博 士 論 文

インターネット環境に適した
構造化 P2P ネットワークソフトウェアの設計と実装

指導教官　篠田 陽一 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

高野 祐輝

2011 年 1 月 7 日

要 旨

P2P ネットワークはサービスに参加しているノード同士が，自律的に相互通信を行いリソースを共有することでサービスの実現を行う分散型のサービスモデルである．P2P ネットワークは大きく分けて，アドレス構造を持つ構造化 P2P ネットワークとアドレス構造を持たない非構造化 P2P ネットワークの二種類存在する．構造化 P2P ネットワークはアドレス構造に基づくデータ検索を行えるため，非構造化 P2P ネットワークと比較して，規模が大きくなっても効率よく検索が行える．

構造化 P2P ネットワークは様々なアルゴリズムが提案されているが，本研究では，設計が比較的シンプルである Kademlia を対象として，構造化 P2P ネットワークのルーティングテーブル検索の効率化問題，NAT 問題，大規模ノード下の Churn 問題に取り組んだ．

Kademlia では，木構造によるルーティングテーブルの管理を行うが，本研究では効率化のため，配列による管理方法を提案した．その結果，100,000 ノードの時，配列の場合では木構造と比較して，30 倍以上効率よくなることが明らかとなった．

構造化 P2P ネットワークは，任意のノード同士が自由に通信を行えるとの仮定に基づいて設計されている．しかし，実際には，インターネットには NAT が存在し，NAT を考慮しなければ任意のノード同士で通信を行うことは出来ない．そこで本研究では，NAT を考慮した構造化 P2P ネットワークを実現する手法である DTUN を提案した．

P2P ネットワークの設計を行った際にはネットワークの状態が頻繁に変わる Churn 状態でも正しく動作することが求められる．そこで本研究では，libcage と呼ぶ構造化 P2P ネットワークライブラリを作成し，Churn 下で，NAT が介在する場合のパフォーマンス計測を行った．その結果，10,000 ノードでも問題なく動作することを確認できた．

現実的には P2P ネットワークは，インターネット環境が持つ制限などにより，その能力を十分に発揮できるとは言いがたい．そこで，本研究では，インターネット環境で，P2P ネットワークの持つポテンシャルを十分に発揮出来るようにすることを目標とした．本研究の成果を用いれば，現在のインターネット環境でも，十分に P2P ネットワークの利点を活かすことが出来るようになる．

目次

第 1 章

序論

Peer-to-Peer ネットワーク（P2P ネットワーク）は，クライアントサーバ型と対比されるサービスモデルである．クライアントサーバ型では，サービスを提供するサーバとサービスを利用するクライアントに明確に分かれていた．一方，P2P ネットワークは，参加しているノードがサービスを利用するとともに，サービスの提供も同時に行うという特徴を持っている．そのため，P2P ネットワークを用いると可用性が高く規模拡張性に優れたサービスを実現可能な技術である．

P2P ネットワークは，大きくわけて 2 種類存在する，1 つ目は，アドレス構造を持たない非構造化 P2P ネットワークである．非構造化 P2P ネットワークを用いている代表的なソフトウェアには，Gnutella [7] や Freenet [25] が存在する．これらはアドレス構造を持たないため，フラッディングやランダムウォークを基にして検索クエリを転送していた．そのため，参加しているノード数が多くなるとクエリ転送の効率が悪くなってしまった．一方，構造化 P2P ネットワークは，アドレス構造を持ち，検索クエリはアドレスに基づいて効率よく転送される．そのため，規模が大きくなっても効率的な検索が行える．本研究では，この構造化 P2P ネットワークに焦点を当てる．

1.1 構造化 P2P ネットワークの利点

構造化 P2P ネットワークの最も大きな利点は，その規模拡張性と効率の良さである．非構造化 P2P ネットワークでは，フラッディングやランダムウォークを基にして検索を行なっている．フラッディングを用いた場合は，ノード数が多くなるとトラフィックが増大してしまい，ランダムウォークを用いた場合では，ノード数が多くなると．検索に非常に大きな遅延が発生してしまうと言う問題が発生してしまう．

一方，構造化 P2P ネットワークはアドレス構造を持ち，アドレスに基づいてトポロジが構成される．そのため，検索クエリはアドレスに基づいて転送されるため，ノード数が多くなっても検索の効率が著しく悪くなるということは無い．構造化 P2P ネットワークの代表的な提案として，Chord [53] や Pastry [49]，Kademlia [37] などがあるが，これらはいずれも，参加しているノード数を N とした場合，$O(\log N)$ のホップ数でデータが検索可能となる[*1]．

1.2 構造化 P2P ネットワークの種類と本研究の対象

構造化 P2P ネットワークは様々な種類が提案されているが，これらは，さらに大きく分けて 3 つの種類に分類される．

1 つ目は，EpiChord [35] や Kelips [30]，OneHop [29] などに代表される，構造化 P2P ネットワークである．これらは O(1)-Hop と呼ばれる種類の構造化 P2P ネットワークであり，検索に必要なホップ数が $O(1)$ で終了するという特徴を持つ．その代わり，各々のノードは基本的にフルメッシュで接続されており，新規ノードの到着やノード離脱の際に発生する経路情報の更新には，マルチキャストやブロードキャストが用いられる．そのため，参加しているノード数が多くなると，経路情報の更新に必要な負荷が著しく増大してしまう [44]．

[*1] 当然，$O(\log N)$ 以外のホップ数となる提案も存在する．例えば，d 次元トーラスを利用する CAN では，$O(dN^{\frac{1}{d}})$ のホップ数となる．

2 つ目は，Koorde [31] や Ulysses [34]，Cycloid [51] などに代表される種類の，一定次数の構造化 P2P ネットワークである．O(1)-Hop の構造化 P2P ネットワークは各ノードがリンクをたくさん持つ代わりに，検索に必要なホップ数を $O(1)$ となるうにしていた．一方，一定次数の構造化 P2P ネットワークでは，各ノードが保持すべきリンクの数をコストと考え，各ノードは一定のリンク数しか持たないようにした方式である．しかし，一定のリンク数しか持たないにもかかわらず，検索に必要なホップ数は $O(\log N)$ で済むという特徴がある．ところが，保持するリンクの数が少ないということは，現実的には，保持している全てのリンクがダウンしてしまう可能性が高くなり，参加しているノードの平均生存時間が短いと，何度も参加を繰り返さなければならず効率が悪くなってしまう．

3 つ目は，先に述べた Chord や Pastry，Kademlia などの，構造化 P2P ネットワークである．普通，構造化 P2P ネットワークと言うと，これらを指すが，本論文では前述した 2 種類と区別するために，これらを一般的な構造化 P2P ネットワークと呼ぶ．一定次数の構造化 P2P ネットワークと違い，一般的な構造化 P2P ネットワークでは，ノード数 N に応じて $O(\log N)$ だけのリンクを持つ．さらに，O(1)-Hop の構造化 P2P ネットワークと違い，データ検索のクエリは多段ホップで転送される．

本研究の対象は 3 つ目の一般的な構造化 P2P ネットワークである．O(1)-Hop や一定次数の構造化 P2P ネットワークにも利点はあるものの，ノード数や生存時間などに大きな条件が課せられており，インターネットのインフラとして利用するには適しているとは言えない．そこで，本研究では，規模拡張性と可用性に優れている一般的な構造化 P2P ネットワークを対象とする．

一般的な構造化 P2P ネットワークにも, 様々な種類があるが, 本研究では特に Kademlia に対象を絞る．Kademlia は基本的に 4 つのメッセージのみからなる，非常にシンプルな設計となっている．そのため，実装が容易であるという大きな利点がある．

1.3 本研究の目的と成果

本研究では，構造化 P2P ネットワークの設計するにあたり特に重要な点である，ルーティングテーブルの検索効率と，NAT 問題，大規模ノード下の Churn 状態について考慮しなければならない．そこで，本研究では，これらの問題について取り組んだ．

1.3.1 構造化 P2P ネットワークの規模拡張性とルーティングテーブルの検索

構造化 P2P ネットワークを設計する上で重要な点は，規模が大きくなっても効率的に検索が行えるかどうかという点である．構造化 P2P ネットワークは，非構造化 P2P ネットワークと比べて，規模が大きくなっても効率的にデータ検索を行えるが，規模が大きくなった場合の効率については，その設計に大きく依存する．特に，構造化 P2P ネットワークでは，各ノードがルーティングテーブルを持ち，検索を行う際にそのルーティングテーブルを何度も引く必要があるため，ルーティングテーブルの検索コストは，ネットワークの規模が大きくなった場合に大きな問題となる．

Kademlia では，ルーティングテーブルを木構造で管理している．そのため，参加しているノード数を N としたとき，データの検索時に $O(\log N)$ 回，木の検索が必要であった．そこで本研究では，ルーティングテーブル検索の効率化を目的として木構造の代わりに配列を用いてルーティングテーブルを管理する方法を提案する．配列を用いた場合，ルーティングテーブルの検索は $O(1)$ 回のテーブルルックアップで終了する．ルーティングテーブルの検索とデータ構造については 4 章にて解説する．

1.3.2 NAT 問題の考慮

P2P ネットワークは参加しているノードがお互いに通信を行い，協調的に動作するモデルである．すなわち，参加している任意のノード同士で，必ず通信が行えるという事が大きな前提とされている．しかしながら，実際のインターネットを考えた場合，任意の

ノード同士で通信を行うことは容易ではない．

任意のノード同士で通信を行うことが容易ではない最大の理由が，Network Address Translation（NAT） [52] の存在である．NAT は，IPv4 [40] のアドレス枯渇問題に対応するために考案された．しかしながら，NAT の仕様は不明瞭な部分が多かったため，挙動が様々に異なる NAT 製品が登場してしまった [54]．そのため，結果的にインターネット上に存在する，任意のノード同士で通信することは容易ではなくなってしまった．

そこで本研究では，NAT の介在するインターネットでも構造化 P2P ネットワークを利用可能とするために，分散環境で NAT 越えの通信を可能な手法である Distributed Traversal UDP through NATs（DTUN）DTUN を提案する．DTUN は Kademlia をベースとした構造化 P2P ネットワークである．通常の Kademlia と最も異なる点は，全てのノードが参加するネットワークに加えて，グローバルアドレスを持つノードのみから成る DTUN ネットワークを設けた点である．DTUN ネットワークは，外部観測的な NAT 判別機構，UDP Hole Punching [27] の為の機構，UDP Hole Punching 不可な NAT 下に居るノードための中継機構を提供する．これらを利用すると，NAT が介在する場合でも，構造化 P2P ネットワークを実現することが出来る．

1.3.3 Churn 下での大規模動作検証

構造化 P2P ネットワークはアドレス構造に基づいたネットワークトポロジを構成し，そのトポロジに従って検索クエリの配送を行う．従って，そのトポロジが正しく構成されていないと，検索クエリが正しく配送されず検索に失敗してしまう．P2P ネットワークはサービスを利用するノード同士が集まって構成されるネットワークである．そのため，各々のノードはいつでも自由に出入りしてしまい，ネットワークがある一定の定常状態に落ち着く事はない．この攪拌された状態のことは Churn と呼ばれるが [43]，Churn 下でも正しくデータを取得できるようすることは，構造化 P2P ネットワークを設計する上で非常に重要である．

Churn 対策の主な手法として，ルーティングアルゴリズム部分での対策と，DHT など

サービス部分での対策を行う方法がある．本研究では，DTUN の実証ライブラリとして libcage の実装を行ったが，libcage ではルーティング部分と DHT 部分の両方で Churn 対策を行った．

さらに，構造化 P2P ネットワークは概念的には，規模拡張性があるが，どの程度の規模まで耐えられるかと言うことは，その設計に大きく依存する．従って，構造化 P2P ネットワークを設計した際は，大規模な環境下で実験を行い，その設計が正しいかどうかを確認しなければならない．

本研究では，NAT が介在する状況下でも，Churn 下の大規模な環境で DTUN が正しく動作し，構造化 P2P ネットワークとして正しく働くかを実際に構造化 P2P ネットワークライブラリの libcage を作成し検証を行った．検証方法は，分散ハッシュテーブルの値取得に必要な時間と，値取得の成功確率を求めた．

実験は PC100 台を利用して，イベント多重により最大 10,000 ノード規模で，Churn 状態の NAT 介在下で行った．また，Churn は各々ノードの生存時間を 500[s] と設定し，NAT 有り（DTUN 利用）の場合は NAT 下にあるノードの数を全体の 70[%] として設定した．その結果，10,000 ノードの時は，NAT 無しの場合は 95[%] が約 6[s] 以内での応答があり，NAT 有りの場合のほうは 95[%] が約 9[s] 以内の応答がありと，やや値取得に必要な待ち時間が多くなったが，NAT 有りでも劇的に待ち時間が長くなるということはなかった．

値取得の成功確率は，Kademlia における DHT の値取得操作で find node の同時問い合わせ数 α と，DHT のデータ複製数 r によって変化する．NAT 無しの場合では $\alpha = 3$, $r = 10$ の時に成功確率が 99[%] となったが，NAT 有りの場合では $\alpha = 6$, $r = 10$ としたときに，成功確率が 99.4[%] となった．DTUN を利用した場合でも，値取得の成功確率はパラメータを変化させることで，DTUN 無しの場合と同程度まで向上させることが可能となった．

第 2 章

Peer-to-Peer ネットワーク

Peer-to-Peer ネットワーク（P2P ネットワーク）とはクライアントサーバと対比される概念である．クライアントサーバ型のシステムでは，サービスを行うサーバとサービスを利用するクライアントは明確に区別されていた．一方，P2P ネットワーク型のシステムではサービスの提供者と利用者の区別はなく，ネットワークに参加しているノードがサービの提供と利用を行う．本章では，P2P ネットワークに関する歴史と特徴について記述する．

2.1 P2P ネットワークの定義

2.1.1 P2P ネットワークの類似技術

P2P ネットワークはアプリケーション層でネットワークを構築して，サービスを実現するモデルである．また，P2P ネットワークは中央集権的なサーバを持たず，参加しているノードはお互いに通信を行い，自律協調的に動作する．すなわち，P2P ネットワークはアプリケーション層で動作する自律分散協調的なシステムであるとみなすことが出来る．

アプリケーション層で動作する，自律分散協調的なシステムは，P2P ネットワーク以外にも存在する．例えば，ニュースサービスの USENET や，IRC，SMTP サービスはサーバ同士で互いに通信を行い，自律分散協調的に動作するシステムである．しかし，これら

は，クライアントサーバ型のシステムであり P2P ネットワーク型のシステムであるとは言えない．

2.1.2 P2P ネットワークとリソースの共有

P2P ネットワークであるための最も重要な条件の一つに，参加しているノード同士がリソースを共有し合うということがある [20, 50]．なお，ここで言うリソースとは，ネットワーク帯域やハードディスク容量，CPU リソースなどを指す．この条件によると，USENET や IRC，SMTP サービスは自律分散的なシステムであるが，P2P ネットワークであるとは言えない．なぜなら，これらは，利用者とサービス提供者は明確に区別されており，利用者のノードはリソース共有を行わないからである．

リソース共有を行うことの最も大きな利点は，コストに関してである [20]．クライアントサーバ型のシステムでは，サービスの規模に応じて，サービスの運用者がリソースを増やしていかなければならなかった．一方，P2P ネットワーク型のシステムでは，基本的にサーバは存在せず，必要なリソースはお互いに共有して利用する．そのため，大規模なサービスであっても，一部のみにコスト負担を強いると言ったことはない．

2.1.3 リソース共有の対称性

リソース共有は，P2P ネットワークであるための最も重要な条件であった．そのリソース共有の方法に関しても条件が課されている [50, 48]，それは，ノード同士がお互いに通信を行え，リソースがお互いに利用できるという事である．P2P ネットワークではリソースの共有が重要であるが，それが片方からのみ利用可能な非対称なものであるべきではないということである．

2.1.4 自己組織化と非中央集権的な制御

P2P ネットワークの大きな特徴として，自律分散協調的なシステムであることが言える．これは，USENET や IRC などと同じ特徴であるが，Roussopoulos らは，P2P ネッ

トワークであるための条件として，これらについても定義している [48]．彼らの定義によると，P2P ネットワークのノードは，グローバルなノード情報やリソースは存在しない状況下で，ノードは各自が得た状況により判断してネットワークを構築する，自己組織化という特徴と，ノードは中央サーバによって制御されず，自律的に自身の振る舞いを制御する，非中央集権的な制御という特徴を持つとしている．

2.2 P2P ネットワークの利点

P2P ネットワークの利点としては，規模拡張性，可用性，アドホック性等が挙げられる．

2.2.1 規模拡張性

クライアントサーバ型では，サーバ側に大きな負担を強いる必要があり，規模の大きさに比例したリソースをサーバ側で容易する必要があり，規模拡張性を維持するためには多大なコストが必要であった．しかしながら，P2P ネットワークはクライアントサーバのように，一部のノードにコスト負担を集中させること無く，参加しているノードでコストを分散させるため，規模拡張性に優れたサービスを最小限の設備投資で実現できる．

Napster [13] は 1999 年に登場した，音楽ファイル共有ソフトウェアである．Napster では，ファイルの検索のみをサーバ側で行い，実際のファイル交換は参加しているノード同士が直接行った．クライアントサーバ型でファイル交換を行うためには，サーバ側でファイル交換用のリソースを容易する必要があるが，Napster では P2P ネットワークのリソース共有を利用して最小限の設備投資でファイル共有を実現した．

2.2.2 可用性

クライアントサーバ型ではサーバが単一障害点となり，サーバの故障やサーバへのネットワーク障害が原因で，サービスを利用できなくなる可能性がある．一方，基本的に P2P ネットワークには単一障害点がなく，一部のノードが故障しても全体としては動作し続けることが可能である．

2.2.3 アドホック性

P2P ネットワークはサーバなどのインフラを必要とせず，各自が自律協調的に動作してシステムを構成する．そのため，恒久的でないシステムや，その場限りのシステムと言った，アドホックなシステムをインフラ整備の必要なく，容易に構築することができる．

2.3 ピュア P2P ネットワークとハイブリッド P2P ネットワーク

P2P ネットワークには大きく分けて，ピュア P2P ネットワークとハイブリッド P2P ネットワークの二種類存在する．ピュア P2P ネットワークとは，サーバのような中央集権的なノードが存在しない P2P ネットワークのことを指す．一方，ハイブリッド P2P ネットワークは，サーバを持ち一部機能をサーバ側で行う．

P2P ネットワークは，様々な利点を持つ反面，管理や把握が難しいといった特徴を持つ．ハイブリッド P2P ネットワークでは，それを補うためにサーバが利用される場合がある．Napster やハイブリッド P2P ネットワークの一つであるが，Napster ではネットワークへのログインなどにサーバが利用された．当然ながら，ハイブリッド P2P ネットワークはサーバが存在するため，そこが単一障害点となってしまう．

一方，ピュア P2P ネットワークの代表的な例としては，Gnutella [7] や Freenet [25] などがある．これらはサーバを持たないため，単一障害点がなく可用性に優れている．

2.4 非構造化 P2P ネットワークと構造化 P2P ネットワーク

ピュア P2P ネットワークを実現する手法としては，大きく分けて非構造化 P2P ネットワークと構造化 P2P ネットワークの二種類存在する．本節ではこれらの違いについて説明する．

2.4.1 非構造化 P2P ネットワーク

非構造化 P2P ネットワークはアドレス構造を持たない P2P ネットワークのことであり，検索クエリの転送には，フラッディングやランダムウォークが用いられる．そのため，非構造化 P2P ネットワークは規模が大きくなると，遅延が大きくなったり，トラフィック消費量が大きくなってしまい，効率が悪くなってしまう．非構造化 P2P ネットワークの代表的な例としては，Gnutella や Freenet が存在する．

2.4.2 構造化 P2P ネットワーク

Consistent Hashing は，1997 年に Karger らによって提案された分散キャッシングの手法である [33]．従来の分散キャッシングでは，ノード数に変化があると，キャッシュの再配置に大きなコストが発生した．しかし，Consistent Hashing では，キャッシュ再配置を局所的なものに留め，規模の変化に強い，効率の良い分散キャッシングを可能にした．

Consistent Hashing を P2P ネットワークで実現するための方法として，Chord [53] や Pastry [49]，CAN [41]，Kademlia [37] などが提案された．これらは，非構造化 P2P ネットワークと区別され，アドレス構造を持つ構造化 P2P ネットワークと呼ばれる．また，P2P ネットワークでの Consistent Hashing は，分散ハッシュテーブル（DHT）と呼ばれる．

非構造化 P2P ネットワークでは，ネットワークの規模が大きくなると，検索の効率が悪くなってしまう．例えば，Puttaswamy らは，DHT を非構造化 P2P ネットワークである Gnutella と Gia [22] へ適用した UDHT [1] を提案している．UDHT では非構造化 P2P ネットワークでの検索には，ランダムウォークの方が効率が良いとしており，検索にはランダムウォークを利用している．そのため，規模が大きくなったとき，検索結果を確実に得るために，ランダムウォークの最大検索深度かデータの複製数を増加させる必要がある．Puttaswamy らは，15,000 ノードまでのシミュレーションを行っているが，15,000 ノードの場合，複製数を 10 とすれば 100[%] 近くの確率でデータを取得できている．しか

し，検索に必要となる平均ホップ数はノード数に対してほぼ線形に推移しており，15,000 ノードの場合，必要な平均ホップ数 100 近くにも達する．

ところが，Chord，Pastry，Kademlia などの構造化 P2P ネットワークでは検索に必要な平均ホップ数は $\log N$ となり，規模が大きくなっても効率的に検索を行うことができる．これは，構造化 P2P ネットワークではアドレス構造にもとづいてトポロジを構築するため，効率的に検索クエリを配送出来るためである．

第 3 章

Kademlia のアルゴリズムとプロトコル

構造化 P2P ネットワークはアドレス構造に基づいてトポロジを形成する P2P ネットワークである．2 章で述べたように，構造化 P2P ネットワークには様々な方式が提案されている．Kademlia [37] は構造化 P2P ネットワークを実現する方式の一つであり，他の構造化 P2P ネットワーク方式と比較して非常にシンプルなアルゴリズムとプロトコルであり実装が容易であるという特徴がある．そこで，本研究では構造化 P2P ネットワーク方式の基礎として Kademlia を利用した．

Kademlia は ID 空間を木構造で管理して構造化 P2P ネットワークを実現する方法であり，BitTorrnet クライアント [3, 18] や，eMule [6] 等のアプリケーションにも広く用いられており，現在，最も利用されている構造化 P2P ネットワークの一つである．本章では，この Kademlia のアルゴリズムとプロトコルについて説明する．

3.1 表記と定義

3.1.1 表記

本章以下では，以下の表記を用いる．

$a \otimes b$　：　a と b の論理積

$a \oplus b$　：　a と b の排他的論理和

3.1.2　定義

共通プレフィクス長

11100 と，**11**000 という 2 進数の数値があったとき，この数値間では上位 2 ビットまでが連続して共通である．この時，この 2 数値間の共通プレフィクス長は 2 であると言う．共通プレフィクス長の定義は以下のようになる．

定義 3.1　共通プレフィクス長とは，上位何ビットが共通かを示す値である．すなわち，ID_A と ID_B があり，両者の上位 n ビットが同じであったとするとき，ID_A と ID_B の共通プレフィクス長は n である．

XOR 距離

構造化 P2P ネットワークでは各ノードにユニークな ID を割り振り，この ID を元にルーティングを行う．Chord や Symphony などは ID 間の距離に，減算を用いて求めた差を用いているが，Kademlia では，距離の導出に排他的論理和を用いる．XOR 距離の定義は以下のようになる．

定義 3.2　ID_A と ID_B 間の XOR 距離は，互いの排他的論理和であり，すなわち XOR 距離 D_{XOR} は，$D_{\mathrm{XOR}}(\mathrm{ID}_A, \mathrm{ID}_B) = \mathrm{ID}_A \oplus \mathrm{ID}_B$ となる．

一般的に，構造化 P2P ネットワークでは ID に 160 ビット等の大きな数値を用いる事が多い．そのため，減算を用いた距離計算には多倍長演算が必要となるが，XOR 距離だとその必要はなく，非常に簡素に記述できる．

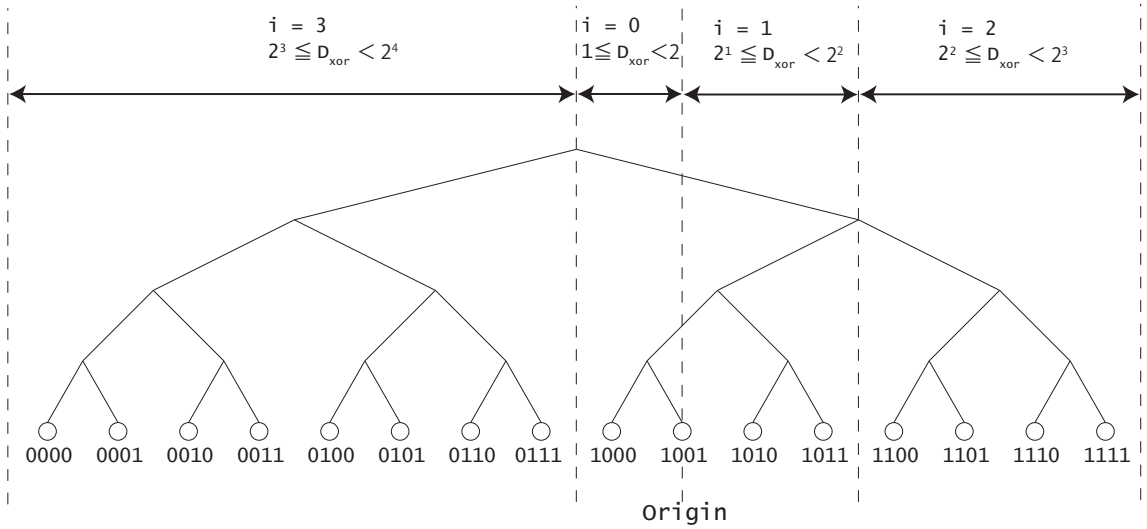


図 3.1 Kademlia の ID 構造とルーティングテーブル

3.2 ルーティングテーブル

3.2.1 ID 構造と k-buckets

Kademlia のルーティングテーブルは，ID のビット数分のリストから成り立つ．つまり，ID が 160 ビットならば各ノードは 160 個のリストを保持する．これらのリストは k-buckets と呼ばれ，$i \in \{0, \ldots, 159\}$ 番目のリストにあるノードの ID は，自身の ID から $[2^i, 2^{i+1})$ だけ離れたものとなっている．各リストには最大で k 個のエントリが挿入され，この k は，Maymounkov らの論文では 20 となっている．なお，このエントリには通信を行うために必要な，IP アドレス，ポート番号，ID 等の情報を含む．

図 3.1 は Kademlia の ID 構造と，k-buckets の各リストが保持する ID の関係を示している．なお，この図では，1001 という ID を持つノードの k-buckets のリストについて描かれている．図からわかるように，リストの番号である i が大きくなるほど，対象とすべき ID の範囲が広くなっていくことがわかる．

3.2.2 ルーティングテーブルのへの追加と検索

アルゴリズム 3.1 は，Kademlia のルーティングテーブルに，新たな ID_{add} を追加するアルゴリズムとなる．

アルゴリズム 3.1　Kademlia のルーティングテーブル追加

Require: ID_{add} is an ID to be added. ID_{mine} is an ID of the node having k-buckets to which ID_{add} is added

1: i = the length of common prefixal bits between ID_{mine} and ID_{add}
2: $bucket$ = k-buckets[i]
3:
4: **if** $bucket$.length $< k$ **then**
5:　$bucket$.enqueue(the infomation of ID_{add})
6:　**return**
7: **end if**
8:
9: $head$ = $bucket$.dequeue()
10: send a *ping* to $head$
11: **if** receive the reply **then**
12:　$bucket$.enqueue($head$)
13: **else** {no reply}
14:　$bucket$.enqueue(the infomation of ID_{add})
15: **end if**

ID_{add} は挿入するノードの ID であり，ID_{mine} は自身の ID となる．1 行目では，新たなノード情報の追加を行う先のリストを求めるため，追加するノードの ID と自身の ID の共通プレフィクスビット長 i を求めている．例えば，1110 と 1100 という ID があった場合，連続する上位 2 ビットが等しいため，両者の共通プレフィクスビット長は 2 となる．

2 行目では i より，挿入先のリストを取得している．挿入先リストを決定した後，リストの最後尾に新たなノードの追加を行う．

4 - 7 行目ではリストのエントリ数を取得し，エントリ数が k 未満であったら，リストの最後尾に ID_{add} を追加して処理を終了させている．

9 行目以降はリストのエントリ数が k 個でに達していた場合の処理となる．9 - 10 行目では，リストの先頭ノードに対して ping メッセージを送信し，生存確認を行っている．12 行目では，ping メッセージに対する応答があった場合の処理となる．先頭ノードの生存が確認できたなら，そのノードを最後尾に移動させ，新たなノードの情報は追加しない．逆に 14 行目では，応答が無かった場合は，そのノードをリストから削除し新たなノードの情報を最後尾に追加している．

このように，Kademlia では，ノードの追加と生存確認が同時に行われることになる．ただし，既に存在するノードの情報を追加しようとした場合は，単純に，リストの最後尾に移動させてテーブルの更新を行う．

ノードを追加するタイミングだが，これは，Kademlia で使用される通常のメッセージを交換した際に行われる．このため，Kademlia ではルーティングテーブルを維持するためのプロトコルが必要ない．Chord や Symphony，Pastry と言ったアルゴリズムでは，経路の維持に複雑なプロトコルを用いる必要があり，そこがボトルネックや，バグの温床となりやすいが，Kademlia での経路維持は比較的容易に行える．

Kademlia では，ある ID をキーとしてルーティングテーブルを検索すると，その ID より XOR 距離が最も近い n 個のノード情報が得られるようにしなければならない．検索アルゴリズムの詳細については，4 章で議論を行う．

