Load Balancing in Distributed Web Server Systems with Partial Document Replication

Ling Zhuo, Cho-Li Wang, Francis C. M. Lau
Department of Computer Science and Information Systems
The University of Hong Kong
Pokfulam Road, Hong Kong
{lzhuo, clwang, fcmlau}@csis.hku.hk

Abstract

How documents of Web site are replicated and where they are placed among the server nodes have an important bearing on balance of load in a Distributed Web Server (DWS) system. The traffic generated due to movements of documents at runtime during load balance could also affect the performance of the DWS system. In this paper, we prove that minimizing such traffic in a DWS system is NP-hard. We propose several heuristic document distribution schemes that perform partial replication of a site's documents at selected server locations so that load balancing is maintained. We carry out simulation of these schemes using both a synthetic workload and real log data. From the simulation results, we find that using an additional 50% of storage for replication, our heuristics can improve the load balancing performance in the DWS system by 48%, and the internal traffic due to movements of documents has an negligible effect on the system's performance.

1. Introduction

With the increasing popularity of the World Wide Web, more people are getting online. According to a recent survey, the size of the "wired population" increases from 407 million in November 2000 to 513 million in August 2001 [25]. This puts a heavy demand on Web servers. And with broadband technology getting more and more pervasive and Internet bandwidth expanding, Web servers could easily face the danger of becoming a bottleneck. Distributed Web Server (DWS) system was proposed to meet this requirement. A DWS consists of multiple server nodes that are connected by a LAN or WAN and uses the aggregate computing power and storage available from these nodes to handle client requests.

In a DWS system, the Web site's documents are distributed among the server nodes using certain rules. These rules determine each document's set of replicas and the placement of these replicas among the server nodes. These rules constitute the "document distribution scheme" which is the subject of this paper. A good scheme is necessary for achieving load balancing in a DWS system.

We classify DWS systems according to the document distribution schemes used. With the first type, “mirroring”, each server node receives and maintains a copy of all the documents. Client requests are copied to multiple server nodes by DNS servers or a centralized scheduler [1,8,10,19,5]. The problem with mirroring is that the caching of IP addresses on the client side or in intermediate DNS servers could easily result in uneven load among the server nodes, and the scheduler may therefore become a performance bottleneck. Moreover, this type of DWS systems can lead to much wasting of disk space by storing documents that are not frequently requested.

Non-duplication DWS systems partition the documents without duplication of contents. They depend on content-aware distributor software to direct or redirect a client request to the server node that has the document requested [18]. There are now products that support content level switching, such as Alteon Content Director [22], CISCO Content Services Switch [23] and Resonate Global Dispatch [26]. Advanced features can be added to such a DWS, such as when a server node gets overloaded, some of its documents are migrated to other nodes [4,13]. However, if the system happens to contain many hot spots (i.e., popular Web pages with extremely high request rates), to equalize the load is absolutely non-trivial.

One way to avoid the load balancing problem due to hot spots is to allow partial-duplication of popular documents on multiple server nodes [11,15,16,20]. Some of these systems adopt a static approach which places the documents and their replicas on selected server nodes based on past access records [15,16]. They aimed at equalizing the average load of each server node over a long period of time, such as one day. Other partial-duplication DWS systems monitor the current global load situation and duplicate popular documents dynamically to maintain a balanced load [11,20]. Although this approach can adapt to the access patterns quickly, it is not clear how it might impact on the traffic inside the DWS system.

In this paper, we focus on geographically distributed DWS with homogeneous nodes and examine partial-duplicated document distribution schemes from a theoretical as well as experimental viewpoint. Geographically distributed DWS systems are becoming more common because of the increased performance of WAN connections and the availability of Web servers in wide areas. The homogeneity feature is preferred here so that the model and its algorithms can be more tractable during analysis. We propose document distribution schemes that can achieve the following goals when deployed:

- Load balancing: Since most requests are for a small part of the entire collection of documents [2], frequently requested documents should be duplicated to avoid bottlenecks. Documents and their replicas should be placed in such a manner that most of the time the load of the participating server nodes is equalized.

- Reduced traffic inside the DWS system: To adapt to users access patterns, documents need to be re-duplicated and re-distributed among the server nodes dynamically or periodically. Such actions should result in reduced traffic inside the system when compared to a system that does only static duplication and

distribution.

- Scalability: Adding a new server node or adding new documents to the system should be easy and transparent, and should not perturb the load-balance currently enjoyed by the system. The same for deletion of nodes or documents. This enables fast expansion of a DWS system to handle increased client requests.

It is easy to see that document distribution in DWS is an optimization problem that is NP-hard. We present various approximation algorithms in this paper. Their performance is evaluated through simulation using synthetic as well as real access log data. The results show that these algorithms can balance the load in the DWS system during run-time efficiently, and the internal traffic generated due to these algorithms is reasonably minimal.

The rest of the paper is organized as follows. Section 2 introduces the DWS system model and gives a proof of the NP-hardness of the problem. Section 3 presents our document distribution algorithms. In Section 4, we describe our simulation methodology and present the performance results. Section 5 surveys related work. Section 6 concludes the paper and discusses future work.

2. System Model

2.1 Architecture

Figure 1 gives a bird-eye view of a DWS system. There is one request redirector, one document distributor, and multiple back-end server nodes connected by a WAN. Their functions are described as follows.

