2005－EVA－13（2）
2005／6／24

サービス時間のばらつきを考慮した クラスタシステムにおける負荷分散方法の提案と評価

佐々木盛朗　　田中淳裕
NEC システムプラットフォーム研究所
{s-sasaki@di, a-tanaka@dc}.jp.nec.com

概要

クラスタシステムは、ウェブサービスのような電子サービスを提供する基盤として広く用いられている。電子サービスにおいては、大部分のリクエストを要求されたレスポンスタイム内で処理するために必要なキャパシティを、システムは持つべきである。ノード間の負荷が不均衡である間はキャパシティが低下するため、システムのキャパシティはノード数だけでなく負荷分散アルゴリズムにも依存する。負荷不均衡によるキャパシティ減少を解決するために、我々は、仮想的な距離に基づいてリクエスト移送を行う負荷分散アルゴリズム、”Nearest Underloaded algorithm (N algorithm)”を提案する。そして、N algorithm は 25%少ないノード数で従来のリクエスト振分アルゴリズムと同等のキャパシティを達成できる場合があることと、従来のリクエスト移送アルゴリズムより多くのリクエストを同じノード数で処理できることを、評価結果で確認した。

A Load Balancing Algorithm for Different Service Times and Its Evaluation

Shigero Sasaki　and　Atsuhiro Tanaka
System Platforms Research Laboratories, NEC
{s-sasaki@di,a-tanaka@dc}.jp.nec.com

Abstract

Cluster systems are prevalent infrastructures that offer e-services, such as those on the Web. Most requests in these systems should be completed within a reasonable period of time to satisfy clients. The capacity of a system depends on the load balancing algorithm as well as the number of nodes because capacity remains low as long as load among the nodes remains imbalanced. To solve the problem of load imbalance, we propose a load balancing algorithm, the Nearest Underloaded algorithm (N algorithm). It is applied to each node and transfers requests to other nodes in order of virtual node distance. One of evaluation results shows that, with 25% fewer nodes, the N algorithm achieved almost the same capacity as a conventional load balancing algorithm which dispatches requests without transferring them. The results also demonstrated that more requests are processed for all workloads with the N algorithm than with the conventional algorithm that transfers requests when the cluster system has nodes of the same number.

1. はじめに

近年、ウェブサービスのような電子サービスを提供する基盤として、クラスタシステムが広く用いられている。クラスタシステムは（計算）ノードとスイッチ、ストレージ、サーバーアプリケーションから構成される。アプリケーションには、ウェブサーバやアプリケーションサーバ、データベースサーバ等がある。

システムは到着するリクエストを処理するのに十分なキャパシティを持つ必要がある。キャパシティが不足すると、クライアントが待ちきれないほどレスポンスタイムが長くなってしまい、機会損失につながるからである。クラスタシステムのキャパシティを増加させる方法の一つはノードの数を増やすことだが、ノード数を増やすとシステムコストが上昇してしまう。そのため、クラスタシステムはできるだけ少ないノードで要求されるキャパシティを達成できるのが望ましい。クラスタのノード数は、ベンチマーク等で使われるダミーのワークロードを使って求めたキャパシティや実システムの運用結果に従って、要求されたキャパシティを満たすように、経験的に決定されることが多い。

クラスタシステムのキャパシティは、ノード数だけでなく負荷均衡の度合いにも依存する。なぜなら、負荷不均衡状態においては高負荷ノードと低負荷ノードが混在し、低負荷ノードは有効に利用されていないためである。負荷不均衡の原因は、リクエストの到着と出発である。到着したリクエ

ストをノードに振り分けることで到着による不均衡は解消できるが、出発による不均衡はリクエスト振分では解消しにくい。なぜなら、不均衡を埋めるだけの到着が必要とするからである。さらに、高負荷ノード上のリクエストのレスポンスタイムを改善できないという問題がある。出発による不均衡を解消する方法の一つは、高負荷ノード上のリクエストを低負荷ノードに移送することである。負荷をより均衡させるには、到着・出発、双方による不均衡を解消するのが望ましい。

リクエスト移送においては、各ノードに既に到着したリクエストを移送するため、個々のノードが移送の判断を行うのが一般的である。そのため、高負荷ノードが独立してリクエスト移送を行った結果、多数の高負荷ノードから少数の低負荷ノードへのリクエスト移送が起き得る。すると、低負荷ノードは高負荷になってしまい、リクエスト移送後も負荷が均衡しないという問題が生じる。ただ単にリクエスト移送を行うだけでは、負荷は均衡しないのである。

キャパシティを向上させ、できるだけ少ないノードで必要なキャパシティを達成するために、我々は、Nearest Underloaded Algorithm (N algorithm)を提案する。N algorithm はノード間に設けた仮想的な距離に基づいてリクエストを移送し、高負荷ノードは、自ノードよりも近い高負荷ノードからの負荷移送を受けたあとも低負荷でいるノードにのみ負荷を移送する。評価では、N algorithm と既存のアルゴリズムを用いて、8 ノードから 16 ノードまでを含むクラスタシステムのキャパシティを測定した。測定では 3 種類のワークロードを用いた。評価結果は、N algorithm は既存のアルゴリズムよりも大きいキャパシティを達成できる、または、より少ないノードで同等のキャパシティを達成できることを示した。

