情報処理学会論文誌 Vol.53 No.12 2752–2761 (Dec. 2012)

推薦論文

Chord# における経路表の維持管理コスト削減手法の提案とその評価

呉 承彦[1,a)] 安倍 広多[1] 石橋 勇人[1] 松浦 敏雄[1]

受付日 2012年6月16日, 採録日 2012年9月10日

概要：Chord# は範囲検索が可能な構造化 P2P（Peer-to-Peer）ネットワークの一種である．Chord# ではショートカットリンク（finger table）を用いることでノード数 n に対して，$O(\log n)$ ホップで検索が可能である．Chord# の finger table は，ノードの挿入や削除，障害に対応するために定期的に更新する必要があるが，本稿ではこの更新処理のコストを削減する方式を提案する．提案手法では，finger table を 2 次元配列に拡張したうえで，隣接するノードの finger table が類似していることを利用して更新処理に必要なメッセージ数を数分の 1 に削減する．さらに，リモートノード離脱時の finger table 更新処理の高速化，およびネットワーク近接性を利用したルーティングも実現した．提案方式の有効性はシミュレーションにより確認した．

キーワード：構造化 P2P ネットワーク，Chord#，経路表維持管理，ネットワーク近接性

A Method for Reducing Maintenance Cost of Routing Table in Chord# and Its Evaluation

Seung eon Oh[1,a)] Kota Abe[1] Hayato Ishibashi[1] Toshio Matsuura[1]

Received: June 16, 2012, Accepted: September 10, 2012

Abstract: Chord# is a kind of structured Peer-to-Peer (P2P) network that supports range queries. Chord# achieves $O(\log n)$ search hops, where n denotes the number of nodes, by using a finger table, a collection of short cut links. A finger table of Chord# must be periodically updated to catch up node insertion, deletion and failure. In this paper, we propose a method to reduce the cost of updating finger tables. Finger tables are extended to two-dimensional arrays and the cost of updating finger tables is reduced by using the similarity of finger tables of adjacent nodes. The method avoids delay in updating finger tables caused by node leaving and also supports proximity routing. The effectiveness of the proposed method is experimentally confirmed by computer simulations.

Keywords: structured P2P network, Chord#, routing table maintenance, proximity routing

1. はじめに

ネットワークによって接続された多数のノードで大容量の情報を分散処理するための技術として，P2P（Peer-to-Peer）技術が注目されている．P2P ネットワークは各ノードが自律的に他のノードと協調して動作し，耐故障性やスケーラビリティ，負荷分散などに優れている．

P2P ネットワークの分野では，分散ハッシュテーブル（DHT）に基づくシステムがよく研究されている．DHT は key と value のペアを P2P ネットワークの多数のノードで分散管理する技術である．代表的な DHT には Chord [2]，Pastry [3]，Tapestry [4] などがある．DHT では key をハッシュすることによってデータを配置するノードを決定する

1 大阪市立大学大学院創造都市研究科
Graduate School for Creative Cities, Osaka City University, Osaka 558–8585, Japan
a) oh@sousei.gscc.osaka-cu.ac.jp

本稿の内容は 2011 年 10 月のマルチメディア通信と分散処理ワークショップにて報告され，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．

© 2012 Information Processing Society of Japan

ため，特定の key に対する検索は可能であるが，指定した範囲の key を探す範囲検索などが困難である．

これに対し，範囲検索が可能な構造化 P2P ネットワークの 1 つに Chord# [5] がある．Chord# は Chord と同様のリングベースの構造化 P2P ネットワークであるが，Chord とは異なりノードが key の値の昇順に並べられるため，範囲検索を容易に実現できる．Chord# の各ノードは，効率的な検索を行うために経路表に Chord と類似した finger table を持つ．これを用いることで，Chord# はノード数を n とするとき，任意のノードを $O(\log n)$ ホップで検索できる．

範囲検索が可能な構造化 P2P ネットワークとしては他に Skip graph [6] がある．Chord# は Skip graph に比べて平均経路長が短く，最大経路長がおさえられるという利点がある一方，ノードの挿入や削除，障害に対応するために finger table を定期的に更新する必要があり，維持管理にコストがかかるという欠点がある[*1]．

本稿では Chord# の finger table 更新処理のコストを削減する手法を提案する．この手法を適用した構造化 P2P ネットワークを **Chord##** と呼ぶ．Chord や Chord# では 1 次元配列であった finger table を Chord## では 2 次元配列に拡張し，隣接するノードの finger table が類似していることを利用して更新処理のコストを削減する．また，Chord## では 2 次元 finger table を利用することで，ネットワーク近接性を考慮したルーティングおよびリモートノード離脱時の finger table 更新処理の高速化も実現する．

本稿の構成は次のとおりである．まず 2 章で Chord# について述べる．3 章では Chord## の詳細について述べ，4 章では Chord## の評価と考察を行う．5 章で関連研究について触れ，最後に 6 章でまとめと今後の課題を述べる．

2. Chord#

本章では，本稿における提案の基礎となる Chord# について述べる．

2.1 概要

Chord# はリングベースの構造化 P2P ネットワークである．各ノードは key を保持しており，リング状の key 空間にマップされる．リングは，時計回りに key の値が大きくなるものとする．Chord とは異なり，リングの key 空間の大きさ（最小値，最大値）をあらかじめ決めておく必要はない．

Chord# の各ノードはいくつかの変数を保持する．以後，ノード u が保持する変数 x を $u.x$ のように表記する．また，Chord# の（あるいは後で述べる Chord## の）全ノードの集合を $V = \{N_0, N_1, \cdots, N_{n-1}\}$ とし（ただし n はノード数），各ノードは時計回りに $N_0, N_1, \cdots, N_{n-1}$ の順に並べられるものとする（$N_i.key < N_{i+1}.key$）．さらに，$a \oplus b \equiv (a+b) \bmod n$ とする．

Chord# の各ノードは key 以外に，successor，predecessor，finger table，および，長さ r の successor list を持つ．successor は key からリング状で時計回りに最も近いノードへのポインタ，predecessor は反時計回りに最も近いノードへのポインタである．すなわち，ノード N_i の successor は $N_{i\oplus 1}$，predecessor は $N_{i\oplus(n-1)}$ と表すことができる．なお，ここでのポインタはノードのロケータ（IP アドレスなど）とノードの key の両方を含むものとする．finger table に関しては次の節で述べる．successor list は，時計回りに最も近い r 個のノードへのポインタであり，successor ノードが離脱または故障した場合に，successor を付け替えてリングを修復するために用いる．ノード N_i の successor list は $N_{i\oplus 1}, N_{i\oplus 2} \ldots, N_{i\oplus r}$ である（最初の要素は successor と等しいことに注意）．

