Peer-to-Peer Caching Schemes to Address Flash Crowds

Tyron Stading Petros Maniatis Mary Baker

Computer Science Department
Stanford University
Stanford, CA 94305, USA
{tstading, maniatis, mgbaker}@cs.stanford.edu
http://identiscape.stanford.edu/

Abstract. Flash crowds can cripple a web site's performance. Since they are infrequent and unpredictable, these floods do not justify the cost of traditional commercial solutions. We describe *Backslash*, a collaborative web mirroring system run by a collective of web sites that wish to protect themselves from flash crowds. Backslash is built on a distributed hash table overlay and uses the structure of the overlay to cache aggressively a resource that experiences an uncharacteristically high request load. By redirecting requests for that resource uniformly to the created caches, Backslash helps alleviate the effects of flash crowds. We explore cache diffusion techniques for use in such a system and find that probabilistic forwarding improves load distribution albeit not dramatically[1].

1 Introduction

Flash crowds have been the bane of many web masters since the web's explosion in mainstream popularity. The term "flash crowd" is used to describe the unanticipated, massive, rapid increase in the popularity of a resource, such as a web page, that lasts for a short amount of time.

Although their long term effects are hardly noticeable, in the short term, flash crowds incur unbearably high loads on web servers, gateway routers and links. They render the affected resources and any collocated resource unavailable to the rest of the world. Flash crowds are also relatively easy to cause. A mere mention of an interesting web page address in a popular news feed can result in an instant flood that lasts as long as the attention span of the news feed audience. In fact, flash crowds have been commonly referred to as "the Slashdot effect," from the name of the popular news feed, which has caused quite a few floods with its stories.

Although the concept of a malicious flash crowd is certainly within the realm of possibility, the intent behind the effect is usually impossible to distinguish in real time. Therefore in practice, it is important to understand how to adapt

[1] Appears in the Proceedings of the 1st International Workshop on Peer-To-Peer Systems (IPTPS 2002), Cambridge, MA, USA. March 2002

efficiently to the changing resource demands so as to distribute the unexpected high load among available resources, regardless of the intent.

Commercial solutions have previously addressed this problem for very popular sites, such as large corporations with extensive web presence. Companies such as Akamai earn their income by distributing the load of highly trafficked web sites across a geographically dispersed network in advance. Akamai's solution focuses primarily on using proprietary networks and strategically placed dedicated caching centers to intercept and serve customer requests before they become a flood.

However, for sites such as non-profit organizations, schools and governments, which do not generally *expect* flash crowds, the cost of a high-profile content distribution solution such as Akamai's is not justifiable. Such sites have currently no recourse other than to overprovision or to pay the price of the occasional disastrous flash crowd including unavailability, prolonged recovery, ISP penalties and loss of legitimate, desirable traffic.

The purpose of this paper is to introduce, motivate, describe and begin evaluating *Backslash*, a grassroots web content distribution system based on peer-to-peer overlays. Backslash is a collaborative, scalable web mirroring service run and maintained by a collective of content providers who do not expect consistently heavy traffic to their sites. It relies on a content-addressable overlay [3, 5, 6, 8] for the self-organization of participants, for routing requests and for load balancing.

We use the remainder of this paper to identify the requirements from such a system and to present the overall design in more detail. We focus on the caching aspects of Backslash and limit the scope of the evaluation section to cache diffusion issues. We conclude with a research agenda for further work in this area.

2 System Requirements

In this section we outline the basic requirements for our grassroots web mirroring system.

Backslash is intended as a drop-in replacement for current web servers and reverse proxy caches. The driving requirement for its development is to make deployment completely transparent to the client web browsers.

The setting in which we hope to deploy the system is, for example, a collective of several universities or research institutions (possibly up to several thousand). Each institution dedicates to Backslash a well-connected low-end PC-grade computer; at the time of this writing any Pentium II-class computer with 128 MB of RAM should suffice. Node-to-node links have bandwidths between one and 10 Mbps, and latencies between 10 and 300ms. Client-to-node link characteristics range from 56 Kbps modems to perhaps cable or ADSL home connections. Each node stores a complete copy of the data collection published by its hosting site—that is, the Stanford Backslash node holds the entire web collection of the `www.stanford.edu` web site— and has enough free storage for caching. We expect the available free space to be a small multiple, say two or three times, of the local collection size.

