A Lightweight, Robust P2P System to Handle Flash Crowds

Angelos Stavrou,*Student Member, IEEE,* Dan Rubenstein, *Member, IEEE,* Sambit Sahu

***Abstract*— Internet flash crowds (a.k.a. hot spots) are a phenomenon that result from a sudden, unpredicted increase in an on-line object's popularity. Currently, there is no efficient means within the Internet to scalably deliver web objects under hot spot conditions to all clients that desire the object. We present PROOFS: a simple, lightweight, peer-to-peer (P2P) approach that uses randomized overlay construction and randomized, scoped searches to efficiently locate and deliver objects under heavy demand to all users that desire them. We evaluate PROOFS' robustness in environments in which clients join and leave the P2P network as well as in environments in which clients are not always fully cooperative. Through a mix of simulation and prototype experimentation in the Internet, we show that randomized approaches like PROOFS should effectively relieve flash crowd symptoms in dynamic, limited-participation environments.**

***Index Terms*— peer-to-peer networks, flash crowds, hot spots, World Wide Web (WWW)**

I. INTRODUCTION

Internet Flash Crowds (a.k.a. hot spots) are a phenomenon that result from a sudden, unpredicted increase in an on-line object's popularity. Examples include the news pages at www.cnn.com and www.nytimes.com on September 11th and immediately following the plane crash in New York on November 12th. During the very times when content reaches its apex in popularity, it becomes unavailable to the majority of users that seek it.

There are several approaches to remedy the problem. A straightforward but costly approach is to provision accessibility based on peak demand. An alternative approach is to dynamically increase server locations of the popular documents. Content distribution companies such as Akamai have identified ways to offload the burden placed on servers to transfer embedded objects. However, to prevent flash crowds from overloading servers with requests for container pages, significant changes must be made to the Domain Name System (DNS) so that clients' initial requests can be also immediately be redirected to available resources. In particular, the time for which a DNS entry is cached must be kept short or a mechanism must be added to allow updated DNS entries to be pushed down the DNS caching hierarchy.

Manuscript received November 15, 2002; revised June 1, 2003. This paper extends results reported in an ICNP 2002 conference paper with the same title. This material was supported in part by the National Science Foundation under Grant No. ANI-0117738. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Angelos Stavrou is with the Electrical Engineering Department, Columbia University. Email: angelos@ee.columbia.edu

Dan Rubenstein is with the Electrical Engineering Department, Columbia University. Email: danr@ee.columbia.edu

Sambit Sahu is with IBM Research. Email: sambits@us.ibm.com

A third approach is to have the clients form a peer-to-peer (P2P) overlay network that allows those clients that have received copies of the popular content to forward the content to those clients that desire but have not yet received it. In this paper, we describe and evaluate our implementation that applies this third approach on top of overlay topologies generated essentially at random to scalably, reliably deliver content whose popularity exceeds the capabilities of standard delivery techniques. We call this system **PROOFS**: **P**2P **R**andomized **O**verlays to **O**bviate **F**lash-crowd **S**ymptoms.

PROOFS is not meant to replace the existing web-based infrastructure that utilizes DNS. It is instead designed to support that infrastructure at times when a flash crowd overwhelms the original architecture's capabilities - an idea similar to that described in [1], [2]. PROOFS consists of two protocols. The first forms and maintains a network overlay that connects the clients that participate in the P2P protocol. The overlay construction is an ongoing, randomized process in which nodes continually exchange neighbors with one another at a low rate. The second protocol performs a series of randomized, scoped searches for objects atop the overlay formed by the first protocol. Objects are located by randomly visiting sets of neighbors until a node is reached that contains the object.

The distributed protocol that forms the overlay is unique in that we seek to "randomize" the connectivity in the overlay. In particular, we make no effort to connect nodes that are "close" to one another with respect to a distance metric such as end-to-end delay. While this approach may cause an increase in search times, the randomness of the topology makes the system more robust: if and when nodes join and leave the overlay, little needs to be done to adjust the topology to accommodate the changes in overlay membership. In addition, since a query results in numerous searches performed in parallel, with high likelihood there will be many paths that choose (at random) nodes that are topologically close. Since the time taken to retrieve an object is the minimum time taken by any search path that locates the object, we conjecture that it is highly likely that some path that locates the object will consist mainly of "short" edges, making expected search times low.

The search protocol itself has many similarities to protocols used by applications such as Gnutella and is not strikingly new. However, unlike the majority of other proposed approaches that are not specifically designed to handle flash crowds, PROOFS does not require users to cache copies of objects or pointers to objects outside from what they have explicitly requested. As demonstrated by our analysis in [3], when run atop a randomly-connected overlay, this simple search protocol quickly retrieves "hot" objects with low bandwidth overheads.

Our work here makes several novel contributions. In particular, we demonstrate that undirected searches atop randomly-formed overlays for “hot” content

- are robust to users (nodes) joining and leaving the overlay network with time.
- are resilient in environments where a majority of users limit their participation in the search process by either restricting the manner in which they forward other users' queries (including not forwarding them at all) as well as not returning copies of a requested object even when a copy exists locally.

Last, we evaluate a prototype implementation on a testbed comprised of end-systems scattered around the world. Although small in scale compared to how we hope the system will eventually be used, the testbed demonstrates that latencies and traffic utilizations by the system are low enough to make the approach feasible in today's networks.

The paper proceeds as follows. In Section II, we overview related work. Section III describes the basic architecture of PROOFS. In Section IV, we evaluate our design's robustness in the face of dynamic changes to overlay membership and clients who offer limited participation. Section V presents experimental results using a prototype version of PROOFS upon the real Internet. We discuss some limitations, future directions, and challenges in Section VI and conclude in Section VII.

II. Related Work

The idea of flash-crowd alleviation via replication was previously considered in [4]. However, the architecture there involves an elaborate communication and exchange mechanism between servers within the network, having been developed before the notion of peer-to-peer communication became popular. The idea of designing a peer-to-peer system that operates as a “backup” to the existing DNS/Web infrastructure to support loads brought on by flash crowds was recently proposed in [1], [2]. Other works that have designed systems that use client collaboration for anonymous and secure content distribution or preservation, but not necessarily within the context of flash crowds, are Crowds [5] and Publius [6].

This paper focuses on the robustness of PROOFS in overlays where protocol participants may limit their participation in the protocol. Our main focus here is not to evaluate its scalability as a function of time to recover objects and messages sent throughout the network as the number of participants grows. A thorough investigation of that problem appears in a separate work [3], where we analyze and simulate a discrete-event version of a randomized, scoped search protocol that is the basis of PROOFS. There, we show that upon randomized topologies, such systems can effectively scale to overlays that contain millions of participating clients.

Randomized overlays are also considered in mobile environments [7], [8]. However, there the “shape” of the overlay, and therefore the performance of the approach, is a function of how users move through the system and their transmission constraints. In PROOFS, we can further optimize the overlay, given that we expect that at all times, any pair of participants can communicate directly atop the underlying IP substrate as long as one participant has the identity (e.g., the IP address) of the other.

