An Effective Pointer Replication Algorithm in P2P Networks

Jian Zhou, Laxmi N. Bhuyan and Anirban Banerjee
University of California, Riverside
{jianz, bhuyan, anirban}@cs.ucr.edu

Abstract

Peer-to-Peer (P2P) networks have proven to be an efficient and successful mechanism for file sharing over the Internet. However, current P2P protocols have long worst case query latencies which prevents them from being employed for real time applications. Popularity of objects in these networks can change rapidly and augurs the need for a rapid and lightweight content replication strategy to reduce search and data-access latencies.

In this paper, we propose an On-line Pointer[1] *Replication (OPR) algorithm in structured P2P networks which yields a significantly low worst case query latency. Also, the degree of replication achieved by OPR is dynamically adaptable to the instantaneous query arrival rate and churn characteristics of the system in order to reduce total control traffic. We evaluate and compare different replica placement strategies on the PlanetLab network as well as with simulations. Experimental results show that OPR outperforms the existing replica placement algorithms by at least 30% in average latency and around 40% in terms of maximum query latency.*

1 Introduction

The recent emergence of Internet-scale distributed systems including storage administration in global companies, entertainment file sharing, and large distributed database systems has led to extensive research on efficient and scalable distributed file sharing architectures. P2P systems, among many other distributed computing models, exhibit good scalability and stability. They have proved to be an efficient mechanism for decentralized file sharing over the Internet. The structured overlay networks are designed to enable scalable P2P file sharing without any centralized control. They enable the file sharing scheme to scale automatically with increasing number of peers. They are robust and self-organizing since they are able to adapt to arrival and departure of nodes with relatively low cost. Due to their potential efficiency, robustness and scalability, the structured P2P networks have been employed to support a variety of applications such as persistent storage [19], query processing [17], domain name services [30], and communication services [36]. A variety of other applications have also demonstrated the significance of structured P2P in large scale distributed systems [8, 28].

Recent research has uncovered that query messages contribute nearly 118TB/month of Gnutella traffic [26]. This massive amount of control messages perform only one function: to locate resources. This initiates an interesting research in P2P networks: locating data sources efficiently across a large number of participating peers to reduce the overall query traffic [22]. This need has forced P2P networks to evolve from Napster [4] based models, wherein a centralized server tracks the pointers of all the data in the network to more decentralized models as GNUtella [3] and BitTorrent [1]. The dynamic nature of peer lifetimes amounts to frequent changes in pointers for data in these networks. This is a natural motivation to devise mechanisms which can satisfy search queries efficiently while incurring low overheads.

An obvious solution is to partition and distribute the pointers among several peers in the network. This mechanism has been deployed in both unstructured [1] and structured P2P networks [29]. In [1], trackers are used to keep a list of potential nodes where the desired data is likely to be found. While in [29], a consistent hash function is used to map the data to its server (root)[2] which stores a pointer to the data. This scheme enables the file sharing scheme to scale automatically with increasing number of peers. However, the search latency may not satisfy client-perceived bounds for large-scale networks [25]. Further, long worst-case latencies prevent these P2P techniques from being employed by latency sensitive applications such as DNS and VOD (video on demand). Another drawback of these schemes is that they suffer from a single point of failure.

Replication algorithms are an attractive option to solve the above problem by placing multiple copies of pointers in the network. For each object, the roots which maintain the pointers are known to all other peers in the network. When a query arrives at any peer in the network, the query is forwarded to the nearest root to ob-

[1]*Pointer* is defined as the location-information of an object.

[2]The *root* of an object is the peer which keeps the location information of that object.

tain the pointer. Recently, a number of replication strategies [12, 21] have been proposed to replicate the objects in the P2P network to reduce the object access time. Our replication framework is different from these approaches since we replicate pointers instead of objects. Objects are usually of much larger size than pointers and incur larger overhead in terms of memory space and replication bandwidth. In current file-sharing P2P systems where both memory space and bandwidth are precious resources, pointer replication is found to be more promising. Some of the questions we intend to answer follow below:

- How much benefit does pointer replication provide over object replication?
- When should a new replica be spawned?

We must mention that we are aware that pointer replication may incurs an update cost when the *home node*, which hosts the actual object data, joins or leaves the network. Also, in order to maintain the freshness of pointer information, messages are transmitted to roots for updating pointer information. Naturally, with multiple roots the cost to update pointer information is increased. In this paper, we propose an On-line Pointer Replication (OPR) algorithm for structured peer-to-peer networks which can efficiently reduce the query search latency while incurring low update overhead.

Briefly, our pointer replication framework displays the following salient features.

- **Low Latency**: The OPR achieves near minimum worst-case query search latency given the number of replications. We prove that OPR can choose the best locations of the replications in polynomial time while guarantying the object search latency within a factor of 2 of the optimal solution.
- **Low Overhead**: OPR generates minimal control traffic[3] to update all the replicas and to locate data objects.
- **High Scalability**: OPR protocol works without any centralized oracle and does not need complete knowledge of the whole network.

The paper is organized as follows. In Section 2, we survey related work followed by section 3 describing the background and problem formulation. Section 4 introduces our replication placement strategy, OPR followed by section 5. Section 6 and 7 evaluate OPR and compare it with various replication algorithms via real-world implementation and simulation. Finally, section 8 concludes the paper.

[3]Here, the control traffic consists of query messages and update messages.

2 Related Work

In this section, we survey related work on replication algorithms in P2P networks. We find two classes of replication mechanisms, namely *reactive* and *proactive* replications.

In reactive replications, as objects are transferred from the home node to the requesting peer, intermediate nodes through whom the data flows, determine independently whether or not to cache the object-content. P2P web caching is an example of such a mechanism [9, 33]. In the context of reactive pointer replication, some mechanisms proposed to cache pointers in order to yield better query search performance [31, 20]. In DiCAS [31], queries are forwarded to peers in a predefined group which passively cache the pointers in unstructured P2P network. However, it would incur a large overhead to update the pointers when the object is moved. When an object is moved or deleted, the updated location information has to be flooded to the whole overlay network. One intuitive approach to decrease the object search latency is to cache pointers on intermediate nodes from the home node to the root. When a query encounters an intermediate node on its way to the root the search latency is cut down since the average route length is reduced. This mechanism is proposed in Plaxton's paper [20], where all the intermediate nodes between the home node and root node contain copies of the pointers, as shown in Fig. 1. Anyhow, our proactive replication algorithm can also be supplemented with intermediate pointers, as described above, to further boost the performance.

