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
Technical Report TR-654-02
June, 2002

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 published state-of-the-art 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, 22]. CDNs deploy geographically-dispersed server surrogates and distribute client requests to an "appropriate" server based on various strategies.

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 wide-spread 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, that is, selects 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 state-of-the-art 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

We assume multiple server surrogates geographically dispersed in different locations as in typical CDNs; any of these servers can potentially serve any request on behalf of the original server. Where to place the servers and how to keep their contents up-to-date has been addressed by other CDN research [1, 13, 22]. Here, we make no explicit assumption about servers' strategic locations.

This paper focuses on *request redirectors*, which are middleware entities that assign client requests to 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 different mechanisms could be used to redirect requests [3]. For example, redirection could be implemented by augmenting DNS servers to return different server addresses to clients. This is essentially the implementation mechanism used by current CDNs. The granularity of DNS-based redi-

rection is usually at the level of a domain or site rather than a URL; with URL rewriting and augmenting the DNS infrastructure, this granularity can be refined. Redirection using DNS also has to deal with client-side caching of mapping entries.

Another possibility is to implement "logical redirectors" at the servers, for example by using HTTP redirects. Unfortunately, server-based redirections incur an additional round trip time, and redirecting servers can be vulnerable to being overloaded by the redirection task itself.

A third alternative is to distribute the redirection function across intermediate nodes of the network, such as routers or proxies. These routers/proxies could use one of two methods to redirect requests: (1) rewrite server addresses (both to redirect request packets and so that reply packets do not confuse the clients), or (2) send HTTP redirect messages back to the clients. In the first case, the router/proxy would need to be at a choke point between the client and server to ensure that it is able to edit all packets. In the second case, either the redirector would need to be on a choke point, or the client would have to cooperate by explicitly addressing the proxy (as with a classical, rather than transparent, proxy). Proxy-based redirectors can be placed either near clients (i.e., serving all the clients at a site), or near servers (i.e., serving as a front-end to a server cluster). In the former case, the client population that a given redirector serves is well-confined and easy to identify; the number of redirectors easily scales with the number of clients. Placing the redirectors near servers implies that redirectors can closely watch server load information.

To reduce the complexity of considering the various combinations outlined in this section, the evaluation experiments presented later in the paper assume redirectors are located at the edge of a client site, and that they receive the full list of cooperating servers through DNS or some other out-of-band communication. The whole request-reply session between a client and a server will go through a redirector through packet rewriting. From these communications, the redirectors passively learn approximate server load information. Redirectors are independent of each other and do not rely on a centralized approach to synchronize. These assumptions—in particular, the imperfect information about server load—do not have a significant impact on the results. They do let us focus more on the redirection strategies.

2.2 Hashing Schemes

Since geographically-distributed redirector placement is desirable, we cannot easily adapt the schemes used in LAN-oriented approaches [17, 26] to redirectors. Those approaches generally use information about the instantaneous state of the entire system. Instead, several of the strategies discussed below use some form of hashing to deterministically map URLs into a small range of values. The primary benefit of this approach is that no inter-redirector communication is required to achieve coordinated operation; no matter which redirector receives a URL, the hashing process produces the same output. Another benefit of hashing is that the range of resulting hash values can be controlled, to trade off precision for the amount of memory consumed by bookkeeping. A good hash function also provides a balanced partition of URL space.

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 – This is the "classic" hashing approach, but is not suitable for this environment. In this approach, the URL is hashed modulo the number of cooperating servers. While this approach is computationally efficient, its unsuitability stems from its behavior when the server set changes. When this occurs, the modulo calculation will result in a diminishing fraction of the documents keeping their same server assignments. While we do not expect frequent changes in the set of servers, the fact that addition of new servers into the set will cause massive reassignment is undesirable.

Consistent Hashing [19, 20] – In this approach, the URL is mapped to a value on a unit circle, as are the cooperating servers. The URL is assigned to the server that lies closest on the circle to its hash value. While this scheme requires finding the closest match, this lookup can be made in constant time by using space equal to the number of possible values on the circle. Otherwise, it can be computed in time logarithmic with the number of servers. If a 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 [32] – This approach is the basis for CARP [8], and consists of hashing each URL with each server, and then 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. The benefit of this approach compared to consistent hashing is that server order is different for each URL, so if one server fails, its load is distributed evenly among the other machines. The drawback to this approach is more computation is required to generate the list, $O(NlogN)$ where N is the number of servers. These lists can be cached, using $O(N \times num_hash_values)$ space, and even this can be reduced by keeping only the top few list entries for each hash value.

The strategies in the next section use the Consistent Hashing (CHash) and Highest Random Weight (HRW) schemes as one component of the request redirection process.

