CUP: Controlled Update Propagation in Peer-to-Peer Networks

Mema Roussopoulos Mary Baker
Department of Computer Science
Stanford University
{mema, mgbaker}@cs.stanford.edu

***Abstract*—**
Recently the problem of indexing and locating content in peer-to-peer networks has received much attention. Previous work suggests it is useful to cache index entries at intermediate nodes that lie on the paths taken by search queries, but until now there has been little focus on how to maintain these intermediate caches. This paper proposes CUP, a protocol for performing Controlled Update Propagation to maintain caches of index entries in peer-to-peer networks. CUP asynchronously builds and maintains caches while answering search queries. CUP is independent of the underlying search mechanism and therefore can be used in the context of any peer-to-peer network.

CUP controls and confines propagation to updates whose cost is likely to be recovered by subsequent queries. CUP allows peer nodes to use their own incentive-based policies to determine when to receive and when to propagate updates. We compare CUP against caching with expiration at intermediate nodes and show that CUP significantly reduces average query latency (by as much as an order of magnitude) across a wide variety of workloads. More importantly, any propagation overhead incurred by CUP is compensated for by a factor of 2 to 200 times in terms of savings in cache misses.

I. Introduction

Peer-to-peer systems are self-organizing distributed systems where participating nodes both provide and receive services from each other in a cooperative effort to prevent any one node or set of nodes from being overloaded. Peer-to-peer systems have recently gained much attention, primarily because of the great number of features they offer applications that are built on top of them. These features include: scalability, availability, fault tolerance, decentralized administration, and anonymity.

Along with these features has come an array of technical challenges. For example, a fundamental problem in peer-to-peer systems is that of locating content. Given the name or a set of keyword attributes (metadata) of an object of interest, how do you locate the object within the peer-to-peer network? Most peer-to-peer networks return a set of index entries in response to a search query. These index entries hold the locations of replica nodes in the network that serve content whose metadata satisfies the search query.

Recent work suggests that metadata-based search queries for index entries can be a performance bottleneck in peer-to-peer systems [1]. As a result, designers of peer-to-peer systems suggest caching index entries at intermediate nodes that lie on the path taken by a search query [2], [3], [4], [5]. We refer to this as *Path Caching with Expiration* (PCX) because cached index entries typically have expiration times after which they are considered stale and require a new search.

PCX is desirable because it balances query load for popular entries across multiple nodes, it reduces latency, and it alleviates hot spots. However, little attention has been given to how to maintain these intermediate caches. The cache maintenance problem is interesting because the peer-to-peer model assumes that the index will change constantly as peer nodes join and leave the network, content is added to and deleted from the network, and replicas of existing content are added to alleviate bandwidth congestion at nodes holding the content. Each of these events causes a change in the current global set of valid index entries. Keeping cached index entries up-to-date therefore requires tracking which entries need to be updated, as well as tracking when interest in updating particular entries at each cache has died down so as to avoid unnecessary update propagation.

In this paper we propose a protocol for performing Controlled Update Propagation (CUP) to maintain caches in a peer-to-peer network. CUP asynchronously builds caches of index entries while answering search queries. It then propagates updates of index entries to maintain these caches. CUP is not tied to any particular search mechanism and therefore can be applied in both networks that perform structured search as well as networks that perform unstructured search. In structured search, search queries follow a well-defined path from the querying node to an authority node that holds the index entries pertaining to the query [4], [6], [5], [7]; in unstructured search, search queries haphazardly travel through the network via flooding or random walks in search of index entries [2],

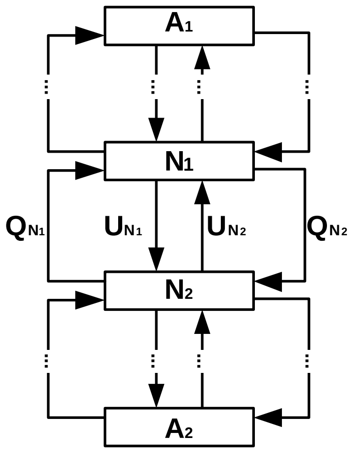

Fig. 1. CUP Query & Update Channels. A_1 and A_2 are authority nodes. A query arriving at node N_2 for an item for which A_1 is the authority is pushed onto query channel Q_{N_1} to N_1. If N_1 has a cached unexpired entry for the item, it returns it to N_2 through U_{N_1}. Otherwise, it forwards the query towards A_1. Any update for the item originating from authority node A_1 flows downstream to N_1 which may forward it onto N_2 through U_{N_1}. The analogous process holds for queries at N_1 for items for which A_2 is one of the authority nodes.

[8].

In the interest of space, in this paper we will describe how CUP works for structured peer-to-peer networks. Details of the CUP algorithms for structured and unstructured networks can be found in [9]. The basic idea is that every node in the peer-to-peer network maintains two logical channels per neighbor: a query channel and an update channel. The query channel is used to forward search queries for objects of interest to the neighbor that is closest to some “authority” node holding the entries for those objects. The update channel is used to forward query responses asynchronously to a neighbor and to update index entries that are cached at the neighbor.

Queries for an item travel “up” the query channels of nodes along the path toward the authority node for that item. Updates travel “down” the update channels along the reverse path taken by a query. Figure 1 shows this process. The process of querying for items and updating cached index entries pertaining to those items forms a CUP tree, similar to an application-level multicast tree where vertices are peer nodes interested in receiving updates for cached index entries.

The query channel enables “query coalescing”. If a node receives two or more queries for an item for which it does not have a fresh response, the node pushes only one instance of the query for that item up its query channel. This approach can have significant savings in traffic, because bursts of requests for an item are coalesced into a single request. Second, coalescing multiple queries for the same item solves the “open connection” problem suffered by some peer-to-peer systems. The asynchronous nature of the query channel relieves nodes from having to maintain many separate open connections while waiting for query responses; instead all responses return through the update channel. Through simple bookkeeping (setting an interest bit) the node registers the interest of its neighbors so it knows which of its neighbors to push the query response to when it arrives.

The cascaded propagation of updates from authority nodes down the reverse paths of search queries has many advantages. First, updates extend the lifetime of cached entries allowing intermediate nodes to continue serving queries from their caches without re-issuing new queries. It has been shown that up to fifty percent of content hits at caches are instances where the content is valid but stale and therefore cannot be used without first being revalidated [10]. These occurrences are called *freshness misses*. Second, a node that proactively pushes updates to interested neighbors reduces its load of queries generated by those neighbors. The cost of pushing the update down is recovered by the first query for the same item following the update. Third, the further down an update gets pushed, the shorter the distance subsequent queries need to travel to reach a fresh cached answer. As a result, query response latency is reduced. Finally, updates can help prevent errors. For example, an update to delete an index entry prevents a node from erroneously answering queries using the invalid entry before it expires.

In CUP, nodes decide individually when to receive updates. A node only receives updates for an item if the node has registered interest in that item. Furthermore, each node uses its own incentive-based policy to determine when to cut off its incoming supply of updates for an item. This way the propagation of updates is controlled and does not flood the network.

Similarly, nodes decide individually when to propagate updates to interested neighbors. This is necessary because a node may not always be able or willing to forward updates to interested neighbors. In fact, a node's ability or willingness to propagate updates may vary with its workload. CUP addresses this by introducing an adaptive mechanism each node uses to regulate the rate of updates it propagates downstream. A salient feature of CUP is that even if a node's capacity to push updates becomes zero, nodes dependent on the node for updates fall back to the case of PCX and incur no overhead.

We compare CUP against PCX with coalescing under typical workloads that have been observed in measurements of real peer-to-peer networks. We show that CUP significantly reduces the average query latency by as much as an order of magnitude. CUP overhead is more than compensated for by its savings in cache misses. The cost of saved misses can be two to 200 times the cost of updates pushed.

