構造化オーバーレイネットワークを用いた条件付きマルチキャストの提案

安倍 広多[1]

概要：構造化オーバーレイネットワークによって，指定した条件を満たすノードに選択的にメッセージを配送する方法を提案する（**条件付きマルチキャスト**と呼ぶ）．各ノード u はユニークなキー (u.key) およびノードの状態を表す値 (u.value) を保持する．条件付きマルチキャストでメッセージを配送するには，配送先のキー範囲 r および条件を指定する関数 $match$ を指定する．メッセージは $(p.\text{key} \in r) \wedge (\text{match}(p.\text{value}) = \text{true})$ を満たすすべてのノード p に配送される．提案手法は Chord$^{\#}$ をベースとした構造を持つ．各ノードが経路表の各エントリに，あるキー区間のノードの value を集約した値を保持することで，当該区間内に条件にマッチするノードが存在する可能性がある場合のみメッセージを配送する．経路表が収束している場合，条件付きマルチキャストに必要な最大ホップ数は $\lceil \log_2 n \rceil$ である（n はノード数）．提案手法には集約値を含む経路表を効率よく収集するアルゴリズムも含む．本稿では提案手法の詳細と性質，応用例などについて述べる．

Proposal of "Conditional Multicasting" using Structured Overlay Networks

KOTA ABE[1]

1. はじめに

構造化オーバーレイネットワークは，自律的に動作するノードが協調して動作することにより，キーにより識別されるノードに効率よくメッセージを配送することが可能なネットワークである．構造化オーバーレイネットワークの中でも，キーの順序関係を保存するもの（2 つのノードのキーが隣接するとき，オーバーレイネットワーク上でも隣接する）はキー順序保存型構造化オーバーレイネットワーク（Key-order preserving structured overlay network）と呼ばれる（以下 KOPSON）．KOPSON はキーの範囲を指定したアプリケーションレベルマルチキャスト（ALM）を効率的に実現できるため，範囲検索，分散 Pub/Sub システム [1]，オンラインゲーム [2] などのさまざまな応用がある．

しかし，ALM は，例えば「ネットワーク上の多数のノードの中で，リソース（CPU 負荷やディスク残量など）やセンサーの値が一定範囲のノードに対してメッセージを配送する」，といったことは得意ではない（これらの値をキーとすると，値が変動するたびにノードの挿入・削除を行う必要があるため）．

本稿では ALM を拡張し，指定したキーの範囲で，指定した条件を満たすノードにのみメッセージを効率よく配送する方法を提案する．このようなマルチキャストを**条件付きマルチキャスト**と呼ぶことにする．

提案手法では，各ノードはキーに加え，自ノードの状態を表す値（$value$）を保持する．条件付きマルチキャストでは，配送先のキー範囲 $[\min, \max)$ に加え，配送先ノードの条件を指定する関数 $match$ を指定する．メッセージは，ノードのキーが $[\min, \max)$ に含まれ，かつ $\text{match}(value) = \text{true}$ であるノードに配送される．

提案手法は KOPSON の 1 つである Chord$^{\#}$ をベースとしている．Chord$^{\#}$ の経路表（finger table）の経路表エントリを拡張し，各エントリが（エントリごとに異なる）キー範囲内に含まれるノードの value を集約した**集約値**を保持する．集約値は当該範囲内に match の条件を満たすノード

[1] 大阪市立大学大学院工学研究科
Graduate School of Engineering, Osaka City Univerity

が存在するかを判定するために用いる．マルチキャストの際，集約値を参照することで，条件を満たすノードが存在する範囲だけにマルチキャスト木を絞る．これにより効率的なマルチキャストを実現する．

一般的な構造化オーバーレイネットワークでは，メッセージの宛先ノードはキーの大小関係のみによって決まるが，本手法では宛先ノードは（キーの大小関係に加えて）各ノードの value と match 関数によって決まる．value は配列などの任意のオブジェクトを利用でき，match 関数の定義も任意であるため，宛先ノードを柔軟に選ぶことができる．

提案方式では，各ノードが保持する集約値が古いと配送もれが発生する可能性がある．このため効率よく集約値を更新する手法（連続的更新手法）も考案した．

本稿の構成は以下のとおりである．2 章で提案システムのベースとなる Chord$^{\#}$ とアプリケーションレベルマルチキャストのアルゴリズム SFB について簡単に説明する．3 章と 4 章で提案手法とその応用例について述べる．5 章で関連研究を述べ，6 章でまとめと今後の課題を述べる．

2. 準備

提案手法のベースとなる Chord$^{\#}$[3] およびアプリケーションレベルマルチキャストのアルゴリズム SFB について簡単に述べる．

2.1 Chord$^{\#}$

Chord$^{\#}$ はリングベースの構造化オーバーレイネットワークである．

2.1.1 構造

Chord$^{\#}$ の各ノードは全順序集合の要素である**キー**を保持する．各ノードはキーが次に大きいノードへのポインタ（successor）と，キーが次に小さいノードへのポインタ（predecessor）を持つ．ただし，最大値のキーを持つノードの successor は最小値のキーを持つノードを，最小値のキーを持つノードの predecessor は最大値のキーを持つノードを指す．これにより全体としてキーの順にソートされた双方向リングを構成する．本稿ではキーの増加方向を右方向とする．ノードの挿入・削除があると，関係するノードの successor と predecessor を更新する必要があるが，このために Chord$^{\#}$ では Chord[4] のスタビライズアルゴリズムを用いる．

高速化のため，各ノードはショートカットリンクの配列（finger table）を保持する．finger table の各要素を *finger* と呼び，i 番目の finger を finger[i] と表記する．finger[0] = successor である．なお，finger にはポインタとキーの両方が含まれるが，簡潔にするため本稿ではこれらを区別しないで扱う（finger[i] をポインタとしてもキーとしても扱う）．

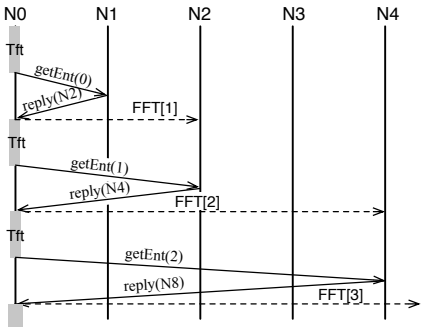

図 1 Chord$^{\#}$ の finger table 更新シーケンス（ノード $N0$）

以下，リング上でノード u から右方向（あるいは左方向）に k 個離れたノードを u_{+k}（あるいは u_{-k}）と表記する．

2.1.2 Finger Table の更新

ノード u の変数 x に，ノード p の y を代入することを以下のように書く（y は p における変数または式）．

$$u.x \Leftarrow p \rightarrow y$$

ノード u の finger[i] $(i > 0)$ は，次式に基づいて更新する:

$$u.\text{finger}[i] \Leftarrow u.\text{finger}[i-1] \rightarrow \text{finger}[i-1].$$