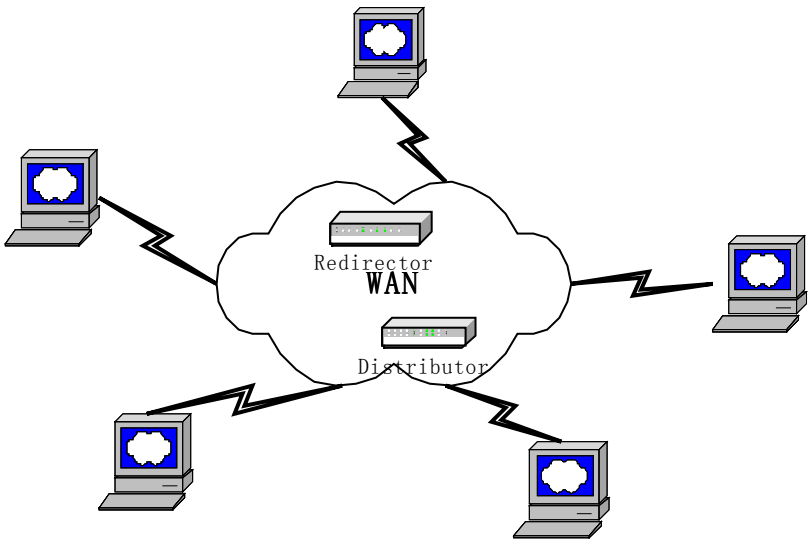

Figure 1 Geographically Distributed Web Server System

- **Request redirector**: The request redirector keeps track of each server node's documents. It redirects a user request to a server node having the requested document. If there are multiple such qualified server nodes, it chooses one based on a certain policy. Note that Figure 1 shows only an example. In reality, there could be multiple request redirectors (using techniques such as Round Robin DNS) to share the load.

- **Document distributor**: The document distributor executes the document distribution scheme. Periodically, it collects the server nodes' log data, and decides on the replicas and their placement based on the access records in the last period and the current placements. The rules it applies in making the decisions aim to maintain load balancing with minimal traffic inside the DWS system.

- **Server nodes**: The server nodes are interconnected and each of them keeps a subset of the site's documents. During each period, they perform internal transfers or copying of documents, following the latest placement decisions coming from the document distributor. When a server node receives a client request, it needs to deal with two cases: (a) if the requested document is locally available, the server node serves the request; (b) if the server node does not have the document, it will forward the request to the redirector since the redirector has the location information of all documents.

2.2 Problem Formulation

In our model, there are N documents, $\{D_1,\ldots D_N\}$, and M server nodes, $\{S_1,\ldots S_M\}$. Each document D_i is characterized by its **size** s_i and its **weight** w_i that is calculated as $w_i = s_i * \delta_i$, where δ_i is the access rate of the document. w_i represents the workload D_i brings to the server node holding it (we use the length of a document to compute its weight [15]). If D_i is replicated, c_i stands for its **number of replicas**. D_i's replicas are $D_i^l (l = 1,\ldots c_i)$, each having size s_i and weight w_i / c_i. In the following discussion, we use "replicas" to refer to all the documents and their replicas. Each server node has storage capacity C, and load capacity L which is the maximum number of simultaneous HTTP connections it can support. The weight W_j of S_j is the sum of the weights of all documents that it has.

We have the following assumptions: at any time, (i) the total number of client requests does not exceed the total load capacity, i.e., $M * L$; (ii) the total size of the replicas does not exceed the total storage capacity, i.e., $\sum_{i=1}^{N}(s_i * c_i) \le M * C$.

We construct a "cost link" between each document and each server: p_{ij} ($i = 1,...N$, $j = 1,...M$), to be associated with the bytes to be transferred when $D_i^l (l \in 1,\ldots c_i)$ is assigned to S_j. We know that p_{ij} equals to s_i if S_j already holds a replica of D_i. Otherwise, p_{ij} is zero.

We also have the following variables:

$$t_{ij}^{l} = \begin{cases} 1, \text{ if } D_i^l \text{ is placed on server } j \\ 0, \text{ otherwise} \end{cases}$$
$$j \in \{1,...M\}, i \in \{1,...N\}, l \in \{1,...c_i\}$$

The replicas are placed on the server nodes under these constraints: (1) each server can only hold replicas whose total size does not exceed its disk space; (2) each server can hold at most one replica of a document; (3) no document is left unassigned to any server node; (4) load is equalized among the server nodes.

Chen [7] has proved that minimizing the maximum load over all server nodes, i.e., satisfying (4) is NP-complete. We will prove that even when the load balancing constraint is removed, the problem of minimizing the communication cost of moving the documents is NP-hard.

The formulation of our replica placement problem is as follows.

minimize
$$z = \sum_{j=1}^{M} \sum_{i=1}^{N} \sum_{l=1}^{c_i} t_{ij}^{l} p_{ij}$$

subject to
$$\sum_{i=1}^{N} \sum_{l=1}^{c_i} t_{ij}^{l} s_i \le C \quad (1)$$

$$\sum_{l=1}^{c_i} t_{ij}^{l} \le 1, j \in \{1,...M\}, i \in \{1,...N\} \quad (2)$$

$$\sum_{j=1}^{M} \sum_{l=1}^{c_i} t_{ij}^{l} = c_i, i \in \{1,...N\} \quad (3)$$

$$t_{ij}^{l} = 0 \text{ or } 1, \ i \in \{1,...N\}, j \in \{1,...M\}, l \in \{1,...c_i\}$$

A replica placement that fulfills all the constraints is a "feasible placement." Our discussion is under the assumption that a feasible placement always exists. We call this optimization problem the *Replica Placement Problem* (RPP). When $c_i = 1$ $(i = 1,...N)$, the problem is a 0-1 RPP.

Lemma *0-1 RPP is NP-hard*.

Proof: We reduce the bin-packing problem, which is NP-hard [12], to the 0-1 RPP. For the bin-packing problem, s_i denotes the sizes of the objects and the bin's size is C. We assume that, in any feasible solution, the lowest indexed bins are used. This means that if there are two bins with the same available storage, the object will be placed in the one with the lower index.

Given the bin-packing problem, we can construct a 0-1 RPP with costs p_{ij} as follows.

$$p_{ij} = \begin{cases} 1, & i \in \{1,...N\}, j = 1 \\ (p_{i,j-1}) * N + 1, & i \in \{1,...N\}, j \in \{2,...M\} \end{cases}$$

With such costs, any set of replicas assigned to $\{S_1,\ldots S_j\}$ has a lower value than any set of replicas assigned to $\{S_1,\ldots S_{j+1}\}$. It is then obvious that the bin-packing problem gets the minimal value if and only if the 0-1 RPP gets the minimal value.

Since the 0-1 RPP is a special case of the RPP, our document placement problem is NP-hard.