以下本稿では、2. 節で解決すべき課題について詳しく述べ、3. 節で N algorithm を導入する。4. 節では評価結果を示し、その考察を行う。5. 節では結論を述べる。

2. 課題と性能指標

クラスタシステムのキャパシティを増加させ、目標キャパシティを達成するために必要なノード数を削減するためには、クラスタを構成するノードの負荷が均衡させることが重要である。負荷の不均衡はリクエストの到着と出発によって生じる。到着による不均衡は、最も負荷の低いノードにリクエストを振り分けることで十分に解消できる。しかし、出発による不均衡を迅速に解消することは一般に困難であり、また、高負荷ノードに到着したリクエストのレスポンスタイムが改善されることはない。

リクエストの出発による不均衡を解消する方法の一つは、高負荷ノードのリクエストを低負荷ノードに移送することである。既にノードに到着したリクエストを移送するため、移送の判断は個々のノードが独立に行うことが多い。そのため、多数の高負荷ノードからの少数の低負荷ノードへのリクエストが起きることで、低負荷ノードが高負荷になってしまって負荷が均衡しない場合がある。これは群集効果と呼ばれ、不適切なリクエスト移送の一例である。

図 1 は、自ノードとリクエスト数が最小であるノードのリクエストの数の差の半分を移送するリクエスト移送アルゴリズムを用いた場合の群集効果の例である。一度に移送するリクエストの数を減らすことで群集効果を緩和することができるが、その場合、負荷均衡状態に遷移するまでに何回かのアルゴリズムの適用を要求するため、負荷の状態によっては均衡状態に遷移するまでに長い時間がかかってしまう。キャパシティという観点から見ると、遷移の遅い負荷移送は、群集効果と同様に不適切な負荷移送である。望ましいのは高速に負荷均衡状態に遷移させるアルゴリズムである。

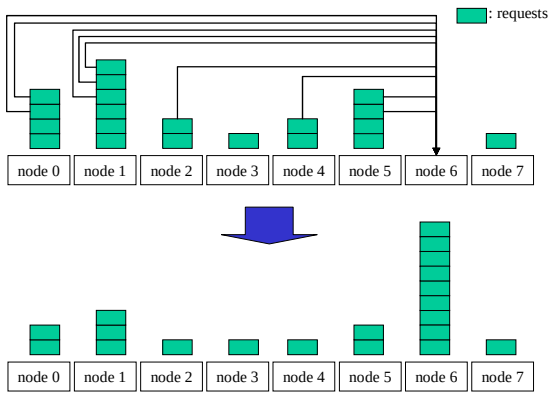

図 1　群集効果

我々の目標は、可能な限り少数のノードで要求されたキャパシティを達成することである。よって、十分短いレスポンスタイムを守りつつ、どれだけの数のリクエストを処理できるか[1]が性能指標となる。レスポンスタイムにも色々あるが、本稿では特に *x* パーセントレスポンスタイム（*x*th PRT）に上限を設けた。一部のリクエスト、5%かもしれないし、1%か 0.5%かもしれない、は状況次第ではレスポンスタイムが長くても許容される。例えば平均レスポンスタイムを用いた場合、レスポンスタイムがある値以上になるリクエストの割合はわからない。そこで *x*th PRT に基づいてキャパシティを算出する。このキャパシティを *x* パーセントキャパシティ（*x*th PC）と呼ぶ。

リクエストの出発による負荷不均衡が生じるのは、少数の到着しかない期間において、各ノードにおける出発数が異なるときである。つまり、バースト到着によるリクエスト到着の偏りと、リクエストサイズ（リクエストのサービス時間）のばらつきの大きさが、リクエストの出発による負荷不均衡に強く影響する。このような性質を持つリクエストの一例としては、HTTP リクエストがある。本稿では、ワークロードは *x*th PC に影響を与えるパラメータによって特性化されるべきである、つまりリクエストの出発による不均衡を生み出すパラメータによって特性化されるべきである、と考え、リクエストサイズの分布のばらつきとリクエスト到着のバースト性によってワークロードを特性化する。

3. 負荷分散アルゴリズム

本節では、既存のアルゴリズムと我々がシステムにおいた仮定について述べる。そして、提案アルゴリズムである Nearest Underloaded algorithm (N algorithm)について述べる。

3.1. 負荷分散アルゴリズムの分類

ここでは、負荷分散アルゴリズムの差異を明らかにするために、アルゴリズムを下記の 5 つのポリシーに分解した上で、既存のポリシーについて説明する[2][3]。

- Initiation policy: 負荷分散を行う主体は何か
- Activation policy: いつ負荷分散アルゴリズムが起動されるか
- Information policy: 負荷指標は何で、いつ更新されるのか
- Transfer policy: どのリクエストを移送するか
- Placement plicy: どのノードの負荷情報が入手でき、どのノードでリクエストを処理するのか

