Breadcrumbs: efficient, best-effort content location in cache networks

Elisha J. Rosensweig
Dept. of Computer Science
University of Massachusetts, Amherst, USA

Jim Kurose
Dept. of Computer Science
University of Massachusetts, Amherst, USA

***Abstract*—For several years, web caching has been used to meet the ever increasing Web access loads. A fundamental capability of all such systems is that of inter-cache coordination, which can be divided into two main types: explicit and implicit coordination. While the former allows for greater control over resource allocation, the latter does not suffer from the additional communication overhead needed for coordination.**

In this paper, we consider a network in which each router has a local cache that caches files passing through it. By additionally storing minimal information regarding caching history, we develop a simple content caching, location, and routing systems that adopts an implicit, transparent, and best-effort approach towards caching. Though only best effort, the policy outperforms classic policies that allow explicit coordination between caches.

I. Introduction

For several years, web caching has been used to meet the ever increasing Web access loads [1]. More recently, advocates of content-centric networking [3], [2] have argued for raising the level of abstraction of the atomic unit of data that is stored and forwarded within the network from the packet to a file, or other higher-level content unit. In both cases, content storage, location, and forwarding within the network are of central concern.

Although content storage (caching) systems come in many forms and flavors, one fundamental capability of all such systems is that of coordination, which can be divided into two main types: explicit and implicit. With explicit coordination, caches share their state (or state summaries), and additional information such as access patterns and content popularity [4] with each other. Using this information, each cache determines what to cache, when to do so, and what to drop. The main cost of such explicit schemes is the additional communication overhead needed for coordination as well as coordination algorithms that can be quite complex and sophisticated.

Implicit coordination, on the other hand, removes the need for such elaborate reporting protocols. Instead, it relies on the local cache management policies [5], as well as the relative position of each cache in the network [6], to achieve good performance. An example of such implicit coordination are hierarchical cache systems [7], where caches are arranged in a tree-like structure. Requests start out at the leaves of the tree, and are routed towards the root until the content is found, either in the tree or at an external source (via the root) when the content is not present in the tree. In such systems, it has been shown that caches lower down the tree tend to contain files that are used more frequently, while upper levels hold files with lower access rates [6]. Here, we see that the architecture provides for implicit coordination to take place between caches, causing them to contain different types of files and so manage resources efficiently.

In this paper, we describe a simple content caching, location, and routing system that adopts an implicit, transparent, and *best-effort* approach towards caching. We consider a network scenario in which each router has a local cache that caches files passing through it. Requests for a file are routed directly towards the source of the file, and en-route check at each router if a copy of the file is present at its cache, and download it directly from there if found. Such caches are commonly referred to as Transparent En-Route Caches (TERC) - ”transparent” as neither the user nor the server are aware that any such cache exists, and ”en-route” since they are accessed during a standard request, on the path to the server [8][9]. The Breadcrumbs approach described in this paper is “best effort” in that coordination is implicit, and forwarded requests may (or may not) locate content while being routed among the caches; if not located, content can always be eventually retrieved from the source. Only a minimal amount of per-file information (which we refer to as a “breadcrumb”) is used in locating content. A breadcrumb stores the direction in which a file was sent in the past, thus tying content routing with content location and caching. We find that although our system promises best-effort only, it performs well even when compared to several classic, more stateful, explicit-cooperation cache systems.

The main contributions of this paper are:

- Presentation of our Breadcrumbs system, that controls query forwarding, cache replacement, and file routing.
- We develop a best-effort policy designed for forwarding queries in search of cached content, and demonstrate its utility for several sample network models.
- We compare the performance of our best-effort policy with other policies, including those systems that are more stateful. We find that our policy performs extremely well in comparison, locating cached content more frequently than policies that use explicit coordination between neighboring caches to control which files are cached locally or dropped.
- We present a preliminary analysis of the impact of employing our forwarding policy on the incoming query

traffic at a specific node. Specifically, we observe that neighbors of a node that experiences an increase in requests for a specific file, will tend to increase the rate of queries for this file that they forward to this node. Thus we get a sense of the nature of the implicit coordination taking place between caches in such a network.

The rest of the paper is structured as follows. In Section II we discuss related work with regards to caching systems in general and, specifically, global ubiquitous caching. In section III we describe at length the Breadcrumbs system, devise a simple Best-Effort Content Search (BECONS) policy, that allows for content to be located in a general caching network, and present a concrete example of this policy. This example helps point out several key behavioral factors of cache netwroks when using BECONS in a Breadcrumbs system, specifically its tendency to forward incoming queries at several nodes to a single node, increasing the chance of retaining the file there as long as it continues to be popular.

In section IV we take a look at the reaction of a the neighbors of a cache in a Breadcrumbs system using BECONS. We classify the trend of changes in the cache miss rate at the neighboring caches and, as a result, the effects on the original cache itself. We present here patterns as they emerge in small-scale systems and try to extrapolate from these to the behavior of a general network. More rigorous analysis is left for future work. Section V presents simulation results, that evaluate the performance of BECONS and compare it to several, both explicitly-coordinated and implicitly-coordinated, cache networks. We demonstrate here that cached information is efficiently located using BECONS, reducing the load experienced by servers and the delay experienced by users requesting content. Finally, section VI presents possible extensions to the models presented in our work and some concluding remarks.

II. Related Work

A. Ubiquitous global caching systems

There have been several attempts at developing ubiquitous global caching systems [5][10][11][12]. Content Delivery Networks (CDN), such as Akamai [14], use a coordinated set of caches throughout the internet to supply content in an efficient manner. Content is pushed to locations where it is requested or, more importantly, predicted to be requested in the future. The managing of the content is primarily done in a centralized manner, and clients are ensured a baseline performance or QoS (Quality of Service).