3. Document Distribution Algorithms

Since the optimization problem at hand is NP-hard, finding the optimal solution in polynomial time is not feasible. We present several heuristics that use the available information in different ways in order to arrive at the best results. The input parameters to these heuristics are *C*, *M*, *N*, $\overline{s}=\{s_1,\ldots s_N\}$, $\overline{w}=\{w_1,\ldots w_N\}$, and $\overline{P}$, the *N*M* matrix of p_{ij}. The output is an *N*M* matrix called $\overline{T}$, formed by t_{ij}. We refer to the ratio between the total storage capacity and the total size of the documents, i.e., $(M*C)/\sum s_i$, *relative capacity*, and denote it by *R*.

We hope to balance the load among the server nodes through appropriately duplicating popular documents. Therefore, before introducing the heuristics, we first describe the replication algorithm some of the heuristics will invoke to obtain the numbers for the documents' replicas, $\overline{c}=\{c_1,\ldots,c_N\}$. The replication algorithm is called the "Density Algorithm" and is shown in Figure 2. It assumes a different interpretation of D_i's access rate δ_i. Notice that $\delta_i = w_i / s_i$, which can be seen as the measure of the workload per one unit of storage of D_i, i.e., **density** of D_i. When D_i is replicated, for each of its replica, the new density is $\delta_i' = (w_i / s_i)*(1/c_i)$.

The Density Algorithm first initializes $\overline{c}$ as $\{1,\ldots 1\}$, thus making sure that each document has at least one copy in the system. In Step2, the minimal density $\delta_{\min}$ is found and each document gets a temporary replica number so that its replicas all have density $\delta_{\min}$. Step3 calculates the ratio between the total size of these temporary replicas and the total size of the original documents, and adjusts each temporary replica number proportionally according to this ratio. The maximal integer that is less than the result is added to c_i, and if c_i is greater than *M*, it is reduced to *M*. This is because a document can only be duplicated integral times, and its replica number will not exceed the number of servers.

This algorithm duplicates the documents according to their densities under the storage limitation, and produces as result, $\delta_k/(c_k-1)\approx\delta_{k+1}/(c_{k+1}-1)\approx\cdots\approx\delta_N/(c_N-1)$, ($\delta_1\ge\delta_2\ge\cdots\ge\delta_N, c_{k-1}=M, c_k<M$). We can assume that generally if $\delta_u>\delta_v$, $\delta_u/c_u>\delta_v/c_v$, a replica of a document with large density has larger density than a replica of a document with small density. The time complexity of the algorithm is ΘN.

```
Variable: S = M * C, t̄ = {t_1, ... t_N}, temp
Begin
    Step1 c_i := 1, ∀i ∈ {1,... N} , temp := (R-1) * Σ s_i
    Step2
       2.1 Go through δ_i, ∀i ∈ {1,... N} and find δ_min
       2.2 Foreach D_i do {
              t_i := δ_i / δ_min ;
              temp := temp + t_i * s_i }
    Step3 Foreach D_i do {
            c_i := c_i + ⌊t_i * S / temp⌋
            temp := temp - t_i * s_i;
            If c_i > M Then c_i := M
            S := S - c_i * s_i }
End
```

Figure 2 Density Algorithm

We use the following heuristic to place the replicas on the server nodes.

3.1 Greedy-cost Algorithm

The Greedy-cost Algorithm is shown in Figure 3. Before placing the replicas, it sorts the pairs (i, j) ($i = 1,\ldots N, j = 1,\ldots M$) by increasing p_{ij}. Then in this order, a replica of D_i is placed on S_j if S_j has enough storage space and does not already hold a replica of D_i.

According to its definition, p_{ij} is either 0 or s_i, depending on whether S_j was assigned a replica of D_i or not during the last period. Therefore, Greedy-cost actually sorts all documents whose c_i is larger than the number of its replicas in the

```
Begin
    Step1 Call Density Algorithm
    Step2
       2.1 Sort (i,j) by p_ij increasingly, i=1,...N, j=1,...M
       2.2 Foreach (i,j) in the sorted list do {
              If (c_i > 0) Then {
                 Assign D_i to S_j if there is enough space on S_j
                        and S_j does not have D_i^1 on it
                 c_i := c_i - 1; } }
End
```

Figure 3 Greedy-cost

system by their sizes. It tries to keep as many replicas as possible, but does not care if the load of the system is balanced or not. Its Step2 takes $\Theta(MN \log MN)$ time, and therefore the total time complexity is $\Theta(N + MN \log MN + MN)$.

3.2 Greedy-load/cost Algorithm

Unlike Greedy-cost, Greedy-load/cost considers balancing the load among the server nodes while minimizing the cost caused when placing the documents.

```
Begin
    Step1 Call Density Algorithm
    Step2 Sort replicas according to δ_i , decreasingly. If they
have same δ_i , sort them by increasing s_i
    Step3 Foreach replica in the sorted list do {
              Sort S_j in increasing communication cost. If two
    servers have the same cost, sort them by decreasing weight
              While (c_i > 0) do {
                  Place D_i in the order of sorted server nodes
                  c_n := c_n - 1; } }
End
```

Figure 4 Greedy-load/cost

This heuristic prefers replicas with higher load and smaller size, that is, larger density. Therefore in Step2, the replicas are sorted in decreasing δ_i' and replicas that possess the same δ_i' are sorted in increasing s_i. In this order, the replicas are assigned to server nodes one by one. When choosing a server node for a single replica, the qualified server nodes are sorted according to their communication costs given the replica. If two server nodes have the same cost, we choose the one with less weight W_j.

The time complexity of Greedy-load is $\Theta(N + X\log X + XM\log M)$, where $X = \sum c_i$, and $X \geq N$. From the result of the Density Algorithm, however, we know that it will be rare for replicas of two different documents to have the same density as well as the same size. All of D_i's replicas, D_i^l $(l = 1,\ldots,c_i, i \in \{1,\ldots N\})$, can be assumed to appear consecutively in the list sorted by Step2. Actually, they can be seen as sorted and then placed as one integral unit in Step2 and Step3. The time complexity of the heuristic is therefore reduced to $\Theta(N + N\log N + NM\log M)$.

