PAPER

Cluster System Capacity Improvement by Transferring Load in Virtual Node Distance Order

Shigero SASAKI[†a)], *Member and* **Atsuhiro TANAKA**[†], *Nonmember*

SUMMARY Cluster systems are prevalent infrastructures for offering e-services because of their cost-effectiveness. The objective of our research is to enhance their cost-effectiveness by reducing the minimum number of nodes to meet a given target performance. To achieve the objective, we propose a load balancing algorithm, the Nearest Underloaded algorithm (N algorithm). The N algorithm aims at quick solution of load imbalance caused by request departures while also preventing herd effect. The performance index in our evaluation is the xth percentile capacity which we define based on throughputs and the xth percentile response times. We measured the capacity of 8- to 16-node cluster systems under the N algorithm and existing Least-Loaded (LL) algorithms, which dispatch or transfer requests to the least-loaded node. We found that the N algorithm could achieve larger capacity or could achieve the target capacity with fewer nodes than LL algorithms could.

key words: *load balancing, herd effect, request departures, cluster systems*

1. Introduction

Cluster systems are prevalent infrastructures for offering e-services because of their cost-effectiveness. Cluster systems are typically composed of computing nodes and switching fabrics. Cost-effectiveness of cluster systems derives from their components, which have become cheaper while being drastically improved. On the other hand, performance of cluster systems is not as simple as to be given as the product of the number of nodes and a single node performance. The performance also depends on many factors which do not exist on single node systems. One of the major factors, for example, is load balancing.

The objective of our research is to enhance cost-effectiveness of cluster systems by reducing the minimum number of nodes to meet a given target performance. A cluster system has to offer sufficient performance to handle incoming requests. Insufficient performance results in excessively long response times that prevent clients from being served. This leads to lost opportunities. A cluster system, on the other hand, should consist of as few nodes as possible to achieve the target performance to reduce system costs. Therefore, the minimum number should be reduced.

To achieve the objective, we propose a load balancing algorithm, the Nearest Underloaded algorithm (N algorithm). The algorithm aims at quick solution of load imbalance caused by request departures while also preventing herd effect [10]. Load imbalance is caused by request arrivals and departures. Imbalances due to both arrivals and departures should be solved as quickly as possible because computing power is not fully utilized while the load is imbalanced. Clearly, request dispatching cannot solve imbalance due to departures immediately. Therefore, imbalance due to departures should be solved by transferring requests on overloaded nodes to underloaded ones, which is called request transfer.

Merely transferring requests to the least-loaded node cannot solve the imbalance due to departures. Many overloaded nodes are likely to transfer requests to a few underloaded nodes independently. As a result, the underloaded nodes become overloaded and request transfers cannot balance the load. This is called *herd effect*. To prevent herd effect, the N algorithm employs a virtual distance.

The performance index in our evaluation is the xth percentile capacity which we define based on throughputs and the xth percentile response times. We define the xth percentile capacity as the number of requests a system can handle per unit of time while retaining the xth percentile response time within a given upper limit. We adopt the capacity because all response times are not necessarily within the limit. Some parts of the requests, which may be 5%, 1%, or 0.5%, are likely to be negligible depending on the circumstances. The average response time does not indicate which parts of the requests will be handled within the limit.

We measured the xth percentile capacity of 8- to 16-node cluster systems under the N algorithm and existing Least-Loaded (LL) algorithms, which dispatch or transfer requests to the least-loaded node. Three synthetic workloads were used in the evaluations. Request sizes (service times for requests) in the first workload followed a lognormal distribution. That in the second workload followed a lognormal distribution which has a bigger standard deviation than that in the first workload. Four requests were issued at the same time in the third workload.

We found that the N algorithm could achieve larger capacity or could achieve the target capacity with fewer nodes than LL algorithms could. The capacity improvement was expanded as the value of x became larger, as the number of nodes increased, and as a workload became deviated and bursty. The 99.5th percentile capacity on a 16-node system for the third workload under the N algorithm was 44.6% larger than that under an LL algorithm which dispatches requests. Furthermore, the capacity on a 12-node system under the N algorithm was almost equal to that on

Manuscript received December 26, 2007.
Manuscript revised August 21, 2008.

[†]The authors are with NEC Corporation, Kawasaki-shi, 211–8666 Japan.
a) E-mail: s-sasaki@di.jp.nec.com

DOI: 10.1587/transinf.E92.D.1

a 16-node system under the LL algorithm which dispatches requests. This implied 25% of reduction of the minimum number of nodes to meet a target performance. The LL algorithm which transfers requests could not work normally on a 16-node system because it caused too many request transfers.

This paper is organized as follows. Section 2 describes our problems in detail, Sect. 3 introduces the N algorithm, and Sect. 4 presents the evaluation results and improvements attained with the N algorithm. Section 5 concludes this paper.

2. Problem Statement and Performance Index

It is easy to balance load if all request sizes (service times for requests) are the same. We only have to dispatch requests in a round robin fashion. It is not difficult to balance load if the size of each request is different but predictable. We only have to dispatch an arriving request to the least loaded node. However, each request size is usually different and unpredictable. There are early and late request departures, and load tends to be measured only by the number of requests. We dealt with this case.