The rest of the paper is organized as follows: Section II describes in detail the design of the CUP protocol. Section III describes the cost model we use to evaluate CUP and presents experimental evidence of the benefits of CUP. Section IV discusses related work and Section V concludes the paper.

II. CUP Protocol Design

We introduce some terminology we use throughout the paper and briefly describe how structured peer-to-peer networks perform their indexing and lookup operations. We then describe the components of the CUP protocol over structured networks. We omit the CUP description for unstructured networks in this paper.

A. Background Terminology

Node: This is a node in the peer-to-peer network. Each node periodically exchanges "keep-alive" messages with its neighbors to confirm their existence and to trigger recovery mechanisms should one of the neighbors fail. Every node also maintains the two logical channels per neighbor as described in Figure 1.

Global Index: The most important operation in a peer-to-peer network is that of locating content. The basic idea in structured peer-to-peer networks is that a hashing scheme maps keys (names of content files or keywords) onto a virtual coordinate space using a uniform hash function that evenly distributes the keys to the space. The coordinate space serves as a global index that stores index entries which are *(key, value)* pairs. The value in an index entry is a pointer (typically an IP address) to the location of a replica node that stores the content file associated with the entry's key. There can be several index entries for the same key, one for each replica of the content.

Authority Node: Each node N in a structured peer-to-peer system is dynamically allocated a subspace of the coordinate space (i.e., a partition of the global index) and all index entries mapped into its subspace are owned by N. We refer to N as the authority node of these entries. *Replicas* of content whose key corresponds to an authority node N send birth messages to N to announce they are willing to serve the content. Depending on the application supported, replicas might periodically send refresh messages to indicate they are still serving a piece of content. They might also send deletion messages that explicitly indicate they are no longer serving the content. These deletion messages notify the authority node to delete the corresponding index entry from its local index directory.

Local index directory: This is the subset of global index entries owned by a node.

Search Query: A search query posted at a node N is a request to locate a replica for key K. The response to such a search query is a set of index entries that point to replicas that serve the content associated with K.

Search/Routing Mechanism: In structured networks, when a node issues a query for key *K*, the query will be routed along a well-defined path with a bounded number of hops from the querying node to the authority node for *K*. The routing mechanism is designed so that each node on the path hashes *K* using the same hash function to deterministically choose which of its neighbors will serve as the next hop. The CUP protocol is aware of but neither affects nor is affected by the underlying routing mechanism.

Query Path for Key K: This is the path a search query for key *K* takes. Each hop on the query path is in the direction of the authority node that owns *K*. If an intermediate node on this path has unexpired entries cached, the path ends at the intermediate node; otherwise the path ends at the authority node. The reverse of this path is the *Reverse Query Path* for key *K*.

Cached index entries: This is the set of index entries cached by a node N in the process of passing up queries and propagating down updates for keys for which N is not the authority. The set of cached index entries and the local index directory are disjoint sets.

Lifetime of index entries: We assume that each index entry cached at a node has associated with it a lifetime during which it is considered fresh and after which it is considered expired.

B. CUP Node Bookkeeping

At each node, index entries are grouped together by key. For each key K, the node stores a flag that indicates whether the node is waiting to receive an update for K in response to a query, and an interest bit vector. Each bit in the vector corresponds to a neighbor and is set or clear depending on whether that neighbor is or is not interested in receiving updates for K.

Each node tracks the popularity or request frequency of each non-local key K for which it receives queries. The popularity measure for a key K can be the number of queries for K a node receives between arrivals of consecutive updates for K or a rate of queries of a larger moving window. On an update arrival for K, a node uses its popularity measure to re-evaluate whether it is beneficial to continue caching and receiving updates for K. We elaborate on this cut-off decision in Section III-C.

Node bookkeeping in CUP involves no network overhead and a few tens of megabytes for hundreds of thousands of entries. With increasing CPU speeds and memory sizes, this bookkeeping is negligible when we consider the reduction in query latency achieved.

C. Update Types

We classify updates into three categories: deletes, refreshes, and appends. Deletes, refreshes, and appends originate from the replicas of a piece of content and are directed toward the authority node that owns the index entries for that content.

Deletes are directives to remove a cached index entry. Deletes can be triggered by two events: 1) a replica sends a message indicating it no longer serves a piece of content to the authority node that owns the index entry pointing to that replica. 2) the authority node notices a replica has stopped sending "keep-alive" messages and assumes the replica has failed. In either case, the authority node deletes the corresponding index entry from its local index directory and propagates the delete to interested neighbors.

Refreshes are directive messages that extend the lifetimes of cached index entries. Refreshes that arrive at a cache do not prevent errors as deletes do, but help prevent freshness misses.

Finally, appends are directives to add index entries for new replicas of content. These updates help alleviate the demand for content from the existing set of replicas since they add to the number of replicas from which clients can download content.

D. CUP Trees

Figure 2 shows a snapshot of CUP in progress. The left hand side of each node shows the set of keys for which the node is the authority. The right hand side shows the set of keys for which the node has cached index entries as a result of handling queries. For example, node A owns K3 and has cached entries for K1 and K5.

For each key, the authority node that owns the key is the root of a CUP tree. The branches of a CUP tree are formed by the paths traveled by queries from other nodes in the network such as {F, D, C, A} for K3 rooted at A. Updates originate at the root (authority node) of a CUP tree and travel downstream to interested nodes.

E. Handling Queries

Upon receipt of a query for a key *K*, there are three basic cases to consider. In each of the cases, the node updates its popularity measure for *K* and sets the appropriate bit in the interest bit vector for *K* if the query originates from a neighbor. Otherwise, if the query is from a local client, the node maintains the connection until it can return a fresh answer to the client. To simplify the protocol description we use the phrase "push the query" to indicate that a node pushes a query upstream toward the authority node. We use the phrase "push the update" to indicate that a node pushes an update downstream in the direction of the reverse query path.

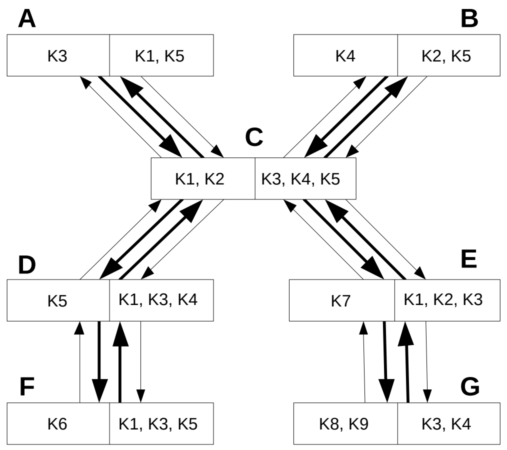

Fig. 2. CUP Trees

Case 1: Fresh Entries for key K are cached. The node uses its cached entries for *K* to push the response to the querying neighbor or local client.

Case 2: Key K is not in cache. The node adds *K* to its cache and marks it with a *Pending-Response* flag. The flag's purpose is to coalesce bursts of queries for *K* into one query. A subsequent query for *K* will be suppressed since the node is already waiting the response for the first query of the burst. Query coalescing results in significant network savings, for both PCX and CUP. In some of the workloads, coalesced queries can form up to 90 percent of the total number of queries that miss.

With every query push, a timer is set so that if the query response is lost, the node pushes up another query.

Case 3: All cached entries for key K have expired. The node must obtain the fresh index entries for *K*. If the *Pending-Response* flag is set, the node does not need to push the query; otherwise, the node sets the flag and pushes the query.

F. Handling Updates

A key feature of CUP is that a node does not forward an update for *K* to its neighbors unless those neighbors have registered interest in *K*. Therefore, with some light bookkeeping, we prevent unwanted updates from wasting network bandwidth.

