大学院論文輪講資料 平成 16 年 1 月 23 日

分散ハッシュテーブルの軽量な負荷分散の一手法

新領域創成科学研究科 青山森川研究室 37302 岡 敏生

The demand for large scale distributed systems is steadily increasing as many people ask for more benefit from the services on the Internet. The large scale distributed systems, however, have been faced with difficulties in terms of robustness, consistency, performance, and cost. To address this problem, many researchers study distributed hashing tables(DHTs) that enable us to construct scalable systems easily. While many people attempt to realize various new applications on top of the DHT platform, we must settle several issues in advance so that the DHT can become a practical platform. In this paper, we discuss the maintenance cost of DHT overlay network topology, and how to reduce the topology maintenance cost.

1 はじめに

インターネットが社会基盤になるにつれ膨大な数のユーザがネットワークの恩恵を受けるようになっている．多くのユーザがひとつのインターネットアプリケーションを利用する際，その裏ではたくさんのサーバが動作しているということも少なくない．複数のサーバを導入する原因は様々で，そもそも単体のサーバでは大人数を収容できないというケースから，収容できるが反応が異常に遅くなってしまうケース，あるいは高性能なサーバを少数そろえるより廉価なサーバをたくさん集めた方が安く構築できるなどが考えられる．いずれの要因にせよ多数のホストを協調的に動作させるというのはしばしば採用されるアプローチである．一般に大規模なシステムでは小規模システムの運用とは全く違う要件が求められると考えられている．まずひとつに必要とされる資源が格段に多くなるということが挙げられる．消費される資源にはユーザの数にほとんど影響を受けない部分以外に，人数に対して線形に増加する要素が存在する．一般的に無制限に資源を利用できるケースは稀であり，技術的もしくは資金的な原因から制限が加わることが多い．二つ目の要件として挙げられるのはパフォーマンスである．処理しなくてはいけないリクエスト数の増加に伴いパフォーマンスの劣化が起きることは明らかであろう．特にホスト間がネットワークによって接続されている場合ホスト内のバスの通信速度とは比較にならないほど遅いため，遅延は無視できない要素となってくる．三つ目の要件として挙げられるのは耐障害性である．システムを利用しているユーザが多い場合，システムの一時的な停止による損害は計り知れない．都合の悪いことに大規模なシステムでは一箇所に問題が生じただけで全体に影響を及ぼしてしまうことも少なくない．以上，大規模システムにおける特殊性について三つほど例示したが，これらの事柄は全く別の問題ではなく互いが密接に関連していることは容易に想像がつくであろう．

本稿では大規模システムで採用されているアプローチの中でもとりわけハッシュ機能 (hash-like functionalities) に着目して取り上げることにする．ハッシュがシステムのパフォーマンス改善に多用されることは周知の事実であるが，本稿で扱うのは分散環境におけるオブジェクト発見に利用される分散ハッシュという概念である．分散ハッシュも通常のハッシュと同様にオブジェクトに対してキーを割り当てることによって，オブジェクトの高速発見や負荷分散に利用される．ただ分散ハッシュでは通常のハッシュの機能のみで利用されることは稀であり，むしろその他の機能が組み込まれて利用されることが一般的である．たとえば，同じキーに対応する複数のオブジェクトの中から（ネットワーク的に）近隣のオブジェクトを発見したり，一部のホストが落ちた場合にも修復する機能，システム内のホスト数の増減があってもシームレスに動作する機能，キーに（一致するアイテムではなく）”類似する”キーを持つアイテムを発見する機能等である．このように分散ハッシュでは多様な機能を備えることが可能なため広範なインターネットアプリケーションに利用可能ではないかと注目を集めている．これらの機能は実際のシステムで適用されユーザが知らないうちに利用しているケースから，まだ実運用はされず研究レベルで考えられているものまで様々である．

このように分散ハッシュは大きな可能性を備えたアルゴリズムであるが，分散ハッシュメカニズムにはオーバーレイネットワークの構造を維持するために頻繁にメッセージを交換しなくてはならないという問題がある．本稿では Chord, Pastry, Koorde 等の分散ハッシュアルゴリズムにおけるメンテナンスコストの軽減手法の提案を行う．これらのアルゴリズムではシステムの障害に対応するために定期的にトポロジー維持メッセージが交換されている．提案手法を利用することで確率的に正確に保証された負荷分散特性を実現する際に必要となるトポロジー維持コストを virtual servers を利用する手法と比較しておよそ $\frac{1}{\log N}$(N:ホスト数) にすることができる．

2 章では既存の分散ハッシュアルゴリズムの説明を行い，3 章では分散環境においてハッシュがどのように応用されうるかについて例を交えながら言及する．4 章では 2 章の

内容を踏まえた上でトポロジー維持コスト低減アルゴリズムの提案，数学的考察，シミュレーションによる確認を行う．最後に 5 章で本稿のまとめを行う．

2 分散環境におけるハッシュ機能

本節では分散環境におけるハッシュ機能について言及する．まず 2.1 節では通常のハッシュについて触れ，それがインターネットのように動的な分散環境でどのような問題を引き起こすか説明する．ついで 2.2 節では CDN で利用されている Consistent Hashing に関して紹介する．そして 2.3 節では Consistent Hashing のスケーラビリティを改善したアルゴリズム Chord について言及する．

