Z-ordering による二次元空間関係を考慮した P2P 情報検索ネットワークの実装

島根大学　総合理工学部　数理・情報システム学科

計算機科学講座　田中研究室

S033037　熊丸 恵太

平成 19 年 2 月 6 日

目次

第 1 章 序論

1.1 研究目的

現在我々の身の回りには、携帯電話やカーナビをはじめとした多くの通信端末が至るところに存在しており、屋外においても手軽に通信手段を得ることが出来るようになった。

そして、これら通信端末を用いることで例えば交通情報や災害情報、近隣のイベントに関する情報といった様々な種類の情報を各人が発信することができ、またそれらを共有、編集することが可能となる。

このような個人レベルでの情報ネットワークはこれまで目を向けられることが少なかったが、個人の身近に存在する情報をより効果的に運用する上で非常に有益であり、特に地震などの広域災害や渋滞などの道路交通問題といった場面においてその価値は大きい。

そこで、本研究ではこのような情報ネットワーク構築の際に重要となるであろう各通信端末の持つデータの位置に基づいた検索に注目し、そのための情報検索ネットワークを構築することを目標に Java による P2P ネットワークの実装を行った。

本研究を通して、今後上記の分野における研究を行っていく際に足掛りとなる知識や経験、能力を身に付けることがねらいである。

1.2 研究概要

本研究では、前述した P2P ネットワークの実装にあたり、ネットワークの構造に Skip Graph[1]を用い、また空間のインデックス付けに Z-ordering[2]を用いた。

これら二つの技術を用いて実装を行ったのは、二次元空間における通信端末の位置関係を P2P ネットワークへ反映させることができ、効率的な P2P ネットワークの構築が期待出来るためである。

また、通信部分には Java の RMI(Remote Method Invocation)機能を用いることで実装の簡単化を図った。

動作確認に関しては、一台の Linux 計算機(DebianLinux)を用いて、一つの仮想端末を一つの通信端末と見なすことで仮想的な P2P ネットワークを構築し、その P2P ネットワークに対して範囲検索を実行した結果を確認することで行った。

第 2 章 P2P ネットワーク

2.1 P2P とは

本研究でネットワークの実装を行う際に用いた P2P(Peer-to-Peer)ネットワークについて文献[3]、[4]に沿って説明する。

現在、一般的なネットワークモデルには P2P 型の他に C/S 型(Client/Server 型)がある(図 2.1)。これは Server と呼ばれる特定のコンピュータがコンテンツやその処理作業を集中的に管理し、Client と呼ばれる他のコンピュータは必要に応じて Server からサービスを受けとるというネットワークモデルである。

一方、今回用いた P2P 型では C/S 型のように各コンピュータを Client や Server で区別することはなく、一台のコンピュータが状況に応じて Client と Server のどちらにもなりうるというネットワークモデルである。

携帯電話やカーナビといった移動可能な通信端末を考えた場合、各端末に Client や Server といった役割を決めておくことは不可能である。また、通信端末とは別にあらかじめ Server を用意していたのでは、即時性や柔軟性に欠けるため現実的とはいえない。

そのため、あらかじめ Server を用意する必要が無く、また各通信端末が Client と Server のいずれの処理も行える P2P ネットワークが、このような通信端末によるネットワークの際には有効である。

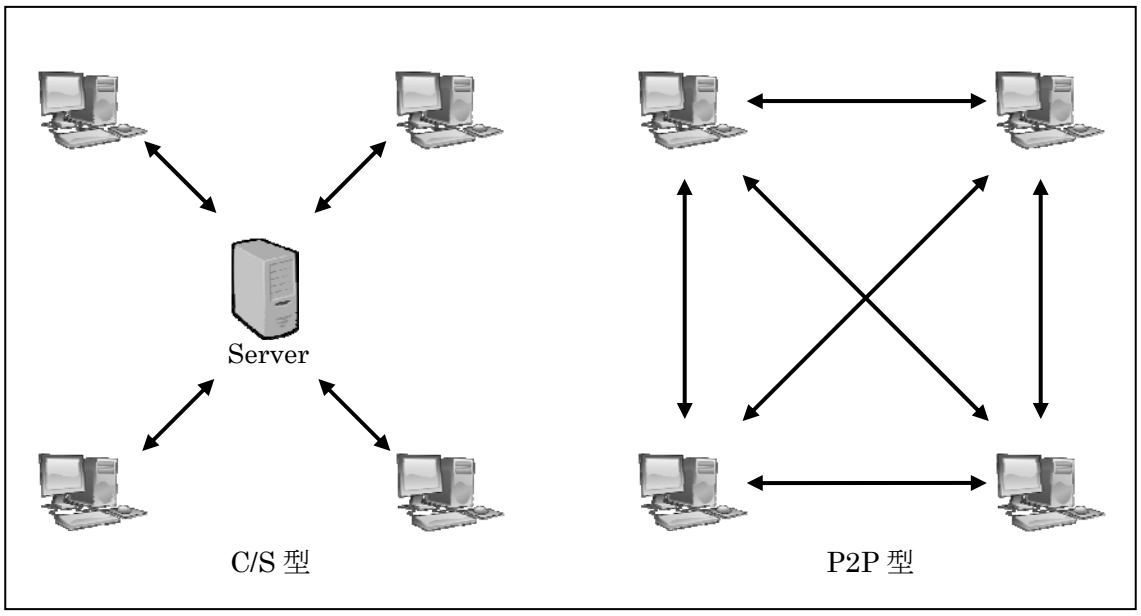


図 2.1 C/S 型と P2P 型

2.2 P2P ネットワークの種類と違い

P2P ネットワークにはデータ検索の方法で分類した場合に、一部の処理を行うために C/S 型の構造を持つ Hybrid 型と、全ての処理を P2P 型を用いて行う Pure 型がある(図 2.2)。

Hybrid 型は server が流通するデータのインデックスとして働くため、求めるデータを持つ通信端末を確実に発見することが出来る反面 C/S 型と同様の問題が発生する可能性がある。

一方、Pure 型では C/S 型の様な問題が発生することは無いが、流通するデータを管理する server が存在しないため、データ検索に関連した問題が起こる。

そして、この Pure 型におけるデータの検索・管理の仕方について、Structured Overlay と Unstructured Overlay がある。

Structured Overlay とはネットワークトポロジに制限を加えることでデータ検索の確実性を保障した P2P ネットワークである。これには、DHT(Distributed Hash Table)や今回用いた Skip Graph が挙げられる。

対して Unstructured Overlay とはネットワークトポロジに制限を加えないことで、データ検索の確実性を犠牲にしながらも、柔軟なネットワークを構築出来るようにした P2P ネットワークである。このようなネットワークを構築するソフトウェアにはファイル共有ソフトの Winny などがある。

前述したように、携帯端末を対象とした場合に C/S 型のネットワークは望ましくない。また、交通渋滞や広域災害を対象とした情報検索を考えた場合、流通する情報が発見出来たり出来なかったりすることがあってはならず、情報検索の確実性は最優先すべき事柄である。そのため、今回の情報検索ネットワークの実装ではこれらを考慮し、Structured Overlay のひとつである Skip Graph を用いて P2P ネットワークの実装を行った。

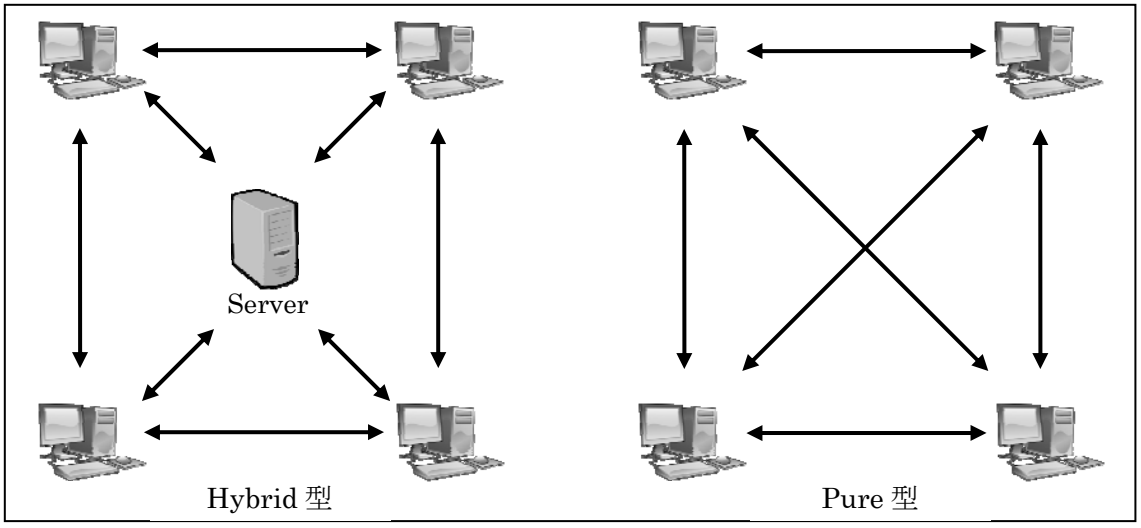


図 2.2 Hybrid 型と Pure 型

第3章 Skip Graph

3.1 Skip Graphを用いた理由

3.1.1 本研究におけるネットワーク構築の必須条件

本研究ではP2Pネットワークの構築にあたり、Skip Graph[1]を用いた。

前述した通り本研究の対象を考えた場合、構築するネットワークはP2P型が望ましい。また、データ検索については探しているデータがネットワーク中に存在する場合には確実に発見出来ることが必須となる。

3.1.2 DHT(Distributed Hash Table)概要

現在、上記の条件を満たす方法として DHT(Distributed Hash Table)が注目されている[4]、[5]、[6]、[7]、[8]。DHT とはハッシュ関数(あるデータが与えられた場合にそのデータを代表する数値を得る関数)を用いた、データ配置、データ検索、経路選択のためのアルゴリズムである。

DHT では各計算機に固有の値(IP アドレスなど)をハッシュ関数にかけることで得た数値をその計算機の ID とし、同様に各計算機の持つデータ(ファイルなど)に固有な値(ファイル名など)をハッシュ関数にかけることで得た数値をそのデータの ID とする。そして、各データは自分の ID と計算機の ID との差が最小になる計算機に管理される。

また、各計算機はネットワーク中で自分の隣に位置する計算機の情報(IP アドレスや ID など)しか管理せず、インターネットにおけるルータと同様の方法でメッセージの伝達を行う。

DHT の基本的な考え方は上記の通りで、経路選択の方法や ID の差の定義といった違いにより、Pastry[6]、CAN[7]、Chord[8]をはじめ様々な種類がある。

この DHT を用いることで、ネットワーク中のデータを中央サーバを用いること無く均等に分散管理することができ、データ検索の確実性や負荷分散を保障している。また、ハッシュ関数の特性によりデータ検索が対数時間で実行出来るのも大きな利点である。

3.1.3 DHT の欠点

DHT ではハッシュ関数を用いており、そのため次に挙げる欠点がある。

データ配置をハッシュ値によって行っているため、データが強制的にネットワーク中の複数の計算機上に分散配置されてしまう。

このようなデータ配置は負荷分散という面では有効であるが、例えば企業の部署内のみで閲覧したいデータや個人情報などの他人に見られたくないデータといった、流通に管理が必要なデータに関しては無闇に複数の計算機上に配置するべきではない。

こうした理由から強制的な分散配置を行うことは欠点となる場合がある。

また、ハッシュ値の取りうる範囲を事前に決めておく必要がある。DHT では計算機やデータの ID にハッシュ値を用いているため、ネットワーク中に同一の ID が存在することのないように、衝突がほぼ確実に発生しない程度の大きさをハッシュ空間に持たせる必要がある。

ハッシュ空間の大きさについては、小さ過ぎる場合には衝突が発生し、大き過ぎる場合にはメモリ空間の無駄につながることになる。そのため、大きさを決定する際には構築されるネットワークの規模(計算機やデータの数)をあらかじめ予測し、その規模に応じて決定する必要がある。

しかし、ネットワークの規模は簡単に予測できるものではないので、大抵その規模は大きめに見積もられることになってしまう。

最後に、ハッシュ関数が Key(ファイル名や IP アドレスなど、データや計算機に固有の値)の順序を破壊するため、ある値の範囲に含まれる Key の検索や名前が A で始まるファイルの検索といった、柔軟な検索が行えないという問題がある。

3.1.4 DHT の欠点に対して

DHT の欠点に対応するための方法として Skip Graph という分散データ構造を用いることが考えられる。Skip Graph とは Skip List[9]と呼ばれる多重リスト構造を分散環境に適応させたデータ構造である。

この Skip Graph は DHT と同様の利点を持つだけでなく、前述した DHT の欠点を補うことが出来る。また、実装面でも DHT より比較的容易に実装出来るという利点がある。

