Adaptive Replication and Replacement in P2P Caching

Jussi Kangasharju Keith W. Ross

***Abstract*— Caching large audio and video files in a community of peers is a compelling application for P2P. Assuming an underlying DHT substrate for the caching community, we propose adaptive object replication and replacement strategies for P2P caches. One such strategy, Top-K MFR, is shown to provide near optimal performance.**

I. Introduction

Web caches are often deployed by institutions (corporations, universities, and ISPs) to reduce perceived user response time and to reduce traffic on access links between the institution and its upstream ISP. Web caching has received significant attention in both industry and research, with numerous companies including Microsoft and Cisco selling Web cache products.

In this paper we explore the design of a similar caching service, but with the objects being cached in intermittently-connected peers rather than in always-on servers. The P2P caching system could be used for caching Web objects, for caching large music and video files emanating from P2P file-swapping services [1], or for caching files emanating from global persistent P2P file storage systems [2] [3] [4]. In a P2P caching system, when a peer in the community wants to obtain an object, it first searches the peers in its community for the object; if the object is not found in the community, then the community retrieves the object from the "outside" (for example, from the Web, from a P2P file-swapping service, or from a persistent P2P global file storage), possibly caches the object in peers in the community, and forwards the object to the requesting peer.

As is the typically the case for Web-caching environments, we assume that the intra-community file transfers occur at relatively fast rates, whereas file transfers into the community occur at relatively slow rates. As an example, the community may be a university or corporate campus, with tens of thousands of peers in the campus community interconnected by high-speed LANs, but with connections to the outside world occurring over congested campus access links. The advantages of a serverless, P2P cache design over the traditional server-centric design include increased storage capacity (important for large music and video files), increased file transfer capacity, improved resilience to faults and attacks, and server cost reduction.

As in a traditional Web caching system, the principal measures of performance are the *hit rate* and the *byte hit rate*, as they typically correlate directly to user perceived response time and to access link traffic. Assuming that intra-community file transfers occur at faster rates than transfers from outside the community, a higher hit rate directly correlates to lower average file transfer times and to reduced stress on access links. There are two big-picture issues in maximizing the hit rate in a P2P caching system.

- **Replication:** Because peers connect and disconnect to the network (or to the "application"), to provide satisfactory hit rates, the popular content needs to be replicated across multiple peers in the community. At the same time, content should not be excessively replicated, wasting bandwidth and storage resources.[1]
- **Object Replacement Policies:** In a P2P caching system, each participating peer has a limited about of storage that it can offer to the caching community. When this storage fills at some peer, the peer needs to determine which objects it should keep and which it should evict.

The principal contribution of this paper is a set of distributed algorithms for dynamically managing cached content in a P2P community. These algorithms replicate and replace content in a near-optimal manner. Importantly, the algorithms make no *a priori* assumptions about object request probabilities nor about the up-down dynamics of the peers. The algorithms are adaptive and fully distributed.

The primary assumption behind our algorithms is that

J. Kangasharju is with the Dept. of Computer Science at TU Darmstadt, Darmstadt, Germany, email: `jussi@tk.informatik.tu-darmstadt.de`

K. W. Ross is with Institut Eurecom, Sophia Antipolis, France, email: `ross@eurecom.fr`

[1] It is problematic when thousands of students on the same campus download and store the same recently-released movie [12].

peers in the community are tied together with an underlying P2P look-up service, such as CAN [8], Chord [9], Pastry [6], or Tapestry [13]. Thus, any participating peer in the P2P cache can give the look-up API the identifier for the object, and the look-up service returns the IP address of the up peers *in the community* that are the *winners* for the object.

We first propose the *Top-K Least Recently Used* algorithm, which provides significantly better performance than non-coordinated schemes. We then propose an alternative algorithm called *Top-K Most Frequently Requested* (Top-K MFR), which through simulation is shown to give nearly optimal performance.

II. Related Work

Squirrel [5] is a recent proposal and implementation of a distributed, serverless, P2P Web caching system. Squirrel, which is built on top of the Pastry [6] look-up substrate, has been carefully designed to serve as a replacement for a traditional Web cache. While such detailed protocol design and implementation issues are clearly important, the work [5] has not focused on the critical issues of replication and object replacement policies for a P2P cache.

