A Lightweight, Robust P2P System to Handle Flash Crowds

Angelos Stavrou* Dan Rubenstein* Sambit Sahu†
*Dept. of Electrical Engineering, Columbia University, New York, NY
†IBM T.J. Watson Research Center, Hawthorne, NY
{angelos,danr}@ee.columbia.edu sambits@us.ibm.com

Columbia University Technical Report EE200321-1
February, 2002

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 analytical modeling, 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.

1 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. Recent 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 accessability 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 be redirected to available resources.

A third approach is to have the clients form a peer-to-peer (P2P) overlay network that allows 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 of a set of protocols that combines

*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.

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 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 slow 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.

PROOFS falls into a class of systems termed *First Generation P2P systems* that also contains P2P systems such as Gnutella [3] in which an object (or information about the precise location of an object) is equally likely to be available at any node within the P2P system. In contrast, *Second Generation P2P systems* (e.g., see [14, 1, 7, 11]) form overlays that, using a variety of clever distributed and dynamic hashing strategies, assign each object to a particular set of clients in the overlay. For an "unpopular" object that resides at a small, fixed number of locations, second generation systems can locate an object using $O(\log n)$ queries, whereas first generation systems require $O(n)$ queries. Thus, second generation systems can provide considerable savings in levels of traffic used for searching as n grows large. However, mathematical and simulation analysis in [12] shows that searches of first generation P2P systems can be designed to have low expected traffic requirements and low latency when searching for objects that are the interest of a flash crowd. This is understood intuitively in that whenever a client locates and subsequently retrieves a copy of the desired object, that client can then service any subsequent queries, cutting down the costs in terms of both time taken and transmissions made of subsequent searches, in effect making the amortized cost of each client's search $O(\log n)$ as well. While in some respects, PROOFS is a step backwards from second generation systems, it has the following advantages over the other proposed state-of-the-art:

- *Clients are not required to cache any objects or pointers to objects other than that which the client has explicitly expressed interest in receiving*. To date, second generation systems that address the flash crowd issue do so by requiring participating clients to explicitly cache copies of objects that are not necessarily of direct interest on the behalf of the system (i.e., for other clients). While technical complications are arguably solvable, it remains unclear whether users would feel comfortable using their own disk space to host unknown content.

- PROOFS handles dynamic changes in overlay membership (i.e., participants joining and leaving the system with time) without any additional mechanism or modification to its fundamental design. In addition, PROOFS is naturally robust even when there exist a substantial number of clients who "take advantage" of the system by using it to obtain popular objects, but who do not fully participate in assisting other clients by either refusing to forward content or even secretly dropping all queries it receives. While some second generation systems have demonstrated certain degrees of robustness against changes in overlay membership, it is unclear how they perform in environments where some clients vary their levels of participation in the forwarding queries and/or delivering stored objects.

- The system is amenable to the formation of complex queries that contain keywords or temporal restrictions (e.g., a copy of an object generated within the last 5 minutes). This is much more difficult to do within second generation systems in which the object description must hash to a unique identifier. In first generation protocols, each client visited parses the query for itself.

Our design is motivated by the observations in [5] that more attention should be paid to the manageability, reliability and robustness of communication systems. Rather than target our main efforts toward minimizing traffic levels and delivery latencies, the system is designed to achieve "good" traffic levels and latencies while remaining robust, reliable, and manageable under a variety of network settings. We demonstrate these claims as well as the scalability of the system to thousands of participants by performing the following tasks:

- Through analytical modeling and simulation, we show that the likelihood that the constructed overlay separates a client from reaching a large fraction of other clients is extremely rare, even in the presence of clients dynamically joining and leaving the overlay.
- Through simulation, we show that traffic levels, latency, and connectivity grow in a tolerable manner as a function of the fraction of overlay nodes cease to perform query and object forwarding (i.e., non-cooperative nodes).
- 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 2, we overview related work. Section 3 describes the basic architecture of PROOFS. In Section 4, we evaluate our design's robustness in the face of dynamic changes to overlay membership and clients who offer limited participation. Section 5 presents experimental results using a prototype version of PROOFS upon the real Internet. We discuss some limitations, future directions, and challenges in Section 6 and conclude in Section 7.

