Load Balancing in Bounded-Latency Content Distribution[*]

Chengdu Huang[†] Gang Zhou[‡] Tarek F. Abdelzaher[†] Sang Hyuk Son[‡] John A. Stankovic[‡]

† Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{chuang30,zaher}@cs.uiuc.edu

‡ Department of Computer Science
University of Virginia
Charlottesville, VA 22904
{gz5d,son,stankovic}@cs.virginia.edu

Abstract

In this paper we present a balanced data replication scheme that provides real-time latency bounds on content retrieval in content distribution networks. Many network applications have ever-increasing requirements on latency sensitive data services. Data replication services have been widely used as an important performance enhancement mechanism to reduce data access latency and throughput. We investigate the problem of provisioning an underlying balanced *data replication service to provide a* global latency bound *on data retrieval in content distribution networks. The solution involves constructing an overlay network based on the given latency bound, and a mechanism to assign content objects to the network nodes so that the workload of all the network nodes is balanced. Our evaluation results drawn from detailed simluations show the efficacy of our load-balancing scheme in meeting the latency bound requirements with high confidence under heavy load.*

1 Introduction

Many large-scale distributed systems, such as peer-to-peer file sharing systems, content distribution networks (CDNs), and wireless sensor networks, utilize data replication to enhance the overall system performance. For example, peer-to-peer applications and many distributed file systems [19] greedily replicate large files so that the average download time of clients is reduced. In CDNs, popular content is replicated on-demand or pre-populated to CDN edge servers to support a higher volume of client requests [1]. In wireless sensor networks, when some event of interest is detected in a certain area and triggers queries from different locations in the network, the data associated with the event may be replicated to prevent network congestion [6]. In these systems, the locations of content replicas have a significant impact on performance. The problem of how and where to replicate content in large-scale network systems has been studied in many different scenarios [17, 12, 21].

While data replication, in general, can improve overall system throughput and service availability, many applications in large-scale distributed systems have stringent timing requirements as well. Namely, many applications require that requests to retrieve content be served within a global latency bound, regardless of where the requests originated. This kind of soft real-time requirements is vital for many import network applications. For example, an online stock trading service provider would want its content to be accessible within a short amount of time for all its worldwide customers. Online gaming, for another example, also requires the state information of the game to be accessible to the gamers very quickly. Whether a (short) global latency bound on content access can be provided has a substantial impact on the quality of many online services.

The problem we investigate in this work is how to achieve load-balancing when there are many content objects with soft real-time latency bounds in a large-scale content distribution network (CDN). When a large number of content objects are to be replicated on the CDN servers, each individual server will host only a subset of the content objects. The fact that different content objects can have very different popularities and servers have different network connectivities can cause servers to have an imbalanced workload. Load-balancing among the network nodes is of primary importance to improve the utilization of the system's resources and the system's ability to provide latency bounds. Different from many existing load-balancing techniques in other distributed systems [4, 14, 10], our load-balancing mechanism is coupled with the process of creating replicas for content objects. This is because our load-balancing mechanisms are subject to the constraint of content objects' latency bound requirements. Ideally, we want to place replicas of the content objects strategically, so that (1) latency bound requirements of the content objects are met; (2) workload of all the network nodes is balanced; and (3) the replication cost is low. Moreover, for a large-scale system, the algorithm has to be decen-

*The work reported in this paper was supported in part by the National Science Foundation under grants CCR-0093144, ANI-0105873, EHS-0208769, and MURI N00014-01-1-0576.

tralized. In this paper, we assume that the content objects do not have stingent consistency requirement and hence their replicas do not need to be strictly synchronized with each other. In the rest of the paper, we describe such a balanced data replication scheme that satisfies these requirements.

The remainder of the paper is organized as follows. Section 2 sketches the architecture and technical challenges of our balanced data replication scheme. Section 3 gives the detailed design of the load-balancing replication algorithm. An extensive performance evaluation of our system is presented in Section 4. Section 5 presents a survey of related work. The paper concludes with Section 6.

2 Overview

2.1 Latency Bound

In our previous work [12], we presented a content distribution service that can provide global latency bounds on content retrievals. For the sake of completeness, we here briefly describe the architecture of the CDN we built.

Consider a CDN that consists of a large number of CDN servers, or servers for short. Content objects can be replicated on these servers. Content objects are discretized into a small number of *content classes* by their latency bounds. Content objects in the same content class share the same latency bound. Latency bounds of the content objects are known to all the servers.

In our CDN model, servers of the CDN provider are deployed on the edge of the Internet backbone and may be co-located with ISPs' Internet Gateway Routers (IGRs). This is known as the *colocation* model in which the requests of clients can be redirected to the nearest CDN server [1, 4]. Hence, clients are presumably very close to their nearest CDN servers. When a CDN server receives a request, if the requested content object is available at the server's local storage, a reply is sent back to the client directly. Otherwise, the request will be redirected to some replica of the requested content object (or to the origin server). When a content object is replicated on a CDN server, we say the server *hosts* the content object.

To provide a latency bound on content retrieval, the CDN provider needs to replicate the content objects on its servers so that when a CDN server receives a client request, it can either find the requested content object in its local storage, or can find a replica of the content object from a remote server that is within the latency bound. To achieve this goal, we first construct an overlay network based on the latency bound. The construction needs to guarantee one important property of the overlay network: neighbors on the overlay can reach each other within the given latency bound.

With an overlay network in which neighbors can reach each other within the global latency bound, to provide a latency bound on content retrieval is to find a set of replicas for every content object so that every CDN server can find the content from its closed neighborhood, which is defined as its neighbors plus itself, on the overlay network.

Note that trivially replicating *all* content objects to *every* server can make the network latencies of serving client requests very short. This naive scheme, however, is not practical because it incurs a prohibitively high replication cost. Hence, our data replication service needs to avoid creating unnecessary replicas when replicating content to meet the latency bound requirements. Specifically, neighbors on the overlay network should largely host different content objects.

2.2 Load-balancing

Workload of a CDN server is simply the total request rate it experiences. As in many other distributed systems, load-balancing is very important for a CDN system to effectively utilize available resources. An imbalanced workload will create hotspots which can cause many latency bound violations. Load-balancing has been studied in many different scenarios such as distributed file systems [19], peer-to-peer systems [10, 18], and CDNs. Existing load-balancing techniques on CDNs are mainly request redirection schemes [4, 22]In our CDN system, content objects are replicated to meet the latency bound requirements and are available on a small subset of the CDN servers only. An effective replication strategy is of ultimate importance to the system's ability to achieve load-balancing. Request redirection mechanisms deal with how to efficiently redirect requests to given replicas so that servers' workload are balanced. How to strategically create replicas for content objects to facilitate load-balancing, however, is out of the scope of request redirection. Our load-balancing mechanism achieves balanced workload along with the replication process. It can also benefit from combining with certain request redirection schemes. Therefore, we believe that request redirection is both orthogonal and complimentary to the load-balancing technique we present in this paper.