A significant amount of attention has been paid to structured P2P architectures such as CAN [9], CHORD [10], [11], PASTRY [12] and Tapestry [13], in which participants have a sense of direction as to where to forward requests. For unpopular documents, structured architectures clearly provide benefit over their unstructured counterparts in terms of the amounts of network bandwidth utilized and the time taken to locate those documents. However, to be able to handle documents whose popularity suddenly spikes without inundating those nodes responsible for serving these documents, these architectures must implement a caching mechanism that caches the objects, as is proposed in [14]. It is unclear whether the transfer overheads such an approach makes sense in a browser-like environment where clients join and leave the system at high frequency. Last, we suspect that members of the overlay who do not participate fully (e.g., drop requests or refuse to transmit objects) will significantly limit the efficiency of these approaches.

Network paradigms such as multicast [15] and anycast [16], [17] can also be used to retrieve “hot” objects at a relatively low cost. Multicast is problematic in that it is difficult to appropriately scope queries to prevent flooding within the network of simultaneous user requests. Anycast is a paradigm that can be used to direct users toward massively replicated content. but does not solve the problem of automating the process of dynamically replicating the content when a hot spot arises.

A wide body of work has considered the theoretical problem of resource location in networks. However, the models considered are often not directly amenable to the problem of delivering information that has suddenly become inaccessible as a result of flash crowds. For instance, in [18] it is assumed that the user seeks a resource that cannot be replicated, and that the network can store state of the search to prevent duplicate traversal along network paths. The massive distribution of popular information is the basis for work in gossip protocols [19], [20], [21]. A limitation of these gossip-style protocols within a flash crowd scenario is that efficient gossiping requires that those nodes that already have the object be in the position to determine whether this object is worth propagating further. In contrast, in a flash crowd, it is the set of nodes that are without the object that make such a determination.

There has been interesting theoretical work that looks at ways to form “good” topologies for scoped searches. One example is that of [22] which focuses on building randomized topologies with bounds on the overlay graph's diameter. The procedure is somewhat more complicated and relies on a central server at various points in the algorithm beyond mere bootstrapping. The overlay generation method considered here does not give any such guarantees on overlay graph diameter, though we expect that in practice the diameter will be small. In its current form, the only centralized component of PROOFS is what is used to bootstrap new clients into the overlay. However, other means such as multicast or anycast can be used in place, removing the need for a centralized component.

Last, there exists a small body of work that has measured or analyzed existing P2P file sharing systems such as Gnutella and Napster [23], [24], [25].

Last, we suspect that the recent caching approach described in [26] will perform well in a flash-crowd environment. While the caching overheads are unnecessary when it is known that the item being searched for is popular, requiring explicit caching could be used to reduce damage caused by false alarms, where users seek unpopular objects that they mistakenly believe are inaccessible because of a flash crowd.

III. Design Description

In this section, we describe the application for which the PROOFS system was designed and describe the details of that design. The objective of PROOFS is to provide additional support to the existing Web/DNS infrastructure. In particular, our goal is to provide timely delivery of web objects that are stored at locations whose availability is compromised as a result of a heavy request load for the objects.

A. PROOFS Design

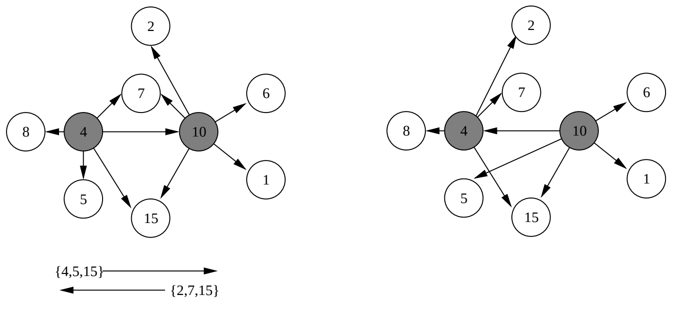
Here we consider the architectural design of the PROOFS system without attempting to optimize its performance in any way whatsoever, i.e., no functionality is added beyond what is necessary to make it functional and robust. Two components comprise the system: the *client* and the *bootstrap server*. From the perspective of PROOFS (without optimizations added), clients are a set of homogeneous end-systems that form the P2P overlay and are used to send searches. Bootstrap servers maintain a finite-sized cache of recent users that have recently joined the overlay, providing a means (prior to hot spot activity) by which clients can learn about and identify other clients in the overlay. In our current implementation (discussed in Section V), we utilize a single bootstrap server. However, it is easy to extend the system to one that employs several bootstrap servers so that users can join the PROOFS system in environments where bootstrap servers can fail.

Each PROOFS client runs two protocols, `ConstructOverlay` and `LocateObject`. `ConstructOverlay` is responsible for determining which sets of clients a client is permitted to query when searching for objects. `LocateObject` is the protocol that participates in searches upon the overlay network formed by `ConstructOverlay`. `ConstructOverlay` is in essence the passive component, running continually, whereas `LocateObject` is initiated by the client only when flash crowd phenomena exist within the network. Below, we give brief descriptions of these two protocols. These protocols rely heavily on randomness to be both simple and robust. All communications between clients occur at the IP level, i.e., each client has an IP address and port that it uses to send and receive communications.

1) `ConstructOverlay`: When a client wishes to participate in the PROOFS system, the `ConstructOverlay` protocol first contacts a bootstrap server to obtain a preliminary list of *neighbors* (an IP address:port combination). A client A's neighbors are the set of nodes with which it is permitted to initiate contact. Hence, if the P2P overlay is viewed as a graph G in which the set of clients are the nodes, then the neighbor relation is indicated via a directed edge. Because we use directed edges, it is possible for node B to be node A's neighbor (such that A can initiate contact with B) while A is not B's neighbor (such that B can only communicate with A directly by responding to A). This set of neighbors is the only state maintained by the `ConstructOverlay` protocol that varies with time. There is a fixed bound, C, on the maximum number of neighbors that a client will maintain.

Clients continually perform what is called a *shuffle* operation. The shuffle is an exchange of a subset of neighbors between a pair of clients and can be initiated by any client. The client C_1 that initiates a shuffle chooses a subset of neighbors of size c that is the no larger than the minimum of the bound, C and its current number of neighbors. It selects one neighbor, C_2 from this subset and contacts that neighbor to participate in the shuffle. C_1 sends the subset of neighbors it selected with C_2 removed from the subset and C_1 added. If C_2 accepts C_1's shuffle, it selects a subset of neighbors from its list of neighbors and forwards this subset to C_1. Upon receiving each other's subsets of neighbors, C_1 and C_2 update their respective neighbor sets by including the set of neighbors sent to them. The replacement is done according to three rules:

1) No neighbor appears twice within the set.
2) A client is never its own neighbor.
3) If the size of the the neighbor set currently lies below the bound, C, new entries to the neighbor set are added without overwriting previous entries (until the bound reaches C).
4) Neighbors in the neighbor set can only be overwritten (i.e., removed) if they were sent to the other neighbor during the shuffle.



(a) Before the shuffle (b) After the shuffle

Fig. 1. An example of a shuffle operation

A sample shuffle operation is shown in Figure 1. There, clients are represented by numbered circles. Directed edges indicate the neighbor relation, where an arrow pointing from A to B means that B is a neighbor of A. Neighbors are depicted only for the darkened clients numbered 4 and 10. These nodes start with the set of neighbors depicted in Figure 1(a) and end with the set of neighbors depicted in Figure 1(b).