Initiation policy は大きく集中ポリシーと分散ポリシーに分けられる[3]。集中ポリシーの下では、一つのノードやロードバランサーによってリクエスト振分が行われる。分散ポリシーの下では、各ノードがリクエスト移送を行う。分散ポリシーはさらに移送元主導ポリシーと移送先主導ポリシーに分類される[5]。しかし、本稿では、以降は移送元主導ポリシーについてのみスコープに含める。両者の差は本稿では本質的ではないためである。Activation policy は、タイムドリブンポリシーまたはイベントドリブンポリシーのいずれかになる。タイムドリブンポリシーは周期的なアルゴリズムの起動を意味することがほとんどであり、イベントドリブンポリシーは、負荷の変化によってアルゴリズムを起動することが多い。Information policy において、負荷指標としてよく用いられるのはリクエストのキュー長である[6]。CPU または他のリソースの使用率もよく用いられる。負荷指標の更新は activation policy で示した方法が用いられる。

Transfer policy は、全てのリクエストを等価だとみなした場合、いくつのリクエストを移送するかを決定するポリシーである。移送するリクエストの数は、移送元ノードと移送先ノードの負荷の差に基づいて経験的に決定されるのが一般的である[7][8]。リクエストは等価ではないとみなし、リクエストをプリエンティブに移送する場合に関しては[9]に示されるポリシーがある。[3]や[7]においては、リクエストは非プリエンプティブに移送されることに注意する。本稿では非プリエンプティブ移送のみを扱い、リクエストは等価であるとみなす。多くの placement policy はノードスコープ内で最も負荷の低いノードを移送先として選択する。ノードスコープは、移送元ノードが負荷情報を共有しているノードの集合を示す。ノードスコープの例としては、[8]、[10]、[11]等のようにランダムに選択されたノードの集合がある。[12]では、各ノードがランダムに選択したノードの負荷を取得し、自ノードが高負荷でランダム選択されたノードが低負荷であれば、そのノードをスコープに含める。逆に、低負荷ノードは高負荷ノードをスコープに含める。

3.2. システムにおいた仮定

我々がターゲットとするクラスタシステムは、同種のステートを持たないノードから構成される。例えば、ウェブサーバや科学技術計算用途のサーバからなるクラスタシステムがそれに該当する。より詳細には、以下の仮定が成り立つシステムがターゲットシステムである。

1. クラスタシステムは等価な（計算）ノードからなり、ノードがボトルネックになる
2. 単一のサーバーアプリケーションが動作する
3. サーバーアプリケーションは状態を持たない
4. リクエスト間に依存性はない
5. 同時に処理されるリクエストは m(≥1)個以下
6. リクエスト移送のオーバーヘッドは小さい

ウェブサーバはステートを持たず、クッキーが全てのノードで同じ方法で処理される限りは、ウェブリクエスト間には依存性はないことに注意する。仮定 5 は、ノードに k 個のリクエストがある場合、$(k-m)$リクエストが移送可能であることを意味する。また、ウェブリクエストは単なる短いテキストであり、その移送には限られた CPU 時間とネットワーク帯域しか消費されない。

3.3. Nearest underloaded algorithm

ここでは、N algorithm の transfer policy と placement policy について詳細に述べる。N

algorithm の transfer policy では、移送されるリクエストの数は、各ノードが持っているリクエストの平均値と移送元ノードが持っているリクエストの数の差である。そして、平均よりも多い数のリクエストを持つノードは高負荷ノードとみなし、平均よりも少ない数のリクエストを持つノードを低負荷ノードとみなす。N algorithm の placement policy は、ノード間にもうける仮想的な距離に基づいて、高負荷ノードが近いノードから順にリクエストの移送数を決定していく。現在のターゲットノードが低負荷であれば、ターゲットノードを高負荷にしない範囲で負荷を移送する。高負荷ノードが高負荷でなくなった時点でそのノードからの負荷移送は終了する。

N algorithm の下では、ノードは絶対ノード ID と相対ノード ID とを持つ。n をクラスタを構成するノードの数として、ノードに 0 から$(n-1)$までの数字を割り当て、これを絶対ノード ID とする。相対ノード ID は、二つのノードの絶対ノード ID の差で表される。ノード i から見たノード j の相対ノード ID は$(i-j+n)\%n$ で与える。例えばノード数が 8 であると仮定すると、ノード 3 から見たノード 5 の相対ノード ID は 2 であり、ノード 7 から見たノード 5 の相対ノード ID は 6 である。

N algorithm のアルゴリズムを以下に示す。負荷指標は整数または実数を想定しているが、いずれにせよ本質的な違いはないため、負荷指標は整数であることを仮定する。以下のステップでは、ノード ID は相対ノード ID であることに注意する。

1. 負荷の平均値 m を算出する
2. 自ノードの負荷と $ceil(m)$の差を $over$ とする
3. $over$ が正でなければアルゴリズムを終了する
4. ターゲットノードの相対ノード ID を表す t を 1 で、ターゲットノードへの負荷の移送量を表す under を 0 で初期化する。
5. m とノード t の負荷の差を $under$ に加える
6. $floor(under)$が正であれば、$floor(under)$と $over$ の小さい方の分の負荷をノード t に移送する
7. $over$ が $floor(under)$よりも小さければ、アルゴリズムを終了する
8. $over$ と $under$ の両方から、$floor(under)$を引き、t をインクリメントする。t が n になったのであれば、アルゴリズムを終了する
9. ステップ 5 に戻る

