Dynamic Replica Placement for Scalable Content Delivery

Yan Chen, Randy H. Katz and John D. Kubiatowicz

Computer Science Division,
University of California, Berkeley
{yanchen, randy, kubitron}@cs.berkeley.edu

Abstract. In this paper, we propose the *dissemination tree*, a dynamic content distribution system built on top of a peer-to-peer location service. We present a replica placement protocol that builds the tree while meeting QoS and server capacity constraints. The number of replicas as well as the delay and bandwidth consumption for update propagation are significantly reduced. Simulation results show that the dissemination tree has close to the optimal number of replicas, good load distribution, small delay and bandwidth penalties for update multicast compared with the ideal case: static replica placement on IP multicast.

1 Introduction

The efficient distribution of Web content and streaming media is of growing importance. The challenge is to provide content distribution to clients with good *Quality of Service (QoS)* while retaining *efficient* and *balanced* resource consumption of the underlying infrastructure. Central to these goals is the careful placement of data replicas and the dissemination of updates.

Previous work on replica placement involves *static* placement of replicas – assuming that clients' distribution and access patterns are known in advance[13, 8]. These techniques ignore server capacity constraints and assume explicit knowledge of the global IP network topology.

Actual Web content distribution requires *dynamic* or *online* replica placement. Most current Content Distribution Networks (CDNs) use DNS-based redirection to route clients' requests [1, 4, 10, 17]. Due to the nature of centralized location services, the CDN name server cannot afford to keep records for the locations of each replica. Thus the CDN often places many more replicas than necessary and consumes unnecessary storage resources and update bandwidth.

For update dissemination, IP multicast has fundamental problems for Internet distribution [5]. Further, there is no widely available inter-domain IP multicast. As an alternative, Application Level Multicast (ALM) tries to build an efficient network of unicast connections and to construct data distribution trees on top of this *overlay* structure [5, 2, 3, 9, 21]. Most ALM systems have scalability problems, since they utilize a central node to maintain state for all existing children [3, 9, 11, 2], or to handle all "join" requests [21]. Replicating the root

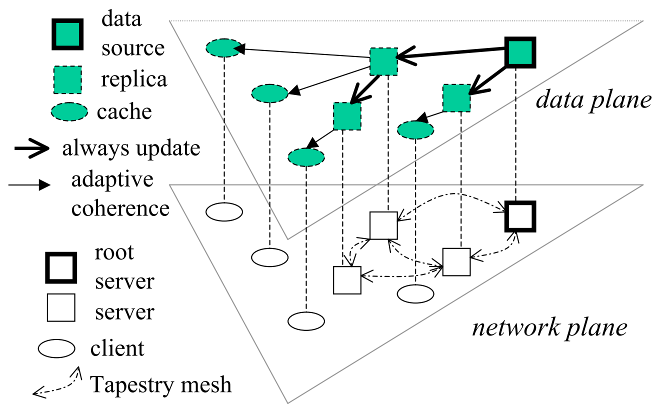

Fig. 1. Architecture of a dissemination tree.

is the common solution [9, 21], but this suffers from consistency problems and communication overhead.

There are two crucial design issues that we try to address in this paper:

1. How to dynamically choose the number and placement of replicas while satisfying QoS requirements and server capacity constraints.
2. How to disseminate updates to these replicas with small delay and bandwidth consumption.

Both must be addressed without explicit knowledge of the global network topology. Further, we would like to scale to millions of objects, clients, and servers.

To tackle these challenges, we propose a new Web content distribution system: *dissemination tree* (in short, *d-tree*). Figure 1 illustrates a d-tree system. There are three kinds of data in the system: *sources, replicas*, and *caches*. The d-tree targets dynamic Web content distribution; hence there is a single source on the Web server. A replica is a copy of source data that is stored on the overlay server and is always kept up-to-date, while a cache is stored on clients and may be stale. These components self-organize into a d-tree and use application-level multicast to disseminate updates from source to replicas. Coherence of caches is maintained dynamically through approaches such as [15]. We assume that d-tree servers are placed in Internet Data Centers (IDC) of major ISPs with good connectivity to the backbone. These servers form a peer-to-peer overlay network called *Tapestry* [20], to find nearby replicas for the clients. Note that Tapestry is shared across objects, while each object for dissemination has a hierarchical d-tree.

