平成21年度 学士論文

# 柔軟な経路表による<br>オーバレイネットワークの設計

東京工業大学 理学部 情報科学科
学籍番号 06-16853

長尾 洋也

指導教員
首藤 一幸 准教授

平成21年2月3日

# 概要

ネットワークに接続されたコンピュータの普及とともに、Web サービスに対する要求が急激に高まっている。また、ネットワークの利用方法も多様化し、既存のネットワーク上に別の仮想的なネットワークを構成するオーバレイネットワークという手法が注目されている。

オーバレイネットワークの代表的な応用例に DHT(Distributed Hasth Table) がある。DHT とは、多数のコンピュータ上に一つの巨大な連想配列を構築する技術であり、耐障害性やスケーラビリティに優れていることが特徴である。そのような特徴から、BitTorrent に代表される Peer-to-Peer ファイル共有ソフトの基盤技術として利用されている。また近年では、データベースのキャッシュとして Web サーバの裏側で利用される事例が増加している。

DHT では、事前に各ノードに ID を割り振る。そして、割り振られた ID に応じて各データの保存先ノードが決定される。具体的な ID の割り振り方や、保存先ノードの決定方法、経路表構築方法などには様々な方式が考えられており、代表的なアルゴリズムとしては Chord[9], Kademlia[5], Tapestry[11], Pastry[7], Koorde[2], CAN[6] などがある。たとえば Chord では、ノード数 $N$ に対して $O(\log N)$ ホップで担当ノードに到達することができる。

しかし、これらのアルゴリズムの経路表管理手法は、ノード ID とノードの IP アドレス情報で構成されるノード情報を厳密に扱い、経路表のどの部分にどんな ID を持ったノード情報が保存されるかを固定してしまっている。その結果、経路表の構成手法に適さないノード情報がむやみに捨てられてしまう問題点や、経路表に保存するエントリ数が変更できないという問題点がある。また、経路表の管理に ID 以外の情報、たとえばネットワーク近接性等の情報を利用することが難しいという問題も存在する。

本研究では、DHT における柔軟な経路表を提案する。柔軟な経路表では、経路表エントリを厳密に決定するのではなく、経路表エントリの分布というより一般化した条件の下管理する。そのような管理を行うことで、Chord のような既存手法と同等のルーティング性能を維持したまま、経路表構成に柔軟性を与える。

また本研究では、柔軟な経路表を Chord の経路表管理に適用した FRT-

Chord を提案する。この FRT-Chord について経路表の性質や経路長に関する実験を行い、FRT-Chord が Chord と同様のルーティング性能を持つことを確認した。また、経路表のサイズを様々に変更させることで、Chord より短い経路長でのルーティングや、ノード数が少ないときにパラメータを変更することなくゼロホップ DHT が行われることを確認した。

# 目 次

# 図 目 次

# 表 目 次

# 第1章　はじめに

ネットワーク性能の向上とともにネットワークの利用方法が多様化し、既存の IP ネットワーク上に仮想的なネットワークを構築するオーバレイネットワークという手法が注目されている。

オーバレイネットワークの例として Peer-to-Peer ファイル共有システムや、アプリケーションレイヤマルチキャスト、CDN などが挙げられる。特に Peer-to-Peer ネットワーク構築に利用される DHT は、オーバレイネットワークの代表的な応用例である。DHT とは、数十から数万、数十万、数百万台規模のオーバレイネットワーク上に連想配列を構築する技術であり、耐障害性やスケーラビリティなどに優れている。この DHT は Peer-to-Peer に利用されるのみでなく、近年、クエリの処理速度向上のためのデータベースキャッシュとしての利用もみられる。

DHT の基本的な構造は次の通りである。はじめに各ノードへ固有の ID が割り当てる。次に、保存する key-value ペアの key に対しても同様に ID が割り当てられ、keyID の対応するノード ID を持つノードへデータが保存される。具体的なノードへの ID 割り当て方法や、保存先ノードの決定方法、ノードの lookup 方法などには様々なアルゴリズムが提案されており、代表的なものに Chord, Kademlia, Tapestry, Pastry, Koorde, CAN などがある。

しかし、これらのアルゴリズムで行われる経路表管理は経路表の構成に強い制限が存在し、経路表のどの部分にどんな ID を持つエントリを保存するかが厳密に固定されてしまっている。その結果、経路表の構成手法に適さないノード情報がむやみに捨てられてしまう問題や、経路表に保存するエントリ数が変更できないという問題がある。また、経路表の管理に ID 以外の情報を活用できないという問題も存在する。この ID 以外の情報としてはネットワーク近接性に関する情報や、ネットワークへの参加時間、ノードの性能などが挙げられる。

本研究では、これらの問題を解決すべく、“柔軟な経路表” を提案する。柔軟な経路表では、経路表エントリを制限し厳密に決定するのではなく、より拡張性に優れた柔軟な条件の下に経路表管理を行う。具体的には、遠くにあるノードより近くにあるノードを経路表へ積極的に取り込むという条件である。このような制限の少ない経路表管理手法を利用したとして

も、Chord のような既存手法と同等のルーティング性能を維持しつつより状況の変化や要求に柔軟に対応できる経路表が構築できる。

また本研究では柔軟な経路表の概念を Chord の経路表構成に適用した FRT-Chord を提案する。この FRT-Chord について実験を行い、FRT-Chord の性質を測定した。また、FRT-Chord と Chord とのルーティング性能の違いに関しても検討した。その結果、FRT-Chord が Chord と同等の経路長でルーティングを行うことが可能というだけでなく、経路表のサイズを様々に変更させることで Chord 以上の性能を引き出すことが可能であることを示した。また、ノード数が少ない場合にゼロホップ DHT が実現されることが示された。

本稿では、まず第 2 章で DHT の導入および、既存のアルゴリズムとその問題点を指摘する。続く第 3 章では提案手法である柔軟な経路表の概念を説明する。第 4 章では Chord に柔軟な経路表を適用したアルゴリズム *FRT-Chord* を提案する。第 5 章では、Overlay Weaver を用いた実験を行う。

# 第2章　研究背景

この章では、本研究が扱うオーバレイネットワークおよび、応用である DHT とその代表的なルーティングアルゴリズムの例について述べる。また、既存のルーティングアルゴリズムの問題点を指摘する。

## 2.1　オーバレイネットワーク

一般的にオーバレイネットワークとは、既存のコンピュータネットワーク上に構築された仮想的なネットワークのことである。オーバレイネットワークでは、下位層に当たるコンピュータネットワークを利用してコンピュータ間の通信を行う。

本研究では、インターネットや LAN などの IP ネットワーク上にアプリケーションを利用して構築された仮想的なネットワークのことを指してオーバレイネットワークと呼ぶ。このようなオーバレイネットワークの応用例に Peer-to-Peer がある。

## 2.2　DHT

DHT とは、オーバレイネットワークを構築することで複数のコンピュータ (ノードと呼ぶ) 上に構成された連想配列のことである。

データを複数ノードに分散配置する際に、DHT を利用することで次のような効果が得られる。

- 通信負荷やディスク容量の負荷などを分散させることができる (負荷分散)。
- 故障ノードやネットワークから離脱するノードが発生したとしても、その影響が全体に及ばない (耐故障性)。
- ノード数が増加したとしても、効率のよいルーティングを行うことができる。

DHT は、key と value のペアを保存し、key から対応する value を取り出す機能を提供する。key は hash 関数にかけられ、その返値に応じて保存する担当ノードが決定される。このとき、担当ノードがネットワークから離脱したとしても、担当ノードの変更が局所的で済むということが DHT の特徴であり、耐故障性を高めている。

### 2.2.1　代表的なルーティングアルゴリズム

DHT には多くのルーティングアルゴリズムが存在する。ここでは、代表的なアルゴリズムである Chord[9] および Kademlia[5] の経路表構成手法について言及する。

#### Chord

Chord[9] では、SHA-1 に代表されるメッセージダイジェスト関数を利用して各ノードに m ビットの ID を設定する。それぞれが仮想的に 0 から $2^m - 1$ までのリング状 ID 空間に配置されていると考える。ここでは、簡単のためにメッセージダイジェスト関数を SHA-1 とし、$m = 160$ とする。

ある ID からリング上を時計回りに進んで最初に出会うノードをその ID の Successor と呼ぶ。このとき、key の担当ノードは ID  SHA-1(key) の Successor ノードとなる。

Chord では、各ノードが Successor リストと Finger table の 2 種類の経路表を保持している (図 2.1)。Successor リストはノード自身の ID から時計回りに進んだときに遭遇するノードを順番に一定個数保持するリストである。また、Finger table はノード自身の ID を $s$ としたときに $i = 0, 1, ..., 159$ それぞれに関して $s + 2^i \pmod{2^{160}}$ の Successor ノードを探索して調べ、そのノードを Finger table の $i$ 番目のエントリとして保持する。

Chord では、ノード $s$ から lookup 先の目的 ID $t$ までの距離をノード $s$ から $t$ までリング上を時計回りに進んだ距離としており、その距離関数 $d$ は、

$$d(s,t) = \begin{cases} t - s & (t \geq s) \\ (t + 2^{160}) - s & (t < s) \end{cases}$$

と定義されている。

Chord のルーティングは目的 ID が減少するように ID リング上を時計回りにホップし、ホップ先の選択に Successor リストと Finger table の 2 つの経路表を利用する。

Successor リストは、ノードの頻繁な加入・離脱があったとしてもルーティングが正しく完了することを保証しており、Successor リストを管理することで、効率的ではないが最低限のルーティングを行うことができる。ただし、Successor リストのみのルーティングは総ノード数を $N$ とした時に経路長が $O(N)$ であり、効率がよくない。そこで、Chord は、もう一つの経路表である Finger table を利用する。

Finger table はルーティングの安定性を保つ Successor リストとは目的が異なり、経路長を短縮するための経路表である。ルーティング時に Finger table 中の目的 ID 直前のエントリノードにホップすることで目的 ID までに存在するノード数を最低でもおよそ半分にすることができ、その結果、総ノード数を $N$ が十分大きいとき高い確率で経路長が $O(\log N)$ となることが証明されている。

### Kademlia

Kademlia[5] も Chord 同様に、SHA-1 ハッシュ値などを利用して各ノードに 160 ビットの ID を設定する。ただし、Chord とは異なり、距離として XOR を利用する。したがって、2 ノード間の距離 $d(s,t)$ は、

$$d(s,t) = s \ \text{XOR} \ t$$

と表すことができる。

Kademlia の経路表は、k-bucket と呼ばれる小さな経路表エントリの配列から構成される。経路表には k-bucket が 160 個存在し、$i(0 \leq i < 160)$ 番目の k-bucket は自ノードからの距離が $I_i = [2^i, 2^{i+1})$ に含まれるノードを最大 k 個保持している。この経路表を用いて、Kademlia のルーティングは次のように行われる。目的 ID を $t$ とし、$t \in I_i$ であるとき、$i$ 番目の k-bucket に含まれているエントリを長さ $\alpha$ の問い合わせリストの初期エントリとして登録する。エントリが $\alpha$ 個に満たない場合は、近くの k-bucket からエントリを追加する。このとき、エントリは $t$ に近い順に並び替えられる。次に、問い合わせリストの先頭ノードに $j$ 番目の k-bucket の内容を問い合わせる。ただし、その先頭ノードの ID を $u$ としたとき $j$ は $u \in I_j$ を満たす値である。受け取ったエントリを問い合わせリストに追加し、$t$ までの距離が近い順に $\alpha$ 個までを問い合わせリストに保持し、残りは問い合わせリストから取り除く。この問い合わせ作業を繰り返し、問い合わせリストに含まれるノードすべてが問い合わせ済みになった時点で探索を終了する。そして、その問い合わせリストに残っているノードが担当ノードとなり、それぞれに value が対応づけられる。