3.3 プロトコル

Kademlia では，ping，find node，find value，store の 4 つのメッセージが用いられる．このうち，find value と store メッセージは DHT を実現するためのメッセージである．本節では，これら 4 つのメッセージについて説明する．

3.3.1　ping メッセージ

ping メッセージはノードの生存確認を行うために用いられる．ping を受け取ったノードは応答として pong メッセージを返信する．

3.3.2　find node メッセージ

find node メッセージは，ある値を宛先 ID として，その ID と最も近い ID を持つノード n 個を検索するのに用いられる．find node メッセージを受信したノードは，find node メッセージの宛先 ID から最も近い ID を持つノード n 個を自身のルーティングテーブルから検索し，その n 個のノード情報（ID, IP アドレス, ポート番号）を応答として返信する．

3.3.3　find value メッセージ

find value メッセージは find node メッセージを少し変えたもので，DHT の Key-Value ペア取得に用いられる．find value メッセージは，まず Key のハッシュ値を取り，そのハッシュ値と Key を宛先 ID として設定する．find value メッセージを受け取ったノードは，自身の DHT エントリを調べ，宛先 ID と Key に該当するデータがあれば，それを返信する．もしも該当するデータを保持していなければ，find node メッセージと同じように，宛先 ID と近い ID を持つ n 個のノードをルーティングテーブルから検索して返信する．

3.3.4　store メッセージ

store メッセージは find node メッセージの結果として得られるノード情報を使って，データの保存を行うためのメッセージである．store メッセージは，Key-Value ペアと，Key のハッシュ値が共に送信され，store メッセージを受け取ったノードは，自身の DHT エントリに，store メッセージ中に含まれる Key-Value ペアを保存する．

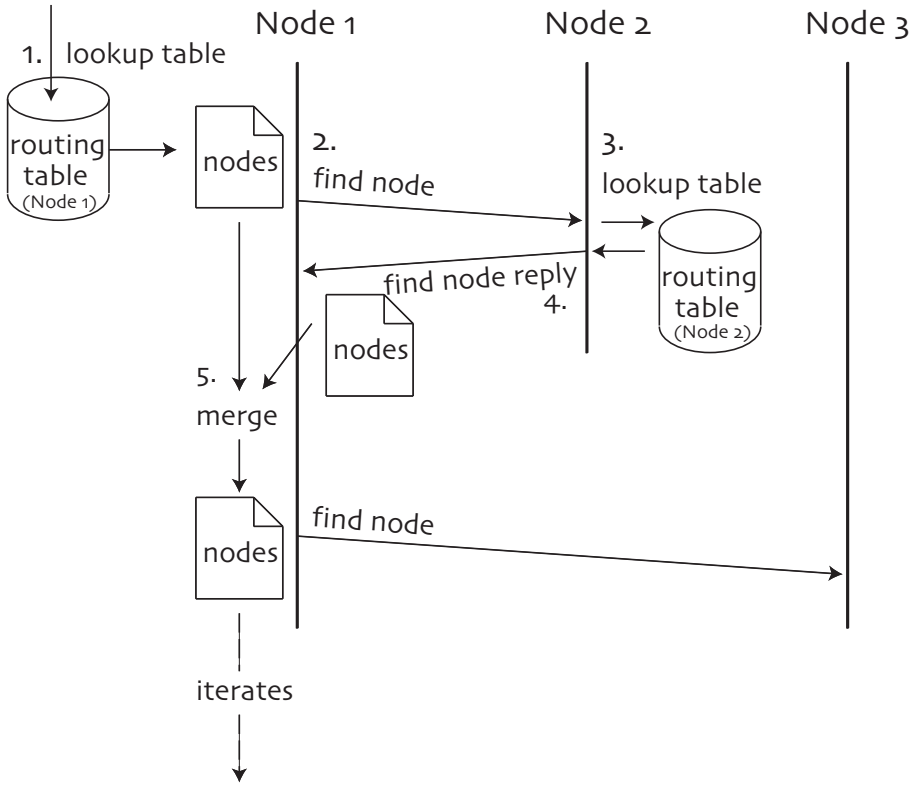


図 3.2 find node 操作

3.4 find node 操作

find node メッセージを複数のノードに対して反復的に送信すると，ある任意の宛先 ID と最も近い ID を持つノードの集合を得ることが出来る．Kademlia では，DHT のデータ保存は find node メッセージの反復送信に得られたノードの集合をもとに行われる．また，DHT でのデータ取得は，find value によって行われるが，基本的に find node メッセージによる反復操作と殆ど変わらない．

find node メッセージを複数のノードに対して反復的に送信してノードの集合を得る操作は，Kademlia を用いて DHT を実現するうえで非常に重要な操作であると言える．そこで，本節では，Kademlia で find node メッセージを利用してノードを得る方法について説明する．なお，本論文では find node メッセージを反復的に送信してノードの情報を得る操作を，find node を行うと記述する．

図 3.2 は，find node を行っている様子を示している．まず始めに，Node 1 は自身のルーティングテーブルから，ID_{key} と近いノードのリストを得る．次に，最も ID_{key} と近い ID を持つ Node 2 に対して find node メッセージを送信する．find node メッセージを受信した Node 2 は，同様に，ID_{key} と近いノードの情報をルーティングテーブルから検索し，Node 1 にその情報を find node reply メッセージとして返信する．Node 1 は，find node reply メッセージに含まれるノードの情報と，操作 1 で得られたノードをマージし，繰り返し問い合わせる．

find node では，取得したいデータのハッシュ値を ID_{key} として問い合わせを行う．find node メッセージを受け取ったノードは，ID_{key} と近い距離にある ID を持つノードの情報を，自身のルーティングテーブルから検索し応答として返信する．find node を行うノードは，ノードのリストを ID_{key} と近い順に保持しておき，応答メッセージから得られたノードとマージしていく．なお，find node を行う際には，ID_{key} と近いノードに対して優先的にメッセージを送信する．先頭 n 個全てのノードに対してメッセージを送信し終えた時点で，find node は終了する．

以上が基本的な find node の説明であるが，実際には検索時の効率を向上させるために，複数のノード α 個に対して同時にメッセージの送信を行う．

3.5 結論

構造化 P2P ネットワークは，ネットワークが大規模になったとしても効率よく検索クエリが配送されることが望まれる．Kademlia ではルーティングテーブルである k-buckets のデータ構造に，木構造を採用しており，その木の高さはノード数を N としたとき，$\log N$ となった．そのため，ノード数が多くなるとテーブル検索に必要なコストが増大していった．そこで，本研究では，木構造の代わりに配列を用いる方法を提案した．

本研究では，木構造の場合はノードを辿る回数をコストとし，配列の場合，バケットをルックアップする回数をコストと定義した．その結果，ランダムな ID をルーティングテーブルから引く場合は，木構造の場合は平均コストが 2 となり，配列の場合は平均コス

トが 1 となり，必要なコストはほとんど変わらないことが明らかとなった．

一方, find node 時に必要となる検索コストの平均は, 木構造の場合は平均して $O(\log N)$ となり，配列の場合は平均して $O(1)$ となった．シミュレーションの結果では，木構造の場合はネットワークの規模が 100,000 ノードの時，検索コストの合計は $k = 20$ の場合，平均して約 35 程度となるのに対して，配列の場合は約 3 のコストが必要となり，大きな差が出ることが明らかとなった．

ルーティングテーブルに含まれる情報が粗な場合，配列の場合でも，複数のバケットをルックアップする必要がある．本研究では粗な場合の検索方法アルゴリズムについても提案を行った．しかし，配列の場合でも粗な場合は最悪で $\log N$ の検索コストが必要となり，これは木構造の場合と等しくなることが明らかとなった．

次に，必要となるメモリ量だが，木構造の場合は必要なメモリ量は $\log N$ 個だけのバケットとなるが，配列の場合は初期状態で ID のビット数分のバケットが必要となる．一般的に，構造化 P2P ネットワークの ID は 128 や 160 ビットなどの大きな値が用いられることが多く，配列の場合は余分にメモリ量が必要となる．しかしながら，実際に利用されるバケットは平均して上位 $\log N$ 番目のバケットまでであるので，配列の場合でも動的に確保することは可能である．

第 4 章

Kademlia のルーティングテーブルとデータ構造

Kademlia のルーティングテーブルは k-buckets により実現されるが，実際の管理，検索には木構造が用いられる．しかしながら，木構造による検索はルーティングテーブルに含まれるノード数が多くなるほど，検索に必要なルックアップの回数が増加していく．そこで本研究では，ノード数が多くなっても効率的にルーティングテーブルの検索が行えるよう，配列を用いて管理する方法を提案する．

本章では，k-buckets の木構造による管理方法と配列による管理方法について説明し，両手法についての議論を行う．

4.1 既存方式：木構造でのルーティングテーブル管理

Maymounkov らの論文 [37] では，Kademlia のルーティングテーブル（k-buckets）を木構造で管理する方法を提案している．図 4.1 は，ID が 0 であるノードのルーティングテーブルが成長していく様子を示している．ただし，ここで，ID のビット長は 160 ビットとする．

図 4.1 の (a) は，ルーティングテーブルの初期状態となる．まずはじめには，すべてのノード情報を入れるための，$i = 0 \cdots 159$ という k-bucket が唯一存在する．初期状態で

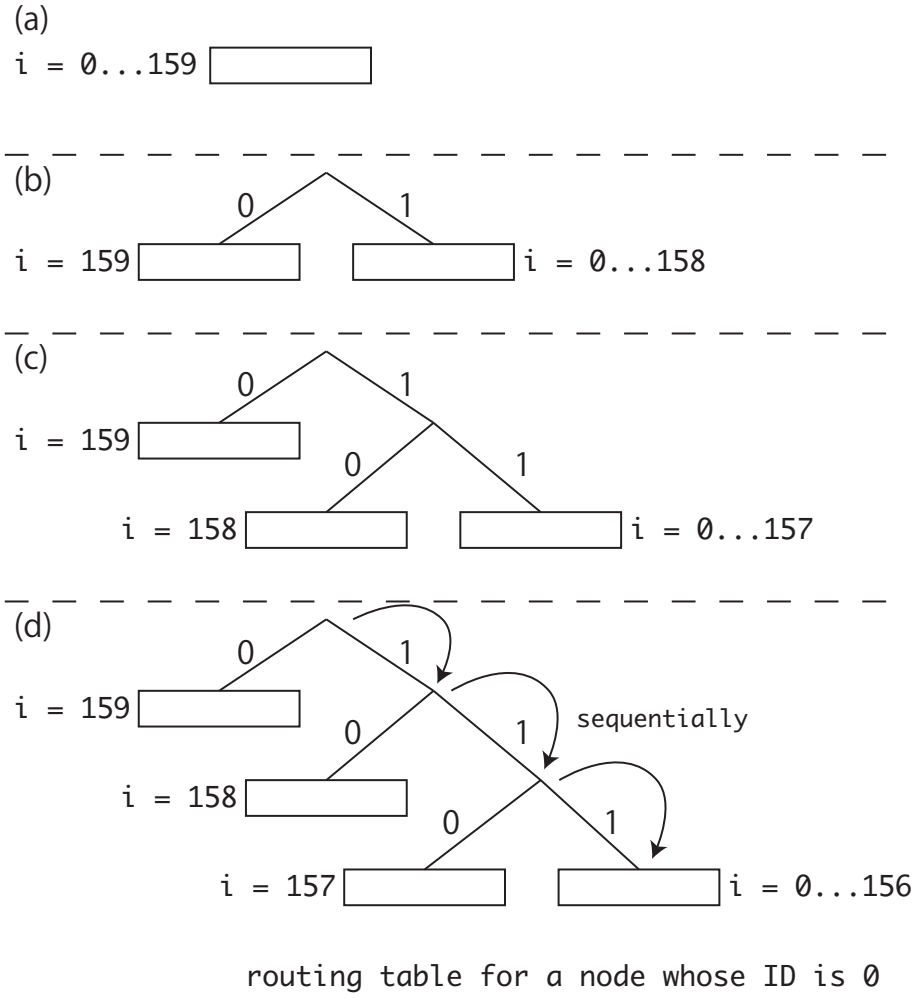


図 4.1　木構造ルーティングテーブルの成長

は，この唯一の k-bucket にノードの情報を追加していき，この k-bucket のサイズが k に達した場合，図 4.1 の (b) で示されるように，$i = 159$ と $i = 0 \cdots 158$ の二つの k-bucket に分割される．

さらに，図 4.1 の (b) にある，$i = 0 \cdots 158$ の k-bucket のサイズが k に達した場合，同じように分割される．図 4.1 の (c) は，図 4.1 の (b) からさらに k-bucket が分割された様子を示している．図 4.1 の (c) では，新たに，$i = 158$ と $i = 0 \cdots 157$ の k-bucket が作成されている．

図 4.1 の (d) は同じように，図 4.1 の (c) から，さらに分割された様子を示している．ところが，この図 4.1 の (d) で示したように，$i = 0 \cdots 156$ の k-bucket に到達するには，

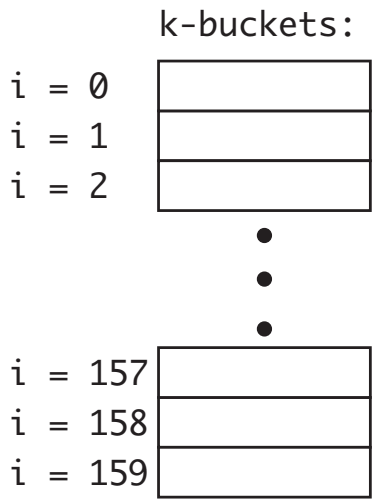


図 4.2 配列でのルーティングテーブル

3 回枝を辿る必要がある．このように，木構造の管理ではルーティングテーブルのサイズが大きくなるとともに，検索効率が悪化していくことがわかる．

4.2 配列でのルーティングテーブル管理

Kademlia では，木構造でルーティングテーブルを保持する代わりに，配列で保持することも可能である．木構造での管理方法では，k-bucket の要素数が k を超えたときに分割を行っていった．この方法だと，ルーティングテーブルが保持するノード数が少ない時に，宛先 ID から近い ID を持つ n 個のノードを得ようとした場合でも，効率的に検索を行うことができる．これは，ルーティングテーブルが粗な時は，bucket の数も少なくデータが固まって存在するためである．例えば，ルーティングテーブルが保持している全ノード数が k であり，ある ID から近い n 個のデータを得たい場合は，図 4.1 の (a) で示されている一つの bucket にのみにアクセスすれば良いことは明らかである．

4.2.1 ルーティングテーブルが粗な場合の検索

一方，配列を用いた管理方法の場合に n 個のノード情報を得ようとすると，$n \leq k$ ならば，ルーティングテーブルが密な場合は一回のテーブルルックアップで検索することが可

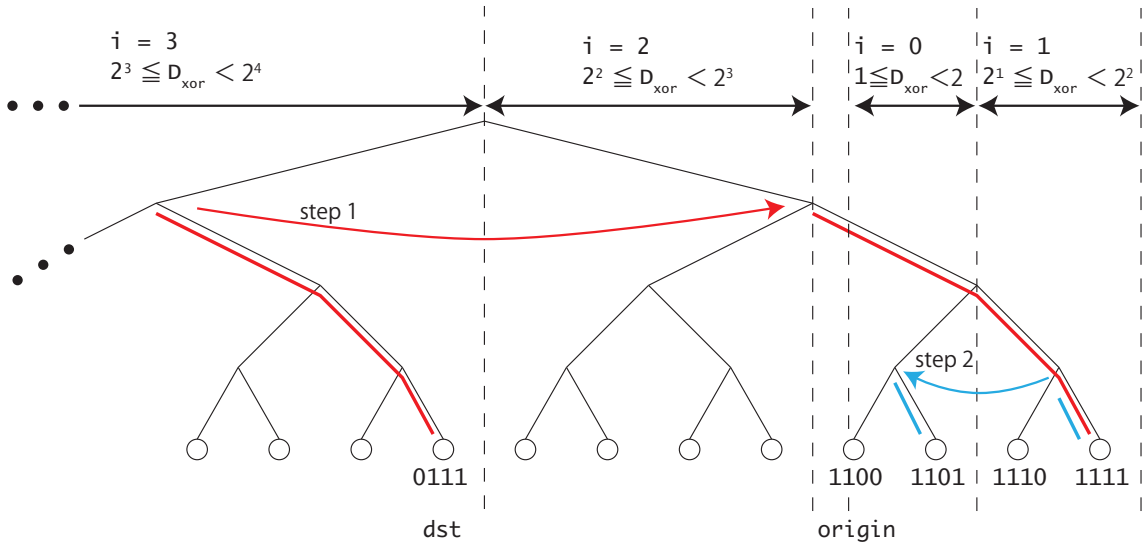


図 4.3　枝の移行によるルーティングテーブルの検索

能である．しかしながら，ルーティングテーブルが粗な時は，複数の bucket にアクセスしなければならない．

図 4.2 は，ルーティングテーブルを配列で保持した様子を示している．いま，ID_{mine} を自身の ID，ID_d を宛先 ID として，p を両者の共通プレフィクス長すると，$i = 159 - p$ となる．したがって，ID_d に近い ID を持つ n 個のノードをルーティングテーブルから検索するためには，まずはじめに，$i = 159 - p$ の bucket を検索することになる．もしも，ルーティングテーブルが密であり $i = 159 - p$ の bucket に十分なデータが含まれていれば，一回のテーブルルックアップで終了する．

検索する bucket に十分なデータが含まれていない場合は，他の bucket も検索しなければならない．最も単純な方法は総当りで検索する方法だが，これは効率が悪い．そこで，本研究では，図 4.3 で示すような，木の枝を移行しつつ検索を行うアルゴリズムを提案する．

図 4.3 は 1100 という ID を持つノードのルーティングテーブルを表しており，そこから 0111 という ID を宛先として検索している様子を表している．ただし，ここでは ID が 4 ビットであるため $i = 3 - p$ となる．

まずはじめに，1100 と 0111 では共通プレフィクス長は $p = 0$ であるため，$i = 3$ の

bucket が検索される．その次に，$i = 3$ との bucket 以外で 0011 と近い ID を保持している bucket を探す必要があるが，これは，図 4.3 の step 1 で示すように，0111 までの枝を反対側に移行することで行える．step 1 から，0111 は $i = 1$ の bucket へと移行しているため，次に検索すべき bucket は $i = 1$ の bucket となる．図より，$i = 3$ 以外で 0111 と最も近い ID を持つ bucket は $i = 1$ の bucket で有ることは明らかである．

その後，同じように今度は図 4.3 の step 2 で示すように，step 1 で移行した枝からさらに反対のサイドへ移行する．すると，$i = 0$ が移行先の bucket となり，これは 0111 と三番目に近い ID を持つノードを保持する bucket である事は図より明らかである．全ての移行が終えたならば，今度は $i = 0$ から順に木の移行時に辿っていない bucket を検索すれば良い．このように，順に木の枝を移行することによってルーティングテーブルのルックアップを効率的に行うことができる．

アルゴリズム 4.1 は枝の移行を行って，Kademlia のルーティングテーブルを検索するアルゴリズムである．ただし，ID_{mine} は自身の ID，ID_{dst} は宛先とする ID，m は取得するノード数の最小数，$blen$ は ID のビット数から 1 引いた値，i_{min} は情報を含む k-buckets のインデクス値のうち最小の値となる．

アルゴリズム 4.1 枝移行による検索

Require: find at least m nodes whose IDs are closer to ID_{dst} than others from a node whose ID is ID_{mine}. $blen$ indicates the bit length of ID, minus 1. i_{min} is the minumun index of nodes holding information

1: $d = \mathrm{ID}_{dst}$

2:

3: **while** $nodes.length < m$ **do**

4: p is the common prefix length between ID_{mine} and d

5: $i = blen - p$

6:

7: **if** $i < i_{min}$ **then**

8: $nodes$.insert(ID_{mine})
9: **break**
10: **else**
11: $nodes$.insert(each of k-buckets[i])
12: **end if**
13:
14: $d = d \oplus (1 \ll i)$
15: **end while**
16:
17: **return** $nodes$

1 行目ではまず，d を宛先 ID の ID_{dst} で初期化する．枝の移行は，この d のビットを反転することで行うことが出来，実際の枝移行は 3 行目以降の while 文中で行われる．この while 文は，検索したノードの数が m 以上となった点で終了する．

4 行目では共通プレフィクス長を求めて，その値を p に代入し 5 行目で，検索すべき bucket のインデクス値 i を求めている．ただし，この i が i_{min} 未満であったなら，8〜9 行目で自身を結果に保存しループを終了している．逆に，もしも i_{min} 以上ならば，i 番目の k-buckets が保持しているデータを結果に保存する．

14 行目が枝の移行を行っている箇所となる．図 4.3 より，枝の移行は i 番目のビット数を反転すれば行えることは明らかであるため，ここでは，排他的論理和を用いて i 番目のビットを反転している．最後に，17 行目で結果を返している．

図 4.3 とアルゴリズム 4.1 から明らかなあるように，枝の移行による検索では全ての k-buckets を検索しない．そのため，先に述べたように枝移行による検索を終了した後に，取得したデータの数が，希望した数に達していなかった場合，$i = 0$ から順に総当りで k-bukcets の検索を行う必要がある．

アルゴリズム 4.2 は，枝移行による探索が終了した後に行う検索アルゴリズムとなる．ただしここで，m は本アルゴリズムで取得したいノードの最低数，ID_{mine} は自身の ID,

ID_{dst} は宛先 ID，invert() 関数はビット反転の関数となる．

アルゴリズム 4.2 枝移行探索後の検索

Require: find at least m nodes whose IDs are closer to ID_{dst} than others from a node whose ID is ID_{mine}. invert() function returns the bit wise inverted value of the argument.

```
1: d = invert(ID_mine ⊕ ID_dst)
2:
3: for bucket each k-buckets do
4:    if nodes.length ≥ m then
5:       break
6:    end if
7:
8:    if d ⊗ (1 ≪ bucket.i) ≠ 0 then
9:       nodes.insert(each of bucket)
10:   end if
11: end for
12:
13: return nodes
```

1 行目では検索すべき k-buckets の判定を行うための，ビット列を生成している．アルゴリズム 4.1 では，自身の ID と宛先 ID で共通のビットが立っていた場合，そこに相当する k-buckets を検索していた．そのため，ビットが異なっている箇所を調べると，検索すべき k-buckets であるかどうかがわかるが，これは，自身の ID と宛先 ID の共通ビットの排他的論理和を取り，ビット反転することで可能となる．

3 行目以降の while 文では，k-buckets を順に走査していき，アルゴリズム 4.1 で検索していない bucket であったなら，9 行目で結果に保存している．もし，結果の数が m より大きくなっているなら 5 行目でループを抜け，最後に 13 行目で結果を返している．

ただし，実際にはこの while ループの開始は，アルゴリズム 4.1 でいう i_{min} 番目の bucket から開始したほうが良い．なぜならば，ルーティングテーブルが粗な状態では，殆どの場合，インデクスが小さい bucket は情報を持っていないためである．

4.3　評価

本章では，木構造で k-buckets を持つ代わりに，配列で持つ方法を提案した．そこで，本節にて，両者の検索コストについて比較を行う．なお，ここで言う検索コストは，配列の場合はエントリをルックアップする回数であり，木構造の場合は枝を辿る回数を，それぞれコストと定義する．

4.3.1　一様ランダムな ID をテーブルから引く場合の検索コスト

一様ランダムに生成された ID をテーブルから引く場合の検索コストについて，配列の場合と木構造の場合に考える．

配列の場合

配列の場合は，ルーティングテーブルが密に埋まっており，検索するノードの数 n が $n \leq k$ であるなら，1 回のテーブルルックアップで済む事は自明である．すなわち，配列の場合は検索コストは 1 となる．

木構造の場合

いま，同じくルーティングテーブルが密に埋まっている状況を考える．この場合，ID $= 0 \cdots 0$ の k-buckets を考えると，$i = 159$ 番目の bucket には，$1 * \cdots *$ の ID を持つノードが入り．また，$i = 158$ 番目の bucket には，$01 * \cdots *$ の ID を持つノードが入る事は明らかである．また，159 番目の bucket を検索するコストは，1 であり，158 番目の bucket を検索するコストは 2 である．このように，157 番目以降の bucket を検索するコストは比例して増えていくことは，図 4.1 よりも明らかである．

なおここで，検索する ID は一様ランダムであるので，$1*\cdots*$ の ID を持つノードは，全体の 50[%] を占め，$01*\cdots*$ の ID を持つノードは，全体の 25[%] を占める．このように，157 番目以降の bucket に格納されるノードの割合は，反比例して減っていく事は明らかである．

従って，ID が n ビットの場合，一様ランダムな ID を木構造のルーティングテーブルから検索する場合の平均検索コスト C は以下のようになる．

$$\begin{aligned} C &= \frac{1}{2} + 2\frac{1}{2^2} + 3\frac{1}{2^3} + \cdots + n\frac{1}{2^n} \\ &= \sum_{i=1}^{n} i \left(\frac{1}{2}\right)^i \end{aligned} \tag{4.1}$$

なお，数式 4.1 は，n が自然数であれば必ず 2 以下となる．すなわち，

$$\sum_{i=1}^{n} i \left(\frac{1}{2}\right)^i \leq \sum_{i=1}^{\infty} i \left(\frac{1}{2}\right)^i = 2 \tag{4.2}$$

となる．

証明．$|x| < 1$ の時，$\dfrac{1}{1-x}$ をマクローリン展開すると，

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots \tag{4.3}$$

となる．この両辺を微分すると

$$\frac{1}{(1-x)^2} = 1 + 2x + 3x^2 + 4x^3 + \cdots \tag{4.4}$$

となる．さらに，この両辺に x を乗算すると，

$$\begin{aligned} \frac{x}{(1-x)^2} &= x + 2x^2 + 3x^3 + 4x^4 + \cdots \\ &= \sum_{i=1}^{\infty} i\, x^i \end{aligned} \tag{4.5}$$

となり，このとき $x = \dfrac{1}{2}$ の場合が式 4.2 に相当し，$\dfrac{x}{(1-x)^2}$ に $x = \dfrac{1}{2}$ を代入すると 2 となる． (証明終)

よって，ランダムな ID をルーティングテーブルから引く場合は，木構造で管理した場合でも平均コストは 2 未満となり，配列の場合と比較してもほとんど変わらない．

4.3.2 FIND NODE 時の検索コスト

次に，FIND NODE を行った場合の平均コストについて説明する．FIND NODE を行った場合，複数のノードに問い合わせることになる．そのため，ここでは 1 ノードあたりの平均コストについて議論する．

ただし，その前に，k-buckets が保持しているノードの数と，FIND NODE 終了までに必要な問い合わせの回数について議論を行う．

ノード数と k-buckets の高さ

今，木構造でルーティングテーブルを管理した場合の木の高さを k-buckets の高さと呼ぶとする．例えば，図 4.1 の (d) では k-buckets の高さは 3 である．この時，Kademlia のネットワークに参加しているノードの数が N であり，各々のノードが持つ ID が一様ランダムに分布していたとしたら，k-buckets の高さは平均して $\log N$ となる．これは，先程の議論と同じく，自身の ID $= 0$ とした場合，$1 * \cdots *$ の ID を持つノードは，全体の 50[%] を占め，$01 * \cdots *$ の ID を持つノードは，全体の 25[%] を占めるからである．

図 4.4 は k-buckets の高さの理論値と，実測値を表したものである．実測値は，ランダムに ID をノード数だけ生成したときの，k-buckets の高さの平均値であり，ここでは生成した全てのノードの k-buckets の高さをサンプルとしている．なお，ここでは $k = 1$ として計測を行った．この結果より，理論通り k-buckets の高さの平均は $\log N$ となることがわかる．

FIND NODE 終了に必要な問い合わせ回数

FIND NODE に必要な問い合わせ数は，k-buckets の k の数に依存する．これは，k の数が多いほど，bucket 中の最大共通プレフィクス長が確率的に大きくなるからである．いま，FIND NODE を行うごとに平均して $L(k)$ ビットずつマッチしていくとすると，FIND NODE 終了に必要な問い合わせの平均回数は，参加しているノードの数を N とすると，$\dfrac{\log N}{L(k)}$ 回となる．

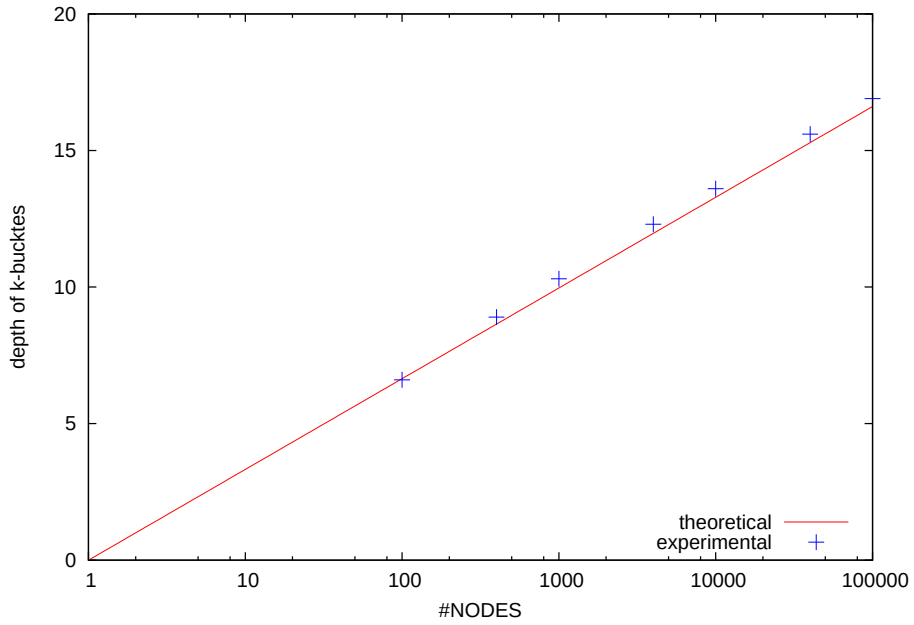


図 4.4 k-buckets の高さ

図 4.5 は，k を 1 から 30 まで変化させたときの，FIND NODE 終了に必要な問い合わせ回数をシミュレーションにて導出したものである．これより，問い合わせ回数は $\log N$ に比例して変化していることが分かる．また，k が大きいほど，必要な問い合わせ回数が減ることもわかる．

