Chord$^{\#}$ における経路表の維持管理コスト削減手法

呉 承 彦[†1] 安 倍 広 多[†1]
石 橋 勇 人[†1] 松 浦 敏 雄[†1]

Chord$^{\#}$ は構造化 P2P(Peer-to-Peer) ネットワークの一種である．Chord などの分散ハッシュテーブルに基づくシステムとは異なり，キーの範囲検索が可能な点に特徴がある．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上で，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のコストを削減する．シミュレーションにより finger table の更新処理コストを削減できることを確認した．

A Method for Reducing Maintenance Cost of Routing Table in Chord$^{\#}$

Seungeon Oh,[†1] Kota Abe,[†1] Hayato Ishibashi[†1] and Toshio Matsuura[†1]

Chord$^{\#}$ is a kind of structured Peer-to-Peer (P2P) network. Unlike those systems based on distributed hash tables (DHT) such as Chord, Chord$^{\#}$ is able to handle 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. In the proposed method, a finger table is extended to two-dimensional array and the cost of updating finger tables is reduced by using the similarity of finger tables of adjacent nodes. We have simulated the algorithm and confirmed that the cost of updating finger tables is actually reduced.

1. はじめに

ネットワークで接続された多数のノードで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することにより，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[1)]， Pastry[2)]，Tapestry[3)] などがある．しかし，DHT では key をハッシュすることでデータを配置するノードを決定するため，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 ネットワークの一つに Chord$^{\#}$[4)] 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 と異なり，ノードは key の値の昇順に並べられる．このため，範囲検索が容易に実現できる．

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 としたとき，$O(\log n)$ ホップで検索できる．

さて，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．本稿ではこの更新処理のコストを削減する方式（**Chord$^{\#\#}$** と呼ぶ）を提案する．Chord$^{\#\#}$ は，finger table を 2 次元配列に拡張した上で，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のコストを削減する．また，Chord$^{\#\#}$ ではネットワーク近接性を利用したルーティングも実現している．

以下，2 章で Chord$^{\#}$ について述べ，3 章で Chord$^{\#\#}$ のアルゴリズムを示す．4 章で Chord$^{\#\#}$ の評価と考察を行い，最後に 5 章でまとめと今後の課題を述べる．

2. Chord$^{\#}$

ここでは，Chord$^{\#}$ について簡単に説明する．

†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

表 1 Chord# で各ノードが保持するデータ

変数	説明
key	キーの値
successor	リング上で時計回りに最も近いノード
predecessor	リング上で反時計回りに最も近いノード
successor_list[]	successor のリスト (長さ r)
finger[]	ショートカットのためのノード集合

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 を保持していて，リング上の 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．なお，Chord と異なり，リングの key 空間の大きさ（最小値，最大値）はあらかじめ決めておく必要はない．

以後，Chord# の（あるいは後で述べる Chord## の）全ノードの集合 $V = \{N_0, N_1, \cdots, N_{n-1}\}$ とし（ただし n はノード数），$N_i.key < N_{i+1}.key$ が成り立つものとする．また，$a \oplus b \equiv (a+b) \bmod n$ とする．

Chord# の各ノードは key 以外に，successor，predecessor，finger table，および，successor list を持つ（**表 1**）．successor は key からリング上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．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．ノード N_i の successor は $N_{i\oplus 1}$，predecessor は $N_{i\oplus(n-1)}$ となる．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る．successor list は successor ノードが離脱や故障した場合に，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）．

以後，ノード u の保持する変数 x を $u.x$ のように表記する．

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 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 は十分大きいものとする）．まず，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．次に N_0.finger[1] には，finger[0] が指すノード N_1 が保持する N_1.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，finger[1] が指すノード N_2 が保持する N_2.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ことになる．なお，各ノードの finger table の高さは $\lceil \log_2 n \rceil$ となる．

ノードの挿入や削除，障害などによって各ノードの finger table は理想的な状態（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 finger[$i-1$] の指すノードの finger[$i-1$] を取得する必要がある．ノードの finger table のすべての行を更新するために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）である．