We make the following contributions in the paper:

– We propose novel algorithms to dynamically place close to minimum number of replicas while meeting the clients' QoS and servers' capacity constraints.

- We self-organize these replicas into an application-level multicast tree with small delay and bandwidth consumption for update dissemination.
- We leverage Tapestry to improve scalability. Tapestry permits clients to locate nearby replica servers without contacting a root; as a result, each node in a d-tree maintains state only for its parent and direct children.

Note that all these are achieved with limited local network topology knowledge only.

The rest of the paper is organized as follows: We formulate the replica placement problem in Sec. 2 and introduce Tapestry in Sec. 3. Sec. 4 describes the protocols for building and maintaining a d-tree. Evaluation and results are given in Sec. 5, and finally conclusions and future work in Sec. 6.

2 Problem Formulation

There is a big design space for modeling 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 [13], we tackle the replica placement 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 $f(s_i) \leq l_i$.

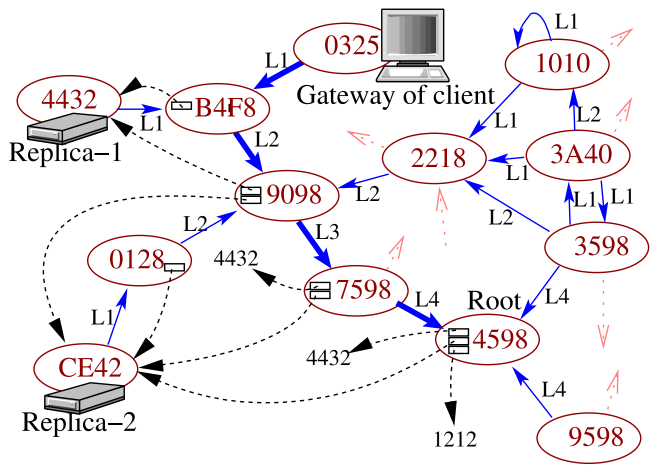

Fig. 2. The Tapestry Infrastructure: *Nodes route to nodes one digit at a time: e.g.* 0325 → *B4F8* → 9098 → 7598 → 4598*. Objects are associated with a particular "root" node (e.g.* 4598*). Servers* publish *replicas by sending messages toward root, leaving back-pointers (dotted arrows). Clients route directly to replicas by sending messages toward root until encountering a pointer (e.g.* 0325 → *B4F8* → 4432*).*

3 Peer-to-peer Location Services: the Tapestry Infrastructure

Networking researchers have begun to explore decentralized peer-to-peer location services [20, 14, 18, 16]. Such services offer a distributed infrastructure for locating objects quickly, with guaranteed success and locality. 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. Our dissemination tree is built on top of Tapestry [20] and takes advantage of two features: *distributed location services* and *search with locality.*

Tapestry is an IP overlay network that uses a distributed, fault-tolerant architecture to track the location of objects in the network. In our architecture (Figure 1), the d-tree servers (i.e., CDN edge servers) and multicast root server (i.e., Web source server) are Tapestry nodes. Each client talks to its nearby Tapestry node (*the gateway*) to send object requests. In practice, the gateway node can be located through certain bootstrap mechanisms.

3.1 Tapestry Routing Mesh

Figure 2 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 [20]. 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 [12].