There can be variants of this heuristic; for example, server nodes can be sorted in increasing available space in Step3.

3.3 Greedy-penalty Algorithm

It is possible that if we do not place D_i^l immediately, extra communication cost may be caused. For example, if we place D_i^l immediately, we can assign it to S_x with small p_{ix}; if we delay placing it for a while, however, S_x may have become full when we try to place D_i^l again and D_i^l has to be placed on S_y with large p_{iy}. In this case, we say we are punished and use f_i to refer to the value of penalty.

To compute the possible penalty for each document, $D_i^l (l = 1,\ldots c_i)$ are treated as an integral unit. Since we cannot predict the status of the server nodes, f_i is computed as the difference between the costs of D_i^l's best and second-best placements according to the current situation of the server nodes. A placement is "better" if it requires less communication cost. To find the best one, the qualified server nodes are sorted in increasing p_{ij}. The first to the c_i^{th} server nodes form the best placement, while the second to the $(c_i+1)^{\text{th}}$ form the second-best placement. f_i is the difference between the communication costs of the first server node and the $(c_i+1)^{\text{th}}$ server node in the sorted list.

Greedy-penalty iteratively places the replicas until they are all assigned to the server nodes. Each time it needs to select and place a document and its replicas, it computes f_i for all unassigned documents. The document yielding the largest penalty is placed in its best placement. This heuristic runs in $\Theta(N + N\log N + NM\log M + N^2\log N)$ time.

```
Variables: f = {f_1,... f_N}
Begin
    Step1 Call Density Algorithm
    Step2 Sort D_i in increasing δ_i
    Step2 While there are unassigned documents do {
            Foreach unassigned D_i do {
                If number of qualified server for D_i <= c_i Then {
                    Assign D_i^1 to qualified servers;
                    goto While }
                Else {
                    Sort S_j in increasing p_ij. If two servers have
                the same cost, sort them in increasing load
                    f_i = cost of (c_i+1)th server node - cost of 1st
server node  } }
              Sort D_i according to f_i, decreasingly
              Assign D_min according to the sorted list of server
          nodes}
End
```

Figure 5 Greedy-penalty

If there is only one placement for D_i^l, we know that there are currently only c_i server nodes having enough storage to hold D_i^l. This implies that we need to place D_i^l immediately. Otherwise, we might leave a replica unassigned, which violates constraint (3). In this case, we set f_i to ∞. If there are multiple documents with infinite penalty, they are placed in the order of decreasing δ_i. As such, Greedy-load/cost is just a special case of Greedy-penalty.

3.4 M/1 Algorithm

The above heuristics all use the Density Algorithm to duplicate documents. The one presented in this section uses a different duplication approach. We sort D_i in decreasing δ_i and replicate as many documents *M-1* times in this order as the storage constraint would allow. As a result, documents that have heavier workload but smaller size are placed on every server node, while other documents would only have one copy in the system. In placing documents whose c_i is 1, the available server node S_j with the smallest p_{ij} is chosen. The time complexity of this approach is $\Theta(N\log N + MN)$.

Although this approach sounds simple, we believe that it can be used with good effects in Web sites whose user accesses tend to target at a tiny part of the total documents. The existence of such Web sites was discussed and justified in [17].

4. Simulation Details and Results

We used the CSIM 18 package for our simulation experiments [24]. We tested our

heuristics with a synthetic trace as well as log files of a real Web server. At time 0, documents are randomly placed on the server nodes with no replication. As time progresses, documents are replicated and distributed to the server nodes using the different heuristics.

We simulated the heuristics presented in Section 3: Greedy-cost (GC), Greedy-load/cost (GL/C), Greedy-penalty (GP) and M/1 (M/1).

4.1 Simulation Model and Metrics

Our simulation model is similar to the one used by Vivek et al. [18], whose "back-end nodes" and "front-end nodes" are actually our "server nodes" and "redirector" respectively. We made several modifications. First, since our focus is on the load balancing performance of our document distribution algorithms, main memory in server nodes is not used for document caching, and all of the documents are assumed to be stored on disk. Second, we assume the technique of HTTP redirection is used for user request redirection.

Based on the cost figures of different processing steps for a given Web request provided in [18], we could see that compared to the disk access time and disk transfer time, the connection establishment time and teardown time are negligible. Therefore, in our model, processing a web request comprises (1) redirection (if necessary), (2) waiting in the queue of the serving server node, and (3) reading the file from disk. We assume that the round-trip time of redirection is 100 ms [6]. The disk reading parameters are derived from Seagate ST360020A technical specification, with the disk access time of about 19 ms and the disk transfer time about 21 MB/s [27].

4.2 Experiment Settings

Experiment 1 – Synthetic Workload

In this experiment, there are 200 documents, each of size 10^x, where x is randomly drawn from a normal distribution with μ=1, and σ=0.3, giving us files sizes that are mostly in the range of (1.26 KB, 79.4 KB).

At any time, the total number of client requests in the system is kept under *0.667*M*L*. The client requests follow a Zipf distribution. According to the Zipf law, if documents are ranked according to their access frequencies, the access rate of the i-th most popular document is proportional to $1/i^{\alpha}$. In our experiment, α is set to 1.4, which has been shown to be suitable for a real Web server [17]. To make the access pattern closer to reality, at the middle of each period, the most popular document is substituted by another document randomly chosen.

Experiment 2 – Real Log files

This experiment uses the dataset and log files of a Web site that hosted mainly personal homepages. For simplicity, we group documents under each personal

account together as one "group" and use the group as the basic unit of replication and distribution in this experiment. In order to expedite the simulation, we prune unpopular groups that account for less than 0.05% of the total traffic. This leads to a 75% reduction in the number of groups while only 3.89% reduction in the bytes transferred to clients. The details of the groups are shown in Table 1.

Table 1. MU statistics

Total Number	201
Minimal Group size	8.2KB
Maximal Group size	63.5MB
Total size of Groups	4,571MB

In this experiment, we use three consecutive days worth of log files from the Web server, as shown in Table 2. The period of document distribution to simulate is 8 hours.

