The Effectiveness of Request Redirection on CDN Robustness

Limin Wang, Vivek Pai and Larry Peterson
Department of Computer Science
Princeton University
{lmwang,vivek,llp}@cs.princeton.edu

Abstract

It is becoming increasingly common to construct network services using redundant resources geographically distributed across the Internet. Content Distribution Networks are a prime example. Such systems distribute client requests to an appropriate server based on a variety of factors—e.g., server load, network proximity, cache locality—in an effort to reduce response time and increase the system capacity under load. This paper explores the design space of strategies employed to redirect requests, and defines a class of new algorithms that carefully balance load, locality, and proximity. We use large-scale detailed simulations to evaluate the various strategies. These simulations clearly demonstrate the effectiveness of our new algorithms, which yield a 60-91% improvement in system capacity when compared with the best published CDN technology, yet user-perceived response latency remains low and the system scales well with the number of servers.

1 Introduction

As the Internet becomes more integrated into our everyday lives, the availability of information services built on top of it becomes increasingly important. However, overloaded servers and congested networks present challenges to maintaining high accessibility. To alleviate these bottlenecks, it is becoming increasingly common to construct network services using redundant resources, so-called Content Distribution Networks (CDN) [1, 13, 21]. CDNs deploy geographically-dispersed server surrogates and distribute client requests to an "appropriate" server based on various considerations.

CDNs are designed to improve two performance metrics: *response time* and *system throughput*. Response time, usually reported as a cumulative distribution of latencies, is of obvious importance to clients, and represents the primary marketing case for CDNs. System throughput, the average number of requests that can be satisfied each second, is primarily an issue when the system is heavily loaded, for example, when a flash crowd is accessing a small set of pages, or a Distributed Denial of Service (DDoS) attacker is targeting a particular site [15]. System throughput represents the overall robustness of the system since either a flash crowd or a DDoS attack can make portions of the information space inaccessible.

Given a sufficiently widespread distribution of servers, CDNs use several, sometimes conflicting, factors to decide how to distribute client requests. For example, to minimize response time, a server might be selected based on its *network proximity*. In contrast, to improve the overall system throughput, it is desirable to evenly *balance* the load across a set of servers. Both throughput and response time are improved if the distribution mechanism takes *locality* into consideration by selecting a server that is likely to already have the page being requested in its cache.

Although the exact combination of factors employed by commercial systems is not clearly defined in the literature, evidence suggests that the scale is tipped in favor of reducing response time. This paper addresses the problem of designing a request distribution mechanism that is both responsive across a wide range of loads, and robust in the face of flash crowds and DDoS attacks. Specifically, our main contribution is to explore the design space of strategies employed by the request redirectors, and to define a class of new algorithms that carefully balance load, locality, and proximity. We use large-scale detailed simulations to evaluate the various strategies. These simulations clearly demonstrate the effectiveness of our new algorithms: they produce a 60-91% improvement in system capacity when compared with published information about commercial CDN technology, user-perceived response latency remains low, and the system scales well with the number of servers. We also discuss several implementation issues, but evaluating a specific implementation is beyond the scope of this paper.

2 Building Blocks

The idea of a CDN is to geographically distribute a collection of *server surrogates* that cache pages normally maintained in some set of *backend servers*. Thus, rather than let every client try to connect to the original server, it is possible to spread request load across many servers. Moreover, if a server surrogate happens to reside close to the client, the client's request could be served without having to cross a long network path. In this paper, we observe this general model of a CDN, and assume any of the server surrogates can serve any request on behalf of the original server. Where to place these surrogates, and how to keep their contents up-to-date, has been addressed by other CDN research [1, 13, 21]. Here, we make no particular assumptions about servers' strategic locations.

Besides a large set of servers, CDNs also need to provide a set of *request redirectors*, which are middleware entities that forward client requests to appropriate servers based on one of the strategies described in the next section. To help understand these strategies, this section first outlines various mechanisms that could be employed to implement redirectors, and then presents a set of hashing schemes that are at the heart of redirection.

2.1 Redirector Mechanisms

Several mechanisms can be used to redirect requests [3], including augmented DNS servers, HTTP-based redirects, and smart intermediaries such as routers or proxies.

A popular redirection mechanism used by current CDNs is to augment DNS servers to return different server addresses to clients. Without URL rewriting that changes embedded objects to point to different servers, this approach has site-level granularity, while schemes that rewrite URLs can use finer granularity and thus spread load more evenly. Client-side caching of DNS mappings can be avoided using short expiration times.

Servers can perform the redirection process themselves by employing the HTTP "redirect" response. However, this approach incurs an additional round-trip time, and leaves the servers vulnerable to overload by the redirection task itself. Server bandwidth is also consumed by this process.

The redirection function can also be distributed across intermediate nodes of the network, such as routers or proxies. These redirectors either rewrite the outbound requests, or send HTTP redirect messages back to the client. If the client is not using explicit (forward mode) proxying, then the redirectors must be placed at choke points to ensure traffic in both forward and reverse directions is handled. Placing proxies closer to the edge yields well-confined easily-identifiable client populations, while moving them closer to the server can result in more accurate feedback and load information.

To allow us to focus on redirection strategies and to reduce the complexity of considering the various combinations outlined in this section, we make the following assumptions: redirectors are located at the edge of a client site, they receive the full list of server surrogates through DNS or some other out-of-band communication, they rewrite outbound requests to pick the appropriate server, and they passively learn approximate server load information by observing client communications. We do not rely on any centralization, and all redirectors operate independently. Our experiments show that these assumptions—in particular, the imperfect information about server load—do not have a significant impact on the results.

2.2 Hashing Schemes

Our geographically dispersed redirectors cannot easily adapt the request routing schemes suited for more tightly-coupled LAN environments [17, 25], since the latter can easily obtain instantaneous state about the entire system. Instead, we construct strategies that use hashing to deterministically map URLs into a small range of values. The main benefit of this approach is that it eliminates inter-redirector communication, since the same output is produced regardless of which redirector receives the URL. The second benefit is that the range of resulting hash values can be controlled, trading precision for the amount of memory used by bookkeeping.

The choice of which hashing style to use is one component of the design space, and is somewhat flexible. The various hashing schemes have some impact on computational time and request reassignment behavior on node failure/overload. However, as we discuss in the next section, the computational requirements of the various schemes can be reduced by caching.

Modulo Hashing – In this "classic" approach, the URL is hashed to a number modulo the number of servers. While this approach is computationally efficient, it is unsuitable because the modulus changes when the server set changes, causing most documents to change server assignments. While we do not expect frequent changes in the set of servers, the fact that the addition of new servers into the set will cause massive reassignment is undesirable.

Consistent Hashing [19, 20] – In this approach, the URL is hashed to a number in a large, circular space, as are the names of the servers. The URL is assigned to the server that lies closest on the circle to its hash value. A search tree can be used to reduce the search to logarithmic time. If a server node fails in this scheme, its load shifts to its neighbors, so the addition/removal of a server only causes local changes in request assignments.

Highest Random Weight [31] – This approach is the basis for CARP [8], and consists of generating a list of hash values by hashing the URL and each server's name and sorting the results. Each URL then has a deterministic order to access the set of servers, and this list is traversed until a suitably-loaded server is found. This approach requires more computation than Consistent Hashing, but has the benefit that each URL has a different server order, so a server failure results in the remaining servers evenly sharing the load. To reduce computation cost, the top few entries for each hash value can be cached.

3 Strategies

This section explores the design space for the request redirection strategies. As a quick reference, we summarize the properties of the different redirection algorithms in Table 1, where the strategies are categorized based on how they address locality, load and proximity.

Category	Strategy	Hashing Scheme	Dynamic Server Set	Load Aware
Random	Random			No
Static	R-CHash	CHash	No	No
	R-HRW	HRW	No	No
Static +Load	LR-CHash	CHash	No	Yes
	LR-HRW	HRW	No	Yes
Dynamic	CDR	HRW	Yes	Yes
	FDR	HRW	Yes	Yes
	FDR-Global	HRW	Yes	Yes
Network Proximity	NPR-CHash	CHash	No	No
	NPLR-CHash	CHash	No	Yes
	NP-FDR	HRW	Yes	Yes

Table 1: Properties of Request Redirection Strategies

The first category, Random, contains a single strategy, and is used primarily as a baseline. We then discuss four static algorithms, in which each URL is mapped onto a fixed set of server replicas—the Static category includes two schemes based on the best-known published algorithms, and the Static+Load category contains two variants that are aware of each replica's load. The four algorithms in these two static categories pay increasing attention to locality. Next, we introduce two new algorithms—denoted **CDR** and **FDR**—that factor both load and locality into their decision, and each URL is mapped onto a dynamic set of server replicas. We call this the Dynamic category. Finally, we factor network proximity into the equation, and present another new algorithm—denoted **NP-FDR**—that considers all aspects of network proximity, server locality, and load.