Upon receipt of an update for key *K* there are three cases to consider.

Case 1: Pending-Response flag is set. This means that the update is a query response carrying a set of index entries in response to a query, the node stores the index entries in its cache, clears the *Pending-Response* flag, and pushes the update to neighbors whose interest bits are set and to local client connections open at the node.

Case 2: Pending-Response flag is clear. If all the interest bits for *K* are clear, the node decides whether it wants to continue receiving updates for *K*. The node bases its decision on *K*'s popularity measure. Each node uses its own policy for deciding whether the popularity of a key is high enough to warrant receiving further updates for it. If the node decides *K*'s popularity is low, it pushes a *Clear-Bit* control message to the sender of the update to notify it that is no longer interested in *K*'s updates. Otherwise, if the popularity is high or some of the neighbor's interest bits are set, the node applies the update to its cache and pushes the update to those neighbors.

Note that a greedy or selfish node can choose not to push updates for a key K to interested neighbors. This forces downstream nodes to fall back to PCX for K. However, by choosing to cut off downstream propagation, a node runs the risk of receiving subsequent queries from its neighbors costing it twice as much, since it must both receive and respond to the queries. Therefore, although each node has the choice of stopping the update propagation at any time, it is in its best interest to push updates for which there are interested neighbors.

Case 3: Incoming update has expired. This could occur when the network path has long delays and the update does not arrive in time. The node does not apply the update and does not push it downstream. If the *Pending-Response* flag is set then the node re-issues another query for K and pushes it upstream.

G. Handling Clear-Bit Messages

A *Clear-Bit* control message is pushed by a node to indicate to its neighbor that it is no longer interested in receiving updates for a particular key from that neighbor.

When a node receives a *Clear-Bit* message for key K, it clears the interest bit for the neighbor from which the message was sent. If the node's popularity measure for K is low and all of its interest bits are clear, the node also pushes a *Clear-Bit* message for K. This propagation of *Clear-Bit* messages toward the authority node for K continues until a node is reached where the popularity of K is high or where at least one interest bit is set.

Clear-Bit messages can be piggybacked onto queries or updates intended for the neighbor, or if there are no pending queries or updates, they can be pushed separately.

H. Adaptive Control of Update Push

Ideally every node would propagate all updates to interested neighbors to save itself from having to handle future downstream misses. However, from time to time, nodes are likely to be limited in their capacity to push updates downstream. Therefore, we introduce an adaptive control mechanism that a node uses to regulate the rate of pushed updates.

We assume each node N has a capacity U for pushing updates that varies with N's workload, network bandwidth, and/or network connectivity. N divides U among its outgoing update channels such that each channel gets a share that is proportional to the length of its queue. This allocation maintains queues of roughly equal size. The queues are guaranteed to be bounded by the expiration times of the entries in the queues. So even if a node has its update channels completely shut down for a long period, all entries will expire and be removed from the queues.

Under a limited capacity and while updates are waiting in the queues, each node can re-order the updates in its outgoing update channels by pushing ahead updates that are likely to have greater benefit. For example, a node can re-order refreshes and appends so that entries that are closer to expiring are given higher priority. Such entries are more likely to cause freshness misses which in turn trigger a new search query.

The strategy for re-ordering depends on the application. For example, in an application where query latency and accuracy are of the most importance, one can push updates in the following order: deletes, refreshes, and appends. In an application subject to flash crowds [11] that query for a particular item, appends might be given higher priority over the other updates. This would help distribute the load faster across a larger set of replicas. For all strategies, during the re-ordering any expired updates are eliminated.

I. Node Arrivals and Departures

The peer-to-peer model assumes that participating nodes will continuously join and leave the network. CUP must be able to handle both node arrivals and departures seamlessly.

Arrivals. When a new node N enters a structured peer-to-peer network, it becomes responsible for a portion of another node M's share of the global index and becomes the authority node for those index entries mapped into that portion. N, M, and all surrounding affected nodes (old neighbors of M) update the bookkeeping structures they maintain for indexing and routing purposes. This is a necessary part of maintaining the connectivity of any structured peer-to-peer network when the set of nodes in the network changes. management.

For CUP, the issues at hand are updating the interest bit vectors of the affected nodes and deciding what to do with the index entries stored at M. This may require bit vector translation. For example, if a node that previously had M as its neighbor now has N as its neighbor, the node must make the bit ID that pointed to M now point to N.

To deal with its stored index entries, M could simply not hand over any of its entries to N. This would cause entries at some of M's previous neighbors to expire and subsequent queries from those nodes will restart update propagations from N. Alternatively, M could give a copy of its stored index entries to N. Both N and M would then go through each entry and patch its bit vector. Both solutions are viable. The first solution requires no bit translation but temporarily loses the CUP update benefits and behaves like PCX for the untransferred entries. The second solution gets the benefits of transferring the entries, at the expense of transferring the index entries and performing the bit vector patching. The metadata and bit vectors for thousands of index entries can be compressed into a few kilobytes and can be piggybacked onto messages that are already being exhanged to reconfigure the topology. Once the transfer occurs, the bit vector patching is an in-memory, local operation that with today's CPU and memory capacities takes only a few seconds for a few million entries.

Departures. Node departures can be either graceful (planned) or ungraceful (due to sudden failure of a node). In either case the index mechanism in place dictates that a neighboring node M take over the departing node N's portion of the global index. To support CUP, the interest bit vectors of all affected nodes must be patched to reflect N's departure.

If N leaves gracefully, N can choose not to hand over to M its index entries. Any entries at surrounding nodes that were dependent on N to be updated will simply expire and subsequent queries will restart update propagations. Again, alternatively N may give M its set of entries. M must then merge its own set of index entries with N's, by eliminating duplicate entries and patching the interest bit vectors as necessary. If N's departure is due to a failure, there can be no hand-over of entries and all entries in the affected neighboring nodes will expire as in PCX.

III. Evaluation

The goal of CUP is to extend and multiply the benefits of PCX. In doing so, there are two key performance questions to address. First, by how much does CUP reduce the average query latency? Second, how much overhead does CUP incur in providing this reduction?

We first present the cost model based on economic incentive used by each node to determine when to cut off the propagation of updates for a particular key. We give a simple analysis of how the cost per query is reduced (or eliminated) through CUP. We then describe our experimental results comparing the performance of CUP with that of PCX.

A. Cost Model

Consider an authority node A that owns key K and consider the tree generated by issuing a query for K from every node in the peer-to-peer network. The resulting tree, rooted at A, is the *Virtual Query Spanning Tree* for K, V(A,K), and contains all possible query paths for K. The *Real Query Tree* for K, R(A,K) is a connected subtree of V(A,K) also rooted at A and contains all paths generated by real queries. The exact structure of R(A,K) depends on the actual workload of queries for K. The entire workload of queries for all keys results in a collection of criss-crossing Real Query Trees with overlapping branches.

Consider a node N within V(A,K) that is at distance D from A. We define the cost per query for K at N as the number of hops in the peer-to-peer network that must be traversed to return an answer to N. When a query for K is posted at N for the first time, it travels toward A. If none of the nodes between N and A have a fresh response cached, the cost of the query at N is $2D$: D hops up and D hops for the response to travel down. If a node on the query path has a fresh answer cached, the cost is less than $2D$. Subsequent queries for K at N that occur within the lifetime of the entries now cached at N have a cost of zero. As a result, caching at intermediate nodes can significantly lower average query latency.

We can gauge the performance of CUP by calculating the percentage of updates CUP propagates that are "justified", i.e., those whose cost is recovered by a subsequent query. Updates for popular keys are likely to be justified more often than updates for less popular keys.