2.2 finger table

Chord# では，検索効率を向上させるために finger table を持つ．finger table はノード数 n に対して，$\lceil \log_2 n \rceil$ 個のエントリを持つ 1 次元配列である．finger table の各エントリは式 (1) で決定される．

$$finger[i] = \begin{cases} successor & (i = 0) \\ finger[i-1].finger[i-1] & (i > 0) \end{cases} \quad (1)$$

例として，ノード N_0 の finger table を考える．N_0.finger[0] には，N_0 の successor すなわち N_1 へのポインタが代入される．N_0.finger[1] には，N_0.finger[0] が指すノードである N_1 が保持する finger[0]，すなわち，N_2 へのポインタが代入される．同様に，N_0.finger[2] には，N_0.finger[1] が指すノードである N_2 が保持する finger[1]，すなわち，N_4 へのポインタが代入される．

このように，ノード N_k の finger[i] には，$N_{k\oplus 2^i}$ へのポインタが代入されるため，finger table の高さは $\lceil \log_2 n \rceil$ となる．この finger table を用いると，検索クエリを他のノードへ転送するごとに検索範囲を 1/2 ずつ絞り込めるため，検索に要するホップ数は $O(\log_2 n)$ となる．

各ノードの finger table はノードの挿入や削除，障害などによって理想的な状態（式 (1) が満たされた状態）ではなくなるため，各ノードは finger table を定期的に更新する必要がある．上で述べたように，ノード u の finger[i]（$i > 0$）を更新するためには，u は自身の finger[$i-1$] が指すノードの finger[$i-1$] を取得する必要がある．1 つのノードが持つ finger table のすべての行を更新するために

[*1] Skip graph は複数の Skip list [7] から構成される分散データ構造である．Skip list は乱数を用いたデータ構造で，検索処理の時間計算量は平衡木と同様 $O(\log n)$ であるが，挿入・削除の際にバランス回復操作が不要という特徴がある．Skip graph は Skip list をベースとしているため，特に経路表を定期的に更新しなくても検索効率が落ちない．

© 2012 Information Processing Society of Japan

必要なメッセージ数は，$2\lceil\log_2 n\rceil$（反復ルーティングで取得した場合），または $\lceil\log_2 n\rceil+1$（再帰ルーティングで取得した場合）となる．

3. 提案手法

本章では，提案手法である Chord$^{\#\#}$ の詳細を述べる．Chord$^{\#\#}$ の主な目標は，finger table の更新処理のコストを削減することにある．これを実現するために，Chord$^{\#\#}$ ではノード u が他のノードから finger table のエントリを取得する際，同時に successor list を取得する．また，finger table を 2 次元化し，取得した successor list を横方向に追加する．これにより，u と u の近隣ノード（u の successor list に含まれるノード）の finger table の間に共通部分ができることを利用し，u から近隣ノードへ共通部分を送信することで，更新に必要なメッセージ数を削減する．

以下，Chord$^{\#\#}$ が Chord$^{\#}$ と異なる点に焦点を絞って詳細を述べる．

3.1 データ構造

Chord$^{\#\#}$ の finger table は図 **1** に示すように Chord$^{\#}$ の finger table を横方向に拡張した 2 次元配列である（fingers[][] と表記する）．ここで，ノード N_u の finger table の高さを $N_u.f_h$，幅を $N_u.f_w$ とする．$N_u.f_h=\lceil\log_2 n\rceil$ であるが，幅 $N_u.f_w$ はノードによって異なる（詳細は 3.2 節で述べる）．

ノード N_u の fingers[i][0]（$0\le i<N_u.f_h$）は，定常状態（ノードの参加や離脱がない状態）ではノード $N_{u\oplus 2^i}$ へのポインタを保持する．また，fingers[i][j]（$0<j<N_u.f_w$）は，finger[i][0] が指すノードの successor_list[$j-1$] を保持する．

図 1 は Chord$^{\#}$ および Chord$^{\#\#}$ におけるノード N_0 の finger table の例である．Chord$^{\#}$ では N_0 の finger table は N_1，N_2，N_4，N_8 へのポインタのみを保持するのに対し，Chord$^{\#\#}$ の N_0 の finger table ではそれに加え，横方向（太枠内）に N_1，N_2，N_4，N_8 の successor list を格納している．

Chord$^{\#\#}$ の finger table について，次の定理が成り立つ．

定理 1． 定常状態にある *Chord*$^{\#\#}$ 上のノード N_u，$N_{u\oplus k}$ に対し，N_u.fingers[i][j] $=N_{u\oplus k}$.fingers[i][$j-k$]（ただし $k\le j<\min(N_u.f_w, N_{u\oplus k}.f_w+k)$）．

証明． Chord$^{\#\#}$ の finger table の構築方式から，定常状態で N_u.fingers[i][j] $=N_{u\oplus(2^i+j)}$ が成り立つ．このため，$N_{u\oplus k}$.fingers[i][$j-k$] $=N_{u\oplus k\oplus(2^i+j-k)}=N_{u\oplus(k+2^i+j-k)}=N_{u\oplus(2^i+j)}=N_u$.fingers[$i$][$j$]．ただし，finger table の幅の制約から，$(0\le j<N_u.f_w)\wedge(0\le j-k<N_{u\oplus k}.f_w)$ であり，これから前提条件 $k\le j<\min(N_u.f_w, N_{u\oplus k}.f_w+k)$ が導かれる． □

3.2 finger table のメンテナンスアルゴリズム

本節では，Chord$^{\#\#}$ の各ノードが保持する finger table を最新の状態に保つためのメンテナンス手法について述べる．

Chord$^{\#\#}$ のノード N_u が自身の finger table を更新する手順は，2.2 節で述べた Chord$^{\#}$ の finger table 更新手順とほぼ同様である．N_u がノード $x=N_u$.fingers[$i-1$][0] に x.fingers[$i-1$][0] を問い合わせるときに，同時に x の successor list を受け取り，N_u.fingers[$i-1$][j]（$0<j$）に格納する．このように他のノードから情報を収集して finger table を更新する手順を**アクティブな更新**と呼ぶことにする．Chord$^{\#\#}$ におけるアクティブな更新に必要なメッセージ数は Chord$^{\#}$ の場合と同じである（2.2 節参照）．

定理 1 で示したように，（定常状態では）Chord$^{\#\#}$ 上のノード N_u と $N_{u\oplus k}$ の finger table には共通部分がある．このため，N_u がアクティブな更新を行った後，共通部分を $N_{u\oplus k}$ に転送することにすれば，$N_{u\oplus k}$ はアクティブな更新を行う必要がない．このように他のノードから finger table を受信して更新することを**パッシブな更新**と呼ぶ．