In proactive replication, contents are pushed to selective peers by the home node in pursuit of better performance in terms of query latency, loss-rate, and load balance etc. There are several works in the object replication in P2P networks [10, 15, 11, 12, 21]. However, the cost of replicating entire objects can be cumbersome in both disk space and bandwidth, particularly for systems that support applications with large objects(e.g., audio, video, software distribution). In [12], the author explores the optimal replication strategy according to the object popularity in unstructured P2P networks. However, it is not suitable to the pointer replication as it does not take the update cost into account. In Beehive [21], replication strategies are designed to achieve a constant lookup performance in average under the assumption that the object popularity follows a power law distribution. Nevertheless, it is not an effective way to reduce the worst-case search latency for all objects. Further, the replication placement algorithm in [21] needs a couple of hours to adapt to the object popularity changes.

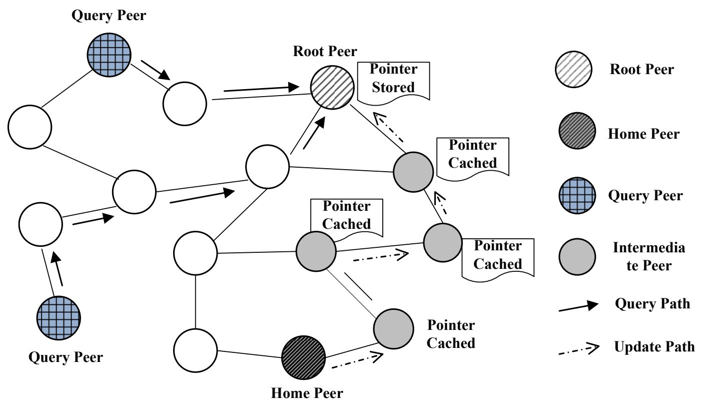


Figure 1. Illustration of Intermediate Pointer

3 Problem Formulation

We consider a set of N nodes in a large distributed network each stores various objects like video files, web pages or documents. Queries requesting a specific object may originate from a node at any time. Once the object location is known to the querying node, the query is considered successful. In order to facilitate the search for a specific object, a location-pointer of the object is kept at a peer (root) in the network. Here we use root node to denote the peer who stores the pointer to the object. Thus, all queries for that object are forwarded to the root node to fetch the location pointer. Latency incurred by a query is denoted by the number of hops it takes to route to the root. This has been clearly demonstrated in Fig. 1.

To reduce the query search latency as well as improve data availability, we propose to replicate the pointers in the networks. In doing so, we identify the tradeoffs between performance gains and the cost paid for replica maintenance. Due to the limited resources available, our replication protocol has to address the following two problems.

- **Placement of Replicas**: Given a number of replicas, how to place the replication pointers in the network to achieve the best performance? We evaluate performance with two metrics, the first being search query latency and the other is resource-overhead.

- **Extent of Replication**: In a given network environment, how to determine the replication degree for each object to achieve the best performance? We use network environment to depict the network condition in terms of bandwidth and latency as well as the dynamic nature of the nodes.

Formal definition: Given the replication number k and the default root node r_1 of the object, we need to find the best $k-1$ replica nodes, denoted as $R = r_2, ...r_k$, such that the worst case latency is minimized. With the knowledge of all the replica nodes R, each query is able to route to the nearest r_i ($i \in [1,k]$) to fetch the pointer. Effectively, the network is actually a forest of k trees with each tree rooted at one replica node r_i. Thus all the k roots cooperatively serve the queries for the object and the workload is distributed among them. Our goal is to identify the best replica placement R to minimize the maximum latency among all the queries.

Thus, given a graph $G = (V, E)$ representing an overlay network topology, given an integer k and given any node $r_1 \in V$, the goal is to compute a subset of $k-1$ vertices $R \subseteq V$, such that the maximum distance between any vertex $v \in V$ and its nearest center $r_i \in R$ is minimized. The objective is to minimize equation 1.

$$\max\{\min\{d(r_i, v) \quad \forall r_i \in R\} \quad \forall v \in V\} \qquad (1)$$

The more replicas we instantiate, the lower the search latency. However, with increasing number of replicas, the system consumes more memory resources and increases communication cost for pointer updates. Given a network environment, the replication degree is a tunable parameter which can balance the performance gain and the resource overhead introduced by the replication.

The overhead involved in replication can be classified in two parts $Cost_{mem}$ and $Cost_{maintenance}$. $Cost_{mem}$ is the memory space overhead and $Cost_{maintenance}$ is the communication overhead. Compared with the small memory requirement for a pointer, the communication overhead to keep the pointer up-to-date is more significant. Further, the communication overhead incurred by the replication consists of two parts $Cost_{publish}$ and $Cost_{update}$. $Cost_{publish}$ is the cost to publish the point-

```
Greedy Approach
Input Parameters:
    (1) Underline Topology: $G = (V, E)$
    (2) Number of Duplications: k
    (3) The Original Root: $r_1$
Output Parameters:
    (1) A set of k Roots: $R$
begin
    (1) For each $v$ in $V$;
            $d[v]$ = INFINITY;
            $R = \{r_1\}$;
    (2) For i = 2 to k;
            If $d[u]$ is maximum;
                $r_i = u$;
                $R = r_i \bigcup R$;
            For each $v$ in $V$;
                $d[v]$ = minimum $D(r_i, v)$, $r_i \in R$
                $D(r_i, v)$ is the distance from $r_i$ to $v$
    (3) return $R$
end
```

Figure 2. Greedy Approach of Selecting k Roots

ers and $Cost_{update}$ represents the cost to update the pointers. Publish denotes the process when a peer joins the network or a new object is made available for sharing by the home node which contacts the root node to register object-pointers. In this framework, we propose a replication strategy that minimizes the communication cost.

$$Cost_{maintenance} = Cost_{publish} + Cost_{update} \quad (2)$$

4 OPR: On-line Pointer Replication Placement

In this section, we describe the OPR framework which can effectively place the replication pointers. We start with a topology aware approach assuming complete knowledge about the network layout and prove the effectiveness of the algorithm. Subsequently, we apply OPR to a structured P2P topology without the global knowledge of the network and prove its efficacy.

