On the Design of Distributed Object Placement and Load Balancing Strategies in Large-Scale Networked Multimedia Storage Systems

Zeng Zeng and Bharadwaj Veeravalli, *Senior Member*, *IEEE*

Abstract—In a large-scale multimedia storage system (LMSS) where client requests for different multimedia objects may have different demands, the placement and replication of the objects is an important factor, as it may result in an imbalance in server loading across the system. Since replication management and load balancing are all the more crucial issues in multimedia systems, in the literature, these problems are handled by centralized servers. Each object storage server (OSS) responds to the requests that come from the centralized servers independently and has no communication with other OSSs in the system. In this paper, we design a novel distributed load balancing strategy for LMSS, in which OSSs can cooperate to achieve higher performance. Such OSS modeled as an $M/G/m$ system can replicate the objects to and balance the requests among other servers to achieve a near-optimal average waiting time (AWT) of the requests in the system. We validate the performance of the system via rigorous simulations with respect to several influencing factors and prove that our proposed strategy is scalable, flexible, and efficient for real-life applications.

Index Terms—Multimedia storage system, request balancing, distributed system, average waiting time, queuing theory.

1 Introduction

Most multimedia application programs run on stand-alone personal computers, with digitized media sourced from local hard disks, CDs, or DVD ROMs. However, the demand for multimedia data servers is increasing, and availing a variety of multimedia services via home computers/laptops or mobile devices has become commonplace for Internet users in this modern era. Recently, more and more researches have focused on large-scale multimedia applications such as video on demand (VoD) [1], [2], IP television (IPTV) [3], [4], etc., and most of the multimedia contents are retrieved from multimedia storage systems [5], [6].

A large-scale multimedia storage system (LMSS), which consists of a single server or multiple servers, manages the storage and retrieval of multimedia data from disk media. As a networked multimedia service is expected to serve a large pool of clients, it is impossible for a single server to meet all multimedia requirements such as continuous-time presentation [1], network bandwidth, etc. Multigigabyte multimedia files are common, and the data sets are fast growing with daily added multimedia files of news, TV serials, etc. System administrators have to regularly handle many terabytes of files. Due to their reliability and flexibility, multiserver storage systems have gained a predominant position in both the academe and industry [8], [9], [15].

• *The authors are with the Computer Networks and Distributed Systems Laboratory, Department of Electrical and Computer Engineering, National University of Singapore, 10 Kent Ridge Crescent, Singapore, 117576. E-mail: {elezz, elebv}@nus.edu.sg.*

Manuscript received 20 Sept. 2006; revised 26 Apr. 2007; accepted 24 Aug. 2007; published online 1 Oct. 2007.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0445-0906. Digital Object Identifier no. 10.1109/TKDE.2007.190694.

Servers can generally be classified in two large categories: centralized and distributed. In a centralized system, a high-end scheduler monitors, serves, and manages the video streams among the servers [13], [14], whereas in a distributed system, workstations or PCs can work as servers, and they cooperate with limited local information to achieve high system performance. As highlighted in the literature, compared with the centralized strategies, the distributed management of the system offers more advantages and, hence, more and more research works have been concentrating on distributed environments [10], [11], [12], [16]. For the traditional distributed computer networks, each server in the system can provide computation capability for every client, whereas in multimedia storage systems, only the server, which has stored the requested objects in its disk(s), can provide the data retrieval service for the clients. It may happen that, assuming that a client request arrives at a server for a movie object, whose bandwidth has been allocated to other served movie streams, although there may be aggregate bandwidth available in other servers without the requested object, the request has to be blocked. One solution to solve such problems is to make some replications among other servers in the system so that other servers can serve the request [20]. Hence, the placement and replication of the multimedia objects are important issues in a multimedia storage system.

In LMSS, if some servers remain idle, whereas others are extremely busy, the system performance will drastically be affected. To prevent this, *load balancing* [12], [23], [24] is often used to distribute the requests and improve performance measures such as the *average waiting time* (AWT), which is the time difference between the time instant at which a request arrives to the system and the time instant at which the client starts receiving the first package of the multimedia stream. In the literature, pertaining to LMSS, by and large, the issues are dealt in an isolated fashion. This means that studies would independently focus on balancing the requests, storing/replicating the media files, and optimal disk/storage utilization problems to quote a few. In this paper, we focus on designing a load balancing strategy for a cluster of workstations or PCs constituting an LMSS. We take a radically different approach in considering the problem in a collective manner and devise a distributed load balancing strategy to determine the placement and number of replications for the media objects. In addition, we attempt to balance the requests among the servers to achieve a minimized AWT of the requests.

1.1 Scope of This Work and Our Contributions

We now define the scope of the formulation adopted in this paper. We focus on dealing mainly with two conflicting issues that are of utmost importance in the context of large-scale multimedia storage related problems. These two issues are media replication and load balancing. For large-scale systems, with widespread geographically scattered nodes/servers, it is obviously beneficial to devise distributed strategies. Thus, first, our problem context considers a distributed system scenario. To this end, we attempt at providing a mathematical formulation that captures and demonstrates the ability of replicating and balancing the load across the media servers. Based on this formulation, we devise distributed strategies to optimize certain key performance metrics. Although there exist a variety of performance metrics for evaluating such distributed strategies, based on the current context of the problem, a natural and primary choice is to consider three important metrics, namely, AWT, storage disk usage/utilization, and Denial of Service (DoS). We will also include several different request arrival patterns within the scope of our simulation studies and attempt at putting forth certain key observations that are crucial to the design of such systems. Now, we explicitly state our key contributions.

Our contributions are summarized as follows: We model the multimedia server as an $M/G/m$ system [36], based on which we propose a model of the server to quantify AWT when a server provides a client a multimedia stream with a fixed bandwidth. We attempt to formulate the load balancing problem as a nonlinear constrained optimization problem. To be specific, we consider an environment where multiple multimedia servers are connected by an arbitrary network as the real-life situation. Clients generate requests for particular objects to the multimedia servers through the Internet. As with the principle of load balancing, requests are allowed to migrate from heavily loaded servers to lightly loaded servers to minimize the AWT of the requests.

We derive a number of practically useful results for the load balancing problem. We compare our proposed algorithm with a recently proposed project named "Neptune," a load balancing strategy that is based on the Polling method [20]. We analyze various request arrival patterns and the waiting behaviors of users to make our quantitative study more practical and usable. From rigorous simulations, we show that our proposed algorithm clearly offers a significant advantage in minimizing the AWT, keeping the rate of DoS within a low level, and, at the same time, balancing the server loading across the system.

The rest of this paper is organized as follows: Section 2 discusses relevant research work. Section 3 describes the system model and formulates load balancing as a nonlinear optimization problem. In Section 4, we propose our strategy to place and replicate the objects and balance the requests among the servers. In Section 5, we discuss the dynamic load balancing strategy and the benchmark algorithm used in this work. Section 6 shows the results of our simulation experiments. We conclude our work in Section 7.

2 Related Work

The pattern of incoming requests is very important in the analysis of LMSS, and different objects may have different request probabilities. By studying the characteristics of Web access, we can obtain some useful hints from client access patterns. The Zipf-like request model is widely accepted and used in the Web traffic modeling literature [18]. It is a general case of the Zipf law, which is originally applied to the relationship between a word's popularity in terms of rank and its frequency of use [19]. It states that if one ranks the popularity of words used in a given text as i and the frequency of use as p, then $p = \frac{k}{i}$, where

$$k = \frac{1}{\sum_{i=1}^{N}(1/i)}$$

is a constant. Bestavros et al. [21] gathered 500,000 Web requests and generalized that the distribution of Web requests follows a Zipf-like distribution as follows:

$$p_i = \frac{k}{i^{\alpha}}, \ where\ k = \frac{1}{\sum_{i=1}^{N}\left(\frac{1}{i^{\alpha}}\right)}. \tag{1}$$

Here, the value of α varies from trace to trace, ranging from 0.64 to 0.83. Nishikawa et al. [22] studied an access log of 2,000,000 requests and found that the request distribution follows the Zipf law, with $\alpha = 0.75$, which we use as a default parameter in this paper.

In [8], Peter proposed a distributed file system named *Lustre*, which has been used in HP servers. In Lustre, the metadata servers (MDSs) handle the request balancing and determine the location of the requested objects among object storage servers (OSSs). However, although the storage strategy is distributed, the architecture of Lustre is still centralized, and it has no communication among the OSSs. In [32], Yao et al. proposed a technique to achieve a highly scalable storage system by applying a random placement of data objects across a group of servers. Their goal was to redistribute a minimum number of objects to achieve a uniform distribution and, hence, a balanced load. In [16], Serpanos et al. proposed a load balancing algorithm named *MMPacking* to achieve equally balanced bandwidths and storage capacities. These algorithms can perfectly be used in homogeneous systems but cannot directly be applied in a generic heterogeneous LMSS. In [20], Shen proposed a clustering architecture named "Neptune" and employed a random polling-based load balancing policy for the clusterwide request balancing. *Peer-to-Peer* (P2P) architectures have become popular recently [34], in which each peer can discover the requested content with a distributed method and provide the data retrieval by point-to-point communications. However, in pure P2P systems, the communication and

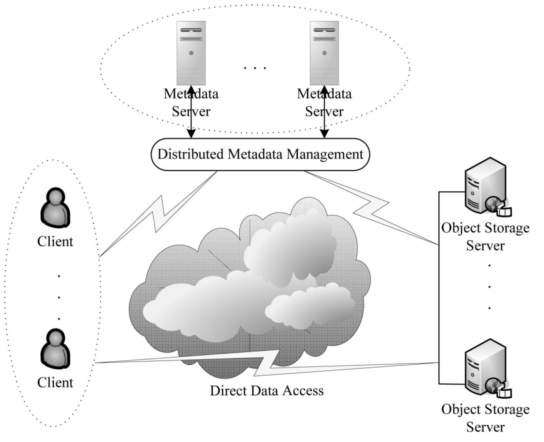


Fig. 1. System architecture of our proposed LMSS.

computation overheads still remain high, and the *Quality of Service* (QoS) cannot satisfy some requirements of multimedia applications. In this paper, we borrow the idea of P2P content distribution technologies and design a distributed load balancing strategy for LMSSs. A good survey on P2P content distribution technologies before 2005 can be found in [33].