Note two important points: first, no client becomes disconnected as a result of a shuffle: it simply moves from being the neighbor of one node to being the neighbor of another. Second, if client A is B's neighbor and B initiates a shuffle with A, then after the shuffle, B is A's neighbor (i.e., the edge reverses direction).

In our current implementation, a client waits a random amount of time sampled from an exponential distribution. A shuffle request is only rejected by neighbors that have placed a request to shuffle but have not yet received a response. Upon receiving a rejection (or a timeout), a client continues the process of choosing the next time to initiate the shuffle from a uniform distribution. The rejection must be explicitly acknowledged. Clients that do not respond to shuffle requests are assumed to be inactive (i.e., are no longer part of the overlay) and are removed from the requesting client's neighbor set.

Shuffling is used to produce an overlay that is "well-mixed" in the a client's neighbors are essentially drawn at random from the set of all clients that participate in the overlay. There is no attempt to optimize the overlay such that neighbors are topologically adjacent. There is no reason to ever terminate the shuffling operation. Once a "random" state is reached, additional shuffles will keep the overlay in a "random" state.

2) `LocateObject`*:* The `LocateObject` protocol is the protocol that attempts retrieval of the desired object by searching among the participating clients that are connected together by the overlay that was constructed using the `ConstructOverlay` protocol. Once a client decides to use PROOFS to retrieve an object (how such a decision can be made is discussed in Section VI), a *query* is initiated at the client. A query contains the following information:

- **Object**: a description of the object being searched for.
- **TTL**: a counter that counts the maximum number of additional hops in the overlay that the query should propagate if a copy of the requested object has not been located.
- **fanout**: a value f that indicates to how many neighbors a client should forward a query that it has received when it does not have a copy of the requested object (assuming the TTL has not expired).
- **Return Address**: the address of the client that initiated the query such that once a suitable object is located, it can be returned directly.

When a client receives a query or initiates a query from another client, it first checks to see if it contains a copy of the requested object. If so, it forwards the object to the return address specified in the query. Otherwise, it decrements the TTL of the query, and if the TTL is non-negative, randomly selects f neighbors from its neighbor set and forwards the query with the decremented TTL to those neighbors. Neighbors that receive the query are expected to acknowledge receipt by sending an ACK packet back to the client that forwarded the query. If no ACK is returned from a client then another client is selected at random and the query is instead forwarded to that client.

If a client that initiates a query does not receive a copy of requested object after a certain period of time, the client assumes that no clients reached by the query had a copy of the object and repeats the query. Currently, we increment the TTL value by one each time a query fails until reaching a given value.

IV. Robustness

In this section, we evaluate the robustness of PROOFS. In particular, we investigate the design's robustness as a function of the following networking phenomena:

- **Overlay partitioning**: Given a fixed set of clients participating in the overlay, it is possible that the `ConstructOverlay` Protocol produces partitions upon the directed overlay graph such preventing communication between all pairs of clients. We analytically prove that when the overlay partitions, the types of partitions caused are never permanent (i.e., they are automatically healed by the protocol eventually with probability 1). We also present simulation results to show that for reasonable neighbor set sizes, partitions are rare occurrences and, when they do occur, are quickly healed.
- **Joins/Leaves**: One expects that over time, clients will join and leave the overlay, and that clients may leave without warning or notification (i.e., they may simply fail). We show via simulation that the majority of clients can still reach a very large fraction of clients in the overlay even when join and/or leave rates are extremely high.
- **Pseudo-participants**: There may exist clients that wish to retrieve objects using the PROOFS system but wish to limit participation assisting other clients within `LocateObject`. We show that, even with up to 80% of clients limiting their participation, our design maintains acceptable traffic levels and times for object delivery.

A. Graph Partitioning

We say that an overlay is **partitioned** if there exists a pair of nodes, n_1 and n_2 in the overlay where no path exists from n_1 to n_2. Such an occurrence would prevent any queries forwarded by n_1 from reaching n_2. In our discussions below, we will consider both partitions in the directed graph (that takes into account the directions of the edges) as well as partitions of the underlying undirected graph (where directions of edges do not matter). Clearly, if the undirected graph is partitioned, then the directed graph must be partitioned as well, but the reverse need not be true.

Partitioning of the undirected, connected graph is of particular concern. It is easy to show that a partitioned undirected graph cannot be repaired via shuffling. In contrast, it is easy to show that a directed graph that is partitioned but whose underlying undirected graph is not partitioned can be repaired by shuffling.

There are practical complications in maintaining an undirected overlay graph (where an edge permits bi-directional communication between the nodes it connects). In particular, when a node leaves the overlay, the nodes with which it connected must find new neighbors that are willing to take on new neighbors as well. This complication compels us to use an overlay whose edges are unidirectional. Unfortunately, it is conceivable that shuffling operations will partition the underlying overlay. However, we can show that any partitioning of the graph due to shuffling is temporary. Eventually, the shuffling process re-connects the separated participants. We emphasize that this theoretical result holds conclusively only when nodes do not leave the overlay.[1]

The result is proved by considering the underlying undirected graph (i.e., removing the directions on edges in the overlay graph). We first prove that shuffling will not partition such a graph, and then show that if the underlying undirected graph is not partitioned, then eventually a path will exist from any node n_1 to another node n_2 within the directed graph.

An undirected graph G is said to be *connected* if a path exists between every pair of nodes, n_1 and n_2.

Lemma 1: Let G be an undirected connected graph, and let G' be the graph that is derived from G by applying an arbitrary shuffle operation. Then G' is an undirected connected graph.

Proof: A shuffle consists of an exchange of a pair of m nodes. This exchange can be done by first removing those nodes that appear in both shuffle sets and then performing the exchange one node at a time (where an exchange might be in a single direction for the case where one node has fewer than m nodes in its cache to swap).[2] Hence, we can restrict our attention to the case where two nodes exchange at most one entry. It follows from induction that if the graph remains connected after a single swap, it remains connected after all swaps performed within the shuffle.

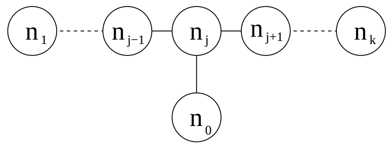
Let $P = \langle n_0, \cdots, n_k \rangle$ be an arbitrary sequence of nodes that forms a path in G as depicted in Figure 2(a). Since we are considering a single swap, there are three cases to consider:

- Case 1: neither node implementing the swap lies on the path. This means that while there may be nodes on the path whose edges change (as a result of the swap), the changed edges connect to the nodes implementing the swap. Hence, no edges that form the path are changed, so the path remains after the swap is complete.
- Case 2: one node, n_j, that implements the swap lies on the path and the other lies off the path (call this other node n_0). Since the nodes that implement the swap are connected both before and after they perform the swap (but the direction of the edge changes within the directed graph), two possible scenarios occur: the node on the path swaps away no edges or swaps away one edge. As can be seen in Figure 2(b), a path between n_1 and n_k remains after the swap.
- Case 3: both nodes lie on the path. It follows that if n_j and n_{j+m} are the nodes implementing the swap, then either n_j is a neighbor of n_{j+m} of n_{j+m} is a neighbor of n_j. In either case, there is an edge connecting n_j and n_{j+m} in the undirected graph. We can restrict our attention to the alternative path $\langle n_1, \cdots, n_j, n_{j+m}, \cdots, n_k \rangle$ connecting n_1 to n_k in G. We use this path instead and relabel all $n_\ell, \ell > j$ as $n_{\ell-m+1}$ such that n_j and n_{j+1} are the nodes that implement the swap. Here, there are three cases to consider: a) no edges on the path are swapped between the nodes, b) n_j forwards to n_{j+1} its connection to node n_{j-1} and n_{j+1} forwards n_j a node that lies off the path or no node (this case also covers the case where the roles of n_j and n_{j+1} are reversed), and c) n_j forwards node n_{j-1} to n_{j+1} and n_{j+1} forwards edge n_{j+2} to n_j. As shown in Figure 3, for all three cases, the resulting graph remains connected. For case b), the new path skips over node n_j and for case c), the new path goes from n_{j-1} to n_{j+1} to n_j to n_{j+2}.