図 4.6 は，図 4.5 で求めた結果から，最小二乗法を用いて $L(k)$ の値を概算した結果である．これより，k が大きいほど $L(k)$ の値が大きくなっていくことがわかる．しかしながら，k が 30 と 40 の場合では，$L(k)$ にはほとんど変化が見られなかった．この結果より，実用的には，Kademlia では k の値は 5 から 30 程度が有用な値であると言える．

配列の場合における FIND NODE 時の検索コスト

配列の場合は，ルーティングテーブルの検索コストは 1 である．従って，FIND NODE 終了に必要な問い合わせ回数は $\dfrac{\log N}{L(k)}$ であるので，合計で，$\dfrac{\log N}{L(k)}$ の検索コストが必要となる．また，これは平均して 1 となる．

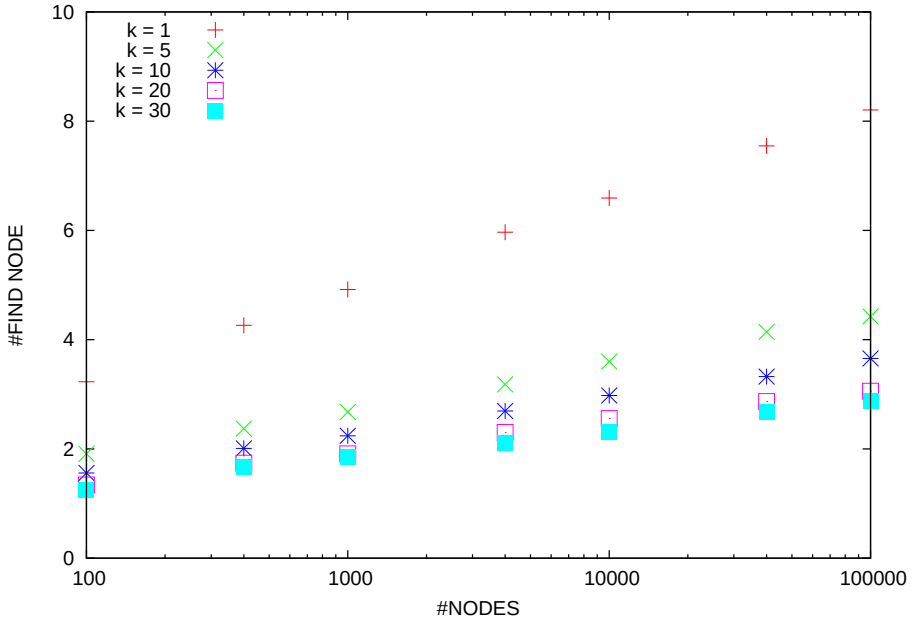


図 4.5　FIND NODE 終了に必要な問い合わせ回数

木構造の場合における FIND NODE 時の検索コスト

木構造の場合は，FIND NODE を行うごとに平均して $L(k)$ だけの検索コストが発生する．これは，$L(k)$ が木を辿るべき高さを意味しているからである．また，k-buckets の高さは $\log N$ となるため，FIND NODE を行う際に発生するルーティングテーブル検索コストの合計は，平均して

$$\sum_{i=1}^{i \cdot L(k) < \log N} i \cdot L(k) + \log N \tag{4.6}$$

となる．なお，平均検索コストは，FIND NODE 終了に必要な問い合わせ回数の $\dfrac{\log N}{L(k)}$ で割ったものになる．

図 4.7 は木構造のときの，FIND NODE に必要な合計コストの平均を，式 4.6 から導出したものである．配列の場合は，FIND NODE 終了に必要な問い合わせ回数と同じなので，合計コストの平均は図 4.5 と等しくなるが，100,000 ノードの場合を比較すると 10 倍以上の差がある事がわかる．

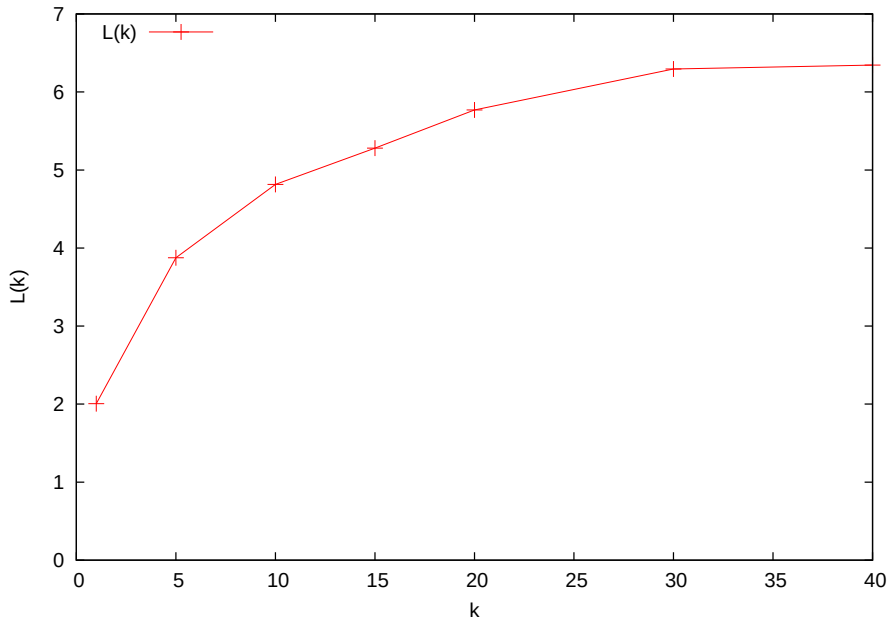


図 4.6 L(k) の概算

4.3.3 ルーティングテーブルが粗な場合のコスト

ルーティングテーブルが密な場合，配列だと検索コストは 1 となる．しかし，粗な場合は，4.2.1 節で説明したように，複数の配列へとアクセスしなければならない．このアクセスの回数だが，最悪の場合は，すべての配列を操作する必要がある．配列の数は，k-buckets の高さに依存するため，配列の場合，ルーティングテーブルが粗なときは最悪 $\log N$ のコストが必要となる．

一方，木構造の場合だが，粗な場合は最も深い bucket まで辿る必要があるため，こちらも同様に $\log N$ のコストが必要となり，配列と同じとなる．

4.3.4 メモリコストの比較

木構造の場合は，k-buckets を動的に確保しているため，必要なメモリ量は k-buckets の高さに依存する．したがって，木構造の場合，平均 $\log N$ 個の bucket が必要となる．

一方，配列の場合は初めに，ID のビット数分だけの bucket を用意しなければならな

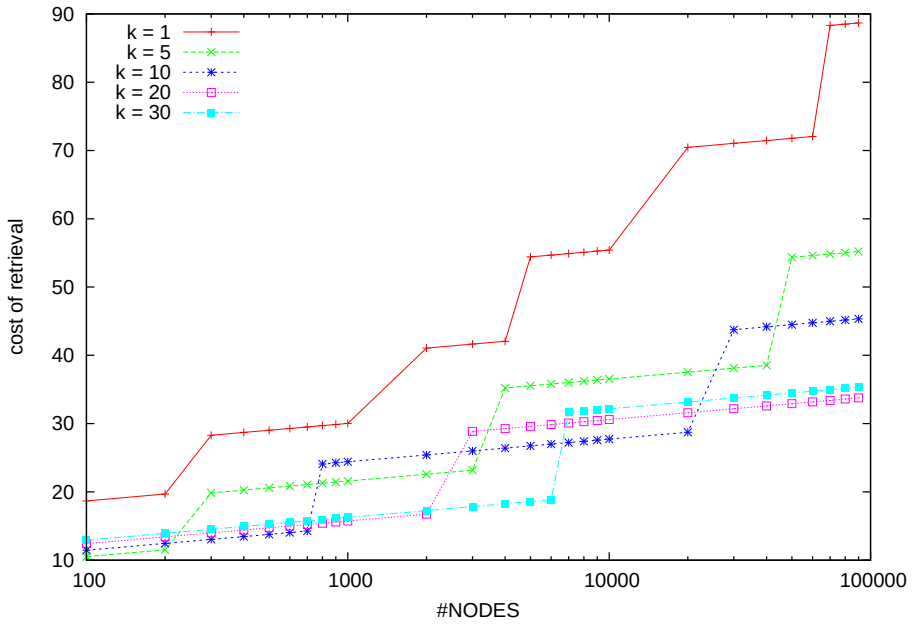


図 4.7　木構造のときの FIND NODE に必要な合計コストの平均

い．すなわち，ID が 160 ビットならば，160 個の bucket を容易する必要がある．配列の場合は，参加しているノードの数が 1,000,000 ノードでも，20 個の bucket で済む．しかし，配列の場合でも，実際に利用されるのは平均して $\log N$ 個の bucket のみであるので，利用される bucket の数に応じて動的確保することは可能である．

第 5 章

NAT 問題とその解決

構造化 P2P ネットワークは P2P ネットワークを実現する手法の一つであり，スケーラビリティや耐障害性が高いといった特徴を持つ．しかしながら，Chord [53] や Symphony [36]，Pastry [49]，Kademlia [37] といったほとんどのアルゴリズムでは，P2P ネットワークに存在するノードは全て，お互いに接続を開始しあう事のできる，対称な接続を行えることが前提となっている．そのため，NAT [42] が存在する場合，対称な通信が自由に行えない場合があり，既存のアルゴリズムをそのまま用いる事は出来ない．

構造化 P2P は大規模な P2P ネットワークを効率的に実現可能な手法である．しかしながら，NAT を考慮した NAT 問題フリーな構造化 P2P を実現しない限り，その効果を十分に発揮することが出来ない．そこで本研究では，NAT 問題フリーな構造化 P2P が必要であると考え，それを実現する構造化 P2P ネットワークの設計と実装を行った．本章では，一般的な NAT 越え手法である UDP Hole Punching [27] について説明した後，提案手法である DTUN 方式の NAT 越え手法と，実証ライブラリである libcage [9, 60] の設計と実装について解説する．

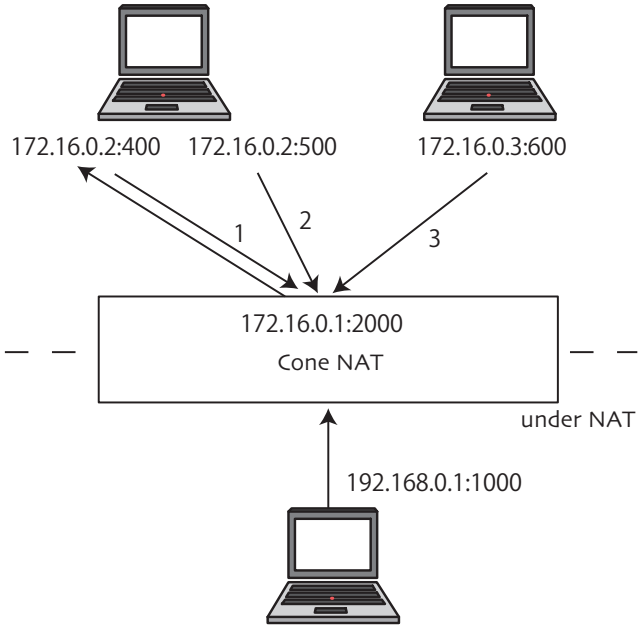


図 5.1　Cone NAT

5.1　既存研究

5.1.1　NAT の種類

Cone NAT と Symmetric NAT

本節では，NAT の種別 [47] について説明する．大きく分けて NAT には，Cone NAT と Symmetric NAT の二種類がある．Cone NAT では，送信元ソースアドレスが同じであれば，宛先が異なる場合でも同じソースアドレスに変換される．一方，Symmetric NAT では，宛先が違う場合には異なったソースアドレスに変換される．

図 5.1 は Cone NAT でソースアドレスが変換される様子を示している．Cone NAT では，ソースアドレスの変換が一対一で行われる．そのため，宛先が違っても，送信元ソースアドレスが同じであれば，同じアドレスに変換される．ただし，Cone NAT ではデータの受信時にフィルタリングを行う場合がある．

図 5.1 は，192.168.0.1:1000 というソースアドレスが，172.16.0.1:2000 に変換され，NAT 下にあるノードは，172.16.0.2:400 へのみデータを送信している様子を示している．

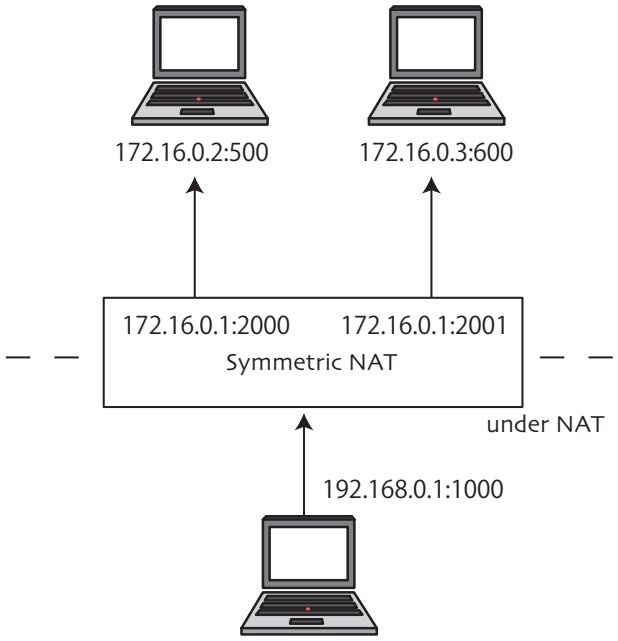


図 5.2 Symmetric NAT

ここで，NAT 下にあるノードにデータを送信するパターンとして，

1. NAT 下にあるノードが過去に送信した宛先からの送信 (172.16.0.2:400 → 172.16.0.1:2000)
2. NAT 下にあるノードが過去に送信した先と同じ IP アドレスであるが違うポート番号からの送信 (172.16.0.2:500 → 172.16.0.1:2000)
3. NAT 下にあるノードが過去に送信していない宛先からの送信 (172.16.0.3:600 → 172.16.0.1:2000)

があり，このパターンのうちどれをフィルタリングするかで，Cone NAT はさらに細かい種類に分けられる．すなわち，1. 2. 3 全てのデータを NAT 下のノードに配信するような NAT を Full Cone NAT，1. 2. のデータのみを配信する NAT を Restricted Cone NAT，1. のデータのみを配信する NAT を Port-Restricted Cone NAT と分類する．

図 5.2 は，Symmetric NAT によってソースアドレスが変換されている様子を示している．この図のように Symmetric NAT では，送信先アドレスが違う場合，異なるソース

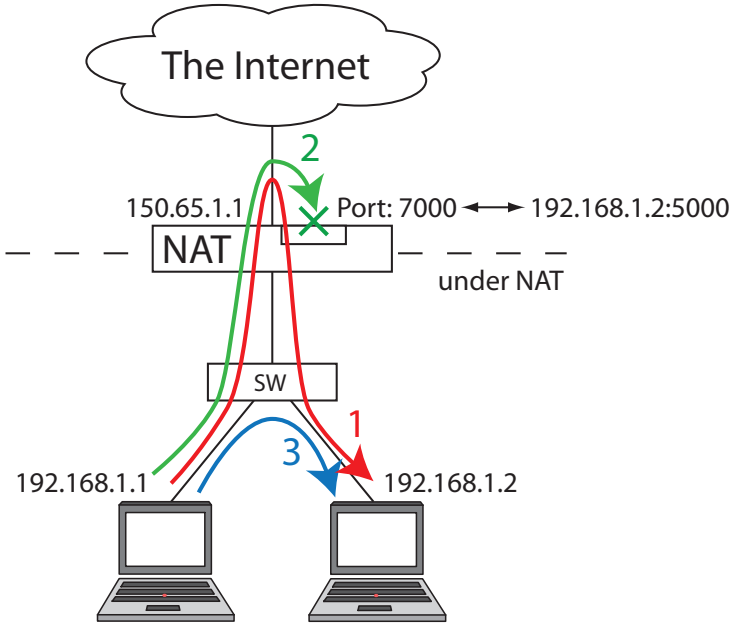


図 5.3　ヘアピンルーティング

アドレスに変換される．そのため，NAT 下にあるノードは，過去に通信を行っていないノードからのデータを受信することが出来ない．

NAT のヘアピンルーティング

次に，本節では NAT の大きな特徴の一つであるヘアピンルーティング [27] に関する問題について説明する．ヘアピンルーティングとは，同じローカルネットワークに存在するノードへ，NAT を中継して行う転送方式のことである．いくつかの NAT では，このヘアピンルーティングはサポートされず，P2P ネットワークアプリケーションで問題となる場合がある．

図 5.3 は，ヘアピンルーティングを行なっている様子を示している．ここでは，NAT は 150.65.1.1 というグローバルな IP アドレスを持っており，この NAT の下に 2 台のホストが存在する．なお，これらホストには，それぞれ 192.168.1.1 と 192.168.1.2 というプライベートな IP アドレスが割り振られており，グローバルアドレスの 150.65.1.1:7000 と，192.168.1.2:5000 というアドレスが NAT によって関連付けられている．

いま，192.168.1.1 のホストから，192.168.1.2:5000 へとデータを送信したいとする．

この時，宛先アドレスに 150.65.1.1:7000 を選択した場合のパケットの転送方法が，ヘアピンルーティングとなり，これは図 5.3 の 1 で示される経路で表されている．ヘアピンルーティングは，このように，ローカルネットワークの内側にあるホストに対して，グローバルアドレスを宛先としてデータを送信した場合に発生する．もしも，NAT が正しくルーティングを行っているのならば，図 5.3 の 1 で示されるように，パケットは正しく 192.168.1.2:5000 へと転送されるはずである．

ところが，実際には，図 5.3 の 2 で示されるように，正しいルーティングを行わずに，パケットを破棄してしまう NAT が存在する．このような NAT はヘアピンルーティング不可の NAT や，単純にヘアピン不可 NAT などと呼ばれ，P2P ネットワークアプリケーションでは問題となる場合がある．

しかしながら，多くの家庭で用いられる NAT では，図 5.3 の 3 で示されるように，ローカルのアドレスである 192.168.1.2 を宛先として指定すれば問題無く通信が出来る．ローカルネットワーク内にあるサービス発見としては，zeroconf [23] と Multicast DNS [24] を用いた Bonjour [4]・Avahi [2] や，UPnP [17] などがあり，P2P ネットワークアプリケーションにこれらを用いれば，ヘアピン不可の NAT であっても正しく通信ができる．

一方，Large Scale NAT [58] などに代表される，ISP レベルでの NAT は図 5.3 の 3 で示すような，ローカル同士の通信は行えない．しかしながら，これらの場合はヘアピンルーティングが可能となっているので，P2P ネットワークアプリケーションとしては問題にならない．

5.1.2 NAT 越えの各種手法

5.1.1 節で説明したように，NAT の種類によってはフィルタリングが行われるため，状況によっては対称な通信が不可能となってしまう．これを解決するための手法として，UDP Hole Punching [27] を用いた STUN [46] や，中継サーバを用いた TURN [45] 等が提案されている．本節では，NAT 越え，すなわち NAT を起因とした対称な通信の障害を解決するための手法について説明する．

UDP Hole Punching

UDP Hole Punching とは，Cone NAT のための NAT 越え手法である．5.1.1 節で説明したように，Restricted Cone NAT 又は Port-Restricted Cone NAT の場合，NAT 下にあるノードが通信したアドレスからのデータしかフォワーディングしない．そのため，NAT 下にあるノードへ向けてデータを送信したい場合は，先に NAT 下にあるノードがデータを送信したいノードへ向けて，何らかのデータを送信しておく必要がある．いま，一方のノードが Cone NAT 下に存在し，もう一方のノードがグローバルアドレスを持つノードであるとする．この場合，NAT 下に存在するノードからグローバルアドレスを持つノードへ向けて通信を行えば，NAT によりフィルタリングされずに対称な通信が可能となる．このような，NAT 下にあるノードから対向ノードへ向けて先に通信を行い，NAT のアドレス変換テーブルを更新する方法が UDP Hole Punching と呼ばれる手法である．

ところが，双方のノードが Cone NAT 下にあった場合には，第三者のグローバルアドレスを持つノードを，待ち合わせのために用意する必要がある．NAT 下にあるノード同士は第三者を通じて，お互いのグローバルアドレスとポート番号を通知し UDP Hole Punching を行う．Cone NAT ではソースアドレスとポート番号が同じであれば，同じグローバルアドレスとポート番号に変換されるため，UDP Hole Punching による通信が可能となる．しかしながら，Symmetric NAT ではソースアドレスとポート番号が同じであっても，宛先アドレスやポート番号が異なると，異なるグローバルアドレスとポート番号に変換されるためこの手法は適用出来ない．

図 5.4 は，グローバルアドレスを持つ仲介サーバを介して，NAT 下にある 2 つのノード，host A と host B が UDP Hole Punching を行っている様子を示している．ただし，ここでは host A と host B はお互いに仲介サーバのアドレスを知っており，host A から host B へ通信を開始しているとする．また，host A の待ち受け IP アドレスは IP1 で，待ち受けポート番号は PORT1，host B の待ち受け IP アドレスは IP2 で，待ち受けポート番号は PORT2 であり，それぞれの IP アドレスとポート番号は NAT により，TIP1,

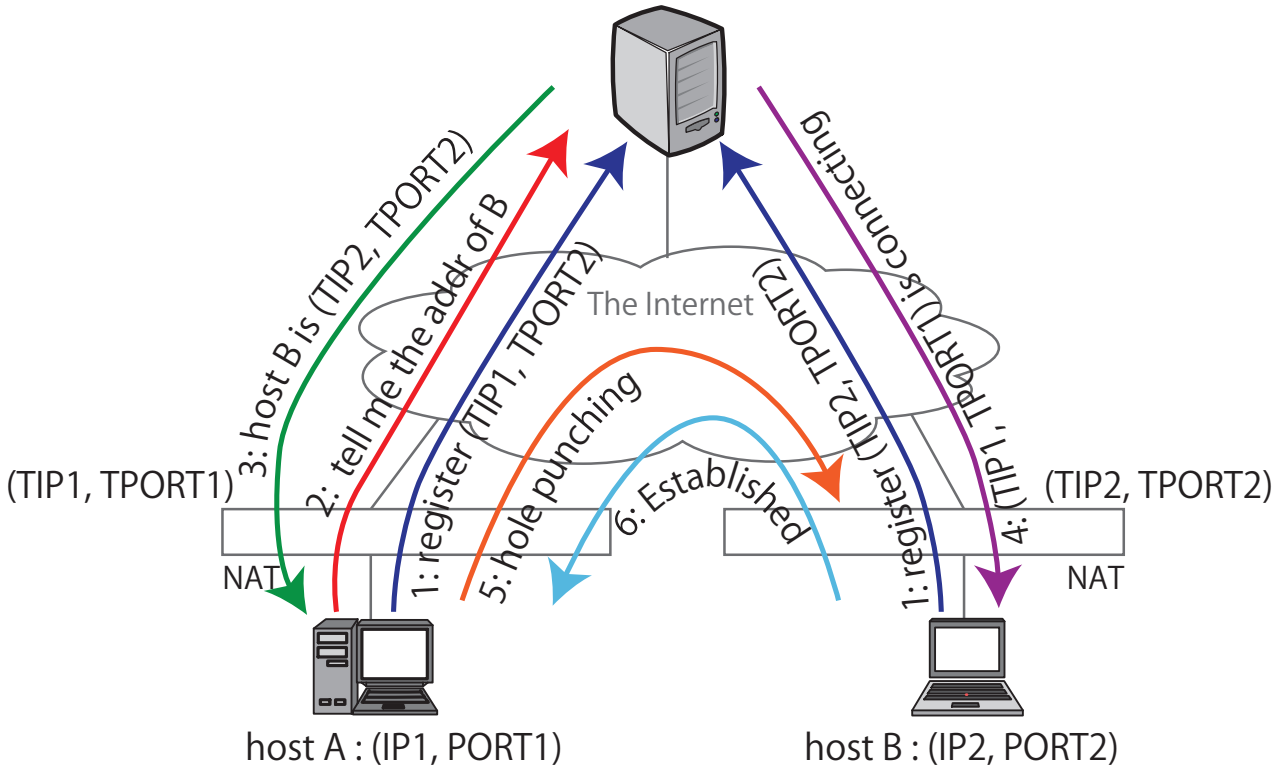


図 5.4 UDP Hole Punching

TPORT1, TIP2, TPORT2 と変換される．仲介サーバを用いた UDP Hole Punching の具体的な手順は以下のようになる．

1. host A と host B は，仲介サーバに対して自身の待ち受けアドレスを通知する．
2. host A が host B のアドレスを仲介サーバに聞きに行く．
3. 仲介サーバは host B のグローバルアドレスである TIP2, TPORT2 を host A に教える．
4. 仲介サーバは host A が接続してきている事と，host A のグローバルアドレスである TIP1，TPORT1 を host B へ伝える．
5. host A は host B の TIP2，TPORT2 へ向けてパケットを送信する．ただし，この時点では host B にはパケットは届かない．
6. host B は host A の TIP1，TPORT1 へ向けてパケットを送信する．ここで，ようやく対称な通信が可能となる．

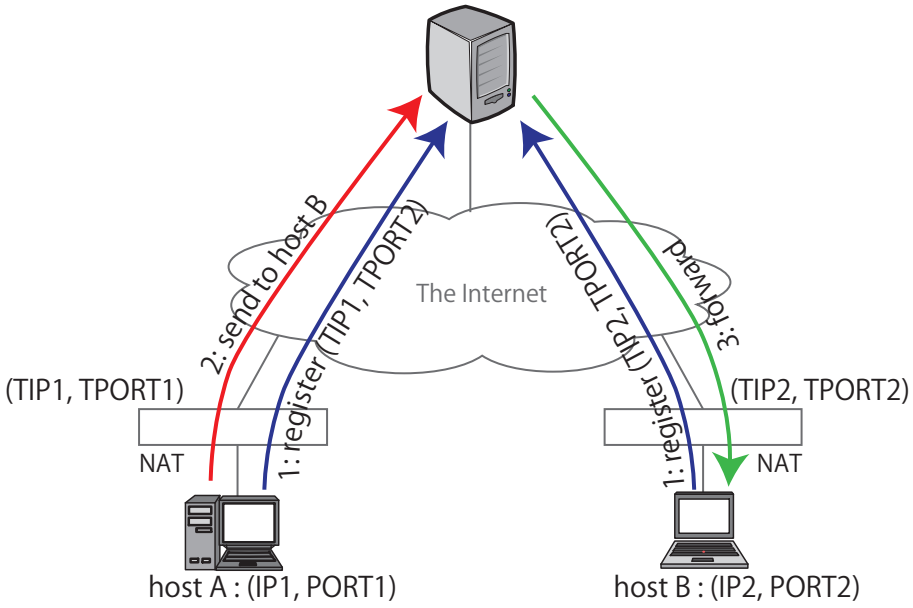


図 5.5　中継方式

この手順 5 と 6 が UDP Hole Punching を行っている箇所となる．手順 5 と 6 は順番が前後する可能性はあるが，最終的な結果は同じとなるため問題ない．

UDP Hole Punching を用いた NAT 越えのプロトコルには STUN がある．STUN では，まずはじめに NAT の種別を判別した後，その種別に応じて UDP Hole Punching を行う．しかしながら，STUN では UDP Hole Punching のみを用いているため，Symmetric NAT の場合には利用できない．

中継方式

Symmetric NAT も越えることの出来る手法に中継方式がある．これはグローバルな接続性を持つ中継サーバを設ける方式であり，通信データはこの中継サーバを経由して配送される．そのため，中継サーバと接続可能な環境であれば，どのような種類の NAT であろうとも越えることが出来る．しかしながら，全てのデータが中継サーバを経由するため，中継サーバに大きな負荷がかかってしまうという問題がある．

図 5.5 は中継サーバを用いて，NAT 下にある 2 つのノード，host A と host B が通信を行っている様子を示している．なお，host A と host B の待ち受け IP アドレスとポー

ト番号は，それぞれ，IP1, PORT1, IP2, PORT2 であり，これらは NAT によって TIP1, TPORT1, TIP2, TPORT2 に変換されるとする．ただし，ここでは，host A と host B は事前に中継サーバのアドレスを知っており，host A から host B へデータを送信するとする．中継サーバを用いた通信の手順は以下のようになる．

1. host A と host B は，仲介サーバに対して自身の待ち受けアドレスを通知する．
2. host A は中継サーバに，host B へのデータを送信する．
3. 中継サーバは host B に，host A からのデータを転送する．

このように行うことで，NAT が存在しても host A から host B へと正しくデータを送信することが出来る．また，host B から host A への通信に関しても同じように行える．中継サーバ方式の NAT 越え手法には TURN [45] が存在する．

P2P ネットワークで NAT 越えを行っている代表的な例として，Skype [16] があり，Skype では NAT 越えのために中継方式を利用している．Skype では，スーパーピアと呼ばれる，一部のグローバルアドレスと多くのリソースを持つノードが中継を行っている [19].