すなわち，u.finger[$i-1$] の指すノード（p とする）から finger[$i-1$] を取得し（q とする），u.finger[i] に q を代入する．

更新は一定周期（T_{ft} とする）で行う．すなわち，u はまず u.finger[1] を更新し，T_{ft} 時間待機した後に u.finger[2] を更新し... というように更新する（**図 1**）．ただし，$q \in [u, p)$ の場合は更新が一周したと判断し，次の周期でレベル 1 の更新に戻る．

ノードの挿入や削除がなければ，ノード u の finger[i] はいずれ u_{+2^i} に収束する．このとき，検索ホップ数の上限と u が保持する finger table のエントリ数はどちらも $\lceil \log_2 n \rceil$ である（n はノード数）．

2.2 SFB

SFB(Split-Forward Broadcasting)[5] は，Skip Graph 上でアプリケーションレベルマルチキャストを行うための単純で効率が良いアルゴリズムである．Chord$^{\#}$ でも利用できる．

SFB の例を**図 2** に示す．ここでは Chord$^{\#}$ で各ノードの Finger Table が収束している場合に，ノード N0 がキー範囲 $[20, 50)$ に対してマルチキャストを行った場合を示している．横線上の黒丸は各ノードの finger が指すノードを示している．N0 は範囲 $[20, 50)$ を finger table エントリによって部分範囲に分割し，次に各部分範囲の処理を，範囲の左端に最も近いノード（実際には左端のキーを超えない最大のキーを持つノード）に委譲する．委譲されたノード

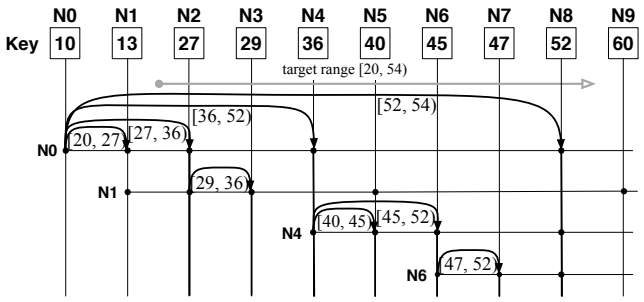

図 2 SFB によるマルチキャストの例

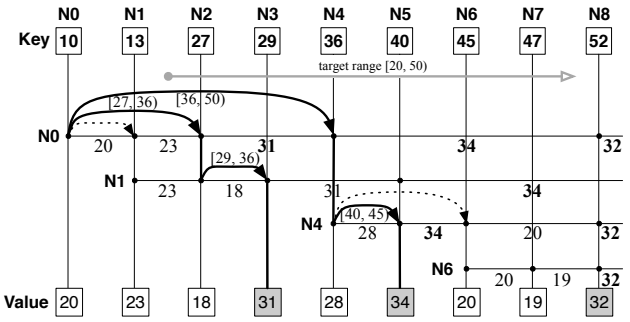

図 3 条件付きマルチキャストの例

も同じ処理を繰り返す．また，各ノードは，自ノードが範囲内であればメッセージをアプリケーションに渡す（太い縦線で示している）．

3. 提案手法

3.1 考え方

各ノード u はキー u.key とは別に任意の値 u.value を持つ．マルチキャストする際は，配送先のキー範囲 r および関数 *match* を指定する．メッセージは $(p.\text{key} \in r) \wedge (\text{match}(p.\text{value}) = \text{true})$ を満たすすべてのノード p に配送する．

単純に実現する場合，SFB によって r 内のすべてのノードにメッセージを配送し，各ノードにおいて match 条件を満たすときのみメッセージをアプリケーション側に渡す方法が考えられるが，この方法は match 条件を満たすか否かに関わらず範囲内のすべてのノードにメッセージが配送されるため効率が悪い．

提案手法では，SFB と同じ方法で分割された各部分範囲に対して，範囲内に match 条件を満たすノードが存在する可能性がある場合のみ当該範囲のマルチキャスト処理を他のノードに委譲する．

このとき，範囲内に match 条件を満たすノードが存在するかどうかの判定方法が問題となる．各ノードが，各部分範囲内のすべてのノードの value を保持していれば判定可能であるが，ノード数が多い場合には非現実的である．このため，提案手法では部分範囲内のすべてのノードの value を 1 つの値に集約して保持する．match の条件判定にはこの値を用いる．複数の値を集約するための関数 (*reduce*) は（match に合わせて）アプリケーション側で定義する．

例を**図 3** に示す．各ノードの value は下部の四角内に，各部分範囲で保持する集約値は範囲の下に示している．例えば，N0 は範囲 $[10, 13)$ に対して集約値 20 を，範囲 $[13, 27)$ に対して集約値 23 を保持している．ここでは集約値は範囲内のノードの value の最大値としている．最大値を集約すると，match 条件として「value が任意の値以上のノード」を使用できる．

図では N0 から，key が 20 以上 50 未満で，value が 30 以上のノードへ条件付きマルチキャストを行った場合のメッセージの流れも示している（実線の黒い矢印）．集約値が 30 未満の範囲には目的ノードが存在しないため，マルチキャスト対象から除く（30 以上の集約値は太字にしてある）．N0 の場合，範囲 $[13, 27)$ の集約値 20 は 30 未満であるため，マルチキャスト対象から除く（点線の矢印）．一方，範囲 $[27, 36)$ および $[36, 50)$ の集約値はいずれも 30 より大きいため，この範囲のマルチキャスト処理は継続する．

提案手法では，アプリケーション開発者が match 関数と reduce 関数を定義することで柔軟にマルチキャストの目的ノードを選択できる．以下，提案手法の詳細について述べる．

3.2 ノードの状態を表す値

前述のように，各ノード u はキー (u.key) に加えて自ノードの状態を表す値 (u.value) を持つ．ノードがネットワークに参加している間，key は変化してはならないが，value は変化してもよい．以下，value の型を V とする．

3.3 match と reduce

match と reduce について述べる．これらの具体的な定義はアプリケーション開発者が決める．

match(v)　条件付きマルチキャストの配送先を指定するために用いる．引数の value が配送先としての条件を満たすときは true を，満たさないときは false を返す．

reduce(v_1, v_2)　2 つの V 型の value を集約して 1 つの V 型の値として返す．あるキー区間内のノードの value を 1 つの値に集約するために用いる．

match と reduce は以下の制約を満たす必要がある．

$$\text{match}(v_1) \vee \text{match}(v_2) \Rightarrow \text{match}(\text{reduce}(v_1, v_2)). \quad (1)$$

すなわち，v_1 と v_2 の少なくとも一方が match の条件を満たすならば，reduce した値も条件を満たす．また，引数が 2 つ以外の場合の reduce を以下のように定義する．

$$\text{reduce}(v) \equiv v$$
$$\text{reduce}(v_1, \ldots, v_n) \equiv \text{reduce}(\text{reduce}(v_1, \ldots, v_{n-1}), s_n) \quad (n > 2)$$

