平成２３年度　東邦大学大学院理学研究科情報科学専攻　修士論文

P2P システムにおける検索負荷分散方式に関する研究

学籍番号　６５１００１３　　氏名　唐叡

要　旨

現在、P2P システムは、ネットワークをより効率よく運用できる方法として、ファイル共有などの様々な応用に利用されている。P2P システムにおける重要な機能に、情報が存在するノードを検索する技術がある。一般に情報検索においては、人気のあるコンテンツに対してクエリ（問い合わせ）が集中する問題があり、キャッシュがその問題点を緩和する方法として利用されてきた。しかし、従来の方式は効率が悪かったり、管理の手間がかかるものであった。本研究ではクエリの到着頻度に応じて、キャッシュの作成確率が高くなることで、人気が高いノードにはキャッシュが作成されやすくなる方式を提案する。また、キャッシュに有効期限を設けることで、アクセスがなくなったキャッシュは自動的に消去される方式にすることで、管理の手間を掛けない方式を提案する。この提案方式を、シミュレーションによって評価し、従来方式と比べて検索ホップ数を改善するとともに、保持すべきキャッシュの量を削減できることを明らかにした。

目 次

第１章 序論

近年、ＡＤＳＬやＦＴＴＨといった広帯域・高速で常時接続可能のインターネットサービスが広く普及している。また、一般のコンピュータの処理能力の向上により、大容量のコンテンツの送受信を比較的簡単に行うことができる。このため、従来のクライアント・サーバモデルに代わり、Ｐ２Ｐ（Peer to Peer）技術が注目されている。

Ｐ２Ｐはサーバを用いず、エンドユーザ同士で直接通信を行うという特徴を持つ。Ｐ２ＰはNapster[1]、Gnutella[2]、Winny[3]、などのファイル共有への応用が広く知られている。これらはユーザ同士が同じ時間を共有する必要がない非同期型アプリケーションである。また、ユーザ同士が同じ時間を共有する同期通信型アプリケーションへの応用例としてIP電話やテレビ会議として利用されるSkype[4]が挙げられる。

また、Gnutella、Winny は非構造化（unstructured）オーバレイ[5]に分類されている。この非構造化オーバレイは、誰を隣接ノードにするかのトポロジに制約がない、存在するオブジェクトでも発見できない可能性がある、基本的にフラッドルーティングを用いた柔軟な検索が可能だが、ネットワークの輻輳や遅延の起きやすさで拡張性に問題がある、という特徴を持っている。

上記のようなＰ２Ｐ（Peer to Peer）システムとは別に、分散配置されたコンテンツ及びノードの検索技術として、CAN[6]、Chord[7]、Pastry[8]、Tapestry[9]などのＤＨＴ（Distributed Hash Table：分散ハッシュテーブル）が盛んに研究開発されている。ＤＨＴはコンテンツやノードの識別にハッシュ値を用い、そのハッシュ値を一定の範囲から受け持つノードを自動的に決定して、検索対象の探索を高速化する技術である。ノードはＤＨＴによりフラットなオーバレイネットワークにまとめられている。このDHTをアルゴリズムとしたネットワークは、構造化（structured）オーバレイ[5]のひとつである。

この構造化オーバレイは、誰を隣接ノードにするかのトポロジに制限がある、存在するオブジェクトは大抵発見できる、柔軟な検索が難しい、拡張性は高いがシステム的にコストが大きくなりやすいという特徴を持っている。

一般に、Ｐ２Ｐシステムの負荷は特定の人気のある情報を管理しているノードに集中する特徴がある。従って、Ｐ２Ｐのアクセス効率を高めるためには、人気のある情報に対する検索に対して効率的に応答するシステムが必要となる。ＤＨＴの一つであるＣＡＮに対して、検索要求の通過する経路が集中するノードに対して、キャッシュノードを設置して負荷を分散する方式が提案されている。しかし、この方式ではどのノードに負荷が集中しているかの情報を集めるコストや、キャッシュノードを起動停止する管理コストがかかる問題がある。一方、DHTの一つのChordでは、隣接するノードに情報をキャッシュしておくことで、負荷を分散する方法が提案されている。しかし、コンテンツの人気に応じてキ

ャッシュが配置されていないため、無駄なキャッシュが配置されるという問題があった。

本研究では、DHT の一つである Chord において、人気のあるコンテンツの情報に対して高い確率でキャッシュを作成し、人気のないコンテンツの情報についてはキャッシュが作られにくくすることで、無駄なキャッシュがない効率的なキャッシュ配置方式を提案する。また、キャッシュに対して有効期限を設けることで、キャッシュが使われなくなった時点でキャッシュが削除されるようにした。その結果、キャッシュを管理する手間がかからないという利点がある方法とした。

本論文における構成を以下に示す。第 2 章では、本研究の関連研究としてＰ２Ｐの概要、DHＴのアルゴリズム、既存の分散ハッシュアルゴリズムの説明を行い。第３章では提案方式について説明する。第４章では提案方式におけるシミュレーションを通して考察を行う。最後に第５章で本研究の結論を述べる。

第 2 章 関連研究

2.1 Ｐ２Ｐ

2.1.1 概要

Ｐ２Ｐとは「Peer to Peer」の略で対等なもの（Peer）同士による通信を行うことである。サーバのように決まった情報の管理・伝達を持たずに、エンドユーザ同士が直接情報のやり取りをするネットワーク形態の一つである。なお、ピアは「ノード（Node）」とも呼ばれる。

2.1.2 クライアント・サーバモデル

現在、インターネットを代表とした主流なネットワークで使用されているのが、クライアント・サーバモデル（図 2.1）である。

クライアント・サーバモデルは、サービスを利用するクライアントと、サービスを提供するサーバによって構成されるネットワーク形態である。クライアントはサーバに対して要求を出し、サーバから結果を受け取る処理を行う。それに対して、サーバは常にサービスを提供しており、クライアントからの要求を受け取り、その要求を処理して結果をクライアントに返す処理を行う。

このように各ピアの役割が固定されているため、クライアントからの要求は全てサーバを通して処理を行うことで、サーバへの付加が大きく、サーバが故障してしまうことがある。サーバが故障するとクライアントはサービスを利用できなくなるという問題点がある。

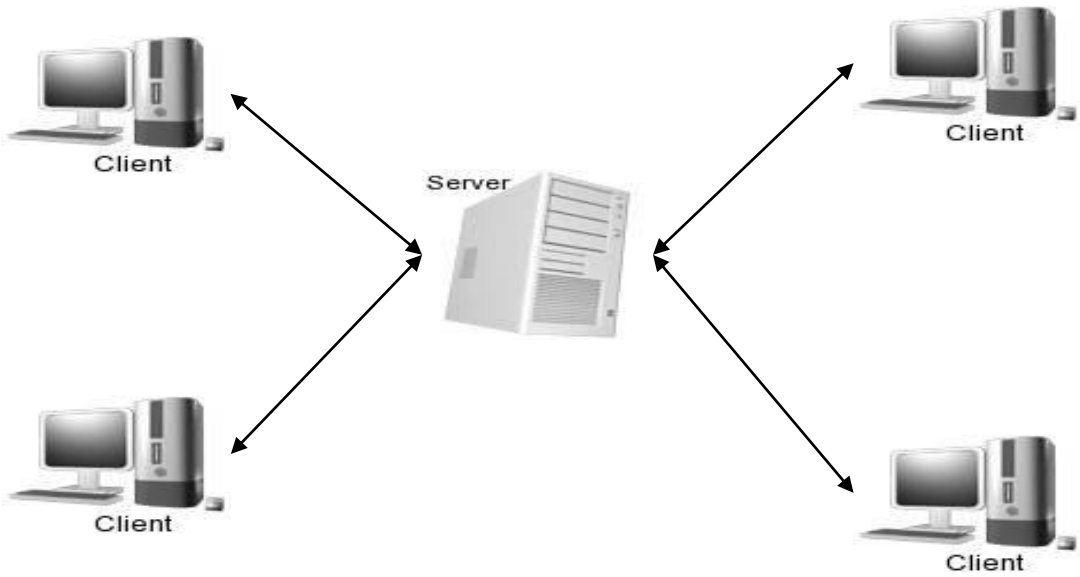

図 2.1　クライアント・サーバモデル

2.1.3　Ｐ２Ｐ（Peer to Peer）モデル

Ｐ２Ｐモデルは、クライアント・サーバモデルのように最初からクライアントとサーバの役割が決まっておらず、処理を行う際要求を出すピアがクライアントとなり、要求を受け取るピアがサーバとなるネットワーク形態である。

Ｐ２Ｐモデルは大きく分けてHybridＰ２Ｐ型とPureＰ２Ｐ型に分類される。

・Hybrid（ハイブリッド）Ｐ２Ｐ型（図2.2）

HybridＰ２Ｐ型は、ネットワーク上で各クライアントの位置と各クライアントのもつ情報を管理するインデックスサーバを用い、情報の交換をピア同士で行う方式である。各クライアントがインデックスサーバに検索情報を問い合わせ、その結果を基に検索した情報を持つと直接通信を行う。

この方式では、インデックスサーバがアカウントの管理やユーザの認証を行うため、各クライアントの負担が少なく、構成が単純でセキュリティ管理が容易であるというメリットがある。しかし、インデックスサーバに処理が集中することで、サーバダウンした場合クライアントがサービスを利用できなくなるデメリットがある。また、ユーザの増加によりサーバの増設をしなければならず、コストが非常にかかる。この方式の代表例としてNapster[1]、WinMX[13]が挙げられる。

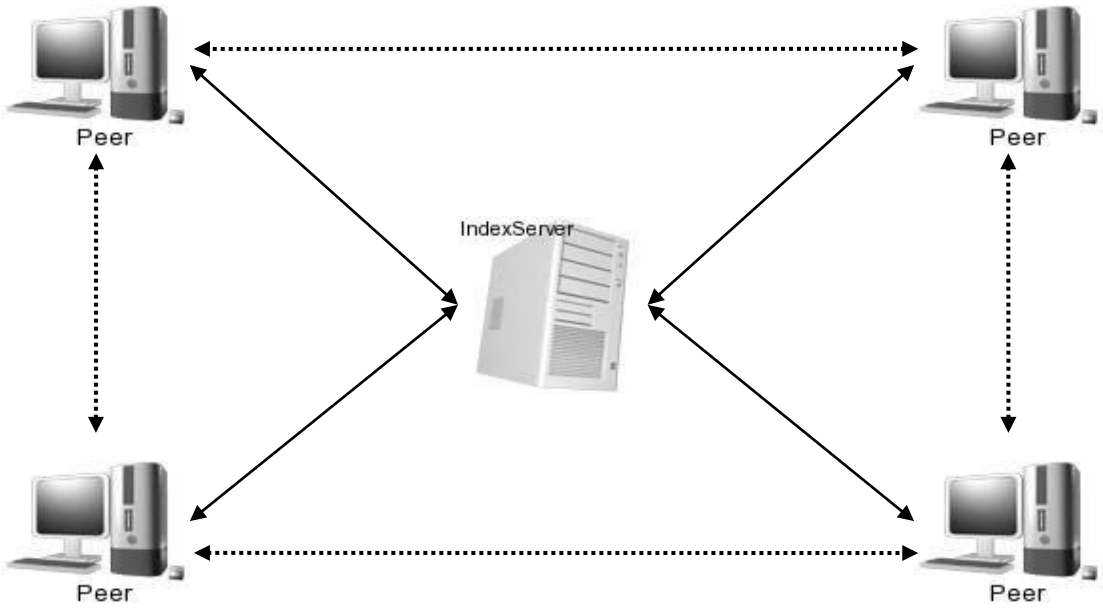

図2.2　HybridＰ２Ｐ

・Pure（ピュア）Ｐ２Ｐ型（図 2.3）

PureＰ２Ｐ型は、HybridＰ２Ｐ型のようなインデックスサーバが存在せず、参加するピア同士がインデックスサーバの役割をもち、情報の交換もピア同士で行う方式である。隣接するピア間でバケツリレー的に検索要求を転送し、ヒットしたピアと直接通信を行う。