負荷を均衡させるのに不必要な負荷移送をさけるため、高負荷ノードは平均以上である負荷の分のみを移送する。N algorithm においては、ノードの過負荷部分は $over$ で表現される。各高負荷ノードは、過負荷部分を移送するため、これらのノードへの負荷移送が起きなければ、負荷移送後に高負荷ではなくなる。一方、低負荷ノードへの過度な負荷移送が起こると、低負荷ノードを高負荷にしてしまうため、負荷が均衡しない。これを防ぐために、N algorithm ではより近いノードからの負荷移送を優先する。$under$ は自ノードよりも近いノードからの負荷移送が発生した後においてターゲットノードにどれ位の負荷を移送しても高負荷にならないかを表す。

N algorithm が各ノードで同時に起動された場合、負荷指標、つまりリクエスト数は平均化される。現在の負荷指標は時間的な遅れのために不正確であり、起動の同時性は保証されないが、これらの問題は本稿のスコープ外とし、情報の正確さと起動の同時性については、キャパシティの観点からのみ扱う。

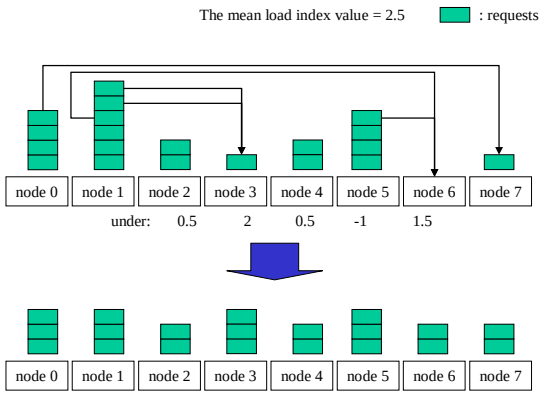

図 2　N algorithm の下での負荷状態の遷移

図 2 に、N algorithm の下での負荷均衡状態への遷移の動作例を示す。図 2 は、8 ノードクラスタ上に 20 個のリクエストがある場合の例であり、ノードには絶対ノード ID が振られている。リクエスト数を負荷指標としているので、負荷指標の平均値は 2.5 である。ノード 2 から 6 の下に示された値は、ノード 1 がこれらのノードをターゲットノードとした時の under の値である。

4. 評価

本節では、リクエスト振分がリクエスト移送よりも平均レスポンスタイムに強く影響し、リクエスト移送は xth PC を増大させることを示す二つの評価結果を示す。xth PC の増大は特に N algorithm の下で顕著に見られた。

4.1. 評価環境

評価は、ギガビットイーサネットで接続された 16 台のノードからなるクラスタ上で行った。ノードの CPU は Xeon 2.4GHz であり、4GB のメモリ容量を持つ。ノードの OS は Linux2.4.7 を使用した。各ノードでは、我々が実装した負荷分散デーモンである Server Wrapper Daemon (SWD)とダ

ミーアプリケーションを動作させた。さらに、簡単な負荷発生プログラムを実装し、これによって負荷を発生させた。以下、実装したソフトウェアと合成したワークロード、評価対象とした負荷分散アルゴリズムについて述べる。

4.1.1. 実装したソフトウェア

SWD はリクエスト移送を可能にするデーモンであり、クラスタの各ノード上で動作し、クライアントからアプリケーションへのリクエストをインターセプトする。インターセプトされたリクエストはアプリケーションまたは他ノードで動作する SWD に移送される。SWD はアプリケーションまたは他ノードの SWD からリプライを受け取ると、クライアントまたは移送元ノードで動作する SWD にリプライを転送する。SWD は与えられた負荷分散アルゴリズムにしたがってリクエストの移送先を決定する。SWD はリクエストのキューを持ち、キューの先頭のリクエストをアプリケーションに移送し、キューの最後尾のリクエストを他ノードで動作する SWD に移送する。同時にアプリケーションに転送するリクエストの数は、本評価では 1 とした。アプリケーションで実際に処理されているリクエストは移送できないことに注意する。負荷移送の機能に加えて、SWD は負荷情報を他の SWD と共有する機能を持っている。本評価での負荷情報とはリクエスト数を指す。

負荷発生プログラムは HTTP リクエストを発行し、そのリプライを受け取る。負荷発生プログラムはリクエスト振分機能を併せ持つ。ダミーアプリケーションは、HTTP リクエストを受け取り、リクエストに指定された回数だけビジーループをまわり、リプライを返す。

