一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
IA2016-65(2016-12)

高いChurn耐性と検索性能を持つ キー順序保存型構造化オーバレイネットワークSuzakuの提案と評価

安倍　広多†　　寺西　裕一††

† 大阪市立大学大学院創造都市研究科　〒 558–8585 大阪府大阪市住吉区杉本 3–3–138
†† 情報通信研究機構　〒 184–8795 東京都小金井市貫井北町 4–2–1
E-mail: †k-abe@gscc.osaka-cu.ac.jp, ††teranisi@nict.go.jp

あらまし　キーの値による範囲検索が可能なキー順序保存型構造化オーバレイネットワークは多くの応用があり，重要性が高い．本研究では，新しいキー順序保存型構造化オーバレイネットワーク Suzaku を提案する．Suzaku は，(1)Churn 時でも最大検索ホップ数が $\log_2 n$ 程度に収まる（n はノード数），(2) キーが大小どちらの方向でも近傍ノードの検索は高速に行える，(3) 構造は単純で実装が容易，といった特徴を備える．本稿では Suzaku の詳細について述べ，シミュレーションによって既存の Chord$^\#$ および Skip Graph と比較する．

キーワード　キー順序保存型構造化オーバレイネットワーク，Churn 耐性

Proposal and Evaluation of Suzaku, a Churn Resilient, Lookup-Efficient and Key-Order Preserving Structured Overlay Network

Kota ABE† and Yuuichi TERANISHI††

† Graduate School for Creative Cities, Osaka City University
3–3–138 Sugimoto, Sumiyoshi-ku, Osaka, 558–8585 Japan
†† National Institute of Information and Communications Technology,
4–2–1 Nukui-Kitamchi, Koganei, Tokyo, 184–8795 Japan
E-mail: †k-abe@gscc.osaka-cu.ac.jp, ††teranisi@nict.go.jp

Abstract　A “key-order preserving structured overlay network,” which enables range queries, has various applications and thus be important. In this study, we propose a novel key-order preserving structured overlay network “*Suzaku*,” which has the following properties: (1) maximum lookup hops is almost $\log_2 n$ even in churn situations, where n is the number of nodes, (2) neighbor search is fast regardless of the direction of their keys, (3) the structure is simple and easy to implement. In this paper, we describe the principles and detailed algorithm of Suzaku. We also show simulation results comparing Suzaku with existing Chord$^\#$and Skip Graph.

Key words　Key-Order Preserving Structured Overlay Network, Churn-Resiliency

1. はじめに

構造化オーバレイネットワーク（以下，構造化オーバレイ）は，ノード間の自律的なルーティングにより，キーによって指定された目的ノードを検索するアプリケーションレイヤのネットワークである．構造化オーバレイのうち，任意の全順序集合の要素をキーとし，キーの隣接関係を保存した構造（キーの値が隣接したノードはオーバレイ上でも隣接する構造）によって範囲検索を可能とする構造化オーバレイ（キー順序保存型構造化オーバレイ）は，アプリケーションレベルマルチキャスト (ALM)，オンラインゲーム [1]，分散 pub/sub [2]，分散 DB [3] などの応用があり，その重要性は高い．

キー順序保存型構造化オーバレイには以下の性質が要求される．

1) 非一様分布のキーを持つノードを対象とする検索の高速性　Chord [4] 等のハッシュ関数を適用したキーを用いる構造化オーバレイと異なり，キーの値が一様に分布しない前提のもと，多数のノードを対象とするルーティング・検索を高速に行なえる必要がある．

2) 近傍ノードを対象とする検索の高速性　ノードが保持するキーの値が近い近傍ノード群を一つのグループと見なす分散 pub/sub などの応用では，グループ内の通信を高速に行う必要

性から近傍ノードへのルーティング・検索を高速に行なえることが求められる．

3) Churn 耐性　キー順序保存型構造化オーバレイでは，特定のキー範囲内に，多数のノードが短時間に挿入・削除されることがある（Churn 状態．例えば分散 pub/sub において特定のコンテンツに購読者が集中したり，あるいは ALM において配信が終了してすべての受信者が離脱する場合など）．このような場合でも，ルーティング・検索が性能劣化なく行なえることが望ましい．