アクティブな更新を行ったノード N_u では，finger table の幅 $(N_u.f_w)$ は $r+1$ である．ここで，N_u がパッシブな更新を行うノードに送信する finger table の列の幅を p とする（$p>1$ はネットワーク距離を考慮した検索を行うために用いる．詳細は 3.4 節で述べる）．このとき，N_u が $N_{u\oplus k}$ に送信する finger table は N_u.fingers[$*$][$k:k+p-1$] となる（[$*$] は配列のすべてのインデックス，[$k:k+p-1$] は配列のインデックス k から $k+p-1$ までの範囲の部分配列を表す）．また，N_u が finger table を送信可能なノードは，$N_{u\oplus 1}, N_{u\oplus 2}, \ldots, N_{u\oplus(r+1-p)}$ となる（図 **2** 参照）．以

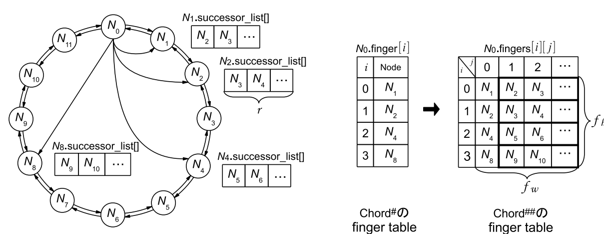

図 1 Chord$^{\#}$ と Chord$^{\#\#}$ の finger table
Fig. 1 Finger tables of Chord$^{\#}$ and Chord$^{\#\#}$.

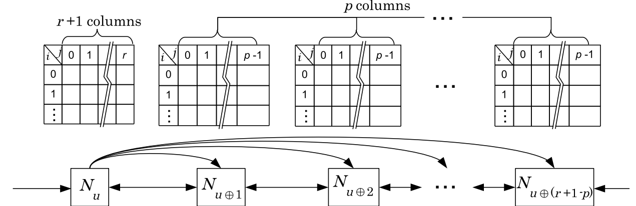

図 2 finger table の送信
Fig. 2 Finger table transfer.

© 2012 Information Processing Society of Japan

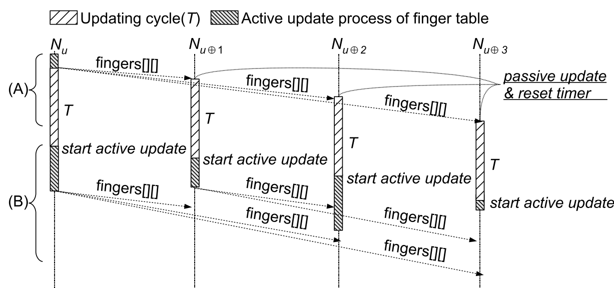

図 3 各ノードが更新周期（T）待つ場合の finger table の更新例
Fig. 3 Finger table update sequence (interval time: T).

下，1 回のアクティブな更新でパッシブな更新を行うことができるノード数を s とする．

$$s = r + 1 - p \tag{2}$$

である．

3.2.1 finger table の更新タイミング

Chord$^{\#\#}$ の各ノードは，一定時間ごとに finger table を更新する必要がある．アクティブな更新を行うノードを自律的に決めるために，各ノードは更新周期を管理するためのタイマを有する．アクティブな更新を行ったノードから finger table を受信した場合はそれを利用してパッシブな更新を行い，受信しないまま更新周期が経過した場合はアクティブな更新を行う．ノード N_u のタイマが満了した場合の動作は次のようになる．

(1) ノード N_u は finger table のアクティブな更新を行い，タイマをリセットする．
(2) ノード $N_{u\oplus k}$（$1 \leq k < s$）に fingers[*][$k : k+p-1$] を送信する．
(3) ノード $N_{u\oplus k}$ は受信した finger table を用いてパッシブな更新を行い，タイマをリセットする．

3.2.2 更新待ち時間

ここで，各ノードの finger table の更新周期について検討する．

まず，単純に各ノードが保持するタイマの設定値をすべて同じ値 T としたときのシーケンス例を**図 3** に示す．ここでは，N_u がアクティブな更新を行い，$N_{u\oplus 1}$ から $N_{u\oplus 3}$ までのノードがパッシブな更新を行っている（図の (A)）．

N_u は更新後，パッシブな更新なしに時間 T が経過すると，N_u は再びアクティブな更新を開始する．しかし，アクティブな更新にはある程度の時間を要するため，N_u からの finger table が到着する前に $N_{u\oplus 1}$ から $N_{u\oplus 3}$ の更新タイマは満了し，アクティブな更新を始めてしまう（図の (B)）．

この状況を改善するため，タイマをリセットする際，システム全体で定められた値 T に，ノードごとに異なる更新待ち時間 $k \times \beta$（$0 \leq k \leq s$）を加えた値をタイマにセットすることにする．すなわち，各ノードの更新周期は $T + k \times \beta$ となる．ここで k の値は，アクティブな更新を行ったノード N_u から finger table を受信したノード $N_{u\oplus j}$（$0 < j < N_u.f_w - p$）では $k = j - 1$ とし，アクティブな更新を行った N_u では，$k = s$ とする．N_u の更新待ち時間を最大値に，$N_{u\oplus 1}$ の更新待ち時間を最小値にするのはアクティブな更新が特定のノードに集中しないようにするためである．β はノードが finger table のアクティブな更新に要する時間の推定値である．なお，Chord$^{\#\#}$ に新たに参加したノードでも $k = s$ とする．

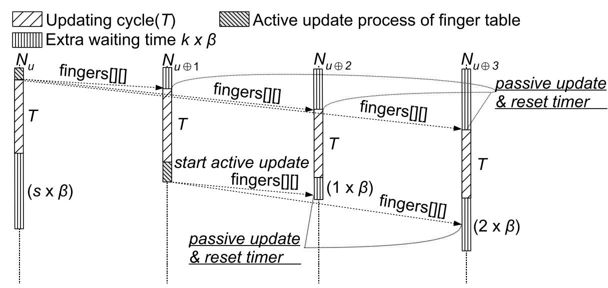

図 4 T に更新待ち時間 $k \times \beta$ を加えた場合の finger table 更新例
Fig. 4 Finger table update sequence (interval time: $T + k \times \beta$).