4.1.2. ワークロード

本評価では、平均リクエストサイズが 40ms である三つの合成ワークロードを用いた。平均リクエストサイズは、[13] の fine-grain trace と medium-grain trace の間の大きさである。第一のワークロードを標準ワークロードと呼ぶ。標準ワークロードにおいては、リクエストはポワソン到着し、リクエストサイズは対数正規分布に従う。この分布の標準偏差と平均リクエストサイズの比は、[14]のファイルサイズの分布にならって 2.16:1 にした。第二のワークロードを分散ワークロードと呼ぶ。分散ワークロードと標準ワークロードでは、リクエストサイズの標準偏差が異なり、2.16 のルート 2 倍の 3.06 になっている。第三のワークロードはバーストワークロードと呼ぶ。バーストワークロードと分散ワークロードの差異は、バーストワークロードでは一度に 4 つのリクエストが発生する点である。

4.1.3. 負荷分散アルゴリズム

本評価で対象とする負荷分散アルゴリズムは、ラウンドロビン(RR)、集中型(CT)、最低負荷(LL)、N algorithm (N)の 4 つである。これらのアルゴリズムのポリシーを表 1 に示す。Information policy の上段には負荷指標を、下段には情報の更新方法が示した。CT の下では負荷情報はリクエストの到着または出発が起こったときに負荷情報が更新される。また、Transfer policy の上段にはどのリクエストが移送され、下段にはいくつのリクエストが移送されるかが示されている。本稿においては、LL は自ノードと最低負荷のノードのキュー長の差の半分未満のリクエストを移送する。LL と N においては、リクエストはラウンドロビンで各ノードに到着する。

表 1　評価対象の負荷分散アルゴリズム

	RR	CT	LL	N
initiation policy	centralized		distributed	
activation policy	event-driven (on arrival)		time-driven (each 40ms)	
information policy	N/A	the length of the request queue		
(index & update)	N/A	event-driven	time-driven (each 20ms)	
transfer policy	N/A		the request at the tail of the queue	
(which & how many)			difference based	described in Section 3.3
placement policy	round robin	least-loaded	least-loaded	

4.2. 平均レスポンスタイム

我々は、16 ノードクラスタ上で 4 つのアルゴリズムに対して平均レスポンスタイムを測定した。リクエスト到着率は 40 req/sec 刻みで 40 req/sec から 360 req/sec まで変化させた。これは、CPU 使用率を 10%刻みで 10%から 90%まで変化させることに相当する。

表 2 は、標準ワークロードに対する平均レスポンスタイムである。CT の平均レスポンスタイムが最も短い理由は、リクエストの到着または出発とともに負荷情報が更新されるために情報が正確なのと、リクエスト移送と情報共有のオーバーヘッドがないためである。N と LL は、20ms 間隔で負荷情報を交換するため、より不正確な情報に基づいてリクエスト移送を行う。ただし、N は迅速に負荷不均衡を解消できるため、LL よりも CT に近い平均レスポンスタイムを達成できた。最も平均レスポンスタイムが長かったのは RR だった。その理由は、RR は動的な負荷情報を一切考慮しないためである。360 req/sec の時の LL の平均レスポンスタイムが測定されていないのは、LL が高負荷に耐えられなかったためである。我々は、16,384 本のコネクションが張られた時に負荷に耐えられなくなったと判断して測定を中止した。

表 3 は分散ワークロードに対する平均レスポンスタイムである。同じリクエスト到着率、同じ

アルゴリズムであれば、標準ワークロードに対する平均レスポンスタイムに対してよりも、分散ワークロードに対する平均レスポンスタイムの方が長くなった。LL の下では 320 req/sec にも耐えられなくなった。

表 4 は、バーストワークロードに対する平均レスポンスタイムである。バーストワークロードに対しては、CT のレスポンスタイムの悪化が N と LL のレスポンスタイムの悪化に比べて大きいことが見て取れる。リクエストのバースト到着は、先に到着したリクエストが終了した、という情報を得にくくするため、CT では扱いにくい到着パターンである。RR のレスポンスタイムは悪化していない。リクエストがバースト到着しても、静的な負荷情報のみを扱うリクエスト振分を行っている場合、リクエストの振分先は変化しないためである。N と LL の下では、レスポンスタイムはそれほど悪化していない。これは RR によって解消しきれなかった負荷の不均衡をリクエスト移送によって解消しているためである。

表 2　標準ワークロードに対する平均レスポンスタイム

	40	120	200	240	280	320	360
RR	41.6	64.88	105.68	139.73	198.63	345.53	728.39
CT	39.16	40.03	41.36	44.29	50.08	62.09	97.01
LL	41.68	51.53	60.92	67.83	79.84	112.13	N/A
N	39.8	43.53	50.87	59	70.99	94.55	162.17

表 3　分散ワークロードに対する平均レスポンスタイム

	40	120	200	240	280	320	360
RR	44.53	93.09	165.11	223.2	327.89	610.25	1253.95
CT	38.8	39.81	42.31	45.74	53.38	71.56	119.59
LL	43.22	55.52	65.37	72.54	88.12	N/A	N/A
N	39.79	44.05	52.51	62.08	77.39	109.95	210.84

表 4　バーストワークロードに対する平均レスポンスタイム

	40	120	200	240	280	320	360
RR	52.86	108.24	202.57	273.44	378.96	568.99	1156.83
CT	39.99	44.75	55.26	64.74	78.15	100.45	157.49
LL	46.42	57.16	69.5	81.05	101.83	N/A	N/A
N	41	45.12	56.84	69.88	87.5	123.91	226.31

