情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2009-ARC-183 No.3
Vol.2009-OS-111 No.3
2009/4/22

データのアクセス頻度を考慮した動的負荷分散機構の Dynamo への適用

川上大輔[†1] 松井俊浩[†1] 齋藤彰一[†1]
津邑公暁[†1] 松尾啓志[†1]

Dynamo は，高い可用性と応答性能を実現した分散 key-value store であるが，データのアクセス頻度の違いはあまり考慮されていない．そこで，本研究ではデータのアクセス頻度を考慮した動的負荷分散機構を Dynamo に適用することで，各ノードの負荷の平衡化を図った．その結果，一部のデータにアクセスが偏っている状況において，各ノードの負荷が平衡化され，全体の処理効率の向上を図れることが確認できた．

Application of Load Balancing Mechanism with Considering Data Access Frequency to Dynamo

Daisuke Kawakami,[†1] Toshihiro Matsui,[†1]
Shoichi Saito,[†1] Tomoaki Tsumura[†1]
and Hiroshi Matsuo[†1]

Dynamo is the distributed key-value storage system with high availability and responsibility. But it doesn't consider the difference of data access frequency well. Therefore, we applied the dynamic load balancing mechanism with considering data access frequency to Dynamo to balance the load of each node. In the result, we confirmed the increase of data access efficiency under the condition of unbalanced data access occurring.

†1 名古屋工業大学
Nagoya Institute of Technology

1. はじめに

近年，通信網を介した分散システムの大規模化，高度化を背景に，Farsite[1] や Google file system[2] などの様々な分散ストレージシステムが提案されている．その中の 1 つに，Dynamo[3] がある．Dynamo は，高い可用性と応答性能を実現した分散 key-value store である．また，ノード間の負荷分散も考慮されており，各ノードが読み書きする key-value の数は，概ね均一になっている．以後，この key-value のペアをデータとする．しかし，各データのアクセス頻度の違いはあまり考慮されていないため，データのアクセス頻度が大きく偏る状況において，ノードの負荷が偏り，全体の処理効率の低下を招く可能性がある．そこで，本研究では各データのアクセス頻度の違いを考慮した動的負荷分散手法を Dynamo に適用し，その有効性を評価・検討した．

本稿の構成は次の通りである．まず，第 2 章で分散ストレージシステムの一般的な特徴について説明する．次に，第 3 章で Dynamo のシステムについて説明し，第 4 章でその問題点，及び解決方針を示す．そして，第 5 章で提案手法について説明し，第 6 章の実験で，その有効性を評価・検討する．最後に，第 7 章で本研究のまとめ，及び今後の課題を示す．

2. 分散ストレージシステムの概要

分散ストレージシステムの一般的な特徴として，**図 1** のように，システムが複数の計算機によって構成されていることがある．この分散ストレージシステムを構成する計算機をノードと呼ぶ．クライアントからデータの書き込み要求があった場合は，複数のノードに書き込まれる．読み出し要求があった場合は，それらの 1 つ，または複数のデータが読み出され，クライアントに応答される．分散ストレージシステムでしばしば問題になるのは，システムのスケーラビリティ，応答性能，耐故障性，及びデータの一貫性を同時に実現することである．例えば，クライアントからの書き込み要求を受け付けるノードが複数あった場合，システムのスケーラビリティ，及び応答性能は良くなる一方で，同じデータの書き込み要求が同時に発生した場合，データの一貫性の保持が困難になる．一方で，クライアントからの書き込み要求を受け付けるノードが 1 つしかないと，システムのスケーラビリティ，及び応答性能が悪化することは明らかである．また，そのノードが単一故障点となり，そのノード 1 つが故障するだけで，システム全体が破綻してしまうという問題点もある．Dynamo では，コンシステント・ハッシュ法[4] や，ベクタクロック[5] などの技法を用いることで，これらの問題を解決している．

