新後のデータが増える度に新しく生成する key の知識をどのようにオーバーレイネットワーク上に広めていくかが課題となる．

key	value
key_0	$data_0$
key_c	$data_1$
	$data_2$
	$data_3$
	$\cdots$
	$data_t$

表 1　same keys における DHT の Map の例
Table 1　example of Map in DHT of same keys

key	value
key_0	$data_0$
key_1	$data_1$
key_2	$data_2$
key_3	$data_3$
$\cdots$	$\cdots$
key_t	$data_t$

表 2　different keys における DHT の Map の例
Table 2　example of Map in DHT of different keys

key	value
key_0	$data_0$
key_1	$data_1$
key_2	$data_2$
	$data_3$
$\cdots$	$\cdots$
key_t	$data_t$

表 3　group keys における DHT の Map の例
Table 3　example of Map in DHT of group keys

2.3 group keys

same keys と different keys では, t 個の更新後のデータに対して key を 1 個または t 個用意したが, いくつか用意するという選択肢も存在する．例えば, 更新後のデータを任意の方法でグループ分けしてそのグループごとに新しい key を割り当てるという案が考えられる．しかし, この方法でも結局は上記と同様の問題が発生する．

2.4 問題点のまとめ

従来の DHT でデータを更新する場合には, データの取得時に余分なデータも同時にダウンロードしてしまいトラフィック量が増える, もしくは新しい key の知識をオーバーレイネットワーク上に広めなくてはならない, という問題が生じてしまう．この問題はデータを頻繁に更新する環境で顕著に顕れることが予想される．

3. 提案手法

本節では, 前節で述べた問題点を解決する手法を提案する．

3.1 検索フレーム

本提案手法では検索システムに下記の 3 つのフレームを設ける．

- 各バージョンのデータを扱う version フレーム
- バージョンの管理を行う history フレーム

frame	key	value
version	$hash(data_0)$	$data_0$
	$hash(data_1)$	$data_1$
	$hash(data_2)$	$data_2$
	$\cdots$	$\cdots$
history	$hash(data_0)$	$\{hash(data_0), hash(data_1), \cdots\}$
attribute	$hash(name_0)$	$hash(data_0)$
	$hash(attr_a)$	$hash(data_0)$
	$hash(attr_b)$	$hash(data_0)$
	$\cdots$	$\cdots$

表 4　提案手法における DHT の Map の例
Table 4　example of Map in DHT of proposal method

- 検索の入り口となる attribute フレーム

各ノードはこれらのフレームごとに専用の HashTable を持つ. 例えば, version フレームで pair の put 要求があった場合には, 要求先ノードは version フレーム用の HashTable にこのペアを加える. そして, version フレームで get 要求があった場合には, 要求先ノードは version フレーム用の HashTable から指定された key に対応付けられている value を返す.

以降の節で, 本提案手法における put, update, remove, get を説明する. その際, 従来の DHT における当該動作と区別するために, 従来の DHT の場合には version-, history-, attribute- というフレーム別の接頭辞をつけて表記する. 例えば, history-remove と表記したときには history フレームで従来の DHT における remove を実行することを意味する. また, $hash()$ は引数を ID 空間に写像するハッシュ関数, {} は中の要素を区別して取り出せる集合とする.

3.2 put

データを公開する手順について述べる. ある初期データ $data_0$ を DHT で公開することを考える. $data_0$ の name を $name_0$ とおく.

はじめに $\langle hash(data_0), data_0\rangle$ を version-put する. 次に $\langle hash(data_0), \{hash(data_0)\}\rangle$ を history-put する. 最後に $\langle hash(name_0), hash(data_0)\rangle$ を attribute-put する. これでデータの公開は完了である.

この動作が本提案手法における put である. history-put した {} 区切りの集合を以降, HistoryInfo と呼ぶことにする. 公開した pair の使用方法についてはおいおい説明していく.

3.3 update

データを更新する手順について述べる. 初期データ $data_0$ をデータ $data_c$ で更新することを考える.