Table 2. Log files statistics

Period	Jan/24/01 – Jan/26/01
Average successful requests per day	227,124

4.3 Load Balancing Analyses

We study the load balancing effects of the proposed heuristics in both experiment settings. The Load Balance Metric (LMB) [3] is used as a performance metric for comparing results. To obtain the LMB value, the peak-to-mean ratio of server load is measured at different sampling points (1 sampling point every hour) in the simulation. The server load is defined as the utilization value of the server node. The LBM value is obtained by calculating the weighted average of the peak-to-mean ratios measured, using the total server load as the weight for the sampling period in question. A smaller value indicates a better load balancing performance.

For the purpose of comparison, we include a dummy heuristic that does not duplicate documents at all. It places the documents in order of increasing w_i, and each document is assigned to the server node with the least W_j at the time. We call this heuristic Non-replication (NR).

We first fix the number of server nodes, M (= 8), and then step by step increase the relative capacity R from 1 to M. We know that with the proposed heuristics, when R = 1, the system is actually a non-replication DWS; on the other hand, when $R = N$, the system becomes a mirroring DWS. Therefore, we ignore these two cases in our discussion.

Figure 6 and Figure 7 show that in both experiments, the proposed heuristics achieve much better load balancing than NR does. With 50% additional storage, the GC/L can improve on its performance by 48% and 31% in the two experiments respectively.

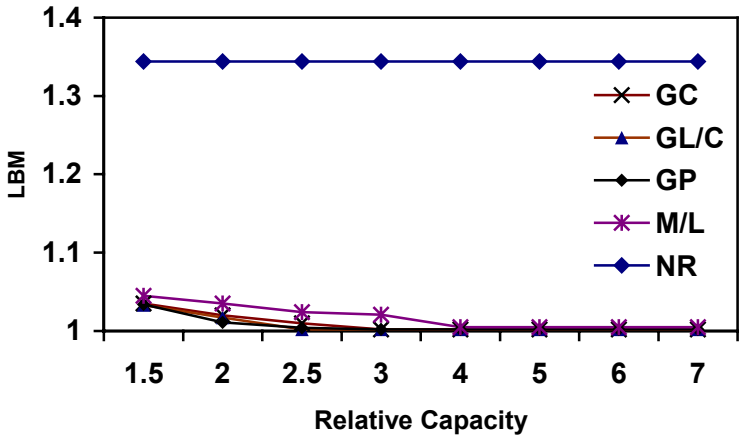

Figure 6 Load Balancing Performance in Experiment 1

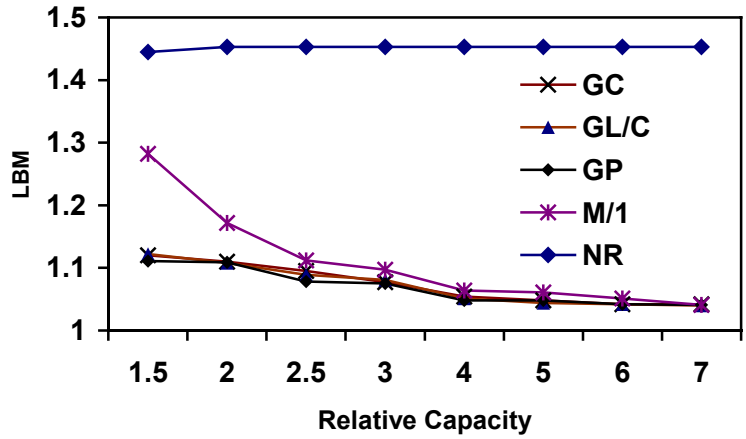

Figure 7 Load Balancing Performance in Experiment 2

We also notice that the performance of the heuristics gets better as R increases. When R is equal to $M/2$, almost half of the documents are duplicated on each server node and the performance becomes very close to mirroring. M/1 is not as good as the other three, because it does not duplicate documents with small δ and only considers communication cost when placing the replicas.

Figure 8 presents the performance of GC, GL/C and GP. GC does not consider load balancing when placing the documents, but it achieves better load balancing than M/1. This proves that the Density Algorithm it employs helps to keep load balanced. When R is small, GL/C's performance and GP's performance are similar. When R gets bigger, GP achieves better load balancing than GL/C.

Because of the large variance of group sizes and access patterns, the load balancing results in Experiment 2 are not as good as those in Experiment 1. But the shapes of the curves are similar to those from Experiment 1, and so are the differences among the curves.