その他の方式と各種方式の比較

その他に，UPnP [17] を用いた手法も存在する．UPnP とはローカルネットワーク内でのサービス発見などを可能にする仕様であるが，UPnP のポートフォワーディングに対応したルータを使用している場合，UPnP を用いてローカルのアドレスと任意のグローバルアドレスをマッピングさせる事が可能となる．これによって，NAT 下にあるノードであっても，グローバルアドレスをもつノードと同等に動作することが可能になる．これは，NAT のポートフォワーディングをアプリケーション側が自動で行うという事に等しい．しかしながら，UPnP は必ずしも全てのルータが対応しているわけではない上に，多段に NAT が使われていた場合には無意味となる．

表 5.1 は，各種 NAT 越え手法の有効性についてまとめた表である．UDP Hole Punching は Cone NAT と多段 NAT に適用でき，対称な通信も実現可能であるが，Symmetric

表 5.1　各種 NAT 越え手法の有効性

	* Cone NAT	Symmetric NAT	多段 NAT	対称通信
Hole Punching	○	×	○	○
中継方式	○	○	○	×
UPnP	○	○	×	○

NAT には適用できない．中継方式では，すべての NAT 種類について適用可能であるが，対称な通信が行えず効率が悪い．UPnP は，Cone NAT と Symmetric NAT に適用でき，対称な通信も可能となるが，多段 NAT の場合には適用できない．

5.2　NAT 介在環境の問題点

P2P ネットワークは，ネットワークに参加しているノード同士がお互いに通信を行ってリソース共有を行う．従って，ほとんどの P2P ネットワークは互いに通信を開始しあう事の出来る状態，すなわち対称な通信が可能であることが大前提とされており，これは，構造化 P2P ネットワークでも例外ではない．しかしながら，現在のインターネットでは NAT が介在するため，任意のノード同士で通信を行うことは必ず出来るわけではない．

今，あるユーザが NAT 越えを考慮されていない構造化 P2P アルゴリズムを用いた P2P ネットワークアプリケーションを利用したいとする．ここで，もし，ユーザの利用している PC が NAT 下に存在した場合，P2P ネットワークアプリケーションを利用するためには，ルータのポートフォワーディング設定を変更する必要がある．しかしながら，ポートフォワーディングの設定はある程度の専門知識を必要とするため，ユーザビリティの観点からして望ましいとは言えない．さらに，複数の P2P アプリケーションが存在した場合に，逐一設定するのはやはりユーザビリティの観点からして望ましいとは言えない．また，そもそもユーザがその権限を持っていない場合もある．

NAT 越えするためのシステムとして，STUN [46] や TURN [45] 等が提案されている．仮に，アプリケーション側がユーザビリティを考慮して STUN や TURN を用いた NAT 越え手法を実装していたとしても，そもそも，STUN や TURN のサーバを誰が運用する

のかという問題が残る．P2P ネットワークは分散された，耐障害性の高いネットワークをアドホックに構築できる手法である．従って，STUN や TURN など，サーバを用いてしまうとそこが単一障害点となり P2P ネットワークの利点を殺してしまう．P2P ネットワークのための NAT 越え提案も存在するが [57]，STUN や TURN を用いているため前述した問題が残る．また，UPnP [17] でポートフォワーディングを自動で行うという方法もあるが，全てのルータが UPnP に対応しているわけではない．また，今後普及するであると考えられる Large Scale NAT [58] などが介在する環境では適用出来ない．

P2P ネットワークを現在のインターネットで活用するために，NAT 問題の考慮は必須事項である．しかし，従来の NAT 越え方式はクライアントサーバ型であり，P2P ネットワークのような分散環境に適さなかった．そこで，本研究では P2P ネットワークに適した分散型の NAT 越え手法である DTUN を提案する．

5.3 Distributed Traversal of UDP through NATs の設計

5.1.2 節では NAT 越えを行う UDP Hole Punching や中継方式などの手法を説明した．しかしながら，これらを用いた標準的なプロトコルである STUN や TURN は，いずれもセントラルなサーバを必要とし，P2P ネットワークアプリケーションには不適当であった．そこで本研究では，分散環境で NAT 越えを実現する DTUN(Distributed Traversal of UDP through NATs) の提案を行った．

提案手法では，グローバルアドレスを持つノードのみからなる DTUN ネットワークを構築し，各ノードは構築した DTUN ネットワークを用いて NAT 越えを実現する．グローバルアドレスを持つノードからなる DTUN ネットワークは分散環境下で NAT 越えを行うための DTUN サービスを提供する．NAT 下にあるノードへの直接通信は DTUN サービスを利用することで可能となる．

DHT の Key-Value Store などを実現するためには，DTUN と別のネットワークを構築する．本論文では，このネットワークのことをサービスネットワークと呼ぶ．DTUN ネットワークと違い，サービスネットワークには，基本的にグローバルアドレスを持つ

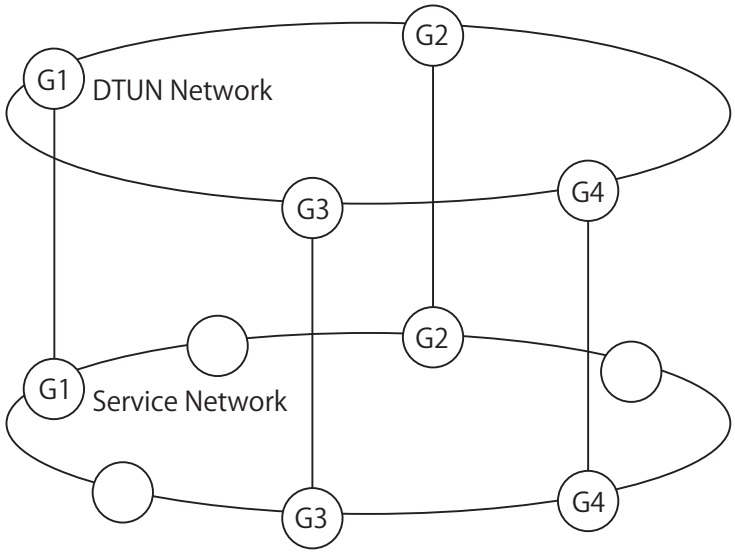


図 5.6　cage 構造による DTUN 及びサービスネットワーク

ノードと NAT 下にあるノードの全てが参加する．なお，ノード同士の直接通信が必要となった場合にも，DTUN サービスを利用して行う事が可能である．

図 5.6 は DTUN ネットワークと，サービスネットワークが構築されている様子を示している．ここでは，ノード G1, G2, G3, G4 がグローバルアドレスを持つノードとなり，これらノードは DTUN 及びサービスネットワークの両方に所属することになる．この二階層ネットワークは鳥かごのように見えるため，本論文ではこれを cage 構造と呼ぶ．

ところで，DTUN ネットワークには，グローバルアドレスを持つノードのみ参加するため，参加するノードはグローバルアドレスを持つか，NAT 下にあるかを判別する必要がある．この判定についても DTUN ネットワークを用いて行うが，詳細については 5.3.2 節で述べる．グローバルアドレスを持つと判別されたノードは，DTUN ネットワークとサービスネットワークの両方に参加し，NAT 下にあると判別されたノードはサービスネットワークのみに参加する．逆に，NAT 下にあるノードは，DTUN ネットワークには参加せず，DTUN ネットワークを利用して NAT 越えの対称通信を行い，サービスネットワークのみに参加する．

このように，基本的に DTUN 方式の NAT 越え手法は，グローバルアドレスを持つノードと持たないノードに分けて行う方法である．本質的に，このコンセプトは，どのよ

うな構造化 P2P アルゴリズムにでも適用可能であるが，本論文ではベースの構造化 P2P に Kademlia を用いた方法について述べる．一般化については 5.6 節で議論を行う．

5.3.1 自ノードの IP アドレスと NAT 判別の問題点

NAT 下に居るかどうかを判別するだけなら，自ノードの IP アドレスがプライベートアドレスであるかどうかを判別するだけで良いように思われるかもしれないが，この方法では判別できない場合がある．これは，アプリケーションレベルで自ノードの IP アドレスを決定することは比較的難しいためである．一般的に自ノードの IP アドレスを決定する手法として，gethostname や gethostbyname，または getaddrinfo を用いる方法がある．しかしながら，必ずしも OS に適切なホスト名が設定されているわけでは無いため，これらの関数は必ずしも正しいアドレスを返すわけではない．

通常，OS のホスト名にはユーザが好きな値をつけて良いため，必ずしも DNS [38, 39] に登録された名前がホスト名として設定されているわけではない．さらに，仮に DNS に登録されたホスト名を利用していたとしても，/etc/hosts ファイルのエントリに，そのホスト名と 127.0.0.1 のループバックアドレスがバインディングされていると，得られるアドレスは 127.0.0.1 となってしまう．実際に，一部 Linux ディストリビューションでは，静的なアドレスを設定した場合，自動的にホスト名とループバックアドレスが/etc/hosts ファイルでバインディングされる．

その他にも，DHCP [26] 等の設定が正しく行われていないと，正しくホスト名からグローバルアドレスに変換できない場合がある．また，マルチホームな環境の場合，正しいホスト名が付いていても実際に利用するアドレスは違っていたり，そもそも実際に利用するアドレスを正しく検出できないといった問題がある．

これらの問題は，ユーザが適切に OS の設定を行えば良く，解決可能な問題でもある．しかしながら，ユーザにとってこれらの設定は，特定のアプリケーションを利用するためだけの特殊な設定である可能性もあり，ユーザに特定の，特殊な操作を強いるような事は，ユーザビリティの観点からもあまり優れているとは言えない．従って，Peer-to-Peer

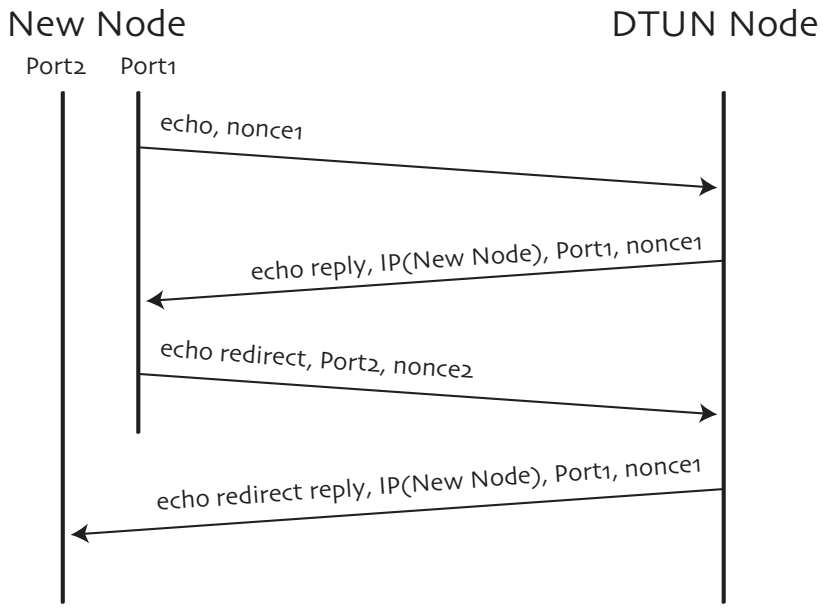


図 5.7　NAT 検出プロトコル

アプリケーションを作成する場合は，その設計時に，gethostname やそれらの関数を用いなくても大丈夫にするべきである．もしも，IP アドレスやポート番号を利用したい場合は，getpeername や recvfrom を利用して得られたものを用いる方が望ましい．

これらの理由から gethostname 等を用いて，自ノードが NAT 下に存在するかどうかを決定することは理想的な方法とは言えない．そこで本研究では，自ノードが NAT 下に居るかどうかを検出するために，DTUN ネットワークを利用した方法を採用した．

5.3.2　DTUN ノードを用いた NAT の検出

新規ノードがサービスネットワークへ参加する際，まず始めに DTUN ノードへとアクセスし，自ノードが NAT 下にあるかそうでないかを判別する．判別する際に用いるプロトコルは図 5.7 のようになる．

まず始めに，新規ノードは DTUN ノードへ echo メッセージを nonce1 と共に送信し，送信した先の DTUN ノードが生存しているか確認を行う．もしも，送信先の DTUN ノードが生存していた場合，DTUN ノードは，echo メッセージ中にあった nonce1 と，recvfrom 関数で得られた IP アドレスとポート番号を含めた echo reply メッセージを新

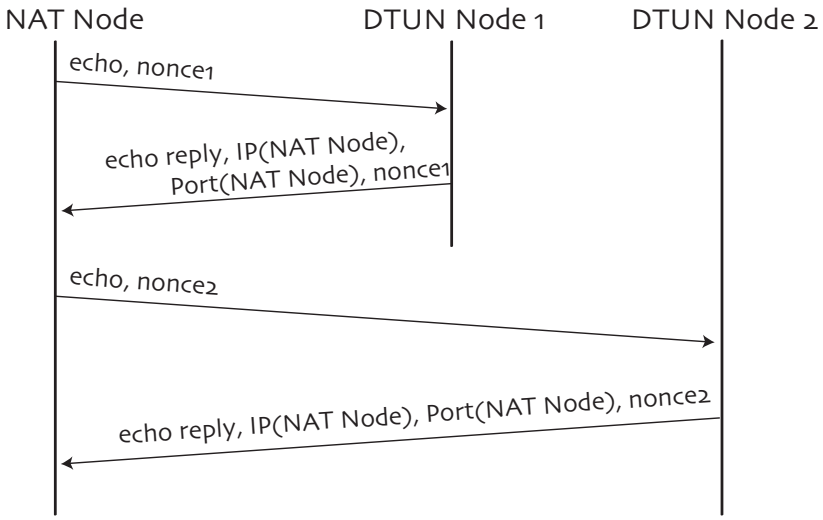


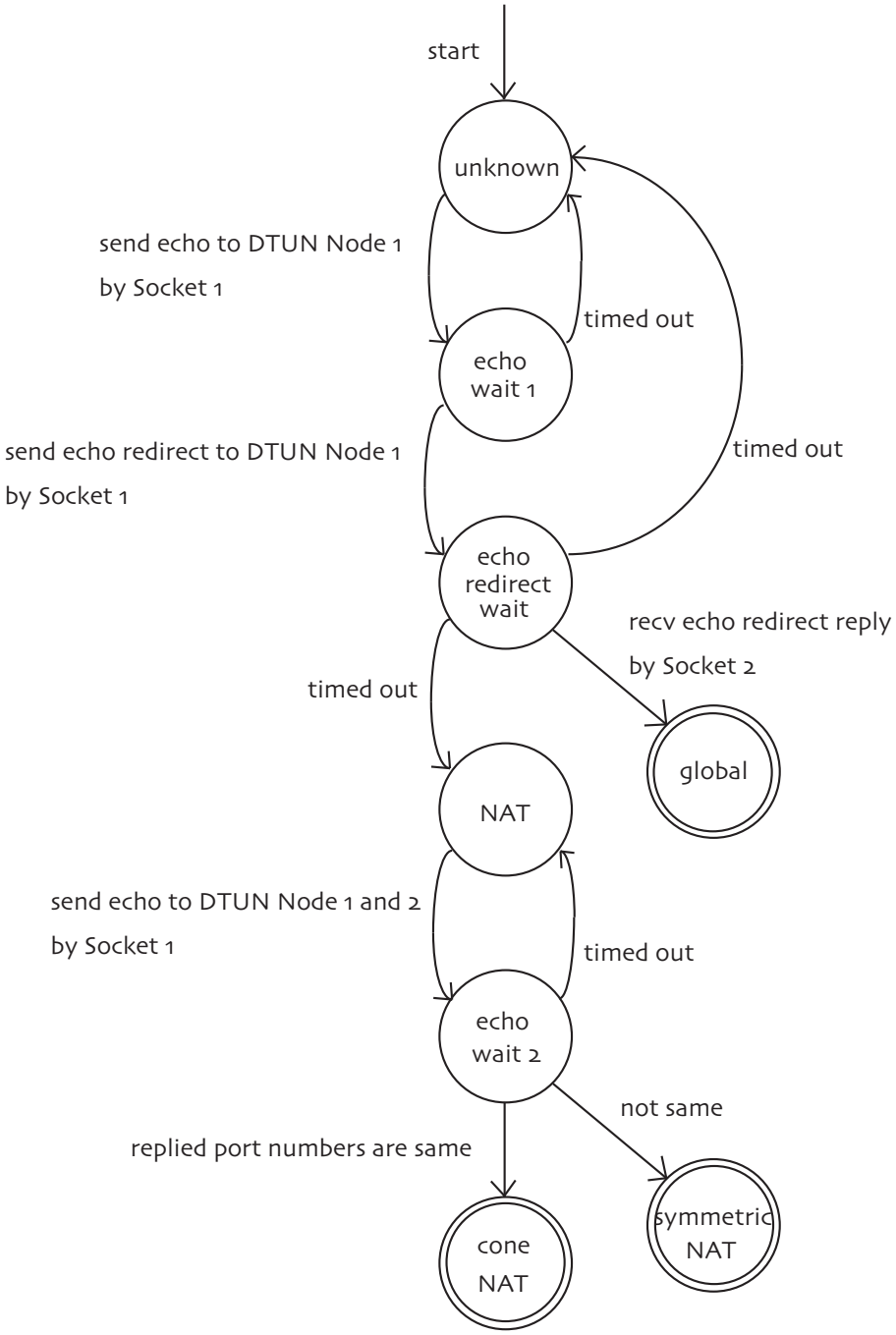
図 5.8 NAT タイプ判定プロトコル

規参加ノードへと返信する．

echo reply メッセージを受け取った新規ノードは，新たに別なポート P_2 を開いた後，echo メッセージを送信したポートと同じポートから，P_2 のポート番号と，nonce2 を含んだ echo redirect メッセージを DTUN ノードへ送信する．echo redirect メッセージを受け取った DTUN ノードは，ポート P_2 へ向けて，同様に，recvfrom 関数で得られた IP アドレスとポート番号を含めた echo redirect reply メッセージを返信する．ただし，この時に返信する宛先ポートは，echo redirect メッセージ中にあるポート P_2 となる．

ここで，もしも新規ノードが NAT 下にあった場合，新たに開いたポート P_2 は何もパケットを送信していないため，echo redirect reply メッセージを受信できないはずである．従って，一定時間内に echo redirect reply メッセージを受信しなかった場合は，新規参加ノードは NAT 下にあると言える．逆に，ポート P_2 で echo redirect reply メッセージを受信できた場合は，新規参加ノードはグローバルアドレスを持っていると言える．

図 5.7 のプロトコルを用いてノードが NAT 下にあると判定された場合，次に図 5.8 のプロトコルを用いて，その NAT が Symmetric NAT であるか Cone NAT であるかを判別する．



actors:

New Node: Socket 1, Socket 2

DTUN Node 1

DTUN Node 2

図 5.9　ノードタイプの状態遷移

5.1 節で述べたように，グローバルアドレス側で利用するポートが違う場合が Symmetric NAT であり，同じ場合が Cone NAT である．従って，グローバルアドレスを持つ 2 つのノードと通信を行えば，Symmetric NAT か Cone NAT かが判別出来ることになる．図 5.8 は，2 つの DTUN ノードにアクセスして NAT 種別の判別を行うプロトコルである．

図 5.8 のプロトコルでは，NAT 下にあるノードは DTUN ノード 1 と 2 に対して，echo メッセージを送信する．echo メッセージを受け取った DTUN ノード 1 と 2 は，NAT ノードに対して echo reply メッセージを返信する．この echo reply メッセージ中には，recvfrom 関数から得られた NAT ノード側のグローバルアドレスとポート番号が含まれている．

このとき，DTUN ノード 1 と 2 が返信したポート番号が同じであれば Cone NAT，違えば Symmetric NAT と判別することが出来る．なお，各 echo，echo reply メッセージ中には nonce が含まれており，NAT ノード側でこの nonce を確認し，正しい echo reply メッセージのみ受信する．

図 5.9 は，図 5.7 と図 5.8 のプロトコルを用いたときの状態遷移を表している．一番はじめの状態は unknown となり，最終的に，global，cone NAT，symmetric NAT のいずれかの状態となる．

このように，NAT の検出には異なるグローバルアドレスを持つ 2 つのノードを利用して行う．そのため，本論文で述べる NAT 越え手法が正しく働くためには，グローバルアドレスを持つノードが 2 つ以上存在し，それらノードが DTUN ネットワークに参加していることが必須となる．

5.3.3 DTUN ネットワークへの参加

先に述べたように，自ノードがグローバルアドレスを持つ場合，新規ノードはまず DTUN ネットワークへ参加し DTUN ノードとなる．DTUN ネットワークは，Kademlia に多少変更を加えたアルゴリズムを利用して構築する．

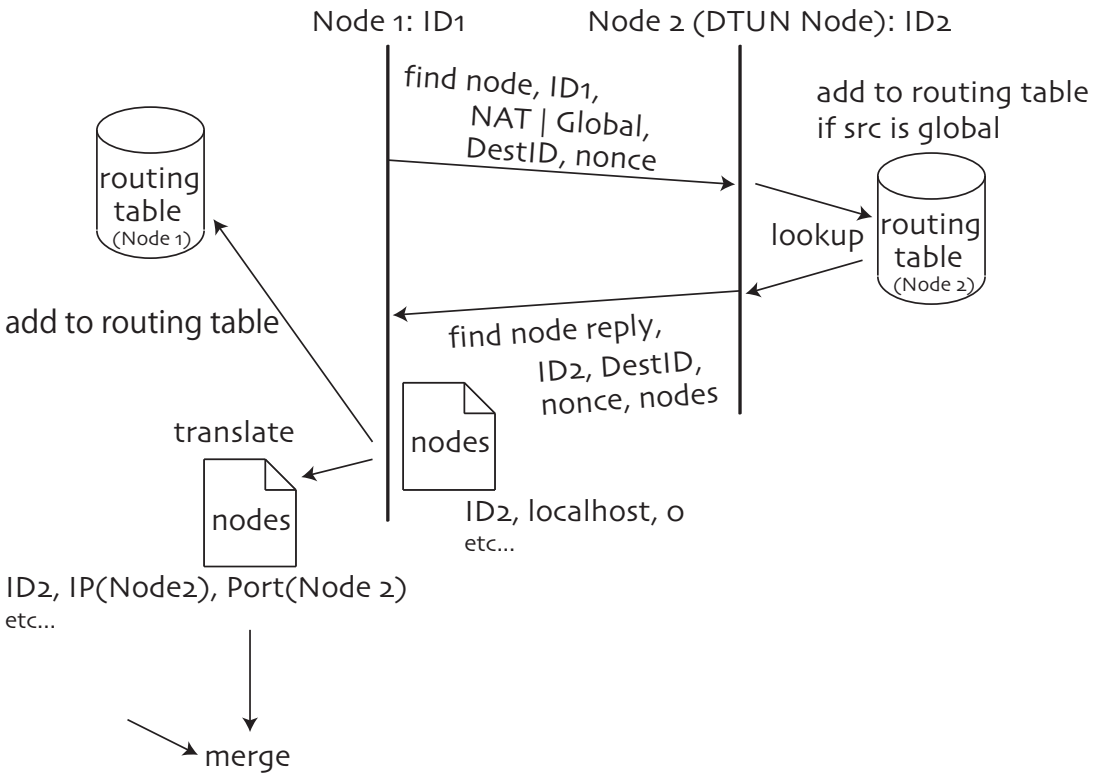


図 5.10　DTUN 上での find node

一方，グローバルアドレスを持たないノードは，DTUN ネットワークに参加しないが，DTUN ネットワークへメッセージを送信するために，ルーティングテーブルの維持のみ行う．ルーティングテーブルの維持のみなので，他ノードからのメッセージ受信やフォワーディングは行わない．

グローバルアドレスを持つ持たないに限らず，Symmetric NAT 下に居るノード以外の全ての新規ノードは，まず DTUN ネットワークへ find node を行う．図 5.10 は，DTUN ネットワークで用いる find node のプロトコルと，メッセージ受信時における動作を表している．ここで，find node を行っているノードを Node 1 とし，find node メッセージを受信するノードを Node 2 とする．

図 5.10 では，まず，Node 1 が Node2 に向けて find node メッセージを送信している．この find node メッセージ中には，Node 1 の ID である ID1，宛先 ID の DestID，nonce と，さらに，Node 1 がグローバルアドレスを持つかどうかを示すフラグも含まれている．

このフラグは，図 5.9 の状態によって決定され，状態が global であるなら，Global のフラグを，状態が NAT，echo wait2 cone NAT であれば NAT のフラグを立てることになる．但し，状態が symmetric NAT であった場合は，5.1.1 節で述べたように，対称な通信を行うことが出来ないため，DTUN ネットワークや，サービスネットワークには直接参加しない．Symmetric NAT の扱いについては 5.3.7 節で述べる．

3.2 節で述べたように，Kademlia では find node など通常のメッセージを受信した際に，ルーティングテーブルの更新を行う．そのため，Node 2 は Node 1 の情報を自身のルーティングテーブルに追加するか決めなければならない．DTUN ネットワークはグローバルアドレスを持つノードのみで構成されるネットワークである．したがって，Node 2 は，find node メッセージ中に含まれる，NAT かグローバルアドレスかのフラグを見て，ルーティングテーブルに追加するかを決める．もしも，Node 1 がグローバルアドレスを持っているのならルーティングテーブルに追加し，NAT 下であるならば追加しない．なお，ルーティングテーブルへの追加処理は，3.2 節で述べたアルゴリズムと全く同じである．

find node を受信した Node 2 は，自身のルーティングテーブルをルックアップし，find node reply メッセージを用いて，その結果を返信する．ただし，ルックアップした結果に自身の情報が含まれていた場合，自身のアドレスを 127.0.0.1:0 とする．これは，5.3.1 節で述べたように，自身の利用しているアドレスを正しく決定することが難しいための措置であり，find node reply メッセージを受信した Node 1 は，127.0.0.1:0 というエントリがあった場合，recvfrom で得られたアドレスに置き換えて処理を続ける．

find node reply メッセージには Node 2 の ID である ID2 と nonce，さらに結果である nodes が含まれている．Node 1 は nonce を確認した後，自身のルーティングテーブルに Node 2 の情報を追加する．find node を行う相手は必ずグローバルアドレスを持つ DTUN ノードであるため，NAT かどうかの情報を得る必要はない．

送信ノードが NAT かそうでないかを区別するフラグを find node メッセージ中に入れておくことで，グローバルアドレスを持つノードのみで構成され，かつ NAT 下からもアクセス可能なネットワークが構成可能となる．

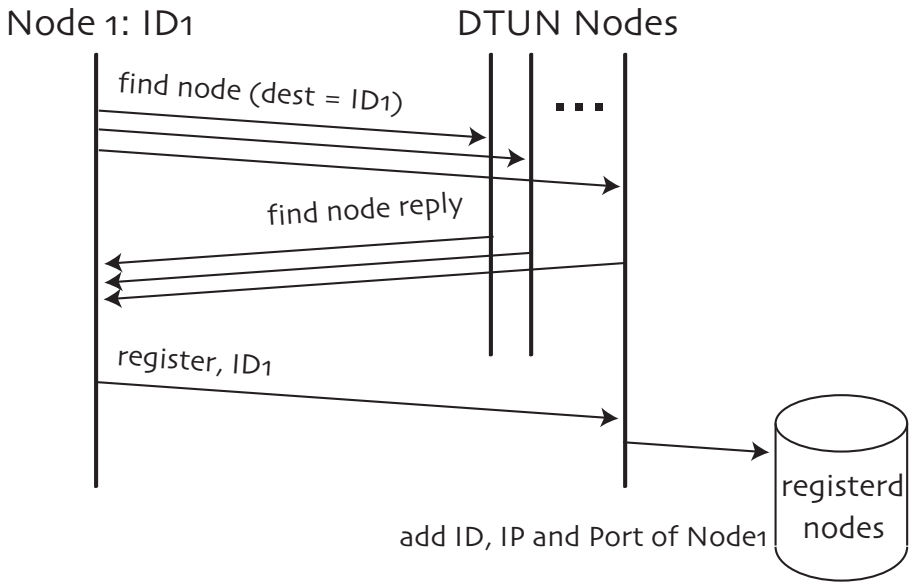


図 5.11　ノード情報の登録

5.3.4　ノード情報の登録と UDP Hole Punching

ノード間の通信は，ノード情報の登録を行う register と，UDP Hole Punching を行う request メッセージを用いて行う．図 5.11 は register を行っている様子を示している．register を行うことで，自ノードの情報，すなわち自ノードの ID，IP アドレス，ポート番号が DTUN ネットワークに登録される．この DTUN ネットワークへの登録は，Kademlia における store とほぼ同じ操作であると考えて良い．

図 5.11 では，Node 1 が自ノードの ID を用いて find node を行っている事が分かる．これは，Kademlia の場合 find node は反復して複数の DTUN ノードへアクセスするためである．正しく find node が終了した場合，ID1 と最も近いノードである DTUN のノードを得ることが出来る．Node 1 は，この得られたノードに対して register メッセージを送信する．register メッセージを受信した DTUN ノードは，Node 1 の ID と，recvfrom 関数から得られた IP アドレス，ポート番号をデータベースへ登録する．

Hole Punching は register を行ったノードに対して request を行うことによって実現する．図 5.12 は request を行っている様子を表している．図 5.12 では，Node 2 が find

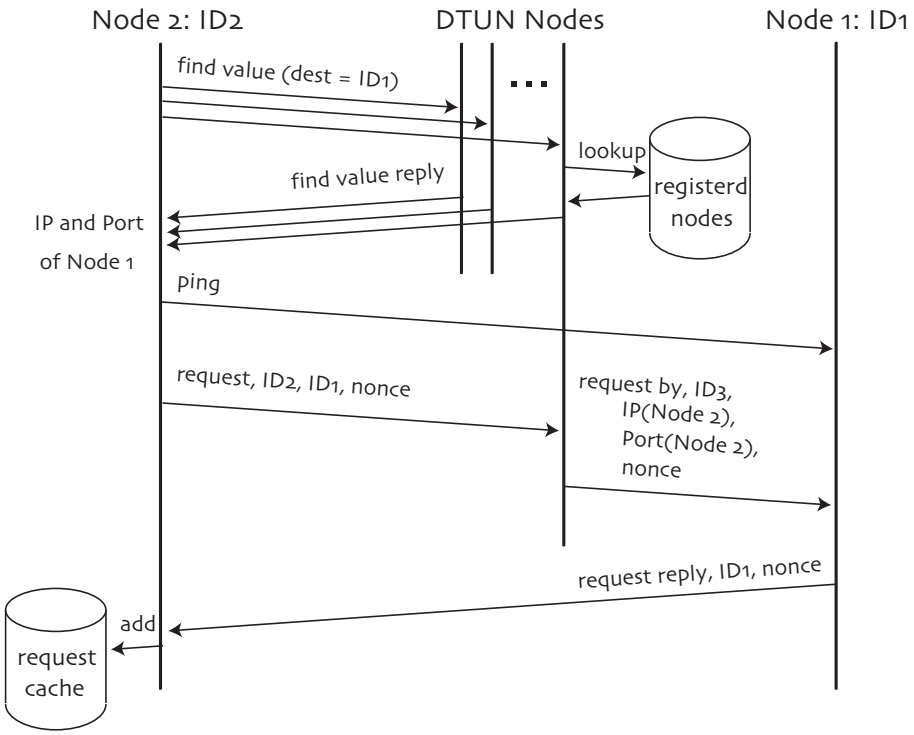


図 5.12 Hole Punching

value メッセージを用いて，Node 1 の IP アドレスとポート番号を取得している．ただし，この Node 1 は図 5.11 のプロトコルを用いて，過去に register を行ったものとする．