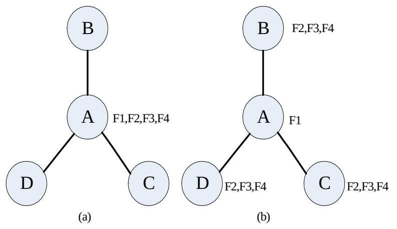


Figure 1. Imbalanced vs. Balanced Replication

The workload of a server is determined by the number of content objects that it hosts, and their popularity. Another factor is the network degree (on the overlay network) of the server. To understand this, consider the example illustrated in Figure 1. Node A has the highest degree among the 4 nodes. Both replication schemes in Figure 1(a) and (b) satisfies the latency bound requirement. However, the load-balance properties of the two schemes differ a lot. Assuming all the content objects share the same popularity r at all the nodes, in Figure 1(a), node A serves *all* the requests while other nodes have no workload at all. In contrast, replication in Figure 1(b) enjoys a much better load-balancing: workload of all the nodes are $4r$. Note, however, the replication i Figure 1(a) enjoys the advantage of having fewer replicas. Actually, there is a trade-off between the number of replicas and the load-balancing property. In Section 4, we investigate this trade-off under different system parameters.

It is worth noting that the total workload in the system is independent of the replication scheme. It is solely determined by request rates of all the content objects. Different replica placements merely make the total workload be distributed among the servers differently.

In this work, we mainly focus on controlling workload of the CDN servers, as opposed to network bandwidth. We argue that server workload is the major concern because the CDN servers are a dedicated resource of the CDN provider. The scheme of distributing the overall workload to different CDN servers largely determines the response time of the CDN servers. Network bandwidth, however, is a shared resource. Our content service is just one of the many services that are running on the internet and sharing the underlying network infrastructure. Hence, the bandwidth consumption of our service is unlikely to noticeably affect the actual network latencies. In addition, our previous measurement study [12] showed that internet latencies are fairly stable for our service. Given the fact that the overlay network is created based on the latency bound and the satisfactory network latency stability, if we can find a balanced data replication scheme on the overlay network to ensure the response time of the servers are bounded, then we can achieve the latency bound guarantee. In the next section, we present the detailed design of such a scheme.

3 System Design

In this section, we present our balanced data replication algorithm. Servers make independent decisions on which content objects to host based on local information about their workload and popularity of the content objects. Servers only need very limited knowledge about their neighbors, making the scheme highly scalable. The algorithm consists of three major phases. In the token calculation phase, servers first exchange their IDs with their neighbors (i.e., the servers that are reachable within the latency bound). Without communication, each server can locally generate a conflict-free token winning sequence using the token calculation algorithm to be described. After that, servers consolidate the token winning sequences of their neighbors and come up with a sequence that contains all the tokens within its closed neighborhood. This sequence will be used to decide on the order in which servers make content allocation decisions. In the content allocation phase, following the consolidated token winning sequence, servers use the content object allocation algorithm to choose appropriate objects to host. We also present a post-adjustment phase to deal with dynamics of network topology and changes in popularity of content objects. In what follows, we describe the three phases in detail, and discuss issues related to multiple content classes.

3.1 Token Calculation

In this phase, each server first collects ID information of all of its neighbors. Note that the neighborhood of a server is the collection of CDN servers that are reachable by the server within a given latency bound. After servers obtain the IDs of their neighbors, each server will create a set of pseudo random number generators (RNGs). For each of its neighbors, a server associates a RNG seeded with the neighbor's ID. Servers also create RNGs associated with themselves, and seed them with their own IDs. Note that *all* the RNGs of all servers should use the same pseudo random number generation algorithm. This scheme is inspired by the NCR algorithm used in TDMA schemes for wireless communications [3, 25].

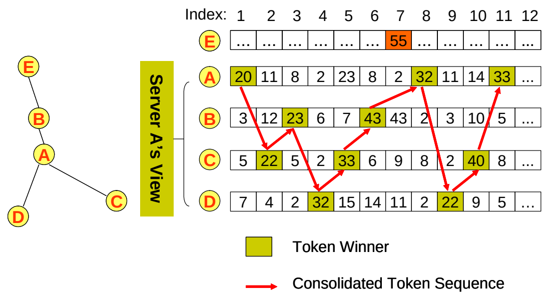

Index: 1 2 3 4 5 6 7 8 9 10 11 12
E ... ... 55 ... ...
A 20 11 8 2 23 8 2 32 11 14 33 ...
B 3 12 23 6 7 43 43 2 3 10 5 ...
C 5 22 5 2 33 6 9 8 2 40 8 ...
D 7 4 2 32 15 14 11 2 22 9 5 ...
Server A's View
E
B
A
D
C
Token Winner
Consolidated Token Sequence

Figure 2. Token Allocation Example

For the ease of explanation, we take server A in Figure 2 as an example. As Figure 2 depicts, servers B, C and D are neighbors of server A, while server E is B's neighbor but not A's. Node A first generates a random number sequence using the RNG seeded with its own ID: $\{20, 11, 8, 2, 23, 8, 2, ...\}$. Similarly, it generates random number sequences for servers B, C and D by RNGs associated with IDs of B, C, and D, respectively.

Having the random number sequences of all four servers in its neighborhood constructed, server A scans the four sequences in parallel in ascending order of the index. For an index i, we say a server wins the *token* if its ith random number is greater than all the ith random numbers of its neighbors, using ID numbers for tie-breaking. Clearly, tokens are associated with index numbers. Token winners will make content allocation decisions as will be discussed in Section 3.2. From Figure 2, we can tell that server A wins tokens at index 1, 8 and 11. When server A calculates the tokens locally, servers B, C and D do the same token calculation in parallel independently. All token winners are marked as shaded blocks in Figure 2. The important feature of this process is that no two servers would win the same token if they are neighbors. This is true because all the RNGs use the same pseudo random number algorithm. When a server sees its ith random number is greater than those of its neighbors, the neighbors should have the same view.