3.1 Random

In the random policy, each request is randomly sent to one of the server surrogates. We use this scheme as a baseline to determine a reasonable level of performance, since we expect the approach to scale with the number of servers and to not exhibit any pathological behavior due to patterns in the assignment. It has the drawback that adding more servers does not reduce the working set of each server. Since serving requests from main memory is faster than disk access, this approach is at a disadvantage versus schemes that exploit URL locality.

3.2 Static Server Set

We now consider a set of strategies that assign a fixed number of server replicas to each URL. This has the effect of improving locality over the Random strategy.

3.2.1 Replicated Consistent Hashing

In the Replicated Consistent Hashing (**R-CHash**) strategy, each URL is assigned to a set of replicated servers. The number of replicas is fixed, but configurable. The URL is hashed to a value in the circular space, and the replicas are evenly spaced starting from this original point. On each request, the redirector randomly assigns the request to one of the replicas for the URL. This strategy is intended to model the mechanism used in published content distribution networks, and is virtually identical[1] to the scheme described in [19] and [20] with the network treated as a single geographic region.

3.2.2 Replicated Highest Random Weight

The Replicated Highest Random Weight (**R-HRW**) strategy is the counterpart to R-CHash, but with a different hashing scheme used to determine the replicas. To the best of our knowledge, this approach is not used in any existing content distribution network. In this approach, the set of replicas for each URL is determined by using the top N servers from the ordered list generated by Highest Random Weight hashing. Versus R-CHash, this scheme is less likely to generate the same set of replicas for two different URLs. As a result, the less-popular URLs that may have some overlapping servers with popular URLs are also likely to have some other less-loaded nodes in their replica sets.

3.3 Load-Aware Static Server Set

The Static Server Set schemes randomly distribute requests across a set of replicas, which shares the load but without any active monitoring. We extend these schemes by introducing load-aware variants of these approaches. To perform fine-grained load balancing, these schemes maintain local estimates of server load at the redirectors, and use this information to pick the least-loaded member of the server set. The load-balanced variant of R-CHash is called **LR-CHash**, while the counterpart for R-HRW is called **LR-HRW**.

3.4 Dynamic Server Set

We now consider a new category of algorithms that dynamically adjust the number of replicas used for each URL in an attempt to maintain both good server locality and load balancing. By reducing unnecessary replication, the working set of each server is reduced, resulting in better file system caching behavior.

[1] The scheme described in these papers also includes a mechanism to use coarse-grained load balancing via virtual server names. When server overload is detected, the corresponding content is replicated across all servers in the region, and the degree of replication shrinks over time. However, the schemes are not described in enough detail to replicate.

3.4.1 Coarse Dynamic Replication

Coarse Dynamic Replication (**CDR**) adjusts the number of replicas used by redirectors in response to server load and demand for each URL. Like R-HRW, CDR uses HRW hashing to generate an ordered list of servers. Rather than using a fixed number of replicas, however, the request target is chosen using coarse-grained server load information to select the first "available" server on the list.

Figure 1 shows how a request redirector picks the destination server for each request. This decision process is done at each redirector independently, using the load status of the possible servers. Instead of relying on heavy communications between servers and request redirectors to get server load status, we use local load information observed by each redirector as an approximation. We currently use the number of active connections to infer the load level, but we can also combine this information with response latency, bandwidth consumption, etc.

```
find_server(url, S) {
  foreach server s_i in server set S,
    weight_i = hash(url, address(s_i));
  sort weight;
  foreach server s_j in decreasing order of weight_j {
    if satisfy_load_criteria(s_j) then {
      targetServer ← s_j;
      stop search;
    }
  }
  if targetServer is not valid then
    targetServer ← server with highest weight;
  route request url to targetServer;
}
```

Figure 1: Coarse Dynamic Replication

As the load increases, this scheme changes from using only the first server on the sorted list to spreading requests across several servers. Some documents normally handled by "busy" servers will also start being handled by less busy servers. Since this process is based on aggregate server load rather than the popularity of individual documents, servers hosting some popular documents may find more servers sharing their load than servers hosting collectively unpopular documents. In the process, some unpopular documents will be replicated in the system simply because they happen to be primarily hosted on busy servers. At the same time, if some documents become extremely popular, it is conceivable that all of the servers in the system could be responsible for serving them.

3.4.2 Fine Dynamic Replication

A second dynamic algorithm—Fine Dynamic Replication (**FDR**)—addresses the problem of unnecessary replication in CDR by keeping information on URL popularity and using it to more precisely adjust the number of replicas. By controlling the replication process, the per-server working sets should be reduced, leading to better server locality, and thereby better response time and throughput.

The introduction of finer-grained bookkeeping is an attempt to counter the possibility of a "ripple effect" in CDR, which could gradually reduce the system to round-robin under heavy load. In this scenario, a very popular URL causes its primary server to become overloaded, causing extra load on other machines. Those machines, in turn, also become overloaded, causing documents destined for them to be served by their secondary servers. Under heavy load, it is conceivable that this displacement process ripples through the system, reducing or eliminating the intended locality benefits of this approach.

```
find_server(url, S) {
  walk_entry ← walkLenHash(url);
  w_len ← walk_entry.length;
  foreach server s_i in server set S,
    weight_i = hash(url, address(s_i));
  sort weight;
  s_candidate ← least-loaded server of top w_len servers;
  if satisfy_load_criteria(s_candidate) then {
    targetServer ← s_candidate;
    if (w_len > 1 &&
        timenow() − walk_entry.lastUpd > chgThresh)
      walk_entry.length −−;
  } else {
    foreach rest server s_j in decreasing weight order {
      if satisfy_load_criteria(s_j) then {
        targetServer ← s_j;
        stop search;
      }
    }
    walk_entry.length ← actual search steps;
  }
  if walk_entry.length changed then
    walk_entry.lastUpd ← timenow();
  if targetServer is not valid then
    targetServer ← server with highest weight;
  route request url to targetServer;
}
```

Figure 2: Fine Dynamic Replication

To reduce extra replication, FDR keeps an auxiliary structure at each redirector that maps each URL to a "walk length," indicating how many servers in the HRW list should be used for this URL. Using a minimum walk length of one provides minimal replication for most URLs, while using a higher minimum will always distribute URLs over multiple servers. When the redirector receives a request, it uses the current walk length for the URL and picks the least-loaded server from the current set. If even this server is busy, the walk length is increased and the least-loaded server is used.

This approach tries to keep popular URLs from overloading servers and displacing less-popular objects in the process. The size of the auxiliary structure is capped by hashing the URL into a range in the thousands to millions. Hash collisions may cause some URLs to have their replication policies affected by popular URLs. As long as the the number of hash values exceeds the number of servers, the granularity will be significantly better than the Coarse Dynamic Replication approach. The redirector logic for this approach is shown in Figure 2. To handle URLs that become less popular over time, with each walk length, we also keep the time of its last modification. We decrease the walk length if it has not changed in some period of time.

As a final note, both dynamic replication approaches require some information about server load, specifically how many outstanding requests can be sent to a server by a redirector before the redirector believes it is busy. We currently allow the redirectors to have 300 outstanding requests per server, at which point the redirector locally decides the server is busy. It would also be possible to calibrate these values using both local and global information—using its own request traffic, the redirector can adjust its view of what constitutes heavy load, and it can perform opportunistic communication with other redirectors to see what sort of collective loads are being generated. The count of outstanding requests already has some feedback, in the sense that if a server becomes slow due to its resources (CPU, disk, bandwidth, etc.) being stressed, it will respond more slowly, increasing the number of outstanding connections. To account for the inaccuracy of local approximation of server load at each redirector, in our evaluations, we also include a reference strategy, **FDR-Global**, where all redirectors have perfect knowledge of the load at all servers.

Conceivably, Consistent Hashing could also be used to implement CDR and FDR. We tested a CHash-based CDR and FDR, but they suffer from the "ripple effect" and sometimes yield even worse performance than load-aware static replication schemes. Part of the reason is that in Consistent Hashing, since servers are mapped onto a circular space, the relative order of servers for each URL will be *effectively* the same. This means the load migration will take an uniform pattern; and the less-popular URLs that may have overlapping servers with popular URLs are unlikely to have some other less-loaded nodes in their replica sets. Therefore, in this paper, we will only present CDR and FDR based on HRW.

3.5 Network Proximity

Many commercial CDNs start server selection with network proximity matching. For instance, [19] indicates that CDN's hierarchical authoritative DNS servers can map a client's (actually its local DNS server's) IP address to a geographic region within a particular network and then combine it with network and server load information to select a server separately within each region. Other research [18] shows that in practice, CDNs succeed not by always choosing the "optimal" server, but by avoiding notably bad servers.

