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と評価

安倍 広多[1,a)]

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すべてのノード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．本稿では提案手法の詳細と解析および評価について述べる．

1. はじめに

構造化オーバーレイネットワークは，複数のノードが自律的に協調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 [2]，オンラインゲーム [3] 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 関数も自由に定義できるため，宛先ノードを柔軟に選ぶことができる．

match の条件を満たすノード u がマルチキャストを受信するためには，すべてのノードが u の value を集約値として収集する必要がある．このため，集約値を効率よく収集する手法（連続的更新手法）も考案した．

[1] 大阪市立大学大学院工学研究科
Graduate School of Engineering, Osaka City University, Japan
[a)] k-abe@osaka-cu.ac.jp
本稿は [1] に加筆・訂正を行ったものである．

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．3，4，5 章で提案手法とその応用例，評価について述べる．6 章で関連研究を述べ，7 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$^{\#}$[4] 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[5] 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，簡潔のため本稿ではこれらを区別しない（finger[i] をポインタとしてもキーとしても扱う）．

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．（なお，ノードが finger を取得するためのメッセージを *getEnt* と呼ぶことにする．）

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．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．

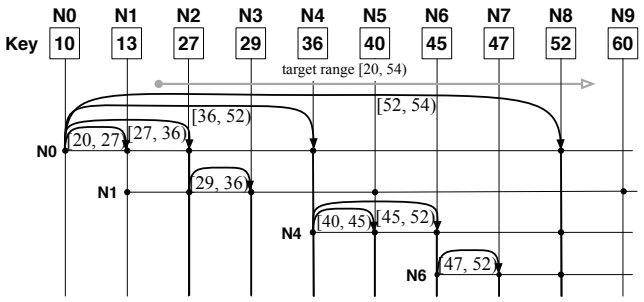

図 1　SFB によるマルチキャストの例

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 のエントリ数はいずれも $\lceil \log_2 n \rceil$ である（n はノード数）．

Chord$^{\#}$ では，各ノードの更新タイミングはばらばらである．すべてのノードの finger[i] が収束するには，すべてのノードの finger[$i-1$] が収束する必要があり，また，各ノードがすべてのレベルを更新するために $T_{ft}\lceil \log_2 n \rceil$ 時間かかることを考慮すると，すべてのノードの finger table が収束するには最大 $T_{ft}(\lceil \log_2 n \rceil^2)$ 時間要する．

2.2 SFB

SFB(Split-Forward Broadcasting)[6] 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 **1** 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（正確には左端のキーを超えない最大のキーを持つノード）に依頼する．依頼されたノード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．ノードがネットワークに参加中，key は一定だが value は可変である．value は任意の型が使用できる（スカラー値である必要はない）．以下，value の型を V とする．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 2** 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．

図の実線の黒い矢印は N0 から key が 20 以上 50 未満かつ 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を示している．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．

以下，提案手法の詳細について述べる．

3.2 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．

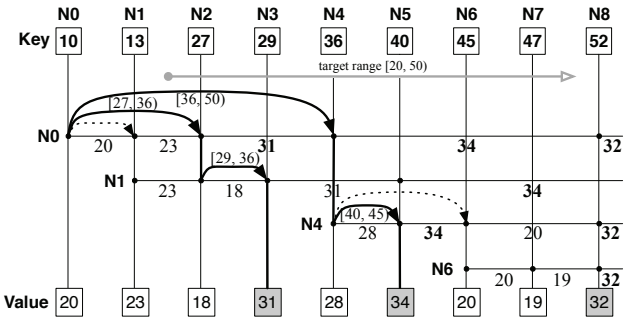

図 2　条件付きマルチキャストの例

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．

$$\begin{aligned} \text{reduce}(v) &\equiv v \\ \text{reduce}(v_1, \ldots, v_n) &\equiv \text{reduce}(\text{reduce}(v_1, \ldots, v_{n-1}), \\ &\qquad v_n) \quad (n > 2) \end{aligned}$$

3.3 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，提案手法では -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．

3.4 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 [7]．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[7] 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，

```
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)
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) {
16   if (finger[i] does not exist) {
17     return (null, null, null)
18   }
19   val ← finger[−1].value
20   j = 0
21   for (; j < i; j++) {
22     val ← reduce(val, finger[j].value)
23   }
24   range ← [u, finger[j−1].range_max)
25   return (finger[i].node, range, val)
26 }
```