2 Related Work

The idea of flash-crowd alleviation via replication was previously considered in [4]. However, the architecture there involves an elobrate communication and exchange mechanism between servers within the network, having been developed before the notion of peer-to-peer communication gained in popularity. A system whose design is similar in several respects to PROOFS is examined in [12]. There, a mathematical model of a discrete-event version of a randomized, scoped search protocol is analyzed and simulated. They show that upon randomized topologies, such systems can effectively scale to overlays that contain millions of participating clients. However, their evaluation, restricted only to simple mathematical models and basic simulations atop these models does not evaluate the robustness of the approach: the effects of clients joining and leaving the overlay are not considered. In addition, it assumes that all clients are "full participants" in that every client is willing to forward queries as well as return copies of requested objects whenever the client downloaded the object. Here, we focus on the performance when these assumptions are relaxed and also look at the performance of a prototype implementation.

A significant amount of attention has been paid to second generation P2P architectures such as CAN [7], CHORD [14, 1] and PASTRY [11], in which participants have a sense of direction as to where to forward requests. For unpopular documents, second generation architectures clearly provide benefit over their first generation 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. 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) can significantly deteriorate the effectiveness of these approaches.

There has been interesting theoretical work that looks at ways to form "good" topologies for scoped searches. One example is that of [6] 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 [13, 8, 9].

3 Design Description

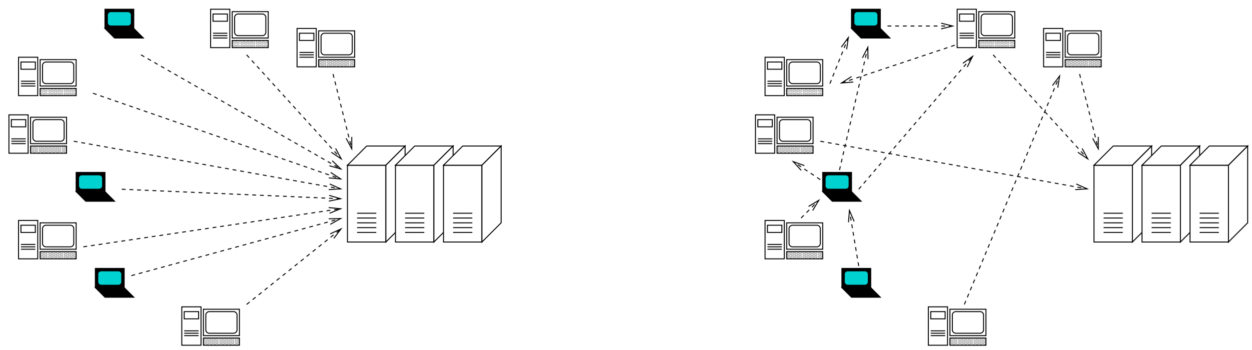
In this section, we describe the application for which the PROOFS system was designed and describe the details of that design. PROOFS purpose is to provide timely delivery of web objects that are stored at locations whose availability is compromised as a result of a heavy request load for the objects. Proofs was designed with the following design objectives:

- Minimize operational complexity: Each client should be responsible for performing a small number of simple tasks (perhaps repeated several times). A flow-chart diagram of a node's operation should be short and simple, minimizing the likelihood of implementation error.
- Minimize state: To form an overlay, clients must maintain a list of neighbors that can be contacted for the purposes of a search. Clients also maintain those pages that are of interest to them. It is preferred not to require clients to maintain additional state for purposes such as monitoring or sharing of network statistics, or for the caching of objects not explicitly requested by that client. Furhtermore, the state should be "soft" in that any incorrect perceptions about the operating environment do not prevent the system from performing its task (but may decrease system efficiency) such that this state can simply expire with time.
- Limit recovery code: often protocols require additional complexity to "heal", e.g., recover from network partitions or adapt the overlay to dynamic changes in membership (clients joining/leaving the overlay). We wish to remove any such additional mechanism except for what is required to bootstrap the system into operation.
- Naturally cope with limited participation: some clients may refuse to deliver objects they have received. Others may refuse to forward queries, and worse yet, some clients may not wish to reveal their refusal to assist. The system should continue to function properly and efficiently even as these non-participants grow to significant, but not overwhelming proportions.
- The ability to put richer semantics within the query, including temporal specifications.

