Dynamic, Scalable, and Efficient Content Replication Techniques

Yan Chen

1 Introduction

Exponential growth in processor performance, storage capacity, and network bandwidth is changing our view of computing. Our focus has shifted away from centralized, hand-choreographed systems to global-scale, distributed, self-organizing complexes – composed of thousands or millions of elements. Unfortunately, large pervasive systems are likely to have frequent component failures and be easily partitioned by slow or failed network links. Thus, use of local resources is extremely important – both for performance *and* availability. Further, pervasive streaming applications must tune their communication structure to avoid excess resource usage. To achieve both local access *and* efficient communication, we require flexibility in the placement of data replicas and multicast nodes.

One approach for achieving this flexibility while retaining strong properties of the data is to partition the system into two tiers of replicas [18] – a small, durable *primary* tier and a large, soft-state, *second-tier*. The primary tier could represent a Web server (for Web content delivery), the Byzantine inner ring of a storage system [6, 29], or a streaming media provider. The important aspect of the primary tier is that it must hold the most up-to-date copy of data and be responsible for serializing and committing updates. We will treat the primary tier as a black box, called simply "the data source". The second-tier becomes soft-state and will be the focus of this chapter. Examples of second-tiers include content-delivery networks (CDNs), file system caches, or web proxy caches.

Because second-tier replicas (or just "replicas") are soft-state, we can dynamically grow and shrink their numbers to meet constraints of the system. We may, for instance, wish to achieve a Quality of Service (QoS) guarantee that bounds the maximum network latency between each client and replicas of the data that it is accessing. Since replicas consume resources, we will seek to generate as few repli-

Department of EECS, Northwestern University, Evanston IL, USA, e-mail: ychen@northwestern.edu.

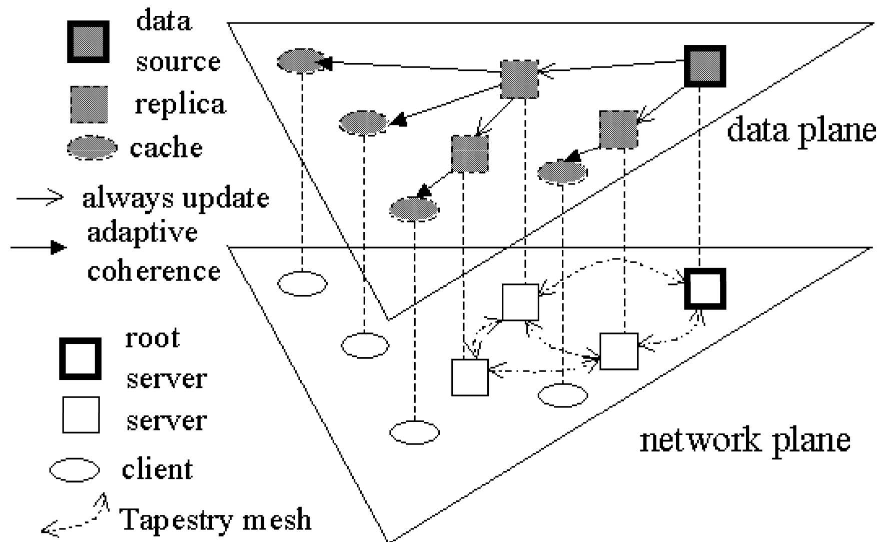

Fig. 1 Architecture of a SCAN system

cas as possible to meet this constraint. As a consequence, popular data items may warrant hundreds or thousands of replicas, while unpopular items may require no replicas.

One difficult aspect of unconstrained replication is ensuring that content does not become stale. Slightly relaxed consistency, such as in the Web [20], OceanStore [29], or Coda [26], allows delay between the commitment of updates at the data source and the propagation of updates to replicas. None-the-less, update propagation must still occur in a timely manner. The potentially large number of replicas rules out direct, point-to-point delivery of updates to replicas. In fact, the extremely fluid nature of the second tier suggests a need to self-organize replicas into a multicast tree; we call such a tree a *dissemination tree* (d-tree). Since interior nodes must forward updates to child nodes, we will seek to control the *load* placed on such nodes by restricting the fanout of the tree.

The challenge of second-tier replication is to provide good QoS to clients while retaining *efficient* and *balanced* resource consumption of the underlying infrastructure. To tackle this challenge, we propose a self-organizing soft-state replication system called SCAN: the *Scalable Content Access Network*. Fig. 1 illustrates a SCAN system. There are two classes of physical nodes shown in the network-plane of this diagram: *SCAN servers* (squares) and *clients* (circles). We assume that SCAN servers are placed in Internet Data Centers (IDC) of major ISPs with good connectivity to the backbone. Each SCAN server may contain replicas for a variety of data items. One novel aspect of the SCAN system is that it assumes SCAN servers participate in a distributed routing and location (DOLR) system, called Tapestry [22]. Tapestry permits clients to locate nearby replicas without global communication.

There are three types of data illustrated in Fig. 1: Data *sources* and *replicas* are the primary topic of this chapter and reside on SCAN servers. *Caches* are the images

of data that reside on clients and are beyond our scope[1] Our goal is to translate client requests for data into replica management activities. We make the following contributions:

- We provide algorithms that dynamically place a minimal number of replicas while meeting client QoS and server capacity constraints.
- We self-organize these replicas into d-tree with small delay and bandwidth consumption for update dissemination.

The important intuition here is that the presence of the DOLR system enables simultaneous placement of replicas and construction of a dissemination tree without contacting the data source. As a result, each node in a d-tree must maintain state only for its parent and direct children.

The rest of this chapter is organized as follows. We first examine the related work in Section 2, then formulate the replica placement problem in Section 3. Next, we present our algorithms in Section 4, evaluation methodology in Section 5 and evaluation results in Section 6.

2 Previous Work

In this section, we first survey existing content distribution systems, namely Web caching (Section 2.1), uncooperative pull-based CDNs (Section 2.2), and cooperative push-based CDNs (Section 2.3). We compare these systems with SCAN, and summarize this in Table 1. Then we discuss the previous work on three building blocks of CDN: object location services (Section 2.4), and multicast techniques for update dissemination (Section 2.5). Finally, we summarize the limitations of previous work in Section 2.6.

2.1 Web Caching

Caching can be *client-initiated* or *server-initiated*. Most caching schemes in wide-area, distributed systems are *client-initiated*, such as used by current Web browsers and Web proxies [32]. The problems with both of these solutions are myopic. A client cache does nothing to reduce traffic to a neighboring computer, and a web proxy does not help neighboring proxies. Thus the effectiveness of caching is ultimately limited to the low level of sharing of remote documents among clients of the same site [4]. A possible solution, *server-initiated caching*, allows servers to determine when and where to distribute objects [3, 4, 21]. Essentially, Content Delivery Networks (CDNs, including our approach) are server-initiated caching *with dedicated edge servers*. Previous server-initiated caching systems rely on unrealistic assumptions. Bestavros *et al.* model the Internet as a hierarchy and any internal

[1] Caches may be kept coherent in a variety of ways (for instance [44]).

Properties	Web caching (client initiated)	Web caching (server initiated)	Uncooperative pull-based CDNs	Cooperative push-based CDNs	**SCAN**
Cache/replica sharing for efficient replication	No, uncooperative	Yes, cooperative	No, uncooperative	Yes, cooperative	**Yes, cooperative**
Scalability for request redirection	No redirection	OK, use Bloom filter [15] to exchange replica locations	Bad, centralized CDN name server	Bad, centralized CDN name server	**Good, decentralized DHT location services**
Granularity of replication	Per URL	Per URL	Per URL	Per Website	**Per cluster**
Distributed load balancing	No	No	Yes	No	**Yes**
Replica coherence	No	No	No	No	**Yes**
Network monitoring for fault-tolerance	No	No	Yes, but unscalable monitoring	No	**Yes, scalable monitoring**

Table 1 Comparison of various Internet content delivery systems

node is available as a service proxy [3, 4]. This assumption is not valid because internal nodes are routers, unlikely to be available as service proxies. Geographical push-caching autonomously replicate HTML pages based on the global knowledge of the network topology and clients' access patterns [21]. More recently, adaptive web caching [34] and summary cache [15] are proposed to enable the sharing of caches among Web proxies. Caches exchange content state periodically with other caches, eliminating the delay and unnecessary use of resources of explicit cache probing. However, each proxy server needs to send index update of cached contents to *all* other proxy servers, and needs to store the content indices of *all* other proxy servers. Thus even with compact content index summary like the Bloom filter [15], the state maintenance and exchange overhead is still overwhelming and unscalable with the number of documents and number of cache servers. For instance, the target number of proxy servers is only in the order of 100 [15]. Furthermore, without dedicated infrastructure like CDN, caching proxies can not adapt to network congestion/failures or provide distributed load balancing.