### 2.2.2　既存のルーティングアルゴリズムの問題点

Chord や Kademlia をはじめとする既存のルーティングアルゴリズムの問題点について述べる。

#### ネットワーク近接性が考慮されない

代表的なアルゴリズムはノードの ID のみを基準にルーティングテーブルの構成やフォワーディングを行っており、ノード間の通信遅延や通信可能性が考慮されておらず、必ずしも適切なルーティングが行われないことが多い。これを解決するためには、ネットワーク近接性を経路表構築時、もしくはフォワーディング時、ノードへの ID 割り当て時に考慮する必要がある。

#### 経路表の厳格性

既存のアルゴリズムでは、経路表の特定のエントリには特定 ID のノードの情報しか保存されず、経路表の構築方法に合致しないノード情報は次々と捨てられてしまう。ノード情報を管理するのに必要なメモリや帯域が十分にあるならば、むやみにノード情報を捨てることはとても非効率である。

実際に、ルーティングアルゴリズムはノード数に対していかに小さい経路表を構築するかが重要視されており、保持するノード情報を削減するためにその多くが捨てられてしまっている。Chord の場合はそれぞれの範囲 $I_i$ につき 1 つのノード情報しか保存することができず、その範囲のノードについてどれだけ情報を取得しようとも、そのうちもっとも手前に存在する 1 つのノード情報以外は捨てられてしまう。また、Kademlia の場合は範囲 $I_i$ に保存されるノードはすべての $i$ に関して $k$ 個と定められており、$i$ が大きい範囲に関しては頻繁にノード情報を捨てることになってしまう。

#### 経路表サイズの動的変更が出来ない

既存のアルゴリズムでは、経路表の管理トラフィックの削減や経路長の短縮など、状況に応じた経路表サイズの動的な変更が出来ない。Kademlia において、$k$ を増減させることにより経路表サイズを変更することが可能ではあるが、$k$ の値はアルゴリズム上のさまざまな部分で共通の値であり、かつネットワークに属しているすべてのノードに関して共通のものであるため、安易に変更することができない。また Chord においては、Finger table の最大サイズは ID のビット長に固定されており、変更が不可能であ

る。Successor リストを増加させることは可能であるが、経路長短縮の目的には適合しない。

#### ゼロホップ DHT が実現されない

ゼロホップ DHT とは、経路表内に全ノードの情報を持つことで、ノードを探索するためのホップが不要な DHT のことである。DHT アルゴリズムは、ノード数が小さい場合は、経路表に全ノードの情報を持ち、ゼロホップ DHT を実現すべきである。しかし、Chord の場合はノード数が少ない場合であっても経路表エントリが溢れてしまい、ゼロホップ DHT は実現されない。同様に、Kademlia の場合には $k$ を $N/2$ 以上まで大きくすることでゼロホップ DHT が可能であるが、ノード数に合わせた $k$ の設定が必要であるだけでなく、$k$ の設定が正しく行われたとしてもデータ構造に大きな無駄が発生してしまう。また、$k$ の設定が増加した場合など、ノード数の変化に応じて動的に変更しなければならないが、前述の通り難しいといえる。また、もともと DHT アルゴリズムはノード数不明の状態で正しく動作することが求められる観点からしても Kademlia によるゼロホップ DHT の実現は困難である。

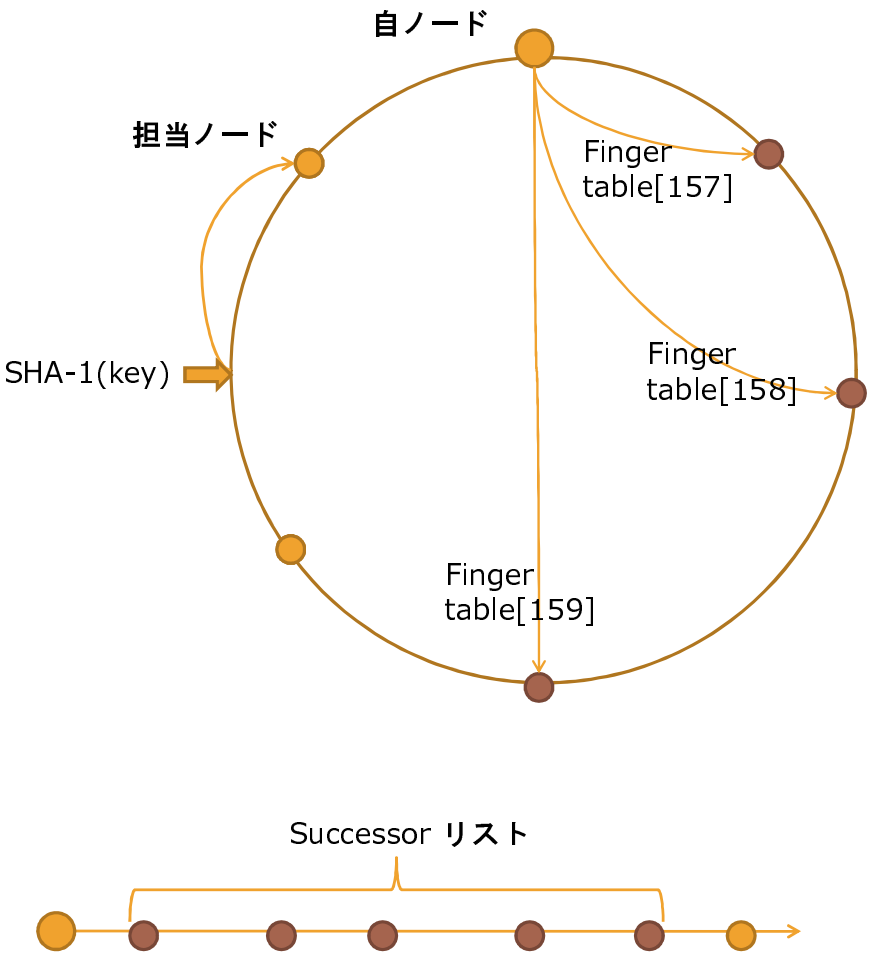


図 2.1: Chord の Successor リストと Finger table

# 第3章　柔軟な経路表

柔軟な経路表とは、経路表エントリを厳格に管理する従来手法とは対照的に、経路表を柔軟に管理する手法である。この柔軟な経路表には、経路表の拡張を容易にし、ノード情報の効率的な利用が可能となる点など、さまざまな利点がある。また、これらの利点は、従来の DHT アルゴリズムに求められる特性を維持しつつ実現される。

## 3.1　経路表エントリの分布

経路表を保持しているノードからの距離に関してどのように経路表エントリが分布しているかがルーティング効率に大きな影響を与える。ここでは、その分布について議論を行う。

### 3.1.1　ID 空間上の距離

DHT ノード数を $N$ とする。それぞれのノードには、0 から $2^m - 1$ までの $m$ ビット整数値 ID がランダムに割り当てられている。そのとき、ID 空間上の距離を表す関数 $d$ を 2 変数の実数値関数として定義する。

$$D = 0, 1, ..., 2^m - 1$$
$$d : D \times D \to \mathbb{R}$$

このとき、距離関数 $d$ は、

$$s, t, u \in D \quad , \quad d(s,t) \geq 0$$
$$s = t \Leftrightarrow d(s,t) = 0$$
$$d(s,t) + d(t,u) \geq d(s,u)$$

という性質を持つ。ただし、

$$d(s,t) = d(t,s)$$

とは限らない。

また、ノード ID $s$ のノードから各ノードへの距離関数 $d_s$ を

$$s, t \in D, d_s(t) = d(s, t)$$

と定義する。

Chord や Kademlia といったアルゴリズムの経路表と同様に、柔軟な経路表はノードからの距離関数に基づいて構成される。

### 3.1.2　Chord・Kademlia の経路表分布

2.2.1 のように、ID 長を m ビットとすると、Chord の距離関数 $d$ は

$$d(s,t) = \begin{cases} t - s & (t \geq s) \\ (t + 2^m) - s & (s > t) \end{cases}$$

と定義される。Chord の 2 つの経路表のうち、経路長短縮を担う Finger table の各経路表エントリ $x_i (i = 0, 1, ..., m-1)$ は、「$d_s(t) \geq 2^i$ を満たす $t$ のうち、$d_s(t)$ が最小となる $t$ である」と定義されている (図 3.1)。

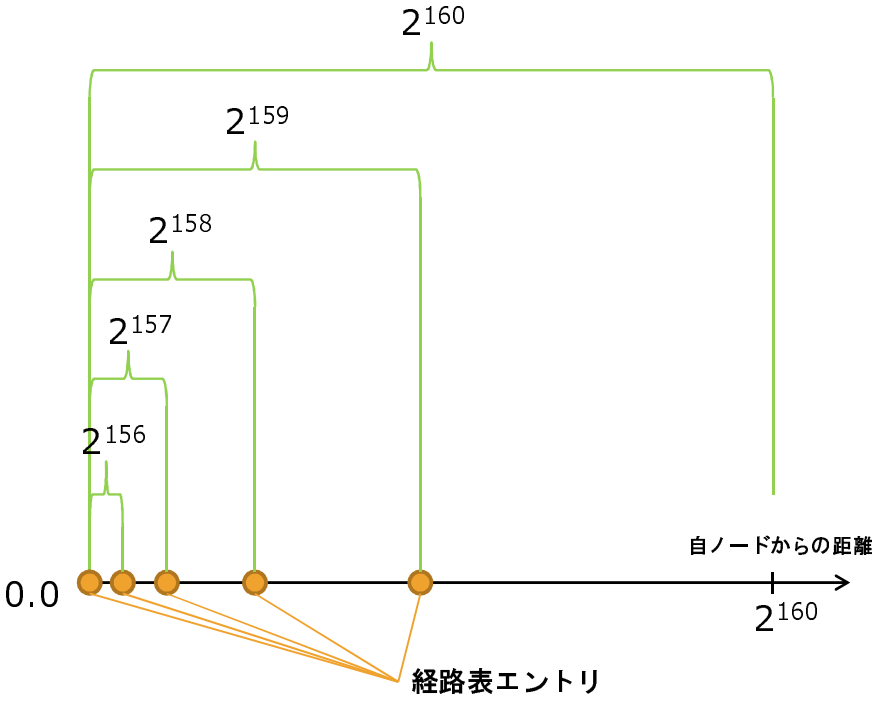


図 3.1: Chord(Finger table) の経路表エントリ分布 (m=160)

また、Kademlia において距離関数 $d$ は、

$$d(s,t) = s \text{ XOR } t$$

と定義され、$i = 1, ..., m$ それぞれに対して $2^{i-1} \leq d_s(t) < 2^i$ を満たすエントリ $t$ が $k$(定数) 個存在する。

