USENIX Association

# Proceedings of the General Track: 2003 USENIX Annual Technical Conference

San Antonio, Texas, USA
June 9-14, 2003

# 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

This paper proposes CUP, a protocol for performing Controlled Update Propagation to maintain caches of metadata in peer-to-peer networks. To moderate propagation without imposing a global policy, CUP introduces the notion of individual node *investment return*. CUP allows each node to determine when it has economic incentive to receive and to propagate updates. A node participates in propagation only when the benefit (*investment return*) it secures from receiving and propagating updates outweighs its cost of propagation.

We extensively evaluate the CUP protocol in maintaining caches of metadata for locating content in peer-to-peer networks. We demonstrate that propagation of updates reduces the average latency of content search queries by as much as an order of magnitude across a variety of workloads. We propose and evaluate the use of *popularity-based* incentives to drive a node's propagation policy. These include incentives based on probabilistic as well as history-based models of investment return. Using these policies, we show that CUP nodes recover their propagation overhead by a factor of 2 to 300, thus offering a lean but powerful protocol.

## 1 Introduction

Peer-to-peer networks are self-organizing distributed systems where participating nodes both provide and receive services from each other in a cooperative effort without distinguished roles as pure clients or pure servers. Peer-to-peer networks 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 desirable 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 metadata in response to a search query. This metadata typically consists of index entries that point to the locations of nodes that serve replicas of the content of interest, but could also include other information such as pricing, trust, connection speed, or load information about these serving nodes.

Recent work suggests that metadata-based search queries for locating content can be a performance bottleneck in peer-to-peer systems [CRSB02]. As a result, designers of peer-to-peer systems suggest caching metadata at intermediate nodes that lie on the path taken by a search query [gnu, SBK02, RFH+01, SMK+01]. We refer to this as *Path Caching with Expiration* (PCX) because cached metadata entries typically have expiration times after which they are considered stale and require a new search.

PCX is desirable because it distributes query load for popular metadata items 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 challenging because the peer-to-peer model assumes that the global set of valid metadata 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. Nodes that cache metadata to serve queries in a more timely fashion need to know about changes to the metadata to serve queries better. Keeping cached metadata up-to-date therefore requires tracking which metadata items need to be updated, as well as tracking when interest in updating particular items at each cache has subsided to avoid unnecessary update propagation for the maintenance of these items.

In this paper we propose a protocol for performing Controlled Update Propagation (CUP) to maintain caches of metadata in a peer-to-peer network. CUP asynchronously builds caches of metadata while answering search queries. It then propagates updates of metadata to maintain these caches. To moderate this propagation, CUP introduces the notion of individual node *investment return*. Rather than imposing a global propagation policy, in CUP, nodes receive and propagate updates only when they have personal economic incentive

to do so. This occurs when the investment return (or benefit) a node secures by propagation outweighs the cost of propagation and thus, all overhead is recovered.

A node proactively receives updates for metadata items from a neighbor only if the node has registered interest with the neighbor. A node that proactively receives an update for a metadata item saves itself from handling a follow-up query for the same item that, without the application of the update, would otherwise miss at the node. Handling a miss involves generating network traffic to forward the query on to one's neighbor(s) and to receive a response. Therefore, from a node's perspective, a received update is *justified* if the update saves the node from the cost of handling queries. A node will only have interest in receiving updates as long as it continues to receive queries for that item.

In CUP, 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 moderated and does not flood the network. We introduce several *popularity-based* incentives to drive a node's decisions to receive metadata updates. The first class of policies is probabilistic where a node computes the probability that a received update is justified using an estimate of the number of nodes that depend on this node for answers to queries for the item. The second class is "history-based," where the node compares the ratio of query arrivals to update arrivals in a sliding window of update arrivals. These policies favor the receipt of updates for popular items since these items generate queries most often.

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. A salient feature of CUP is that even when 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 under typical workloads that have been observed in measurements of real peer-to-peer networks. We show that CUP reduces the average query latency by as much as an order of magnitude. CUP propagation overhead is more than compensated for by its savings in cache misses. The cost of saved misses can be two to 300 times the cost of updates pushed. Finally, since nodes make propagation decisions independently and without coordination from other nodes, CUP is simple to implement, which is crucial for a peer-to-peer network with potentially thousands of participants.

