2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops

Update Management in Decentralized Social Networks

Simon Forsyth, Khuzaima Daudjee
David R. Cheriton School of Computer Science
University of Waterloo
{swforsyt, kdaudjee}@uwaterloo.ca

***Abstract*—Decentralized social networks in the form of blogs and wikis often replicate dynamic content through caching. Maintaining replicated data in these systems is challenging due to the high cost of update management. We propose a technique that provides a structure to support efficient updates for cached data, and we demonstrate the efficacy of our approach through performance studies.**

I. Introduction

Blogs and wikis are forms of decentralized social networks that allow individuals to interact with each other through dyanamic content that is made available on a large scale. For example, Wikipedia allows users to change parts of articles while blogs and news articles provide sources for people to comment on. Effort has been expended to create unstructured peer-to-peer (P2P) versions of web pages [1], content delivery networks (CDN) [2], and social networks [3].

To reduce the cost of network accesses, content in these systems is replicated through caching [4]. In many types of decentralized online social networks that are unstructured P2P networks, this cached data changes over time [5]. An unstructured P2P system must therefore provide a mechanism for replacing old content with new versions to avoid stale data. Thus, update management is important in these systems. Proposals exist to maintain data synchronously or using quorums to limit the number of peers required [6] [7], but in addition to other factors, such schemes must know the number of replicas currently in the system, introducing its own problems. Therefore, updates must be applied in a lazy (asynchronous) manner to make cached data maintenance feasible and effective [8]. However, lazy maintenance can result in data that is updated at a slower rate, resulting in declining freshness. This problem is mitigated in structured systems as the number and location of cached copies is easily determined. Ready access to knowledge about the state of cached copies does not exist in an unstructured system, making techniques that require knowledge of the number or location of copies difficult, and therefore expensive, to obtain.

In this paper, we address the problem of maintaining cached data in unstructured P2P systems with on-demand replication that is often the hallmark of online social networks. Results of queries are cached in the process of being returned to the peer that issued the query. These cached replicas are maintained lazily, allowing the system to scale with respect to the number of peers, replicas and queries.

Specifically, in Section III-A, we address how updates can be managed through path replication [9] to improve search efficiency. Our proposal applies updates in the same order to all replicas in the system to maintain cached data. We show using both real and synthetic workloads traces (Section IV) that our techniques are efficient and effective in maintaining freshness for cached data.

II. System Model

The system models a participatory CDN, in which the providers and consumers of data cooperate to share the load. This system provides up-to-date content to those who request it. Therefore, updates must be supported and distributed efficiently. Efficient deletion is not required as cache turnover will eventually force deleted items out of a cache. Deletion of the original copies will eventually cause removal from all caches without any special effort.

We assume each data item has one peer, identified as its *master* peer, that controls all updates for that data. A peer in the network may be the master for any number of data items. We assume that providers are interested in maintaining their data; consequently we assume that churn among master peers will be minimal as they will choose not to leave to ensure their data remain available. Conversely, peers which are not masters may undergo greater churn, possibly behaving in a manner similar to that observed in existing P2P networks, with most peers remaining for very short periods of time [10]. Finally, we assume peers do not exhibit malicious behaviour. They may crash, but all messages sent are trustworthy.

The network modelled is an unstructured network, which allows peers to join and leave with little coordination as peers simply connect to those that are available rather than having to find a unique location in the structure and then assuming the responsibilities associated with that location. In an unstructured network, data becomes harder to find and thus maintain. The network topology is a regular random graph constructed using the SwapLinks algorithm by Vishnumurthy and Francis [11]. Our work makes no assumptions regarding topology so our algorithms will work successfully on any network topology.

Search and replication are related operations. The choice of where an object is replicated affects the search efficiency and some search strategies impose conditions on where caches are located. For search and replication, the system uses multiple

978-0-7695-5023-7/13 $25.00 © 2013 IEEE
DOI 10.1109/ICDCSW.2013.54

IEEE computer society

random walks and path replication respectively. In an unstructured system, these choices provide a low network overhead, allowing scaling while still providing reasonably fast search times [9].