既存のキー順序保存型構造化オーバレイの代表例として，Chord$^{\#}$ [5]，Skip Graph [6] がある．どちらも 1) を念頭に設計されているが，a) Chord$^{\#}$ は経路表が収束すると最大検索ホップ数は $\lceil \log_2 n \rceil$ となるものの（n はノード数），収束するまではこれを大きく超える．また，b) Skip Graph は最大検索ホップ数は $\log_2 n$ の数倍程度となることが一般的であり，どちらも改善の余地がある．また，Chord$^{\#}$ は 2)，3) を満たさない．Skip Graph は 2)，3) を満たすが，構造が複雑で実装が難しいという問題がある（これら既存方式については 2. 章に詳しく述べる）．

本稿では，1) 経路表収束時，最大検索ホップ数は $\lceil \log_2 n - 1 \rceil$，2) キーの大小に関わらず近傍ノードは高速に検索可能，3)Churn 状態においても検索ホップ数はほとんど変化しない，といった特徴を備え，実装も容易な新たなキー順序保存型構造化オーバレイ *Suzaku* を提案する (3. 章)．また，Churn 時における検索性能，近傍ノードの検索性能，オーバレイを維持するためのトラフィックについて，Chord$^{\#}$ および Skip Graph と比較を行い，Suzaku の優位性を示す (4. 章)．

2. 関 連 研 究

既存のキー順序保存型構造化オーバレイの代表例である Chord$^{\#}$ と Skip Graph について述べる．これらはリングベースの構造化オーバレイであり(注1)，各ノードは自ノードの次にキーが大きいノードへのポインタ（successor と呼ぶ）と，キーが次に小さいノードへのポインタ（predecessor と呼ぶ）を持つ（ただし，最大キーのノードの successor は最小キーのノード，最小キーのノードの predecessor は最大キーのノードを指す）．各ノードの successor と predecessor で構成される双方向リングをレベル 0 リングと呼ぶ．なお，本稿ではキーが大きくなる方向を時計回り，小さくなる方向を反時計回りとする．

また，各ノードは複数のレベルを持つ経路表を保持し，レベル i に 2^i 個（程度）離れたノードへのポインタを格納するスキップ構造を構築する．検索キーと経路表中のポインタが指すノードのキーに基づいた greedy ルーティングを行うことで，ノード数 n に対し平均検索ホップ数 $O(\log n)$ を実現する．

2.1 Chord$^{\#}$

Chord$^{\#}$ の経路表は finger table と呼ばれる．finger table は $\lceil \log_2 n \rceil$ 要素の 1 次元配列であり，各要素は次の式によって決定される．

$$\mathrm{finger}[i] = \begin{cases} \text{successor} & (i = 0) \\ \mathrm{finger}[i-1] \rightarrow \mathrm{getFinger}(i-1) & (i > 0) \end{cases}$$

finger[0] は successor と等しい．ノード p の finger[i] ($i > 0$) は，定期的に finger[$i-1$] が指すノードから finger[$i-1$] を取得することで更新する．finger table エントリ取得のための問い合わせを **finger エントリ取得リクエスト**と呼ぶ．全ノードの経路表が収束すると，各ノードの finger[i] は時計回り方向に 2^i 個離れたノードへのポインタを保持することになり，最大検索ホップ数は $\log_2 n$ となる．

2.2 Skip Graph

Skip Graph では，各ノードは固有の Membership Vector (MV) と呼ばれる乱数を保持し，これを基に経路表を構築する（本稿では MV は 2 進数とする）．Skip Graph の経路表は多段の双方向連結リストで構成される．レベル 0 では successor ノードと predecessor ノードと接続し，レベル i ($i > 0$) では，両方向で MV の先頭 i 桁が一致する最も近いノードと接続することでスキップ構造を構築する．

2.3 議　　論

Chord$^{\#}$ と Skip Graph とを比較する．

近傍ノードの検索　Skip Graph では，時計回り方向，反時計回り方向の両方にスキップ構造を構築するのに対し，Chord$^{\#}$ では時計回り方向のみを構築する．このため Chord$^{\#}$ で反時計回り方向に近いノードを検索する場合，リング構造を一周近くたどるルーティング（あるいは反時計回り方向へのレベル 0 を用いたルーティング）が必要となり，検索遅延は大きくなる．