Load imbalance is caused by request arrivals and departures. While imbalance due to arrivals can be solved by dispatching requests, imbalance due to departures cannot be solved quickly by doing this because a certain number of request arrivals are needed to solve this imbalance. In addition, the response times of requests on overloaded nodes, which have many requests, cannot be shortened by dispatching requests. Imbalance due to departures occurs when the number of departures on each node is different with few request arrivals. Unpredictable and small request sizes, wide deviations in request sizes, and bursty arrivals of requests tend to cause imbalance due to departures. HTTP requests, for instance, have these characteristics.

To improve the capacity of a system and to reduce the nodes needed to achieve the target capacity, the imbalance due to departures should be resolved quickly. Transferring requests on overloaded nodes to underloaded nodes is one approach to the problem. However, request transfer is generally difficult to control because each overloaded node transfers requests to underloaded ones independently. As a result, many overloaded nodes are likely to transfer requests to a few underloaded nodes when transfers occur almost simultaneously. This is called herd effect [10], being an example of incorrect request transfers.

Figure 1 shows an example of herd effect when each overloaded node transfers requests to the least-loaded node so that the transferring and transferred nodes have the same number of requests. We can alleviate herd effect by reducing the number of requests transferred, although more activations of the load balancing algorithm are needed to balance the load. This implies that transition to a balanced state becomes slower. Slow transition as well as herd effect can be regarded as a result of incorrect request transfers from the capacity point of view. We need a load balancing algorithm that will produce quick transitions to a balanced state.

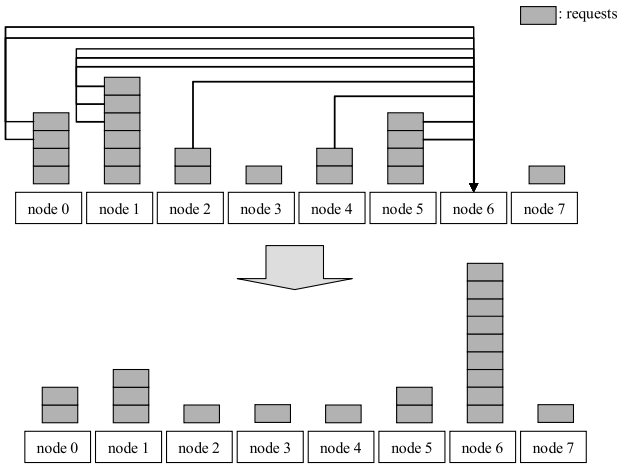

Fig. 1 Example of herd effect.

We define and adopt the xth percentile capacity as a performance index. We define the xth percentile capacity as the number of requests a system can handle per unit of time while retaining the xth percentile response time within a given upper limit. We adopt the capacity because all response times are not necessarily within the limit. Some parts of the requests, which may be 5%, 1%, or 0.5%, are likely to be negligible depending on the circumstances. The average response time is one of the most important performance indices as Menasce et al. showed by examples of capacity planning in [1]. However, this does not indicate which parts of the requests will be handled within the limit. Imbalance due to departures tends to be caused when workloads have wide distributions in request sizes and produce bursty request arrivals. The wide distributions increase differences between early and late request departures. Bursty arrivals cause lengthy periods where no request arrives or is dispatched. The workloads should be characterized by parameters that affect the xth percentile capacity, to put it more concretely, parameters that affect load imbalance due to request departures. Therefore, we employed variations in the sizes of requests and the burstiness of arrivals as parameters for workloads.

3. Load Balancing in a Cluster System

This section describes existing load balancing algorithms and our assumptions on the system. We then describe our load balancing algorithm, the Nearest Underloaded algorithm (N algorithm).

3.1 Classification of Load Balancing Algorithms

We considered a load balancing algorithm to be composed of five components based on Milojicic et al. [2] and Cardellini et al. [3]. These were:

- Initiation policy: Who decides which node requests are processed on?

- Activation policy: When are load balancing algorithms activated?
- Information policy: What are load indices and when are they updated?
- Transfer policy: Which requests are transferred?
- Placement policy: On which nodes is load information available and requests processed?

To clarify the differences between load balancing algorithms, we will describe existing policies and not the algorithms themselves in this section.

Initiation policies can roughly be classified into centralized or decentralized [4]. With a centralized policy, a node or a load balancing fabric dispatches each arriving request, while each node transfers requests with a decentralized policy. Decentralized policies can further be classified into sender-initiated or receiver-initiated policies [5]; however, we have only addressed sender-initiated policies because the differences between sender and receiver-initiated policies are outside the scope of this paper.

Activation policies fall into time-driven or event-driven policies. A time-driven policy usually implies periodic activations. An event-driven policy often triggers load balancing when the load state changes. The changes are caused by, for example, request arrivals or departures.

A major index of load is the length of the request queue [6]. CPU utilization or utilization of other resources can also be adopted as a load index according to the circumstances. Load information is updated in the same way as the activation policies that were previously described.

Transfer policies determine how many requests are transferred when all requests are regarded as being identical. The number of requests to be transferred has been determined by static thresholds and the difference in load on the sender node and receiver node under decentralized load balancing algorithms in Stankovic [7] and Eager et al. [8]. The thresholds have been empirically determined. When requests are not regarded as being identical, they are usually preemptively migrated as described by Harchol-Balter and Downey [9]. To focus on herd effect, we have only dealt with non-preemptive transfer, and have regarded requests as being identical as Stankovic [7] and Zhou [4] did.