3.1 Application of PROOFS

Figure 1 pictorially demonstrates how PROOFS alleviates symptoms associated with flash crowds. In Figure 1(a), a set of end-systems is attempting to receive the same object at roughly the same time directly servers containing the object. DNS requests issued by these end-systems would point these systems to a small set of servers from which they can receive the object. For instance, a recent query to the DNS for cnn.com[1] returned 6 IP addresses to which http queries can be transmitted. This small number of sites cannot handle sudden, huge increases in requests. To redirect these initial requests, DNS would require substantial modifications in order to quickly update DNS entries throughout the Internet to prevent clients sending queries

[1] on 1/9/2002 from host medellin.cs.columbia.edu via the Linux command *host cnn.com*



(a) "Standard" architecture

(b) PROOFS-enhanced architecture

Figure 1: An example of how PROOFS assists in timely object retrieval.

to the overloaded servers. While deployed content delivery solutions are able to offset the load imposed on these servers for sub-objects (JPGs, ads, etc.), they are unable to offset the load for the original request for what is often referred to as the *container page*. Figure 1(a) is an example where several end-systems place temporally-adjacent requests for the same object, overloading the capacity of the servers. As a result, the majority of requests remain unanswered and fail, frustrating many of the users that placed the request. In Figure 1(b), with the PROOFS system activated, end-systems can query other end-systems for a copy of the object. Presumably, the system works well if end-system queries for the object reach end-systems that have already obtained a copy of the requested object. Our design leverages off the theoretical and simulation results in [12] that analytically demonstrate that in theory, randomized scoped searches between P2P participants all looking for the same popular object scale well in terms of number of packets transmitted and time taken to retrieve the object.

3.2 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. There are two components to 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 provide a means by which clients can learn about and gain access to the overlay. In our current implementation, we utilize a single bootstrap server. For the system to be robust, it is likely necessary to have multiple bootstrap servers. Below, we limit discussion to a system that contains a single bootstrap server. We briefly discuss some straightforward ways to provide multiple servers in Section 6. A detailed exploration is beyond the scope of this paper.

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` runs 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. The underlying routing system is not of concern in this paper.

3.2.1 `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 no greater than 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. Increase the size of the the neighbor set if below the bound before overwriting previous entries
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

Figure 2: An example of a shuffle operation

A sample shuffle operation is shown in Figure 2. 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 2(a) and end with the set of neighbors depicted in Figure 2(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.

3.2.2 `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 6), a *query* is initiated at the client. A query contains the following information:

- **Object**: a description of the object being searched for. In our current implementation, the description is restricted to the URL that describes the original location of the page. However, the description can easily be extended to handle more sophisticated queries (keywords, temporal specifications, etc.) since the set of locations searched are independent of the object specification.
- **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. Because each search is randomized, even the first few hops of the new query can visit clients that were not visited on previous queries. The time time a client waits between subsequent queries is $t\mathcal{T}$, where t is the TTL of the query and $\mathcal{T}$ is some rough estimate of propagation delay. The size of the TTL must be chosen carefully. The number of visits to clients grows exponentially in the size of the TTL, so rapid increases in its value can cause unnecessary flooding within the network. $\mathcal{T}$ must also be set carefully: large values increase potential waiting times, but small values can lead to the initiation of new queries prior to the completion of previous queries that may yet return a copy of the object. We investigate how the setting of $\mathcal{T}$ affects retrieval times and traffic levels in Section 5, and discuss ways to avoid traffic flooding in Section 6.

4 Robustness

In this section, we evaluate the natural robustness of PROOFS. By "natural", we mean that no additional functionality is introduced beyond what implements the most basic functions needed by the protocol (as is described in Section 3). 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 paritions, 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 occurences 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. 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`. Clients that do not participate in the `ConstructOverlay` Protocol maintain a fixed set of neighbors throughout the duration of their session. This limits their own ability to retrieve content as some of these neighbors may leave the system.[2] We show that, even with up to 80% of clients limiting their participation, our design maintains acceptable traffic levels and times for object delivery.

4.1 Overlay Partitioning

We say that an overlay is **partitioned** if there exists a pair of clients, n_1 and n_2 in the overlay where no path exists from n_1 to n_2. Such an occurence 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.

The practical complications in maintaining an undirected overlay graph (where an edge permits bi-directional communication between the nodes it connects) compels us to use an overlay whose edges are

[2]Note that under the bootstrapping process, these nodes are assigned as neighbors to other nodes and remain as parents until explicitly removed.

unidirectional. Unfortunately, it is conceivable that shuffling operations will partition the underlying overlay. However, we now present a theoretical result that demonstrates that any partitioning of the graph due to shuffling is only temporary. With probability one (given enough time), the shuffling process will eventually reconnect the separated participants. We emphasize that this theoretical result holds conclusively only when nodes do not leave the overlay.[3]

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.