Random walks provide a message efficient method to find items. A random walk is performed by choosing a single neighbouring peer at random to forward the query to and repeating until the desired item is found or the search is cancelled. Apart from the peer it just came from, a query may be forwarded to any neighbour. Performing multiple simultaneous random walks decreases the time required to find existing data while maintaining a limit on overhead due to network traffic.

We use *checking* to control the duration of random walks, in which peers receiving a query send messages back to the peer that initiated it. The originating peer may reply with a continue or cancel message to control the duration of the random walk. While this method adds additional traffic and time to find a result, it also allows the other random walks to terminate early once a result has been found. It also avoids issues associated with selecting the correct value for the TTL [9]. As an additional benefit, a user can cancel a search and have its overhead quickly removed from the network.

When a query finds a result using a random walk, that query was forwarded by one or more peers including the source of the query. Using path replication, the result is forwarded back through the chain of peers to the originating peer and each peer in the chain caches the data before forwarding it. These caches form a path on the network overlay graph.

In the next section, we describe our algorithms for update management.

III. Update Management

A. Cache Structure

Path replication (described in the previous section) does not track updates, allowing stale copies to remain in the system. To address this deficiency, the cached copies of data are augmented with additional information about that data on a local level. This information is exploited to push updates to peers with cached copies without maintaining global system state or using expensive alternatives such as flooding to send updates.

1) Data Caching: A successful random walk provides two pieces of information about the query: the data and the path used to find that data. While path replication uses the path to create replicas, it does not store any information about the path. We propose replicating information about the path in addition to the data satisfying the query. Each peer therefore stores the data, a version number assigned by the master, the estimated distance in hops to the master, the identity of the peer from which it received the data (its *parent*) and the identity of the peer to which it sent the data (its *child*). For example, after the random walk in Figure 1a, three cached copies and two parent/child links are created (Figure 1c). Peers may have any number of children, limited only by the number of neighbours. It is possible that more than one of the walks

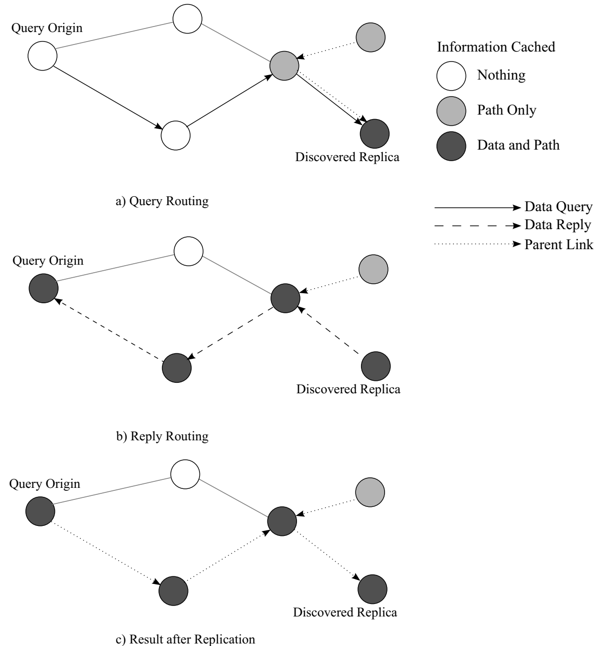

Fig. 1. Random walks and path replication. (a) A random walk visits peers until it finds a replica. (b) The peers replicate the data back along the path followed by the random walk. (c) Peers also record the path. References to parents are shown. There are also references from each parent to their children.

will succeed. In that case, path replication occurs along all successful routes.

It is not safe to abort a random walk or replication initiated by that walk until a result reaches the peer that originated the query. Thus a peer may participate in more than one path between the querying peer and the master peer as multiple walks may succeed and start replication before the querying peer receives a result. In such cases, both parents remember the peer, but it remembers only the parent with the shortest estimated distance to the master. To improve response time for a query, cycles are eliminated. In the absence of cache eviction, the edges connecting the parents to children form a directed acyclic graph rooted at the master copy of an item.