3.4 Finger Table エントリの拡張

Chord$^{\#}$ の finger[i] はノードへのポインタとキーを保持するが，提案手法では，これに加えて，キーの範囲と，その範囲内のノードの value を集約した値を保持する．このため，提案手法では finger[i] を構造体とし，ノードへのポインタとキー（Chord$^{\#}$ における finger[i]）を finger[i].node，キーの範囲を finger[i].range，value の集約値を finger[i].value に格納するものとする．

ここで範囲は $[a, b)$ という形式であり，環状キー空間においてキーの昇順方向に a から b までの範囲を表す．ただし a は含み，b は含まない．また，$[a, a)$ はすべての範囲を表す．さらに，範囲 $r = [a, b)$ のとき，$r_{\min} = a$，$r_{\max} = b$ とする．

Chord$^{\#}$ の Finger Table はインデックス 0 から始まるが，提案方式では -1 から始める．任意のノード u について，u.finger[-1].node $= u$，u.finger[-1].range $= [u, u.\text{successor})$，$u$.finger[$-1$].value $= u$.value とする．また，Chord$^{\#}$ と同様，u.finger[0].node $= u$.successor とする．

3.5 successor と predecessor の更新

ノードの挿入・削除のために Chord$^{\#}$ が採用している Chord のスタビライズアルゴリズムは収束に時間がかかることが知られている [6]．提案手法ではこれらのポインタが（可能な限り）常に正しいノードを指していることを期待するため，DDLL[6] のような，ポインタをアクティブに更新するアルゴリズムを使用する．

DDLL では，(1) 各ノードの successor は常に正しい，(2)predecessor は，対応する（反対方向の）successor が更新された後，1 片方向遅延時間で正しいノードを指す，(3) 左右両方向で正しく（挿入済みのノードをスキップすることなく）トラバースが可能，といった性質を備える．提案手法はこれらの特徴を前提とする．

3.6 Finger Table の更新（1 レベル）

ノード u の u.finger[i] ($i \geq 0$) の更新方法を述べる．

まず，u.finger[i].node ($i > 0$) は，Chord$^{\#}$ の finger table 更新アルゴリズムと同じ方法で更新する．すなわち，

$$u.\text{finger}[i].\text{node} \Leftarrow u.\text{finger}[i-1].\text{node} \rightarrow \text{finger}[i-1].\text{node}. \tag{2}$$

また，u.finger[i].value および u.finger[i].range は，以下の方法により更新する ($i \geq 0$)．（ただし初期値は null とする）．

$$u.\text{finger}[i].\text{value} \Leftarrow u.\text{finger}[i].\text{node} \rightarrow \text{reduce}(\text{finger}[-1].\text{value}, \ldots, \text{finger}[i-1].\text{value}) \tag{3}$$

```
1  // i: level to update (i >= 1)
2  u.updateFinger(i) {
3    p ← finger[i-1].node
4    (node, range, val) ← p.getEnt(i-1, u)
5    if (node = null ∨
6      node ∈ [u, finger[i-1].node]) { // circulated
7      truncate finger so that finger[i-1] is
8        the highest entry
9    } else {
10     finger[i].node = node
11   }
12   finger[i-1].range = range
13   finger[i-1].value = val
14 }
15 u.getEnt(i, h) {
16   if (finger[i] does not exist) {
17     return (null, null, null)
18   }
19   val ← finger[-1].value
20   j = 0
21   // optimized:
22   //   j < i ∧ finger[j].node ∈ [u, h)
23   for (; j < i; j++) {
24     val ← reduce(val, finger[j].value)
25   }
26   range ← [u, finger[j-1].range_max)
27   return (finger[i].node, range, val)
28 }
```

図 4　Finger Table 更新アルゴリズム（1 レベルのみ）

$$u.\text{finger}[i].\text{range} \Leftarrow u.\text{finger}[i].\text{node} \rightarrow [\text{key}, \text{finger}[i-1].\text{range}_{\max}). \tag{4}$$

アルゴリズムを図 4 に示す．u が u.finger[i].node を更新する契機で u.finger[$i-1$].value と u.finger[$i-1$].range を更新する．

各ノードは，updateFinger(1) から順番に updateFinger(2)，updateFinger(3)... と実行し，一周すると updateFinger(1) の実行に戻る（詳しくは 3.9 節で議論する）．

finger[i].node の更新は Chord$^{\#}$ と同じ方法で行うため，いずれ任意のノード u に対して u.finger[i].node $= u_{+2^i}$ となる．

図 5 はノード $N0, N1, \ldots$ があるとき，$N0$〜$N4$ が保持する finger[i].range の範囲と，その集約の流れを示している．水平方向の矢印は各ノードの finger[i].range を示す．ただし，矢印の根元は端点を含み，先端は端点を含まない．また，N0 から伸びている実線の矢印は $N0$.finger[i].node を示している．なお，ここではすべてのノードの finger table が収束した状況を示している．

まず，

$$N0.\text{finger}[0].\text{node} = N0.\text{successor} = N1$$

である．また，$N0$ が finger[0].value と finger[0].range を更新すると，式 3 および式 4 より

$$\begin{aligned} N0.\text{finger}[0].\text{value} &= \text{reduce}(N1.\text{finger}[-1].\text{value}) \\ &= N1.\text{value} \\ N0.\text{finger}[0].\text{range} &= [N1, N2) \end{aligned}$$

となる．また，$N0$.finger[1].node $= N2$ であるから，（N2 と N3 が finger[0].value と finger[0].range を更新した後で）$N0$ が finger[1].value と finger[1].range を更新すると，

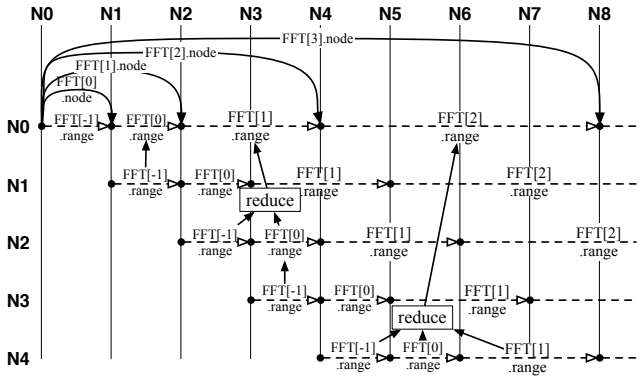

図 5 データ構造と集約の流れ

$$N0.\text{finger}[1].\text{value} = \text{reduce}(N2.\text{finger}[-1].\text{value}, N2.\text{finger}[0].\text{value}) = \text{reduce}(N2.\text{value}, N3.\text{value})$$