この方式では、集中管理するサーバがないため、耐故障性に優れており、匿名性の保証などのメリットがある。一方でサーバが存在しないことにより、アカウントの管理やユーザの認証といったセキュリティ管理が難しいというデメリットがある。また、ピアの増加によるネットワークの輻輳や遅延が起こりやすい。この方式の代表例として Gnutella[2]、Winny[3]が挙げられる。

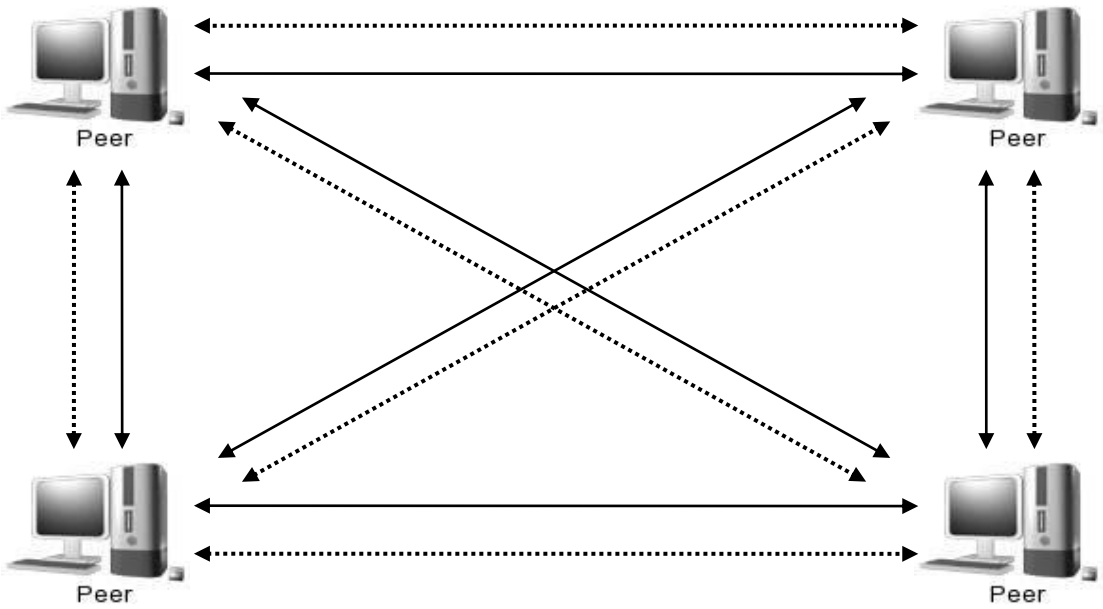

図 2.3　PureＰ２Ｐ型

上記で述べたモデルは、サーバやネットワークの輻輳などによる問題点があり、スケーラビリティも高くはない。そこで、近年研究されている検索技術として「ＤＨＴ（Distributed Hash Table：分散ハッシュテーブル）」がある。次節でこの分散ハッシュテーブルについて説明する。

2.2　分散ハッシュテーブル（ＤＨＴ）

2.2.1　概要

ＤＨＴは、Ｐ２Ｐにおいてサーバを必要としないデータ検索システムを構築するのに用いられる。基本的な仕組みは、コンテンツとノードの両方に同じハッシュ関数のハッシュ値を割り当て、コンテンツのハッシュ値を受け持つピアへ分散配置する。検索する際には、検索したいコンテンツのハッシュ値を計算し、そのハッシュ値を受け持つノードに順々に問い合わせることで検索を行う。その後、ピア間で直接通信を行う。この DHT は、前節で述べた HybridＰ２Ｐ型や PureＰ２Ｐ型よりもスケーラビリティが高く、全てのノードに対して検索でき、負荷の分散などの点で優れている。

ＤＨＴを利用した検索アルゴリズムとして CAN[6]、Chord[7]、Pastry[8]、Tapestry[9]、Kademlia[10]などがある。本研究では、他の検索アルゴリズムに比べて比較的単純かつ、耐故障性に優れている。

2.2.2　Chord

Chord は、Stoica らが提案したＤＨＴ検索アルゴリズムの一つである。ノードの参照やネットワークへの参加・脱退にサーバを必要とせず、各ノードが完全に分散して処理を行う。Gnutella 等の検索クエリをフラッティングするネットワークにおいては、検索可能ノード数Ｎの増加に伴い線形のパケットの増加が起こるが、Chord ではパケット数の増加はＯ(log N)であり、規模拡張性に優れている。

各キーID とノード ID は SHA-1[15]というハッシュ関数を用いて作成され、同じ空間にマッピングされる(キーID＝hash(key)、ノード ID＝hash(IP address))。160bit の ID 空間の場合、１０の４８乗もの ID が存在することになり、コンテンツと IP アドレスをマッピングしてもほぼ衝突しないことが前提とされる。ここでは説明しやすいように 6bit(0～63)の ID 空間を用いて説明を行う。

Chord ネットワークにおいて、ID を持った各ノードは仮想的なリングを形成し、時計方向に接続を行っている。図 2.4 では N1→N8→N15→N22→N31→N36→N43→N47→N52→N56→N1 のように接続されている。この接続はルータ同士の接続ではなく、ノード同士で作られる接続である。そのため、ノード間には複数のルータが存在し、ネットワークの近さをまったく考慮しない仮想的なネットワークである。

各キーの値とデータのペアは、時計回りに最も近いノードに格納され、このノードを代表ノードと呼ぶ。Key10 は時計回りに最も近いノード 15 に格納され、Key24、Key30 はノード 31、Key36 はノード 36 に、Key54 はノード 56 に格納される。

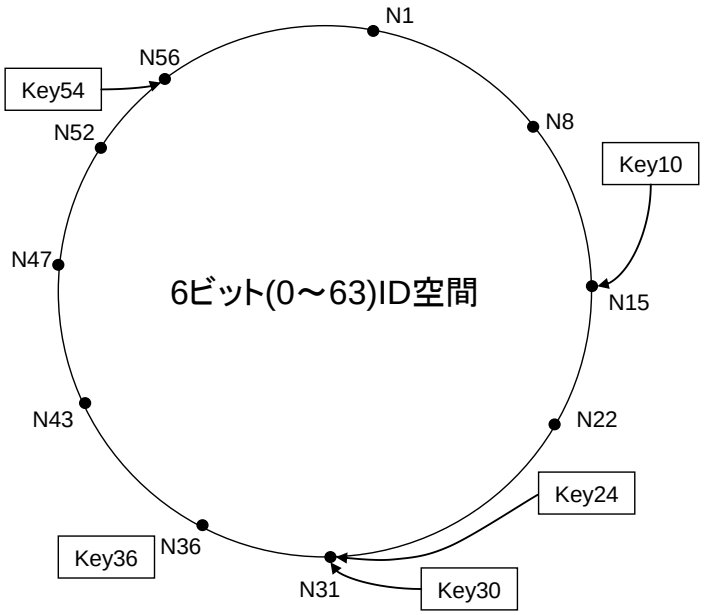

図 2.4　キーID のマッピング

図 2.5 で基本的な参照方法を示す。

検索者であるノード８が Key54 に対応するデータを取得したい場合、N8→N15→N22→N31→N36→N43→N47→N52→N56→N8 へと検索クエリが転送される。ノード 56 は Key54 を保持しているため、Key54 に対応するデータをノード８に送信する。

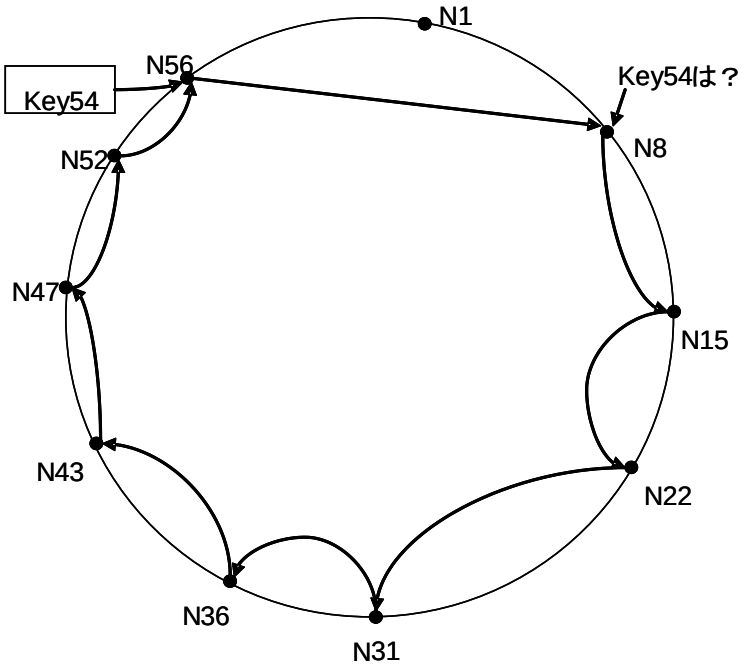

図 2.5　基本的な参照方法

基本的な参照方法では、N ノードで構成されるネットワークの検索に O(N)のホップ数がかかることになる。そこで Chord では性能を改善するために、フィンガーテーブルを用いた高速な参照方法（図 2.6）を定義している。フィンガーテーブルとは、6bit のハッシュ空間を用いて 64 の ID 空間が存在する場合、自ノードの ID より 2^k（0<=k＝6）先のノードの IP アドレスを常に保持しているテーブルであり、このテーブルを用いて検索キーをショートカットしクエリの転送を行う。よって logN のルーティングテーブルを保持することになり、O(logN)のホップ数で高速に検索を行うことができる。