3.2 Tapestry Distributed Location Service

Tapestry employs this infrastructure for data location. Each object is associated with a *Tapestry location root* through a deterministic mapping function. This root is for location purposes only and has nothing to do with the multicast root server (such as the Web content server in Figure 1). To advertise an object o, the server s storing the object sends a publish message toward the Tapestry location root for that object, depositing *location pointers* in the form of <Object-ID(o), Server-ID(s)> at each hop. These mappings are simply pointers to the server s where o is being stored, and not a copy of the object itself. A node s that keeps location mappings for multiple replicas keeps them sorted in the order of distance from s.

Figure 2 shows two replicas and the Tapestry root for an object. Location pointers are shown as dotted arrows that point back to replica servers. To locate an object, a client sends a message toward the object's root. When the message encounters a pointer, it routes directly to the object. It is shown in [12] that the average distance traveled in locating an object is *proportional* to the distance from that object in terms of the number of hops traversed. In addition, it is proved that for any node c that requests object o, Tapestry can route the request to the asymptotically optimal node s (in terms of the shortest path network distance) that contains a replica of o [12].

4 Dissemination Tree Protocols

4.1 Replica Placement and Tree Construction

In this section, we present an algorithm that dynamically places replicas and organizes them into an application-level multicast tree with only limited knowledge of the network topology. This algorithm attempts to satisfies both client latency and server capacity constraints. Our goal is to minimize the number of replicas deployed and to self-organize the servers with replicas into a load-balanced tree. We contrast static solutions that assume global knowledge of clients and topology.

Dynamic Replica Placement We consider two algorithms: *naive placement* and *smart placement*, for comparison. We describe these as procedures for a new client c to join the tree of object o, possibly generating new replicas in the process. Following the notations in Sec. 2, the latency constraint of c is d_c and the capacity constraint of s is l_s. We define the following notations: current load of s: lc_s; remaining capacity of s: $rc_s = l_s$ - lc_s; overlay distance on Tapestry: $dist_{overlay}$ and IP distance: $dist_{IP}$. As periodically there are "refresh" messages

procedure DynamicReplicaPlacement_Naive(c, o)

1 c sends a "join" request to s with o through Tapestry, piggybacks the IP addresses, $dist_{overlay}(c, s')$ and $rc_{s'}$, for each server s' on the path
2 **if** $rc_s > 0$ **then**
3   **if** $dist_{overlay}(c,\ s) \leq d_c$ **then** s becomes c's parent, **exit**.
    else
4   s pings c to get $dist_{IP}(s,\ c)$
5   **if** $dist_{IP}(\text{s, c}) \leq d_c$ **then** s becomes c's parent, **exit**.
    end
end
6 From the closest one to c, **foreach** *server* s' *on the path* **do**
    search for t that satisfies $rc_t > 0$ **and** $dist_{overlay}(t, c) \leq d_c$
end
7 s puts a replica on t and becomes its parent, t becomes c's parent
8 t publishes o in Tapestry, **exit**.
9 **foreach** *path server* s_i *whose* $rc_{s_i} > 0$ **do** s_i pings c to get $dist_{IP}(s_i,\ c)$
10 c chooses t which has the smallest $dist_{IP}(t, c) \leq d_c$
11 Same as steps 7 and 8.

Algorithm 1: Dynamic Replica Placement (Naive)

going from a child server to its parent for soft state management, we assume that each parent server knows the current remaining capacity of each child server.

Naive placement: Client c sends the request for object o through Tapestry and is routed to server s. For the naive approach, s only considers itself to be c's parent server, i.e., whether $rc_s > 0$ and $dist_{IP}(s, c) \leq d_c$ are satisfied. If unsatisfied, it will try to place a replica on the overlay path server that is as *close* to c as possible (see Algorithm 1). Note that given the limited search, the naive approach may not always find the suitable parent server for every client, even when such a parent exists.