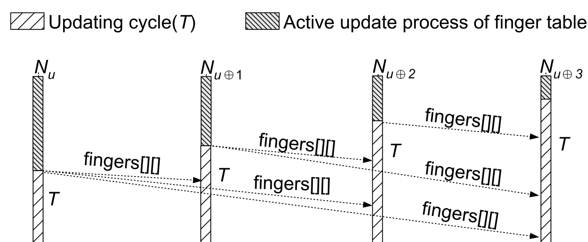

図 5 アクティブな更新のタイミングによる，finger table 更新のオーバラップ例
Fig. 5 Finger table update sequence (update overlap due to timing of active update).

更新待ち時間を追加した場合のシーケンス例を**図 4** に示す．図 3 と比べて無駄なアクティブな更新を行うノードが減少することが分かる．

また，各ノードのアクティブな更新のタイミングによっては，一時的に**図 5** のような状況が生じることがある．図 5 では，$N_{u\oplus 3}$，$N_{u\oplus 2}$，$N_{u\oplus 1}$，N_u の順にアクティブな更新を行っているため，パッシブな更新によるメッセージ数の削減効果が得られない（このような状況を**オーバラップ更新**と呼ぶ）．しかし，この状況は N_u がアクティブな更新を行って，$N_{u\oplus 1}$，$N_{u\oplus 2}$，$N_{u\oplus 3}$ のタイマをリセットすると解消される．

なお，他のノードの参加や離脱などより，N_u が 2 回連続してアクティブな更新を行う場合，2 回目の更新では後続のノードに finger table を送信しない．これは，2 回目の N_u のアクティブな更新よりも $N_{u\oplus 1}$ の方が先にアクティブな更新を行うため，N_u から $N_{u\oplus 1}$ に finger table を送信する意義はほとんどないためである．

タイマの設定値 T は，大きすぎると finger table がネットワーク内のノード集合の変化に追随できないため性能が

© 2012 Information Processing Society of Japan

劣化し，また小さすぎると finger table 更新処理のオーバヘッドが大きくなる．詳細は文献 [8] を参照されたい（T の値について 30 秒から 20 分の間で評価している）．

3.2.3 アルゴリズムの詳細

以下に Chord$^{\#\#}$ の finger table 更新アルゴリズムを疑似コードの形で示す．

```
1  process P
2  var // ノードPが保持する変数
3      P.key // キー
4      P.fingers[P.f_h][P.f_w] // finger table
5      （ただしfingers[0][0] = successorであり，
6        本アルゴリズムとは別に管理されることを想定する）
7      P.successor_list[r] // successor list
8      P.k // 更新待ち時間のパラメータ.初期値は−1
9  // タイマの満了によりアクティブな更新を行う
10 upon timer timeout
11     i = 0
12     while (true) {
13         i = i + 1
14         // P.fingers[i − 1][0]からfingers[i][0]と
15         // fingers[i − 1][1 : r]を取得
16         send ⟨get_fingerOp, P, i⟩ to
               P.fingers[i − 1][0]
17         upon receiving ⟨reply_fingerOp, finger′,
               successors[]⟩
18         for j = 1 to r
19             P.fingers[i − 1][j] = successors[j − 1]
20         if (finger′ ∈ [P.key, P.fingers[i − 1][0]))
21             break
22         P.fingers[i][0] = finger′
23     }
24     // T + (β × s)秒後に
25     // timeout イベントが発生するように設定
26     timer(T + (β × s))
27     // 2 回連続してアクティブな更新をしていなければ，
28     // 更新したfinger table を後続ノードに送信する
29     if (P.k ≠ s)
30         for k = 1 to s
31             send ⟨updateOp, P.fingers[∗][k : k + p],
                   k − 1⟩ to P.successor_list[k − 1]
32     P.k = s
33 // updateOp メッセージの受信によりパッシブな更新を行う
34 upon receiving ⟨updateOp, fingers′[][], seq⟩:
35     P.fingers[∗][∗] = fingers′[∗][∗]（ただし，successor が
36     格納されているP.fingers[0][0]は除く）
37     P.k = seq
38     timer(t + (β × P.k))
39 // P′から get_fingerOp メッセージを受信した場合，
40 // reply_fingerOP メッセージをP′に送信する
41 upon receiving ⟨get_fingerOp, P′, index⟩:
42     send ⟨reply_fingerOp, P.fingers[index − 1][0],
           P.successor_list[∗]⟩ to P′
```

ノードが保持する finger table は一部の要素が重複している（たとえば $fingers[0][1] = fingers[1][0]$）．擬似コードでは明示していないが，ネットワーク帯域を節約するため，finger table を送信する際は（擬似コード 42 行目），重複する要素は省いて送信し，また受信の際は（35 行目）省かれた要素を補ってコピーするものとする．なお，後の 4.5 節ではこの前提でメッセージのバイト数を計算している．

3.2.4 アルゴリズムの解析

Chord$^{\#}$ ではすべてのノードが finger table をアクティブに更新するが，Chord$^{\#\#}$ ではあるノード u が finger table のアクティブな更新を行うと，u に引き続く s 個のノードはパッシブな更新で済ませることができる．このため，理想的にはアクティブな更新を行うノード数は $\lceil n/(s+1) \rceil$ となる．

Chord$^{\#\#}$ において，u がアクティブな更新を行うために必要なメッセージ数は，2.2 節で述べた Chord$^{\#}$ の場合と同じく $2\lceil \log_2 n \rceil$（反復ルーティングの場合），または $\lceil \log_2 n \rceil + 1$（再帰ルーティングの場合）である．また，$s$ 個のノードがパッシブな更新を行うために必要なメッセージ数は s である．そのため，ノード u と，u に引き続く s 個の合計 $s+1$ ノードが finger table を更新するために必要なメッセージ数の合計は $2\lceil \log_2 n \rceil + s$（または $\lceil \log_2 n \rceil + s + 1$）となる．これは，Chord$^{\#}$ において $s+1$ 個のノードが finger table を更新するために必要なすべてのメッセージ数 $(s+1) \times 2\lceil \log_2 n \rceil$（または $(s+1) \times (\lceil \log_2 n \rceil + 1)$）と比較するとほぼ $1/(s+1)$ である．

表 1 に，Chord$^{\#}$ と Chord$^{\#\#}$ における，アクティブな更新を行うノード数と，1 ノードが 1 回の更新に必要とする平均メッセージ数を示す．なお，これらの値は上で述べた理想的な場合の値（**最良値**）である．