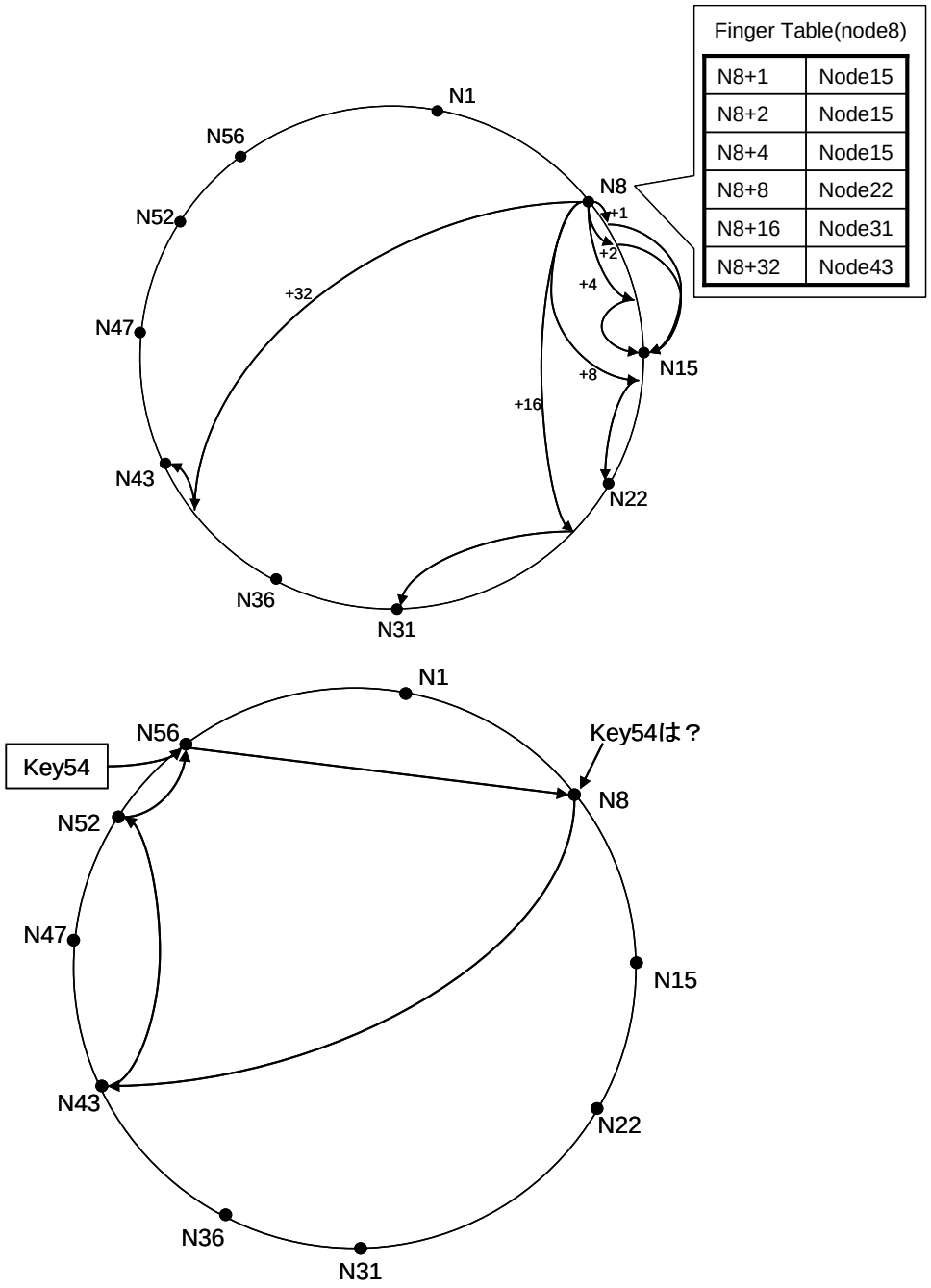

図 2.6　高速な参照方法(key54 を検索)

Chord におけるクエリの転送方法における特徴から、人気のある情報へのクエリが特定のノードへのクエリの集中を招き、同時にそのノードの直前のノードにそのクエリの転送要求が集中することが分かる。従って、クエリの集中するノードの負荷を分散することで、応答時間を改善できる可能性がある。しかも、その負荷を分散すべきノードは直前のノードであることがわかる（図 2.7)。

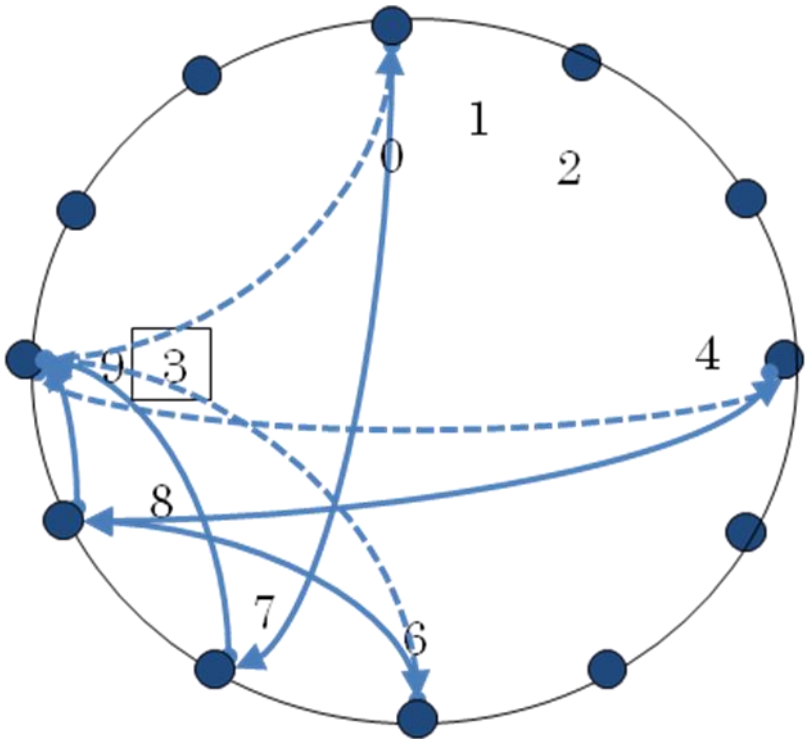

図 2.7　ノード 9 へのクエリの集中と、直前ノード 8 への転送要求の集中

2.3　CAN

CAN では分散ハッシュテーブルの空間として一般に N 次元トーラスを使うが、多くの場合 (1,0)×(0,1)の 2 次元正方形を分散ハッシュテーブルの空間とする。

はじめに、ネットワークに新たに参加するノードは乱数(rand_x,rand_y)を発生する。ただし、０＜rand_x＜１，０＜rand_y＜１である。この乱数は、分散ハッシュテーブルの空間のノード ID となる（Node_ID=(Node_ID_x,Node_ID_y)=(rand_x,rand_y)）。

今、ハッシュ空間のエリアが下図のようになっているとする。A,B,C は各ノードの Node_ID を示している。最初に A がハッシュ空間全体を管理していたとする。次に B は A から空間を半分譲り受けその管理ノードとして参加する。次に、C はハッシュ空間を B から半分譲り受け、その管理ノードとして参加する。各領域に対応する ID の情報を各ノードは管理する。検索要求は、検索する ID に近づく方向に、隣接する管理ノードに対して転送

される。CAN では構造化された P2P ネットワークを構築するために、各ノードが以下の情報を管理する。

・ノード情報（自身の IP と管理領域のセット）

・ルーティングテーブル（隣接ノードのノード情報）

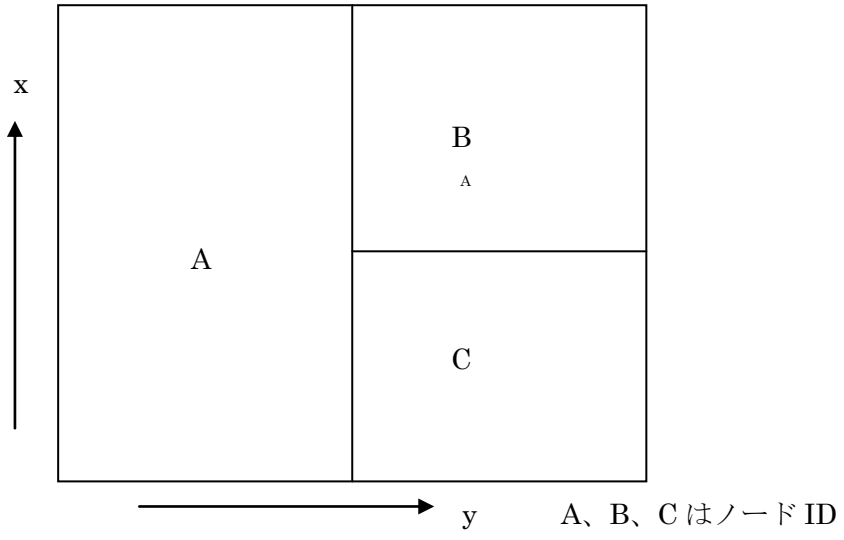

図 2.8　CAN のハッシュ空間

2.3.1　CAN ルーティング方式

CAN では各ノードが管理するルーティングテーブルの整合性を保つために、各ノードが自身のノード情報とルーティングテーブルを含む更新要求メッセージを隣接ノードへブロードキャストする。ブロードキャストは自身の領域が変化した場合、もしくは一定周期で行われる。そして更新要求を受信したノードは受信した情報をもとにルーティングテーブルの更新を行う。ノードの参加・離脱により隣接ノードが常に変化している状態において、常に全てノードのルーティングテーブルの整合性を保つことはできないが、一定周期で送信される更新要求により、各ノードのルーティングテーブルの整合性を高めることができる。

図 2.9 におけるノード D を例に挙げるとルーティングテーブルとして隣接ノード B、E のノード情報を管理している。他ノードの情報はルーティングテーブルに格納しない。ノード D から、ノード A まで検索するには、まず D のルーティングテーブルにおいてノード A を探し、存在しない場合は A に近づくための経由ノードを検索する。

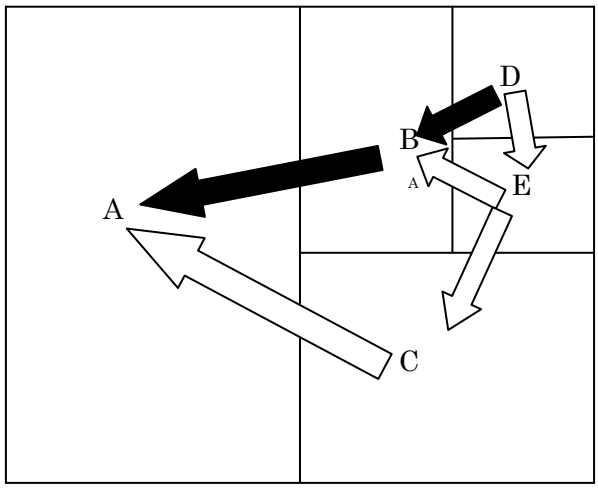

図 2.9　CAN のルーティング

2.4　分散環境におけるハッシュ機能

キャッシュは、CPU のバスやネットワークなど様々な情報伝達経路において、ある領域から他の領域へ情報を転送する際、その転送遅延を極力隠蔽化させ転送効率を向上させるために考案された記憶階層の実現手段である。実装するシステムに応じてハードウェア・ソフトウェア双方の形態がある。

キャッシュは転送元と転送先の中間に位置し、データ内容の一部とその参照を保持する。データ転送元への転送要求があり、それへの参照が既にキャッシュに格納されていた場合は、元データからの転送は行わずキャッシュが転送を代行する（この状態をキャッシュヒット、キャッシュに所望のデータが存在せず元データから転送する状態をキャッシュミスヒットという)。もしくは出力データをある程度滞留させ、データ粒度を高める機能を持つ。これらによりデータの 2 種の局所性、すなわち時間的局所性と空間的局所性を活用し、データ転送の冗長性やオーバヘッドを低減させることで転送効率を向上させる。

時間的局所性とは、データの再利用率とその時間的特性を示す。ある領域のデータ転送が行われて、同一データの転送が再度、近い時間内に行われている場合を時間的局所性があるという。

空間的局所性とは、データの格納位置に対する偏在性を意味する。ある領域のデータ転送が行われて、近い時間内に、連続ないし近傍領域のデータ転送が行われている場合を空間的局所性があるという。真にランダムに転送されるべきデータというのは少なく、大抵のデータには空間的局所性が存在する。