4.3. *x*th percentile capacity

ここでは、CT, LL, N の下でのクラスタシステムの *x*th PC について述べる。我々が実際に測定したのは、8〜16 ノードのクラスタ上でリクエスト到着率は変化させたときのレスポンスタイムである。このレスポンスタイムから *x*th PRT を算出し、隣接する測定点の間においては、リクエスト到着率と *x*th PRT の間に比例関係が成り立つとして *x*th PC を算出した。以下で述べる評価結果は、CT, LL, N のアルゴリズムの下で 3 つの合成ワークロードに対する *x*th PC を求めたものである。*x*th PRT の上限は、16 ノードクラスタに N algorithm を適用して、320 req/sec の負荷をかけたときの *x*th PRT に設定した。

図 3 に、95th PRT の上限を 279ms としたときの、標準ワークロードに対する 95th PC を示す。図 3 で最も大きい 95th PC を示したのは CT であった。16 ノードクラスタにおいては、CT は N に対して 4.7%大きいキャパシティを達成し、N は LL に対して 8.8%大きい 95th PC を達成した。図 4 は、95th PRT の上限は 341ms としたときの、分散ワークロードに対する 95th PC を示す。図 4 では、図 3 に比べると CT と N または LL の 95th PC の差が縮まっている。これは、ワークロードにおけるリクエストサイズのばらつきが大きい状況では、リクエスト移送を行わなければ、行う場合に比べて *x*th PC がより大きく減少することが原因である。LL を用いたときの 16 ノードでの 95th PC がプロットされていないのは、LL が対応する負荷に耐えられなかったためである。図 5 は、95th PRT の上限を 391ms としたときのバーストワークロードに対する 95th PC である。図 5 では、N の下での 95th PC は CT の下での 95th PC とほぼ同じになっている。また、LL と CT の 95th PC の差もより小さくなっている。リクエストのバースト到着は、先に到着したリクエストの終了を待たずにリクエスト振分の判断を行う必要を生じさせるため、リクエスト振分では扱いにくい。そのため、バースト到着によってリクエスト移送を行わない CT は、リクエスト移送を行う N と LL よりも大きく 95th PC が減少する。

95th PRT を制限するのが適切な場合もあるが、より多くのレスポンスタイムに制限をつけるのが適切な場合もある。そこで、99th と 99.5th PC も計算した。元になるレスポンスタイムは 95th PC の計算に使ったものと同じである。

図 6 は、99th PRT の上限を 517ms としたときの標準ワークロードに対する 99th PC である。図 6 で、最も大きい 99th PC を示したのは N を用いた時であった。また、既存のリクエスト移送アルゴリズムである LL を用いた場合の 99th PC は、低負荷時には CT の 99th PC を上回った。より多数のリクエストのレスポンスタイムを考慮すると、より少数のレスポンスタイムの悪化がキャパシティに影響を及ぼす。したがって、リクエストの出発による負荷不均衡を迅速に解消することが、99th PC の増加につながる。そのため、CT より LL が、LL より N が大きい 99th PC を示した。図 7 は 99th PRT の上限を 700ms としたときの分散ワークロードに対する 99th PC であり、図 8 は、99th PRT の上限を 737ms としたときのバーストワークロードに対する 99th PC である。これらの図からは、ワークロードにおけるリクエスト

サイズのばらつきが大きいほど、リクエスト到着のバースト性が高いほど、N algorithm の既存のアルゴリズムに対する優位性が強まることが読み取れる。図 8 における N の下での 16 ノードクラスタの 99th PC は、CT のそれよりも 21.9%大きい。

図 9 は 99.5th PRT の上限を 643ms としたときの標準ワークロードに対する 99.5th PC、図 10 は、99.5th PRT の上限を 900ms としたときの分散ワークロードに対する 99.5th PC、図 11 は 99.5th PRT の上限を 927ms としたときのバーストワークロードに対する 99.5th PC である。いずれの図においても、同一ワークロードに対する 99th PC に比べて、N と LL の CT に対する優位性が大きくなっていることがわかる。図 11 では、N の下での 16 ノードクラスタの 99.5th PC は CT のそれに比べて 44.6%大きいことが見て取れる。また、16 ノードクラスタで CT を用いたときの 99.5th PC は、12 ノードクラスタで N を用いたときの 99.5th PC にほぼ等しい。つまり、N を用いることで、25%のノード数削減が可能であることがわかる。LL も CT に比べて大幅に 99.5th PC を増加させるが、N に比べると増加幅は小さく、高負荷時の安定性に問題があった。