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，上で述べた finger table の更新に必要なメッセージ数を削減することにある．これを実現するために，Chord## では (1) リモートのノードから 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，(2)finger table を 2 次元化し，取得した successor list を列方向に追加する．こうすることで隣接したノード間の finger table に共通部分ができるため，(3)finger table を更新したノードは共通部分を隣接ノード（successor ノード）に順次転送する．finger table を受信したノードは自身で finger table を更新する必要がないため，必要なメッセージ数を削減できる．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 Chord# の finger table を列方向に拡張した 2 次元配列 (fingers[][]) である．ノード u の finger table の高さを $u.f_h$，幅を $u.f_w$ とする．高さ $u.f_h = \lceil \log_2 n \rceil$ であるが，幅 $u.f_w$ はノードによって異なる（詳細は 3.3 節で述べる）．$u.f_w \le r + 1$ である．

Chord## の fingers[i][0] は，Chord# の finger[i] と同様，式 (1) で求めたノードへのポインタを保持する．また，ノード u の fingers[i][j] ($0 < j < u.f_w$) は，u.finger[i][0] が指

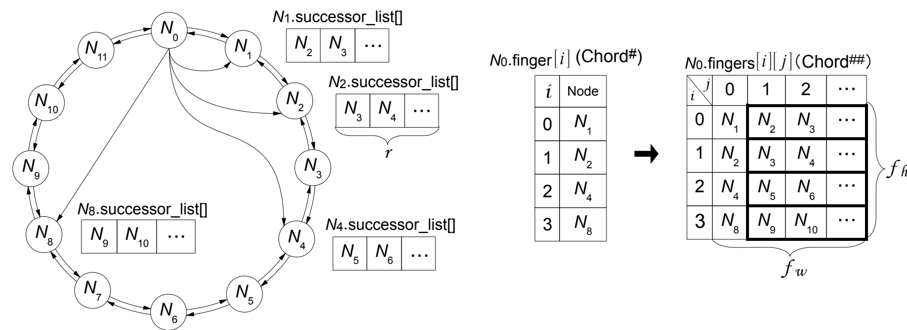

図 1　Chord$^{\#}$ と Chord$^{\#\#}$ の finger table

すノードの successor_list$[j-1]$ を保持する．

図 1 は Chord$^{\#}$ と Chord$^{\#\#}$ におけるノード N_0 の finger table の例である．Chord$^{\#}$ での 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$^{\#\#}$ での N_0 の finger table ではそれに加え，列方向（太枠内）に N_1，N_2，N_4，N_8 の各々の successor list を格納している．

Chord$^{\#\#}$ の finger table について，以下の性質が成り立つ．

定理 1． *Chord*$^{\#\#}$ 上のノード u，v に対し，v が u の *successor* で，$u.f_w - v.f_w = 1$ のとき，$u.fingers[i][j] = v.fingers[i][j-1]$ $(0 < j < u.f_w)$.

証明．$u = N_u$ とすると，$v = N_{u\oplus 1}$ である．Chord$^{\#\#}$ の finger table の構築方式から，N_k.fingers$[i][j] = N_{k\oplus(2^i+j)}$ が成り立つ．このため，$0 < j < N_u.f_w$ に対して，N_v.fingers$[i][j-1] = N_{v\oplus(2^i+j-1)} = N_{u\oplus(2^i+j)} = N_u$.fingers$[i][j]$.　□

3.2　検索アルゴリズム

Chord$^{\#\#}$ では finger table の各行に複数のポインタを保持していることを利用することで，ネットワーク距離 (Round Trip Time など) を考慮した検索を行うことができる．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)　$k \in (u.\mathrm{key}, u.\mathrm{successor.key})$ 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，$k_x \in (k_1, k_2)$ はリング上の key 空間で k_1 から k_2 の範囲に k_x が含まれるかどうかを判定する（ただし両端は含まない）．

(3)　finger table に $u.\mathrm{fingers}[i][j] = k$ であるようなエントリがあれば，当該エントリの指すノードに検索要求を転送し，終了する．

(4)　$u.\mathrm{fingers}[i][0] \leq k < u.\mathrm{fingers}[i+1][0]$ であるような i を求める．また，ノード集合 $S = \{v | v \in u.\mathrm{fingers}[i][j]\ (0 \leq j) \wedge (v.key \in (u.key, k))\}$ 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要求を転送する．
- $i > 0$ 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要求を転送する．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[5] などのネットワーク座標を用いるなどの方法が考えられる．