Theorem 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 neighbor set to swap).[4] 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 3(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.

(a) Initial topology and no swaps

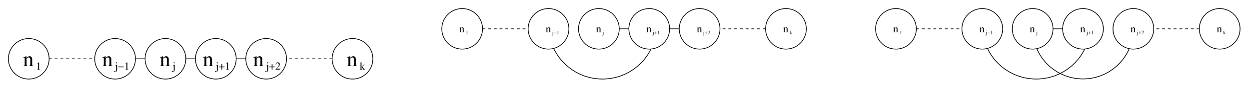
(b) One swap on the path

Figure 3: A generic path where the node being swapped with is off the path.

- 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 3(b), a path between n_1 and n_k remains after the swap.

[3]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.

[4]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

(c) Two swaps on the path

Figure 4: A generic path where the node being swapped with is on the path.

- 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} or 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 4, 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}.

■

Theorem 2 *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: Due to lack of space, we simply present a sketch of the proof. Consider the undirected path between n_1 and n_k. Then, by induction on i, we perform a procedure that builds a directed path from n_1 to a node n' where n' is at most $k-i$ hops from n_k along an undirected path. Once i reaches k we have achieved our result. By choosing the closest node n' to n_k (in the undirected sense) that can be reached via directed edges from n_1 and via undirected edges to n_k, the direction of the edge between n' and the next hop node, n'' on the undirected path can be reversed by having n'' initiate a shuffle operation with n'. The undirected component of the path to n_k is now one hop shorter, either because of the additional hop on the directed path from n' to n'', or because n'' transfered its directed edge to the next hop on the path to n' during the shuffle. Since the proper sequence of shuffling operations is a finite set of shuffles, with probability one an appropriate sequence is eventually selected. ■

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.

4.2 Handling Dynamic Joins and Leaves

We now evaluate 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. However, we use the following result of Erdös and Rényi in [2] to argue that partitions in the directed graph are highly unlikely.

Theorem 3 (Erdös) *Let G be a graph containing n nodes and $1/2n \log n + \alpha n + o(n)$ edges where the nodes connected by the edge are drawn from a uniform distribution over the set of all possible edges. Then the graph is not paritioned (i.e., is connected) with probability* $\exp(-e^{-2\alpha})$ *as* $n \to \infty$.

By setting α to a reasonable-sized value (e.g., 11), the probability of such a random graph being partitioned is less than 10^{-9}. Since nodes in such graphs have an expected degree of $1/2 \log n + \alpha + o(1)$, a one million node graph, with $\alpha = 11$, would attain this low partitioning probability when each node has an average of 13 neighbors. We suspect that the distribution of graphs generated by shuffling combined with dynamic joins and leaves of clients is similar, though not exactly the same, to the distribution of graphs generated in Theorem 3 such that a similar result would hold and hence, the likelihood of the undirected graph paritioning for a reasonably-sized neighbor set is miniscule. However, we have yet to formally prove this result. Instead, we resort to simulation to make our case.

We now present simulation results to evaluate how clients joining and leaving the overlay affect the overlay's ability to provide communication between arbitrary sets of clients. 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 left 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 would inform the bootstrap server of its arrival, obtaining a new list of neighbors from the bootstrap server and updating the bootstrap server's (potentially short) list of active participants. We varied the likelihood with which a client would inform the bootstrap server of its departure from the overlay. However, we found 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 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, the fact that the times between samples are exponentially distributed guarantees that the samples indeed reflect steady-state behavior.[5]