2.2 Un-cooperative Pull-based CDNs

Recently, CDNs have been commercialized to provide Web hosting, Internet content and streaming media delivery. Basically, the contents are pulled to the edge servers upon clients' requests. Various mechanisms, such as DNS-based redirection, URL rewriting, HTTP redirection, *etc.*[1], have been proposed to direct client requests for objects to one of the CDN servers (*a. k. a.* CDN nodes or edge servers). Most of the

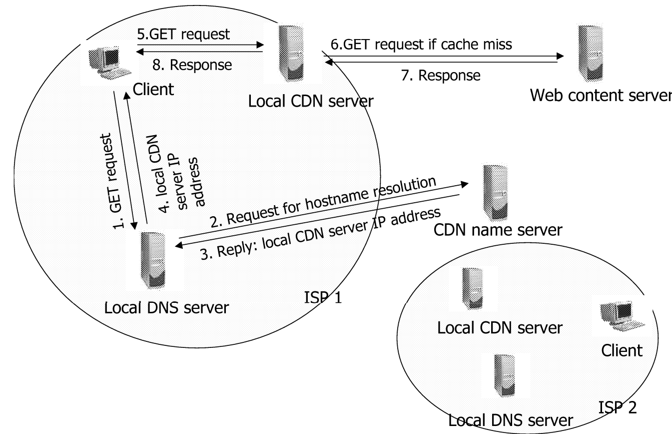

Fig. 2 Un-cooperative pull-based CDN architecture

commercial CDN providers, such as Akamai [14], LimeLight Networks [31], and Mirror Image [35], use DNS-based redirection due to its transparency [28]. Fig. 2 shows the CDN architecture using DNS-based redirection. Given the rapid growth of CDN service providers, such as Akamai (which already has more than 25,000 servers in about 900 networks spanning across 69 countries [14]), we assume that for each popular clients cluster, there is a CDN server as well as a local DNS server. The client cluster is the group of clients that are topologically close. The clients can be grouped by their BGP prefix [27] or by their local DNS servers. The latter is simple and adopted in practice, but it is not very accurate [33].

Fig. 2 gives the sequence of operations for a client to retrieve a URL. The hostname resolution request is sent to the CDN name server via local DNS server. Due to the nature of centralized location service, the CDN name server cannot afford to keep records for the locations of each URL replica. Thus it can only redirect the request based on network proximity, bandwidth availability and server load. The CDN server that gets the redirected request may not have the replica. In that case, it will pull a replica from the Web content server, then reply to the client.

Due to the uncooperative nature, current CDNs often places more replicas than necessary and consumes lots of resources for storage and update. Simulations reveals that with reasonable latency guarantees, cooperative push-based CDN (defined in Section 2.3) only uses a small fractional number of replicas (6-8%) and less than 10% of the update dissemination bandwidth than the uncooperative schemes [10, 11].

As a research effort, Rabinovich and Aggarwal propose RaDaR, a global Web hosting service with dynamic content replication and migration [41]. However, it requires the DNS to give the complete path from the client to the server, which in practice is often unavailable.

2.3 Cooperative Push-based CDNs

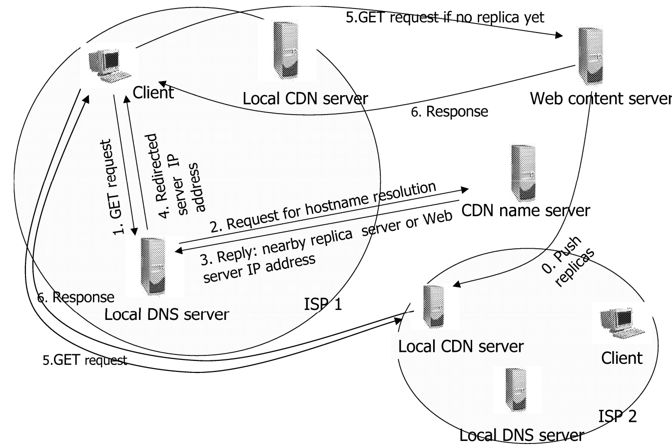

Fig. 3 Cooperative push-based CDN architecture

Several recent works proposed to pro-actively push content from the origin Web server to the CDN edge servers or proxies according to users' access patterns and global network topology, and have the replicas cooperatively satisfy clients' requests [30, 25, 40, 48].

The key advantage of this cooperative push-based replication scheme over the conventional one does not come from the fact that we use push instead of pull (which only saves compulsory miss), but comes from the cooperative sharing of the replicas deployed. This cooperative sharing significantly reduces the number of replicas deployed, and consequently reduces the replication and update cost [10, 11].

We can adopt a similar CDN architecture as shown in Fig. 3 to support such a cooperative push-based content distribution. First, the Web content server incrementally pushes contents based on their hyperlink structures and/or some access history collected by CDN name server [10, 11]. The content server runs a "push" daemon, and advertises the replication to the CDN name server, which maintains the mapping between content, identified by the host name in its (rewritten) URL, and their replica locations. The mapping can be coarse (e.g., at the level of Web sites if replication is done in units of Web sites), or fine-grained (e.g., at the level of URLs if replication is done in units of URLs).

With such replica location tracking, the CDN name server can redirect a client's request to its closest replica. Note that the DNS-based redirection allows address resolution on a per-host level. We combine it with content modification (e.g., URL rewriting) to achieve per-object redirection [1]. References to different objects are rewritten into different host names. To reduce the size of the domain name spaces, objects can be clustered as studied by Chen *et al.* [10, 11], and each cluster shares

the same host name. Since the content provider can rewrite embedded URLs *a priori* before pushing out the objects, it does not affect the users' perceived latency and the one-time overhead is acceptable. In both models, the CDN edge servers are allowed to execute their cache replacement algorithms. That is, the mapping in cooperative push-based replication is soft-state. If the client cannot find the content in the redirected CDN edge server, either the client will ask the CDN name server for another replica, or the edge server pulls the content from the Web server and replies to the client.

Li *et al.* approach the proxy placement problem with the assumption that the underlying network topologies are trees, and model it as a dynamic programming problem[30]. While an interesting first step, this approach has an important limitation in that the Internet topology is not a tree. More recent studies [40, 25], based on evaluating real traces and topologies, have independently reported that a greedy placement algorithm can provide content delivery networks with performance that is close to optimal. To simplify the assumption about detailed knowledge of global network topology and clients' distribution, topology-informed Internet replica placement was proposed to place replicas on the routers with big fanout [42]. They show that the router-level topology based replica placement can achieve average client latencies within a factor of 1.1 - 1.2 of the greedy algorithm, but only if the placement method is carefully designed.

2.4 Object Location Systems

Networked applications are extending their reach to a variety of devices and services over the Internet. Applications expanding to leverage these network resources find that *locating objects* on the wide-area is an important problem. Further, the read-mostly model of shared access, widely popularized by the World-Wide-Web, has led to extensive object replication, compounding the problem of object location. Work on location services has been done in a variety of contexts [13, 19, 23, 50]. These approaches can be roughly categorized into the following three groups: *Centralized Directory Services (CDS)*, *Replicated Directory Services (RDS)*, and *Distributed Directory Services (DDS)*.