The objective of Backslash is to offer fair load distribution in the face of flash crowds. Our primary interest is to limit the load on any participating node so as not to overwhelm it, by distributing requests among as many participants as possible. However, we consider the task of identifying and penalizing documents that consistently exhibit disproportionately high popularity to be out of scope for this paper. Similarly, we ignore the security implications of malicious Backslash nodes at this early stage of this work.

Finally, we ignore problems with the mirroring of dynamically generated content. The problem of mirroring static content is, by itself, a formidable one in the grassroots context. Consequently, we tackle it first, before taking on the much harder problem of dynamic content.

3 Design

In the next few sections we describe the design of Backslash at a high level. We first present how Backslash bridges the gap between the resource location subsystem and the traditional browser-server relationship (Section 3.1). Then, we describe the resource location subsystem, which is based on the peer-to-peer *Distributed Hash Table* paradigm (Section 3.2). Finally, we go into cache diffusion in more detail (Section 3.3).

3.1 Redirection

Every Backslash node is primarily a regular web server for the document collection of the hosting site. During its normal mode of operation, that is, as long as the request load perceived by the node is manageable, a Backslash node does little more than what a normal web server does.

When an increased request load is perceived, the Backslash node switches into one of two special modes of operation: the *pre-overload* mode, in which the node sees uncharacteristically high load but is still not overwhelmed, and the *overload* mode, in which the node is nearly overrun with requests. In the pre-overload mode of operation, the node satisfies all requests that arrive, but diverts subsequent requests to associated resources, such as embedded images, away from itself. In the overload mode, the node redirects all requests it receives to surrogate Backslash nodes and otherwise serves no content. Every node has two locally defined *load thresholds* that determine the boundaries of the normal, pre-overload and overload modes.

Backslash nodes diffuse some of the load directed at a flooded document collection via the use of URL rewriting. A node in pre-overload mode overwrites the embedded URLs of the documents it returns so as to divert subsequent follow-up requests. Such requests—for example, embedded images—are directed instead to surrogate Backslash nodes. URL rewriting takes advantage of the two stages of which web requests commonly consist: the DNS lookup and the HTTP request. In fact, every Backslash node runs a simplified DNS server to intercept DNS requests caused by URL rewrites.

Both types of URL rewrites have the same goal: to cause the client browser to look elsewhere for the flooded document. The DNS-based rewrite accomplishes this by directing the DNS lookups for the hostname of the rewritten URL to a Backslash DNS server. For example, the original URL `http://www.backslash.stanford.edu/image.jpg` is rewritten as `http://<hash>.backslash.berkeley.edu/www.backslash.stanford.edu/image.jpg`, so as to redirect the requester to a surrogate Backslash node at Berkeley, where `<hash>` denotes the base-32 encoding of a SHA-1 hash of the entire original URL.

Similarly, the HTTP-based rewrite accomplishes the same thing by naming a specific surrogate IP address within the rewritten URL. For example, the original URL `http://www.backslash.stanford.edu/image.jpg` is rewritten as `http://a.b.c.d/www.backslash.stanford.edu/image.jpg`, where `a.b.c.d` is the IP address of a Backslash node at Berkeley.

Although functionally similar, the two rewrite techniques have different performance implications. The DNS-based rewrite can overlap the document location task, triggered by an intercepted DNS request at the surrogate node, with the HTTP/TCP client connection establishment that follows. On the other hand, DNS requests can result in long latencies in high-loss environments because of UDP time-outs, especially for wireless clients. As a result, embedded links served to client browsers coming from "nearby" network locations use DNS rewriting, whereas HTTP rewriting is used for more remote clients, based on local policy.

When URL rewriting is not an option, specifically in the case of the first request to an overloaded node from a particular client, plain redirection is used, again either via DNS or HTTP. For example, the mini DNS server responsible for the `backslash.stanford.edu` domain (which is a Backslash node itself) can return the IP address of a surrogate node when asked for the `A` record of `www`, when the Stanford site is in overload mode. Similarly, HTTP redirection uses `REDIRECT` responses to cause browsers to retry a request at a rewritten URL.