For the sake of studying system capacity, we make a conservative simplicifaction by treating the entire network topology as a single geographic region. We could also simply take the hierarchical region approach as in [19], however, to see the effect of *integrating* proximity into server selection, we introduce three strategies that explicitly factor intra-region network proximity into the decision. Our redirector measures servers' geographical/topological location information through *ping*, *traceroute* or similiar mechanisms and uses this information to calculate an "effective load" when choosing servers.

To calculate the *effective load*, redirectors multiply the raw load metric with a normalized *standard distance* between the redirector and the server. Redirectors gather distances to servers using round trip time (RTT), routing hops, or similar information. These raw distances are normalized by dividing by the minimum locally-observed distance, yielding the standard distance. In our simulations, we use RTT for calculating raw distances.

FDR with Network Proximity (NP-FDR) is the counterpart of FDR, but it uses effective load rather than raw load. Similarly, **NPLR-CHash** is the proximity-aware version of LR-CHash. The third strategy, **NPR-CHash**, adds network proximity to the load-oblivious R-CHash approach by assigning requests such that each surrogate in the fixed-size server set of a URL will get a share of total requests for that URL inversely proportional to the surrogate's distance from the redirector. As a result, closer servers in the set get a larger share of the load.

The use of effective load biases server selection in favor of closer servers when raw load values are comparable. For example, in standard FDR, raw load values reflect the fact that distant servers generate replies more slowly, so some implicit biasing exists. However, by explicitly factoring in proximity, NP-FDR attempts to reduce global resource consumption by favoring shorter network journeys.

Although we currently calculate effective load this way, other options exist. For example, effective load can take other dynamic load/proximity metrics into account, such as network congestion status through real time measurement, thereby reflecting instantaneous load conditions.

4 Evaluation Methodology

The goal of this work is to examine how these strategies respond under different loads, and especially how robust they are in the face of flash crowds and other abnormal workloads that might be used for a DDoS attack. Attacks may take the form of legitimate traffic, making them difficult to distinguish from flash crowds.

Evaluating the various algorithms described in Section 3 on the Internet is not practical, both due to the scale of the experiment required and the impact a flash crowd or attack is likely to have on regular users. Simulation is clearly the only option. Unfortunately, there has not been (up to this point) a simulator that considers both network traffic and server load. Existing simulators either focus on the network, assuming a constant processing cost at the server, or they accurately model server processing (including the cache replacement strategy), but use a static estimate for the network transfer time. In the situations that interest us, both the network and the server are important.

To remedy this situation, we developed a new simulator that combines network-level simulation with OS/server simulation. Specifically, we combine the NS simulator with Logsim, allowing us to simulate network bottlenecks, round-trip delays, and OS/server performance. NS-2 [23] is a packet-level simulator that has been widely-used to test TCP implementations. However, it does not simulate much server-side behavior. Logsim is a server cluster simulator used in previous research on LARD [25], and it provides detailed and accurate simulation of server CPU processing, memory usage, and disk access. This section describes how we combine these two simulators, and discusses how we configure the resulting simulator to study the algorithms presented in Section 3.

4.1 Simulator

A model of Logsim is shown in Figure 3. Each server node consists of a CPU and locally attached disk(s), with separate queues for each. At the same time, each server node maintains its own memory cache of a configurable size and replacement policy. Incoming requests are first put into the holding queue, and then moved to the active queue. The active queue models the parallelism of the server, for example, in multiple process or thread server systems, the maximum number of processes or threads allowed on each server.

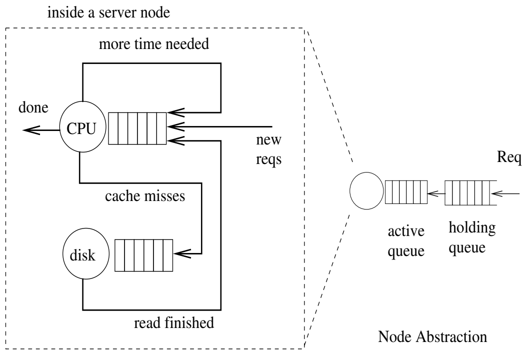


Figure 3: Logsim Simulator

We combined Logsim with NS-2 as follows. We keep NS-2's event engine as the main event manager, wrap each Logsim event as a NS-2 event, and insert it into the NS-2 event queue. All the callback functions are kept unchanged in Logsim. When crossing the boundary between the two simulators, tokens (continuations) are used to carry side-specific information. To speed up the simulation time, we also re-implemented several NS-2 modules and performed other optimizations.

On the NS-2 side, all packets are stored and forwarded, as in a real network, and we use two-way TCP. We currently use static routing within NS-2, although we may run simulations with dynamic routing in the future.

On the Logsim side, the costs for the basic request processing were derived by performing measurements on a 300MHz Pentium II machine running FreeBSD 2.2.5 and the Flash web server [24]. Connection establishment and tear-down costs are set at 145μs, while transmit processing incurs 40μs per 512 bytes. Using these numbers, an 8-KByte document can be served from the main memory cache at a rate of approximately 1075 requests/sec. When disk access is needed, reading a file from the disk has a latency of 28ms. The disk transfer time is 410μs per 4 KBytes. For files larger than 44 KBytes, and additional 14ms is charged for every 44 KBytes of file length in excess of 44 KBytes. The replacement policy used on the servers is Greedy-Dual-Size (GDS)[5], as it appears to be the best known policy for Web workloads. 32MB memory is available for caching documents on each server and every server node has one disk. This server is intentionally slower than the current state-of-the-art (it is able to service approximately 600 requests per second), but this allows the simulation to scale to a larger number of nodes.

The final simulations are very heavy-weight, with over a thousand nodes and a very high aggregate request rate. We run the simulator on a 4-processor/667MHz Alpha with 8GB RAM. Each simulation requires 2-6GB of RAM, and generally takes 20-50 hours of wall-clock time.

4.2 Network Topology

It is not easy to find a topology that is both realistic and makes the simulation manageable. We choose to use a slightly modified version the NSFNET backbone network T3 topology, as shown in Figure 4.

In this topology, the round-cornered boxes represent backbone routers with the approximate geographical location label on it. The circles, tagged as R1, R2..., are regional routers;[2] small circles with “C” stand for client hosts; and shaded circles with “S” are the server surrogates. In the particular configuration shown in the figure, we put 64 servers behind regional routers R0, R1, R7, R8, R9, R10, R15, R19, where each router sits in front of 8 servers. We distribute 1,000 client hosts evenly behind the other regional routers, yielding a topology of nearly 1,100 nodes. The redirector algorithms run on the regional routers that sit in front of the clients.

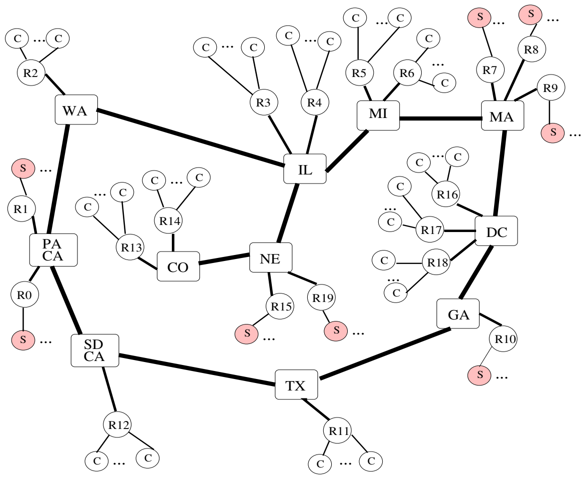


Figure 4: Network Topology

The latencies of servers to regional routers are set randomly between 1ms to 3ms; those of clients to regional routers are between 5ms and 20ms; those of regional routers to backbone routers are between 1 to 10ms; latencies between backbone routers are set roughly according to their geographical distances, ranging from 8ms to 28ms.

To simulate high request volume, we deliberately provision the network with high link bandwidth by setting the backbone links at 2,488Mbps, and links between regional routers and backbone routers at 622Mbps. Links between servers and regional routers are 100Mbps and those between clients and their regional servers are randomly between 10Mbps and 45Mbps. All the queues at routers are drop tail, with the backbone routers having room to buffer 1024 packets, and all other routers able to buffer 512 packets.

[2]These can also be thought of as edge/site routers, or the boundary to an autonomous system

4.3 Workload and Stability

We determine system capacity using a trace-driven simulation and gradually increase the aggregate request rate until the system fails. We use a two month trace of server logs obtained at Rice University, which contains 2.3 million requests for 37,703 files with a total size of 1,418MB [25], and has properties similar to other published traces.