2) Path Caching: While the data cache allows updates to propagate by following child links as described in III-B, cache eviction will cause breaks in the graph. However, the metadata associated with each data item require less space to store than all but the smallest data items. Since this means a peer may store the metadata for many more data items than full replicas in the same amount of space, the system additionally includes a *path cache* containing only metadata. We refer to the cache containing the data items as the *data cache* to differentiate the caches. When data is purged from the data cache, the metadata is copied to the path cache. This allows the graph structure to be maintained for a greater period of time, allowing updates to propagate.

In addition, the path cache improves the performance of random walks. When a random walk visits a peer with relevant information in its path cache, the walk next visits the parent

peer stored in the cache instead of a random peer (as in the last message in Figure 1a). Such hints help a walk reach the relevant master but do not guarantee success since a peer identified in the path cache may have removed the path information from its cache, left the network, or crashed. In such cases, the walk reverts to random selection to continue the search.

B. Updates

As stated earlier, each data item is associated with a master peer that manages updates for that item. It determines the order of updates and records a strictly increasing version number on successive updates. While other peers may initiate an update, it is always applied at the master copy and the updated version propagated to other peers.

New versions are pushed lazily along the edges of the directed graph produced by following the child links in the caches (the *child graph*). Peers that contain a copy of the data in their data cache update their copy and forward the update message to their children, while peers that have a copy in the path cache forward but do not store the data item. As the graph may contain more than one path to the same peer, peers check the version of the update. If the version in the update message is the same or older than the present version, the update message is discarded. Therefore, these version numbers ensure that all updates are applied in the same order as at the master, ensuring a globally consistent order for updates. Since it is possible that an update may be missed by a peer, each update must contain a complete copy of the data to ensure that any missed updates do not affect the final result.

The master acts as a source vertex for the complete child graph associated with a data item. Therefore, unless the graph has been cut due to a cache eviction from a non-sinkpeer or a peer has failed, an update is guaranteed to reach every copy of the item. Each peer contacts only the subset of its neighbours known to have had a copy of the data previously. This ensures that peers that have never been part of a search for a data item will not be contacted for an update.

IV. Performance Evaluation

A. Experimental Setting

Experiments were performed on the PlanetSim simulator to evaluate the capability of the algorithms. These experiments used synthetic workloads and two trace-based workloads, one from Wikipedia [12] and one from Metafilter [13]. The synthetic workload modelled uniform and Zipfian distributions of queries and explored the effect of changing parameters such as the update frequency. The Wikipedia trace represents a workload with a very large number of data items, while the Metafilter trace describes a flash-crowd-like workload where specific data items are popular at particular times.

1) Configuration: The following baseline is used for all experiments. The network graph contains 10 000 peers, each connected to 32 other peers. 20% of the peers are designated as master peers and all data items are placed on them using a uniform distribution. The data cache size is 25 items and the path cache size is 125 items for all peers. While the cache sizes are pessimistic, machines do not gain capacity as the number of data items in the network grows, so any cache size will eventually represent a very small proportion of the total amount of data. We therefore choose a small cache size relative to the number of data items to model the network when there is a significant amount of data relative to the capacity of each peer's ability cache. In addition, our experiments demonstrate that a small path cache has enough room to maintain path information for the most frequently accessed data on every peer.

To issue queries, a random peer in the network is designated uniformly at random to issue a request for a data item determined by the workload. The peer issues 16 random walks (per the suggested range in [9]) and reports success when the first of these walks returns with a result. No walk is terminated until one returns a result; especially, no TTL value has been set as a second cutoff method. While this is impractical in a real system as overhead would gradually increase as searches for non-existent items accumulate, it permits an examination of the worst case behaviour of a search.