3 System Model and Problem Formulation

In this section, we shall formally introduce the problem that we are tackling and propose a mathematical model used in our analysis. We will also introduce the required terminology, notations, and definitions that will be used throughout this paper. In the following, we first present a system architecture in the design of our strategy.

We consider the system architecture of LMSS as shown in Fig. 1, where we only present the main path of the requests and omit the technical details such as Authentication, Authorization, and Accounting (AAA), the setup of secure sessions, etc. In this figure, there are three main steps for the requests: 1) the registered users send requests to the MDSs of LMSS, 2) the MDSs forward the requests to the primary OSSs, which are in charge of the requested objects, and 3) one of the OSSs pushes the data stream to the client who has sent the request.

The LMSS is composed of N multimedia OSSs, storing a total of M different objects $o_1, o_2, \dots, o_M$, where $M \gg N$. We use O to denote the set of objects in the system. For every server S_j, there are M_j objects saved in it, and O_j is denoted as the set of objects belonging to S_j. Obviously, $O_j \subset O$ and $O = O_1 \cup O_2 \cup \cdots \cup O_N$ hold. Initially, we assume that $O_1 \cap O_2 \cap \cdots \cap O_N = \emptyset$, and we define S_j as the primary server of object $o \in O_j$, which will take charge of operations on the objects in O_j. Each S_j has a fixed network bandwidth BW_j. When a client retrieves an object o_i from some server in the system, bw_i bandwidth should be assigned to this session to guarantee the continuous-time presentation. Obviously, in S_j, the total bandwidths used to serve different requests should be not more than BW_j at any time.

Clients generate requests for data retrievals from LMSS. Let λ be the total request arrival rate in the entire system. Let object o_i be requested with probability p_i. Clearly, $\sum_{i=1}^{M} p_i = 1$. Without loss of generality, we assume that the distribution of the probabilities fits the Zipf law, and we sort the objects in descending order by using their corresponding probability p_i as the key. Hence, $p_1 \geq p_2 \geq \cdots \geq p_i \geq \cdots \geq p_M$, where $p_i (1 \leq i \leq M)$ is given by (1). Obviously, the request arriving rate for o_i is $\lambda_i = p_i \cdot \lambda$. In our system architecture, when a request for o_i arrives, the request will arrive at a server who is the primary OSS of o_i first. Then, the primary OSS of o_i will decide where the request should be served, locally or on another OSS with a replica of o_i, according to an optimal load balancing solution. If it is decided to be served on another OSS, the request will be forwarded to that OSS, where it will be made to wait in the queue of that OSS and then be served. We assume that the length of the request message can be very short, and the time delay for forwarding can be negligible.

With object o_i requiring bw_i bandwidth to play back without hiccup, the playback time of o_i is $t_i = s_i / bw_i$. Here, we assume that the random variables $t_1, t_2, \dots, t_M$ are identically distributed and mutually independent. Therefore, the service rate $\mu_i = 1/t_i = bw_i / s_i$, where s_i is the size of o_i. Considering (1), we can obtain that the expected service rate of a request for an object o in the system is $\mu_o = E[\mu_i] = \sum_{o_i \in O} p_i \cdot \mu_i$, and the expected playback time of an object is $\overline{X} = E[t_i] = 1/\mu_o$. Here, we use $\overline{X^2}$ to denote the second moment of playback time, which is given by $\overline{X^2} = E[t_i^2] = \sum_{o_i \in O} p_i \cdot t_i^2$. We can also obtain the expected bandwidth used for retrieving an object o as $bw_o = E[bw_i] = \sum_{o_i \in O} p_i \cdot bw_i$. Hence, the expected number of requests that can simultaneously be served on S_j is $n_j = \lfloor \frac{BW_j}{bw_o} \rfloor$, which can be interpreted as the expected number of channels that exist in S_j. Then, we can model the OSS as an $M/G/m$ discipline. We assume that $\sum_{j \in system} \lambda_j = \lambda$, where λ_j is the request arrival rate served on S_j. Based on the deductions in [17], [36], we can obtain the following equations. In an $M/G/m$ system, the probability that a request has to wait (approximately) is given as

$$P_Q = P\{Queuing\} = \frac{p_{j0} (n_j \rho_j)^{n_j}}{n_j! (1 - \rho_j)}, \tag{2}$$

where $\rho_j = \frac{\lambda_j}{n_j \mu_o}$, $(\rho_j < 1)$, and p_{j0} is the probability that there is no request waiting in the queue of S_j, which is

$$p_{j0} = \left[\sum_{k=0}^{n_j - 1} \frac{(n_j \rho_j)^k}{k!} + \frac{(n_j \rho_j)^{n_j}}{n_j! (1 - \rho_j)} \right]^{-1}. \tag{3}$$

The expected number of requests waiting in queue (not in service) is given by

$$N_Q = \frac{\rho_j \overline{X^2}}{2 \overline{X}^2 (1 - \rho_j)} \cdot P_Q. \tag{4}$$

Using Little's Theorem [17], we can obtain that the AWT of a request in S_j is

$$T_j(\lambda_j) = \frac{N_Q}{\lambda_j}. \tag{5}$$

Therefore, we can obtain our objective function AWT of the requests as follows:

$$Minimize: \quad F(\lambda) = \frac{1}{\lambda}\sum_{j=1}^{N} \lambda_j \cdot T_j(\lambda_j), \tag{6}$$

$$subject\ to: \quad \sum_{j=1}^{N} \lambda_j = \lambda \ and \ n_j\mu_o > \lambda_j. \tag{7}$$

Hence, in each S_j, it will determine how the arrival requests λ_j can independently be distributed among other OSSs in order to minimize (6). For the derivation of (2), interested readers can refer to [36] for more details on the AWT of requests in an $M/G/m$ system. In the following, we shall describe our proposed strategy used in every OSS to attack the optimization problem (6) and (7). In the Appendix, we present Table 3 which contains a list all the notations used in this paper.

Note that recently, a number of researchers have proposed new architectures that make use of network multicast to achieve vastly improved efficiency such as batching and patching [25], [35]. Batching groups requests for the same object and then serves them by using a single multicast channel. Patching exploits the client-side bandwidth and buffer space to merge requests from separate transmission channels into an existing multicast channel. However, all of these technologies are based on the assumption that when a request starts to be served, the probability that there are one or more requests for the same object arriving at the system in a short time is very high. In our models, the number of objects is large, and the request arrival rate for a particular object is very low. That probability in our system may be very small. Hence, in this paper, we consider one channel used for only one request.

4 Proposed Strategy for an Optimal Solution

As discussed in Section 1, we shall determine the placement and number of replications of the objects existing in the system. First, we shall attempt at answering the question: *Which OSS should be the primary OSS of each object $o_i \in O$ in order to balance the objects among the OSSs*? To address this problem, we have many strategies such as random distribution [32] and the hashing strategy [8]. Random distribution can guarantee that the workload is uniformly distributed in the server cluster, and it is easy for implementation. However, when a request arrives at the MDS server for an object, the server has to search in the data structure, for example, B^+ tree, to determine the primary server of the object. The size of the structure grows in proportion to the number of objects. On the other hand, the hashing strategy widely distributes objects across servers based on some unique object identifiers such as the object ID or pathname, providing more balanced workload in the cluster. With well-defined hashing functions, the workload can evenly be distributed across the cluster, and clients can directly locate and contact the responding multimedia server. Hence, in this paper, we choose the hashing strategy to distribute the objects in O among the servers. Once an object o_i is determined to store in S_j, S_j becomes the primary OSS of o_i and takes charge of o_i. For ease of understanding, we shall discuss our algorithm used in each OSS to balance the requests in the following section.

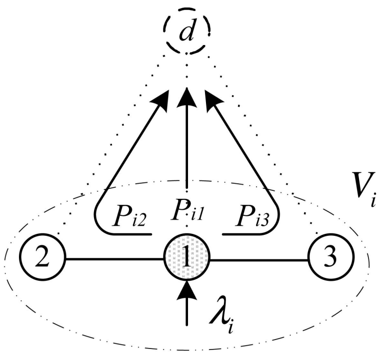


Fig. 2. Virtual routing path with virtual destination node d.

4.1 Balancing the Requests among the Replications

Without loss of generality, we first assume that there are some replications for object o_i existing in the system, and we attempt to balance the requests for o_i among the OSSs. If we consider object o_i only, we can observe that there is a set of OSSs V_i, and for each $S_j \in V_i$, $o_i \in O_j$. Now, we can define a subproblem of (6) defined as follows:

$$Minimize: \quad F(\lambda_i) = \frac{1}{\Phi}\sum_{j\in V_i} \lambda_j \cdot T_j(\lambda_j), \tag{8}$$

where $\Phi = \sum_{j\in V_i} \lambda_j$. We use λ_{ij} to denote the rate of the requests for o_i arriving at S_j. Clearly, we have $\sum_{j\in V_i} \lambda_{ij} = \lambda_i$. Note that the difference between λ_i and λ_j is that λ_i is the request arrival rate for object o_i in the system, whereas λ_j is the request arrival rate that will be served by server S_j.

Now, we have obtained the subsystem, as shown in Fig. 2, for example, where servers S_1, S_2, and S_3 belong to V_i, and S_1 is the primary OSS of object o_i. As shown in this figure, all of the requests for o_i will arrive at S_1 first, and S_1 shall determine an optimal solution $\bar{\lambda} = \{\bar{\lambda}_{i1}, \bar{\lambda}_{i2}, \bar{\lambda}_{i3}\}$ to minimize the function (8).

At first, we add a *virtual* destination node d into the subsystem in Fig. 2. Connect node d with each $S_j \in V_i$ by *virtual* links, respectively. Let the AWT function (2) of S_j be treated as the communication delay function on link (j, d). We can also consider some communication delays of the paths, for example, (1, 2) and (1, 3), for the requests forwarding following the models used in [11], [17]. Note that the primary OSS may not be the neighboring node of the other OSSs with a replication, and there may be multiple links along the paths between them. After these modifications, the procedure of balancing the requests can be described in an alternative way. As shown in Fig. 2, a three-node system has been transformed into a datagram network, in which S_i basically acts as a *router*. Referring to this figure, we observe that for the primary OSS, it can consider three paths to reach node d via S_j $(j \in V_i)$, and the paths are $S_1 \rightarrow d$, $S_1 \rightarrow S_2 \rightarrow d$, and $S_1 \rightarrow S_3 \rightarrow d$, denoted as P_{i1}, P_{i2}, and P_{i3}, respectively. Here, we denote $\mathcal{P}_i$ as the set of routing paths of o_i.