4.1 Topology Aware Approach

In this subsection, we describe the heuristic approach of the algorithm to find the best replica placement, analyze the time complexity of the algorithm and prove that this approach produces results within a factor of two of the optimal solution.

Intuitively, the k roots should be selected as far apart from each other as possible. With this in mind, starting from the original root r_1, we use a greedy approach to select the $k - 1$ replica nodes one by one.

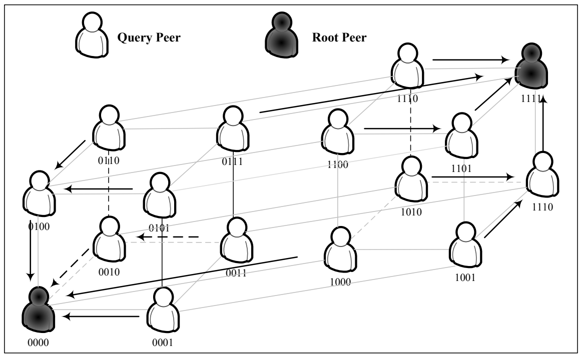
Greedy approach: we compute the shortest distances between the original root r_1 and all the other vertices in graph G. Then, we consider the vertex farthest from the first root r_1, designate it as the next root r_2 and compute the shortest distances from each vertex in the graph to the nearest vertex in $\{r_1, r_2\}$. Now, we consider the vertex with the maximum shortest-distance to be the next root r_3 and recursively proceed until all k roots have been selected. In each iteration of our greedy approach, there may be several nodes in the network having the same maximum distance to the closest root in R. We solve this tie by randomly selecting one of them. Fig. 2 shows the pseudo code for the greedy algorithm.

Given the network topology, the complexity of our root selection algorithm is $O(k(N + E)logN)$, where E denotes the number of edges in the overlay network. It is easy to see that each step of the algorithm includes a multiple-source shortest-path problem, which can be solved by the Dijkstra's algorithm [13]. The run time of Dijkstra's algorithm is $O((N+E)logN)$ and our greedy approach is k times the run time of Dijkstra's algorithm. When the N^2 node to node distance is given, the time complexity of our root selection algorithm is only $O(k)$.

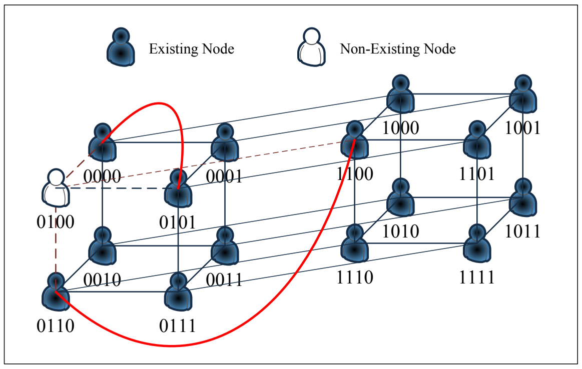
From the analysis, we deduce that the greedy algorithm executes fairly fast. Also, for any heuristic approach, it is important to know how well it can approximate the optimal solution. Let d_{opt} denote the optimal solution and d_g denote the maximum latency getting from our greedy approach, we can prove the bounds for our approach using methods outlined in [14]. The greedy approach is a 2-approximation algorithm of the NP-hard problem, that is $d_g/d_{opt} \leq 2$.

4.2 Decentralized Approach

A natural concern with a topology aware approach is that the global knowledge of the network layout may not be available in a large-scale P2P network. Also, it may incur large overheads to keep the topology information updated on each individual node in a dynamic environment. The large amount of control traffic generated to propagate topology changes may impact the scalability of the systems. However, in a structured P2P network, we can address this problem easily since the layout of a structured P2P network can provide the information to find the best placement of the root nodes. In a structured P2P network, the overlay topology of the network is constructed based on some interconnection topologies such as hypercube [20, 27, 35], generalized *k-ary-d-cube* [23], and Moore graph [16] and etc. Without loss of generality, we investigate the widely used hypercube topology in this paper.



(a) Illustration of Two Roots



(b) Illustration of Non-existing Node

Figure 3. Illustration of On-line Pointer Replication

Hypercube overlay has been widely used in the area of distributed and parallel computing as it features a small inter-node distance(diameter) and good fault-tolerance characteristics due to a large number of alternate paths existing between pairs of nodes. These are desirable properties in a P2P scenario. In a static network, where the network size is $N = 2^n$, a hypercube can be easily embedded by connecting any two nodes that are one hamming distance apart. Here, hamming distance of two nodes is the number of bits that are different in IDs of two nodes. For example, ID 0000 and 1110 differs in all positions except the last bit, so their hamming distance is 3. In this topology, each node in the hypercube is connected to $log_2 N$ neighbors, where N is the total number of nodes. With such a setup, it is easy to compute the difference between two nodes given their node IDs.

Peers in the P2P networks are known to be unstable. They may join and leave the network arbitrarily. Thus, the network size N is not necessary to be 2^n in most P2P networks. We assume that network size does not expand and shrink frequently, which is a reasonable assumption in structured P2P networks. Now, we select a hypercube with size 2^n where $2^{n-1} \leq N < 2^n$. When the number of nodes is less than 2^n, we might not be able to construct the complete hypercube overlay topology. Thus, to keep the hypercube well connected, neighbors of the missing node are connected with each other. As shown in Fig. 3(b), in a hypercube with 2^4 nodes, node with ID 0100 is missing. Its neighbor 0000 is connected to 0101 and 0110 is connect with 1100. Thereby, all the nodes in the system still have $log_2 N$ neighbors and queries can be routed properly within the network.