検索ホップ数　Chord$^{\#}$ は，経路表が収束した状態ならば最大検索ホップ数は $\log_2 n$ となるが，収束していない場合は $\log_2 n$ を超える．特に隣接する 2 ノード間に短時間に多数のノードが挿入される場合は，経路表の更新が間に合わず，検索ホップ数が $O(x)$ となることがある（x は挿入したノード数）．一方，Skip Graph の平均検索ホップ数は $O(\log n)$ であるが，経路表が収束した Chord$^{\#}$ の平均検索ホップ数よりも大きい．また，最大検索ホップ数は一般に $\log_2 n$ の数倍程度となる．これは経路表の構築に乱数を利用していることが原因である．Skip Graph の検索ホップ数を改善する方法も提案されているが [7]，Chord$^{\#}$ と同程度まで改善するには時間を要する．

削除済みノードへのメッセージ送信　Chord$^{\#}$ の経路表には対称性がないため（任意のノード p に対して，p の経路表エントリがノード q を指しているならば q の経路表エントリが p を指している場合に経路表に対称性があるという），ノードを削除するときに，当該ノードを finger table で参照しているノードに削除通知を送ることができない．このため，Chord$^{\#}$ では削除済みノードに検索クエリが転送されることがある．タイムアウトを待って検索処理を再実行すればよいが，タイムアウト待ちの分，検索時間は長くなる．Skip Graph では，経路表に対称性があるため，このような問題は生じない．

構造の複雑さ　Skip Graph は多数の双方向連結リストによって構成されるが，分散環境において，並行挿入や並行削除，障害が発生する中でこれらを一貫性を維持しながら更新する処理

(注1)：Skip Graph の原論文では両端が繋がっていない双方向連結リストを用いているが，本稿ではリングとして扱う．

は単純ではない．これに対し，Chord$^\#$ は finger table にノードへのポインタを格納するだけの単純な構造である．このため，Chord$^\#$ は Skip Graph に比べて実装が容易である．

3. Suzaku

本章では Suzaku について述べる．Suzaku は，finger table 更新に要するメッセージ数を抑えつつ，2.3 節で述べた既存のキー順序保存型構造化オーバレイの欠点を解決・改善する．

3.1 設計方針

Suzaku の設計方針は以下の通りである．

finger table の使用　構造の単純性を考慮し，Suzaku は Chord$^\#$ と同様，経路表として finger table を用いる．

finger table の双方向化　ただし，反時計回り方向に近いノードを素早く検索するために，双方向で finger table を持つ．従来の時計回り方向の finger table を forward finger table（FFT），新たに設ける反対方向の finger table を backward finger table（BFT）と呼ぶ．

パッシブ更新の導入　Suzaku では，Chord$^\#$ と同様，各ノード p は他のノード q に finger エントリ取得リクエストを送信することでエントリを取得するが（これを**アクティブな更新**と呼ぶ），このとき，q の反対方向の finger table のエントリも同時に更新する（**パッシブな更新**と呼ぶ）．これによって finger table 更新に要するメッセージ数を節約する．

ノード挿入直後のアクティブな finger table 更新　Chord$^\#$ では挿入直後のノードの finger table は空であるが，Suzaku ではノード挿入直後は FFT と BFT のすべてのエントリをアクティブに更新する．パッシブ更新との組み合わせにより，多数のノードが短時間に挿入された場合の性能低下を抑制する．

一貫性のあるリング管理アルゴリズムの導入　FFT と BFT を正しく更新するためには，successor と predecessor ポインタが正しいノードを指している必要があるが，これらのポインタの管理のために Chord$^\#$ が採用している Chord [4] のスタビライズアルゴリズムは，多数のノードが短時間で挿入された場合に正しい値に収束するまで時間を要することが知られている [8]．このため，Suzaku では，successor と predecessor の一貫性を保ちつつノードの挿入・削除が可能なリング管理アルゴリズム（DDLL [8] など）を用いる．

finger table 更新時のノード生存確認　アクティブな更新を行う際，Chord$^\#$ ではリモートノードから取得したポインタが指すノードが生存していることを確認せずに finger table に格納するが，Suzaku では確認してから格納する．これによって，検索時に削除済みノードや障害ノードに対して検索クエリを送信してしまう可能性を低下させる．

リバースポインタの導入　各ノード p は，finger table のレベル 1 以上で p へのポインタを保持しているノードへのポインタ（**リバースポインタ**）の集合（**リバースポインタ集合** R）を持つ．ノード削除時には，リバースポインタ集合に含まれる各ノードに削除通知を送信する．削除通知に p の代替ノードを含めることで，受信した各ノードは p を指す finger table エントリを代替ノードに変更する．これにより以後の p へのメッセージ送信を抑制する．