まず, $\langle hash(data_c), data_c\rangle$ を version-put する. 次に, put 時に history-put した $\langle hash(data_0), \{hash(data_0)\}\rangle$ の value を $\{hash(data_0), hash(data_c)\}$ に変更する. データの更新操作はここまでである.

この動作が本提案手法における update である. さらに $data_0$ の更新を続ける場合には, 同様の処理を繰り返せばよい. このとき, history フレームでは $\{hash(data_0), hash(data_1), hash(data_2), \cdots\}$ のように HistoryInfo の要素を増やしていく.

3.4 remove

データを削除する手順について述べる. 初期バージョンが $data_0$ であるデータ $data_c$ を削除することを考える.

まず, history-put してある $\langle hash(data_0), \{hash(data_0), \cdots, hash(data_c), \cdots\}\rangle$ の HistoryInfo の要素から $hash(data_c)$ を削除する. 次に $\langle hash(data_c), data_c\rangle$ を version-remove する. これでデータの削除操作は完了である.

ただし HistoryInfo が空, つまり {} になるときには, history フレームの $\langle hash(data_0), \{\}\rangle$ および attribute フレームの $\langle hash(name_0), hash(data_0)\rangle$ を history/attribute-remove するか, その削除を DHT の TTL（pair の有効期間）機構による自然消滅に委ねる.

この動作が本提案手法における remove である.

3.5 get

データを取得する手順について述べる. 初期バージョンが $data_0$ であるデータ $data_c$ を取得することを考える.

まず, $hash(name_0)$ を key として $hash(data_0)$ を attribute-get する. 次に, $hash(data_0)$ を key として $\{hash(data_0), \cdots, hash(data_c), \cdots\}$ を history-get する. あとはこの HistoryInfo から $hash(data_c)$ を取り出して, これを key とすれば目的の $data_c$ を version-get できる.

この動作が本提案手法における get である.

3.6 属性検索

属性検索とは name 以外の情報で検索することである. 例えば, データの作者・公開者・作成日時・公開日時などによる検索が挙げられる.

本提案手法では, データの name の代わりに属性を ID 空間へ写像しても, 同様に検索を行うことができる. 例えば, 属性 $attr$ で検索を行えるようにするには, put するときに $hash(name)$ の代わりに $hash(attr)$ で attribute-put しておけばよい. get するときには $attr$ の知識があるノードは, $hash(name)$ ではなく $hash(attr)$ で attribute-get すれば目的のデータまで辿り着ける. つまり, 本研究手法においては ID 空間に写像できさえすれば name も属性も違いはない. 別の言い方をすれば, 本研究手法では name も 1 つの属性として考えられる. ゆえに, 検索の入り口となるフレームを attribute フレームと名付けたのである.

3.7 分散特性

本提案手法で初期データおよび更新後のデータを公開するとき, value の $data$ に対応する key はすべて $hash(data)$ である. このハッシュ関数が異なる引数を十分な確率で異なる値に写像するならば, key と value は十分な確率で一対一対応となるので, $hash(data_c)$ を key とした get 要求に $data_c$ 以外のデータがヒットする事態は, 十分な確率で防げる. これにより, 2. 節で述べた, 従来の DHT でデータを更新するときの問題点を解決できた. また, $data_c$ に対して $hash(data_c)$ が ID 空間に十分に散らばるならば, データの追加先ノードを散らばらせるという DHT の分散特性を保つ.

```
<history>
  <version>
    <id>xxxxxxxxxxxxxxx</id>
    <id>xxxxxxxxxxxxxxx</id>
  </version>
</history>
```

表 5　HistoryInfo の例
Table 5　example of HistoryInfo

3.8 HistoryInfo

HistoryInfo の形式は本研究では限定しないが表 5 の XML フォーマットを例示しておく.