The simulation starts with the clients sharing the trace and sending requests at a low aggregate rate in an open-queue model. Each client gets the name of the document sequentially from the shared trace when it needs to send a request, and the timing information in the trace is ignored. The request rate is increased by 1% every simulated six seconds, regardless of whether previous requests have completed. This approach gradually warms the server memory caches and drives the servers to their limits over time. We configure Logsim to handle at most 512 simultaneous requests and queue the rest. The simulation is terminated when the offered load overwhelms the servers.

Flash crowds, or DDoS attacks in bursty legitimate traffic form, are simulated by randomly selecting some clients as *intensive* requesters and randomly picking a certain number of hot-spot documents. These intensive requesters randomly request the hot documents at the same rate as normal clients, making them look no different than other legitimate users. We believe that this random distribution of intensive requesters and hot documents is a quite general assumption since we do not require any special detection or manual intervention to signal the start of a flash crowd or DDoS attack.

We define a server as being overloaded when it can no longer satisfy the rate of incoming requests and is unlikely to recover in the future. This approach is designed to determine when service is actually being denied to clients, and to ignore any short-term behavior which may be only undesirable, rather than fatal. Through experimentation, we find that when a server’s request queue grows beyond 4 to 5 times the number of simultaneous connections it can handle, throughput drops and the server is unlikely to recover. Thus, we define the threshold for a server *failure* to be when the request queue length exceeds five times the simultaneous connection parameter. Since we increase the offered load 1% every 6 seconds, we record the request load exactly 30 seconds before the first server fails, and declare this to be the system’s maximum capacity.

Although we regard any single server failure as a system failure in our simulation, the strategies we evaluate all exhibit similar behavior—significant numbers of servers fail at the same time, implying that our approach to deciding system capacity is not biased toward any particular scheme.

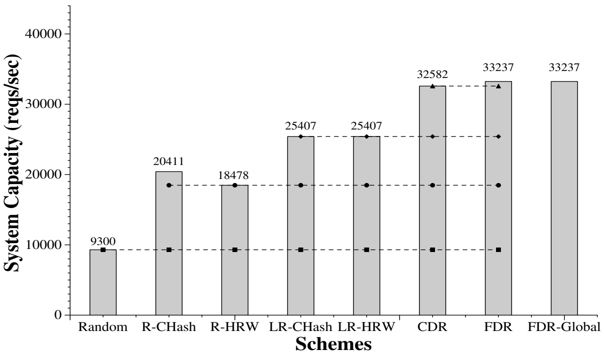


Figure 5: Capacity Comparison under Normal Load

5 Results

This section evaluates how the different strategies in Table 1 perform, both under normal conditions and under flash crowds or DDoS attacks. Network proximity and other factors that affect the performance of these strategies are also addressed.

5.1 Normal Workload

Before evaluating these strategies under flash crowds or other attack, we first measure their behavior under normal workloads. In these simulations, all clients generate traffic similar to normal users and gradually increase their request rates as discussed in Section 4.3. We compare aggregate system capacity and user-perceived latency under the different strategies, using the topology shown in Figure 4.

5.1.1 Optimal Static Replication

The static replication schemes (R-CHash, R-HRW, and their variants) use a configurable (but fixed) number of replicas, and this parameter's value influences their performance. Using a single replica per URL perfectly partitions the file set, but can lead to early failure of servers hosting popular URLs. Using as many replicas as available servers degenerates to the Random strategy. To determine an appropriate value, we varied this parameter between 2 and 64 replicas for R-CHash when there are 64 servers available. Increasing the number of replicas per URL initially helps to improve the system's throughput as the load gets more evenly distributed. Beyond a certain point, throughput starts decreasing due to the fact that each server is presented with a larger working set, causing more disk activity. In the 64-server case—the scenario we use throughout the rest of this section—10 server replicas for each URL achieves the optimal system capacity. For all of the remaining experiments, we use this value in the R-CHash and R-HRW schemes and their variants.

5.1.2 System Capacity

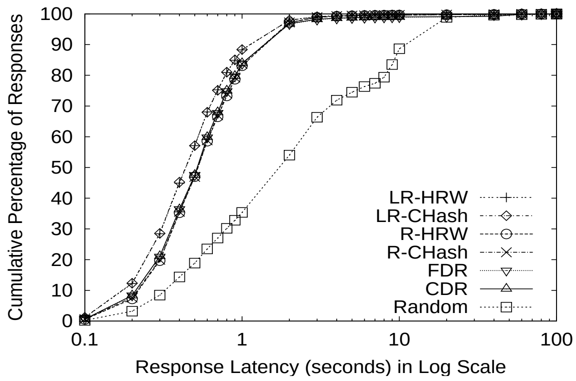
The maximum aggregate throughput of the various strategies with 64 servers are shown in Figure 5. Here we do not plot all the strategies and variants, but focus on those impacting throughput substantially. Random shows the lowest throughput at 9,300 req/s before overload. The static replication schemes, R-CHash and R-HRW, outperform Random by 119% and 99%, respectively. Our approximation of static schemes' best behaviors, LR-CHash and LR-HRW, yields 173% better capacity than Random. The dynamic replication schemes, CDR and FDR, show over 250% higher throughput than Random, or more than a 60% improvement over the static approaches and 28% over static schemes with fine-grained load control.

The difference between Random and the static approaches stems from the locality benefits of the hashing in the static schemes. By partitioning the working set, more documents are served from memory by the servers. Note, however, that absolute minimal replication can be detrimental, and in fact, the throughput for only two replicas in in Section 5.1.1 is actually lower than the throughput for Random. The difference in throughput between R-CHash and R-HRW is 10% in our simulation. However, this difference should not be over emphasized, because changes in the number of servers or workload can cause their relative ordering to change. Considering load helps the static schemes gain about 25% better throughput, but they still do not exceed the dynamic approaches.

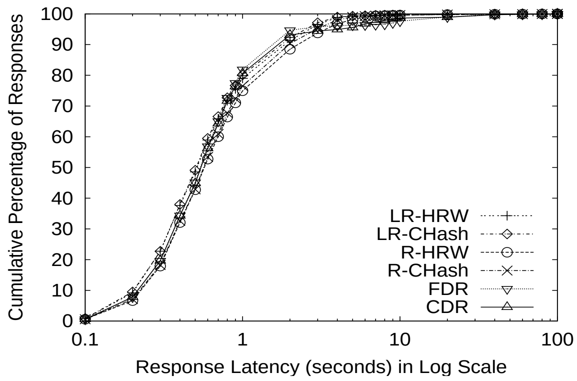
The performance difference between the static (including with load control) and dynamic schemes stems from the adjustment of the number of replicas for the documents. FDR also shows 2% better capacity than CDR.

Interestingly, the difference between our dynamic schemes (with only local knowledge) and the FDR-Global policy (with perfect global knowledge) is minimal. These results suggest that request distribution policies not only fare well with only local information, but that adding more global information may not gain much in system capacity.

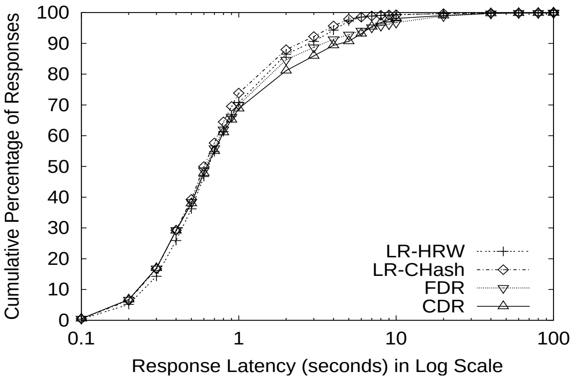
Examination of what ultimately causes overload in these systems reveals that, under normal load, the server's behavior is the factor that determines the performance limit of the system. None of the schemes suffers from saturated network links in these non-attack simulations. For Random, due to the large working set, the disk performance is the limit of the system, and before system failure, the disks exhibit almost 100% activity while the CPU remains largely idle. The R-CHash, R-HRW and LR-CHash and LR-HRW exhibit much lower disk utilization at comparable request rates; but by the time the system becomes overloaded, their bottleneck



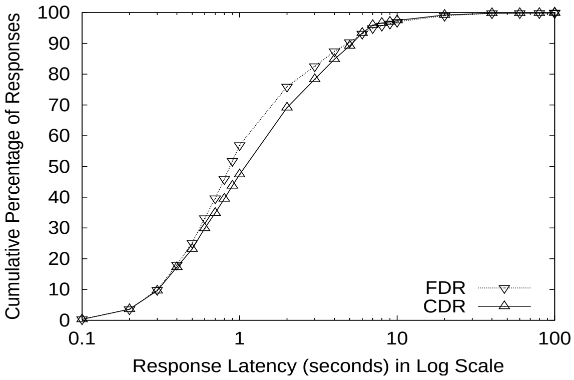
(a) Random's limit: 9,300 req/s



(b) R-HRW's limit: 18,478 req/s



(c) LR-HRW's limit: 25,407 req/s



(d) CDR's limit: 32,582 req/s

Figure 6: Response Latency Distribution under Normal Load

Utilization Scheme	*CPU (%)*		*DISK (%)*	
	Mean	Stddev	Mean	Stddev
Random	21.03	1.36	100.00	0.00
R-CHash	57.88	18.36	99.15	3.89
R-HRW	47.88	15.33	99.74	1.26
LR-CHash	59.48	18.85	97.83	12.51
LR-HRW	58.43	16.56	99.00	5.94
CDR	90.07	11.78	36.10	25.18
FDR	93.86	7.58	33.96	20.38
FDR-Global	91.93	11.81	17.60	15.43

Table 2: Server Resource Utilization at Overload

also becomes the disk and the CPU is roughly 50-60% utilized on average. In the CDR and FDR cases, at system overload, the average CPU is over 90% busy, while most of the disks are only 10-70% utilized. Table 2 summarizes resource utilization of different schemes before server failures (not at the same time point).