Vol.2009-ARC-183 No.3
Vol.2009-OS-111 No.3
2009/4/22

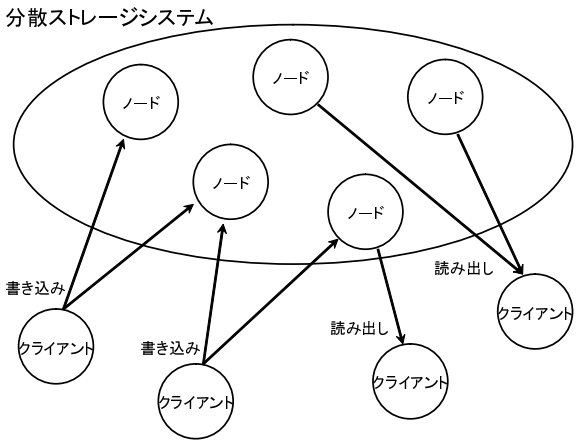

図 1 分散ストレージシステムの概略図
Fig. 1 Image of distributed storage system.

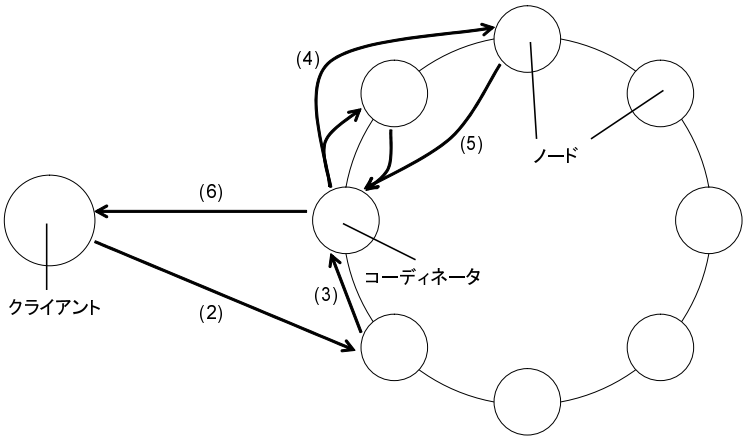

図 2 Dynamo の動作の流れ
Fig. 2 Process flow of Dynamo.

3. Dynamo の概要

ここで，Dynamo の動作の概要について説明する．Dynamo におけるデータの書き込み，及び読み出し処理の大まかな流れを以下に示す．なお，各項目の番号は，それぞれ**図 2** 中の番号に対応する．

(1) Dynamo のシステムには，複数のノードが参加する．

(2) クライアントは，システム中のいずれかのノードに書き込み，又は読み出しリクエストを送る．

(3) クライアントからリクエストを受け取ったノードは，そのリクエストが対象とするデータの読み書きを管理するノードに，リクエストを転送する．
このあるデータの読み書きを管理するノードを，コーディネータと呼ぶ．各データに対応するコーディネータは，基本的に一定である．

(4) コーディネータは，リクエストを受け取ると，実際にそのデータの読み書きを行うノード群にリクエストを転送する．
このデータの読み書きを行うノード群を管理する表をプリファレンスリストと呼び，あるデータに対応するプリファレンスリストに含まれるノード群は，コーディネータと同様，基本的に一定である．

(5) リクエストを受け取ったプリファレンスリスト内のノードは，データの読み書き処理を行い，その結果得られたデータをコーディネータに送る．

(6) コーディネータは，プリファレンスリスト内のノードからデータを受け取ると，そのデータをまとめてクライアントに送る．
この時，コーディネータは，プリファレンスリスト内の全てのノードからデータを受け取るまで待つのではなく，ある一定数のノードからデータを受け取った時点で，クライアントにデータを送る．

以降で，それぞれの動作の詳細について説明する．

3.1 Dynamo システムへの参加，及び離脱

Dynamo では，リング状のハッシュ空間を用いるコンシステント・ハッシュ法という技法によって，システム中のノード，及びデータの読み書きを管理している．

Dynamo のシステムに新しいノードが参加する場合，参加するノードは，システム中の特定のノード，あるいはシステム外の管理ノードに対して，参加する旨を伝える．管理ノードは，そのノードの名前などをキーとしたハッシュ値によって，このハッシュリング上にそのノードを配置する．この時，ノード数が少ないと，配置される位置が偏る．Dynamo において，各データはハッシュリング上のそれぞれの点に対応するため，各ノードの位置が偏