Smart placement: Essentially, the smart approach (Algorithm 2) attempts to optimize the "best" parent selection for c in a larger set: including s, its *parent, siblings* and its *other server children*. Among qualified candidates, c chooses the one with the lightest load as parent. If none of them meet the client's latency and server's load constraints, s will try to place a replica on the overlay path server that is as *far* from c as possible. We call it *lazy placement*. All these steps aim to distribute the load with the greedy algorithm to reduce the number of replicas needed while satisfying the constraints.

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

procedure DynamicReplicaPlacement_Smart(c, o)

1 c sends a "join" request to s with o through Tapestry

2 s sends c's IP address to its parent p and other server children sc **if** $rc_{sc} > 0$

3 p forwards the request to s's siblings ss **if** $rc_{ss} > 0$

4 s, p, ss and sc send c its rc **if** its $rc > 0$

5 **if** *c gets any reply* **then**

6   c chooses the parent t which has the biggest rc and $dist_{IP}(t, c) \leq d_c$, **exit**.

else

7   c sends a message to s through Tapestry again and the message piggybacks the IP addresses, $dist_{overlay}(c, s')$ and $rc_{s'}$ for each server s' on the path

8   From the closest one to s, **foreach** *server s' on the path* **do**

        search for t that satisfies $rc_t > 0$ **and** $dist_{overlay}(t, c) \leq d_c$

    end

9   Same as steps 7, 8 and 9 in **procedure** DynamicReplicaPlacement_Naive.

10   c chooses t which has the biggest $dist_{IP}(t, c) \leq d_c$

11   Same as step 11 in **procedure** DynamicReplicaPlacement_Naive.

end

Algorithm 2: Dynamic Replica Placement (Smart)

Sec. 5. The tradeoff between the smart and naive approaches is that the smart one 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 Sec. 5.

Static Replica Placement 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 Sec. 2 can be converted to a special case of the capacitated facility location problem [7] 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 [7]. 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 $s \in S$ which has the largest value of min(cardinality $|C_s|$, remaining capacity rc_s)
    $S' = S' \bigcup \{s\}$
    $S = S$ - $\{s\}$
    if $|C_s| \leq rc_s$ **then** $C = C$ - C_s
    else
        Sort each element $c \in C_s$ 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 $\forall\ c \in C$
end
return S'.

Algorithm 3: Static Replica Placement with Distributed Load Balancing

We consider two types of static replica placement: with only overlay path topology vs. with global IP topology. For the former, to each client c, the root only knows the servers on the Tapestry path from c to root which can cover that client (in IP distance). On the other hand, the latter assumes the knowledge of global IP topology and gives close-to-optimal bound on the number of replicas.

4.2 Soft State Tree Maintenance

The liveness of the tree is maintained using a soft-state mechanism. Periodically, we send "heartbeat" messages from the root down to each member. We assume that all the nodes are loosely synchronized through the Network Time Protocol (NTP) [6]. Thus if any member (except the root) gets the message within a certain threshold, it will know that it is still alive on the tree. Otherwise it will time out and start rejoining the tree. Meanwhile, each member will periodically send out a "refresh" message to its parent. If the parent does not get the "refresh" message within a certain threshold, it will kick out the child's entry.

5 Evaluation

In this section, we evaluate the performance of our d-tree algorithms. We use the GT-ITM transit-stub model to generate five 5000-node topologies [19]. The