3.2 詳　　細

ノード p の FFT を p.FFT のように表記する．p.FFT[0] および p.BFT[0] はリング管理アルゴリズムによって管理される successor および predecessor と等しい．

3.2.1 ノード挿入

ノード p を挿入する場合，オーバレイ上で p の挿入位置を検索し，リング管理アルゴリズムによりレベル 0 リングに挿入する．次に，以下の方法で finger table を更新する（3.2.6 節のシーケンス例と擬似コードも参照されたい）．

- p.FFT[$i+1$] ($i \geqq 0$) を更新するために，p.FFT[i] が指すノード q から q.FFT[i]（x とする）を取得する．
- その際，q.BFT[i] を p に更新する（**パッシブな更新 1**）．ただし，$i = 0$ の場合は除く（successor と predecessor はリング管理アルゴリズムで管理するため）．
- この時点ではノード x は生存が確認できていないが，次に x から x.FFT[$i+1$] を取得する際に x の生存が確認できるため，このときまで p.FFT[$i+1$] への x の格納を遅らせる．
- ノード挿入直後は FFT だけではなく BFT の更新も行う．その際，FFT の更新と BFT の更新は交互に行う（p は p.FFT[0]，p.BFT[0]，p.FFT[1], p.BFT[1] ... の順に finger エントリ取得リクエストを送信）．
- p.BFT[i] が指すノード q から q.BFT[i] を取得する場合を考える．このとき，パッシブな更新 1 により q.FFT[i] を p に更新する．また，q から見ると p.FFT[i] が指すノードは（finger table が収束していれば）時計回り方向に 2^{i+1} 離れたノードであり，直前に更新済み（生存も確認済み）であるため，q.FFT[$i+1$] を p.FFT[i] で更新する（**パッシブな更新 2**）．
- finger table の更新が一周したら（取得したノードが自ノードを超えたら），$T_{ft} \times$ random の後に次節で述べる定常的な更新に移る．ここで，random は 0 から 1 までの一様乱数であり，ノード挿入が集中した場合でも finger table 更新周期を分散させるために導入する．

3.2.2 定常的な finger table 更新

定常状態ではレベル 1 から順に周期 T_{ft} で p.FFT[i] の更新を行う．その際，パッシブな更新 1 により p.FFT[i] の指すノードの BFT[i] を p に更新する．パッシブな更新 2 は行わない．

3.2.3 リバースポインタの管理

ノード p がアクティブ更新に finger table エントリを更新してノード q を指すとき，パッシブ更新 1 により q の finger table エントリは p を指す．このため，各ノードはアクティブ更新およびパッシブ更新 1 で更新したエントリはリバースポインタ集合に追加する．

パッシブ更新 2 では，ノード p の FFT[i] が指すノード r は，ノード p の BFT[i] が指すノード q の FFT[$i+1$] によって指される．このため，r のリバースポインタ集合 $r.R$ に q を追加する．p は直前に r に r の FFT[i] を取得するための getEnt リクエストを送るため，この契機で $r.R$ に q を追加する（図 1 の “$R = R \cup Nx$”）．

ノード p の finger table エントリがアクティブあるいはパッシブに更新され，指しているノードが x から y に変化する場合，q の finger table の他のエントリに x が存在しないならば，x は q に removeRev リクエストを送信して x を削除する．

3.2.4 ノード削除

ノード p を削除する場合，まずリング管理アルゴリズムによりレベル 0 リングから削除する．その際，他ノードの finger table で p を指すエントリを，p の predecessor ノード q に付け替える（リング管理アルゴリズムが predecessor ノードの successor リンクを先に更新する場合を想定）．そのため，p は q に p のリバースポインタ集合 $p.R$ を送る．q は $p.R$ に含まれる各ノードに対して，p を指しているエントリを q に付け替えるためのメッセージを送信する．また，$q.R$ に $p.R$ をマージする．レベル 0 リングからノードを削除した後も，しばらくの間は p は削除直前に送信されたメッセージを受信する可能性があるため，ルーティング処理を継続する．

3.2.5 障害への対応

ノードの障害（削除手続きを実行せずに離脱する場合を含む）が発生した場合，レベル 0 リングの修復と finger table の修復が必要である．前者はリング管理アルゴリズムが行う．以下，後者の方法を述べる．