A refresh update is justified if a query arrives sometime between the previous expiration of the cached entry and the new expiration time supplied by the refresh update. An append update is justified if at least one query arrives between the time the append is performed and the time of its expiration. Finally, a deletion update is considered justified if at least one query arrives between the time the deletion is performed and the expiration time of the entry to be deleted.

For each update, let T be the critical time interval described above during which a query must arrive in order for the update to be justified. Consider a node N at distance D from A in R(A,K). An update propagated down to N is justified if at least one query Q is posted within T time units at any of the nodes of the virtual subtree V(N,K).

Given the distribution of query arrivals at each node in the tree V(N,K), we can find the probability that the update at N is justified by calculating the probability that a query will arrive at some node in V(N,K). If the queries for K arrive at each node N_i in V(N,K) according to a

Poisson process with parameter λ_i, then it follows that queries for K arrive at V(N,K) according to a Poisson process with parameter Λ equal to the sum of all $\lambda' s$. Therefore, the probability that a query for K will arrive within T time units is $1 - e^{-\Lambda T}$ and equals the probability that the update pushed to N is justified. The closer to the authority N is, the higher the Λ and thus the higher the probability for an update pushed to N to be justified. For $\Lambda = 1$ query arrival per second and $T = 6$ seconds, the probability that an update arriving at N is justified is 99 percent.

The benefit of a justified CUP update goes beyond just recovery of its cost. For each hop a justified update u is pushed down to the root N of subtree V(N,K), exactly one hop is saved since without the propagation, the first subsequent query landing at a node N_i in V(N,K) within T time units will cause two hops, from N to its parent and back. This halves the number of hops traveled between N and its parent which in turn reduces query latency. In fact all subsequent queries posted somewhere in V(N,K) within T time units will benefit from N receiving u. The cumulative benefit an update u brings to subtree V(N,K) increases when N is closer to the authority nodes since there is a higher probability that queries will be posted within V(N,K). We define "investment return" as the cumulative savings in hops achieved by pushing a justified update to node N. The experiments show that the return is large even when CUP's reduction in latency is modest and is substantially large when the latency reduction is high.

High IR in some nodes, especially those close to the authority nodes may provide enough benefit margin for more aggressive CUP strategies. For example, a more aggressive strategy would be to push some updates even if they are not justified. As long as the number of justified updates is at least fifty percent the total number of updates pushed, the overall update overhead is completely recovered. Therefore, if network load is not the prime concern, an "all-out" push strategy achieves minimum latency.

B. Experiment Setup and Metrics

CUP can be viewed as extending the caching benefits of PCX. Therefore, we evaluate CUP by comparing it with PCX. We perform our simulation experiments using a wide range of parameters based on measurements of real peer-to-peer workloads [12], [13], [8], [14].

For our experiments, we simulate a content-addressable network (CAN) [4] using the Stanford Narses simulator [15]. Again, we stress that CUP is independent of the specific search mechanism used by the peer-to-peer network and can be used as a cache maintenance protocol in any peer-to-peer network.

As in previous studies (e.g., [4], [5], [16], [1], [17], [6]), we measure CUP performance in terms of the number of hops traversed in the overlay network. *Miss cost* is the total number of hops incurred by all misses, i.e. freshness and first-time misses. CUP overhead is the total number of hops traveled by all updates sent downstream plus the total number of hops traveled by all clear-bit messages upstream. (We assume clear-bit messages are not piggybacked onto updates. This somewhat inflates the overhead measure.) *Total cost* is the sum of the *miss cost* and all overhead hops incurred. Note that in PCX, the *total cost* is equal to the *miss cost*. *Average query latency* is the average number of hops a query must travel to reach a fresh answer plus the number of hops the answer must travel downstream to reach the node where the query was posted. (For coalesced queries we count the number of hops each coalesced query waits until the answer arrives.) Thus, the average latency is over all queries, including hits, coalesced and non-coalesced misses.

We compute investment return (IR) as the overall ratio of saved miss cost to overhead incurred by CUP:

$$IR = \frac{MissCost_{PCX} - MissCost_{CUP}}{OverheadCost_{CUP}}$$

Thus, as long as IR is greater than 1, CUP fully recovers its cost.

The simulation takes as input the number of nodes in the overlay peer-to-peer network, the number of keys owned per node, the distribution of queries for keys, the distribution of query inter-arrival times, the number of replicas per key, the lifetime of replicas in the system, and the fraction of the replica lifetime remaining at which refreshes are pushed out from the authority node. We present experiments for n = 2^k nodes where k ranges from 7 to 14. After a warm-up period for allowing the peer-to-peer network to connect, the measured simulation time is 3000 seconds. We present results for experiments with index entry lifetimes of five minutes to reflect the dynamic nature of peer-to-peer networks where it is prudent to assume nodes might only serve content for a few minutes at a time [13]. Refreshes of index entries occur one minute before expiration. Since both Poisson and Pareto query inter-arrival distributions have been observed in peer-to-peer environments [8], [12], we present experiments for both distributions. Nodes are randomly selected to post queries. We also present experiments where queries are posted at particular "hot spots" in the network.

We present seven sets of experiments. First, we compare the effect on CUP performance of different incentive-based cut-off policies and compare the performance of these policies to those of PCX. Second, using the best cut-off policy of the first experiment, we study how CUP performs as we scale the network. Third, we study the effect on CUP performance of varying the topology of the network by increasing the average node degree, thus decreasing the diameter of the network. Fourth, we study the effect on CUP performance of limiting the outgoing update capacities of nodes. Fifth, we study how CUP performs when queries arrive in bursts, as observed with Pareto inter-arrivals. Sixth, we study how CUP performs when there are hot spots of querying nodes in the network. These six experiments show the per-key benefits of CUP according to the query rates observed by each key. In the last experiment, we show the overall benefits of CUP when keys are queried for according to a Zipf-like distribution.

C. Varying the Cut-Off Policies

As discussed in Section III-A, the propagation of updates is beneficial only if the updates are justified; when a node's incentive to receive updates for a particular key fades, continuing to propagate updates to that node simply wastes network bandwidth. Therefore, each node needs an independent and decentralized way of controlling its intake of updates.

We base a node's incentive to receive updates for a key on the *popularity* of the key at the node. The more popular a key is, the more incentive there is to receive updates for that key, because the more likely updates for that key will be justified. For a key K, the popularity is the number of queries a node has received for K since the last update for K arrived at the node. (Note that the popularity metric is node-dependent and could be defined in another way such as with a moving average of query arrivals for K.)

We examine two types of thresholds against which to test a key's popularity when making the cut-off decision: probability-based and log-based.

A probability-based threshold uses the distance of a node N from the authority node A to approximate the probability that an update pushed to N is justified. Per our cost model of section III-A, the further N is from A, the less likely an update at N will be justified. We examine two such thresholds, a linear one and a logarithmic one. With a linear threshold, if an update for key K arrives at a node at distance D and the node has received at least αD queries for K since the last update, then K is considered popular and the node continues to receive updates for K. Otherwise, the node cuts off its intake of updates for K by pushing up a clear-bit message. The logarithmic popularity threshold is similar. A key K is popular if the node has received $\alpha \lg(D)$ queries since the last update. The logarithmic threshold is more lenient than the linear in that it increases at a slower rate as we move away from the root.

A log-based threshold is one that is based on the recent history of the last *n* update arrivals at the node. If within *n* updates, the node has not received any queries, then the key is not popular and the node pushes up a clear-bit message. A specific example of a log-based policy is the "second-chance policy", $n = 2$. When an update arrives, if no queries have arrived since the last update, the policy gives the key a "second chance" and waits for the next update. If at the next update, still no queries for K have been received, the node pushes a clear-bit message. The philosophy behind this policy is that pushing these two updates down from the node's parent costs the same as one query miss occuring at the node, since a query miss incurs one hop up to the parent and one hop down. This means that just one query arriving at the node between the two updates is enough to recover the propagation cost.