3.3 リモートノード離脱時の finger table の更新

ここでは finger table 更新時にリモートノードが離脱（故障を含む）していた場合の処理について議論する．

2.2 節で述べたように，Chord$^{\#}$ では，ノード u が finger table を更新する際，u.finger[i] は u.finger[$i-1$] の指すノードから取得する．このとき u.finger[$i-1$] が指すノードが離脱していた場合，u.finger[$i-1$] が正しい（離脱していない）ノードを指すように修復する必要がある．しかし，u.finger[$i-1$] は，u.finger[$i-2$] が指すノード（v とする）から取得するため，v が finger table を更新するまで，u は u.finger[$i-1$] を更新できない．さらに，（再帰的に）v が自身の finger table を更新する際も同じ問題が発生する．このように，Chord$^{\#}$ ではノードの離脱によって複数のノードの finger table 更新処理が遅延するという問題がある．

Chord$^{\#\#}$ では，この問題は，アクティブな更新を行うノードの finger table の各行に複数のエントリがあることを利用して解決できる．u.fingers[$i-1$][0] が指すノー

© 2012 Information Processing Society of Japan

表 1 アクティブな更新を行うノード数と 1 ノードが 1 回の更新に必要とする平均メッセージ数（n nodes）
Table 1 Average number of messages for updating a finger table.

	アクティブな更新を行うノード数	1 ノードあたりの平均メッセージ数（パッシブな更新に必要なメッセージを含む）	
		反復ルーティング	再帰ルーティング
Chord$^{\#}$	n	$2\lceil\log_2 n\rceil$	$\lceil\log_2 n\rceil + 1$
Chord$^{\#\#}$	$\left\lceil\frac{n}{s+1}\right\rceil$	$\left\lceil\frac{n}{s+1}\right\rceil \times \frac{2\lceil\log_2 n\rceil + s}{n}$	$\left\lceil\frac{n}{s+1}\right\rceil \times \frac{\lceil\log_2 n\rceil + s + 1}{n}$

ド（x とする）が離脱していた場合，u.fingers[$i-1$][0] は x の successor に変更すればよい．ここで u は，x を，u.fingers[$i-2$][0] が指すノード（v とする）から得たことに注意すると (i.e., $x = u$.fingers[$i-2$][0].fingers[$i-2$][0])，x の successor は，u.fingers[$i-2$][1] から得ることができる (i.e., x.successor $= u$.fingers[$i-2$][1].fingers[$i-2$][0]) [*2]．このように，Chord$^{\#\#}$ では finger table 更新時にリモートノードが離脱していた場合でも，他のノードによる修復を待つ必要がないため，finger table の更新をすみやかに行うことができる．

3.4 ネットワーク距離を考慮したルーティング

Chord$^{\#\#}$ ではパッシブな更新を行うノードの finger table の幅 (p) を 2 以上にすることでルーティング時にノード間のネットワーク距離（Round Trip Time など）を考慮することができる．検索クエリを転送する際に，次ホップとして finger table の各行の複数のエントリの中から最もネットワーク距離が近いノードを選択することで検索時間を短縮する．以下は，ノード u がキー k を検索する場合のアルゴリズムである．

(1) u が k を保持していれば，それを結果として返し，終了する．
(2) $k \in (u.\text{key}, u.\text{successor.key})$ ならば，キー k を保持するノードが存在しないため，nil を返し，終了する．なお，$k \in (k_1, k_2)$ はリング上の key 空間で k_1 から k_2 の範囲（ただし両端は含まない）に k が含まれる場合に真となる．
(3) u の successor_list か finger table にキー k のエントリがあれば，当該エントリの指すノードに検索クエリを転送し，終了する．
(4) u.fingers[i][0] $< k$ を満たす最大の i を求める．また，ノード集合 $S = \{v | v \in u.\text{fingers}[i][j]\ (0 \le j < u.f_w) \wedge (v.key \in (u.key, k))\}$ とする．
- $i = 0$ の場合，S の中でキーが k に最も近いノードに検索クエリを転送し，終了する．
- $i > 0$ の場合，S の中で，最もネットワーク距離が近いノードに検索クエリを転送し，終了する．最もネットワーク距離が近いノードを見つけるためには，finger table 中の各ノードとの Round Trip Time を計測しておく，あるいは Vivaldi [9] のようなネットワーク座標を用いる．

このアルゴリズムは，可能なすべての経路の中からネットワーク距離が最短な経路を選べるわけではないが，4.4 節で示すように検索時間を大幅に短縮できる．また，p を大きくすると次ホップとして選べるノードの候補が増えるため，検索時間をより短縮できるが，finger table 更新にかかるネットワーク帯域は増加する．これは 4.5 節で評価する．

4. 評価

提案手法をシミュレーションにより評価した．

4.1 ノードあたりの平均メッセージ数

finger table の更新に必要なメッセージ数を測定した．$n = 2^{10}$ 個のノードを用意し，パラメータ $T = 60$ 秒，$\beta = 0.5$ 秒に設定した．各ノードには 0 から $2^{31}-1$ の範囲の乱数で key を付与した．1 回のアクティブな更新によって，パッシブな更新を行うノードの数（s）を 0 から 15 まで変化させながら，10 時間相当のシミュレーションを行い，その間に各ノードが finger table をアクティブ，あるいはパッシブに更新するために送受信したメッセージ数を測定した．結果を**図 6** に示す．横軸は s，縦軸は 1 ノードが 1 回の更新に必要とする平均メッセージ数である．なお，アクティブな更新には反復ルーティングを用いた．また，Chord$^{\#}$ と Chord$^{\#\#}$ における最良値（表 1）もプロットした．

Chord$^{\#\#}$ のプロットより，1 ノードが 1 回の更新で必要な平均メッセージ数は s の増加にともなって表 1 の最良値とほぼ同じように減少することが確認できる．なお，最良値よりも若干値が大きいのは，3.2.2 項で述べたオーバラップ更新が生じているためである．s が大きくなるに従って最良値との差が縮まっているのは，1 回のアクティブな更新でオーバラップ更新を解消できるノード数が増加するためである．

4.2 ノードの参加・離脱（Churn）による影響

ノードの参加・離脱の頻度が高い場合における有効性

*2 x の successor も離脱していた場合は，u.fingers[$i-2$][2] を使用すれば良い．

© 2012 Information Processing Society of Japan

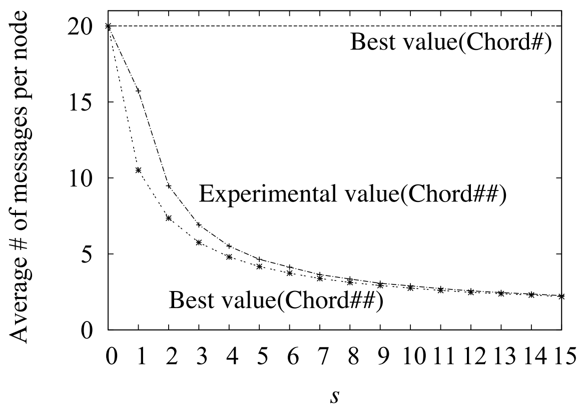

図 6 finger table の更新に要する 1 ノードあたりの平均メッセージ数（iterative）
Fig. 6 Average number of messages per node for updating finger tables (iterative).