図 3　Finger Table 更新アルゴリズム（1 レベルのみ）

(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．

3.5　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）．

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

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

アルゴリズムを図 **3** 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 3.8 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 **4** 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．

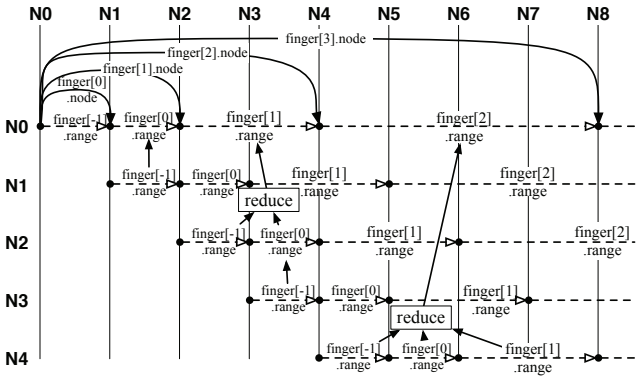

図 4　データ構造と集約の流れ

まず，$N0$.finger[0].node $= N0$.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 $N0$.finger[0].value $=$ reduce($N1$.finger[-1].value) $= N1$.value，$N0$.finger[0].range $= [N1, N2)$ 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，$N0$.finger[1].value $=$ reduce($N2$.finger[-1].value, $N2$.finger[0].value) $=$ reduce($N2$.value, $N3$.value)，$N0$.finger[1].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．