Extensive work on these directory services have been proposed as we will discuss in more detail in this subsection. However, to the best of our knowledge, there is no attempt to benchmark and contrast their performance.

2.4.1 Centralized and Replicated Directory Services

A *centralized directory service* (CDS) resides on a single server and provides location information for every object on the network (See Fig. 4). Because it resides on a single server, it is extremely vulnerable to DoS attacks. A variant of this is the *replicated directory service* (RDS) which provides multiple directory servers. An

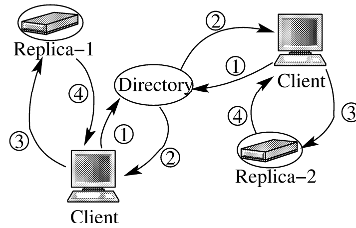

Fig. 4 A Centralized Directory Service (CDS): Clients contact a single directory to discover the location of a close replica. Clients subsequently contact the replica directly. A Replicated Directory Service (RDS) provides multiple directories.

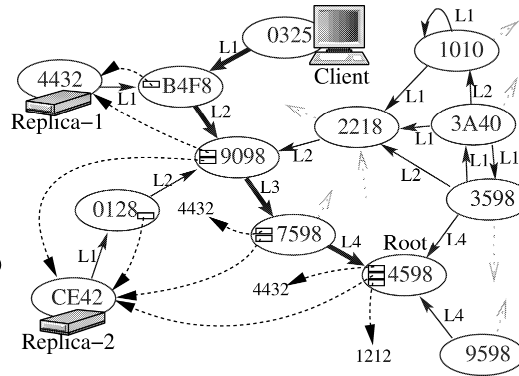

Fig. 5 A Distributed Directory (Tapestry): Nodes connected via links (solid arrows). Nodes route to nodes one digit at a time: e.g. 1010 → 2218 → 9098 → 7598 → 4598. Objects are associated with one particular "root" node (e.g. 4598). Servers publish replicas by sending messages toward root, leaving back-pointers (dotted arrows). Clients route directly to replicas by sending messages toward root until encountering pointer (e.g. 0325 → $B4F8$ → 4432).

RDS provides higher availability, but suffers consistency overhead. Here we do not consider the partitioned directory service because it often requires extra meta directory server for maintaining the partitioning information, such as the root server of DNS.

2.4.2 Distributed Directory Services: the Tapestry Infrastructure

Networking researchers have begun to explore decentralized peer-to-peer location services with distributed hash table (DHT), such as CAN [43], Chord [47], Pastry [45] and Tapestry [50]. Such services offer a distributed infrastructure for locating objects quickly with guaranteed success. Rather than depending on a single server to locate an object, a query in this model is passed around the network until it reaches a node that knows the location of the requested object. The lack of a single target in decentralized location services means they provide very high availability even under attack; the effects of successfully attacking and disabling a set of nodes is limited to a small set of objects.

In addition, Tapestry exploits locality in routing messages to mobile endpoints such as object replicas; this behavior is in contrast to other structured peer-to-peer overlay networks [43, 47, 45]. Thus we leverage on Tapestry to build SCAN.

Tapestry is an IP overlay network that uses a distributed, fault-tolerant architecture to track the location of objects in the network. It has two components: a *routing mesh* and a *distributed location services*.

Tapestry Routing Mesh Fig. 5 shows a portion of Tapestry. Each node joins Tapestry in a distributed fashion through nearby surrogate servers and set up *neighboring* links for connection to other Tapestry nodes. The neighboring links are shown as solid arrows. Such neighboring links provide a route from every node to every other node; the routing process resolves the destination address one digit at a time. (e.g., ***8 $\Longrightarrow$ **98 $\Longrightarrow$ *598 $\Longrightarrow$ 4598, where *'s represent wildcards). This routing scheme is based on the hashed-suffix routing structure originally presented by Plaxton, Rajaraman, and Richa [39].

Tapestry Distributed Location Service Tapestry assigns a globally unique name (GUID) to every object. It then deterministically maps each GUID to a unique *root* node. Storage servers *publish* objects by sending messages toward the roots, depositing *location pointers* at each hop. Fig. 5 shows two replicas and the Tapestry root for an object. These mappings are simply pointers to the server s where object o is being stored, and not a copy of the object itself. Thus for nearby objects, client search messages quickly intersect the path taken by publish messages, resulting in quick search results that exploit locality. It is shown that the average distance travelled in locating an object is *proportional* to the distance from that object [39].

2.5 Multicast for Disseminating Updates

For update dissemination, IP multicast has *fundamental* problems as the architectural foundation for Internet distribution. For instance, it works only across space, not across time, while most content distribution on the Internet works across both [16]. Further, there is no widely available inter-domain IP multicast.

As an alternative, many application-level multicast (in short, ALM) systems have been proposed [16, 12, 17, 38, 7, 51]. Among them, some [12, 7, 38] target small group, multi-source applications, such as video-conferencing, while others [16, 17, 51] focus on large-scale, single-source applications, such as streaming media multicast. Bayeux [51] is also built on top of Tapestry. It uses the Tapestry location service to find the multicast root(s), and then uses Tapestry routing to route both the control (e.g., "join") and data messages. In contrast, we only use the Tapestry location mechanism to find the nearby replica.

Most ALM systems have scalability problems, since they utilize a central node to maintain states for all existing children [12, 17, 38, 7], or to handle all "join" requests [51]. Replicating the root is the common solution [17, 51], but this suffers from consistency problems and communication overhead. On the other hand, Scribe [46] and the update multicast system of SCAN (namely dissemination tree) leverage peer-to-peer routing and location services, and do not have the scalability problem. Scribe is a large-scale event notification system, using overlay DHT for both subscription and dissemination. The dissemination tree is more efficient because we use overlay DHT only for subscription, and use IP for dissemination directly.

2.6 Summary

In summary, we find that previous work on CDNs and its related techniques have the following limitations.

1. Client-initiated web caching is myopic, while the server-initiated web caching has unscalable content state exchange overhead. Neither can adapt to network congestion/failures or provide distributed load balancing.
2. CDNs rely on centralized location services, thus they have to either apply inefficient and pull-based replication (uncooperative CDN), or replicate at the granularity of per Website and sacrifice the performance to clients (cooperative CDN).
3. There is no performance or DoS attack resilience benchmark for existing location services. This makes it difficult to compare the alternative proposals.
4. No coherence to replicas/caches: IP multicast doesn't exist in the Internet, while the existing application-level multicast has scalability problem.

In SCAN, the first two limitations are addressed with distributed location services, Tapestry, and we propose a network DoS resilience benchmark to contrast its performance with other alternatives [8]. For limitation 4, we dynamically place replicas and self-organize them into a scalable application-level multicast tree to disseminate updates as presented next.

3 Dynamic Replica Placement Problem Formulation

As shown in Fig. 1, replica placement is a key component of SCAN. According to users' requests, it dynamically places a minimal number of replicas while meeting client QoS and server capacity constraints. The location services discussed in last section are notified about the new replicas via Tapestry *PUBLISHOBJECT* API [50].

There is a large design space for modelling Web replica placement as an optimization problem and we describe it as follows. Consider a popular Web site or a CDN hosting server, which aims to improve its performance by pushing its content to some hosting server nodes. The problem is to dynamically decide where content is to be replicated so that some objective function is optimized under a dynamic traffic pattern and set of clients' QoS and/or resource constraints. The objective function can either minimize clients' QoS metrics, such as latency, loss rate, throughput, *etc.*, or minimize the replication cost of CDN service providers, e.g., network bandwidth consumption, or an overall cost function if each link is associated with a cost. For Web content delivery, the major resource consumption in replication cost is the network access bandwidth at each Internet Data Center (IDC) to the backbone network. Thus when given a Web object, the cost is linearly proportional to the number of replicas.

As Qiu *et al.* tried to minimize the total response latency of all the clients' requests with the number of replicas as constraint [40], we tackle the replica place-

ment problem from another angle: minimize the number of replicas when meeting clients' latency constraints and servers' capacity constraints. Here we assume that clients give reasonable latency constraints as it can be negotiated through a service-level agreement (SLA) between clients and CDN vendors. Thus we formulate the Web content placement problem as follows.