2.1 最も基本的なハッシュ

本節では基本的なハッシュスキームの一つ，線形探索法 (Linear Probing) について説明する．線形探索法では固定サイズのバケット（データの格納場所）B_0 から B_{M-1} を予め確保しておく．オブジェクト a を格納する場合には鍵 K_a を生成し，ハッシュ関数 $f(x)$ に基づいてデータ格納バケットを決定する．例えば，$f(K_a) = 3$ のときにはオブジェクト a は B_3 に格納される．ただし，$f(K_i) = f(K_j)(i! = j)$ の時データ格納バケットに衝突が発生してしまうことに注意しなくてはならない．線形探索法では衝突が発生した場合，そのバケットから順に空いているバケットを探し，空いていることが確認されたバケットにデータを格納する．格納したデータを取得する場合には鍵を元に探し出せばいい．例えば，鍵 K_b を探したい場合には $f(K_b)$ 番目のバケットから順に該当する鍵に対応するデータを探せばいい．データ数がバケット数に対して十分小さい場合にはデータの格納，取得にかかるコストは $O(1)$ であることは周知の事実である（図 1）．

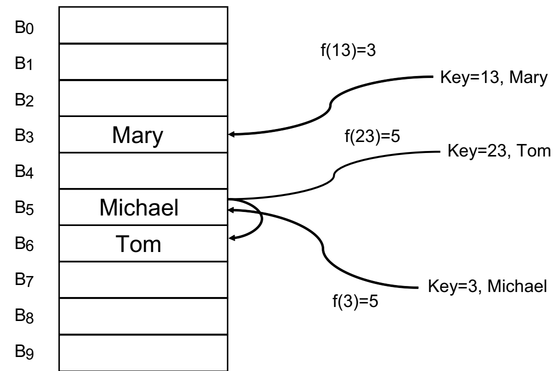

図 1: Linear Probing

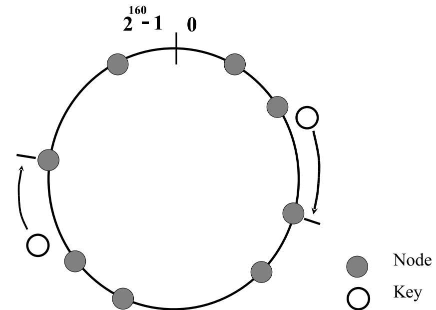

図 2: Consistent Hashing

2.2 Consistent Hashing

2.1 節で述べたハッシュスキームは単純だという利点があるが，バケットサイズが固定であるというのが大きな問題となっている．分散環境下では先に述べたバケットはネットワーク上に分散したホストに配置される．このような環境下ではホストがクラッシュしたり，ネットワークが切断されるといったことが考えられる．逆にシステムを増設したい場合などにはホスト数が増加するといったことも考えられる．このような環境下では先ほどの単純なハッシュスキームでは適切に動作することができない．そのような問題を解決するのが Consistent Hashing[8] である．

Consistent Hashing ではまず巨大な ID 空間 S を想定する（ここでは ID 空間のサイズを 2^{160} とする）．ホストの ID と鍵はこの空間に属しているものとする．Consistent Hashing に参加するホストはこの空間上のランダムな ID が割り当てられる．オブジェクト a はその鍵 $K_a(K_a \epsilon S)$ から見て時計回りに最初のホストに格納される．その格納メカニズムを図 2 に図示した．図から明らかであるが，ノードの参加・離脱があってもオブジェクトと格納先のバケット（ホスト）の関係はほとんどほとんど変化しない．

ここで負荷分散の特性やノードの参加・離脱に伴うデータの移動量が具体的にどのようになっているかが重要である．詳細な議論は省略するが，Consistent Hashing では高い確率で $(1+\epsilon)L/N$ の負荷分散特性を実現することができる (L:システム総負荷, N:ノード数, ϵ:ある実数)．そして Consistent Hashing では参加・離脱するホストに格納される/されていたデータのみが移動される．当然ながら ϵ を十分小さく取ったときこれはデータ移動量の観点から見て最小である．

2.3 Chord

Chord とは 2.2 節で説明した Consistent Hashing を大規模分散環境のために拡張したアルゴリズムである．先の Consistent Hashing では各ホストがシステム内のほぼすべてのホストの IP やそのホストへのネットワーク距離，そしてそのホストが Node Failure を起こしていないかなどの情報を知っているという条件の下で動作した．しかしながら大規模な分散環境ではその仮定は必ずしも想定できない．各ホスト間で常に最新の情報を取得するためには大量の制御パケットを要することになるためである．Chord アルゴリズムはこのような問題点を解決するために考え出された．Chord は Consistent Hash ファミリーに属するハッシュアルゴリズムであり，Consistent Hashing の多くの性質を踏襲している．しかし Chord はノード数 N の分散システムにおいて $O(\log N)$ 個のホストに関しての情報を持っているだけでよく，Consistent Hashing に比べて各ホストが保持しておくべき情報の量が格段に少

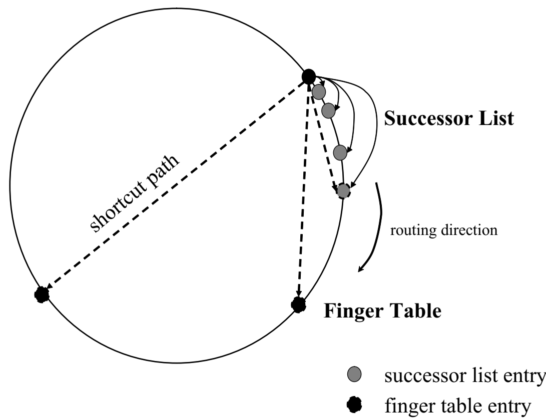

図 3: Chord

ない．また Chord は極めてロバスト性の高いアルゴリズムであり，全体の半分のホストが同時に Fail Stop したとしても，各ホストは $(1-\frac{1}{N})$ の確率で即座に正確なリンク情報を取得することができる．つまりホスト全体の中で数ノードのみが再びシステムに再加入する処理を行えばよく，$O(\log N)$ 時間でシステムは復旧可能である．