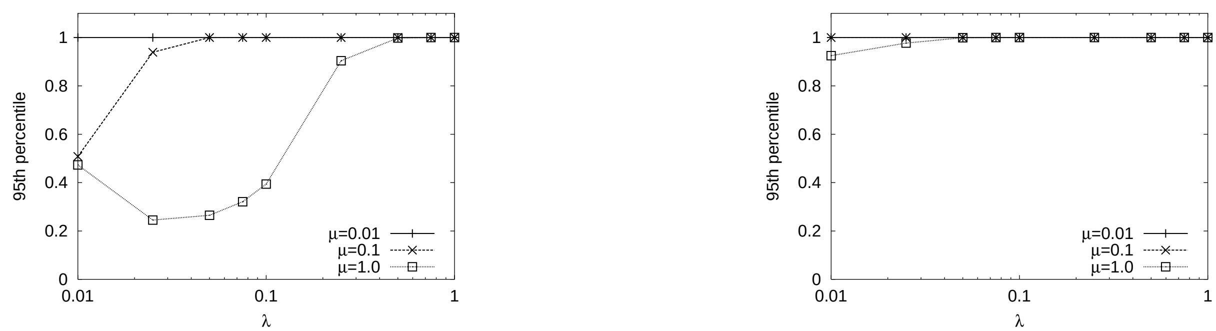
During each sample, for each active client $\mathcal{C}$ (currently joined to the overlay), we computed the fraction of other active clients that can be reached by $\mathcal{C}$ via some path along a sequence of directed edges within the overlay graph. We call this quantity the *reachability* of $\mathcal{C}$. During each sample, we compute the minimum, mean, median, and maximum reachability over all active clients. Table 1 lists the set of parameters varied during experiments as well as the values to which the different parameters were set.

Table 1: 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
bootsrap server cache	5, 10, 50, 100

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

4.2.1 Results



(a) minimum

(b) average

Figure 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 level of reachability that is exceeded in at least 95% of the samples by *all* clients. In other words, fewer than 1 of 20 samples should contain a client whose reachability is lower than the values plotted in the figure. 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 used in place 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 now discuss the case where the set of clients are fixed is covered by setting $\mu = 0$ (clients join and never leave). We omit the plots since they overlap with the case where $\mu = 0.01$. In summary, in our experiments with $\mu = 0$, client reachability dipped below 1 less than 5% of the time, and never dipped below 0.95.

We now examine how the size of clients' neighbor sets affects minimum reachability. In Figure 6, a point plotted at (x, y) indicates that the minimum client has reachability of y for at least a fraction x of the time. The different curves are for the different sizes of neighborhoods. We see that increasing the neighborhood size has a dramatic effect on the reachability when $\lambda << \mu$. Theorem 3 gives us intuition that the size of the neighborhood must grow at a rate proportional to the $\log$ of the number of participating clients.

We conclude the examination of the reachability within overlays generated by the shuffling algorithm by noting 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

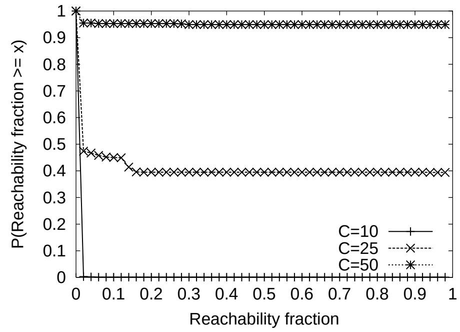


Figure 6: Effect of neighborhood size

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.

4.3 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, we must account for the fact that not all clients will be willing to fully participate. In some cases, clients may even attempt to deceive others about their levels/ability to participate [13]. Often, 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.[6]
- **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[7] and the average time taken for a client to receive the requested object. In these simulations, time is measured in *hops*: the time for a client to communicate with another client (i.e., forward a query) takes a single time unit. A client can transmit an unlimited number of queries to neighboring clients within the same time unit.

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

[7] 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.

Following the lead of [12], 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. This is implemented by probabilistically initializing a client's search that has not yet begun at each time unit. The probability for time unit t is $p + q^{n(t)}$, where p and q are constant 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 p or is "told" about the object by each other client that has already started its search independently with probability q. In our experiments, we set $p = 0.001$ and $q = 0.01$.