Given a network G with C clients and S server nodes, each client c_i has its *latency constraint* d_i, and each server s_j has its load/bandwidth/storage *capacity constraint* l_j. The problem is to find a smallest set of servers S' such that the distance between any client c_i and its "parent" server $s_{c_i} \in S'$ is bounded by d_i. More formally, find the minimum K, such that there is a set $S' \subset S$ with $|S'| = K$ and $\forall\ c \in C$, $\exists\ s_c \in S'$ such that distance(c, s_c) $\leq d_c$. Meanwhile, these clients C and servers S' self-organize into an application-level multicast tree with C as leaves and $\forall\ s_i \in S'$, its fan-out degree (i.e., number of direct children) satisfies $f(s_i) \leq l_i$.

4 Replica Placement Algorithms

The presence of an underlying DOLR with routing locality can be exploited to perform simultaneous replica placement and tree construction. Every SCAN server is a member of the DOLR. Hence, new replicas are published into the DOLR. Further, each client directs its requests to its proxy SCAN server; this proxy server interacts with other SCAN servers to deliver content to the client.

Although we use the DOLR to locate replicas during tree building, we otherwise communicate through IP. In particular, we use IP between nodes in a d-tree for parents and children to keep track of one another. Further, when a client makes a request that results in placement of a new replica, the client's proxy keeps a cached pointer to this new replica. This permits direct routing of requests from the proxy to the replica. Cached pointers are soft state since we can always use the DOLR to locate replicas.

4.1 Goals for Replica Placement

Replica placement attempts to satisfy both *client latency* and *server load* constraints. *Client latency* refers to the round-trip time required for a client to read information from the SCAN system. We keep this within a pre-specified limit. *Server load* refers to the communication volume handled by a given server. We assume that the load is directly related to the number of clients it handles and the number of d-tree children it serves. We keep the load below a specified maximum. Our goal is to meet these constraints while minimizing the number of deployed replicas, keeping the d-tree balanced, and generating as little traffic during update as possible. Our success will be explored in Section 6.

4.2 Dynamic Placement

Our dynamic placement algorithm proceeds in two phases: *replica search* and *replica placement*. The replica search phase attempts to find an existing replica that meets the client latency constraint without being overloaded. If this is successful, we place a link in the client and cache it at the client's proxy server. If not, we proceed to the replica placement phase to place a new replica.

Replica search uses the DOLR to contact a replica "close" to the client proxy; call this the *entry* replica. The locality property of the DOLR ensures that the entry replica is a reasonable candidate to communicate with the client. Further, since the d-tree is connected, the entry replica can contact all other replicas. We can thus imagine three search variants: *Singular* (consider only the entry replica), *Localized* (consider the parent, children, and siblings of the entry replica), and *Exhaustive* (consider all replicas). For a given variant, we check each of the included replicas and select one that meets our constraints; if none meet the constraint, we proceed to place a new replica.

procedure DynamicReplicaPlacement_Naive(c, o)
1 c sends *JOIN* request to o through DOLR, reaches entry server s. Request collects $IP_{s'}$, $dist_{overlay}(c,s')$ and $rc_{s'}$ for each server s' on the path.
2 **if** $rc_s > 0$ **then**
  if $dist_{overlay}(c, s) \leq d_c$ **then** s becomes parent of c, **exit**.
  else
3 s pings c to get $dist_{IP}(c, s)$.
4   **if** $dist_{IP}(c, s) \leq d_c$ **then** s becomes parent of c, **exit**.
  end
end
5 At s, choose s' on path with $rc_{s'} > 0$ *and* smallest $dist_{overlay}(t,c) \leq d_c$
if $\nexists$ *such* s' **then**
6 for each server s' on the path, s collects $dist_{IP}(c,s')$ and chooses s' with $rc_{s'} > 0$ *and* smallest $dist_{IP}(t,c) \leq d_c$.
end
7 s puts a replica on s' and becomes its parent, s' becomes parent of c.
8 s' publishes replica in DOLR, **exit**.

Algorithm 1: Naive Dynamic Replica Placement. Notation: Object o. Client c with latency constraint d_c. Entry Server s. Every server s' has remaining capacity $rc_{s'}$ (additional children it can handle). The overlay distance ($dist_{overlay}$(x,y)) and IP distance ($dist_{IP}$(x,y)) are the round trip time (RTT) on overlay network and IP network, separately.

We restrict replica placement to servers visited by the DOLR routing protocol when sending a message from the client's proxy to the entry replica. We can locate these servers without knowledge of global IP topology. The locality properties of the DOLR suggest that these are good places for replicas. We consider two placement strategies: *Eager* places the replica as close to the client as possible and *Lazy* places the replica as far from the client as possible. If all servers that meet the latency

procedure DynamicReplicaPlacement_Smart(c, o)

1 c sends *JOIN* request to o through DOLR, reaches entry server s

2 From s, request forwarded to children (sc), parent (p), and siblings (ss)

3 Each family member t with $rc_t > 0$ sends rc_t to c. c measures $dist_{IP}(t,c)$ through TCP three-way handshaking.

4 **if** $\exists$ *t and* $dist_{IP}(t,c) \leq d_c$ **then**

5 $\quad$ $c$chooses t as parent with biggest rc_t and $dist_{IP}(t,c) \leq d_c$, **exit**.

else

6 $\quad$ c sends *PLACEMENT* request to o through DOLR, reaches entry server s

$\quad$ Request collects $\text{IP}_{s'}$, $dist_{overlay}(c,s')$ and $rc_{s'}$ for each server s' on the path.

7 $\quad$ At s, choose s' on path with $rc_{s'} > 0$ *and* largest $dist_{overlay}(t,c) \leq d_c$

$\quad$ **if** $\nexists$ *such* s' **then**

8 $\quad\quad$ for each server s' on the path, s collects $dist_{IP}(c,s')$ and chooses s' with $rc_{s'} > 0$ *and* largest $dist_{IP}(t,c) \leq d_c$.

$\quad$ **end**

9 $\quad$ s puts a replica on s' and becomes its parent, s' becomes parent of c.

10 $\quad$ s' publishes replica in DOLR, **exit**.

end

Algorithm 2: Smart Dynamic Replica Placement. Notation: Object o. Client c with latency constraint d_c. Entry Server s. Every server s' has remaining capacity $rc_{s'}$ (additional children it can handle). The overlay distance ($dist_{overlay}$(x,y)) and IP distance ($dist_{IP}$(x,y)) are the round trip time (RTT) on overlay network and IP network, separately.

constraint are overloaded, we replace an old replica; if the entry server is overloaded, we disconnect the oldest link among its d-trees.

4.2.1 Dynamic Techniques

We can now combine some of the above options for search and placement to generate dynamic replica management algorithms. Two options that we would like to highlight are as follows.

- *Naive Placement:* A simple combination utilizes *Singular* search and *Eager* placement. This heuristic generates minimal search and placement traffic.
- *Smart Placement:* A more sophisticated algorithm is shown in Algorithm 4. This algorithm utilizes *Localized* search and *Lazy* placement.

Note that we try to use the overlay latency to estimate the IP latency in order to save "ping" messages. Here the client can start a daemon program provided by its CDN service provider when launching the browser so that it can actively participate in the protocols. The locality property of Tapestry naturally leads to the locality of d-tree, i.e., the parent and children tend to be close to each other in terms of the number of IP hops between them. This provides good delay and multicast bandwidth consumption when disseminating updates, as measured in Section 6. The tradeoff between the naive and smart approaches is that the latter consumes more "join" traffic to construct a tree with fewer replicas, covering more clients, with less delay and multicast bandwidth consumption. We evaluate this tradeoff in Section 6.