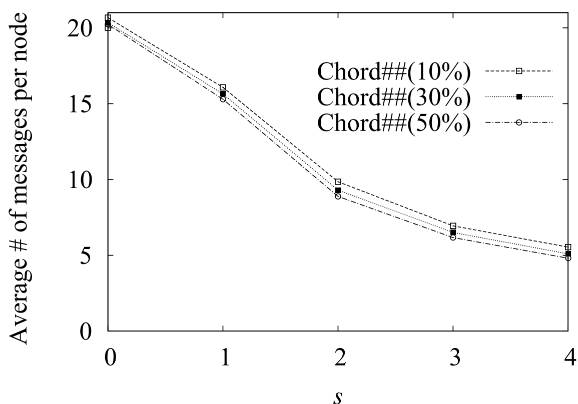

図 7 Churn rate の変化による 1 ノードあたりの平均メッセージ数
Fig. 7 Churn rates and average number of messages per node.

を確認するため，ノードが参加・離脱する状況で 4.1 節と同様の実験を行った．実験では，ノードはパラメータ $\lambda = nx$ [nodes/h] のポアソン分布に従って参加・離脱するものとした（n はノード数）．x は 1 時間あたりの参加・離脱するノードの，全ノードに対する割合（Churn rate）である．ここでは，x の値を 10%，30%，50%とし，それぞれ 100 回試行した．また，s は 0 から 4 まで変化させた．なお，3.3 節で述べた，ノード離脱時の finger table 更新処理は実装している．結果を**図 7** に示す．横軸は s，縦軸は 1 ノードが 1 回の更新に必要とする平均メッセージ数である．図 6 の $0 \leq s \leq 4$ の部分と比較すると，ノードの参加・離脱の頻度が高くなっても，finger table の更新に必要なメッセージ数はほとんど変化しないことが分かる．なお，Churn rate の値が大きいほど平均メッセージ数が若干小さくなるのは，ネットワークに参加してからアクティブ，あるいはパッシブな更新を 1 回も行わずに離脱するノードが多くなるためである．

4.3 アクティブな更新を行うノードの分布

アクティブな更新を行うノードが均等に分布しているか

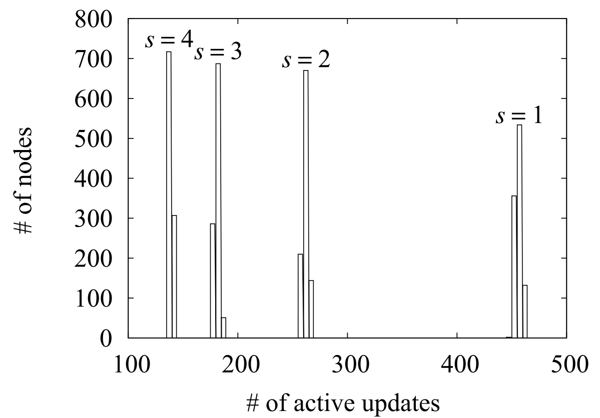

図 8 ノードあたりのアクティブな更新回数の分布
Fig. 8 Distribution of the number of active updates per node.

どうかを確認するため，4.1 節と同じ設定で，パッシブな更新を行うノード数 s を 1 から 4 まで変化させながら，10 時間の間に各ノードがアクティブな更新を行う回数を測定した．各 s におけるアクティブな更新回数のヒストグラムを**図 8** に示す．横軸はアクティブな更新回数（階級の幅は 5），縦軸はノード数（頻度）である．たとえば $s = 1$ の場合，459 回をピークとして前後のごく狭い範囲に分布が集中しており，アクティブな更新を行うノードは均等に分布していることが確認できる．その他の s についても同様である．

なお，T の値が 60（秒）であるため，測定期間内に各ノードが finger table をアクティブに更新する回数は，理想的には $(10 \times 3600)/(60 \times (s+1)) = 600/(s+1)$ 回となる．測定結果はこれより大きいが，これもオーバラップ更新の影響である．

4.4 ネットワーク距離を考慮したルーティング

3.4 節で述べたように，Chord$^{\#\#}$ ではパッシブな更新を行うノードの finger table の幅（p）を 2 以上にすることで，ネットワーク距離を考慮した検索が可能である．ここではこの効果を確認した．

まず，インターネットトポロジジェネレータ Inet-3.0 [10] を用いて 4,096 ノードの仮想ネットワークを作成し，ここからランダムにノードを選択して Chord$^{\#}$ と Chord$^{\#\#}$ のネットワークを構築した．その上で，Chord$^{\#}$ および Chord$^{\#\#}$ のそれぞれにおいて，ノード u と v を総当たりで選び，u から v に到達するまでの検索時間を求めた．Chord$^{\#}$ と Chord$^{\#\#}$ のネットワーク構築はそれぞれ 10 回ずつ行い，平均値を求めた．また，$r = 10$ とした．Inet-3.0 が生成したノード間の遅延時間には具体的な単位がないため，ここでは遅延時間の値を 100 で割り，ミリ秒として扱った．Chord$^{\#}$ では，文献 [2] にならって finger table に加え successor list も利用して検索するようにした．他の設定は 4.1 節と同様である．

© 2012 Information Processing Society of Japan

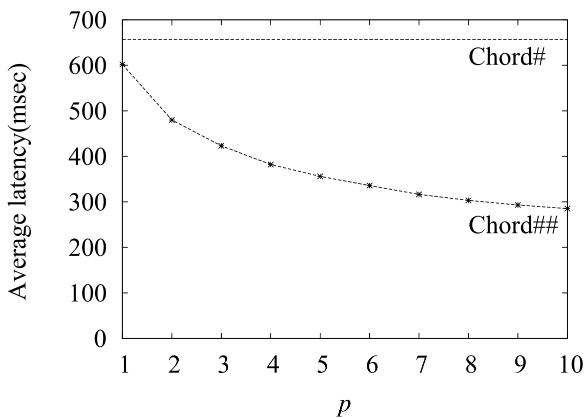

図 9　ネットワーク距離を考慮した検索（1,000 nodes）
Fig. 9　Average latency of lookup operations with proximity routing (1,000 nodes).