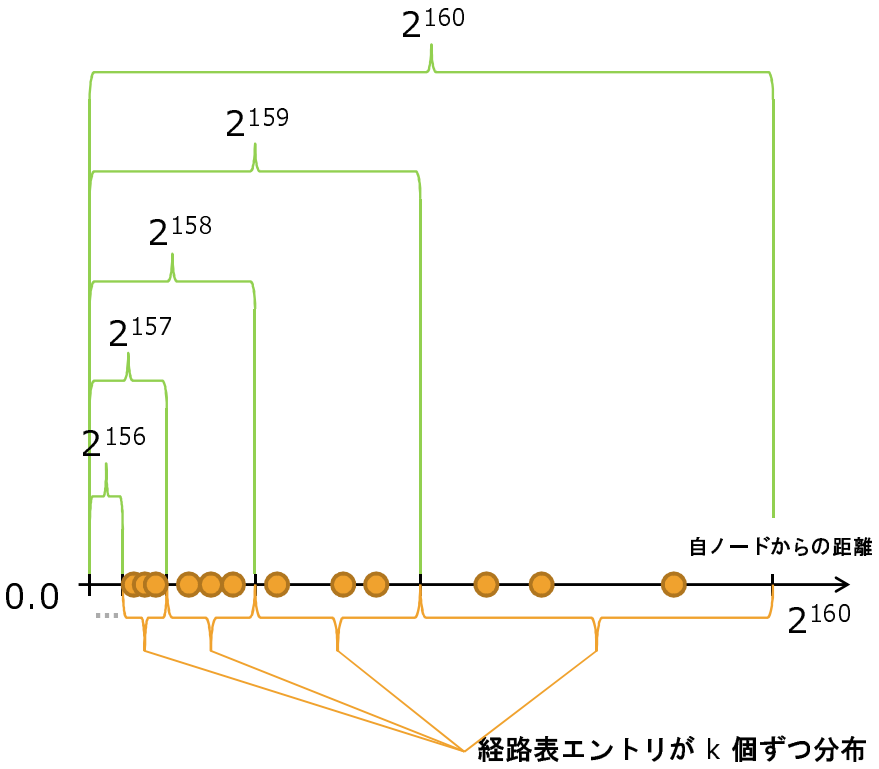


図 3.2: Kademlia(k-bucket) の経路表エントリ分布 (m=160)

これら 2 つのアルゴリズムには、自ノードからの距離が小さい (近い) ノードが経路表に多く含まれるという共通点が存在する。この共通点はより具体的に言えば、$2^{i-1} \leq d_s(t) < 2^i$ を満たすエントリ $t$ が定数個含まれるという性質のことである。

ここで、距離 $d_s(e) = x$ 以内に存在する経路表エントリ $e$ の経路表エントリ全体に対する割合をノード分布関数 $F(x)$ で表すと、Chord および Kademlia のノード分布には

$$F(2^i) - F(2^{i-1}) = (\text{一定})$$

を満たすという共通の性質があるといえる。

このように経路表分布を維持することで、小さい経路表であっても経路長が小さく抑えられていることに我々は注目する。

### 3.1.3　目標分布関数

Chord や Kademlia においては、経路表分布が $2^i$ という区切りごとに管理されていた。これに対して柔軟な経路表では、分布関数の性質を

$$F(2x) - F(x) = (\text{一定}) \tag{3.1}$$

という区切りのない性質へ拡張し、この性質を満たすことを目標とする。ここで、

$$\bar{F}(x) = \frac{1}{m}\log x \quad (1 \leq i \leq 2^m) \tag{3.2}$$

を柔軟な経路表の目標分布関数と呼ぶ。ここで $\bar{F}(x)$ は式 (3.1) を満たす (式 (3.3))。

$$\bar{F}(2x) - \bar{F}(x) = \frac{1}{m}\log 2x - \frac{1}{m}\log x = \frac{1}{m} = (\text{一定}) \tag{3.3}$$

このとき、経路表の目的密度関数 $\dfrac{d}{dx}\bar{F}$ は

$$\frac{d}{dx}\bar{F}(x) \sim \frac{1}{x}$$

であり、これは文献 [3] が利用する密度関数と一致している。

## 3.2　経路表エントリの管理方法

柔軟な経路表のエントリは、エントリ分布が $\bar{F}(2x) - \bar{F}(x) = (\text{一定})$ に近づき、目的分布 $\bar{F}(x) = \frac{1}{m}\log x$ に近くなるように管理される。

### 3.2.1　経路表エントリの追加

経路表へのエントリ追加方法を追加アルゴリズムと呼ぶ。この追加アルゴリズムは、応用目的や管理指針によりいくつかの方式が考えられる。また、これらの方式は組み合わせることが可能であるが、メモリ消費量やトラフィックの増加に注意する必要がある。

**Join 時に追加**

Join 時に ID の近いノードから経路表エントリを受け取り、自身の経路表へ追加する。一般的に、DHT アルゴリズムではネットワークへの Join 時に自身のノード ID について lookup を実行し、担当ノードを探索する。このとき、ID 的に Join 中のノードと担当ノードとは近く、目的となる経路表の内容も近くなると考えられるため、経路表エントリをその担当ノードから受け取ることで効率よくエントリを収集することができる。ただし、その各エントリとの通信を行ったわけではないので、通信可能性は不明である。

**通信時に追加**

必要なノードとの通信が成功した際に、随時そのノード情報を経路表へ追加する。柔軟な経路表はノード情報を高効率で利用することを重要視しており、できる限り経路表エントリを入手したい。そこで、lookup 時や、Join の通信時を中心としたあらゆる通信時にノード情報を経路表へ追加する。このとき、通信に “成功した” ということが重要である。

**ノード情報を得るための lookup を実行する**

必要ではない lookup を行い、通信したノードを追加する。定期的に、もしくは経路表エントリの状態に応じて自主的に lookup を実行し、ノード情報を収集する。この方式では、経路表が必要としている ID のノード情報を取得できるという利点がある一方、実行間隔を十分に確保せずに実行すると、トラフィックが増大し、本来必要な通信を妨げる可能性があるので注意が必要となる。

**経路表エントリの交換**

ノードとの通信の際に、経路表エントリを交換する。もしくは、経路表エントリを交換するために通信を行う。経路表エントリをより多く追加するために、積極的にノード情報を入手する方法である。この方式では、通信量や通信回数が増加するというデメリットがある。また、受け取った経路表エントリを何のチェックも行わずに経路表へ追加することは好ましくない。つまり、通信可能性の評価や、すでに存在していないノードの情報が転送され続けてしまうことを防ぐ必要がある。他にも、悪意のあるノードがネットワーク中に存在した場合、経路表を交換するシステムを通じて他のノードの経路表を破壊することを防ぐ必要がある。

### 3.2.2　経路表エントリの削除

経路表エントリの削除は、通信失敗時および、経路表エントリ数 $l$ が経路表エントリ数の最大値 (経路表サイズ)$L$ を越えたときに実行される。

### 3.2.3　削除する経路表エントリの決定方法

経路表エントリの追加時に $l = L$ であった場合、$L$ 個の既存エントリと追加されようとしている 1 エントリの中から 1 つを選択し、削除を行う。このとき、どのエントリを削除するかが重要である。

このとき、$e$ の選択理由には以下のような指針が考えられる。

**通信不能ノード**

通信不能ノードを選択する。通信不能となったノードは、いつまた通信可能となるか予想が付かないばかりでなく、通信可能となった後、再び通信不能となる可能性が高いと予想されるため、優先的に削除されるべきである。

**目的分布**

経路表分布が目標分布関数により近づくように選択する。もっとも中心となる削除方針であり、このように削除することで、経路長を短縮し、高いルーティング効率の維持を行う。

**ネットワーク近接性**

ノードからのレイテンシが大きいノードを選択する。ID 的に適切なノードであったとしても、レイテンシが著しく大きい場合、フォワーディング先として好ましくない。そのため、前回通信時のレイテンシを保存しておき、削除ノードの決定時に参考にすることが考えられる。

**通信経路**

ネットワーク上の特定範囲で通信が完結するノードが残るように選択する。たとえば、無線ネットワークと有線ネットワークが存在するとき、無線ネットワークに属するノードは無線ネットワーク内で通信を優先的に行い、有線ネットワークに属するノードは有線ネットワーク内で通信を優先的に行うようにしたい。もしくは、他プロバイダ回線を極力利用しない経路を利用するようにしたい場合などが考えられる。そのような場合に、同一ネットワーク上のノードを経路表に残すようにすれば、異なるネットワークをまたぐ通信の削減につながる。ただし、このように経路表構成を生かすには Recursive なルーティングが必要となる。

**ネットワーク参加時間**

ネットワークへの参加時間が長いノードを選択する。これは、ネットワークへの参加時間が長いノードの方が、短いノードと比較して今後もネットワークへ参加し続ける時間が長いという傾向が知られているためである。つまり、このように選択することで、通信不能ノードの発生そのものを防ぐ効果がある。

**その他**

その他のノード情報から削除ノードを選択する。各ノードには、さまざまな特徴があり、それらを利用して削除ノードを選択する。

これらの削除方針に基づき削除を実行するアルゴリズムを削除アルゴリズムと呼ぶ。削除アルゴリズムが、柔軟な経路表を特徴付けることになり、非常に重要といえる。

また、このように様々な要因を考慮して経路表を構築可能であることがこの柔軟な経路表の大きな特徴である。

## 3.3　ルーティング方法

柔軟な経路表を用いたルーティングは基本的に、フォワーディング先ノード $h$ を $d(h,t)$ がもっとも小さくなるように経路表エントリから選択する。削除アルゴリズム同様、このフォワーディング先選択アルゴリズムも、ルーティング性能に直結する。

フォワーディング先の選択基準はの基本は ID 間距離であるが、削除アルゴリズムの例として紹介したレイテンシ情報なども利用可能である。経路表エントリの最大数 $L$ を大きくとることで、互いに ID が近いエントリを複数保持することが可能となり、それらから ID 以外の要因を考慮してフォワーディング先を選択することが出来る。

また、ルーティング方式には大きく分けて Iterative と Recursive の 2 通りの方式が考えられ、それぞれ利点・欠点が存在する。

### Iterative ルーティング

Iterative ルーティングは、lookup を実行するノードがホップ先と順番に通信する方式である。この方式を利用することで、自身の経路表に含まれないノードとの通信が多く発生し、新たなノード情報を効率よく取得することができる。一方で、lookup を開始したノードがホップ先ノードそれぞれと直接通信できる必要があるため、基本的に全ノードが互いに通信可能であることが求められる。

#### Recursive ルーティング

Recursive ルーティングは、それぞれのホップ先が lookup を次々と再帰的に行う方式である。この方式では、担当ノードに到達したとき、再帰的にホップした経路を戻るのではなく直接 lookup を開始したノードと通信を行うことで Iterative ルーティングと比較してレイテンシを短縮することができる。また、経路表エントリにレイテンシの小さいノードを登録する場合、経路表エントリノードと直接通信する Recursive ルーティングでなければそのレイテンシの小ささを生かすことができない。その反面、経路表に含まれないエントリノードとの通信が少なく、新しいノード情報は入手しにくい。

## 3.4　柔軟な経路表の利点

このように構成した柔軟な経路表には以下のような利点がある。

#### ノード数の増減に対応可能

ノードが少なく、$L \geq N - 1$ を満たす時、全ノード情報が経路表に登録されることから、パラメータを変更することなくゼロホップ DHT が実現される。つまり、一つの経路表構築アルゴリズムを Peer-to-Peer のような数十万から数百万ノード規模のネットワークだけでなく、サーバの裏側でサービスを支えるような数ノードから数十、数百ノードまでのネットワークの構築に利用できることになる。既存のアルゴリズムではノードが少ない場合であっても、このように経路表へすべてのノード情報を効率よく登録することはできない。

#### ノード情報の有効利用

経路表の構成方法が柔軟であるため、ノード情報をむやみに捨てることを防ぐことが出来る。従来のアルゴリズムでは、経路表の各スペースに保存されるエントリの ID に制限があり、それを越えたノード情報を保存できない。一方で柔軟な経路表では、経路表の総エントリ数が制限されているのみで特定の ID 空間に対する制限が存在しない。

#### 経路表サイズの動的な変更が可能

経路表サイズを各ノードが個別に変更することができる。これにより、ノードの性能やネットワーク負荷、要求されるレイテンシなどに応じて

lookup レイテンシを小さくするために経路表を拡大したり、省メモリや更新作業を抑えるために経路表を縮小することが可能となる。そのように経路表サイズを変更したとしても、柔軟な経路表はサイズによらず経路表の分布を一貫して適切に維持しようとするため、常によりルーティング効率のよい経路表エントリを保持しておくことが可能である。