Distributed caching systems, in which no centralized authority manages the distribution of data, have also been discussed. The OceanStore project [12], to use a common example, is designed to ensure ”global persistent data store” [15] by having users contribute storage space to the system, and use redundancy to ensure content is almost never unavailable or lost. Our work here aims at best-effort caching instead, accepting that content might, in theory, be unavailable within the caching network, but in practice is usually available.

In our work here, we address an architecture in which caching takes place in the network itself, and cache management is done in a distributed manner. The basic concept is to use something similar to Transparent En-Route Caches, in which content is stored at caches associated with routers, and requests for files check the contents of these caches en-route to a public server, which is a known source for the content. Most of the research regarding such caches has focused on how to place a small number of these in an efficient way [8] and where to cache specific objects [4], and has generally addressed the question in a limited number of topologies. Our work here assumes that all routers have been equipped with a TREC-like cache, and that content is cached blindly as it passes through.

Our work focuses on a similar framework to that discussed extensively in [10][5]. Here, the authors attempt to solve the problem of efficient cache replacement by introducing an adaptive caching system named ACME. ACME uses machine learning techniques in order to determine when and what to cache locally, without explicit communication between caches. It uses a pool of virtual caches, each managed by a different static policy, that simulate the behavior of the cache had it been using a specific static cache replacement policy. ACME assigns weights to virtual caches and uses machine learning algorithms to select the best current policies from the virtual cache pool. In such a manner, ACME achieves improved performance compared to any specific static policy. BECON and the Breadcrumbs system differ from ACME in that we use intelligent *query routing*, instead of adaptive caching, in order to improve performance. In this sense, the two architectures are orthogonal to one another, and it is possible that combining them would be advantageous. Such a task is beyond the scope of this work.

B. Heterogeneous caching

It has been shown [16][6] that, in 2-level hierarchical cache systems, performance can be improved by using different cache replacement techniques at different levels. Using different policies at different levels is a form of implicit cache coordination, such that a cache at a lower level skews the patterns of incoming traffic in such a way that the resulting traffic pattern allows a more efficient caching to take place at the next level. This observation becomes less useful, however, when considering the behavior of caches distributed throughout the network, where a clear hierarchy does not exist. In such systems, two neighboring nodes may reverse positions in the hierarchy with respect to different sources. Specifically, if nodes A and B are placed somewhere along the route between servers S_1 and S_2, then A will be higher up in the hierarchy than B with respect to (wlog) S_1 and lower down with respect to S_2. Thus, in such cases, there is no *static* policy that can be assigned to each cache that can enjoy the advantages of heterogeneous caching. In [10][5] adaptive caching is used to solve this problem. In our work, we assume that all caches are using the same cache replacement policy (e.g. LRU, FIFO), and improve performance by employing other tools.

III. Breadcrumbs, and where they lead

A. Basic architecture

We envision a caching network where each node - a router with an associated cache - sets aside some of its cache space for the purpose of storing routing history, or breadcrumbs (BC), of previously seen files. Each BC is a $4-tuple$ entry, indexed by a global file ID (FID), containing the following information:

- Time when the file past through the node. Only the most recent such event is recorded.
- ID of node from which the file arrived.
- ID of node to which the file was forwarded.

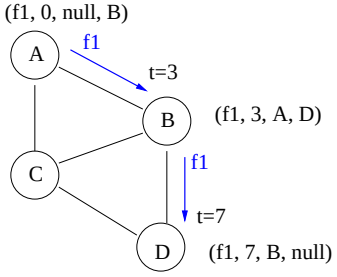

Fig. 1. Breadcrumbs example

In the simple case portrayed in Figure 1, file f_1 was sent along the route $A-B-D$ and then delivered to its destination. The source (server) and destination (user) of the file do not use such caches, and so are ignored by entering *null* for entries that point to these non-router locations. Thus, as the file is downloaded, it leave behind it a *trail of breadcrumbs* at the caches along its download path. As a BC requires very little in terms of storage, we assume throughout the paper that a BC for each file can be maintained at each cache indefinitely.

To begin our discussion of the breadcrumbs architecture, lets consider a request for a file. This request/query will be routed towards the server at which that file is known to always reside. As this request is routed towards the server, it may encounter a router with a breadcrumb for that file, thus *intercepting* a trail of breadcrumbs for that file. At this point, the query could be satisfied locally, if the file is contained in the intercepting routers cache. Alternatively, the query can be routed up or down the breadcrumb trail in an attempt to locate the file. Note that the file can always be found by following the breadcrumb trail upstream, but that the file may be also be found (possible more quickly) downstream, as discussed below. We also note that a similar notion of routing towards a source, but then exploiting state found at an intercepting node is used in multicast tree construction in core-based multicast routing trees [13].

The obvious question that arises from this architecture is in which direction to follow a trail: *upstream*, towards the last origin of the file, or *downstream*, in the same direction as the file was last sent. In the example presented in Fig. 1, a query arriving at node B could be routed upstream to node A or downstream to node D. In what follows we focus on the scenario where the file originated from some public server, and the trail was discovered by first routing towards the source. In such cases, heading upstream is equivalent to continue heading towards the server, where the file is always available for download.

Cache replacement schemes play a large part in determining where queries should be routed. Specifically, if recently cached or referenced items remain in the cache longer than older items, as is the case with the standard LRU cache replacement policy and others, routing a query downstream increases the chance of finding a cached copy compared to routing upstream. We assume that these policies are being used for cache replacement for the rest of the paper, as they are commonly used in practice, and discuss other types of cache replacement policies at section VI. On the other hand, since f's server is upstream, routing in that direction will definitely result in downloading the file, while a downstream search might reach the end of the trail without finding a cached copy. In such an event, the query will need to be re-routed back upstream in search of the file, lengthening the download time.