Table I compares the total cost of PCX with CUP using the linear and logarithmic polices for various α values, with CUP using second chance, and with a version of CUP that does not use any cut-off policy but instead pushes updates until the optimal push level is reached. To determine the optimal push level we make CUP propagate updates to all querying nodes that are at most p hops from the authority node. By varying the push level p, we determine the level which achieves minimum total cost. This is shown by the row labelled optimal push level and used as a baseline against which to compare PCX and CUP with the cut-off policies described.

In Table I we show cut-off policy results for a network of 2^{10} nodes and Poisson λ rates of 1, 10, 100 and 1000 queries per second. In each table entry, the first number is the total cost and the number in the parentheses is the total cost normalized by the total cost for PCX. First, we see that regardless of the cut-off policy used, CUP outperforms PCX. Second, for the lower query rates, the performance of the linear and the logarithmic policies is greatly affected by the choice of parameter α, whereas for the higher query rates, the choice of α is less dramatic. These results show that choosing a priori an α value for the linear and logarithmic policies that will perform well across all workloads is difficult.

For the higher query rates, the log-based second-chance policy performs comparably to the probability-based policies, and for the lower query rates outperforms the probability-based policies. In fact, across all rates, the second-chance policty achieves a total cost very near the optimal push level total cost. This is because, unlike the probability-based policies, the second-chance policy adapts to the timing of the queries within the workload in a manner that is independent of the distance of the node. In all remaining experiments, we use second-chance as the cut-off policy.

TABLE I

TOTAL COST PER KEY PER QUERY RATE FOR VARYING CUT-OFF POLICIES.

Policy	**1 q/s Total Cost**	**10 q/s Total Cost**	**100 q/s Total Cost**	**1000 q/s Total Cost**
PCX	61568 (1.00)	154502 (1.00)	476420 (1.00)	2296869 (1.00)
Linear, $\alpha = 0.10$	41281 (0.67)	34311 (0.22)	47132 (0.10)	196650 (0.09)
Linear, $\alpha = 0.01$	31110 (0.51)	24697 (0.16)	48330 (0.10)	196797 (0.09)
Logarithmic, $\alpha = 0.25$	30683 (0.50)	24695 (0.16)	48330 (0.10)	196797 (0.09)
Logarithmic, $\alpha = 0.10$	30683 (0.50)	24695 (0.16)	48330 (0.10)	196797 (0.09)
Second-chance	16958 (0.28)	23702 (0.15)	48330 (0.10)	196797 (0.09)
Optimal push level	15746 (0.26)	23696 (0.15)	45325 (0.095)	153309 (0.07)

TABLE II

PER-KEY COMPARISON OF CUP WITH PCX FOR VARYING NETWORK SIZES, POISSON ARRIVALS OF 1 QUERY/SECOND.

Network Size	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**
CUP / PCX MissCost	0.10	0.10	0.15	0.17	0.19	0.22	0.20	0.21
PCX AvgLat (σ)	1.51 (2.8)	2.67 (4.0)	4.49 (5.9)	6.74 (8.3)	11.01 (12.1)	17.47 (17.5)	29.29 (27.8)	45.56 (40.3)
CUP AvgLat (σ)	0.21 (1.1)	0.46 (1.6)	1.25 (3.2)	2.17 (4.4)	4.18 (7.1)	7.70 (11.3)	11.48 (15.1)	19.17 (23.7)
IR	4.15	4.88	6.29	7.83	11.43	16.14	24.85	35.98

TABLE III

PER-KEY COMPARISON OF CUP WITH PCX FOR VARYING NETWORK SIZES, POISSON ARRIVALS OF 10 QUERIES/SECOND.

Network Size	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**
CUP / PCX MissCost	0.08	0.09	0.09	0.08	0.09	0.09	0.10	0.11
PCX AvgLat (σ)	0.37 (1.58)	0.87 (2.89)	2.28 (5.72)	4.21 (8.78)	7.48 (13.35)	14.42 (21.37)	25.87 (32.73)	43.85 (48.33)
CUP AvgLat (σ)	0.03 (0.44)	0.09 (0.96)	0.26 (2.14)	0.47 (3.17)	1.14 (5.74)	2.53 (9.93)	5.26 (16.23)	9.97 (25.09)
IR	6.79	8.30	11.76	13.00	14.89	21.52	30.88	50.52

TABLE IV

PER-KEY COMPARISON OF CUP WITH PCX FOR VARYING NETWORK SIZES, POISSON ARRIVALS OF 100 QUERIES/SECOND.

Network Size	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**
CUP / PCX MissCost	0.08	0.08	0.09	0.08	0.10	0.08	0.09	0.10
PCX AvgLat (σ)	0.16 (1.07)	0.33 (1.87)	0.91 (3.65)	1.77 (5.99)	3.91 (10.79)	8.60 (18.65)	17.94 (30.63)	35.96 (49.28)
CUP AvgLat (σ)	0.01 (0.28)	0.03 (0.53)	0.08 (1.14)	0.14 (1.73)	0.36 (3.51)	0.76 (5.92)	2.06 (11.98)	5.50 (21.89)
IR	28.41	29.05	39.80	39.96	44.08	52.62	60.48	83.34

D. Scaling the Network

In this section we study CUP performance as we scale the size of the network.

Table II compares CUP and PCX for network sizes between 128 and 16384 nodes for a Poisson λ rate of 1 query per second. The first row shows the CUP miss cost as a fraction of the PCX miss cost. The second and third rows show the average query latency in hops for PCX and CUP respectively. The number in parentheses is the standard deviation. As can be observed, CUP reduces average query latency respectively by 9.77, and 17.81, and 26.39 hops for the 4096, 8192, and 16384 node networks. This shows a substantial reduction in average query latency that improves with increasing network size. Comparing the standard deviations of CUP and PCX we see that PCX also has more variability around its average query latency than CUP does.

The fourth row in Table II shows the IR per overhead push performed by CUP. We observe a growth in the rate of return with returns of 16.14, 24.85, and 35.98 for the last three network sizes. These numbers are encouraging, especially considering the overhead investment is completely recovered.

Tables III, IV and V show the corresponding table for λ rates of 10, 100, and 1000 queries per second.

To get a quick idea of how CUP performs across all four query rates, figure 3 shows the IR of CUP versus network size for Poisson with λ = 1, 10, 100, and 1000 queries per second. From the figure we see that for a particular network size, if we increase the query rate the IR increases, and for a particular query rate, if we increase the network size, the IR also increases. This demonstrates that CUP scales to higher query rates and higher network sizes.

E. Varying the Network Topology

In general, different peer-to-peer networks exhibit different topologies and thus different network diameters.

TABLE V

PER-KEY COMPARISON OF CUP WITH PCX FOR VARYING NETWORK SIZES, POISSON ARRIVALS OF 1000 QUERY/SECOND.