#### トラフィックの有効利用

入手したノード情報を積極的に取り入れる。経路表の都合に合わせた経路表を作るための通信を行うのではなく、通常に行う通信から得られるノード情報を経路表に生かすことができる。これにより、経路表メンテナンスのためのトラフィックの増大を防ぐとともに、トラフィック量を有効利用することが出来る。

#### 経路表が容易に拡張可能

経路表の維持に削除アルゴリズムを利用することで、ノード ID 以外の様々なノード情報を利用した経路表構築が容易となる。これは、経路表に様々な情報を取り入れるという拡張の容易さを表している。

#### フォワーディング先ノードを柔軟に選択可能

経路表サイズを大きくすることで、ID 的に大きな違いのない複数のフォワーディング先候補から、ID 以外のノード情報を考慮して最終的なフォワーディング先を選択、もしくは複数ノードへフォワーディングすることが可能となる。

# 第4章　FRT-Chord:柔軟な経路表のChordへの適用

代表的なルーティングアルゴリズムである Chord に柔軟な経路表という概念を適用したルーティングアルゴリズム *FRT-Chord* を提案する。

## 4.1　基本となるアルゴリズム

FRT-Chord のアルゴリズムは Chord のアルゴリズムを基本とし、経路表の管理・構築に柔軟な経路表の概念を利用したものである。そのため、この章では Chord との相違点を中心に扱う。また、このアルゴリズムではさまざまなノード情報のうちノード ID(ビット長 $m = 160$) を利用したルーティングを行う。

基本となるアルゴリズムとして Chord を選択したのにはいくつか理由がある。一つ目は、先行研究が多く、経路長が高い確率で $O(\log N)$ になること等、いくつかの性質に関して証明が与えられていることが挙げられる。二つ目は、Chord が比較的シンプルなアルゴリズムであること、三つ目としては、DHT 実装に広く利用されており利用実績が豊富であることが挙げられる。これらの理由により、本研究においては Chord を選択したが、柔軟な経路表は Chord に限らずさまざまなルーティングアルゴリズムに対して利用できると私は考えている。

## 4.2　追加アルゴリズム

FRT-Chord の追加アルゴリズムは、すべての通信をトリガとして経路表へノード情報の追加を行う。lookup だけでなく、フォワーディングやネットワークへの加入 (Join) 時など、様々な通信時に得たノード情報を経路表へ追加する。また、Join 時に初期ルーティングテーブルとして Successor ノードから経路表エントリ情報を受け取る。また、必要でない lookup や、経路表エントリの交換は実行しないものとする。

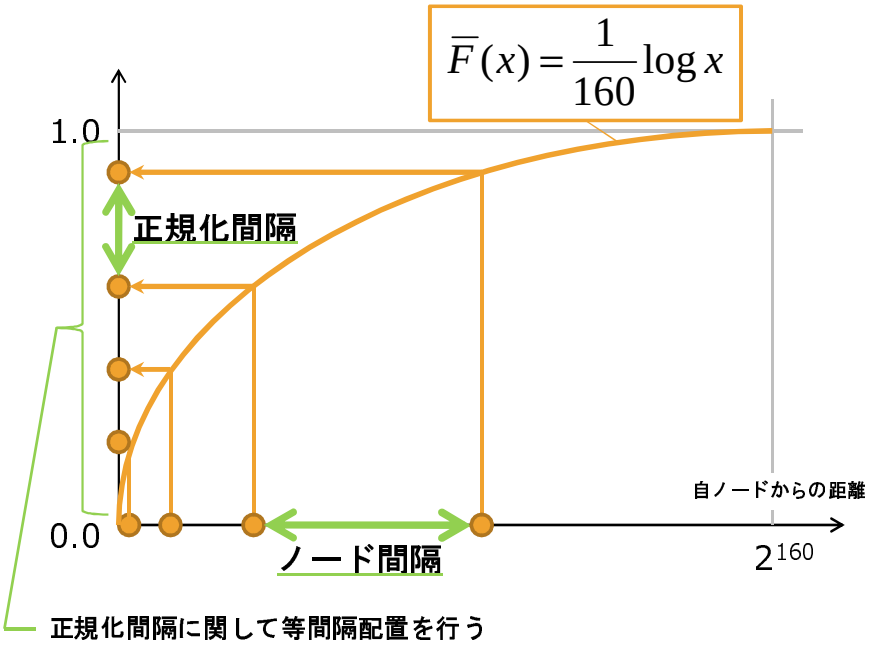


図 4.1: FRT-Chord の分布管理手法

## 4.3　削除アルゴリズム

経路表エントリの削除は、通信の失敗時およびエントリ追加時に経路表サイズ $l$ がその最大サイズ $L$ を越える場合に実行される。

経路表サイズが大きくなりすぎたとき、目標分布 $\bar{F}(x) = \frac{1}{160}\log x$ に従うように削除する経路表エントリを選択する。

ここで、分布に従うように削除ノードを選択するにあたり、経路表エントリ間の ID 間隔に注目し、ID 間隔がある条件を満たすように調節する。

ここで、経路表エントリを自ノードから近い順に $x_0, x_1, ..., x_{l-1}$ とし、$x_{i-1}$ と $x_i$ との正規化ノード間隔 $C_i$ を

$$C_i = \bar{F}(d_s(x_i)) - \bar{F}(d_s(x_{i-1}))$$

と定義する (図 4.1)。

FRT-Chord は、この正規化間隔 $C_i$ が均一となるように削除エントリとして $x_j$ $(C_j = \min_i C_i)$ を選択するとする。つまり、正規化間隔が最小となるエントリを取り除き拡大することで、正規化間隔の均一化を図る。また、Successor ノードとなる $x_0$ を取り除かないために、$C_0 = \infty$ とする。さらに、Chord の Successor リストによる安定性を維持するために、経路

表の先頭から $k$ 個のノードを保持し続けるようにする。結果 $C_i$ の最終的な定義は次のようになる。

$$C_i = \begin{cases} \infty & (0 \leq i < k) \\ F(d_s(x_i)) - F(d_s(x_{i-1})) & (k \leq i) \end{cases}$$

経路表への追加や削除が発生したとき、$C_i$ の変更は局所的であり、また、$C_i$ を昇順で管理することで、削除対象エントリの探索も効率よく行われる。従って、この削除アルゴリズムは時間効率よく実行されるといえる。つまり、経路表を柔軟にしたとしても、管理コストのオーダーが大きく増すことはないといえる。

## 4.4　ルーティング方法

ルーティングは、経路表へのエントリ追加頻度の多い、Iterative ルーティングとする。目的 ID を $t$ とすると、経路表の中から $d_s(x_i) < d_s(t)$ を満たす最大の $i$ を取得し、$x_i$ へホップを行う。これをホップ先ノードと Successor ノードとの間に $t$ が来るまで繰り返し、最後のホップをその Successor ノードへ行いルーティングは終了する。Chord では Finger table で目的 ID へ近づいた後、Successor リストに切り替える必要があるが、FRT-Chord では経路表が一つに統一されているため、そのような切り替えは不要である。

## 4.5　Join 方法

Join 時には、他のルーティングアルゴリズムと同様に、自 ID の担当ノードを探索する。この探索中も常に通信相手を経路表へ登録する。担当ノードの探索の完了後、担当ノードの経路表エントリのうち、ノード近くのエントリを定数個取得する。

# 第5章　評価

この章では、Overlay Weaver 上に FRT-Chord を実装する。次にその実装を利用して実験を行い、提案手法である FRT-Chord が既存の DHT アルゴリズムに求められる性質を満たしていることを確認する。また、ノードが少ないときにゼロホップになるなど FRT-Chord が Chord より優れている点を確認する。

## 5.1　実装

この章では、Overlay Weaver[12, 8] を利用した FRT-Chord の実装について詳しく述べる。

### 5.1.1　FRT-Chord の Overlay Weaver 上での実装

FRT-Chord を、オーバレイ構築ツールキットである Overlay Weaver 上に実装した。

Overlay Weaver 上に新たなルーティングアルゴリズムを実装するためには、最低限以下の API を Java 言語を用いて実装するのみでよい。少ないコードでシミュレータのみでなく実ネットワーク上で動作するアルゴリズムを記述できることが Overlay Weaver の特徴である。

- BigInteger distance ( ID to, ID from)
  ノード from からノード to までの距離を返す。
- IDAddressPair[ ] closestTo ( ID target, int maxNumber, RoutingContext context)
  経路表エントリのうち、ノード target に近いノード配列を返す。
- IDAddressPair[ ] rootCandidates ( ID target, int maxNumber)
  経路表エントリのうち、target を担当するノードの候補配列を返す。
- IDAddressPair[ ] adjustRoot (ID target)
  経路表を用いて target へ向けての接近が自ノードで終了したとき、target の担当ノードの候補配列を返す。

- void join ( IDAddressPair[ ] neighbors)
  Join 時に Join を行ったノードで呼び出される。

- void join ( IDAddressPair joiningNode, IDAddressPair lastHop, boolean isRootNode)
  Join 時に、Join 先 ID への Hop 中に経由したノード上で呼び出される。

- void touch ( IDAddressPair from)
  from ノードと通信を行った際に呼び出される。

- void forget ( IDAddressPair node)
  node ノードとの通信に失敗したとき、呼び出される。

代表的なアルゴリズムである Chord, Kademlia, Pastry, Tapestry, Koorde などの実装が Overlay Weaver に含まれているため、それらの実装を参考にすることで素早く実装することが可能である。

### 5.1.2　経路表のデータ構造

自ノード ID を $s$、経路表に保持されている ID 集合を $X$、経路表の最大サイズを $L$ とする。経路表エントリは経路表内の検索効率のために常に $d_s$ に関して昇順に管理される。このとき経路表エントリノード ID を先頭から $x_0, x_1, ..., x_{|X|-1}$ とおく。また、それらの正規化間隔 $\{C_i\}_{i=0,...,|X|-1}$ をノードの追加・削除にあわせて更新し、正規化間隔リストと呼ぶ。また、正規化間隔リストは昇順に管理され、それぞれのノード $C_i$ には、同じインデックス値を持つエントリ $x_i$ への参照を保持させておく。

ノードの追加時に $|X| = L$ であった場合、正規化間隔リストの先頭 $C_{i_0}$ を参照し、その参照から $x_{i_0}$ を取得し、削除を行う。この処理のコストは $O(1)$ である。

経路表へのノード追加時、追加された経路表エントリが $x_a$ であるとすると、新たに計算する必要がある正規化間隔は $C_a$ および $C_{a+1}$ の最大 2 つである。また、ノード $x_b$ の削除時には、新たに計算する必要がある正規化間隔は $C_{b+1}$ のみである。よって、経路表への追加・削除に対して、正規化間隔の計算量は $O(1)$ となる。

経路表への追加・削除時、正規化間隔リストに計算した正規化距離を追加するだけでなく、古くなった $C_{a+1}$ や $C_{b+1}$ を取り除く必要がある。この作業のためにあらかじめ $x_i$ それぞれに正規化間隔リストの要素 $C_i$ への参照を持たせておくことで、正規化間隔リストからの削除も $O(1)$ で実行可能である。

以上のことから、経路表および正規化間隔リストの更新にかかるコストは昇順に並べられたリストの探索コスト $O(\log L)$ に抑えられ、管理コストが Chord に対してオーダー的に不利になっていないといえる。

## 5.2　実験環境

実験では、Overlay Weaver 上に実装した FRT-Chord アルゴリズムを利用する。また、今回は実験を 1 台のマシン上におけるシミュレーションにより行う。各ノードの挙動管理は、Overlay Weaver 用のシナリオファイルを Perl により生成し、それを利用した。