## 2 Background Terminology

The following terms give some background on how structured peer-to-peer networks perform their indexing and lookup operations. These help clarify the description of CUP over structured networks in the next section.

*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.

*Global Index*: A fundamental 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 node that stores a replica of the content 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*.

*PCX*: Recently, researchers have suggested caching metadata with expiration times along the reverse query path [gnu, SBK02, RFH+01, SMK+01] as the query response is propagated down to the querying node.

*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 query responses 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*: 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.

## 3 CUP Protocol Design

We give a brief overview of CUP and then describe the components of the CUP protocol in detail.

### 3.1 CUP Overview

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. As described above, in structured search, queries follow a well-defined path from the querying node to an authority node that holds the index entries pertaining to the query [RFH+01, RD01a, SMK+01, ZKJ01]; in unstructured search, queries haphazardly travel through the network via flooding or random walks in search of index entries [gnu, LCC+02].

In the interest of space, in this paper we describe and evaluate how CUP works to maintain caches of index entries in structured peer-to-peer networks. 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 the 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

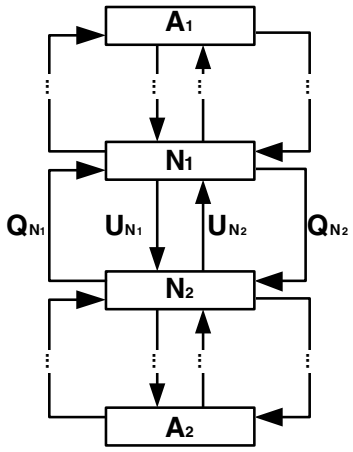


Figure 1: CUP Query & Update Channels. $A_1$ and $A_2$ are authority nodes for some objects. A query arriving at node $N_2$ for an item for which $A_1$ is the authority is pushed onto query channel $Q_{N_1}$ to $N_1$. If $N_1$ has a cached unexpired entry for the item, it returns it to $N_2$ through $U_{N_1}$. Otherwise, it forwards the query towards $A_1$. Any update for an 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.

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 queries for an item are coalesced into a single request. 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 re-validated [CK01c]. 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. 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, search query latency is reduced.

Reducing search query latency is important because the user must wait until the search query has successfully returned a set of index entries before choosing from which replica node to download the content. Finally, updates can help prevent errors by invalidating outdated entries. For example, an update to delete a fresh but invalid index entry prevents a node from erroneously answering queries using the entry before it expires.

## 3.2 CUP 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.

## 3.3 CUP Node Bookkeeping

At each node, index entries are grouped together by key. For each key K, the node stores a "Pending-Response" flag that indicates whether the node is waiting to receive a response to a query for K, 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 in a sliding window of time. 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 4.4.

Node bookkeeping in CUP involves no network overhead and a few 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.

## 3.4 Handling Queries in CUP

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.

**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 awaiting 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 we evaluate, 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 delayed, 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.

## 3.5 Handling Updates in CUP

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, CUP does not push unwanted updates.

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 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 which would cost it more, since it must both receive and respond to these 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.

## 3.6 Handling Clear-Bit Messages in CUP

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.

## 3.7 Node Arrivals and Departures in CUP

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.

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 would establish new 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 their bit vectors. 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 CUP benefits for the transferred entries, at the expense of transferring them 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 exchanged 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 peer-to-peer index mechanism 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 establish new 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.

## 4 Evaluation

The main goal of CUP is to continuously harvest 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 define the notion of a CUP tree. We use this definition to present a 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.

### 4.1 CUP Trees

Figure 2 shows a snapshot of CUP in progress for a network with seven peer nodes. The left half of each node shows the set of keys for which the node is the authority. The right half shows the set of keys for which the node has cached index entries as a result of handling queries. For example, node C owns K1 and K2 and has cached entries for K3, K4, and K5.