The way in which the requests for o_i are balanced among servers in V_i can be described as follows: The requests for o_i that arrive at its primary OSS according to a Poisson process with an average external arrival rate of λ_i are routed to destination node d via every S_j. The goal of load balancing now can alternatively be stated as a problem that attempts at minimizing the mean link delay for each request for o_i in the system. Then, the primary OSS only considers the routing paths of $\mathcal{P}_i$. For each routing path, we use the *first derivative length* (FDL) to denote the first derivative of the delay function along the path. For example, the FDL of path $S_1 \rightarrow d$ is

$$[\lambda_1 \cdot T_1(\lambda_1)]' = \frac{\partial[\lambda_1 \cdot T_1(\lambda_1)]}{\partial \lambda_1}.$$

Then, we use *Newton's method* [17], [31] to obtain an optimal request routing rate of each path. Eventually, we obtain an optimal request transferring rate λ_{ij}. Now, we state the following theorem:

Theorem 1. *The set of values of $\bar{x}_i$ is an optimal solution to problem (8)* **only if** *the request flow travels along the minimum FDL (MFDL) paths for each S-D pair. In addition, if F is assumed to be convex, then $\bar{x}_i$ is optimal* **if and only if** *the request flow travels along the MFDL for each S-D pair.*

Proof. We shall first prove the first part of the Theorem 1. Let $\bar{x}_i = \{\bar{x}_{P_{i1}}, \ldots, \bar{x}_{P_{ik}}\}$, $o_i \in O$, and $k \in V_i$ be an optimal path flow vector. Then, if $\bar{x}_{P_{ik}} > 0$ for some path P_{ik} of an S-D pair $\mathcal{P}_i$, we must be able to transfer a small amount $\delta > 0$ from path P to any other path P' of the same S-D pair without decreasing the mean response time. Otherwise, the optimality of $\bar{x}_i$ would be violated. For the first derivative, the change in the mean response time from this transfer is

$$\delta \frac{\partial F(\bar{x}_i)}{\partial x_{P'}} - \delta \frac{\partial F(\bar{x}_i)}{\partial x_P}.$$

Since this change must be nonnegative, we obtain

$$\bar{x}_P > 0 \Rightarrow \frac{\partial F(\bar{x}_i)}{\partial x_{P'}} \geq \frac{\partial F(\bar{x}_i)}{\partial x_P}, \; for \; all \; P' \in \mathcal{P}_i. \tag{9}$$

In other words, the optimal path flow is positive only on paths with an *MFDL*.

Thus, the condition (9) is a *necessary* condition for the optimality of $\bar{x}_i$. The above proof is also valid for the *only if* part of the second part of Theorem 1 for convex functions. Now, the above proof can be backtracked to prove the second part of the theorem when we assume that the functions F are convex, for example, when the second derivatives F'' exist and are positive in the domain of the definition of F. □

Referring to Fig. 2, we can state the following lemma, the proof of which is evident according to this figure:

Lemma 1. *The optimal request rate $\bar{x}_P$ on path P is equal to an optimal solution of $\bar{\lambda}_{ij}$, where j is the virtual router on path P.*

From Theorem 1, we can observe that an optimal solution results only if a request flow travels along MFDL paths for each S-D pair. However, by transferring all the flows of each S-D pair to the MFDL path will lead to an oscillatory behavior. Thus, it is more appropriate to transfer only part of the flow from other paths to the MFDL path. We shall determine the amount of these flows from each of the other paths between an S-D pair to be transferred to the MFDL in each iteration r to seek an optimal solution.

TABLE 1
Procedure of Balancing λ_i in V_i

```
Step 1: Initialization
    Find a feasible solution $x$ to satisfy $\sum_{P \in \mathcal{P}_i} x_P^{(r)} = \lambda_i$.
    $r = 0$ ($r$: iteration index).
Step 2: Solution Procedure
    Let $r = r + 1$.
    For $k = 1$ to $v_i, \; v_i = |V_i|$
        Calculate FDL of the paths $P$, $d_P^{(r)}, \; P \in \mathcal{P}_i$.
    End For
    Find MFDL paths $\bar{P}^{(r)}$ among $\mathcal{P}_i$.
    For $k = 1$ to $v_i - 1$
        To path $P_k, \; P_k \in \mathcal{P}_i, \; P_k \neq \bar{P}^{(r)}$, calculate $H_{P_k}$.
        Let $x_{P_k}^{(r)} = max\{0, \; x_{P_k}^{(r-1)} - \alpha^{(r-1)} H_{P_k}^{-1} (d_{P_k}^{(r-1)} -$
            $d_{\bar{P}^{(r-1)}}^{(r-1)})\}$. Let $\lambda_{ij}^{(r)} = x_{P_k}^{(r)}$, if $j$ in path $P_k$.
    End For
    Let $\lambda_{ij}^{(r)} = \lambda_i - \sum_{P \in V_i, \; P \neq \bar{P}^{(r)}} x_P^{(r)}$, $j$ in path $\bar{P}$.
Step 3: Stopping Rule
    If $|F(\lambda_i^{(r)}) - F(\lambda_i^{(r-1)})|/F(\lambda_i^{(r)}) < \varepsilon$, then stop, where $\varepsilon$
        is a desired acceptable tolerance for solution quality;
    otherwise, go to Step 2.
```

For each S-D pair (s, d), let $\bar{P}$ be an MFDL path. The minimization problem (8) can now be transformed to a problem involving only active (strictly positive) constraints by expressing the flows of MFDL paths $\bar{P}$ in terms of the other path flows while eliminating the passive (equality) constraints. From [17], [29], we can obtain that a basic iteration takes the form

$$x_P^{(r+1)} = max\{0, (x_P^{(r)} - \alpha^r H_P^{-1}(d_P - d_{\bar{P}}))\}, \tag{10}$$

$$x_{\bar{P}}^{(r+1)} = \lambda_i - \sum_{P \in \mathcal{P}_i, P \neq \bar{P}^{(r)}} x_P^{(r)}, \tag{11}$$

where $P \in \mathcal{P}_i$, $P \neq \bar{P}$, d_P, and $d_{\bar{P}}$ are the FDL of the paths P and $\bar{P}$, respectively. H_P is the sum of the "second derivative length" of P and $\bar{P}$, which are the second derivative of the delay functions along paths P and $\bar{P}$, respectively. For example, if j_1 is in P, and j_2 is in $\bar{P}$, then H_P is given by

$$H_P = [x_P^{(r)} \cdot T_{j_1}(x_P^{(r)})]'' + [x_{\bar{P}}^{(r)} \cdot T_{j_2}(x_{\bar{P}}^{(r)})]''. \tag{12}$$

The step size α^r is some positive scalar, which may be chosen by a variety of methods available in the literature [30]. We set α^r to 0.5 for all r, which works well in our method. A pseudocode of the algorithm is also shown in Table 1.

4.2 Object Replication Determination

In order to obtain an optimal combination of servers[1] to minimize the AWT function (6) and (7), we need to test every potential combination of the servers and find an optimal set of V_i for every object o_i. However, this incurs a

1. This involves both the number of servers and the choice of servers.

complexity of $O((N-1)!)$ for each o_i, and it is impractical in real-life situations. To handle this overly complex scenario, we propose a heuristic method to determine an optimal set of V_i for object o_i in S_j. In the worst case, the complexity of our method is $O(N \log N)$.

From Theorem 1, we can observe that in an optimal solution, if some part of λ_i is determined to travel along path P, the FDL of path P should be a minimum among all the paths. In our method, we attempt at searching and finding an optimal set of V_i for each o_i in order to minimize (6) according to the FDL of each path in the system. For $o_i \in O_j$, S_j takes charge of its replication and request balancing. Initially, $V_i^{(0)} = \{S_j\}$ only. S_j calculates the FDL of all the potential paths via every other server in the system and then sorts the servers in ascending order according to the value of FDL. Without loss of generality, we use the set $\{S_{j_1}, S_{j_2}, \dots, S_{j_{N-1}}\}$ to denote the servers (excluding S_j) in such an order. Then, we compute the system AWT $F(\lambda^{(0)})$. In the following, we use $T_{gain}^{(r)} = F(\lambda^{(r-1)}) - F(\lambda^{(r)})$ as the time gain in the rth iteration. Now, we can describe a heuristic procedure of our algorithm to obtain an optimal set of servers $\bar{V}_i$ for each o_i.

In the first iteration, we add server S_{j_1} into $V_i^{(0)}$ and, then, we obtain $V_i^{(1)} = \{S_j, S_{j_1}\}$. Server S_{j_1} will have a replication of o_i, and it can serve some requests for o_i. Using the algorithm described in Table 1, we can obtain an optimal solution of $\bar{\lambda}_i^{(1)}$, which is $\{\bar{\lambda}_{ij}^{(1)}, \bar{\lambda}_{ij_1}^{(1)}\}$. We can calculate $F(\lambda^{(1)})$ and, then, we can obtain $T_{gain}^{(1)}$. If $T_{gain}^{(1)} > 0$, then the procedure comes into the second iteration; else, the procedure stops, and an optimal solution of $\bar{V}_i$ is $\bar{V}_i = V_i^{(0)} = \{S_j\}$.

In the second iteration, server S_{j_2} is added into $V_i^{(1)}$, and $V_i^{(2)}$ becomes $\{S_j, S_{j_1}, S_{j_2}\}$. Using the algorithm in Table 1 again, we can obtain an optimal solution of the requests for o_i, which is $\{\bar{\lambda}_{ij}^{(2)}, \bar{\lambda}_{ij_1}^{(2)}, \bar{\lambda}_{ij_2}^{(2)}\}$. Calculating $F(\lambda^{(2)})$ by using the optimal solution, we can then obtain $T_{gain}^{(2)}$. If $T_{gain}^{(2)} \le 0$, stop the procedure, and $\bar{V}_i = \{S_j, S_{j_1}\}$; else, the procedure will come into the next iteration. This iterative procedure will be repeated until the last server in the system.