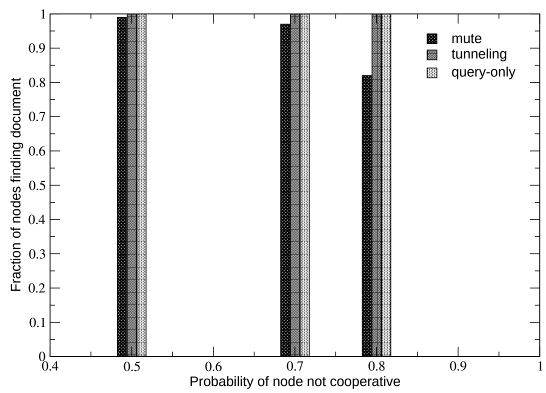
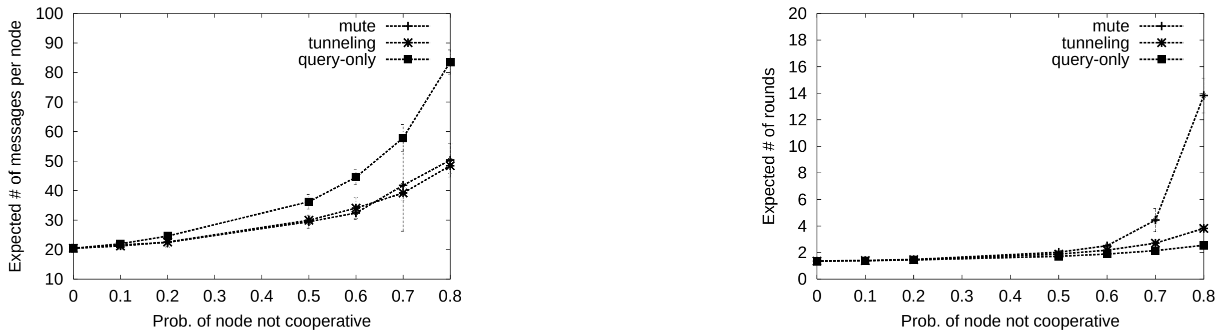


Figure 7: Non-cooperation search completion rates: Isolated arrival process

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 7 and 8 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 7 and different curves in Figure 8. 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).

Figure 7 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 8 plots the average number of messages per client and average time units required using the isolated arrival process. From Figure 8(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



(a) average number of messages

(b) average number of time units

Figure 8: Non-cooperation overhead: Isolated arrival process

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 note a rather 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 use a significantly larger amount of traffic.

Figure 8(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 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 ran 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.

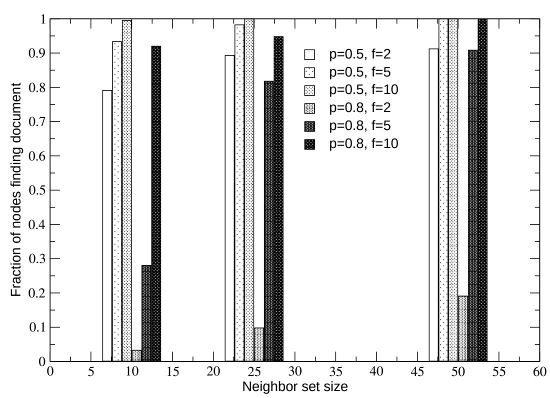
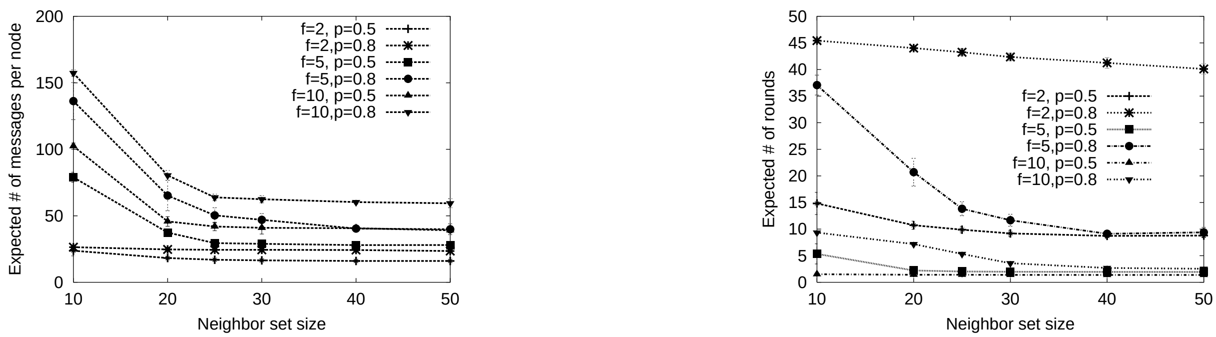


Figure 9: Non-cooperation search completion rates: Isolated arrival process

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 9 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 succed 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