$$N0.\text{finger}[1].\text{range} = [N2, N4)$$

となる．

各ノードが finger table の更新を繰り返すことで，いずれ任意のノード u の finger table は以下の状態に収束する．

$$u.\text{finger}[i].\text{node} = u_{+2^i} \tag{5}$$

$$u.\text{finger}[i].\text{value} = \text{reduce}(u_{+2^i}.\text{value}, u_{+(2^i+1)}.\text{value}, \ldots, u_{+(2^{i+1}-1)}.\text{value}) \tag{6}$$

$$u.\text{finger}[i].\text{range} = [u_{+2^i}, u_{+2^{i+1}}). \tag{7}$$

つまり，u.finger[i].value は，範囲 u.finger[i].range に含まれるすべてのノードの value の集約値である．

図 4 のアルゴリズムはノード数がちょうど 2 の累乗ではない場合，最後のエントリで本来不要な範囲の集約値まで追加してしまう．例えばノード数が 11 で（$N0$〜$N10$），各ノードの finger table は収束しているとき，$N0$ が $N8$ に対して getEnt($i = 3$) を要求した場合，N8 が返す集約値の範囲は $[N8, N5)$ である必要はなく（$N5 = N8_{+8}$），$[N8, N1)$ で十分である（$N1 = N8_{+4}$）．getEnt 中のコメントアウトされている条件に変更することで改善できる．

3.7 ノードの挿入

提案手法では各ノードが finger table エントリに集約値を保持している必要があるため，ノードの挿入時には左ノードから finger table をコピーする．また，3.9.2 節で述べる連続的更新方式を用いる場合は右ノードからの update メッセージのタイムアウトをチェックするためのタイマーをスタートする．

3.8 条件付きマルチキャスト

条件付きマルチキャストのアルゴリズムを**図 6** に示す．条件付きマルチキャストを行うノード u は condcast(r, match) を実行する．u は必要に応じて他のノードにメッセージを送信し，condcast を再帰的に実行する．なお，ここでは単純のため r は [min, max) 形式に限定する．

```
1  // r: target range in the form of [min, max)
2  // match: predicate function
3  u.condcast(r, match) {
4    // N: set of ranges that does not contain
5    // any matching node
6    N ← {e.range | (e ∈ ∀u.finger) ∧ (e.range ≠ null) ∧
7          ¬match(e.value)}
8    // S: set of ranges within r that may contain
9    // matching nodes
10   S ← r − N
11   // T: split each element in S by finger table
12   // entries
13   T ← {split(s) | s ∈ S}
14   for t ∈ T {
15     p ← closest_preceding_node(t_min)
16     if (p = u) {
17       if (match(u.key)) {
18         // receive the message (send to app)
19       }
20     } else {
21       p.condcast(t, match)
22     }
23   }
24 }
25
26 // split a range with finger[].node.
27 // returns a set of ranges.
28 u.split(r) {
29   R ← ∅ // set of ranges
30   E ← all u.finger[].node sorted in the order of
31     clockwise distance from u
32   for e ∈ E {
33     if ((e.node ∈ r) ∧ (r_min ≠ e.node)) {
34       R ← R ∪ [r_min, e.node)
35       r ← [e.node, r_max)
36     }
37   }
38   R ← R ∪ r
39   return R
40 }
41
42 // find the closest node in finger[].node.
43 // returns a pointer to a node
44 u.closest_preceding_node(k) {
45   p ← the most rightward distant node from u in
46     {n | n ∈ finger[].node ∧ n ∈ [u.key, k]}
47   return p
48 }
49 }
```

図 6 条件付きマルチキャストのアルゴリズム

condcast において，N は，u が保持するすべての finger の範囲のうち，match の条件を満たさないものの集合である．また，S は r から N のすべての範囲を差し引いて残った範囲の集合であり，r の中で match の条件を満たすノードが存在する可能性がある範囲を示している．さらに T は，S の各要素をさらに finger[].node で分割したものである（分割できなければそのまま）．アルゴリズムは T の各要素（範囲）に対し，範囲の左端に最も近いノードに範囲の処理を委譲する．

図 3 の例において，N0 が condcast を実行したときの様子を**図 7** に示す．

finger table が収束しているとき，このアルゴリズムはすべての目的ノードに $\lceil \log_2 n \rceil$ ホップで配送できる．なお，メッセージ数は目的ノードの数に依存する．

なお，実際の実装では match 関数は各ノードで予め定義しておき，match 関数への引数をネットワークで転送してもよい．

3.9 Finger Table の更新（全体）

finger table の 1 エントリの更新方法は 3.6 節で述べた．

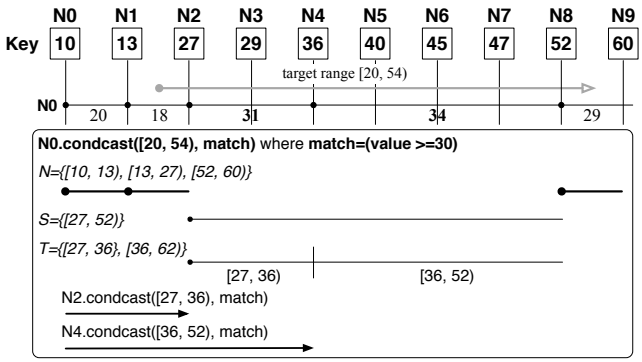

図 7 N0 における条件付きマルチキャストの実行

ここでは，finger table 全体の更新方法について述べる．

まず，Chord$^{\#}$ の finger table エントリ更新と同じタイミングで更新する**離散的更新方式**について述べ，次により短時間で集約値を収集できる**連続的更新方式**について述べる．

3.9.1 離散的更新方式

この方式では，Chord$^{\#}$ の finger 更新のタイミング（2.1 節参照）をそのまま用いて finger table を更新する．すなわち，各ノードは T_{ft} 時間のインターバルで updateFinger(1), updateFinger(2), ... を実行する．ノード間で updateFinger を実行するタイミングはばらばらである．

ここで，新規ノード q が挿入，もしくは既存ノード q の value が変更された時刻を $t = 0$ とし，以後ノードの挿入や削除，value の変化がない場合を想定し，すべてのノードの finger table が収束する時間を考える（収束すればすべてのノードから q へマルチキャスト可能）．

すべての FFT エントリの更新に要する時間を T_{all} とすると，$T_{all} = T_{ft}\lceil \log_2 n \rceil$ である（T_{ft} に比べてメッセージ遅延時間は非常に小さいため，getEnt の実行時間は無視している）．