Using the greedy approach, the node with the largest distance to the nearest existing roots is selected as the next root. As the hamming distance represents a distance metric, we can easily identify the farthest node in the system. For example, in a network with sixteen nodes, the farthest node from peer 0000 should be the peer with complementary node ID 1111 because of a complete bit-inversion at all the positions. Thus, with one root fixed at r_1, the secondary root r_2 can easily be identified by converting all the bits in r_1, so $r_2 = \bar{r_1}$. The distance between r_1 and r_2 is $log_2 N$ which is also the diameter of the hypercube. For any node ID id in the network, let d_1 and d_2 denote the distance between id and r_1, r_2 ($\bar{r_1}$) respectively. Interestingly, the sum of d_1 and d_2 always equals $log_2 N$. This means with two roots located at r_1 and $\bar{r_1}$, nodes are able to route to the nearer one within $\lfloor \frac{log_2 N}{2} \rfloor$ hops. The hypercube is actually a double rooted tree. In an incomplete hypercube, less nodes in the system may in turn result in less hops to reach the destination. Fig.3(a) shows example with two root nodes in four dimensional sixteen node hypercube. Given $k = 2$, the worst case distance in our OPR framework is $\lfloor \frac{log_2 N}{k} \rfloor$.

When $k = 4$, according to the greedy approach, the next root should be the node with the maximum distance to the nearest root in 0000,1111. In this case, nodes with ID 0011, 0101, 1001, 0110, 1010 and 1100 are all with two hamming distance to either 0000 or 1111. Generally speaking, any of them are qualified to be the next root. In our case, we use 0011 and 1100 which is numerically nearest to 0000 and 1111 respectively.

In a scalable P2P network, every node uses the local information to route the incoming queries to the destination. In order to reach the destination, queries are forwarded to one of the neighbors whose IDs are progressively closer to the destination node D.

In OPR, each query is routed to the neighbor that has the least hamming distance with the destination node. Each node has a neighbor table (routing table) with $log_2 N$ entries. And each neighbor is one hamming distance away from the node. With $log_2 N$ neighbors, any node can find a neighbor which is at least one hop nearer to the destination. Thus, the destination can be reached within log_2N hops. When nodes are missing in the hypercube, they are connected to the nodes which are two hops away from it. Thus, queries can be routed to the destination even faster.

When a query cannot find any node in the routing table that is nearer to the destination node, it indicates that the destination node is not available in the system. The query will instead be forwarded to a node which is numerically closest to the destination node. Eventually, the query will reach the node which is numerically nearest to the destination node in the system where the data is located.

5 Replication Strategy

In the previous section, we discussed where to place the replica pointers. Naturally, the larger number of replicas, the better performance we can obtain. However, more replicas could also introduce more communication overhead in the system which may degrade the performance. An optimal replication strategy helps maximize the network resource utilization and in turn minimizes the absolute resource demand. In this section, we will demonstrate how to achieve such an optimal replication strategy in a hypercube based structured P2P network.

5.1 Resource Analysis

Both query messages and the system-maintenance messages consume network resources to an extent. We term the union of these as *total control messages*. In the replication framework, system-maintenance messages are used to keep the pointers up-to-date. To best utilize the limited resources, we propose an optimal replication strategy with the minimal resource consumption.

In a network with N nodes and replication degree k, the cost for each query can be calculated by Equation 3, where S_q denotes the size of the query message.

$$Cost_{query} = (\frac{\lfloor log(N-1) \rfloor + 1}{2} - logk + 1) * S_q \quad (3)$$

Our replication algorithm is designed to obtain the minimum worst-case query search latency for all objects in the system such that the P2P system could be employed for latency sensitive applications. We use the same replication degree for all objects for mathematical ease.

Overhead analysis: By replicating pointers on multiple roots, we introduce communication overhead in the system. When a new object is inserted into the network, instead of routing to the original root r_1 to publish the object pointer, we need to route to all the k roots. This also happens when the object is moved or deleted from the system. This implies that the communication overhead to initialize the pointer information increases by a factor of k. However, by paying this cost for initialization we benefit significantly during the query-phase. The cost for publishing and updating multiple pointers are the same and are calculated via Equation 4 where S_m denotes the size of the publish/update message.

$$Cost_{maintenance} = ((\lfloor log(N-1) \rfloor + 1) + k - 2) * S_m \quad (4)$$

The total cost of the system is calculated by Equation 5 where R_q denotes the query arrival rate and R_n denotes the number of objects inserted or deleted from the systems per second.

$$Cost_{total} = Cost_{query} * R_q + Cost_{maintenance} * R_n \quad (5)$$

From these equations, it is clear that the total cost incurred by queries reduces as the number of replications increases. To take full advantage of the benefit of this replication scheme, it is required to find the optimal replication degree such that the total cost is minimized. We can rewrite the Equation 5 by substituting $Cost_{query}$ and $Cost_{maintenance}$ by Equation 3 and Equation 4 respectively.

$$Cost_{total} = \{(\frac{\lfloor log(N-1) \rfloor + 1}{2} - logk + 1) * S_q\} * R_q + \\ \{((\lfloor log(N-1) \rfloor + 1) + k - 2) * S_m\} * R_n$$

To get the optimal replication degree k, we differentiate Equation 5 for k. We can compute k which satisfies $\frac{dCost_{total}}{dk} = 0$ and $\frac{dCost_{total}^2}{dk} > 0$. Finally, we obtain the optimal k requiring minimal communication cost as shown in Equation 6.

$$k = \frac{S_q * R_q}{\ln 2 * S_m * R_n} \quad (6)$$

The query message is usually of the same size as the pointer maintenance message. Therefore, Equation 6 can be further simplified as follows:

$$k = \frac{R_q}{\ln 2 * R_n} \quad (7)$$

From the above equation, we conclude that **the optimal replication degree is directly proportional to the query arrival rate and inversely proportional to the system churn rate**. In OPR, we adapt the replication degree dynamically according to the estimation of the query arrival rate and the system churn rate. The query arrival rate and the system churn rate can be estimated using the sampling technique proposed in [6].