Having received the token winning sequences of servers B, C, and D, server A can combine all the sequences and get a consolidated sequence, illustrated by the arrows in Figure 2. This sequence is used in the content object allocation phase to be discussed in the next section.

It is worth pointing out that in index 7, no server in A's neighborhood wins the token. This is because B's 7th random number is greater than those of A, C, and D, but smaller than that of E. This does cause some problem in TDMA design [3] because it introduces "holes" in the sequence of token winners. Our scheme is not impacted much by these holes, because we only care about the relative decision sequence for content object allocation. It is true, though, that we need to generate more random numbers to get the same number of tokens because of the holes.

3.2 Content Object Allocation

The consolidated token sequence (illustrated as the arrows in Figure 2) of a server is basically a sequence of tokens of its neighbors. If a server finds that the token with the smallest index belongs to itself, it volunteers to host a content object, using the algorithm to be presented later in this section. This decision or a dummy decision (to be discussed later in this section) is broadcast to its neighbors. The token is then removed from the server's consolidated token sequence. If a server finds that some neighbors hold tokens with smaller indices, it simply waits for the content object allocation decisions from those neighbors, with a time out to handle token loss in rare cases. Every time it hears a content object allocation decision from a neighbor, the corresponding token is removed from its consolidated token sequence and the content object selected by the neighbor is marked as "unavailable". After all those tokens with smaller indices than its own token are removed, it can make its own content object allocation decision again and broadcast it. For example, in Figure 2 after server A makes the first content object allocation decision, it will need to wait for decisions from C, B, D, C, B before it makes the next content objection allocation decision. Note that in the token calculation phase, the possibility of winning tokens for a server is inversely proportional to the number of neighbors of the server. Statistically, the more neighbors a server has, the fewer chances it will get to host content objects. This is ideal in our situation because when a server hosts an object, its neighbors will forward their requests for the object to the server. The workload that will be brought to the server by hosting an object is therefore proportional to the number of neighbors of the server. Hence, making servers that have more neighbors host fewer objects is helpful in making servers' workload balanced.

To implement the mechanism described above, each server maintains an *available objects list* (AVL). This list maintains all content objects that have *not* been hosted by any server in its closed neighborhood, as well as their popularity. Servers only choose objects from their AVL to host. When new content objects are introduced to the CDN system, they are inserted into the servers' AVLs. The algorithm for content object allocation will then be triggered to create replicas for the new objects. Note that the popularity of objects maintained by a server are those popularity witnessed by the server locally, not global popularity. The list is sorted in the descending order of popularity of the objects. When a server hears that some neighbor has volunteered to host a content object, the content object will be removed from its AVL. Similarly, when the server decides to host a content object, the object will be removed from the AVL as well. This way, if two servers are neighbors, they would not both be replicas of the same content object, assuming no packet loss. Ideally, for each content object, there will be one and only one replica in each local neighborhood.

Note that although the tokens are associated with indices, the content allocation phase is not a synchronous process. All the decision making and broadcasting can happen in parallel in different neighborhoods throughput the the whole network. The decisions a server is waiting for do not have to arrive in order. For example, in Figure 2, D's content allocation decision can arrive at A before those of B and C. In addition, when one server broadcasts its decision to its neighbors, it may trigger multiple servers to make decisions in parallel, which speeds up the whole process. In this scheme, there is no network wide information propagation, making it highly scalable.

We assume the popularity of content objects can be predicted based on history information. Measurement studies show that relative popularity of content objects is pretty stable over time [17, 15]. It is true, however, that popularity of content objects can be subject to abrupt changes in some situations (e.g., flash events). We do have a mechanism (to be discussed in Section 3.4) to accommodate popularity changes. Having relatively accurate history information about popularity of content objects can only be beneficial.

The response time of a server obviously depends on its workload. It is well known that server response time does not linearly scale with the offered load. In fact, in many cases there exists a knee in the load-response curve. Imposing a workload that exceeds the knee will cause a substantial increase on the response time, and hence cause a lot of latency bound violations. Hence, for a given bound on server processing time t and a target latency miss ratio p, we define the *capacity threshold* of a server as the workload at which $1 - p$ of the requests can be served within t. Imposing a workload that exceeds the capacity threshold can cause a substantial increase of latency bound violations.

From the definition of the capacity threshold, we need to keep the workload of all the servers below their capacity thresholds if possible. Specifically, when a server picks a content object to host, it needs to make sure this action will not bring its workload beyond its capacity threshold. Hence, a content object allocation decision actually is twofold: the server needs to decide (1) which content object to host; and (2) which neighbors should be allowed to forward requests for this object to the server. The goal is to minimize the servers' chance of transcending their capacity thresholds.

We use a greedy heuristic in making the content allocation decisions. When it is the turn for a server, say $\mathbb{A}$, to pick a content object to host, it greedily chooses the object $\mathcal{O}$ that has the highest popularity among the objects in its AVL that does not make workload of $\mathbb{A}$ exceed its capacity threshold. If there exists such an object, the server then collects the popularity of this object on its neighbors. Note that, if all its neighbors forward the requests for $\mathcal{O}$ to $\mathbb{A}$, it is possible that the server will be overloaded. Hence, the server needs to be selective in notifying the neighbors of its object allocation decision. This notification process proceeds from the neighbor with the lowest network degree (number of neighbors on the overlay) to the highest. If the available capacity of the server is sufficient to serve the requests for the picked object from a neighbor, a notification reporting that the object is hosted is sent to the neighbor. Otherwise, a *dummy decision* is sent to the neighbor. A dummy decision is a placeholder just to let the neighbor know the server has already made a content allocation decision so that the neighbor can proceed if it is waiting for the server's decision. After every notification is sent, the server updates its available capacity. In the case that the available capacity of $\mathbb{A}$ is not sufficient for any available object, the server will choose the to host the least popular object with a probability that is proportional to its degree.

Note that for the sake of simplicity, we assume that all servers have homogeneous capacities. Our algorithm can be trivially extended to handle heterogeneity of server capacities without affecting the validity of the discussions in this paper. For example, when servers have different capacities, instead of using the absolute workload as the metric of deciding load-balancing, we can use the ratio of a server's workload over its total capacity.