各ノードは $t = T_{all}$ までに 1 回は finger[0].value（と range，以下省略）の更新を行うため，この時点ですべてのノードの finger[0].value は正しい値に更新される（finger[−1].value は常に正しいことに注意）．同様に，$t = 2T_{all}$ ですべてのノードの finger[1].value は正しい値に更新される．このように考えると，ノードの Finger Table エントリは $\lceil \log_2 n \rceil$ 個あるので，結局すべてのノードの finger[].value が正しい値に収束するための最大時間は，

$$T_{all}\lceil \log_2 n \rceil = T_{ft}(\lceil \log_2 n \rceil)^2 \qquad (8)$$

となる（すなわち $O((\log_2 n)^2)$）．

3.9.2 連続的更新方式

以下のように各ノードの finger table 更新タイミングを制御することにより，少ないメッセージ数で finger table を収束させることができる．

- ノード u が finger table を更新する際，（1 レベルごとに T_{ft} 待つのではなく）レベル 1 から最大レベルまでの更新を一気に行う．

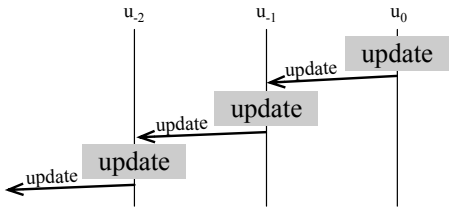

図 8 連続的更新方式の更新フロー

- その後，u の左ノードが同様に（レベル 1 から最大レベルまで）finger table を更新する．

流れを**図 8** に示す．灰色の四角は finger table のレベル 1 から最大レベルまでの更新処理を表している．更新が終了すると，左ノードに update メッセージを送信し，同じ処理を開始する．この更新の流れを**更新フロー**と呼ぶことにする．更新フローはリングをぐるぐると周回する．

3.9.2.1 連続的更新方式の性質

連続的更新方式では，以下の性質が成り立つ．

定理 1. u の左ノード u_{-1} から更新フローを開始したとき，u_{-k} が更新を終えた時点で，

$$u_{-k}.\text{finger}[i].\text{node} = u_{-k+2^i} \quad (i \le \lfloor \log_2 k \rfloor).$$

証明. $k = 1$ で定理は成立する．以下，定理が $k < 2^g$ で成立すると仮定し，$2^g \le k < 2^{g+1}$ でも成立することを示す（帰納法）．

このとき，$\lfloor \log_2 k \rfloor = g$ であるため，$u_{-k}.\text{finger}[g].\text{node} = u_{-k+2^g}$ を示せばよい（finger[0] から finger[$g-1$] までは，更新フローの開始位置を u_0 から適当に左にずらせば正しいことが分かる）．

式 2 より，$u_{-k}.\text{finger}[g].\text{node} \Leftarrow u_{-k}.\text{finger}[g-1].\text{node} \rightarrow \text{finger}[g-1].\text{node}$ であるが，$u_{-k}.\text{finger}[g-1].\text{node} = u_{-k+2^{g-1}}$ である．$k' = k - 2^{g-1}$ とすると，$k' < 2^g$ であり，仮定から $u_{-k'}.\text{finger}[g-1].\text{node} = u_{-k'+2^{g-1}}$ である．このため，$u_{-k}.\text{finger}[g].\text{node} = u_{-k'}.\text{finger}[g-1].\text{node} = u_{-k'+2^{g-1}} = u_{-k+2^g}$ となる． □

定理 2. u の左ノード u_{-1} から更新フローを開始したとき，u_{-k} が更新を終えた時点で，$u_{-k}.\text{finger}[i].\text{value}$ は $u.\text{value}$ を集約値の元に含む（$i = \lfloor \log_2 k \rfloor$）．

証明. $k = 1$ で定理は成立する．以下，定理が $k < 2^g$ で成立すると仮定し，$2^g \le k < 2^{g+1}$ でも成立することを示す（帰納法）．

このとき，$\lfloor \log_2 k \rfloor = g$ であるため，$u_{-k}.\text{finger}[g].\text{value}$ に $u.\text{value}$ を集約値の元に含むことを示す．

式 3 より $u_{-k}.\text{finger}[g].\text{value} \Leftarrow u.\text{finger}[g].\text{node} \rightarrow \text{reduce}(\text{finger}[-1].\text{value}, \ldots, \text{finger}[g-1].\text{value})$ である．定理 1 より，$u_{-k}.\text{finger}[g].\text{node} = u_{-k+2^g}$ である．$k' = k - 2^{g-1}$ とすると，$k' < 2^g$ であり，仮定から $u_{-k'}.\text{finger}[g-1].\text{value}$ は $u.\text{value}$ を集約値の元に含む．このため，

reduce(finger[−1].value, . . . , finger[$g-1$].value) は u.value を集約値の元に含む．□

定理 2 から，ノード q が挿入された，もしくは既存のノード q の value が変更されたとき，q の左ノードから更新フローがリングを一周すれば，すべてのノードの finger table に q.value を集約した値が含まれることになる．また，任意の状態から収束させるには更新フローが 2 周すれば良い．

離散的更新方式では収束までに各ノードあたりに要するメッセージ数は $O((\log_2 n)^2)$ であったのに対し，連続的更新方式では $O(\log_2 n)$ である（update を 2 回実行するのに要するメッセージ数）．

3.9.2.2 連続的更新方式のアルゴリズム

連続的更新方式のアルゴリズムは，以下の性質を備えることが望ましい．

(1) 各ノードが update を実行する周期（更新周期）を設定可能とすること（短すぎるとオーバヘッドが大きく，長すぎると新規ノードへ条件付きマルチキャスト可能になる時間が遅れる）．
(2) ノード数の増減に対応する．
(3) ノード数によっては更新フローが一周する時間が目標周期より大きくなることがある．この場合は別の更新フローを自律的にスタートする．
(4) 更新フローが過剰な場合は削減する．
(5) ノード障害などで更新フローが中断した場合，自律的に更新フローをリスタートする．

これらを実現する分散アルゴリズムを考案した（ただし (4) は実現できておらず，今後の課題である）．アルゴリズムは以下の考えに基づく．ここでは，注目するノードを u，その predesessor を p，u が successor から update メッセージを受信した時刻を r，u が p に update メッセージを送信する時刻を s，u が前回 p に update メッセージを送信した時刻を last，目標とする更新周期を PERIOD とする．また，参考図を**図 9** に示す．

- u が長時間（前回 update メッセージを受信してから PERIOD + GRACE 時間）update メッセージを受信しない場合，タイムアウトとして u が更新フローを開始する．
- u でタイムアウトが発生したときに p でもタイムアウトが発生すると p でも更新フローが生成されてしまう．u が（タイムアウト後に）更新フローを開始して update メッセージを p に送信したときに p ではタイムアウトしないようにするために，p のタイムアウト時刻を u のそれよりもある程度遅らせる必要がある．このため，$s - r \geq$ MINDELAY となるようにする（MINDELAY は最低限の遅らせる時間）．
- 周期を保つ観点からすると，s = last + PERIOD が望ましいが，更新フローの連続性を考えると，$s = r +$ MINDELAY が望ましい．双方を一定の割合で満足させるため，

$$s = \alpha(\text{last} + \text{PERIOD}) + (1-\alpha)(r + \text{MINDELAY}) \tag{9}$$