3.6　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，3.8.3 節で述べるタイマをスタートする．

3.7　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 **5** 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

```
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.value)) {
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 by 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 5 条件付きマルチキャストのアルゴリズム

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．

condcast において，N（6 行目）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（10 行目）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（13 行目）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 2 の例において，N0 が condcast を実行したときの様子を**図 6** に示す．

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．

V と reduce の定義によっては，match が true を返した範囲内にマルチキャストの対象ノードが存在しない場合もある（**偽陽性**と呼ぶ）．このとき，送信したメッセージは無駄になるため，偽陽性の確率は低いほうが良い．

3.8 Finger Table の更新（全体）

finger table の 1 エントリの更新方法は 3.5 節で述べた．ここでは，finger table 全体の更新方法について述べる．

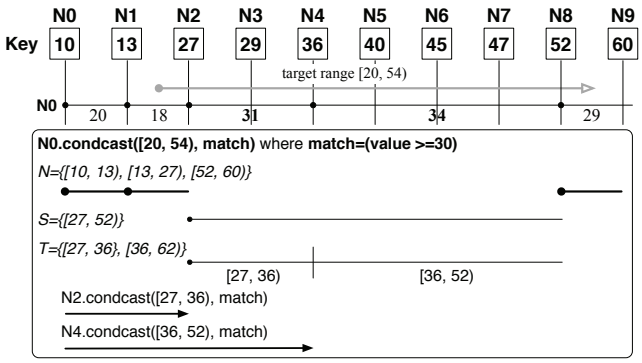

図 6 N0 における条件付きマルチキャストの実行

任意のノード u が，u が宛先の条件に含まれるすべての条件付きマルチキャストを受信するためには，u の value の集約値がすべてのノードに伝播している必要がある．以下，高速に集約値を収集するために考案した手法（連続的更新手法）について述べる．

ノードが finger table を更新する際，右方向に 2 の累乗個離れたノードの finger table を参照するため，右ノードから左ノードの方向に更新処理を行うと効率が良い．このため，以下の方針で finger table を更新する．

- ノードは右ノードからの update メッセージ受信を契機に，finger table のすべてのレベルを更新する．
- 時間調整のためにしばらく待機した後，左ノードに update メッセージを送信する．p

この左方向への更新の流れを**更新フロー**（もしくは「フロー」）と呼ぶ．

この方式を実現するにあたって，以下の目標を立てた．

(1) 自律的に動作する．
(2) 各ノードが finger table を更新する目標周期（以下 P とする）を設定可能とする．
(3) フローが乱れないように，なるべく右ノードの更新に引き続いて更新する．
(4) ネットワーク中のフローの数が最適に近くなるようにフロー数を制御する．
(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．

3.8.1 考え方

注目するノードを u，u の predesessor を p とする．また，u が前回 p に update メッセージを送信した時刻を last，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，update メッセージを受信してから送信するまでの間隔（$s - r$）を δ，u が前回 update メッセージを送信してから次に送信するまでの間隔を ρ とする（**図 7** 参照）．

u が前回 update メッセージを受信してから次に update メッセージを受信するまでの間隔によって，以下の 3 つの場合に分けて考える．

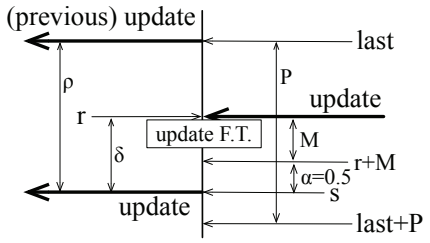

図 7　update 送信タイミング

- (Timeout) u が長時間（$P+G$ 時間）update メッセージを受信しない場合（G は猶予時間）: このとき，u は新しい更新フローを開始する．
  ただし，提案方式ではフローが過剰な場合は削除するようになっている（3.8.2 節で述べる）．削除したフローを復活させないようにするため，タイムアウトしたときは，フローが過剰かどうかを判定し（この方法も 3.8.2 節で述べる），過剰と判断した場合は新しい更新フローを開始しない．