Node 1 の IP アドレスとポート番号を取得した Node 2 はまず，ping メッセージを Node 1 へ送信した後，返答を返してきた DTUN ノードに request メッセージを送信する．request メッセージを受信した DTUN ノードは request by メッセージを Node 2 へ送信する．request by メッセージには Node 2 の IP アドレスと Port 番号が入っており，request by メッセージを受信した Node 1 は，request reply メッセージを Node 2 へ返信し，request reply メッセージを受信した Node 2 は Node 1 との疎通がとれたことを確認する．また，これら request メッセージには nonce が含まれており，Node 1 は request reply メッセージの受信時に nonce の確認を行う．

UDP Hole Punching が成功するためには，DTUN ノードから Node 1 へパケットを送信することが出来なければならない．すなわち，Node 1 が NAT 下にいた場合，request by メッセージが送信されるより前に Node 1 から DTUN ノードに，register が行われて

いなければ，DTUN ノードは Node 1 へデータが送信できない可能性がある．佐藤らの報告 [59] によると，ルータのポートマッピングは 300 秒程度でリフレッシュされるとある．従って，register は最低でも 300 秒以内で繰り返し行う必要がある．

5.3.5　サービスネットワークとその構築

提案手法では，DTUN ネットワークを用いた NAT 越え機能を利用して，サービスネットワークを構築する．本手法では，DTUN によって対称通信が可能となるが，サービスネットワークでは，これを利用して，Key-Value Store を実現するための分散ハッシュテーブル（DHT）や，オーバレイマルチキャストなどを実現する．先に述べたように，このサービスネットワークも P2P ネットワークであり，サービスネットワークも Kademlia を利用して P2P ネットワークを構築する．ただし，本提案では構造化 P2P ネットワークの最も基本的なサービスである DHT の構築にのみ言及する．

Kademlia では find node や find value といったメッセージを行う際，複数のノードに対して反復してメッセージを送信するが，送信先のノードが NAT 下にある場合正しくメッセージが配送されない可能性がある．従って，find node やその他のメッセージの送信を行う前に，その find node などのメッセージの送信先ノードに対して request を行う必要がある．本質的に，メッセージの送信先ノードが NAT 下にいるかどうかは，事前に知ることは出来ないため，送信先ノードがグローバルアドレスを持つノードであろうと無かろうと関係なく行う．メッセージの送信は，request メッセージに対する request reply メッセージの受信後，すなわち，対称な通信が可能になった事を確認した後に行われ，request reply メッセージが受信できなかった場合は，メッセージは送信されない．

なお，対称通信でのメッセージ送信は，送信先ノードの ID を宛先として行い，sendto で利用する IP アドレスとポート番号は，request reply メッセージの受信時に recvfrom 関数から得られたものを使う．

5.3.6 リクエストキャッシュ

5.3.5 節では，メッセージを送信する際に，request を行うことによって疎通確認を行うと説明した．しかしながら，毎回リクエストメッセージを送信することは，ネットワークトラフィックの増大に繋がり，レスポンス速度低下の原因ともなる．そこで，トラフィックの増大を防ぐために，一旦 request reply メッセージを受信したら，リクエストキャッシュにその情報，すなわち，ID 対 IP アドレスとポート番号のマッピング情報を保存する．ただし，このキャッシュに追加された情報は一定時間後に失効させる必要がある．

リクエストキャッシュに追加するタイミングは，request reply メッセージを受信したときのみではなく，通常のメッセージ，すなわち DTUN 及びサービスネットワークのメッセージ受信時などにも行う．このようにすることで，無駄に request を行う必要が無くなる．

ところで，一旦リクエストキャッシュに追加されたエントリは，そのエントリが期限切れで失効するまで，上書きされない事に注意する必要がある．これは，悪意のあるノードからのキャッシュポイズニング攻撃を避けるためである．ただし，リクエストキャッシュに既に存在するエントリを追加しようとした場合は，そのエントリの追加時刻のみを更新する．

5.3.7 プロクシモード

ここまで説明してきたプロトコルは，Cone NAT 下にあるノード，もしくはグローバルアドレスを持つノードに対して適用されるプロトコルであった．Symmetric NAT の場合，外部からのデータをパッシブに受信することが難しいため，Symmetric NAT 下にあるノードがサービスネットワークが提供する DHT のルックアップを行う場合，DTUN ノードをプロクシとした代理問い合わせを行う必要がある．本提案では，このような，Symmetric NAT 下にあるノードが DTUN ノードを介したサービスネットワークを利用する方式をプロクシモードと呼ぶ．

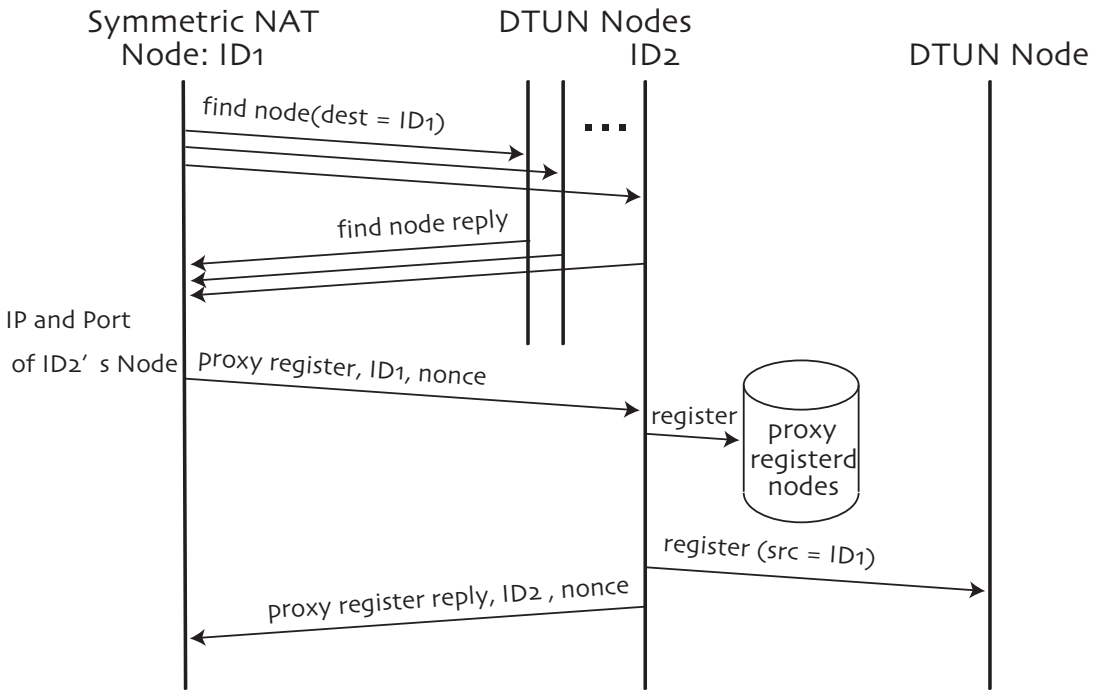


図 5.13　プロクシ登録

DHT のルックアップを行う場合，Symmetric NAT 下にあるノードが DTUN ノードへ proxy find value メッセージを送信する．proxy find value メッセージを受信した DTUN ノードは，代理で find value を行い，得られた結果を Symmetric NAT 下にあるノードに返信する．また，DHT への store も DTUN ノードを介して行う．

DHT のルックアップ等はこのようにして行うことが出来るが，その他のノードから Symmetric NAT 下にあるノードにデータの送信を可能にするため，Symmetric NAT 下にあるノードは，DTUN ノードに対して proxy register メッセージを発行する．図 5.13 は，Symmetric NAT 下にあるノードが proxy register メッセージを発行している様子を表している．proxy register メッセージを受信した DTUN ノードは，図 5.11 で説明した register メッセージを代理で発行する．ただしこのとき，ソース ID に自身の ID を利用せず，Symmetric NAT 下にあるノードの ID を用いる．このようにすると，Symmetric NAT 下にあるノードの ID に対して送信されたメッセージは，全て DTUN ノードが代理として受け取るようになる．DTUN ノードが代理として受け取ったメッセージは転送され，Symmetric NAT 下にあるノードへ配信される．

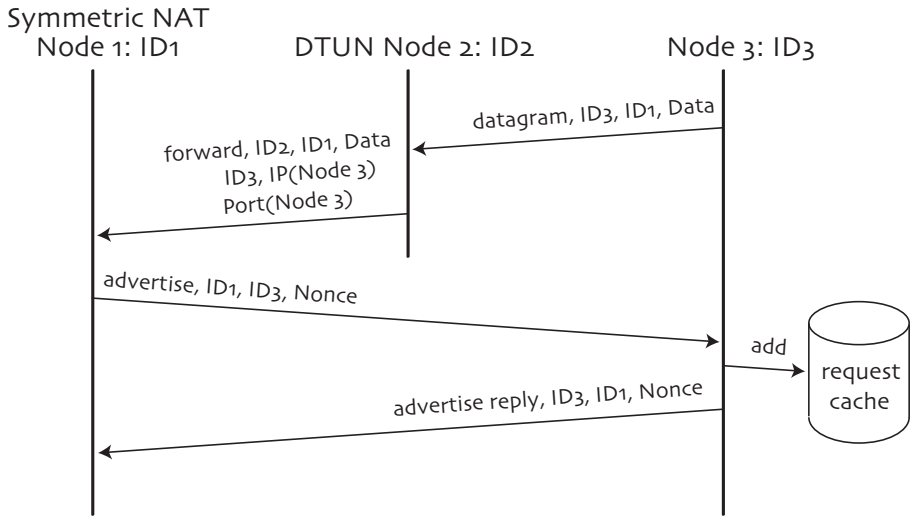


図 5.14 IP アドレスとポート番号の通知

Symmetric NAT 下にあるノードがメッセージの送信元のノードに対して，自身の IP アドレスとポート番号を伝えることによって，DTUN ノードを介さない対称通信による直接通信が可能となる．図 5.14 は，Symmetric NAT 下にあるノードが送信元ノードに対して自身の IP アドレスとポート番号を伝える，advertise メッセージを発行している様子を示している．

通常，メッセージを送信する場合，リクエストキャッシュに登録されている ID 対 IP アドレスとポート番号の情報を元に登録を行う．このリクエストキャッシュは，5.3.6 節で説明したように request メッセージ等を受信した際に更新される．前述したように，Symmetric NAT 下にあるノードは，代理で register メッセージを発行してもらっている．そのため，Symmetric NAT 下にあるノードに向けて送信されたメッセージは，一旦 DTUN ノードへ配信されることになる．

DTUN ノードは Symmetric NAT 下宛のメッセージを受信したら，そのメッセージの送信元ノードの IP アドレスとポート番号をつけて，Symmetric NAT 下にあるノードへ向けて転送しなければならない．直接通信のメッセージを受け取った Symmetric NAT 下にあるノードは，送信元ノードに対して，advertise メッセージを発行する．advertise メッセージを受け取ったノードは，自身のリクエストキャッシュの更新を行う．これに

よって，以降の Symmetric NAT 下にあるノードに対する直接通信メッセージは，DTUN ノードを介さず送信される．

ただし，advertise メッセージを受信できるノードは，グローバルアドレスを持つノードか，Restricted 又は Port-Restricted 以外の Cone NAT 下にあるノードである．従って，Symmetric NAT 下にあるノードは，advertise reply メッセージが受信できなかったノードの情報を保持しておき，以降はそれらノードに対しては，advertise を行わないようにする．

5.4　実装

5.4.1　実証ライブラリ libcage

libcage [9, 60] は，本章で提案した DTUN 方式の NAT 越えを実装した構造化 P2P ネットワーク用のライブラリである．C++ にて実装されており，行数はサンプルコードやテストコードを含めて約 19,000 行である．依存ライブラリは, Boost [5], libevent [10], libcrypto [14] となっている．ソースコードはインターネット上に BSD ライセンスで公開しており，誰でも自由に閲覧・改変が可能となっている．

libcage では，初期化時に IPv4 を利用するか IPv6 を利用するかの設定を行い，IPv4 を利用する場合は DTUN が利用される．DTUN は本章で記述した通りに実装されており，ネットワークの参加時に自動的に NAT の判別と DTUN ネットワークへの参加の判別が行われる．libcage では，サービスネットワークに DHT と，ID に基づいた通信を実装している．DHT は単純な分散型の Key-Value Store を実現している．また，ID に基づいた通信を用いると，TCP の様な End-to-End の通信が行える．TCP と異なるところは，NAT が存在しても通信が行えるという点である．

5.4.2　ノードタイプの決定

libcage では，まず，P2P ネットワーク参加時に，5.3.2 節で述べたプロトコルを用いて NAT の検出と判別を行う．さらに，それと同時に，DTUN ネットワークに対して，自

ノード ID を宛先とした find node を行いネットワークへ参加する．ただし，5.3.3 節および図 5.9 で述べたように，初期のノードタイプは unknown であるため DTUN ネットワークには完全に参加せず，自身のルーティングテーブルに DTUN ノードの情報を追加するのみである．

5.3.2 節では，ノードタイプを決定するために 2 つのノードと通信を行うと説明した．しかしながら，libcage では，ネットワーク参加時に明示的に指定しなければならないのは 1 つのノードのみとしてある．前述したように，参加時にはノードタイプ決定と同時に DTUN ネットワークへの参加も行う．DTUN ネットワークへの参加は 1 つのグローバルアドレスを指定すれば出来る．一方，ノードタイプの決定は 2 つ以上の DTUN ノードの情報が必要であるが，DTUN ネットワークへ参加するときには，2 つ以上のノードとアクセスしているため，DTUN ネットワークのルーティングテーブルから適当にノード情報を取得すれば，ノードタイプを決定できる．

5.4.3 サービスネットワークのルーティングテーブル追加

3.2 節で，Kademlia ではメッセージを受け取ったときにルーティングテーブルの追加を行うと説明した．ところで，全ての DTUN ノードはサービスネットワークに所属しているため，DTUN ネットワークのメッセージを受け取った際にもサービスネットワークのルーティングテーブルにノードの情報を追加しても良いことになる．そこで，libcage では，DTUN ネットワークとサービスネットワークのメッセージを受信したときどちらも，サービスネットワークのルーティングテーブルへとノードの追加を行う．

しかしながら，サービスネットワークへのメッセージ送信は，ノードタイプが決定されるまでは行わない．これは，Symmetric NAT の場合にノードを送信して，サービスネットワークのルーティング情報を乱さないようにするためである．

5.4.4　タイムアウトとタイマー

ノードタイプ決定時のタイムアウト

図 5.9 では，状態遷移の条件にタイムアウトがあるが，libcage では，この全てのタイムアウトは 3 秒と設定してある．したがって，echo redirect wait から NAT へ遷移するためにはタイムアウトが必要であるため，状態が cone NAT あるいは symmetric NAT と決定されるまでには最低でも 3 秒以上の時間が必要になる．

DTUN へのノード情報登録のタイムアウト

提案方式では，DTUN ネットワークへとノード情報を定期的に登録しなければならないが，libcage では，t 秒の間隔で呼び出されるタイマーが定期的に自ノード情報の登録を行う．ただしここで，t は [30, 60) から選ばれた一様乱数となる．一方，登録されたデータは 300 秒経過すると，データベースから削除される．

プロクシへのノード情報登録のタイムアウト

プロクシノードの登録も，DTUN ノードの登録と同じく，t 秒の間隔で呼び出されるタイマーが定期的にプロクシへと登録を行う．同様に，プロクシの情報も 300 秒経過すると，データベースから削除される．

5.5　評価

5.5.1　他ライブラリとの比較

5.1.1 節で述べたように，NAT には幾つか種類があり，P2P アプリケーションを作成した場合は，NAT の種類毎に利用可能かどうかを考慮しなければならない．本節では，libcage と他ライブラリの比較評価を NAT の種類別に行う．

表 5.2 は libcage と他の著名なライブラリの，各種 NAT 下での動作状況を表したものである．Khashmir [8] は Python での Kademlia 実装であり，BitTorrent [3] でも利用さ

表 5.2 各種 NAT 下での動作

	Full Cone	Restricted Cone	Port-Restricted Cone	Symmetric
libcage	○	○	○	△ (中継のみ)
Khashmir	○	×	×	×
Mojito	○	×	×	×
Overlay Weaver	○	×	×	×

れているライブラリである．Mojito [12] は P2P アプリケーションの LimeWire [11] で用いられているライブラリであり，Java での Kademlia 実装である．Overlay Weaver [15] は Java で実装された構造化 P2P ネットワークライブラリであり，ルーティング部分には Kademlia 以外のアルゴリズムも実装されている．

libcage 以外のライブラリでは，Full Cone NAT 以外の NAT 下にノードが存在した場合，正しく通信を行うことが出来ない．ただし，Overlay Weaver は UPnP を用いたポートマッピングに対応しているため，アプリケーション利用者のルータが UPnP に対応している場合のみ，Full Cone NAT 以外でも正しく通信ができる．

libcage では，Symmetric NAT 下にあるノードでも正しく通信を行うことが出来るが，プロクシモードという特殊な扱いを行っている．理想的には，Symmetric NAT 下にあるノードでも，対称に通信を行い DHT ネットワークに参加することが望まれるため，本評価では△とした．

Khashmir をはじめ，構造化 P2P ネットワークのアルゴリズムを素直に実装した場合は Full Cone NAT 以外の NAT 下においては正しく通信を行うことが出来ない．しかしながら，libcage では，あらかじめ NAT を考慮したプロトコルを実装したため，全ての状況において対応が可能となっている．

5.5.2 多段 NAT 下の通信

5.1.2 節で述べたように，多段 NAT の場合，内部から設定を行う UPnP などを用いても NAT を越えて通信を行うことは出来ない．しかし，DTUN では外部観測的に NAT の

種別などの判別を行い，その判定に基づいて NAT 越えを行うため，多段 NAT の場合でも正しく通信を行うことが出来る．

もしも多段にある NAT 箱のうち，すべてが UDP Hole Punching 可能であるならば，各々の NAT 箱でソースアドレスのみに基づいてポートマッピングが行われるため，DTUN では UDP Hole Punching 可と判別される．そのため，多段 NAT 下にあるノードは UDP Hole Punching を用いてサービスネットワークを利用することが出来る．

逆に，一つ以上 UDP Hole Punching 不可の NAT が存在する場合，DTUN では Hole Punching 不可であると判別される．そのため，多段 NAT 下にあるノードは DTUN が提供する中継機構を用いてサービスネットワークを利用することが可能である．これらより，原理的には多段 NAT 下のノードであっても DTUN を用いれば P2P ネットワークサービスを構築することが出来る．

5.5.3　ヘアピンルーティング不可な NAT 下の通信

5.1.1 節で説明したように，NAT の種類によってはヘアピンルーティングが行えない場合がある．しかしながら，本章で説明した NAT 越え手法では，ヘアピンルーティングが不可能な NAT 下にあるノード同士の通信に問題が生じる．なぜなら，DTUN 方式の通信では，お互いの通信にグローバルアドレスを用いるが，ヘアピンルーティング不可の場合，グローバルアドレスを用いたローカルエリアネットワークでの通信は不可能となってしまうからである．

これを解決するには，同一のローカルエリアネットワークに存在するノード同士の通信にはローカルアドレスを用いて行い，他ノードの通信にはグローバルアドレスを用いた通信を行わなければならない．ローカルエリア同士の通信を判別するための方法として，Multicast DNS [24] を用いた方法がある．

Bonjour [4] や Avahi [2] などの Multicast DNS ソフトウェアを利用すると，ローカルエリアネットワーク内でサービス広告・発見が行える．つまり，これを利用すると，ローカルエリアネットワーク内にある他のノードを発見し，ローカルアドレスで通信すること

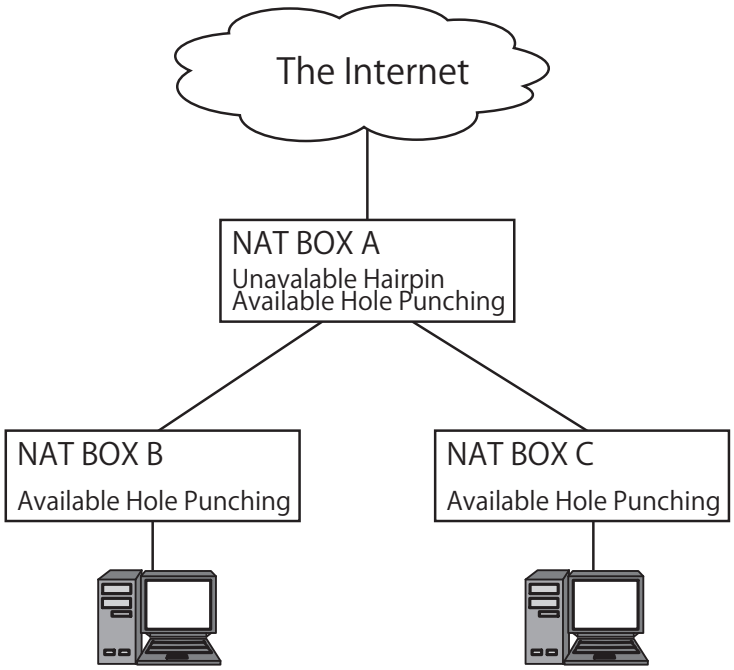


図 5.15 通信不可な多段 NAT の構成

が可能になる．構造化 P2P ネットワークソフトウェアは，Multicast DNS を用いて発見したノードの情報を保持しておき，その情報をルーティング時に利用することでヘアピンルーティングが不可の場合でも，正しく同一ローカルエリアネットワーク内にあるノード同士で通信が行えるようになる．このように，同一ローカルエリアネットワーク内での通信は特に難しくはないため，DTUN ではヘアピンルーティングについては特に対策を行わなかった．ただし，実際のアプリケーション作成者は，ヘアピンルーティング不可の NAT について考慮する必要がある．

ヘアピンルーティングが最も問題となる場合として，多段 NAT の最上段にヘアピンルーティング不可の NAT が存在する場合が挙げられる．図 5.15 は，互いに対称な通信が不可な多段 NAT の構成例を示している．図 5.15 では，全ての NAT 箱が UDP Hole Punching 可であり，最上段にある NAT 箱 A がヘアピンルーティング不可となっている．この例では，NAT 箱 B と C の下にあるホストは，外部観測的には UDP Hole Punching 可であるため，中継方式ではなく，UDP Hole Punching を用いて P2P ネットワークへアクセスする．しかし，NAT 箱 B と C の下にあるホスト同士がグローバルアドレスで通

信を行おうとしても，NAT 箱 A はヘアピンルーティング不可であるため，対称な通信を行うことが出来ない．また，Multicast DNS などを利用しても，両者が存在するのは異なるネットワークであるため，やはり対称な通信を行うことが出来ない．

これを解決する方法は，NAT 箱 B と C によって変換される外部アドレスを用いて通信を行うか，中継方式を用いて通信を行うかである．NAT 箱 B と C の外部アドレスを用いるためには，NAT 箱 A と B，C 間に Hole Punching 用のノードを用意しなければならず，現実的とは言えない．したがって，この場合では，中継方式が最も適していると言える．

ところで，多段 NAT となる場合のほとんどは，最上段の NAT 箱が ISP 等によって提供される NAT 箱の場合がほとんどであると考えられる．ISP 等によって提供される NAT 箱の場合，ヘアピンルーティングが可でありローカルネットワーク間の通信が不可となる [27]．すなわち，多段 NAT で最上段の NAT がヘアピンルーティング不可という，図 5.15 の様な状況は極めて稀であると言える．また，各々の NAT 箱が ISP 等から提供されたものではなく，各自で用意したものならば，ネットワークの設定を変更することも可能である．このような理由から DTUN 方式では，図 5.15 については考慮を行わなかった．

5.6　議論

5.6.1　TCP と UDP の選択

本研究では NAT 越えの手法に UDP Hole Punching を用いた方法を提案したが，実のところ，UDP 通信のみではなく TCP 通信でも Hole Punching は可能である [27]．Ford らの報告によると，380 種の NAT 機器のうち UDP Hole Punching が可能なのは全体の 310 種（82%）であるのに対して，TCP Hole Punching が可能なのは全体の 286 種のうち 184 種（64%）しか無かった [27]．

Hole Punching は対称通信を実現するための技術であるが，Hole Punching が出来ないと中継方式に頼るしか無い．ところが，中継方式は中継ノードのオーバーヘッドが大き

くなり，非効率的である．そのため，NAT を考慮して P2P ネットワークを設計するには，TCP よりも UDP を用いたほうが望ましい．

UDP は信頼性のない通信であるため，パケットロスが生じてしまったり，エンド間の MTU（最大転送サイズ）を超える通信を行った場合に問題が生じる．そのため，UDP を用いる場合には何らかの方法で，データ転送時における再送制御などを行わなければならい．

libcage では簡易なデータの転送制御プロトコルである RDP [55] を用いて，UDP 上で信頼性のある通信を実現している．RDP を用いているのは，DHT でデータを取得・保存する部分であり，パケットロスが生じたり，MTU を超えるサイズのデータであっても，確実に取得・保存が出来る．UDP を用いて通信を行う場合は，P2P ソフトウェア側でなんらかの転送制御を行う必要がある．

5.6.2 他の構造化 P2P ネットワークへの適用

本研究では，DTUN ネットワークを用いて NAT 越えを行う手法を提案した．その実現方法及び実装方法は，Kademlia に依存したものであった．しかしながら，本質的に DTUN ネットワークは，グローバルなアドレスのみから成る構造化 P2P ネットワークのことであるため，Kademlia 以外の構造化 P2P ネットワークに対しても，提案手法を適用することが出来る．

DTUN ネットワークを構成する DTUN ノードが負うべき役割は次の 3 点である．1 つめが，ID を鍵とする IP アドレス及びポート番号を保存するノードであり，2 つめが，Cone NAT 下にあるノードが UDP Hole Punching を行うための待ち合わせノード，3 つめが，Symmetric NAT 下にあるノードのためにデータの中継を行うノードとなる．DTUN ノードに，これら 3 つの役割を持たせることで，NAT が介在する環境でもノード間の直接通信を可能とする．

通常の構造化 P2P ネットワークでは，DHT でデータの保存と取得を行うために，そのネットワークへ所属する必要がある．しかし，DTUN ネットワークにも参加してい

ないノードもデータの取得と保存を行えるようにアルゴリズムを変更する必要がある．DTUN ネットワークを用いて任意のノード間での通信が可能となると，その後は NAT によるフィルタリングを考えずに P2P ネットワークを構築することが出来る．

構造化 P2P ネットワークには，NAT の他にも Churn の問題もある．提案手法を Chord や Symhony などに用いる際は Churn も考慮した設計と実装を行う必要がある．ただし，Churn 下での性能評価については，DHT のアルゴリズムと実装に大きく依存する．そのため，後の 6.3 章では，本研究にて行った実装である libcage の性能評価を行っているが，この結果と他の DHT アルゴリズムを実装した場合の結果が同じになるとは限らない．

5.7　結論

P2P ネットワークは，参加しているノード同士が相互に通信してリソースを共有し合うサービスモデルである．そのため，P2P ネットワークでは，任意のノード同士で相互に通信を開始し合うことが出来る事が必要である．構造化 P2P ネットワークにおいても，全く同じような仮定がされており，構造化 P2P ネットワークが正しく動作するためには，任意のノード同士で相互に通信を開始できる必要がある．

しかし，現在のインターネット環境では，任意のノード同士で相互に通信を行うことは容易ではない．その最も大きな理由として，NAT の存在が挙げられる．NAT が介在する場合，NAT 下に居るノード同士のみでは通信を開始することが出来ない．そのため，グローバルアドレスと持つノードが通信を仲介する必要がある．仲介を行う方法としては UDP Hole Punching を用いた STUN や中継方式を用いた TURN が存在するが，これらはいずれもサーバ型のサービスであり，P2P ネットワークに適しいているとは言えない．

そこで本研究では，分散型の NAT 越え手法である DTUN を提案した．DTUN では，グローバルアドレスを持つノードのみからなる DTUN ネットワークと，DTUN ネットワークを利用して DHT などのサービスを実現するサービスネットワークに分けて，構造化 P2P ネットワークを実現する方式である．

NAT には，大きく分けて UDP Hole Punching 可能な種類の Cone NAT と，Hole Punching 不可な種類の Symmetric NAT が存在する．DTUN では，外部観測的に NAT 種別の判別を行い，UDP Hole Punching 可能な NAT の場合は DTUN ノードを介して直接通信を行いサービスネットワークが提供するサービスを利用する．一方，UDP Hole Punching 不可な NAT の場合は，DTUN ノードがデータの中継を行って，サービスネットワークを利用する．

さらに，NAT が原因となる問題として，多段 NAT の問題もある．1 段の NAT の場合，NAT 箱自体にポートマッピングの設定を行えばインターネットからの接続を受け付けることは可能であった．しかし，多段 NAT の場合，1 段目の NAT 箱にポートマッピングの設定を行っても，それより上位にある NAT 箱の設定を正しく行わなければ，接続を受け付けることは出来ない．Large Scale NAT などが介在する環境では，基本的に多段 NAT となり，インターネットの出口側の NAT 箱はユーザが自由に設定することは困難となる．そのため，P2P ネットワークアプリケーションは多段 NAT 下でも動作可能なように設計する必要がある．DTUN では，外部観測的に NAT と UDP Hole Punching 不可の判別を行い，NAT 下の場合は自動的に NAT 越えを行うため，多段 NAT 下のノードでも正しく P2P ネットワークのための通信を行うことができる．

このように，DTUN を用いることで，NAT や多段 NAT が介在する環境でも Hole Punching や中継方式を用いて，構造化 P2P ネットワークを構築し DHT などのサービスを実現することが出来るようになった．

第 6 章

Churn と大規模環境下の性能

基本的に，P2P ネットワークはエンドノード同士が相互に接続しあってネットワークを形成している．そのため，ルータやスイッチ等から形成されるインターネットなどと違い，P2P ネットワークはエンドユーザの行動に応じて常に変化するという特徴を有する．したがって，P2P ネットワークにはネットワークの変化に対応可能なトポロジ維持や，ルーティングの安定化を行う方法が必須となる．このような，ネットワークの状態が一定の定常状態に収まらずに常に撹拌されている状態は Churn [43] と呼ばれる．

Churn の最適な対策手法は設計や実装によって異なるが，大きく分けてルーティングアルゴリズム部分での Churn 対策と，DHT などサービス部分での対策に分けられる．libcage では，Kademlia のルーティング部分に対する Churn 対策と，DHT 部分での対策の両方を行った．6.1 節では，ルーティング部分に対する対策手法を，6.2 節では，DHT 部分に対する対策手法について述べる．さらに，6.3 節では，Churn 下で 10,000 ノード規模で DHT の値取得に必要となった遅延時間と，値取得の成功確率を測定した．

6.1 トポロジの維持とルーティングの安定化

P2P ネットワークが考慮すべき点は色々あるが，Churn 下における安定性はルーティングの効率化などとともに最も重要な要素であり，実際に構造化 P2P ネットワークを設計・実装する際には Churn 下におけるトポロジ維持や，ルーティングの方法について考

慮する必要がある．そこで本節では，Churn 下における Kademlia のトポロジの維持とルーティングの安定化について議論を行う．

6.1.1 トポロジとルーティングテーブルの維持

3 章で述べたように，Kademlia は基本的に find node などのメッセージを交換する際にルーティングテーブルの更新を行ないトポロジを維持する．これは，メッセージのやりとりが十分行われている場合には有効な方法である．しかしながら，ノードが離脱していき，かつ，メッセージがあまりやり取りされていない場合，ルーティングテーブルに既に離脱したノードの情報が大量に含まれることになる．そのため，正確なトポロジが形成されず，正しくルーティングを行う事が出来なくなってしまう．

そこで本研究では，トポロジ及びルーティングテーブルの維持を行うためのアルゴリズムを提案し，libcage に適用した．アルゴリズム 6.1 は，ルーティングテーブル維持のアルゴリズムである．ただしここで，ID_{mine} は自身の ID であり，*blen* は ID のビット長，inverse 関数はビット反転の関数，sleep 関数は一定時間スリープを行う関数となる．

アルゴリズム 6.1　Kademlia のルーティングテーブル維持アルゴリズム

Require: ID_{mine} is an ID of the node maintaining the routing table. *blen* is the bit length of ID.