3.3 Multiple Content Classes

In the CDN system, content objects are categorized into different content classes by their latency bound requirements. Content objects of the same content class share the same latency bound. Since construction of the overlay network depends on the latency bounds, different content classes actually have different overlay networks. The token calculation and content object allocation algorithms presented above assume that all content objects have the same latency bound. These algorithms can be easily extended to handle multiple content classes. When there are content objects that belong to multiple content classes, the system can sequentially run the algorithms for each content class. Since the number of content classes is usually small, this is not an issue. The sequence of running the algorithms on the content classes, however, may have an impact on the performance of load-balancing. Conceivably, the latency bounds of the content classes can affect the load-balancing property. The popularity of content objects in each content class may also be a factor that needs to be considered. In Section 4, we study the implications of different strategies of handling multiple content classes.

3.4 Post-Adjustment

After the process described above terminates, every server will be able to find at least one replica for any of the content objects within its one-hop closed neighborhood. Moreover, the workload of all the servers should be below their capacity thresholds due to the way replicas of the content objects are created.

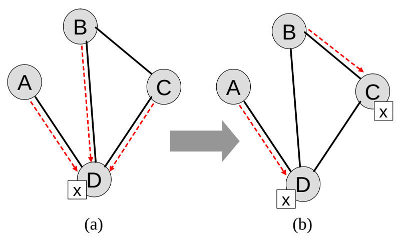


Figure 3. Post-Adjustment

However, dynamics of workload can disrupt the load-balancing. Consider the example depicted in Figure 3(a). Server D is a replica of object x. Servers A, B, and C, can forward requests asking for x to D because they are neighbors of D. For the same reason, they would not host x. Now, if x for some reason (e.g., a flash event) suddenly becomes extraordinarily popular, it can cause server D to suffer a much higher workload than the other servers, and potentially makes D overloaded.

To deal with this situation, we introduce a post-adjustment phase to readjust the content object allocations to achieve better load-balancing after the regular replica assignment process. The post-adjustment process still proceeds in a distributed fashion, meaning servers make decisions independently based on their local knowledge about their neighbors. The basic idea of the post-adjustment algorithm is that when a server has a workload significantly higher than its capacity threshold, it can "push" some of the hot content objects it hosts to some of its relatively underloaded neighbors, as illustrated in Figure 3(b). Server D can push x to server C and hence create another replica of x. After that, B can forward requests asking for x to C. This way, the total workload brought by object x is shared by C and D. The cost is that an extra replica of x is created.

Formally, when a node, say node A, finds that its workload $Load(A)$ satisfies

$$\begin{aligned}&\exists s \in Neighbors(A), Load(A) > Load(s) \text{ and} \\ &\qquad Load(A) > F \times CapacityThreshold(A) \qquad (1)\end{aligned}$$

it picks the object with the highest popularity among all the objects it hosts, say $\mathcal{O}$, and asks its least loaded neighbor to be a replica of $\mathcal{O}$ too. The neighbor then broadcasts an announcement to notify its neighbors that a new replica of $\mathcal{O}$ has been created.

In order to choose the least loaded neighbor, servers periodically exchange workload information with their neighbors. This information can also be piggybacked in the content object allocation decisions. The condition specified by Eq. 1 indicates that the server is not the least loaded in its neighborhood, and its workload is currently F times higher than its capacity threshold, where F is a constant called *tolerance factor*. Obviously, F should be greater than 1. The tolerance factor is an indicator of how much overload is tolerated by the post-adjustment algorithm. Generally, the smaller the tolerance factor, the better load balancing can be achieved, but at a higher cost. Servers periodically check their workload against the condition specified in Eq. 1, and make adjustment to content object allocations independently. Workload of those heavily loaded servers gradually gets shed off and shared by other underloaded servers.

Note that, it is possible that after a heavily loaded server pushes out a hot content object to some of its neighbors, the content object may be pushed back to the server by some neighbor. In that case, there is a possibility that this content object will be pushed back and forth indefinitely. To avoid this situation, every server needs to maintain a list of "hot-potatoes". If a server pushes out a content object in the post-adjustment phase, and later on the content object gets pushed back to the server, the content object is added to the hot-potatoes list. Content objects in the hot-potatoes list will *not* be picked to be pushed out by the server again within a relatively long period.

4 Evaluation

4.1 Experiment Setup

We implemented our balanced data replication service by instrumenting a Squid Proxy Cache [20] and tested it on PlanetLab [16]. Because PlanetLab is a shared platform and its servers are constantly in heavy loaded state, it is very hard to run repeatable and controllable experiments on it. In our performance evaluation study, we used a "hybrid" experiment methodology. We first selected certain servers on PlanetLab and deployed our measurement daemons to these servers. Distance maps of the servers can be constructed from the measurement results. We implemented the data replication algorithm described in Section 3 in a discrete event simulator. The distance maps constructed from PlanetLab measurement data were then fed to the simulator. We configured different latency bounds so that different overlay network topologies can be generated. We used two different experiment scenarios. One has 30 servers, all from sites in North America. The other has 75 servers from both North America and Europe. Figure 4 shows the experiment scenario with 30 servers. Unless otherwise specified, the experimental data presented in this section are the average of the results running on both scenarios. Before embarking on presenting detailed evaluation results, we first validate the simulation results of one of our experiments against the results we obtained from experimenting on PlanetLab. Figure 5 gives the workload of all the 30 servers when running the same experiment on PlanetLab and in simulation. As we can tell from the figure, the results match fairly well. This gives us confidence in the fidelity of our simulation results.

We evaluate our replication algorithm using the following two primary metrics:

- *Latency bound miss ratio:* The latency bound miss ratio is the fraction of requests that are served with latencies longer than the latency bounds associated with the requested objects. When there are multiple content classes, the latency bound miss ratios are the aggregated ratios of all the classes.

- *Average number of replicas:* As content objects are replicated in the network, the number of replicas is

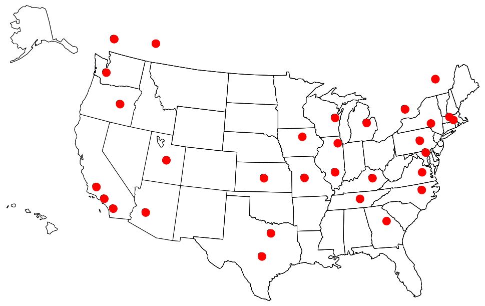


Figure 4. Experiment setup on PlanetLab

of interest because it is directly related to the replication cost. Fewer replicas imply lower replication cost. The average number of replicas is defined as the total number of replicas of all the content objects over the number of content objects.