Updates are performed at the master copy of the data item. The master pushes updated versions to children who identified themselves through queries as described in the model.

2) Measurement: We measure the time to find a result using the number of times a message is forwarded before reaching its destination (*hops*). This is a scale invariant feature of the network and so remains relevant regardless of the network.

The number of queries that have not completed is monitored and measurement begins once the number stabilizes. At this point the number of queries completed each cycle approximately equals the number of queries issued each cycle. In addition, both the data and path caches are full and are distributed according to the query load.

The first random walk to return a result is used to measure freshness, even if another random walk associated with the search returned a more recent version.

All experiments were run five times and results averaged over all runs. There was little variation in the results: the 95% confidence interval for all freshness results is less than 0.2%.

B. Workloads

1) Synthetic: The number of data items is 10 000. Data items for queries follow two distributions: a Zipf distribution with an exponent equal to 1 and a uniform (normal) distribution. Within the Zipf distribution, the most popular item represents about one percent of the total number of queries. For brevity, we refer to the frequency of data access according to each distribution as *popularity*. Data to be updated is chosen uniformly at random.

Queries are issued at a rate of 100 queries per cycle (1% of the total number of peers per cycle) while updates are applied at a ratio equal to 20% of the number of queries issued. For many applications, such a ratio is pessimistic, as far more people consume updates than produce them. It is therefore

likely that most real scenarios will, like those from our traces, exhibit better results.

2) Wikipedia: The first source of real data for traces is Wikipedia. Traces from Wikipedia access logs between 18 Sep 2007 20:10:48 GMT and 21 Sep 2007 13:11:44 GMT were used as a source for the workload [12]. From the raw logs we extracted all reads and updates to English articles. Searches, special pages that are not part of the content, and images were discarded. Because the update rate is low (about 1 update per 1000 requests), the workload was modified to increase the effective update rate without biasing the data by collecting updates from a greater period of time and compressing them so they they remain identically distributed but issued at a higher rate. The reads were issued at a constant rate of 100 per time step, while the mean number of updates is 5.5 updates per time step. The number of articles (data items) referenced by the million reads in the trace is 409 744. The most popular item in the trace represents 0.2% of the total number of queries. The trace was run with data cache sizes of 1000 items to match the proportion in the synthetic workload.

3) Metafilter: We also ran experiments based on a workload trace from Metafilter [13]. Metafilter is a community blog, where any user may add a story with links and the other users discuss it. The available data spans the entire history of the site so we used data for 2012. User comments represent updates to the data, but a trace of the reads was not available.

Based on the read queries for cached data in CoralCache for Slashdot, a similar site [14], a similar flash crowd with a quick onset and a trail of requests after was modelled. To achieve this model, a time step in the simulator represents one second. The clock was started at the time of the first update in 2012, and updates were applied at the time they occurred. As the updates already model a flash crowd and provide a reason for a user to make another request, a random number of reads were issued after each update. The number of reads was normally distributed with a mean of 75 and a standard deviation of 25. These reads were applied using a Poisson distribution for the intervals. As a result, the number of reads per item is proportional to the number of updates, with a normal distribution and distributed in time in the same form as the updates.

C. Results

1) Query Response Time: The median number of hops to find the first replica is 87 using a uniform distribution and 5 using a Zipf distribution for the popularity of the data items. This shows that caching is more effective when some data items are more popular than others. When the experiments were run without caching, there were far more incomplete than complete queries as the system ran out of memory due to excess incomplete searches.

D. Freshness

Both the Zipf distribution and the uniform distribution of data access frequency return similar results for freshness at a 20% update ratio. Under the Zipf distribution, 85.5% of queries returned fresh results while the uniform distribution returned fresh data for 88.3% of queries (Figure 2). Because a uniform distribution is less realistic and because it matches the behaviour of the least popular data in the Zipf distribution, we do not discuss it further. Fresh data is returned more often as the update ratio decreases, with 97.7% of queries returning fresh data at a 5% update ratio for the Zipf distribution. Query results that are more than one version older than the current fresh version are rare; 97.8% of the results are no more than one version older than a fresh copy.