こうした理由から、本研究では Skip Graph を用いた。

3.2 Skip Graph 概要

Skip Graph によって構築されるネットワークは図 3.1 のように概念的に表される。

各通信端末はノードと呼ばれ(以降はネットワーク中の通信機器はノード、その他の場合は必要に応じ計算機や通信端末などと記述)、それぞれが高さ(レベル)を持ち、レベル毎に双方向のリンクによって繋がれている。リンクに関してノードは、各レベルで自分の隣に位置するノード(隣人ノード)についての情報しか管理しない。

そのため、ノード検索やノード参加などの操作は、各操作に関連するメッセージをネットワーク中のノード間でバケツリレーのように転送することによって実現される。

ノードは自分が保持する一意な値(Key)によって判別され、ネットワーク中にはこの Key の値(Key 値)で昇順に整列された状態で配置される。ノード検索やノード参加などの操作はこの Key 値をもとにして行われる。

また、ノードはメンバーシップベクタと呼ばれる任意の値を持っており、各レベルでリンクするノードを決定する際に用いる。この値は DHT のハッシュ値とは異なり、新しい桁が必要になる度に一桁ずつ値を決定すれば良いため、あらかじめネットワークサイズを予測し、それをもとにメンバーシップベクタの桁数を決定しておく必要が無い。

Skip Graph は各レベルでの複数のリンクにより構成されているが、レベル 0 のリンクには全てのノードが参加しており、これによりメッセージがネットワーク中のノード全てに行き渡ることを保障している。また、レベル 1 以上のリンクではネットワークのリンクに冗長性を持たせることで耐故障性を高めると共にメッセージ伝達の高速化を行っている。

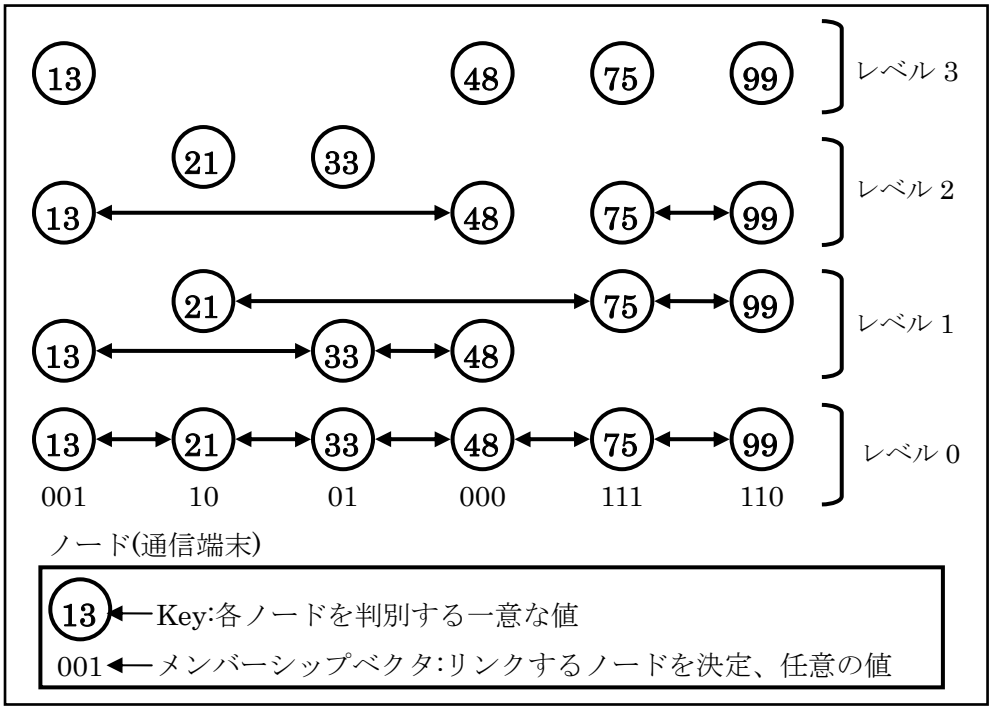


図 3.1 Skip Graph の概念図

3.3 ノード検索

3.3.1 ノード検索の手順

Skip Graph において重要な操作の一つである Key 値によるノード検索について、その手順を示す。

ノード検索は始点となるノードのトップレベルから開始し、レベルを降下しながら検索メッセージを適切なノードに転送することで実現する。

検索手順(検索メッセージを受け取ったノード n の処理)

1. 検索開始(検索開始ノードならばトップレベル、中継ノードならばメッセージを受け取ったレベルから開始する)。
2. n の Key 値と検索する Key 値が同じならば、n が検索対象のノード。n に関する情報を検索開始ノードに返して検索終了。
3. n が検索対象でないならば、検索する Key 値を超えない最大(もしくは最小)の Key 値を持つ隣人ノードを探す。
   検索する Key 値より n の Key 値が小さいならば、検索する Key 値より小さい Key 値を持つ右隣のノードとなる。
   検索する Key 値より n の Key 値が大きいならば、検索する Key 値より大きい Key 値を持つ左隣のノードとなる。
4. 手順 3 で隣人ノードが見つからない場合はレベルを一つ下げた後、手順 3 を行う。
5. 手順 3,4 によって見つかった隣人ノードに検索メッセージを転送。
6. 検索メッセージを受け取ったードも同様に手順 1,2,3,4,5 によって隣人ノードを探し、検索メッセージを転送。
7. 上記の処理を隣人ノードが見つかる限り繰り返し続ける、レベルが 0 より低くなった時点で検索しているノードが見つからなければ検索する Key 値を超えない最大(もしくは最小)の Key 値を持つノードの情報を検索開始ノードに返して終了。

ノード検索の例を図 3.2 に示す。図 3.2 の例では Key:13 のノードが Key:95 を検索する場合を示している。点線が検索メッセージの伝播を表す。

このとき、検索は Key:13 のノードから開始されるが、Key:13 のノードはレベル 3 において隣人ノードを持っていないので、レベルを一つ下げる(1)。レベル 2 において Key:13 のノードは右隣にある Key:48 のノードとリンクしており、Key:48 は検索している Key:95 より小さいので、検索メッセージを Key:48 のノードに転送する(2)。

検索メッセージを受け取った Key:48 のノードはメッセージを受け取ったレベルであるレベル 2 においてその隣人ノードを確認するが、Key:48 のノードはレベル 2 において隣人ノードを持っていないのでレベルを一つ下げる(3)。しかし、Key:48 のノードはレ

ベル 1 においても隣人ノードを持っていないので、さらにレベルを一つ下げる(4)。ここで、レベル 0 において Key:48 のノードの右隣には Key:75 のノードがあり、Key:75 は検索している Key:95 より小さいので、検索メッセージを Key:75 のノードに転送する(5)。

次に、検索メッセージを受け取った Key:75 のノードの右隣には Key:99 のノードがあるが、Key:99 は検索している Key:95 より大きいので、検索メッセージは転送されずに検索はこの時点で終了、検索を開始した Key:13 のノードには Key:95 のノード(検索している Key 値を超えない最大の Key 値を持つノード)の情報が返される(6)。

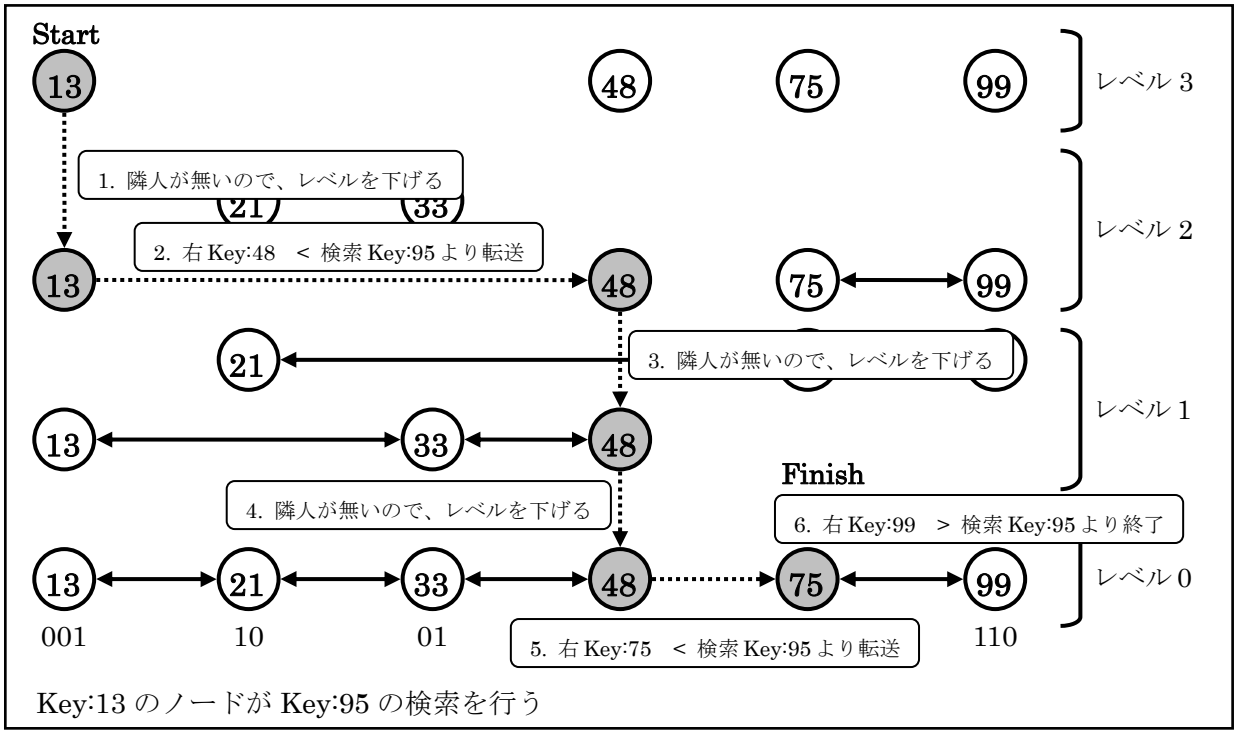


図 3.2 ノード検索の例

3.3.2 ノード検索の計算量とその証明

ノード数 n の Skip Graph におけるノード検索操作は、$O(\log n)$ 時間、$O(\log n)$ メッセージを要する。

以下にその証明を示す。
まず、証明に関連して補題を示す。

補題

メンバーシップベクタに用いられるアルファベットを Σ とし、$\{S_w\}$ をアルファベット Σ による Skip Graph とする。また、Σ^∞ を Σ のクリーネ閉包、$z\mathrm{I}i$ を長さ i の z の接頭辞、$|\Sigma|$ を Σ の個数とする。このとき以下が成り立つ。

任意の $z \in \Sigma^\infty$ について $S_i = S_{z\mathrm{I}i}$ とするとき、連続する $S_0, S_1, S_2, \cdots,$ は $p = |\Sigma|^{-1}$ の Skip List。

証明開始

Skip Graph S に含まれるノードのメンバーシップベクタに用いられるアルファベットを Σ とし、z を検索開始ノード、$\mathrm{m}(z)$ を z のメンバーシップベクタとする。

ここで補題より、連続する $S_{m(z)} = S_0, S_1, S_2, \cdots,$ は Skip List である。

Skip Graph 中で z から開始した検索は、S において $S_{m(z)}$ と同様のパスを辿る。そのため、文献[9]に述べられている Skip List における検索操作の分析を S における検索操作の分析にそのまま適用することが出来る。

この分析では、n 個のノードでは平均 $O\left(\log n \,/\, \log(1/p)\right)$ レベルであるとされている（$p = |\Sigma|^{-1}$）。殆どの場合において平均して $\dfrac{1}{1-p}$ 個のノードが各レベルで検索されるので、合計して $O\left(\log n \,/\, (1-p)\log(1/p)\right)$ 時間、$O\left(\log n \,/\, (1-p)\log(1/p)\right)$ メッセージとなる。

よって、p は定数なのでノード検索操作は $O(\log n)$ 時間、$O(\log n)$ メッセージを要する。

■

3.4 ノード参加

3.4.1 ノード参加の手順

Skip Graph において重要な操作の一つであるノード参加について、その手順を示す。

ノード参加では、ノードはレベル 0 における適切な位置に自分を配置することから開始する。次に、自分のメンバーシップベクタの各桁の値を決定し、同レベルでリンクしているノードの持つメンバーシップベクタの各桁の値とそれらの比較を行い、より高いレベルでのリンクを作成していく。

参加手順(新規参加ノード n の処理)

1. n は初期ノードに自分の Key 値で検索を依頼。
2. 検索結果から自分の隣人ノードのアドレスを得る。
3. 手順 2 で得たノードに依頼することで、レベル 0 における適切な位置に自分を配置する。
4. n は自分のメンバーシップベクタの一桁目の値を決定。
5. n はレベル 0 における隣人ノードにメンバーシップベクタの一桁目の値を尋ねる。
6. 自分のメンバーシップベクター桁目の値と値が同じならばレベル 1 でそのノードとリンクし手順 8 を行う、異なるならばさらに隣のノードにメッセージを転送し手順 5 以降を行う。
7. 手順 6 において、転送するノードが無ければ処理を終了する。
8. 二桁目以降も手順 4,5,6,7,8 と同様の処理を行う。