Figure 10: Non-cooperation overhead: Isolated arrival process

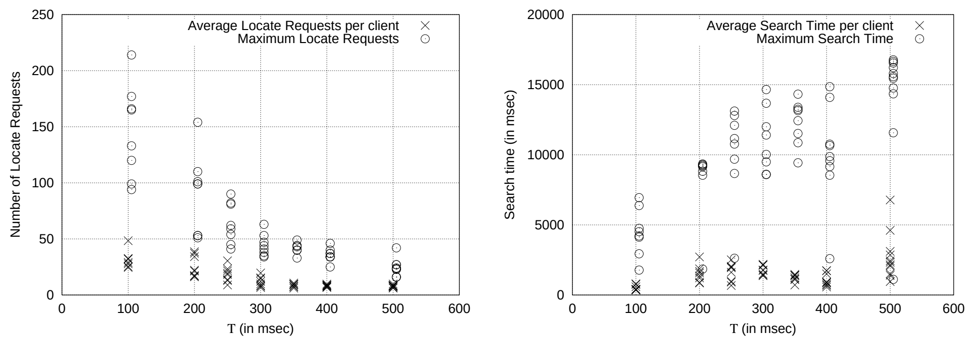
Figure 10 illustrates the average number of messages per client and average number of time units for the same set of simulations used to plot Figure 9. We observe that increasing neighbor set size significantly reduces the messages and the time required to locate the document for smaller fanouts and 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 8 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 0.5.

5 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 (Minnessota), and University of Maryland (MD). The hosts yielded a hetergeneous mix of operating systems (mostly Linux and Solaris), bandwidth capabilities, processor speeds and memories.

Our goal was to examine PROOFS within a wide scale experiment containing thousands of participating



(a) Traffic Levels

(b) Delivery Time

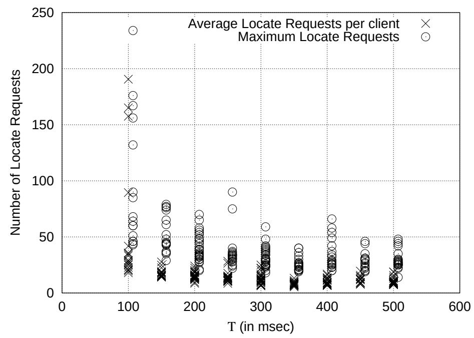
Figure 11: Experiments with 180 clients, simultaneous searches

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.

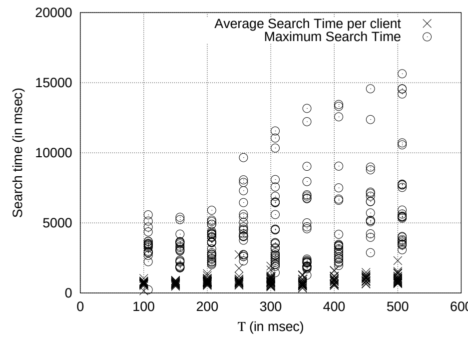
Our prototype is a multi-threaded Java executable that uses TCP sockets to form and maintain connections between neighbors in the overlay. We selected Java because of its inherent portability to all the machines, though the executable code is slower than what can be achieved by coding in C. By using 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 initations. We let the shuffling proceed for a half hour before initiating our experiments to give the overlay time to "randomize" itself.

Figure 11 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 11(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 distinguihs between average and maximum points). Figure 11(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



(a) Traffic Levels



(b) Delivery Time

Figure 12: Experiments with 80 clients, simultaneous searches

another.

Figure 12 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 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.

6 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. We envision two simple ways to control such flooding:

- **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). This can lead to a high query drop rate (i.e., processing only one out of every f queries), worsening performance. However, it should effectively bound the amount of traffic that PROOFS adds to the network, irrespective of the number of clients searching or participating in the overlay. Second, we have run other sets of simulations (not presented here) in which each client participating in the overlay drops requests with a probability of p. The results are more favorable than what we have observed when a fraction, p, of clients drop all requests. Hence, PROOFS should continue to locate objects efficiently even when the query rate modestly exceeds r.

- **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.