とする（α は $0 \leq \alpha \leq 1$ を満たすパラメータ）．

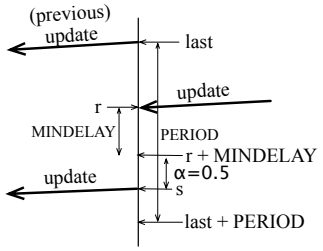

図 9 update 送信タイミング

```
1  last = 0
2  u.update(is_timeout) {
3    r = current_time()
4    for (i = 1;; i++) {
5      updateFinger(i)
6      break if circulated
7    }
8    if (¬is_timeout) {
9      if (last = 0 ∨ last + PERIOD < r + DELAY) {
10       s = r + MINDELAY
11     } else {
12       s = α(last + PERIOD) + (1 − α)(r + DELAY)
13     }
14     diff = s − current_time()
15     if (diff > 0) sleep(diff)
16   }
17   last = current_time()
18   predecessor.update(false)
19   schedule a timer to call u.update(true)
20     if u does not receive an update request
21     until (r + PERIOD + GRACE).
22 }
```

図 10 連続的更新方式のアルゴリズム

アルゴリズムを**図 10** に示す．タイムアウトによって update() を実行する場合は MINDELAY 時間待たない．また，sleep 実行中に受信した update メッセージは無視する．

ノード数やネットワーク遅延時間の変動によって update 受信のタイミングは変動するため，GRACE はある程度大きく取る必要がある．また，u がタイムアウトしたとき，u はすべてのレベルの finger table を更新してから update メッセージを p に送信するが，p がこの update メッセージを p のタイムアウトより前に受信するための条件は以下のとおり．

$$\text{MINDELAY} + T_{\text{oneway}} > T_u \tag{10}$$

ただし，T_u はすべてのレベルの finger table 更新にかかる時間，T_{oneway} はメッセージ片方向遅延時間である．

3.9.2.3 シミュレーション

連続的更新方式の簡単なシミュレーションを行った．ノードは N0〜N7 の 8 ノードで固定とした．更新フローは N7→N6→N5. . . の方向に流れる．PERIOD=30,000, MINDELAY=1,500，GRACE=15,000，$\alpha = 0.5$，FFT 更新時

間=1,000，ノード間の片方向メッセージ遅延時間=20 とした（単位 msec）．

N7 から更新フローを開始したときの，(T1) 更新フロー上のあるノードと次のノードの update メッセージ送信時刻の差と，(T2) 各ノードの update メッセージ送信の間隔のグラフを**図 11** と**図 12** に示す．update 実行を繰り返すことで T1 は約 3670，T2 は約 29,400 に近づく（$3670 \times 8 = 29360 \approx 29400$）．

また，複数の更新フローができた場合を想定し，N7 と N6 から同時に更新フローを開始した場合のグラフを**図 13** と**図 14** に示す．このとき，T1 は約 6,620，T2 は約 26,400 に近づく（$\frac{1}{2}(6620 \times 8) = 26480 \approx 26400$）．

連続的更新方式では，全体のノード数や更新フローの数がわからなくても各ノードの update 送信タイミングを調整することで，PERIOD に近い周期に収束することが確認できる．なお，更新フローの数が増えると，それに応じて更新フローが一周する時間が伸びる．また，ある更新フローにおける finger table 更新が別の更新フローにおける finge table 更新の影響を受ける．これらの詳細な分析は今後の課題である．

なお，$\alpha = 0$ もしくは $\alpha = 1$ の場合，T1 と T2 のいずれかが収束しない（N7 と N6 から更新フローを開始した場合の例を**図 15**，**図 16**，**図 17**，**図 18** に示す）．

4. 提案手法の応用

提案手法の応用例をいくつか挙げる．

4.1 value が一定値以上のノードへのマルチキャスト

ノード u が value としてスカラー値 $u.c$ を保持していて，value が C 以上のノードへマルチキャストする場合は以下のようにする．

- V: スカラー値
- u.value $= u.c$
- match$(v) \equiv v > C$
- reduce$(v_1, v_2) \equiv \max(v_1, v_2)$

4.2 1 次元範囲マルチキャスト

ノード u が value としてスカラー値 $u.c$ を保持しているときに，value がある 1 次元の範囲 $R = [R_{\min}, R_{\max}]$ 内のノードへマルチキャストする場合は以下のようにする．

- V: 1 次元の範囲
- u.value $= [u.c, u.c]$
- match$(v) \equiv (v \cap R \neq \emptyset)$
- reduce$(v_1, v_2) \equiv$ merge(v_1, v_2)

ただし，merge(x, y) は 2 つの 1 次元範囲 x, y に対して，x と y の両方を包含した 1 つの範囲を返す関数である．x，y のどちらも閉区間ならば，

$$\text{merge}(x, y) \equiv [\min(x_{\min}, y_{\min}), \max(x_{\max}, y_{\max})].$$

4.3 多次元範囲マルチキャスト

4.2 節の方法は n 次元に拡張できる．ノード u が n 次元のベクトル $u.p = (u.p_1, \ldots, u.p_n)$ を保持しているときに，n 次元空間における範囲 $R = [a, b] = \{(x_1, \ldots, x_n) \mid a_1 \leq x_1 \leq b_1, \ldots, a_n \leq x_n \leq b_n\}$ 内のノードにマルチキャストする場合，

- V: n 次元の範囲（超直方体）
- u.value: $[u.p, u.p]$
- match$(v) \equiv (v \cap R \neq \emptyset)$
- reduce$(v_1, v_2) \equiv$ merge(v_1, v_2)

とする．ただし merge は，2 つの n 次元範囲を包含した 1 つの n 次元範囲を返す関数である．x，y のどちらも閉区間ならば，

$$\text{merge}(x, y) \equiv [X, Y] \quad \text{where}$$
$$X = (\min(x_{\min_1}, y_{\min_1}), \ldots, \min(x_{\min_n}, y_{\min_n})),$$
$$Y = (\max(x_{\max_1}, y_{\max_1}), \ldots, \max(x_{\max_n}, y_{\max_n}))$$

なお，ノードの key に $u.p$ の 1 次元を割り当て，u.value に残りの $n-1$ 次元を割り当ててもよい．

多次元範囲マルチキャストは，例えばセンサーノードが 2 次元平面上に分布しているときに，指定した 2 次元範囲内でセンサー値が特定の範囲内のノードにマルチキャストしたい場合や，ノードが複数の属性値を持っているときに，複数の属性値に関する AND 検索を行う場合などで利用できる．

4.4 特定のカテゴリのノードへの送信

何らかのカテゴリ $\{c_0, \ldots, c_{n-1}\}$ があって，ノード u が 0 個以上のカテゴリに属しているものとする（u が属すカテゴリの集合を $u.C$ とする）．このとき，あるカテゴリ c_x に属すノードにマルチキャストする場合を考える．なお，ここではカテゴリが予め決まっていてそれほど多くない場合を想定する．