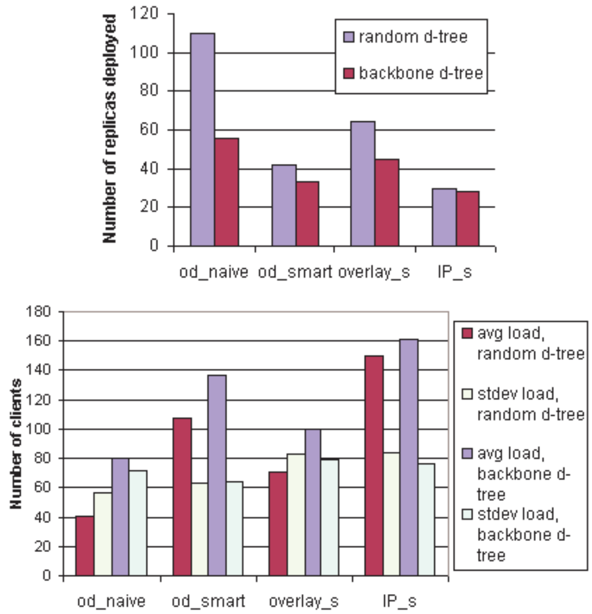

Fig. 3. *Number of replicas deployed (top) and load distribution on selected servers (bottom) (500 d-tree servers).*

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.

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

We couple the server placement with four different replica placement techniques: overlay dynamic naive placement (*od_naive*), overlay dynamic smart placement (*od_smart*), overlay static placement (*overlay_s*), and static placement on IP network (*IP_s*). 500 nodes are chosen to be d-tree servers with either "random" or "backbone" approach. The rest of nodes are clients and join the d-tree in a random order. We randomly choose one non-transit d-tree server to be the multicast source and set as 50KB the size of data to be replicated. Further, we assume the latency constraint is 50ms and the load capacity is 200 clients/server.

In the following, we consider three 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. A smaller ratio implies better load distribution.

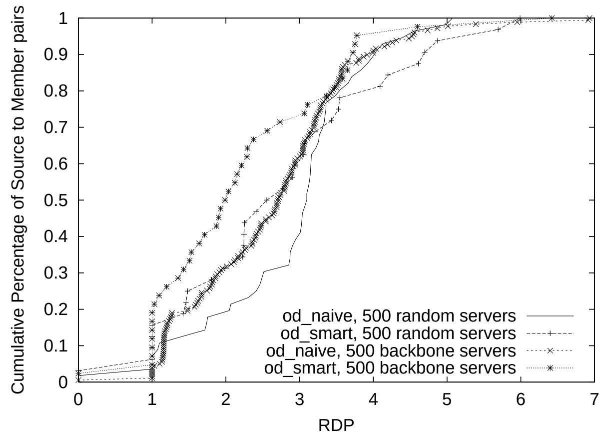

Fig. 4. *Cumulative distribution of RDP with various approaches (500 d-tree servers).*

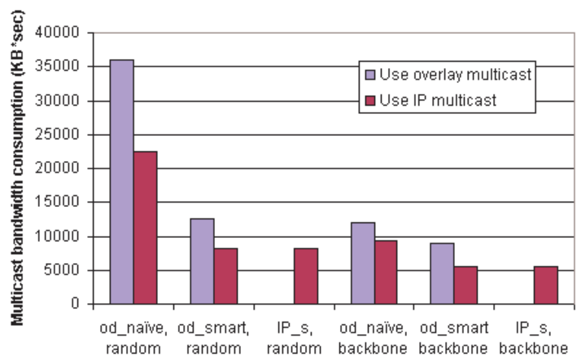

Fig. 5. *Bandwidth consumption when multicast 1MB update data (500 d-tree servers).*

- **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.
- **Tree construction traffic:** We count both the number of application-level messages sent and the bandwidth consumption for constructing the d-tree.

Figure 3 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 d-tree*.

In Figure 4, *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. Figure 5

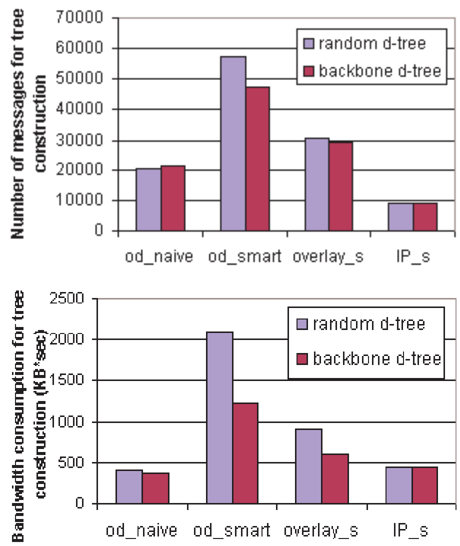

Fig. 6. *Number of application-level messages (top) and total bandwidth consumed (bottom) for d-tree construction (500 d-tree servers).*

contrasts the bandwidth consumption of various d-tree construction techniques with optimal IP placement. The results are very encouraging: the bandwidth consumption of *od_smart* is quite close to the optimal *IP_s* and is much less than that of *od_naive*.