5.2 Addressing Peer Churn

In a dynamic environment, it is important to address node churn. When nodes join the network, they try to construct their routing tables by contacting other nodes in the network. Also, the new incoming nodes propagate their existence to other nodes in the network. When nodes leave the network, we need to make sure that the pointers they store are transferred over to other nodes in the network to ensure consistent data availability. Also the nodes which registered the leaving node in their neighbor tables need to update their routing tables.

When a node, say P, leaves the network, it informs its neighbors about its intention, along with the corresponding *replacement node* for updating route entries. Upon receiving the exit message from node P, the neighbors update their routing entries. The pointers kept on node P are pushed to the neighbor which has the ID that is numerically nearest to P.

However, in a dynamic and failure-prone environment, nodes may just become unreachable due to node and link failure which partition the network. Many mechanisms have been proposed to monitor node availability, update routing tables and find back the lost pointers stored on the failed node. These methods are complex and time-consuming and before the system gets updated, it may result in routing failures and querying failures for acquiring the lost data (pointers). In other words, the static resilience of the design is quite important. Here, we define static resilience as how well routing can continue to function as nodes fail and other nodes try to establish connections to compensate.

Replication algorithms are known to enhance the performance, improve the availability of the data and resist to node failure [7]. With multiple roots exiting in the system, the pointers of the objects are always available in the system unless all the multiple roots fail simultaneously. A query will be first routed to the nearest root asking for the pointer. If this root is unable to respond, it is assumed to be failed. Then the query will be sent to the nearest root among all other active roots. This step will be repeated until the pointer is fetched.

6 Experiments On PlanetLab

To verify the efficiency and effectiveness of our replica placement algorithm, we conduct experiments on PlanetLab [5], a global overlay network designed by and for the research community.

Setup: We secure 100 geographically distributed nodes connected by a diverse collection of links to form a corpus. For our experiment, we use 16 nodes to form our network as a hypercube overlay. For each run, we choose 16 nodes from our corpus, based on CoMoN [2] data which allows us to rank nodes based on their relative stability and available bandwidth and CPU resources. We select the best 16 candidates from the dataset and implement our hypercube structure atop it. We inject queries for different objects in this structured network in order to observe search latencies. Unless otherwise noted, each object has the same probability to be queried (Random Workload). Also, each node in the network also has the same probability to issue a query. The arrival time between consecutive queries follows a Poisson distribution (i.e. $p(x, \lambda) = e^{-\lambda} * (\lambda)^x/(x!)$), where λ is the mean value and x is a non-negative integer.

The following performance measures are used in the experiments,

- *Average Query Latency:* The average latency seen by a query in the network to get the pointer.
- *Maximum Query Latency:* Among all the queries, the maximum latency for one successful query.

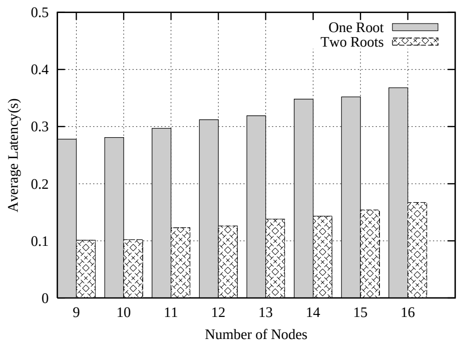
To highlight the effectiveness of our algorithm, we compare our algorithm with three other replication algorithms as follows,

Competing schemes: One intuitive approach is to randomly place the replications and forward the query to the nearest replica, we term this as RMP(Random Placement). We also compare our scheme with Beehive where replications are placed on all nodes that logically precede the root on all querying paths. Besides RMP and Beehive, we also put duplicate pointers on some intermediate nodes, called Intermediate Pointer Placement (IPP). As a result, a query may encounter an intermediate node with the requested pointer on its way to the root. Thus, the average route length for searching the pointer is cut down, and the workload for the whole system is reduced as it prevents the query from being forwarded to the rest of the nodes on the path. This mechanism is proposed in Plaxton's paper [20], where all the intermediate nodes between the home node and root node contain copies of the pointers. Again, the home node is one which has a copy of the original object and the root is the node to which the object maps.

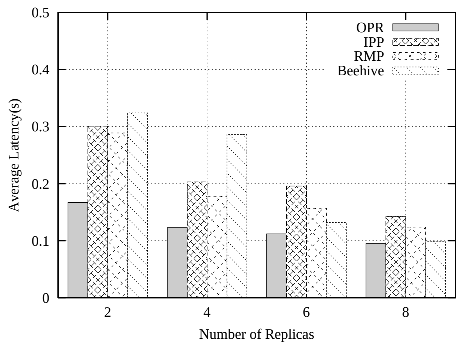
Table 1. Latency with Intermediate Pointer Caching

Num of nodes	Traffic Intensity(Msg/s)	Max(s)	Avg(s)
8	10	1	0.14
8	100	2	0.21
8	1000	3	0.24
8	100000	10	0.33
16	10	1	0.26
16	100	2	0.31
16	1000	6	0.33
16	100000	21	0.41

The OPR scheme can be supplemented with intermediate pointers (IPP) to further boost the performance. However, we would like to ascertain whether it is more beneficial to store these pointers on intermediate nodes or replicated root nodes. Therefore, we fix the number of replications to k and compare both cases. Let p be the number of nodes on the path from the home node to the root. When $p > k$, we replicate the pointers on the nodes which lie on the path to the root and are near it too. This is because pointers if placed near the root are logically situated on the higher levels of the tree and may satisfy queries with higher probability. While in the other situation where $p <= k$, we put the $k - p$ extra duplications on the neighbors of the root. So the total number of pointer duplications is k including multiple roots and intermediate pointers. We summarize the competing schemes below:



(a) Average Case Latency



(b) Difference Replication Placement

Figure 4. Experimental Results in Planetlab

- *Random Plancement(RMP) [23]:* Randomly select several nodes in the network as roots that store replicas and each query is forwarded to the nearest replica.
- *Intermediate Pointers Placement(IPP) [20]:* With one root, pointers are also kept on the intermediate nodes which happen to be on the path from the original file location (home node) to the root of the tree.
- *Beehive [21]:* The replicas are instantiated on all nodes logically preceding the root on all query paths.