3.3　メンテナンスアルゴリズム

本節では，Chord$^{\#\#}$ 上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．

基本的には Chord$^{\#\#}$ のノード 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．u がノード $x = u.\mathrm{finger}[i-1][0]$ に finger$[i-1][0]$ を問い合わせるときに，同時に x の successor list を受け取ればよい．このようにリモート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，Chord$^{\#\#}$ 上のノード N_u は $N_{u\oplus 1}$ $(= N_u.\mathrm{successor})$ の fingers$[i][j]$ $(0 \leq j < r)$ に相当する部分を保持している．したがって，N_u が finger table を上記の手順で更新した際に，finger table の第 1 列以降（fingers$[i][j]$ $(1 \leq j \leq r)$）を $N_{u\oplus 1}$ に転送することにすれば，$N_{u\oplus 1}$ は自ら情報を収集することなく fingers$[i][j]$ $(0 \leq j < r)$ を得ることができる．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**と表現する．

ここで，N_u の finger table が $r+1$ 列からなるのに対して，その successor である $N_{u\oplus 1}$ ではこのうち r 列のみが有効となる点に注意すると，$N_{u\oplus j}$ においては $r+1-j$ 列のみが有効となることがわかる (**図 2** 参照)．すなわち，各ノードが保持する finger table の幅は一定ではない．また，$r+1-j > 0$ でなければならないので，N_u の finger table の情報を利用することができるのは最大 r 個のノードとなる．ところで，3.2 節で述べた複数のポインタを利用してネットワーク距離を考慮した検索を行なうためには，一定数のノードが必

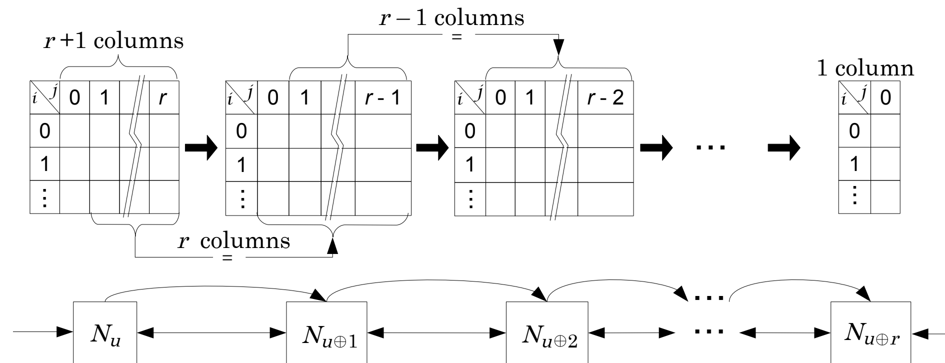

図 2 finger table の送信

要であり，この数を p $(2 \leq p \leq r)$ とすると，各ノードは少なくとも p 列の finger table を保持しなければならない．このため，実際に可能な転送回数 s は，$r - p$ 回となる．

3.3.1 アルゴリズムの概要

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．ある時点で他のノードから 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なうが，更新がないままに一定時間が経過した場合はアクティブな更新を行う．このために，各ノードは更新周期を管理するためのタイマーを有している．

今，ノード N_u のタイマーが満了したとすると，更新手順は次のようになる．

(1) ノード N_u は finger table のアクティブな更新を行ない，タイマーをリセットする．

(2) 更新した finger table のうち，fingers$[i][j](1 \leq j \leq r)$ を successor に転送する．

(3) successor ノードは受信した finger table を用いてパッシブな更新を行い，タイマーをリセットする．

(4) 受信した successor が新たな転送元となり，手順 2 以降を繰り返す．ただし，設定した転送回数 $s(= r - p)$ に達したところで繰り返しを終了する．

3.3.2 更新待ち時間

ここで，各ノードが finger table を更新する間隔について検討する．

Chord$^{\#\#}$ の各ノードが保持するタイマーの設定値がすべて同じ値 t であるとしたとき，N_0 がアクティブな更新を行ない，それに伴って N_1 から N_4 までのノードがパッシブな更新を行なった場合の動作シーケンスを**図 3** に示す．