では次に Chord の具体的なアルゴリズムの説明を行う．Chord の場合も Consistent Hashing と同様に広大な ID 空間からランダムに NodeID および鍵が生成されるようにする．そして Consistent Hashing と同様にオブジェクトはその鍵から見て時計回りに最初のホストに格納される．Chord では各ノードは自ノードから見て時計回りに $O(\log N)$ 個の隣接ノードへのリンクを保持することとする（図 3:successor list）．このようにリング状にリンクを形成することでバケツリレー的にパケットを転送してゆけば任意のホストから任意のホストへパケットを送ることができる．ここで $O(\log N)$ 個のリンクを維持するのは，一部のホストが Fail Stop した場合でもリングを維持するためである．そのため Chord では大量のホストが同時に Fail Stop した場合でも，ID 空間上における次のホストの IP，Port 番号を非常に高い確率で保持していることになる．ただ，このままでは明らかにパケットの転送に平均で $O(N)$ のホップ数が必要になってしまう．そこで Chord では ID 空間上でおよそ $2^i (i = 0, 1, .., 159)$ 離れたホストに対してショートカットリンクを作成することでルーティングホップ数を $O(\log N)$ にまで削減している (図 3:finger table)．finger table が正確に維持されている場合，パケットが一回ホストを転送されるたびに目的ホストまでの ID 空間上の距離は半分以下になることになる．またノードの参加により finger table が古い情報になってしまったとしても高い確率でこの性質が達成される．当然のことながらノードの離脱によってショートカットリンクが切れてしまった場合にはこの性質が失われてしまうので，ショートカットリンクを再構成する必要がある．以上のようなメカニズムで Chord は Consistent Hashing より格段に少ないリンク情報で同様の性質を実現している．

Chord が純粋な Consistent Hashing に対して劣っている点は原理的には Consistent Hashing が目的のオブジェクトを発見するのに 1 hop で到達可能であるのに対して，Chord では $O(\log N)$ ホップ必要である点である．しかしながら，Consistent Hashing を利用する局面では実際には 3.1 節で説明する Random Tree のようにキャッシュを利用した負荷メカニズムを利用することがあり，そのような場合には Consistent Hashing であっても $O(\log N)$ のホップ数が必要となってくる．したがってこのような利用形態ではホップ数に本質的な相違はない．

3 分散環境におけるハッシュ機能の応用

本節では分散ハッシュがインターネットアプリケーションにおいてどのような利用形態を提案されているか紹介する．

3.1 CDN(Content Distribution Network)

CDN とはインターネットにおいてコンテンツを効率的に配信するためのネットワークである．このような用途にも分散的なハッシュ機能が利用されている．近年大量のディジタルコンテンツがインターネット上で公開されているが，そのパフォーマンス改善に広く利用されているのが CDN である．一般にユーザのウェブサイトへのアクセスは Zipf-like 分布に従うといわれており，CNN のような有名なサイトにアクセスが集中する傾向にある．ひとつのサイトが消費できるネットワーク帯域には制限があり，大量のアクセスがあるとネットワークに輻輳 (congestion) が起こってしまう．またサーバ自体も高負荷になり，場合によっては正常に動作しないようになりうる．そのようなウェブサイトにおいて何ら改善を行わない場合，全くアクセスできないという状況が発生してしまう．一つの解決方法はミラーリングによって複数のクラスタサーバを各地に運用し，各サイトをマルチホーミングするというアプローチである．しかしこの方法は実際に求められる容量以上に設備投資する必要があり，効率的な解決方法ではない．[1]

近年一般的になっているアプローチは CDN(Content Distribution Network) によるアプローチである．CDN では配布するコンテンツのレプリカやキャッシュを分散配置することによってこのような問題を解決している．ネットワーク上に複数存在するコピーから近隣のものを選択して配信することによって，ユーザからみたパフォーマンスも改善することが期待される．CDN の実現方法にも多様な実現方法があるが，代表的な例が Akamai Technologies による DNS を利用した CDN であろう．Akamai のキャッシュサーバの選択では Consistent Hashing と Random Tree と呼ばれるアルゴリズムが利用されている．ここでは Akamai における適切なキャッシュサーバの選択のメカニズムについて概説する．

まず図に示すように Akamai のキャッシュサーバ $C = 1, 2, 3, .., c$ が全世界に分散している環境を想定する．キャッシュサーバは各キャッシュサーバへのネットワーク上の距離とそのサーバの (IP, Port) を知っているが，各キャッシュサーバはどのコンテンツがどのホストに格納されているかは知らないものとする．このような条件下で Akamai が提供する機能は所望のコンテンツを保持している近隣のキャッシュサーバを見つけ出すことである．Akamai のすべ