FarSite [15] (see also [16] [17]) is a P2P file system with the strong persistence and availability of a traditional file system. The FarSite filesystem uses the same number of replicas for each object. In contrast with a file system, the goal of a P2P cache is not to provide strong file persistence and availability, but instead maximal content availability. Thus, in a P2P cache, the number of replicas of an object depends on the popularity of the object.

Lv et al [10] and Cohen and Shenker [11] studied optimal replication in an unstructured peer-to-peer network in order to reduce random search times. Our work differs in that our goal is to replicate content to maximize hit probabilities in P2P caches, taking intermittent connectivity explicitly into account. There has also been recent work comparing replication and erasure coding in persistent P2P storage infrastructures [7].

III. Adaptive Management of a P2P Cache

A P2P cache consists of a community of peers that collectively provide a distributed content cache. The peers providing the caching service could be workstations, desktop PCs, and portables.[2] Each participating peer allocates a fraction of its storage to the P2P cache. We suppose that the content in a peer's shared storage is not lost when a peer disconnects; when a peer comes back up, all the content in its shared storage is again available. (This is generally the case in P2P file-swapping systems such as KaZaA and Gnutella.)

[2]However, PDAs, and other low-bandwidth, low-storage devices would not likely be included in the cache community (although they may be permitted to use the caching service).

A. Location substrate

We suppose that each peer has a persistent name, which is assigned when the peer initially subscribes to the application. (Because peers typically change their IP addresses each time they come up, the IP address cannot be used as the persistent name.)

Our algorithms assume the existence of a substrate with the following functionality. The substrate has a function call that takes as input an object name j and creates internally an ordered list of all the up peers. The substrate then returns for a desired value of K the first K peers on the list, $i_1, i_2, \ldots, i_K$. The peer i_1 is said to be the current first place winner for j; the peer i_2 is said to be the current second-place winner for j, etc.

We assume that each peer has access to an API for this substrate. Thus an application running on a peer can give the API an object name j and get from the API the current ordered list of peers $i_1, i_2, \ldots, i_K$. There are number of substrates today that provide this functionality for first-place winners, including CAN [8] Chord [9], Pastry [6] and Tapestry [13]. These substrates are easily extended to provide the top K winners.

B. Top-K LRU Algorithm

Our adaptive replication algorithms replicate objects on-the-fly, at times of object requests. Moreover, each of the algorithms layer on top of the object location substrate. Consequently, our adaptive algorithms not only replicate content in a coordinated fashion, but also provide each peer with a means of locating a copy of a desired object.

We begin with a simple, intuitive algorithm for replicating content on the fly. Suppose X is a peer that wants object j. X will get access to j as follows:

Top-K LRU Algorithm

1) X uses the substrate to determine i_1, the current first-place winner for j.
2) X asks i_1 for j.
   - If i_1 doesn't have j, i_1 determines $i_2, \ldots, i_K$ and pings each of the $K-1$ peers to see if any of them have j.
   - If any of $i_2, i_3, \ldots, i_K$ have j, i_1 retrieves j from one of them and puts a copy in its shared storage. If none of them have j (a "miss" event), i_1 retrieves j from outside the community and puts a copy in its shared storage.

- If i_1 needs to evict an object to make room for j in its shared storage, i_1 uses the LRU (least recently used) replacement policy.

3) i_1 makes j available to X (either for streaming or for downloading into X's private storage). Note that X does not put j in its shared storage unless $X = i_1$.

We see that the Top-K LRU Algorithm possesses many desirable properties. It replicates content on-the-fly without any *a priori* knowledge of object request patterns or nodal up probabilities. It is fully distributed. It is possible that there will be a miss even when the desired object is in some up peer in the community; however, we shall show that if K is appropriately chosen, the probability of such a miss is low.

To study the hit probability performance, we have performed simulation experiments with 100 peers and 10,000 objects. All object sizes are of the same size b. (We also did extensive experiments with heterogeneous object sizes and obtained similar results.) For each experiment, each peer contributes the same amount of shared storage to the community. (We also did extensive experiments with heterogeneous storage, and obtained similar results.) Our experiments run from 5 objects per peer to 30 objects per peer. We suppose that the request probabilities for the various objects follow a Zipf-like distribution; our experiments use parameter 1.2 [14].