Most placement policies choose the least-loaded node in a node scope, which is composed of nodes with which a sender node shares load information. One of the simplest scopes includes all nodes. Eager et al.'s [8], Dahlin's [10], and Mitzenmacher's [11] node scopes were randomly selected nodes. Kremien et al.'s [12] scope was composed of a group of nodes. In Kremien et al.'s scope, an overloaded node probes randomly selected nodes and forms a group with them if they are underloaded. An underloaded node also forms a group with randomly probed overloaded nodes.

Based on classification, we will explain the algorithm we propose and compare it with existing ones in the following sections. Classification helps to compare the algorithms because a load balancing algorithm has many parameters and design choices.

3.2 System Assumptions

Our targeted cluster systems are homogeneous stateless kinds such as Web server clusters or various scientific computing clusters. More concretely, the following assumptions hold with our targeted systems:

1. A cluster system contains identical computing nodes, and these are bottlenecks,
2. Single server software works on a cluster system,
3. The software does not have any state,
4. There are no dependencies among requests,
5. Less than or equal to m requests are processed on a node simultaneously for a given m (≥ 1), and
6. Request transfers incur little cost.

Note that Web servers do not have a state (cookies are in Web requests), and Web requests never depend on one another if cookies are processed in the same way on all nodes. Assumption 5 implies that $(k-m)$ requests can be transferred when there are k ($> m$) requests on a node. A Web request is only a short message, so its transfer consumes limited CPU time and network bandwidth.

3.3 Nearest Underloaded Algorithm

We will now describe each policy for the N algorithm in detail. The initiation policy for the N algorithm is decentralized and sender-initiated. The activation policy is periodic. The information policy is that each node disseminates the number of requests on it to all nodes periodically. Even though these three policies are existing ones, the transfer policy is characterized by employing a dynamic threshold. The placement policy can choose multiple nodes based on virtual distance between nodes.

The transfer policy of the N algorithm is based on a dynamic threshold, an average number of requests. Only overloaded nodes, which have more than the average number of requests, transfer requests under the N algorithm. The number of requests transferred is the difference between the number of requests on the overloaded node and the average number. Therefore load is balanced if the requests are transferred to underloaded nodes without overloading them. We employ the dynamic threshold though most existing policies employ empirical and static thresholds.

The dynamic threshold aims at quick solution of short-term load imbalance. Even though the imbalance could scarcely deteriorate the average response time or the throughput, it should deteriorate the xth percentile response time and capacity. This is because a few response times decide the xth percentile response time. The short-term imbalance is usually caused by request departures especially when request sizes are small and widely distributed and request arrivals are bursty. Consequently, we have to quickly find an appropriate threshold which indicates whether a node is overloaded or underloaded and transfer requests on overloaded nodes to underloaded ones.

Under the N algorithm, a node has an absolute node ID and relative node IDs. Let n denote the number of nodes in a cluster system. The nodes are numbered 0 to $n-1$, which are called absolute node IDs. A relative node ID, on the other hand, is calculated from a pair of absolute node IDs. Node j considers node i to have relative node ID $(i-j+n)$ % n. For example, node 3 considers node 5 to have relative node ID 2, and node 7 considers node 5 to have relative node ID 6 when n is 8. The fewer relative node IDs it has, the nearer the node is.

A relative node ID denotes a virtual distance between two nodes. The distance is statically given because a relative node ID is calculated from two absolute node IDs which are also statically given. On a flat network, we assign the absolute node IDs to nodes in an arbitrary way, and our main target is a cluster system on a flat network. A typical example is a system which is composed of identical stateless nodes and one gigabit Ethernet switch.

The N algorithm, however, should be effective where example cluster systems are connected via a high-speed network. Suppose that we assign consecutive absolute node IDs to nodes in a cluster, and that the placement policy of the N algorithm transfers requests to near underloaded nodes. Then, transfers to nodes of another cluster incur limited cost because few requests are transferred to nodes of another cluster and network is high-speed.

The placement policy of the N algorithm chooses the nearest underloaded node as a destination of request transfers. Most placement policies choose the least-loaded node in a node scope and intend to balance load among the nodes in the scope. A wide scope produces herd effect because many overloaded nodes choose the same node as a destination of transfers. To alleviate herd effect, they usually employ narrow scopes even though it does not always balance load among all nodes. To prevent herd effect, we transfer requests to the nearest underloaded node. An overloaded node regards a node as the nearest underloaded when the average number of requests on nodes between the overloaded one and the node is less than that on all nodes and when the node is the nearest. For example, node 0 regards node 4 as the nearest underloaded if the average number of requests on node 1–4 is less than that on all nodes but that on node 1–3 is more than or equal to that on all nodes. An overloaded node assesses, starting from the nearest to furthest, whether a node is the nearest underloaded. Therefore, overloaded nodes may choose different nodes as the nearest underloaded among all nodes.

The placement policy can choose multiple nodes as destinations of request transfer. Most existing transfer policies transfer few requests to alleviate herd effect. This results in slow transition to a balanced state even though the imbalance is short-term. As has been mentioned, the dynamic threshold enables solving the short-term imbalance if the requests are transferred to underloaded nodes without overloading them. Aiming at quick transition to a balanced state, the placement policy chooses multiple nodes because one underloaded node cannot always accept excessive requests on an overloaded node without overloading itself.