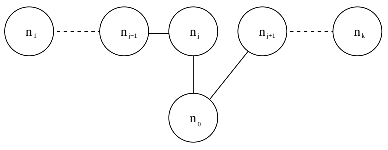
∎

[1]It is trivial to construct cases where leaving nodes can create a partition that cannot be healed without outside intervention. Subsequent simulation results will demonstrate that such permanent partitions are extremely rare.

[2]Once duplicates are removed, the swapping operation is associative, i.e., the order in which nodes are actually exchanged does not alter the final outcome.

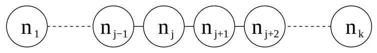


(a) Initial topology and no swaps

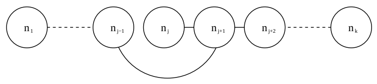


(b) One swap on the path

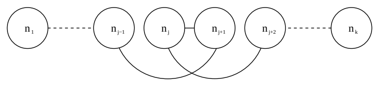
Fig. 2. A generic path where the node being swapped with is off the path.



(a) Initial topology and no swaps



(b) One swap on the path



(c) Two swaps on the path

Fig. 3. A generic path where the node being swapped with is on the path.

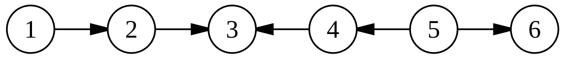


Fig. 4. An example of an unconnected directed path whose underlying undirercted path is connected.

Theorem 1: Let G be a directed graph for which a path (in the undirected sense) $n_1, n_2, \cdots, n_k$, exists connecting n_1 to

n_k, but where no directed path exists from n_1 to n_k. Then there exists a series of shuffle operations that will form the directed path.

Proof: Consider the undirected path between n_1 and n_k. Some directed edges along this path point toward n_k and some point toward n_1. An example is illustrated in Figure 4, where the edge connecting node 3 to node 4 and the edge connecting node 4 to node 5 point in the wrong direction. For any i where node n_{i+1} has an edge pointing toward n_i, if n_{i+1} initiates a shuffle operation with node n_i, this operation terminates with the edge pointing from n_i to n_{i+1}. For instance, in Figure 4, node 4 must initiate a shuffle operation with node 3, and node 5 must initiate a shuffle operation with node 4.

Since Lemma 1 ensures that the graph remains connected in the undirected sense at all times, and since the proper sequence of shuffling operations is a finite set of shuffles, with probability one an appropriate sequence is eventually selected that forms the directed path. ■

Last, we have performed simulation results that demonstrate that when the set of clients remains fixed, there is a partition in the directed sense less than 95% of the time, and that during these partitions, all clients are still able to reach more than 95% of the clients in the graph. These simulations are discussed in the next subsection.

B. Handling Dynamic Joins and Leaves

We now evaluate via simulation the likelihood of a partition for the case where clients dynamically join and leave the PROOFS system. Clearly, one can construct sample paths of joins and leaves that cause a partition in the underlying directed graph. In each simulation, an upper bound, N, is placed on the number of clients participating in the overlay. These clients join and leave the overlay, each client's join and leave times are exponentially distributed with rates of λ and μ, respectively. Each client initiates shuffles where the time between these initiations is exponentially distributed with mean rate 1. In these experiments, when clients leave the overlay, there are no explicit attempts to self-heal the overlay, i.e., edges that pointed to clients since departed subsequently point to nowhere until the client returns. Upon their return to the overlay, a client contacts the bootstrap server of its arrival and obtains a new list of neighbors. We vary the likelihood with which a client would inform the bootstrap server of its departure from the overlay. However, we find this announcement to have negligible impact on performance, so results presented here are only for the case where clients *do not* inform the bootstrap server of their departures.

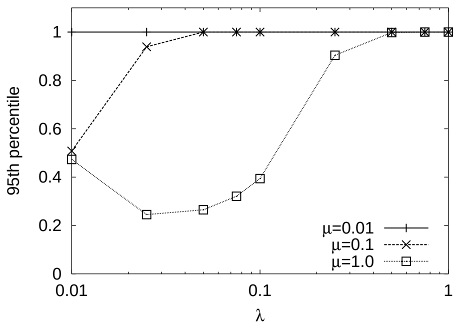
During a simulation, we sample the status of the overlay at an average rate of $1/N$, with the time between samples drawn from an exponential distribution. We collect 1200 samples and discard the first 200 to allow the experiment time to converge toward a steady state. By PASTA (Poisson Arrivals See Time Averages), the fact that the times between samples are exponentially distributed guarantees that the samples reflect steady-state behavior.[3]

[3]This of course assumes that the system has reached steady state by the 200th sample.

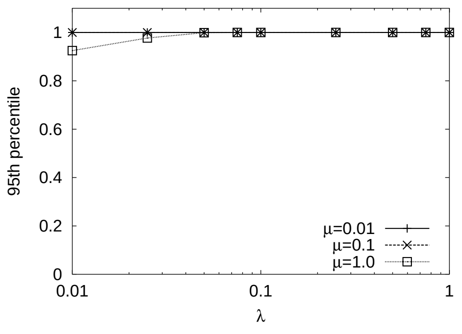
During each sample, for each active client $\mathcal{C}$ (currently joined to the overlay), we compute the fraction of other active clients that can be reached by $\mathcal{C}$ via some path along a sequence of directed edges within the overlay graph. We call this quantity the *reachability* of $\mathcal{C}$. During each sample, we compute the minimum, mean, median, and maximum reachability over all active clients. Table I lists the set of parameters varied during experiments as well as the values to which the different parameters were set.

TABLE I
PARAMETERS VARIED FOR PARTITION SIMULATIONS

# clients	50,100,500,1000,2000
client neighborhood size	5,10,25,50
λ	0.01 through 1
μ	0,0.01,0.1,1
shuffle size	2,5,10
bootstrap server cache	5, 10, 50, 100



(a) minimum



(b) average

Fig. 5. 95% bounds on reachability

Figure 5 presents results for experiments in which $N = 2000$ clients formed the overlay, each with a bound of 25 on the size of its neighbor set. The shuffle size is set to 5 and bootstrap cache size is set to 25. Figure 5(a) plots, as a function of λ and μ, the fraction of the time for which the reachability exceeds 95% for *all* clients. In other words, fewer than 1 of 20 samples should contain a client whose reachability is lower than the indicated values. λ is varied on the x-axis with the different curves plot the results for differing values of μ. Figure 5(b) is similar to Figure 5(a) except that the average reachability is plotted instead of the minimum reachability.