Network Size)	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**
CUP / PCX MissCost	0.08	0.08	0.08	0.08	0.12	0.08	0.09	0.09
PCX AvgLat (σ)	0.12 (0.88)	0.23 (1.42)	0.54 (2.54)	0.92 (3.89)	1.90 (6.77)	4.01 (11.67)	8.55 (19.93)	18.36 (34.40)
CUP AvgLat (σ)	0.01 (0.26)	0.02 (0.41)	0.05 (0.75)	0.07 (1.10)	0.16 (2.05)	0.34 (3.59)	0.79 (6.55)	2.22 (12.29)
IR	202.45	185.99	226.84	192.11	192.01	207.12	221.55	264.80

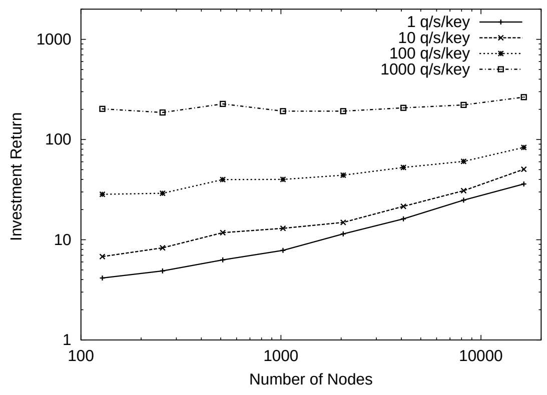

Fig. 3. IR vs. net size. (Log-scale axes.)

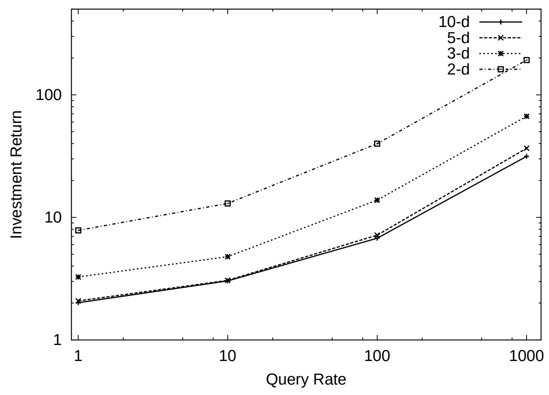

Fig. 4. IR vs. query rate, varying dimensions. (Log-scale axes.)

The particular topology created depends on the protocol the peer nodes use to join the network and to keep it connected. The CAN design is based on a d-dimensional coordinate space, with our experiments thus far having been for $d = 2$. Increasing the number of dimensions results in a topology where nodes have higher degree and the network has smaller diameter. Smaller diameter implies that the average path length of a query on a miss is shorter for both PCX and CUP. Shorter query paths means that PCX miss latencies decrease, which implies that the benefits of CUP may be less pronounced. On the other hand, CUP total update cost also decreases since there will be shorter distances for updates to travel. As a result, we find that CUP continues to provide significant savings in terms of both overall total cost, latency reduction, and IR per overhead push.

In this set of experiments we study the effect of increasing the number of CAN dimensions on a network with 1024 nodes. The dimensions chosen for this experiment are 2, 3, 5, and 10. These dimensions result in network diameters of 24, 12, 8, and 8 respectively. (For a network of 1024 nodes, increasing beyond five dimensions does not reduce the network diameter any further.) The queries arrive according to a Poisson process with λ rate of 1, 10, 100, and 1000 queries per second for a network of 1024 nodes. Figure 4 shows the IR versus the query rate for each dimension. From the figure we see that the curves for dimensions 5 and 10 are very similar because they have equal network diameters. We also see that dimension 2 achieves the highest IR across all query rates, and that the IR decreases with dimension. However, even for the higher dimensions (5 and 10), the IR per overhead hop is at least 2.1 per overhead hop for 1 q/s and increases to 36.6 per overhead hop for 1000 q/s.

In Tables VI and VII, for these dimensions, we show the CUP miss cost as a fraction of PCX miss cost, the CUP miss latency, the PCX miss latency, and the ratio of saved misses to CUP update cost.

F. Varying Outgoing Update Capacity

Our experiments thus far show that CUP outperforms PCX under conditions where all nodes have full outgoing update capacity. A node with full outgoing capacity is a node that can and does propagate all updates for which there are interested neighbors. In reality, an individual node's outgoing capacity will vary with its workload, network connectivity, and willingness to propagate updates. In this section we study the effect on CUP performance of reducing the outgoing update capacity of nodes.

We present an experiment run on a network of 1024 nodes. In this experiment, after a five minute warm up period, we randomly select twenty percent of the nodes and reduce their outgoing capacity to a fraction of their full capacity. These nodes operate at reduced capacity for ten minutes after which they return to full capacity. After another five minutes for stabilization, we randomly select another set of twenty percent of the nodes and reduce their capacity for ten minutes. We proceed this way for the

TABLE VI

COMPARISON OF CUP WITH PCX FOR VARYING DIMENSIONS

Average Rate (q/s)	**1**	**1**	**1**	**1**	**10**	**10**	**10**	**10**
Number of dimensions)	**10**	**5**	**3**	**2**	**10**	**5**	**3**	**2**
CUP / PCX MissCost	0.33	0.33	0.26	0.17	0.09	0.09	0.09	0.08
PCX AvgLat (σ)	2.09 (2.27)	2.21 (2.38)	3.14 (3.49)	6.74 (8.25)	0.94 (1.80)	0.95 (1.82)	1.51 (3.15)	4.21 (8.78)
CUP AvgLat (σ)	1.04 (1.53)	1.10 (1.59)	1.37 (2.22)	2.17 (4.37)	0.11 (0.68)	0.11 (0.67)	0.17 (1.14)	0.47 (3.17)
IR	2.01	2.08	3.26	7.83	3.03	3.06	4.78	13.00

TABLE VII

COMPARISON OF CUP WITH PCX FOR VARYING DIMENSIONS

Average Rate (q/s)	**100**	**100**	**100**	**100**	**1000**	**1000**	**1000**	**1000**
Number of dimensions)	**10**	**5**	**3**	**2**	**10**	**5**	**3**	**2**
CUP / PCX MissCost	0.08	0.08	0.09	0.08	0.08	0.08	0.08	0.08
PCX AvgLat (σ)	0.31 (1.41)	0.30 (1.37)	0.61 (2.50)	1.77 (5.99)	0.16 (0.97)	0.14 (0.90)	0.30 (1.59)	0.92 (3.89)
CUP AvgLat (σ)	0.03 (0.40)	0.03 (0.42)	0.06 (0.78)	0.14 (1.73)	0.01 (0.28)	0.01 (0.25)	0.02 (0.46)	0.07 (1.10)
IR	7.16	6.75	13.82	39.96	36.68	31.46	66.81	192.11

entire 3000 seconds during which queries are posted, so capacity loss occurs three times during the simulation.

Figure 5 shows the ratio of CUP total cost to PCX total cost versus reduced capacity c for this experiment and for four different Poisson query rates λ. The capacity reduction c ranges from 0, implying that no updates are propagated to 1 in which nodes have full outgoing capacity. $c = .25$ means that a node is only capable/willing of pushing out one-fourth the updates it receives.

Note that even when one fifth of the nodes do not propagate any updates, the total cost incurred by CUP is about half that of PCX. As the outgoing capacity increases, the total cost decreases smoothly until $c = 1$ where CUP achieves its full potential. A key observation from these experiments is that CUP's performance degrades gracefully as the capacity c decreases. This is because reduction in update propagation also results in reduction of its associated overhead. Therefore, the capacity reduction should be seen as a missed opportunity for higher returns rather than as an overall loss. Clearly though, CUP achieves its full potential when all nodes have maximum propagation capacity.

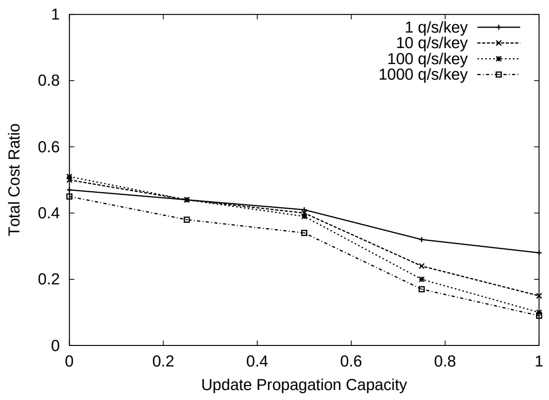

Fig. 5. Total cost ratio vs. update propagation capacity

G. Pareto Query Arrivals

Recent work has observed that in some peer-to-peer networks, query inter-arrivals exhibit burstyness on several time scales [12], making the Pareto distribution a good candidate for modeling these inter-arrival times. Therefore, in this section we compare CUP with PCX under Pareto inter-arrivals.