Let a, o, and u be the average number of requests, the number of requests on an overloaded node, and the number of requests on the underloaded node nearest the overloaded one, respectively. When $(o-a)$ is less than or equal to $(a-u)$, the overloaded node can transfer $(o-a)$ requests to the nearest underloaded one without overloading the nearest underloaded one. And then, the overloaded one becomes balanced. When $(o-a)$ is more than $(a-u)$, the overloaded node transfers $(a-u)$ requests to the nearest underloaded one. We note that the nearest underloaded node become balanced but does not remain the nearest underloaded after the transfer. Then, the overloaded one finds the new nearest underloaded node and transfers requests to that node as long as the overloaded node remains overloaded.

The placement policy does not make underloaded nodes overloaded; in other words, it prevents herd effect. Each overloaded node can transfer its excessive requests and becomes balanced because it transfers $(o-a)$ requests and an overloaded one never chooses any other overloaded ones as destinations of transfer. Therefore, all overloaded nodes get balanced after request transfer. An overloaded node does not transfer requests to underloaded nodes that nearer overloaded nodes are going to transfer requests to when the transfers are going to make the underloaded ones balanced. Therefore, underloaded nodes, which are going to be balanced, will never be overloaded.

The steps involved in the N algorithm are summarized as follows. Although a load index can be an integer or a real number, we will only describe the N algorithm when it is an integer because there is no intrinsic difference between them. We note that the number of requests on a node is the load index and that nodes are specified by relative node IDs in the steps below.

1. Calculate the mean load index value a from shared load information.
2. Let *over* be the difference between its own load index value and *ceil*(a).
3. If *over* is not positive, finish this algorithm.
4. Initialize c with 1 and *under* with 0. c denotes the relative node ID of a candidate node. An overloaded node determines how much load is transferred to the candidate node.
5. Add the difference between a and the load index value of node c to *under*.
6. If *floor*(*under*) is positive, transfer load by the smaller *over* or *floor*(*under*).
7. If *over* is smaller than *floor*(*under*), finish this algorithm.
8. Decrease *over* and *under* by *floor*(*under*) and increment c. If c is equal to $n-1$, finish this algorithm.
9. Go to step 5.

Once the N algorithm is activated on each node in a cluster system at the same time, the load index values or the numbers of requests on each node are averaged. Although the load index value can be inaccurate and

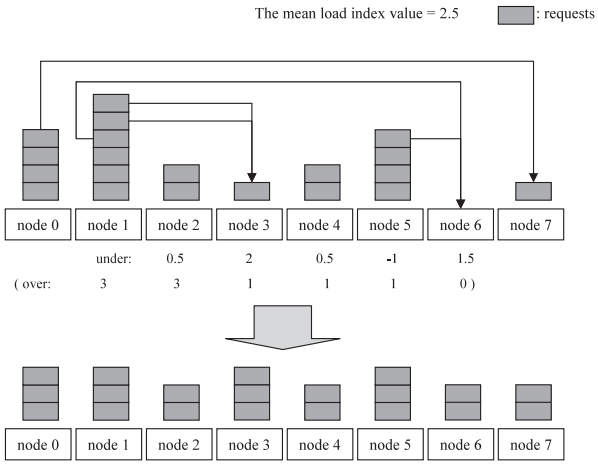

Fig. 2 Transition to balanced state under N algorithm.

simultaneous activation may not always hold, we did not include them within the scope of this section. The effects of accuracy and simultaneity are evaluated from the point of view of the *x*th percentile capacities in Sect. 4.

To simplify our understanding of the transition to a balanced state under the N algorithm, we included an example of the algorithm's behavior in Fig. 2. We can see an 8-node cluster and 20 requests waiting to be processed. The nodes are specified by absolute IDs. The load index is the number of requests on a node, and the mean load index value is 2.5 in the figure. The numbers below nodes 2–6 give us the values of *under* when node 1 has calculated *under* on the nodes.

On node 1, there are 6 requests and *over* equals to 3 ($= 6 - ceil(2.5)$). On node 2, there are 2 requests and *under* is equal to 0.5 ($= 2.5 - 2 + 0$). On node 3, there is 1 request and *under* is equal to 2 ($= 2.5 - 1 + 0.5$). Because *floor*(*under*) is positive and *floor*(*under*) is less than *over*, node 1 transfers 2 (= *floor*(*under*)) requests to node 3. Then, *under* becomes 0 ($= 2 - 2$) and *over* becomes 1 ($= 3 - 2$). On node 4, there are 2 requests and *under* is equal to 0.5 ($= 2.5 - 2 + 0$). On node 5, there are 4 requests and *under* is equal to -1 ($= 2.5 - 4 + 0.5$). On node 6, there is no request and *under* is equal to 1.5 ($= 2.5 - 0 + (-1)$). Because *floor*(*under*) is more than or equal to *over*, node 1 transfers 1 (= *over*) request to node 6. We note that node 0 transfers one request to node 7 because the value of *under* is not positive on node 1–6.

4. Evaluation

We will present two evaluation results in this section that demonstrate centralized request dispatching has more influence on the mean response time than decentralized request transfer, and that request transfer, especially the N algorithm, can increase the *x*th percentile capacity significantly in some circumstances.