We see at least 95% of the time, the average reachability equals one (all clients are able to reach all other active clients). A client with the minimum reachability within a sample can drop as low as 20%, which means that a clients' query can reach at most 400 of the 2000 clients participating in the system. We emphasize that these plots are based on the reachability of the client with lowest reachability at each sample. A single client remains "the worst" for short periods of time and so an individual client's average reachability is much higher than what is plotted here. In addition, we note that low levels of reachability occur only in extreme cases where the expected time for which a client remains in the system is 50 times smaller than the expected time for which the client is exited from the system. Note that such a ratio corresponds to a scenario in which clients that use web browsers twice a day run the browser on average for less than 15 minutes per sitting. This makes these high ratios unlikely in practice. We therefore expect under realistic conditions, reachability will be high for all active clients at all times. We omit the plots for the case where $\mu = 0$ (clients join and never leave) since the curves simply overlap with the curves plotted for $\mu = 0.01$. In other words, in our experiments with $\mu = 0$, client reachability dipped below 1 less than 5% of the time, and never dipped below 0.95.

We have also evaluated the impact that neighborhood size has on minimum reachability. Due to lack of space, we simply summarize our finding that increasing neighborhood size greatly increases the minimum reachability in the overlay graph when $\lambda << \mu$.

Note that we have examined the algorithm in an environment where we make no explicit attempts to repair partitions. In practice, it would be simple if desired to add an additional mechanism to explicitly perform repairs. For instance, a client, upon detecting an unresponsive neighbor could remove the neighbor from its neighbor set and shuffle with an active neighbor to replenish its neighbor set. On the rare occasions that a client finds itself partitioned or unable to increase its neighborhood to the desired size by shuffling can contact the bootstrap server to obtain a fresh set of neighbors. Such types of mechanisms would reduce the likelihood of partitioning, improving reachability, if deemed necessary.

C. Non-cooperating clients

We now turn our attention to evaluating the robustness of PROOFS as we vary the level of participation of clients within protocol `LocateObject`. Because PROOFS is designed to run on users' desktop machines, not all clients are necessarily willing to fully participate. In some cases, clients may even attempt to deceive others about their levels/ability to participate [23]. In fact, the ability to adjust the level of participation is a feature in file-sharing systems.We introduce three basic means by which a client can limit its participation in PROOFS:

- **Query-only**: a query-only client will act as though it has not received a copy of the object. However, the client will forward queries further in the normal fashion (forwarding the query to f neighbors after decrementing the TTL as long as the TTL is larger than 0.)
- **Tunneling**: upon receiving a query, a tunneling client selects a single neighbor and forwards the query to the neighbor with a decremented TTL.[4]
- **Mute**: a mute client drops all queries it receives without notifying other clients of this behavior. We assume that other clients are not aware that a given client is mute and therefore no action is taken to compensate for mute clients.

Using discrete-event simulation, we evaluate the performance of PROOFS as a function of the number of messages transmitted to each client[5] and the average time taken for a client to receive the requested object. In these simulations, time is measured in *hops*: where each transmission between a pair of clients requires a single time unit. A client can transmit an unlimited number of queries to neighboring clients within the same time unit.

Following the lead of [3], we evaluate these measures of performance using two different client arrival processes that determine the proximity in time with which clients become interested in the "hot" object and initiate queries. In the **isolated** arrival process, only one client is interested at any given time. A client's search for the object must complete before the next client's search commences. In the **joined** arrival process, the times at which client searches are initiated follow the distribution of a branching process. Here, each client that has not already initiated its search by time t initiates its search at time t with probability $q_1 + {q_2}^{n(t)}$, where q_1 and q_2 are constants and $n(t)$ is the number of clients that had been initiated by time unit $t-1$. This emulates a scenario where a client self-initializes with probability q_1 or is "told" about the object by each other client that has already started its search independently with probability q_2. In our experiments, we have arbitrarily selected $q_1 = 0.001$ and $q_2 = 0.01$.

We begin by considering the fraction of searches that fail to locate a copy of the desired object. As the fraction of clients that are willing to forward queries or return copies of objects decreases, the likelihood of a search failing increases. Figures 6 and 7 plot results of simulations using the isolated arrival process. In both figures, the x-axis indicates the fraction of clients that are non-participants. The type of non-participants (query-only, tunneling, or mute) is indicated by the different bars in Figure 6 and different curves in Figure 7. When $x = 0$, all clients are "behaving" following the basic rules of the protocol. Here, the overlay used to generate these plots contains 1000 clients, each with a neighbor set of size 25. The fanout, f, used here is 5. Each point plotted is the the average of 300 runs. When shown, 95% confidence intervals are generated from 20 samples that average 15 data points at a time (such that each sample is drawn from a distribution that is approximately normal).

[4]Our original intention was to not decrement the TTL but this created large bandwidth overheads as the number of limited-participation clients was large.

[5]A subtle point should be made here that the average number of queries received equals the average number of queries sent (since every query that leaves a client must arrive at another client.

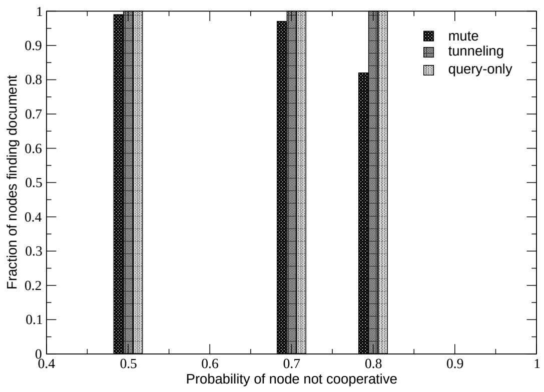


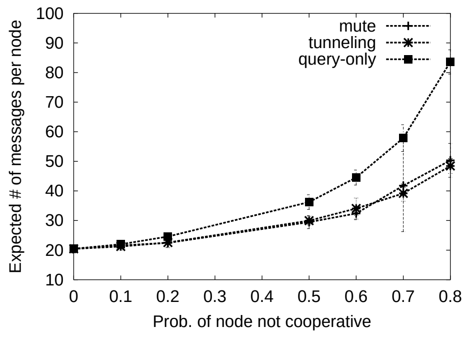
Fig. 6. Non-cooperation search completion rates: Isolated arrival process

Figure 6 illustrates the fraction of clients that locate a document as a function of the fraction of non-cooperative clients. Those clients who limit their participation all do so in an identical fashion: the different curves indicate the type. We see that even when the fraction of non-cooperating clients is as high as 0.5, all clients' queries are successful when the non-cooperation type is query-only or tunneling. When the type is mute, a client's query is successful 99.5% of the time. We also observe that the fraction of clients that find the document does not degrade as the fraction of non-cooperating clients increases further with the exception of the mute type of non-cooperation. There, fewer than 20% of searches fail to locate the object.