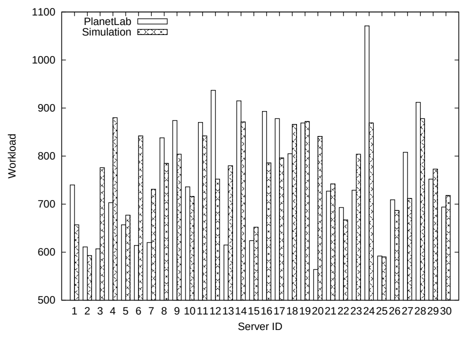


Figure 5. Validation of Simulation Results

The end-to-end latency of serving a request consists of two parts. The first part is the network latency between the server where the request is originated and the replica the request is forwarded to. This value is immediately available from our measurement data. The second part is the server end processing delay. This time is related to the workload of the CDN servers. Since we are discouraged from running overloading experiments on PlanetLab, we measured the server end latencies from using our local boxes. Basically, we setup a server in our LAN that serves client requests generated by an array of clients running SURGE [5]. The request intensities of the clients were configured according to our simulation results, which contain the information of workload experienced by each CDN server. The server processing delay can then be measured on the clients. The end-to-end latency of serving a request is the sum of the two parts. When the *end-to-end latency* of a request exceeds its latency bound, we count it as a miss.

Our replication algorithm requires every server to know its capacity threshold. As discussed in Section 3.2, the capacity threshold of a server depends on a latency bound t and a target miss ratio p. In our experiments, we choose $t = 100ms$ and $p = 0.01$. In other words, we make the capacity threshold of a server the workload that the server can serve 99% of the requests within 100ms.

Different content objects can have very different popularity. Measurement studies on traffic of Web caches [7], Web servers [2, 15], and peer-to-peer file sharing systems [11] discover that popularity of content objects in these applications follow Zipf-like distributions or a variation of Zipf distribution. To understand the system performance under realistic conditions, we use a Zipf-like distribution as the popularity model. The value of the α parameter of Zipf is 1.2 in our experiments. Note that using Zipf distribution as the model of content objects' popularity brings tremendous difference of popularity among content objects.

In our experiments, we used a parameter called *load factor*, defined as the sum of request rates of all the content objects on all the servers over the total capacities of all the servers, to control the overall workload of the system.

4.2 Basic Performance

We first demonstrate the efficacy of our load-balancing algorithm in achieving the latency bounds with high confidence. As discussed in Section 2.2, there is a trade-off between load balancing and the replication cost (i.e., the number of replicas). Greedily minimizing the number of replicas can cause imbalanced workload and hence jeopardize the system's overall performance in meeting the latency bounds. To the best of our knowledge, there is no existing load-balancing algorithm in literature on content distribution services with latency bounds. We compared our replication algorithm with a simple baseline heuristic that attempts to minimize the number of replicas and avoids overloading servers when creating replicas for content objects. The purpose of this comparison is to demonstrate the necessity of considering popularity of objects, servers' capacities, as well as network topology, which is what our load-balancing algorithm does. In this baseline algorithm, every server repeatedly picks a content object that has not been hosted by itself or any of its neighbor randomly, and *nominates* the server with the highest network degree among its neighbors (including itself) that have not exceeded their capacities to host the object. If all of its neighbors have exceeded their capacities, the neighbor with the highest degree is nominated. When a server receives a nomination from its neighbor, it will become a replica of the object and sends out an announcement to *all* of its neighbors.

Figure 6 gives the comparison study between our load-balancing replication algorithm and the baseline heuristic. In this experiment, there are three classes of content objects with their latency bounds being 250ms, 350ms, and 450ms, respectively. We plotted both the latency bound miss ratio and the average number of repli-

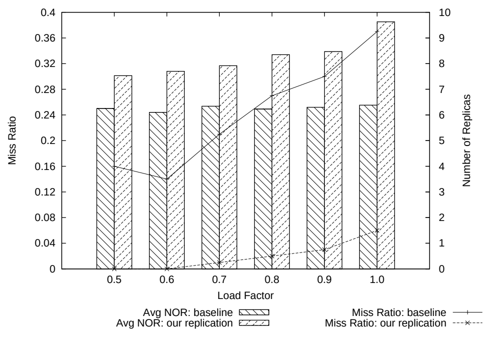
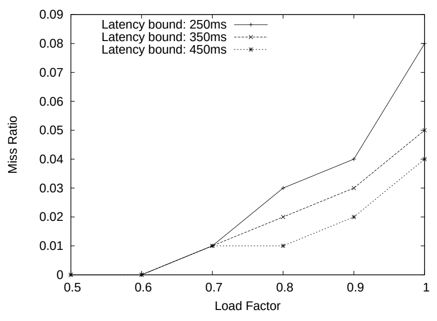
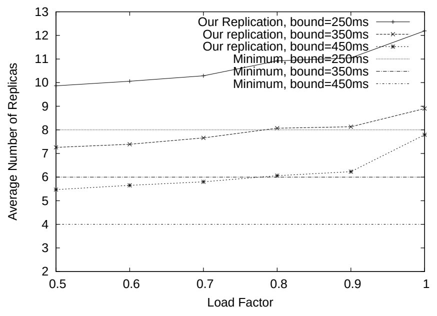


Figure 6. Comparison with Baseline



(a) Miss Ratio



(b) Average Number of Replicas

Figure 7. Impact of Latency Bound

cas (NOR) in Figure 6. The two algorithms exhibit different behaviors on both of performance metrics. When the load factor is low, the miss ratio is nearly 0 using our replication algorithm. The miss ratio slowly climbs as the load factor increases. Even when the load factor is close to 1.0 in which case the system's overall workload approaches its total capacity, the miss ratio is still blow 8%. In contrast, the miss ratio of the baseline algorithm is significantly higher across all load factors. As to the average number of replicas, the baseline algorithm expectedly produces fewer replicas. Note that, the number of replicas created by our replication algorithm increases with the load factor. This is reasonable because when the system's workload gets higher, more replicas are needed to keep the servers below their capacity thresholds.

Next, we investigate the impact of the latency bounds on the performance of our load balancing algorithm. For the purpose of better understanding this impact, we made all the content objects belong to the same content class in this experiment, and varied the latency bound of the class. The latency bound can have an impact on load balancing because the overlay network is constructed based on it. A more generous latency bound makes servers generally have more neighbors on the overlay, which should be helpful in achieving load-balancing especially when the load factor is high. This is because the more neighbors a server has, the better chance it has to have some neighbors to share its workload when its workload approaches its capacity threshold. Experimental results summarized in Figure 7(a) confirm our analysis. Regarding the number of replicas, a longer latency bound conceivably creates fewer replicas because the network is more connected, as shown in Figure 7(b).