4.2.2 Static Comparisons

The replica placement methods given above are unlikely to be optimal in terms of the number of replicas deployed, since clients are added sequentially and with limited knowledge of the network topology. In the static approach, the root server has complete knowledge of the network and places replicas *after* getting all the requests from the clients. In this scheme, updates are disseminated through IP multicast. Static placement is not very realistic, but may provide better performance since it exploits knowledge of the client distribution and global network topology.

The problem formulated in Section 3 can be converted to a special case of the capacitated facility location problem [24] defined as follows. Given a set of locations i at which facilities may be built, building a facility at location i incurs a cost of f_i. Each client j must be assigned to one facility, incurring a cost of $d_j c_{ij}$ where d_j denotes the demand of the node j, and c_{ij} denotes the distance between i and j. Each facility can serve at most l_i clients. The objective is to find the number of facilities and their locations yielding the minimum total cost.

To map the facility location problem to ours, we set f_i always 1, and set c_{ij} 0 if location i can cover client j or ∞ otherwise. The best approximation algorithm known today uses the primal-dual schema and Lagrangian relaxation to achieve a guaranteed factor of 4 [24]. However, this algorithm is too complicated for practical use. Instead, we designed a greedy algorithm that has a logarithmic approximation ratio.

Besides the previous notations, we define the following variables: set of covered clients by s: C_s, $C_s \subseteq C$ and $\forall\, c \in C_s$, $dist_{IP}(c, s) \leq d_c$; set of possible server parents for client c: S_c, $S_c \subseteq S$ and $\forall\, s \in S_c$, $dist_{IP}(c, s) \leq d_c$.

```
procedure ReplicaPlacement_Greedy_DistLoadBalancing(C, S)
input   : Set of clients to be covered: C, total set of servers: S
output  : Set of servers chosen for replica placement: S′
while C is not empty do
    Choose s ∈ S which has the largest value of min(cardinality |C_s|, remaining capacity
    rc_s)
    S′ = S′ ⋃ {s}
    S = S - {s}
    if |C_s| ≤ rc_s then C = C - C_s
    else
        Sort each element c ∈ C_s in increasing order of |S_c|
        Choose the first rc_s clients in C_s as C_sChosen
        C = C - C_sChosen
    end
    recompute S_c for ∀ c ∈ C
end
return S′.
```

Algorithm 3: Static Replica Placement with Load Balancing

We consider two types of static replica placement:

- *IP Static:* The root has global IP topology knowledge.
- *Overlay Static:* For each client *c*, the root only knows the servers on the Tapestry path from *c* to the root which can cover that client (in IP distance).

The first of these is a "guaranteed-not-to-exceed" optimal placement. We expect that it will consume the least total number of replicas and lowest multicast traffic. The second algorithm explores the best that we could expect to achieve gathering all topology information from the DOLR system.

4.3 Soft State Tree Management

Soft-state infrastructures have the potential to be extremely robust, precisely because they can be easily reconfigured to adapt to circumstances. For SCAN we target two types of adaptation: fault recovery and performance tuning.

To achieve fault resilience, the data source sends periodic *heartbeat* messages through the d-tree. Members know the frequency of these heartbeats and can react when they have not seen one for a sufficiently long time. In such a situation, the replica initiates a *rejoin* process – similar to the replica search phase above – to find a new parent. Further, each member periodically sends a *refresh* message to its parent. If the parent does not get the refresh message within a certain threshold, it invalidates the child's entry. With such soft-state group management, any SCAN server may crash without significantly affecting overall CDN performance.

Performance tuning consists of pruning and re-balancing the d-tree. Replicas at the leaves are pruned when they have seen insufficient client traffic. To balance the d-tree, each member periodically rejoins the tree to find a new parent.

5 Evaluation Methodology

We implement an event-driven simulator for SCAN because *ns*2 [5] can only scale up to one thousand nodes. This includes a packet-level network simulator (with a static version of the Tapestry DOLR) and a replica management framework. The soft-state replica layer is driven from simulated clients running workloads. Our methodology includes evaluation metrics, network setup and workloads.

5.1 Metrics

Our goal is to evaluate the replica schemes of Section 4.2. These strategies are dynamic naive placement (*od_naive*), dynamic smart placement (*od_smart*), overlay static placement (*overlay_s*), and static placement on IP network (*IP_s*). We compare the efficacy of these four schemes via three classes of metrics:

- *Quality of Replica Placement*: Includes number of deployed replicas and degree of load distribution, measured by the ratio of the standard deviation vs. the mean of the number of client children for each replica server.
- *Multicast Performance*: We measure the relative delay penalty (RDP) and the bandwidth consumption which is computed by summing the number of bytes multiplied by the transmission time over every link in the network. For example, the bandwidth consumption for 1K bytes transmitted in two links (one has 10 ms, the other 20 ms latency) is 1KB $\times$ (10+20)ms = 0.03(KB.sec).
- *Tree Construction Traffic:* We count both the number of application-level messages sent and the bandwidth consumption for deploying replicas and constructing d-tree.

In addition, we quantify the effectiveness of capacity constraints by computing the *maximal load* with or without constraints. The maximal load is defined as the maximal number of client cache children on any SCAN server. Sensitivity analysis are carried out for various client/server ratios and server densities.

5.2 Network Setup

We use the GT-ITM transit-stub model to generate five 5000-node topologies [49]. The results are averaged over the experiments on the five topologies. A packet-level, priority-queue based event manager is implemented to simulate the network latency. The simulator models the propagation delay of physical links, but does not model bandwidth limitations, queuing delays, or packet losses.

We utilize two strategies for placing SCAN servers. One selects all SCAN servers at random (labelled *random SCAN*). The other preferentially chooses transit and gateway nodes (labelled *backbone SCAN*). This latter approach mimics the strategy of placing SCAN servers strategically in the network.

To compare with a DNS-redirection-based Web content delivery network (CDN), we simulate typical behavior of such a system. We assume that every client request is redirected to the closest CDN server, which will cache a copy of the requested information for the client. This means that popular objects may be cached in every CDN server. We assume that content servers are allowed to send updates to replicas via IP multicast.

5.3 Workloads

To evaluate the replication schemes, we use both a synthetic workload and access logs collected from real Web servers. These workloads are a first step toward exploring more general uses of SCAN.

Our synthetic workload is a simplified approximation of *flash crowds*. Flash crowds are unpredictable, event-driven traffic surges that swamp servers and dis-

Web site	Period	# Requests total - simulated	# Clients	# Client groups total - simulated	# Objects simulated
MSNBC	10-11 am, 8/2/99	1604944 - 1377620	139890	16369 - 4000	4186
NASA	All day, 7/1/95	64398 - 64398	5177	1842 - 1842	3258

Table 2 Statistics of Web site access logs used for simulation

rupt site services. For our simulation, all the clients (not servers) make requests to a given hot object in random order.

Our trace-driven simulation includes a large and popular commercial news site, MSNBC [36], as well as traces from NASA Kennedy Space Center [37]. Table 5.3 shows the detailed trace information. We use the access logs in the following way. We group the Web clients based on BGP prefixes [27] using the BGP tables from a BBNPlanet (Genuity) router [2]. For the NASA traces, since most entries in the traces contain host names, we group the clients based on their domains, which we define as the last two parts of the host names (e.g., a1.b1.com and a2.b1.com belong to the same domain). Given the maximal topology we can simulate is 5000 (limited by machine memory), we simulate all the clients groups for NASA and 4000 top client groups (cover 86.1% of requests) for MSNBC. Since the clients are unlikely to be on transit nodes nor on server nodes, we map them randomly to the rest of nodes in the topology.

6 Evaluation Results

In this section, we evaluate the performance of the SCAN dynamic replica management algorithms. What we will show is that:

- For realistic workloads, SCAN places close to an optimal number of replicas, while providing good load balance, low delay, and reasonable update bandwidth consumption relative to static replica placement on IP multicast.
- SCAN outperforms the existing DNS-redirection based CDNs on both replication and update bandwidth consumption.
- The performance of SCAN is relatively insensitive to the SCAN server deployment, client/server ratio, and server density.
- The capacity constraint is quite effective at balancing load.

We will first present results on synthetic workload, and then the results of real Web traces.