get において, $hash(data_0) = hash(data_c)$ ならば, $data_c$ に関する history-get 先ノードと version-get 先ノードは一致するので, ノード探索を 1 回短縮できる. $\langle hash(data_0), \{hash(data_0), hash(data_1), hash(data_2), \cdots\}\rangle$ を history-put していることにこれは起因する. ここで, この pair の key に使われるデータ（上の例では $data_0$）を primary version と呼ぶことにする. もし, update などのときに primary version を $data_0$ から $data_1$ に変更して, $\langle hash(data_1), \{hash(data_0), hash(data_1), hash(data_2), \cdots\}\rangle$ を history-put し直したならば, $hash(data_1) = hash(data_c)$ のときにノード探索を 1 回短縮することができるようになる. したがって, あらかじめ get 要求の多いデータがわかるならば, 全体の探索回数を減らせる可能性がある. ただしこの方法を使う場合には, 先の例で言えば attribute-put する pair も合わせて $\langle key_c, hash(data_0)\rangle$ から $\langle key_c, hash(data_1)\rangle$ へと変更する必要がある.

また, HistoryInfo を閲覧したうえで取得するデータのバージョンを決定するという環境では, HistoryInfo に各バージョンの要約や Bloom Filter [13] を用いた縮約などを含めておくことで, バージョン決定の補助とすることができる.

3.9 提案手法のまとめ

2. 節で, 従来の DHT でデータを更新した場合には次のどちらかもしくは両方の問題点があることを述べた.

- データ取得時に目的以外のデータも同時にダウンロードしてしまうため, 無駄なトラフィックが生じる.
- 各バージョンのデータに対応する key の知識をどのようにネットワーク上に広めていくかが課題となる.

これらの問題はデータを頻繁に更新する環境で顕著に顕れることが予想される.

その解決策として本研究では DHT の新しい検索方式を提案した. この手法は name での検索と同様に属性検索も可能である. また, 提案手法はルーティングアルゴリズムに依存していないので, システム構築時に目的に合わせてトポロジとノードの探索方法の選定をできるという DHT の利点を維持している.

4. 評　　価

本節では, 評価実験と, そこから得られる考察について述べる.

本研究の提案手法の優位性は 3.9 節で述べたとおりである.

```
<?xml version="1.0" encoding="UTF-8"?>
<history>
  <primary>
    <id>xxxxxxxxxxxxxxx</id>
  </primary>
  <version>
    <id>xxxxxxxxxxxxxxx</id>
    <id>xxxxxxxxxxxxxxx</id>
  </version>
</history>
```

表 6　本実験における HistoryInfo の形式
Table 6　format of HistoryInfo in this experience

このうち, データ取得時に目的以外のデータも同時にダウンロードしなくなることにより無駄なトラフィック量を軽減できる, という利点を実験によって確かめる. 合わせて, 提案手法によってオーバーレイ・ネットワーク上のトラフィック量がどのように増減するのかを調べる.

本実験は, 1 台のマシンで複数のノードをエミュレートして DHT ネットワークを形成して行う. エミュレーション環境には Overlay Weaver [11] [12] を利用した.

4.1 データの設定

本実験では DHT の value に, 実際のファイルではなくそれをシミュレートしたデータを使用する. このデータは, どのフレームの value であるかを表す情報や, 識別子などを記録している. 実験ではこれを元にデータサイズを計測する. 本小節では, 本実験における各フレームの value のサイズについて述べる.

attribute フレームの value, つまり ID 形式の value のサイズはこの ID を 16 進出力したときの容量とする. したがって, ID 空間が 160 bit のときにはこの value のサイズは 40 byte になる.

history フレームの value, すなわち HistoryInfo のサイズは表 6 に示す XML フォーマットに従うファイルの容量とする. primary タグは 3.8 節で述べた primary version を ID 空間にマップしたハッシュ値を示している. ただし, 本実験では primary version の変更は行わないので primary version は必ず初期データとなる.

version フレームの value, つまりバージョンごとの共有データのサイズは起動時のパラメータで与える.

4.2 計 測 対 象

実験では以下の 4 つを計測する. すべて DHT 上でのみ計測し, 下位ネットワークは考慮しない. また, タグが Overlay Weaver の標準実装で計測対象となっているメッセージのみを本実験での計測対象とする.