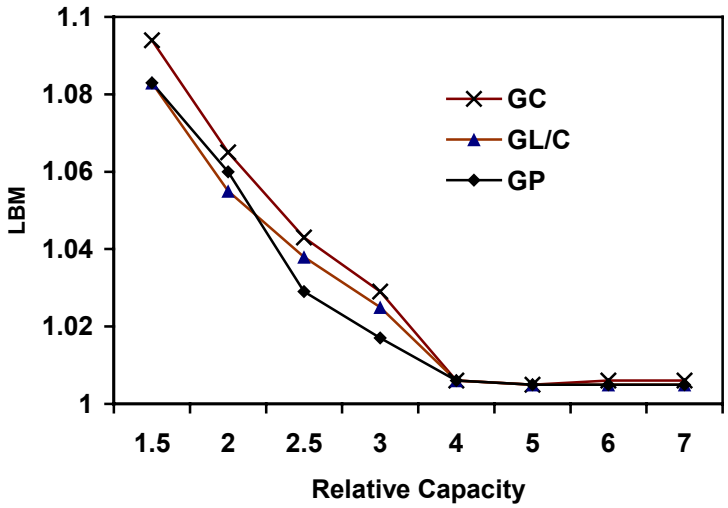

Figure 8 Load Balancing Performance of heuristics in Experiment 1

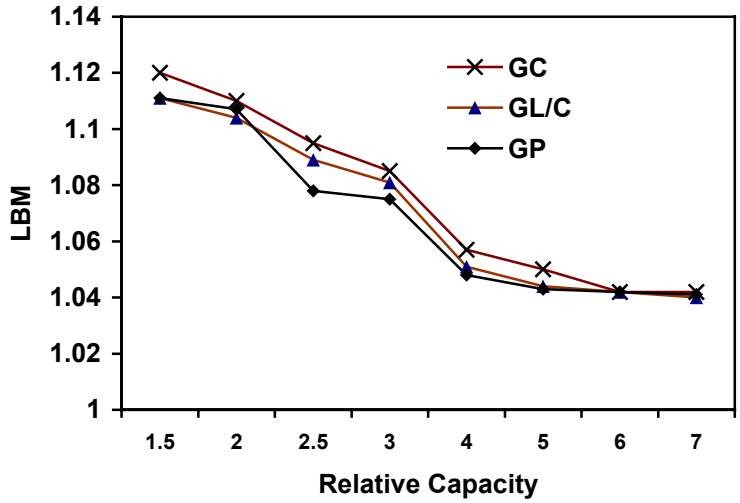

Figure 9 Load Balancing Performance of heuristics in Experiment 2

We then fix *R* at 1.5, and increase *M* from 8 to 64 to see how well the proposed heuristics can adapt to the change of number of server nodes.

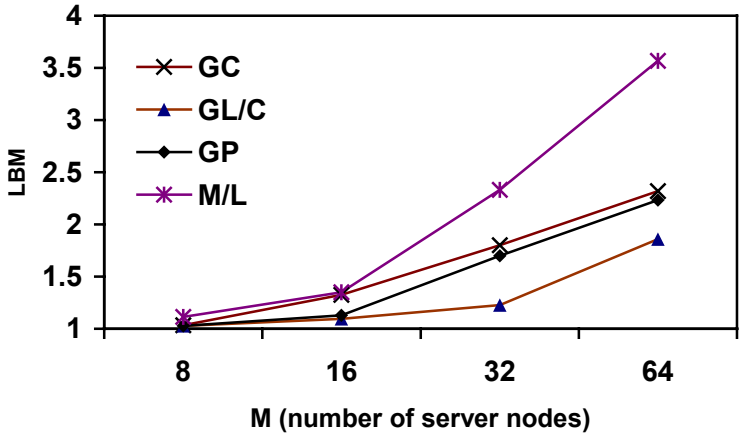

Figure 10 R = 1.5 in Experiment 1

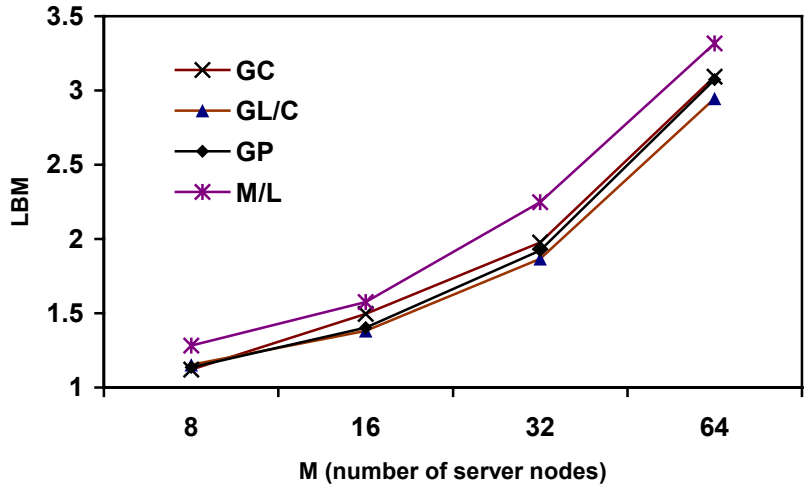

Figure 11 R = 1.5 in Experiment 2

When M increases, the capacity of each server node, C, decreases. Therefore in Figure 8, all the LBM values become larger as R increases. M/1's performance deteriorates the fastest, because it duplicates all the hottest documents on every server node. When M is large, it can only duplicate very few documents.

We see that GP's performance is between GC/L and GC, of which GC/L takes load balance into account when placing the replicas, and GC considers only communication costs. When C is small, GP is similar in performance to GL/C. When C decreases, GP gets closer and closer to GC. This is consistent with the result we obtained when M is fixed.

4.4 Traffic Analyses

We use the algorithm in Figure 12 to transfer the documents inside the DWS system. If S_x needs to give a replica $D_i^l (l = 1, \ldots c_i)$ to S_y which doesn't have the replica,

we record the bytes S_x needs to transfer, i.e. s_i. When a server node needs to fetch a replica from other server nodes, it chooses the one which has had to transfer least bytes to other server nodes so far. We record the total bytes transferred inside the system in each period and compute the average at the end of the simulation. The

```
Output: O = {O_1,...,O_M}
Begin
    Step1 O_j := 0,  ∀j ∈ {1,...,M}
    Step2 Foreach D_i, do {
            Foreach S_j do {
            If D_i is assigned to S_j and p_ij == s_i Then {
                Find S_l that holds D_i and O_l is minimal
                    O_l := O_l + s_i } } }
End
```

Figure 12 Algorithm of moving MUs

smaller this average is, the less affected is the performance of the DWS system due to movements of the documents.