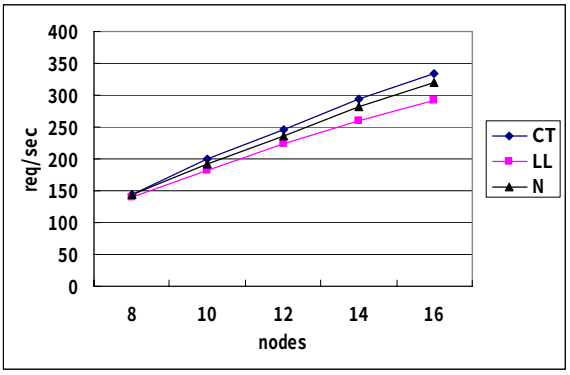

図 3 標準ワークロードに対する 95th PC

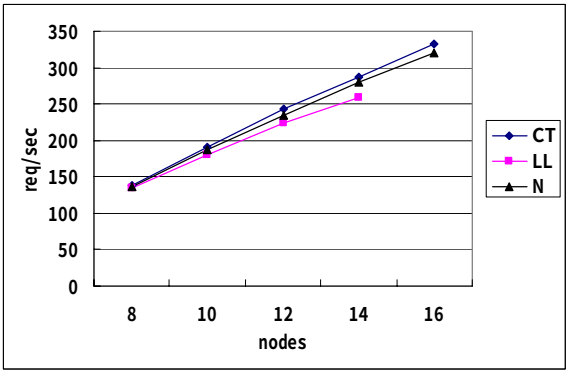

図 4 分散ワークロードに対する 95th PC

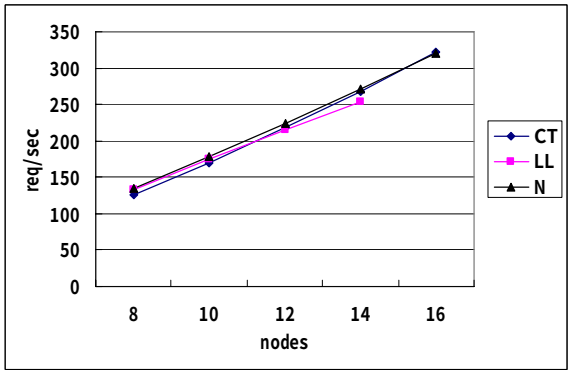

図 5 バーストワークロードに対する 95th PC

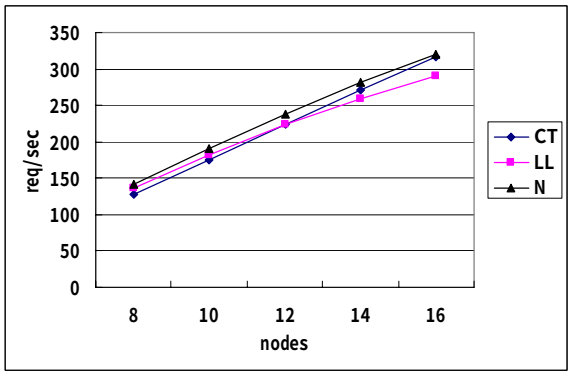

図 6 標準ワークロードに対する 99th PC

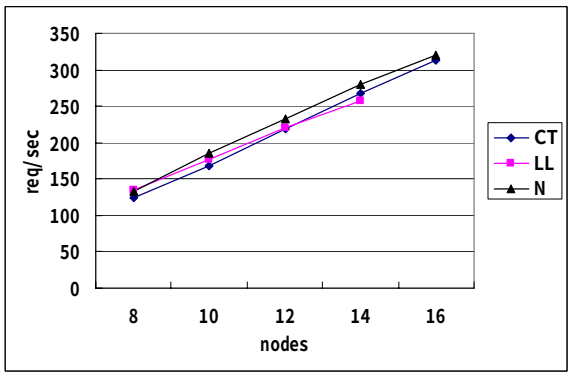

図 7 分散ワークロードに対する 99th PC

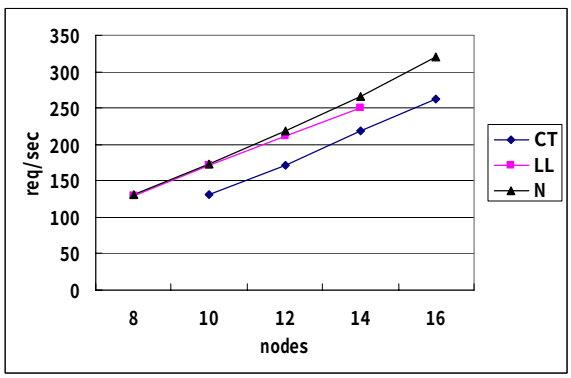

図 8 バーストワークロードに対する 99th PC

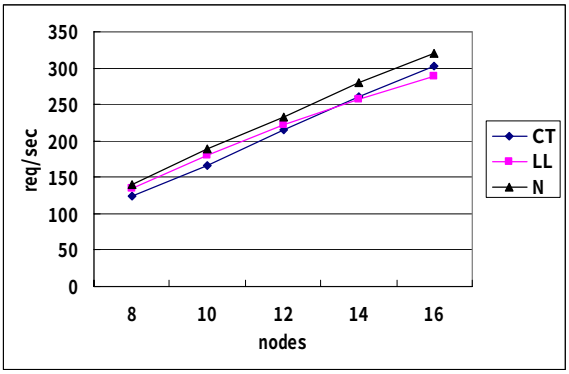

図 9 標準ワークロードに対する 99.5th PC

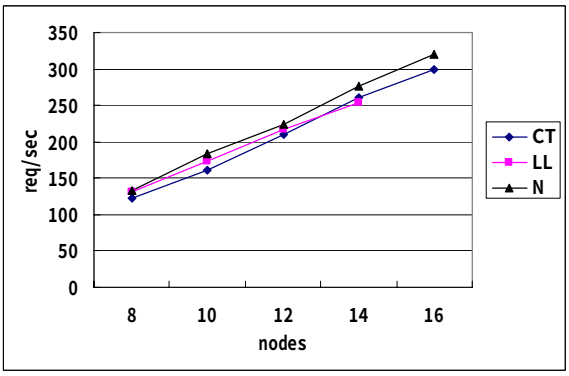

図 10 分散ワークロードに対する 99.5th PC

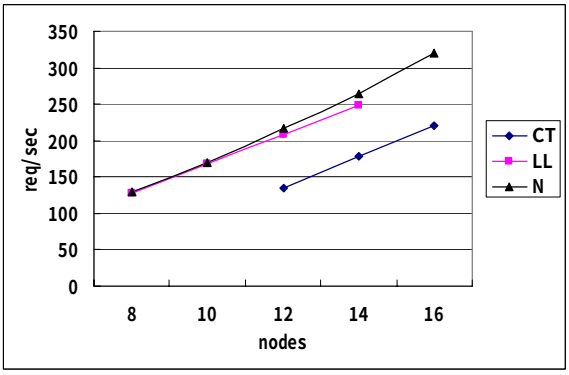

図 11 バーストワークロードに対する 99.5th PC

5. おわりに

クラスタシステムのキャパシティを改善し、要求されたキャパシティをより少ないノードで達成するために、我々は新たな負荷分散アルゴリズム、Nearest Underloaded algorithm (N algorithm)を本稿で提案した。N algorithm は、ノード間に仮想的な距離を設け、これに基づいてリクエスト移送を行う。評価結果では、N algorithm を用いることで、従来のリクエスト振分を行うアルゴリズムを用いた場合に比べて 44.6%のキャパシティの増加を達成できる場合があることを確認した。また、25%少ないノードで同等のキャパシティを達成できる場合があることを確認した。さらに、従来のリクエスト移送を行う負荷分散アルゴリズムは高負荷時に不安定になる傾向があるが、N algorithm は高負荷時でも安定して動作し、より大きいキャパシティを達成できることを示した。

謝辞

本研究は、新エネルギー・産業技術総合開発機構、基盤技術研究促進事業、「大規模・高信頼サーバの研究」の支援の下に行われた。

参考文献

[1] D.A. Menasce and V.A.F Almeida, *Capacity Planning for Web Services*, Prentice Hall PTR, 2002.

[2] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process Migration. *ACM Computing Surveys*, Vol. 32, No. 3, pp. 241—299, September 2000

[3] V. Cardellini, E. Casalicchio, and M. Colajanni. The State of the Art in Locally Distributed Web-server Systems. *ACM Computing Surveys*, Vol. 34, No. 2, pp. 263—311, June 2002.

[4] S. Zhou. A Trace-driven Simulation Study of Dynamic Load Balancing. *IEEE Trans. on Software Engineering*, Vol. 14, No. 8, pp. 1327-1341, September 1988.