Vol.2009-ARC-183 No.3
Vol.2009-OS-111 No.3
2009/4/22

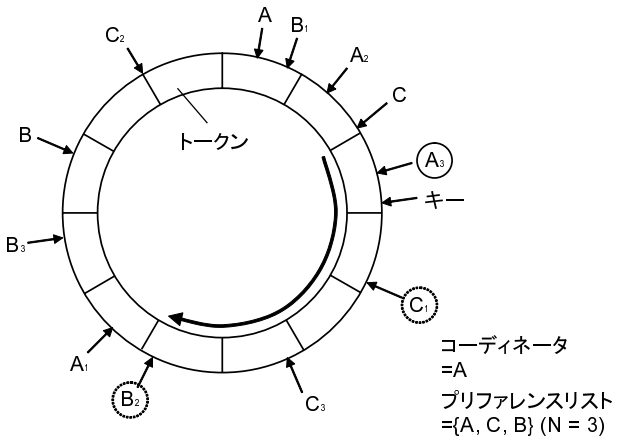

図 3 コーディネータ，及びプリファレンスリストの決定
Fig. 3 Determination of coordinator and preference list.

ると，読み書きするデータの数にもばらつきが生じる可能性がある．

そこで，各ノードで読み書きするデータの数がなるべく均一になるよう，各ノードに対応する複数の仮想ノードを配置する．例えば，ノードの名前をキーとしてハッシュ値を求めていた場合，その名前の最後に 1, 2, ... などの特定の値を付加することによって，異なるハッシュ値を求めることができる．このハッシュ値と元のノードを対応付けたものが，仮想ノードである．

また，システム中のノードのダウンを検知するため，システム中の各ノードは，ランダムに選択したノードに対して，定期的に生存確認を行う．

生存確認できないノードがあった場合は，そのノードはダウンしていると判断し，システムから離脱させる．システムからノードを離脱させる場合は，リング上に配置されたノード，及びそれに対応する仮想ノードを全て除去する．

3.2 コーディネータ，及びプリファレンスリストの決定

Dynamo では，各データにユニークなキーが設定されている．各データのコーディネータ，及びプリファレンスリストは，この値をキーとしたハッシュ値によって決められる．この時，通常のコンシステント・ハッシュ法を用いると，データ毎にコーディネータ，及びプリファレンスリストが異なり，コーディネータやプリファレンスリストの管理が非常に煩雑になる．そこで，Dynamo では，ハッシュ空間を複数の**トークン**と呼ばれる部分空間に分割し，そのトークン単位でコーディネータ，及びプリファレンスリストを管理している．

プリファレンスリスト決定までの具体的なイメージを，**図 3** に示す．まず，データのキーから求めたハッシュ値を Dynamo のハッシュ空間に照らし合わせ，対応するトークンを求める．そのトークンの範囲の先頭から順にハッシュリング上を探索していき，最初に見つかったノード，又は仮想ノードに対応するノードが，そのデータに対応するコーディネータとなる．図の例では，最初に見つかるのが仮想ノード A_3 なので，ノード A がコーディネータとなる．同様に，最初に見つかった N 個のノードが，そのデータに対応するプリファレンスリストに登録される．結果的に，あるデータに対するコーディネータは，そのデータに対するプリファレンスリストにも含まれることになる．この N の値はユーザがあらかじめ指定するもので，多いほど耐故障性が増すが，データの整合性の維持や，プリファレンスリストの管理が煩雑になる．また，この探索中に，もしこれまでに見つかったプリファレンスリストのノードと同じノードに対応する仮想ノードが見つかった場合は，そのノードはプリファレンスリストに登録せず，次のノードを探索する．すなわち，プリファレンスリストには，同じノードを登録しないようにする．図の例では，仮想ノード C_1 の後に仮想ノード C_3 が見つかるが，これらに対応するノードは共通なので，C_3 の代わりに，次の B_2 に対応するノード B がプリファレンスリストに登録される．これは，耐故障性を高めるためである．