[1] マルチホーミングは経路が変化したときに，BGP によるルーティングが安定するまでに時間がかかるという問題もある

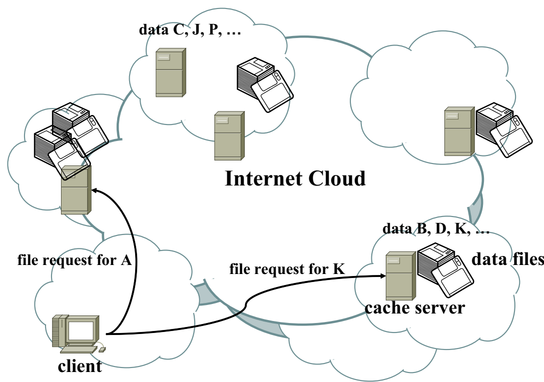

図 4: Content Distribution Network

ての仕組みを説明するのは本稿の目的ではないので，コンテンツのコピーをどのように配置し，どのキャッシュサーバに配置されているか発見する部分に特化して述べる．先に説明した Consistent Hashing は次に説明する Random Tree の各ノードに該当するホストの IP を割り出す際に利用される．Random Tree では図 5 に示すようにキャッシュサーバをランダムに選んで作成する木構造である．この木構造はコンテンツごとに作成されるため，異なるコンテンツには異なる木構造が（仮想的に）作成される．コンテンツの本体（レプリカと呼ぶ）は木の Root ノードに格納される．クライアントは木構造のランダムな Leaf ノードとなるキャッシュサーバに接続し，もしこのキャッシュサーバが所望のコンテンツのキャッシュを持っていれば，クライアントに渡す．逆にもし持っていない場合は，親ノードとなるサーバになるホストにリクエストを転送する．この動作を繰り返すことで Root ノードにたどり着くまでにキャッシュを発見できる．ここで注意すべき点は二つである．一つは頻繁にアクセスされるコンテンツは途中ノードでキャッシュされることに注意してほしい．図の例の場合，左のコンテンツ (file A) の方が頻繁にアクセスされるコンテンツであることが分かる．キャッシュについて一般的にいえることだが頻繁にアクセスされるコンテンツほど高速に取得できる．もう一つはツリーがコンテンツごとにランダムに構成されるため，特定のホストに負荷が集中してしまわない点である．

この Random Tree の構成に Consistent Hashing が利用されていることを再度確認しておく．ハッシュはこのように負荷分散に対して強力な枠組みである．ランダム化を施すことによって恣意的に木を構築する場合に比べて容易に負荷分散できる．単純なウェブプロキシサーバのツリーの場合すべてのコンテンツについて同じ木構造が利用されるため，木の Root ノードに負荷が集中してしまう．Random Tree の場合コンテンツ毎にルートノードが分散するため，サーバの負荷を分散することができる．ただしコンテンツ毎に上流（親）が異なるためクエリを転送する際，上流の番号に対応するキャッシュサーバの (IP, Port) を解決する必要がある．上流のキャッシュサーバの解決に Consistent Hashing が利用されている．

このように動的に構成ホストが変化するような環境で一貫したハッシュ機能を提供することが負荷分散性を提供する上で重要になっている．分散ハッシュはこのような機能を提供するために利用されることが多い．尚，オブジェクトを取得する際に発生する遅延を削減する仕組みについては省略する．

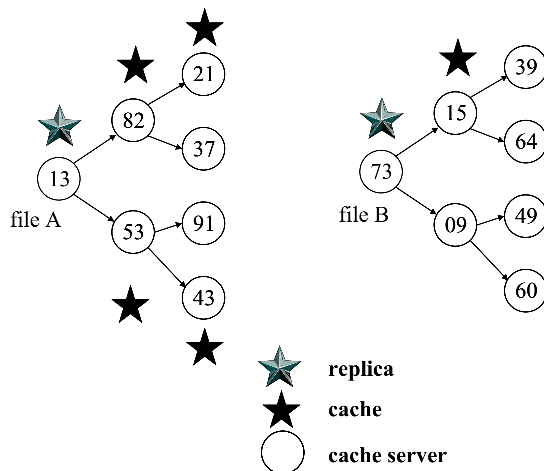

図 5: Random Tree

3.2　グローバルストレージ

グローバルストレージも分散ハッシュメカニズムの有力なアプリケーションの一つである．グローバルストレージとは多くの組織や人がストレージ資源を互いに共有することで実現する巨大なストレージである．ファイルのコピーが世界中にばらまかれた環境下で位置・複製透過なファイルアクセスを提供することで，ユーザが自由に移動するようなモバイル環境でも高速なファイルアクセスが提供できると考えられる．またファイルのコピー（レプリカ）を複数作成し，分散配置することでディスクの故障，ネットワークの切断による可用性の低下を防ぐことができる．前節で紹介した CDN は基本的にウェブコンテンツを対象にしたシステムであったのに対し，グローバルストレージでは膨大な数のホスト，ファイルを対象にしているため，スケーラビリティが極めて重要になってくる．Consistent Hashing ではそのような膨大なホストを対象にしたアルゴリズムではないため，Chord や Pastry[13], Tapestry[16] などといった分散ハッシュテーブルが利用されると考えられている．

3.3　pSearch

pSearch は純粋な意味で分散ハッシュではないが，分散ハッシュと同一のシステムを利用している点において注目すべきアプリケーションである．通常ハッシュでは提示した鍵に完全一致する鍵をもつオブジェクトを返す．pSearch ではこの制約を取り払い，提示する鍵に類似した鍵をもつオブジェクトを返す仕組みを提供している．pSearch では d 次元ベクトルを鍵とする分散ハッシュテーブル CAN(Content Addressable Network)[12] を利用し，d 次元ベクトル鍵と近い鍵を持つオブジェクトの取得を行う．これによって，ある画像に近い特徴量を持つ画像を検索したり，ある文書に近い特徴量を持つ文書を検索したりすることができると考えられている．pSearch では次元の呪いを回避するため，LSI(Latent Semantic Indexing)[3] を利用している．