To re-iterate, object pointers are stored on paths from root to the actual location of the object. We measure amount of time needed for a query to reach the final node which stores the actual object. In Fig. 1, we depict a typical scenario and measure time beginning at query insertion point (grey node, bottom left) to the final object location node (dark node, bottom right). The latency results are presented in table 1. We also present Fig. 4 which depicts the efficacy of OPR versus other replication schemes. We find that the object-search latency decreases by nearly 60% by augmenting the number of roots, as shown in Fig. 4a . Further, in Fig. 4b we observe that OPR performs significantly better in terms of search latency compared to RMP, IP and Beehive.

7 Simulation Results

In order to evaluate the performance of our algorithm with large network sizes, we implement an event-driven simulator to emulate the algorithm behavior on a variety of network topologies and web workloads. Additionally, the robustness of our algorithms is evaluated by varying the failure rate of nodes in the network.

Simulation setup: The experiment al network consists of 4096 nodes conforming to a GT-ITM generated Transit-Stub (TS) topology [34]. Each node is assigned a unique node ID randomly when it enters the network. TS models the network using a two-level hierarchy of routing domains, with transit domains interconnecting the lower level stub domains. By default, the latency of intra-transit domain links, stub-transit links and intra-stub domain links are set to 20ms, 5ms and 2ms respectively [24]. There are 100,000 different objects distributed randomly in the network, with each object having a unique object ID in an identifier space of 2^{128}.

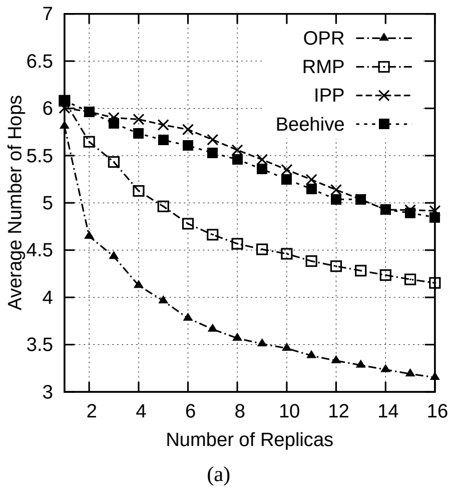
In the following subsections, we study the performance of our algorithm against various network and workload configurations. Effects of varying the number of duplications, workloads and node failure rate are observed. The effectiveness of our algorithm in improving the system's load balance caracteristics is also demonstrated.

7.1 Performance of Replication Placement

Given the number of duplications, our algorithm tries to place the duplicated roots intelligently to achieve minimum object search latency. Intuitively, as the number of replicas increase, it is more likely that the queries can get to the pointers with less number of hops since more information is available in the network. This trend is clearly captured in Fig. 5(a) and 5(b) which depicts the average case latency and worst case latency as a function of the number of roots when there are no intermediate pointers in the network. When there is only one pointer in the system, the average latency is 6 and the worst case is 12 in a 4096 node network. With increase in the number of replications, the object search latency in both average case and worst case decrease moderately.

We now compare our results with the intermediate pointer (IP) case. Note that the number of pointers is kept same as the number of roots to maintain a level playing field. Among all the four schemes shown in Fig. 5(a) and 5(b), our duplication placement algorithms OPR outperform the random roots selection scheme and the intermediate pointers scheme consistently. This is because our placement strategy actively situates the replica roots intelligently, based on the overlay topology. With the same number of replications, approximately 30% improvement can be observed via OPR compared with the IPP scheme. The random roots selection strategy performs better than the intermediate pointer scheme.

The same trend can be observed in Fig. 6(a) and Fig. 6(b) with intermediate pointers. In this case the pointers exist in all the intermediate nodes. It may be observed that the average



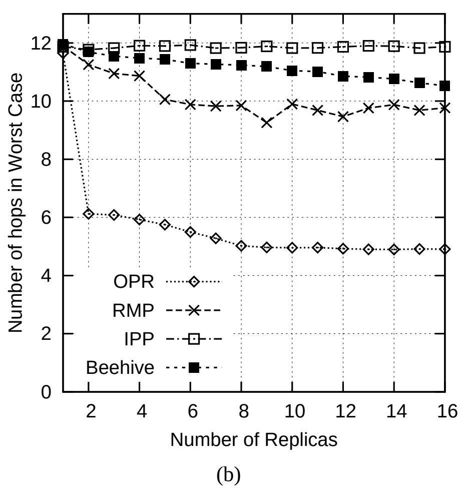
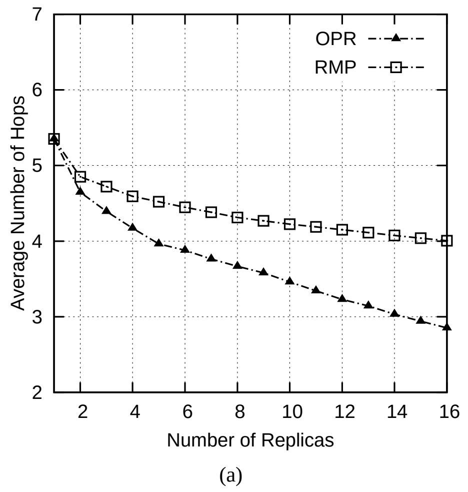


Figure 5. Perf. with Varying Number of Replications without Intermediate Pointers



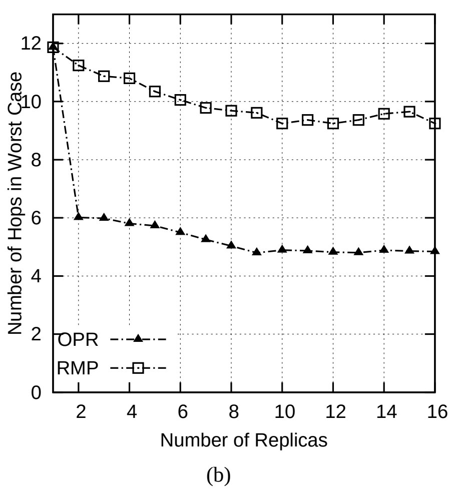


Figure 6. Perf. with Varying Number of Replications with Intermediate Pointers

latency is reduced from 6 (Fig. 5(a)) to 5.3 when there is one root and pointers at all intermediate nodes [20]. Adding multiple roots through RMP, OPR techniques reduces the latency further.