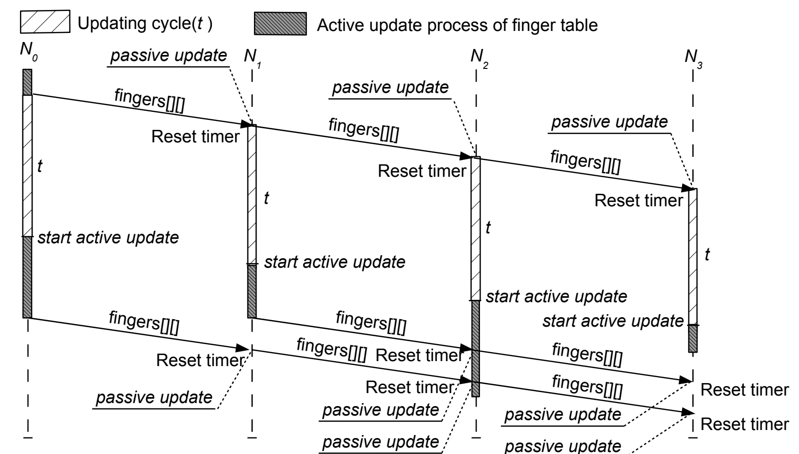

図 3 各ノードが更新周期 (t) 待つ場合の finger table の更新例

図のように，N_0 が更新を行った後，パッシブな更新なしに時間 t が経過すると，N_0 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時間を要するため，N_0 の更新が終了して finger table が到着する前に N_1 の更新タイマーが満了となり，N_1 もまたアクティブな更新を始める．ここで，N_1 の更新が N_0 よりも短い時間で終了したとすると，N_1 はアクティブな更新が終了した後に N_0 からの finger table を受け取ることになり，パッシブな更新によるメッセージ数削減の効果が失われてしまう．

このような状況を減らすため，finger table を受信した後，更新タイマーをリセットする際に，システム全体で定められた更新周期 t に，ノードごとに異なる受信待ち時間 $k \times \beta$ $(0 \leq k \leq s)$ を加えた値をタイマーにセットすることにする．β は Chord$^{\#\#}$ 上の各ノードが finger table のアクティブな更新に要する最大時間である．受信待ち時間 $k \times \beta$ を設定することで，あるノードがアクティブな更新を行なっている間に後続するノードがアクティブな更新を行なう可能性を減らすことができる (**図 4**).

k の値は，次のように定める．アクティブな更新を行なったノード (図 4 の N_0 とする) から最初に finger table を受信したノード N_1 では $k = 0$，2 番目に受信したノード N_2 で

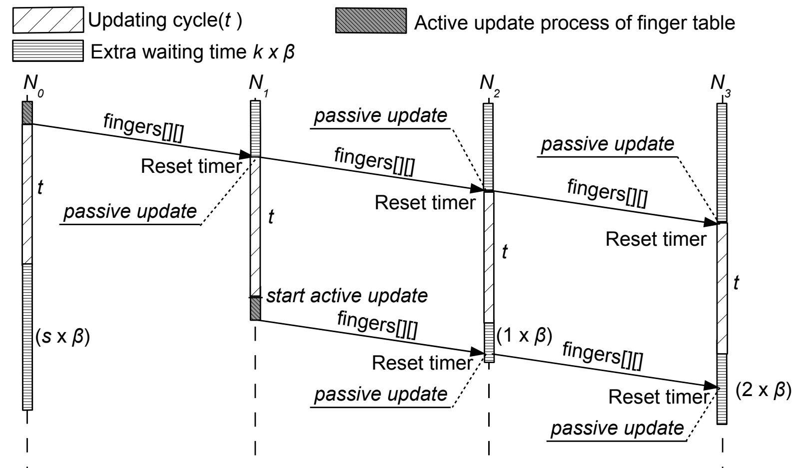

図 4 t に受信待ち時間 $k \times \beta$ を加えた場合の finger table 更新例

は $k = 1$，3 番目に受信したノード N_3 では $k = 2$，$\cdots$ とする．アクティブな更新を行なったノード N_0 では，$k = s$ とする．

アクティブな更新を行なった N_0 の受信待ち時間を最大値に，N_1 の受信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負荷分散するためである．