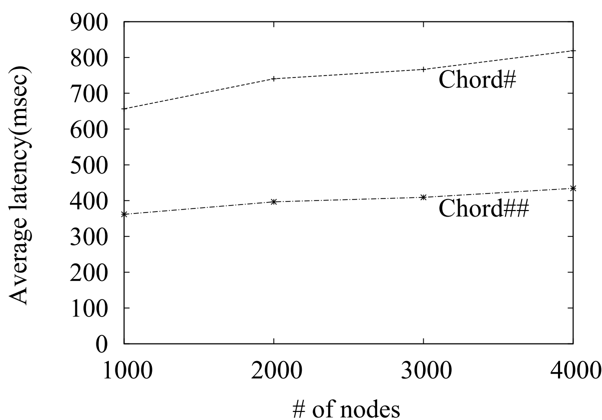

図 10　ネットワーク距離を考慮した検索（$p=5$）
Fig. 10　Average latency of lookup operations with proximity routing ($p=5$).

図 9 はノード数を 1,000 に固定し，p を 1 から 10 まで変化させたときの検索時間の平均値をプロットしたものである（このときの s は式 (2) で決まる）．参考に Chord$^{\#}$ の平均値もプロットしている．グラフより，p が大きいほど検索時間が減少することが確認できる．

図 10 はノード数を 1,000，2,000，3,000，4,000 と変化させたときのグラフである．Chord$^{\#\#}$ のパラメータとしては $p=5$，$s=r+1-p=6$ を用いた．グラフより，ノード数が増加してもネットワーク距離を考慮したルーティングは有効であることが確認できる．

4.5　finger table の更新に必要なネットワーク帯域

Chord$^{\#}$ と Chord$^{\#\#}$ のそれぞれにおいて finger table を更新するために必要なネットワーク帯域を評価した．ここでは，ノード間の通信に UDP を用いることを想定し，IP レベルでのメッセージサイズを用いて評価する．IP パケットは IP ヘッダ（20 byte），UDP ヘッダ（8 byte），ペイロードで構成され，1 パケットにつき送信可能な最大ペイロード長は 1,024 byte とする．また，文献 [5] と同様，1 つのポインタは 24 byte（ノードの IP アドレス 4 byte とキー 20 byte）とする．

表 2　finger table の更新処理に必要なメッセージのバイト数
Table 2　Sizes of messages for updating finger tables.

メッセージ	byte
$\langle$get_fingerOp, $pointer$, $index\rangle$	$\langle 1, 24, 1\rangle$
$\langle$reply_fingerOp, $finger'$, $successors[]\rangle$	$\langle 1, 24, 24r\rangle$
$\langle$updateOp, $fingers'[][]$, $seq\rangle$	$\langle 1, 24(p\lceil\log_2 n\rceil-\alpha_p), 1\rangle$

Chord$^{\#\#}$ では $p>1$ のときネットワーク距離を考慮した検索ができるが，このときメッセージサイズは増加する．Chord$^{\#\#}$ と Chord$^{\#}$ を平等な条件で比較するため，Chord$^{\#}$ においてもネットワーク距離を考慮した検索ができるように finger table を 2 次元化した場合を想定する．Chord$^{\#}$ ではすべてのノードはアクティブな更新を行うが，このとき successor list を $p-1$ 列取得するものとする．

次に finger table の更新時に送受信されるメッセージ（get_fingerOp，reply_fingerOp，updateOp）のサイズを検討する．各メッセージは先頭 1 バイトでタイプを指定し，次にメッセージごとに異なる引数列が続くものとする．

Chord$^{\#\#}$ での各メッセージのバイト数は**表 2** のとおりである．ここで α_p は 3.2.3 項で述べた，finger table を送信する際に省ける finger table の要素数であり，$\alpha_1=0$，$\alpha_2=1$，$\alpha_3=3$，$\alpha_4=5$，$\alpha_5=8$ である[*3]．1 回のアクティブな更新では get_fingerOp と reply_fingerOp をそれぞれ $\lceil\log_2 n\rceil$（finger table の高さ）回送信するため，必要なメッセージのバイト数は $(51+24r)\lceil\log_2 n\rceil$，パッシブな更新では，updateOp を 1 回送信するため $2+24(p\lceil\log_2 n\rceil-\alpha_p)$ となる．

Chord$^{\#}$ の場合は get_fingerOp と reply_fingerOp のみを用いる．reply_fingerOp メッセージでは successor list を $p-1$ 列取得するため，必要なメッセージのバイト数は $(51+24(p-1))\lceil\log_2 n\rceil$ となる．

finger table の更新に必要なネットワーク帯域を求めるためには，各ノードのアクティブおよびパッシブな更新の回数が必要となる．このため，4.1 節と同様の設定で，パッシブな更新を行うノード数（s）を 0 から 15 まで変化させながら，10 時間の間にすべてのノードが行うアクティブおよびパッシブな更新の回数をそれぞれ測定した．平均値を**図 11** に示す．

これらの値を用いて Chord$^{\#}$ と Chord$^{\#\#}$ における finger table の更新に必要なネットワーク帯域の平均値を求めた．グラフを**図 12** に示す．p は 1 から 5 まで，s は 0 から 15 まで変化させている．Chord$^{\#\#}$ における r は式 (2) で求めた．

*3　$\alpha_p=\sum_{i=0}^{\lfloor\log_2 p\rfloor}(p-2^i)$ によって求められる．

© 2012 Information Processing Society of Japan

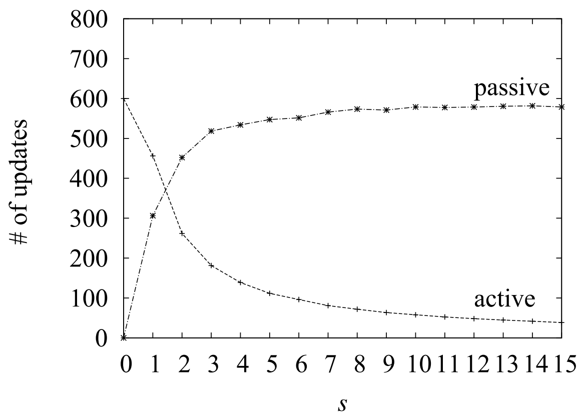

図 11 アクティブおよびパッシブな更新回数
Fig. 11 Number of active/passive updates.