Consequently, a reasonable first step in devising a content search algorithm would be to define a threshold value T_f, and send queries downstream if-and-only-if the file was last cached and forwarded within the last T_f time units; Otherwise continue searching upstream. For lower values of T_f, queries will have a higher chance of finding the file downstream, but less queries will be taking advantage of possible copies cached there. Conversely, for higher values of T_f more queries will be sent downstream, but a larger fraction of them will reach a dead end.

A more complete policy can be reached, however, once we consider the effects of queries that download the content from a downstream cache. Returning to Figure 1, imagine a series of queries sent down by node B to node D, where a cached copy is found. After T_f time has passed since the file was last cached at B, it will cease to forward queries downstream, even though the file might still be cached there. More importantly, since during this period queries were sent to node D, it is expected that they refreshed the existing copies at D or its descendants, extending the period during which a copy is cached there. Thus, it would be advantageous to continue forwarding queries downstream as long as queries are sent there at a high rate.

Based on these observations, we propose the following **B**est **E**ffort **CON**tent **S**earch (BECONS) query routing policy. Let c be some cache-node, and assume a query q_f arrived at time t, discovering that f is not present at c. Then, for some set of values T_f, T_{q_f}, node c forwards q_f downstream if-and-only-if

1) File f was cached or refreshed (via successful query) at c within $[t-T_f, t]$; or
2) A q_f query passed through c within $[t-T_{q_f}, t]$ and sent downstream.

This policy does not involve any explicit communication between neighboring caches. However, T_f and T_{q_f} can be chosen in several ways. They can be identical for all files or tailored separately for each, and can change in response to

traffic fluctuations and network topology as well. Additional factors, other than elapsed time, can be incorporated into computing the utility of looking upstream and downstream, and different nodes might select different threshold values.

B. S-BECONS: description and analysis

In this section we present Simple BECONS (S-BECONS), a specific instance of the general BECONS policy, and analyze two of its useful properties that make it an attractive candidate for a query forwarding policy: *trail stability* and *trail invalidation*. Let $c_1, ..., c_n$ be a downstream trail and assume a query has begun its search downstream at time $t = 0$.

Definition 1: A BC is said to be *valid* if it is being used to forward unanswered queries.

As mentioned in the previous section, a BC is generated at a node when a file f passes through the node, and remains in use (i.e. valid) with BECONS policy as long as queries continue to arrive at the node no more than T_{q_f} time units apart. A node becomes *invalid* when the interval between queries is longer than this, in which case the BC *times out*, or when the node determines somehow that the file is not present downstream anymore.

Definition 2: A trail $c_1, ..., c_n$ is said to be *broken* if there exist indices $1 < i < j < k < n$ s.t. the breadcrumbs at c_i and c_k are valid while the breadcrumb at c_j in invalid.

Definition 3: The trail $c_1, ..., c_j$ is said to be *stable* if it does not become broken *during* a download search.

A query starting a search along a stable downstream trail will therefore end its search if one of two things occurs: either it has found a cached copy of the file being requested, or it will reach an invalid BC and be rerouted back upstream, towards the source. Since the trail is not broken at this time, all BCs lower down are also known to be invalid.

Definition 4: A policy is said to have *trail invalidation* built in to it, if there is a way in which c_i $(1 \leq i \leq n)$ can determine that the entire downstream trail (starting from it) does not contain a copy of the file.

Surprisingly, it turns out that this property is readily available to a simple instance of BECONS, without the use of any explicit communication between caches. We prove this below.

Let us look at the following instance of BECONS, termed S-BECONS (Simple BECONS). In this policy , each file has a pair of system-wide constants, T_f and T_{q_f}. That is, for each file these values are used at *all* the nodes. We focus here on a single file, so we use T_q instead of the more cumbersome T_{q_f}. Queries are forwarded downstream when they arrive within T_f and T_q of the last file or query (respectively) seen at the node; if a BC at a node becomes invalid, queries of that type are rerouted towards the source. We require that $T_f \geq T_q$, since a new file refreshes the presence of a file along an entire downstream trail for certain, whereas a new query may or may not refresh some nodes. Finally, we make the following assumptions about the network properties:

- The propogation and queuing delay at links and routers (respectively) are constant.
- Let h_f be the delay associated with sending a file a single hop, and h_q the delay associated with forwarding a query one hop and checking the content of a cache. Then $h_f \geq h_q$. This is a reasonable assumption, as files are assumed to be much larger than a query.

Before we continue to analyze the behavior of S-BECONS, a note about following downstream trails is in order. A trail is created at download time by the file being cached along a specific route. As the file gets flushed out of some nodes, the breadcrumbs remain to point the direction of downstream search. However, when a query is forwarded downstream, the path it takes may traverse several different trails, as some nodes downstream might have been refreshed recently. We refer the reader to Figure 2. We see here two original trails that were created independently: $ABXC$ and later $DEXF$. A query heading down the trail starting at node A will then follow the path $A - B - X - F$, taking the freshest direction at each node. This can only increase the probability of finding the file lower down, as only the fresher trail is followed.

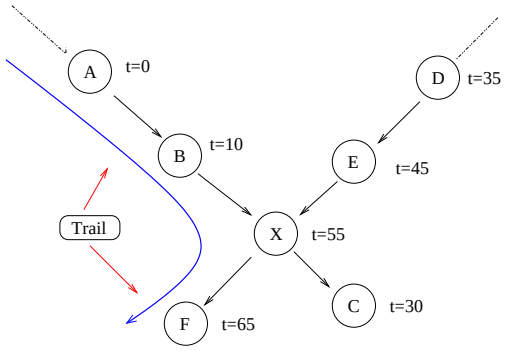

Fig. 2. **Trail intersection.** The arrows denote the direction the file was download in the past when each trail was created ($h_f = 10$). The freshest trail of breadcrumbs might be a concatenation of several subsections of download paths. As trail $DEXF$ is fresher than $ABXC$, a query heading downstream from node A will follow the path $ABXF$.