- アップロードサイズ
- ダウンロードサイズ
- メッセージサイズ
- メッセージ数

アップロードサイズはアップロードした value の通信量を示す. Overlay Weaver の DHT 実装では put, remove メソッド

key	value
$hash(name)$	$data$
$hash(attr_a)$	$data$
$hash(attr_b)$	$data$
...	...

表 7　従来手法 ① における DHT の Map の例
Table 7　example of Map in DHT of existing method ①

key	value
$hash(name)$	$data$
$hash(attr_a)$	$hash(name)$
$hash(attr_b)$	$hash(name)$
...	...

表 8　従来手法 ② における DHT の Map の例
Table 8　example of Map in DHT of existing method ②

でその引数の key にそれまで割り当てられていた value をダウンロードしているが, 現実のアプリケーションでは必ずしもそうする必要はなく, その value の用途がなければ通信コストの無駄となるだけである. 今回の実験では, put, remove メソッドでその成否のみを応答として受け取る環境を想定し, そのときにダウンロードする value のサイズは計測対象としない. ただし, value 以外にメッセージに含まれるシグネチャやタグなどのヘッダについては, 成否を示す応答としてメッセージサイズの項目で計測する.

ダウンロードサイズはダウンロードした value の通信量を示す.

メッセージサイズはアップデートサイズとダウンロードサイズを除いた通信量を示す. つまり, value を除いた通信量を示すことになる.

メッセージ数は通信回数を示す.

4.3　比 較 対 象

本実験では 3 つのアルゴリズムを比較する.

比較対象の 1 つは本研究の提案手法である.

次の 1 つは従来の DHT である. これを従来手法 ① と呼ぶことにする. データの更新は, 2.1 節で述べた same keys で行う. 本実験では, 更新データ用の key に初期データの key を使用することとした. same keys を選んだ理由は, 2.2 節の different keys, 2.3 節の group keys では, key の知識をネットワーク上に広めることが課題となり, それを解決しない限り検索が成立しないためである.

最後の 1 つも従来の DHT である. これを従来手法 ② と呼ぶことにする. データの更新も同様に same keys で行う. 従来手法 ① との違いは次節で述べる.

従来手法 ① と ② の違いは属性検索の方法である. まず, どちらの手法でもデータを公開するノードは $\langle hash(name), data\rangle$ を put する. これで通常通り, name からデータの取得が可能になる. ここで, 属性を ID 空間にマップした値を $hash(attr)$ とおく. 次に, 従来手法 ① では, データの公開元ノードは $\langle hash(attr), data\rangle$ を put する. これにより, attr からもデータの取得が可能になる. それに対して, 従来手法 ② では, データの公開元ノードは $\langle hash(attr), hash(name)\rangle$ を put する. この場合には, attr から hash(name) を取得した後, これを key とすることでデータの取得が可能になる. 従来手法 ② ではノード探索が 1 回増えることになるが, 従来手法 ① にはデータを更新する度に属性の数だけそのデータを put しなおすという欠点がある.

4.4　実験シナリオ

初期ノード数を 10000 とする. ノードは途中参加／離脱をしない. 要するにノード数は常に 10000 である.

get を 3000 個, put, remove, update を 1000 個ずつの計 6000 個のコマンドを用意する. コマンドは一様分布に従いランダムにシャッフルして並び替えた. ただし, put コマンドを先に実行しなければ DHT 上に共有データが存在しないためにその他のコマンドを実行できないので, 例外的に 500 個の put コマンドを先頭に並べる.

ID 空間は 160 bit, ルーティングアルゴリズムは Kademlia [6], メッセージングサービスは Iterative である. 耐 churn 手法 [14] は導入していない.

属性数とデータのサイズを変更したときの各比較対象の様子を計測・考察する.

4.5　実 験 結 果

本節では実験結果を示す. この実験結果において属性数は name を含めて数えている. グラフの凡例は上から順に提案手法, 従来手法 ①, 従来手法 ② である.

図 1, 2, 3 は順に, データサイズを変更したときのアップロードサイズ, ダウンロードサイズ, その合計を示している. 属性数は 3 個である.