3.3 データの書き込み，及び読み出しリクエストの処理

クライアントからのデータの書き込み，及び読み出しリクエストは，まずそのデータのコーディネータに送られ，コーディネータからプリファレンスリスト内の各ノードに転送される．プリファレンスリスト内の各ノードは，要求されたデータの読み出し，及び書き込み処理を実行し，その結果得られたデータをコーディネータに送る．コーディネータは，プリファレンスリスト内の一定数のノードからデータを受け取った時点で，クライアントにデータを送る．全てのノードのデータを待たないのは，ダウンしていたり，負荷が高くてデータの送信が遅れるノードがある場合を考慮しているためである．このコーディネータが返事を待つノードの数は，読み出し処理の場合は R，書き込み処理の場合は W と，ユーザがあらかじめ別々に指定する．例えば，R を減らして W を増やすと，読み出しの性能を重視したシステムになる．いずれにせよ，データの一貫性を保つため，この R と W の和は，プリファレンスリスト内のノードの数よりも大きくなることが推奨されている．

また，読み出し処理の結果については，ノードの離脱や参加などが原因で，各ノードのデータが異なる場合がある．このデータの不整合の解決には，ベクタクロックが用いられ

Vol.2009-ARC-183 No.3
Vol.2009-OS-111 No.3
2009/4/22

不整合データのベクタクロック	解決後データのベクタクロック
{N_1, 2, 1297342100} {N_1, 1, 1210734889}	{N_1, 2, 1297342100}
{N_1, 2, 1284955612} {N_1, 2, 1254132656}	{N_1, 2, 1284955612}
{N_1, 3, 1243672844} {N_1, 2, 1318308409}	解決不可（ユーザ依存）
{N_1, 3, 1236214948} {N_1, 3, 1236214948}, {N_2, 1, 1246938791}	{N_1, 3, 1236214948}, {N_2, 1, 1246938791}
{N_1, 2, 1287392133}, {N_2, 1, 1309837210} {N_1, 2, 1287392133}, {N_3, 1, 1299324026}	解決不可（ユーザ依存）

※ {X, Y, Z} = {コーディネータ, コーディネート数, 最近コーディネート時のタイムスタンプ}

図 4 ベクタクロックを使った不整合データの解決例
Fig. 4 Resolution of inconsistency data with vector clock.

る．このベクタクロックには，そのデータの読み書きをコーディネートしたノードと，コーディネートした回数，及び各コーディネート時のタイムスタンプが記録されている．複数のコーディネータがコーディネートした場合は，そのコーディネータの数だけ記録される．

ベクタクロック，及びベクタクロックによる不整合データの解決例を，**図 4** に示す．1 番目の例のように，ベクタクロックのコーディネータが同じ場合は，コーディネート数が多く，かつ最近にコーディネートされたデータが最新のデータであると判断される．2 番目の例のように，コーディネート数が同じ場合は，最近にコーディネートされたデータが最新のデータであると判断される．3 番目の例のように，もしコーディネート数が多いデータと最近にコーディネートされたデータが異なっていた場合は，ベクタクロックでの不整合の解決は困難であると判断し，ユーザが不整合を解決する．ベクタクロックのコーディネータが異なっていた場合も，同様にベクタクロックによる不整合の解決はできない．また，4 番目と 5 番目の例のように，ベクタクロックに複数のコーディネータが存在していた場合は，各コーディネータについて，それぞれ上記と同様に処理する．

いずれの場合も，不整合が解決されたら，それを不整合が発生していたノードに書き戻すことで，データの整合性を回復する．

4. Dynamo の問題点と解決方針

Dynamo の問題点は，上記のように，あるデータの読み書きを行うノードが常に一定となるため，あるデータのアクセス頻度が他と比べて大きかった場合に，そのデータの読み書きを行うノード，とりわけ，コーディネータとなるノードの負荷が大きくなり，全体的な処理のボトルネックになる可能性があることである．これは，Dynamo がコンシステント・ハッシュ法を利用していることによる．一方で，コンシステント・ハッシュ法を利用することで，ノードの離脱や参加の影響を小さく抑えられるという利点がある．また，あるデータの読み書きを行うノードが一定であるということは，データの一貫性を保持する上では有効である．そこで，基本となるシステムにはそのままコンシステント・ハッシュ法を利用をし，その上で，データのアクセス頻度が偏っていた場合に，負荷分散を行う方法を考える．