The process of querying for a key K and updating cached index entries pertaining to K forms a tree which we refer to as the *Real CUP Tree*. This tree, denoted R(A,K), is similar to an application-level multicast tree and has as its root the authority node A for K. The exact structure of R(A,K) depends on the actual workload of queries for K. The branches of the tree are formed by the paths traveled by queries from other nodes in the network. For example, in Figure 2, the tree R(C,K1) has grown branch {F, D, C} as the result of a query for K1 at node F. Updates for K1 originate at the root (authority node) C and travel down the tree to interested nodes A, D, E, and F. The entire workload of queries for all keys results in a collection of criss-crossing Real CUP Trees with overlapping branches.

We define the *Spanning CUP Tree* for key K, S(A,K) as the tree that contains all possible query paths for K. This is the tree that would be generated by issuing a query for K from every node in the peer-to-peer network. For example, in Figure 2, S(C,K1) is rooted at C (level 0), has nodes A, B, D, E at level 1, and nodes F and G at level 2.

### 4.2 Cost Model

Consider a node N within spanning tree S(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.

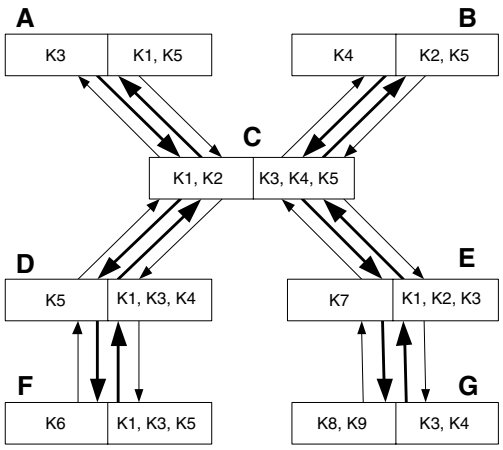


Figure 2: CUP Trees

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 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 is posted within $T$ time units at any of the nodes of the spanning subtree S(N,K). For example, if we assume a Poisson query arrival rate $\lambda$ of one query per second at nodes in S(N,K) and $T = 6$, then the probability that an update arriving at N is justified is $1 - e^{-\lambda T} = 1 - e^{-1*6} = .99$.

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 S(N,K), exactly one hop is saved since without $u$'s propagation, entries in all nodes of S(N,K) will expire and the first subsequent query landing at a node $N_i$ in S(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 S(N,K) within $T$ time units will benefit from N receiving $u$. The cumulative benefit an update $u$ brings to subtree S(N,K) increases when N is closer to the authority node since there is a higher probability that queries will be posted within S(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.

## 4.3 Experiment Setup and Metrics

We evaluate CUP by comparing it with PCX with coalescing. We perform our simulation experiments using models derived from measurements of real peer-to-peer workloads [Mar02, SGG02, LCC+02, Sri01].

For our experiments, we simulate a content-addressable network (CAN) [RFH+01] using the Stanford Narses simulator [MGB01]. 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., [RFH+01, SMK+01, RD01b, CRSB02, RKCD01, RD01a, ZKJ01]), 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 misses 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 or equal to 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 index entries in the system, and the fraction of an entry's lifetime remaining at which refreshes for the entry 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. Since both Poisson and Pareto query inter-arrival distributions have been observed in peer-to-peer environments [LCC+02, Mar02], we present experiments for both distributions. Nodes are randomly selected to post queries. We also performed experiments where queries are posted at particular "hot spots" in the network and found similar results. These, as well as other results which we omit in the interest of space, can be found elsewhere [Rou02].

We present results for experiments where index entry lifetimes are five minutes and refreshes occur one minute before expiration. We choose these values to reflect the dynamic and unpredictable nature of peer-to-peer networks. It has been found that the median user session duration of a peer is approximately sixty minutes [SGG02]. However, content may become available on a peer or be deleted from the peer at any point during the user session. This results in actual content availability that is on the order of a few minutes [CLL02]. We therefore take the safe approach of validating that the content is still available every few minutes. This is also in line with designers of structured peer-to-peer networks who advocate periodic refreshes (keep-alive messages) between the peers storing replicas of a particular content and the authority node for that content [RFH+01, RD01a]. If there were some way to ensure that lifetimes of entries could be set for longer, then we find that CUP continues to provide benefits, albeit reduced, since PCX would incur fewer misses. Unfortunately, making such guarantees would require placing a global availability policy across autonomous peer nodes.