ノード参加の例を図 3.4_1、図 3.4_2、図 3.4_3、図 3.4_4 に示す。図 3.4 の例では図 3.1 で示される状態のネットワークに Key:50 のノードが参加する場合を示している。

このとき、ネットワークに参加する Key:50 のノードは、初期ノードである Key:13 のノードに Key:50 で検索を依頼する(1)。この処理によって、Key:50 のノードはレベル 0 における自分の左隣にあるノード(Key:50 を超えない最大の値を Key 値に持つノード)のアドレスを得ることが出来る。また、この左隣のノードは Key:50 のノードの右隣に位置することになるノードのアドレスを知っている。そのため、Key:50 のノードは左隣のノードに、自分をレベル 0 においてこれらノードの間に配置するよう依頼する(2)。

次に、Key:50 のノードは自分のメンバーシップベクタの一桁目の値を決定する。この決定方法は任意で良く、今回の実装では乱数で 0,1 のいずれかを発生させるようにしている(3)。そして、メンバーシップベクタの一桁目の値を決定した Key:50 のノードはレベル 0 におけるノードに対してメンバーシップの一桁目の値を尋ねる。このとき、レベル 0 において Key:50 のノードの左隣にある Key:48 のノードはメンバーシップベクタの一桁目の値が Key:50 のノードと一致している、そこで Key:48 のノードと Key:50 のノ

ードはレベル 1 においてリンクする(4)。

これにより、Key:50 のノードは一つ上のレベルでリンクすることが出来たので、続けてメンバーシップベクタの二桁目の値を決定する(5)。二桁目の値を決定した Key:50 のノードは先程と同様にレベル 1 におけるノードに対してメンバーシップベクタの二桁目の値を尋ね、自分のメンバーシップベクタの二桁目の値と比較する。このとき、Key:33 のノードが持つメンバーシップベクタの値と同じ値であるのでレベル 2 において Key:50 のノードと Key:33 のノードはリンクする(6)。

さらに、Key:50 のノードはメンバーシップベクタの三桁目の値を決定し(7)、レベル 2 におけるノードに対して、そのメンバーシップベクタの三桁目の値を尋ねる(8)。このとき、Key:50 のノードからメンバーシップベクタの値を尋ねられた Key:33 のノードはメンバーシップベクタの三桁目が未定であるので、この Key:50 のノードに尋ねられた時点で三桁目の値を決定する(9)。レベル 2 に存在する Key:33 のノードと Key:50 のノードはメンバーシップベクタの三桁目の値が互いに異なっているので、レベル 3 においてリンクは行わず、ノード参加の処理を終了することになる(10)。

こうした一連の処理によって、ノードのレベルが上がり、ネットワークが成長していく(11)。

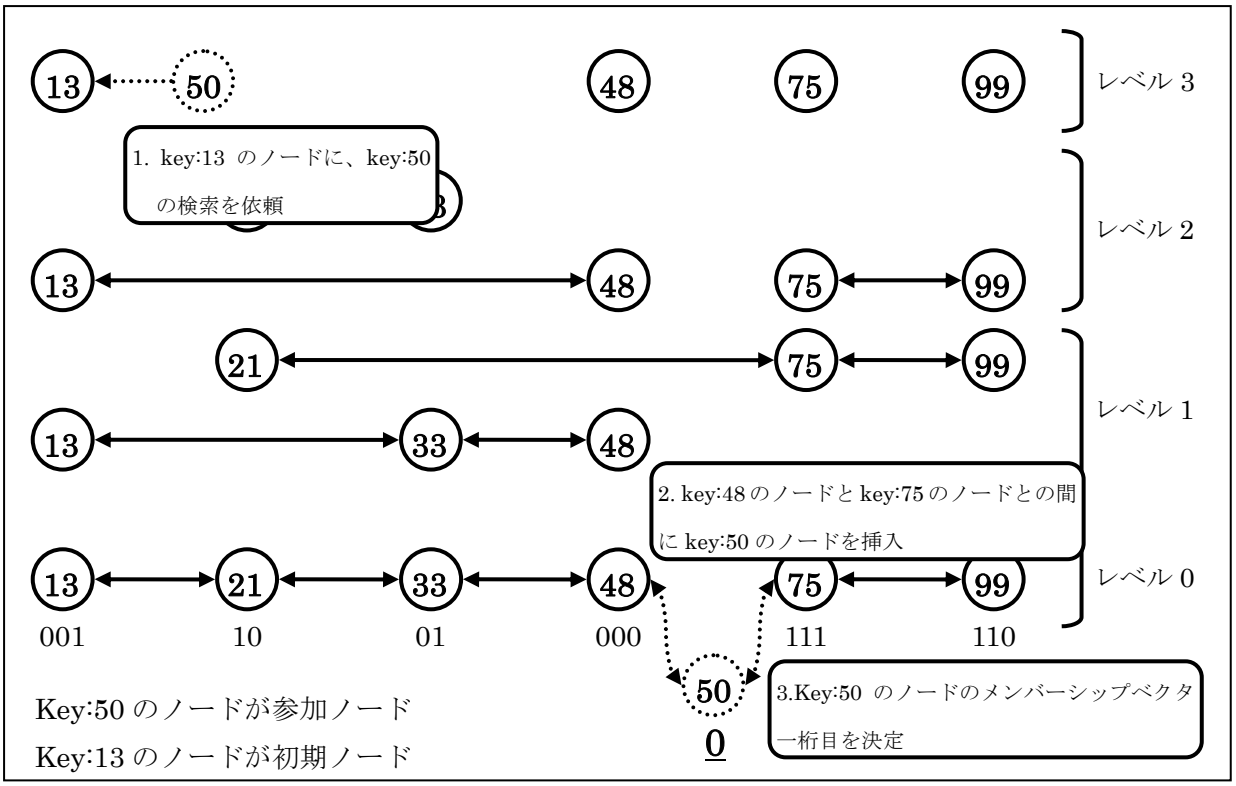


図 3.4_1 ノード参加の例(1/4)

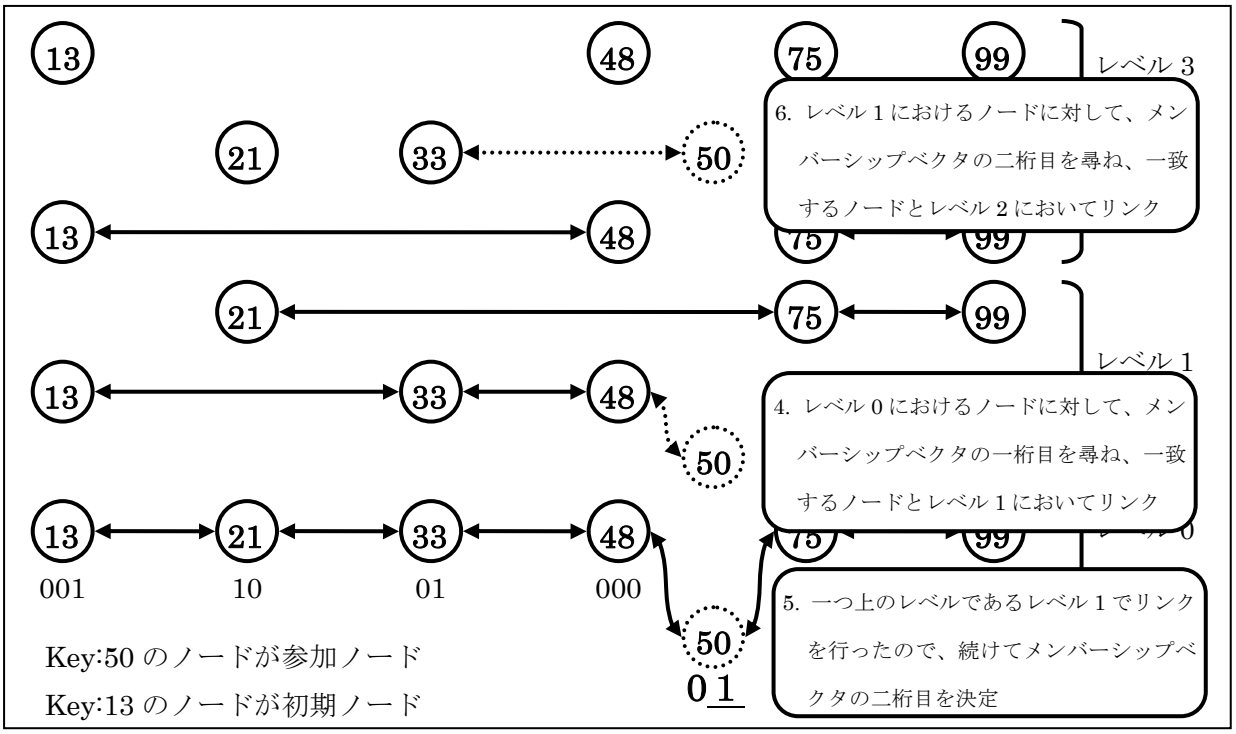


図 3.4_2 ノード参加の例(2/4)

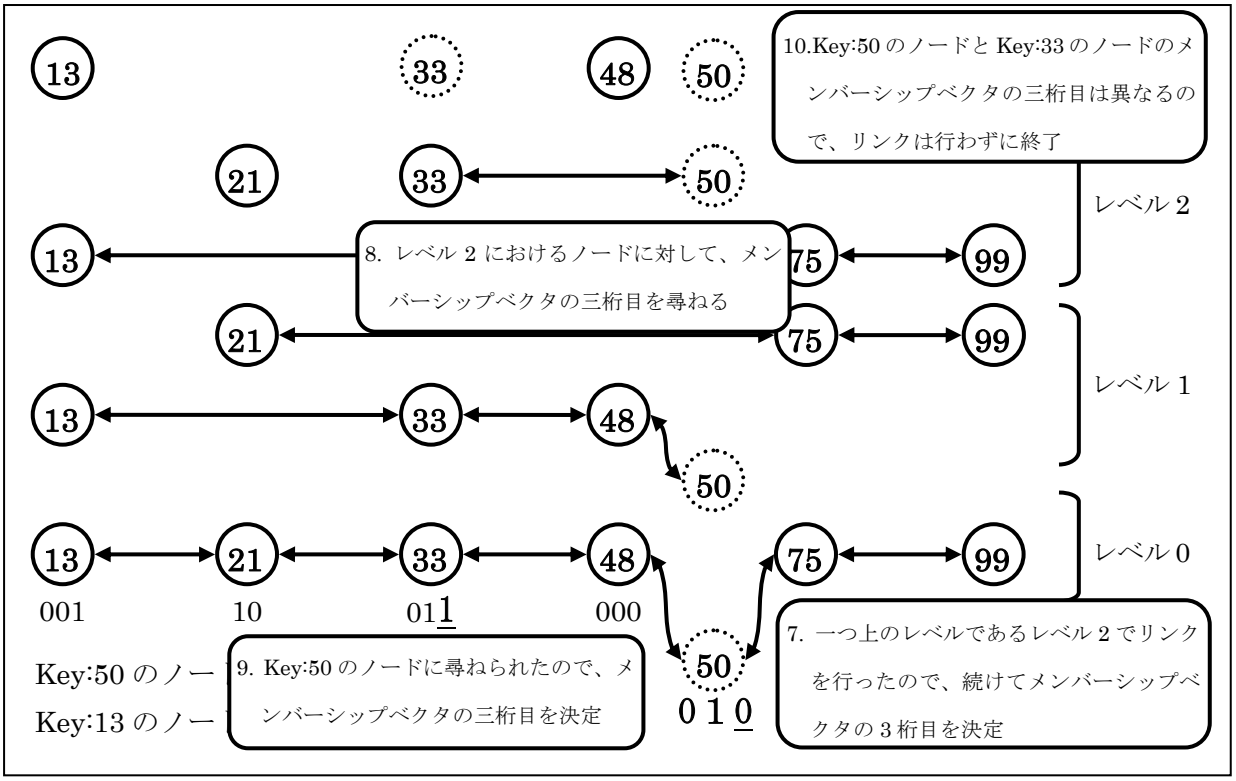


図 3.4_3 ノード参加の例(3/4)