5. データのアクセス頻度を考慮した動的負荷分散手法

アクセス頻度が高いデータを多くコーディネートするノードがある場合に，いくつかのデータのコーディネータを別のノードに変更することで，ノード間の負荷分散を行う方法を提案する．以後，この手法を適用した Dynamo を **DAB-Dynamo(Data Access Balanced Dynamo)** と呼ぶ．まず，各トークンのコーディネータ，及びプリファレンスリストを管理する表を作成する．この表は，負荷分散処理時のほか，新たなノードがシステムへ参加したり，システム中からノードが離脱した場合に更新される．次に，全てのノードの中から，1 つの代表ノードを選ぶ．代表ノードは，ノード名をキーとしたハッシュ値の最も小さいもの，あるいは最も大きいものとする．また，全てのノードは，トークンごとにデータの書き込み，及び読み出しリクエストのコーディネートを何回行ったかという負荷情報を記録しておく．負荷情報に実際のデータの読み書きを考慮に入れないのは，データのコーディネート処理量が，実際のデータの読み書きの処理量に比べて十分大きく，コーディネート数だけで全体の負荷を近似できると判断したためである．代表ノードは，定期的にこの負荷情報を要求するメッセージを全てのノードに対して送信する．負荷情報を要求されたノードは，これまでに記録してきた負荷情報を代表ノードに対して送信し，負荷情報を初期化する．全てのノードから負荷情報が得られると，代表ノードはその負荷情報を元に，負荷分散情報を作成する．負荷分散情報を作成するためのアルゴリズムを，**図 5** に示す．

まず，負荷情報から，コーディネート数の最も多かったノード `MaxNode` と，コーディネート数の最も少なかったノード `MinNode` を求める．次に，この 2 つのノードのコーディネート数の差を埋めるため，この 2 つのノードのコーディネート数の差を 2 で割った値 `Threshold` を求め，これを負荷分散対象トークンの決定基準とする．そして，`MaxNode` がコーディネートしたトークンの中で，コーディネート数が多い方から順に探索していき，合計コーディネート数が `Threshold` を超えない範囲で，複数の負荷分散対象トークンを決定し，それら

Vol.2009-ARC-183 No.3
Vol.2009-OS-111 No.3
2009/4/22

```
sort(負荷情報);
MaxNode = max(負荷情報).ノード名;
MinNode = min(負荷情報).ノード名;
Threshold = (MaxNode.負荷値 - MinNode.負荷値) / 2;
foreach(Node = 負荷情報.ノード名) {
  sort(Node.負荷情報)
  foreach(Token = Node.負荷情報.トークン名) {
    if(Token.負荷値 + TotalLoad < Threshold) {
      TotalLoad = TotalLoad + Token.負荷値;
      負荷分散処理対象リストに Token を追加;
    }
  }
  if(負荷分散処理対象リストが空ではない) {
    break;
  }
}
if(負荷分散対象リストが空ではない) {
  負荷分散対象リスト内のトークンのコーディネータを MinNode に変更;
}
```

図 5 負荷分散情報作成のアルゴリズム
Fig. 5 Algorithm of making load balancing information.

のコーディネータを MinNode に変更する．多い方から順に探索するのは，負荷分散対象トークンを減らすことで，負荷分散処理を軽くするためである．もし該当するデータが無ければ，2 番目にコーディネート数の多かったノードに対して，同様の処理を行う．全てのノードについて該当するデータが無ければ，負荷分散は行わない．

具体例を，**図 6** に示す．この場合，総コーディネート数が最も多い N_1 が最大負荷ノードとなり，同じく最も小さい N_3 が最小負荷ノードとなる．この 2 つのノードの総コーディネート数の差 1972 を 2 で割った値は 986 であり，これを閾値とする．次に，最大負荷ノードである N_1 におけるトークンごとのコーディネート数を確認する．ここで，コーディネート数の多いトークンから順に探索していき，合計コーディネート数が閾値 986 未満で最大となる組み合わせを求める．この場合，$660 + 312 = 972 < 986$ が最大となる組み合わせなので，トークン 34 と 45 のコーディネータを N_3 に変更するという負荷情報が作成される．