As mentioned above, in order to maintain servers' workload below their capacities, our load-balancing algorithm may create more replicas than the minimum possible number of replicas. To understand how many more replicas are created by our algorithm, we plotted the minimum number of replicas for each latency bound in Figure 7(b). The minimum number of replicas is calculated using existing minimum dominating set (MDS) algorithms [13, 12]. Note, however, the minimum number of replicas shown here is just the theoretical lower bound. It could be achievable only when those servers forming the minimum dominating set have enough capacity to host *all* the content objects. This generally is not true in practice. In reality, servers can reach their capacity thresholds after hosting part of the objects only. The rest of the objects will have to be hosted by more servers, resulting in an average number of replicas that is larger than the minimum.

Our replication algorithm is expected to achieve good load-balancing. However, this is not to say that the workload of all servers should be strictly balanced all the time. The primary goal of our load-balancing is to keep latency bound miss ratio as low as possible. In fact, when the overall workload of the system is very low, the servers are free to have imbalanced workload as long as the miss ratio is low. When the system is heavily loaded, however, it is important that the workload of all the servers be balanced because when the overall system workload is high, imbalanced workload will inevitably cause some of the servers to exceed their capacity thresholds. Figure 8 depicts the workload imbalance of our system under different load factors. We used *coefficient of variation*, defined as $\frac{stdev}{mean}$, as the metric to describe the workload imbalance. Large coefficient of variation values imply significant imbalance. As we expected, when the system is relatively underloaded, imbalanced workload is observed. The imbalance gradually diminishes when the system workload increases.

Finally, Figure 9 compares the miss ratios at different

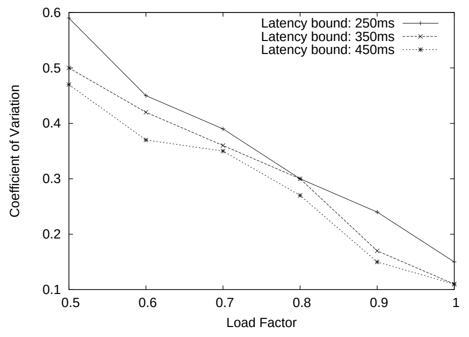


Figure 8. Coefficient of Variation

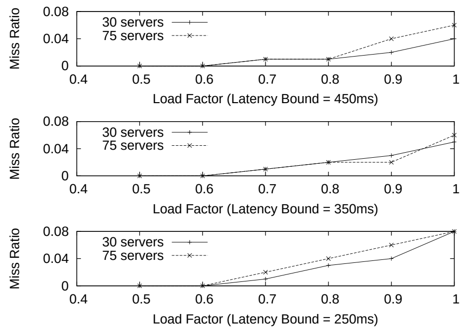


Figure 9. Different Network Scales

network scales (30 and 75 servers). Encouragingly, the miss ratios of the two scenarios do not have appreciable difference across all the load factors, which serves as evidence of good scalability of our algorithm.

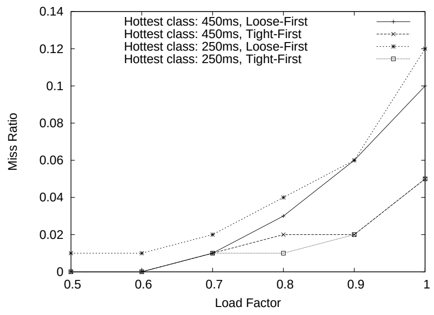
4.3 Multiple Content Classes

When there exist multiple content classes, the replication algorithm needs to run on each content class sequentially. With content classes having different latency bounds, there are basically two options regarding the sequence of running the algorithm on the content classes. We can start running the algorithm on the content class that has the most stringent latency bound, then on the less stringent bound class, and on forth. Alternatively, we can start from the class that has the most generous latency bound and proceed to the classes with more stringent bounds. We call these options "tight-first" and "loose-first", respectively. Besides latency bounds of the classes, the popularity of the objects in each content class may also have an impact on the overall load-balancing performance.

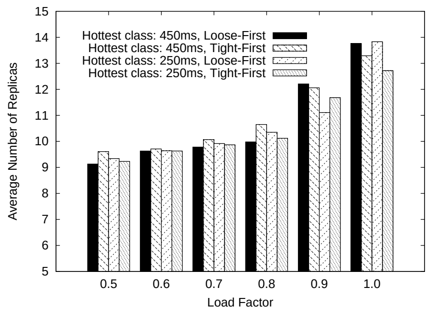
In this section, we are going to find out the performance of the tight-first and loose-first policies. Moreover, we want to reveal the impact of the objects' popularity when there are multiple content classes. Again, there are three content classes (class0, class1, class2) with latency bounds of 250ms, 350ms, and 450ms, respectively. Each content class has the same number of content objects. Regarding the popularity, we call the content class that has the most "hot objects" the "hottest class". An object is a "hot object" if it has a popularity that is 10 fold greater than the average popularity of all the content objects.

Intuitively, a more generous latency bound gives servers better chances of finding some neighbors to share their workload when overloaded. When content objects are to be replicated on servers that are largely under heavy load, it would be beneficial if the latency bound of the objects was generous. Hence, we expect the tight-first policy to outperform the loose-first policy. For similar reasons, the hottest class should be processed first. However, when these two conditions are at odds (i.e., when the hottest class is *not* the class with the most stringent latency bound), it is not obvious which factor has the dominant impact. Figure 10(a) shows the miss ratios of the 4 combinations of policies under different load factors. As shown in the figure, tight-first policy always outperforms the loose-first policy for both cases where the hottest class has the most stringent latency bound, and the hottest class has the most generous bound. Furthermore, among the 4 combinations, using tight-first when the hottest class has the tightest bound is the best. Conversely, using loose-first when the hottest class has the tightest bound is the worst. This can be observed from Figure 10(a).