- Overlay Weaver 0.9.7
- JRE build 1.6.0_17-b04
- OS:Windows Vista SP2 32bit
- CPU:Intel Core 2 Duo E8400 3.00GHz
- メモリ:4.00GB

## 5.3　実験 1:経路表構成手法の検証

提案手法による経路表管理を実行することで経路表エントリ分布が

$$\bar{F}(2x) - \bar{F}(x) = (\text{一定}) \tag{5.1}$$

に近づくことを確認する。

### 5.3.1　実験内容

ノード数 $N = 10000$ とし、経路表サイズ L=5, 10, 20, 30, 40, 50, 100 それぞれに関して、ランダムに選択した 1 ノードからランダムに選択した ID への lookup を 10000 回実行する。実行後、ノードの経路表の分布を取得し、目標分布との比較を行う。また、FRT-Chord の $k$ の値を 1 と設定する。

目標分布との比較に際し、取得した経路表エントリの分布関数 $\hat{F}$ を

$$\hat{F}(x) = \frac{1}{|X|} \sum_{d=1}^{\lfloor x \rfloor} \hat{f}(d) \quad (1 \leq x \leq 2^m)$$

とする。ここで、$\hat{f}$ は、

$$\hat{f}(d) = \begin{cases} 1 & (s + d \bmod 2^m \in X) \\ 0 & (s + d \bmod 2^m \notin X) \end{cases}$$

である。この経路表分布関数を目標分布関数

$$F(x) = \frac{1}{m} \log x \quad (1 \leq x \leq 2^m)$$

と比較する。

### 5.3.2　実験結果および考察

それぞれの経路表サイズごとの分布関数を経路表エントリの位置ごとにプロットしたものが図 5.1 および図 5.2 である。

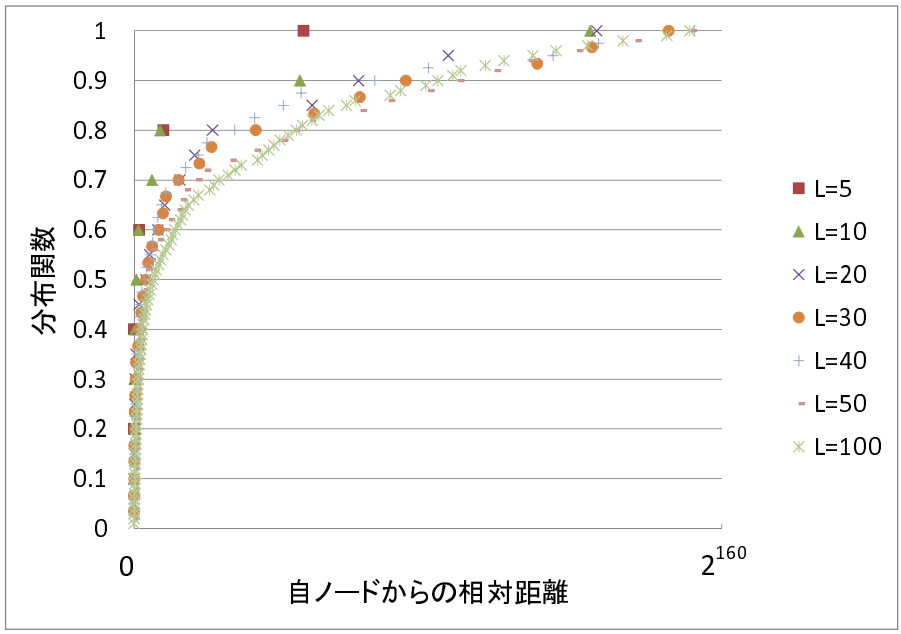


図 5.1: 分布関数

#### 経路表エントリ分布

図 5.2 において、$x$ の小さいところで経路表エントリの分布が疎になっている。これは、ノードが密に分布しておらず、経路表のもっとも近くに

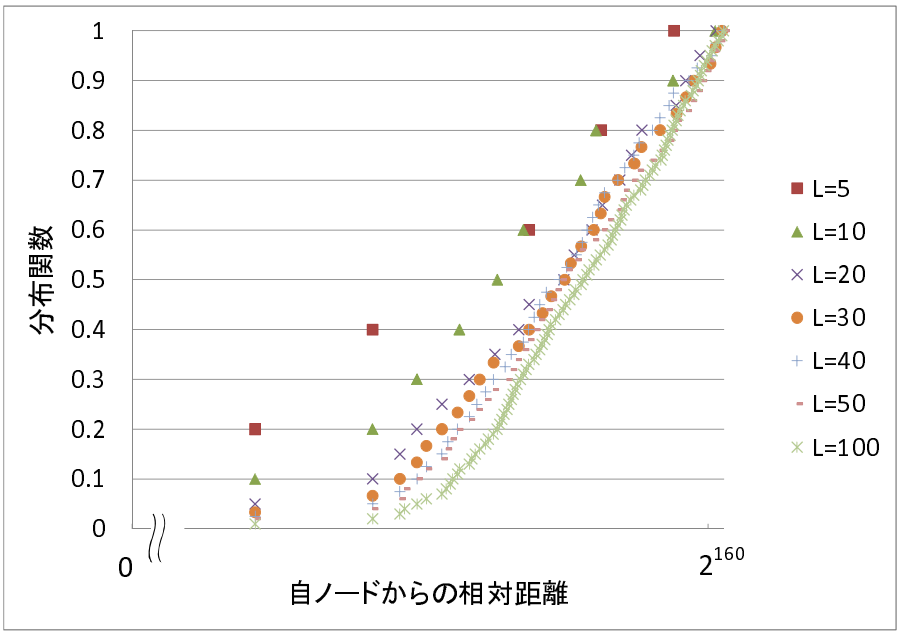


図 5.2: 分布関数 (log スケール)

ある Successor ノードまでノードが存在しないことによる。したがって、$\hat{F}_L(x)$ の一番左の点は同じ Successor ノードが経路表に含まれていることを示している。

経路表エントリの先頭部分を除けば、$\hat{F}_L(x)$ が一番左の Successor ノードを除き、ほぼ直線上に分布していることから、もともとの目標である

$$\bar{F}(2x) - \bar{F}(x) = (\text{一定})$$

という特徴が達成されていることが確認できる。ここで、経路表中の二番目のエントリが一番目から離れているのは、一番目である Successor ノードと二番目ノードとの間にもともとノードが存在せず、二番目のエントリノードが Successor の Successor であるからである。また、$L$ が大きい経路表の場合、二番目のエントリと三番目のエントリとの間隔も大きくなっているが、これもその間にノードが存在しないことに起因する。$L$ が大きい場合にこの特徴が見られるのは、$L$ が大きいほどエントリ間隔が狭くなる力が働いているからであり、エントリ間隔を狭くしようとしてもノードが存在しない部分に関してはほかの部分に比べてエントリ間隔が開いてしまうからである。ただし、この範囲で分布が条件 $\bar{F}(2x) - \bar{F}(x) = (\text{一定})$ に近づいていないとしても、その範囲の担当ノードが経路表に含まれている以上、直接ホップできるため問題にならない。

このことから、提案手法では、ネットワーク上のノード数などのパラメータを利用することなくノードから近い部分と遠い部分とで異なる特徴を持っていることが確認できる。これは、Chord が Successor リストと Finger table を別々に構築していることと比較して興味深い特徴である。

以上より、自ノード近くを除いた部分において、ノード分布の特徴がうまく実現されていることが示された。

### 5.3.3　結論

この実験により、提案した経路表管理手法によって経路表分布が $\bar{F}(2x) - \bar{F}(x) = (\text{一定})$ に近づくことが確認できた。

## 5.4　実験 2:経路表の更新と経路長の関係

提案手法では、経路表エントリ数が経路表サイズ $L$ に達するまでは単純にノード情報を追加し続け、$L$ を越えた時点で目標分布に従うような削除を繰り返し行い、経路長の短縮を試みる。本実験では、経路表がエントリで埋まること、および経路表の更新が経路長短縮へ貢献していることを示す。

### 5.4.1　実験内容

ノード数を $N = 1000$ とし、経路表の最大サイズを $L = 100$ として 20 万回の lookup を実行する。各 lookup は全ノードから無作為に選択したノードが実行し、lookup 対象 ID も同様に無作為に選択した。このとき、lookup の経路長を計測する。

### 5.4.2　実験結果および考察

実験結果をグラフにしたものが図 5.3 である。

全 20 万回の lookup を 100 回ごとに区切り、その区間それぞれに関して平均経路長および平均経路表エントリ数を集計したものである。横軸は lookup 回数を示している。縦軸は 2 つあり、左の縦軸が経路長を、右の縦軸が経路表エントリ数を示す。2 つのグラフのうち減少傾向にある方が左の縦軸に対応する平均経路長であり、増加傾向にある方が右の縦軸に対応する平均経路表エントリ数である。

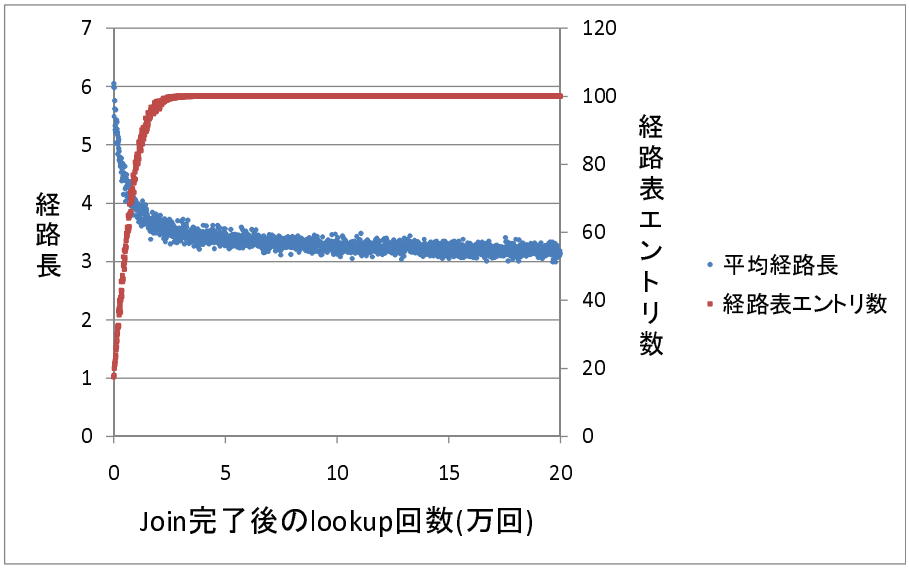


図 5.3: lookup 回数と平均経路長の関係 (N=1000, L=100)

### Join 直後に関して

ネットワークへの Join が完了した直後、経路表中には平均約 18 エントリが存在している。一方、Join 完了直後に行われた 100 回の lookup において、平均経路長は 5.99 回であった。このことから、Join の完了直後の時点で 1000 ノード中をわずか経路長 6 で到達したことになる。Join 直後ですでに約 18 エントリが存在したのは、ノードの Join 自体が lookup を含んでいるからである。また、そのとき通信相手は ID のランダム性からランダムに選ばれたノードであり、通信相手が偏らず、提案手法の更新作業を行わずとも十分に効率的なホップができたものと考えられる。

### 経路表エントリ数が最大に達するまでに関して

実験結果より、3 万回の lookup を実行した時点で経路表エントリ数がその最大数である $L = 100$ にほぼ到達している。経路表エントリを 100 取得するために 3 万 lookup を必要としているが、この 3 万回の lookup は 1000 ノードそれぞれが実行するのではなく、1000 ノードの合計であるので、実際には 1 ノード平均 30 回程度の lookup しか実行していないことになる。つまり、各ノードが 30 回ずつ lookup 作業をするのみで経路表に 100 エントリを追加することが可能であることが分かる。このことから、経路表エントリを埋めるための通信を行わずとも、非常に大量のノード情報を取得できることが確認できる。