Based on the assumptions mentioned above, we prove the following two claims.

Theorem 1: S-BECONS can determine trail invalidation if a query q_f arrives at c_1 from c_2.

Proof: We prove this by way of induction on the length of the trail. For the base case of a single link (2 nodes), if c_2 sends a query upstream to c_1, this means that the file is not cached at c_2, and so the trail can be invalidated.

For the induction step, assume that the claim has been proven for a trail of length $k - 1$ and now we prove it for the case of length k. c_2 forwarded a query upstream to c_1, so obviously c_2 does not contain the file. In addition, the query was forwarded upstream instead of downstream, so the BC at c_2 is invalid. This can be for one of two reasons:

- The BC at c_2 has timed out, since no file or query passed through within the required threshhold times. This, however, is not possible, since we are assuming that the BC at c_1 is still valid. This BC is older than the one at c_2 by at least h_q, and since it has not timed out neither could the one at c_2.

- Node c_2 has determined that the trail $c_3...c_n$ is invalid, based on the induction step.

Thus, based on the induction step we know that the file was not cached along the trail when the query traversed it, and so the entire trail can be invalidated. ■

Theorem 2: The downstream trail of a S-BECONS breadcrumb trail is stable.

Proof: Let q_f be a query, and $t = 0$ be the time at which it began its search downstream. At time $t = 0$, we know that c_1 had a valid breadcrumb. We treat the two possible causes for this:

- A file passed through c_1 within the last T_f time, the earliest time being $t = -T_f$. Thus, the earliest time for which a BC will timeout at node $j > 1$ is $-T_f + (j - 1)h_F + T_f = (j-1)h_f$. On the other hand, for all $j > 1$ that q_f reaches, the time will be $t_j = (j - i)h_q$. Since we assume $h_f \geq h_q$, the query passes before timeout can occur.
- An older query q_f^* passed through c_1 within the last T_q time, the earliest time being $t = -T_q$. If q_f^* did not locate the file until node j, then using the same technique as before we know that q_f^* reached node j no earlier than $(j-1)h_q - T_q$, and the breadcrumb will timeout not before $(j - 1)h_q$, by which time q_f will reach node j.
  Otherwise, the file was located by q_f^* at some node $1 < h < j$, and a fresher trail continued for the next $j - h$ hops. Since we know $T_f > T_q$, this trail will also not timeout until the new query reaches node j.

From all this we know that there can be no breaks in the trail due to timeouts. What is left is to address the case of a break due to other types of invalidations - namely, a query backtracking up the trail. However, as we saw in Theorem 1, if this happens than all the nodes from node j until the end of the trail have been invalidated, so there is no break in the trail. ■

There are many advantages to a forwarding policy that ensures stability and has the capacity for trail-invalidation. To begin with, when a break in a trail occurs, the local information at each node is not sufficient to know that lower downstream there is a section that is still valid, and that the file may be found there. Stability, therefore, ensures that a search downstream will cover all valid breadcrumbs in the trail while searching for the file.

Trail invalidation is a critical tool for an efficient routing policy. Without trail invalidation, a node might continue forwarding queries downstream even though the file is no longer available there. These queries would then experience a large delay, and eventually the file is downloaded from the source. Furthermore, the fact that S-BECONS is able to perform trail invalidation without tagging individual queries and keeping track of all their identities makes the system fast, efficient and practical.

Finally, we observe here the emergence of a *border node* - a node from which point and down all queries are forwarded downstream, and for all other nodes queries are forwarded upstream. This is a result of the stability property - if a breadcrumb at node j has not timed out, the same holds for all nodes lower down. The existence of such a node is important since it means that queries intercepting the trail anywhere lower than the border node will all be forwarded down until the file is located. The cache containing this file will enjoy a considerable increase in the incoming rate of queries q_f, and allow it to keep the file stored for an extended period of time. A policy that does not allow the emergence of border nodes, on the other hand, will not allow such an aggregation of queries to form, increasing the number of cache misses and allowing less coordination between caches overall.

C. File download path

Once a cached copy is discovered, it is downloaded to the user that requested it. For this download, the file may be routed to its destination in two ways:

- **Download Follows Query (DFQ)** - the file backtracks along the route the query took.
- **Download Follows Shortest Path (DFSP)** - the file is sent along the shortest path to the destination.

These download policies have different delays associated with them, but more importantly, they determine the new locations where the file will be re-cached on its way to the destination. These differences are illustrated in Figure 3.

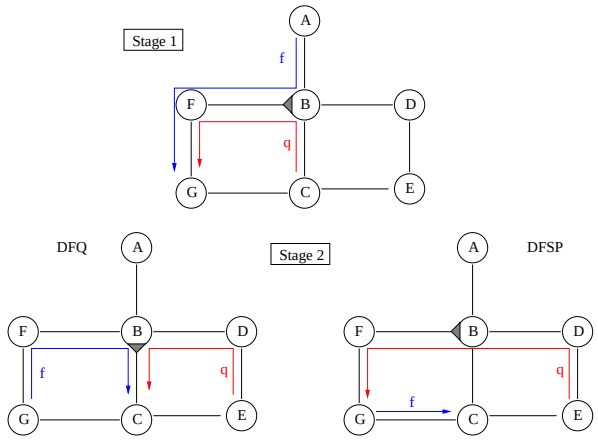

Fig. 3. DFQ vs. DFSP. In each diagram, the file download (blue line) occurs prior to the query (red line). The top diagram shows how the query is routed downstream to find a copy of the file. The bottom two diagrams demonstrate the differences and implications of using different file download policies. The shaded triangle represents the direction in which q will be sent.