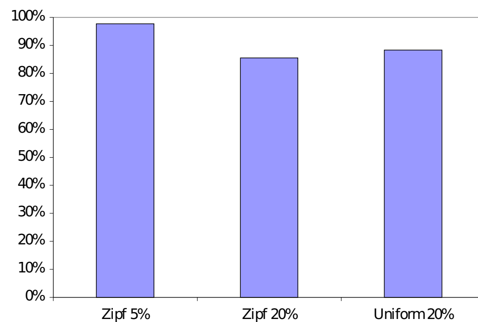

Fig. 2. Number of queries returning fresh data

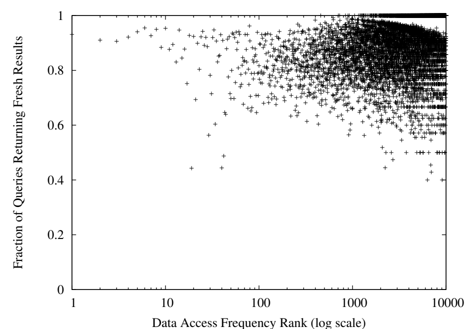

Fig. 3. Fraction of fresh results at 20% updates (Zipf)

Figure 3 shows the fraction of fresh results as a function of the data items ranked by access frequency. We use a logarithmic scale on the horizontal axis, causing results to be plotted according to the frequency at which they occur. Thus, the fraction of total queries for a particular data access frequency rank is equivalent to the horizontal space occupied. Freshness depends on the data item access frequency, with three bands of effectiveness. The first band, which consists of roughly the 10 most popular data items and one quarter the total number of queries, has excellent results with 92.7% of queries returning fresh data. The second band consists the next 90 most popular items and one quarter the total number of queries with 82.8% of these queries returning fresh data. The third band, containing all remaining items and one half the total queries, represents the least popular data items and shows much variation in the results for individual items. Overall, a very high proportion (89.9%) of queries in this band returned fresh data.

For Wikipedia, the number of fresh results returned is very high, matching the 5% update scenario with 97.8% of requests returning fresh data. The low popularity of even popular items had a significant effect on the time to find results. The median number of hops was 242.

For Metafilter, 91.7% of the queries returned fresh results. As the Metafilter trace simulates a flash crowd, caching was very effective, with the median number of hops equal to zero, indicating that over 50% of queries are satisfied by a locally cached copy. This shows that the caches adapt quickly to changes in the popularity of data. As nearly all items become the most popular item for a short period of time, we do not include a graph showing results by popularity.

E. Cache Eviction

To ensure that updated versions are successfully propagated to all copies in the data cache, the child graph must remain connected. To this end, we explore several eviction policies for the caches. These are *random*, *first-in-first-out* (FIFO), *least-frequently-used* (LFU), *least-recently-used* (LRU), *sink-first*, and *root-first*. Note that the LFU, LRU and sink-first policies break the square root replication requirement but they are included to test the possibility that despite the increase in search overhead they may allow more searches to return fresh results.

The *sink-first* policy is motivated by the idea that copies at the end of a path are potentially the least valuable. Update messages reach them last, and there are the greatest number of hops between them and the master, increasing the chance for the path to become disconnected. Moreover, sinks represent peers that can be removed without disconnecting the graph, suggesting that removing them should help keep the graph connected. For this policy, the number of child peers associated with a data item are considered when choosing an item to remove. If there are no children associated with the data item at a peer, then removing the path information associated with that data item cannot break any paths to a copy in a data cache and therefore one of these items is removed at random. It is possible that there are no sinks in the path cache. In this case, we revert to the least-frequently-used policy to select an item to evict on the grounds that it is more important to maintain connections and ensure freshness for frequently accessed and/or updated data.