また、経路表エントリ数が増大するにつれて、平均経路長が急激に減少していることが分かる。

### 経路表エントリ数が最大に達して以降に関して

経路表エントリ数が最大に達し、これ以上経路表エントリ数が増えることがなくなって以降も平均経路長が減少し続けていることがグラフから読み取れる。

これは、経路表に残すエントリを提案手法の削除アルゴリズムに従って選択し、より効率のよい経路表になるように修正を繰り返している成果である。提案手法により、同じ経路表サイズであっても、その経路表サイズにおけるよりよい経路表へ修正し、結果その経路表サイズを十分に利用した経路長短縮が行われていることが示されている。また、経路表エントリ数が最大に達した直後、平均経路長が大きく減少しているのは、エントリ数が最大に達した直後はエントリ分布の調節がほとんど行われていないからである。

### 最大経路長の推移に関して

5.3 のグラフに、100 回ごとの最大経路長を加えたものが図 5.4 である。平均経路長の減少とともに、各区間の最大経路長も減少していることが

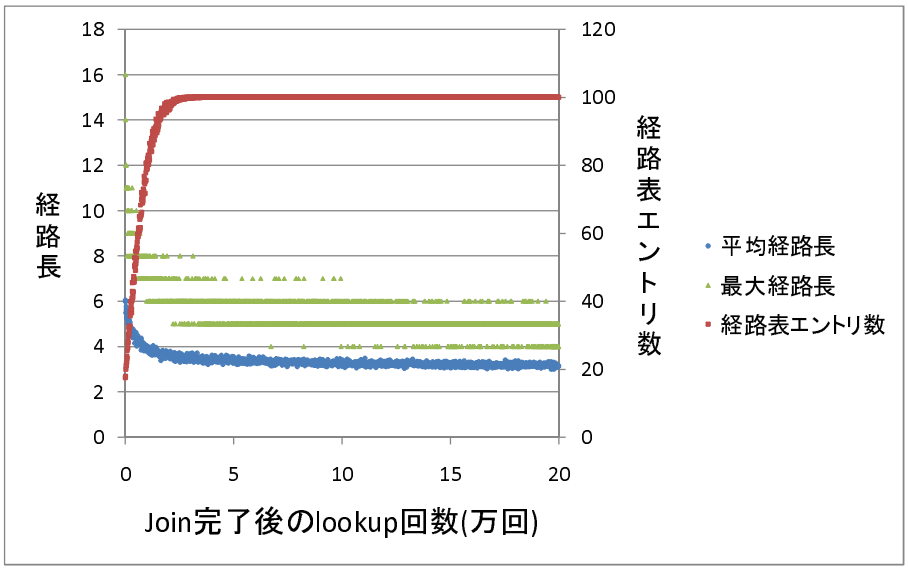


図 5.4: lookup 回数と平均経路長・最大経路長の関係 (N=1000, L=100)

はっきりと読み取れる。

### 5.4.3　結論

この実験から、提案手法による経路表の更新が経路長を確実に短縮していることが示された。また、少ない lookup 回数であっても経路表エントリを埋めることが可能であることが示された。

## 5.5　実験 3:異なる経路表サイズにおける経路表更新と経路長変化の関係

5.4 節の実験では、経路表サイズ $L = 100$ で実験を行い、経路表更新と経路表変化の関係について議論した。ここでは、より小さい経路表サイズ $L = 10$ を指定したときに、どのような違いが見られるかを評価する。

### 5.5.1　実験内容

5.4 節の実験における経路表サイズを $L = 10$ として実験を行う。このときノード数は $N = 1000$ である。

### 5.5.2　実験結果および考察

実験結果を 5.4 節と同様の方法で集計したものが図 5.5 である。

#### 経路表エントリ数について

$L = 10$ の経路表エントリ数は、Join 直後から最大値である 10 に達していることがグラフから読み取れる。これは、経路表サイズが小さい場合は Join に伴う lookup のみで経路表を埋めることが可能であることを示している。この点は $L = 100$ の場合と異なる。

#### 経路長に関して

経路長は、Join 終了直後におよそ 7 である。そして十分時間が経過した後、経路長平均はおよそ 6 となる。経路長に関して $L = 100$ のときと比較すると、次のようなことが分かる (図 5.6)。

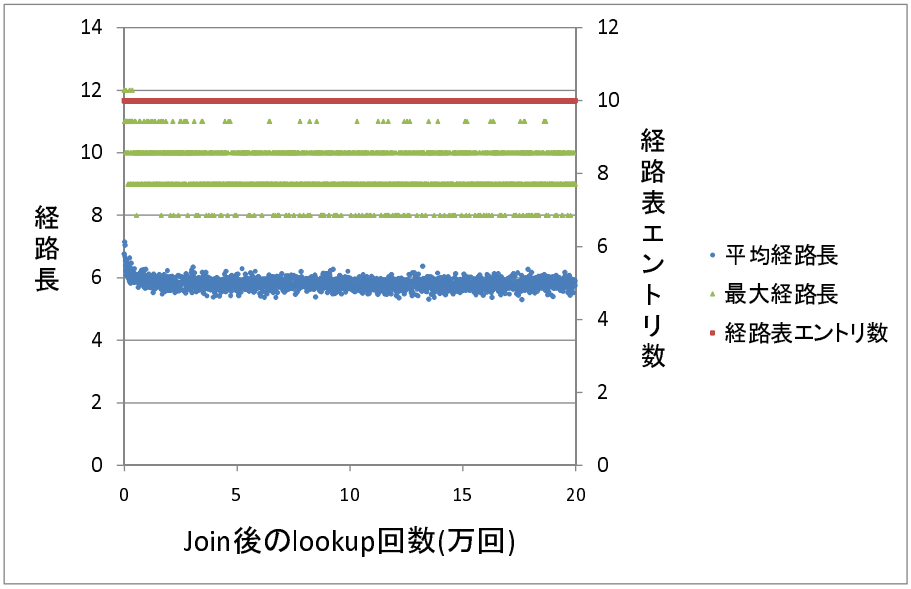


図 5.5: 経路長比較 (N=1000, L=10)

- Join 終了直後の時点で、$L = 100$ のとき平均経路長がおよそ 6 であるのに対し、$L = 10$ のときおよそ 7 と、大きな差はないが、経路表エントリ数の増加に伴い、その差が大きくなる。これは、$L = 100$ の経路表に含まれるエントリの数が急激に増加していることによる。
- 経路表が安定し、経路長が一定してくるまでにかかる時間に大きな違いが見て取れる。これは、経路表の最大サイズが小さい場合の方がより早く経路表が安定することを示している。
- 図 5.6 を見ると、平均経路長のばらつきに違いがあることが確認できる。このことから、経路表サイズを大きくすることで経路長を短縮するだけではなく、経路長のばらつきを抑えることも可能であることが読み取れる。

### 5.5.3　結論

この実験から、経路表サイズの大小により、経路長が安定するまでにかかる lookup 回数や、経路長のばらつきに違いがあることが確認できた。一方で、経路表の大小に依らず提案手法の経路表管理により経路長を短縮できていることが示された。

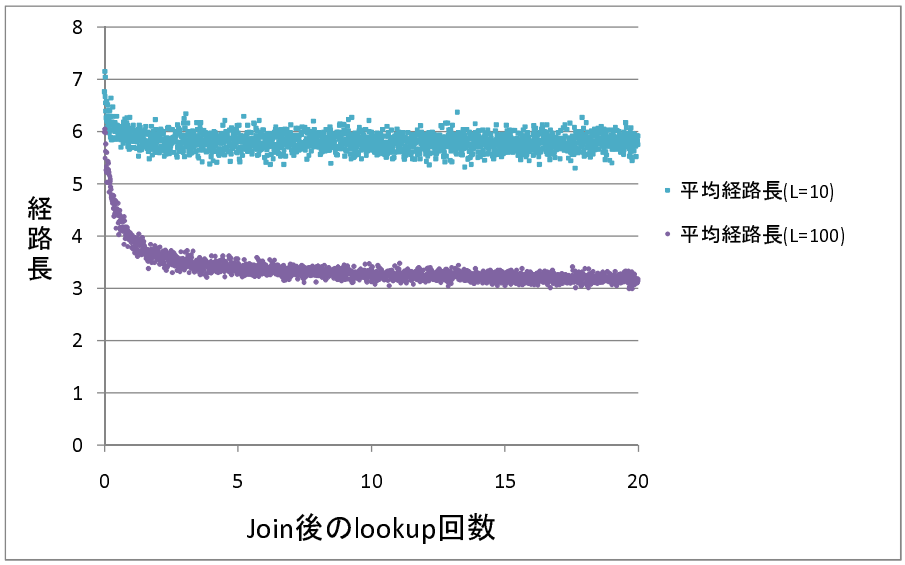


図 5.6: 経路長比較 (N=1000, L=10, 100)

## 5.6　実験 4:経路表サイズの違いによる経路長の変化

経路表サイズの設定により、経路長にどのような違いが出るかを測定し、Chord と同等の経路長や Chord より短い経路長でのルーティングが行われることを示す。また、経路長と経路表サイズとのトレードオフを柔軟に変更可能であることを示す。

ここでは、FRT-Chord との比較対象として Chord, Koorde, Kademlia の経路長も同じ条件の下測定する。

### 5.6.1　実験内容

ノード数は 5.4 節と 5.5 節同様 $N = 1000$ とし、経路表サイズ $L$ を 5, 6, 7, 8, 9, 10, 11, 20, 30, 40, 50, 100 と変化させ実験する。ここで、$L = 5$ に関して、FRT-Chord が $k = 1$ の場合 Successor ノードの Successor ノードの情報を経路表から取り除いてしまうことが Overlay Weaver 上で問題となるので、$k = 2$ とし、取り除かれないようにする。

5.4 節の結果を踏まえ、経路長測定前に十分な lookup を行うこととした。経路表サイズが小さいほどすぐに経路表が安定し、$L = 100$ において 20 万回の lookup で十分と考えられるため、それぞれの $L$ について lookup を 20 万回繰り返すこととする。安定化の 20 万 lookup の終了後、1 万回

の lookup を実行し経路長を測定した。lookup はランダムに選択したノードからランダムに決定した ID への探索とする。

### 5.6.2　実験結果および考察

#### FRT-Chord と他のアルゴリズムとの比較

それぞれにおける FRT-Chord(L=5) の経路長および Chord、Koorde、Kademlia の経路長のグラフが図 5.7 である。

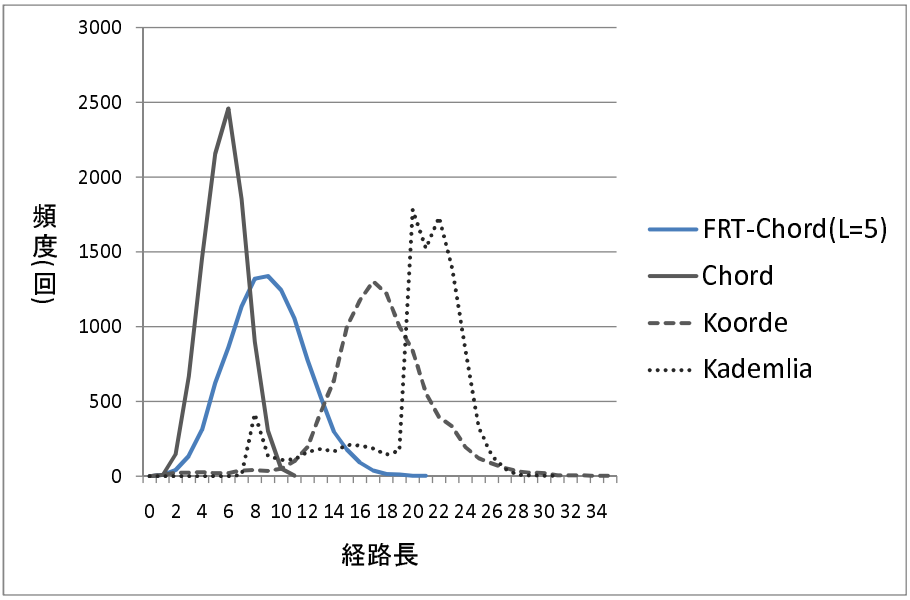


図 5.7: 経路長の度数分布 [FRT-Chord(L=5), Chord, Koorde, Kademlia]