DFQ has the file cached at the locations in the hierarchy where the query arrived, and thus prepares the ground for additional queries that might arrive if the file exhibits local popularity. However, in such a case only some of the caches will experience it, since successfully-answered requests at a node will not be forwarded upstream, and so its ancestors will experience a much lower rate of queries of the same type. DFSP, on the other hand, will deliver the file in the shortest possible manner to the user, in the given state of the system, though perhaps place file copies in locations where the file will not be requested at all.

Notation	Meaning
$I_x(k)$	Incoming rate of q_k at node x.
$O_x(k)$	Cache miss rate of q_k at node x.
$+[y]$	An increase of variable y (e.g. $+[I_x(k)]$
$-[y]$	A decrease of variable y (e.g. $-[I_x(k)]$

TABLE I
FREQUENTLY USED NOTATION

Another major implication of the downloading route is the stability of the query forwarding table at the interception point c_0. When employing DFQ, the file f is cached only along nodes through which q_f passed, and specifically c_0. From c_0, f is forwarded to the requested destination using shortest path routing. This causes a change in the query forwarding table at c_0, since this new destination is now the most recent direction in which the file was sent. Thus, next time a cache-miss occurs at c_0, the query will be forwarded along this fresher trail.

DFSP, alternately, ensures a more stable query forwarding table. Since the file is routed along the shortest path, it might not pass through c_0. Node c_0 is then oblivious to what happens downstream, and is only aware that a copy of the file was successfully found. It can thus continue to forward queries downstream in the same direction. As we discussed in the previous section when addressing border nodes, if the flow of queries is high enough, this can ensure with high probability that a copy shall remain cached downstream. This stability of forwarding tables and cache contents downstream can therefore improve the performance of BECONS compared to when using DFQ, as the probability of finding a cached copy downstream increases. This behavior is supported by the simulations we performed (section V).

IV. IMPLICIT LRU CACHE COORDINATION

Our BECONS query forwarding policy modifies the direction in which queries are routed in order to locate cached files in the network. A modification will occur only when the rate of queries q_f is above a certain threshold, as expressed by T_f and T_{q_f}. When such a change in routing takes place, this will cause a sudden increase in queries coming in to nodes downstream. It is not clear, however, how neighboring nodes will react to such an influx. Therefore, given a node x that experiences an increase in queries of type q_f from upstream, we would like to know how the combined rate of q_f at node x is affected, as x's neighboring nodes react to this change at x.

A network cache can be thought of as a query filter, allowing incoming queries to move on to the next hop only when a cache miss occurs. This filter tends to be tighter, and allow a smaller fraction of queries of type q_i to proceed, as q_i takes up a larger part of the incoming query distribution. Formally, assume that the steady-state distribution of arriving queries is $p = (p_1, ..., p_n)$ where p_i is the probability that the next request will be for file f_i, and $\sum_{i=1}^{n} p_i = 1$. If r_i is the rate of such requests, $R = \{r_1, ..., r_n\}$ $r = \sum_{r' \in R} r'$, we get $p_i = r_i/r$. Then, as p_i increases the probability of a cache miss decreases.

The *probability* of a cache miss for of q_i, as a function of the arrival rate of q_j $(j \neq i)$, is monotonic in its behavior. As r_j increases, f_j takes over a cache slot for longer stretches of time and in such a manner forces other files to be dropped more frequently. Using the notation in Table I we can write that for any cache x,

$$+[I_x(i)] \Rightarrow +[O_x(j)] \quad (j \neq i) \tag{1}$$

The same cannot be said, however, for the miss *rate* of q_i as a function of r_i, when the rate of incoming queries of all other files, termed *background traffic*, remain unchanged. At one extreme we know that when $p_i \to 0$, less queries are coming in and so the miss rate decreases as well, as it is bounded by the incoming rate. At the other extreme, as $p_i \to 1$, cache misses become rare and the cache miss rate goes to 0 as well, and no cache misses occur for q_i when $p_i = 1$. Assuming therefore that the background traffic remains the same, the miss rate as a function of the incoming rate has an upper bound.

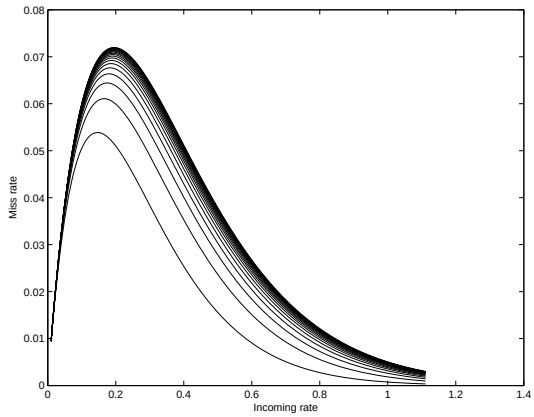

Fig. 4. **LRU miss-rate unimodality**. We plot here the miss rate of a single file as it's incoming rate of queries increases, while the background traffic remains the same. Background traffic normalized to 10, cache size is 50. The peak miss rate increases as the overall number of users in the system grows from 150 to 1050.