The combination of URL rewriting and redirection allows unaware client browsers to reach an unloaded surrogate Backslash node that will serve their requests. We explain how surrogate Backslash nodes serve requests for content from other sites' collections in the next section.

3.2 Resource Location

Once a surrogate Backslash node has received a request for a document of the afflicted site, it has to act as a gateway between the HTTP client browser and the collaborative mirroring portion of Backslash.

Mirroring is implemented on a peer-to-peer overlay following the *distributed hash table* paradigm. Systems in this category [3, 5, 6, 8] implement a hash table over a large number of self-organized nodes. Each node is responsible for a chunk of the entire hash table. If the hash function used by the table is uniform, then regardless of the distribution of resource names stored, resources are distributed uniformly over the hash space. As long as the chunks of the hash space assigned to participating nodes are of roughly equal size, then each node maintains a roughly equal portion of all resources stored into the distributed hash table, thereby achieving load balancing.

Backslash is specifically implemented on the Content Addressable Network [5], but does not rely on the specifics of CAN for its operation. The overlay used underneath Backslash is mostly interchangeable with any other distributed hash table. In addition to hash table operations, Backslash requires knowledge about the neighborhood of an overlay node, but all such popular systems can be easily modified to export this information through their APIs.

3.3 Caching and Replication

Although using a distributed hash table, such as a CAN, explains how we find a copy of a popular document within the Backslash web mirror, it does not explain how the copy was created or propagated through the system. In this section we explain the basic cache diffusion techniques we explore in the context of Backslash.

Each Backslash node has some available storage for use in caching (a few times the size of its local document collection). This storage is split in two categories: *replica* space and *temporary cache* space. On one hand, a replica is a cached copy of a document that is guaranteed to be where it was placed. Replicas are placed in the overlay by insertion operations of the distributed hash table. A temporary cache, on the other hand, is a cached copy of a document that is placed opportunistically at a node of the overlay to speed up subsequent retrievals. Temporary caches are created in response to retrieval operations of the distributed hash table and are not guaranteed to remain where they are placed. In fact, they might be replaced very soon after they are created if they are the least recently used temporarily cached document of a node. A fixed portion of the available free space of each node is allocated as replica space. Whatever remains unused in the replica space and the remainder of the free space is allocated as temporary cache space.

The Backslash replica space is used exclusively for the first copy of each file in the participating mirrored web collections. Every Backslash node periodically injects the documents in its local document collection into the distributed hash table. The single copy of each such document created at insertion time is a replica. In the cache diffusion schemes we explore in the remainder of this paper we create no other replicas.

The first cache diffusion method we consider is *local diffusion*. In local diffusion, each node serving a document as a replica or temporary cache monitors the rate of requests it receives for that document. When the node determines that the request rate has reached a predetermined *push threshold*, it pushes out a new temporary cache of the document one overlay hop closer to the source of the last request. This technique aims to offload some of the demand by having more nodes in the locality of an observed flood intercept and serve requests. In a sense, a node that observes a local flood creates a "bubble" of temporary caches around itself, diffusing its load over its neighborhood. The diameter of the bubble grows in relation to the intensity of the flood, until no node on the perimeter of the bubble observes high request rates for the document.

The second cache diffusion method we consider is *directory diffusion*. In this method, the distributed hash table stores directories of pointers to document copies instead of the document copies themselves. Replicas and temporary caches are also stored in Backslash nodes, but their location is not related to the hash table structure. When a node receives a newly inserted document, it creates a directory for it, picks a random Backslash node and stores a replica for the document at that node, documenting it in its own directory. When the directory receives a request for the document, it returns as many permuted directory entries pointing to individual copies of the document as it can fit in a single response packet. To create new temporary caches, the directory node monitors the request rate for the document. When the request rate reaches a predetermined threshold, the directory responds to the requester with an invitation to become a new temporary cache along with the list of pointers to copies of the file.

Both cache diffusion techniques require that a node serve a request if it holds a copy of the requested document. We explore a modification of this requirement whereby a node may choose (at random) to forward a request even if it already holds a copy of the requested document. This allows the node to shed probabilistically a fraction of the request load it observes, without creating new temporary caches. We introduce this variation, called *probabilistic forwarding*, to increase the reuse of already existing caches and curtail the creation of new ones. This is especially the case for the local diffusion method, where all requests originating outside the "bubble" are handled by the nodes at the perimeter, leaving caches inside the bubble practically unused. Probabilistic forwarding enables the use of caches in the interior.