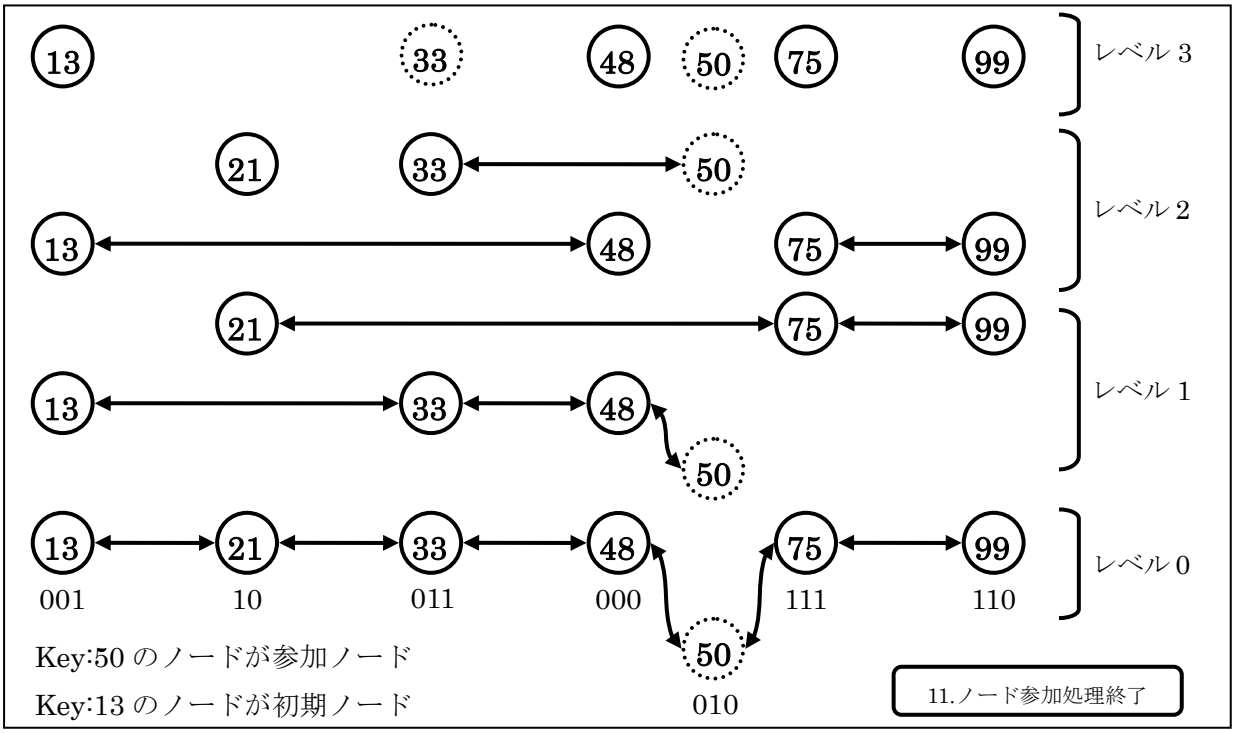


図 3.4_4 ノード参加の例(4/4)

3.4.2 ノード参加の計算量とその証明

ノード数 n の Skip Graph におけるノード参加操作は、$O(\log n)$時間、$O(\log n)$メッセージを要する。

以下にその証明を示す。

証明開始

メンバーシップベクタに用いられるアルファベットをΣ、$|\Sigma|$をアルファベットΣの個数、 p を$|\Sigma|$とする。

n個のノードにおける Skip Graph は平均$O(\log n)$レベル。レベル 0 でのリンクのため、新規ノード uは一度検索操作を行う。3.3.2 より、この検索操作は$O(\log n)$時間、$O(\log n)$メッセージである。

各レベルl($l \geq 0$)において u はレベル$l+1$における自分の一つ前に位置するノード s'を発見するまでに$1/p$個のノードと通信を行い、この処理には$O(1/p) = O(1)$時間と$O(1/p) = O(1)$メッセージが必要になる。この点からノードの挿入処理には$O(1)$時間要することになる。

よって、全てのレベルにおける挿入処理の合計に、最初に行われる検索操作を含めると、ノード参加操作は$O(\log n)$時間、$O(\log n)$メッセージを要する。

■

第 4 章 二次元空間関係を考慮したネットワーク構築

4.1 二次元空間関係を考慮する利点

ネットワークを構築する際に、リンクするノードの決定を適当に行っていたのでは都合が悪い。

ネットワークにおけるノード間の距離というのは現実世界における通信端末間の距離とは関連性が無いため、現実世界で大きく離れた位置にある通信端末がネットワーク中で隣同士としてリンクされる場合がある。例えば、現実世界において自分の近くに他の通信端末があるにもかかわらず、それとは異なる地球の裏側にある通信端末と、ネットワーク中で隣同士としてリンクしてしまう、ということが起こる。

一般的に通信を行う機器間の距離が物理的に離れるほど、通信速度は低下する傾向にあるため、このようにリンクされてしまうのは望ましくない。

また、Skip Graph によるネットワークの場合、隣接したノード間でのリンクは、リンクしているノードの間に新しいノードが参加するか、リンクしているノードの一方が脱退する場合でなければ変化することが無いため、非効率なリンクを行った際の影響が大きい。

そこで、この問題に対応するために二次元空間における通信端末間の位置関係を反映してネットワークを構築するのが効果的である。

二次元空間関係を考慮したネットワークでは、上述の通信速度における利点のほか、次に挙げる利点が考えられる。

一つは通信範囲の局所化が挙げられる。通常、通信端末間の距離が物理的に近いほどそれらの間に含まれるルータなどの通信機器の数は少なく、遠いほどその数は増える。そのため、位置関係を考慮したリンクを行うことで、通信時に間に含まれる通信機器の数を抑えることが出来る。

Skip Graph ではネットワークを構造化しており、ネットワーク中のノードが故障などの理由で不意に失われると、構造化されたネットワークが崩れてしまい、修復が必要になる場合がある。

災害や故障などによる影響は、通信している二点間に含まれる通信機器が多いほど受けやすくなるため、この通信範囲の局所化という特性は有効である。

その他に、無駄な通信の削減が挙げられる。これは上記の局所化とも関連するが、通信に含まれる通信機器が少ないほど必要なメッセージ転送が減るため、通信量の削減に繋がることになる。

また、二次元空間を対象にした範囲検索を効率化することが出来る。二次元空間で範囲検索を行う場合、検索の対象は二次元空間中において近い通信端末群なので、ネットワーク中の一定箇所に現実世界で近い位置にあるノードが集中している方が検索に必要なメッセージ転送の数を減らすことが出来るため効率的である。

4.2 方法

以下では二次元空間を扱うためのネットワーク構造について、多次元による場合と一次元による場合の二種について述べる。

4.2.1 多次元のネットワーク構造

ネットワークで二次元空間の位置関係を扱うためには、多次元のネットワーク構造を利用することが考えられ、その方法の一つとして、DHT の一種である CAN[7]が挙げられる。

CAN は d 次元デカルト座標空間を DHT 抽象化の実装に用い、座標空間はゾーンと呼ばれる hyper-rectangles に分割される。ネットワーク中の各ノードはそれらゾーンのいくつかに対して責任を持ち、ノードはそのゾーンの境界によって識別されることになる。

ネットワーク中に流通するデータは Key によって識別されるが、その Key は座標空間上の点に対応付けられ、その点の座標を含むゾーンの責任を持つノードに保持されることになる。

ある二つのノードの持つゾーンが$(d - 1)$次元の hyper-plane を共有するとき、これら二つのノードは隣人であると呼ぶ。各ノードは座標空間における全ての隣人の経路表を持っている。

図 4.2.1 に二次元で CAN を用いることによって構築されるネットワークの概念図を示す。この例では node 0 から node 4 までの五つのノードが存在している。各ノードは一次元の hyper-plane を共有しているノード同士で隣人となる。例えば、node 0 は node 3 と node 4 が隣人となり、node 0 はネットワーク中でこれらのノードとリンクする。

また node 0 はデータ固有の情報(ファイル名など)をハッシュ関数にかけることで得たハッシュ値(Key)が、自分の担当するゾーン(今回の場合では(0.0-0.5, 0.0-0.5)の範囲)に含まれている場合は、そのデータを管理することになる。

このように CAN では構築されるネットワークの状態を仮想的な二次元空間で表すことが出来る。そのため、ネットワークにノードが参加する際のノード配置に、現実世界における通信端末の位置を仮想的な二次元空間上の座標にマッピングする、という方法を用いることで現実世界における通信端末の位置関係を扱うことが出来る。

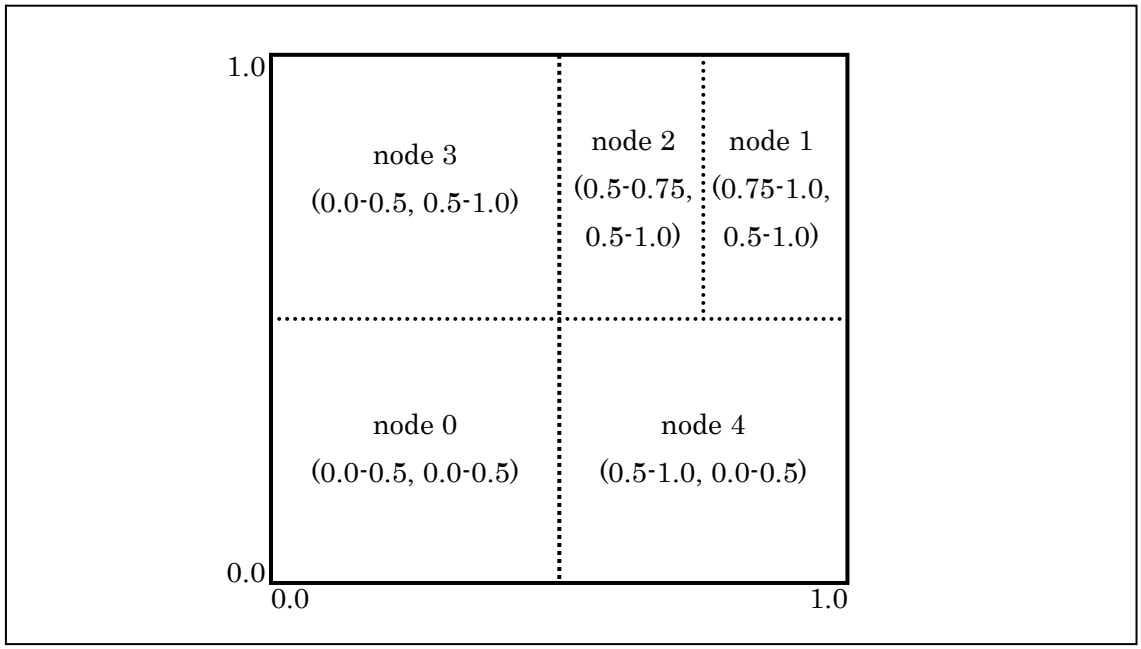


図 4.2.1 CAN の概念図

4.2.2 一次元のネットワーク構造

ネットワークで二次元空間を扱うためには、前述した多次元のネットワーク構造の他に、Skip Graph のような一次元のネットワーク構造を利用する方法が考えられる。

この場合、二次元空間を扱うためには、まず二次元空間の位置関係を一次元にマッピングする必要がある。そして、このマッピングによって得た結果を用いることで、二次元空間関係を反映した一次元のネットワークを構築することが出来る。このようにネットワークを構築した場合、以下に挙げる利点がある。

一次元のネットワーク構造は多次元のネットワーク構造と比べてその構造がシンプルになる。ネットワーク構造がシンプルな場合、実装が容易になるだけでなく、ネットワークの構築に必要な処理量も少なくなるため、通信量や処理速度といった面で有利である。また、その他にネットワーク中に流通するデータの管理や検索の方法が容易になるという利点があり、ネットワークの維持や運用の面でも効果的であると考える。

第 5 章 二次元空間の一次元への対応

二次元空間を一次元へマッピングするために、二次元空間を正方形の小領域に分割し、各小領域にコード付けを行うという方法をとる。

その際のコード付けの方法について、ジグザグな順序によるものと Z-ordering によるものの二種について述べる。

5.1 ジグザグな順序によるコード付け

分割された小領域にコード付けを行う際、図 5.1 に示すようにジグザグな順序でコード付けを行う方法がある。

このような方法でコード付けを行うと、二次元空間中の二点間において座標値が近い場合でもコードが大きく離れる可能性がある。

また、二次元空間を対象にした範囲検索の際に必要になる検索範囲の分割を行う場合に、分割数が多くなってしまう可能性がある。

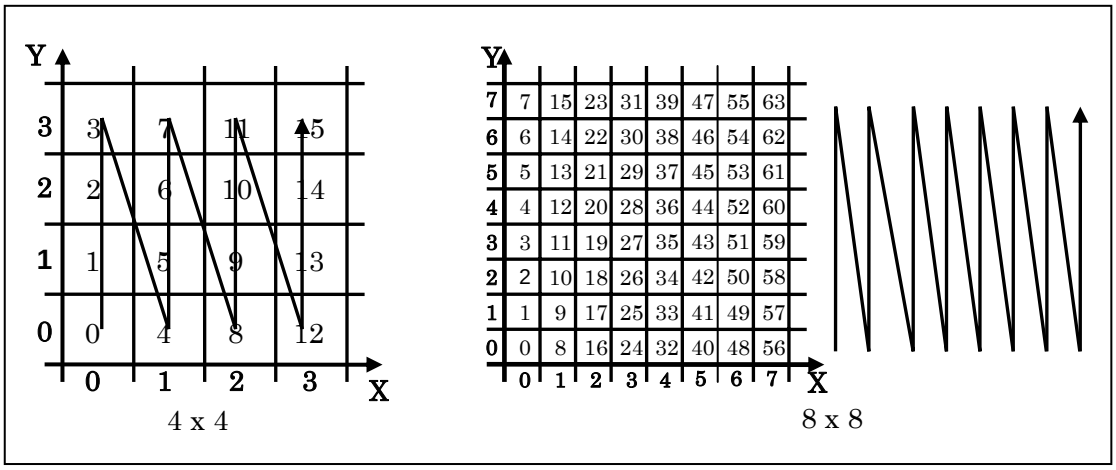


図 5.1 ジグザグな順序によるコード付け

5.2 Z-ordering によるコード付け

分割された小領域にコード付けを行う際に、図 5.2 に示す Z 曲線と呼ばれる空間充填曲線の一種を用いてコード付けを行う方法がある。この Z 曲線を用いた方法を Z-ordering と呼ぶ。