3 Strategies

This section discusses the design space for the request distribution strategies, and introduces two new algorithms that factor both load and locality into their decision and another new one that considers all aspects of network proximity, server locality and load.

3.1 Random

We use the random assignment policy as a baseline to determine a reasonable level of performance. In this policy, each request is randomly sent to one of the cooperating servers. We expect this approach to scale with the number of servers in the set, and since its request distribution has no pattern, it is unlikely to exhibit any pathological behavior. It has the drawback that the working set of each server increases with the number of cooperating servers. Since server performance increases as a higher fraction of requests that can be served from main memory, this approach is at a disadvantage versus schemes that exploit URL locality.

3.2 Replicated Consistent Hashing (R-CHash)

In the Replicated Consistent Hashing strategy, each URL is assigned to a set of replicated servers. The URL is hashed to a point on the unit circle, and the replicas are evenly spaced starting from this original point. On each request, the redirector calculates the set of replicas for the URL and randomly assigns the request to one of the replicas. The number of replicas is fixed, but is configurable. This strategy is intended to model the mechanism used in published state-of-the-art content distribution networks, and is virtually identical to the scheme described in [19] and [20].

3.3 Replicated Highest Random Weight (R-HRW)

The Replicated Highest Random Weight strategy is the counterpart to Replicated Consistent Hashing, but with a different underlying 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 ordered list of servers for each URL is determined using the Highest Random Weight approach, and then the top N servers are treated as possible targets for the URL. On each request, the redirector randomly picks one member of the appropriate set and sends the request to that server. The reason we expect different behavior between this scheme and the previous approach is because this scheme is less likely to have two URLs generate the same set of replicas. 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.

3.4 Coarse Dynamic Replication (CDR)

This scheme stems from the belief that the number of replicas used by redirectors to serve one URL should be dynamically adjusted in response to server load and demand for that URL. Like Replicated Highest Random Weight, HRW hashing is used to generate an ordered list of servers, but rather than using a fixed number of replicas, based on coarse-grained server load information, the first "available" server on the list is chosen as the target server for the request. By reducing unnecessary replication, the working set of each server is reduced, resulting in better file system caching behavior.

Figure 1 shows how a request redirector picks the destination server for each request. Notice that 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 instead 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;
  for each 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.

3.5 Fine Dynamic Replication (FDR)

The Fine Dynamic Replication (FDR) scheme addresses the problem of unnecessary replication in CDR by keeping information regarding the popularity of URLs 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 or throughput. This scheme uses the HRW hashing as in CDR, but additionally, each URL is also hashed to a smaller identifier, in the range of thousands to millions of values. With each of these identifiers is kept the number of servers to use in the sorted list, allowing URLs to be replicated across a number of servers based on their popularity.

The introduction of finer-grained bookkeeping is an attempt to counter the possibility of a "ripple effect" in Coarse Dynamic Replication, 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 effects 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 > changeThresh)
      walk_entry.length −−;
  } else {
     foreach remaining 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, this scheme keeps an auxiliary structure at each redirector that maps from the URL to a "walk length," indicating how many servers in the HRW list should be used for this URL. Using a minimum value of one for entries in this table provides minimal replication for most URLs, and a higher minimum can be used to try to 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.

The rationale behind this approach is to try to keep popular URLs from overloading servers and displacing unpopular URLs in the process. The size of the auxiliary structure is contained by hashing the URL into a range in the thousands to millions. While this hashing will produce some imprecision, the collisions will only cause some small number of 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, the two 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 director 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. In the future, we can 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.

3.6 FDR with Network Proximity (FDR-NP)

Many commercial CDNs start server selection with network proximity matching. For an instance, [19] indicates that CDN's hierarchical authoritative DNS servers can map 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. Other research [18] shows that in practice, CDNs succeed not by always choosing the "optimal" server, but by avoiding notably bad servers.

We introduce a new strategy, called FDR with Network Proximity (FDR-NP), which also explicitly factors network proximity into server selection, but in a more unified fashion. Our redirector measures servers' geographical/topological location information through *ping*, *traceroute* or similiar mechanisms; and different from FDR, when evaulating servers' status, it uses "effective load " instead of raw server load information to choose a proper server.

$$min_distance = MIN\{distance_i, i = 1...n\} \quad (1)$$

$$std_distance_i = distance_i / min_distance \quad (2)$$

$$effect_load_i = load_i \times std_distance_i \quad (3)$$

In our evaluation, we define *effective load* in the above equations. Redirectors first calculate *minimum distance* among all servers using gathered redirector-server distance information. Here, distance can take the form of either round trip time (RTT) or routing hops; we choose to use RTT. Then

all raw distances are normalized to *standard distances* using this *minimum distance*. *Effective load* is thus defined as the product of raw load numbers and *standard distance*. The rationale behind this is that, although the server load metric piggybacks some distance information implicitly, since replies from remote servers take longer time to finish, plain FDR actually tries to use servers with various distances equally by only considering load. FDR-NP, on the other hand, takes server distances into consideration explicitly. A remote server with the same raw load as a nearby server should be regarded as "effectively" more loaded, since redirecting a new request to that remote server will result in long network journeys and resources being potentially held longer. The tradeoff between server's load and proximity is thus achieved through assigning each server's load a weight inversely proportional to its distance. Using minimum distance to normalize all distances conservatively ensures that even the closest server is not significantly overloaded.