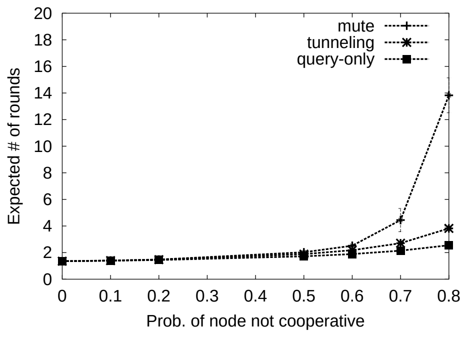
Figure 7 plots the average number of messages per client and average time units required using the isolated arrival process. *We emphasize that this is the expected number of messages that each client expects to receive so that the desired object can be delivered to* **all** *999 other clients (including itself) to obtain a copy of the object.* It is also the expected number of messages that each client must send for this purpose.[6] such From Figure 7(a) we observe that when the fraction of non-cooperating clients is 0.5, the average number of messages does not even double. In fact, for mute and tunneling types, levels of traffic increase by only 50%. We see that types tunneling and mute have less of an impact on traffic than does type query-only. We observe large confidence intervals at $x = 0.7$ for the mute type. These are a result of the small number of searches that do not locate the object because no path exists through non-mute clients to the object. This creates a small set of searches that generate significantly larger levels of traffic.

Figure 7(b) plots the average number of time units taken for a client to retrieve the object. We observe here that types query-only and tunneling cause minimal increases in search

[6]This seems somewhat counter-intuitive at first, since a client typically sends f messages for each message received. However, there are occasions when the TTL reaches 0 at the client such that it forwards nothing when it receives a message. The observation is proved by noting that each message sent by a client is received by another.



(a) average number of messages



(b) average number of time units

Fig. 7. Non-cooperation overhead: Isolated arrival process

times. The mute type causes a minimal increase when the fraction non-cooperating clients falls below 0.6. However, the time increases dramatically once this fraction is passed.

We run similar experiments for the case where clients initiated queries according to the joined arrival process. There, we observe similar trends in both the average number of messages and the average number of time units. The only difference is that the averages are slightly (no more than 20%) higher than for the isolated arrival process.

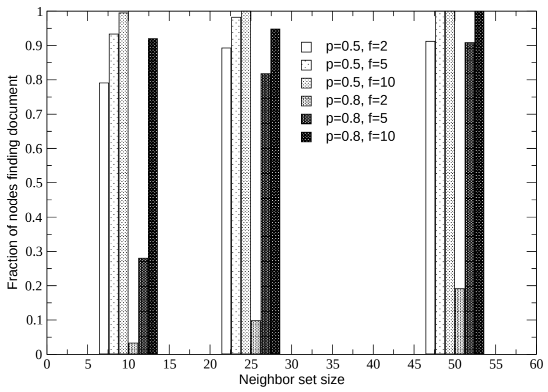
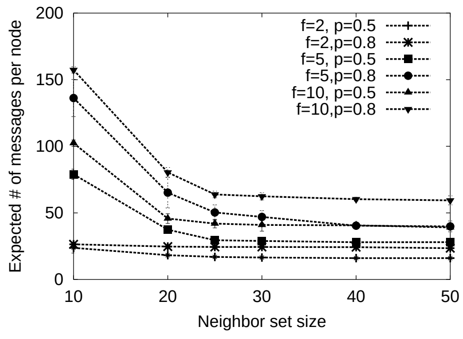


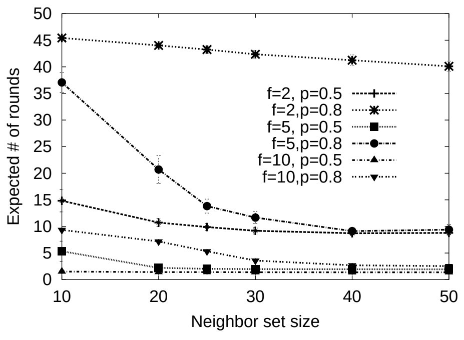
Fig. 8. Non-cooperation search completion rates: Isolated arrival process

Next we examine the effect of varying the size of the neighbor set. The parameters for the number of clients and fanout remain similar to those in the previous experiments. Figure 8 illustrates the fraction of clients that are able to locate the document. The different bars plot these values for various fractions of mute non-cooperating clients and various fanouts. We find when fanout $f = 2$, increasing the neighbor set size does little to improve the likelihood of a search succeeding when the fraction of non-cooperative clients is large. However, such an increase does yield significant improvements when the fanout is 5: increasing the neighbor set size from 10 to 50 changes the fraction of searches that succeed from 0.25 to 0.82 when 80% of the clients are non-cooperative.

Our findings indicate that a fanout of $f = 5$ is sufficient to handle overlays in which large fractions of client are non-cooperative of type mute. With query-only, and tunneling type of non-cooperation, we find that the fraction of clients that are able to locate the object is near 100% even with a fanout as low as $f = 2$ and half of the clients are non-cooperative.



(a) average number of messages



(b) average number of rounds

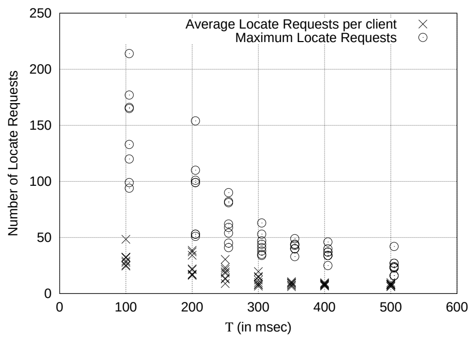
Fig. 9. Non-cooperation overhead: Isolated arrival process

Figure 9 illustrates the average number of messages per client and average number of time units for the same set of simulations used to plot Figure 8. We observe that increasing neighbor set size significantly reduces the messages and the time required to locate the document for smaller fanouts and larger fractions of non-cooperative clients. Again, we observe that a fanout of 5 upon an overlay in which clients' neighbor sets are size 25 keeps the average number messages received per client small (around 25), and the time required less than 5 hops. Even a neighbor set size as small as 10 is sufficient to locate the document within 5 hops when half the clients are non-cooperative. We observe similar trends to that explored in Figure 7 with query-only and tunneling non-participants.

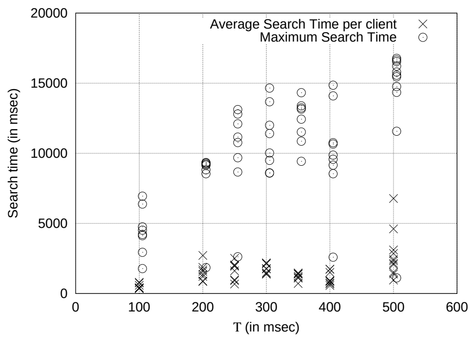
In summary, these simulation results indicate that PROOFS is robust in overlays even when the fraction of clients that are non-cooperative is 0.5.

V. Experiments

In this section, we present results of our use of an experimental prototype within a wide-area network setting. Our experimental testbed consists of a variety of machines gathered at the following academic institutions around the globe: MIT(MA), USC(CA), Columbia (NY), UCL (London), GeorgiaTech (GA), UKentucky (KY), NTUA (Athens, Greece), UNC (NC), CMU (PA), UCSD (CA), UDelaware (DE), UMass (MA), UWisconsin (WI), UoA(Athens, Greece), UMN (Minnesota), and University of Maryland (MD). The hosts yielded a heterogeneous mix of operating systems (mostly Linux and Solaris), bandwidth capabilities, processor speeds and memories.