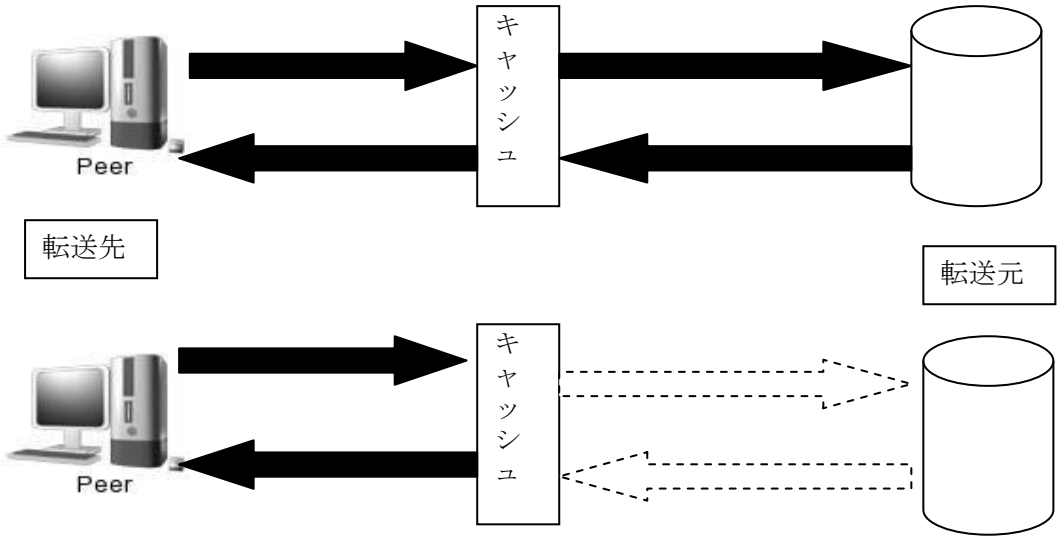

図 2.10　キャッシュの役割

2.5 Chord に対するキャッシュの組み込み方式[7]

Chord にキャッシュを導入することで検索負荷を分散し、検索時間を短縮する方式が提案されている。Chord のリングで隣接するノードに、インデックス情報をキャッシュ（複製）する方法が提案されている。特に直前ノードインデックス情報をキャッシュしておくことで、オリジナルのノードにアクセスするクエリと、直前ノードにアクセスするクエリとに負荷を分散することができる。また、直前ノードが検索結果を返すことから、検索の転送回数（ホップ数）が削減されて、応答時間が短くなる利点もある。

ただし、インデックス情報は事前にアクセス頻度が分からないため、キャッシュはすべてのノードについて行われる。従って、アクセスセス頻度が低くキャッシュが有効でない場合でもキャッシュが作られる問題がある。

2.6 CAN に対するキャッシュの組み込み方式

CAN にキャッシュを導入することで検索負荷を分散し、検索時間を短縮する方式が提案されている。特に P2P ネットワーク上でのキャッシュの配置方法が考察されている。

2.6.1　CC 方式

CC 方式では、各ノードが一度検索したインデックスをローカルの記憶領域上にキャッシュし、同一の検索を行う場合にはキャッシュを利用する。またキャッシュを他のノードと共有することで、キャッシュの利用性を高めている。例えば、他のノードからの検索要求を転送する前に、自身のキャッシュ内に対応するインデックスを保持しているかを確認し、保持している場合は検索要求を転送せずにキャッシュを利用して応答する。

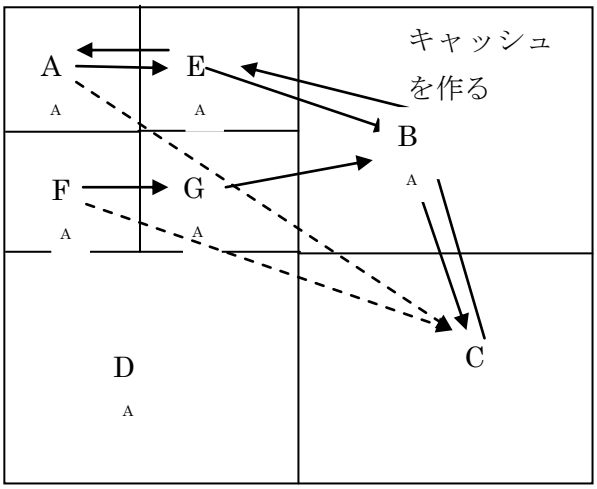

図 2.11　CAN におけるキャッシュの例

図 2.11 では、A、B、C、D、E、F、G はノード ID である。ノード A からノード C の領域の ID を検索すると、A→E→B→C という経路で検索要求が転送されたとする。すると、ノード B、E には A の検索結果がキャッシュされる。次に、ノード F がノード C の領域にある同じ ID を検索すると、F→G→B と検索要求が転送されるが、ノード B にあるキャッシュにヒットすると、ノード B から検索結果が返される。

2.6.2　CCPR 方式

CCPR 方式は CC 方式を拡張した方法である。CCPR ではキャッシュの効果を高めるために、中継ノードと呼ぶノードを設定し、中継ノードにもそのインデックスをキャッシュする。検索時には、中継ノードを経由する経路を採用することで、他のノードが検索した同じ要求に対するキャッシュを利用することができる。

中継ノードは、検索要求が多数経由する経路に配置され、検索要求を効率的に処理することができる。しかし、検索要求が少なくなったときには、中継ノードはオーバーヘッドになるために、削除される。これらの制御を行うために、各ノードの負荷の情報を定期的に収集し、負荷に応じて中継ノードを設置したり削除するための管理コストが必要となる。

第３章 提案方式

3.1 目的

情報検索においては、人気のあるコンテンツに対してクエリ（問い合わせ）が集中する問題がある。このような課題に対して、予めキャッシュ（または複製）を配置して、負荷を分散する方式が提案されてきた。しかし、予めどのコンテンツに人気がでるのかは分からないため、すべてのコンテンツに対してキャッシュが作成されることで無駄なキャッシュが作られることがあった。また、人気が出ることで負荷が高くなったノードにキャッシュを配置する方式が提案されているが、ノードの負荷情報を定期的に収集する手間や、人気がなくなった時点でキャッシュを削除するなどの管理上のコストが大きくなる問題があった。それに対して、本研究ではクエリの到着頻度に応じて、キャッシュの作成確率が高くなることで、人気があるノードにはキャッシュが作成され、人気のないコンテンツにはキャッシュが作成されにくい方式にした。また、キャッシュに有効期限を設けることで、アクセスがなくなったキャッシュは自動的に消去される方式にすることで、管理の手間が少ない方式を提案する。

3.2 概要

本研究では、Chord ネットワークを採用する。システムに参加するノードは、N ビットの一意なＩＤを保有することとする。Chord のリング上を検索要求が転送され、目的ノードにおいて検索対象が見つかった場合、検索元に検索結果が返送されるとともに、検索要求が目的ノードに到達する直前のノードに一定の確率（キャッシュ確率）でキャッシュが作成される。作成されたキャッシュには有効期間が与えられ、その期限を越えるとキャッシュは自動的に削除される。キャッシュは、オリジナルのノードから１回だけコピーされる単一モードと、キャッシュからさらにキャッシュが作成される反復モードとがあり、それぞれ評価を行う。

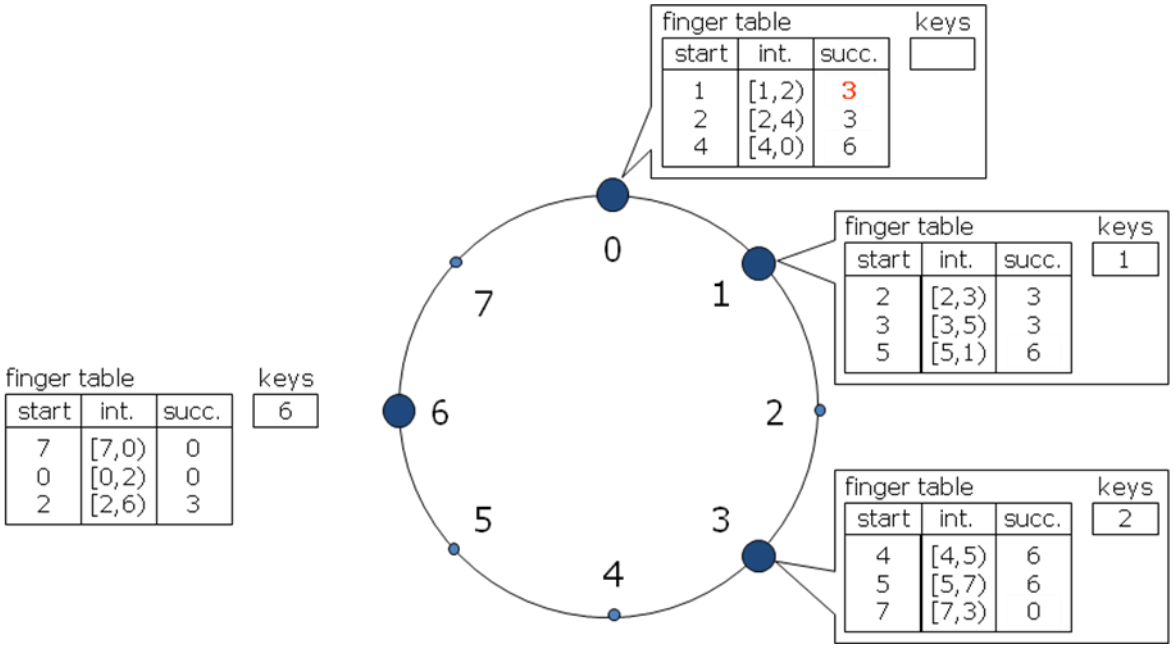

図 3.1　Chord の例

システム全体のノード数は最大で 128、256、512、1024 個とした。1 回検索にホープ数の最大は 20 ことを構成することになる。なお、今回は検証を簡単にするために、ノードの故障については考慮しないこととした。（故障率 0 %とした）

3.3　キャッシュ

3.3.1　キャッシュの構成

キャッシュすべき情報は、コンテンツに対する ID と値（key-value ペア）とキャッシュの有効期間である。今回の実験では、値自体は重要ではないので、ID と有効期間のみを保持することとした。プログラムコードにおけるキャッシュの定義を以下に示す。

Coding 定義

```
class Cache{
        long    cache_id;// cache id
        double time_limit;// time limit

 }
```

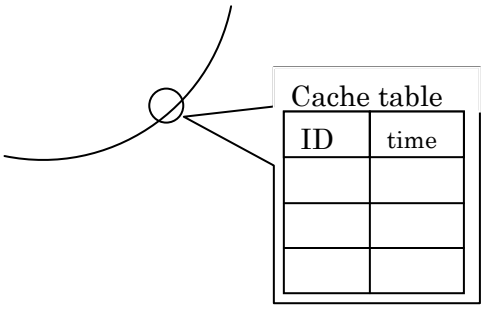

図 3.2 ノードにキャッシュを示す

以下はキャッシュの作成に関するコードである。このコードが動作するノードは目標 ID が見つかったノードの直前のノードである。

```
static void e_cache(Event e){

		Cache c = new Cache();

		NodeInstance node1;
		for(int i = 0;i<node.size();i++){
			node1 = node.elementAt(i);
			if(node1.Node_ID ==e.node_id) //イベントが生じたノード
			{
				c.cache_id =e.query_id;//目標ID
				c.time_limit = time + CACHELIMIT;//有効期限
				node1.add_cache(c);//キャッシュのノードへのセット
			}

		}
```

つまり、すべてのノードにはキャッシュ表があるが、キャッシュ表には最初は何もない。直後のノードから依頼があると、キャッシュをセットする。

ノードにキャッシュを追加する定義

```
public void add_cache(Cache c){

		this.Cache.addElement(c);//ベクタクラスを利用する

	}
```

ノードにおいて、検索要求が転送されてくると、管理領域をチェックするとともに、保持するキャッシュを検索する。もし、キャッシュに検索対象のIDが見つかれば、検索要求を転送せずに検索元に応答を直接返す。