These results suggest that the CDR and FDR schemes are the best suited for technology trends, and can most benefit from upgrading server capacities. The throughput of our simulated machines is lower than what can be expected from state-of-the-art machines, but this decision to scale down resources was made to keep the simulation time manageable. *With faster simulated machines, we expect the gap between the dynamic schemes and the others to grow even larger.*

5.1.3 Response Latency

Along with system capacity, the other metric of interest is user-perceived latency, and we find that our schemes also perform well in this regard. To understand the latency behavior of these systems, we use the capacity measurements from Figure 5 and analyze the latency of all of the schemes whenever one category reaches its performance limit. For schemes with similar performance in the same category, we pick the lower limit for the analysis so that we can include numbers for the higher-performing scheme. In all cases, we present the cumulative distribution of all request latencies as well as some statistics

Req Rate	9,300 req/s				18,478 req/s				25,407 req/s				32,582 req/s			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
Random	3.95	1.78	11.32	6.99												
R-CHash	0.79	0.53	1.46	2.67	1.01	0.57	1.98	3.58								
R-HRW	0.81	0.53	1.49	2.83	1.07	0.57	2.28	3.22								
LR-CHash	0.68	0.44	1.17	2.50	0.87	0.51	1.82	2.74	1.19	0.60	2.47	3.79				
LR-HRW	0.68	0.44	1.18	2.50	0.90	0.51	1.89	3.13	1.27	0.64	2.84	3.76				
CDR	1.16	0.52	1.47	5.96	1.35	0.55	1.75	6.63	1.86	0.63	4.49	6.62	2.37	1.12	5.19	7.21
FDR	1.10	0.52	1.48	5.49	1.35	0.54	1.64	6.70	1.87	0.62	3.49	6.78	2.22	0.87	4.88	7.12
FDR-Global	0.78	0.50	1.42	2.88	0.97	0.54	1.58	5.69	1.11	0.56	1.86	5.70	1.35	0.66	2.35	6.29

Table 3: Response Latency of Different Strategies under Normal Load. μ — Mean, σ — Standard Deviation.

about the distribution.

Figure 6 plots the cumulative distribution of latencies at four request rates: the maximums for Random, R-HRW, LR-HRW, and CDR (the algorithm in each category with the smallest maximum throughput). The x-axis is in log scale and shows the time needed to complete requests. The y-axis shows what fraction of all requests finished in that time. The data in Table 3 gives mean, median, 90th percentile and standard deviation details of response latencies at our comparison points.

The response time improvement from exploiting locality is most clearly seen in Figure 6a. At Random's capacity, most responses complete under 4 seconds, but a few responses take longer than 40 seconds. In contrast, all other strategies have median times almost one-fourth that of Random, and even their 90th percentile results are less than Random's median. These results, coupled with the disk utilization information, suggest that most requests in the Random scheme are suffering from disk delays, and that the locality improvement techniques in the other schemes are a significant benefit.

The benefit of FDR over CDR is visible in Figure 6d, where the plot for FDR lies to the left of CDR. The statistics also show a much better median response time, in addition to better mean and 90th percentile numbers. FDR-Global has better numbers in all cases than CDR and FDR, due to its perfect knowledge of server load status.

An interesting observation is that when compared to the static schemes, dynamic schemes have worse mean times but comparable/better medians and 90th percentile results. We believe this behavior stems from the time required to serve the largest files. Since these files are less popular, the dynamic schemes replicate them less than the static schemes do. As a result, these files are served from a smaller set of servers, causing them to be served more slowly than if they were replicated more widely. We do not consider this behavior to be a significant drawback, and note that some research explicitly aims to achieve this effect [10, 11]. We will revisit large file issues in section 5.4.2.

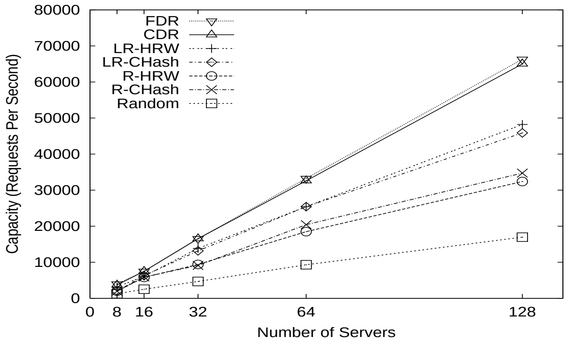


Figure 7: System Scalability under Normal Load

5.1.4 Scalability

Robustness not only comes from resilience with certain resources, but also from good scalability with increasing resources. We repeat similar experiments with different number of servers, from 8 to 128, to test how well these strategies scale. The number of server-side routers is not changed, but instead, more servers are attached to each server router as the total number of servers increases.

We plot system capacity against the number of servers in Figure 7. They all display near-linear scalability, implying all of them are reasonably good strategies when the system becomes larger. Note, for CDR and FDR with 128 servers, our original network provision is a little small. The bottleneck in that case is the link between the server router and backbone router, which is 622Mbps. In this scenario, each server router is handling 16 servers, giving each server on average only 39Mbps of traffic. At 600 reqs/s, even an average size of 10KB requires 48Mbps. Under this bandwidth setup, CDR and FDR yield similar system capacity as LR-CHash and LR-HRW, and all these 4 strategies saturate server-router-to-backbone links. To remedy this situation, we run simulations of 128 servers for all strategies with doubled bandwidth on both the router-to-backbone and backbone links. Performance numbers of 128 servers under these faster links are plotted in the graph instead. This problem can also be solved by placing fewer servers behind each pipe and instead spreading them across more locations.

5.2 Behavior Under Flash Crowds

Having established that our new algorithms perform well under normal workloads, we now evaluate how they behave when the system is under a flash crowd or DDoS attack. To simulate a flash crowd, we randomly select 25% of the 1,000 clients to be *intensive* requesters, where each of these requesters repeatedly issues requests from a small set of pre-selected URLs with an average size of about 6KB.

5.2.1 System Capacity

Figure 8 depicts the system capacity of 64 servers under a flash crowd with a set of 10 URLs. In general, it exhibits similar trends as the no-attack case shown in Figure 5. Importantly, the CDR and FDR schemes still yield the best throughput, making them most robust to flash crowds or attacks. Two additional points deserve more attention.

First, FDR now has a similar capacity with CDR, but still is more desirable as it provides noticeably better latency, as we will see later. FDR's benefit over R-CHash and R-HRW has grown to 91% from 60% and still outperforms LR-CHash and LR-HRW by 22%.

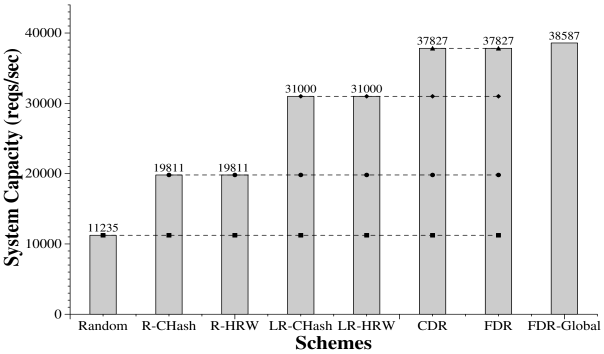


Figure 8: Capacity Comparison Under Flash Crowds

Second, the absolute throughput numbers tend to be larger than the no-attack case, because the workload is also different. Here, 25% of the traffic is now concentrated on 10 URLs, and these attack URLs are relatively small, with an average size of 6KB. Therefore, relative difference among different strategies within each scenario yields more useful information than simply comparing performance numbers across these two scenarios.