4.3 Load Balancing Strategy

Now, we shall present our strategy for solving the load balancing problem (6) by using the methods described above. Initially, some objects are assigned to a server, and this server becomes their primary server. All of the requests for these objects come to the primary server first, and the server takes charge of replicating the objects and distributing the requests among the system. Although it is a distributed system, due to the fact that the load balancing solution can be computed *offline*, each server can obtain some system information and can cooperate to achieve a global optimal solution.

On S_j, the request rate for different $o_i \in O_j$, λ_i, is different, and the service rate of each o_i, μ_i, is also different. From [17], we can obtain the expected number of channels used for o_i, which is $E[o_i \text{ Channels}] = \frac{\lambda_i}{\mu_i}$. Without loss of generality, we sort the objects in O_j as

TABLE 2
Our Proposed Strategy

```
For $j = 1$ to $N$.
    Calculate the expected number of channels of each object $o_i \in O_j$.
    Sort the objects in descending order according $E[o_i\ Channels]$
        and construct $O_j = \{o_{j_1}, o_{j_2}, \cdots\}$.
End for.
$r = 1$.
Loop:
    $l = 1$.
    Calculate $F(\lambda^{(r)})$.
    Loop1:
        For $j = 1$ to $N$, in server $j$, do.
            Find optimal replications of $o_{j_l} \in O_j$, $\bar{V}_{j_l}$.
            Using the method in Table 1 to balance the requests for $o_{j_l}$.
        End for.
        If all objects have been calculated, break loop1.
        $l = l + 1$.
    End loop1.
    $r = r + 1$.
    Calculate $F(\lambda^{(r)})$
    If $|F(\lambda^{(r)}) - F(\lambda^{(r-1)})|/F(\lambda^{(r)}) < \varepsilon$, then stop;
    otherwise continue.
End loop.
```

$\{o_{j_1}, o_{j_2}, \cdots\}$ in descending order by using the corresponding expected number of channels as the key. If it happens that two objects have the same expected number of channels, we place the object with smaller size before the other. In Table 2, we present the workings of our strategy for static load balancing. In the first iteration, each S_j in the system will determine an optimal set of servers for object $o_{j_1} \in O_j$ and balance the requests for it by the method shown in Table 1. In the second iteration, each S_j will determine the replications and balance the requests for o_{j_2} and so on. The server will continue the iteration until the last object in O_j. If all objects are considered, the first loop terminates. We can run the loop several times to achieve an acceptable tolerance, hence obtaining a near-optimal[2] solution to our original problem (6).

5 Dynamic Load Balancing Strategy

The load balancing strategy proposed in Section 4 is based on the assumption that the total request arrival rate of the system λ is known and fixed. However, in real-life situations, the request arrival rate is dynamically changed. Due to the high computation and communication overheads, it is impossible to frequently run the proposed strategy. In this section, we will extend our method to dynamic situations. First, we shall discuss the request arrival patterns.

5.1 Request Arrival Pattern Description

As the Internet-connected population increases, we can observe that events external to the Internet conspire to create periods of significant fluctuation, and we shall employ a model for creating varied request arrival patterns.

As in [7], it is wiser to model the request arrival based on a 24-hour period. In Fig. 3, we propose a baseline steady-state request arrival pattern and three nonsteady-state request

2. If the tolerance is small enough, the solution can be considered as an optimal one for real-life applications. Interested readers can refer to [11], [23] for the proof.

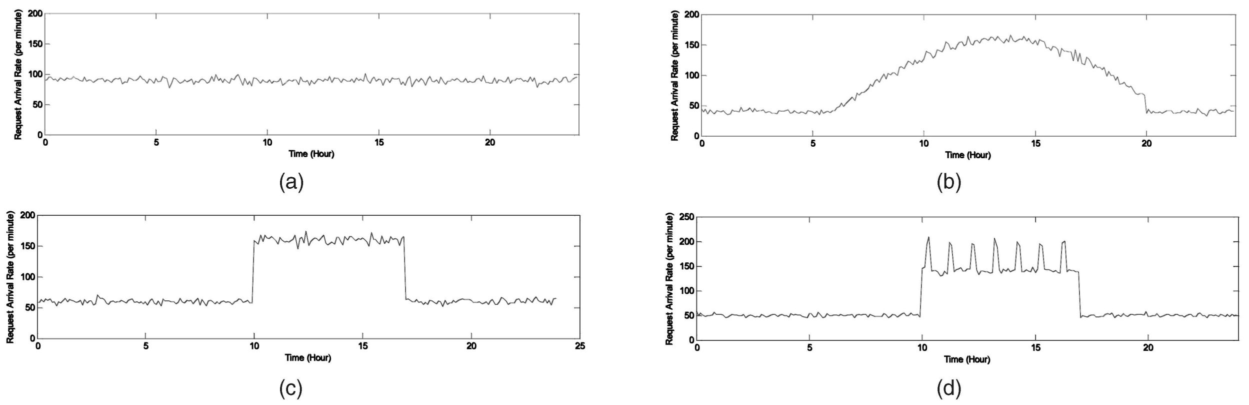


Fig. 3. Various request arrival patterns. (a) Steady. (b) Gradual increase and decrease. (c) Square waveform. (d) Square waveform with hourly spikes.

arrival patterns. The horizontal axis represents time, and the vertical axis shows the number of requests per minute during a 6-minute (0.1-hour) time interval. Fig. 3a shows a steady-state request arrival pattern with a Poisson distribution, which is used as a comparison with the other patterns. For the nonsteady-state patterns, we use the statistics that suggests that the request arrival rates during the peak hour can approximately be four to five times greater than the normal rate. Figs. 3b and 3c show two patterns: one with a gradual increase to high rate and then a gradual decrease to low rate and one with a "jump" to a high rate. The pattern shown in Fig. 3d is a special case, with the rate surging at hourly intervals. Changes in the request arrival patterns are implemented using an exponential function with changing averages [24].

5.2 User Waiting Behavior

Internet users can, in general, exhibit completely arbitrary and random behavior, and we must select an appropriate model of behavior in order to evaluate the system performance. The model used in this paper is based on the users' unwillingness to wait for service more than some finite amount of time [37], which closely resembles how users behave in the Internet nowadays. As an illustrative example, consider a user surfing the Web, who will click and wait. Unless there is some indication of progress, the user will become impatient and very quickly switch elsewhere. Because most systems process requests by using the *FCFS* queuing principle, the user's request would have eventually been satisfied. However, during high-load periods, the waiting time for a user may be very long, beyond some user-imposed maximum tolerable waiting time. As a result, our model assumes that users agree to wait at least some predefined amount of time plus a given average from an exponential distribution. For example, a user might be willing to wait for 5 minutes to watch a movie plus some random amount of additional time up to 2.5 minutes. If a channel has not been assigned to the user after this period of time, the user is assumed to withdraw the request. Following the terminology consistent with related work, we call this process as *Denial of Service* (DoS), which is analogous to a request that cannot be satisfied and is a key performance metric for our systems.

5.3 Dynamic Load Balancing

The strategy proposed in Section 4 can determine the number of replications for each object existing in the system and balance the requests among the OSSs. This is based on the assumption that the request arrival rate is constant and fixed. In real-life applications, it is impossible to assume λ as a constant when the system is running. Hence, we shall modify and extend our proposed strategy to dynamic situations.

Through the analysis of the request arrival patterns, we can observe that the number of requests for a particular object may rapidly change. In this case, due to the fact that the size of the object may be large and the communication capacities of the network are limited, it is impossible for the system to add new replication of the object in a short time in order to meet the increasing requests. Hence, before the system starts running, we shall estimate the worst cases and make sure that if the request arrival rate is increasing or decreasing, the system utilization of each server can be proportionally increasing or decreasing.

In order to maximize the number of clients that the system can simultaneously serve, we shall maximize the utilization of the entire system. However, if the utilization achieves 100 percent, according to queuing theory, we can observe that the AWT of each request will be infinite, and the system will be overloaded [17]. Hence, in our proposed strategy, we aim at a threshold, for example, 90 percent system utilization. If the system utilization exceeds the threshold for a long time, the system will start sending warning signals to the administrator and remind that new servers shall be added into the system to meet the current requirement. Once the system utilization is determined, we can obtain the threshold value of request arrival rate. Then, we can determine the set of replications for each object and balance the requests among the replications according to the request arrival rate by using the strategy presented in Table 2.

Once the system has been balanced by our proposed strategy according to the threshold value of request arrival rate, the primary server of each object o_i has a mapping of the servers with the replications of o_i, V_i, and then transfers the request to one of the servers in V_i according to the near-optimal solution obtained by our proposed algorithm. We can obtain the probability of a request for o_i served in server

$k(k \in V_i)$ by $p_{ik} = \frac{x_{P_k}}{\lambda_i}$, where x_{P_k} is the request rate on path P_k. Obviously, $\sum_{k \in V_i} p_{ik} = 1$. When the system is running, the servers can determine where a request for o_i should be served according to the probability p_{ik}. We name the method described above as a probability-based distribution method. In the following, we shall present a lemma, which is very important for the dynamic situations:

Lemma 2. *By using the probability-based distribution method in our proposed strategy, the system utilization of each server is proportional to the request arrival rate.*

Proof. We assume that the threshold request arrival rate is $\lambda_{threshold}$, with which we can obtain p_{ik} for each $o_i \in O$ and each server $k \in V_i$ by using our proposed strategy. We define $\lambda(t)$ and $\lambda_i(t)$ as the request arrival rate of the system and the request arrival rate for object o_i at time t, respectively. For server S_k, we have $O_k \in O$, and for each object $o_i \in O_k$, it has its own arrival rate on S_k at time t which is

$$\lambda_{ik}(t) = p_{ik} \cdot \lambda_i(t) = p_{ik} \cdot p_i \cdot \lambda(t), \tag{13}$$

where p_{ik} and p_i are constants. From (13), we can obtain that the request arrival rate that will be served by server S_k is

$$\lambda_k(t) = \sum_{o_i \in O_k} \lambda_{ik} = \sum_{o_i \in O_k} p_{ik} \cdot p_i \cdot \lambda(t) = C_k \cdot \lambda(t), \tag{14}$$

where C_k is a constant. Using (14), we can obtain the system utilization of S_k:

$$\rho_k(t) = \frac{\lambda_k(t)}{n_k \cdot \mu_o} = \frac{C_k}{n_k \cdot \mu_o} \cdot \lambda(t).$$

Obviously, $\rho_k(t) \propto \lambda(t)$. □

From (4) and (5), we can observe that the AWT of the requests is optimal only at the point when $\lambda(t) = \lambda_{threshold}$. To overcome this disadvantage, we can follow the method suggested in [24], where the authors repeat the algorithm to regularly balance the requests at a fixed time interval. However, when the system utilization of the server is very high, the communication and computation overheads prevent the server from repeating such calculations. If we carefully choose $\lambda_{threshold}$, we can achieve the optimal AWT in most cases, and in other cases, Lemma 2 guarantees that the system can near optimally be balanced. Hence, in our strategy, we repeat the computation only when it is necessary, for example, when some predefined load unbalancing occurs.