ノードのキャッシュを検索するコード

```
public boolean search_cache(long target){
        Cache c,c1;
        int i;
        long nodeid;
        for(int i=0;i<this.Cache.size();i++){
                c = this.Cache.elementAt(i);
                nodeid = c.cache_id;
                if(nodeid == target){
                return true;
                }
        }
        return false;
}
```

3.3.2　キャッシュの作成確率とキャッシュの削除

キャッシュは、検索が成功したノードの直前のノードに作成されるが、常に作成されるわけではない。なぜなら、利用頻度の低いキャッシュを作っても無駄になるからである。本研究では、キャッシュをある確率（キャッシュ確率）で作成することとした。この結果、検索頻度が高いノードでは１回１回のキャッシュ確率が小さくても、全体としてキャッシュが作成される確率が高くなる。一方、検索頻度が低いノードではキャッシュが作成されにくくなる。

また、キャッシュが一度作成されると、それをどのように削除するかが難しいが、我々は有効期限方式をとった。これは、有効期限をキャッシュに設定しておき、有効期限が切れた場合キャッシュを自動的に削除してしまう方式であり、集中的な管理や負荷状態の通知などの手間が不要である。

ノードにキャッシュを削除

```
public boolean del_cache(double time){
        Cache c1;
        for(int i=0;i<this.Cache.size();i++){
                c1= this.Cache.elementAt(i);
```

```
                if(c1.time_limit<=time)
                {
                        this.Cache.remove(i);
                        return true;

                }

        }
```

3.3.3　反復キャッシュ

キャッシュは、検索が成功したノードの１ホップ前のノードに作成される。つまり、本方式は Chord の検索ルート上にキャッシュを配置することになる。この考えを拡張して、キャッシュにヒットしたノードの１ホップ前のノードに更にキャッシュを配置することを考える。これを本研究では反復キャッシュ（キャッシュのキャッシュ）と呼ぶ。反復キャッシュは、キャッシュの増大につながるが、キャッシュ確率と有効期限の調整によって適切に管理すれば、検索対象と検索元との間に適切にキャッシュが配置されることになる。

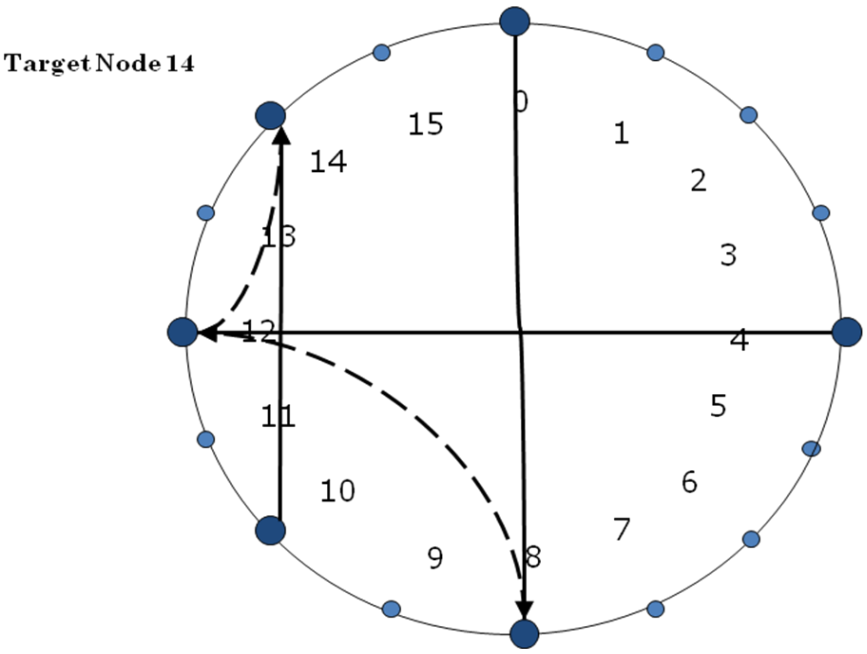

図 3.3　キャッシュのキャッシュ

図(1)のように、ノード１４はノード１４の前ノード１２にキャッシュを作る。更にノード１２は、前ノード８にノード１２のキャッシュのキャッシュを作る。そして、ノード８

のキャッシュ表にはノード１４からノード１２に作られたキャッシュの情報がキャッシュされる。

キャッシュのキャッシュを実現するコード

```
if(CACHE_OF_CACHE){
//---------> cache of cache
        double  percent1;
        percent1 = r2.nextDouble();
        if(percent1<=CACHERATE)
        {
                Cache_total++;
                Event event2 = new Event();
                event2.query_id = e.query_id;//検索ID
                event2.event_type = 8;//キャッシュ作成要求
                event2.node_id = e.sender;//直前ホップノード
                event2.sender = e.node_id;
                schedule(event2,DELAY);
        }
}
```

第 4 章　シミュレーション

4.1　シミュレーション方法

（1）ノード数を変更したときの評価を行う。以下にシミュレーション条件を示す。

・ノードは、シミュレーション開始時に初期化されランダムに配置されるものとする
・最大ホップ数を 20 回とし、これを越えたクエリは破棄とする
・クエリ数１０００００回
・参加各ノードのノード数：
128、256、512、1024 個
・キャッシュ確率を 1 にする
・タイムリミットは１０００００
・キャッシュのキャッシュは利用しない

（2）タイムリミットを変化させた場合の評価を行う。以下にシミュレーション条件を示す。

・ノードの配置は、シミュレーション開始時に初期化されランダムに配置されるものとする
・最大ホップ数を 20 回とし、これを越えたクエリは破棄とする
・クエリ数１０００００回
・参加各ノードのノード数：
128、256、512、1024 個
・キャッシュ確率を 1 にする
・タイムリミットは１０００００、１０００，１００，１０，０
・キャッシュのキャッシュを利用する

(3) キャッシュの確率を変化させた場合の比較を行った。以下にシミュレーション条件を示す。

・最大ホップ数を 20 回とし、これを越えたクエリは破棄とする
・クエリ数１０００００回
・参加各ノードのノード数：
128、256、512、1024 個
・キャッシュ確率は以下の値をとる
0 ,0.1, 0.5, 1

(4) キャッシュのキャッシュ（反復キャッシュ）を利用した場合の比較を行った。
以下にシミュレーション条件を示す。

・最大ホップ数を 20 回とし、これを越えたクエリは破棄とする
・クエリ数１０００００回
・参加各ノードのノード数：
128、256、512、1024 個
・キャッシュの確率
0 ,0.1, 0.5, 1
・タイムリミット
0,10000,5000,1000,100,10
・キャッシュのキャッシュを利用する場合としない場合を比較

(5)クエリの発生条件

クエリの発生条件は、負荷の高い（人気のあるコンテンツ情報を持つ）ノードと低い（人気のないコンテンツ情報を持つ）ノードが存在するモデルとした。このようなモデルとして一般的に利用されている、Zipf の法則に基づいて、１００個のコンテンツ情報があると仮定して各コンテンツごとに検索要求発生確率を設定して、クエリを発生させた。

数学定義：

$$f(k; s, N) = \frac{1/k^s}{\sum_{n=1}^{N} 1/n^s}$$

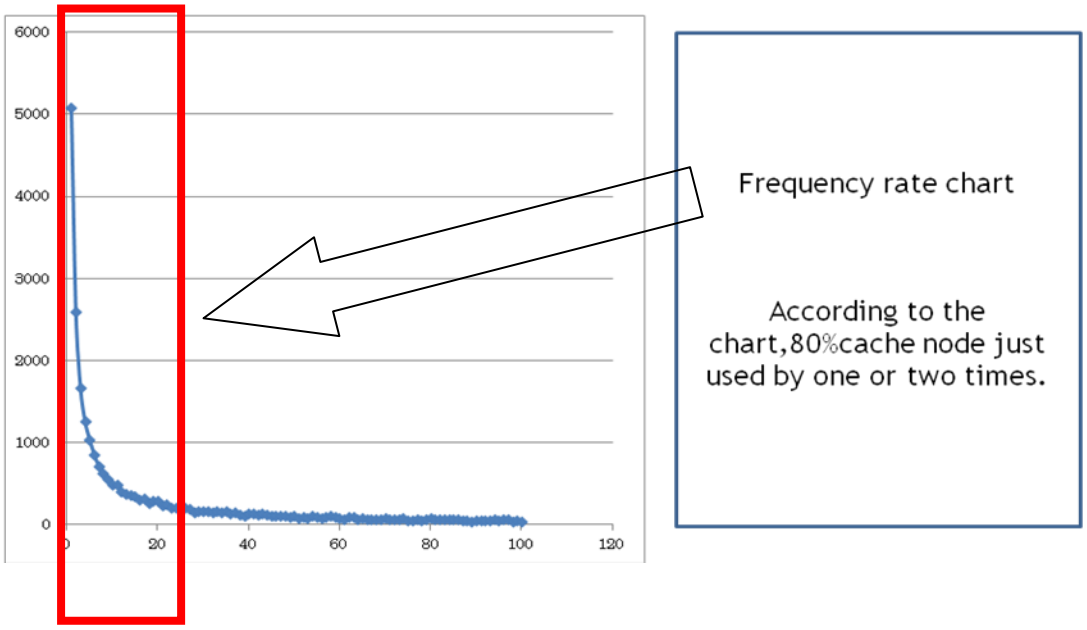

図 4.1　zipf 分布

コード例

```
public static void zipfnode()
      {
            int i,j;
            double total = 0;
            ////////////////////////////////////////////////////
            for(i= 0;i<100;i++){
                  nodeprob[i]= (double)(1)/(i+1);//(1/n)
                  total =(double)(1)/(i+1)+total;// sum of (1/n)
            }
            for(i=0;i<100;i++){
                  nodeprob[i]= nodeprob[i]/total; // (1/n)/sum of (1/n)
            }
            for(i=1;i<100;i++){
                  nodeprob[i]=nodeprob[i-1]+nodeprob[i];//
                  if(i==99)
                  nodeprob[i]=1.0;
                  System.out.println("nodeprob["+i+"]="+ nodeprob[i]);
            }
      }
```