5.2.2 Response Latency

The cumulative distribution of response latencies for all seven algorithms under attack are shown in Figure 9. Also, the statistics for all seven algorithms and FDR-Global are given in Table 4. As seen from the figure and table, R-CHash, R-HRW, LR-CHash, LR-HRW CDR and FDR still have far better latency than Random, and static schemes are a little better than CDR and FDR at Random, R-HRW's and LR-HRW's failure points; and LR-CHash and LR-HRW yields slightly better latency than R-CHash and R-HRW.

As we explained earlier, CDR and FDR adjust the server replica set in response to request volume. The number of replicas that serve attack URLs increases as the attack ramps up, which may adversely affect serving non-attack URLs. However, the differences in the mean, median, and 90-percentile are not large, and all are probably acceptable to users. The small price paid in response time for CDR and FDR brings us higher system capacity, and thus, stronger resilience to various loads.

5.2.3 Scalability

We also repeat the scalability test under flash crowd or attack, where 250 clients are *intensive* requesters that repeatedly request 10 URLs. As shown in Figure 10, all strategies scale linearly with the number of servers. Again, in the 128-server cases, we use doubled bandwidth on the router-to-backbone and backbone links.

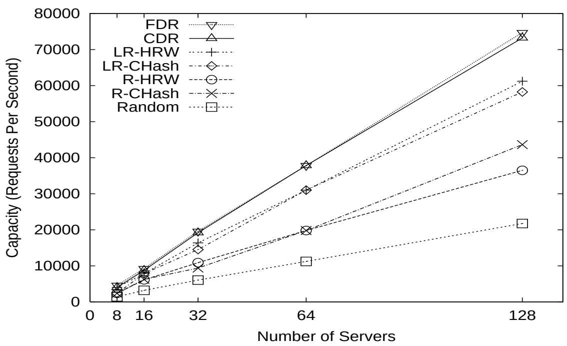
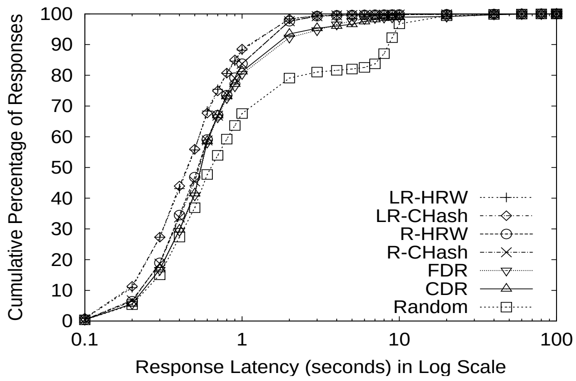


Figure 10: System Scalability under Flash Crowds

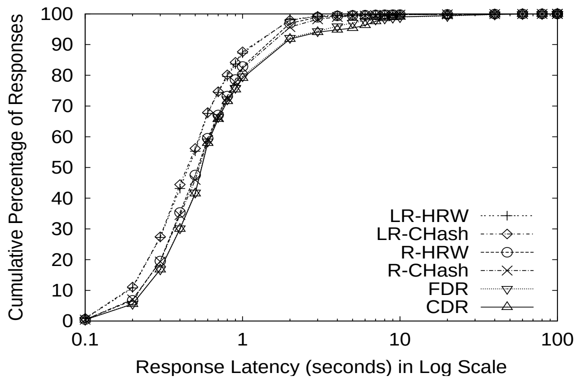
5.2.4 Various Flash Crowds

Throughout our simulations, we have seen that a different number of *intensive* requesters, and a different number of hot or attacked URLs, have an impact on system performance. To further investigate this issue, we carry out a series of simulations by varying both the number of intensive requesters and the number of hot URLs. Since it is impractical to exhaust all possible combinations, we choose two classes of flash crowds. One class has a single hot URL of size 1KB. This represents a small home page of a website. The other class has 10 hot URLs averaging 6KB, as before. In both cases, we vary the percentage of the 1000 clients that are intensive requesters from 10% to 80%. The results of these two experiments with 32 servers are shown in Figures 11 and 12, respectively.

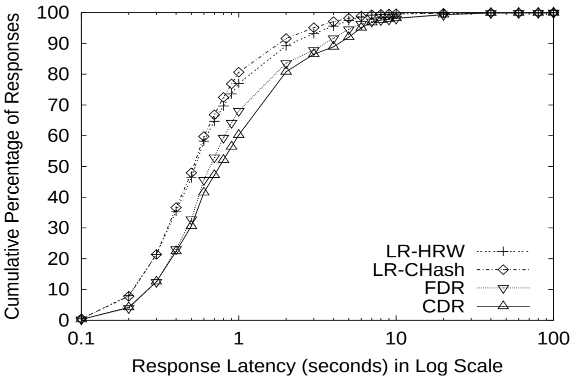
In the first experiment, as the portion of *intensive* requesters increases, more traffic is concentrated on this one URL, and the request load becomes more unbalanced. Random, CDR and FDR adapt to this change



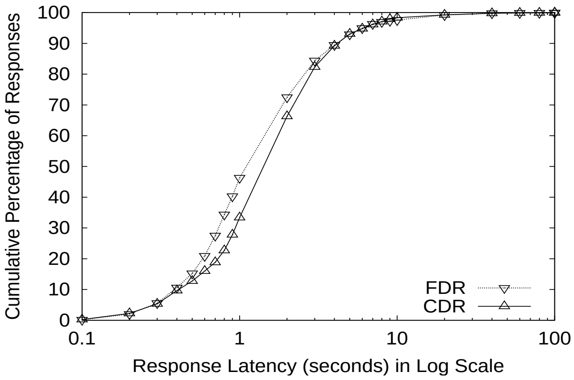
(a) Random's limit: 11,235 req/s



(b) R-HRW's limit: 19,811 req/s



(c) LR-HRW's limit: 31,000 req/s



(d) CDR's limit: 37,827 req/s

Figure 9: Response Latency Distribution under Flash Crowds

well and yield increasing throughput. This benefit comes from their ability to spread load across more servers. However, CDR and FDR behave better than Random because they not only adjust the server replica set on demand, but also maintain server locality for less popular URLs. In contrast, R-HRW, R-CHash, LR-HRW and LR-CHash suffer with more intensive requesters or attackers, since their fixed number of replicas for each URL cannot handle the high volume of requests for one URL. In the 10-URL case, the change in system capacity looks similar to the 1-URL case, except that due to more URLs being intensively requested or attacked, FDR, CDR and Random cannot sustain the same high throughput. We continue to investigate the effects of more attack URLs and other strategies.

Another possible DDoS attack scenario is to randomly select a wide range of URLs. In the case that these URLs are valid, the dynamic schemes "degenerate" into one server for each URL. This is the desirable behavior for this attack as it increases the cache hit rates for all the servers. In the event that the URLs are invalid, and the servers are actually reverse proxies (as is typically the case in a CDN), then these invalid URLs are forwarded to the server-of-origin, effectively overloading it. Servers must address this possibility by throttling the number of URL-misses they forward.

To summarize, under flash crowds or attacks, CDR and FDR sustain very high request volumes, making overloading the whole system significantly harder and thereby greatly improving the CDN system's overall robustness.

5.3 Proximity

The previous experiments focus on system capacity under different loads. We now compare the strategies that factor network closeness into server selection—Static (NPR-CHash), Static+Load (NPLR-CHash), and Dynamic (NP-FDR)—with their counterparts that ignore proximity. We test the 64-server cases in the same scenarios as in Section 5.1 and 5.2.

Req Rate	11,235 req/s				19,811 req/s				31,000 req/s				37,827 req/s			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
Random	2.37	0.64	8.57	5.29												
R-CHash	0.73	0.53	1.45	2.10	0.81	0.53	1.57	2.59								
R-HRW	0.73	0.52	1.45	2.11	0.76	0.52	1.51	2.51								
LR-CHash	0.62	0.45	1.15	1.70	0.67	0.45	1.23	2.42	0.96	0.52	1.86	3.55				
LR-HRW	0.63	0.45	1.18	1.80	0.67	0.46	1.26	2.65	1.07	0.53	2.19	3.52				
CDR	1.19	0.55	1.72	5.40	1.25	0.55	1.86	5.51	1.80	0.76	4.35	6.08	2.29	1.50	4.20	6.41
FDR	1.22	0.55	1.81	5.71	1.18	0.55	1.83	5.27	1.64	0.66	3.57	5.95	2.18	1.14	4.15	6.63
FDR-Global	0.91	0.55	1.66	4.09	0.90	0.53	1.60	4.59	0.98	0.54	1.74	5.08	1.20	0.56	1.99	5.53

Table 4: Response Latency of Different Strategies under Flash Crowds. μ — Mean, σ — Standard Deviation.