  u でタイムアウトが発生したとき，u の左側ノードでも連鎖的にタイムアウトが発生すると更新フローが過剰に生成される可能性がある．これを防ぐために，最小遅延時間 M を設け，次の Normal ケースで $\delta \geq M$ となるようにする．これによって p のタイムアウト時刻を u のそれよりも最低でも M だけ遅らせる．
- (Normal) u が，前回 update 受信から $P-M$ 時間経過前に次の update メッセージを受信した場合：
  送信周期を保つ観点からすると，$s = \text{last} + P$ が望ましいが，更新フローの連続性を考えると，$s = r + M$ が望ましい．双方を一定の割合で満足させるため，

$$s = \alpha(\text{last} + P) + (1-\alpha)(r + M) \qquad (8)$$

  とする（α は $0 \leq \alpha \leq 1$ を満たすパラメータ）．
- (Delayed) u が，前回 update 受信から $P-M$ 時間経過後，かつ $P+G$ 時間経過前に update を受信した場合：このときはすばやく update メッセージを送信するため，$s = r + M$ とする．

3.8.2　フローの削除

フローがリングを一周するためには，最低 Mn 時間必要である．各ノードの更新周期が P の場合，少なくとも

$$\lceil \frac{M}{P} n \rceil \qquad (9)$$

個のフローが必要である．3.8.5 節で述べるが，フロー数は式 9 に近いほうが良いため，フロー数が過剰な場合は削除する．ネットワーク中のフロー数を正確にカウントすることは難しいため，以下の方法を用いる．

finger table の最大レベルを i，ノード数を n とすると，$2^i < n \leq 2^{i+1}$ である．ここでは $n' = 2^i$ とし，フロー数の上限値 F' を式 9 から $\frac{M}{P} n' + 1$ とする．また，n' と F' から，想定される δ を後述の式 12 から求める（これを δ' とする）．update 処理を実行する際，自ノードにおける δ の値を記録しておき，連続して DEL_FLOW_THRES 回，$\delta > \delta' \times$ DELTA_MARGIN となった場合はフロー過剰と判断してフローを削除する（DELTA_MARGIN は 1 以上のパラメータ）．ただし，フローの削除しすぎを避けるため，フローの削除は，上の条件が満たされた上で，さらに確率的に行う（確率 DEL_FLOW_POSS で削除する）．

3.8.1 節の (Timeout) で述べたが，あるノードで update の受信がタイムアウトした場合，フロー過剰かどうかを判定する必要がある．この判定にも上述の方法を用いる．ただし，削除したフローがすぐに復活する可能性を下げるため，（DEL_FLOW_THRES ではなく）直前 1 回の δ が閾値を上回っただけでフロー数が過剰と判断する（過剰の場合は新規のフローを開始しないため）．

3.8.3　アルゴリズム

擬似コードを**図 8** に示す．ここで，schedule(t, lambda) は t 時間後に lambda を実行するタイマを設定する，cancel(timer) は指定したタイマをキャンセルする，current_time() は現在時刻を返す，random() は 0 以上 1 未満の乱数を生成する関数である．->はラムダ式を表す．

新規ノード挿入時には，`timer = schedule((2 + random())×P, () -> u.update(true))` を実行する．乱数が入っているのは，短時間に多数のノードが挿入された場合に，過剰なフローが生成される確率を下げるためである．

3.8.4　実行例

図 8 のアルゴリズムをシミュレータにより実行したときの update メッセージのシーケンス例を**図 9** に示す．パラメータは以下の通りである：ノード数 = 50，$P = 30$ 秒，$M = 2$ 秒, $G = 5$ 秒, $\alpha = 0.2$, 片方向ネットワーク遅延時間 = 20 ミリ秒, DEL_FLOW_THRES= 3, DEL_FLOW_POSS= 0.1, DELTA_MARGIN= 1.2．なお，このとき式 9 の値 = 4 である．

N49（右端のノード）を最初のノードとし更新フローを開始した．他のノードはその後一度に挿入している．左側のノード（N0〜N18）では update のタイムアウトによりいくつかのフローが生成されていることが確認できる．フローの増加によって δ が大きくなり，フロー間の間隔が徐々に調整されている．さらに，時刻 177 付近で N10 が，179 付近で N48 が，189 付近で N41 がフロー過剰と判定し削除している．この例では最終的にフロー数 4，平均 $\rho = 28.9$ 秒，平均 $\delta = 2.3$ 秒となった．

図 8 のアルゴリズムが自動的にフロー数と更新間隔を調整できていることを確認できる．

3.8.5　解析

以下，更新フローが安定している場合（ノード数に変化がなく，各ノードの δ が収束している場合）を考える．