4.2　シミュレーション結果

4.2.1 ノード数の変化

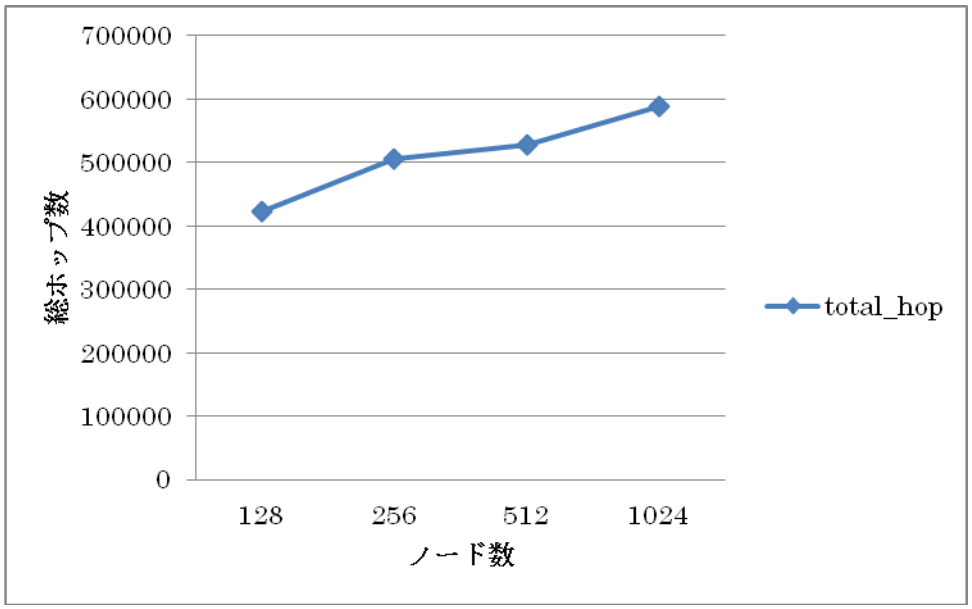

図 4.2 ノード数を変化させたときの総ホップ数

図 4.2 において、横軸はノード数、縦軸はクエリの総ホップ数である。従って、chord のノード数が増えれば総ホップ数が増えていることがわかる。

4.2.2 タイムリミットの変化

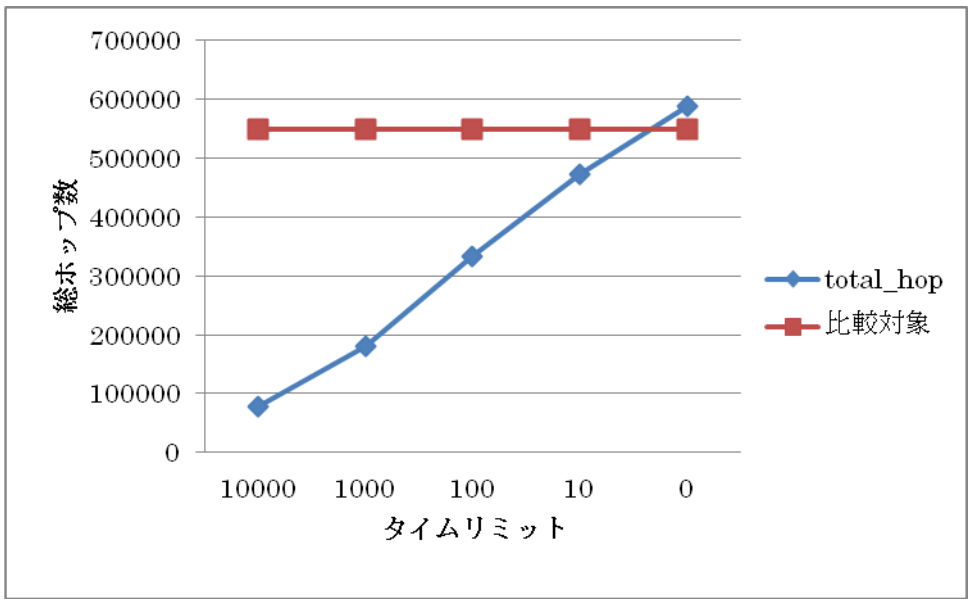

図 4.3 ノード数 1024 の時のタイムリミットに対する総ホップ数

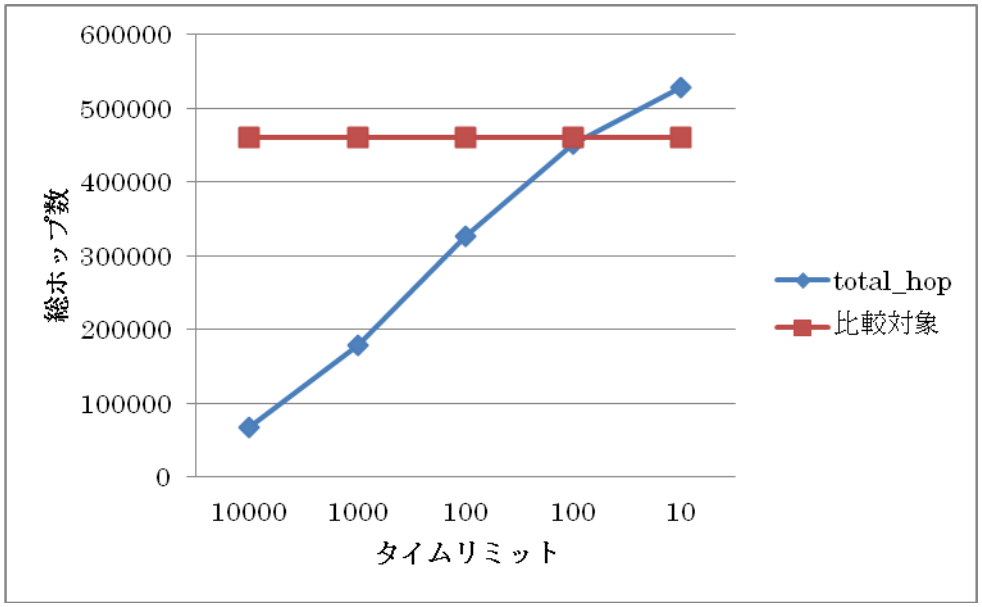

図 4.4 ノード数 512 の時のタイムリミットに対する総ホップ数

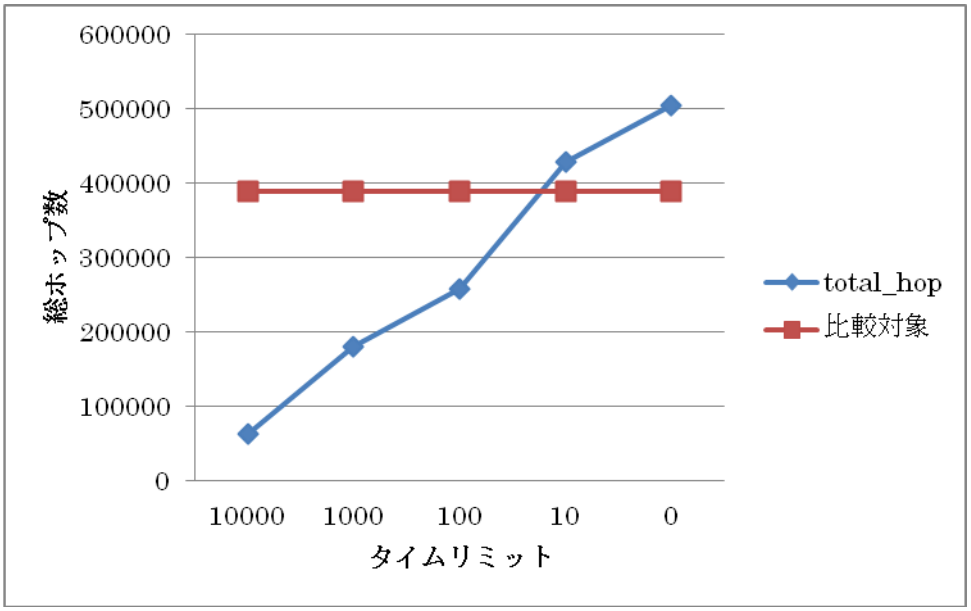

図 4. 5 ノード数 256 の時のタイムリミットに対する総ホップ数

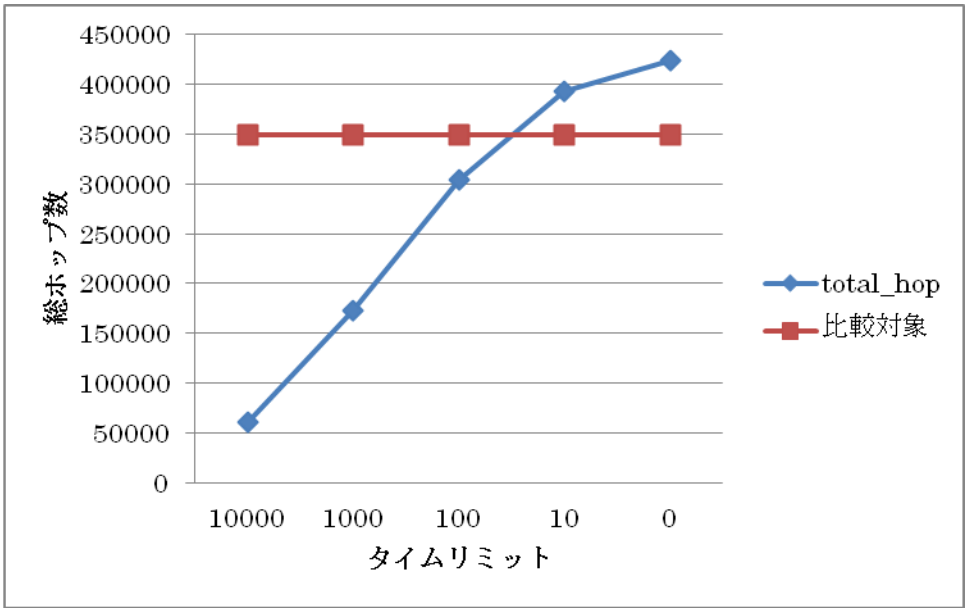

図 4. 6 ノード数 128 の時のタイムリミットに対する総ホップ数

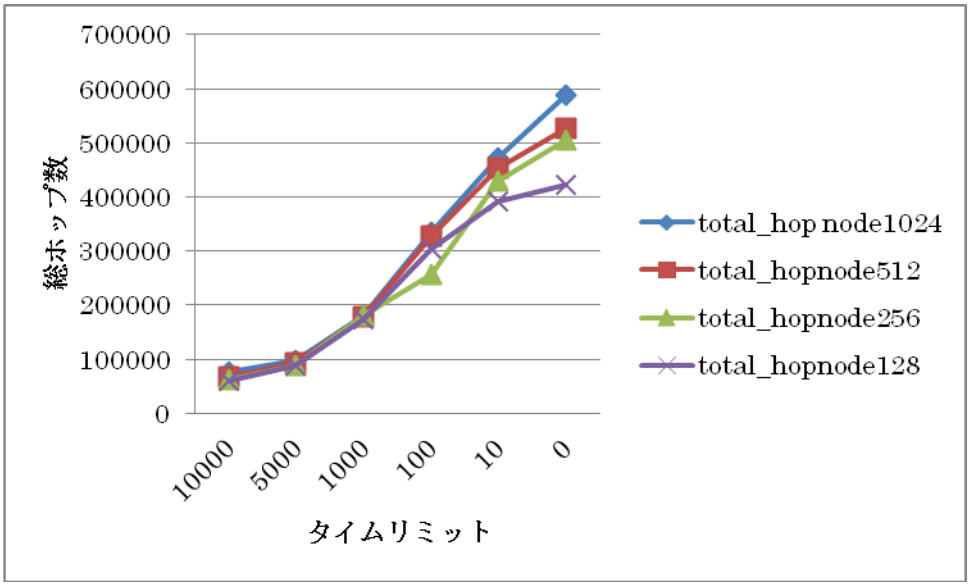

図 4. 7 タイムリミットに対する総ホップ数(まとめ)

図 4.7 からわかるように、タイムリミットを増やすことで総ホップ数が削減できることがわかった。これは、キャッシュの保持時間が長くなることで、キャッシュにヒットする確率が高くなるためである。キャッシュの保持時間が長くなるため、キャッシュのコストはタイムリミットを増やすほど大きくなる。