空間充填曲線とは空間中に存在する全ての点を交差することなく必ず一度通過する曲線である。この空間充填曲線にはヒルベルト曲線やルベーグ曲線などの種類が存在するが、実装の容易性から本研究では Z 曲線(Z-curve)を用いた。

Z-ordering を用いることで、二次元空間中の二点間において座標が近い場合でもコードが大きく離れてしまう可能性を下げることが出来る。

また、二次元空間を対象にした範囲検索の際に必要な検索範囲の分割において、その分割数を抑えることが出来る。

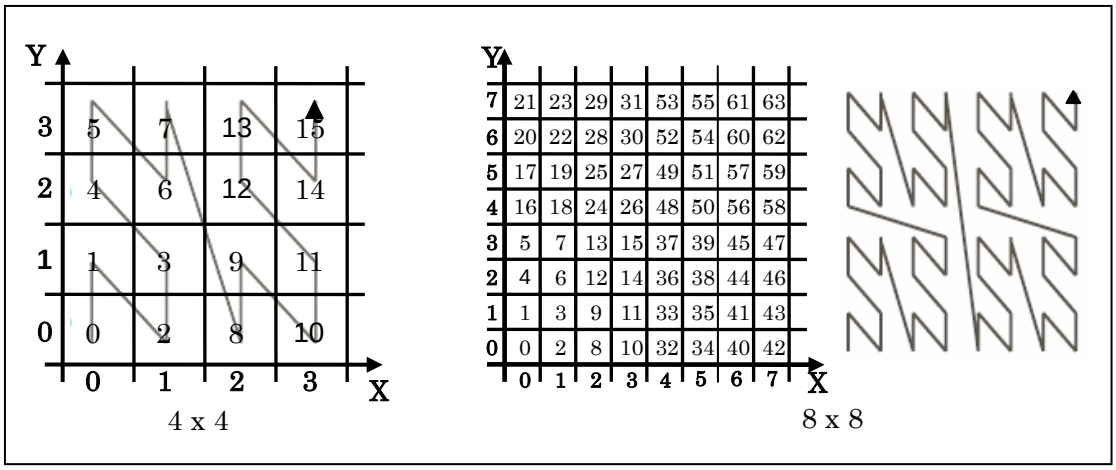


図 5.2 Z-ordering によるコード付け

第 6 章 Z-ordering と Skip Graph による P2P 情報検索ネットワーク

通信端末の二次元空間中における位置座標から Z-ordering の値を求め、この値を Skip Graph におけるノードの Key とすることで、二次元空間における位置関係の Skip Graph へのマッピングを行った。

これにより、二次元空間中で互いに近い位置関係にある通信端末同士は Skip Graph 中でもおおよそ近くに配置されることになる。

図 6 に二次元空間と Skip Graph によるネットワークの対応関係を示した。左図は二次元空間における通信端末の位置を表し、右図はネットワークの概念図である。左右の図における色は対応しており、例えば左図の青マスの位置にある通信端末は右図の青丸で表されるノードと対応する。

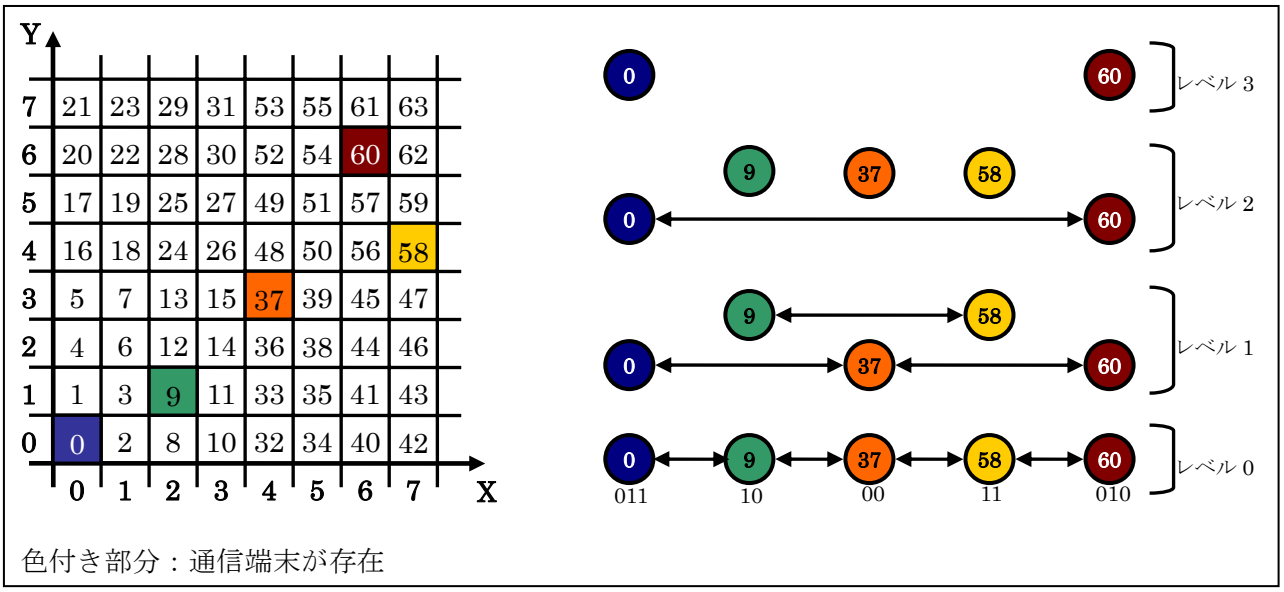


図 6 二次元空間と Skip Graph ネットワークの対応関係

第 7 章 実装概要

7.1 実装環境

7.1.1 Java RMI(Remote Method Invocation)

実装には、通信部分の簡単化のために Java の RMI (Remote Method Invocation)機能を用いた。

RMI とは Java で書かれたプログラム間の ORB (Object Request Broker) で、RPC のオブジェクトに相当する機能を果たすための Java API である。

ORB とはオブジェクト指向と組み合わされた RPC (Remote Procedure Call)。

この RMI には、異なる Java 仮想マシンにあるオブジェクトのメソッドを呼び出せる、トランスポート層などを意識する必要が無い、ソケット通信である、などの特徴がある。

7.1.2 オブジェクト管理

RMI では RMI レジストリと呼ばれる名前付けの仕組みを用いて、オブジェクトの配置に関する情報を管理しているが、その RMI レジストリへオブジェクトを登録する際の各ノードの識別に、ノードの IP アドレスと Skip Graph 上でのノードの名前を組にしたもの用いた。

これにより、一台の計算機上で仮想的なノードを複数起動することが出来るので、複数台の計算機を用意する必要なく動作確認が行える。

7.2 参考論文の擬似コードと実装

7.2.1 ノード検索の実装

ノード検索操作について文献[1]の擬似コードと今回実装した Java によるプログラムの対応関係を述べる。

図 7.2.1_1 は文献[1]の擬似コードで、検索メッセージを受け取ったノード n の処理を記述している。

処理の流れを簡単に説明する。まず、ノード n は検索開始ノードと検索 Key、検索を行うレベルを含んだ検索メッセージ(searchOp)を受け取る(*l*.1)。次に、自分の Key 値と比較し、自分の Key 値が検索する Key 値と同じならば、検索開始ノードに自分の情報を返す(*l*.2,3)。もし、検索する Key 値が自分の Key 値より大きいならば(*l*.4)、レベルが 0 以上の間、自分の右隣のノードの Key 値と検索する Key 値を比較し、検索する Key 値の方が大きい場合は右隣のノードに検索メッセージを転送する(*l*.5-9)。検索する Key 値が自分の Key 値より小さいならば左のノードに対して同様の処理を行う(*l*.10-15)。自分の Key 値と検索する Key 値の大小関係を比較した後に、レベルが 0 以上の間、右もしくは左の隣人ノードどちらか一方のみを確認するだけで済むのは、Skip Graph のノード配置が Key 値によって昇順に整列されているためである。

上記の処理を検索メッセージを受け取った各ノードで行い、最終的にレベルが 0 より小さくなった時点で検索する Key 値を持つノードは発見出来なかったとして、検索する Key 値を超えない最大(もしくは最小)の Key 値を持つノードの情報が、検索開始ノードに返されることになる(*l*.16,17)。

Algorithm 1: `search` for node n

```
1  upon receiving ⟨searchOp, startNode, searchKey, level⟩:
2  if n.key = searchKey then
3     send ⟨searchOp, n⟩ to startNode
4  if n.key < searchKey then
5     while level ≥ 0 do
6        if (nR_level).key < searchKey then
7           send ⟨searchOp, startNode, searchKey, level⟩ to nR_level
8           break
9        else level←level-1
10 else
11    while level ≥ 0 do
12       if (nL_level).key > searchKey then
13          send ⟨searchOp, startNode, searchKey, level⟩ to nL_level
14          break
15       else level←level-1
16 if level < 0 then
17    send ⟨searchOp, n⟩ to startNode
```

図 7.2.1_1 ノード検索の擬似コード

次に、今回実装した Java プログラム(SkipGraphOpImpl.java の searchOp メソッド)と擬似コードの対応関係を説明する。

このメソッドは擬似コードによる検索操作と同様に検索開始ノードと検索 Key、検索を行うレベルを引数として呼び出される。また、図中 1 は擬似コード *1*.2,3、図中 2 は擬似コード *1*.4、図中 3 は擬似コード *1*.5-9、図中 4 は擬似コード *1*.10-15、図中 5 は擬似コード *1*.16,17 とそれぞれ対応しており、処理内容は擬似コードのアルゴリズムとほぼ同様である。

```
//Key 値による検索操作を行うメソッド

public SearchNodePack searchOp(Node startNode, Key searchKey, int level) throws RemoteException{

    SearchNodePack snp = null;

    try{
        //検索 Key と同じノード Key を持つノードを発見
        if(SkipGraph.key.value == searchKey.value){
            snp = new SearchNodePack(SkipGraph.np, true);                    | 1
        }

        //左から検索メッセージ到着
        if(SkipGraph.key.value < searchKey.value){  2
            while(level >= 0){
                                                                             |
                //検索 Key の値を越えない最大のノード Key を持つノードを探す   |
                if(SkipGraph.neighborsR.getNeighbor(level).nFlag != false &&|
                        SkipGraph.neighborsR.getNeighbor(level).key.value <= searchKey.value){
                    try{                                                     |
                        SkipGraphOp sgo = *1                                 |
                        //右の隣人ノードの検索操作を呼び出す                 |
                        snp = sgo.searchOp(startNode, searchKey, level);     | 3
                        return snp;                                          |
                    }catch(Exception e){                                     |
                        e.printStackTrace();                                 |
                    }                                                        |
                }                                                            |
                --level;                                                     |
            }                                                                |
        }
        else if(SkipGraph.key.value > searchKey.value){
                                                                             |
            //左の隣人に関する処理                                           | 4
                                                                             |
        }
        //検索 Key と同じノード Key を持つノードが見つからない
        if(level < 0){                                                       |
            snp = new SearchNodePack(SkipGraph.np, false);                   | 5
        }
    }catch(Exception e){}
}

*1: (SkipGraphOp)Naming.lookup(SkipGraph.neighborsR.getNeighbor(level).np.getNodePointer());
```

図 7.2.1_2 ノード検索のプログラム(searchOp)

7.2.2 ノード参加の実装

ノード参加操作について文献[1]の擬似コードと今回実装した Java によるプログラムの対応関係を述べる。

図 7.2.2_1 は文献[1]の擬似コードで、新しくネットワークに参加するノード *n'* の処理(左)と、ノード参加に関するメッセージを受けとる既存のノード *n* の処理(右)を記述している。以下で処理の流れを *n'*、*n* の順で簡単に説明する。

まず *n'* の処理について、新規参加ノード *n'* が introducer(初期ノード)ならば、*n'* はネットワークに唯一のノードなので、隣人情報を初期化して終了する(*1*.1-3)。既に初期ノードが存在するとき、もし初期ノードの Key 値より *n'* の Key 値が大きければノード参加の処理に関する以後のメッセージは左から右に流れることを表す side = R となる。逆の場合は side = L。その後、初期ノードにレベル 0 におけるネットワーク中での自分の位置を検索させる(*1*.5-7)。検索によって得た隣人ノードにレベル 0 におけるリンク依頼を送信する(*1*.8,9)。