4 Evaluation

In our preliminary evaluation efforts, we have focused on the behavior of cache diffusion techniques. The mechanisms responsible for interjecting Backslash into the protocol stream of unaware client browsers (delineated in Section 3.1) or for building simple self-maintained overlays (pointed to in Section 3.2) are available and pose no significant challenges for the purposes of our target application.

Our experimental setup involves 1,000 nodes participating in a single two-dimensional CAN overlay. Each node has twice the size of its own collection in available free space, of which exactly half is allocated to replicas, and the other half to temporary caches. For simplicity, the document collection owned by each node consists of a single document and all documents have exactly the same size. We present a brief preliminary exploration of two particular design choices in our cache diffusion mechanism: diffusion agility and probabilistic forwarding.

Diffusion agility is the speed with which Backslash reacts to a new flash crowd. A highly agile diffusion mechanism spreads out cached copies of the flooded document rapidly, so as to reach a state where the downpour of requests for that document can be served collectively by as many nodes as possible. How-

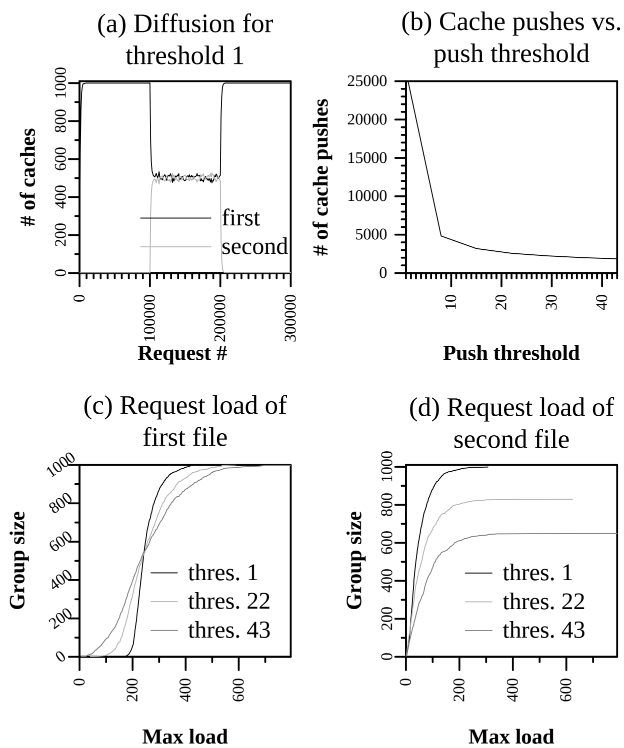

Fig. 1. The simultaneous flash crowd scenario discussed in the text. (a) Cache diffusion during the evolution of the scenario. (b) Number of cache pushes as a function of the push threshold. (c) and (d) Cumulative load distribution of requests served as a function of the number of requests served per node for the first, long flood and the second, short flood, respectively.

ever, high agility also carries an early commitment of heavy resources (storage space, cache diffusion bandwidth) to a flood that might not necessitate them. By controlling agility, we allow the system to moderate the amount of resources it commits to a particular flood.

We represent this agility parameter by a *push threshold*, the number of requests a node must serve before it decides to push to its neighbors a copy of the flooded file. A push threshold of one means that every time a Backslash node receives a request, it also pushes out a copy of the requested file to the neighbor that forwarded it the request for caching. A push threshold of 100 means that the node only pushes out a new cache of a requested file after every 100-th request.

To illustrate the effects of the push threshold, we have simulated a scenario where two floods are handled at the same time by Backslash. The first flood starts alone and manages to saturate the system by causing a copy of the first flooded file to be placed at every node. After saturation, and while the first flood is ongoing, the second flood begins, gradually displacing cached copies of the first file for copies of the second file. Finally, the second flood terminates, allowing the first file to saturate the system again. Figure 1(a) shows how the diffusion evolves in this scenario for a push threshold of one. Note how agility is very high; the system responds very rapidly to changes in offered load.