The Pareto distribution has two parameters associated with it: the shape parameter $\alpha > 0$ and the scale parameter $\kappa > 0$. The cumulative distribution function of inter-arrival time durations is $F(x) = 1 - (\frac{\kappa}{(x+\kappa)})^{\alpha}$. This distribution is heavy-tailed with unbounded variance when $\alpha < 2$. For $\alpha > 1$, the average number of query arrivals per time unit is equal to $\frac{(\alpha-1)}{\kappa}$. For $\alpha <= 1$, the expectation of an inter-arrival duration is unbounded and therefore the average number of query arrivals per time unit is 0.

We ran experiments for a range of α and κ values and present representative results here. Table VIII compares CUP with PCX for α equal to 1.1 and 1.25 respectively for a network of 1024 nodes. We set the value of κ in each run so that the average rate of arrivals $\frac{(\alpha-1)}{\kappa}$ equals 1, 10, 100, and 1000 queries per second to match the λ rate of the Poisson experiments in previous sections.

As α decreases toward 1, query interarrivals become more bursty. Queries arrive in more frequent and more intense bursts, followed by idle periods of varying lengths. If an idle period occasionally falls in the heavy-tail portion of the Pareto distribution (i.e. very long idle period), then second chance CUP propagation cost could be unrecoverable, since the next query may arrive long after the cached entry has expired. However, CUP does well under bursty conditions because when it is able to refresh a cache before a burst of queries, it saves a large penalty which by far outweighs any unrecovered overhead that occurs during the occasional, very long idle period. Therefore, refreshing the cache in time provides greater benefits with increasing burstyness. The table results confirm this. In going from $\alpha = 1.25$ to $\alpha = 1.1$, we see that the average query latency reduction CUP achieves generally improves and the IR increases for all query rates.

H. Query Hot Spots

In this experiment we show how CUP performs when there are hot spots in the network, that is when a small portion of the network is posting queries for particular keys. It is possible that some portions of the network will have high interest whereas others will show only low interest for a particular key. We examine how CUP performs in such a scenario.

We show results for an experiment for a 1024 node network in which queries for a particular key were generated according to a Poisson process with 1, 10, 100, and 1000 queries per second. Nodes to post the queries were chosen according to a zipf distribution of the node IDs. This has the effect of having a small number of active querying nodes and a large number of nodes which issue very few queries for the key throughout the simulation.

Table IX compares CUP with PCX. From the table we see that the results of the hot spot experiment are similar to those where nodes are randomly chosen to post queries. This is because the CUP tree simply grows branches in the direction of nodes that have interest in the key. Each node decides individually when to cut off its intake of updates and this decision is independent of the query rates or propagation behavior of other nodes that lie in a different part of the network.

Table X compares CUP with PCX in a network with hot spots and Pareto arrivals, with $\alpha = 1.1$ and average rate of arrivals of 1, 10, 100, and 1000 queries per second. Again the results here are similar to those observed when nodes are randomly selected to post queries.

I. Zipf-like Key Distributions

A recent study has shown that queries for multiple keys in a peer-to-peer network can follow a Zipf-like distribution, with a small portion of the keys getting the most queries [14]. That is, the number of queries received by the *i'th* most popular key is proportional to $\frac{1}{i^\alpha}$ for constant α.

In this section we compare CUP with PCX in a network of 1024 nodes, where each node owns one key. The query distribution among the 1024 keys follows a Zipf-like distribution with parameter $\alpha = 1.2$. Table XI shows results for Poisson arrivals where the λ rates are 100, 1000, 10000, and 100000 queries per second. (We also ran simulations with α = 0.80 and 2.40 and with Pareto arrivals with equivalent average rates and the results were similar.)

TABLE XI
CROSS-KEY COMPARISON OF CUP WITH PCX, FOR POISSON ARRIVALS AND ZIPF-LIKE KEY DISTRIBUTION.

Avg Rate (q/s)	**100**	**1000**	**10000**	**100000**
CUP / PCX MissCost	0.45	0.23	0.10	0.08
PCX AvgLat (σ)	10.6 (9.9)	6.9 (8.9)	3.4 (7.5)	1.53 (5.47)
CUP AvgLat (σ)	7.4 (8.5)	2.6 (5.2)	0.4 (2.7)	0.13 (1.67)
IR	6.57	8.52	10.98	30.02

From the table we see that CUP outperforms PCX with IR ranging from 6.57 to 30.02. The latency reduction ranges from 3.2 (for 100 q/s) to an order of magnitude reduction (for 100000 q/s, latency dropped from 1.53 to 0.13). The Zipf-like distribution causes some of the keys to get a large percentage of the queries, leaving others to be asked for quite rarely. For such keys, caching does not help since the entry expires by the time the key is queried for again, and the query rate for these keys is not high enough to recover the update propagation. However, the IR for the very hot keys is high enough to by far offset the unrecoverable cost of other less popular keys. As a result, CUP achieves an overall IR of at least 6.57 for 100 q/s and as much as 30.02 for 100000 q/s.

For comparison, Table XII shows representative results for Pareto arrivals with α equal to 1.1 and 1.25 respectively for a network of 1024 nodes. We set the value of κ in each run so that the average rate of arrivals $\frac{(\alpha-1)}{\kappa}$ equals 1, 10, 100, and 1000 queries per second.

IV. RELATED WORK

To our knowledge, CUP is the first protocol aimed at maintaining caches of index entries to improve search queries in peer-to-peer networks. While designers of peer-to-peer systems [2], [4], [5], [6], [7] advocate caching index entries to improve performance, there has been little follow-up work studying when and where to cache entries and how to maintain these cached entries.

Cox et al. study providing DNS service over a peer-to-peer network [18]. They cache index entries which are DNS mappings along search query paths. Similarly, the TerraDir Distributed Directory caching scheme has nodes along the search query path cache pointers to other nodes previously traversed by the query [3]. In each of these examples, cached index entries have expiration times and are not refreshed or maintained until a miss or failure occurs.

Path caching of content in peer-to-peer systems has received more attention. Freenet [19], CFS [20], PAST [16], and Lv et al [21] each perform path caching, or caching

TABLE VIII

PER-KEY, PER-QUERY RATE COMPARISON OF CUP WITH PCX FOR PARETO ARRIVALS.