各 finger table エントリには，それが指すノード x に加えて x の successor list（x の時計回り方向のノード集合）も格納する．x の successor list は finger table 更新の際に x から取得する．ノード p が finger table エントリを参照してノード q にメッセージを送信するときは，q は p へ確認メッセージを送信する．p は一定時間以内に q から応答がなかった場合，当該エントリの successor list から代替ノードを選び，エントリを修正する．（紙数の関係で詳細は割愛する）．

3.2.6 シーケンス例と擬似コード

レベル 0 リングへの挿入後から最初の定常的な finger table 更新までのシーケンス例を図 1 に示す．ここでは，ノード N1 から N7 までが挿入され，finger table が収束している状態で，後から N0 が挿入された場合を示している．斜線はメッセージの転送，水平方向の破線は finger table エントリの更新（細い破線はアクティブ，太い破線はパッシブ）を表す．

getEnt が finger エントリを取得・更新するリクエストである．最初の 2 つの引数は，FFT あるいは BFT の i 番目の要素を取得する要求であることを示す．3 つ目の引数はリクエスト送信ノードへのポインタであり，受信ノード側のパッシブな更新 1 のために用いる．4 つ目の引数はリクエスト送信ノードからリクエスト受信ノードの反対方向に i 個離れたノードへのポインタであり，パッシブな更新 2 もしくはリバースポインタの追加に用いる．

N0 は，N1 から N2 へのポインタを受け取るが，これを N0 の FFT[1] に代入する処理は N2 に getEnt リクエストを送って応答を受信した後である．また，N6 はパッシブな更新 1 によって FFT[1] を N0 に，パッシブな更新 2 によって FFT[2] を N2 に更新している．

ノード n における finger table 更新処理の擬似コードを図 2 に示す．NodeAndKey はノードのロケータとキーの両方を含む型である．ノードを挿入する場合，オーバレイに参加している既知のノードから自身の挿入位置を検索した後，リング管理アルゴリズムによってレベル 0 リングに参加する．その後，`updateBoth(FFT, 0, null, null)` を実行して finger table を更新する．この処理が終了したら，$T_{ft} \times$ random 時間待ってから `updateFFT(0, null)` により finger table を定期的に更新する．

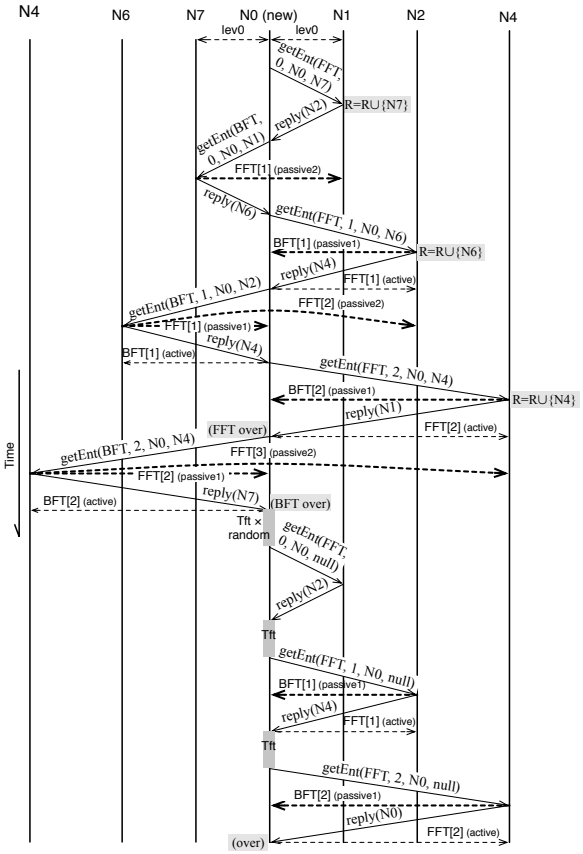

図 1　Suzaku の Finger Table 更新シーケンス
Fig. 1　Finger table update sequence of Suzaku.