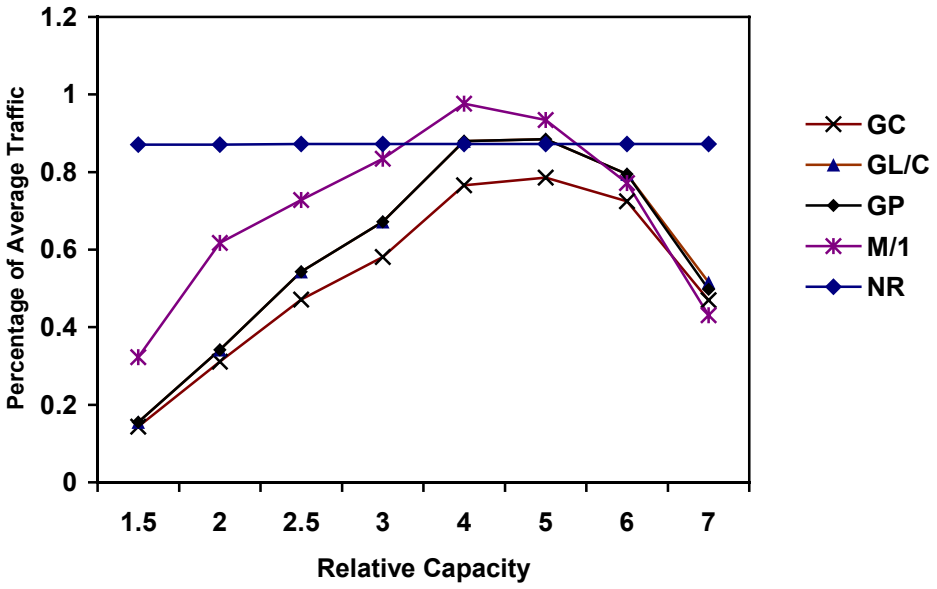

Figure 13 Traffic Analyses in Experiment 2

Figure 13 plots the internal traffic figures generated by the heuristics in Experiment 2. The x-axis is *R*, and the y-axis is the average we mentioned above, as the percentage of the total document size. *M* is fixed to 8. For NR, its traffic remains unchanged as *R* changes. In most cases, its cost is much larger than the other heuristics. The largest cost generated by GP and GL/C is only 1.01% of corresponding NR's costs.

For the proposed heuristics, as *R* increases, the traffic increases at first. This is because there are more replicas in the system, and if the access pattern changes, more replicas needed to be moved around. When R is larger than M/2, however, the traffic begins to decrease because many documents are now duplicated on each server node, thus decreasing the need to move them.

It is easy to understand that GC, which cares most about communication costs,

incurs the least cost. The M/1 approach generates the most traffic because c_i can only be M or 1, and so when the ranking of documents changes, more replicas would be affected. GP and GL/C both consider the request rate of documents as well as the current placement when placing documents, and thus their costs are in between the other ones and they are very close to each other. Although we use total bytes transferred as the metric, the actual time needed to move documents is $t = max(o_i)/bandwidth$. From the algorithm of moving documents, we know that

$$t \approx \sum_{j}^{M} o_j / (bandwidth * M)$$

Therefore, in this experiment, if we assume that the network bandwidth is 1 MB/s, moving documents would not take more than several minutes. Since during this period, the DWS system can continue to serve requests with documents not in the move, this length of time is acceptable.

From the simulation results, we see that our heuristics can improve load balancing in a DWS system significantly and generate acceptable traffic. Among the four schemes, M/1 is the simplest, but the traffic it generates is heaviest and its load balancing capability is the worst. GC generates the least internal traffic, but its load balancing performance is not as good as GP and GL/C. GP and GL/C incur similar costs. When the server capacity, C, is small, GL/C balances the load among the server nodes better than GP, and when C is large, the other way around. Since GP requires more computation, it might not be as practical as GL/C. Therefore, we conclude that GL/C, which considers the documents' load and size and the current placement when placing documents, is most suitable for DWS systems.

5. Related Work

In a full-mirroring DWS system, each server node has all the documents of the Web site. The system relies on a mechanism to direct client requests to selected server nodes. This mechanism may reside on the client side or on the server side, called client-based or server-based respectively [5]. Client-based approaches put the burden of choosing a server node in the DWS system and routing the request to the selected node on the client [14, 15]. Although these approaches make clients share the load of the Web server, they cannot guarantee good load balancing at the server side because of clients' ignorance of server nodes' load status. Smart client [21], designed by Yoshikawa et al, solves this problem by transmitting server nodes status to clients through a Java applet. The continuous communication between each applet and server node, however, may generate excessive additional network traffic.

Server-based approaches used in full-mirroring DWS systems are mainly DNS-based [8,10,19]. A summary of these approaches is provided in [5]. Due to caching in intermediate DNS servers and clients' browsers, a high degree of load balance is not easily achievable [8,9]. It is also possible for requests to continue to go to a server node even the node is already overloaded. To address this problem,

Cardellini et al proposed a load balancing strategy for geographically DWS systems [6]. In their system, a DNS server assigns clients' requests to a selected server node first. If a server node is overloaded, it can redirect some of its requests to an appropriate peer node. SWEB [1] integrates Round Robin DNS with redirection. It takes the CPU load, disk I/O channels and interconnection network bandwidth into consideration when distributing HTTP requests to the server nodes.

Mourad et al. proposed a simple document distribution scheme in their non-duplication DWS system [13]. Each server node only holds part of the Web site's documents and periodically reports its load to a monitoring coordinator. If a server node is overloaded, some of its content is migrated to another server with less load based on analysis of the access pattern. The author claimed that this system can support data replication by replicating some of the highly accessed content that is on an overloaded server, but they did not provide concrete information on how this was done.

Another non-duplication DWS, DCWS, is proposed by Baker and Moon [4]. They make use of a graph-based Web document-partitioning algorithm in their document distribution scheme. Each document resides on its home server and can be migrated to a co-op server for load balancing reason. To redirect client requests from the home server to the co-op server, all hyperlinks pointing to the document need to be modified, which may require substantial computation. Since one hyperlink can only refer to one document, this system cannot support document replication.

In the partial-duplication DC-Apache system [11], each document has a home server that keeps its original copy. When the document becomes popular, however, it is replicated instead of being migrated to co-op servers. Each time the replicas of a document or their locations change, the document's home server regenerates all the hyperlinks pointing to this document based on global load information to order to maintain load balancing. Although such dynamic document distribution scheme can adapt to user access patterns quickly, it generates much traffic in transferring documents among the nodes, and much computation in updating the hyperlinks.

Unlike DC-Apache, RobustWeb [15] is a partial-duplicated DWS system featuring a static document distribution scheme. The sets of replicas are determined on the basis of past access records and placed on the server nodes beforehand. If a document has several replicas, each replica carries a number indicating its probability of being accessed. Front-end redirection servers choose a server node for a request according to this probability. Instead of moving documents like we do in this paper, in RobustWeb, only the redirection probability is computed periodically. RobustWeb aims to equalize the average access rate for each server over a long time, for example, one day.