The benefits of using lower push thresholds are illustrated in Figure 1(d). The figure graphs the cumulative load distribution over the system for the short second flood for three representative thresholds: 1, 22 and 43. On one hand, with a threshold of one, the second flood causes no higher a load than 300 requests to any node. On the other hand, with the highest threshold of 43 only 600 nodes participate in caching the second file at any one time and the maximum per node load reaches almost 800 requests.

However, higher agility makes the satisfaction of requests for the second file more expensive. For the same number of requests satisfied, higher thresholds result in much fewer cache pushes in the face of contention. Figure 1(b) graphs the total number of cache pushes as a function of the push threshold. Threshold one results in almost 25,000 cache pushes, of which 23,000 are mainly due to the oscillations caused by contention between the first and second files. The hysteresis introduced by the highest threshold of 43 mitigates this effect, as indicated by the almost tenfold reduction in cache pushes shown in the graph. Note, however, that the actual point where the benefit of lower threshold justifies the cost is specific to the underlying topology and resource restrictions of each Backslash participant, for whom a temporarily high load might be justified by overall lower traffic.

As described in Section 3.3, in local diffusion cached copies of a flooded document inside a cache "bubble" are only used by requests initiated locally, whereas caches on the perimeter of the bubble are used locally and also by requests initiated outside the bubble. This makes perimeter caches much hotter than internal bubble caches. We explore the use of probabilistic forwarding as a method to spread out the load of the perimeter incurred by requests initiated outside the bubble over all the nodes within the bubble.

We use a probability function that assigns a linearly decreasing forwarding probability to every bubble node on the path from an external request originator to the authority node. In this way, a cache at the perimeter of the bubble has a maximum forwarding probability (60% in our experiments). Subsequent next hop nodes toward the center of the bubble decrease their forwarding probability proportionally as they get closer to the center. We would expect to see a better load distribution among the nodes of a bubble as a result of this technique.

In Figure 2(b) we show the effects of using probabilistic forwarding during a single flood. We have calibrated the push threshold of the probabilistic run so as to achieve similar diffusion patterns between the two runs (see Figure 2(a)). Surprisingly, although we initially expected probabilistic forwarding to even out the distribution of load among nodes with caches of the flooded file, the graph shows only a very small improvement; specifically, there are slightly more lower-load nodes and slightly fewer higher-load nodes. We ascribe this surprising result

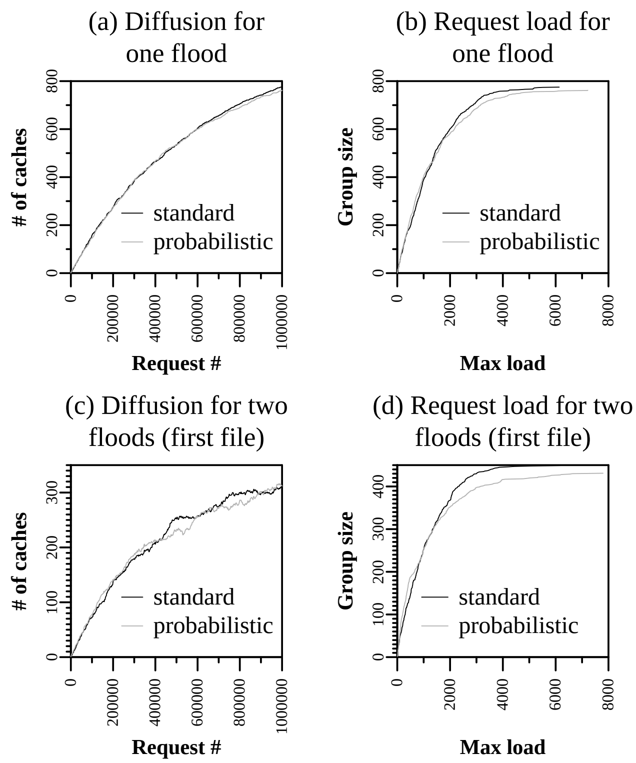

Fig. 2. The effects of probabilistic forwarding in one flood ((a) and (b)), and in two concurrent floods of equal intensity ((c) and (d)). (a) and (c) graph cache diffusion during the evolution of the two experiments, as achieved by calibrating the push threshold in the probabilistic experiments to 220; the standard experiments used a threshold of 500. (b) and (d) show the cumulative load distributions for the two flood scenarios with and without probabilistic forwarding.