5.4 Benchmark Algorithm

In [20], Shen et al. employed a random polling-based load balancing policy for the clusterwide request distribution. In their project named "Neptune," each data has several replications, and they exist in different servers with a uniform distribution. For every request access, the request client polls several randomly selected servers for load information and then directs the request access to the server responding with the lightest load. In practice, Neptune's cluster load balancing is based on a random polling policy with a poll size of three, which means that the request client only sends a polling request to three randomly chosen servers instead of all the servers with a replication. If there are not more than three servers that have a replication, the client will send polling requests to all the servers with a replication. In their work, Shen et al. measured that a UDP-based polling round-trip cost is around 290 μs when the servers are idle. However, if the server is busy, it may take much longer than that to respond to the polling request. Hence, the polling policy will discard polls not responded to within 10 ms in order to achieve fine-grained services.

In Neptune, the waiting time of each request includes the polling round-trip delay, which is determined by the latest response from the servers and the time delay for directing the request to the corresponding server. Since it can provide the client with the most accurate real-time system status with relatively small communication overhead, it is natural to use Neptune as the benchmark algorithm in our work and compare it with our proposed strategy to examine their performances.

6 Performance Evaluation and Discussions

In our simulations, we employ a discrete-event approach to model, simulate, and evaluate the systems [28]. We assume that the multimedia servers are connected by an intranet or Internet. It is assumed that the queue discipline of the requests in each server is *FCFS*. We also assume that the multimedia servers can be normal PCs or workstations, and the system can be homogeneous or heterogeneous. Each object is assigned to a server by a simple hashing function using the object ID as the key, and the server will become the primary server of the object. In order to guarantee the real-time delivery requirement of the continuous media, the servers have to provide each request with a minimum bandwidth for playback. For ease of simplicity, we fix the bit rate of each object to be 1 megabit per second (Mbps) with an advanced codec (for example, MPEG-4). The lengths of the objects are also randomly generated following a uniform distribution with the range from 1,200 to 3,600 seconds. Hence, the expected playback time of each object is 2,400 seconds (40 minutes). We simply assume that the size of each object is equal to its length times the bit rate (CBR). We assume that the delay of each communication such as polling sending, responding, and request forwarding follows a uniform distribution within [1, 5] ms, and the probability that the communication may be failed is set to 1 percent.

For each simulation, the warming time is 86,400 seconds (24 hours), and the simulation time is set to 86,400 seconds in order to obtain an accurate statistical AWT of the requests arriving at the system. We believe that such a setting is sufficient for illustrative purposes. We repeat each simulation 15 times and use the average value as the final result. The confidence of all the simulations is over 95 percent. Here, we use ρ_{system} to denote the entire system utilization, which is given as follows:

$$\rho_{system} = \frac{\lambda}{\sum_{j=1}^{N} n_j \cdot \mu_o}.$$

We use synthetic request arrival patterns in our simulations on the performance comparisons. Two categories of

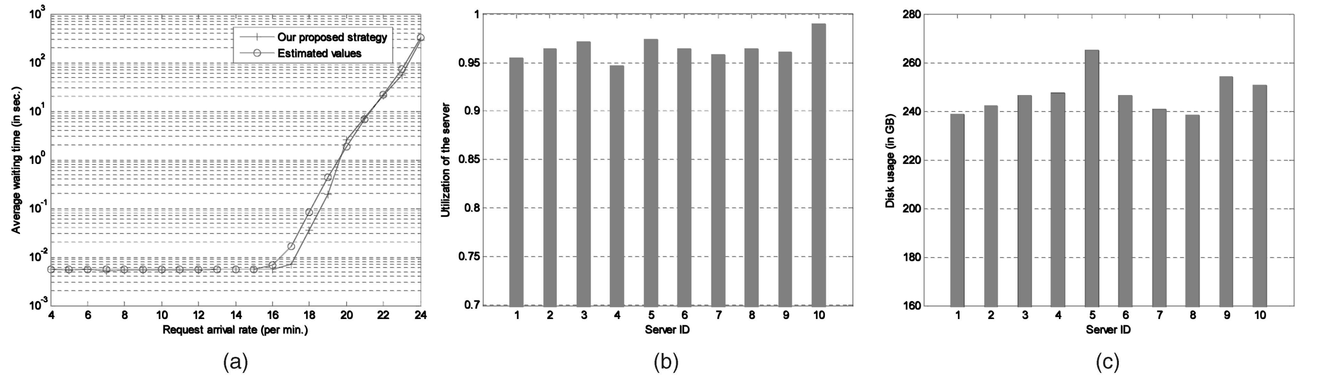


Fig. 4. (a) Comparison of our proposed strategy and estimated values with respect to the AWT. (b) System utilization of each server when $\lambda = 24$. (c) Disk utilization of each server (in gigabytes) when $\lambda = 24$.

request arrival patterns are generated for this purpose: 1) *Steady* requests, which includes the steady arrival patterns, as shown in Fig. 3a, and 2) *Dynamic* requests, which includes gradual increase and decrease arrival patterns, sudden increase and decrease arrival patterns, and sudden increase with hourly spikes arrival patterns, as shown in Figs. 3b, 3c, and 3d, respectively. For ease of comparison, we conduct simulations on Neptune under no replication and replication degrees of 1 and 2. The simulation software is written in C#, and the platform used is Intel Pentium 4 (3.2 GHz) with 1 Gbyte of RAM.

6.1 Load Balancing under Steady Requests

In this section, we will carry out two types of experiments by using a small-scale computer system, which consists of 10 servers and 1,000 objects. The expected playback time $\overline{X}$ and standard deviation σ of the objects are 2,430.4 and 711.0, respectively. In these simulations, the total size of the objects is 2.24 Tbytes. In Experiment 1, we shall examine the performance of our proposed strategy under different system utilizations and compare the AWT of the requests with the theoretically estimated values. In Experiment 2, we shall compare our proposal strategy with the polling-based algorithms with respect to the AWT of the requests, the rate of DoS, and disk utilizations.

Experiment 1. In this experiment, we assume that the users will wait indefinitely and no DoS will occur. The system under consideration is a homogeneous system, and each server has 100-Mbps bandwidth and 500-Gbyte hard disks. First, we set $\lambda = 4/\text{minutes}$ and then increase the value of λ by 1 per step until $\lambda = 24$, at which $\rho_{system} = 0.97$, and the system is overloaded. At each step, we use (4) and (5) to obtain the estimated AWT of the system. We present the simulation results in Fig. 4.

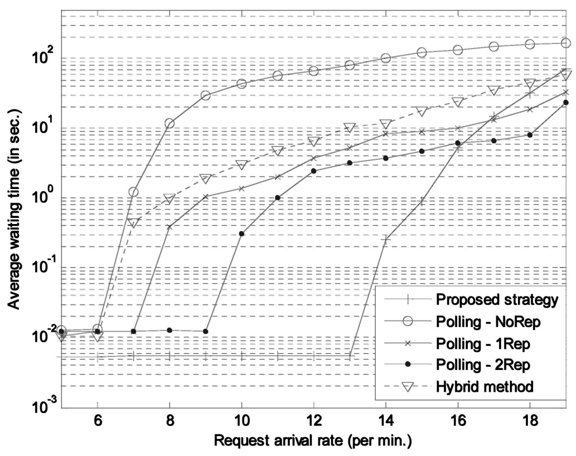
In Fig. 4a, we compare the AWT of our proposed strategy with the estimated values given by (4) and (5). To our surprise, we find that in some cases, the AWT of our proposed strategy is even better than the estimated values. However, it is true, because as compared with the well-established $M/M/m$ system, the $M/G/m$ system is much more complicated, and there is no accurate expression for it in the literature. Equations (4) and (5) given in [36] are approximate solutions, and the numerical results in this reference have shown that these approximations tend to overestimate the AWT of the requests in an $M/G/m$ system. In [36], Hokstad mentioned that there are some other more accurate solutions for $M/G/m$ systems. However, these approaches are too complicated to be used in real-life situations, and (4) and (5) are good enough for applications.

In order to prove the load balancing ability of our proposed strategy, in Fig. 4b, we present the system utilization of each server when the system is very heavily loaded ($\rho_{system} = 0.97$). Based on this figure, we can observe that even in such an extreme situation, our proposed strategy works well, and no server is overloaded. In Fig. 4c, we also show the disk usage of each server when the system is heavily loaded. We can observe that the disk usage in the system is also well balanced. It is worth noting that the total disk usage in this case is 2.37 Tbytes, which is only 0.13 Tbytes or 5.8 percent more than the total size of the objects existing in the system. This means that only a small part of the objects, which are the most popular, have replications among the system.