Additionally, we conjecture that the miss rate will, in fact, be *unimodal*, experiencing a single peak miss rate and then decreasing monotonically as p_i increases. Support for this can be found in Figure 4, which was produced by simulating the behavior of LRU using the approximation algorithm presented in [17]. In this simulation, we set the combined rate of the background traffic to be constant, and gradually increased the rate of incoming queries for a specific file, plotting its miss rate as a result. The background traffic is distributed uniformly - each file has the same probability of being the next to arrive. The different curves in the figure represent the miss rate for different amounts of files in the system. As the number of files increases, so does the miss rate, since there is greater chance that a new file will arrive at the cache and require some file to be dropped.

To characterize this behavior we say that if $I_x(i)$ is *pre-peak*,

$$+[I_x(i)] \Rightarrow +[O_x(i)]. \quad (2)$$

and otherwise, if $I_x(i)$ is *post-peak*,

$$+[I_x(i)] \Rightarrow -[O_x(i)]. \quad (3)$$

Some details regarding the location of this peak point can be found in Section VI. For our purposes here, we rely solely on the unimodal structure - the existence of such a peak point - in order to characterize the changes in the query rates at the surrounding nodes.

Relationships (1)-(3) are the building blocks of the analysis that follows. We focus our attention in this paper on the 3-node network presented in Figure 5, and leave an extensive discussion of the problems and methodologies presented here to a future work.

First, consider the case of two neighboring nodes, x, y, in the aforementioned figure. Cache misses of q_1 at node x are forwarded to y, and cache misses of q_2 at node y are forwarded to x. We observe the following behavior:

Lemma 1: $+[O_x(1)] \Longleftrightarrow +[O_y(2)]$

Proof: Assuming that an increase occurred in the output of node x (wlog), this will lead to the following series of rate changes:

$$+[O_x(1)] \Rightarrow +[I_y(1)] \Rightarrow_{exp.(1)} \ +[O_y(2)]$$

The increase in the miss rate at node y will have the same effect on node x. ■

This behavior is one of *reciprocation* - if x increases the load on node y, y in return increases the load on node x, though of a different type of query. Reciprocation can also clarify what a new steady-state of the system may look like. As x sends more of the load for q_1, y reacts by sharing some of the load for q_2 with node x in return. The converse tendency can be seen to exist when there is a decrease in miss rates from one node. From this result we get the following property as well:

Corollary 1: $+[I_y(1)] \Longleftrightarrow +[O_x(1)]$

Proof: We've seen that $+[I_y(1)] \Rightarrow +[O_y(2)]$, and Lemma 1 completes the proof. ■

The importance of Corollary 1 is that x increases the rate of q_1 being sent to y as a result of the original increase at y, even though x may have had nothing to do with this original increase. Thus, we observe here an implicit form of coordinated load-balancing between neighboring caches: as node y dedicates more resources to store f_1, some of its neighbors increase the rate of q_1 queries that are sent its way. This increase in queries has the direct effect of reducing cache misses for q_1, allowing y to *specialize* in storing this file. At the same time, node x is free to dedicate more resources for storing other files, such as f_2.

For a more complete understanding of this mechanism, we expand this model to include a third node, z (Fig. (5)). Node z forwards (receives) cache misses of q_2 (q_1) to (from) node y. As we shall see, it is here that the unimodularity of the miss rate comes into play.

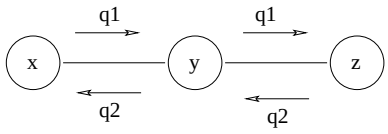

Fig. 5. 3-node cache network

If node y is in pre-peak state for both q_1 and q_2, we get the following sequence:

$$\begin{aligned} +[I_y(1)] \ &\Rightarrow_{ex.(2)} \ +[O_y(1)] \Rightarrow +[I_z(1)] \\ &\Rightarrow_{ex.(1)} \ +[O_z(2)] \Rightarrow +[I_y(2)] \Rightarrow_{ex.(2)} +[O_y(2)] \end{aligned}$$

If node y is in post-peak state for both q_1 and q_2, we get the following sequence:

$$\begin{aligned} +[I_y(1)] \ &\Rightarrow_{ex.(3)} \ -[O_y(1)] \Rightarrow -[I_z(1)] \\ &\Rightarrow_{ex.(1)} \ -[O_z(2)] \Rightarrow -[I_y(2)] \Rightarrow_{ex.(3)} +[O_y(2)] \end{aligned}$$

Both of these cases behave the same from the perspective of node x: an increase in $I_x(2)$ flowing in from y. As we have seen in Lemma 1, this causes an increase in $+[O_x(1)] \Rightarrow +[I_y(1)]$, which is the starting event of the previous two sequences. This means that the traffic coming in from both x and z shifts the system in the same direction: an increase of q_1 at node y.

The third and final case is when q_1 is post-peak while q_2 is pre-peak. Once again, we trace the sequence of changes in the query rates:

$$\begin{aligned} +[I_y(1)] \ &\Rightarrow_{ex.(3)} \ -[O_y(1)] \Rightarrow -[I_z(1)] \\ &\Rightarrow_{ex.(1)} \ -[O_z(2)] \Rightarrow -[I_y(2)] \Rightarrow_{ex.(2)} -[O_y(2)] \end{aligned}$$

As opposed to what we saw in the first two cases, here the traffic coming in from z seems to have two opposing effects on the rate of q_1 at y. On the one hand, $-[O_y(2)] \Rightarrow -[O_x(1)]$, while on the other, an increase in q_1 queries at y is what started the entire process. Based solely on our high-level analysis, we are unable to determine what the new trend of the system will be as a result. We do, however, propose the following conjecture:

Conjecture 1: When q_1 is post-peak and q_2 is pre-peak, the reduction in queries coming in to y from x is smaller than the increase that causes the reduction.

The basic justification for this conjecture is this: Since it is the influx in queries that causes the reduction from x, this reduction will be proportional to the influx.