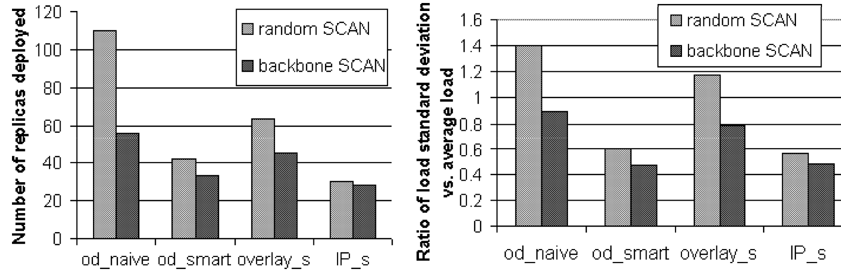

Fig. 6 Number of replicas deployed (left) and load distribution on selected servers (right) (500 SCAN servers)

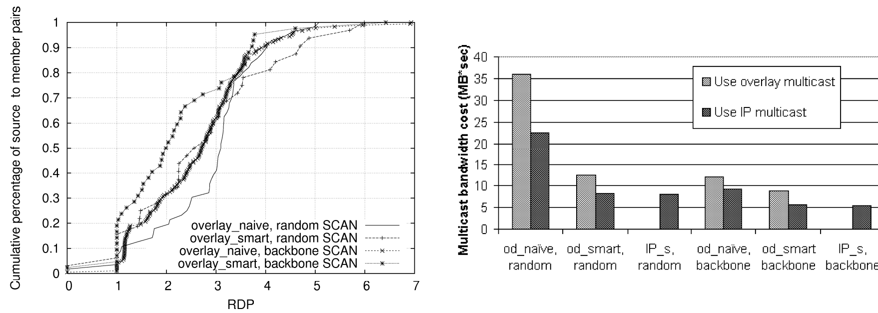

Fig. 7 Cumulative distribution of RDP (500 SCAN servers)

Fig. 8 Bandwidth consumption of 1MB update multicast (500 SCAN servers)

6.1 Results for the Synthetic Workload

We start by examining the synthetic, flash crowd workload. 500 nodes are chosen to be SCAN servers with either “random” or “backbone” approach. Remaining nodes are clients and access some hot object in a random order. We randomly choose one non-transit SCAN server to be the data source and set as 50KB the size of the hot object. Further, we assume the latency constraint is 50ms and the load capacity is 200 clients/server.

6.1.1 Comparison Between Strategies

Fig. 6 shows the number of replicas placed and the load distribution on these servers. *Od_smart* approach uses only about 30% to 60% of the servers used by *od_naive*, is even better than *overlay_s*, and is very close to the optimal case: *IP_s*. Also note that *od_smart* has better load distribution than *od_naive* and *overlay_s*, close to *IP_s* for both *random* and *backbone SCAN*.

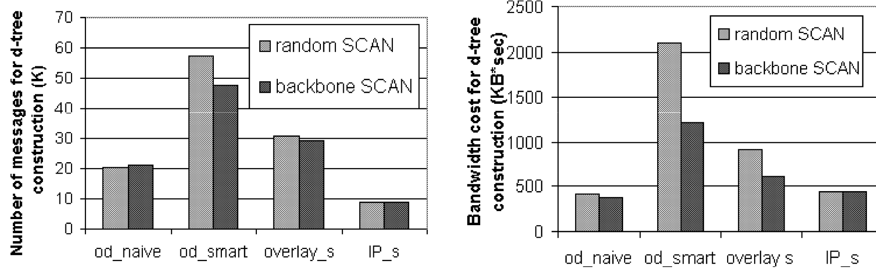

Fig. 9 Number of application-level messages (left) and total bandwidth consumed (right) for d-tree construction (500 SCAN servers)

Relative Delay Penalty (RDP) is the ratio of the overlay delay between the root and any member in d-tree vs. the unicast delay between them [12]. In Fig. 7, *od_smart* has better RDP than *od_naive*, and 85% of *od_smart* RDPs between any member server and the root pairs are within 4. Fig. 8 contrasts the bandwidth consumption of various replica placement techniques with the optimal IP static placement. The results are very encouraging: the bandwidth consumption of *od_smart* is quite close to *IP_s* and is much less than that of *od_naive*.

The performance above is achieved at the cost of d-tree construction (Fig. 9). However, for both *random* and *backbone SCAN*, *od_smart* approach produces less than three times of the messages of *od_naive* and less than six times of that for optimal case: *IP_s*. Meanwhile, *od_naive* uses almost the same amount of bandwidth as *IP_s* while *od_smart* uses about three to five times that of *IP_s*.

In short, the smart dynamic algorithm has performance that is close to the ideal case (static placement with IP multicast). It places close to an optimal number of replicas, provides better load distribution, and less delay and multicast bandwidth consumption than the naive approach – at the price of three to five times as much tree construction traffic. Since d-tree construction is a much less frequent than data access and update this is a good tradeoff.

Due to the limited number and/or distribution of servers, there may exist some clients who cannot be covered when facing the QoS and capacity requirements. In this case, our algorithm can provide hints as where to place more servers. Note that experiments show that the naive scheme has many more uncovered clients than the smart one, due to the nature of its unbalanced load. Thus, we remove it from consideration for the rest of synthetic workload study.

6.1.2 Comparison with a CDN

As an additional comparison, we contrast the overlay smart approach with a DNS-redirection-based CDN. Compared with a traditional CDN, the overlay smart ap-

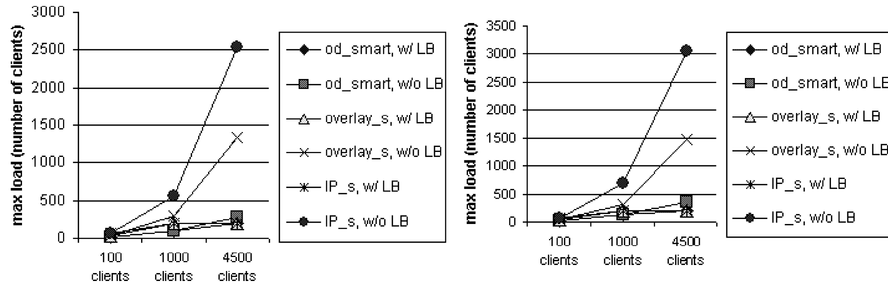

Fig. 10 Maximal load measured with and without load balancing constraints (LB) for various numbers of clients (left: 500 random servers, right: 500 backbone servers)

proach uses a fraction of the number of replicas (6-8%) and less than 10% of bandwidth for disseminating updates.

6.1.3 Effectiveness of Distributed Load Balancing

We study how the capacity constraint helps load balancing with three client populations: 100, 1000 and 4500. The former two are randomly selected from 4500 clients. Fig. 10 shows that lack of capacity constraints (labelled *w/o LB*) leads to hot spot or congestion: some servers will take on about 2-13 times their maximum load. Performance with load balancing is labelled as *w/ LB* for contrast.

6.1.4 Performance Sensitivity to Client/Server Ratio

We further evaluate SCAN with the three client populations Fig. 11 shows the number of replicas deployed. When the number of clients is small, *w/ LB* and *w/o LB* do not differ much because no server exceeds the constraint. The number of replicas required for *od_smart* is consistently less than that of *overlay_s* and within the bound of 1.5 for *IP_s*. As before, we also simulate other metrics, such as load distribution, delay and bandwidth penalty for update multicast under various client/server ratios. The trends are similar, that is, *od_smart* is always better than *overlay_s*, and very close to *IP_s*.

6.1.5 Performance Sensitivity to Server Density

Next, we increase the density of SCAN servers. We randomly choose 2500 out of the 5000 nodes to be SCAN servers and measure the resulting performance. Obviously, this configuration can support better QoS for clients and require less capacity for servers. Hence, we set the latency constraint to be 30 ms and capacity constraint 50 clients/server. The number of clients vary from 100 to 2500.