(a) Traffi c Levels



(b) Delivery Time

Fig. 10. Experiments with 180 clients, simultaneous searches

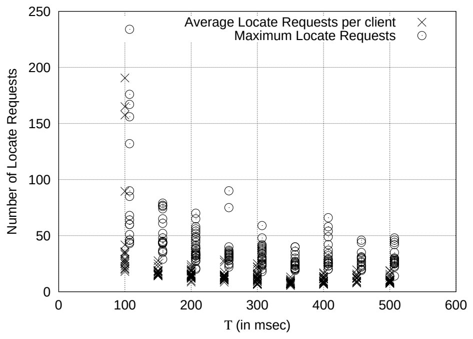
Our goal was to examine PROOFS within a wide scale experiment containing thousands of participating clients. However, doing so would have overloaded the small number of distributed machines to which we had access. To generate more participants, multiple clients (between 5 and 15) were assigned to a single machine as separate processes. Since a client's neighbors are assigned randomly via the shuffling process, the selection of neighbors is not biased by their network or physical proximity. Hence, the only effect that this artificial proximity has on the experiments is that approximately $1/n$th of the time, the end-to-end transmission delay between pairs is smaller than would be expected in practice, where n is the number of hosts.

Our prototype is a multi-threaded Java executable that uses TCP sockets to form and maintain connections between neighbors in the overlay. We selected Java because of its inherent portability to all the machines, though the executable code is slower than what can be achieved by coding in C. By using TCP sockets, we did not need to concern ourselves with handling lost transmissions within the network. When a client shuffles a neighbor away, it closes the TCP socket that leads to the departed neighbor. When a client is informed of a new neighbor (during a shuffle) it then initiates a TCP connection with that neighbor. We also implemented a bootstrap server to provide the clients with a valid sets of neighbors during their startup. In all experiments, the times at which each client initiates shuffle operations are exponentially distributed with an expected time of two minutes between shuffle initiations. We let the shuffling proceed for a half hour before initiating our experiments to give the overlay time to "randomize" itself.

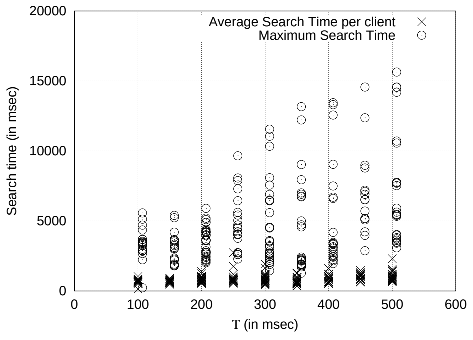
Figure 10 plots results of 8 experiments using an overlay consisting of 180 clients with a neighbor set size of 15. In each experiment, a single client starts with a copy of the object. All other clients simultaneously search for that object using a fanout $f = 2$. Figure 10(a) plots, for each experiment, the average number of query requests received by each client, as well as the maximum number of requests received over all clients. On the x-axis, we vary $\mathcal{T}$, where a client waits $\mathcal{T}t$ milliseconds after initiating a query with TTL t before initiating its next query (the maximum values are shifted slightly to the right to more easily distinguish between average and maximum points). Figure 10(b) plots the corresponding average and maximum times taken from the time that a client's search is initiated to the first time that the client retrieves the object (since multiple copies can be returned due to the parallel nature of the search).

We see that by setting $\mathcal{T}$ to small values, the expected time to delivery is reduced. However, there can be substantial increases in traffic levels due to premature transmission of queries (before previous queries have had a chance to complete). We see that for values of $\mathcal{T} > 300$, average traffic levels are approximately the same, with each client receiving on average fewer than 25 queries to allow all clients to obtain the content. This follows from our observation that typical response times to queries varied between 100ms and 350ms. The results indicate that a client should give ample time for a query to complete its search before starting another.

Figure 11 plots results of between 10 and 25 experiments for each 50 ms increment of $\mathcal{T}$ using a similar setup as before except that here, only 87 clients participate. The conclusions we draw from these plots are roughly the same. We note that traffic levels and retrieval times are roughly the same as for the 180 client case. This indicates that the number of requests and



(a) Traffi c Levels



(b) Delivery Time

Fig. 11. Experiments with 80 clients, simultaneous searches

the retrieval time does indeed grow slowly with the number of clients participating in the system.

These experiments demonstrate that (admittedly, on a smaller scale), PROOFS can retrieve popular objects in an efficient fashion. The time between queries should be no less than 250 msec, giving ample time for the large majority of queries to reach their intended destinations.

VI. Discussion

The appeal of PROOFS is the simplicity, scalability, and robustness of its basic architecture. The fact that often nodes receive redundant copies of queries does increase the levels of traffic it adds to the network. However, this redundancy proves to be helpful in naturally prevent partitions and allows the system to operate effectively even when a large fraction of clients limit their participation.

While we have demonstrated PROOFS' ability to scalably and robustly deliver objects under heavy demand, we have not evaluated the potential damages to the network via misuse or intentional abuse. PROOFS' scalability relies on the fact that the object a client searches for is also being searched for by many other clients in the network. In practice, it is necessary to limit the amount of flooding caused by searches that are not looking for popular content. The impact of such flooding is sufficiently limited by introducing caching mechanisms, such as those recently described in [26]. However, such caching does require users to cache objects that they have not specifically requested. A second option is to place limits on the rate at which clients are willing to service queries. If all clients bound the rate at which they process queries by some fixed r, then each client can only inject queries into the network at a maximum rate of fr (the rate can be lower due to queries for which a copy of the object can be returned). Another way is to place limits on the maximum TTLs for queries. Large TTLs are required when few clients are searching for an object so that their queries cover the majority of nodes in the overlay. In contrast, when numerous clients search for a common object, repeated searches with small TTLs will spread the objects around the overlay quickly as a result of the overlay's randomly connected structure. Hence, the number of small scoped searches that find the object is expected to grow exponentially with time. This rate limiting reduces the usefulness of PROOFS under non-flash crowd conditions, but we expect will not significantly affect performance during actual flash crowds.

We are currently actively investigating mechanisms to prevent denial of service attacks through overlay networks. Our technique compares the rates of incoming traffic from different neighbors, taking into account the "structure" of the overlay. For a PROOFS-like overlay, two neighbors are expected to send queries to a node at roughly the same rate, so that when one neighbor sends queries at a much higher rate than the other, this is likely the result of a DoS attack through that first neighbor. Such an approach not only shows promise in undirected search networks such as that used by PROOFS, but we are also looking at means to implement similar protection in directed search networks proposed in [9], [10], [11], [12], [13].

Our simulations have assumed a homogeneous collection of participants. One possible future direction is to explore the performance of PROOFS where the various participants have differing bandwidth capabilities and differing join/leave patterns.

We have demonstrated that PROOFS can provide robust performance in delivering hot objects as long as some node in the overlay contains a copy of the sought after object. PROOFS can incur significant overheads when used to search for an object that is not "hot" or is unavailable within the P2P network. For the former case, we are investigating mechanisms that can be used to predict the "heat" of an object. The latter remains an open problem, not only in PROOFS, but in all P2P search protocols.