Although we currently calculate *effective load* this way, it is not the only possibility. In fact, *effective load* can take other dynamic load/proximity metrics into account, for example network congestion status through real time measurement, thereby reflecting instantaneous load conditions.

3.7 Categories and Other Variants

To summarize, we classify the above strategies along three different dimensions. We also introduce a few new variants.

Replication. Based on how replication is achieved, there are three types of strategies: purely random replication—Rand, static replication (R-CHash and R-HRW), and dynamic replication (CDR, FDR and FDR-NP).

Server Load Awareness. Strategies either select servers in a load oblivious way (Rand, R-CHash and R-HRW); or consider server load deliberately (CDR, FDR and FDR-NP). Although it is not clear how fine grained load information is used in current CDN systems, we try to approximate their best behaviors by introducing two variants of static replication strategies, Load-aware R-CHash (LR-CHash) and Load-aware R-HRW (LR-HRW), into our evaluation. In short, instead of randomly choosing a server from members of the fixed size of server set for each URL as in R-CHash and R-HRW, these two new strategies pick the least loaded server from the server set.

Network Distance Awareness. Except for FDR-NP, most of the above strategies treat each server surrogate as "equally" distant from clients. This may not have much impact on the system's aggregate capacity, but will affect response time. As mentioned earlier, current CDNs rely on DNS region mapping to achieve network closeness. However, the proximity obtained that way is usually at the granularity of an Autonomous System, or ISP, and the actual topological or geographical distance between server and client could still be substantial [21]. Factoring fine-grained network proximity into server selection can still be beneficial. To approximate this improved proximity matching at the redirectors, we derive a variant from R-CHash, called NP-CHash, which assigns requests such that each surrogate in the fixed-size server set of an URL will get a share of total requests for that URL inversely proportional to the surrogate's distance from the redirector. Similarly, NPLR-CHash is a variant of LR-CHash using *effective load* as FDR-NP.

Not all possible combinations of these three factors yield practical strategies. Rather than exhaustively considering all possibilities, we choose to concentrate on those strategies that seem most promising or illustrative.

4 Evaluation Methodology

The goal of this work is to examine how different strategies respond under different loads and especially how robust they are in the face of flash crowds and DDoS attacks. Although DDoS attacks take many different forms, the most-difficult-to-detect versions generate legitimate traffic, virtually indistinguishable from flash crowds. So we use high request volumes to stress the system.

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 we are interested in, both the network and the server are important.

To remedy this situation, we develop 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 [24] 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 [26], 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 simulation model of Logsim is shown in Figure 3. Each server node consists of a CPU and locally attached disk(s), with a separate queue 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 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.

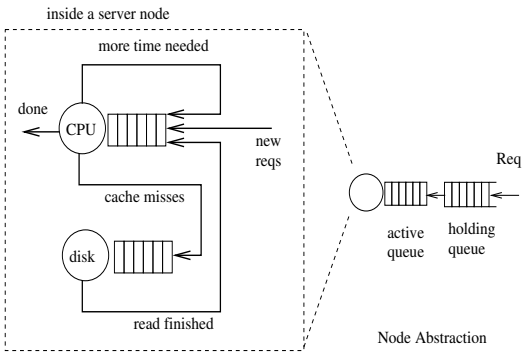


Figure 3: Logsim Simulator

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 side, all packets are stored and forwarded, as in a real network, and we use two-way TCP. We use static routing within NS-2, although it would be interesting to run simulation under dynamic routing.

On the Logsim side, the costs for the basic request processing were derived by performing measurements on a 300 MHz Pentium II machine running FreeBSD 2.2.5 and the Flash web server [25]. Connection establishment and tear-down costs are set at 145μs, while transmit processing incurs 40μs per 512 bytes. Using these numbers, an 8KByte 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 4KBytes. 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. This is a server that is somewhat 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. Although we could use a topology generation tool to get a power-law topology, we instead choose to slightly modify the NSFNET backbone network T3 topology, as shown in Figure 4. The reason is not only because this topology is easy to manage, but also because to a large extent, it resembles an ISP's network, or a simplified backbone. 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[1]; small circles with "C" stands for client hosts; and shaded circles with "S" are the cooperating servers. In the particular configuration shown in the figure, we put 64 cooperating 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, which ends up with a topology of nearly 1,100 nodes. The redirector algorithms run on the regional routers that sit in front of the clients.