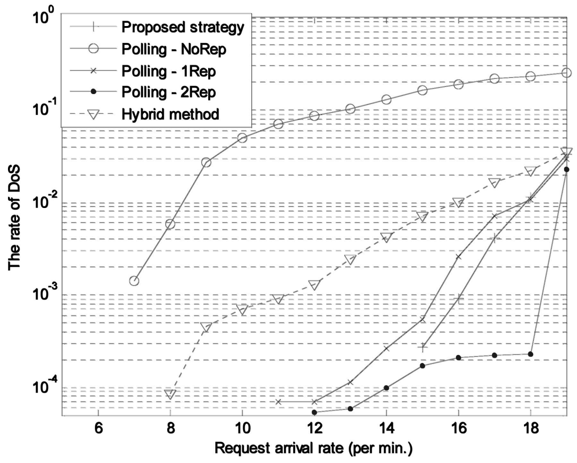
Through these serial simulations, we can conclude that in static situations, our proposed strategy can efficiently balance the load of homogeneous systems by determining the number of replications and placements for a few most frequently requested objects.

Experiment 2. In this experiment, we compare our proposed strategy with three settings of the benchmark algorithm: polling-based algorithms without replication (referred to as *Polling-NoRep*), with one replication (referred to as *Polling-1Rep*), and with two replications (referred to as *Polling-2Rep*). We assume that each user can wait in the queue for 300 seconds, and after that, he can wait for another random time, which is within [0, 150] sesconds, if he still cannot obtain service. The user will withdraw from the queue if the time exceeds and a DoS occurs. The rate of DoS is the ratio of the total DoS to the total requests arrived in the simulation. Here, the system under examination is a heterogeneous system. Six servers have 100-Mbps bandwidth and 500-Gbyte hard disks, and the other four servers have 50-Mbps bandwidth and 300-Gbyte hard disks. We set $\lambda = 2$ first and then increase the value of λ by 1 until $\lambda = 19$, at which ρ_{system} is over 0.96.

We show the simulation results in Figs. 5 and 6. Based on Figs. 5a and 5b, we can observe that when the system load is low and moderate ($\lambda < 15$ or $\rho_{system} < 0.75$), our proposed

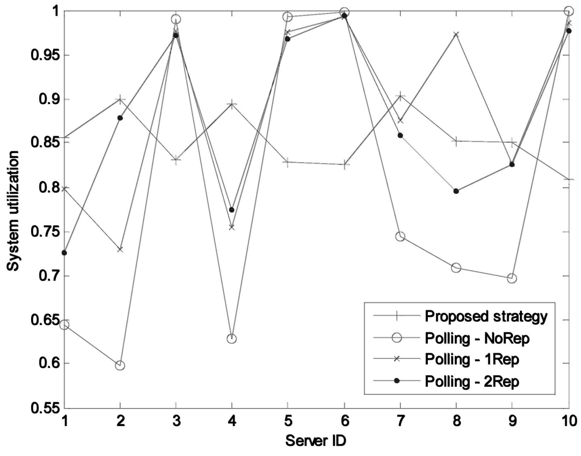


(a)

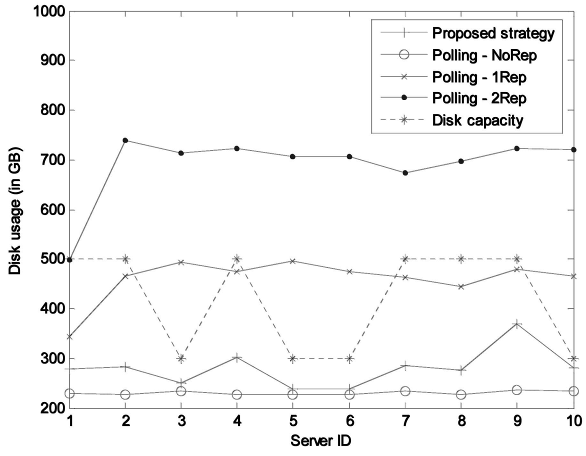


(b)

Fig. 5. (a) Comparison of the algorithms with respect to the AWT. (b) Comparison of the algorithms with respect to the rate of DoS.



(a)



(b)

Fig. 6. (a) System utilization of each server when $\lambda = 17$. (b) Disk usage of each server when $\lambda = 17$.

algorithm performs best among all the algorithms with respect to both AWT and rate of DoS. When the system load becomes higher, *Polling-2Rep* becomes the best one. Through a careful analysis, we find that *Polling-1Rep* performs slightly better than our proposed algorithm only for $\lambda > 17$. The worst performance is by *Polling-NoRep*. From these simulations, we can observe that for polling-based algorithms, one more replication can significantly improve the performance. Note that we omit the points whose values are zero in Fig. 5b, because the y-axis of this figure uses logarithmic scale, and it will be the same for the rest of the figures. Note that in these figures, we also present the performance of a "Hybrid" method, which we will describe later in this section.

Fig. 6a compares the algorithms with respect to the system utilization of each server when $\lambda = 17$ ($\rho_{system} = 0.85$). Based on this figure, we can observe that among the four algorithms, our proposed strategy is the best one, in which all the servers are well balanced, and *Polling-NoRep* is the worst one, the curve of which rapidly fluctuates. It is interesting to find that when $\lambda > 17$, *Polling-2Rep* is the best one, as shown in Fig. 5, where the servers are not balanced well. The reason is that for polling-based algorithms, the requests choose the servers with the shortest queue, regardless of the current server's performance. For example, there are one high-performance server and one low-performance server. A request queues in the low one, because at that moment, the queue length of the low-performance server happens to be slightly shorter. Soon, the high-performance server finishes some requests, and its queue becomes empty, whereas that request is still in the queue. Hence, the probability that low-performance servers will become busier is higher than that of high-performance servers. It can be proven according to Fig. 6a, where the low-performance servers 3, 5, 6, and 10 are much busier than the high-performance servers in the polling-based algorithms. We also compare the algorithms with respect to the disk usage of each server and show the results in Fig. 6b. Based on this figure, we can observe that in our proposed strategy, the disk usage of servers is slightly higher than *Polling-NoRep* but is much smaller than *Polling-1Rep* and *Polling-2Rep*. Furthermore, if we consider the disk constraint of each server (which is indicated by dashed line in this figure), we can find that *Polling-1Rep* and *Polling-2Rep* cannot be applied in this system.

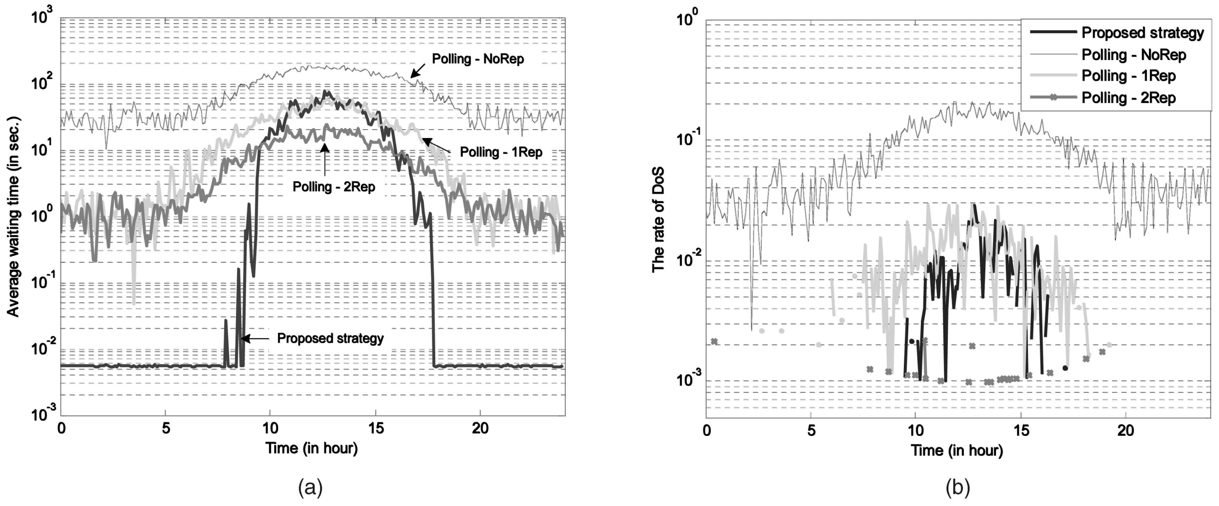


Fig. 7. Gradual arrival rate pattern. (a) Comparison of the algorithms with respect to the AWT. (b) Comparison of the algorithms with respect to the rate of DoS.

From these simulations, we can conclude that our proposed algorithm is very space efficient. This can be explained as follows: Our proposed algorithm can adjust the number and the position of replications for each object based on its request rate. For some most popular objects, they can have more replications; otherwise, for other "cold" objects, they will have no replication, and there is only one copy that exists in the system. Furthermore, the request distribution follows a global optimal solution, and this can guarantee to achieve a best solution. However, for polling-based algorithms, each object has a number of replications, and the position is randomly determined. Although it can achieve some good performance in some cases, it needs more storage space to hold more replications.

We conduct some other interesting simulations, in which we use our proposed algorithm to determine the number and positions of the replications for each object and then use the polling-based method to distribute the requests. This is referred to as the $Hybrid$ method. In order to save some space, we show the results in Fig. 5, which are indicated by dashed lines. Through these simulations, we find that only when $\lambda = 19$ does $Hybrid$ slightly outperform our proposed algorithm, and in the other cases, our algorithm is much better. Comparing $Hybrid$ with the polling-based methods, we find that $Hybrid$ is much better than *Polling-NoRep* but is worse than *Polling-1Rep* and *Polling-2Rep*. Polling-based methods use some system resource as index such as queue length, and it may bring some unfairness into the system. We can observe that in $Hybrid$, the requests for popular objects have many choices, but the requests for "cold" ones have only one choice. This means that the polling-based method will favor the requests coming for popular objects, whereas the requests for "cold" objects will be jammed in their primary servers. Such unfairness can lead to some kind of load imbalance in the system. Our proposed algorithm distributed the requests according to a near-optimal solution and can perform better than the polling-based methods in most cases. However, when the system load is extremely high, it may happen that some servers are jammed by fast incoming requests. The reason is that our proposed algorithm uses a probability-based method to distribute the requests, and there is no mechanism to rebalance the system. Hence, in such situations, polling-based algorithms perform better than our proposed algorithm.