レベル 0 でのリンク後、自分のレベルを 1 に設定し、以下の処理を必要に応じてレベルを上昇させながら break するまで行う(*1*.10,11)。まず、左の隣人ノードが存在するならばそのノードに自分がリンクするべきノード(buddy ノード)を確認する、このとき buddy ノードが存在すればそのノードと現在のレベルにおいてリンクする(*1*.12-16)。左の隣人ノードが存在しないならば同様の処理を左右の方向を逆にして、右の隣人ノードについて行う(*1*.17)。

しかし、このような条件で buddy ノードを探した場合、左の隣人ノードが存在する限り、同レベルにおいて右方向へのリンクが行われないため、実装にあたってはこの部分の修正が必要になる。また、newBuddy という変数も while ループの初回に *1*.17 部分の処理を実行した段階では存在しておらず、この変数に関しても修正が必要になる。

上記処理の break 条件は、現在のレベルにメンバーシップベクタの値が一致するノードが存在しない場合と、隣人ノードが存在しない場合である(*1*.23,24)。*1*.25-27 ではノードのレベルを成長させている。

次に *n* の処理について説明する。リンクメッセージ(linkOp)を受け取ったノード *n* は引数の side を確認し、それに応じて右もしくは左の隣人ノードと Key 値の比較を行う(*1*.2-4)。もし *1*.4 の条件が成り立つならば、リンクメッセージは右もしくは左の隣人ノードに転送される(*1*.5)。この処理は、メッセージが伝播している間に *n'* とは別の新規ノードがこの位置に配置されていた場合などに、そのまま *n'* をこの位置に配置すると Key の順序が乱れネットワーク構造が崩れてしまうため、それを防ぐためである。

1.4 が成り立たない場合は右もしくは左の隣人ノードとして *n'* を配置し、*n' を挟んで*もう一方のノードに *n'* の参加によるリンクの繋ぎかえを依頼する(*1*.7,8)。

しかし、このリンクメッセージの処理にはメッセージを無視する条件が含まれていないため、*l*.8 のようにリンク依頼を出していたのではリンクメッセージが無限に発生し続けてしまうので、実装の際にはこのようなリンクメッセージの無限発生を防ぐ処理を加える必要がある。

buddy ノードメッセージ(buddy0p)を受け取ったノード *n* はまず、尋ねられているメンバーシップベクタの桁を確認し、まだ決定されていない場合は該当する桁をランダムに決定し、左右の隣人情報を初期化する(*l*.9-13)。

もし、該当する桁のメンバーシップベクタの値が *n'* のメンバーシップベクタの値と一致するならば、自分の情報を *n'* に返す(*l*.14, 15)。異なるならば、メッセージが左から到達した場合は右の隣人ノード、逆の場合は左の隣人ノードへ buddy ノードメッセージを転送する(*l*.17, 18)。これにより、同レベルにおいて到達可能なノード全てに対して、buddy ノードが見つかるまで buddy ノードメッセージを転送出来る。

もし、該当する桁の値が一致するメンバーシップベクタを持つノードが存在しなかった場合は、発見出来なかったという情報を *n'* に返す。

Algorithm 2: `insert` for new node n'

```
1  if introducer = n' then
2    n'L_0 = ⊥
3    n'R_0 = ⊥
4  else
5    if introducer.key < n'.key then side = R
6    else side = L
7    send ⟨searchOp, n', n'.key, 0⟩ to introducer
8    upon receiving ⟨searchOp, neighbor⟩:
9    send ⟨linkOp, n', side, 0⟩ to neighbor
10   level← 1
11   while true do
12     if n'L_{level-1} ≠ ⊥ then
13       send ⟨buddyOp, n', level, m(n')_level⟩ to
         n'L_{level-1}
14       upon receiving ⟨buddyOp, newBuddy, level⟩:
15       if newBuddy ≠ ⊥ then
16         send ⟨linkOp, n', R, level⟩ to newBuddy
17     else if (n'R_{level-1} ≠ ⊥) ∧ (newBuddy = ⊥)
18     then
19       send ⟨buddyOp, n', level, m(n')_level⟩ to
         n'R_{level-1}
20       upon receiving ⟨buddyOp, newBuddy, level⟩:
21       if newBuddy ≠ ⊥ then
22         send ⟨linkOp, n', L, level⟩ to newBuddy
23       else break
24     else break
25     level←level+1
26     n'L_level = ⊥
27     n'R_level = ⊥
```

Algorithm 3: `Insert` for existing node n

```
1  upon receiving ⟨linkOp, n', side, level⟩:
2  if side = R then cmp = <
3  else cmp = >
4  if (n side_level).key cmp n'.key then
5    send ⟨linkOp, n', side, level⟩ to n side_level
6  else
7    adjust links to add n' as side neighbor
8    send ⟨linkOp, n', otherSide, level⟩ to n side_level
9  upon receiving ⟨buddyOp, n', level, val⟩ from side L(R):
10 if m(n)_level = ⊥ then
11   m(n)_level = getCoin()
12   nL_level = ⊥
13   nR_level = ⊥
14 if m(n)_level = val then
15   send ⟨buddyOp, n, level⟩ to n'
16 else
17   if nR_level(L_level) ≠ ⊥ then
18     send ⟨buddyOp, n', val, level⟩ to nR_level
       (nL_level)
19   else
20     send ⟨buddyOp, ⊥, level⟩ to n'
```

図 7.2.2_1 ノード参加の擬似コード

図 7.2.2_2 に示すプログラムは *n'* の処理を実装したものである。以下で擬似コードとこのプログラム(InsertOpThread.java)の対応関係を説明する。

まず、図中 **1** 部分で自分が初期ノードであるかの確認を行っている。もし、そうであるならば隣人情報を初期化して終了する(*l*.1-3)。

図中 **2** では初期ノードの Key 値と自分の Key 値を比較することで、ノード参加に関するメッセージをどの方向に転送するかの決定を行っている(*l*.4-6)。

この決定を行った後に図中 **3,4** の処理によって、*n'* は初期ノードにレベル 0 における自分の Key 値による検索を依頼する。この検索時点では *n'* と同じ Key 値を持つノードはネットワーク中に存在しないので、必ず自分の Key 値を超えない最大(もしくは最小)の Key 値を持つノード(自分の Key 値が初期ノードの Key 値より大きい場合は左の隣人ノード、逆の場合は右の隣人ノード)の情報が返ってくることになる(*l*.7)。そして、*n'* はこの検索によって得た隣人ノードに対して、自分とのリンクを依頼する(*l*.8,9)。

レベル 0 で適切な位置に自分を配置した後、図中 **5** の処理によって *n'* は 1 以上のレベルにおいてリンクを行うことになる。そのために、まず *n'* は自分のレベルを一つ上げ(レベルが 0 の時のみ)、次に level - 1 において右もしくは左に隣人ノードが存在しているか確認する、このとき隣人ノードが存在しているならばその隣人ノードに対して自分とリンク可能なノードがあるか尋ねる。

ここで、自分とリンクするノードの決定には buddyOp メソッドを呼び出すことによって行うが、この buddyOp メソッド呼び出し時にメンバーシップベクタの level 桁目が必要になるので、まずメンバーシップベクタの level 桁目を決定(図中 **5** 中程)した後に隣人ノードの buddyOp メソッドを呼び出す。そして、この buddyOp メソッドによってリンク可能なノードが発見できた場合はそのノードとリンクを行う。

前述したように、この 1 以上のレベルにおいてリンクを行うための一連の処理を擬似コード通りの条件文で実装したのでは左方向のリンクしか行われなくなってしまう。

そこでこの問題に対処するために *l*.17 では else if 文で右方向のリンクに関して条件分岐していた部分を if 文に変更した(図中 **5** 下部)。

また、break の条件については buddy ノードが存在しない場合と隣人ノードが存在しない場合の二種類で、擬似コードと同様(図中 **6,8**)であるが、擬似コードと同じ場所に記述したのでは正しく動作しないため、図中 **6** に示すようにその位置を変更した。

レベルを上げる位置についても問題があったため、その位置を変更した(図中 **5** 上部,**7**)。

その他の部分についての処理は、擬似コードとほぼ同様の処理を行っている。

```
//挿入操作を行うスレッド

class InsertOpThread implements Runnable{
    Node introducer;
    SearchNodePack snp;
    BuddyNodePack bnpL,bnpR;
    SkipGraphOp sgo;

    public InsertOpThread(Node introducer){
        this.introducer = introducer;
        bnpL = new BuddyNodePack(new Node(), false, 0);
        bnpR = new BuddyNodePack(new Node(), false, 0);
    }

    public void run(){
        Boolean side;

        //自分が初期ノードであるかを確認
        if(introducer.np.getNodePointer() == SkipGraph.np.getNodePointer()){
            SkipGraph.neighborsL.setNeighbor(0, new Node());
            SkipGraph.neighborsR.setNeighbor(0, new Node());
        }else{
            if(introducer.key.value < SkipGraph.key.value) side = true;
            else side = false;

            //初期ノードに自分の隣人の検索を依頼
            try{
                sgo = SkipGraph.connect(introducer.np);
                snp = sgo.searchOp(new Node(SkipGraph.key, SkipGraph.np), SkipGraph.key, 0);
            }catch(Exception e){}

            //レベル 0 でリンクを張る
            try{
                sgo = SkipGraph.connect(snp.np);
                sgo.findLinkOp(new Node(SkipGraph.key, SkipGraph.np), side, 0);
            }catch(Exception e){}

            //以下でレベル 1 以上のリンクを張る
            SkipGraph.level = 1;
            SkipGraph.neighborsL.setNeighbor(SkipGraph.level);
            SkipGraph.neighborsR.setNeighbor(SkipGraph.level);
            for(;;){
                //右もしくは左どちらかの隣人ノードが存在する
                if(SkipGraph.neighborsL.getNeighbor(SkipGraph.level - 1).nFlag == true ||
                        SkipGraph.neighborsR.getNeighbor(SkipGraph.level - 1).nFlag == true){

                    //左の隣人ノードが存在するならば、
                    //buddyOp に右からのメッセージであることを伝えるために side = false
                    if(SkipGraph.neighborsL.getNeighbor(SkipGraph.level - 1).nFlag != false){
                        side = false;
                        //メンバーシップベクタを参照して隣人ノードかどうかを判定
                        try{
                            //左の隣人ノードへ接続
                            sgo = SkipGraph.connect(*1);
                            //メンバーシップベクタの level 桁目を決定する
                            SkipGraph.mv.extMembershipVector(SkipGraph.level);
```

1

2

3

4

5

```
                        bnpL = sgo.buddyOp(new Node(SkipGraph.key, SkipGraph.np), side,
                                SkipGraph.level, *2);
                    }catch(Exception e){}

                    //buddy ノード発見時の処理
                    if(bnpL.bFlag != false){
                        try{
                            sgo = SkipGraph.connect(bnpL.buddyNode.np);
                            //隣人ノードに自分をリンクするように依頼
                            sgo.findLinkOp(new Node(SkipGraph.key, SkipGraph.np),
                                true, SkipGraph.level);
                        }catch(Exception e){}
                    }
                }

                //右の隣人ノードが存在する場合には左右の方向を逆にして同様の処理を行う
                if(SkipGraph.neighborsR.getNeighbor(SkipGraph.level - 1).nFlag != false){

                    右の隣人ノードが存在する場合の処理

                }

                //buddy ノードが存在しない
                if(bnpL.bFlag == false && bnpR.bFlag == false) break;

                //レベルを 1 上げる
                SkipGraph.level++;
                SkipGraph.neighborsL.setNeighbor(SkipGraph.level);
                SkipGraph.neighborsR.setNeighbor(SkipGraph.level);

            //隣人ノードが存在しない
            }else break;
        }
    }
}
*1: SkipGraph.neighborsL.getNeighbor(SkipGraph.level - 1).np
*2: SkipGraph.mv.getMembershipVectorAt(SkipGraph.level)
```

5

6

7

8

図 7.2.2_2 *n’*の処理プログラム(InsertOpThread)

図 7.2.2_3 に示すプログラムは *n* の処理(findLinkOp)を実装したものである。以下で擬似コードとこのプログラム(SkipGraphOpImpl.java の findLinkOp メソッド)の対応関係を説明する。

今回の実装では、前述したメッセージの無限発生に対して図中 **1** の条件文が成立する際に空処理を行うことによって対処している。これはリンクメッセージ(findLinkOp)が複数回発生し続けるとメッセージの送信ノードと受信ノードが同一のノードになることを利用している。

図中 **2** の処理は図 7.2.2_1 の右側コード *1*.4,5 と対応しており、リンクしようとする二つのノード間に不適切なノードが入り込んでいる場合にリンクメッセージを隣人ノードに転送することでネットワークの構造が崩れることを防いでいる。

図中 **3** ではノード右側での双方向リンクの作成を行っている。その手順は、まず tmpNode に *n* の右の隣人ノードの参照を保持する。次に、*n* の右の隣人ノードとして、findLinkOp の呼び出し元である *n'* を登録する。その後、*n'* に自分とリンクするよう依頼する。この時点で、ノード *n* の右側において *n'* と双方向リンクが作成される。