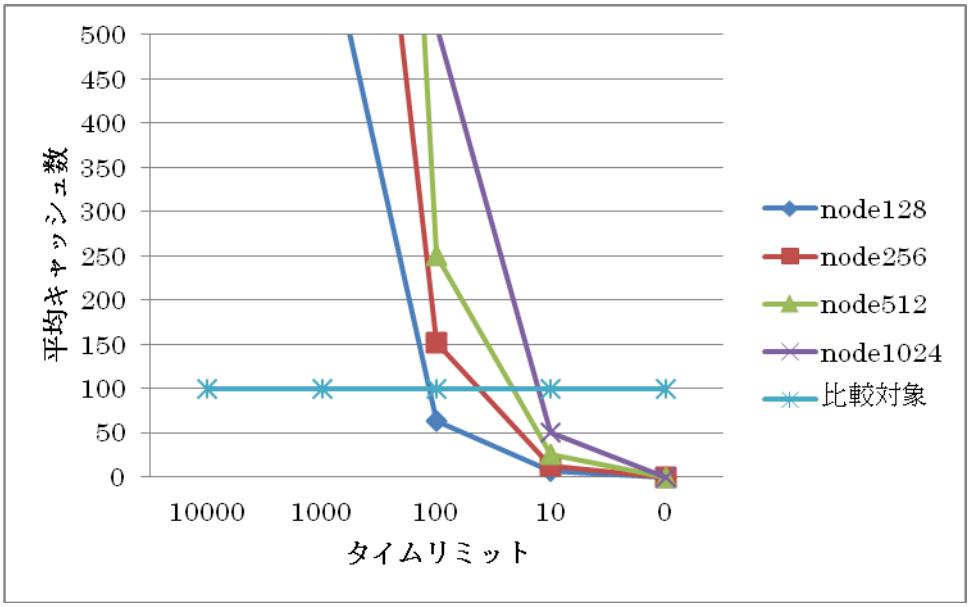

図 4.8 タイムリミットに対する保持されるキャッシュ数

図 4.8 から、タイムリミット増やすと、保持されるキャッシュが多くなることが分かる。ここで、図 4.3 から図 4.6 及び図 4.8 に比較対象として、Chord のリングの隣のノードに静的にキャッシュを作成した場合のシミュレーション結果を表示した。この結果から、比較対象の方式よりもキャッシュ数が少なくても、総ホップを削減できていることが分かる。たとえば、図 4.6 において、１２８ノードでタイムリミット 100 のときに、比較対象の結果が約３５万ホップかかっているのに対して、提案方式は約３０万ホップとなっている。このとき、図 4.8 から１２８ノードの平均キャッシュ数は、比較対象が１００なのに対して、およそ６０程度となっており、キャッシュのコストも少なくて済んでいることがわかる。

4.2.3 キャッシュ確率

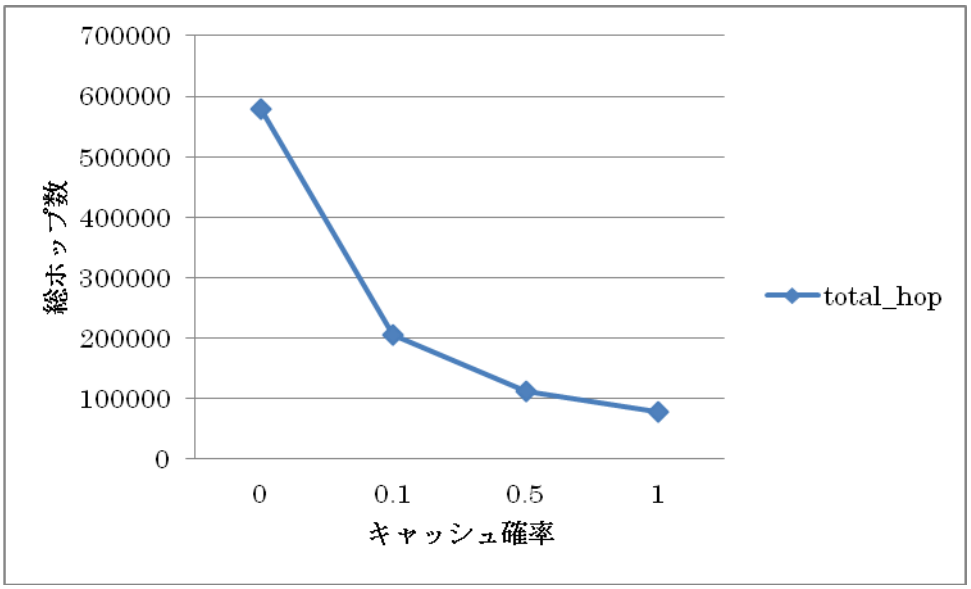

図 4.9 ノード数１０２４のときのキャッシュ確率に対する総ホップ数

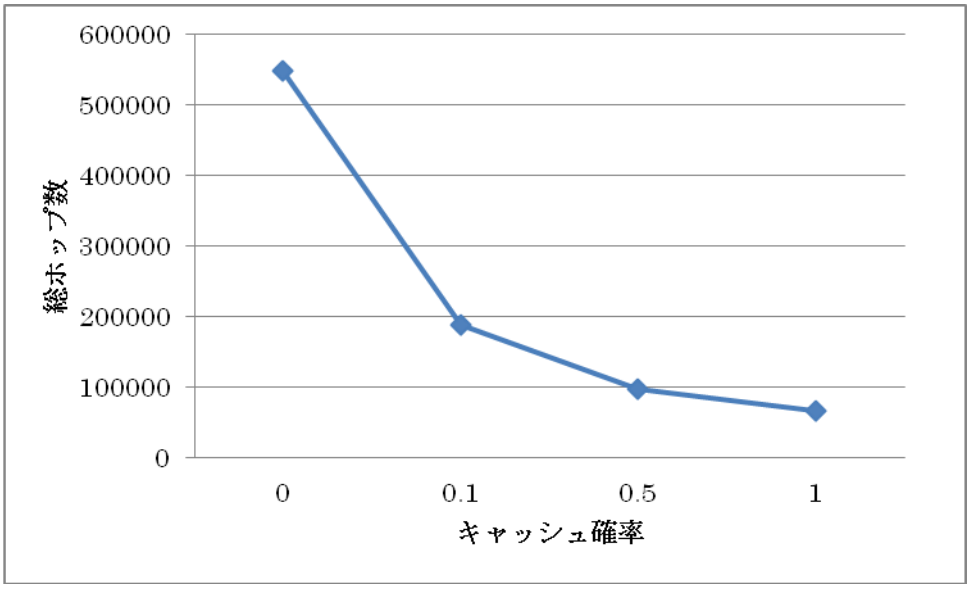

図 4.10 ノード数５１２時のときのキャッシュ確率に対する総ホップ数

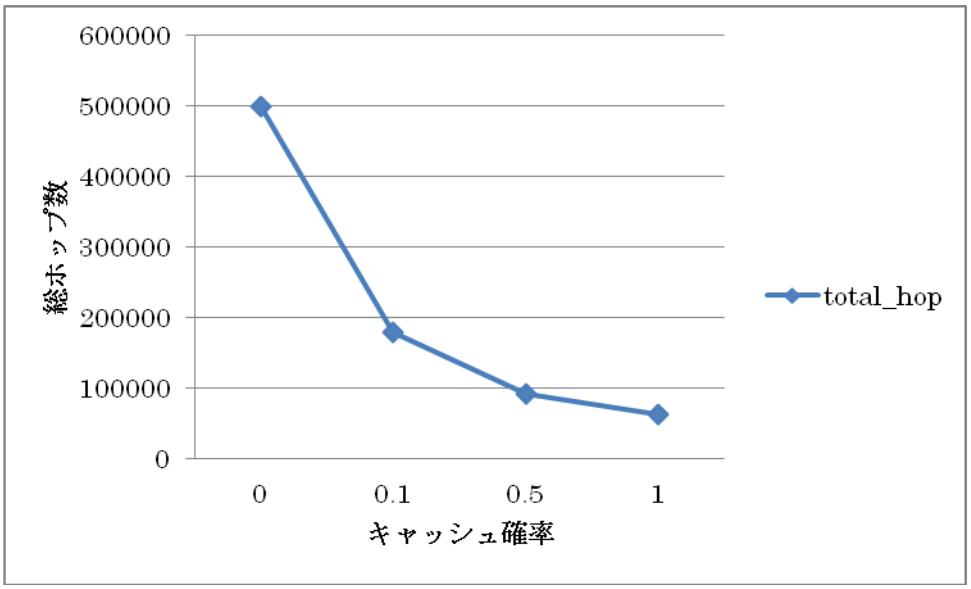

図 4.11 ノード数２５６のときのキャッシュ確率に対する総ホップ数

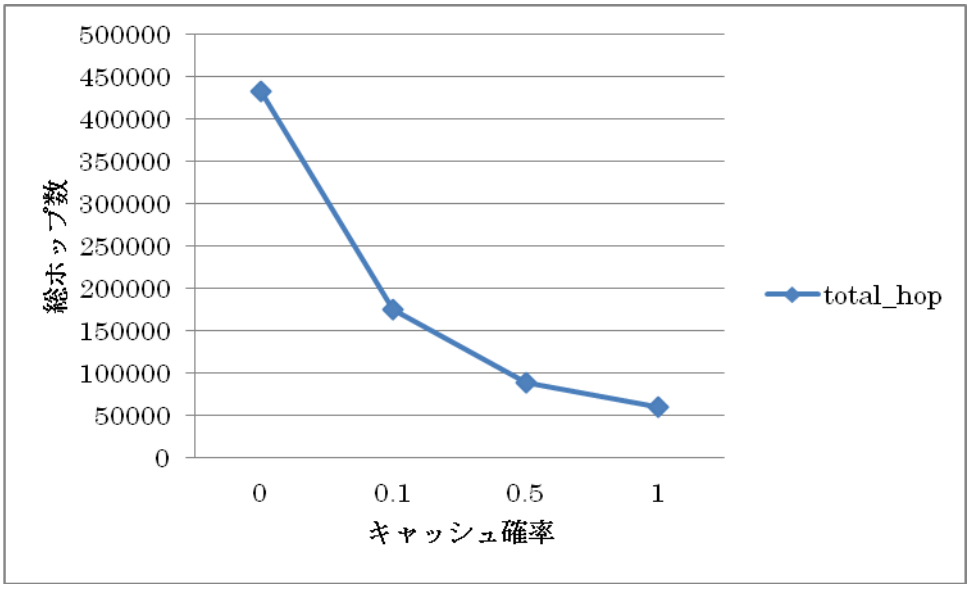

図 4.12 ノード数１２８のときのキャッシュ確率に対する総ホップ数

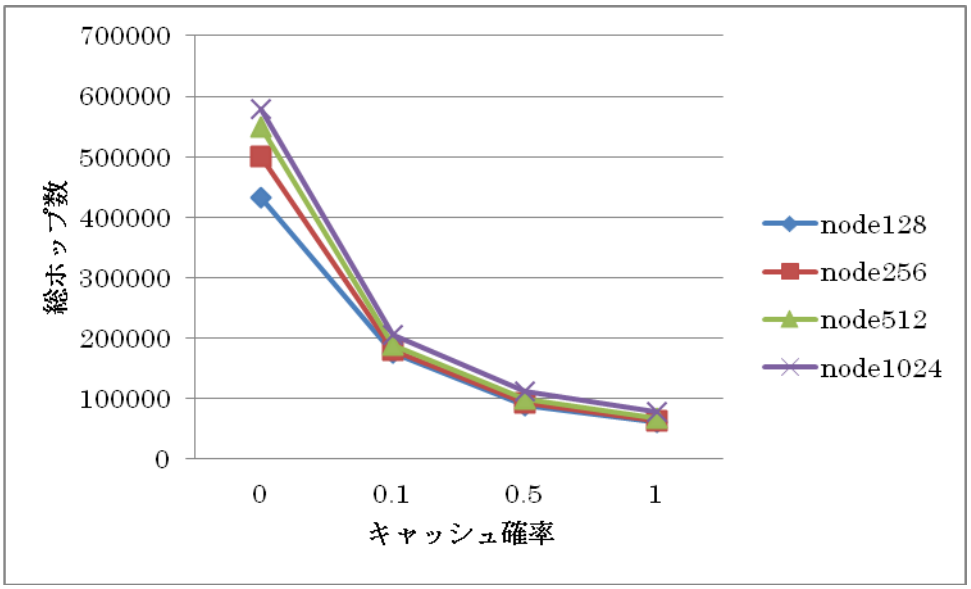

図 4.13 キャッシュ確率に対する総ホップ数のまとめ

図 4.12 から、キャッシュ確率が高くなると、キャッシュの作成される割合が高まるため、キャッシュへのヒット率が増えることで総ホップ数が削減される。しかし、キャッシュが大きくなると、その分キャッシュを保持するためのコストがかかる。