The following is a list of open issues that still require further investigation:

- **Search Initiation**: We are interested in automating PROOFS inside of a web browser to automatically retrieve objects during flash crowd conditions. A reasonable approach is to first attempt to contact the server and, after a short timeout, initiate `LocateObject`.
- **DoS attacks**: Protocols that fan out requests can be used to generate large amounts traffic in the network by placing bogus queries. While the rate-limiting and TTL-bounding techniques can protect the rest of the network from being overwhelmed with queries, the generation of a large number of bogus queries can suppress the ability to service valid queries. One fix would be to prioritize the servicing of queries to sites that are more likely to legitimately contain flash crowds such as news sites. Another possibility is to prioritize service of the more frequently occuring queries, which are more likely to be legitimate.
- **Unavailable Objects**: When an object does not reside anywhere within the overlay, it cannot be retrieved, nor replicated at intermediate points in the search space. This means that searches for that object will flood the system. Handling this case remains an open problem. We point out that this problem also exists within second generation approaches that rely on caching to prevent flooding the focal point of a directed query for a popular object [1, 7, 11].
- **Neighbor Proximity**: We have not made any attempt whatsoever to shape the overlay to the underlying network topology. We find that clients can recover popular objects in a small number of seconds upon an overlay produced by simple shuffle operations. It remains to be seen whether or not optimizing the overlay topology can significantly reduce search times, given that it will also reduce the "randomness" of the search within the overlay graph since searches will tend to cluster more within local geographical areas.
- **Failed Bootstrap Server**: Having a single bootstrap server is a limitation that is easily addressed by replicating the bootstrap server at several fixed locations.
- **Object verification**: A client participating in the overlay could easily transmit a fake copy of the requested object upon receiving a query. For sites that are visited frequently, a browser could obtain a copy of a public key used by the site before a flash crowd arrives at the site. By including a unique certificate into an object (such as an MD5 digest of the object [10]), encrypted by the originating site's private key, this certificate could be used to verify that an object did indeed originate from where it was claimed to have originated from.

There are several ways to optimize the manner in which the overlay is constructed that could potentially improve the protocol's performance. Our simulations assumed that all clients had identical capabilities and our experiments were conducted upon well-connected, well-provisioned end-systems at academic institutions. One immediate direction of future work is to determine how to construct overlays upon which randomized searches proceed efficiently through the overlay graph with an increased use of high bandwidth clients and a reduced use of low bandwidth clients.

One example is using the method described in [6] to generate bounded-diameter overlays. Another would be to give preference to neighbors who are nearby (either topologically, via hop-count, or end-to-end delay). A third is to focus design toward graphs that exhibit small world phenomena. We are interested in pursuing these directions as future work. It would also be interesting to theoretically prove results involving the "shape" of the graphs generated by shuffling. For instance, are they truly random, and if so, how quickly do they converge to a random shape? It would also be of interest to analytically model the behavior of this type and other protocols as membership, levels of participation, and processing capabilities of clients are varied.

Finally, we note that it is convenient to have connections between neighbors in the overlay maintained via open TCP connections so that we need not worry about lost messages. Since these connections are bidirectional, it would be worth considering allowing them to be used by both endpoints, effectively making the overlay graph an undirected graph. In theory, since directed overlays can partition much more frequently than their underlying undirected structures, making each edge in the overlay graph bidirectional would improve clients' expected reachabilities.

7 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.

References

[1] 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.

[2] P. Erdös and A. Rényi. On the Strength of Connectedness of a Random Graph. *Acta Mathematica Academiae Scientiarum Hungaricae*, 12, August 1961.

[3] The Gnutella Protocol Specification v0.4, revision 1.2. Available from http://gnutella.wego.com.

[4] K. Kong and D. Ghosal. Mitigating Server-Side Congestion in the Internet through Pseudoserving. *Transactions on Networking*, 7(4), August 1999.

[5] Committee on Research Horizons in Networking. *Looking Over the Fence at Networks: A Neighbor's View of Networking Research*. National Academy Press, Washington D.C., 2001.

[6] 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.

[7] 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.

[8] M. Ripeanu and I. Foster. Peer-to-Peer Architecture Case Study: Gnutella Network. Technical report, University of Chicago, TR-2001-26, July 2001.

[9] J. Ritter. Why Gnutella Can't Scale. No, Really, February 2001. Available from http://www.monkey.org/∼dugsong/mirror/gnutella.html.

[10] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, April 1992.

[11] 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.

[12] D. Rubenstein and S. Sahu. An Analysis of a Simple P2P Protocol for Flash Crowd Document Retrieval. Technical report, Columbia University, November 2001.

[13] S. Saroiu, P. Gummadi, and S. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. Technical report, University of Washington, UW-CSE-01-06-02, July 2001.

[14] 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.