We present six 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 that 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

Table 1: 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, $\alpha = 0.25$ | 55475 (0.90) | 72022 (0.47) | 49341 (0.10) | 196650 (0.09) |
| Linear, $\alpha = 0.10$ | 41281 (0.67) | 34311 (0.22) | 47132 (0.10) | 196650 (0.09) |
| Logarithmic, $\alpha = 0.5$ | 31658 (0.51) | 27311 (0.18) | 47785 (0.10) | 196797 (0.09) |
| Logarithmic, $\alpha = 0.25$ | 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) |

with Pareto inter-arrivals. These five experiments show the per-key benefits of CUP when keys are queried for according to a uniform distribution. In the last experiment, we show the overall benefits of CUP when keys are queried for according to a Zipf-like distribution.

## 4.4 Varying the Cut-Off Policies

As discussed in Section 4.2, 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 update propagation 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 updates for that key are more likely to 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 history-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 4.2, 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 $\alpha D$ queries for K since the last update for some constant $\alpha \geq 0$, 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 history-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 history-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 occurring 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 first update and the expiration of the second update is enough to recover their propagation cost.

Table 1 compares PCX with CUP using the linear and logarithmic polices for various $\alpha$ 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 labeled "optimal push level" and used as a baseline against which to compare PCX and CUP with the cut-off policies described.

In Table 1 we show the cut-off policy results for a network of 1024 nodes and Poisson $\lambda$ 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 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 $\alpha$, whereas for the higher query rates, the choice of $\alpha$ is less dramatic. These results show that choosing a priori an $\alpha$ value for the linear and logarithmic policies that will perform well across all workloads is difficult.

For the higher query rates, the history-based second-

Table 2: 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 ($\sigma$) | 1.51 (2.77) | 2.67 (3.96) | 4.49 (5.92) | 6.74 (8.25) | 11.01 (12.11) | 17.47 (17.49) | 29.29 (27.79) | 45.56 (40.31) |
| CUP AvgLat ($\sigma$) | 0.21 (1.10) | 0.46 (1.60) | 1.25 (3.19) | 2.17 (4.37) | 4.18 (7.13) | 7.70 (11.28) | 11.48 (15.08) | 19.17 (23.75) |
| IR/CUPOvhd Hop | 4.15 | 4.88 | 6.29 | 7.83 | 11.43 | 16.14 | 24.85 | 35.98 |

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 policy achieves a total cost very near the optimal push level total cost. In all remaining experiments, we use second-chance as the cut-off policy.

## 4.5 Scaling the Network

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

Table 2 compares CUP and PCX for network sizes between $2^7 = 128$ and $2^{14} = 16384$ nodes for a Poisson $\lambda$ 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 is a substantial reduction in average query latency that improves with increasing network size. Comparing the standard deviations of CUP and PCX we see that CUP also has less variability around its average query latency.

The fourth row in Table 2 shows the IR per overhead push performed by CUP. We observe a growth in the rate of return with 16.14, 24.85, and 35.98 for the last three network sizes. These numbers are quite strong, considering that the overhead is completely recovered.

Figure 3 shows the IR of CUP versus network size for Poisson with $\lambda$ = 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.