4.1 Environment

We used 16 nodes for this evaluation. Each node contained a Xeon 2.4-GHz and 4-GB RAM, and ran on Linux 2.4.7. The nodes were connected via a gigabit Ethernet. A Server Wrapper Daemon (SWD), the load balancing daemon we implemented, and a dummy application ran on each node. A simple workload generator was also implemented and ran on another node. This section describes the implemented software, synthetic workloads, and load balancing algorithms.

4.1.1 Implemented Software

An SWD enables us to transfer requests. This works on each node and intercepts requests from clients to an application running on the same node as the SWD. Intercepted requests are forwarded to the application or to another SWD that works on another node. An SWD receives replies from the application or from other SWDs; it then forwards replies to clients or other SWDs that send requests to itself. An SWD forwards requests to SWDs running on other nodes selected by a given load balancing algorithm. An SWD has a request queue. The request at the head of the queue is the only one that is forwarded to the application, and it cannot be transferred to another node. Requests are transferred from the tail of the queue. In addition to the request transfer facility, an SWD has a facility to share the number of requests in its queue with other SWDs.

A workload generator was substituted for clients in this evaluation. This made HTTP requests and received replies. It had the facility of dispatching requests under various centralized load balancing algorithms. We used a dummy application to accept HTTP requests, and this went into a busy loop for a specified period of time during a request, and then sent a reply.

4.1.2 Workloads

Three synthetic workloads were used in this evaluation. The mean request size was 40 ms for the workloads, which was between the mean request size in the fine-grain trace and that in the medium-grain trace described by Shen et al. [13]. The first workload was called a standard workload where requests arrived according to a Poisson process and the distribution of request sizes was lognormal. The ratio of the standard deviation and the mean of request sizes was 2.16 : 1, which was modeled on the file size distribution in SURGE [14].

The second workload was called a wide deviation workload, which was different from the standard workload in its ratio of standard deviation to mean for request sizes. The ratio in the wide deviation workload was 3.06, which was the product of 2.16 and the square root of 2. The third workload was called a bursty workload, which was different from the wide deviation workload in terms of the arrival

Table 1 Policies for four algorithms.

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

Table 2 Mean response times for standard workload.

	40	80	120	160	200	240	280	320	360
RR	41.6	52.74	64.88	83.49	105.68	139.73	198.63	345.53	728.39
CT	39.16	40.12	40.03	40.28	41.36	44.29	50.08	62.09	97.01
LL	41.68	47.14	51.53	56.25	60.92	67.83	79.84	112.13	N/A
N	39.8	42.06	43.53	46.65	50.87	59	70.99	94.55	162.17

process. Four requests arrived at the same time with an exponential inter-arrival time in the bursty workload. This was a simplified model that reflected short-term fluctuations in arrival rate.

4.1.3 Load Distribution Algorithms

Four load distribution algorithms were employed in this evaluation. They were RR (round robin), CT (centralized algorithm), LL (least-loaded algorithm), and N (N algorithm). Their policies are listed in Table 1. The upper row for information policy describes the load index and the lower row describes how information is updated. Note that CT updates load information when requests arrive and depart. The upper row for transfer policy describes which request is to be transferred and the lower row describes how many requests are to be transferred. Note that the number of requests transferred under LL is equal to half the difference between the queue length on a sender node and that on the least-loaded node.

RR and CT do not have a transfer policy because they do not transfer requests. Request dispatches under LL and N are done in a RR fashion. The nodes in a scope are all other nodes under LL and N.

4.2 Mean Response Times

We measured the mean response times on a 16-node cluster using the four algorithms, increasing each request arrival rate from 40 req/sec to 360 req/seq in 40 req/sec increments. The arrival rates yielded approximately 10% to 90% CPU utilization with increases of 10% because the mean request size was 40 ms and there were 16 nodes in the cluster.

Table 2 lists the mean response times for the standard workload. The reason CT performed best was due to relatively accurate load information that the workload generator had acquired by counting how many requests were on each node. Moreover, there were fewer overheads under CT than under LL and N because no load information was shared and no request transfers occurred. RR was worst because it never utilized dynamic load information. The N algorithm performed close to CT because it quickly solved load imbalance caused by RR. With LL, the mean response time at 360 req/sec was not measured because LL could not withstand such a heavy load. We ceased measurements when the workload generator had more than 16,384 connections to the cluster system.

Table 3 Mean response times for wide deviation workload.

	40	80	120	160	200	240	280	320	360
RR	44.53	67.57	93.09	127.81	165.11	223.2	327.89	610.25	1253.95
CT	38.8	39.8	39.81	40.43	42.31	45.74	53.38	71.56	119.59
LL	43.22	50.61	55.52	60.48	65.37	72.54	88.12	N/A	N/A
N	39.79	42.23	44.05	47.49	52.51	62.08	77.39	109.95	210.84

Table 4 Mean response times for bursty workload.

	40	80	120	160	200	240	280	320	360
RR	52.86	73.82	108.24	154.8	202.57	273.44	378.96	568.99	1156.83
CT	39.99	40.72	44.75	49.5	55.26	64.74	78.15	100.45	157.49
LL	46.42	50.71	57.16	63.12	69.5	81.05	101.83	N/A	N/A
N	41	41.71	45.12	49.75	56.84	69.88	87.5	123.91	226.31

Table 3 lists the mean response times for the wide deviation workload. These are greater than the corresponding response times in Table 2 although there are no significant changes in the relations of the mean response times under the four algorithms. The wide distribution in request sizes deteriorated the mean response times with both centralized and decentralized algorithms. LL could not withstand the load even at 320 req/sec.