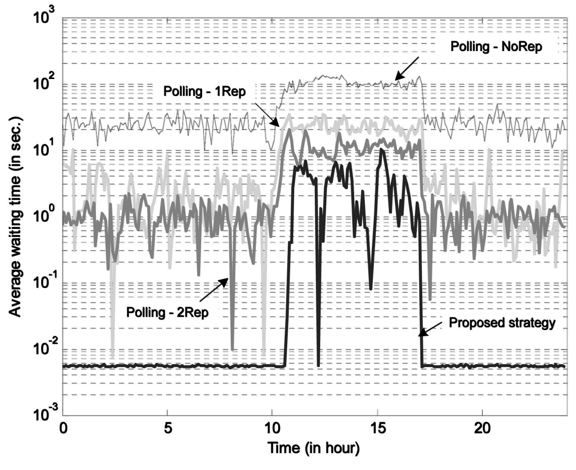
6.2 Load Balancing under Nonsteady Requests

In this section, we will carry out three serials of simulations, in which we set the request arrival patterns to be gradual (Fig. 3b), sudden (Fig. 3c), and spiked (Fig. 3d), respectively. In these simulations, every 6 minutes (0.1 hour), we record the number of requests that obtain the service and the number of requests that quit the system. We also record the waiting time of each request and use the average value of all the requests that leave the waiting queue as the AWT of the time interval.

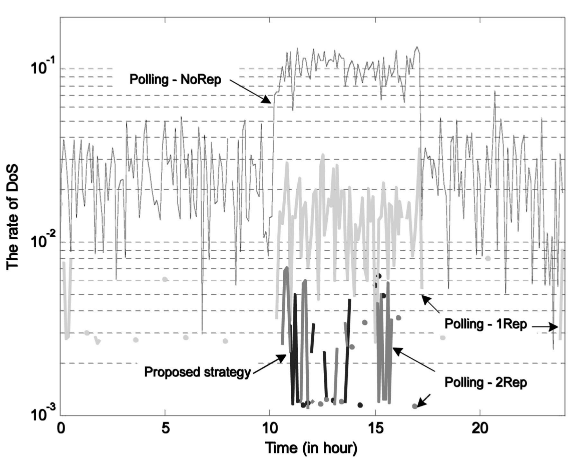
Experiment 3. In this experiment, we construct a large-scale heterogeneous system with 100 servers, among which 50 percent of the servers are low-performance systems with 50-Mbps bandwidth and 300-Gbyte hard disks, and the rest of the servers are high-performance ones with 100-Mbps bandwidth and 500-Gbyte hard disks. We also generate 10,000 objects to be used in the system, $\overline{X}$ and σ of which are 2,435.6 and 689.3 respectively.

For gradual patterns, we show the simulation results in Fig. 7. In Fig. 7a, we can observe that the AWT of all the algorithms increases or decreases following the request arrival pattern. In most cases, our proposed algorithm outperforms the polling strategies. When the request arrival rate increases, the AWT of our proposed algorithm increases, and when the request arrival rate reaches its peak, our proposed algorithm becomes worse than *Polling-2Rep* and very similar to *Polling-1Rep*. The worst one is always *Polling-NoRep*, as one can expect. In Fig. 7b, we can point out that the rate of DoS of our proposed algorithm is kept within a very low level and is similar to that of *Polling-2Rep*, which is much smaller than that of *Polling-NoRep* and *Polling-1Rep*. In all of these simulations, the threshold λ of our proposed algorithm is set to a medium-high value, which is 140 requests per minute.

We also conduct simulations for sudden patterns and show the results of the AWT and rate of DoS in Figs. 8a and 8b respectively. The worst one is still *Polling-NoRep*. For the AWT, our proposed strategy is slight better than *Polling-2Rep* and is much better than *Polling-NoRep* and *Polling-1Rep*. We can observe that the rate of DoS of our proposed

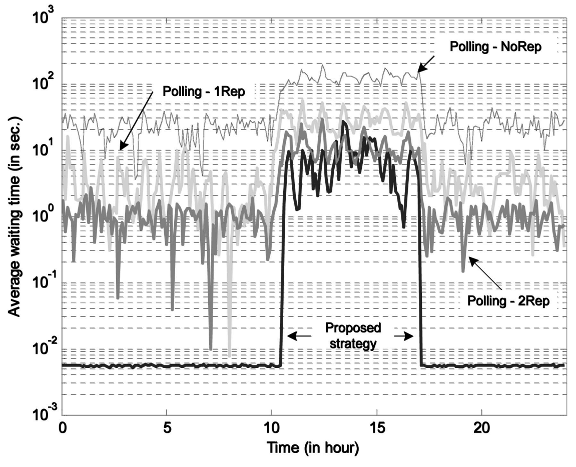


(a)

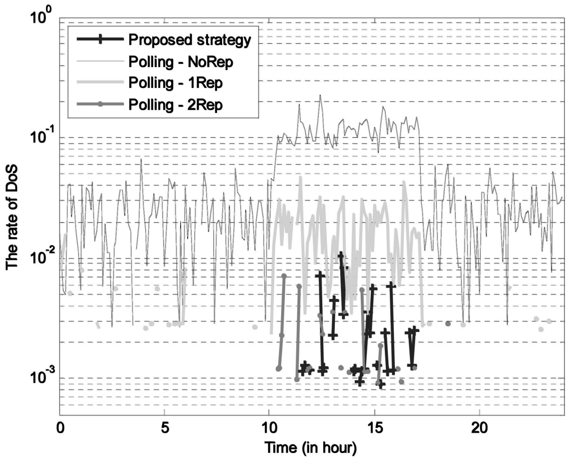


(b)

Fig. 8. Sudden arrival rate pattern. (a) Comparison of the algorithms with respect to the AWT. (b) Comparison of the algorithms with respect to the rate of DoS.



(a)



(b)

Fig. 9. Arrival rate with spikes. (a) Comparison of the algorithms with respect to the AWT. (b) Comparison of the algorithms with respect to the rate of DoS.

strategy is also better than that of the polling-based algorithms. The reason is that for this request arrival pattern, the peak rate is 140 per minute ($\rho_{system} = 0.8$), and at this system utilization, our proposed strategy is the best one among all the algorithms. The results here are consistent with the results shown in Fig. 5.

Furthermore, we carry out simulations to examine the performance of the algorithms under the request patterns of sudden rising with spikes and show the simulation results in Fig. 9. We can observe that during the spikes, the AWT of each algorithm also rapidly rises. However, for our proposed algorithm, in the worst cases, the AWT can be kept within a low level, which is not more than 40 seconds. In these simulations, *Polling-NoRep* and *Polling-1Rep* perform the worst. In Fig. 9b, we can point out that in most cases, the rate of DoS of our proposed algorithm is the best among all the algorithms.

From these experiments, we can conclude that one more replication of the object in the system can significantly improve the system performance. More replications can achieve smaller AWT and lower rate of DoS. However, with the consideration of disk capacities, we cannot add replications unlimited into the system. Although *Polling-2Rep* performs slightly better than our proposed algorithm in some peak situations, our proposed algorithm is the most space efficient, with very small extra disk usage, and the performance is excellent under varying request arrival patterns. Hence, we can draw the conclusion that our proposed algorithm is efficient, flexible, and scalable.

7 Conclusions and Future Work

In this paper, a novel distributed load balancing strategy has been proposed for large-scale multimedia storage systems, which consist of a large number of OSSs. In traditional distributed file systems, request balancing is handled by the centralized MDSs, and the MDSs are the potential bottlenecks in achieving higher performance. In our proposed strategy, we distribute such workloads of load balancing among all the OSSs in the system and let the MDSs focus on searching for the locations of the requested objects only. We model OSS as an $M/G/m$ system, based on which we formulate our objective function for the AWT of the requests that arrive at the system. Each OSS can determine the replications of the objects and balance the requests for these objects among their replications independently according to the near-optimal solution of the objective function.

For each object, it has a primary OSS in the system, which takes charge of the object. The primary server uses a heuristic method for finding other OSSs to store a replication of the object and construct the set of servers for the object. The number of replications of each object is based on the request rate for the object. For example, some "hot" objects can have several replications to meet the client requests, whereas some "cold" object may have no replication, and they just have one copy for backup in the system. To the best of our literature survey, this work is the first of its kind, in which the request-based replication policy is proposed. This replication policy can use the disk storage capacity more efficiently and can save more disk space in the system.

We proposed a novel distributed load balancing algorithm to distribute the requests for an object among its set of servers and applied the algorithm in dynamic situations. We constructed several request arrival patterns and analyzed the client waiting behavior. In our simulation experiments, we compared our proposed algorithm with polling-based algorithms. Our simulation experiments conclusively demonstrated that the proposed algorithm clearly offers a significant advantage in minimizing the AWT, keeping the rate of DoS in a low level, and, at the same time, balancing the server loading across the system. Our study demonstrated that our proposed strategy with not more than one replication, on the average, performs better than the polling-based algorithm with two replications in most cases. When the system load is high, the performance of our proposed algorithm becomes worse. However, the AWT and rate of DoS are still kept within an acceptable level.

Some open-ended issues are apparent from this work. First, it would be normal to analyze and implement the dynamic prefetching [26], migration [27], and deletion of the objects that exist in the system. It would also be interesting to migrate some "cold" objects from the servers with high bandwidth but relatively low disk capacities to other servers with much more disk space. Furthermore, system failover and recovery are also important parts for LMSS. In this work, most of the objects have only one copy in the system, and the failure of even one server can lead to many objects out of service. Hence, fast failover and recovery can significantly improve system robustness. Last, we should consider more about the system security and the QoS of the networks, to quote a few. Some of these studies are currently underway.

APPENDIX

LIST OF NOTATIONS

See Table 3.