```
1  last = 0
2  updating = false
3  timer = null
4  d = []
5  u.update(new_flow) {
6    if (updating) return
7    updating = true
8    if (timer != null) {
9      cancel(timer)
10     timer = null
11   }
12   r = current_time()
13   for (i = 1;; i++) {
14     updateFinger(i)
15     break if circulated
16   }
17   if (new_flow) {          // start a new flow
18     s = 0
19     d = []
20   } else if (last = 0 ∨ last + P < r + M) {
21     s = r + M              // first time or delayed
22     d = []
23   } else {                 // normal
24     s = α(last + P) + (1 − α)(r + M)
25     delta = s − r
26     d.push(delta)
27   }
28   if (s − current_time() > 0) {
29     sleep(s − current_time())
30   }
31   if (decrease_flow_p(DEL_FLOW_THRES)
32       ∧ random() < DEL_FLOW_POSS) {
33     last = 0               // delete the flow
34     d = []
35   } else {
36     last = current_time()
37     predecessor.update(false)
38   }
39   // schedule a timer for timeout job
40   timer = schedule(P + G + r − current_time(),
41                    () −> u.on_timeout())
42   updating = false
43 }
44
45 u.on_timeout() {
46   if (decrease_flow_p(1)) {
47     // do not start a new flow for now
48     timer = schedule(P + G, () −> u.update(true))
49   } else {
50     u.update(true) // start a new flow
51   }
52 }
53
54 u.decrease_flow_p(count) {
55   if (delta.length ≥ count) {
56     compute δ′
57     dlasts = d[d.length−count : d.length−1]
58     return true if ∀x > δ′×DELTA_MARGIN
59       where x ∈ dlasts
60   }
61   return false
62 }
```

図 8 連続的更新方式のアルゴリズム

式 8 において，簡単のため last = 0 とする．更新フローが安定しているとき，$r = \rho - \delta$，$s = \rho$ であるため，式 8 は $\rho = \alpha P + (1-\alpha)(\rho - \delta + M)$ となり，以下の式を得る．

$$\alpha(P - \rho) + (1-\alpha)(M - \delta) = 0 \tag{10}$$

フロー数を F とすると，ノードは F 個のフローの update を受信すると最初のフローがリングを一周して戻ってくるので，$\rho F = \delta n$ が成り立つ．これから，

$$\rho = \frac{\delta n}{F} \tag{11}$$

が得られる．また，この式と式 10 から次式が得られる．

$$\delta = \frac{\alpha(P - M) + M}{\alpha(\frac{n}{F} - 1) + 1} \tag{12}$$

任意のノード u が，ある更新フロー f_1 の update メッセージを受信して finger table を更新する場合を考える（**図 10** 参照）．finger table を更新する際に，u は $u_{+1}, u_{+2}, u_{+4}, \ldots$ に getEnt メッセージを送信する．ここで，$u_{+1}, \ldots, u_{+2^{m-1}}$

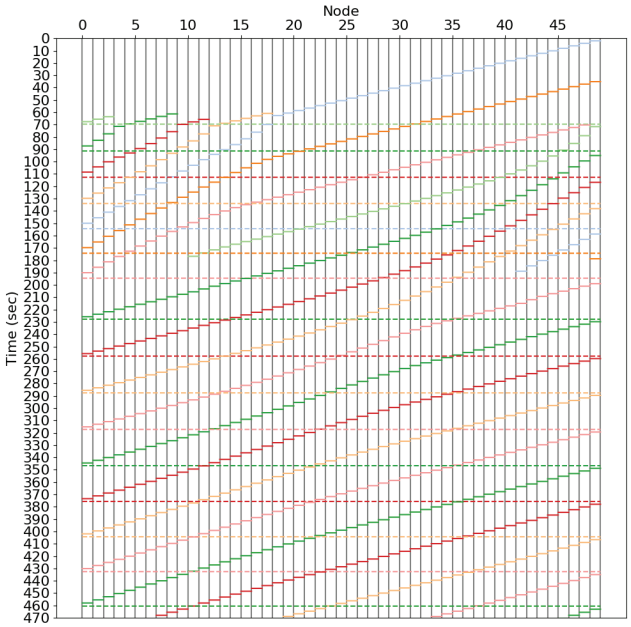

図 9 update メッセージのシーケンス例

までは最後の finger table 更新が同じフロー f_1 で行われ，u_{+2^m} では次のフロー f_2 で行われたとする．m は $\rho < 2^m\delta$ を満たす最小の整数になるので，