- V: n ビットの配列（ビットマップ）
- u.value: $\sum_{c_i \in u.C} 2^i$
- match$(v) \equiv (v \overset{\text{bit}}{\wedge} 2^x) \neq 0$
- reduce$(v_1, v_2) \equiv v_1 \overset{\text{bit}}{\vee} v_2$

ただし，$\overset{\text{bit}}{\wedge}$ と $\overset{\text{bit}}{\vee}$ はそれぞれビット単位の論理積と論理和である．

4.5 Bloom Filter による配送

各ノードが 1 つ以上のキーワードを保持しているときに，指定した 1 つ以上のキーワードをすべて持つノードにマルチキャストする場合（AND 検索），Bloom Filter[7] を用いる方法がある．ノード u が保持するキーワード集合を

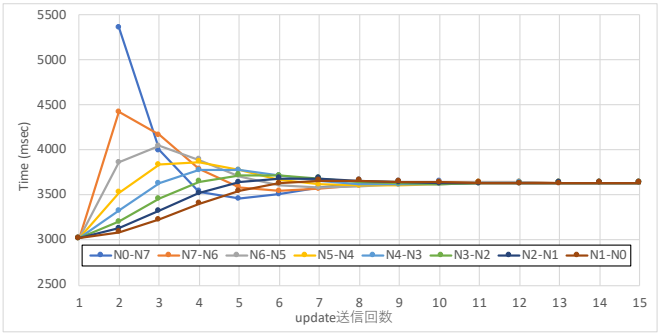

図 11 隣接ノード間の update メッセージ送信時刻の差（更新フロー数=1）

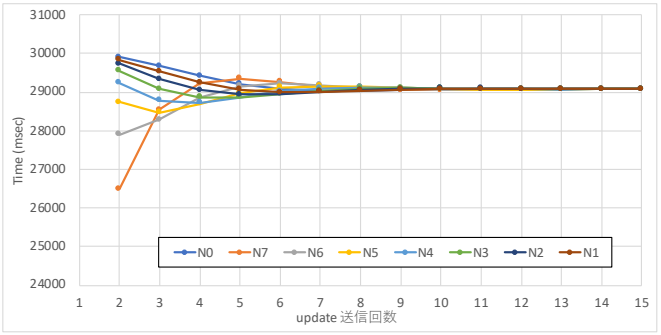

図 12 各ノードにおける update 送信間隔（更新フロー数=1）

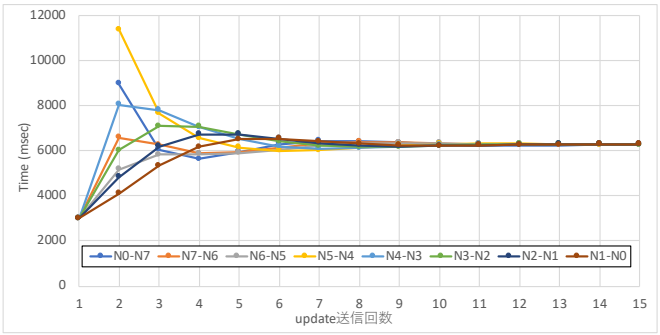

図 13 隣接ノード間の update メッセージ送信時刻の差（更新フロー数=2）

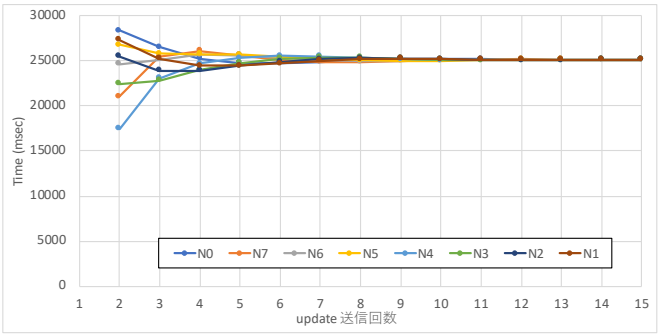

図 14 各ノードにおける update 送信間隔（更新フロー数=2）

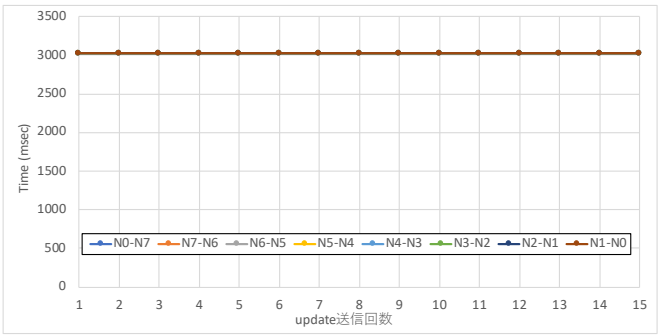

図 15 隣接ノード間の update メッセージ送信時刻の差（更新フロー数=2，$\alpha = 0$）

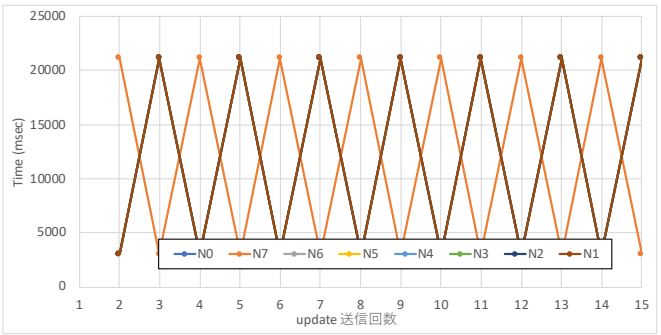

図 16 各ノードにおける update 送信間隔（更新フロー数=2，$\alpha = 0$）

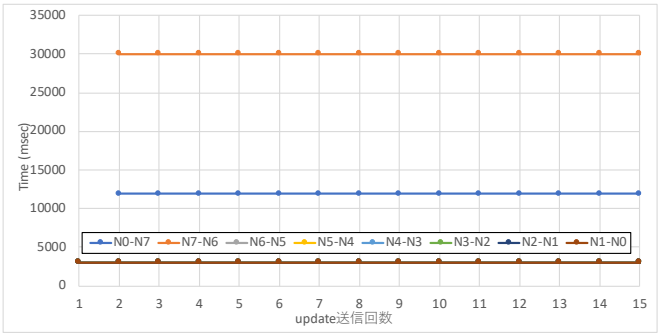

図 17 隣接ノード間の update メッセージ送信時刻の差（更新フロー数=2，$\alpha = 1$）

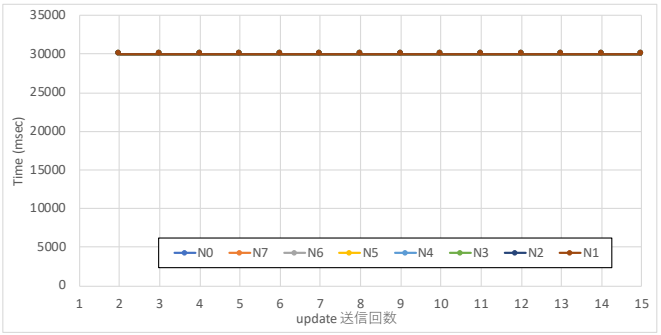

図 18 各ノードにおける update 送信間隔（更新フロー数=2，$\alpha = 1$）

$u.S$，マルチキャスト対象のキーワード集合を K とする．

- V: m ビットの Bloom Filter（ビット配列）
- u.value: $u.S$ の各要素を登録（追加）した Bloom Filter
- match$(v) \equiv$ (Bloom Filter v contains $\forall K$)
- reduce$(v_1, v_2) \equiv v_1 \overset{\text{bit}}{\vee} v_2$