このグラフにおいて、経路長の大小はパラメータによるものなので大きな意味を持たないことに注意する。このグラフから、Chord と FRT-Chord、Koorde が非常に似た形をしていることが分かる。一方で Kademlia のみ経路長分布特性が異なっている。これは、FRT-Chord と Koorde がともに Chord をベースとしたアルゴリズムであるからであると考えられる。

#### FRT-Chord と Chord との比較

図 5.8 は、L=5, 6, 7, 8, 9, 10, 11 の FRT-Chord と Chord の経路長分布である。このグラフから、FRT-Chord の経路表サイズ $L$ を変更することで柔軟に経路長分布を変更可能であることが分かる。特に、今回の実験

条件下においては、FRT-Chord の経路表サイズを $L = 10$ とすることでほぼ Chord と同等の経路長特性を持たせることが可能であるといえる。

また、それぞれのグラフの形が Chord に類似していることから、それぞれの経路表サイズに関して Chord と似た傾向があると読み取れる。

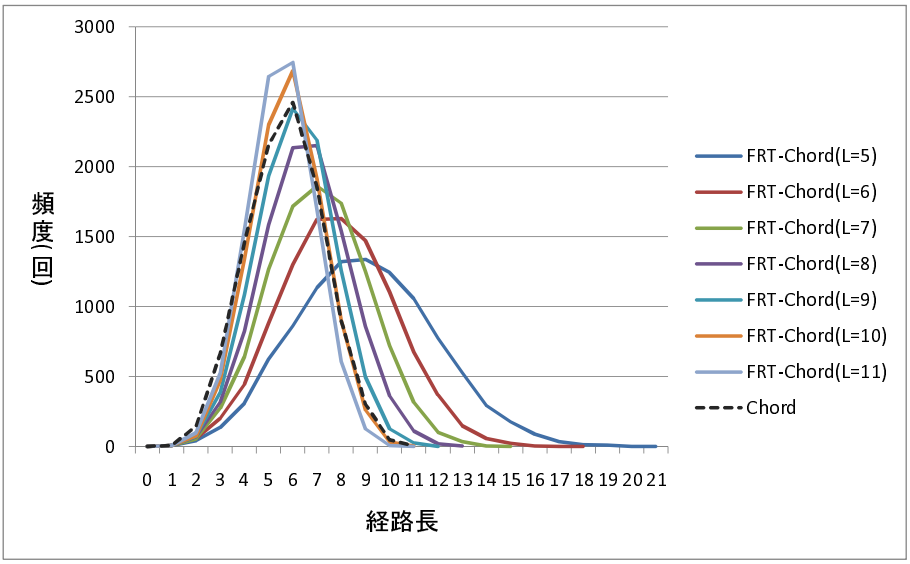


図 5.8: 経路長の度数分布 [FRT-Chord(L=5,6,7,8,9,10,11),Chord]

**FRT-Chord の経路表サイズ L の変化による経路長への影響**

図 5.9 は、L=5, 10, 20, 30, 40, 50, 100 の FRT-Chord および Chord の経路長分布である。

このグラフから、経路表サイズを変更することで、Chord より経路長を短くしたり長くしたりという経路長特性の変更が可能であると分かる。特に経路長を 10 より大きくした場合に、FRT-Chord が Chord よりも経路長に関して優れていることが分かる。また経路表サイズを大きくすることで平均経路長を短縮するだけでなく経路長のばらつきや最大経路長が小さくなっていることが確認できる。特に経路表サイズが小さい場合には経路表サイズを大きくすることによる経路長特性への影響が大きいことがわかる。

このことから、FRT-Chord の経路表サイズ $L$ を変更することで、経路長と経路表サイズのトレードオフを柔軟に変更可能であることが分かる。Chord において、Finger table のサイズを変更することは出来ない。また、

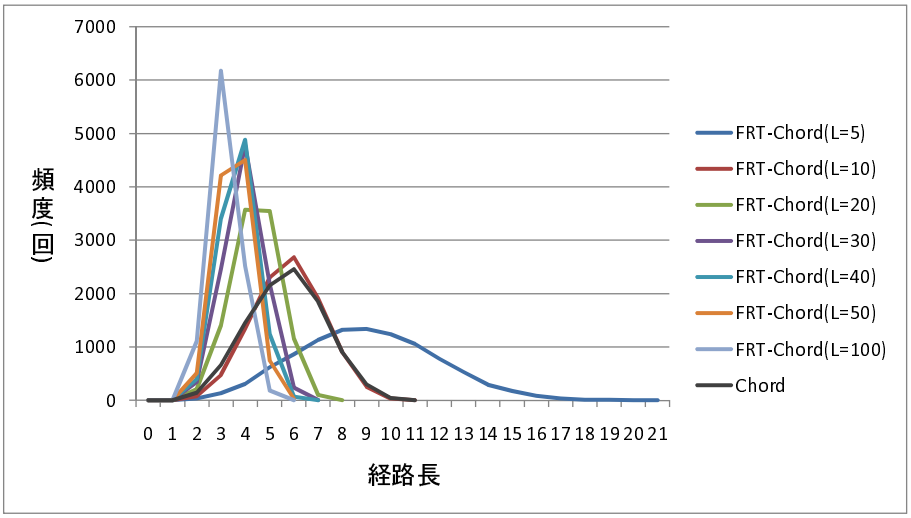


図 5.9: FRT-Chord の経路表サイズと経路長分布の関係

Successor リストのサイズを大きくすることでは、このトレードオフを実現することは出来ない。

### 5.6.3　結論

この実験から、経路表サイズを変更することで、Chord と同様の経路長でルーティングが可能であることが示された。特に、$L = 10$ という小さい経路表サイズにおいても Chord 相当の経路長が得られた。また、経路表サイズを大きくすることで、Chord より優れた経路長分布でのルーティングが可能であることも同様に確認され、経路表サイズを変更することで経路長と経路表サイズのトレードオフを柔軟に調整できることが示された。

## 5.7　実験 5:ノード数と経路長の関係

ノード数を $N$ としたとき、Chord の経路長は $O(\log N)$ となる。この利点を FRT-Chord も同様に持つことを、Chord の経路長と比較することにより確かめる。

### 5.7.1　実験内容

ノード数 N に対する経路長の変化を見るため、$N$ を 10, 100, 1000, 10000 と変化させて実験を行う。また、経路表の最大サイズは Chord と同様の

160 の場合と、比較的小さな 20 という値を利用する。まず経路表の構成のため 20 万回の lookup を実行する。そしてその後新たに 1 万回の lookup を実行し、経路長を測定する。

### 5.7.2　実験結果と考察

実験結果は表 5.1 および表 5.2 である。また、それらをグラフにしたものが図 5.10 および図 5.11 である。

FRT-Chord および Chord の lookup では目的 ID 直前のノードへのルーティング後、担当ノードである Successor への最後の 1 ホップを行うため、全ノード情報が経路表に登録されている状態での最大経路長は、経路表エントリノードへの経路と Successor ノードへの経路を合わせた 2 であり、1 とはならない点に注意が必要である。

| N | Chord | FRT-Chord(L=20) | FRT-Chord(N=160) |
|---:|---:|---:|---:|
| 10 | 1.99 | 1.89 | 1.89 |
| 100 | 3.74 | 2.95 | 1.99 |
| 1000 | 5.72 | 4.41 | 3.00 |
| 10000 | 8.46 | 6.78 | 5.06 |

表 5.1: ノード数と平均経路長の関係

| N | Chord | FRT-Chord(L=20) | FRT-Chord(N=160) |
|---:|---:|---:|---:|
| 10 | 3 | 2 | 2 |
| 100 | 7 | 5 | 2 |
| 1000 | 11 | 8 | 6 |
| 10000 | 17 | 14 | 11 |

表 5.2: ノード数と最大経路長の関係

#### 経路長オーダーに関して

図 5.10 から、平均経路長に関して FRT-Chord が Chord より優れていることが分かる。また、同様に図 5.11 から、最大経路長に関しても FRT-Chord が Chord より優れていることが分かる。

よって、経路長が $O(\log N)$ である Chord と同様に FRT-Chord のグラフが対数軸で直線的であることに加え、FRT-Chord が最大経路長に関し

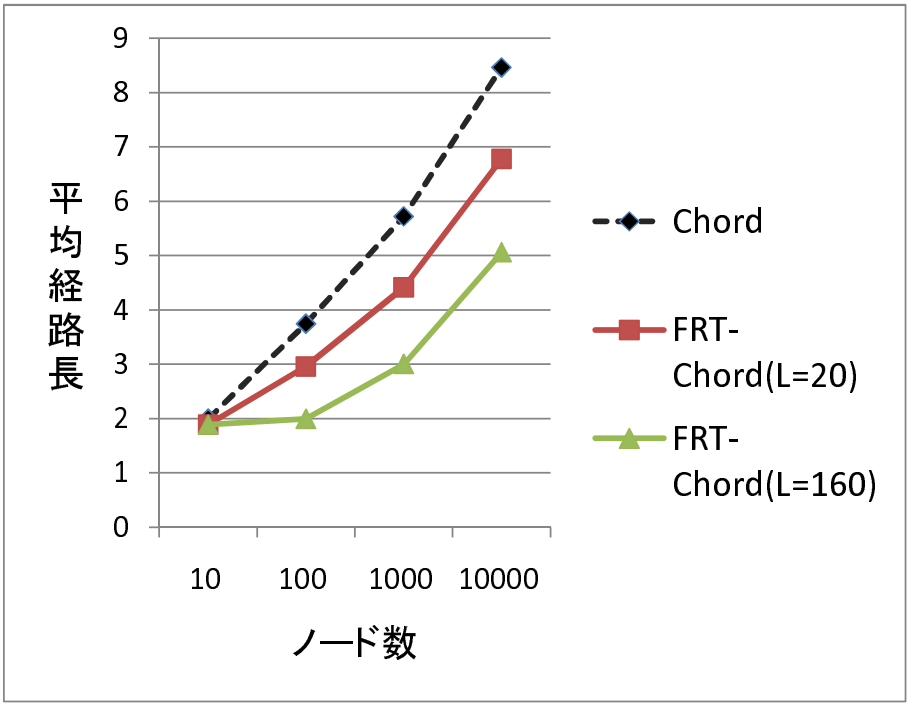


図 5.10: ノード数と平均経路長の関係

て Chord より優れていることから、FRT-Chord の経路長も Chord 同様に $O(\log N)$ であると考えられる。

**ゼロホップ DHT に関して**

図 5.11 に注目すると、ノード数 10 の Chord において、FRT-Chord と異なり、最大経路長が 3 となっている。これは、Chord の経路表が厳格であるためにノード数が少ない場合にも経路表エントリがあふれてしまっていることに起因する。一方で、柔軟な経路表の概念を取り入れた FRT-Chord では、経路表サイズよりノード数が少ない場合に全ノードが経路表エントリに追加され、経路長 1 で担当ノード直前のノードへホップしている (経路長が 2 の場合を指す)。