## 4.6 Varying the Network Topology

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

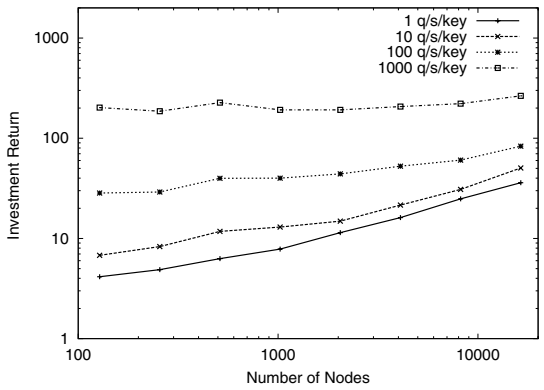


Figure 3: IR vs. net size. (Log-scale axes.)

and the network has smaller diameter. Smaller diameter means that the average path length of a query on a miss is shorter for both PCX and CUP, 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 $\lambda$ rate of 1, 10, 100, and 1000 queries per second. 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 is at least 2.1 for 1 q/s and increases to 36.6 for 1000 q/s.

## 4.7 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 capac-

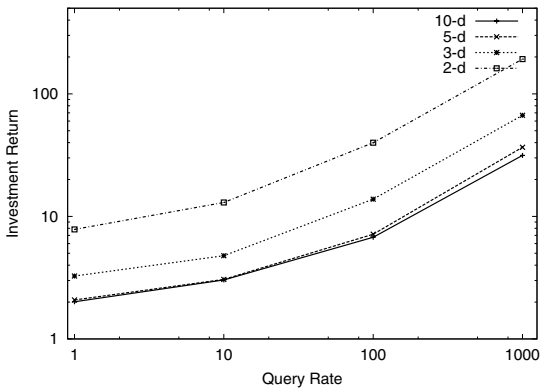


Figure 4: IR vs. query rate, varying dimensions. (Log-scale axes.)

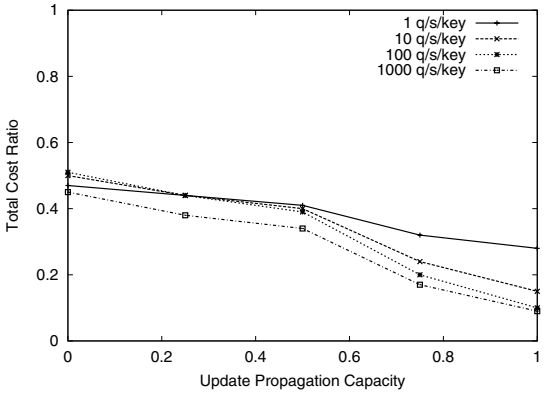


Figure 5: Total cost ratio vs. update propagation capacity

ity 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 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 capacity $c$ for this experiment and for four different Poisson query rates $\lambda$. The capacity $c$ ranges from 0, implying that no updates are propagated, to 1, where 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.

## 4.8 Pareto Query Arrivals

Recent work has observed that in some peer-to-peer networks, query inter-arrivals exhibit burstiness on several time scales [Mar02], 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 $\alpha$ and $\kappa$ values but can only present representative results here. Table 3 compares CUP with PCX for $\alpha$ equal to 1.25 and 1.1 respectively for a network of 1024 nodes. We set the value of $\kappa$ 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 $\lambda$ rate of the Poisson experiments in previous sections.

As $\alpha$ 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., it is a 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

Table 3: 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 ($\sigma$) | 7.77 (9.28) | 6.99 (9.43) | 3.84 (8.41) | 4.01 (8.75) | 1.75 (5.88) | 1.61 (5.53) | 1.00 (4.02) | 1.10 (4.16) |
| CUP AvgLat ($\sigma$) | 3.16 (5.75) | 1.71 (4.44) | 0.42 (3.03) | 0.37 (2.80) | 0.13 (1.66) | 0.15 (1.71) | 0.08 (1.17) | 0.09 (1.24) |
| IR/CUPOvhd Hop | 6.41 | 7.49 | 13.09 | 16.03 | 43.25 | 53.57 | 223.97 | 293.30 |

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 burstiness. 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.

## 4.9 Zipf-like Key Distributions

A recent study has shown that queries for multiple keys in a peer-to-peer network follow a Zipf-like distribution, with a small portion of the keys getting the most queries [Sri01]. That is, the number of queries received by the *i'th* most popular key is proportional to $\frac{1}{i^\alpha}$ for constant $\alpha$.