$$\begin{aligned} m &= \lceil \log_2 \frac{\rho}{\delta} + \epsilon \rceil \\ &= \lceil \log_2 \frac{n}{F} + \epsilon \rceil. \end{aligned} \tag{13}$$

最大集約時間（ノードが取得した集約値の中で最も古い value の “古さ”）を E とする．

n 個のノードはフローによって（収束時は等分に）F 個の区画に分割される．ノード u があるフローの update メッセージを受信して finger table を更新する際，finger table のレベル m 以上では異なる区間のノードに getEnt を送信する．これらのノードは ρ 以内に更新しているはずである．u から，レベル m 以上の getEnt メッセージを逆方向にたどると，すべてのノードに到達するにはおよそ $(\log_2 F)\rho$ 時間必要である（図 10 参照）．また，レベル m 未満のノードは同じ区画に存在するため，最大 $(2^m - 2)\delta$ 時間かかる．以上より，E の近似式として以下を得る．

$$E \approx (\log_2 F)\rho + (2^m - 2)\delta \tag{14}$$

次に，最適なフロー数を求めるために，各ノードがすべての value の集約値を収集するために必要な update 実行回数（E/ρ）を求める．

$$\begin{aligned} \frac{E}{\rho} &\approx \log_2 F + (2^m - 2)\frac{\delta}{\rho} \\ &\approx \log_2 F + (\frac{n}{F} - 2)\frac{F}{n} \\ &= \log_2 F - \frac{2F}{n} + 1 \end{aligned} \tag{15}$$

F の値域 $\lceil \frac{M}{P} n \rceil \le F \le n$ では，式 15 は F が最小値の場

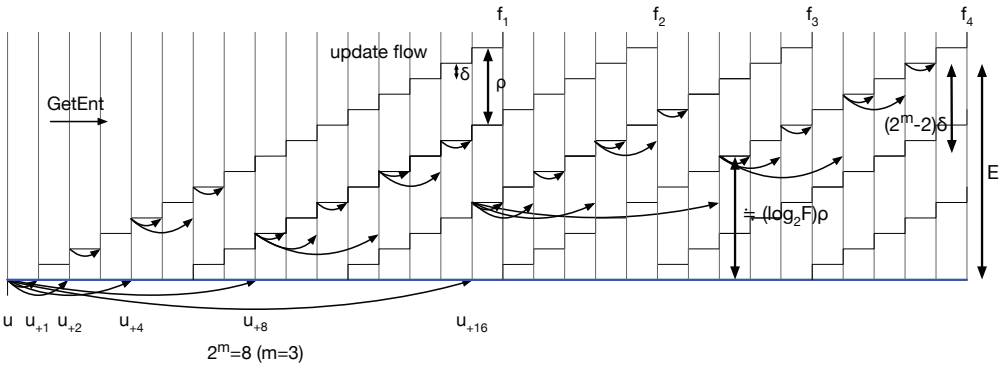

図 10　更新フローと最大集約時間の関係

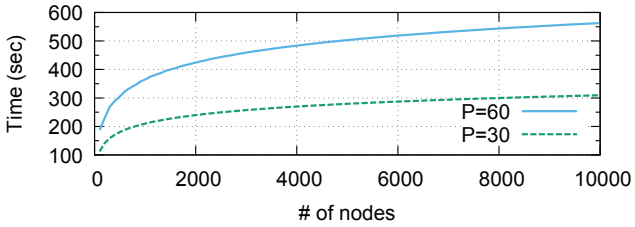

図 11　ノード数に対する最大集約時間

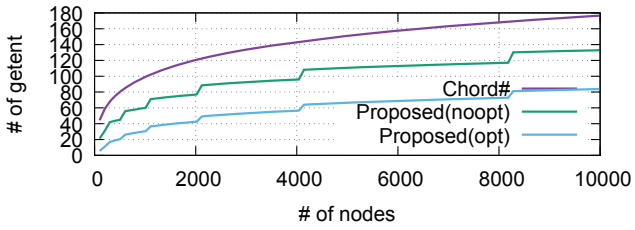

図 12　すべてのノードの value 集約に必要な getEnt メッセージ数

合（$F = \lceil \frac{M}{P} n \rceil$）に最小となる．つまり，フローは少ないほうが良いということになる．

E（式 14）のグラフを図 **11** に示す（パラメータは $P = \{30, 60\}$ 秒，$M = 2$ 秒，$\alpha = 0.2$，$F = \lceil \frac{M}{P} n \rceil$）．$E$ はノード数に対して対数オーダで増加する．

また，すべてのノードが最新の集約値を取得するために必要な，ノードあたりの getEnt メッセージ数（すなわち，最大集約時間内に各ノードが送信する getEnt メッセージ数）を図 **12** に示す．ここでは，Chord$^{\#}$ 方式（2.1.2 節参照）と連続的更新方式（3.9 節で述べる最適化を行った場合と行わなかった場合）をプロットしている（パラメータは，$P = 60$ 秒，$M = 2$ 秒，$\alpha = 0.2$，$F = \lceil \frac{M}{P} n \rceil$．Chord$^{\#}$ では理論値 $(\log_2 n)^2$ を用いた）．連続的更新方式（最適化あり）では，Chord$^{\#}$ 方式の半分程度のメッセージ数ですべての集約値を集められている．

3.9　getEnt メッセージの削減

update メッセージが順次左ノードに送信されることを利用して，getEnt メッセージの数を削減できる．

update メッセージが配列 d$[2^m][m]$ を保持する．ノードが update メッセージを左ノードに送信する際に，d$[x+1][y]$=d$[x][y]$ となるようにシフトし（d$[2^m$-1$]$ は削除），getEnt(i) の返り値を d$[0][i]$ に格納する．

update メッセージを受信したノードは，finger table のレベル i $(i < m)$ については getEnt メッセージを送信せず，代わりにメッセージ中の配列 d を参照して更新する（詳細は割愛する）．これによって，一回の update で送信する getEnt メッセージ数を $\log_2 n$ から $\log_2 n - m$ に削減できる．

4.　応用例

提案手法では様々な条件付きマルチキャストを実現できるが，そのうちのいくつかの例を述べる．例えば (1)〜(3) はノードが持つリソースやセンサーの値が一定範囲内のノードにメッセージを配送するために，(4)〜(5) は Pub/Sub システムなどに利用できる．

(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 がスカラー値 $u.c$ を保持するときに，値が C 以上のノードへマルチキャスト（V: スカラー値，u.value: $u.c$，match$(v) \equiv v > C$，reduce$(v_1, v_2) \equiv \max(v_1, v_2)$）．

(2) 1 次元範囲マルチキャスト　5.2 節で述べる．