4. 評　　価

4.1 シミュレーションの方法

メッセージレベルで動作する離散イベントシミュレータを実装した．シミュレータでは Suzaku，Chord$^{\#}$，Skip Graph を実装した．

すべてのアルゴリズムにおいて，リング管理アルゴリズムには筆者らが考案した DDLL [8] を用いた．ノード間の End-to-End 遅延時間はメッセージの大きさに関わらず 20ms，Chord$^{\#}$ と Suzaku の finger table の更新周期 T_{ft} は 1 分とした．

Chord$^{\#}$ では，レベル 0 リングへの挿入直後に predecessor ノードから finger table をコピーすることで finger table が空の期間を減らし，性能向上を図る．また，障害時の finger table 修復は Suzaku と同じ方法で行う．

Skip Graph では実装を単純にするため，レベル 1 以上の連結リストではノードの挿入・削除は一瞬で行われ，他のノード

```
1  const FFT = 0, BFT = 1;
2  // ft[FFT] is FFT, ft[BFT] is BFT
3  // ft[FFT][0] = successor, ft[BFT][0] = predecessor
4  var ft[2][]: Array of {Array of NodeAndKey};
5  var R: Set of NodeAndKey; // reverse pointers
6  n.updateBoth(dir, i, n1, n2) {
7    tab ← ft[dir];
8    if (i = 0) n1 ← ft[dir][0]; // succ or pred
9    if (n1 = null or n1 = n) {
10     x ← null; goto next;
11   }
12   x ← n1.getEnt(dir, i, n, ft[1 − dir][i]);
13   if (n1 failed) { we omit the detail of this case }
14   if (i ≠ 0) change(tab, i, n1, true); // active update
15   if (x ∈ [n, tab[i]]) x ← null; // circulated
16 next:
17   if (x = null and n2 = null) return;
18   if (dir = FFT) { n.updateBoth(BFT, i, n2, x); }
19   else { n.updateBoth(FFT, i + 1, n2, x); }
20 }
21 n.updateFFT(i, n1) {
22   if (i = 0) n1 ← ft[FFT][0]; // successor
23   x ← n1.getEnt(FFT, i, n, null);
24   if (n1 failed) { we omit the detail of this case }
25   if (i ≠ 0) change(ft[FFT], i, n1, true); // active update
26   if (x ∈ [n, ft[FFT][i]]) { // circulated
27     // truncate FFT and BFT to size i
28     for (j←i to ft[FFT].length−1)change(ft[FFT],i,null,false);
29     for (j←i to ft[BFT].length−1)change(ft[BFT],i,null,false);
30     sleep(T_ft);
31     n.updateFFT(0, null);
32   } else {
33     sleep(T_ft);
34     n.updateFFT(i + 1, x);
35   }
36 }
37 n.getEnt(dir, level, p, q) {
38   if (level ≠ 0) {
39     change(ft[1 − dir], level, p, true);// pasv. update 1
40   }
41   if (q ≠ null) {
42     if (dir = BFT) {
43       change(ft[1−dir], level+1, q, false);// pasv. update 2
44     } else R ← R ∪ {q};
45   }
46   return ft[dir][level];
47 }
48 n.change(tab, level, new, addtorev) {
49   old ← tab[level];
50   tab[level] ← new;
51   if (addtorev and new ≠ null) R ← R ∪ {new};
52   if (old ≠ null and (FFT and BFT do not contain old)) {
53     old.removeRev(n);
54   }
55 }
56 n.removeRev(p) {
57   R ← R \ {p};
58 }
```

図 2　Suzaku: finger table 更新アルゴリズム

Fig. 2　Finger table update algorithm of Suzaku.

の挿入・削除とは衝突しない．また障害検出時には経路表の修復は一瞬で行われるものとした（これらは Skip Graph にとって有利な条件である）．

すべての実験はまず最初のノード（初期ノード）を挿入してから開始した．

4.2　多数ノード挿入時の検索ホップ数

多数のノードを短時間で挿入した場合の検索ホップ数を測定した．初期ノード挿入後，ランダムな順に 255 ノードを挿入した．挿入完了時点を時刻 0（分）とし，以後乱数で選んだノード間の検索を 30 秒あたり 2000 回行った．時刻 0 から時刻 20 までの平均検索ホップ数の推移を図 3 に示す．時刻 i における値は検索開始時刻が $[i, i+0.5)$（分）のクエリの集約値である．また，図 4 に，挿入完了直後および経路表収束後（Chord$^\#$ は時刻 44，Suzaku は 47）における検索ホップ数の分布を示す．

挿入完了直後における Chord$^\#$ の最大ホップ数は 51 に達し

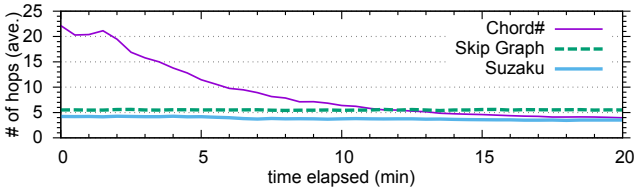

図 3　平均検索ホップ数の推移

Fig. 3　Changes of # of lookup hops.