m は想定するキーワード数と許容する Bloom Filter の

偽陽性確率に応じて選択する．reduce は，Bloom Filter ではビット単位の論理和によって和集合が得られる特性を利用している．なお，偽陽性のため，最終的にノード u が受信したメッセージをアプリケーションに渡すかどうかは $u.S$ と K によって判定する必要がある．

この方法は，トピックベースの Pub/Sub システムや（キーワードとしてトピック, 名を用いる），各ノードが分散保持する文書の全文検索に応用できる．

5. 関連研究

構造化オーバーレイネットワークを用いた ALM は一般的である（[5], [8] など）が，ALM の対象ノードをキー以外の条件によって絞る方法は著者の知る限り知られていない．

本研究に関連した研究として構造化オーバーレイネットワークを用いて各ノードの持つ値の集約値（MIN，MAX，AVERAGE など）を求める手法がある（[9], [10] など）．このようなクエリを**集約クエリ**と呼ぶ．これらと提案手法は，経路表に一部のノードの値の集約値を保持する点が共通しているが，集約クエリではこれを（任意のキー範囲内の）値の集約値を求めるために用いるのに対し，提案手法ではルーティングに用いる点が異なる．また，本研究で提案している連続的更新方式は集約クエリにおいて正しい（最新の）集約値を求めるために利用できる（なお，提案手法でも集約クエリは実行可能である）．

多次元範囲検索が可能な構造化オーバーレイネットワークはいくつか提案されている（[3], [11] など）．提案手法でも多次元の範囲検索が可能であるが，提案手法では多次元範囲検索を含むさまざまなマルチキャストが可能な汎用的な枠組みを提供している点が異なる．なお，提案手法を用いた多次元範囲検索の評価と改善は今後の課題の 1 つである．

6. おわりに

本稿では，構造化オーバーレイネットワークを用いた条件付きマルチキャストの手法を述べた．配送先ノードはアプリケーション側で定義する関数によって指定するため，柔軟な条件を指定できる．また，各ノードの集約値を含む経路表を効率よく更新する手法（連続的更新方式）を考案した．連続的更新方式は Chord$^\#$ をベースとする構造化オーバーレイネットワークでも利用できる．

提案手法は P2P 基盤ソフトウェア PIAX[12] において試験的な実装を行ったが，Chord$^\#$ ではなく構造化オーバーレイネットワーク Suzaku[13] をベースとしている．また連続的更新手法は未実装である．

今後の課題としては，連続的更新手法において，複数の更新フローが存在するときの動作の分析，過剰な更新フローを削減する仕組みの検討，4 節で挙げた応用手法の評価などが挙げられる．また，4.2 節および 4.3 節で述べた 1 次元および多次元範囲に対するマルチキャストは，各ノードの key と value に相関がない場合，マルチキャスト時に無駄な配送が発生する可能性が高い（キーが近いいくつかのノードの value を reduce するだけで集約値の範囲が広くなるため）．これを改善する方法も検討したい．

参考文献

[1] Banno, R., Takeuchi, S., Takemoto, M., Kawano, T., Kambayashi, T. and Matsuo, M.: Designing Overlay Networks for Handling Exhaust Data in a Distributed Topic-Based Pub/Sub Architecture, *Journal of Information Processing*, Vol. 23, No. 2, pp. 105–116 (2015).

[2] Yahyavi, A. and Kemme, B.: Peer-to-peer architectures for massively multiplayer online games: A Survey, *ACM Computing Surveys*, Vol. 46, No. 9, pp. 1–51 (2013).

[3] Schütt, T., Schintke, F. and Reinefeld, A.: Range queries on structured overlay networks, *Computer Commun.*, Vol. 31, No. 2, pp. 280–291 (2008).

[4] Stoica, I., Morris, R., Liben-Nowell, 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. on Net.*, Vol. 11, No. 1, pp. 17–32 (2003).

[5] Banno, R., Fujino, T., Takeuchi, S. and Takemoto, M.: SFB: a scalable method for handling range queries on Skip Graphs, *IEICE Communications Express*, Vol. 4, No. 1, pp. 14–19 (online), DOI: 10.1587/comex.4.14 (2015).

[6] Abe, K. and Yoshida, M.: Constructing Distributed Doubly Linked Lists without Distributed Locking, *Proc. of the IEEE Intl. Conf. on P2P Computing 2015*, pp. 1–10 (2015).

[7] Bloom, B. H.: Space/Time Trade-offs in Hash Coding with Allowable Errors, *Communications of the ACM*, Vol. 13, No. 7, pp. 422–426 (online), DOI: 10.1145/362686.362692 (1970).

[8] González-Beltrán, A., Milligan, P. and Sage, P.: Range queries over skip tree graphs, *Computer Commun.*, Vol. 31, No. 2, pp. 358–374 (2008).

[9] Abe, K., Abe, T., Ueda, T., Ishibashi, H. and Matsuura, T.: Aggregation Skip Graph: A Skip Graph Extension for Efficient Aggregation Query over P2P Networks, *International Journal on Advances in Internet Technology*, Vol. 4, No. 3, pp. 103–110 (2012).

[10] Takeda, A., Oide, T. and Takahashi, A.: A Structured Overlay Network for Aggregating Sensor Data, *2012 Seventh International Conference on Broadband, Wireless Computing, Communication and Applications*, pp. 684–689 (2012).

[11] Shu, Y., Chin Ooi, B., Tan, L. and Zhou, A.: Supporting multi-dimensional range queries in peer-to-peer systems, *5th IEEE International Conference on Peer-to-Peer Computing (P2P'05)*, pp. 173–180 (2005).

[12] PIAX development team: PIAX distributed computing framework, Osaka University (online), available from ⟨http://piax.org/en/⟩ (accessed 2019/05/12).

[13] Abe, K. and Teranishi, Y.: Suzaku: a Churn Resilient and Lookup-Efficient Key-Order Preserving Structured Overlay Network, *IEICE Transactions on Communications*, Vol. E102-B, No. 9 (online), DOI: 10.1587/transcom.2018EBT0001 (2019).