(3) 多次元範囲マルチキャスト　ノード u が d 次元の値 $u.p$ を保持しているとき，値が d 次元の範囲 R に含まれるノードにマルチキャスト（V: d 次元の範囲, u.value: $[u.p, u.p]$，match$(v) \equiv (v \cap R \neq \emptyset)$，reduce$(v_1, v_2) \equiv$ d 次元範囲 v_1，v_2 を包含する d 次元範囲）．

(4) 特定のカテゴリに所属するノードへのマルチキャスト
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ときに，あるカテゴリ c_x に属すノードにマルチキャスト（V: n ビットのビットマップ，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）．

(5) 特定のキーワードを持つノードへのマルチキャスト

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 つ以上のすべて（もしくはいずれか）のキーワードを持つノードにマルチキャスト（V: 適切な大きさの Bloom Filter[8]，u.value: u が保持するキーワードを追加した Bloom Filter，match(v): Bloom Filter v がすべて（もしくはいずれか）のキーワードを含むなら true，reduce$(v_1, v_2) \equiv v_1 \overset{\text{bit}}{\vee} v_2$）．

(6) 複数の条件の AND・OR によるマルチキャスト

各ノード u が複数の value $\{v_1, \ldots, v_k\}$ を持つ場合，u.value $= \{v_1, \ldots, v_k\}$ とし（配列），reduce 関数は配列の要素ごとに集約することで，match は $v_1, \ldots, v_k$ に関する AND 条件や OR 条件を指定できる．

5. 評価

5.1 マルチキャストのホップ数

提案手法では，どのような match，reduce を用いたとしても，すべての目的ノードへのホップ数は最大 $\lceil \log_2 n \rceil$ である（経路表収束時）．

5.2 マルチキャストの効率

マルチキャストに要するメッセージ数は少ない方が良い．対象ノードが多いほどメッセージ数は多くなるため，マルチキャストの効率（対象ノード数 / 送信したメッセージ数）を評価した．値が 1 に近いほど効率的である．

効率は match と reduce によって変化する．ここでは 1 次元範囲マルチキャスト（各ノード u が value として整数 $u.c$ を持つ．指定した 1 次元範囲 $R = [R_{\min}, R_{\max}]$ に $u.c$ が含まれるすべてのノード u にマルチキャストする）を取り上げ，以下の 2 つの方式を比較する．

方式 1: 範囲による集約

- 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)（merge(v_1, v_2) は 2 つの 1 次元範囲 v_1, v_2 の両方を包含した 1 つの範囲を返す関数）

方式 2: ビットマップによる集約

- V: m ビットの配列（m は $u.c$ の最大値 $+1$）
- u.value: $v[u.c] =$ true，その他の要素は false である配列 v
- match(v): $v[R_{\min} \ldots R_{\max}]$ に true の要素があれば true，なければ false
- reduce(v_1, v_2): 配列 v_1 と v_2 のビット単位の論理和

評価はシミュレーションで行った．諸条件は以下の通り：ノード数は 100，value は 0 から 99 までの整数を一様乱数で生成，マルチキャスト対象の value の範囲の幅 w は 10 から 100 まで 10 刻みで増加，対象 value の範囲は（上限下限とも 0 から 100 までに収まるように）乱数で決定．

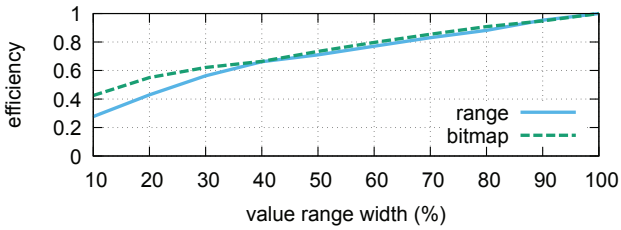

図 13 1 次元範囲マルチキャストの効率

各 w について試行回数は 10．なお，各試行は finger table が収束した後（すべての集約値を収集した後）に行った．

結果を**図 13** に示す．横軸は w，縦軸は効率の平均値である．グラフの右端では，すべてのノードが対象となるため効率 $= 1$ である．

方式 2 では基本的に偽陽性がない（finger table エントリの value（ビットマップ）で match=true ならば，その範囲に条件を満たすノードが必ず存在する）が，目的ノードに到達するために条件を満たさないノード（非目的ノード）を経由する場合があり，w が小さいほどその確率は高い．

方式 1 は方式 2 よりも効率が劣るが，これは次の理由による：方式 1 では，finger table エントリの範囲（.range）が少し広くなっただけで，集約した範囲が value の値域のほとんどを含む程度に広くなることがある（隣接ノードの value が 0 と 99 の場合を考えてみよ）．このため，w が小さいと偽陽性確率が高くなる．

提案手法を一次元範囲マルチキャスト以外のマルチキャストに適用した場合，（偽陽性がない）方式 2 よりも効率が良くなることはないと考えられる．つまり，全体の w%のノードが対象となるマルチキャストの平均効率は，最も良い場合で図 13 の方式 2 程度と予想される（ただし各ノードの key と value に相関がない場合）．