全ノードの負荷情報

ノード名	総コーディネート数
N_1	2489
N_2	1465
N_3	517
N_4	1772

N_1 の詳細負荷情報

トークン	コーディネート数
45	312
51	1201
34	660
94	238
32	78

最大負荷ノード = N_1
最小負荷ノード = N_3
閾値 = (最大総コーディネート数 – 最小総コーディネート数) / 2 = 986
負荷分散対象トークンリスト(合計コーディネート数 < 986) = {34, 45}

図 6 負荷分散対象トークンの決定例
Fig. 6 Example of determination of load balancing target.

また，負荷分散情報を一度に全ノードに対して適用しようとすると，その適用タイミングがずれ，負荷分散情報を適用したノードと適用していないノードで別々のコーディネータに同時にリクエストを転送する場合があり，多くのデータの不整合の発生が想定される．ベクタクロックで解決できる可能性もあるが，その処理の負担や，上記のように，別々のコーディネータによって書き込まれたデータの不整合は解決できないことを想定すると，データの不整合の発生は極力抑えるべきである．そこで，負荷分散情報の適用には，以下の手法を用いる．なお，各項目の番号は，それぞれ**図 7** 中の番号に対応する．

(1) コーディネータの変更元ノードに対して，負荷分散情報を通知する．
(2) 変更元ノードは，負荷分散情報を受け取ると，自身に負荷分散情報適用した上で，負荷分散情報と，コーディネータの変更対象となるデータをまとめてコーディネータの変更先ノードに送る．
これは，コーディネータの変更先のノードが，変更対象トークンのプリファレンスリストに含まれていない可能性があり，その場合，そのトークンに属するデータを一切持っていないためである．

Vol.2009-ARC-183 No.3
Vol.2009-OS-111 No.3
2009/4/22

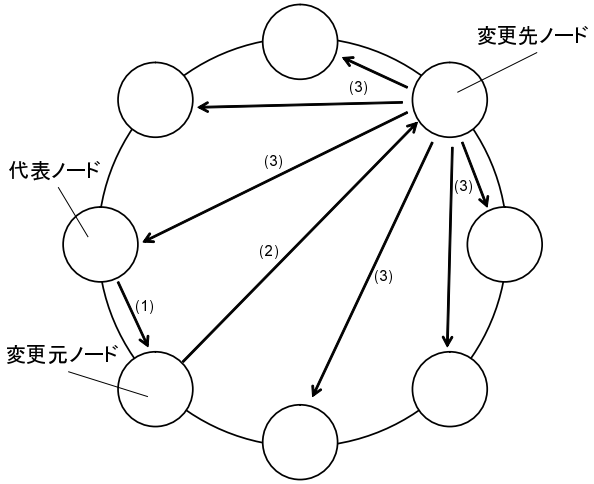

図 7 データの不整合回避機構
Fig. 7 Mechanism of avoidance of data inconsistency.

(3) 負荷分散情報とデータを受け取ったコーディネータの変更先ノードは，受け取ったデータを記録した上で，自身に負荷分散情報を適用し，負荷分散情報を他の全てのノードに送る．

(4) 負荷分散情報を受け取ったその他のノードは，受け取り次第，負荷分散情報を適用する．

この時，コーディネータの変更元ノードが負荷分散情報を適用してから，その他のノードが負荷分散情報を適用するまでの間にタイムラグがあり，その間にその他のノードに届いた負荷分散対象トークン内のデータへのリクエストは，コーディネータの変更元ノードの方に送られてしまう．そこで，ノードにデータのコーディネートリクエストが送られた際に，そのデータのコーディネータが本当に自分であるかどうかをチェックするようにし，自分でなかった場合は，本来のコーディネータにリクエストを転送するようにする．この手法により，その間に発生しうるデータの不整合を回避できるようになる．