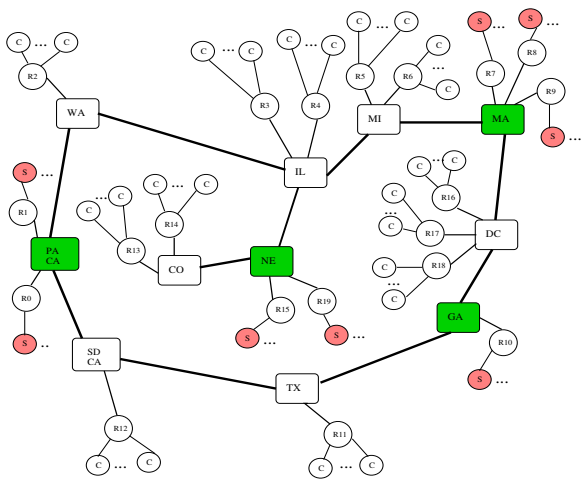


Figure 4: Network Topology

The latencies of servers to regional routers are set randomly between 1ms to 3ms; those between clients to regional routers are between 5 ms and 20ms; those 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.

4.3 Workload and Stability

We drive our simulations using 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 [26].

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

Since populating cooperating servers with documents and maintaining content consistency are not our primary focus, how close these logs are to real life user accesses to web sites is not the most important issue. In fact, we only use the name and size of the documents, and ignore the timing information contained in the trace.

We carry out stress tests as follows. Each client starts requesting documents at a certain rate from the very beginning of the test, with the documents' information drawn from the trace. The request redirector decides where the request should be sent and then the client opens a new connection to that server for this request, mimicking instant packet rewriting. Clients keep sending requests at a rate-defined time interval without waiting for previous ones to finish. We start with a low aggregate rate—with all clients contributing equally to this rate—and then increase the rate by 1% every simulated 6 seconds. In doing so, we warm the server memory caches at the beginning, and drive servers to their limits gradually over time. The parallelism parameter of Logsim is set to 512, allowing each server to handle at most 512 simultaneous requests. We keep increasing the offered load in this way until the servers become saturated, as defined below.

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; flash crowds and DDoS attacks can happen without pre-knowledge and present less obvious patterns.

It is not obvious how to decide when the system reaches its maximum capacity. What we want to determine is the state where servers are stable in delivering responses, yet another small increase in the offered load could result in server overload or instability. Contrary to real servers, our simulated servers do not crash. This complicates the decision on server stability.

The length of a server's request queue (active + holding) seems to be a good indication of how busy a server is. However, queue length may grow quickly due to a short burst of requests, which we do not want to misinterpret as server failure. On the other hand, if the server has a persistently long queue that significantly exceeds the server's parallelism, chances are that server will not catch up with offered load soon, and under an ever-increasing request volume, the request queue can only become longer and the server will become unstable.

We conducted a series of simple tests in which we monitor the request queue of each server at 15 seconds intervals, allowing the queue to grow infinitely. We check the throughput of the system against queue length in units of the servers' parallelism settings. Empirically, allowing longer queues yields higher throughput, but when queue length exceeds 4 or 5 times the parallelism parameter, throughput flattens out and then drops. Thus, we define the threshold for a server *failure* to be when the request queue length exceeds five times the parallelism 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 choose to compare 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.

5 Results

This section evaluates how the different strategies perform, both under normal conditions and under flash crowds or DDoS attacks. The strategies we examine include: Random, R-CHash, R-HRW, CDR, FDR, FDR-NP and FDR-Ideal. The last one, FDR-Ideal, is a reference strategy where all redirectors have perfect knowledge of the load at all servers. We also consider a couple of variants: LR-CHash, LR-HRW, NP-CHash and NPLR-CHash.

5.1 Normal Workload

Before evaluating these strategies under flash crowd or 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. We place 64 servers behind 8 regional routers and 1000 clients behind the remaining 12 regional routers. Request redirectors are placed at the clients' regional routers.