1: $n = 1$
2: **while true do**
3: $mask = \mathrm{inverse}(1 \ll (blen - n))$
4: $\mathrm{ID}_{dst} = \mathrm{ID}_{mine} \otimes mask$
5: find_node(ID_{dst})
6: **if** $n == 159$ **then**
7: $n = 1$
8: **end if**
9: sleep()

10: $n = n + 1$

11: **end while**

まず，1 行目では n を 1 で初期化しており，3 行目で n の値に応じて $mask$ を生成している．4 行目では $mask$ と ID_{mine} の論理和が計算され，find node の宛先である ID_{dst} が決定される．すなわち，この ID_{dst} は，ID_{mine} の上から n ビット目が反転された値となり，5 行目で実際に ID_{dst} を宛先として find node を行っている．6 行目の if 文内では，n が 159 に達した場合に初期値に戻しているだけとなる．基本的に，2 行目以降の無限ループ中で find node を行い，ルーティングテーブルの情報を更新している．

ただし，9 行目にある sleep で示されるように，これは一定の間隔をおいて行われる．これは，find node の大量実行による無用なトラフィックの増加を避けるためである．先にも述べたが，基本的に，Kademlia はルーティングテーブルの維持のための特別なプロトコルは必要としないため，この sleep の時間は比較的長くても問題ない．

6.1.2 Churn 下における find node

3.4 節では Kademlia で find node を行う方法について述べた．しかしながら，必ずしも全ての時点において，ルーティングテーブルの一貫性が保たれているわけではないため，この方法では Churn 下で正しく find node を行うことが出来ない．そこで本研究では，クエリのタイムアウト時にルーティングテーブルからタイムアウトしたノードを取り除く方法と，タイムアウトしたノードを記憶しておく方法を用いて，Churn 耐性を Kademlia に付与した．本節では，これらの方法と具体的な改良版のアルゴリズムについて述べる．

タイムアウトしたノード情報の保存

Kademlia では，find node を行う際に複数のノードにアクセスして，宛先と近いノードの情報の取得を行う．しかしながら，Kademlia は基本的に UDP で通信を行うため，各々のルーティングテーブルは必ずしも最新の情報となっているわけではない．そのた

め，既にネットワークから離脱したノードの情報も，ルーティングテーブルに入っている可能性がある．これが Churn 下で find node が正しく行われない大きな理由となっている．

既にネットワークから離脱したノードの情報が，複数ノードのルーティングテーブルに含まれていた場合，find node 時に，そのノードに何度もアクセスを試みてしまう可能性がある．存在しないノードへのアクセスが失敗したことを確認するには，タイムアウトからしか分からない．そのため，何度もタイムアウトが発生し find node のレイテンシが非常に大きくなってしまい非効率的である．

そこで，これを回避するために，メッセージ送信時にタイムアウトが発生した場合は，そのメッセージの送信先を記憶しておき，再度タイムアウトが発生した宛先へメッセージを送信しないようにする必要がある．ただし，一度タイムアウトが発生したとしても，ノードの再参加やネットワークの状態などの理由により，メッセージが到達可能となる可能性もあるので，一定時間後にその情報を失効させる必要がある．

タイムアウトとルーティングテーブル

タイムアウトが発生した情報を保持しておけば，何度も無駄なメッセージを送信しなくても良くなる．しかしながら，このままでは，ルーティングテーブルにはタイムアウトしたノードの情報が残ったままとなってしまう．

基本的にタイムアウトしたノードの情報を持っているだけで，find node は正しく行える．しかしながら，ルーティングテーブルにそのノードの情報が残っているということは，他のノードから find node メッセージが送られてきたときに，その，到達不可能なノードの情報を返答として返してしまい，メッセージを送信したノードは無用なメッセージ送信を行ってしまう．そのため，タイムアウトが発生した場合には，自身のルーティングテーブルからタイムアウトしたノードの情報を削除する必要がある．このようにすることで，他ノードへの誤った情報伝達は行われず，より効率的にルーティングが行える．

Churn 耐性のある find node 操作のアルゴリズム

アルゴリズム 6.2 は Churn 耐性を持たせた find node 操作のアルゴリズムとなる．ここで，$n, \mathrm{ID}_{mine}, \mathrm{ID}_{key}$ はそれぞれ，取得するノードの最大数，自身の ID，宛先の ID であるが，メッセージ送信時に $nodes_{time\_out}$ はタイムアウトした発生したノードの集合を示している．

アルゴリズム 6.2 Churn 耐性を持たせた find node 操作

Require: n is the maximun number of nodes to get. ID_{mine} is an ID of the source. ID_{key} is an ID of the destination. $nodes_{time\_out}$ is a set including nodes, which are not reachable

1: $i = 0$ /* the number of nodes, to which *find node* is being sent */
2: $ids = [\mathrm{ID}_{mine}]$ /* the set of IDs, to which *find node* has been sent */
3: $nodes = k$-buckets.lookup(ID_{key}) /* looking up nodes, which are closest to ID_{key} */
4: sort $nodes$ by the distance from ID_{key}
5:
6: $j = 0$
7: **while** $j < \alpha$ **do**
8: **if** $nodes[j] \in nodes_{time\_out}$ **then**
9: remove $nodes[j]$ from $nodes$
10: remove $nodes[j]$ from the routing table
11: **else**
12: send $find\_node_{key}$ to $nodes[j]$
13: ids.insert($nodes[j]$.id)
14: $i = i + 1$
15: $j = j + 1$

```
16:    end if
17: end while
18:
19: while i > 0 do
20:    if receive nodes_from from ID_from then
21:       nodes.merge(nodes_from)
22:       sort nodes by the distance from ID_key
23:    else if receive time out event of ID_tout then
24:       add ID_tout.id to nodes_time_out
25:       remove ID_tout from nodes
26:       remove ID_tout from the routing table
27:    end if
28:    i = i − 1
29:
30:    k = 0
31:    for each node in nodes do
32:       if k == n then
33:          if i == 0 then
34:             return  nodes[0 : n]
35:          end if
36:          break
37:       end if
38:
39:       if node in nodes_time_out then
40:          continue
41:       end if
42:
```

```
43:       if not node.id in ids then
44:         send find_node_key to node
45:         ids.insert(node.id)
46:         i = i + 1
47:         if i == α then
48:           break
49:         end if
50:       end if
51:
52:       k = k + 1
53:
54:     end for
55: end while
56:
57: return nodes[0 : n]
```

7 行目からの while 文では，α 個までのノードに find node メッセージを送信している．しかしながら，ここで送信する先は $nodes_{time\_out}$ 集合に含まれていないノードのみとなる．9，10 行目では，送信しようとした先のノードの ID が $nodes_{time\_out}$ に含まれていた場合に，$nodes$ とルーティングテーブルからそのノードの情報を削除している．なお，while 文の 12 行目以降は元のアルゴリズムと同じである．

19 行目からの while 文は，find node メッセージに対する応答を受信し，さらに，他のノードへ向けて find node メッセージを反復して送信を行っている．20 から 22 行目でノードの集合を受け取っているのは元のアルゴリズムと同じであるが，ここでは，23 行目から 26 行目に，送信した find node メッセージに対してタイムアウトが発生した場合の処理が追加している．24 行目では，タイムアウトが発生したノードの情報を $nodes_{time\_out}$ に追加し，25, 26 行目で $nodes$ とルーティングテーブルからそのノードを削除している．

31 行目からの for 文は，実際に find node メッセージを送信している箇所であり，ここは基本的に元のアルゴリズムと変わらない．追加したのは，39 から 41 行目の箇所であり，ここでは，送信しようとしているノードがタイムアウトしたノードで無いかを調べている．

6.1.3 実装

ルーティングテーブル維持を行う間隔とクエリの並列化

libcage [9, 60] では，アルゴリズム 6.1 によるルーティングテーブル維持はタイマーが呼び出しており，このタイマーの間隔は 300〜900 秒の一様ランダムな間隔となっている．ただし，アルゴリズム 6.1 では，同時に 2 つの k-buckets に対して更新を行う．すなわち，実際の実装を正確に記すとアルゴリズム 6.3 となる．

アルゴリズム 6.3 実際の Kademlia のルーティングテーブル維持アルゴリズム

Require: ID_{mine} is an ID of the node maintaining the routing table. *blen* is the bit length of ID.

1: $n = 1$
2: **while true do**
3: $mask = \mathrm{inverse}(1 \ll (blen - n))$
4: $\mathrm{ID}_{dst} = \mathrm{ID}_{mine} \otimes mask1$
5: find_node(ID_{dst})
6:
7: $mask = \mathrm{inverse}(1 \ll (blen - n + 1))$
8: $\mathrm{ID}_{dst} = \mathrm{ID}_{mine} \otimes mask$
9: find_node(ID_{dst})
10: **if** $n \geq 159$ **then**
11: $n = 1$

12: **end if**

13: sleep()

14: $n = n + 2$

15: **end while**

3 から 9 行目までで，同時に 2 つの宛先に find node を行っていることが分かる．見ての通り，本質的にはアルゴリズム 6.1 と変わらないが，このように，同時に find node を行う k-buckets を増やすことで，よりトポロジを安定させることが出来る．ただし，増やしすぎると無駄なトラフィックが多くなるので，ここは利用するアプリケーションの性質によって適切に設定する必要がある．

クエリのタイムアウト時間

アルゴリズム 6.2 では，タイムアウトを考慮した find node アルゴリズムを示した．libcage では，このタイムアウト時間は 3 秒としており，3 秒以内に応答が帰ってこなかった場合にタイムアウトの処理を行っている．このタイムアウト時間を長くとると，反応の遅いノードからの応答も確実に取得することができる．しかしながら，離脱したノードが多い場合には，find node が完了するまでの時間が長くなってしまうので注意する必要がある．逆に，タイムアウト時間を短くすると，今度は反応が速いノードからの応答しか得られなくなってしまう．

タイムアウト時間は P2P ネットワークの下に位置する，現実のネットワークに応じて変化させる必要があるが，経験上，現在のインターネット環境では libcage が採用している 3 秒でほとんど問題が無い．

6.1.4 議論

Kademlia やその他の構造化 P2P ネットワークで，ルーティングが正しく働かない理由の最大の理由として，ネットワーク全体でのルーティングテーブルの一貫性が保たれていない事が挙げられる．すなわち，テーブル中に存在すべきノードがテーブル中に存在

しなかったり，既にネットワークから離脱したノードがテーブル中に存在している事が，ルーティングが正しく働かない最大の理由である．

テーブル中に存在すべきノードがない場合は，ノード情報の交換などによって，ルーティング情報をアップデートする必要がある．Kademlia の場合は，通常のメッセージ交換によって新規ノードの追加が行われると説明した．一方，本来テーブル中に存在すべきではないノードが存在してしまうという問題だが，一見すると，これはネットワークの離脱時にノードが離脱する事を他のノードに報告すれば良いように思うかもしれない．しかしながら，この方法は正しく働かない．

ノード離脱時に離脱した旨を伝えることが働かない最大の理由は，ノードの離脱が様々な要因によって発生するからである．P2P ネットワーク的に最も綺麗な離脱の仕方としては，ソフトウェアの正常終了により離脱するということである．ソフトウェアが正常終了した場合は，終了時に離脱する旨を正しく伝達可能であるだろう．しかしながら，ソフトウェアが正しく終了しなかった場合には，離脱する旨を正しく伝えることは出来ない．ソフトウェアの異常終了は様々な要因によって引き起こされる．例えばそれは，ソフトウェア自体が内包するバグであるかも知れないし，OS の異常動作によるものかも知れない．あるいは，メモリ不足などのハードウェア的問題から引き起こされる可能性もある．

離脱する旨の伝えられない，もう一つの要因としてネットワーク的な障害による離脱が挙げられる．ここで言うところのネットワークは，P2P ネットワークではなく，インターネットやローカルエリアなどのネットワークのことであるが，ソフトウェアが動作しているネットワーク自体の障害によって，P2P ネットワークから離脱してしまうと，そもそも離脱する旨を伝えることなど出来はしない．

これらから分かるように，様々な要因により離脱した旨を正しく伝えることが出来ない場合が多々ある．そのため，離脱した旨を伝えることに大きく依存した P2P ネットワークの安定化方法は，正しく働かない可能性が高い．P2P ネットワークの場合，正しくソフトウェアが終了することを期待した設計を行ってはならない．

さらに，離脱する旨を伝えることは，効率の面でも問題があると考えれる．離脱した旨を伝えるのは，ルーティングテーブルに自己の情報を含んでいるノードに対してである．

しかしながら，P2P ネットワークは大規模なネットワークであり，全ノードのルーティングテーブル情報を把握することは不可能である．そのため，離脱した旨を伝達するためには，何らかの方法で，自己のことをルーティングテーブルに保持しているノードを把握しておかなければならない．これを行うには，また別のプロトコルが必要となってしまう上，その情報管理のコストが発生してしまう．

6.2 DHT のデータ再配置と複製

DHT は，分散されたノードに Key-Value ペアを効率的に保存・取得することを可能とする手法である．しかしながら，DHT の実現には構造化 P2P ネットワークが用いられるため，ノードの出入りがある Churn 下では，正しく Key-Value ペアを取得できなくなる場合がある．

ノードの出入りに対しての耐性を持たせるために，保存するデータを再配置する方法と，複製する方法がある．多くの DHT 実装では，この再配置と複製を行っているが，libcage でも再配置と複製によってデータの信頼性を向上させている．そこで，本節では，libcage で採用したデータの再配置と複製の方法について，有効性および効率性の観点から議論する．

6.2.1 表記と定義

表記

本節以下では，以下の表記を用いる．

$$\mathbb{Z} \quad : \quad \text{整数空間}$$

$$h(key) \to \mathrm{ID} \quad : \quad \text{ハッシュ関数 } h \text{ による } key \text{ から ID への写像}$$

定義

DHT を用いると分散化された Key-Value データの保存機構が実現可能となる．データを保存したり取得したりすることを，put や get と呼ぶ．put と get の定義は以下のよう

になる．

定義 6.1　DHT 上に Key-Value データを保存する操作を，put すると言う．

定義 6.2　DHT 上から Key-Value データを取得する操作を，get すると言う．

構造化 P2P ネットワークのノードは各ノードに ID が割り当てられており，ID 間の距離測度が定義されている．本論文では，自身の ID との近い ID を持つノードの集合を近隣ノードと呼ぶ．近隣ノードの定義は以下のようになる．

定義 6.3　自身の ID を ID_m としたとき，ID_m と最も近い ID を持つノードの集合を近隣ノードと呼ぶ．

本論文では，DHT ではデータを put した際に，そのデータを put したノードの事をオリジンノードと呼ぶ．オリジンノードの定義は以下のようになる．

定義 6.4　あるノード N が Key-Value ペア (K, V) を DHT 上に put したとき，ノード N を (K, V) のオリジンノードと呼ぶ．

6.2.2　データの再配置

構造化 P2P では，ある値を Key とすると，その Key から最も近い ID を持つノードの情報（IP アドレスやポート番号など）を取得することが出来る．この特性を利用して，構造化 P2P ネットワークは DHT を実現するのだが，ノードの出入りがあった場合は，データの再配置を行わなければ，正しくデータを取得することが出来ない．

いま，ID 空間を $\mathrm{ID} \in \mathbb{Z}$，距離測度を $D_{sub}(\mathrm{ID}_1, \mathrm{ID}_2) = |\mathrm{ID}_1 - \mathrm{ID}_2|$ と定義した構造化 P2P が存在したと考える．図 6.1 は，この構造化 P2P で Key-Value ペアの保存と再配置を行った例となる．図 6.1 の (a) は，ノードの初期配置となり，初期状態では 4，7，8 の ID を持つノードが存在している．

ここで，Key を k（ただし $h(k) = 6$），Value を v とした Key-Value ペア (k, v) を，この構造化 P2P に挿入したとする．すると，k のハッシュ値である $h(k) = 6$ と一番近い

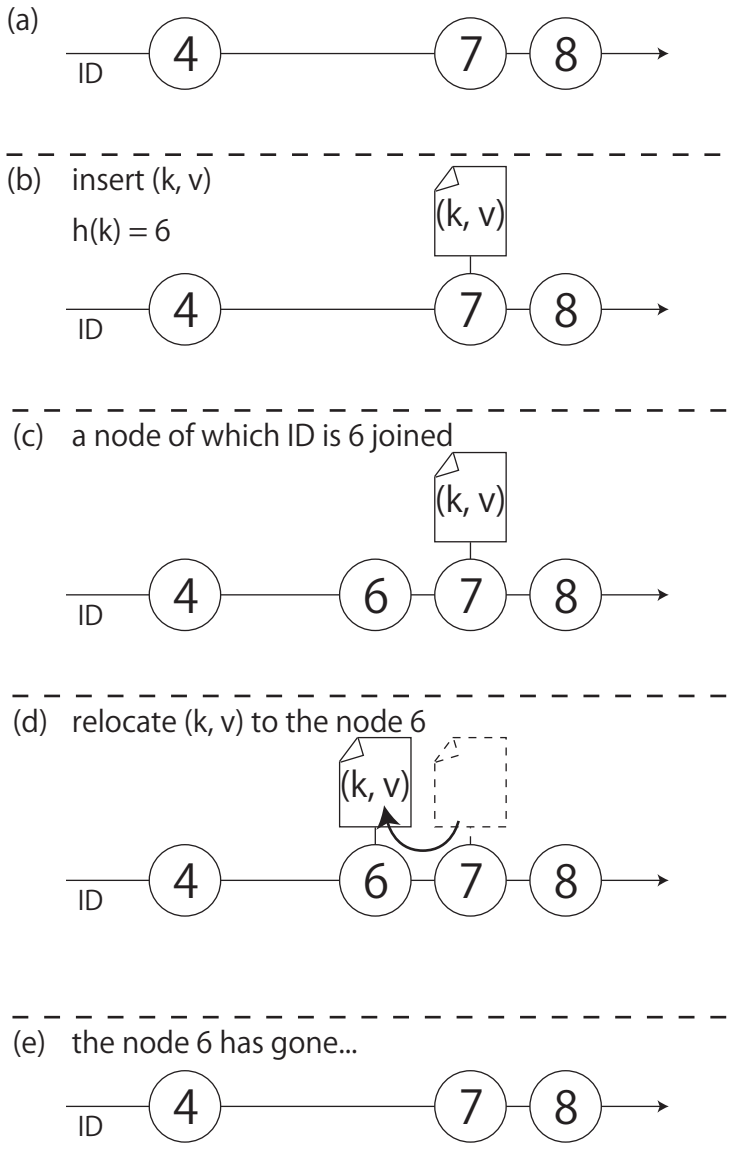


図 6.1 データの再配置

ID は 7 であるため，ID に 7 を持つノードに (k, v) が保存されることになる．この状態を示したのが図 6.1 の (b) となる．図 6.1 の (b) では，6 に最も近いノードを取得するとノード 7 の情報が得られるため，(k, v) データの取得には成功する．

次に，ID に 6 を持つノードが新たに，このネットワークへ参加してきたとする．図 6.1 の (c) は，この時の状態を示している．図 6.1 の (c) では，6 に最も近いノードを取得するとノード 6 の情報が得られてしまう．しかしながら，ノード 6 は，新たに参加したノードであるため，データ (k, v) を保持しておらず，$h(k) = 6$ を検索の Key としたデータの

取得は失敗してしまう．

データの取得に失敗する理由は，ノード 6 が参加する前は，$h(k) = 6$ より最も近いノードはノード 7 であったのに，ノード 6 が参加するとことで，最も近いノードはノード 6 となってしまったためである．そのため，ノード 7 が保持しているデータ (k, v) は，ノード 6 が参加した時点でノード 6 へと再配置される必要がある．この，再配置を行った様子を表しているのが図 6.1 の (d) となる．再配置を行ったあとは，$h(k) = 6$ を Key としてデータを正しく取得することが可能となる．

しかしながら，再配置を行ったとしてもノード 6 が離脱してしまった場合は，そもそもデータを保持しているノードが存在しなくなってしまい，データ (k, v) は，この DHT から無くなってしまう．この様子を表しているのが，図 6.1 の (e) となる．DHT では，データの喪失を防ぐためにデータの複製が行われるが，これについては次の 6.2.3 節で述べる．

6.2.3　データの複製

6.2.2 節では，データの再配置について述べた後，再配置を行ったとしても，ノードの離脱によりデータが取得できなくなる場合があると説明した．ノード離脱が起きてもデータが有効性を維持するためには，データの複製を行うことが有効である．そこで本節では，DHT におけるデータの複製に関して議論を行う．

図 6.2 は，データの複製と再配置を行った例を示している．ただし，ここで用いている構造化 P2P では 6.2.2 節と同じく，ID 空間を $\mathrm{ID} \in \mathbb{Z}$，距離測度を $D_{sub}(\mathrm{ID}_1,\ \mathrm{ID}_2) = |\mathrm{ID}_1 - \mathrm{ID}_2|$ と定義している．

図 6.2 の (a) は，Key を k (ただし $h(k) = 3$)，Value を v とした Key-Value ペア (k, v) を，この DHT に挿入した状態を示している．ただし，初期状態では ID に 2，5，6 をもつノードのみ存在しているとする．いま，Key から最も近い 3 つのノードにデータを複製するとすると，$h(k) = 3$ と近いのはノード 2, 5, 6 であるので，これらノードにデータが複製され置かれることになる．また，データの取得時には，複数のノードに問い合わせを

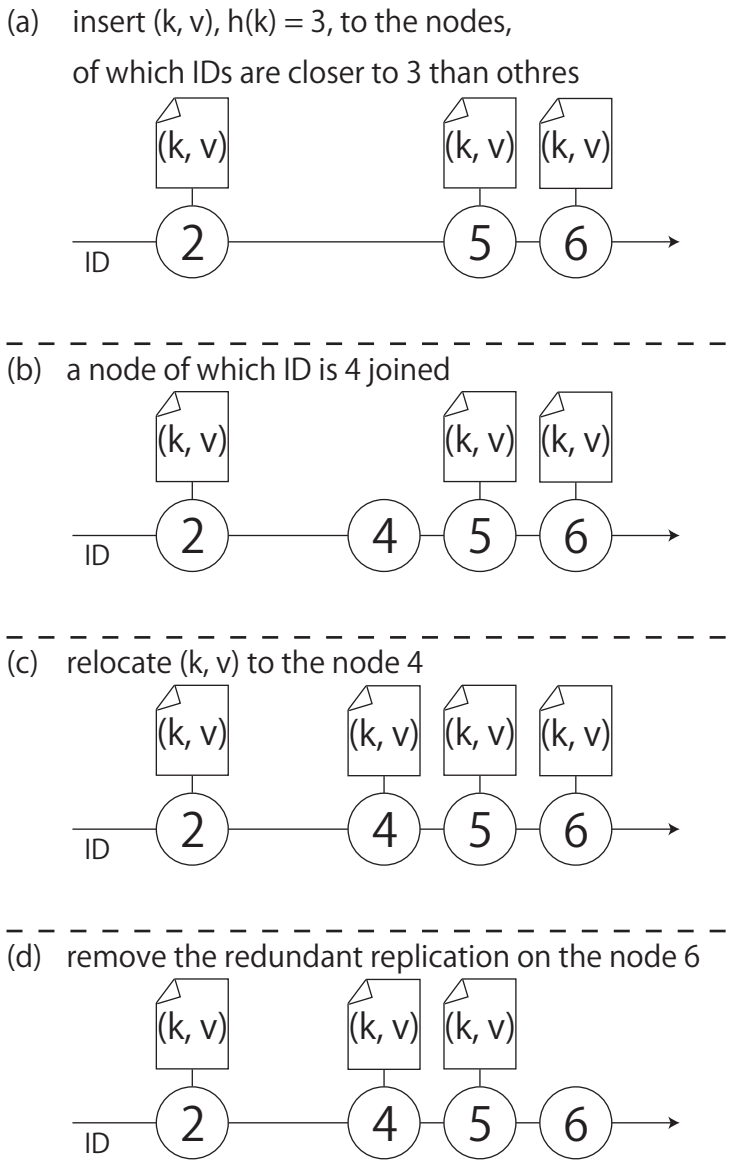


図 6.2 データの複製

行う．

図 6.2 の (b) は，新たに ID に 4 を持つノードが参加した状態を示している．6.2.2 節では，Key との距離が他のノードより近い ID を持つノードが参加すると，データが取得できなくなるという問題があった．ところが，データの複製を行い複数のノードへと問い合わせを行うため，新たにノードが参加したとしてもデータの取得は成功する．

新たなノードが参加したとしても，ある程度はデータ取得が正しく行えるが，ノード参加が非常に多く発生したあとでは，やはりデータを正しく取得できなくなってしまう．

そこで，6.2.2 節と同様に，データを再配置する必要がある．再配置を行った様子を示しているのが，図 6.2 の (c) となる．ここでは，$h(k) = 3$ と最も近い ID を持つノードは，ノード 2，4，5 であるので，ノード 4 にもデータ (k, v) が保存される．なお，ノードの離脱が起きた場合も同様に，データの複製数を維持するために，再配置が行われる．

新たに参加したノードにデータの複製を行ったら，ノード 6 はデータを保持する必要は無くなる．何故なら，Key と最も近い 3 つのノードに複製を保存しておくというのがルールであったためである．そこで，図 6.2 の (d) で示されるように，冗長なデータはノード 6 から削除される．

このように，常に複数のデータを DHT 上に保存しておくことで，ノードの参加・離脱に対する耐性を付与することが可能となる．

6.2.4　再配置の戦略

6.2.2 と 6.2.3 節では，データの再配置が必要な理由を述べた．再配置を実現する方法として，put したノードが再 put する方法と，データを保持しているノードが再 put する方法があるが，本節ではこれら二種類の方法について議論を行う．

データのオリジンノードによる再 put

データのオリジンノードによる再 put を以下のように定義する．

定義 6.5　ノード N がデータ $(k, v)_N$ のオリジンノードであったとき，ノード N が定期的にデータ $(k, v)_N$ を put して行うデータの再配置方法を，オリジンノードによる再 put と呼ぶ．

オリジンノードによる再 put の利点

DHT でデータを put する理由としては，自身の持っている何らかの情報を広告するためである場合が多いと考えられる．例えば，他のノードが発見できるように，自分の持っているファイルや，自分の名前などを広告するといったことは，DHT の利用のされ方と

しては典型的な例であるといえる．

いま，オリジンノードがデータを put するのは，オリジンノードに何らかの利益があるからであると仮定する．すると，データのオリジンノードが，その put したデータの可用性を保証することは理にかなっている方法と言える．なぜなら，put したデータが他のノードから正しく入手可能であることと，オリジンノードの利益とが直結するからである．

オリジンノードによる再 put の問題点

再 put を行うためには，構造化 P2P のルーティングを行ってデータを put しなければならない．しかしながら，構造化 P2P のルーティングを頻繁に行うと，ネットーワーク帯域を大きく消費してしまう．

さらに言うと，オリジンノードによる再 put は，再配置に係るトラフィック削減の為の最適化を行なうことが難しい．なぜなら，オリジンノードとデータを保存すべきノードの位置には相関が無く，オリジンノードと put 先ノードの位置は，構造化 P2P のネットーワーク的に距離が離れている場合が多いと考えられるため，再 put を行うには構造化 P2P のルーティングが必須となるからである．

データの保持ノードによる再 put

データの保持ノードによる再 put を以下のように定義する．

定義 6.6 Key-Value ペア (k, v), $h(k) = h_k$ というデータがあったとき，そのデータを保持しているノードの集合を $\mathcal{N}_{h_k}$ とする．このとき，$\mathcal{N}_{h_k}$ に含まれるノードがデータの再配置のために再 put を行う方法を，データの保持ノードによる再 put と呼ぶ．ただし，$\mathcal{N}_{h_k}$ の初期値はオリジンノードがデータを put する際に put 先として選ばれたノードの集合であり，それぞれのノードは h_k と近い ID を持つ．

定義から分かるとおり，すなわち，データの保持ノードによる再 put とは，$\mathcal{N}_{h_k}$ に含まれるノード自身が，$\mathcal{N}_{h_k}$ の集合を操作（追加や削除）することである．

保持ノードによる再 put を行う際は，構造化 P2P のルーティングを用いて put を行う方法と近隣ノードの情報を元にして行う方法が考えられる．構造化 P2P のルーティングを行う方法での再 put は，オリジンノードによる再 put と同様にトラフィックの増加を引き起こしてしまう．しかしながら，近隣ノードの情報を元に行う方法では，トラフィックの増加を引き起こさずに，効率的に再 put 出来る可能性がある．なぜなら，構造化 P2P では近隣ノードの情報をより詳細に持って居る場合が多く，さらに，特定の構造化 P2P アルゴリズムではルーティングテーブルの維持と共に，近隣ノードの情報も更新するためである．

6.2.5　再 put のアルゴリズム

単純な再 put アルゴリズム

まずはじめに，最も単純な再 put アルゴリズムを以下に示す．

アルゴリズム 6.4　最も単純な再 put

Require: *data* is a set of key-value pairs, which should be re-put.