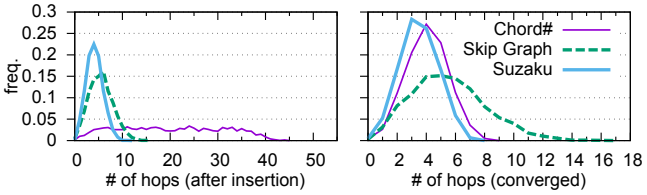

図 4　検索ホップ数の分布

Fig. 4　Distribution of lookup hops.

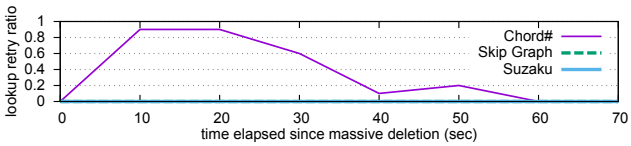

図 5　多数ノード削除時の検索リトライ率

Fig. 5　Lookup retry ratio in massive node deletion.

たのに対し，Suzaku は 10 であった．また，経路表収束時の最大ホップ数は，Chord$^\#$ は 8 $(=\lceil \log_2 256 \rceil)$，Suzaku では 7$(=\lceil \log_2 256 \rceil - 1)$ であった．また，すべての期間における Skip Graph の最大ホップ数は 17 であった．

Chord$^\#$ では挿入直後の検索ホップ数は大きく，finger table 収束に従い徐々に減少するが，Suzaku では挿入直後から収束値に近いホップ数で検索でき，Chord$^\#$ および Skip Graph よりも高速である．Suzaku は多数のノード挿入に対する耐性が高いと言える．

4.3　多数ノード削除時の挙動

多数のノードを短時間に削除した際の挙動を確認するため，次の実験を行った．256 個のノード（N0〜N255）を挿入し，経路表が収束した後，N32〜N96 を一斉に削除した．並行して 1 秒間に 1 回のペースで，N0〜N31 の範囲のノードから N97〜N127 の範囲のノードを検索した．このとき，削除済みノードに送ってタイムアウトの後に再送した検索の割合を求めた．1 回の検索で複数回のタイムアウトとなった場合も 1 回とカウントしている．10 秒ごとに集約した結果を図 5 に示す．横軸はノード削除完了時点からの経過時間である．

Chord$^\#$ では経路表から削除ノードが取り除かれないため，再送が必要な場合が多かったが，Suzaku と Skip Graph では再送の必要はなかった．Suzaku（と Skip Graph）は多数のノード削除への耐性があると言える．

4.4　検索対象の位置とホップ数との関係

近傍ノードを検索する場合の性能を確認するため，127 個のノード（N0〜N63，N65〜N127）をランダムな順序で挿入し，その直後に N64 を挿入してから，N64 からすべてのノードへ

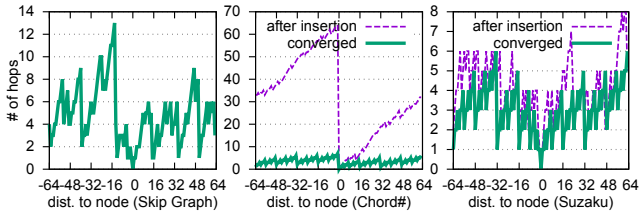

図 6　検索対象までの距離とホップ数の関係

Fig. 6　Relation between distance to dest. node and # of hops.

の検索に要するホップ数を測定する実験を行った．結果を図 6 に示す．横軸は N64 から見た検索対象ノードのキー (0〜127) の差である．Chord$^{\#}$ と Suzaku では，N64 挿入直後（破線）と finger table 収束後（実線）の 2 つを示している．

グラフより，(1)Chord$^{\#}$ は反時計回り方向の検索には時間を要し，特に挿入直後はノード数 n に対して $O(n)$ 時間を要すること，(2)Suzaku は挿入直後から両方向で近傍ノードを高速に（ノード数でカウントした距離の対数オーダで）検索できること，などを確認できる．

4.5　トラフィック

4.5.1　ノード挿入に要するトラフィック

ノードの挿入では，(1) 挿入位置の検索，(2) レベル 0 リングへの挿入，および (3) 経路表の構築，のそれぞれでメッセージを送受信する．(1) は 4.2 節で評価している．(2) はリング管理アルゴリズムによって異なり，DDLL では 3 である．