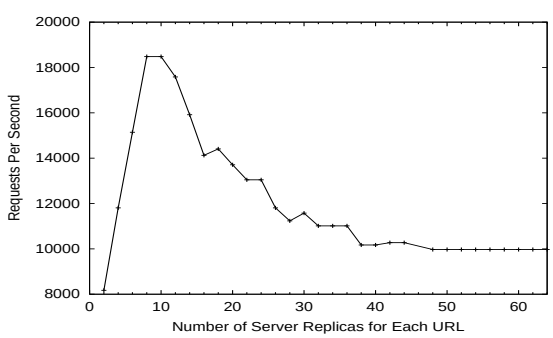


Figure 5: Finding Optimal Number of Replicas for R-HRW in 64 Server Case

5.1.1 Optimal Static Replication

The two static replication schemes, R-CHash and R-HRW, and their variants use a configurable (but fixed) number of replicas, and this parameter's value can significantly influence their performance. In the extreme case with only one target server per URL, requests for a very popular URL can

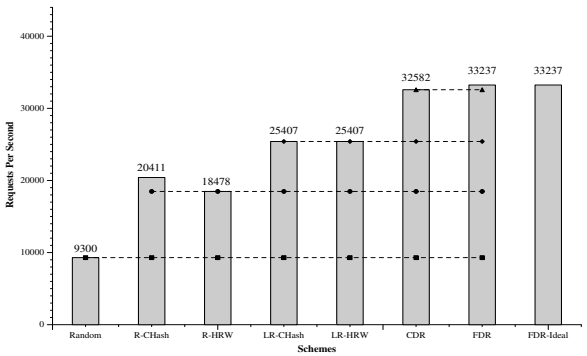


Figure 6: Capacity Comparison under Normal Load

Utilization	*CPU (%)*		*DISK (%)*	
Scheme	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-Ideal	91.93	11.81	17.60	15.43

Table 1: Server Resource Utilization at Overload

easily overwhelm a single server. To determine an appropriate value, we varied this parameter between 2 and 64 replicas. The system capacity results of these tests for R-HRW are shown in Figure 5; the results for R-CHash are similar.

As seen from this figure, increasing the number of replicas per URL initially helps to improve the system's throughput because the load is more evenly distributed. But this increase stops after the number of replicas surpasses a threshold, and then throughput starts decreasing due to lessening locality. At the extreme, using 64 replicas to serve each URL, R-HRW and R-CHash degenerate to the Random strategy. 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 are shown in Figure 6. Here we do not plot all the strategies and variants, but focus on those impacting throughput substantially. To simplify the discussion, we sometimes group the algorithms into four categories: Random, static replication (R-CHash and R-HRW), static load replication (LR-CHash and LR-HRW) and dynamic replication (CDR, FDR, FDR-Ideal). 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.

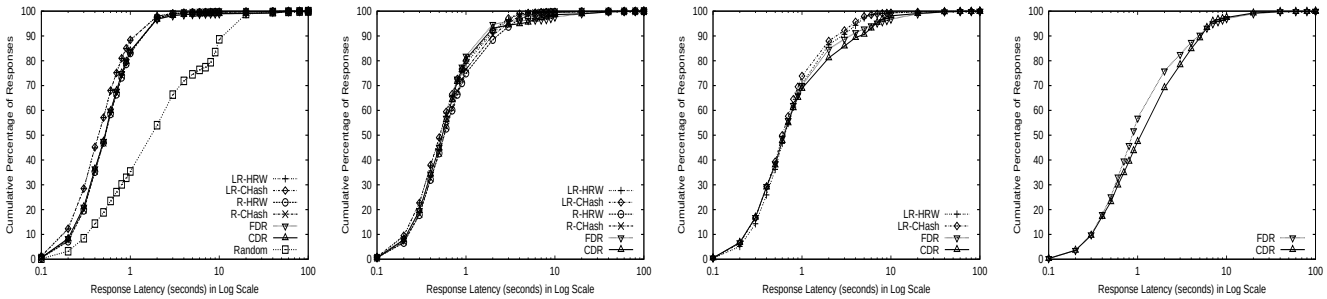
The difference between Random and the static approaches stems from the locality benefits of the hashing in the static schemes. By partitioning the working set, more documents are served from memory by the servers. Note, however, that absolute minimal replication can be detrimental, and the throughput for only two replicas in Figure 5 is actually lower than the throughput for Random. The difference in throughput between R-CHash and R-HRW is 11% in our simulation. It appears that R-CHash spreads load somewhat more evenly since two URLs that have one replica in common will have all replicas in common. As a result, no single server gets overloaded before others. This difference should not be overly emphasized, because changes in the number of servers or workload can cause their relative ordering to change. Considering load helps 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-Ideal 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-RHW 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 also becomes the disk and the CPU is busy roughly half the time. 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 1 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 roughly half of 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.*



(a) Random limit: 9,300 req/s (b) R-HRW limit: 18,478 req/s (c) LR-HRW limit: 25,407 req/s (d) CDR limit: 32,582 req/s

Figure 7: Response Latency Distribution under Normal Load

Req Rate	*9,300 req/s*				*18,478 req/s*				*25,407 req/s*				*32,582 req/s*			
Latency	Mean	Median	90%	Stddev	Mean	Median	90%	Stddev	Mean	Median	90%	Stddev	Mean	Median	90%	Stddev
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-ideal	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 2: Response Latency of Different Strategies under Normal Load

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 6 and analyze the latency of all of the schemes whenever one of the schemes reaches its performance limit. For schemes with similar performance, 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 about the distribution.