VII. Conclusion

We have presented PROOFS, a system designed to deliver objects whose servers of origins are experiencing flash crowd conditions. The system uses overlays that are formed via a distributed shuffling procedure such that neighbors are selected at random. Randomized, scoped, flooding searches are then used by clients upon the overlay to locate the object that cannot be retrieved from the overwhelmed server. We have shown via a mix of theoretical results, simulation, and experimentation that by relying on randomness, PROOFS can achieve low latency delivery utilizing modest traffic levels, even when membership to the overlay changes dynamically with time and when there exist members that limit their participation in the system.

Acknowledgments

The authors would like to thank the anonymous reviewers for JSAC and ICNP whose comments helped improve the quality of this work.

References

[1] V. Padmanabhan and K. Sripanidkulchai, "The Case for Cooperative Networking," in *Proceedings of IPTPS'02*, Cambridge, MA, March 2002.

[2] V. Padmanabhan, H. Wang, P. Chou, and K. Sripanidkulchai, "Distributing Streaming Media Content Using Cooperative Networking," in *Proceedings of NOSSDAV'02*, Miami Beach, FL, May 2002.

[3] D. Rubenstein and S. Sahu, "An Analysis of a Simple P2P Protocol for Flash Crowd Document Retrieval," Columbia University, Tech. Rep., November 2001.

[4] K. Kong and D. Ghosal, "Mitigating Server-Side Congestion in the Internet through Pseudoserving," *Transactions on Networking*, vol. 7, no. 4, August 1999.

[5] M. Reiter and A. Rubin, "Crowds: Anonymity for Web Transactions," *ACM Transactions on Information and System Security*, vol. 1, no. 1, pp. 66–92, 1998.

[6] A. R. M. Waldman and L. Cranor, "Publius: A Robust, Tamper-evident, Censorship-resistant, Web Publishing System," in *Proc. 9th USENIX Security Symposium*, Denver, CO, August 2000.

[7] M. Papadopouli and H. Schulzrinne, "Effects of Power Conservation, Wireless Coverage and Cooperation on Data Dissemination among Mobile Devices," in *7th Annual Int. Conf. Mobile Computing and Networking (ACM SIGMOBILE)*, Rome, Italy, July 2001.

[8] ——, "Design and Implementation of a Peer-to-Peer Data Dissemination and Prefetching Tool for Mobile Users," in *Proceedings of the First NY Metro Area Networking Workshop*, Hawthorne, New York, March 2001.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," in *Proceedings of ACM SIGCOMM'01*, San Diego, CA, August 2001.

[10] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications," in *Proceedings of ACM SIGCOMM'01*, San Diego, CA, August 2001.

[11] F. Dabek, , F. Kaashoek, R. Morris, D. Karger, and I. Stoica, "Wide-Area Cooperative Storage with CFS," in *Proceedings of ACM SOSP'01*, Banff, Canada, October 2001.

[12] A. Rowstron and P. Druschel, "Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility," in *Proceedings of ACM SOSP'01*, Banff, Canada, October 2001.

[13] B. Zhao, J. Kubiatowicz, and A. Joseph, "Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing," UC Berkeley, Tech. Rep., April 2001.

[14] T. Stading, P. Maniatis, and M. Baker, "Peer-to-Peer Caching Schemes to Address Flash Crowds," in *Proceedings of IPTPS'02*, Cambridge, MA, March 2002.

[15] S. Deering and D. Cheriton, "Multicasting routing in datagram internetworks and extended LANs," *ACM Transactions on Computer Systems*, vol. 8, no. 2, pp. 85–110, May 1990.

[16] E. Zegura, M. Ammar, Z. Fei, and S. Bhattacharjee, "Application-Layer Anycasting: A Server Selection Architecture and Use in a Replicated Web Service," *Transactions on Networking*, vol. 8, no. 4, August 2000.

[17] D. Katabi and J. Wroclawski, "A Framework for Scalable Global IP-Anycast (GIA)," in *Proceedings of ACM SIGCOMM'00*, Stockholm, Sweden, September 2000.

[18] J. Bernabeu-Auban, M. Ammad, and A. Ammar, "Resource Finding in Store and Forward Networks," *Acta Informatica*, vol. 28, 1991.

[19] D. Kempe, J. Kleinberg, and A. Demers, "Spatial Gossip and Resource Location Protocols," in *Proceedings of the Thirty Third Annual ACM Symposium on Theory of Computing (STOC)*, Crete, Greece, July 2001.

[20] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic Algorithms for Replicated Database Maintenance," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, 1987.

[21] R. Karp, S. Shenker, C. Schindelhauer, and B. Vocking, "Randomized rumor spreading," in *41st Symposium on Foundation on Computer Science (FOCS'00)*, Redondo Beach, CA, November 2000.
[22] G. Pandurangan, P. Raghavan, and E. Upfal, "Building Low-Diameter P2P Networks," in *42nd Symposium on Foundation on Computer Science (FOCS'01)*, Las Vegas, NV, October 2001.
[23] S. Saroiu, P. Gummadi, and S. Gribble, "A Measurement Study of Peer-to-Peer File Sharing Systems," University of Washington, UW-CSE-01-06-02, Tech. Rep., July 2001.
[24] M. Ripeanu and I. Foster, "Peer-to-Peer Architecture Case Study: Gnutella Network," University of Chicago, TR-2001-26, Tech. Rep., July 2001.
[25] J. Ritter, "Why Gnutella Can't Scale. No, Really," February 2001, available from http://www.monkey.org/∼dugsong/mirror/gnutella.html.
[26] E. Cohen and S. Shenker, "Replication Strategies in Unstructured Peer-to-Peer Networks," in *Proceedings of the ACM SIGCOMM '02*, Pittsburgh, PA, August 2002.

Angelos Stavrou Angelos Stavrou received his B.S. in Physics with distinction from University of Patras, Greece and an M.S. in theoretical CS from University of Athens, Greece. He also holds an M.S. in Electrical Engineering from Columbia University and he is currently working toward the Ph.D degree at the same university. He is currently a Research Assistant at the COMET Laboratory of Electrical Engineering at Columbia University. His research interests are Peer-to-peer and Overlay Networks, Network Reliability, and Statistical Inference.

Dan Rubenstein Dan Rubenstein has been an Assistant Professor of Electrical Engineering and Computer Science at Columbia University since 2000. He received a B.S. degree in mathematics from M.I.T., an M.A. in math from UCLA, and a PhD in computer science from University of Massachusetts, Amherst. His research interests are in network technologies, applications, and performance analysis, with an emphasis on large-scale Internet design for continuous media transmission. He received a Best Student Paper Award for his ACM SIGMETRICS 2000 paper entitled "Detecting Shared Points of Congestion via End-to-end Measurement" and an NSF CAREER Award in 2002 to continue his investigation of peer-to-peer and overlay networking systems.

Sambit Sahu Sambit Sahu has been a Research Staff Member in the Networking Software and Services group at IBM T.J. Watson Research Center since 2000. He received his Ph.D. degree in Computer Science from the University of Massachusetts, Amherst. Since joining IBM, his research has focused on overlay based communication, content distribution architecture, and design and analysis of high-performance network communication protocols. He has published a number of papers in the areas of differentiated services, multimedia and peer-to-peer communications.