4　軽量な確率的負荷分散アルゴリズム

本節では本題の確率的負荷分散アルゴリズムについて述べる．

4.1　Chord におけるトポロジー維持コスト

2.3 節では Chord アルゴリズムの概要について説明したが，ここでは Chord においてトポロジー維持に必要となるコストについて述べる．大きく分類すると Chord トポロジー維持コストは successor list と finger table の維持の二つに分かれる．successor list とはすでに触れた通り ID 空間上で隣接するホストのリストであり，このリストが Chord アルゴリズムの動作上で極めて重要な位置を占めている．このリストの中に適切なデータが含まれない場合，Chord は正常な動作に戻るまで長い時間を要することになってしまう．したがって Chord では定期的に（以下 30 秒として考える）メッセージ交換を行い，常に正しい Successor が含まれるように successor list を更新している．

次に finger table の維持コストについて説明する．finger table はルーティングを高速化するためのものであり，successor list ほど正確さを要求されることはない．finger table が仮に本来のホストに向けられていなかったとしても，必要ホップ数が増えるかあるいは全く変化しないかのどちらかである．とはいえ長期間更新しないとルーティングホップ数の増加を招くため，やはり定期的に更新を行っている．[14] では，30 秒毎に $O(\log N)$ エントリの中から 1 エントリを選び出し更新を行っている．finger table ではメッセージ交換の対象となるホストが $O(\log N)$ であるため，結果的に successor list と finger table は全体でほぼ同じ維持コストを必要としていることが分かる．

本来，トポロジーに維持に必要なコストはこれだけであるはずだが，実際にはこの $O(\log N)$ 倍のコストが必要である．なぜならば Consistent Hashing では $(1+\epsilon)L/N$ の負荷分散特性を実現するためには各ホストに $O(\log N)$ 個の virtual servers を動作させせなくてはならないからである．virtual server はそれぞれ ID 空間上のランダムな点にマッピングされるため，それぞれについて successor list と finger table の維持が必要である．その結果，先に述べた維持コストの $O(\log N)$ 倍のコストが必要となっている．

4.2　実現した事項（比較）

既存手法と提案手法の定性的な比較を Table 1 に示した．Virtual servers とは Consistent Hashing に基づいて行われる負荷分散機能であり，元来の Chord で利用されている手法である．この手法ではリンク維持に必要となるコストが全般的に大きくなっていることが分かるであろう．その反面，一つのノードが参加・離脱したとき，並列的にデータの移動がなされるため近隣ノード (predecessor) とやりとりするデータ量が単位ホスト当たり少なくなっている．これはノードの参加・離脱にあたって高速に定常状態へ修復できることを示している．一方，Power of Two はリンク維持コストは少ないがノードの参加・離脱にあたって移動すべき総データ量 $(1+\epsilon)L/N$ がすべて 2 ホスト間でなされるため，比較的ゆっくりと修復が行われることを示している．Power of Two に関しては 4.7 節で説明する．

Virtual Servers

論理リンク数	$O(\log^2 N)$
finger table の維持	$O(\log^2 N)$
ノード参加のコスト	実質 $O(\log^2 N)$ メッセージ
successor list の維持	$O(\log N)$ times / 30(s)
参加に伴うデータ移動量（1 ノードあたり）	$L/(N \times O(\log N))$
離脱に伴うデータ移動量（1 ノードあたり）	$L/(N \times O(\log N))$
負荷分散特性	$(1+\epsilon)L/N$

Power of Two

論理リンク数	$O(\log N)$
finger table の維持	$O(\log N)$
ノード参加のコスト	実質 $O(\log N)$ メッセージ
successor list の維持	once / 30(s)
参加に伴うデータ移動量（1 ノードあたり）	L/N
離脱に伴うデータ移動量（1 ノードあたり）	L/N
負荷分散特性	$\frac{\ln \ln n}{\ln d} + O(L/N)$?

提案手法

論理リンク数	$O(\log N)$
finger table の維持	$O(\log N)$
ノード参加のコスト	実質 $O(\log N)$ メッセージ
successor list の維持	once / 30(s)
参加に伴うデータ移動量（1 ノードあたり）	$L/(N \times O(\log N))$
離脱に伴うデータ移動量（1 ノードあたり）	$L/(N \times O(\log N))$
負荷分散特性	$(1+\epsilon)L/N$

表 1: 比較表

本稿の提案手法は基本的なアイデアを virtual servers のアイデアから引き継いでおり，負荷分散などについては同じ特性を持っている．しかしながら，リンク維持コストはそれに比べて少ないものとなっている

4.3　準備

本節では次節以降で用いる用語の説明，定義を行う．

- N ：システム内のホスト数
- L ：システム内に格納するオブジェクトの総数
- ϵ ：ある実数（典型的には 1 以下を想定）
- s ：successor list のサイズから 1 引いたもの
- r ：耐障害性のためオブジェクトを複製する個数
- I_U ：ID 空間のサイズ (2^{160})
- I_a ：ノード a の ID
- H ：ハッシュ関数
- w_i ：ホスト i の容量に基づいて決められる値