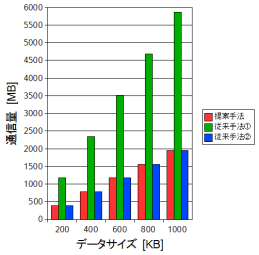

図 1　データサイズを変更したときのアップロードサイズ
Fig. 1　upload size about changing data size

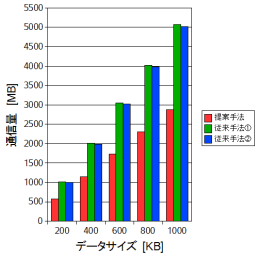

図 2　データサイズを変更したときのダウンロードサイズ
Fig. 2　download size about changing data size

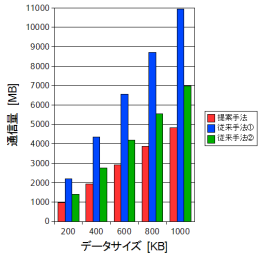

図 3　データサイズを変更したときのアップ／ダウンロードサイズ
Fig. 3　up/download size about changing data size

図 4, 5, 6 は順に, 属性数を変更したときのアップロードサイズ, ダウンロードサイズ, その合計を示している. データサイズは 500 KB である.

図 7, 8 は順に, 属性数を変更したときメッセージ数, メッセージサイズを示している. 図 8 の通信量に図 6 の通信量を加算したものが図 9 である.

メッセージ数とメッセージサイズの計測ではデータサイズは含まれないので, データサイズを変更させたときのメッセージ数とメッセージサイズは計測していない.

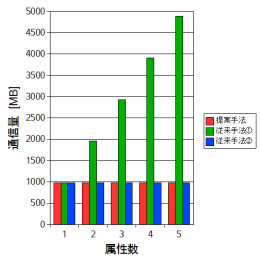

図 4　属性数を変更したときのアップロードサイズ
Fig. 4　upload size about changing attribute num

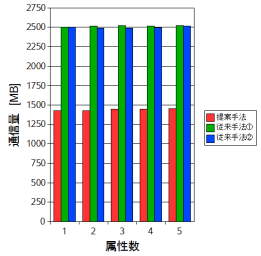

図 5　属性数を変更したときのダウンロードサイズ
Fig. 5　download size about changing attribute num

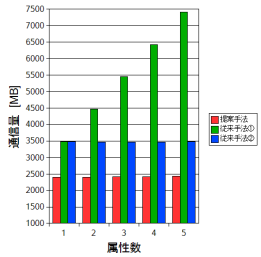

図 6　属性数を変更したときのアップ／ダウンロードサイズ
Fig. 6　up/download size about changing attribute num

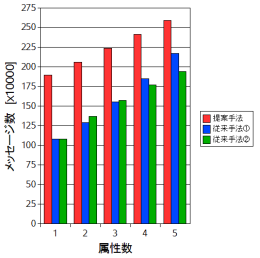

図 7　属性数を変更したときのメッセージ数
Fig. 7　message num about changing attribute num

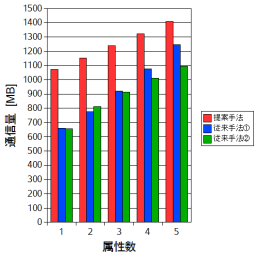

図 8　属性数を変更したときのメッセージサイズ
Fig. 8　message size about changing attribute num

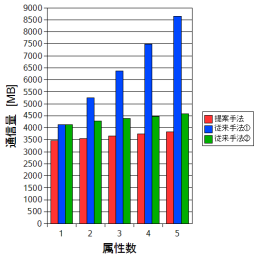

図 9　属性数を変更したときの全通信量
Fig. 9　all communication size about changing attribute num

4.6　考　　察

本節では実験結果についての考察を行う.