Based on all the above, we formulate the following two principles that help explain the manner in which neighboring caches react to changes in queries at a specific cache.

Theorem 3: The reaction of the neighboring caches to an increase (decrease) of queries at y depends *only* on the state of these queries at node y, and not on the state at the neighbors.

Proof: From Lemma 1 we know that each neighbor reciprocates the behavior it observes (via queries) at node y.

Thus, the pre-peak and post-peak state of a query at node y is the only thing that determines the manner in which neighbors will respond. ■

This property is a very useful one, as it allows for some insight into the behavior of a cache system, even when only a small portion of it is available, or when analysis of the complete system is intractable.

Theorem 4: In the 3-node network discussed here, if q_1 is past its peak point at node y, an increase of q_1 queries will result in a reduction in the miss rate of q_i.

Proof: As we've shown in the previous section, the result of an increase of q_1 from some source is an overall increase in $I_y(1)$. At the same time, we know that when q_1 is post-peak we get

$$+[I_y(1)] \Rightarrow -[O_z(2)]$$

that is, a reduction in the overall rate of q_2 coming in. Thus, p_1 grows at the expense of p_2. This, in turn, decreases the cache misses w.r.t. q_1. ■

This theorem relies on Conjecture 1. One implication of this theorem is that when a query at some node has past its peak point, an increase in queries at it will greatly stabilize the presence of the file at the cache. In order to pass this peak point, a large volume of queries must be focused on a single cache. As discussed earlier, using BECONS with DFSP file forwarding is designed to achieve just this.

V. Simulations

As part of evaluating the behavior and performance of the Breadcrumbs network, we simulated and compared the behavior of several routing and cache-replacement algorithms. We built an event-driven simulator that generates requests for files at every node and sends them into the network. The requests (queries) are routed within the network until they find the file, either cached or at a source, and then the file is sent towards the requesting node. Both the location of the public servers and the number of files assigned to each were chosen at random (uniformly).

We let all delays in the system be constant. These include query and file propagation delays, queuing delays and download time at server. These assumptions are an approximation of the behavior when the network is not congested. In the future, however, we plan to have our simulation environment expanded to allow load-dependent delays to be incorporated as well. The values used in the simualtions discussed here used the parameters displayed in Table II.

Requests are generated at each node in the network with the identical exponential distribution. Files being requested were selected at random, using the same distribution at each node, for which we chose both uniform and zipf. For every sequence of such events, we simulated the behavior of the entire system using different combinations of routing policies and cache replacement algorithms. We looked at the following:

- Routing to the source, with LRU. This is the simplest policy, and the baseline for performance evaluation.

Parameter	Value	Parameter	Value
Download from source	25	File hop	10
Query hop	1	Cache Access	1
# sources	10	# nodes	100
# files	300		

TABLE II
Simulation parameters

- S-BECONS with LRU. T_f and T_{q_f} were set to be identical for all files. File routing was tested with both DFSP and DFQ.
- Routing to the source, using two types of explicitly-coordinated LRU:
  - a file enters a cache only if it is not located in any direct neighbor, and a file is dropped from the cache using LRU, but choosing the LRU file that is also cached in one of the neighbors, if it exists.
  - a file enters a cache only if it is not located in the neighbor cache en-route to the source of the file being requested. Dropping a file is also done using LRU but selecting from the files that are cached at the next hop, if possible.

We found that DFQ performed approximalty the same as the simple, route-to-source policy, and so we present here our results only with regards to DFSP. In Figure 6 we present the *relative* number of downloads from a source as a function of cache size and policy. We simulated a system with 300 distinct files, when cache sizes are 10, 20, 30 and 40. As can be seen from the results, S-BECONS performs well in comparison to explicitly-cooperative systems, and outperforms them when the cache size is small. Cooperating caches show performance gains mainly due to the fact that a group of caches acts as a single larger cache. However, when the cache size is realtively small compared to the number of different files, as in our framework, these gains are depleted, and finding cached material by following breadcrumbs reduces the load on servers much more.

We analyzed as well the time associated with an average download, and found similar behavior to emerge in the data here as well: for smaller cache sizes, the time per download decreases when employing our S-BECONS policy. These results, however, rely heavily on the system parameters, such as propagation and queueing delay. More realistic and trace-driven simulation are required in order to validate this behavior in real systems. This is left for future work.

VI. Model Extensions and Closing Comments

A. Cache replacement policies

In this work we focused primarily on the LRU cache replacement policy, and in general on policies in which recency is explicitly considered when determining what file to drop when needed. Other than LRU, policies such as FIFO and LFU are included in this category. Here we refer to the implications of our work to systems that employ other types of cache replacement policies.

Policies that do not explicitly reward recency can be divided into two categories: Those that do not consider recency at all, and those that prefer less recent items. The latter are usually not used for cache replacement purposes, but rather as an admission control algorithm, determining which files to cache in the first place [10]. When such admission control is in use, the first nodes past the interception point might not contain the file, but past them the same considerations used in this work should apply.

Static policies that only use other considerations, such as file size or type, can be thought of as implicitly rewarding recency. For example, if caches prioritize large files over small ones, the routing algorithm can compute the probability of finding a cached copy downstream based on its size and the traffic characterization of the network. Here too, recently arrived files will tend to remain longer in the cache compared to files with *the same properties* (e.g. same size). Even when same property files are selected randomly for dropping, the longer a file remains in a cache the more random selections it must survive. Thus, border nodes will emerge in many systems with a variety of cache replacement policies.

B. Peak node characteristics

We noted earlier that the miss rate of a query is unimodal. As can be seen from our analysis thus far, the exact location of this peak rate can be crucial to the behavior of the system as a whole. Preliminary results seem to indicate that, when the background traffic is assumed to be uniformly distributed, the peak miss rate will tend to occur around when $p_i = 1/K$ (Fig. 7), where K is the volume of the cache in terms of number of cachable files.