For the case of homogeneous up probabilities, we can derive a tight upper bound (over the set of all replication/replacement policies). We do not discuss this bounding technique here, due to lack of space. Because such a bound is available, the experimental results we report all use homogeneous up probabilities for the peers. We have considered two up probabilities: .2 and .9. We have also performed extensive testing with heterogeneous up probabilities, and have found that our algorithms have similar performance behavior.

Figure 1 shows two graphs, one for each of the up probabilities. Each graph plots hit probabilities as a function of peer storage. The top curve in each of these figures is the upper bound (optimal). Each figure has a curve for $K = 1$ and $K = 5$. The bottom curve is the hit probability for the case when each peer independently retrieves and stores content (in its shared storage) without regard to the other peers in the community. The figure also includes curves for the MFR algorithm, which will be discussed shortly. We observe from the figure that the adaptive algorithm with $K = 1$ performs significantly better than the non-coordinated algorithm, but significantly worse than the theoretical optimal.

Examining how objects are replicated provides important insight. Figure 2 shows, as a function of object popularity, the number of replicas per object for the theoretical optimal and for the adaptive LRU algorithm with $K = 1$. For the adaptive algorithm, the number of replicas per object is changing over time; the graphs therefore report the average values. The difference in how the theoretical optimal and the adaptive algorithm replicate objects is striking. The optimal algorithm replicates the more popular objects much more aggressively than does the adaptive algorithm. Furthermore, it doesn't store the less popular objects, whereas the adaptive algorithm provides temporary caching to the less popular objects.

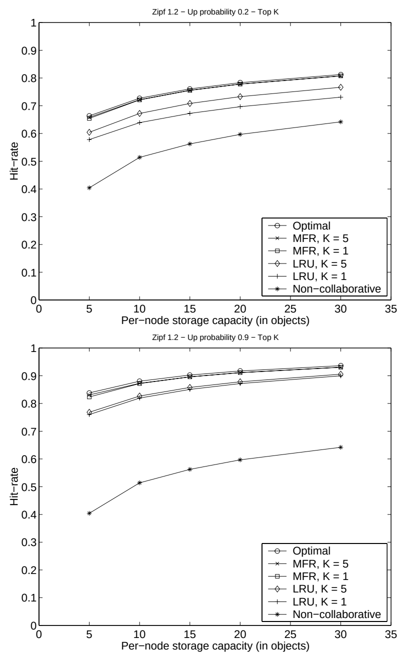

Fig. 1. Hit-rate as function of node storage capacity, top is node up probability .2 and bottom is .9

C. Top-K Most Frequently Requested Algorithm

The top-K LRU algorithm is simple and intuitive, and has reasonably good performance in terms of hit probabilities. But can we do better? To this end, we make two observations:

- LRU lets unpopular objects linger in peers. When an unpopular object is requested, it gets stored in one of the peers and remains there until it is evicted with LRU. Intuitively, if we do not store the less popular objects, the popular objects will grab the vacated space and there will be more replicas of the popular objects.
- Searching more than one peer (that is, the top-K procedure) is needed to find objects in the aggregate storage.

Based on these observations, we will now create a new adaptive algorithm, which will be shown to have near op-

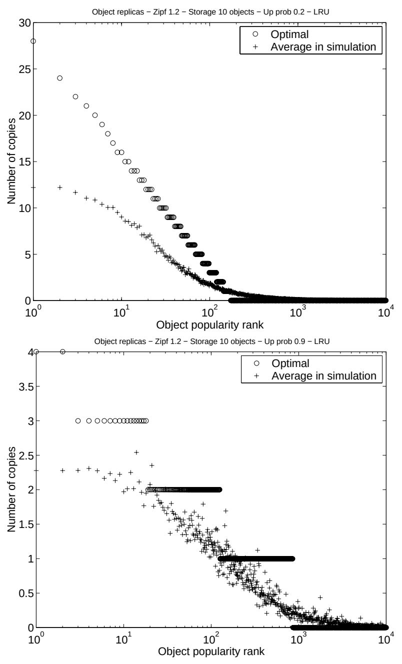

Fig. 2. Number of replicas per object with 10 objects of per-node storage capacity and LRU replacement policy, top .2 up probability, bottom .9

timal performance. To this end, we introduce the Most Frequently Requested (MFR) retrieval and replacement policy:

MFR retrieval and replacement policy

- Each peer i maintains a table for all objects for which it has received a request. For an object j in the table, the peer maintains an estimate of $\lambda_j(i)$, the request rate for the object. In the simplest form, $\lambda_j(i)$ is the number of requests peer i has seen for object j divided by the amount of time peer i has been up.
- Each peer i stores the objects with the highest $\lambda_j(i)$ values, packing in as many objects as possible.