Table 4 lists the mean response times for the bursty workload. Note that the mean response times under CT have deteriorated more in comparison with the corresponding ones in Table 3, although this did not happen with the other algorithms. Bursty arrivals prohibited us from acquiring knowledge on request sizes at the time of dispatch from completed requests during inter-arrivals. The mean response times under RR did not change much because bursty arrivals did not affect which node a request was dispatched to under static request dispatches. The reason the mean response times under N and LL did not deteriorate much is that inappropriate dispatches could be compensated for by request transfers.

4.3 *x*th Percentile Capacity

The *x*th percentile capacity of cluster systems under algorithms CT, LL, and N is discussed in this section. What were, or could actually be, measured were the response times of requests with several arrival rates on systems that had 8 to 16 nodes. Therefore, we calculated the *x*th percentile response time from the response times actually measured, and assumed a linear increase or decrease in the *x*th percentile response time for two adjacent arrival rates. The following evaluation results indicate the *x*th percentile capacity under CT, LL, and N for three synthetic workloads with an upper limit for the *x*th percentile response time; this upper limit equals that under N on a 16-node cluster for each workload.

In Fig. 3, we can see the 95th percentile capacity of the systems. The upper limit for the 95th percentile response time was 279 ms. CT performed best under these circumstances. On a 16-node system, for example, CT could

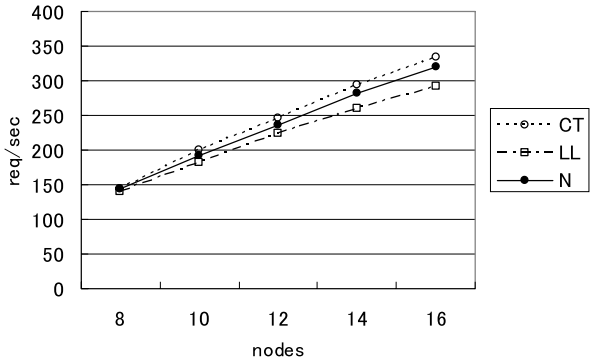

Fig. 3 95th percentile capacity for standard workload (279 ms).

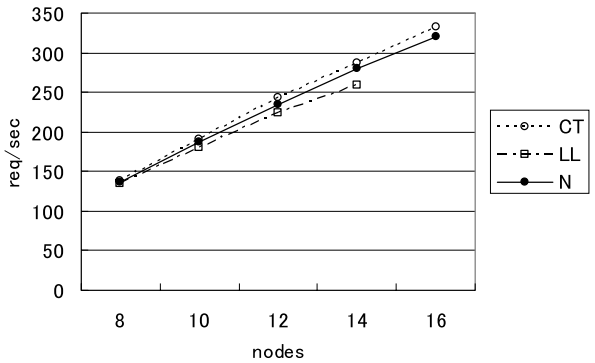

Fig. 4 95th percentile capacity for wide deviation workload (341 ms).

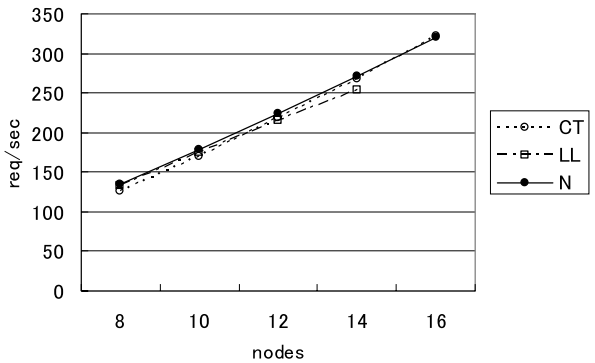

Fig. 5 95th percentile capacity for bursty workload (391 ms).

achieve a 4.7% larger capacity than N could, and the capacity under LL was 8.8% smaller than that under N. Figure 4 plots the 95th percentile capacity for the wide deviation workload where the upper limit was 341 ms. Although CT performed best, the difference in the capacity under CT and N was smaller than the difference plotted in Fig. 3. Figure 5 plots the 95th percentile capacity for the bursty workload where the upper limit was 391 ms. In the figure, the capacity under N was almost the same as that under CT. The capacity under LL for the wide deviation workload and the bursty one was not plotted when the 16-node system was highly loaded. LL caused incorrect request transfers due to its transfer and placement policies. These transfers made the system unstable.

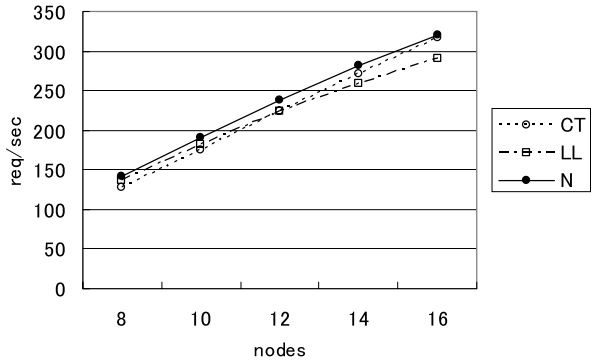

Fig. 6 99th percentile workload for standard workload (517 ms).