6. 実　　験

データのアクセス頻度を考慮した動的負荷分散手法の効果を評価・検討するため，Dynamo, 及び DAB-Dynamo に対して，各データのアクセス頻度が一様である場合と，偏っている場合の 2 種類のリクエストを発生させた．なお，純粋な負荷分散の効果を測定するため，実験中の新たなノードのシステムへの参加，及びシステム中のノードの離脱は無いものとする．その他，主な実験環境，及び実験条件を，それぞれ**表 1**，**表 2** に示す．リクエストに関しては，一般的には書き込みよりも読み出しの回数の方が多いと考えられるので，読み出しと書き込みの割合を 2:1 とした．また，データのキーの範囲とトークン数については，各ノードに対してリクエストが十分に分散する値となっている．プリファレンスリストのノード数，読み出し同期ノード数，及び書き込み同期ノード数については，Dynamo で一般的に使われる値を用いた．負荷分散の頻度については，十分に負荷情報が集まり，データの読み書きに影響を与えない範囲でなるべく早く負荷を分散させるため，1 回/秒とした．その他の細かいパラメータ，及び実装方法については，Kai[6] を参考にした．

表 1 実験環境
Table 1 Experimentation environment.

CPU	Intel Pentium 4 3.0GHz
メモリ	2GB
LAN	Ethernet(100Mbps)
言語	Erlang[7] ver. 5.6.3

表 2 実験条件
Table 2 Experimentation condition.

ノード数	8
クライアント数	256(32 クライアント× 8 ノード)
リクエスト数	5000/クライアント (読み出し:書き込み=2:1)
データのキーの範囲	0〜4095
ハッシュ関数	MD5(上位 32 ビット)
ハッシュ空間	0〜2^{32}-1
トークン数	256
プリファレンスリストのノード数	3
読み出し同期ノード数	2
書き込み同期ノード数	2
負荷分散の頻度	1 回/秒

6.1 実 験 結 果

処理時間の測定結果を**表 3** に示す．なお，表中の「一様」とは，全データに対してランダムにリクエストを発生させた場合で,「偏り」とは，0 以上 1 未満の乱数を 4 乗した後，デー

Vol.2009-ARC-183 No.3
Vol.2009-OS-111 No.3
2009/4/22

表 3 実験結果（処理時間）
Table 3 Experimentation result (execution time).

Dynamo		DAB-Dynamo	
一様	偏り	一様	偏り
59.4 秒	70.2 秒	59.1 秒	62.3 秒

タの数 (4096) を掛けることで求めた値のデータに対して，リクエストを発生させた場合である．この場合，全リクエストの約 12%が，最も頻度の高いデータに対するリクエストとなる．

表のように，偏ったデータに対するリクエストを発生させた場合，ランダムなデータに対するリクエストを発生させた場合と比較して，Dynamo では約 11 秒処理時間が増加しているのに対し，DAB-Dynamo では，約 3 秒の増加にとどまっている．また，ランダムなデータに対するリクエストを発生させた場合の処理時間がほぼ同一であることから，負荷分散のコストは無視できるぐらいに小さいと考えられる．

次に，各ノードの平均処理リクエスト数，及び平均ロードバランスを**図 8** から**図 11** に示す．なお,「一様なリクエスト」と「偏ったリクエスト」は，それぞれ前述したランダムなデータと偏ったデータに対するリクエストを発生させた場合である．また，処理リクエスト数とは，コーディネートしたリクエストの数のことで，ロードバランスとは，各ノードの平均処理リクエスト数を，最大処理リクエスト数で割ったものである．図のように，DAB-Dynamo では，時間が経過するにつれ，各ノードの負荷が分散され，各ノードの平均処理リクエスト数，平均ロードバランス共に向上している様子が確認できる．また，Dynamo において，ランダムなデータに対するリクエストを発生させているにも場合にも多少の負荷の偏りが見られるが，これはコンシステント・ハッシュ法の性質により，必ずしも各ノードがコーディネートするデータの数が一様とならないためである．