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 4 shows results for Poisson arrivals where the overall $\lambda$ rates are 100, 1000, 10000, and 100000 queries per second. (We also ran experiments with $\alpha$ = 0.80 and 2.40 and with Pareto arrivals, and the results were similar.)

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 rare 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 unrecovered cost of the unpopular 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.

# 5 Related Work

We describe related work specifically in the peer-to-peer literature, followed by related work in the systems literature in general.

## 5.1 Related Peer-to-Peer 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 advocate caching index entries to improve performance [gnu, RFH+01, SMK+01, RD01a], there has been little follow-up work studying when and where to cache entries and how to maintain these cached entries in a peer-to-peer system.

Cox et al. [CMM02] study providing DNS service over a peer-to-peer network as an alternative to traditional DNS. They cache index entries, which are DNS mappings, along search query paths. Similarly, the TerraDir Distributed Directory caching scheme [SBK02] has nodes along the search query path cache pointers to other nodes previously traversed by the query. 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 [CSWH00], CFS [DKK+01], PAST [RD01b], and Lv et al. [LCC+02] each perform path caching, or caching 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 implicitly prevent 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 [RKCD01] and Bayeaux [ZZJ+01]. Scribe is a publish-subscribe infrastructure built on top of Pastry [RD01a] 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 benefit from our CUP ideas to provide update propagation for cache maintenance in Pastry.

Table 4: Cross-Key Comparison of CUP with PCX, for Poisson arrivals and Zipf-like key distribution

| Overall AvgRate q/s | 100 | 1000 | 10000 | 100000 |
|---|---|---|---|---|
| CUP/PCX MissCost | 0.45 | 0.23 | 0.10 | 0.08 |
| PCX AvgLat ($\sigma$) | 10.6 (9.9) | 6.9 (8.9) | 3.4 (7.5) | 1.53 (5.47) |
| CUP AvgLat ($\sigma$) | 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 |

## 5.2 Related Distributed Caching Work

DNS [Moc87a, Moc87b] is the largest and best known distributed directory service for the Internet. Name servers, like CUP nodes, can be viewed as distributed caches that hold index entries (DNS name-to-IP address mappings) with Time-to-Live (TTL) fields indicating how long they should be considered valid. The maintenance of DNS caches has typically been *pull-driven*, where name servers either pull a fresh version of a stale cached mapping in response to a client request, or proactively, in anticipation of a request [CK01b]. CUP maintains caches through a *proactive push-driven* approach, where updates are pushed to all interested nodes in the overlay network. DNS is generally intended to support slowly-changing mappings with TTLs on the order of hours (e.g., 24 hours) [CK01b], whereas CUP is geared toward maintaining caches of metadata that change frequently, on the order of minutes.

Distributed caching techniques have been looked at in the context of distributed file systems (e.g., [HO93, ADN+95], where the focus is on achieving cache coherence amongst groups of participating file writers that have cached files and communicate over a local-area network. CUP is designed for peer-to-peer environments, where there may be thousands of participating nodes spread across the Internet, and where updates for a particular metadata item are typically generated by only one peer node.

Distributed caching techniques have also been looked at in the context of web caching. Many previous studies have focused on cache replacement policies since cache size becomes a finite source when caching content for potentially thousands of clients [Mog96, WAS+96]. In CUP, cache size is not an issue since metadata are small.

Data caching and movement techniques based on economic models of locally computed interest have been studied in the context of the Mariposa Distributed Database Management System [SDK+94]. Mariposa builds a market-based system with a virtual currency where servers advertise prices to provide resources such as CPU cycles and storage services for query processing such that they maximize their local revenue income per time unit. If a server is underutilized, it will lower the price of its resources to attract more requests. In CUP, the notion of economic benefit is different; a node that derives benefit by propagating an update is saving itself from future work (query requests).

Many schemes have been proposed for the maintenance of cached web content. Some propose push-based invalidation schemes where a web server/proxy notifies proxies/clients when cached objects are modified (e.g., [LC99]), pull-based validation schemes, where the proxy/client validates with the server/proxy cached objects that have expired [CK01c], and hybrid schemes, where the server piggybacks validations on responses to requests for related objects (e.g., [KW97]. CUP differs from previous web maintenance schemes by using push-driven propagation that is driven by the individual economic incentive of participating nodes.