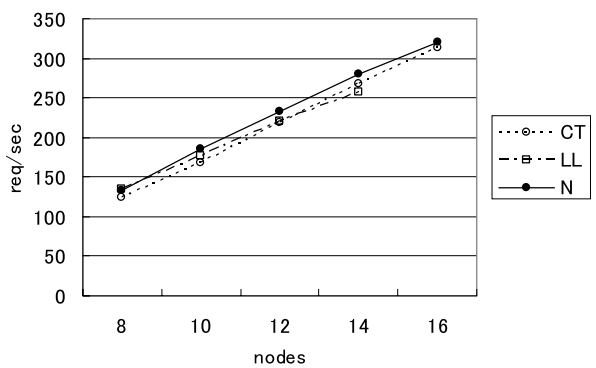

Fig. 7 99th percentile workload for wide deviation workload (700 ms).

The 95th percentile response time as a restriction implies that 5% of the response times do not matter, however long they are. Although the 95th percentile response time can be an appropriate restriction, more response times must be restricted in some circumstances. Therefore, we calculated the 99th and 99.5th percentile capacities. The calculations were based on the same response times as those for the 95th percentile ones.

Figure 6 plots the 99th percentile capacity for the standard workload where the upper limit on the 99th percentile capacity was 517 ms. The figure shows that the N algorithm performed best because it solved the imbalance due to request departures. The simple transfer and placement policies of LL worsened performance as the number of nodes in the system increased, and CT, which did not transfer requests, outperformed LL when the system had more than 12 nodes. Figure 7, which plots the 99th percentile capacity for the wide deviation workload where the upper limit was 700 ms, resembles Fig. 6. Figure 8 plots the 99th percentile capacity for the bursty workload where the upper limit was 737 ms. In the figure, we can see that the capacity under CT deteriorated significantly and that the capacity under N was 21.9% larger than that under CT on a 16-node system.

Figure 9 plots the 99.5th percentile capacity for the standard workload where the upper limit for the 99.5th percentile response time was 643 ms. Figure 10 plots the capacity for the wide deviation workload where the upper limit was 900 ms and Fig. 11 plots the capacity for the bursty

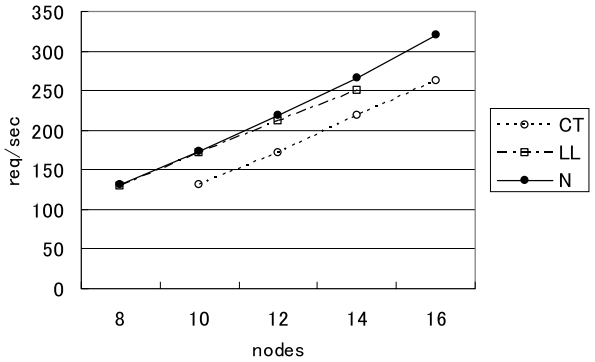

Fig. 8 99th percentile workload for bursty workload (737 ms).

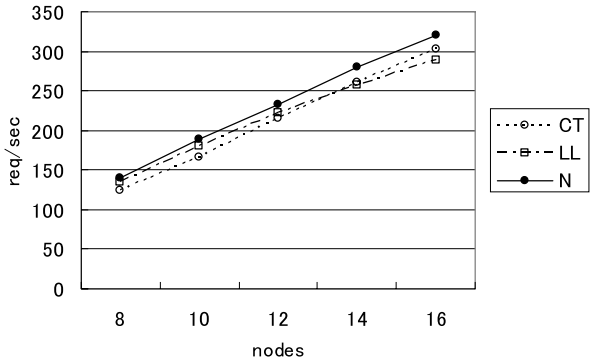

Fig. 9 99.5th percentile workload for standard workload (643 ms).

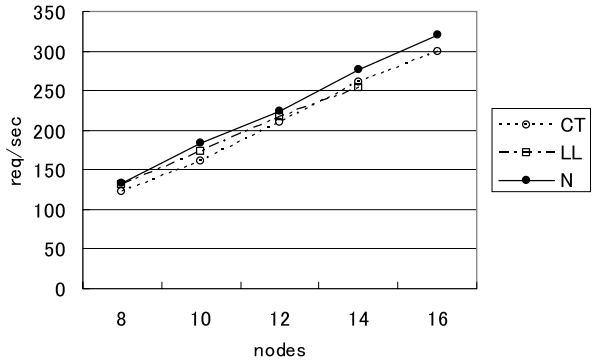

Fig. 10 99.5th percentile workload for wide deviation workload (900 ms).

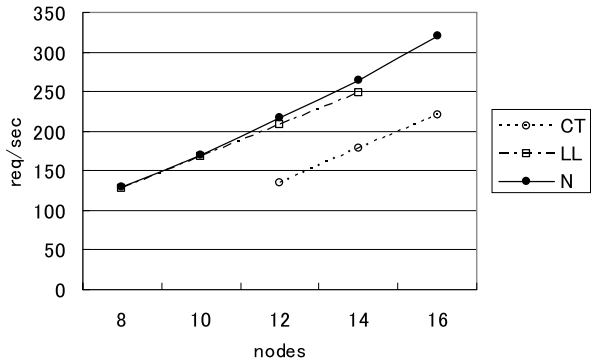

Fig. 11 99.5th percentile workload for bursty workload (927 ms).

workload where the upper limit was 927 ms. In these figures, we can see that the N algorithm could achieve the prominent capacity. In Fig. 11, we can see that the capacity under N was 44.6% larger than that under CT on a 16-node system, and that the capacity under N on a 12-node system was almost the same as that under CT on the 16-node system.