### 5.7.3　結論

この実験から、FRT-Chord のノード数に対する経路長の増加が Chord 同様に $\log N$ に従っており、$O(\log N)$ であると示された。また、ノード数が少ない状態で、ゼロホップ DHT が達成されることが確かめられた。

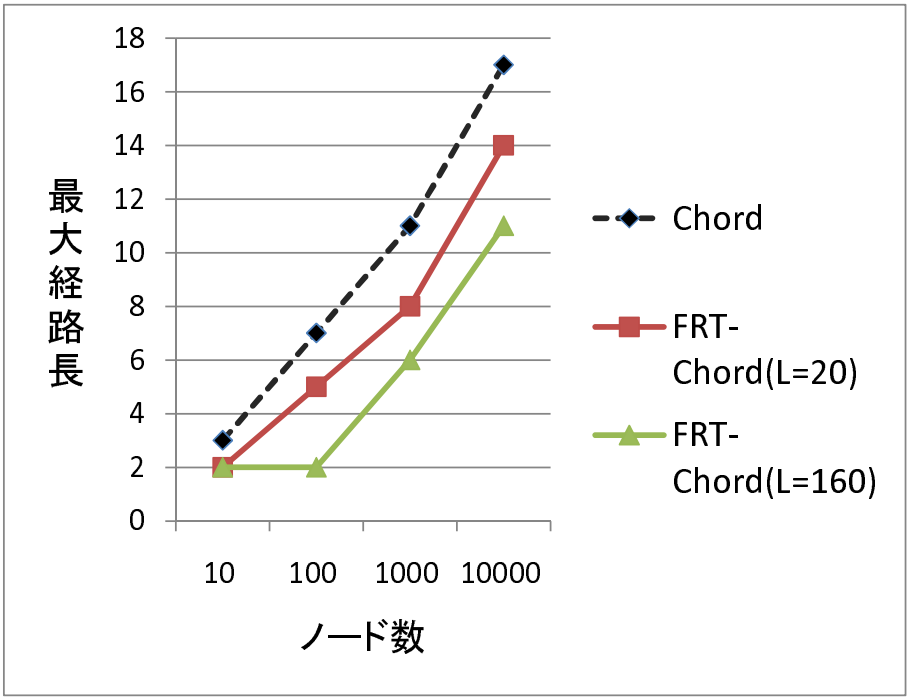


図 5.11: ノード数と最大経路長の関係

# 第6章　関連研究

関連研究を以下に挙げる

### LPRS[10]

LPRS では、lookup 時の通信を利用してレイテンシを計測し、レイテンシの小さいノードを優先的に経路表に取り入れることにより、ルーティングにネットワーク近接性を利用する。LPRS-Chord では、Finger table の i 番目のエントリを $[2^i, 2^{i+1})$ の範囲で最もレイテンシの小さいノードとしている。こうして $[2^i, 2^{i+1})$ から 1 つノードを選択することで、lookup 時の経路長を $O(\log N)$ に保ったまま経路表エントリの選択に幅を持たせ、よりレイテンシが小さいノードをフォワーディング先として選択することが出来る。こうして LPRS は lookup 時のレイテンシを小さくしている。

文献 [1] によれば、DHT のルーティングにおけるネットワーク近接性の考慮方法は、ネットワーク近接性を考慮するタイミングにより、ID 割り当て時に近接性を考慮する PIS(Proximity Identifier Selection)、経路表構成時に近接性を考慮する PNS(Proximity Neighbor Selection)、フォワーディング先ノードの選択時に近接性を考慮する PRS(Proximity Route Selection) の 3 つに分類することができる。これに従うと、LPRS は経路表構成時にネットワーク近接性を考慮する PNS に分類される。

LPRS はエントリの選択に幅を持たせている点が本研究との共通しているが、本研究の提案手法では経路表構成時にレイテンシ情報を考慮する PNS だけでなく、経路表サイズを大きくすることでフォワーディング先の選択に幅を持たせ PRS を行うことも可能である点が異なる。

また、LPRS はレイテンシの向上を目的としている点が柔軟性の向上を目的としている本研究とは異なる。本研究の提案手法は、より柔軟性の高い経路表を目標としている。

### NoN-GREEDY ルーティング [4]

NoN-GREEDY ルーティングでは、自身の経路表エントリだけでなく、自身の経路表エントリに含まれているノードの経路表エントリすべてか

ら最も目的 ID に近いノードをホップ先に選択する。このようにすることで、自身の経路表エントリ中で最も目的 ID に近いノードをホップ先に選択するよりも結果として経路長を小さく抑えることが可能となる。

NoN-GREEDY ルーティングは、フォワーディング先を従来より多くの候補から選択しようとしており、その点は本研究と重なる。しかし、経路表エントリに含まれているノードすべての経路表エントリを取得・更新する必要があるだけでなく、他のノードの経路表エントリを参照しなくてはならないことが問題となる。また、この方法を利用して経路表を拡大したとしても、ゼロホップ DHT を実現することは出来ない。

### Small-World Phenomenon

文献 [3] 中では、経路表エントリを近接ノードとある確率分布に従い決定した 1 ノードとすることで、経路長の効果的に短縮することが可能であることが示されている。分布を定義し、それに従うように経路表を構成するという点が本研究と共通している。

文献中で扱われているネットワークにはノードが $k$ 次元格子状に規則正しく分布しており、各ノード $s$, $t$ 間の距離は $s = (s_0, s_1, ..., s_{k-1})$, $t = (t_0, t_1, ..., t_{k-1})$ としたときに $d(s,t) = \sum_{i=0}^{k-1} |s_i - t_i|$ となる。また、確率分布は自ノード ID を $s$、エントリ候補ノード ID を $t$ としたとき、定数 $r$ と、確率の和を 1 とするための定数 $c$ を用いて

$$p(t) = c\frac{1}{d(s,t)^r}$$

とされている。このとき、$r = k$ が最も効率的であるとされている。ここで、$k = 1$ としたとき、この確率分布は提案手法の目的密度関数と一致する。このことから、構造化オーバレイの $O(\log N)$ という性質が Small-World と同じ理由から来ており、この Small-World 性を従来手法に比べてより自然な形で取り入れたものが提案手法であると考えることが出来る。

# 第7章　まとめと今後の課題

## 7.1　まとめ

DHT ルーティングアルゴリズムでは、経路表の構成方法が厳格に定められており、経路表の柔軟な拡張や管理が難しいという問題点があった。本研究では、そのような厳格な経路表に対し、経路表エントリの目標分布を利用した柔軟な経路表および柔軟な経路表を Chord に適用した FRT-Chord を提案した。

提案手法 FRT-Chord を Overlay Weaver 上に実装し、経路表管理やルーティング性能に関する実験を行い、Chord と同等の経路長を維持したまま経路表を柔軟に管理可能であることを示した。また、ノード数が少ない状況では全ノードを経路表に載せることにより、ゼロホップ DHT が可能であることを示した。

FRT-Chord を利用することで、ノードの搭載メモリや負荷、経路長等に応じて経路表サイズを変更することが可能である。

## 7.2　今後の課題

### 7.2.1　柔軟な経路表の他のアルゴリズムへの適用

柔軟な経路表は Chord 以外のアルゴリズムにも適用できると考えている。また、従来のアルゴリズムに適用するだけではなく、より柔軟な経路表に適したアルゴリズムが作成できないか検討する。

### 7.2.2　他の削除アルゴリズムを利用する

FRT-Chord で利用したエントリ ID 間隔に注目した分布維持方法を他の方法と比較し、よりよい分布維持方法を検討する。

### 7.2.3　諸性質の定量的な証明

今回、FRT-Chord の性能を実験を中心に示した。しかし、より多くのノードをあつかった場合にどのような挙動を示すのか、ノードの ID が特

定の並びになったときに不都合が起きないか等、すべての可能性を考慮して性質を示すためには、定量的な証明が必要不可欠である。そのため、定量的なアプローチによる証明を検討する。

また、その他の定量的なアプローチとして、経路表サイズ $L$ およびノード数 $N$ などによる経路表安定までに必要な lookup 回数の違いなどに関する相関を求めたい。

### 7.2.4　適切な経路表サイズの決定

適切な経路表サイズは保証したい遅延やノード数に依存すると考えられる。一方でむやみに経路表を拡大することは更新頻度やそのメモリ効率からして適切とは言えない。そのため、以下のような 2 つのアプローチを検討する。

- 必要なレイテンシ (経路長) および予想される (準備した) ノード数などのパラメータから、適切な経路表サイズを静的に決定する。
- 初期状態では小さい経路表を構築し、レイテンシを測定しながら目標として設定されたレイテンシを維持できるような経路表サイズを動的に決定し、最低限の経路表サイズに変更する。

### 7.2.5　将来の広域ネットワークへの応用

現在のインターネットに換わる新しいネットワークの形が現在も検討されている。有線通信と移動体通信を統合する FMC(Fixed Mobile Convergence) でのルーティングを考えた場合、全対全で通信可能であるという仮定を必要としないルーティングを行う必要がある。また、異なる通信網間の通信を最小限に抑える経路を利用するようにしなければならない。こういったルーティングを実現するためには、経路表エントリやフォワーディング先を様々な要素を考慮して選択する必要がある。このようなルーティングに柔軟な経路表が利用できないかを検討する。

# 謝辞

本研究を進めるにあたり、指導教員の首藤一幸准教授には多くの議論を通じて様々な知識を共有させていただいただけでなく、研究の進め方やOverlay Weaverの詳細な利用法などについて丁寧な指導を賜りました。ここに感謝の意を表します。また、合同でのミーティングを行っていただき、研究に対するコメントをくださった千葉滋教授に感謝いたします。最後に、研究活動を支えてくださった首藤研究室・千葉研究室の皆様に心から感謝します。

# 参考文献

[1] Gummadi, K., Gummadi, R., Gribble, S., Ratnasamy, S., Shenker, S. and Stoica, I.: The Impact of DHT Routing Geometry on Resilience and Proximity, *Proceedings of SIGCOMM 2003* (2003).

[2] Kaashoek, M. F. and Karger, D. R.: Koorde: A Simple Degree-Optimal Distributed Hash Table, *IPTPS*, pp. 98–107 (2003).

[3] Kleinberg, J.: The Small-World Phenomenon: An Algorithmic Perspective, Technical report, Ithaca, NY, USA (1999).

[4] Manku, G. S., Naor, M. and Wieder, U.: Know thy Neighbor's Neighbor: the Power of Lookahead in Randomized P2P Networks, *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, New York, NY, USA, ACM, pp. 54–63 (2004).

[5] Maymounkov, P. and Mazières, D.: Kademlia: A Peer-to-Peer Information System Based on the XOR Metric, *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, London, UK, Springer-Verlag, pp. 53–65 (2002).

[6] Ratnasamy, S., Francis, P., Handley, M., Karp, R. and Schenker, S.: A Scalable Content-Addressable Network, *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA, ACM, pp. 161–172 (2001).

[7] Rowstron, A. I. T. and Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems, *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, London, UK, Springer-Verlag, pp. 329–350 (2001).

[8] Shudo, K., Tanaka, Y. and Sekiguchi, S.: Overlay Weaver: An Overlay Construction Toolkit, *Comput. Commun.*, Vol. 31, No. 2, pp. 402–412 (2008).

[9] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F. and Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA, ACM, pp. 149–160 (2001).

[10] Zhang, H., Goel, A. and Govindan, R.: Incrementally Improving Lookup Latency in Distributed Hash Table Systems, *SIGMETRICS Perform. Eval. Rev.*, Vol. 31, No. 1, pp. 114–125 (2003).

[11] Zhao, B. Y., Kubiatowicz, J. D. and Joseph, A. D.: Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and, Technical report, Berkeley, CA, USA (2001).

[12] 首藤一幸: Overlay Weaver, `http://overlayweaver.sourceforge.net/`.