Figure 7 shows the latency cumulative distribution plots at four request rates: the maximums for Random, R-HRW, LR-HRW, and CDR. 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 2 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 7a. 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 7d, 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-Ideal has better numbers in all cases than CDR and FDR, due to its perfect knowledge on 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 what occurs in the static scheme. 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].

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 8. They all display near-linear scalability, implying all of them are reasonably good strategies when the system becomes larger. The only noticeably sub-linear increase seen

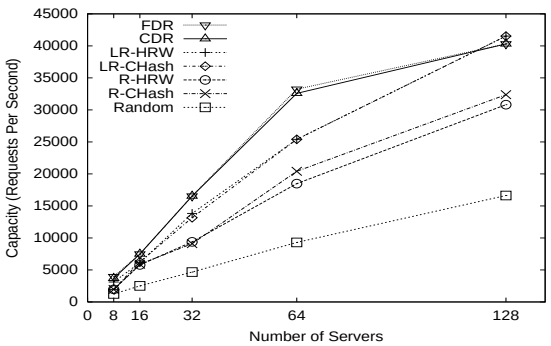


Figure 8: System Scalability under Normal Load

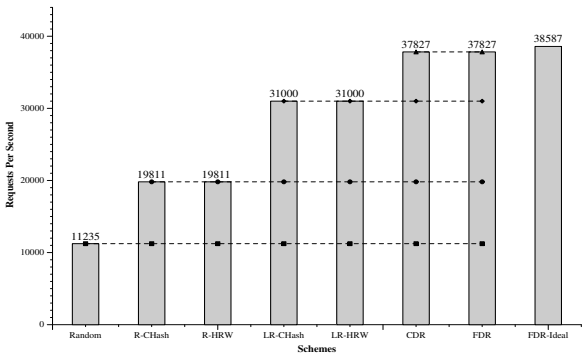


Figure 9: Capacity Comparison Under Flash Crowds

is when dynamic schemes, CDR and FDR increases from 64 servers to 128 servers. However, closer inspection of the simulation shows that 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. Rerunning the simulation with doubled bandwidth on the router-to-backbone links once again shows linear scalability at 128 servers.

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 a 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 preselected URLs with an average size of about 6KB.

5.2.1 System Capacity

Figure 9 depicts system capacity under flash crowd with a set of 10 URLs. In general, it exhibits similar trends as the no-attack case shown in Figure 6. 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%.

Second, most of the absolute throughput numbers are larger compared to the no-attack case. One reason is that at least 25% of the traffic is now concentrated on 10 URLs, which may increase the servers' hit rates. Moreover, these attack URLs are relatively small, with an average size of 6KB, which results in more responses delivered using the same resources.

5.2.2 Response Latency

The cumulative distribution form of response latencies for all seven algorithms, when under attack, are shown in Figure 10. Also, the full statistics for all seven algorithms and FDR-Ideal are given in Table 3. 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.

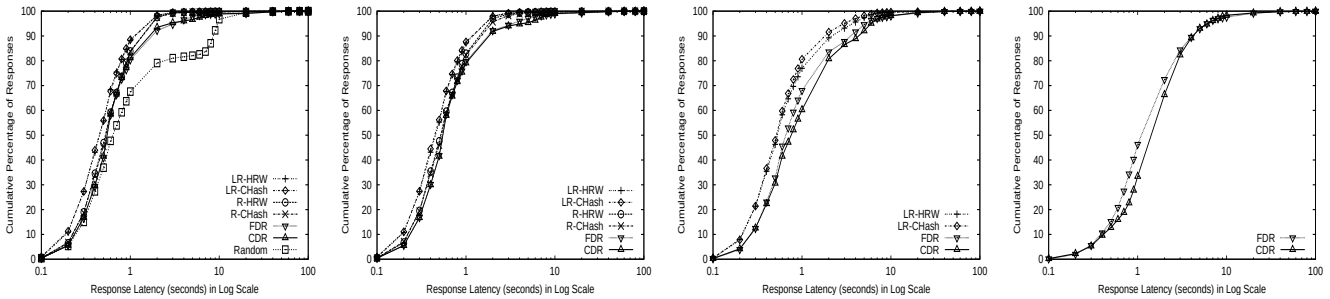
As we explained earlier, CDR and FDR adjust the server replica set in response to request volume. The number of replicas that serve attack URLs increases as the attack ramps up, which may adversely affect serving non-attack URLs. However, the differences in the mean, median, and 90-percentile are not 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 11, all strategies scale linearly with the number of servers, with the exception of CDR and FDR's 128 server case. Again, the network link becomes a bottleneck in this case, which can be solved either by greater bandwidth provisioning or placing fewer servers behind each pipe and instead spreading them across more locations.