まず図 1, 2, 3 の結果について考察する. 従来手法 ① では更新の都度すべての属性に対して更新後のデータを put するので, データサイズが大きくなるにつれてアップロードサイズが増大している. 提案手法ではデータ取得時に目的以外のデータも同時にダウンロードすることがないので, ダウンロードサイズがどちらの従来手法よりも小さくなっている. さらに, データサイズが大きくなるにつれ, 余分なデータをダウンロードするか否かの違いが顕著になり, ダウンロードサイズの差が広がっていく. このダウンロードサイズの差により, アップ／ダウンロードの合計サイズは, データサイズが大きいほど提案手法のほうが従来手法よりも小さくなった. 以上の結果から提案手法の利点を確認できたことになる.

次に図 4, 5, 6 の結果について考察する. 先ほどと同様に, 従来手法 ① では属性数が増えるにつれてアップロードサイズが増大している. 同じく, 提案手法ではダウンロードサイズがどちらの従来手法よりも小さくなっている. これにより, アップ／ダウンロードの合計サイズは, 提案手法のほうが従来手法よりも小さくなった. ただし, 提案と従来手法 ② の差は属性数を

増加させても開いていかない．これらの結果からも提案手法の利点を確認できたことになる．

最後に図 7, 8, 9 の結果について考察する．ここでも同様に, 従来手法 ① では属性数が増えるにつれてメッセージ数とメッセージサイズが増大している．そして, 提案手法でもこれらが増大していると同時に, 提案手法では従来手法と比較して高い数値となっている．この原因は検索フレームを導入したことでノード探索が増えたためと思われる．メッセージサイズにアップ／ダウンロードの合計サイズを加えた結果においては, 提案手法が従来手法より少ない通信量を維持している．これは, アップ／ダウンロードの合計サイズの差がメッセージサイズの差よりも大きいためである．したがって, 本実験のようにデータサイズがメッセージ一つあたりのサイズに対して十分大きい環境でなければ, 提案手法は従来手法と比べて通信量が多くなってしまう可能性がある．ゆえに, 本研究手法はデータサイズが十分に大きい環境で真価を発揮する手法であると言える．

実験結果についての考察をまとめる．データ取得時に目的以外のデータも同時にダウンロードしなくなることにより無駄なトラフィック量を軽減できる, という提案手法の利点を確認できた．しかしその一方, 提案手法ではメッセージ数とメッセージサイズが増大するという弱点も示された．この理由として, 検索フレームの導入でノード探索が増えたことが考えられる．本実験のようにデータサイズがメッセージ一つあたりのサイズに対して十分大きい環境でなければ, 提案手法は従来手法と比べて通信量が多くなってしまうかもしれない．つまり, 本研究手法はデータサイズが十分に大きい環境において利用価値がある．

5. 今後の課題

関根らの研究 [15] も構造化オーバーレイでのデータの更新を扱っている．[15] ではデータとして, リアルタイムで変動するセンサデータを想定している．そして, DHT に PHT（Prefix Hash Tree）[16] [17] を適用することで範囲検索に対応させた構造化オーバーレイにおいて, 負荷の分散やメッセージ数／サイズの削減などを目指している．本研究とは想定している環境が異なるが, データの更新時にノード探索の履歴を利用してメッセージ数を削減するといった, 本研究の提案手法と併用できる手法も考えられている．とくに, 本研究の提案手法では 4.6 節で述べたようにメッセージ数／サイズがネックとなっているので, 今後このような手法の導入を検討していく必要がある．

また, 松田の研究 [18] では複数属性の範囲を指定して範囲検索を行える構造化オーバーレイを提案している．本研究の提案手法も PHT を適用すれば範囲検索を行えるようになる．しかし, [18] のように同時に複数属性を指定することはできない．したがって, このようなより柔軟な検索に対応するためには本提案手法をさらに改良していく必要がある．

6. 本研究のまとめ

本論文では, データの更新を考慮した, DHT の新しい検索方式の提案を行った．

従来の DHT でデータを更新した場合には次のどちらかもしくは両方の問題点が存在する．

- データ取得時に目的以外のデータも同時にダウンロードしてしまうため, 無駄なトラフィックが生じる．
- 各バージョンのデータに対応する key の知識をどのようにネットワーク上に広めていくかが課題となる．