The *root-first* policy takes a different approach to the problem of disconnected graphs. Instead of trying to keep the graph from disconnecting, it instead tries to destroy the disconnected subgraph more quickly when a disconnection occurs. That is, when a peer discovers that it represents the root for a subgraph and is not the master, it assumes that it is disconnected and therefore should remove itself as it will become stale. This assumption is not always correct since children can become parents if they discover that updates are coming from an alternate route but it is always true if the subgraph is not connected.

Experiments with different cache eviction policies were performed with larger caches. The data cache was set to 50 items and the path cache was enlarged to 250 to magnify the differences between cache algorithms.

TABLE I
FRESHNESS FOR CACHE EVICTION ALGORITHMS

		Data Cache Policy		
		Random	Root First	FIFO
Path Cache Policy	LFU	82.54%	90.00%	90.06%
	Random	76.07%	82.12%	84.63%
	Leaf First	70.45%	78.27%	82.69%
	FIFO	80.58%	87.24%	88.88%
	LRU	80.33%	87.52%	89.43%

The *sink-first* policy, which preferentially removes sinks from the graph in the path cache is clearly sub-optimal. It performs more poorly than any other method, even though its fallback algorithm, LFU, had the best performance for the same cache. The scheme successfully kept graphs locally connected, however, when a break did occur, due to churn or a lack of sinks in a particular peer's cache, the newly disconnected graph remained disconnected, creating islands of stale data.

While the random policy allows for square root replication, it appears to be an inferior policy for maintaining freshness of that same data. FIFO offers the same quality of replication as a random policy while greatly increasing freshness. While an LFU policy offers further increase, its benefits appear modest and are offset by an increase in read costs for less popular data. The difference in freshness between the LRU and FIFO policies is insignificant, though this may be an indication that the cache sizes remain too small to make a distinction between the policies even after the increase in size for this experiment and so not indicate that they are equivalent in performance.

Preferentially removing the root when it has no connection slightly decreases freshness when applied to the data cache over plain FIFO. This means that removing data that was added earlier is more valuable than removing data that cannot be updated. It seems likely that such a policy may perform better for data that is very frequently updated, especially if other data in the system is updated less frequently.

V. RELATED WORK

Lv et al. demonstrate the case for using path replication in unstructured networks [9]. It does not address updates.

CoralCDN is a P2P decentralized CDN intended to help websites deal with flash crowds that has been available since 2004 [2]. It uses expiry-based pulls to discover updated data.

Freenet uses path replication along paths that are chosen based on a greedy search within a key space [1]. The original version of Freenet made no allowance for updates. Later versions of Freenet along with other approaches place updated data without removing or updating stale copies [15] [16] [17]. In the case of Freenet, the version is part of the document name, increasing the difficulty of finding the document by that name. It also does not guarantee that the most recent version is found.

Datta et al. use a probabilistic flooding algorithm to send update messages to neighbours that contain replicas [18].

Instead of relying on randomness to limit message passing, we take advantage of the path information provided by the replica placement policy to achieve the same results deterministically. Other push-pull algorithms for updates have been labelled as gossip or epidemic protocols in other work [19]. Generally, gossip protocols are concerned with completeness of updates, not freshness.

Another method for applying updates relies on quorums [6] [7]. This allows for eager replication and perfect consistency while reducing the overhead for applying updates. While quorums are easy to track in a structured P2P network where data items may be identified by keys, they are much more difficult in an unstructured network — Henry et al. require that the entire network be contacted to generate an initial list of peers that could participate and Vecchio and Son provide no algorithm to find the set of interested peers.

Data may also be assigned expiry times (time-to-live) after which a replica is considered invalid [20] [21]. This approach creates a pull-based system in which replicas keep track of the origin. Typically, the replicas are arranged into a hierarchy to avoid bottlenecks at the master.

VI. Conclusion

In this paper, we addressed the problem of managing updates in decentralized online social networks. Our techniques for caching and exploiting path information effectively control update propagation to provide fresh data to most queries. This contribution opens up a spectrum of design choices that can support data freshness versus performance trade-offs in decentralized social networks.