As to the cost of running the load-balancing algorithm on multiple classes, there is barely discernable difference among different policies, as shown in Figure 10(b). Finally, Figure 11 gives the miss ratios of the three classes when tight-first policy is used. As can



(a) Miss Ratio



(b) Average Number of Replicas

Figure 10. Multiple Content Classes

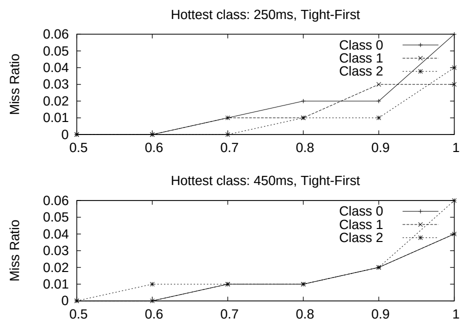


Figure 11. Fairness of Multiple Classes

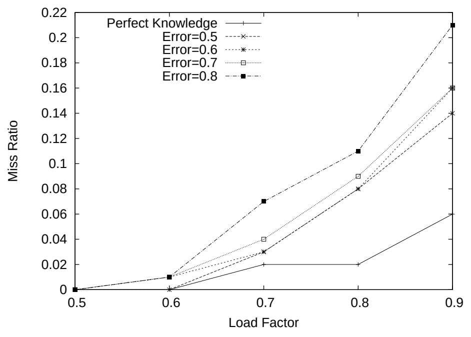


Figure 12. Imperfect Knowledge on Popularity

be observed in the figure, for both cases that the hottest class has the tightest bound and the hottest class has the loosest bound, fairness among the classes is achieved, which is ideal. From these experiment results, we conclude that tight-first policy is the best policy for running our algorithm on multiple content classes. In fact, all other experiments involving multiple classes presented in this section used this policy.

4.4 Imperfect Knowledge

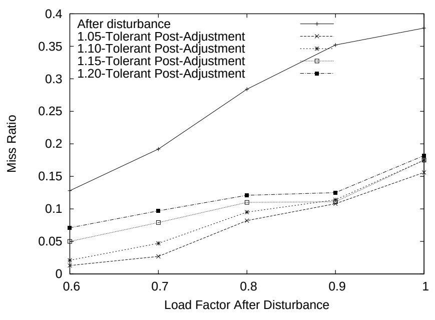
The evaluation results above are based on the assumption that the servers have perfect knowledge of the popularity of the content objects. In practice, popularity of objects can be predicted from history. However, the predictions can only be rough estimates. In this section, we examine the performance of our algorithm when servers have imperfect knowledge about the objects' popularity. We hope our replication algorithm can tolerate imperfect knowledge to a certain extent. In particular, we hope the performance in terms of miss ratio will have only a graceful degradation.

Our approach is to salt the popularity of the content objects with random noise, and vary the amount of noise. We used a parameter called *error* in adding random noise to the popularity information. With an error of ϵ, the popularity of the content objects fed to the servers will be a random number in the range of $[(1-\epsilon)p, (1+\epsilon)p]$, where p denotes the actual popularity of the content objects. Note the way we salt the popularity knowledge keeps the total workload of the system unchanged from a statistical perspective. This excludes the impact of changes to the system overall workload, and hence enables us to examine the impact of imperfect knowledge of popularity only.

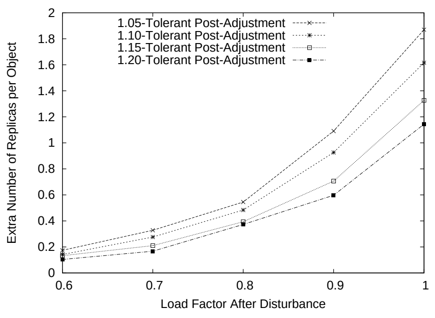
Figure 12 shows the latency bound miss ratios under different load factors and error levels. For comparison, we also gave the miss ratios when perfect knowledge about popularity was available. As can be observed in the figure, the miss ratio increases when the error is enlarged, which is understandable. Besides, the performance degradation becomes more noticeable when the load factor increases. The reason is that the imperfect knowledge of popularity basically brings imbalance to servers workload. When the system's overall workload gets close to its capacity limit, imbalanced workload will make some of the servers overloaded and hence cause more latency bound misses. With an error of 8.0, and a load factor of 0.9, the miss ratio can reach 20%. Despite this, our load-balancing replication algorithm exhibits a fairly good tolerance to imperfect knowledge of objects' popularity. This is partly because that some of the disturbance brought by imperfect knowledge can actually cancel out itself to some extent: the increase of some objects' popularity is averaged out by the decrease of some other objects. Moreover, the post-adjustment mechanism to be evaluated in the next section mitigates the performance degradation observed here.

4.5 Post-Adjustment

In this section, we study the performance of our load-balancing replication algorithm in the presence of workload dynamics. As discussed in Section 3.4, our load-balancing mechanism proceeds with the process of replicating content objects to provide bounded latencies. If the popularity of the content objects are stable over time, the overall workload of the system would be stable as well. However, in practice there exist various system dynamics. The most relevant dynamics are the changes in content objects' popularity. Operationally, the CDN system should run the replication algorithm periodically but at a low frequency, say once a day. Hence, slight or slow changes on content popularity and system's workload can be naturally covered. It is possible, however, that the objects' popularity can experience abrupt changes between two invocations of the algorithm. For example, a flash event can dramatically increase the popularity of a small set of content objects. This unquestionably has negative impact on the system's performance in terms of latency bound miss ratio. Such kind of changes not only bring higher workload to the servers, but also affects the relative popularity of the content objects. We expect that abrupt changes to content popularity can substantially increase latency bound misses, especially when the system is already heavy loaded and hence lacks the cushion to accommodate extra workload.



(a) Miss Ratio



(b) Overhead (extra number of replicas per object)

Figure 13. Post-Adjustment Mechanism

The post-adjustment mechanism (Section 3.4) is designed to deal with this situation. Servers trigger the post-adjustment mechanism when they find out that their workload transcend their capacity thresholds by more than a tolerance factor. An overloaded server will greedily attempt to "push" some of its most popular content objects to some of its relatively underloaded neighbors. Post-adjustment allows servers to offload themselves at the cost of creating more replicas for some of the content objects.

Figure 4.5 summarizes the results of our experiments to evaluate the efficacy and overhead of the post-adjustment mechanism. We simulated the dynamics of popularity by adding random disturbance to the popularity of the content objects so that the system's overall workload reaches a target level. In these experiments, we first chose a target load factor. Pairs of object-server were randomly picked to be disturbed in the way that the popularity of the object on the server was scaled up by a factor randomly chosen in the range of $[5, 20]$. Note this always increases the system's workload. The disturbance process was repeated until the overall system workload reached the target load factor. The post-adjustment mechanism was then triggered to adjust the replicas of the content objects. The load factor of the system before the disturbance was 0.5 for all the experiments. In Figure 13(a) we plotted the latency bound miss ratio with the disturbance and the miss ratio after post-adjustment settled down, varying the target workload after disturbance. Clearly, for all the load factors post-adjustment significantly reduces the latency bound misses brought by the disturbance. Besides, post-adjustment with smaller tolerance factors generally can bring down the miss ratios further. Figure 13(b) studies the major overhead of running post-adjustment, namely the extra replicas created. It gives the average extra number of replicas per content object created by post-adjustment process with different tolerance factors, varying the target workload. From the figure, it is clear that while using smaller tolerance factors can create more replicas, the cost is generally fairly moderate. Another observation worth mentioning is that when the load factor increases, the difference in performance among different tolerance factors diminishes (as shown in Figure 13(a)), while the difference in overhead increases (as shown in Figure 13(b)). This is because when the system's workload comes close to its capacity limit, many servers are saturated with their own workload and hence can not share much of the workload of those overloaded servers even though many more replicas are created.