Chen et al. extended RobustWeb's work theoretically [7]. They showed that if memory constraints are present, achieving the optimal load balancing through 0-1 document placement is NP-hard. They also presented several greedy approaches, but these approaches do not consider the traffic requirement, making their work not suitable for practical DWS systems.

Ng at el. [16] included a prefetching feature in their EWS system, a

partial-duplication DWS. In this system, documents that are always accessed together are grouped and placed on the same server node. Only the first request of a session has to go through the redirection server, thus cutting down on the redirection overhead. A revised document placement algorithm of the one used in RobustWeb is used to maintain load balance. Our work can be considered a derivative from theirs by taking storage constraints and communication cost into account. The algorithms we propose here can be used in EWS.

6. Conclusion and Future Work

In this paper, we study the rules used in a document distributed scheme for determining document replicas and their placement in a DWS system. We proved that the placement optimization problem is NP-hard for homogeneous DWS systems. We presented several heuristics and studied their performance via simulation. The results showed that the heuristics can balance the load among the server nodes during run-time of the DWS system, and traffic due to movements of the documents is negligible.

Our next step is to incorporate geographical information into our document distribution schemes so that they would duplicate a document at a location where the document is most wanted, which is expected to reduce access latencies substantially.

References

[1] Andresen, D.; Yang, T.; Holmedahl, V.; Ibarra, O. H., 1996, "SWEB: Towards a Scalable World Wide Web Server on Multicomputers", presented at Proceedings of IPPS.

[2] Arlitt, M. F.; Williamson, C. L. 1996, "Web Server Workload Characterization: The Search for Invariants", presented at ACM Proceedings of the ACM SIGMETRICS conference on Measurement & Modeling of Computer Systems.

[3] Bung, R. B.; Eager, D. L.; Oster, G. M.; Williamson, C. L. 1999, "Achieving Load Balance and Effective Caching in Clustered Web Servers". Presented at Proceedings of the 4th International Web Caching Workshop.

[4] Baker, S. M.; Moon, B.1999. "Scalable Web Server Design for Distributed Data Management", presented at Proceedings of the 15th International Conference on Data Engineering.

[5] Cardellini, V.; Colajanni, M.; Yu, P. S. 1999, "Dynamic Load Balancing on Web-Server Systems", *IEEE Internet Computing*, vol. 3.

[6] Cardellini, V.; Colajanni, M.; Yu, P.S. 2000, "Geographic Load Balancing for Scalable Distributed Web Systems", presented at Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems.

[7] Chen L. C.; Choi H. A. 2001, "Approximation Algorithms for Data Distribution with Load Balancing of Web Servers", presented at Proceedings of the 2001 IEEE

International Conference on Cluster Computing.
[8] Colajanni, M.; Yu, P. S.; Dias, D.M. 1998, "Analysis of Task Assignment Policies in Scalable Distributed Web-Server Systems", *IEEE Trans. Parallel and Distributed Systems*, vol. 9.
[9] Dias, D. M.; Mukherjee, W. K. R.; Tewari, R. 1996. "A Scalable and Highly Available Web-Server", presented at Proceedings of 41st IEEE Computer Society International Conference.
[10] Kwan, T. T.; McGrath, R. E.; Reed, D. A. 1995, "NCSA's World Wide Web Server: Design and Performance", *Computer*, vol. 28.
[11] Li, Q. M.; Moon, B. 2001, "Distributed Cooperative Apache Web Server", presented at Proceedings of the 10th International World Wide Web Conference.
[12] Martello. S; Toth, P. *Knapsack Problems: algorithms and computer implementation*: John Wiley & Sons Ltd, 1990.
[13] Mourad, A.; Liu, H. Q. 1997, "Scalable Web Server Architectures", presented at Proceedings of the 2nd IEEE Symposium on Computers and Communications.
[14] Mosedale, D.; Foss, W.; McCool, R. 1997, "Lessons Learned Administering Netscape's Internet Site", IEEE Internet Computing, vol. 1.
[15] Narendran, B.; Rangarajan, S.; Yajnjk, S.1997, "Data Distribution Algorithms for Load Balanced Fault -Tolerant Web Access", presented at Proceedings of the 16th Symposium on IEEE Reliable Distributed Systems.
[16] Ng, C. P.; Wang C. L. 2001, "Document Distribution Algorithm for Load Balancing on an Extensible Web Server Architecture", presented at Proceedings of 1st IEEE/ACM International Symposium on Cluster Computing and the Grid.
[17] Padmanabhan V. N.; Qiu, L. L. 2000, "The Content and Access Dynamics of a Busy Web Site: Findings and Implications", presented at Proceedings of ACM SIGCOMM 2000.
[18] Pai, V. S.; Aron, M.; Banga, G.; Svendsen, M.; Druschel, P.; Zwaenepoel, W.; Nahum, E. 1998, "Locality-Aware Request Distribution in Cluster-based Network Servers", presented at Proceedings of 8th International Conference on Architectural Support for Programming Languages and Operating Systems.
[19] Schemers, R. J. 1995, "lbmnamed: A Load Balancing Name Server in Perl", presented at Proceedings of USENIX 9th Systems Administration Conference.
[20] Yang, C. S.; Luo, M.Y. 2000, "A Content Placement and Management System for Distributed Web-Server Systems", presented at Proceedings of 20th International Conference on Distributed Computing Systems.
[21] Yoshikawa, C.; Chun, B.; Eastham, P.; Vahdat, A.; Anderson, T.; Culler, D. 1997, "Using Smart Clients to Build Scalable Services", presented at Proceedings of the USENIX 1997 Annual Technical Conference.
[22] Alteon Content Director
http://www.nortelnetworks.com/products/01/pcd/fandb.html
[23]. Cisco CSS 11000 Series Content Service Switches
http://www.cisco.com/univercd/cc/td/doc/product/webscale/css/index.htm
[24] "CSIM 18 User Manual",
http://ringer.cs.utsa.edu/faculty/boppana/courses/csim18man.htm

[25] "How many online?" http://www.nua.net/surveys/how_many_online/index.html
[26] Resonate Global Dispatch
http://www.resonate.com/solutions/products/global_dispatch/data_sheets.php
[27] Tom's Hardware Guide, www.tomshardware.com