ノード *n* の右側においてリンクが完成した後、図中 **4** の処理によって *n'* を挟んで反対側に位置するノードでも同様に双方向リンクを作成する。この時点で *n'* は左右の隣人と双方向リンクを作成することになる(*1*.7,8)。

図中 **3** において tmpNode を用いたのは、図中 **4** で *n'* を挟んで反対側に位置するノードとリンクを行う際に右の隣人ノードの参照を使用するが、この参照は図中 **3** で行う右の隣人ノードの修正によって失われてしまうので、これを防ぐためである。

今回は side == true の場合を例にしたが、false の場合はこの処理の左右を逆にして同様の処理が行われる。

```
//startNode を挟んだ両隣のノードでリンクを行うメソッド

public void findLinkOp(Node startNode, Boolean side, int level) throws RemoteException{

    //startNode の左の隣人ノードでの処理
    if(side == true){

        //最初の if 文によってこの処理の無限ループを回避
        if(SkipGraph.key.value == startNode.key.value){}                    1

        //startNode とその左隣のノードの間に他のノードが存在する場合は隣の Node に転送
        else if(SkipGraph.neighborsR.getNeighbor(level).nFlag != false &&
                SkipGraph.neighborsR.getNeighbor(level).key.value < startNode.key.value){      2

            SkipGraphOp sgo = SkipGraph.connect(SkipGraph.neighborsR.getNeighbor(level).np);
            sgo.findLinkOp(startNode, side, level);

         }else{

            //通常の処理
            try{
                Node tmpNode = SkipGraph.neighborsR.getNeighbor(level);

                //右の隣人集合に startNode を追加し、レベルを修正
                SkipGraph.neighborsR.setNeighbor(level, startNode);
                if(SkipGraph.level < level) SkipGraph.level = level;                    3

                //startNode に自分を登録させる
                //adjustLinkOp:side で指定した方向に隣人ノードを追加するメソッド
                SkipGraphOp sgo = SkipGraph.connect(startNode.np);
                sgo.adjustLinkOp(new Node(SkipGraph.key, SkipGraph.np), !side, level);

                //startNode の右隣のノードに同様の処理を依頼
                if(level == 0 && tmpNode.nFlag != false){                               4
                    sgo = SkipGraph.connect(tmpNode.np);
                    sgo.findLinkOp(startNode, !side, level);
                 }
            }catch(Exception e){}
         }
    }else{
        startNode の右の隣人ノードでの処理
    }
}
```

図 7.2.2_3 *n* の処理プログラム(findLink0p)

図 7.2.2_4 に示すプログラムは *n* の処理(buddyOp)を実装したものである。以下で擬似コードとこのプログラム(SkipGraphOpImpl.java の buddyOp メソッド)の対応関係を説明する。

図中 **1** で *n* はメンバーシップベクタ level 桁目を確認する、この時点で level 桁目が決まっていない場合は、ランダムに 0,1 を発生させることによってその値を決定し、さらに隣人情報の初期化を行う(*l*.10-13)。図中 **2** で、*n* と *n'* のメンバーシップベクタの値が一致しているかを確認し、一致するならば自分の情報を buddy ノードとして返す(*l*.14,15)。異なるならば buddy メッセージがどちらの方向から届いたのか side によって確認し、適切な方向の隣人ノードにメッセージを転送する。この転送を繰り返し、転送出来る隣人ノードが無くなった時点で処理を終了、ノードを発見出来なかったという情報を返す(*l*.16-20)。

```
//buddy ノードの検索を行うメソッド

public BuddyNodePack buddyOp(Node startNode, Boolean side, int level, String prifix)
                                                                    throws RemoteException{

    BuddyNodePack bnp = new BuddyNodePack(new Node(SkipGraph.key, SkipGraph.np),
                                                                        false, level);

    if(*1 == null) SkipGraph.mv.extMembershipVector(level);

    SkipGraph.neighborsR.setNeighbor(level);                                    1
    SkipGraph.neighborsL.setNeighbor(level);
    if(SkipGraph.level < level)SkipGraph.level = level;

    //自分のメンバーシップベクタの値と等しい
    if(SkipGraph.mv.getMembershipVectorAt(level).equals(prifix)){                          2
        bnp = new BuddyNodePack(new Node(SkipGraph.key, SkipGraph.np), true, level);
    }else{

        //自分の右隣のノードが存在すれば、そのノードにメンバーシップベクタの判定を依頼
        if(side == true){
            if(SkipGraph.neighborsR.getNeighbor(level - 1).nFlag != false){
                try{                                                                                3
                    SkipGraphOp sgo = SkipGraph.connect(*2);
                    return sgo.buddyOp(startNode, true, level, prifix);
                }catch(Exception e){}
            }else bnp = new BuddyNodePack(new Node(SkipGraph.key, SkipGraph.np), false, level);
        }
        //自分の左隣のノードが存在すれば、左右方向を逆にして同様の処理
        else if(side == false){
            左の隣人ノードでの処理
        }
    }
    return bnp;
}

*1: SkipGraph.mv.getMembershipVectorAt(level)
*2: SkipGraph.neighborsR.getNeighbor(level - 1).np
```

図 7.2.2_4 *n* の処理プログラム(buddyOp)

7.3 プログラム構成

今回実装したプログラムファイルは以下の通りである。

BuddyNodePack.java

Buddy ノードに関する情報を扱うクラス

隣人ノードを探す際に使用

CommandThread.java

実験用のコマンド受け付けスレッドのクラス

範囲検索などのコマンドを待ち受けるのに使用

DeleteOpThread.java

ノード削除操作を行うクラス

このスレッドが他の計算機上の RMI メソッドを呼び出し、削除処理を実行する

InsertOpThread.java

ノード参加操作を行うクラス

このスレッドが他の計算機上の RMI メソッドを呼び出し、参加処理を実行する

Key.java

ノードの Key を定義するクラス

今回は int 型を使用したが、論文中に型の指定は無いので変更可能

MembershipVector.java

ノードのメンバーシップベクタを定義するクラス

今回は String 型を使用したが、論文中に型の指定は無いので変更可能

値の決定は 0、1 を一桁ずつランダムに発生させることで行う

Neighbors.java

ノードの隣人ノードに関する情報を扱うクラス

Node.java

ノードに関する情報を扱うクラス

Key クラス、NodePointer クラス、MembershipVector クラスが含まれる

NodePointer.java

RMI によって他計算機に接続する際に用いる計算機の参照を扱うクラス
IP アドレスとノード名を組にしたもの

SearchNodeMvPack.java
メンバーシップベクタによるノード検索時に必要な情報を扱うクラス
ノードへの参照やメッセージ転送のためのフラグなどが含まれる

SearchNodePack.java
Key によるノード検索時に必要な情報を扱うクラス
ノードへの参照と発見確認用のフラグが含まれる

SearchNodeRqPack.java
範囲検索によるノード検索時に必要な情報を扱うクラス
発見される複数のノードを扱うために Vector クラスを使用

SearchOpThread.java
ノード検索操作を行うクラス
このスレッドが他の計算機上の RMI メソッドを呼び出し、検索処理を実行する

SkipGraph.java
Skip Graph 本体
仮想端末からこのプログラムを起動することで、ネットワークを構築する
他の計算機に接続するためのメソッド、Key を座標値に変換するメソッド、座標値を
Key に変換するメソッド、あるノードとの距離を返すメソッドなどが含まれる

SkipGraphOp.java
RMI メソッドのインターフェースを定義
RMI を使用するにはこの宣言が必要

SkipGraphOpImpl.java
RMI メソッドの本体
他の計算機から呼び出される RMI メソッドの処理を記述

第 8 章 動作確認

8.1 確認手順

動作確認は一台の Linux 計算機を用いて、一つの仮想端末を一つのノードとみなし、仮想的な P2P ネットワークを構築することで行った。

手順は、まず今回作成したプログラムを複数の仮想端末上で起動することで、端末毎にノードを作成し仮想的な P2P ネットワークを構築する。

そしてこの P2P ネットワークに対して、指定した方形範囲内に存在するノードの検索(範囲検索)と、特定のノードの近傍に存在するノードの検索(近傍検索)を実行し、その結果を確認した。

8.2 実行画面

今回作成したプログラムによるノード情報の出力画面が図 8.2 である。今回の実装では各ノードはレベル、メンバーシップベクタ、Key、そして左右の隣人ノードの情報を保持している。隣人ノードの情報(実行に用いた計算機のネットワーク設定を行っていないため、IP アドレスが 127.0.0.1)にはノードの参照先とそのノードが持つ Key 値が含まれており、この情報によって各種メッセージの転送先を決定している。

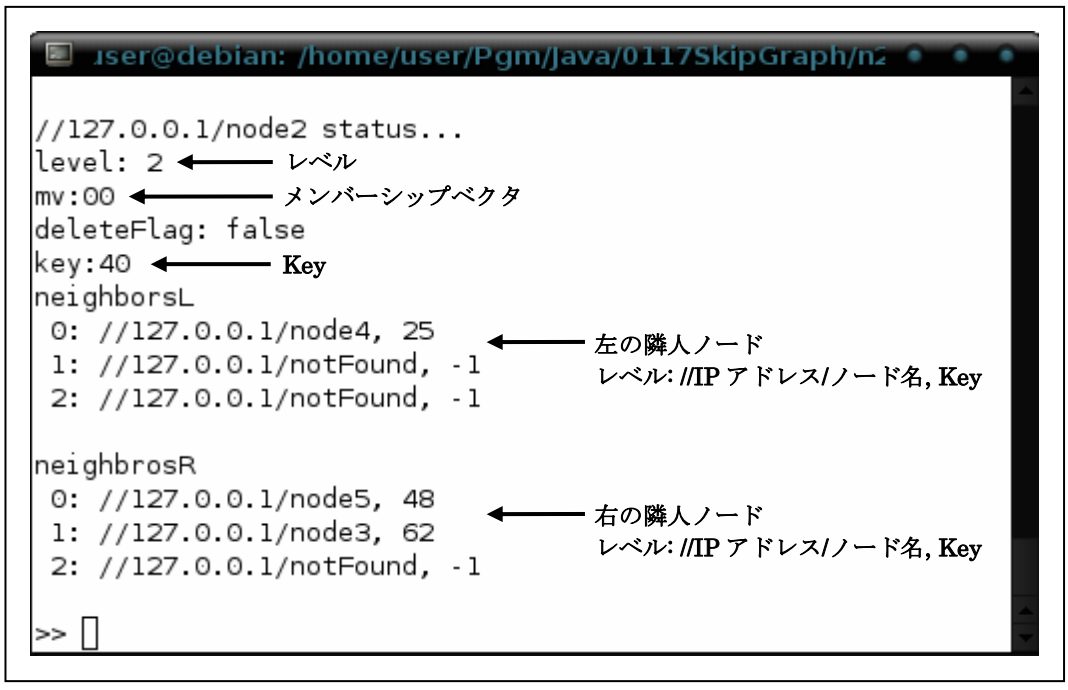


図 8.2 実行画面

8.3 実行結果

図 8.3_1 に今回想定した二次元空間における通信端末の配置と実際にプログラムを実行することで構築された P2P ネットワークの概念図を対応させて示す。

また、図 8.3_2 ではノード毎に実際のプログラム実行画面を示し、図 8.3_3 と図 8.3_4 ではそれぞれ範囲検索と近傍検索の実行結果を抜粋して示す。

図 8.3_1 の左図における色付きのマスは二次元空間中に存在する通信端末の位置を表している、また右図ではこの位置関係を想定してプログラムを実行した際に構築された P2P ネットワークを示している。左右の図において色は対応しており、例えば左図の青マスにある通信端末は右図の青丸で表されるノードと対応する。

この図 8.3_1 より二次元空間中でのノード間の位置関係が P2P ネットワーク中でも大よそ反映されていることが確認できる。

ただし、Z-ordering によるコード付けを行った場合でも、例えば座標(4,3)と座標(4,4)のように、二次元空間上で近い位置関係であってもコードが大きく離れてしまう可能性がある。

図 8.3_3 では座標(2,0)と座標(5,4)によって作られる方形の範囲内に含まれるノードの検索を行った結果を示している。この場合の検索範囲内には座標(2,1)にあるノード(node 2)と座標(4,3)にあるノード(node 4)が該当するが、正しく発見していることを確認出来る（図 8.3_3 実行結果下線部より）。

図 8.3_4 では座標(6,6)にあるノード(node 3)の近傍ノードを検索した結果を示している。この場合では座標(7,4)にあるノード(node 1)が該当するが、正しく発見していることを確認出来る(図 8.3_4 実行結果下線部より)。