node

ID 空間上の一点にマッピングされる計算機能を持ったエンティティ．
実在するホストまたは virtual server を指す

predecessor
: ID 空間上で反時計回り側に見て最初にある node

successor
: ID 空間上で時計回り側に見て最初にある node

predecessor list
: ID 空間上で反時計回り側の隣接 node 群

successor list
: ID 空間上で時計回りの隣接 node 群

zone
: 二つの node に挟まれた ID 空間

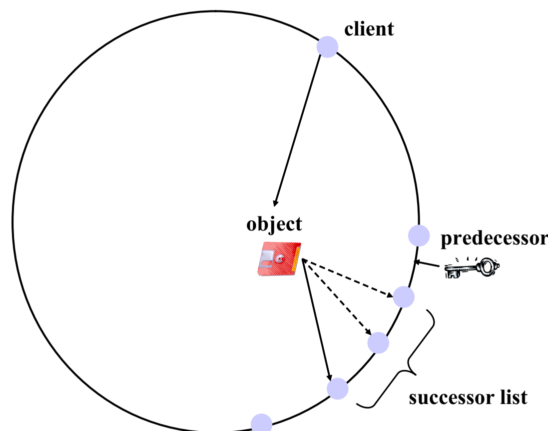

図 6: 提案手法

4.4 仕組み

本節ではトポロジー維持コストの削減アルゴリズムの具体的な仕組みを述べるが，その前に元来の Chord における virtual servers を用いた負荷分散について説明しておく．先に説明した通り Consistent Hashing では各ホストは ID 空間内で predecessor node と自ノードに挟まれる領域に存在するオブジェクトを管理するのであった．図 2 にも示した通り，各ホストは円弧に相当する領域を管理している．十分多いオブジェクトがランダムに ID 空間上に配置された場合，各ホストが保持するべきオブジェクトの数は円弧の長さにおよそ比例する．

ではこの円弧の長さはどのような分布を示すのであろうか？結論を言うとこの分布は幾何分布に従う．そのため各ホストが保持すべきオブジェクトの数は大きくばらついてしまうことになる．そこで Consistent Hashing では virtual servers という概念を導入することでこの問題を解決している．ホストを ID 空間上の一点にマッピングするのではなく，ホスト内に $O(\log N)$ 個の virtual servers を生成し，各 virtual server 毎に ID 空間上のランダムな一点をマッピングする．当然ながら各 virtual server が保持すべきオブジェクトの数はおよそ幾何分布に従うが，ホストから見ると各 virtual server 同士でばらつきを相殺し，結果的にほぼ均一な数のオブジェクトが割り当てられることになる．2.2 節で述べた通り，高い確率で各ホストには $(1+\epsilon)L/N$ の負荷がかかることになる．

では Chord においてこの virtual servers による解決方法を適用するとどのような問題があるのだろうか？答えは至極単純で，トポロジー維持コストの増大を招くのである．各 virtual server についてトポロジー維持が必要であれば，$O(\log N)$ 個の virtual servers にはその $O(\log N)$ 倍のトポロジー維持コストが必要である．仮に各 virtual server が 30 秒に一回の割合でトポロジー維持のためのメッセージを送っているとすると，$N = 2^{10}$ のシステム内ではおよそ 3 秒に一度の頻度でメッセージを送らなくてはならない．

提案手法では virtual servers を使わずに負荷分散を実現するというアプローチを用いる．Consistent Hashing で virtual servers を必要としたそもそもの原因はノード (host or virtual server) の管理するべき空間（円弧）の大きさが幾何分布に従っており，その空間を一つのノードで管理していることに起因している．提案手法ではその空間を一つのノードで管理するのではなく $\Theta(\log N)$ のノードで管理することで負荷分散を図る．仮にノード－ノード間の円弧が長くなったとしても，その負荷を複数のホストで分散するので負荷の分散が図れるであろうというのが直観的な解釈である．これまでの Consistent Hashing ではオブジェクトは ID 空間上で次にくるノード (successor) に格納されていたが，提案手法では図 6 に示した通り，successor list 内のノードから選ばれたノードに格納される．

```
SuccessorList* GetSuccessorList(ObjectID objID)
{
    Predecessor* predecessor;
    predecessor = GetPredecessor(objID);
    return( predecessor->GetSuccessorList() );
}

HostList* GetResponsibleHost(ObjectID objID)
{
    HeapTree heap;    HeapNode entry;
    HostList* hlist = new HostList;

    for(int i=0; i<s+1, i++){
      entry.metric = H(objID, node[i]->ID, node[i]->weight);
        entry.node = node[i];
        heap->InsertHeap(entry);
    }

    for(int j=0; j<r; j++)
        hlist->Insert( heap->DeleteMax() );

    return(hlist);
}
```

表 2: 疑似コード

次いで具体的な処理の説明にうつることとする．表 2 にこの処理の疑似コードを示した．ここで各ノードは 30 秒ごとに successor にメッセージを送り，相手のプロセスが起動しているかチェックしているものとする．仮にレスポンスが一定時間帰ってこない場合，以上終了したとみなして近隣ホストに向けてハードステート[2]の告知を行う．これによって各ノードは常に最新の predecessors と successors を知ることができる．

[2]状態に変化があったとき，直ぐに伝達する方式をハードステートという．一方，一定時間ごとに相手の状況をチェックし，そのときに状態の更新を行う方式をソフトステートと呼ぶ

このような状況下でオブジェクト a をシステム内に格納したいノードではオブジェクトの predecessor をまず調べ，その successor list(s+1 個) を取得する．その中の最後のエントリは無視して，s 個のホストの中から $H(a, nodeID_i, w_i)$ の値が最大となる r 個のホストを選び出し，オブジェクトの格納を行う．各ホストの容量が同じ $(w_i = const.)$ という条件下ではある zone の負荷は successor list 内のホストで均一に分散される．