Thus when peer i receives a request (from any other peer) for object j, it updates $\lambda_j(i)$. It then checks to see if it currently has j in its storage. If i doesn't have j and MFR says it should, then i retrieves j from the outside, puts j in its storage, and possibly evicts one or more objects from its storage according to MFR.

Now that we have defined the retrieval and replacement policy, we need to define the request dynamics. We want the request dynamics to influence the rates so that the replicas across all peers become nearly optimal. One approach might be for X (the peer that wants the object) to ping the top-K winners in parallel, and then retrieve the object from any peer that has the object. Each of the pings could be considered a request, and the peers could update their request rates and manage their storage with MFR accordingly. But it turns out that this approach doesn't give better performance than Top-K LRU.

It turns out that the correct approach is for X to *sequentially* request j from the top-K winners, and stop the sequential requests once j is found. Sequential requests influence the locally-calculated request rates in a manner such that the global replication is nearly optimal. In particular the value of $\lambda_j(i)$ at any peer i will be reduced by hits at "upstream" higher-placed peers for j. We now summarize the algorithm. Suppose X wants object j. Initialize $k = 1$.

Top-K MFR Algorithm

```
While $k \leq K$ and $X$ has not obtained $j$:
    1) $X$ uses substrate to determine $i$, the $k$th place winner for $j$.
    2) $X$ requests $j$ from $i$.
        • Peer $i$ updates $\lambda_j(i)$.
        • If peer $i$ already has $j$, peer $i$ sends $j$ to $X$; stop.
        • If peer $i$ does not have $j$ but it should (according to MFR), $i$ gets $j$, stores $j$ and evicts objects $o$ with low $\lambda_o(i)$ values if necessary. Peer $i$ sends $j$ to $X$.
    3) $k = k + 1$
```

If after K iterations, X still does not have j, X gets j from the outside directly (but does not put j in its shared storage).[3]

Figure 3 shows, as a function of object popularity, the number of replicas per object for the theoretical optimal and for Top-K MFR Algorithm with $K = 5$. We see that, in contrast with LRU, the number of replicas given by the MFR algorithm is very close to the optimal. In fact for most objects, the number of replicas given by the Top-5 MFR algorithm is equal to the optimal; a small fraction of objects are off by one replica (or less on average) from the optimal. Figure 1 compares the hit rate of MFR (with $K = 1$ and $K = 5$) with the adaptive LRU algorithms and with the optimal hit rate. We see from Figure 1 that the MFR algorithms give hit rates that are very close to optimal over the entire parameter space considered.

We have also developed a performance evaluation technique for MFR, and have used it to show that Top-I MFR generally provides near optimal results (where I is total number of nodes). The small and insignificant differences between MFR and optimal replication/replacement (when they occur) are due to imperfect load-balancing in the location substrate and to packing non-constant-size objects

[3]There is a subtlety in how i gets j at the end of Step 2. The peer i could simply retrieve i from the outside. The peer i could also ping the remaining internal winners for the object, which improves marginally the hit probability. (But in this latter approach, it is important not to count the pings as requests.)

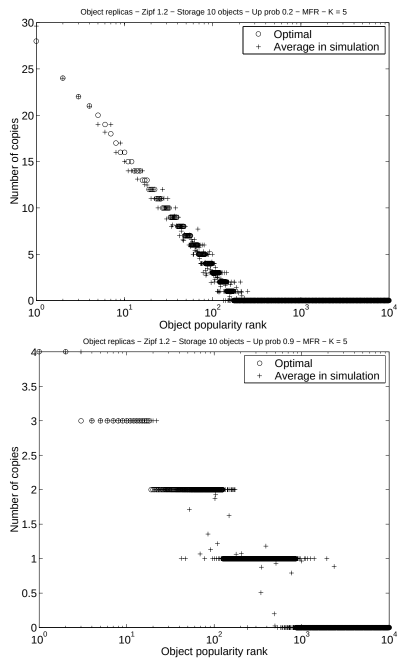

Fig. 3. Number of replicas per object with 10 objects of per-node storage capacity and MFR replacement policy with $K = 5$, top .2, bottom .9 up prob.

into the peers' storage. Because of lack of space, we not present the performance modeling technique.