7.2 Churn Scenario

In order to examine the impact of churn rate on the performance of our algorithm, we conduct our simulation under churn. According to a study of churn model [32], we set Poisson arrivals and pareto stay time for nodes in our simulator. By default, the Poisson arrival rate is at 10 joins per second, and the pareto stay time with a minimum duration of 90 seconds and a mean of 300 second with the pareto parameter $\alpha = \frac{10}{7}$.

Fault tolerance mechanism to ensure the consistency of the routing table and the object availability is not considered in this experiment. This is done so that the benefit of replicated roots can be observed in isolation. The number of nodes in the network is kept roughly constant by matching the node arrival and failure rates. Every node in the system has the same probability of failing. Queries are considered to be unsuccessful when either all the roots in the system have failed or the paths to the roots are broken.

Fig. 7 depicts the success rate of the queries with varying churn rate in OPR when there are 2 or 4 roots. With the same failed nodes, the success rate increases as the number of roots increases. Fig. 7 shows that the reliability of OPR with roots 4 and 2 is much better than that with a single root.

Overhead evaluation: In Fig. 8 we depict the total network bandwidth consumption with varying number of replicas. Here, the bandwidth cost consumed by the query messages and update messages is measured by the total number of hops seen by all the messages per second. As shown in Fig. 8, starting from a single replication, the total cost decreases as the number

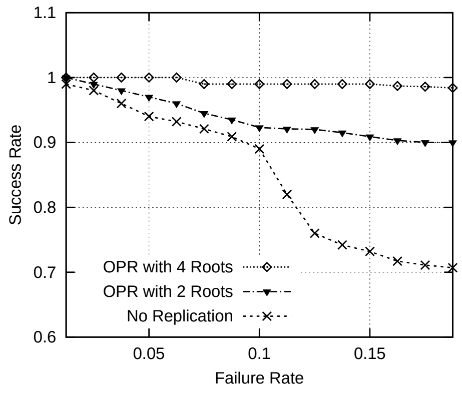


Figure 7. Perf. with Varying Churn Rate

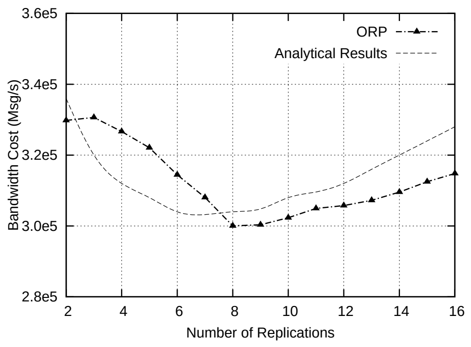


Figure 8. Replication Strategy Verification

of replications increases. This trend ends when the number of replicas reaches 8. Beyond this point, the total cost increases with increasing number of replicas. Clearly, the curve closely matches the analytical results which is derived from theoretical results, for the ease of comparison.

7.3 Load Balance Analysis

In this subsection, we evaluate the impact of our replication algorithm on load-balance characteristics. Load balance is defined as the variation in the number of queries satisfied per node and the number of queries forwarded per node. When the popularity of each object remains the same in the system and objects are uniformly mapped to nodes, each peer receives roughly the same number of queries. Hence, the hash function maps the objects to the roots uniformly. However, when the number of queries for each object follows a Zipf distribution, the amount of queries sent to each node can vary significantly.

For our experiments, we also use the ZipfII workload distribution, and vary the number of roots in the network. Figure 9(a) plots the mean and the 1st and 99th percentiles of the number of queries satisfied by each node for OPR scheme. The average number queries per node remains the same, but the numbers exhibit large variations. As the number of roots increase, the gap between the maximum and the minimum diminishes significantly. The same trend can also be observed in Fig. 9(b) which depicts the number of queries forwarded (instead of satisfied) by each node.

With Zipf distribution, some roots may receive thousands of queries while others only get a few queries. When several roots exist for one object, the queries for that object are actually partitioned and allocated to different roots. With such a design, the queries can be evenly distributed among the peers as the number of replicated roots increases.

8 Conclusion

Through our research we develop an on-line topology-aware algorithm, OPR to intelligently replicate roots in structured P2P networks based on a greedy, 2-approximation of the k-center problem. We show using extensive simulations and implementation on PlanetLab, that our methods are superior to various other competing options, by consistently performing nearly 30% better in terms of average latency and around 40% in terms of maximum query latency than IP and random schemes. We also present detailed analysis for the costs associated with updating pointer information and spawning multiple replicas. Moreover, our mechanism allows replication degree, a tunable parameter, to dynamically control the replication mechanism in order to best adapt to rapid changes in network conditions.

References

[1] BitTorrent. *http://bittorrent.com,* 2003.

[2] CoMon. *http://comon.cs.princeton.edu/*

[3] Gnutella. *http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf*, 2001.

[4] Napster Inc. The napster homepage. In *http://www.napster.com*, 2001.

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

[6] B. Arai, G. Das, D. Gunopulos and V. Kalogeraki, "Approximating Aggregation Queries in Peer-to-Peer Networks", *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, 2006.

[7] S. Androutsellis-Theotokis, D. Spinellis, "A survey of peer-to-peer content distribution technologies", *ACM Comput*, 2004.

[8] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron and A. Singh, "SplitStream: High-bandwidth content distribution in a cooperative environment", *Proceedings of IPTPS*, February 2003.

[9] J. Cao, Y. Zhang, L. Xie and G. Cao, "On peer-to-peer client web cache sharing", *ICC*, 2005.