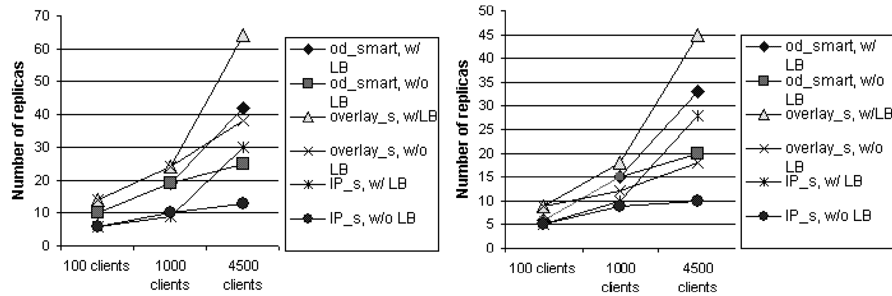

Fig. 11 Number of replicas deployed with and without load balancing constraints (LB) for various numbers of clients (left: 500 random servers, right: 500 backbone servers)

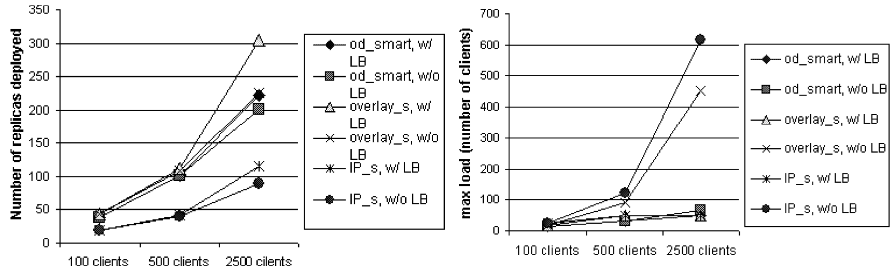

Fig. 12 Number of replicas deployed (left) and maximal load (right) on 2500 random SCAN servers with and without the load balancing constraint (LB)

With very dense SCAN servers, our *od_smart* still uses less replicas than *overlay_s*, although they are quite close. *IP_s* only needs about half of the replicas, as in Fig. 12. In addition, we notice that the load balancing is still effective. That is, overloaded machines or congestion cannot be avoided simply by adding more servers while neglecting careful design.

In summary, *od_smart* performs well with various SCAN server deployments, various client/server ratios, and various server densities. The capacity constraint based distributed load balancing is effective.

6.2 Results for Web Traces Workload

Next, we explore the behavior of SCAN for Web traces with documents of widely varying popularity. Fig. 13.a characterizes the request distribution for the two traces used (note that the x-axis is logarithmic.). This figure reveals that the request number for different URLs is quite unevenly distributed for both traces.

For each URL in the traces, we compute the number of replicas generated with *od_naive*, *od_smart*, and *IP_s*. Then we normalize the replica numbers of *od_naive* and *od_smart* by dividing them with the replica number of *IP_s*. We plot the CDF

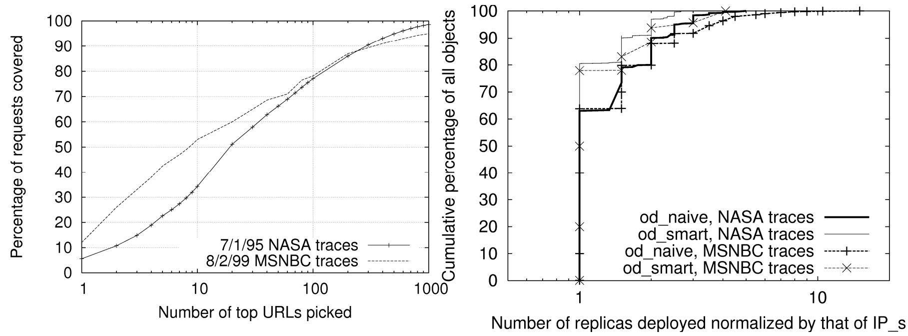

Fig. 13 Simulation with NASA and MSNBC traces on 100 backbone SCAN servers. (a) Percentage of requests covered by different number of top URLs (left); (b) the CDF of replica number deployed with *od_naive* and *od_smart* normalized by the number of replicas using *IP_s* (right)

of such ratios for both NASA and MSNBC in Fig. 13.b. The lower percentage part of the CDF curves are overlapped and close to 1. The reasons are most of the URLs have very few requests, and we only simulate a limited period, thus the number of replicas deployed by the three methods are very small and similar. However, *od_smart* and *od_naive* differ significantly for popular objects, exhibited in the higher percentage part. *Od_smart* is very close to *IP_s*, for all objects, the ratio is less than 2.7 for NASA and 4.1 for MSNBC, while the ratio for *od_naive* can go as high as 5.0 and 15.0, respectively.

In addition, we contrast the bandwidth consumption for disseminating updates. Given an update of unit size, for each URL, we compute the bandwidth consumed by using (1) overlay multicast on an *od_naive* tree, (2) overlay multicast on an *od_smart* tree, and (3) IP multicast on an *IP_s* tree. Again, we have metric (1) and (2) normalized by (3), and plot the CDF of the ratios. The curves are quite similar to Fig. 13.b.

In conclusion, although *od_smart* and *od_naive* perform similarly for infrequent or cold objects, *od_smart* outperforms dramatically over *od_naive* for hot objects which dominate overall requests.

6.3 Discussion

How does the distortion of topology through Tapestry affect replica placement? Notice that the overlay distance through Tapestry, on average, is about 2-3 times more than the IP distance. Our simulations in Section 6, shed some light on the resulting penalty: *Overlay_s* applies exactly the same algorithm as *IP_s* for replica placement, but uses the static Tapestry-level topology instead of IP-level topology. Simulation results show that *overlay_s* places 1.5 - 2 times more replicas than IP_s. For similar reasons, *od_smart* outperforms *overlay_s*. The reason is that *od_smart* uses "ping" messages to get the real IP distance between clients and servers. This observation also explains why *od_smart* gets similar performance to *IP_s*. One could imagine

scaling overlay latency by an expected "stretch" factor to estimate real IP distance – thereby reducing ping probe traffic.

7 Conclusions

The importance of adaptive replica placement and update dissemination is growing as distribution systems become pervasive and global. In this chapter, we present SCAN, a scalable, soft-state replica management framework built on top of a distributed object location and routing framework (DOLR) with locality. SCAN generates replicas on demand and self-organizes them into an application-level multicast tree, while respecting client QoS and server capacity constraints. An event-driven simulation of SCAN shows that SCAN places close to an optimal number of replicas, while providing good load distribution, low delay, and small multicast bandwidth consumption compared with static replica placement on IP multicast. Further, SCAN outperforms existing DNS-redirection based CDNs in terms of replication and update cost. SCAN shows great promise as an essential component of global-scale peer-to-peer infrastructures.

8 Acknowledgments

Some of the materials presented in this chapter appeared in a preliminary form at Pervasive'02 (the first International Conference on Pervasive Computing) [9]. I would like to thank other co-authors who contributed to the previous form of this work: Prof. Randy H. Katz and Prof. John D. Kubiatowicz from UC Berkeley and Prof. Lili Qiu from UT Austin.

References

1. BARBIR, A., CAIN, B., DOUGLIS, F., GREEN, M., HOFMANN, M., NAIR, R., POTTER, D., AND SPATSCHECK, O. Known CN request-routing mechanisms. http://www.ietf.org/internet-drafts/draft-ietf-cdi-known-request-routing-00.txt.
2. BBNPLANET. telnet://ner-routes.bbnplanet.net.
3. BESTAVROS, A. Demand-based document dissemination to reduce traffic and balance load in distributed information systems. In *Proc. of the IEEE Symposium on Parallel and Distributed Processing* (1995).
4. BESTAVROS, A., AND CUNHA, C. Server-initiated document dissemination for the WWW. In *IEEE Data Engineering Bulletin* (Sep. 1996).
5. BRESLAU, L., ESTRIN, D., FALL, K., FLOYD, S., HEIDEMANN, J., HELMY, A., HUANG, P., MCCANNE, S., VARADHAN, K., XU, Y., AND YU, H. Advances in network simulation. *IEEE Computer 33*, 5 (May 2000), 59–67.
6. CASTRO, M., AND LISKOV, B. Proactive recovery in a byzantine-fault-tolerant system. In *Proc. of USENIX Symp. on OSDI* (2000).