5.3 従来手法との比較

従来の構造化オーバーレイネットワーク上の ALM を用いて条件付きマルチキャストと同様の処理を実現する場合，次の方法が考えられる（以下従来手法）：(1) 各ノードの value を key にマップし，挿入する．マップする際，マルチキャストで送信する相手ノードがキー空間上で連続する範囲に収まるようにする．(2) マルチキャストの際は，この連続範囲に対して ALM で送信する．

提案手法は従来手法と比較して次のメリットがある：(1)key は全順序集合の要素であるため，value を key にマップすることが困難な場合がある（value が多次元の値である場合など）が，提案手法ではそのような問題はない，(2) 提案手法では key を value と別の目的に使用できるが（対象ノードの絞り込みなど），従来手法ではできない．(3) 従来手法は value に変化がある度にノードを挿入し直す必要があるが，提案手法はその必要はない，(4) 従来手法では，

場合によっては 1 つの物理ノードが複数の（論理）ノードを挿入する必要がある．例えば 4 章の「特定のカテゴリに所属するノードへのマルチキャスト」を従来手法で実現した場合，各物理ノードは所属するカテゴリ数だけ論理ノードを挿入する必要がある．論理ノード数に比例して経路表の維持コストがかかるため，1 物理ノードがあまりに多数の論理ノードを挿入することは現実的ではないが，提案手法では挿入する論理ノード数は 1 つでよい．

一方，提案手法は次のデメリットがある：(1) 偽陽性がある場合，不要なメッセージを送信する可能性がある，(2) 従来手法ではノードの挿入直後からマルチキャストを受信できるのに対し，提案手法ではノードの value がすべてのノードに伝播するまで待つ必要がある．(3) オーバーレイネットワークの経路表に加え，集約した value を転送・保持する必要がある（V のサイズが大きい場合には問題となる可能性がある）．

6. 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（[6], [9] 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（[10], [11] 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．なお，提案手法でも finger table 内の集約値を用いて集約クエリを高速に実行することは可能である．集約クエリでは，なるべく新しい集約値を収集する必要があるが，そのためには本研究で提案している連続的更新方式が利用できる．

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（[4], [12] 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．

7. 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．配送先となるノードはアプリケーション側で定義する関数によって柔軟に指定できる．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を考案した．さらに，提案手法の基礎的な解析と評価を行った．

3.8.5 節で述べたように，提案方式で新規ノードへマルチキャスト可能となるまでにはある程度の時間が必要である．今後，新規ノードが挿入されたときにアクティブに他のノードの集約値を更新する方法を検討したい．

参考文献

[1] 安倍広多：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，マルチメディア，分散，協調とモバイル (DICOMO2019) シンポジウム予稿集，pp. 274–283 (2019).

[2] 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).

[3] 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).

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

[5] 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).

[6] 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).

[7] 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).

[8] 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).

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

[10] 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).

[11] 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).

[12] 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).