The wide deviations in request sizes created larger differences between actual request sizes and the mean request size. This caused load imbalance after requests were dispatched and deteriorated response times on overloaded nodes. Because there were a relatively small number of deteriorated response times, the deviations did not increase the mean response time under CT more than those under N or LL. However, this increased the xth percentile response time under CT more than those under N and LL. Bursty arrivals greatly reduced the capacity under CT because requests were dispatched before preceding requests had been completed. The high x value made the capacity sensitive to load imbalance because a few deteriorated response times on overloaded nodes affected the capacity. On the other hand, the N algorithm significantly outperformed LL when the system contained many nodes because the behavior of the decentralized algorithms had a much greater impact on capacity in many nodes.

The number of nodes used in our evaluation was small because we did not have enough hardware resource. However, these evaluation results indicate that load imbalance due to request departure has more significance when we intend to achieve the high xth percentile capacity than when we do high throughputs. We note that Dynamo [15] is built for latency sensitive applications that require at least 99.9% of read and write operations to be performed within a few hundred milliseconds.

The short-term imbalance due to departure should be solved quickly because few long response times deteriorate the capacity. The N algorithm could quickly solve the imbalance and achieved higher capacity when the imbalance was caused.

Once the N algorithm is activated, excessive requests on overloaded nodes are transferred to underloaded nodes without overloading them. While existing algorithms balance load in narrow node scopes such as randomly selected nodes, the N algorithm includes all nodes in scope. In the system-wide scope, the proposed algorithm dynamically calculates how much nodes are overloaded or underloaded and solves the imbalance. The algorithm employs a virtual distance to determine destinations of request transfer appropriately to balance multiple overloaded nodes, which have to transfer multiple requests to be balanced, and multiple underloaded nodes.

5. Conclusion

We propose a load balancing algorithm, the Nearest Under-

loaded algorithm (N algorithm), in an effort to improve cost-effectiveness of cluster systems. The N algorithm transfers requests to other nodes in order of virtual node distance, and reduces the minimum number of nodes to meet a target performance. The performance index in this paper is the *x*th percentile capacity, which denotes the number of requests a system can handle per unit of time while retaining the *x*th percentile response time within a given upper limit. The maximum improvement in the 99.5th percentile capacity we measured is 44.6% and the maximum reduction in the number of nodes is 25%. These improvements were achieved by employing the N algorithm instead of an existing least-loaded algorithm which dispatches requests on a 16-node system for bursty requests.

Acknowledgments

This work was carried out under the "Large-scale Dependable Servers" project and was supported by the New Energy and Industrial Technology Development Organization (NEDO).

References

[1] D.A. Menasce, V.A.F. Almeida, and L.W. Dowdy, Capacity Planning for Web Services, Prentice Hall PTR, 2002.
[2] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," ACM Comput. Surv., vol.32, no.3, pp.241–299, Sept. 2000.
[3] V. Cardellini, E. Casalicchio, and M. Colajanni, "The state of the art in locally distributed Web-server systems," ACM Comput. Surv. vol.34, no.2, pp.263–311, June 2002.
[4] S. Zhou, "A trace-driven simulation study of dynamic load balancing," IEEE Trans. Softw. Eng., vol.14, no.8, pp.1327–1341, Sept. 1988.
[5] D. Eager, E. Lazowska, and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated adaptive load sharing," Perform. Eval., vol.6, no.1, pp.53–68, May 1986.
[6] D. Ferrari and S. Zhou, "A load index for dynamic load balancing," Proc. Fall Joint Computer Conference, pp.684–690, Nov. 1986.
[7] J. Stankovic, "Simulations of three adaptive, decentralized controlled job scheduling algorithms," Comput. Netw., vol.8, pp.199–217, Aug. 1984.
[8] D. Eager, E. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," IEEE Trans. Softw. Eng., 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 Trans. Comput. Syst., vol.15, no.3, pp.253–285, 1997.
[10] M. Dahlin, "Interpreting stale load information," IEEE Trans. Parallel Distrib. Syst., vol.11, no.10, pp.1033–1047, Oct. 2000.
[11] M. Mitzenmacher, "How useful is old information?" IEEE Trans. Parallel Distrib. Syst., vol.11, no.1, pp.6–20, Jan. 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, Aug. 1993.
[13] K. Shen, T. Yang, and L. Chu, "Cluster load balancing for fine-grain network services," Proc. 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," Proc. Performance'98/SIGMETRICS'98, pp.151–161, July 1998.
[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," Proc. Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, pp.205–220, Oct. 2007.

Shigero Sasaki received the B.S. degree in Mathematics from Keio University in 1997 and the M.S. degree in Information Science from University of Tokyo in 1999. Currently, he is working at NEC Corporation. His research interests include real-time scheduling, Web caching algorithms, load balancing, and clustered databases.

Atsuhiro Tanaka received the B.S. degree in Mathematical Engineering and Information Physics in 1990 and the M.S. degree in Information Engineering in 1992 from University of Tokyo. He joined NEC Corporation in 1992 and is engaged in system performance evaluation.