TABLE 3
List of Notations

bw_i	the bandwidth needed to play object i
bw_o	the expected bandwidth need to play an object
d_P	the first derivative length of path P
i	index of objects
j	index of servers
n_j	the expected number of channels in server j
o_i	object i
p_i	the probability that a request is for object i
r	the index of iterations
s_i	the size of object i in byte
t_i	the playback time of object i
x_P	the request rate on path P
λ	the total request arrival rate in the system
λ_i	the request arrival rate for object i
λ_j	the request arrival rate in server j
λ_{ij}	the rate of the requests for object i arriving at server j
μ_i	the service rate of object i
μ_o	the expected service rate of objects
σ	the standard deviation of playback time
ρ_j	the system utilization of server j
ρ_{system}	the utilization of the entire system
BW_j	the bandwidth of server j
H_P	the sum of second derivative lengths of P and $\bar{P}$
M	the number of objects in the system
N	the number of servers in the system
O	the set of objects in the system
O_j	the set of objects in server j
$\bar{P}$	the minimum first derivative length path
P_{ik}	the k-th path of the requests for object i
P_k	the path k
$\mathcal{P}_i$	the set of routing paths of the requests for object i
S_j	server j
$T_{gain}^{(r)}$	the time gain in r-th iteration
V_i	the set of servers with a replication of object i
$\overline{X}$	the expected playback time of an object
$\overline{X^2}$	the second moment of playback time

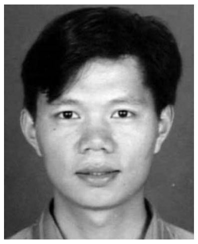
ACKNOWLEDGMENTS

The authors thank the anonymous reviewers whose comments significantly improved this work. They would also like to acknowledge the support by the Agency for Science, Technology and Research (A*STAR) SERC, Singapore, under Grant R-263-000-345-305.

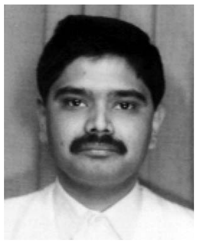
REFERENCES

[1] B. Veeravalli and G. Barlas, *Distributed Multimedia Retrieval Strategies for Large Scale Networked Systems.* Springer-Verlag, 2005.
[2] C.Y. Choi and M. Hamdi, "A Scalable Video-on-Demand System Using Multi-Batch Buffering Techniques," *IEEE Trans. Broadcasting,* vol. 49, no. 2, pp. 178-191, June 2003.
[3] W.K. Park, C.S. Choi, D.Y. Kim, Y.K. Jeong, and K.R. Park, "IPTV-Aware Multi-Service Home Gateway Based on FTTH Access Network," *Proc. Ninth Int'l Symp. Consumer Electronics (ISCE '05),* pp. 285-290, June 2005.
[4] U. Jennehag and T. Zhang, "Increasing Bandwidth Utilization in Next Generation IPTV Networks," *Proc. Int'l Conf. Image Processing (ICIP '04),* Oct. 2004.
[5] Y.J. Oyang, C.H. Wen, C.Y. Cheng, M.H. Lee, and J.T. Li, "A Multimedia Storage System for On-Demand Playback," *IEEE Trans. Consumer Electronics,* vol. 41, no. 1, pp. 53-64, Feb. 1995.
[6] B. Ozden, R. Rastogi, and A. Silberschatz, "Buffer Replacement Algorithms for Multimedia Storage Systems," *Proc. Third IEEE Int'l Conf. Multimedia Computing and Systems (ICMCS '96),* pp. 172-189, June 1996.
[7] K.C. Almeroth, "Adaptive Workload-Dependent Scheduling for Large-Scale Content Delivery Systems," *IEEE Trans. Circuits and Systems for Video Technology,* vol. 11, no. 3, Mar. 2001.
[8] P. Braam, *Luster,* http://www.lustre.org, 2005.

[9] S. Ghemawat, H. Gobioff, and S.T. Leung, "The Google File System," *Proc. 19th ACM Symp. Operating Systems Principles (SOSP '03),* Oct. 2003.
[10] B. Nq, R.W.H. Lau, A. Si, and F.W.B. Li, "Multiserver Support for Large-Scale Distributed Virtual Environment," *IEEE Trans. Multimedia,* vol. 7, no. 6, Dec. 2005.
[11] Z. Zeng and B. Veeravalli, "Design and Analysis of a Non-Preemptive Decentralized Load Balancing Algorithm for Multi-Class Jobs in Distributed Networks," *Computer Comm.,* vol. 27, pp. 679-694, 2004.
[12] A.E. Kostin, I. Aybay, and G. Oz, "A Randomized Contention-Based Load-Balancing Protocol for a Distributed Multiserver Queuing System," *IEEE Trans. Parallel and Distributed Systems,* vol. 11, no. 12, Dec. 2000.
[13] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam, "An Integrated Approach to Parallel Scheduling Using Gang-Scheduling, Backfilling, and Migration," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 3, pp. 236-247, Mar. 2003.
[14] E. Choi, "Performance Test and Analysis for an Adaptive Load Balancing Mechanism on Distributed Server Cluster Systems," *Future Generation Computer Systems,* vol. 20, pp. 237-247, 2004.
[15] *Tiger Shark File System,* IBM Almaden, http://www.research.ibm.com/webvideo/ shark.html., 2007.
[16] D.N. Serpanos, L. Georgiadis, and T. Bouloutа, "MMpacking: A Load and Storage Balancing Algorithm for Distributed Multimedia Servers," *IEEE Trans. Circuits and Systems for Video Technology,* vol. 8, no. 1, pp. 13-17, Feb. 1998.
[17] D. Bertsekas and R. Gallager, *Data Networks.* Prentice Hall, 1992.
[18] G. Zipf, *Human Behavior and the Principle of Least Effort.* Addison-Wesley, 1949.
[19] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira, "Characterizing Reference Locality in the WWW," *Proc. Fourth IEEE Int'l Conf. Parallel and Distributed Information Systems (PDIS '96),* Dec. 1996.
[20] K. Shen, T. Yang, and L. Chu, "Cluster Support and Replication Management for Scalable Network Services," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 11, Nov. 2003.
[21] A. Bestavros, C.R. Cunha, and M.E. Crovella, "Characteristics of WWW Client-Based Traces," technical report, Boston Univ., July 1995.
[22] N. Nishikawa, T. Hosokawa, Y. Mori, K. Yoshidab, and H. Tsujia, "Memory-Based Architecture with Distributed WWW Caching Proxy," *Proc. Seventh Int'l Conf World Wide Web (WWW '96),* Apr. 1998.
[23] L. Jie and H. Kameda, "Load Balancing Problems for Multiclass Jobs in Distributed/Parallel Computer Systems," *IEEE Trans. Computers,* vol. 47, no. 3, pp. 322-332, Mar. 1998.
[24] Z. Zeng and B. Veeravalli, "Design and Performance Evaluation of Queue-and-Rate-Adjustment Dynamic Load Balancing Policies for Distributed Networks," *IEEE Trans. Computers,* vol. 55, no. 11, pp. 1410-1422, Nov. 2006.
[25] J.Y.B. Lee, "Channel Folding: An Algorithm to Improve Efficiency of Multicast Video-on-Demand Systems," *IEEE Trans. Multimedia,* vol. 7, no. 2, pp. 366-378, Apr. 2005.
[26] B. Wu and A.D. Kshemkalyani, "Object-Optimal Algorithms for Long-Term Web Prefetching," *IEEE Trans. Computers,* vol. 55, no. 1, pp. 2-17, Jan. 2006.
[27] L.F. Zeng, D. Feng, F. Wang, and K. Zhou, "Object Replication and Migration Policy Based on OSS," *Proc. Fourth Int'l Conf. Machine Learning and Cybernetics (ICMLC '05),* Aug. 2005.
[28] J.J. Kinney, *Probability: An Introduction with Statistical Applications.* John Wiley & Sons, 1997.
[29] M. Avriel, *Nonlinear Programming Analysis and Methods.* Prentice Hall, 1997.
[30] D.P. Bertsekas, *Nonlinear Programming.* Athena Scientific, 1995.
[31] V.V. Mitin, D.A. Romanov, and M.P. Polis, *Modern Advanced Mathematics for Engineers.* John Wiley & Sons, 2001.
[32] S.D. Yao, C. Shahabi, and P.A. Larson, "Hash-Based Labeling Techniques for Storage Scaling," *The VLDB J.—The Int'l J. Very Large Data Bases,* vol. 14, pp. 222-237, Apr. 2005.
[33] A.T. Stephanos and D. Spinellis, "A Survey of Peer-to-Peer Content Distribution Technologies," *ACM Computing Surveys,* vol. 36, no. 4, pp. 335-371, Dec. 2004.
[34] Y. Lu, A. Zhang, H.F. He, and Z.Q. Deng, "Stochastic Fluid Model for P2P Content Distribution Networks," *Proc. Seventh Int'l Symp. Autonomous Decentralized Systems (ISADS '05),* pp. 707-712, Apr. 2005.
[35] J.Y.B. Lee, "On a Unified Architecture for Video-on-Demand Services," *IEEE Trans. Multimedia,* vol. 4, pp. 38-47, Mar. 2002.
[36] P. Hokstad, "Approximations for the M/G/m Queue," *Operations Research,* vol. 26, no. 3, pp. 510-523, May/June 1978.
[37] Z.X. Zhao, S.S. Panwar, and D. Towsley, "Queuing Performance with Impatient Customers," *Proc. IEEE INFOCOM '91,* pp. 400-409, Apr. 1991.



Zeng Zeng received the BS and MS degrees in automatic control from Huazhong University of Science and Technology, Wuhan, China, in 1997 and 2000, respectively, and the PhD degree in electrical and computer engineering from the National University of Singapore in 2005. He is currently a research fellow at the National University of Singapore. His research interests include distributed/grid computing systems, multimedia storage systems, wireless sensor networks, and controller area networks.



Bharadwaj Veeravalli received the BSc degree in physics from Madurai-Kamaraj University, India, in 1987, and the master's degree in electrical and communications engineering and the PhD degree from the Indian Institute of Science, Bangalore, India, in 1991 and 1994, respectively. He was a postdoctoral researcher in the Department of Computer Science, Concordia University, Montreal, in 1996. He is currently with the Department of Electrical and Computer Engineering, National University of Singapore, as a tenured associate professor. He is also with the editorial board of the *IEEE Transactions on Computers*, the *IEEE Transactions on Systems, Man, and Cybernetics*, and the *International Journal of Computers and Applications* as an associate editor. His main research interests include multiprocessor systems, cluster/grid computing, scheduling in parallel and distributed systems, bioinformatics and computational biology, and multimedia computing. He is one of the earliest researchers in divisible load theory (DLT). He is a senior member of the IEEE and the IEEE Computer Society.