Cooperative caching has been proposed to allow groups of participating caches to exchange cached web content amongst themselves. The overall goal is to bring a particular web object to the cache that is closest to the clients requesting that web object. Previous proposals include hierarchical cache schemes (e.g., [CDN+96, KLL+97, squ, CK01a]), hash-based schemes [KLL+97, VR98], directory-based schemes [FCAB98, MIB98, TDVK99], and multicast-based schemes (e.g., [Tou98]). Of these cooperative caching studies, those most related to CUP are work on refreshment policies for cascaded caches by Cohen et al. [CK01a] and work on distributing location hints across a hierarchy of caches by Tewari et al. [TDVK99].

Cohen and Kaplan study the effect that aging through cascaded caches has on the miss rates of web client caches [CK01a]. 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 cache; instead, the client cache 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. 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, whenever a parent cache gets a refresh so does the interested child node. In such situations, we find the miss rate at the child node actually

improves.

Tewari et al. [TDVK99] cache location hints in addition to web content at web caches in a web cache hierarchy. Location hints are used by requesting leaf caches to access copies of web content directly from remote caches holding the content, rather than waiting for the content to travel through the root and down to them. Propagation of hint updates is considered inexpensive, and occurs proactively and independently of the request pattern of the web object the hint represents. CUP emphasizes *recovering* propagation overhead. CUP makes the propagation decision by comparing the cost of propagating a particular update with the benefit (investment return) the update will bring to the tree below the node. CUP only propagates updates that are likely to benefit subsequent queries in the subtree below.

## 6 Conclusions

CUP provides a general purpose framework for maintaining caches of metadata in peer-to-peer networks, where continuous updates are expected, yet nodes must have personal economic incentive to participate in the maintenance. CUP is a complete protocol with query channels for coalescing bursts of queries and update channels for asynchronous delivery of query responses and updates of cached metadata. To moderate propagation without imposing a global policy, CUP introduces the notion of *investment return* for motivating each node to participate in the update propagation and policies for estimating when the benefit ceases to outweigh the overhead. For the case of locating content in a peer-to-peer network, we find that CUP secures an investment return of 2 to 300 times the propagation cost and significantly reduces query latency.

We have leveraged the CUP protocol to deliver metadata required for effective load-balancing of content downloads across multiple replica nodes [Rou02]. As with regular searches, the economic incentive-based model helps to moderate and control the amount of metadata update propagation in a highly dynamic environment where load information changes very rapidly. Future work includes the use of 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.

## 7 Acknowledgments

This research is supported by the Stanford Networking Research Center, and by DARPA (contract N66001-00-C-8015).

We thank Brian Noble, our paper shepherd, for his guidance. The work presented here has benefitted greatly from discussions with Petros Maniatis, Armando Fox, Nick McKeown, and Rajeev Motwani. We thank them for their invaluable feedback.

## References

[ADN+95] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. In *SOSP*, 1995.

[CDN+96] A. Chankhunthood, P.B. Danzig, C. Neerdaels, M.F. Schwartz, and K.J. Worrell. A Hierarhical Internet Object Cache. In *USENIX*, January 1996.

[CK01a] E. Cohen and H. Kaplan. Aging Through Cascaded Caches: Performance Issues in the Distribution of Web Content. In *Sigcomm*, 2001.

[CK01b] E. Cohen and H. Kaplan. Proactive caching of DNS records: Addressing a performance bottleneck. In *SAINT*, 2001.

[CK01c] E. Cohen and H. Kaplan. Refreshment Policies for Web Content Caches. In *Infocom*, 2001.

[CLL02] J. Chu, K. Labonte, and B. N. Levine. Availability and Locality Measurements of Peer-to-Peer File Systems. In *Proc. ITCom: Scalability and Traffic Control in IP Networks II Conferences*, July 2002.

[CMM02] R. Cox, A. Muthitacharoen, and R. T. Morris. Serving DNS using a Peer-to-Peer Lookup Service. In *IPTPS*, March 2002.