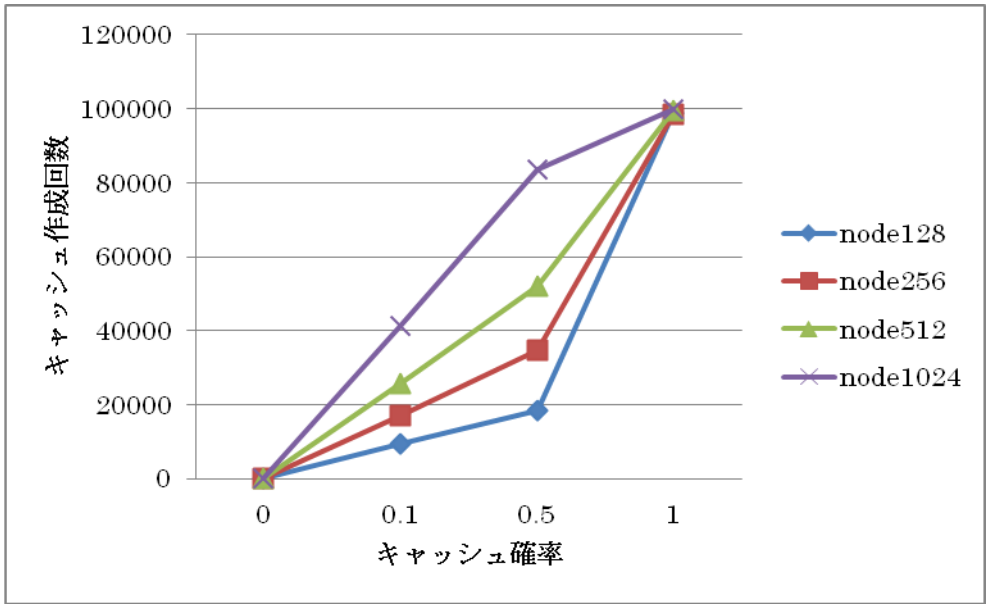

図 4.14 キャッシュ確率とキャッシュ作成回数の関係

図 4.14 において、キャッシュ確率に応じたキャッシュ作成回数を評価した。キャッシュは確率が高くなるにつれて作成回数は増えていくが、ノード数が小さいとキャッシュのヒット率が高まるため、キャッシュが作成されないことから、作成回数が少なくなる傾向がある。一方、ノードが大きいと様々な経路でオリジナルのノードにクエリが到着し、キャッシュがヒットしない割合が増えてくることから、キャッシュの作成回数が増加している。

4.2.4 反復キャッシュの結果

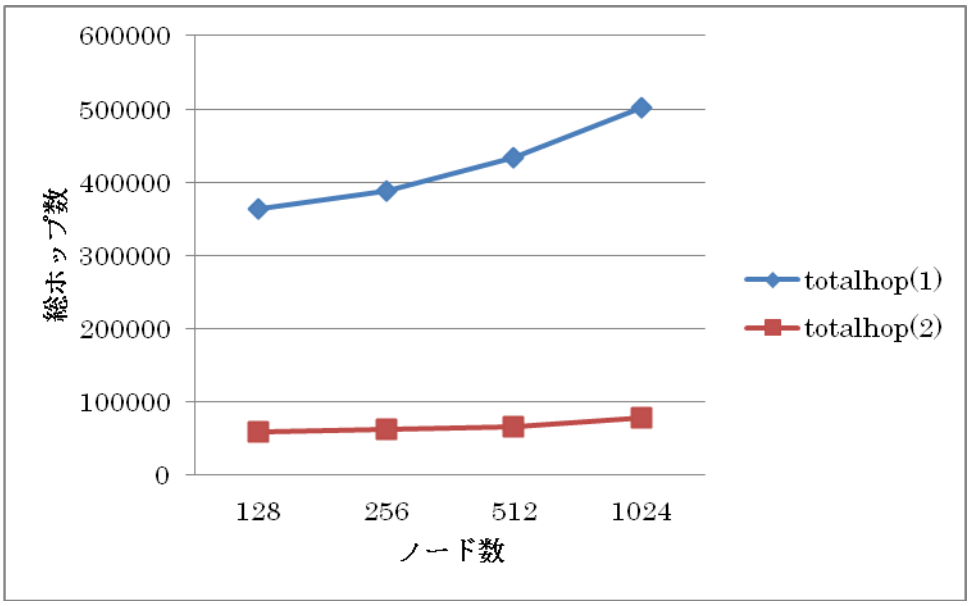

図 4.15　反復キャッシュを利用する場合と利用しない場合の総ホップ数比較
Totalhop(1) 反復キャッシュは使わない
Totalhop(2) 反復キャッシュを使う

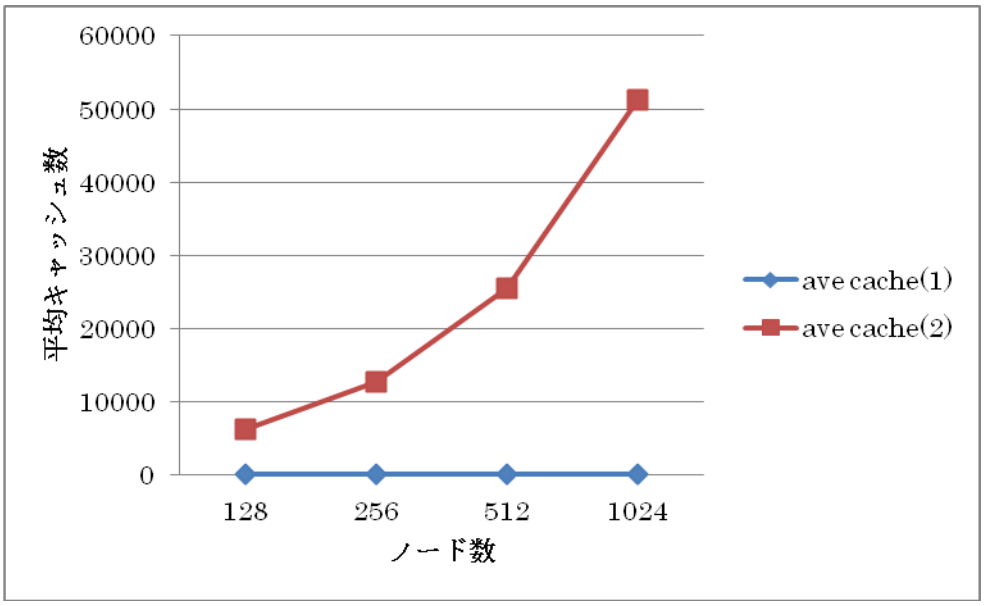

図 4.16　反復キャッシュを利用する場合と利用しない場合の平均キャッシュ数比較

Ave cache(1) 反復キャッシュは使わない

Ave cache(2) 反復キャッシュを使う

反復キャッシュを使うことで、図 4.15 に示すように総ホップ数は大きく削減することが可能である。一方で、図 4.16 に示したように、平均キャッシュ数は非常に大きくなる。従って、キャッシュ数に制限があるような環境では、反復キャッシュは利用しにくい。そのため、適切なキャッシュの保持時間（タイムリミット）や、キャッシュ確率を設定することで、平均キャッシュ数を少なく制限して利用することが必要となる。

第５章　結論

本研究では、Ｐ２Ｐネットワークシステムにおける検索負荷が人気のあるコンテンツに対して集中する問題に対し、適切なキャッシュを動的に配置することで性能を改善する方式を提案した。Ｐ２Ｐネットワークとしては、分散ハッシュテーブル（Distributed Hash Table：ＤＨＴ）の一つであるＣｈｏｒｄアルゴリズムを対象とした。分散ハッシュテーブルにキャッシュを導入する従来方式としては、ＣＡＮに対してキャッシュノードを設置することで、負荷を分散する方式が提案されているが、検索経路の負荷の状況を収集するコストやキャッシュノードを起動したり停止する手間がかかる問題があった。また、Ｃｈｏｒｄでは、各隣接ノードで直後のノードが保持する情報をキャッシュするアルゴリズムが提案されているが、負荷の状況を考慮していないため、負荷の少ないノードにもキャッシュが配置され、無駄なコストがかかる問題があった。本研究では、検索要求が転送されて検索が成功したノードの直前のホップのノードにキャッシュを配置した。また、配置方法は確率的に行うことで、人気のある情報を管理しているノードに沢山の検索要求が集まることを利用して、その場合にのみキャッシュが作成される方式とした。その結果、負荷の高いノードにのみキャッシュが作成され、負荷の少ないノードにはほとんどキャッシュが作成されないようになる。また、キャッシュは一定時間経過すると、自動的に削除されるようにした。そのため、キャッシュを削除する手間が必要なく、管理コストが軽減される。提案方式をシミュレーションによって評価した。検索要求の発生確率は、人気のある情報を管理するノードに沢山の検索要求が送られ、人気のない情報を管理するノードには少ない検索要求が送られるように設計した。これには、Ｚｉｐｆの法則と呼ばれる確率分布を用いて作成した。その結果、従来のＣｈｏｒｄにおいて隣接ノードに全情報をキャッシュする方式にくらべて、提案方式はキャッシュ効率が良く、ホップ数も少なくてすむ設定が可能であることを明らかにした。また、キャッシュのキャッシュ（反復キャッシュ）を入れる場合と入れない場合についてもその性能を明らかにした。反復キャッシュを入れると、単一のキャッシュでは得られないホップ数削減効果が得られる。ただし、キャッシュの量は非常に多くなる。キャッシュ量について、制約が少ない状況であれば、反復キャッシュは非常に有効な方式であることを明らかにした。

謝辞

本研究の全過程において懇切丁寧なご指導を頂きました東邦大学の佐藤 文明教授に厚く御礼申し上げます。

参考文献

[1]”Napster.” http://www.napster.com/

[2]”Gnutella.” http://www.gnutelliums.com/

[3]金子 勇，アスキー出版社，“Winny の技術”，2005

[4]”Skype.” http://web.skype.com/intl/ja/

[5]首藤一幸，田中良夫，関口智嗣:"オーバレイ構築ツールキット Overlay Weaver",情報処理学会論文誌: コンピューティングシステム, Vol.47, No.SIG12 (ACS 15), pp.358-367, 2006.

[6]Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, “A Scalable Content-Addressable Network”, In Proceedings of ACM SIGCOMM, 2001.

[7]Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, “Chord:A Scalable Peer-to-peer Lookup Service for Internet Applications”, In the Proceedings of ACM SIGCOMM, 2001.

[8]Antony Rowstron; Peter Druschel: Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems, Lecture Notes in Computer Science **2218/2001**. Springer Berlin, p. 329, (2001).

[9]Ben Y. Zhao, John D. Kubiatowicz, Anthony D. Joseph:“Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing”. *Technical Report: CSD-01-1141*. Berkeley, CA, USA: University of California at Berkeley, (2001).

[10] Petar Maymounkov and David Mazières, Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In 1st International Workshop on Peer-to-peer Systems (IPTPS'02), 2002.