(3) は，Suzaku は $6\log_2 n$（finger table 収束時），Chord$^{\#}$ は 0（挿入時に経路表を構築しないため），Skip Graph は平均して概ね $6\log_2 n$ であり，Suzaku は Skip Graph と同程度のメッセージ数となる．（Suzaku は FFT と BFT をあわせたエントリ数は $2\log_2 n$，1 エントリの更新に getEnt，reply，removeRev の 3 メッセージ．Skip Graph は MV が一致するノードに到達するのに平均 2，その応答に 1，連結リストへの挿入に 3（DDLL を用いる場合））．

4.5.2　定常的なトラフィック

Chord$^{\#}$ と Suzaku は，T_{ft} 毎に finger table の更新を行う．Chord$^{\#}$ では 1 エントリの更新に 2 メッセージ（getEnt, reply）必要だが，Suzaku ではこれに加えて removeRev メッセージで 1 メッセージ必要である（ただしエントリに変更がある場合のみ）．

Chord$^{\#}$ では，短時間に多数のノードが挿入された場合の検索ホップ数を抑えようとすると，T_{ft} を小さくする必要があるが，Suzaku ではその必要はほとんどないため，Chord$^{\#}$ に比べて T_{ft} を大きくできると考えられる．ただし，どの程度大きくできるかについては未評価である．

Skip Graph では，構造を維持するために各レベルで隣接ノードの生存確認を行う必要がある（行わない場合，ノードが孤立する可能性が高まる）．このため，定常的なトラフィックは必要であるが，Suzaku との比較は今後の課題である．

4.6　構造の複雑さ

Suzaku が用いる経路表やリバースポインタ集合は Chord$^{\#}$ の finger table と同様，ノードへのポインタを格納するだけの簡単な構造である．このため，実装の難易度は Chord$^{\#}$ と大差はない（2.3 節参照）．

表 1　既存方式との比較

Table 1　Comparison with existing methods.

	Chord$^{\#}$	Skip Graph	Suzaku
反時計回り方向の近傍ノード検索	低速	高速	高速
検索ホップ数（収束後）	最大 $\lceil\log_2 n\rceil$	平均 $O(\log n)$	最大 $\lceil\log_2 n - 1\rceil$
検索ホップ数（最大）	$O(n)$	$\log_2 n$ の数倍程度	ほぼ $\log_2 n$
削除ノードへのルーティング	あり	なし	なし
構造	単純	複雑	単純

4.7　比　　較

Chord$^{\#}$，Skip Graph と Suzaku の比較表を表 1 に示す．

5.　おわりに

本稿で提案した Suzaku は，Chord$^{\#}$ の長所（経路表収束時の高速性，構造の単純さ）と Skip Graph の長所（Churn 時の性能劣化がない，キーの大小に関わらず近傍ノードは高速に検索可能）を兼ね備え，またこれらよりも高速という特徴を持つ．このため，さまざまなアプリケーションにとって有用であると考えられる．

今後の課題としては，より詳細な評価を行うこと，理論的な検索ホップ数の上界を求めることが挙げられる．

(謝辞) 本研究は JSPS 科研費 JP16K00135 の助成を受けている．

文　　献

[1] A. Yahyavi, et al., “Peer-to-peer architectures for massively multiplayer online games: A survey,” ACM Computing Surveys, vol.46, no.9, pp.1–51, 2013.

[2] R. Banno, et al., “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] X. Shao, et al., “Effective Load Balancing Mechanism for Heterogeneous Range Queriable Cloud Storage,” Proc. of CloudCom 2015, pp.1–8, 2015.

[4] I. Stoica, et al., “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] T. Schütt, et al., “Range queries on structured overlay networks,” Computer Communications, vol.31, no.2, pp.280–291, 2008.

[6] J. Aspnes and G. Shah, “Skip graphs,” ACM Trans. on Algorithms, vol.3, no.4, pp.1–25, 2007.

[7] T. Kawaguchi, et al., “Self-refining skip graph: A structured overlay approaching to ideal skip graph,” Proc. of COMPSAC 2016, pp.377–378, 2016.

[8] K. Abe and M. Yoshida, “Constructing distributed doubly linked lists without distributed locking,” Proc. of the IEEE Intl. Conf. on P2P Computing 2015, pp.1–10, 2015.