This approximation is closer to the mark as the number of files in the system grows, since then more new files arrive that cause file drops. Such behavior can be explained by the fact that once the number of queries takes up $1/K$-th of the queries, a query will arrive on average within the time needed to refresh the position of the file in the LRU queue, and thus maintain the file in the cache for a high percentage of time.

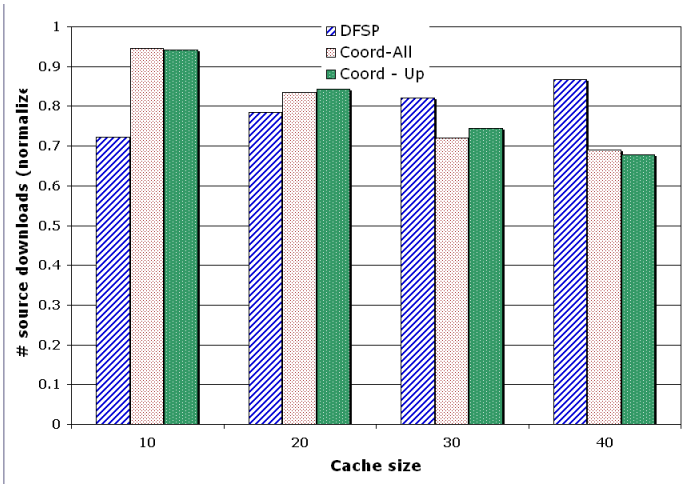

Fig. 6. Relative number of downloads from sources, using DFSP and two types of coordinated LRU caching. All values are normalized by the number of downloads from source experienced when routing to the source with standard LRU. The smaller the value, the less load experienced at the sources.

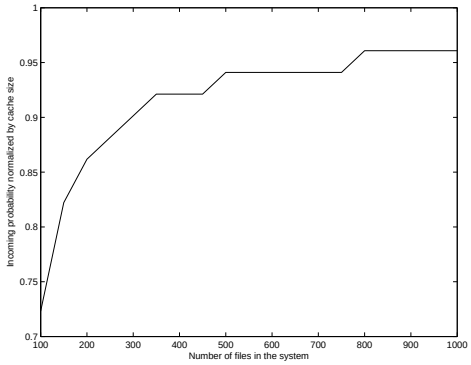

Fig. 7. **Peak miss rate**. Values are normalized by $1/K$. As the number of files in teh system increases, the miss rate reaches its peak when the incoming ratio is in the vicinity of $1/K$.

REFERENCES

[1] A. Datta et. al. *World Wide Wait: A Study of Internet Scalability and Cache-Based Approaches to Alleviate It*, Management Science Volume 49 , Issue 10, October 2003, pp. 1425 - 1444.

[2] D. Raychaudhuri, R. Yates, S. Paul, J. Kurose, "The Cache-and-Forward Network Architecture for Efficient Mobile Content Delivery Services in the Future Interne t, ITU-T Innovations in NGN," ITU-T Innovations in NG,N May 2008.

[3] Van Jacobson, "A New Way to Look at Networking", http://video.google.com/videoplay?docid=-6972678839686672840

[4] X. Tang, S. T. Chanson, *Coordinated En-Route Web Caching* IEEE Transactions on Computers, Vol 51 No. 6, 2002 pp. 595-607.

[5] I. Ari et al *ACME: Adaptive Caching Using Multiple Experts*, Proceedings in Informatics, vol. 14, Carleton Scientic, 2002.

[6] H. Che, Z. Wang, and Y. Tung, *Analysis and Design of Hierarchical Web Caching Systems*, IEEE INFOCOM 2001 pages 1416-1424.

[7] A. Chankhunthod et. al. *A hierarchical Internet object cache*. In Proceedings of the 1996 USENIX Annual Technical Conference, San Diego, CA, 1996.

[8] P. Krishnan, D. Raz, Y. Shavit, *The Cache Location Problem*, IEEE/ACM Transactions on Networking (TON) Volume 8 , Issue 5 (October 2000) Pages: 568 - 582

[9] Y. Jin,W. Qu, K. Li,*A Survey of Cache/Proxy for Transparent Data Replication* Second International Conference on Semantics, Knowledge and Grid, 2006. SKG '06.

[10] I. Ari, *Design And Management Of Globally Distributed Network Caches*, PhD dissertation 2004, http://www.soe.ucsc.edu/ ari/Ari-PhD-Thesis.pdf

[11] S. Bhattacharjee, K. L. Calvert, E.W. Zegura, *Self-organizing wide-area network caches* IEEE INFOCOM 1998 vol. 2 pp. 600-608.

[12] J. Kubiatowicz et. al. *OceanStore: An Architecture for Global-Scale Persistent Storage*, ACM SIGARCH Computer Architecture News Volume 28 , Issue 5 December 2000, pp. 190-201.

[13] A. J. Ballardie, P. F. Francis, and J. Crowcroft (August 1993). "Core Based Trees", *ACM SIGCOMM Computer Communication Review*, 23 (4): 85 95.

[14] http://www.akamai.com.

[15] http://oceanstore.cs.berkeley.edu/info/overview.html

[16] M. Busari and C. Williamson. *Simulation evaluation of a heterogeneous web proxy caching hierarchy*. In IEEE Proceedings of the 9th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 01), pages 379388, Cincinnati, OH, Aug. 2001.

[17] A. Dan, D. Towsley, *An approximate analysis of the LRU and FIFO buffer replacement schemes*, Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems 1990 pp. 143 - 152.