本研究の提案手法は以下の 3 つの要件を満たす．

- 上記のデータの更新に関する問題を解決する．
- 属性検索が可能である．
- ルーティング・アルゴリズムに非依存である．

実験結果から, データ取得時に目的以外のデータも同時にダウンロードしなくなることにより無駄なトラフィック量を軽減できる, という本提案手法の利点を確認できた．しかしその一方, 提案手法ではメッセージ数とメッセージサイズが増大するという弱点も実験結果で示された．この理由として, 提案手法においてノード探索が増えたことが考えられる．データサイズがメッセージ一つあたりのサイズに対して十分大きい環境でなければ, 提案手法は従来手法と比べて通信量が多くなってしまう可能性も考えられる．したがって, 本研究手法はデータサイズが十分に大きい環境において高い効果を得られる手法である．

文　献

[1] “Gnutella.com”, http://www.gnutella.com/.

[2] “Winny - Wikipedia”, http://ja.wikipedia.org/wiki/Winny.

[3] J. Aspnes and G. Shah: “Skip graphs”, In Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 384-393 (Jan. 2003).

[4] W. Pugh: “Skip lists: a probabilistic alternative to balanced trees”, Communications of the ACM 33, pp. 668-676, (Jun. 1990).

[5] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan: “Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications”, In Proc. of ACM SIGCOMM 2001, pp. 149-160 (Aug. 2001).

[6] P. Maymounkov and D. Mazieres: “Kademlia: A Peer-to-peer Information System Based on the XOR Metric”, In Proc. of IPTPS 2002, pp. 53-65 (Mar. 2002).

[7] A. Rowstron and P. Druschel: “Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems”, In Proc. of IFIP/ACM Middleware 2001, pp. 329-350 (Nov. 2001).

[8] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph: “Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing”, U. C. Berkeley Technical Report, UCB/CSD-01-1141 (Apr. 2001).

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker: “A Scalable Content-Addressable Network”, In Proc. of ACM SIGCOMM 2001, pp. 161-172 (Aug. 2001).

[10] “PlanetLab — An open platform for developing, deploying, and accessing planetary-scale services”, http://www.planet-lab.org/.

[11] 首藤一幸, 田中良夫, 関口智嗣: “オーバレイ構築ツールキット Overlay Weaver”, 先進的計算基盤システムシンポジウム (SACSIS 2006), pp.183-191 (May 2006).

[12] “Overlay Weaver: An Overlay Construction Toolkit”, http://overlayweaver.sourceforge.net/index-j.html.

[13] B. Bloom: “Space/Time Trade-Offs in Hash Coding with Allowable Errors”, Communications of ACM, vol. 13, no. 7, pp. 422-426 (Jul. 1970).

[14] 首藤一幸: “下位アルゴリズム中立な DHT 実装への耐 churn 手法の実装”, 第 19 回 コンピュータシステム・シンポジウム

(ComSys 2007), pp. 191-198 (Nov. 2007).
[15] 関根理敏, 瀬崎薫: “P2P ネットワークにおける高頻度なデータ更新を考慮した適応的コンテンツ配置法”, 電子情報通信学会技術研究報告 情報ネットワーク (IN 2006), pp. 197-202 (Mar. 2007).
[16] S.Ramabhadran, S.Ratnasamy, J.M.Hellerstein, and S.Shenker: “Prefix Hash Tree: An Indexing Data Structure over Distributed Hash Tables”, IRB Technical Report (Feb. 2004).
[17] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, J. Hellerstein, and S. Shenker: “A Case Study in Building Layered DHT Applications”, In Proc. of ACM SIGCOMM 2005, pp. 97-108 (Aug. 2005).
[18] 松田哲史: “複数属性を持つデータの属性値範囲指定検索用分散ネットワーク構成方式の提案”, 情報処理学会研究報告 分散システム/インターネット運用技術 (DSM-44), pp. 19-23 (Mar. 2007).