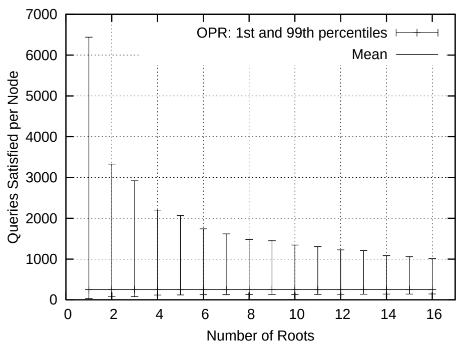
[10] T. Chang and M. Ahamad, "Improving service performance through object replication in middleware: a peer-to-peer approach", *Fifth IEEE International Conference on Peer-to-Peer Computing*, 2005.

[11] Y. Chen, R. Katz, and J. Kubiatowicz, "Dynamic replica placement for scalable content delivery", *IPTPS'02.*

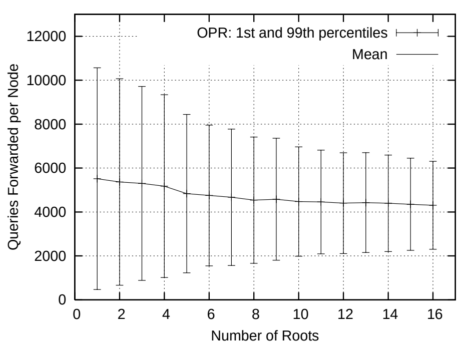
[12] E. Cohen and S. Shenker, "Replication strategies in unstructured peer-to-peer networks", *SIGCOMM'02.*

[13] E. W. Dijkstra, "A note on two problems in connetion with graphs", *Numerische Mathematik* , 1959.

[14] T. F. Gonzalez, "Clustering to Minimize the Maximum Intercluster Distance", *Theoretical Computer Science*, June 1985.



(a) # of Queries Satisfied by Each Node



(b) # of Queries Forwarded by Each Node

Figure 9. Load Balance Analysis with Varying Number of Roots

[15] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee and P. Keleher, “Adaptive replication in peer-to-peer systems”, *Proceedings. 24th International Conference on Distributed Computing Systems*, 2004.

[16] D. Guo, J. Wu, H. Chen and X. Luo, “Moore: An Extendable Peer-to-Peer Network Based Incomplete Kautz Digraph With Constant Degree”, *INFOCOM 07*, 2007.

[17] R. Huebsch, J. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica, “Query the Internet with PIER”, *Proceedings of VLDB*, Sep. 2003.

[18] O. Kariv and S. Hakimi, “An algorithm approach to network location problems. I. the *p-center*”, *SIAM Journal of Applied Mathematics 37*, 1979.

[19] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, “OceanStore: An Architecture for Global-scale Persistent Storage”, *Proceedings of ASPLOS*, Boston, MA, November 2000.

[20] C. G. Plaxton, R. Rajaraman, A. W. Richa, “Accessing Nearby Copies of Replicated Objects in a Distributed Environment”, *SPAA*, 1997.

[21] V. Ramasubramanian and E. G. Sirer, “Beehive: O(1) Lookup Performance for Power-Law Query’ Distributions in Peer-to-Peer Overlays”, *Symposium on NSDI*, San Francisco CA, Mar 2004.

[22] P. Rao and B. Moon, “psiX: Hierarchical Distributed Index for Efficiently Locating XML Data in Peer-to-Peer Networks”, *Technical Report*, 2005.

[23] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A Scalable Content-Addressable Network”, *ACM SIGCOMM*, San Diego, CA, Aug. 2001.

[24] S. Ratnasamy, M. Handley, R. Karp and S. Shenker, “Topologically Aware Overlay Construction and Sever Selection”, *IEEE Infocom*, 2002.

[25] S. Rhea, B. Chun, J. Kubiatowicz, S. Shenker, “Fixing the Embarassing Slowness of OpenDHT on PlanetLab”, *Proceedings of the Second Workshop on Real, large Distributed System (WORLDS’ 05)*, 2005.

[26] M. Ripeanu and I. Foster, “Mapping the Gnutella Network: Macroscopic Properties of Large-Scale Peer-to-Peer Systems”, *In Proceedings of the first International Workshop on P2P Systems*, March 2002.

[27] A. Rowstron, P. Druschel, “Pastry: Scalable, Decentralized Object Location and Routing for LargeScale Peer-to-Peer Systems”, *The 18th IFIP/ACM Conf. on Distributed Systems Platforms*, Heidelberg, Nov. 2001.

[28] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana, “Internet indirection infrastructure”, *ACM SIGGCOM*, August 2002.

[29] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for Internet applications”, *In Proc. ACM SIGCOMM*, August 2001.

[30] M. Walfish, H. Balakrishnan, and S. Shenker, “Untangling the web from DNS”, *Symposium on Networked Systems Design and Implementation(NSDI)*, 2004.

[31] C. Wang, L. Xiao, Y. Liu, P. Zheng, “DiCAS: An Efficient Distributed Caching Mechanism for P2P Systems”, *IEEE Transactions on Parallel and Distributed Systems Vol. 17, Issue 10* , 2006.

[32] X. Wang, Z. Yao and D. Loguinov, “Residual-Based Measurement of Peer and Link Lifetimes in Gnutella Networks”, *INFOCOM*, May 2007.

[33] L. Xiao, X. Zhang, A. Andrzejak, S. Chen, “Building a large and efficient hybrid peer-to-peer Internet caching system”, *IEEE Transactions on Knowledge and Data Engineering, Vol.16, Issue 6* June 2004.

[34] E. Zegura, K. Calvert and S. Bhattacharjee, “How to Model an Internetwork”, *IEEE Infocom*, May 1996.

[35] B. Y. Zhao and J. D. Kubiatowicz and Aerence. D. Joseph, “Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing”, *Proc. of the Second International Conf. on P2P Computing*, Feb. 2001.

[36] 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”, *Proceedings of NOSSDAV*, June 2001.