IV. Dealing with Hot Spots

Up until this point our focus has been on replicating content to maximize the probability of having a hit in the community. However, consider the case when a very popular object j has a first-place winner i that is almost always up, that is, $p_i \approx 1$. In this case, our algorithms will only create one copy of object j, which will be permanently stored on peer i. If the demand for this object is very high, then peer i will become overloaded with file transfers. In this section we outline some solutions to this hot-spot problem.

One simple solution to this problem is to segment all objects (or just popular objects) into multiple pieces, and give each piece a unique name. Each piece is then treated as a separate object in the Top-K MFR algorithm, and the file-transfer load imposed by this popular object is spread over many peers. One drawback to this above approach is that, with multiple pieces per object, a hit requires having a hit for each of the individual pieces. A further refinement of the approach is to use redundant pieces, that is, to create $m + n$ pieces for the popular objects in a manner such that the original object can be reconstructed from any m of the $m + n$ pieces. We are currently studying this approach in more detail.

V. Conclusion

Institutional P2P caching of large multimedia files is a compelling P2P application. A fundamental characteristic of a large-scale P2P caching system is that the participating peers are intermittently connected. In contrast with traditional server-centric Web caching, LRU performs poorly in in P2P caching. In this paper we have introduced a distributed, adaptive algorithm, Top-K MFR, which gives essentially optimal performance for P2P caching. We have briefly outlined how hot spots could be handled in a P2P cache. We are currently exploring this research direction in more depth.

References

[1] KaZaA, http://www.kazaa.com

[2] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, Wide-area cooperative storage with CFS, *ACM SOSP 2001*, Banff, October 2001

[3] P. Druschel and A. Rowstron, ”PAST: A large-scale, persistent peer-to-peer storage utility”, *HotOS VIII*, , May 2001

[4] J. Kubiatowicz et al. “OceanStore: An Architecture for Global-Scale Persistent Storage,” *ASPLOS 2000*, Nov. 2000.

[5] S. Iyer, A. Rowstron and P. Druschel, ”SQUIRREL: A Decentralized, Peer-to-Peer Web Cache”, *PODC 2002*.

[6] A. Rowstron and P. Druschel, ”Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems”. *IFIP/ACM International Conference on Distributed Systems Platforms*, Heidelberg, Germany, Nov. 2001

[7] H. Weatherspoon and J. D. Kubiatowicz, “Erasure Coding vs. Replication: A Quantitative Comparison,” *IPTPS ’02*.

[8] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A scalable content-addressable network,” in *SIGCOMM 2001*, San Diego, CA.

[9] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” in *SIGCOMM 2001*, San Diego, CA.

[10] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, “Search and replication in unstructured peer-to-peer networks,” in *ACM International Conference on Supercomputing*, June 2002.

[11] E. Cohen and S. Shenker, “Replication strategies in unstructured peer-to-peer networks,” in *SIGCOMM 2002*.

[12] Chronicle of Higher Education, “Napster was nothing compared with this year’s bandwidth problems,” Sept. 28, 2001.

[13] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, “Tapestry: An infrastructure for fault-tolerant wide-area location and routing,” Tech. Rep. UCB//CSD-01-1141, Apr. 2000.

[14] K. Sripanidkulchai, “The popularity of Gnutella queries and its implications on scalability,” Mar. 2001, Unpublished.

[15] A. Adya et al. “FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment,” in *OSDI’02*, December 2002

[16] J.R Douceur, A. Adya, W.J. Bolosky, D. Simon, M. Theimer, “Reclaiming Space from Duplicate Files in a Serverless Distributed File System,” in *ICDCS, 2002*

[17] J.R. Douceur, R.P. Wattenhofer, “Optimizing File Availability in a Secure Serverless Distributed File System,” Proceedings of *20th IEEE SRDS, 2001*