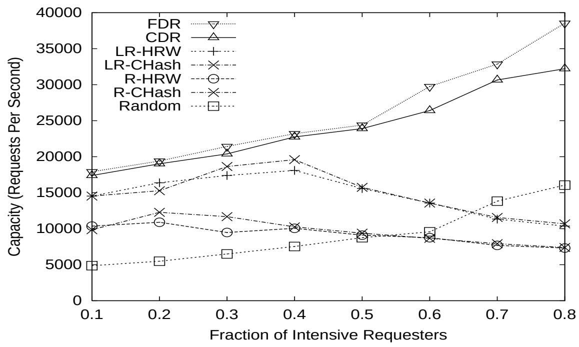


Figure 11: 1 Hot URL, 32 Servers, 1000 Clients

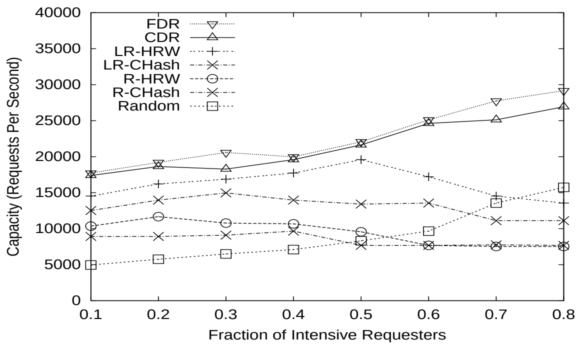


Figure 12: 10 Hot URL, 32 Servers, 1000 Clients

	System Capacity (reqs/sec)		
Category	Scheme	Normal	Flash Crowds
Static	*NPR-CHash*	14409	14409
	R-CHash	20411	19811
Static +Load	*NPLR-CHash*	24173	30090
	LR-CHash	25407	31000
Dynamic	*NP-FDR*	31000	34933
	FDR	33237	37827

Table 5: Proximity's Impact on Capacity

Table 5 shows the capacity numbers of these strategies under both normal load and flash crowds of 250 intensive requesters with 10 hot URLs. As we can see, adding network proximity into server selection slightly decreases systems capacity in the case of NPLR-CHash and NP-FDR. However, the throughput drop of NPR-CHash compared with R-CHash is considerably large. Part of reason is that in LR-CHash and FDR, server load information already conveys the distance of a server. However, in the R-CHash case, the redirector randomly choosing among all replicas causes the load to be evenly distributed, while NPR-CHash puts more burden on closer servers, resulting in unbalanced server load.

We further investigate the impact of network proximity on response latency. In Table 6 and 7, we show the latency statistics under both normal load and flash crowds. As before, we choose to show numbers at the capacity limits of Random, NPR-CHash, NPLR-CHash and NP-FDR. We can see that when servers are not loaded, all schemes with network proximity taken into consideration—NPR-CHash, NPLR-CHash and NP-FDR—yield better latency. When these schemes reach their limit, NPR-CHash and NP-FDR still demonstrate significant latency advantage over R-CHash and FDR, respectively.

Interestingly, NPLR-CHash underperforms LR-CHash in response latency at its limit of 24,173 req/s and 30,090 req/s. NPLR-CHash is basically LR-CHash using effective load. When all the servers are not loaded, it redirects more requests to nearby servers, thus shortening the response time. However, as the load increases, in order for a remote server to get a share of load, a local server has to be much more overloaded than the remote one, inversely proportional to their distance ratio. Unlike NP-FDR, there is no load threshold control in NPLR-CHash, so it is possible that some close servers get significantly more requests, resulting in slow processing and longer responses. In a summary, considering proximity may benefit latency, but it can also impact capacity. NP-FDR, however, achieves a good balance of both.

5.4 Other Factors

5.4.1 Heterogeneity

To determine the impact of network heterogeneity on our schemes, we explore the impact of non-uniform server network bandwidth. In our original setup, all first-mile links from the server have bandwidths of 100Mbps. We now randomly select some of the servers and re-

Req Rate	*9,300 req/s*				*14,409 req/s*				*24,173 req/s*				*31,000 req/s*			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
Random	3.95	1.78	11.32	6.99												
NPR-CHash	0.66	0.42	1.21	2.20	0.76	0.44	1.51	2.30								
R-CHash	0.79	0.53	1.46	2.67	0.82	0.56	1.63	2.50								
NPLR-CHash	0.57	0.36	0.93	2.00	0.68	0.39	1.33	2.34	1.34	0.55	2.63	4.73				
LR-CHash	0.68	0.44	1.17	2.50	0.71	0.48	1.43	2.19	1.04	0.50	1.95	3.44				
NP-FDR	0.70	0.50	1.42	1.63	0.67	0.49	1.33	1.56	0.80	0.49	1.55	2.82	1.08	0.53	1.96	3.54
FDR	1.10	0.52	1.48	5.49	1.25	0.54	1.71	5.87	1.60	0.57	2.10	6.84	1.88	0.59	3.72	7.25

Table 6: Proximity's Impact on Response Latency under Normal Load. μ — Mean, σ — Standard Deviation.

Req Rate	*11,235 req/s*				*14,409 req/s*				*30,090 req/s*				*34,933 req/s*			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
Random	2.37	0.64	8.57	5.29												
NPR-CHash	0.61	0.42	1.15	1.76	0.63	0.41	1.08	2.34								
R-CHash	0.73	0.53	1.45	2.10	0.73	0.52	1.38	2.50								
NPLR-CHash	0.53	0.36	0.90	1.75	0.55	0.35	0.91	2.29	1.29	0.61	2.65	3.94				
LR-CHash	0.62	0.45	1.15	1.70	0.64	0.44	1.13	2.56	0.90	0.49	1.73	3.44				
NP-FDR	0.70	0.50	1.45	1.68	0.66	0.45	1.34	1.63	0.81	0.47	1.64	2.55	0.99	0.51	1.92	3.26
FDR	1.22	0.55	1.81	5.71	1.07	0.54	1.67	5.47	1.60	0.66	3.49	5.90	1.84	0.78	4.15	6.31

Table 7: Proximity's Impact on Response Latency under Flash Crowds. μ — Mean, σ — Standard Deviation.

duce their link bandwidth by an order of magnitude, to 10Mbps. We want to test how different strategies respond to this heterogeneous environment. We pick representative schemes from each category: Random, R-CHash, LR-CHash and FDR and stress them under both normal load and flash crowd similar to network proximity case. Table 8 summarizes our findings on system capacities with 64 servers.

Redirection Schemes	Portion of Slower Links					
	Normal Load			Flash Crowds		
	0%	10%	30%	0%	10%	30%
Random	9300	8010	8010	11235	8449	8449
R-CHash	20411	7471	7471	19811	7110	7110
LR-CHash	25407	23697	19421	31000	26703	22547
FDR	33237	31000	25407	37827	34933	29496

Table 8: Capacity (reqs/sec) with Heterogeneous Server Bandwidth,

From the table we can see, under both normal load and flash crowds, Random and R-CHash are hurt badly because they are load oblivious and keep assigning requests to servers with slower links thereby overload them early. In contrast, LR-CHash and FDR only suffer slight performance downgrade. However, FDR still maintains advantage over LR-CHash, due to its dynamic expanding of server set for hot URLs.

5.4.2 Large File Effects

As we discussed at the end of section 5.1.3, the worse mean response times of dynamic schemes come from serving large files with a small server set. Our first attempt to remedy this situation is to handle the largest files specially. Analysis of our request trace indicates that 99% of the files are smaller than 530KB, so we use this value as a threshold to trigger special large file treatment. For these large files, there are two simple ways to redirect requests for them. One is to redirect these requests to a random server, which we call T-R (tail-random). The other is to redirect these requests to a least loaded member in a server set of fixed size (larger than one), which we call T-S (tail-static). Both of these approaches enlarge the server set serving large files. We repeat experiments of 64 server cases in Section 5.1 and 5.2 using these two new approaches, where T-S employs a 10-replica server set for large files in the distribution tail. Handling the tail specially yields slightly better capacity than standard CDR or FDR, but the latency improves significantly. Table 9 summarizes latency results under normal load. As we can see, the T-R and T-S versions of CDR and FDR usually generate better latency numbers than LR-CHash and LR-HRW. Results under flash crowds are similar. This confirms our assertion about large file effects.

6 Related Work and Discussion

Cluster Schemes: Approaches for request distribution in clusters [8, 12, 17] generally use a switch/router through which all requests for the cluster pass. As a result, they can use various forms of feedback and load information from servers in the cluster to improve system performance. In these environments, the delay between the redirector and the servers is minimal, so they can have tighter coordination [2] than in schemes like ours, which are developed for wide-area environments. We do, however, adapt the fine-grained server set accounting from the LARD/R approach [25] for our Fine Dynamic Replication approach.