[5] D. Eager, E. Lazowska, and J. Zahorjan. A Comparison of Receiver-initiated and Sender-initiated Adaptive Load Sharing. *Performance Evaluation*, Vol. 6, No. 1, pp. 53—68, May, 1986.

[6] D. Ferrari and S. Zhou. A Load Index for Dynamic Load Balancing. Proceedings of *the Fall Joint Computer Conference*, pp. 684—690, November 1986.

[7] J. Stankovic, Simulations of Three Adaptive, Decentralized Controlled Job Scheduling Algorithms, *Computer Networks*, Vol. 8, pp. 199—217, August 1984.

[8] D. Eager, E. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, Vol. 12, No. 5, pp. 662—675, June 1986.

[9] M. Harchol-Balter and A. Downey, Exploiting Process Lifetime Distributions for Dynamic Load Balancing, *ACM Transactions on Computer Systems*, Vol. 15, No. 3, pp. 253—285, 1997.

[10] M. Dahlin. Interpreting Stale Load Information. *IEEE Transaction on Parallel and Distributed Systems*, Vol. 11, No. 10, October 2000.

[11] M. Mitzenmacher, How Useful Is Old Information? *IEEE Trans. on Parallel and Distributed Systems*, Vol. 11, No. 1, pp. 6—20, January 2000.

[12] O. Kremien, J. Kramer, and J. Magee. Scalable, Adaptive Load Sharing for Distributed Systems. *IEEE Parallel and Distributed Technology*, Vol. 1, No. 3, pp. 62—70, August 1993.

[13] K. Shen, T. Yang, and L. Chu. Cluster Load Balancing for Fine-grain Network Services. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pp. 51—59, April, 2002.

[14] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation, In *Proceedings of Performance'98/SIGMETRICS'98*, pp. 151—161, July 1998.