The performance above is achieved at the cost of d-tree construction (Figure 6). However, for both *random* and *backbone d-tree*, *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 replica placement has a close-to-optimal number of replicas, 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. Usually, tree reconstruction is a much less frequent event than Web data access and update. Further, its performance is quite close to the ideal case: static placement on IP multicast. Hence, the "smart approach" is more advantageous.

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. And the experiments show that the naive scheme has many more uncovered clients than the smart one, due to the nature of its unbalanced load.

6 Conclusions and Future Work

In this paper, we explore techniques for building the dissemination tree, a dynamic content distribution network. First, we propose and compare several replica placement algorithms which reduce the number of replicas deployed and self-organize them into a balanced dissemination tree. Second, we use Tapestry, a peer-to-peer location service, for better scalability and locality. In the future, we would like to continue evaluation with more diverse topologies and workloads, add dynamic replica deletion to d-tree, and investigate how to build a better CDN with other peer-to-peer techniques.

7 Acknowledgments

We graciously acknowledge sponsorship and grants from DARPA (grant N66061-99-2-8913), California Micro Grant #01-042, Ericsson, Nokia, Siemens, Sprint, NTTDoCoMo and HRL laboratories. We thank Hao Chen, Matthew Caesar and Chen-nee Chuah for reviewing the draft of the paper and thank the anonymous reviewers for their valuable suggestions.

References

1. Akamai Technologies Inc. http://www.akamai.com.
2. Y. Chawathe, S. McCanne, and E. Brewer. RMX: Reliable multicast for heterogeneous networks. In *Proceedings of IEEE INFOCOM*, 2000.
3. Y. Chu, S. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of ACM SIGMETRICS*, June 2000.
4. Digital Island Inc. http://www.digitalisland.com.
5. P. Francis. Yoid: Your own internet distribution. Technical report, ACIRI, http://www.aciri.org/yoid, April, 2000.
6. J. D. Guyton and M. F. Schwartz. Experiences with a survey tool for discovering network time protocol servers. In *Proc. of USENIX*, 1994.
7. K. Jain and V. Varirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation. In *Proc. of FOCS*, 1999.
8. S. Jamin, C. Jin, A. Kurc, D. Raz, and Y. Shavitt. Constrained mirror placement on the internet. In *Proceedings of IEEE Infocom*, 2001.
9. J. Jannotti et al. Overcast: Reliable multicasting with an overlay network. In *Proceedings of OSDI*, 2000.
10. Mirror Image Internet Inc. http://www.mirror-image.com.
11. D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An application level multicast infrastructure. In *Proceedings of 3rd USITS*, 2001.
12. C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of the SCP SPAA*, 1997.
13. L. Qiu, V. N. Padmanabhan, and G. Voelker. On the placement of web server replicas. In *Proceedings of IEEE Infocom*, 2001.
14. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, 2001.

15. P. Rodriguez and S. Sibal. Spread: Scaleable platform for reliable and efficient automated distribution. In *Proceedings of WWW*, 2000.
16. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware 2001.*
17. Speedera Inc. http://www.speedera.com.
18. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of ACM SIGCOMM*, 2001.
19. E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proceedings of IEEE INFOCOM*, 1996.
20. B. Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. UCB Tech. Report UCB/CSD-01-1141.
21. S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of ACM NOSSDAV*, 2001.