to the monolithic fashion in which we measure load in our simulations. We conjecture that although the cumulative load per node in the duration of the experiment seems only a little affected by the use of probabilistic forwarding, it is the distribution of that load *over time* within a single node that improves, that is, becomes less bursty, in this case. Deterministic forwarding creates high bursts of load at the perimeter of the bubble since all nodes must service their requests. Upon reaching its threshold, the bubble expands outward and a new perimeter services incoming requests. Once a node is no longer at the perimeter of the bubble, it does not receive requests from outside and its load drops significantly. Probabilistic forwarding allows interior nodes to continue servicing requests even after they are no longer at the perimeter of the bubble.

The results are similar when two floods of equal intensity compete against each other. Figure 2(d) shows the difference in load distribution with and without probabilistic forwarding for one of the two simultaneous floods. While probabilistic forwarding evens out the load distribution in favor of lower loads, the effect is not quite as significant as we had anticipated. We hope to experiment with different forwarding probability functions to achieve a more pronounced benefit.

This is a very preliminary evaluation of cache diffusion in Backslash. We hope to perform a more thorough analysis in the near future.

5 Related Work

Our work shares many goals with the pioneering work done in the Adaptive Web Caching project [1]. Our local diffusion method is similar to the diffusion method used by AWC. However, AWC offers the benefits of a proxy cache, whereas Backslash replaces a reverse proxy cache.

A lot of work has been done on building self-maintainable overlay networks that follow the distributed hash table paradigm [3, 5, 6, 8]. We use results from that area extensively.

A set of HTTP extensions for the "content addressable web" were proposed recently [2]. Backslash would certainly benefit from the extended HTTP functionality offered by this work.

Other work has explored the use of client Web browser plug-ins to diffuse the effects of flash crowds [4]. However, Backslash differs by requiring a server-side implementation that is completely transparent to the client.

Finally, Rubenstein and Sahu [7] analyze theoretically a simple peer-to-peer protocol for flash crowd document retrieval based on random walks in an unstructured overlay, similar to Gnutella. We hope to compare the latency characteristics of the two designs in the near future.

6 Conclusions

There exists a need for a cost effective method to combat flash crowds. Backslash addresses this problem and, given preliminary results, is a promising method of mitigating flash crowd effects.

The next steps in this research involve a deeper exploration of different forwarding probability functions and their interactions with the other aspects of cache diffusion, the development of a hybrid local/directory diffusion method to exploit the benefits of both methods, closer cooperation with a cache invalidation scheme, and a higher-fidelity simulation and trial deployment plan.

7 Acknowledgments

This work started out as a summer internship project at the ICSI Center for Internet Research, under the guidance of Mark Handley, Scott Shenker and Sylvia Ratnasamy. We are grateful to them for the unfaltering motivation, guidance and feedback they provided during the formulation of this research.

We are also thankful for the generous funding we have received from the Stanford Networking Research Center, DARPA (contract N66001-00-C-8015) and Sonera Corporation.

References

1. The Adaptive Web Caching Project. `http://irl.cs.ucla.edu/AWC/`.
2. The Content-Addressable Web. `http://onionnetworks.com/caw/`.
3. Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of ASPLOS 2000* (Cambridge, MA, USA, Nov. 2000), pp. 190–201.
4. Padmanabhan, V., Sripanidkulchai, K. The Case for Cooperative Networking. In *1st International Workshop on Peer-to-Peer Systems (IPTPS 2002)* (Cambridge, MA, USA, March 2002).
5. Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. A Scalable Content-Addressable Network. In *Proceedings of SIGCOMM 2001* (San Diego, CA, U.S.A., Aug. 2001), ACM SIGCOMM, pp. 161–172.
6. Rowstron, A., and Druschel, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM Middleware 2001* (Heidelberg, Germany, Nov. 2001).
7. Rubenstein, R. and Sahu, S. An Analysis of a Simple P2P Protocol for Flash Crowd Document Retrieval. Available as *Columbia University Technical Report EE011109-1* (Columbia University, New York, NY, USA, Nov. 2001).
8. Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of SIGCOMM 2001* (San Diego, CA, U.S.A., Aug. 2001), ACM SIGCOMM, pp. 149–160.