7. CHAWATHE, Y., MCCANNE, S., AND BREWER, E. RMX: Reliable multicast for heterogeneous networks. In *Proceedings of IEEE INFOCOM* (2000).
8. CHEN, Y., BARGTEIL, A., BINDEL, D., KATZ, R. H., AND KUBIATOWICZ, J. Quantifying network denial of service: A location service case study. In *Proceeding of Third International Conference on Information and Communications Security (ICICS)* (2001).
9. CHEN, Y., KATZ, R. H., AND KUBIATOWICZ, J. D. SCAN: a dynamic scalable and efficient content distribution network. In *Proc. of the First International Conference on Pervasive Computing* (Aug. 2002).
10. CHEN, Y., QIU, L., CHEN, W., NGUYEN, L., AND KATZ, R. H. Clustering Web content for efficient replication. In *Proc. of the 10th IEEE International Conference on Network Protocols (ICNP)* (2002).
11. CHEN, Y., QIU, L., CHEN, W., NGUYEN, L., AND KATZ, R. H. Efficient and adaptive Web replication using content clustering. *IEEE Journal on Selected Areas in Communications (J-SAC), Special Issue on Internet and WWW Measurement, Mapping, and Modeling 21*, 6 (2003), 979–994.
12. CHU, Y., RAO, S., AND ZHANG, H. A case for end system multicast. In *Proceedings of ACM SIGMETRICS* (June 2000).
13. CZERWINSKI, S., ZHAO, B., HODES, T., JOSEPH, A., AND KATZ, R. An architecture for a secure service discovery service. In *Proc. of ACM/IEEE MobiCom Conf.* (1999).
14. DILLEY, J., MAGGS, B., PARIKH, J., PROKOP, H., SITARAMAN, R., AND WEIHL, B. Globally distributed content delivery. *IEEE Internet Computing* (September/October 2002), 50–58.
15. FAN, L., CAO, P., ALMEIDA, J., AND BRODER, A. Summary cache: A scalable wide-area Web cache sharing protocol. In *Proc. of ACM SIGCOMM Conf.* (1998).
16. FRANCIS, P. Yoid: Your own Internet distribution. Technical report, ACIRI, http://www.aciri.org/yoid, April, 2000.
17. GIFFORD, D. K., JOHNSON, K. L., KAASHOEK, M. F., AND J. W. O'TOOLE, J. Overcast: Reliable multicasting with an overlay network. In *Proc. of USENIX Symp. on OSDI* (2000).
18. GRAY, J., HELLAND, P., O'NEIL, P., AND SHASHA, D. The dangers of replication and a solution. In *Proc. of ACM SIGMOD Conf.* (June 1996), vol. 25, 2, pp. 173–182.
19. GUTTMAN, E., PERKINS, C., VEIZADES, J., AND DAY, M. Service Location Protocol, Version 2. IETF Internet Draft, November 1998. RFC 2165.
20. GWERTZMAN, J., AND SELTZER, M. World-Wide Web Cache Consistency. In *Proceedings of the 1996 USENIX Technical Conference* (1996).
21. GWERTZMAN, J., AND SELTZER, M. An analysis of geographical push-caching. In *Proceedings of International Conference on Distributed Computing Systems* (1997).
22. HILDRUM, K., KUBIATOWICZ, J., RAO, S., AND ZHAO, B. Distributed data location in a dynamic network. In *Proc. of ACM SPAA* (2002).
23. HOWES, T. A. The Lightweight Directory Access Protocol: X.500 Lite. Tech. Rep. 95-8, Center for Information Technology Integration, U. Mich., July 1995.
24. JAIN, K., AND VARIRANI, V. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation. In *Proc. of IEEE FOCS* (1999).
25. JAMIN, S., JIN, C., KURC, A., RAZ, D., AND SHAVITT, Y. Constrained mirror placement on the Internet. In *Proceedings of IEEE Infocom* (2001).
26. KISTLER, J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 3–25.
27. KRISHNAMURTHY, B., AND WANG, J. On network-aware clustering of Web clients. In *Proc. of SIGCOMM* (2000).
28. KRISHNAMURTHY, B., WILLS, C., AND ZHANG, Y. On the use and performance of content distribution networks. In *Proceedings of SIGCOMM Internet Measurement Workshop* (2001).
29. KUBIATOWICZ, J., ET AL. Oceanstore: An architecture for global-scale persistent storage. In *Proceeedings of 9th ASPLOS* (2000).
30. LI, B., GOLIN, M. J., ITALIANO, G. F., DENG, X., AND SOHRABY, K. On the optimal placement of Web proxies in the Internet. In *Proceedings of IEEE INFOCOM* (1999).

31. LIMELIGHT NETWORKS INC. http://www.limelightnetworks.com/.
32. LUOTONEN, A., AND ALTIS, K. World-Wide Web proxies. In *Proc. of the First International Conference on the WWW* (1994).
33. MAO, Z. M., CRANOR, C., DOUGLIS, F., RABINOVICH, M., SPATSCHECK, O., AND WANG, J. A precise and efficient evaluation of the proximity between Web clients and their local DNS servers. In *Proc. of USENIX Technical Conf.* (2002).
34. MICHEL, S., NGUYEN, K., ROSENSTEIN, A., ZHANG, L., FLOYD, S., AND JACOBSON, V. Adaptive Web caching: Towards a new caching architecture. In *Proceedings of 3rd International WWW Caching Workshop* (June, 1998).
35. MIRROR IMAGE INTERNET INC. http://www.mirror-image.com.
36. MSNBC. http://www.msnbc.com.
37. NASA kennedy space center server traces. http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html.
38. PENDARAKIS, D., SHI, S., VERMA, D., AND WALDVOGEL, M. ALMI: An application level multicast infrastructure. In *Proceedings of 3rd USENIX Symposium on Internet Technologies* (2001).
39. PLAXTON, C. G., RAJARAMAN, R., AND RICHA, A. W. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of the SCP SPAA* (1997).
40. QIU, L., PADMANABHAN, V. N., AND VOELKER, G. M. On the placement of Web server replica. In *Proceedings of IEEE INFOCOM* (2001).
41. RABINOVICH, M., AND AGGARWAL, A. RaDaR: A scalable architecture for a global Web hosting service. In *Proceedings of WWW* (1999).
42. RADOSLAVOV, P., GOVINDAN, R., AND ESTRIN, D. Topology-informed Internet replica placement. In *Proceedings of the International Workshop on Web Caching and Content Distribution* (2001).
43. RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM* (2001).
44. RODRIGUEZ, P., AND SIBAL, S. SPREAD: Scalable platform for reliable and efficient automated distribution. In *Proceedings of WWW* (2000).
45. ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of ACM Middleware* (2001).
46. ROWSTRON, A., KERMARREC, A.-M., CASTRO, M., AND DRUSCHEL, P. SCRIBE: The design of a large-scale event notification infrastructure. In *Proceedings of International Workshop on Networked Group Communication (NGC)* (2001).
47. STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of ACM SIGCOMM* (2001).
48. VENKATARAMANI, A., YALAGANDULA, P., KOKKU, R., SHARIF, S., AND DAHLIN, M. The potential costs and benefits of long term prefetching for content distribution. In *Proc. of Web Content Caching and Distribution Workshop 2001* (2001).
49. ZEGURA, E., CALVERT, K., AND BHATTACHARJEE, S. How to model an Internetwork. In *Proceedings of IEEE INFOCOM* (1996).
50. ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications* (2003).
51. ZHUANG, S. Q., ZHAO, B. Y., JOSEPH, A. D., KATZ, R. H., AND KUBIATOWICZ, J. D. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of ACM NOSSDAV* (2001).