Average Rate (q/s)	**1**	**1**	**10**	**10**	**100**	**100**	**1000**	**1000**
Pareto rate (a)	**1.25**	**1.1**	**1.25**	**1.1**	**1.25**	**1.1**	**1.25**	**1.1**
CUP / PCX MissCost	0.24	0.14	0.08	0.07	0.07	0.09	0.08	0.08
PCX AvgLat (σ)	7.77 (9.3)	6.99 (9.4)	3.84 (8.4)	4.01 (8.8)	1.75 (5.9)	1.61 (5.6)	1.00 (4.0)	1.10 (4.2)
CUP AvgLat (σ)	3.16 (5.7)	1.71 (4.4)	0.42 (3.0)	0.37 (2.8)	0.13 (1.7)	0.15 (1.7)	0.08 (1.2)	0.09 (1.2)
IR	6.41	7.49	13.09	16.03	43.25	53.57	223.97	293.30

TABLE IX

COMPARISON OF CUP WITH PCX FOR VARYING QUERY RATES AND HOT SPOT, POISSON ARRIVALS

Average Rate (q/s)	**1**	**10**	**100**	**1000**
CUP / PCX MissCost	0.16	0.08	0.08	0.09
PCX AvgLat (σ)	7.07 (8.85)	3.58 (7.76)	1.62 (5.57)	0.96 (3.95)
CUP AvgLat (σ)	2.06 (4.29)	0.39 (2.40)	0.13 (1.59)	0.08 (1.22)
IR	8.87	11.47	32.35	193.89

TABLE X

COMPARISON OF CUP WITH PCX FOR VARYING QUERY RATES AND HOT SPOT, PARETO ARRIVALS

Average Rate (q/s)	**1**	**10**	**100**	**1000**
CUP / PCX MissCost	0.16	0.08	0.08	0.09
PCX AvgLat (σ)	6.87 (9.41)	3.56 (7.93)	1.59 (5.42)	1.13 (4.23)
CUP AvgLat (σ)	1.68 (4.39)	0.35 (2.40)	0.14 (1.63)	0.10 (1.32)
IR	7.98	14.20	47.56	294.89

TABLE XII

CROSS-KEY COMPARISON OF CUP WITH PCX FOR PARETO ARRIVALS AND ZIPF-LIKE KEY DISTRIBUTION.

Average Rate (q/s)	**1**	**1**	**10**	**10**	**100**	**1000**
Pareto rate (a)	**1.25**	**1.1**	**1.25**	**1.1**	**1.1**	**1.1**
CUP / PCX MissCost	0.85	0.79	0.65	0.66	0.44	0.23
PCX AvgLat (σ)	14.10 (10.54)	13.85 (10.51)	12.67 (10.32)	12.44 (10.29)	10.48 (9.90)	6.87 (8.91)
CUP AvgLat (σ)	13.39 (10.46)	12.88 (10.38)	11.03 (9.92)	10.73 (9.82)	7.19 (8.38)	2.49 (5.16)
IR	1.38	1.93	3.94	3.75	6.69	8.69

of content along the search path of a query. These studies do not focus on cache maintenance, but rather depend on expiration or cache size constraints to limit the use of stale content.

CUP trees are similar to application-level multicast trees, particularly those built on peer-to-peer networks. These include Scribe [17] built on top of Pastry [6] and Bayeaux built on top of Tapestry [7]. Here, we focus on Scribe. Scribe is a publish-subscribe infrastructure where subscribers interested in a topic join its corresponding multicast group. Scribe creates a multicast tree rooted at the rendez-vous point of each multicast group. Publishers send a message to the rendez-vous point which then transmits the message to the entire group by sending it down the multicast tree. The multicast tree is formed by joining the Pastry routes from each subscriber node to the rendez-vous point. Scribe could apply the ideas CUP introduces to provide update propagation for cache maintenance in Pastry.

Cohen and Kaplan study the effect that aging through cascaded caches has on the miss rates of web client caches [22]. For each object an intermediate cache refreshes its copy of the object when its age exceeds a fraction v of the lifetime duration. The intermediate cache does not push this refresh to the client; instead, the client waits until its own copy has expired at which point it fetches the intermediate cache's copy with the remaining lifetime. For some sequences of requests at the client cache and some v's, the client cache can suffer from a higher miss rate than if the intermediate cache only refreshed on expiration. Their model assumes zero communication delay. A CUP tree could be viewed as a series of cascaded caches in that each node depends on the previous node in the tree for updates to an index entry. The key difference is that in CUP, refreshes are pushed down the entire tree of interested nodes. Therefore, barring communication delays, whenever a parent cache gets a refresh so does the interested child node. In such situations, the miss rate at the child node actually improves.

V. Conclusions

CUP is a protocol for maintaining caches of index entries in peer-to-peer networks. CUP query channels coalesce bursts of queries for the same item into a single query. CUP update channels asynchronously transport query responses and refresh intermediate caches. Through light book-keeping and incentive-based propagation cutoff policies, CUP controls and confines propagations to updates that are likely to be justified. In fact, CUP's overhead is compensated for by a factor of 2 to 300 times in terms of savings in cache misses.

When compared with path caching with expiration (PCX), CUP significantly reduces the average query latency over a wide variety of workloads, including Poisson and Pareto query arrivals, networks of increasing size and various topologies, and uniform and Zipf-like multi-key query distributions. We have also shown that even with limited update propagation, CUP continues to outperform PCX. The overall conclusion is that CUP benefits increase with network size and query rates. Although Poisson query interarrivals may result in less unrecovered overhead than Pareto arrivals, the sheer force of Pareto bursts results in higher CUP benefit (investment return).

We believe that CUP provides a general purpose framework for maintaining metadata in peer-to-peer networks. We have leveraged the CUP protocol to deliver metadata required for effective load-balancing of content demand across replica nodes [9]. Future work includes using CUP to enhance management of dynamic content replication, publish-subscribe applications, and price negotiation and auctioning of services amongst nodes in a peer-to-peer network.

References

[1] Y. Chawathe, S. Ratnasamy, S. Shenker, and L. Breslau, "Can Heterogeneity Make Gnutella Scale?," May 2002, http://research.att.com/ yatin/publications/docs/gdr-stanford.ppt.

[2] "The Gnutella Protocol Specification v0.4," http://gnutella.wego.com/.

[3] B. Silaghi, B. Bhattacharjee, and P. Keleher, "Routing in the TerraDir Directory Service," 2002, http://motefs.cs.umd.edu/terradir/.

[4] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," in *Sigcomm*, 2001.

[5] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," in *Sigcomm*, 2001.

[6] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *MiddleWare*, Nov. 2001.

[7] 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, U. C. Berkeley, Apr. 2001.

[8] P. Cao, "Search and Replication in Unstructured Peer-to-Peer Networks," in *Sigmetrics*, June 2002.

[9] M. Roussopoulos, *Controlled Update Propagation in Peer-to-Peer Networks*, Ph.D. thesis, Stanford University, 2002.

[10] E. Cohen and H. Kaplan, "Refreshment Policies for Web Content Caches," in *Infocom*, 2001.

[11] T. Stading, P. Maniatis, and M. Baker, "Peer-to-Peer Caching Schemes to Address Flash Crowds," in *IPTPS*, Mar. 2002.

[12] E. P. Markatos, "Tracing a large-scale Peer-to-Peer System: an hour in the life of Gnutella," in *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

[13] S. Saroiu, P. K. Gummadi, and S. D. Gribble, "A Measurement Study of Peer-to-Peer File Sharing Systems," in *MMCN*, 2002.

[14] K. Sripanidkulchai, "The Popularity of Gnutella Queries and its Implication on Scalability," Feb. 2001, http://www-2.cs.cmu.edu/ kunwadee/research/p2p/gnutella.html.

[15] P. Maniatis, T.J. Giuli, and M. Baker, "Enabling the Long-Term Archival of Signed Documents through Time Stamping," Tech. Rep. cs.DC/0106058, Stanford University, June 2001.

[16] A. Rowstron and P. Druschel, "Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility"," in *SOSP*, Oct. 2001.

[17] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel, "SCRIBE: The design of a large-scale event notification infrastructure," in *NGC*, 2001.

[18] R. Cox, A. Muthitacharoen, and R. T. Morris, "Serving DNS using a Peer-to-Peer Lookup Service," in *IPTPS*, Mar. 2002.

[19] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System," in *DIAU*, July 2000.

[20] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area Cooperative Storage with CFS," in *SOSP*, 2001.

[21] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and Replication in Unstructured P2P Networks," in *ICS*, 2002.

[22] E. Cohen and H. Kaplan, "Aging Through Cascaded Caches: Performance Issues in the Distribution of Web Content," in *Sigcomm*, 2001.