1: **for** (k, v) **each** *data* **do**
2:　　put(k, v)
3: **end for**

アルゴリズム 6.4 の *data* は，再 put すべき Key-Value ペアの集合となる．また，4 行目にある put() 関数は，構造化 P2P のルーティングを行ってデータの put を行う関数となる．

この再 put アルゴリズムは最も単純な方法であり，オリジンノードによる再 put とデータの保持ノードによる再 put の両方で利用可能である．しかしながら，6.2.4 節で述べたように，全てのデータを構造化 P2P のルーティングを行って put を行うのは効率が悪い．

近隣ノードベースの再 put アルゴリズム

Kademlia [37] や Pastry [49] など，いくつかの構造化 P2P アルゴリズムはルーティングテーブルに自身の ID と近い，近隣ノードの情報を保持する．この近隣ノードの情報を用いると，データの再配置に係るコストを抑えることが可能となる．

近隣ノードの情報を用いた再 put のアルゴリズムは次のようになる．

アルゴリズム 6.5　近隣ノードベースの再 put

Require: *data* is a set of key-value pairs, which should be re-put.

1: update information about *neighbors*
2: **for** (k, v) **each** *data* **do**
3: find *nodes* which are close to $h(k)$ than others from the *neighbors* and *me*
4: **if** *me* **in** the *nodes* **then**
5: **for** *node* **in** *nodes* **do**
6: **if** not (k, v) **in** $nodes\_sent_{(k,v)}$ **then**
7: send (k, v) to the *node*
8: insert the *node* into $nodes\_sent_{(k,v)}$
9: **end if**
10: **end for**
11: **else**
12: remove (k, v) from *data*
13: remove $nodes\_sent_{(k,v)}$
14: **end if**
15: **end for**

アルゴリズム 6.5 は，1 行目で，まずはじめに，近隣ノードの情報をアップデートを行う．その後，再 put すべき全てのデータに対して，近隣ノードおよび自ノードの集合から $h(k)$ と近い ID を持つノードを検索する（3 行目）．もし，$h(k)$ と近いノードに自身が含

まれていたなら，6 行目でそれらのノードに Key-Value ペア (k, v) を送信する．すなわちこれは，再配置を行うと同時に，複製を行っていることになる．逆に自身が含まれていなかったら，図 6.2 の (c) と (d) で示した冗長な複製を保持しているとみなし，12 行目で $data$ から (k, v) を削除している．

なお，$nodes\_sent_{(k,v)}$ は，同じデータを何度も同じノードに送信するのを防ぐ為に使われる集合となる．Key-Value ペア (k, v) を $node$ へ送信した後，8 行目で，その (k, v) を $node$ へ送信したことを記録しておく．このようにすることで，さらに冗長なトラフィックを削減することが可能となる．

アルゴリズムから明らかなように，この方法は，データの保持ノードによる再 put にのみ適用可能であり，オリジンノードの再 put には適用できない．

6.2.6　複数ノードへの put

6.2.3 節ではデータの複製を行う利点について述べたが，データの put 時に複製を生成する方法として，宛先ノードによる方法と，オリジンノードによる方法の 2 種類が考えられる．そこで，本節では，これら 2 種類の複数 put 方法について議論を行う．

6.2.7　データの宛先ノードによる複数 put

図 6.3 は，データの宛先ノードによる複数 put が行われている様子を示している．オリジンノードが Key-Value ペア (k, v), $h(k) = 4$ を put したとすると，このとき，$h(k) = 4$ と最も近いノードは ID に 4 をもつノードとなる．そのため，まず，データはノード 4 に送信される．その後，データを受け取ったノード 4 は，近隣のノードへとデータの再 put を行う．この再 put は，アルゴリズム 6.5 を用いると，効率的に行うことができる．

これは，近隣ノードの情報を用いて put を行う方法であるため，Pastry や Kademlia など，近隣ノードの情報を持つ構造化 P2P アルゴリズムならば，効率的な複数 put が可能となる．

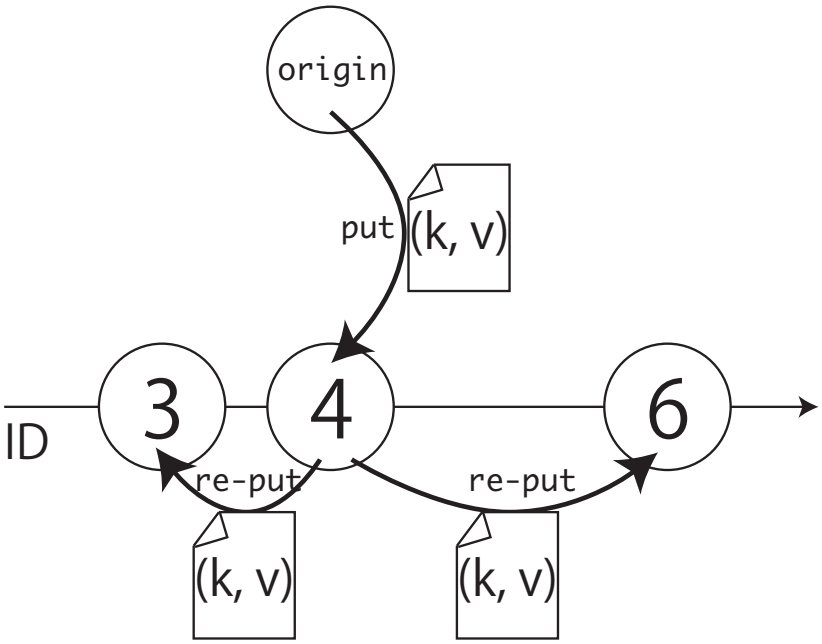


図 6.3　宛先ノードによる複数 put

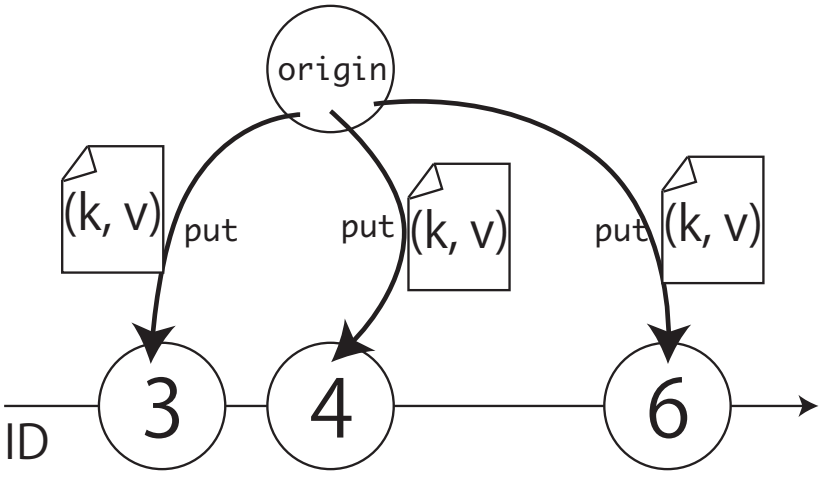


図 6.4　オリジンノードによる複数 put

オリジンノードによる複数 put

図 6.4 は，オリジンノードによる複数 put が行われている様子を示している．ここでは，図 6.3 の時と同じく，オリジンノードが Key-Value ペア (k, v), $h(k) = 4$ を put しているが，put する先はが $h(k) = 4$ と近い ID を持つ複数のノードとなっている点が大き

く異なっている．

オリジンノードによる複数 put の問題点は，Key から近い複数のノード情報を取得しなければならない点と，実質，反復ルーティングを用いた構造化 P2P にしか適用できない点にある．ところが，Kademlia では，find node を行うと Key と最も近い複数ノードの情報が得られる．そのため，find node を用いて複数 put を行う場合には宛先ノードによる複数 put よりも，オリジンノードによる複数 put の方がより効率的である．

しかしながら，基本的には，宛先ノードによる複数 put の方が多くの構造化 P2P アルゴリズムに適用が容易であると考えられる．複数 put を効率的に行うには，その構造化 P2P アルゴリズムに応じて，宛先ノードによる複数 put と，オリジンノードによる複数 put のどちらを用いるかを考慮しなければならない．

6.2.8　実装

libcage [9, 60] は，Kademlia をベースとした構造化 P2P ネットワーク・DHT ライブラリである．そのため，Key-Value ペア (k, v) の put を行うためには，まずオリジンノードが $h(k)$ を計算し，$h(k)$ を宛先として find node を行う必要がある．find node を行うと，$h(k)$ と近い ID を持つ複数のノードの情報が得られるが，それを元に $h(k)$ と近いノードに対してデータを送信する．これは，6.2.7 節で述べたオリジンノードによる put に相当する．送信するノードの数，すなわちデータの複製数は，デフォルトでは 10 としてある．

再 put は，オリジンノードによる方法と，宛先ノードによる方法の両方を併用している．ただし，オリジンノードによる再 put は 6.2.4 節で述べたようにネットワークトラフィックの消費が大きいため，初めに put を行ってから 3 回までしか行わない．一方，宛先ノードによる再 put はデータの有効期限が切れるまで行われる．なお，宛先ノードによる再 put には，6.2.5 節で示した近隣ノードベースの再 put アルゴリズムを用いている．

データの再 put は，10 分から 20 分の間の一様ランダムな間隔で行っている．これは，等間隔で再 put を行った場合における，トラフィックのバースト的な発生を避けるためで

ある．各ノードの再 put は，10 分から 20 分の間隔を開けて行われ，また，複製数が 10 であるため，全体では平均 1.5 分に 1 回の再 put が行われる計算となる．

6.2.9 議論

DHT における再配置の問題は，新規ノードの参加とノードの離脱時に発生する．ノード参加時の再配置は定期的な再 put によって行う必要があるが，ノード離脱時の再配置は離脱するノードが再 put を行えば良いように思われる．しかしながら，6.1.4 節で述べたように，ノードの離脱が常にソフトウェアの正常終了によって引き起こされるわけではない．そのため，ノードの離脱時の終了動作に依存した再配置は，6.1.4 節で言及したのと同じ問題を引き起こす．

ただし，全てのノードが終了時に再配置を行わなくても，一部のノードが終了時に再配置を行うだけで，効果的な再配置が行われる可能性がある．ノード離脱を通知するためには，自身の情報がどのノードによって保持されているかを知る必要があり，P2P ネットワーク全体の情報を知る必要があったが，再配置は自身が保持している情報のみで行えるため効率の悪化にはつながらない．

6.3 評価

libcage [9, 60] は，Kademlia をベースに用いた構造化 P2P ネットワーク及び，DHT ライブラリであり，本論文で提案したトポロジ維持とルーティングの安定化方法，DTUN 方式による NAT 越え手法，DHT データの再配置と複製方法を実装している．は，libcage の性能評価を目的として，10,000 ノード規模での実験を行った．本節ではその実験結果について記述する．

本評価では，主に，DHT のデータ取得に的を絞っている．これは，ユーザが最も関心のあるのは，DHT によるデータ取得がどの程度安定して行えるかであると考えられるためである．DHT の性能評価は定常状態及び，Churn 下における安定性の議論をおこなう．DHT は P2P ネットワークを構築するアルゴリズムであり，Churn 下での安定動作

が強く求められる．

本実験は，情報通信研究機構北陸リサーチセンターが有する StarBED [61] の，Group-G を用いて行われた．Group-G は Opteron 146HE(2GHz)，メモリ 4GB の PC150 台からなる PC クラスタである．本評価では，PC1 台に付き最大 100 ノードまで起動させ，合計 10,000 ノードでの実験を行った．なお，実験に用いた OS は FreeBSD 6.2 である．

6.3.1　DHT での値取得時の待ち時間

本節では，DHT で値を取得する際の遅延時間について議論を行う．待ち時間の計測は，NAT が介在しない場合とする場合において，ノードの出入りがない定常状態と出入りのある Churn 下で計測を行った．NAT が介在しないときの計測では，DTUN を利用しない．また，NAT が介在する Churn 下での待ち時間計測を，データのレプリケーション数 r と find value 時の同時問い合わせ数 α を変更して計測を行った．また，特に説明がない場合におけるパラメータの値は $r = 10, \alpha = 3$ である．表 6.2 は本節で説明するグラフの，80 と 95 パーセンタイル値を表している．

NAT が介在しない場合

本節では，NAT が介在しない場合における待ち時間について説明する．図 6.5 は，ノードの出入りが無い定常状態において，ノード数を 100〜10,000 まで変化させた時に待ち時間の計測を行った結果である．図 6.5 は，縦軸が累積確率密度，横軸が待ち時間となっており，待ち時間の累積分布関数（CDF）を表している．ノード数が 100 または 1,000 の時は，待ち時間の差はあまり見られなかったが，ノード数が 10,000 となると，全体的に約 0.5[ms] ほど待ち時間が大きくなった．表 6.2 のパーセンタイル値を見ると，その値は，どのノード数でも数ミリ秒以内となっており，ほぼ全ての応答は数ミリ秒以内に返ってきていることが分かる．

図 6.6 はノードの出入りがある，Churn 下での待ち時間の計測を行った結果である．既存の P2P ネットワークのノードの生存時間は，平均 5,000[s] の指数分布に従うことが知られている [28]．そこで本評価では，より条件を厳しくするため，ノードの生存時間を平

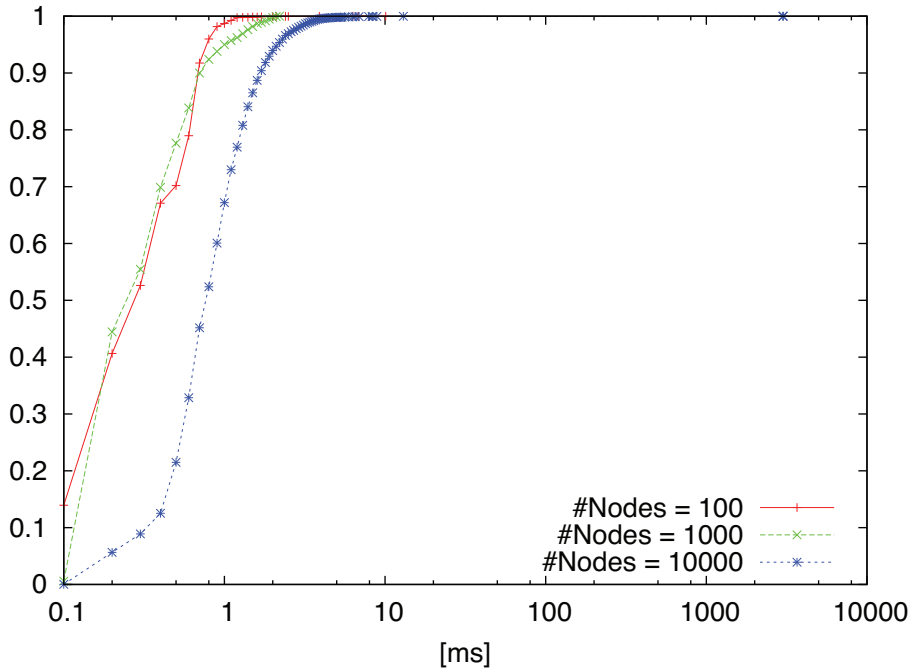


図 6.5 待ち時間 (CDF) - NAT 無し，定常状態，$r = 10, \alpha = 3$

均 500[s] の指数分布に従うように設定した．ただし，一度ノードを離脱させたあとすぐにネットワークに参加するようにしたため，全体のノード数に変化はない．

図 6.5 と図 6.6 を比較すると，100 ノードの場合はあまり変化はみられない．しなしながら，1,000 と 10,000 ノードの場合は，Churn 下の方が全体的に待ち時間が大きくなっていることがわかる．これは，Churn 下では Kademlia のルーティングテーブルが正しく維持されないために起こると考えられる．表 6.2 によると，80 パーセンタイル値の変化はあまり見られないが，95 パーセンタイル値が 1,000 ノードの時で約 3[s]，10,000 ノードの時で約 6[s] と大幅に大きくなっている．

Kademlia は UDP で実装されることを想定されたアルゴリズムである．UDP はコネクションレスな通信プロトコルであるため，ネットワーク上に既に存在しないノードの情報をルーティングテーブルに保持し続けてしまう．その結果，find value または find node 時に，存在しないノードへもリクエストを送信してしまい，そのリクエストのタイムアウトを待たなければならなくなる．このタイムアウトが発生すると，結果的に待ち時間も大きくなると考えられる．このメッセージのタイムアウトだが，6.1.3 節で述べたよ

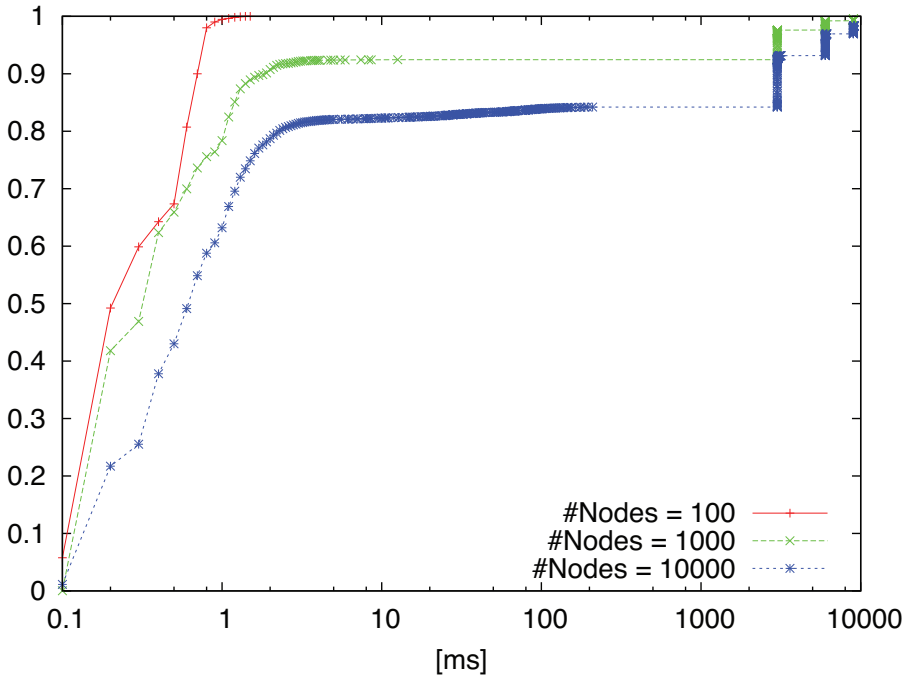


図 6.6　待ち時間 (CDF) - NAT 無し，Churn 下, $r = 10, \alpha = 3$

うに libcage では 3[s] として実装している．

NAT が介在する場合

本節では，NAT が介在する場合の待ち時間について議論を行う．本実験では，NAT 下に存在するノードに対するパケットは ipfw を用いて Port-Restricted Cone NAT と同等のパケットフィルタリングを行った．図 6.7 は DTUN を利用したときに，ノードの出入りがない定常状態において DHT の値を取得したときの待ち時間を CDF で表したグラフであり，縦軸が確率密度，横軸が待ち時間となっている．Casado らの研究 [21] によると，インターネットに存在するノードのうち，60[%] 以上が NAT 下に存在し，さらに，およそ 15[%] のノードがプロクシを用いてインターネットに接続をしている．そこで，本実験では，NAT 下にいるノードの割合は 70[%] であると仮定して実験を行った．すなわち，NAT 下のノードとグローバルアドレスを持つノードの比率は 7:3 となる．

図 6.5 と比較すると，ノード数が 10,000 のときに待ち時間が 0.2[ms] ほど大きくなっているが，100 または 1,000 ノードのときはほとんど待ち時間の変化は見られない．これ

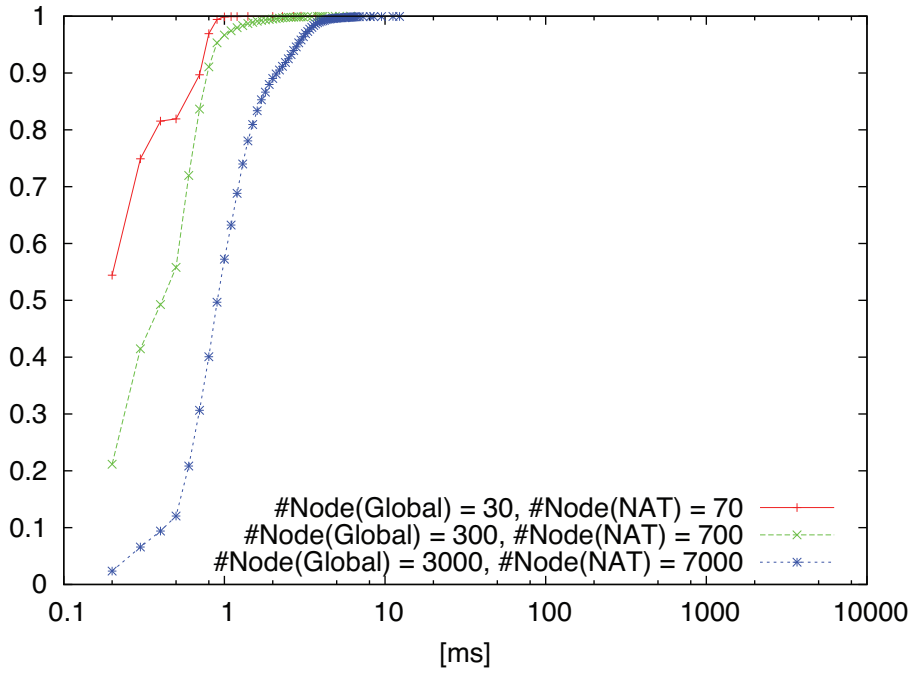


図 6.7 待ち時間 (CDF) - NAT 有り，定常状態, $r = 10, \alpha = 3$

より，定常状態においては DTUN を利用した場合でも，待ち時間への影響はほとんど無いことが分かる．また，表 6.2 のパーセンタイル値を見ても，NAT が介在しない状況とほとんど変わらない事が分かる．

図 6.8 は，Churn 下での待ち時間の計測を行った結果である．NAT 下のノードとグローバルアドレスを持つノードの比率は，図 6.6 の時と同じで 7:3 である．また，ノードの生存時間も同じく，平均 500[s] の指数分布に従うように設定した．

図 6.8 の結果より，100 ノードの時は図 6.6 及び図 6.7 と比較してもあまり変化は見られない．しかしながら，1,000 または 10,000 ノードの場合は，全体的に待ち時間がかなり大きくなっているのがわかる．表 6.2 のパーセンタイル値を見ると，1,000 ノードの場合は 95 パーセンタイル値が約 3[s] となっており，10,000 ノードの場合は，80 パーセンタイル値が約 6[s]，95 パーセンタイル値が約 9[s] となっており大幅に大きくなっている．

これは, NAT が介在するときは DTUN を用いる必要があり, その場合は二回 Kademlia の find node を行うため，タイムアウトによるクエリの失敗が待ち時間に影響する可能性が高くなるためであると考えられる．そのため，DTUN を用いない場合における Churn

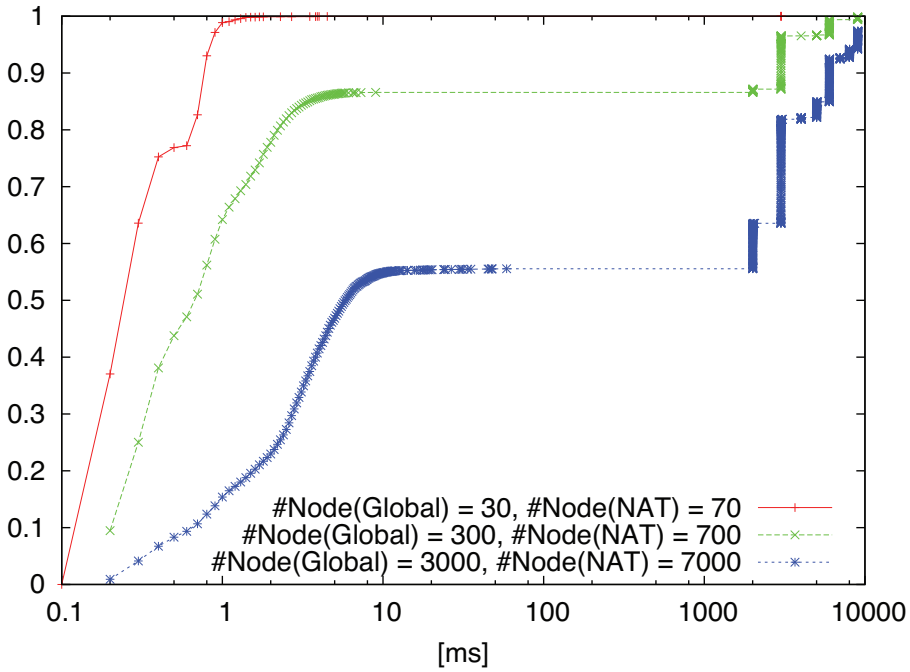


図 6.8　待ち時間 (CDF) - NAT 有り，Churn 下, $r = 10, \alpha = 3$

下での待ち時間よりも，DTUN を用いた方が待ち時間が大きくなってしまう．

次に，DHT のレプリケーション数 r と，find value 時の同時問い合わせ数 α を変化させて，Churn 下で同等の計測を行った．図 6.9 は r を，図 6.10 は α を変化させたときの，待ち時間を CDF で表したグラフである．ただし，図 6.9 では $\alpha = 3$，図 6.10 では $r = 10$ と固定した．また，ノード数はグローバルアドレスを持つノードが 3,000 ノード，NAT 下のノードが 7,000 ノード，合計 10,000 ノードである．Churn 頻度はこれまでと同じく，ノードの生存時間を平均 500[s] の指数分布に従うように設定した．

図 6.9 では，r を 6, 10, 14 と変化させている．r の値が大きいほど，待ち時間が小さくなっていくことが分かる．これは，r の値が大きいほど，find value 時に値が返ってくる確率が高くなるためであると考えられる．しかしながら，その差はわずかであり，r が待ち時間に与える影響は小さいと言える．表 6.2 のパーセンタイル値を見ても，80 および 90 パーセンタイル値すべてが 3[s] より大きな値となってしまっている．

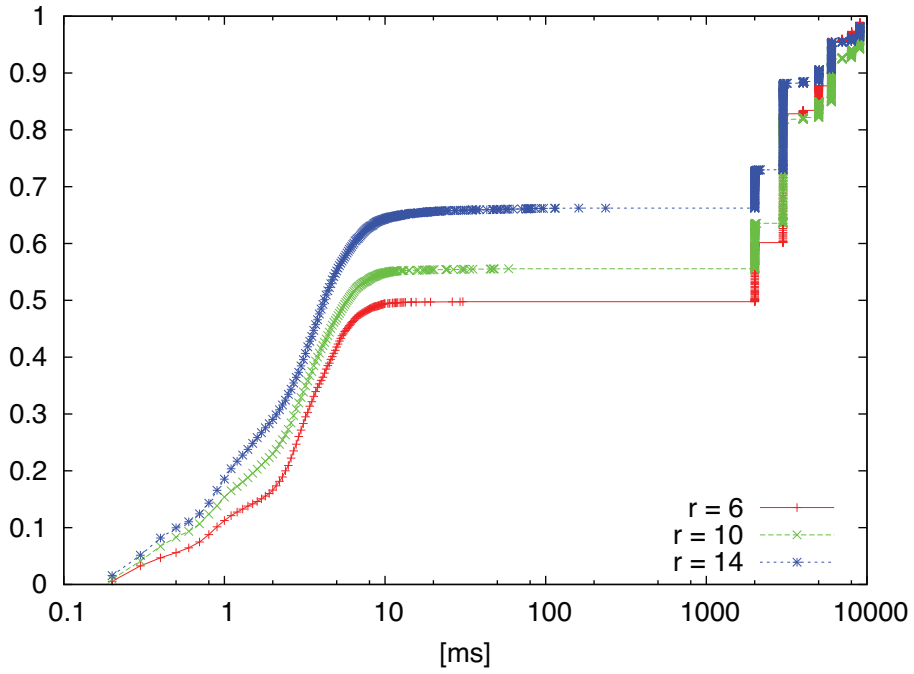


図 6.9 待ち時間 (CDF) - NAT 有り，Churn 下, $r = \{6, 10, 14\}, \alpha = 3$

6.3.2 値取得の成功確率

Churn 下では，待ち時間の問題のみならず，正しく値が取得できないという問題が発生する．これを回避する手法として，同時問い合わせ数 α とレプリケーション数 r を大きくする方法が考えられる．そこで，本節ではパラメータ α, r を変化させたときの，DHT 値取得の成功確率を議論する．