Acknowledgment

Funding for this project was provided by the Natural Sciences and Engineering Research Council of Canada.

References

[1] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: a distributed anonymous information storage and retrieval system," in *International workshop on Designing privacy enhancing technologies: design issues in anonymity and unobservability*. New York, NY, USA: Springer-Verlag New York, Inc., 2001, pp. 46–66.

[2] M. J. Freedman, E. Freudenthal, and D. Mazières, "Democratizing content publication with coral," in *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*. Berkeley, CA, USA: USENIX Association, 2004, pp. 18–18.

[3] S. Buchegger and A. Datta, "A case for P2P infrastructure for social networks - opportunities and challenges," in *Proceedings of WONS 2009, The Sixth International Conference on Wireless On-demand Network Systems and Services*, Snowbird, Utah, USA, February 2-4, 2009.

[4] Oversi, "Overcache p2p caching and delivery platform," 2010.

[5] R. Narendula, T. Papaioannou, and K. Aberer, "Towards the realization of decentralized online social networks: An empirical study," in *ICDCS Workshops*, 2012, pp. 155–162.

[6] D. Del Vecchio and S. Son, "Flexible update management in peer-to-peer database systems," in *Database Engineering and Application Symposium, 2005. IDEAS 2005. 9th International*, july 2005, pp. 435 – 444.

[7] K. Henry, C. Swanson, Q. Xie, and K. Daudjee, "Efficient hierarchical quorums in unstructured peer-to-peer networks," in *OTM Conferences (1)*, 2009, pp. 183–200.

[8] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz, "Update propagation protocols for replicated databases," in *SIGMOD Conference*, 1999, pp. 97–108.

[9] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," in *Proceedings of the 16th international conference on Supercomputing*, ser. ICS '02. New York, NY, USA: ACM, 2002, pp. 84–95.

[10] D. Stutzbach and R. Rejaie, "Understanding churn in peer-to-peer networks," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, ser. IMC '06. New York, NY, USA: ACM, 2006, pp. 189–202.

[11] V. Vishnumurthy and P. Francis, "On heterogeneous overlay construction and random node selection in unstructured p2p networks," in *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, april 2006, pp. 1 –12.

[12] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, July 2009.

[13] Metafilter, "Metafilter infodump," 2012, stuff.metafilter.com/infodump/.

[14] M. J. Freedman, "Experiences with coralcdn: a five-year operational view," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 7–7.

[15] I. Clarke, O. Sandberg, M. Toseland, and V. Verendel, "Private communication through a network of trusted connections: The dark freenet," *Network*, 2010.

[16] W. W. Terpstra, J. Kangasharju, C. Leng, and A. P. Buchmann, "Bubblestorm: resilient, probabilistic, and exhaustive peer-to-peer search," in *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '07. New York, NY, USA: ACM, 2007, pp. 49–60.

[17] C. Leng, W. W. Terpstra, B. Kemme, W. Stannat, and A. P. Buchmann, "Maintaining replicas in unstructured p2p systems," in *Proceedings of the 2008 ACM CoNEXT Conference*, ser. CoNEXT '08. New York, NY, USA: ACM, 2008, pp. 19:1–19:12.

[18] A. Datta, M. Hauswirth, and K. Aberer, "Updates in highly unreliable, replicated peer-to-peer systems," in *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ser. ICDCS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 76–.

[19] M. K. Mujtaba, "Push-pull gossiping for information sharing in peer-to-peer communities," in *In Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), pages 1393 1399, Las Vegas*. CSREA Press, 2003, pp. 1393–1399.

[20] X. Tang, J. Xu, and W.-C. Lee, "Analysis of ttl-based consistency in unstructured peer-to-peer networks," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 19, no. 12, pp. 1683 –1694, dec. 2008.

[21] X. Tang, H. Chi, and S. Chanson, "Optimal replica placement under ttl-based consistency," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, no. 3, pp. 351 –363, march 2007.