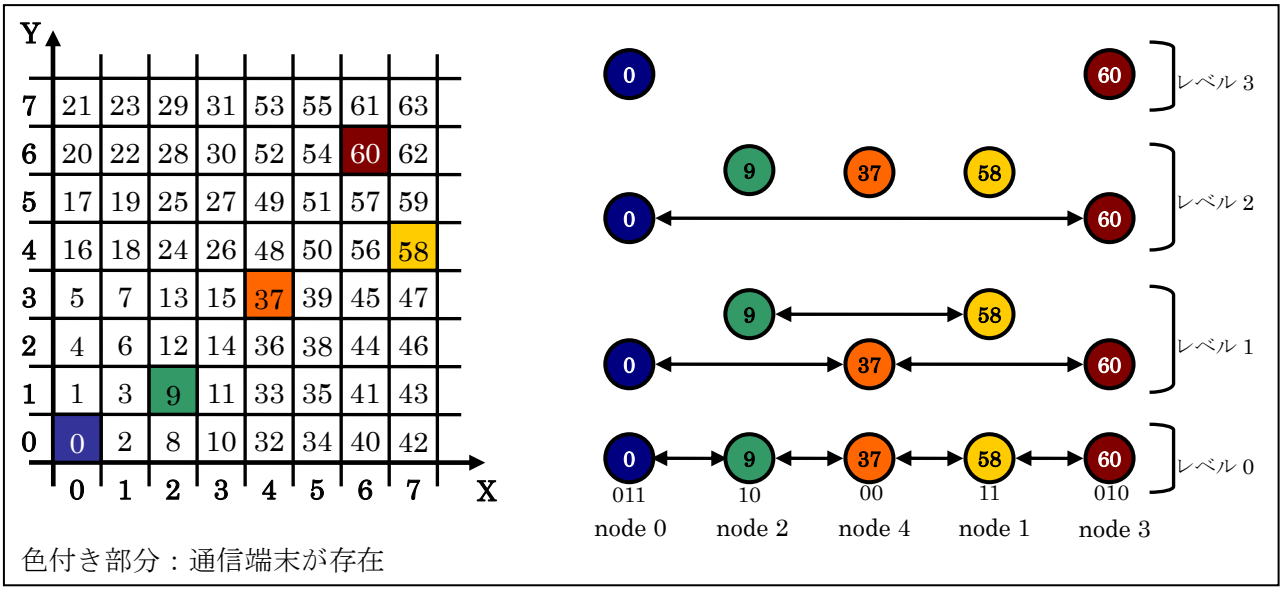


図 8.3_1 二次元空間と構築された P2P ネットワークの対応関係

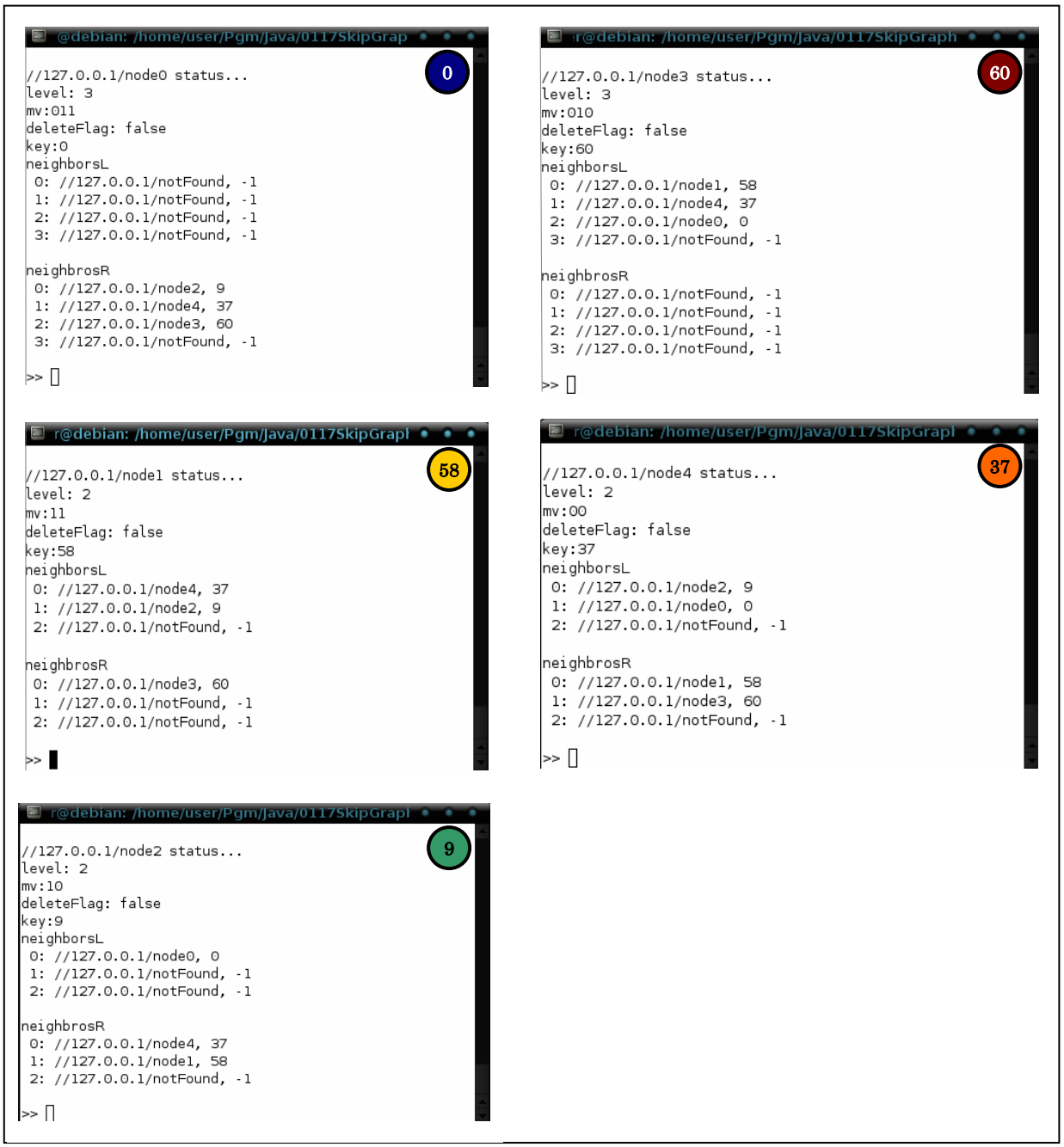
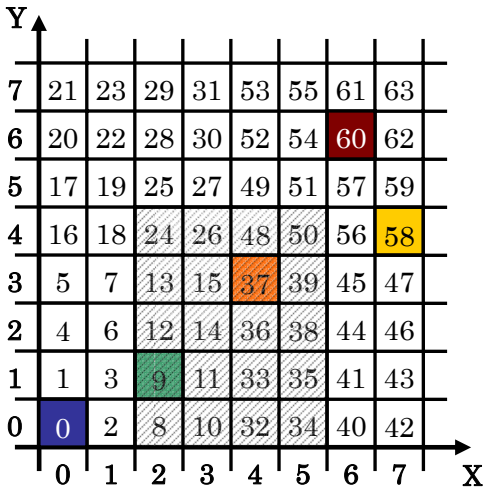


図 8.3_2 ノード毎のプログラム実行画面



左図の灰色網掛け部分の範囲に含まれるノードの検索を行った。
この場合では座標(2,1)にあるノード(node 2)と座標(4,3)にあるノード(node 4)が該当する。

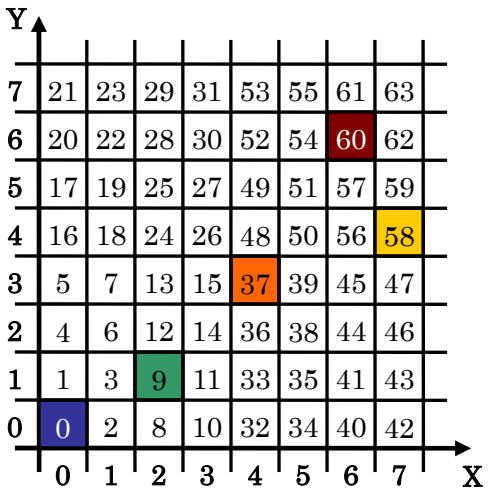
```
r@debian: /home/user/Pgm/Java/0117SkipGraph

>> r 2-0-5-4
2
0
5
4
RangeQuery: Start
[8, 15, 24, 24, 26, 26, 32, 39, 48, 48, 50, 50]
value8
value15
SearchOpRq: Start
SearchOp: Start
searchOpImpl: Start
searchOpImpl: Key > searchKey
//127.0.0.1/node3?//127.0.0.1/node4????
searchOpImpl: Return
SearchOp: Finish
result: //127.0.0.1/node2
value24
value24
```

中略

```
SearchOp: Finish
result: //127.0.0.1/node4
value48
value48
SearchOpRq: Start
SearchOp: Start
searchOpImpl: Start
searchOpImpl: Key > searchKey
//127.0.0.1/node3?//127.0.0.1/node1????
searchOpImpl: Return
SearchOp: Finish
value50
value50
SearchOpRq: Start
SearchOp: Start
searchOpImpl: Start
searchOpImpl: Key > searchKey
//127.0.0.1/node3?//127.0.0.1/node1????
searchOpImpl: Return
SearchOp: Finish
>>
```

図 8.3_3 範囲検索の実行結果



左図の座標(6,6)にあるノード(node 3)の近傍のノードを検索した。
この場合では座標(7,4)にあるノード(node 1)が該当する。

```
er@debian: /home/user/Pgm/Java/0117SkipGraph/
>> n
tKey: 58 dicord[0]: 7 dicord[1]: 4 di: 2.23606797749979
4488
NearestNeighborSearch: Start
[48, 63, 96, 96, 98, 98, 104, 104, 106, 106, 144, 145, 1
48, 149, 192, 192]
SearchOpRq: Start
SearchOp: Start
searchOpImpl: Start
searchOpImpl: Key > searchKey
//127.0.0.1/node3?//127.0.0.1/node1????
searchOpImpl: Return
SearchOp: Finish
//127.0.0.1/node1
```

中略

```
searchOpImpl: Finish
SearchOp: Finish
//127.0.0.1/node1
SearchOpRq: Start
SearchOp: Start
searchOpImpl: Start
searchOpImpl: Key < searchKey
searchOpImpl: False
searchOpImpl: Finish
SearchOp: Finish
//127.0.0.1/node1
Found: //127.0.0.1/node1
>>
```

図 8.3_4 近傍検索の実行結果

第 9 章 結論

今回、Skip Graph と Z-ordering を用いて情報検索のための P2P ネットワークを自ら実装し、実際に動作させることで正しく実装出来ていることを確認した。

このような情報検索ネットワークはカーナビや携帯電話、無線 IC タグの普及に伴い、例えば道路交通情報やイベント情報の共有、センサーネットワークによる情報の収集といった分野への応用が期待され、その重要度は増すと考えられる。

今後は本研究で得た知識や経験を生かし、複数台の通信端末を用いての動作確認や実際の利用環境への応用、その他の実装方法との比較、また今回の実装では非常に効率の悪かった範囲検索の効率化や柔軟性を持たせるために発生した冗長なクラス構成の見直し、RMI のバケツリレーのような呼び出しによる障害耐性の低さを改善するためのノード間通信方法の変更などを行っていきたい。

第 10 章 謝辞

最後に、本研究を進めるにあたりゼミを中心に最後まで熱心なご指導、ご助言を頂きました田中章司郎教授に深く感謝の意を表すとともに心より御礼申し上げます。また、同じ研究室の今井拓也さん、木村眞吾さん、的場祥仁さん、皆木健太さんにも色々とご協力、ご助言頂いたことに御礼申し上げます。なお、本研究で作成したプログラム、発表資料などの全ての著作権を田中章司郎教授に譲渡いたします。

文献

[1] James Aspnes, Gauri Shah. Skip Graphs. *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2003, pp. 384-393.

[2] Orenstein.J, Merrett.T. A class of data structures for associative searching. *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1984, pp. 181-190.

[3] Michael Miller 著 トップスタジオ 訳 大和総研 情報技術研究所 監修
“P2P コンピューティング入門” (翔泳社 2002)

[4] “UNIX magazine October 2006 AUTUMN”

[5] Hari Balakrishnan, M.Frans Kaashoek, David Karger, Robert Morris, Ion Stoica. Looking Up Data in P2P Systems. *Communications of the ACM.* 2003, Vol.46, No.2, pp. 43-48.

[6] Antony Rowstron, Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *International Conference of Distributed Systems Platforms(Middleware).* 2001, pp.329-350.

[7] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, Scott Shenker. A Scalable Content-Addressable Network. *Proceedings of the ACM Symposium on Communications Architectures and Protocols(SIGCOMM)*, 2001, pp.161-172.

[8] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *Proceedings of ACM SIGCOMM.* 2001.

[9] William Pugh. Skip Lists: a Probabilistic Alternative to Balanced Trees. *Communications of the ACM.* 1990, Vol.33, No.6, pp.668-676.

その他関連文献

・ Kenneth L. Calvert/Michael J. Donahoo 共著　小高知宏 監訳
“TCP/IP ソケットプログラミング JAVA 編”（オーム社 2003）

・ Nicholas J.A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, Alec Wolman.
SkipNet: A Scalable Overlay Network with Practical Locality Properties.
Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03).

・ 小高 知宏 著
“TCP/IP JAVA ネットワークプログラミング”（オーム社 2002）

・ Wikipedia
http://ja.wikipedia.org/