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. Here, we show the results for three representative strategies: Random, R-HRW and CDR. Since it is impractical to exhaust all possible combinations, we choose two classes of flash crowds. One class has a single



(a) Random limit: 11,235 req/s (b) R-HRW limit: 19,811 req/s (c) LR-HRW limit: 31,000 req/s (d) CDR limit: 37,827 req/s

Figure 10: Response Latency Distribution under Flash Crowds

Req Rate	*11,235 req/s*				*19,811 req/s*				*31,000 req/s*				*37,827 req/s*			
Latency	Mean	Median	90%	Stddev	Mean	Median	90%	Stddev	Mean	Median	90%	Stddev	Mean	Median	90%	Stddev
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-ideal	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 3: Response Latency of Different Strategies under Flash Crowds

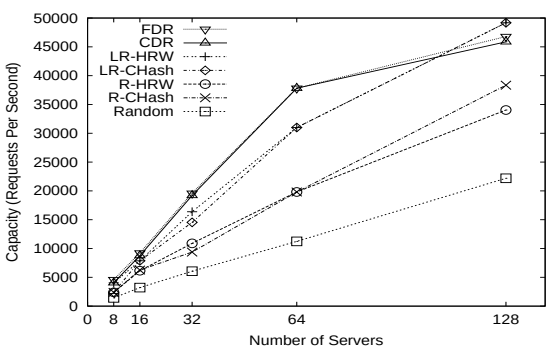


Figure 11: System Scalability under Flash Crowds

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 are shown in Figures 12 and 13, 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 and CDR adapt to this change well and yield increasing throughput. This benefit comes from their ability to spread load across more servers. However, CDR behaves better than Random because it not only adjusts the server replica set on demand, but it also maintains server locality for less popular URLs. In contrast, R-HRW suffers with more intensive requesters or attackers, since its 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, 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 will "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 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 bringing it down. Servers must address this possibility by throttling the number of URL-misses they forward.

To summarize, under flash crowds or attacks, our CDR and FDR sustain very high request volumes. This makes overloading the whole system significantly harder. Considering CDR's maximum capacity of 37,827 req/s, each of the 1000 clients we use in the simulation has to maintain a request rate of roughly 37 req/s, which is quite a high number for a typical browsing user. In this regard, our clients are more aggressive than a single attacker might be, and should be viewed as a small group of clients.

5.3 Network Proximity and Heterogeneity

This section takes the topological distance between servers and clients and network bandwidth into consideration.

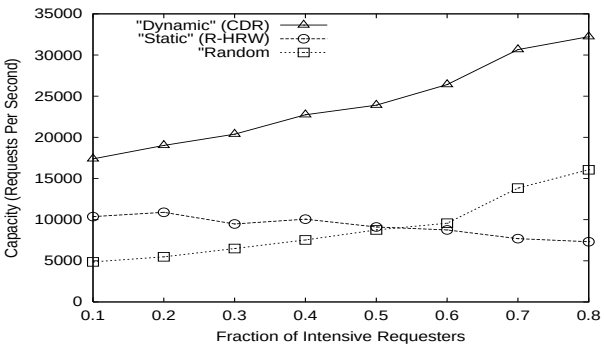


Figure 12: 1 Hot URL, 32 Servers, 1000 Clients

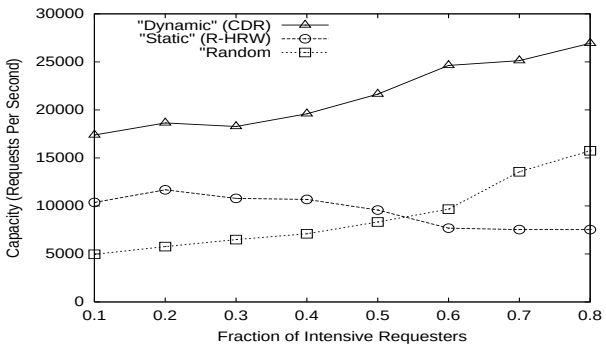


Figure 13: 10 Hot URL, 32 Servers, 1000 Clients

5.3.1 Proximity

Previous experiments focus on system's capacity under different loads. Here, we will compare those strategies that factor network closeness into server selection, namely static (NP-CHash), load-static (NPLR-CHash), and dynamic (FDR-NP) with their counterparts that ignore proximity. We test 64 server case in the same set up as that of Section 5.2.