4.5 確率から見た考察

先に述べたアルゴリズムの負荷分散の特性について，確率の観点から検証する．本節では $\Theta(N)$ の successor list を用いて先のアルゴリズムを適用することによって，高い確率でノードの負荷が平均ノード負荷に近い値になることを示す．なお，ここでは簡単のため静的な環境を考えるが，動的な環境についてもほぼ同様の考えが適用できる．また先に述べたアルゴリズムでは各ホストの容量には隔たりがある状況を考慮していたが，ここではすべてのホストが全く均一な負荷になることを目標にした際の負荷分散特性について考慮する．また ID 空間は十分大きく取ってあり，ノード ID の衝突は起こらないものとして考察する．
ではそのような条件下においてあるホスト A から見て s 番目の successor ノード B にかかる負荷 L_b が平均負荷 L_m の $(1+\epsilon)$ 倍より大きくなる確率が低くできることを示す．$L_b \simeq L/(I_U) * (I_b - I_a)/s$ より

$$
\begin{aligned}
& prob\{L_b > (1+\epsilon)L_m\} && (1)\\
= \;& prob\{L_b > (1+\epsilon)L/N\} && (2)\\
\simeq \;& prob\{L/I_U * (I_b - I_a)/s > (1+\epsilon)L/N\} && (3)\\
= \;& prob\{(I_b - I_a) > (1+\epsilon)I_U/N * s\} && (4)
\end{aligned}
$$

つまりノード B にかかる負荷が平均ノード負荷の $(1+\epsilon)$ 倍より大きくなる確率は ID 空間上のノード B とノード A の距離が $I_U/N * s$ の $(1+\epsilon)$ 倍より大きくなる確率に等しい．この確率は $(I_a, I_a + (1+\epsilon)I_U/N * s]$ の空間において存在するホストの数が s 個より小さい確率に等しいので，以下ではこの空間におけるホストの数に関して考察を加える．

この空間上の点においてホストが存在するかどうかは各点独立であり，ベルヌーイ試行に従うと考えられる．したがってこの空間内に存在するホストの数は二項分布にしたがっており，$Bi((1+\epsilon)I_U/N * s, N/I_U)$ の確率変数 X に関して $prob\{X < s\}$ を求めればよい．ここで導き出したい結果は s を $\Theta(\log N)$ にとったときに，この確率が十分小さくできるということである．
幸いにして二項分布に関するこの確率は Chernoff の不等式によって上限を与えられる．

$X_1, X_2, ..., X_n$ が独立なベルヌーイ確率変数であり，$prob\{X_i\} = p_i(0 < p_i < 1, 1 \le i \le n)$ のとき，$X = \sum_{i=1}^{n} X_i, \mu = E[X] = \sum_{i=1}^{n} p_i$ に関して以下の不等式が成り立つ．

$$0 < \delta \le 1, \quad prob\{X < (1-\delta)\mu\} < \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu} \quad (5)$$

$$0 < \delta \le 1, \quad prob\{X < (1-\delta)\mu\} < e^{-\frac{\mu\delta^2}{2}} \quad (6)$$

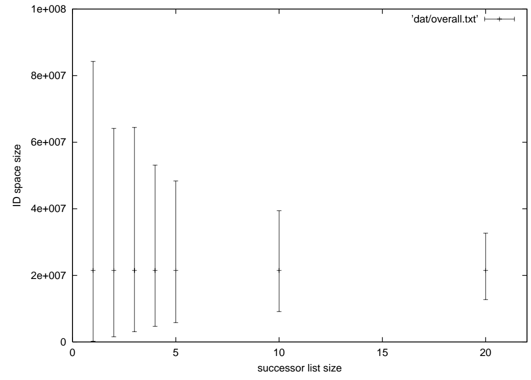

図 7: 負荷のばらつき

上の式の方がより正確な上限を与えるが，ここでは簡単のため下の式を用いることとする．ここで $\delta = \frac{\epsilon}{1+\epsilon}$ となるように δ をとると，$1 - \delta = \frac{1}{1+\epsilon}$ なので

$$
\begin{aligned}
prob\{X < s\} &= prob\{X < \frac{\mu}{1+\epsilon}\} && (7)\\
&= prob\{X < (1-\delta)\mu\} && (8)\\
&< e^{-\frac{\mu(\epsilon/1+\epsilon)^2}{2}} && (9)
\end{aligned}
$$

ここで $\mu = (1+\epsilon)s$ を代入して，

$$prob\{X < s\} < e^{-s\epsilon^2/2(1+\epsilon)} \quad (10)$$

したがって，$s = \left(2(1+\epsilon)/\epsilon^2\right)\ln N$ となるように，s をとれば $prob\{X < s\} < \frac{1}{N}$ となり，非常に高い確率でノード B の負荷は平均負荷の $1+\epsilon$ 倍よりも低くなる．

またこの式において例えば，$N = 10000, \epsilon = 1, s = 2\ln N$ を代入すれば，$prob\{X < s\} < \frac{1}{100}$ となるので，平均負荷の 2 倍以上の負荷となるホストは全体の 1 パーセントとなる．s を Ω にすれば，任意の負荷分散特性を実現することができる．

4.6 評価

先ほど Chernoff の不等式によって示した結果をシミュレーションによって検証した．シミュレーションでは ID 空間を 32bit とし，その空間中に 200 個のホストをランダムに分布させた．図の横軸は successor list の大きさ s，縦軸はホストが担当する空間の広さである．ホストが担当する平均的な空間の広さは $2^{32}/200$ である．図は同じシミュレーションを 20 回繰り返した結果の平均となっている．図には両端 0.5%ずつ除外した 99%のホストが縦の線分として描いた．縦線の上端は負荷が極端に重いホストを示しており，下端は負荷が極端に軽いホストを表している．s が大きくなるにつれ，両端が平均に近づいていくことが分かる．このことから，s の増加に伴い負荷分散の特性が良くなっているといえる．