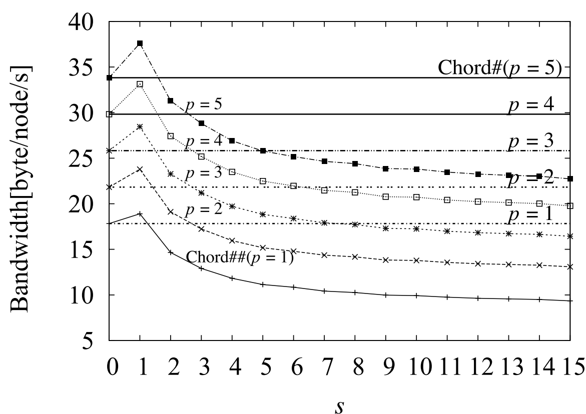

図 12 finger table 更新に必要なネットワーク帯域
Fig. 12 Bandwidth consumption for updating finger tables.

同一の p どうしのプロットを比較すると，$s \geq 2$ の領域で Chord## は Chord# よりも必要なネットワーク帯域が小さくて済むことが確認できる．

5. 関連研究

構造化 P2P ネットワークにおける経路表の維持管理コストを削減するために，ノード間で送受信される検索クエリを利用するという方法がある [11]．この方法では，経路表を維持管理するために追加のメッセージを必要としない．しかし，この手法はノードの間で十分に検索クエリが送受信されることと，ノードの参加・離脱の頻度（churn rate）より検索の頻度（lookup rate）が高いことを前提としているため，検索クエリの頻度が低い場合や，lookup rate より churn rate が高い場合は経路表の正確さが低下する．

提案手法は，定期的に経路表を更新するメリット（経路表の正確性が検索クエリの頻度や churn rate に依存しない）を生かしつつ，経路表の維持管理に必要なメッセージ数やネットワーク帯域を削減したところに意義がある．

6. おわりに

本稿では範囲検索が可能な構造化 P2P ネットワークの 1 つである Chord# の finger table 更新コストを削減する手法（Chord##）を提案した．Chord## では，finger table に successor list を加えて 2 次元化することで，finger table 更新に必要なメッセージ数の削減，リモートノード離脱時の finger table 更新処理の高速化，ネットワーク近接性を利用したルーティングを実現した．

今後の課題としては，Chord## を実ネットワーク上で実装し，評価することがあげられる．

参考文献

[1] 呉 承彦，安倍広多，石橋勇人，松浦敏雄：Chord# における経路表の維持管理コスト削減手法，情報処理学会第 19 回マルチメディア通信と分散処理ワークショップ（DPSWS2011）論文集，pp.250–256 (2011).

[2] Stoica, I., Morris, R., Liben-Lowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F. and Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications, *IEEE/ACM Trans. Networking*, Vol.11, No.1, pp.17–32 (2003).

[3] Rowstron, A. and Druschel, P.: Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems, *IFIP/ACM International Conference on Distributed Systems Platforms* (*Middleware*), pp.329–350 (2001).

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

[5] Schütt, T., Schintke, F. and Reinefeld, A.: Range queries on structured overlay networks, *Computer Communications*, Vol.31, pp.280–291 (online), DOI: 10.1016/j.comcom.2007.08.027 (2008).

[6] Aspnes, J. and Shah, G.: Skip graphs, *ACM Trans. Algorithms*, Vol.3, No.4, pp.1–25 (2007).

[7] Pugh, W.: Skip Lists: A Probabilistic Alternative to Balanced Trees, *Comm. ACM*, Vol.33, No.6, pp.668–676 (1990).

[8] Schütt, T., Schintke, F. and Reinefeld, A.: Structured overlay without consistent hashing: Empirical results, *GP2PC* (2006).

[9] Dabek, F., Cox, R., Kaashoek, F. and Morris, R.: Vivaldi: a decentralized network coordinate system, *SIGCOMM Computer Communication Review*, Vol.34, No.4, pp.15–26 (online), DOI: 10.1145/1030194.1015471 (2004).

[10] Winick, J. and Jamin, S.: Inet-3.0: Internet Topology Generator, Technical Report CSE-TR-456-02, University of Michigan (2002).

[11] Alima, L., El-Ansary, S., Brand, P. and Haridi, S.: DKS(N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications, *The 3rd International Workshop on Global and P2P Computing on Large Scale Distributed Systems, CCGRID 2003* (2003).

推薦文

分散ハッシュ表 Chord に対して順序性を保持する改良を行った Chord# に対して，finger table 更新のメッセージ数を削減する手法を提案している．提案手法では，隣接ピ

© 2012 Information Processing Society of Japan

アの経路表の重複エントリに着目し，重複部分のメッセージの削減を行っている．本研究は大規模な分散システムを構築する基本アルゴリズムとして新規性および汎用性が高く，評価実験結果の信頼性も高い．以上から DPS ワークショップ 2011 からの推薦論文に値する．

（マルチメディア通信と分散処理研究会主査　勝本道哲）

呉 承彦　（学生会員）

平成 24 年大阪市立大学大学院創造都市研究科修士課程修了．同年同研究科博士（後期）課程入学．現在に至る．修士（都市情報学）．P2P システムの研究に従事．

安倍 広多　（正会員）

平成 4 年大阪大学基礎工学部情報工学科卒業．平成 6 年同大学大学院博士前期課程修了．同年 NTT 入社．平成 8 年大阪市立大学学術情報総合センター助手．平成 12 年同講師．平成 15 年同大学院創造都市研究科講師．平成 17 年同助教授．平成 24 年より同教授．博士（工学）．基盤ソフトウェア，分散システム等に興味を持つ．IEEE，電子情報通信学会各会員．

石橋 勇人　（正会員）

昭和 62 年京都大学大学院工学研究科修士課程情報工学専攻修了．平成元年同博士後期課程情報工学専攻退学後，京都大学大型計算機センター助手．平成 10 年大阪市立大学学術情報総合センター講師．平成 14 年同助教授．平成 15 年同大学院創造都市研究科助教授．平成 19 年より同教授．博士（情報学）．高速ネットワーク，ネットワーク管理システム等に関する研究に従事．人工知能学会，電子情報通信学会，IEEE，ACM 各会員．

松浦 敏雄　（正会員）

昭和 50 年大阪大学基礎工学部情報工学科卒業．昭和 54 年同大学大学院基礎工学研究科（情報工学専攻）博士後期課程退学後，大阪大学基礎工学部情報工学科助手．平成 4 年同大学情報処理教育センター助教授．平成 7 年大阪市立大学生活科学部教授．平成 8 年同大学学術情報総合センター教授．平成 15 年同大学院創造都市研究科教授現在に至る．工学博士．ソフトウェア開発環境，ユーザインタフェース，マルチメディア，情報教育等に興味を持つ．ACM，IEEE，電子情報通信学会各会員．

© 2012 Information Processing Society of Japan