Static	*NP-CHash*	14409
	R-CHash	19811
Load Static	*NPLR-CHash*	30090
	LR-CHash	31000
Dynamic	*FDR-NP*	35280
	FDR	37827

Table 4: Proximity Impact on Capacity under Flash Crowds

Table 4 shows the capacity numbers of these strategies under flash crowds of 250 intensive requesters with 10 hot URLs. As we can see, adding network proximity into server selection slightly decrease systems capacity in the case of NPLR-CHash and FDR-NP. However throughput drop in NP-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, redirector round robin among all servers causes the load to be evenly distributed, while NP-CHash puts more burden on closer servers, resulting in unbalanced server load.

We further investigate the impact of network proximity on response latency. In Figure 14, instead of plotting relevant strategies together all the way, we use a different format. We first plot all relevant strategies against the Random scheme at Random's limit of 11,235 req/s, then we plot the six schemes pair-wise at the lower capacity in each category. From this graph we can see that when servers are not loaded, all schemes with network proximity taken into consideration—NP-CHash, NPLR-CHash and FDR-NP—yield much better latency. When these schemes reach their limit, NP-CHash and FDR-NP still demonstrate significant latency advantage over R-CHash and FDR, respectively.

To our surprise, NPLR-CHash underperforms LR-CHash at its limit of 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. Since unlike FDR-NP, there's is no load threshold control in NPLR-CHash, it is possible that some close servers get a lot more requests, resulting in slow processing and longer response. In a summary, considering proximity might bring latency benefit, but it could also slightly hurt the capacity. FDR-NP, however, achieves a good balance.

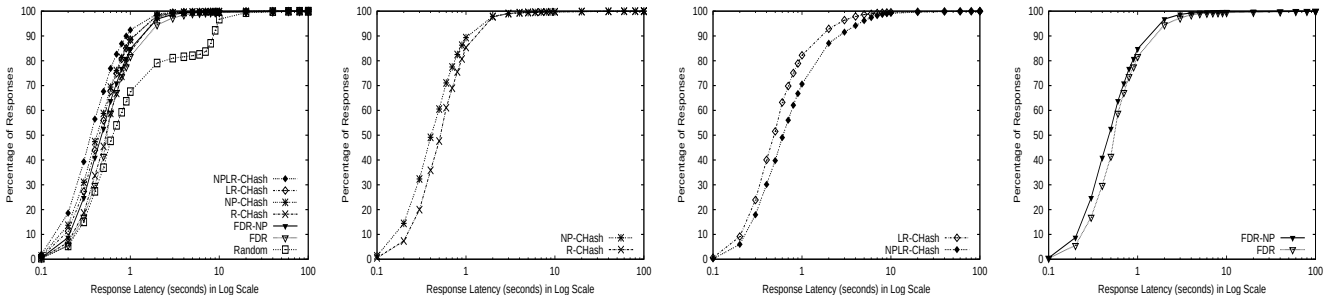
5.3.2 Heterogeneity

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 reduce their link bandwidth by an order of magnitude, to 10Mbps. We want to test how different strategies respond to this heterogeneous environment. We pick some representative schemes: Random, R-CHash, LR-CHash and FDR. Table 5 summarizes our findings for the flash crowds test similar to Section 5.3.1.

Scheme	*Portion of Slower Links*		
	0%	10%	30%
Random	11235	8449	8449
R-CHash	19811	7110	7110
LR-CHash	31000	26703	22547
FDR	37827	34244	28065

Table 5: Capacity under Flash Crowds with Heterogeneous Server Bandwidth

From the table we can see, with 10% server links 10 times slower than others, FDR's capacity downgrades by 9.5% and LR-CHash decreases by 14%; when 30% of the links are slower, the decrease changes to 25.8% and 27.3%, respectively. However, 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.



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

(b) NP-CHash limit: 14,409 req/s

(c) NPLR-CHash limit: 30,090 req/s

(d) FDR-NP limit: 35,280 req/s

Figure 14: Proximity Impact on Response Latency

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 the performance of request distribution. In these environment, 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 [26] for our Fine Dynamic Replication approach.

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. Our current techniques are primarily focused on balancing load, locality and latency, 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 [33]. Other schemes involve overlapping meshes [34] 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 [23]. Researchers have recently developed techniques to identify the source of attacks using various traceback techniques, such as probabilistic packet marking [29] and SPIE [30]. 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 to provide 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 [27], Chord [31], and Pastry [28] 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 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.

7 Conclusions

This paper demonstrates that request redirection can effectively improve CDN's robustness. The key is to balance locality, load and proximity. Detailed end-to-end simulations show that even when the redirectors have imperfect information about server load, an algorithm that dynamically adjusts the number of servers selected for a given object allows the system to support a 60-91% greater load than published state-of-the-art 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.

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] Z. M. Mao, C. D. Cranor, F. Douglis, M. Rabinovich, O. Spatscheck, and J. Wang. A Precise and Efficient Evaluation of the Proximity between Web Clients and their Local DNS Servers. In *USENIX Annual Technical Conference*, 2002.

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

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

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

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

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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