4.7 関連研究

本稿では virtual servers に基づく負荷分散アルゴリズムとの対比を中心に行ってきたが，現在知られている分散

ハッシュテーブルの負荷分散法として"Power of Two"[1]を利用したものがある．通常のハッシュではオブジェクト a を格納するバケットを決める際，$f(K_a)$ に基づいて決定している．しかし，"Power of Two"では一つのハッシュ関数によって決めるのではなく，$d(d \geq 2)$ 個のハッシュ関数 $f_1(K_a), f_2(K_a), ..., f_d(K_a)$ に対応するホストの中から最も負荷の軽いバケットに実体を格納し，他のバケットにはそのホストへのポインタを格納するというアプローチをとっている．この手法の利点は最も負荷の大きいバケットでもかなり良好な負荷分散特性を示す点である．この手法は virtual servers に基づく手法ではないため，トポロジー維持コストは定性的には提案手法と同じである．

Power of Two で不利な点として，まずオブジェクトの登録コストが通常の d 倍になることが挙げられる．一般にオブジェクトの数はホストの数より格段に多いため，このコストは決して無視できるコストではない．また virtual servers や提案手法ではノード参加・離脱の際，複数のホストで並列にデータ転送が行われるため，高速に修復が行われるが，Power of Two を用いた方法ではデータ転送が一対一で行われるため比較的にゆっくりと行われる．最後に Power of Two では node failure が起きたときポインタが古くなってしまうため，ポインタの修復作業が必要になるという多少複雑な処理が必要になる．

一方，提案手法が virtual servers による手法や Power of Two に比べて問題となりうる点は，システムの動作が successor list の正確さに依存している部分である．提案手法では successor list の管理をハードステートメカニズムを加えることでこれを解決しようとしているが，実際に動的な環境下でどの程度正確に行えるかは今後の検討が必要である．もう一つ不利な点としては他手法ではホストの容量に大きな差がある状況でも対応できるのに対して，提案手法では比較的小さな差についてしか対応できない点である．

5 まとめと今後の課題

本稿では分散的なハッシュ機構について説明し，そのためのオーバーレイネットワークトポロジーの維持コストの削減手法について議論した．提案した手法では virtual servers を利用した負荷分散メカニズムと同等の負荷分散特性を持ちながら，必要なメッセージコストを削減することに成功した．今後の課題としては，提案手法が動的な環境においてどの程度の一貫性を維持できるかについて検証する必要があると考えている．

参考文献

[1] John Byers, Jeffrey Considine, and Michael Mitzenmacher: "Simple Load Balancing for Distributed Hash Tables," In Proceedings of the 2nd International Workshop on Peer-to-Peer Systems(IPTPS2003), March 2003.

[2] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica: "Wide-area cooperative storage with CFS," In Proceedings of the 18th ACM Symposium on Operating Systems Principles(SOSP'01), pp.202-215, October 2001.

[3] Scott Deerwester, Susan T. Dumais, Geroge W. Furnas, Thomas K. Laundauer, and Richard Harshman: "Indexing by Lantent Semantic Analysis," Journal of the American Society of Information Science, 41(6), pp. 391-407, 1990.

[4] John Dilley, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl: "Globally Distributed Content Delivery," IEEE Internet Computing, pp. 50-58, September/October 2002.

[5] Peter Druschel and Antony Rowstron: "PAST : A large-scale, persistent peer-to-peer storage utility," In Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS VIII), pp.75-80, May 2001.

[6] Kevin Fu, M. Frans Kaashoek, and David Mazières: "Fast and Secure Distributed Read-only File System," In Proceedings of the 4th Symposium on Operating Systems and Design & Implementation(OSDI2000), October 2000.

[7] M. Frans Kaashoek and David R. Karger: "Koorde: A Simple Degree-optimal Distributed Hash Table," In Proceedings of the 2nd International Workshop on Peer-to-Peer Systems(IPTPS2003), February 2003.

[8] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, and Daniel Lewin: "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," In Proceedings of the 29th ACM Symposium on Theory of Computing (STOC97), pp.654-663, May 1997.

[9] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummandi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao: " OceanStore : An Architecture for Global-Scale Persistent Storage," In Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS-IX), pp.190-201, November 2000.

[10] Petar Maymounkov and David Mazières: "Kademlia: A Peer-to-peer Information System Based on the XOR Metric," In Proceedings of the 1st International Workshop on Peer-to-Peer Systems(IPTPS2002), March 2002.

[11] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica: "Load Balancing in Structured P2P Systems," In Proceedings of the 2nd International Workshop on Peer-to-Peer Systems(IPTPS2003), March 2003.

[12] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker: "A Scalable Content-Addressable Network," In Proceedings of the Conference of the ACM Special Interest Group on Data Communication(SIGCOMM'01), pp.161-172, August 2001.

[13] Antony Rowstron and Peter Druschel: "Pastry : Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems," In Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), pp.329-350, November 2001.

[14] Ion Stoica, Robert Morris, M.Frans Kaashoek, and Hari Balakrishnan: "Chord : A Scalable Peer-to-Peer Lookup Service for Internet Applications," In Proceedings of the Conference of the ACM Special Interest Group on Data Communication(SIGCOMM'01), pp.149-160, August 2001.

[15] Chuanqiang Tang, Zhichen Xu, and Mallik Mahalingam: "pSearch: Information Retrieval in Structured Overlays," In Proceedings of 1st Workshop on Hot Topics in Networks(HotNets-I), October, 2002.

[16] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph: "Tapestry : An Architecture for Fault-tolerant Wide-area Location and Routing," Technical Report, U.C. Berkeley, CSD-01-1141, April 2000.