7. まとめと今後の課題

本研究では，分散 key-value store の Dynamo にデータのアクセス頻度を考慮した動的負荷分散機構を適用することで，Dynamo のシステム中の各ノードの負荷の平衡化を図った．その結果，一部のデータにアクセスが偏っている状況において，各ノードの負荷が平衡化され，全体の処理効率の向上を図れることが確認できた．

今後の課題として，新たなノードのシステムへの参加，及びシステム中のノードの離脱が発生する状況下での評価，及び負荷分散手法の改良が挙げられる．

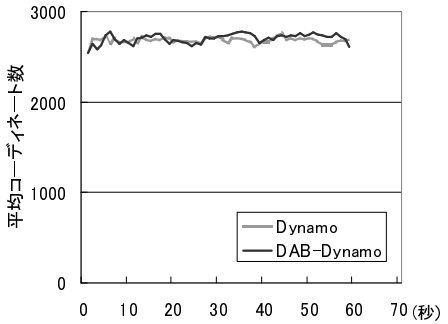

図 8 各ノードの平均処理リクエスト数
（一様なリクエスト）
Fig. 8 Number of average process requests (balanced request).

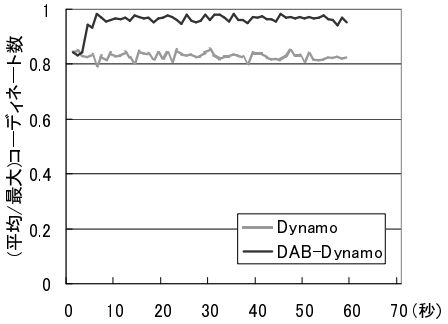

図 9 各ノードの平均ロードバランス
（一様なリクエスト）
Fig. 9 Average load balance (balanced request).

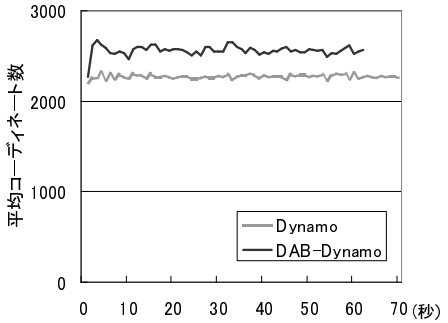

図 10 各ノードの平均処理リクエスト数
（偏ったリクエスト）
Fig. 10 Number of average process requests (balanced request).

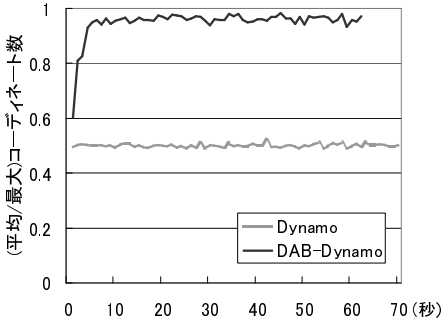

図 11 各ノードの平均ロードバランス
（偏ったリクエスト）
Fig. 11 Average load balance (balanced request).

Vol.2009-ARC-183 No.3
Vol.2009-OS-111 No.3
2009/4/22

参 考 文 献

1) Adya, A., Bolosky, W.J., Castro, M., Cermak, G., Chaiken, R., Douceur, J.R., Howell, J., Lorch, J.R., Theimer, M. and Wattenhofer, R.P.: FARSITE: Federated, available, and reliable storage for an incompletely trusted environment, *In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pp.1–14 (2002).
2) Ghemawat, S., Gobioff, H. and Leung, S.-T.: The Google File System, *In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, pp.29–43 (2003).
3) Hastorun, D., Jampani, M., Kakulapati, G., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W.: Dynamo: amazon's highly available key-value store, *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ACM, pp.205–220 (2007).
4) Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D. and Panigrahy, R.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web, *In ACM Symposium on Theory of Computing*, pp.654–663 (1997).
5) Lamport, L.: Time, clocks, and the ordering of events in a distributed system, *Commun. ACM*, Vol.21, No.7, pp.558–565 (1978).
6) たけまる：Kai. `http://teahut.sakura.ne.jp/b/cat_kai.html`.
7) Armstrong, J.: Making reliable distributed systems in the presence of hardware errors, PhD Thesis, The Royal Institute of Technology Stockholm, Sweden (2003).