表 6.1 は，Churn 下で値を取得した時の成功確率を表したものである．ノード数の合計は 10,000 であり，NAT 有りの場合は NAT 下にあるノードとそうでないノードの比率は 7:3 としてある．ノード生存時間は，これまでと同様に平均 500[s] の指数分布に従うようにした．成功確率の算出は，find node を行ったときに正しく値を取得できた場合を成功，そうでない場合を失敗として，10,000 回試行して行った．

表 6.1 によると，NAT 無しで $\alpha = 3, r = 10$ の場合では，値取得の成功確率は 99.0[%] となり，高い値となった．NAT 有りの場合は，$\alpha = 3, r = 6$ の場合の成功確率が 76.5[%] となり，他と比較して小さな値となった．r を小さくしすぎると，Churn 下では DHT で

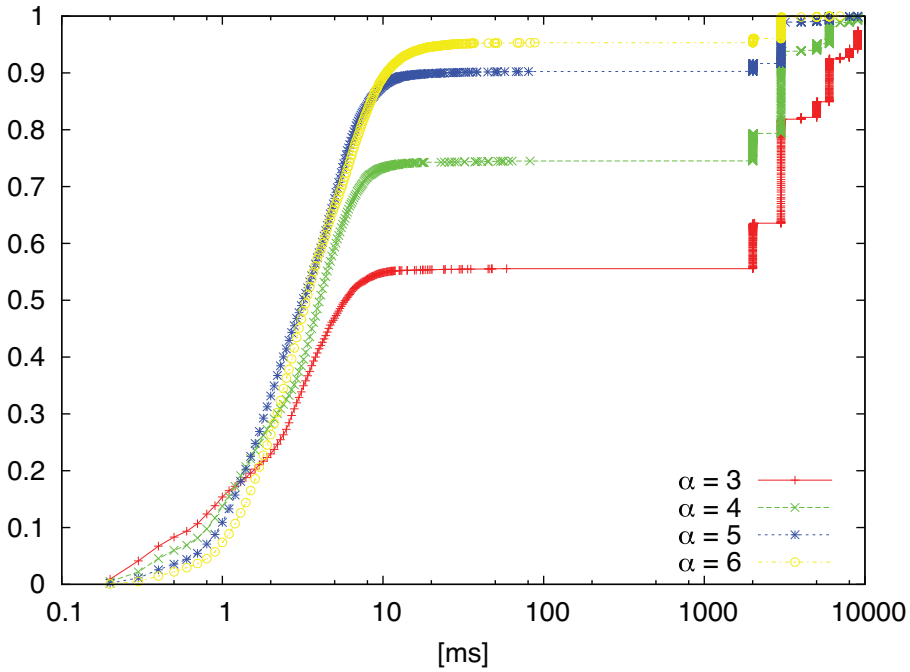


図 6.10　待ち時間 (CDF) - NAT 有り，Churn 下, $r = 10, \alpha = \{3, 4, 5, 6\}$

表 6.1　値取得の成功確率

	α	r	成功確率 [%]
NAT 無し	3	10	99.0
NAT 有り	3	6	76.5
	3	10	92.9
	3	14	96.9
	4	10	97.0
	5	10	98.6
	6	10	99.4

値を取得出来なくなる可能性が高くなると言える．これは，Churn 下では Kademlia のルーティングテーブルを正しく維持出来ないのと，値の保存先ノードがデータの保持期間より先に脱落してしまう事が原因と考えられる．逆に，r を大きくする毎に成功確率は上がっていき，$\alpha = 3, r = 14$ のときには 96.9[%] まで上昇した．一方，r を固定し α を変

化させた場合，$\alpha = 6, r = 10$ のときに成功確率が 99.4[%] までに達している．

これより，NAT 無しの場合 (DTUN を用いない場合) には，値取得の成功確率が大きいが，NAT 有りの場合には成功確率が若干低下してしまう事が分かる．α や r の値を大きくすることで，成功確率を高くすることが出来ると言える．特に，α を大きくすることは，待ち時間を小さくするのに加え，値取得の成功確率も大きくするため，効果的であると言える．

ただし，α の値を大きくすることは，瞬間トラヒックの増加が見込まれることに注意しなければならない．これは，同時に α の数だけメッセージを送信するからで，単純に α の数に比例して瞬間トラヒックは多くなると考えられる．しかしながら，総トラヒック数はそれほど変化しないと思われる．特に，find node の場合は，最終的に送信する必要のあるメッセージの数は α の値が大きくなろうと変化しない．find value の場合は，多くても，最低限必要なメッセージ数 $+\alpha$ の数だけメッセージが必要になると考えられ，無駄になるメッセージ数は高々 α 個となる．

6.3.3 議論

今回の評価では，NAT 下のノードの割合を全体の 70[%] と設定して実験を行った．IPv4 アドレス枯渇問題はますます深刻となっているため，グローバルアドレスを持つノードは，さらに減っていく可能性がある．そのため，グローバルアドレスを持つノードが大きく減少した場合についても，今後，検討する必要がある．

また，提案手法ではグローバルアドレスを持つノードの方がリソースを大きく消費するため，グローバルアドレスを持つノードは，リソース消費を減らすために NAT 下にあるノードになりすます可能性が考えられる．これは，DTUN ネットワークのノードとなることにインセンティブが無いからである．これも同様に，グローバルアドレスを持つノードの割合が低下する原因となりえる問題である．この問題は，リソースの提供に関して何らかのインセンティブが発生するようにすると解決する可能性がある．どのようにインセンティブを設定するかについても，同様に今後の課題であると言える．

表 6.2　80 および 90 パーセンタイル値

		パーセンタイル	
	#Nodes	80	95
NAT 無し	100	0.7	0.8
定常状態	1,000	0.6	1.0
図 6.5	10,000	1.3	2.2
NAT 無し	100	0.6	0.8
Churn 下	1,000	1.1	3002.1
図 6.6	10,000	2.3	6003.7
NAT 有り	100	0.4	0.8
定常状態	1,000	0.7	0.9
図 6.7	10,000	1.5	2.9
NAT 有り	100	0.7	0.9
Churn 下	1,000	2.3	3003.6
図 6.8	10,000	3007.1	9004.7
	r		
レプリカ数 r 変化	6	3006.4	6008.0
図 6.9	10	3007.1	9004.7
	14	3003.7	6009.7
	α		
同時問い合わせ数	3	3007.1	9004.7
α 変化	4	3001.3	5011.1
図 6.10	5	6.7	3003.7
	6	7.2	26.6

単位:[ms]

6.4　結論

P2P ネットワークは，頻繁にノードの出入りがあるため，同じ状態を保ち続ける定常状態となることはなく，常に攪拌された状態である Churn 状態となっている．Churn 状態

では，ルーティングテーブルと実際のネットワークトポロジの乖離が発生したり，DHT のデータがネットワーク上から失われたりしてしまう．これらを回避するためには，Churn 対策を行う必要がある．

Churn 対策の主な方法としては，ルーティングアルゴリズム部分で対策を行う方法と，DHT などのサービス部分で対策を行う方法に分類される．本研究では，DTUN の実証用ライブラリとして libcage の作成を行ったが，libcage では，ルーティング部分と DHT 部分の両方で対策を行った．

本研究では，ルーティング部分の対策方法として，タイムアウトが発生した場合の取り扱いの改良を行った．Kademlia は，基本的に UDP を用いた実装が想定されている．そのため，ネットワーク上からノードが存在しなくなっても，他のノードは，存在しなくなったノードの情報を長い時間保持し続けてしまう．Kademlia では，これら存在しなくなったノードの情報がルーティングテーブルから排除されるには，k-buckets の更新アルゴリズムが働くまで待つ必要があった．しかしながら，存在しなくなったノードの検出は，タイムアウトの発生を検出することで可能である．そこで，本研究では，タイムアウトが発生した場合に，ルーティングテーブルから削除する方法を提案した．

しかし，自身のルーティングテーブルから削除したとしても，他のノードが存在しなくなったノードの情報を持ち続ける可能性がある．この場合，find node 時などに，再びタイムアウトを待たなければならないという問題がある．そこで本研究では，一度タイムアウトしたら，そのノードの情報をしばらく持ち続けるタイムアウトキャッシュの提案を行った．libcage では，メッセージを送信する際，タイムアウトキャッシュを参照し直近の過去にその送信先がタイムアウトしていないかを検査して，無駄なタイムアウトが発生するのを防ぐ．

DHT 部分での Churn 対策は，Key-Value データを put したオリジンノード自身が再 put を行う方式と，データの保存先となるノードが再 put を行う方式を用いた．libcage では，ルーティングの維持のため近隣ノードの情報を定期的に更新する．そのため，データの保存ノードが再 put を行う方式では，構造化 P2P ネットワーク部分でのルーティングを行わずに済むため，オリジンノードの再 put より効率的に行える．逆に，オリジン

ノードの再 put は構造化 P2P ネットワーク部分でのルーティングが必要となるが，再 put を行うことが自身の利益につながる．そのため，libcage では両者の再 put 方式を行うが，効率化のため，オリジンノードの再 put は初めに put を行ってから 3 回まで行うよう実装した．

本研究では，libcage の性能を，Opteron 146HE(2GHz)，メモリ 4GB の PC100 台の上で動作させ，イベント多重により最大 10,000 ノード規模で実験を行った．計測を行ったのは DHT の値取得時に必要となった待ち時間と，DHT の値取得の成功確率である．Churn の状態は，各ノードの生存時間を平均 500[s] の指数分布として設定し，ノードが離脱したら，再び即参加するという事を繰り返した．また，値取得の試行回数は，100 個の値を 100 回ずつ取得した．さらに，これらを NAT 有りと無しの両方で測定した．なお，NAT 有りの場合は，全体の 70[%] のノードが NAT 下に存在するとした．

その結果，NAT 無しの Churn 下では，10,000 ノードのとき 80[%] が数 [ms] 以内で応答があり，95[%] が 6[s] 以内で応答があった．一方，NAT 有りの Churn 下では，10,000 ノードの時 80[%] が 3[s] 以内で応答があり，95[%] が 9[s] 以内で応答があった．NAT 有りの場合は DTUN を利用しているため，DTUN の負荷が原因でタイムアウトが発生し，待ち時間が長くなってしまったと考えられる．

値取得の成功確率は，find node の同時問い合わせ数 α と DHT の複製数 r を変化させて測定した．その結果，NAT 無しの場合，10,000 ノードでは $\alpha = 3$, $r = 10$ としたときに 99[%] の確率で正しく値が取得できたのに対し，NAT 有りの場合は，$\alpha = 6$, $r = 10$ としたときに，99.4[%] の確率で正しく値を取得できることが明らかとなった．

本実験では，平均生存時間を 500[s] として設定したが，実際の P2P ネットワークの平均生存時間はこれより長く，Gnutella では約 5,000[s] で有ることが知られている．そのため，本評価は最悪の場合の結果であり，実際にはこれよりも良い結果となると考えられる．

第 7 章

結論

本研究では，構造化 P2P ネットワークの設計時に重要となる点である，大規模時のルーティングテーブルの検索効率と，NAT 問題，大規模ノード下の Churn 状態について取り組んだ．

3 章では，Kademlia のルーティングテーブルの検索効率の効率化を目指し，理論と実験に寄る両面からの解析を行った．従来，Kademlia のルーティングテーブルは木構造にて管理されたが，本研究では配列を用いた管理方法を提案した．その結果，ネットワークに存在するノード数を N として，木構造の場合，枝を辿る回数のコストを 1 とし，配列の場合，エントリをルックアップするコストを 1 と定義した場合，木構造の場合は平均して $O(\log N)$，配列の場合は平均して $O(1)$ のコストが必要となることが明らかとなった．

構造化 P2P ネットワークは規模追従性の高いモデルであるが，それを実現とするには，ルーティングテーブルの検索も効率的に行われなければならない．本研究の提案は，Kademlia の規模追従性を支える重要な要素の一つであるといえる．

5 章では，NAT 問題とそれの解決方法について議論を行った．元々インターネットは，End-to-End の原則に基づいて設計されたため，ネットワークの中間では NAT などの複雑な制御は行われないのが原則であった．しかし，IPv4 アドレスの枯渇により，NAT が広く用いられるようになると，任意のノード同士で End-to-End の通信を行うことが難しくなってしまった．これは，基本的にノード同士が直接通信しあう Peer-to-Peer ネット

ワークを構築する際に，大きな問題となった．

そこで本研究では，DTUN と呼ぶ，二重構造の構造化 P2P ネットワークを構築して，NAT が介在する環境でもエンドのノード同士が直接通信を行えるようにする方式を提案した．本方式では，グローバルアドレスを持つノードのみから構成される DTUN ネットワークと，すべてのノードのみから構成されるサービスネットワークの二種類の構造化 P2P ネットワークが構成され，サービスネットワークのノードは，DTUN ネットワークのノードを介して NAT 越えの為の処理を行い，相互に直接通信を行う．

NAT 越えのための方式としては，STUN などの方式が存在するが，これらの方式では，サーバが必要とされるため，P2P ネットワークの利点を殺してしまう．ところが，本本式はサーバレスであるため，P2P ネットワークの利点を殺さず，より規模追従性・可用性が高い NAT 越え機構を提供出来る．また，NAT 越えを行うための別の方法として，UPnP などのを用いた方式も存在するが，これらの方式では多段 NAT の場合に適用できない．今後は Large Scale NAT の普及により，多段 NAT で構成されるネットワーク環境が増えると考えられるが DTUN 方式では多段 NAT の場合でも通信を行うことが可能である．

6.3 章では，3 章で提案した配列を用いたルーティングテーブルの管理方式と，5 章で提案した DTUN 方式を実装した，実証用ライブラリである libcage を用いて，これら提案方式のパフォーマンス測定を行った．実験では，NAT が介在する場合と，介在しない場合の両方で，Churn 下における DHT の値取得における遅延時間と値取得の成功確率を計測した．実験時の最大ノード数は 10,000 ノードであり，Churn の頻度はノードの生存時間を平均 500[s] の指数分布に沿うようにし，NAT 下に居るノードの割合を 7 割として設定した．

その結果，NAT 無しかつ Churn 下の条件では，10,000 ノードのとき 80[%] が数 [ms] 以内で応答があり，95[%] が 6[s] 以内で応答があった．一方，NAT 有りの Churn 下では，10,000 ノードの時 80[%] が 3[s] 以内で応答があり，95[%] が 9[s] 以内で DHT の値を取得することが出来た．本実験から，NAT 有りの場合では Churn 下における値取得の遅延が大きくなることがわかった．これは，DTUN とサービスモードの両方で DHT のルックアップが行われるからと考えられる．しかしながら，NAT が存在する場合でも 8

割は 3[s] 以内で取得できたこともわかった．

値取得の成功確率だが，find node の同時問い合わせ数 α と DHT の複製数 r を変化させて測定した．その結果，NAT 有りで Churns 下の状況でも，$\alpha = 3$, $r = 10$ と設定することで，99.4[%] の確率で正しく値を取得できることが明らかとなった．これは，NAT が介在する場合でも，Churn 下でほとんどのノードが正しく値を取得できることを意味する．

謝辞

本論文は，多くの人々からの支えがあったからこそ完成することが出来た．ここに関係者各位に感謝の意を表明したい．

まずは，北陸先端科学技術大学院大学・篠田研究室のスタッフに感謝したい．篠田陽一教授には研究指導はもちろん，学生生活に関する様々なことに関して助言いただき，本当に感謝している．知念賢一准教授も同様に，研究や学生生活に関して様々な助言を頂いた．特に，JSSST の論文誌へ投稿する際に大変御迷惑をおかけした．知念准教授の助けがなければ，本論文を仕上げるには至らなかったであろう．本当に感謝している．また，宇多仁助教と小原泰弘助教にも，研究生活を行う上で色々と助言を頂き，非常に感謝している．

次に，同・篠田研究室の修了生，現メンバーにも感謝したい．同じ研究仲間として，彼らと日々議論を行い互いに研鑽できたことは非常に誇りに思っている．特に，同じ博士後期課程の学生であった，井上朋哉氏と安田真悟氏には，研究と学生生活の両方から大きな支えとなってくれた．彼らには本当に感謝している．また，先輩，同期，後輩の，宮地利幸博士，Latt Khin Tida 博士，Nguyen Lan Tien 博士，芳炭将氏，Nguyen Nam Hoai 博士，Muhammad Imran Tariq 氏，我如古津世史氏，出口眞人氏，木下稔氏，内田幸治氏，磯崎直樹氏，阪上武寛氏，三浦良介氏，上野隆資氏，中井浩氏，梅木孝志氏，野中雄太氏，Saber Zrelli 博士，千装俊幸氏，栗原良尚氏，佐川喜昭氏，明石邦夫氏，松井大輔氏，川瀬拓哉氏，立花一樹氏，中村祐輔氏，橋本将彦氏，山田悠介氏，吉岡慎一郎氏，鍛治祐希氏らと共に学生生活を歩めたことを感謝したい．

また，北陸リサーチセンターの方々には実験などの面で大きなお世話になった．特に，

三輪信介博士には大変お世話になった．研究を支えてくださった北陸リサーチセンターの方々にも感謝したい．

さらに，WIDE プロジェクトの方々に方々に感謝したい．同じネットワークの研究をする仲間として交流できたことは，研究を進める上でたいへん大きな刺激となった．特に，IDEON メンバーの存在は，P2P ネットワークの研究を進める上で大きな心の支えとなった．WIDE プロジェクト並びに，IDEON のメンバーには本当に感謝している．

そして，慶応義塾大学の中村修教授と砂原秀樹教授，北陸先端科学技術大学院大学の丹康雄教授，情報通信研究機構の中川晋一教授らには，博士論文の審査をして頂いた上，厳しくも温かい意見を頂き感謝している．

最後に，研究生活を送る上で一番大きな支えとなったのが，父・和明と母・なり子の存在である．両親には本当に迷惑をおかけし，感謝してもしきれない思いである．

本研究に関する発表論文

論文誌（査読有り）

- 高野 祐輝，井上 朋哉，知念 賢一，篠田 陽一, ”NAT 問題フリーな DHT を実現するライブラリ libcage の設計と実装”, 日本ソフトウェア科学会学会誌『コンピュータソフトウェア』 2010 年 11 月号 「ソフトウェア論文」特集, pp. 58 - 76.

国際会議（査読有り）

- Yuuki Takano, Tomoya Inoue, Hiroshi Nakai and Yoichi Shinoda, ”Virtualization of Decentralized IP Networks using Structured P2P Network”, IEEE First International Global Information Infrastructure Symposium 2007, Poster Proceedings pp. 17-18.
- Yuuki Takano, Naoki Isozaki and Yoichi Shinoda, ”Multipath Key Exchange on P2P Networks”, The First International Workshop on Dependable and Sustanable Peer-to-Peer Systems, DAS-P2P 2006, In conjunction with The First International Conference on Availability, Reliability and Security, ARES 2006, Vienna, Austria, April 20th-22nd, 2006, Proceedings. IEEE Computer Society P2567, ISBN 0-7695-2567-9, pp.748-755.

国内会議（査読有り）

- 井上 朋哉，安田 真悟，高野 祐輝，宇多 仁，篠田 陽一, ”PacketX: エンドホスト

におけるアプリケーション指向 IP パケット制御機構”, インターネットコンファレンス 2008 論文集, 日本ソフトウェア科学会, 研究会 資料シリーズ No.57, ISSN 1341-870X, pp.99-107, 2008.10

国内研究会

- 高野 祐輝，井上 朋哉，篠田 陽一, ”P2P ネットワークを用いた仮想 IP ネットワーク構築”, 情報処理学会 マルチメディアと分散処理／コンピュータセキュリティ，ISSN 0919-6072，pp.237-242, 2007.3
- 高野 祐輝，井上 朋哉，篠田 陽一, ”P2P@i: Peer-to-Peer Attached Tunneling Interface の設計”, 電子情報通信学会 次世代ネットワークソフトウェア研究会, 2006.2
- 磯崎 直樹，高野 祐輝，篠田 陽一, ”大規模な散布型センサネットワークにおけるルーティング方式の検討”, 情報処理学会 モバイルコンピューティングとユビキタス通信／高度交通システム研究報告，ISSN 0919-6072，pp.135-140, 2005.11
- 高野 祐輝，磯崎 直樹，篠田 陽一, ”複数経路を用いた P2P ネットワーク上でのネゴシエーションと鍵共有法”, 情報処理学会 マルチメディアと分散処理／電子化知的財産・社会基盤研究報告，ISSN 0910-6072，pp.25-30, 2005.11

参考文献

[1] *A case for unstructured distributed hash tables*. IEEE, 2007.

[2] Avahi. http://avahi.org/.

[3] BitTorrent. http://www.bittorrent.com/.

[4] Apple - Support - Bonjour. http://www.apple.com/jp/support/bonjour/.

[5] Boost C++ Libraries. http://www.boost.org/.

[6] eMule-Project.net - Official eMule Homepage. Downloads, Help, Docu, News... http://www.emule-project.net/.

[7] Gnutella - A Protocol for a Revolution. http://rfc-gnutella.sourceforge.net/index.html.

[8] Khashmir. http://khashmir.sourceforge.net/.

[9] libcage. http://github.com/ytakano/libcage.

[10] libevent. http://www.monkey.org/~provos/libevent/.

[11] LimeWire. http://www.limewire.com/.

[12] The Mojito DHT. http://wiki.limewire.org/index.php?title=Mojito.

[13] Internet Archive - Napster's web site. http://web.archive.org/web/*/http://www.napster.com/.

[14] OpenSSL: The Open Source toolkit for SSL/TLS. http://www.openssl.org/.

[15] Overlay Weaver: An Overlay Construction Toolkit. http://overlayweaver.sourceforge.net/.

[16] Free Skype calls and cheap calls to phones - Skype. http://www.skype.com/.

[17] Welcome to UPnP^{TM} Forum! `http://www.upnp.org/`.

[18] Azureus, now called Vuze: Bittorrent Client. `http://azureus.sourceforge.net/`.

[19] Salman Baset and Henning Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. In *INFOCOM*. IEEE, 2006.

[20] G. Camarillo and IAB. Peer-to-Peer (P2P) Architecture: Definition, Taxonomies, Examples, and Applicability. RFC 5694 (Informational), November 2009.

[21] Martin Casado and Michael J. Freedman. Peering Through the Shroud: The Effect of Edge Opacity on IP-Based Client Identification. In *NSDI*. USENIX, 2007.

[22] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like P2P systems scalable. In Anja Feldmann, Martina Zitterbart, Jon Crowcroft, and David Wetherall, editors, *SIGCOMM*, pp. 407–418. ACM, 2003.

[23] S. Cheshire, B. Aboba, and E. Guttman. Dynamic Configuration of IPv4 Link-Local Addresses. RFC 3927 (Proposed Standard), May 2005.

[24] Stuart Cheshire and Marc Krochmal. Multicast DNS draft-cheshire-dnsext-multicastdns-11, March 2010.

[25] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In Hannes Federrath, editor, *Workshop on Design Issues in Anonymity and Unobservability*, Vol. 2009 of *Lecture Notes in Computer Science*, pp. 46–66. Springer, 2000.

[26] R. Droms. Dynamic Host Configuration Protocol. RFC 2131 (Draft Standard), March 1997. Updated by RFCs 3396, 4361, 5494.

[27] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-Peer Communication Across Network Address Translators. In *USENIX Annual Technical Conference, General Track*, pp. 179–192. USENIX, 2005.

[28] P. Krishna Gummadi, Stefan Saroiu, and Steven D. Gribble. A measurement study of Napster and Gnutella as examples of peer-to-peer file sharing systems. *Computer Communication Review*, Vol. 32, No. 1, p. 82, 2002.

[29] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. Efficient Routing for Peer-to-Peer Overlays. In *NSDI*, pp. 113–126. USENIX, 2004.

[30] Indranil Gupta, Kenneth P. Birman, Prakash Linga, Alan J. Demers, and Robbert van Renesse. Kelips: Building an Efficient and Stable P2P DHT through Increased Memory and Background Overhead. In Kaashoek and Stoica [32], pp. 160–169.

[31] M. Frans Kaashoek and David R. Karger. Koorde: A Simple Degree-Optimal Distributed Hash Table. In Kaashoek and Stoica [32], pp. 98–107.

[32] M. Frans Kaashoek and Ion Stoica, editors. *Peer-to-Peer Systems II, Second International Workshop, IPTPS 2003, Berkeley, CA, USA, February 21-22,2003, Revised Papers*, Vol. 2735 of *Lecture Notes in Computer Science*. Springer, 2003.

[33] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, pp. 654–663, 1997.

[34] Abhishek Kumar, Shashidhar Merugu, Jun Xu, Ellen W. Zegura, and Xingxing Yu. Ulysses: a robust, low-diameter, low-latency peer-to-peer network. *European Transactions on Telecommunications*, Vol. 15, No. 6, pp. 571–587, 2004.

[35] Ben Leong, Barbara Liskov, and Erik D. Demaine. EpiChord: Parallelizing the Chord lookup algorithm with reactive routing state management. *Computer Communications*, Vol. 29, No. 9, pp. 1243–1259, 2006.

[36] Gurmeet Singh Manku, Mayank Bawa, and Prabhakar Raghavan. Symphony: Distributed Hashing in a Small World. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[37] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron, editors, *IPTPS*, Vol. 2429 of *Lecture Notes in Computer Science*, pp. 53–65. Springer, 2002.

[38] P.V. Mockapetris. Domain names - concepts and facilities. RFC 1034 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936.

[39] P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966.

[40] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[41] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pp. 161–172, 2001.

[42] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address Allocation for Private Internets. RFC 1918 (Best Current Practice), February 1996.

[43] Sean C. Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT (Awarded Best Paper!). In *USENIX Annual Technical Conference, General Track*, pp. 127–140. USENIX, 2004.

[44] Rodrigo Rodrigues and Charles Blake. When Multi-hop Peer-to-Peer Lookup Matters. In Voelker and Shenker [56], pp. 112–122.

[45] J. Rosenberg, R. Mahy, and P. Matthews. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN) draft-ietf-behave-turn-16, July 2009.

[46] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389 (Proposed Standard), October 2008.

[47] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489 (Proposed Standard), March 2003. Obsoleted by RFC 5389.

[48] Mema Roussopoulos, Mary Baker, David S. H. Rosenthal, Thomas J. Giuli, Petros Maniatis, and Jeffrey C. Mogul. 2 p2p or not 2 p2p? In Voelker and Shenker [56], pp. 33–43.

[49] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In Rachid Guerraoui, editor, *Middleware*, Vol. 2218 of *Lecture Notes in Computer Science*, pp. 329–350. Springer, 2001.

[50] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Proceedings of the First International Conference on Peer-to-Peer Computing*, P2P '01, pp. 101–, Washington, DC, USA, 2001. IEEE Computer Society.

[51] Haiying Shen, Cheng-Zhong Xu, and Guihai Chen. Cycloid: A constant-degree and lookup-efficient P2P overlay network. *Perform. Eval.*, Vol. 63, No. 3, pp. 195–216, 2006.

[52] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational), January 2001.

[53] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, Vol. 11, No. 1, pp. 17–32, 2003.

[54] D. Thaler, L. Zhang, and G. Lebovitz. IAB Thoughts on IPv6 Network Address Translation. RFC 5902 (Informational), July 2010.

[55] D. Velten, R.M. Hinden, and J. Sax. Reliable Data Protocol. RFC 908 (Experimental), July 1984. Updated by RFC 1151.

[56] Geoffrey M. Voelker and Scott Shenker, editors. *Peer-to-Peer Systems III, Third International Workshop, IPTPS 2004, La Jolla, CA, USA, February 26-27, 2004, Revised Selected Papers*, Vol. 3279 of *Lecture Notes in Computer Science*. Springer, 2005.

[57] Arno Wacker, Gregor Schiele, Sebastian Holzapfel, and Torben Weis. A NAT Traversal Mechanism for Peer-To-Peer Networks. In Klaus Wehrle, Wolfgang Kellerer, Sandeep K. Singhal, and Ralf Steinmetz, editors, *Peer-to-Peer Computing*, pp. 81–83. IEEE Computer Society, 2008.

[58] I. Yamagata, T. Nishitani, S. Miyakawa, A. Nakagawa, and H. Ashida. Common requirements for IP address sharing schemes draft-nishitani-cgn-04, March 2010.

[59] 佐藤 良. P2P 通信技術: NAT 越え ～STUN と UPnP と，時々，TURN～. 株式会社コナミデジタルエンタテイメント，プロジェクトソリューションセンター，R&D 推進グループ，`http://homepage3.nifty.com/toremoro/study/voip2008/NATTraversal.pdf`.

[60] 高野祐輝, 井上朋哉, 知念賢一, 篠田陽一. NAT 問題フリーな DHT を実現するライブラリ libcage の設計と実装. 日本ソフトウェア科学会学会誌『コンピュータソフトウェア』「ソフトウェア論文」特集, pp. 58–76, 11 2010.

[61] 宮地利幸, 中田潤也, 知念賢一, ラズバン・ベウラン, 三輪信介, 岡田崇, 三角真, 宇多仁, 芳炭将, 丹康雄, 中川晋一, 篠田陽一. StarBED: 大規模ネットワーク実証環境. Technical report, 1 月 2008. 情報処理 第 49 巻 第 1 号，pp.57—70.