3.3.3 アルゴリズムの詳細

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

```
1  process P
2  // タイマーの満了によってfinger table のアクティブな更新を開始する
3  upon receiving timeout
4          fingers[0][0] = successor
5          i = 0
6          while (true) {
7                  i = i + 1
8                  // fingers[i − 1][0]へ get_fingerOp メッセージを送信
9                  send ⟨get_fingerOp, P, i⟩ to P.finger[i − 1][0]
10                 upon receiving ⟨get_fingerOp, f, succlist[]⟩
11                 for j = 1 to r
12                         P.fingers[i − 1][j] = succlist[j]
13                 if (f ∈ [P.key, P.finger[i − 1][0]))
14                         break
15                 P.fingers[i][0] = f
16         }
17         // タイマーをt + (β × s)にリセット
18         // timer(x)はx秒後に timeout イベントを発生させる関数
19         timer(t + (β × s))
20         // 更新したfinger table を successor に送信
21         // fingers[][1:]はfingers[i][j](0 ≤ i < P.f_h, 1 ≤ j < P.f_w)
22         send ⟨updateOp, fingers[][1:]⟩ to successor
23 // updateOp メッセージを受信した場合，パッシブな更新を行う
24 upon receiving ⟨updateOp, fs[][]⟩:
25         // 受信したfinger table に更新
26         P.fingers[][] = fs[][]
27         // タイマーをt + (β × (r−P.f_w)にリセット
28         timer(t + (β × (r − P.f_w))
29         if (P.f_w > p)
30                 // 更新したfinger table を successor に送信
31                 send ⟨updateOp, P.fingers[][1:]⟩ to successor
32 // get_fingerOp メッセージを受信した場合，経路表の情報を返す
33 upon receiving ⟨get_fingerOp, P′, index⟩:
34         send ⟨get_fingerOp, P.fingers[index − 1][0], P.successor_list[]⟩ to P′
```

3.3.4 アクティブな更新を行うノード数とノードあたりの平均メッセージ数

Chord$^{\#}$ では，finger table を 1 回更新する周期の間にすべてのノードが 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，平均的にはアクティブな更新を行うノード数は $\lceil \frac{n}{s+1} \rceil$ 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 $2s$（または s）である．そのため，ノード u と，u に引き続く s 個のノード（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 $\lceil \log_2 n \rceil + 1 + s$）となる．これは Chord$^{\#}$ 上の $s + 1$ 個のノード

表 2 アクティブな更新を行なうノード数とノードあたりの平均メッセージ数 (n nodes)

	アクティブな更新を行なうノード数	1 ノードあたりの平均メッセージ数	
		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##	$\left\lceil \frac{n}{s+1} \right\rceil$	$\left\lceil \frac{n}{s+1} \right\rceil \times \frac{2(\lceil \log_2 n \rceil + s)}{n}$	$\left\lceil \frac{n}{s+1} \right\rceil \times \frac{\lceil \log_2 n \rceil + 1 + s}{n}$

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 $s+1$ に反比例して少ない．**表 2** に，Chord# と Chord## におけるアクティブな更新を行なうノード数と 1 ノードあたりの平均メッセージ数を示す．

4. 評　　価

提案手法をシミュレーションにより評価した．

まず，finger table の更新にかかるメッセージ数の削減効果を測定した．2^{10} 個のノードを用意し，更新周期 $t = 20$ 秒，$\beta = 0.5$ 秒に設定した．転送回数 (s) を 0 から 19 まで変化させながら，$100t (= 2000)$ 秒の間にすべてのノードが finger table を更新するために送受信したメッセージ数を測定した．これを 100 回試行し，その平均値をプロットしたグラフが**図 5** である．横軸は転送回数 s，縦軸はノードあたりの平均メッセージ数である．なお，ルーティングは反復ルーティングを用いている．また，Chord# と Chord## における理論値（表 2）もプロットした．

グラフより，転送回数 s が大きいほど，ノードあたりの平均メッセージ数が減少することが確認できる．なお，理論値よりも若干値が大きいのは，パッシブな更新で済むノードの一部がタイムアウトによってアクティブな更新を行っているものと考えられる（β の調整で改善する可能性がある）．

次に，アクティブな更新を行うノードが分散しているかどうかを確認するため，上記と同じ設定で $1000t (= 20000)$ 秒の間に Chord## の各ノードがアクティブな更新を行なう回数を測定した（転送回数 $s = 4$ とした）．これを 10 回試行し，その平均値を**図 6** に示す．横軸はアクティブな更新回数，縦軸はノード数である．

グラフより，アクティブな更新を行うノードはほぼ均等に分散していることがわかる．なお，理想的には，測定期間内に各ノードが finger table を更新する回数は $20000/t = 1000$ であり，これを $s+1$ 個のノードが分担してアクティブな更新を行うため，ノードあたりの