Req Rate	9,300 req/s				18,478 req/s				25,407 req/s				32,582 req/s			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
LR-CHash	0.68	0.44	1.17	2.50	0.87	0.51	1.82	2.74	1.19	0.60	2.47	3.79				
LR-HRW	0.68	0.44	1.18	2.50	0.90	0.51	1.89	3.13	1.27	0.64	2.84	3.76				
CDR	1.16	0.52	1.47	5.96	1.35	0.55	1.75	6.63	1.86	0.63	4.49	6.62	2.37	1.12	5.19	7.21
CDR-T-R	0.78	0.52	1.43	2.77	0.76	0.52	1.40	2.80	1.05	0.57	1.90	3.06	1.58	0.94	3.01	3.55
CDR-T-S	0.74	0.52	1.43	2.17	0.72	0.52	1.38	2.44	1.01	0.56	1.93	2.96	1.53	0.68	3.69	4.18
FDR	1.10	0.52	1.48	5.49	1.35	0.54	1.64	6.70	1.87	0.62	3.49	6.78	2.22	0.87	4.88	7.12
FDR-T-R	0.78	0.52	1.43	2.77	0.75	0.52	1.40	2.82	1.01	0.57	1.87	2.98	1.39	0.77	2.82	3.68
FDR-T-S	0.74	0.52	1.43	2.17	0.72	0.52	1.37	2.55	0.98	0.56	1.84	2.95	1.41	0.63	2.88	3.88

Table 9: Response Latency with Special Large File Handling, Normal Load. μ — Mean, σ — Standard Deviation.

Distributed Servers: In the case of geographically distributed caches and servers, DNS-based systems can be used to obliviously spread load among a set of servers, as in the case of round-robin DNS [4], or it can be used to take advantage of geographically dispersed server replicas [6]. More active approaches [9, 14, 16] attempt to use load/latency information to improve overall performance. We are primarily focused on balancing load, locality and latency, meanwhile, we also demonstrate a feasible way to incorporate network proximity into server selection explicitly.

Web Caches: We have discussed proxy caches as one deployment vehicle for redirectors, and these platforms are also used in other content distribution schemes. The simplest approach, the static cache hierarchy [7], performs well in small environments but fails to scale to much larger populations [32]. Other schemes involve overlapping meshes [33] or networks of caches in a content distribution network [19], presumably including commercial CDNs such as Akamai.

DDoS Detection and Protection: DDoS attacks have become an increasingly serious problem on the Internet [22]. Researchers have recently developed techniques to identify the source of attacks using various traceback techniques, such as probabilistic packet marking [28] and SPIE [29]. These approaches are effective in detecting and confining attack traffic. With their success in deterring spoofing and suspicious traffic, attackers have to use more disguised attacks, for example by taking control of large number of slave hosts and instructing them to attack victims with legitimate requests. Our new redirection strategy is effective in providing protection against exactly such difficult-to-detect attacks.

Peer-to-Peer Networks: Peer-to-peer systems provide an alternative infrastructure for content distribution. Typical peer-to-peer systems involve a large number of participants acting as both clients and servers, and they have the responsibility of forwarding traffic on behalf of others. Given their very large scale and massive resources, peer-to-peer networks could provide a potential robust means of information dissemination or exchange. Many peer-to-peer systems, such as CAN [26], Chord [30], and Pastry [27] have been proposed and they can serve as a substrate to build other services. Most of these peer-to-peer networks use a distributed hash-based scheme to combine object location and request routing and are designed for extreme scalability up to hundreds of thousands of nodes and beyond. We also use a hash-based approach, but we are dealing one to two orders of magnitude fewer servers than the peers in these systems, and we expect relatively stable servers. As a result, much of the effort that peer-to-peer networks spend in discovery and membership issues is not needed for our work. Also, we require fewer intermediaries between the client and server, which may translate to lower latency and less aggregate network traffic.

7 Conclusions

This paper demonstrates that improved request redirection strategies can effectively improve CDN robustness by balancing locality, load and proximity. Detailed end-to-end simulations show that even when redirectors have imperfect information about server load, algorithms that dynamically adjust the number of servers selected for a given object, such as FDR, allow the system to support a 60-91% greater load than best published CDN systems. Moreover, this gain in capacity does not come at the expense of response time, which is essentially the same both when the system is under flash crowds and when operating under normal conditions.

These results demonstrate that the proposed algorithm results in a system with significantly greater capacity than published CDNs, which should improve the system's ability to handle legitimate flash crowds. The results also suggest a new strategy in defending against DDoS attacks: each server added to the system multiplicatively increases the number of resources an attacker must marshal in order to have a noticeable affect on the system.

Although we believe this paper identifies important trends, much work remains to be done. We have conducted the largest detailed simulations as current simulation environment allows. We also find that approximate load information works well. We expect our new algorithms scale to very large systems with thousands of servers, but it requires a lot more resources and time to evaluate. We would like to run simulations at an even larger scale, with faster, more powerful simulated servers. We would also like to experiment with more topologies such as those generated by power-law based topology generators, use more traces, real or synthetic (such as SPECweb99). Finally, we plan to deploy our new algorithms on a testbed and explore other implementation issues.

Acknowledgments

This research is supported in part by Compaq and DARPA contract F30602–00–2–0561. We thank our shepherd, David Wetherall, for his guidance and helpful inputs. We also thank our anonymous reviewers for their valuable comments on improving this paper.

8 REFERENCES

[1] Akamai. Akamai content delivery network. http://www.akamai.com.

[2] D. Andresen, T. Yang, V. Holmedahl, and O. Ibarra. Sweb: Towards a scalable world wide web server on multicomputers, 1996.

[3] A. Barbir, B. Cain, F. Douglis, M. Green, M. Hofmann, R. Nair, D. Potter, and O. Spatscheck. Known CN Request-Routing Mechanisms, Feb. 2002. Work in Progress, draft-ietf-cdi-known-request-routing-00.txt.

[4] T. Brisco. DNS support for load balancing. Request for Comments 1794, Rutgers University, New Brunswick, New Jersey, Apr. 1995.

[5] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies an d Systems (USITS)*, Monterey, CA, Dec. 1997.

[6] V. Cardellini, M. Colajanni, and P. Yu. Geographic load balancing for scalable distributed web systems. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Aug. 2000.

[7] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *USENIX Annual Technical Conference*, pages 153–164, 1996.

[8] J. Cohen, N. Phadnis, V. Valloppillil, and K. W. Ross. Cache array routing protocol v1.1. http://ds1.internic.net/internet-drafts/draft-vinod-carp-v1-01.txt, September 1997.

[9] M. Colajanni, P. S. Yu, and V. Cardellini. Dynamic load balancing in geographically distributed heterogeneous web servers. In *International Conference on Distributed Computing Systems*, pages 295–302, 1998.

[10] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[11] M. Crovella, M. Harchol-Balter, and C. D. Murta. Task assignment in a distributed system: Improving performance by unbalancing load (extended abstract). In *Measurement and Modeling of Computer Systems*, pages 268–269, 1998.

[12] O. Damani, P. Y. Chung, Y. Huang, C. M. R. Kintala, and Y. M. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. In *Proceedings of the Sixth International World-Wide Web Conference*, 1997.

[13] Digital Island. http://www.digitalisland.com.

[14] 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 *INFOCOM (2)*, pages 783–791, 1998.

[15] L. Garber. Technology news: Denial-of-service attacks rip the Internet. *Computer*, 33(4):12–17, Apr. 2000.

[16] J. D. Guyton and M. F. Schwartz. Locating nearby copies of replicated internet servers. In *SIGCOMM*, pages 288–298, 1995.

[17] G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. Technical report, IBM T.J. Watson Research Center, May 1997.

[18] K. L. Johnson, J. F. Carr, M. S. Day, and M. F. Kaashoek. The measured performance of content distribution networks. In *Proceedings of The 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[19] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. In *Proceedings of the Eighth International World-Wide Web Conference*, 1999.

[20] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[21] Mirror Image. http://www.mirror-image.com.

[22] D. Moore, G. Voelker, and S. Savage. Inferring internet denial of service activity. In *Proceedings of 2001 USENIX Security Symposium*, Aug. 2001.

[23] NS. (Network Simulator). http://www.isi.edu/nsnam/ns/.

[24] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX Annual Technical Conference*, June 1999.

[25] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. M. Nahum. Locality-aware request distribution in cluster-based network servers. In *Architectural Support for Programming Languages and Operating Systems*, pages 205–216, 1998.

[26] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM'01*, Aug. 2001.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.

[28] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In *Proceedings of the 2000 ACM SIGCOMM Conference*, Aug. 2000.

[29] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-based ip traceback. In *Proceedings of ACM SIGCOMM'01*, Aug. 2001.

[30] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM'01*, Aug. 2001.

[31] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, Feb. 1998.

[32] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. R. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *Symposium on Operating Systems Principles*, pages 16–31, 1999.

[33] L. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching. In *Proceedings of the 1997 NLANR Web Cache Workshop*, June 1997.