5 Related Work

Existing load-balancing mechanisms for CDNs are mainly efficient request redirection algorithms. In the case of geographically distributed replicas, DNS-based request redirection has been widely adopted to spread client requests to servers because of its transparency to clients. More sophisticated approaches [8, 9, 4, 22] try to use information of servers' workload, network proximity, and content availability to improve overall performance. In our system, content objects are replicated on different sets of CDN servers so that a soft real-time latency bound on content retrieval is provided. This property drives our load-balancing technique to be coupled with the replica selection algorithm.

With the emergence of distributed hash table (DHT) based P2P systems, researchers recently looked into the load-balancing problem in such systems. In P2P systems structured by DHT, content objects are "hashed" to nodes by certain consistent hashing mechanisms. Karger and Ruhl [14] proposed an address-space balancing protocol that balances the distribution of identifier space to nodes to improve overall load-balancing. Godfrey et al.[10] and Rao et al. [18] addressed the load-balancing problem by using the concept of virtual servers. A virtual server looks like a single node to the underlying DHT, but each physical node can be responsible for multiple virtual servers. In our system, content objects are not subjected to the constraint of an underlying consistent hash. Instead, the placement of replicas of the content objects is subject to *latency* bound requirements.

Xiong et al.[24] presented a concurrency control mechanism for replica management. In their application scenario, replicas have stringent consistency requirements, which different from our case. ten the replicas are replicated. Wei et al.[23] investigated a problem of dynamically creating replicas of in distributed database to improve transaction processing time. Our work is very different in that our system creates replicas under the constraint of global latency bound requirment.

6 Conclusions and Future Work

In this paper, we proposed a load-balancing data replication scheme for a content distribution network providing latency bounds on content access. The approach involves a distributed algorithm to decide which servers should host which content objects. The algorithm guarantees that every server can find all the content objects within its one-hop closed neighborhood. Most importantly, the replica assignment plus a distributed post-adjustment mechanism enable the system to achieve very good load-balancing among servers. Evaluation results drawn from simulations and experiments on PlanetLab [16] using realistic content access request models show that our scheme achieves good performance in terms of meeting the latency bounds and achieving load-balancing at a moderate replication cost. For future work, we are interested in extending the performance evaluation with real traces of client traffic as opposed to synthetic workload. We also plan to investigate the performance implications when the network latencies between servers are more dynamic. Another interesting avenue for future exploration is how to extend our work to treat content classes with different tolerance to misses.

References

[1] Akamai. `http://www.akamai.com`.

[2] V. Almeida, A. Bestavros, M. Crovella, and A. Oliveira. Characterizing reference locality in the www. In *Proceedings of PDIS'96*, December 1996.

[3] L. Bao and J. J. Garcia-Luna-aceves. A new approach to channel access scheduling for ad hoc networks. In *Proceedings of ACM MobiCom 2001*, July 2001.

[4] A. Barbir, B. Cain, R. Nair, and O. Spatscheck. Known content network (CN) request-routing mechanisms, July 2003.

[5] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of ACM SIGMETRICS '98*, 1998.

[6] S. Bhattacharya, H. Kim, S. Prabh, and T. Abdelzaher. Energy-conserving data placement and asynchronous multicast in wireless sensor networks. In *Proceedings of MobiSys'03*, May 2003.

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of the IEEE INFOCOMM'99*, 1999.

[8] M. Colajanni, V. Cardellini, and P. S. Yu. Dynamic load balancing in geographically distributed heterogeneous web servers. In *Proceedings of IEEE ICDCS'98*, 1998.

[9] Z. Fei, S. Bhattacharjee, E. W. Zegura, and M. H. Ammar. A novel server selection technique for improving the response time of a replicated service. In *Proceedings of IEEE INFOCOM'98*, 1998.

[10] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured p2p systems. In *Proceedings of IEEE INFOCOM'04*, March 2004.

[11] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *19th ACM Symposium on Operating Systems Principles*, October 2003.

[12] C. Huang and T. F. Abdelzaher. Bounded-latency content distribution: Feasibility and evaluation. *IEEE Transaction on Computers*, November 2005.

[13] L. Jia, R. Rajaraman, and T. Suel. An efficient distributed algorithm for constructing small dominating sets. In *Proceedings of PODC'01*, August 2001.

[14] D. R. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings of the Third International Workshop on Peer-to-Peer Systems (IPTPS'04)*, February 2004.

[15] V. N. Padmanabhan and L. Qiu. The content and access dynamics of a busy web site: Findings and implications. In *Proceedings of ACM SIGCOMM 2000*, August 2000.

[16] PlanetLab. http://www.planet-lab.org.

[17] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *Proceedings of IEEE INFOCOM'01*, April 2001.

[18] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS'03)*, February 2003.

[19] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of OSDI'02*, December 2002.

[20] Squid Proxy Cache. `http://www.squid-cache.org`.

[21] X. Tang and J. Xu. On replica placement for qos-aware content distribution. In *Proceedings of IEEE INFOCOM'2004*, March 2004.

[22] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on CDN robustness. In *Proceedings of OSDI'02*, Boston, MA, December 2002.

[23] Y. Wei, A. Aslinger, S. Son, and J. Stankovic. ORDER: A dynamic replication algorithm for periodic transactions in distributed real-time databases. In *Proceedings of RTCSA 2004*, August 2004.

[24] M. Xiong, K. Ramamritham, J. Haritsa, , and J. Stankovic. Mirror: A state-conscious concurrency control protocol for replicated real-time databases. *Information Systems*, Vol. 27, No. 4, April 2002.

[25] G. Zhou, T. He, J. A. Stankovic, and T. F. Abdelzaher. RID: Radio interference detection in wireless sensor networks. In *Proceedings of IEEE INFOCOM 2005*, March 2005.