[CRSB02] Y. Chawathe, S. Ratnasamy, S. Shenker, and L. Breslau. Can Heterogeneity Make Gnutella Scale? May 2002. http://research.att.com/ yatin/publications/.

[CSWH00] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *DIAU*, July 2000.

[DKK+01] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area Cooperative Storage with CFS. In *SOSP*, 2001.

[FCAB98] L. Fan, P. Cao, J. Almeida, and A.Z. Broder. Summary Cache: A scalable Wide-area Web Cache Sharing Protocol. In *SIGCOMM*, 1998.

[gnu] The Gnutella Protocol Specification v0.4. http://gnutella.wego.com/.

[HO93] J. Hartman and J. Ousterhout. The Zebra Striped Network File System. In *SOSP*, 1993.

[KLL+97] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC*, 1997.

[KW97] B. Krishnamurthy and C.E. Wills. Study of Piggyback Cache Validation for Proxy Caches in the World Wide Web. In *USITS*, 1997.

[LC99] Dan Li and David Cheriton. Scalable Web Caching of Frequently Updated Objects Using Reliable Multicast. In *USITS*, 1999.

[LCC+02] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and Replication in Unstructured P2P Networks. In *ICS*, 2002.

[Mar02] 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.

[MGB01] P. Maniatis, T.J. Giuli, and M. Baker. Enabling the Long-Term Archival of Signed Documents through Time Stamping. Technical Report cs.DC/0106058, Stanford University, June 2001.

[MIB98] J.M. Menaud, V. Issarny, and M. Banatre. A New Protocol for Efficient Transversal Web Caching. In *Symposium on Distributed Computing*, 1998.

[Moc87a] P. Mockapetris. Domain names - Concept and Facilities. In *RFC 1034*, 1987.

[Moc87b] P. Mockapetris. Domain names - Implementation and Practice. In *RFC 1035*, 1987.

[Mog96] J. Mogul. Hinted Caching in the Web. In *SIGOPS*, 1996.

[RD01a] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *MiddleWare*, November 2001.

[RD01b] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility". In *SOSP*, October 2001.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Sigcomm*, 2001.

[RKCD01] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *NGC*, 2001.

[Rou02] M. Roussopoulos. *Controlled Update Propagation in Peer-to-Peer Networks*. PhD thesis, Stanford University, 2002.

[SBK02] B. Silaghi, B. Bhattacharjee, and P. Keleher. Routing in the TerraDir Directory Service, 2002. http://motefs.cs.umd.edu/terradir/.

[SDK+94] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staeline. An Economic Paradigm for Query Processing and Data Migration in Mariposa. In *3rd International Conference on Parallel and Distributed Information Systems*, 1994.

[SGG02] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *MMCN*, 2002.

[SMK+01] 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.

[squ] Squid Internet Object Cache. http://squid.nlanr.net.

[Sri01] K. Sripanidkulchai. The Popularity of Gnutella Queries and its Implication on Scalability, February 2001. http://www-2.cs.cmu.edu/ kunwadee/research/p2p/gnutella.html.

[TDVK99] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design Considerations for Distributed Caching on the Internet. In *ICDCS*, 1999.

[Tou98] J. Touch. The LSAM Proxy Cache - A Multicast Distributed Virtual Cache. In *3rd WWW Caching Workshop*, June 1998.

[VR98] V. Valloppilli and K.W. Ross. Cache Array Routing Protocol v1.0 (Work in Progress), February 1998. ftp://ftp.isi.edu/internet-drafts/draft-vinod-carp-v1-02.txt.

[WAS+96] S. Williams, M. Abrams, C.R. Standbridge, G. Abdulla, and E.A. Fox. Removal Policies in Network Caches for World-Wide Web Document. In *Sigcomm*, 1996.

[ZKJ01] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, U. C. Berkeley, April 2001.

[ZZJ+01] S.Q. Zhuang, B.Y. Zhao, A.D. Joseph, R.H. Katz, and J. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. In *NOSSDAV*, June 2001.