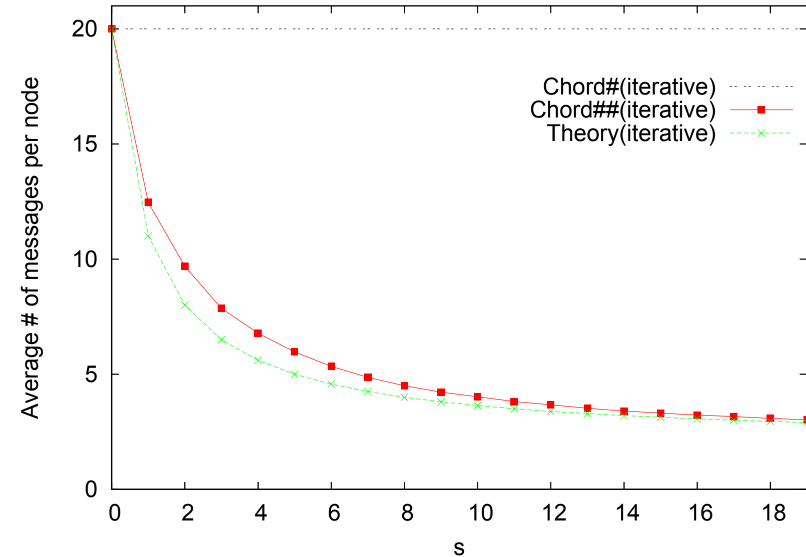

図 5 s を変化させたときの finger table の更新に要する 1 ノードあたりの平均メッセージ数

アクティブな更新回数は $1000/(s+1) = 200$ 回となる．測定結果はこれよりも若干大きいが，これも上で述べた理由によるものと考えられる．

5. お わ り に

本稿では構造化 P2P ネットワークの 1 つである Chord# の finger table を更新するコストを削減する手法（Chord##）を提案した．finger table の更新処理は各ノードが定期的に行う必要があるため，このコストを削減することは重要である．提案手法では，あるノードが更新した finger table を他のノードも利用することで，finger table の更新にかかるメッセージ数を数分の一に削減できる．その効果はシミュレーションによって確認した．また，提案手法では 2 次元に拡張した 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についても考慮している．

今後の課題としては，ノードの参加離脱 (Churn) による影響の解析，ネットワーク近接性を考慮したルーティングの有効性を実ネットワークに近いネットワークトポロジで評価す

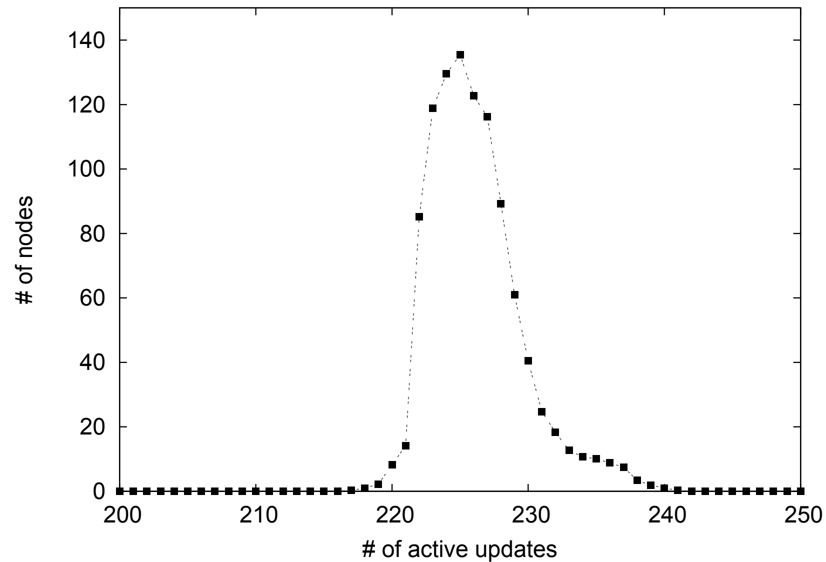

図 6 ノードあたりのアクティブな更新回数の分布

ることなどが挙げられる.

参 考 文 献

1) Stoica, I., Morris, R., Karger, D., Kaashoek, M.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 (2001).
2) Rowstron, A. and Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems, *Lecture Notes in Computer Science*, Vol.2218, pp.329–350 (2001).
3) 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).
4) Schütt, T., Schintke, F. and Reinefeld, A.: Range queries on structured overlay networks, *Computer Communications*, Vol.31, pp.280–291 (2008).
5) Dabek, F., Cox, R., Kaashoek, F. and Morris, R.: Vivaldi: A Decentralized Network Coordinate System, *Proceedings of the ACM SIGCOMM '04 Conference*, Portland, Oregon (2004).