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

Elisha J. Rosensweig
Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003–9264
Email: elisha@cs.umass.edu

Jim Kurose
Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003–9264
Email: kurose@cs.umass.edu

***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 [2], [3] have argued for raising the level of abstraction of the atomic unit of data that is stored and forwarded within the network from a 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. The system topology provides for implicit cache coordination to take place, as the position of each cache will cause it to hold 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 initially towards the source of the file, and en-route check at each router whether 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" in that 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 (termed "breadcrumb") is used in locating content. A breadcrumb stores the most recent direction and time that a file was forwarded 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:

- We describe the architecture of our Breadcrumbs system, and present a best-effort caching policy designed for forwarding queries in search of content, that utilizes this architecture. We demonstrate the utility of this policy for sample network models.
- We compare the performance of our best-effort policy with other policies, including those that are more stateful. We find that our policy performs extremely well in comparison, locating cached content more frequently than

This work was supported in part by the National Science Foundation under CNS-0626874. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation.

policies that use explicit coordination between neighboring caches.
- We present a description and preliminary analysis of the implicit cache coordination that is a result of our query-forwarding policy. Specifically, we show that neighbors of a cache respond to changes in its incoming query distribution in a manner that results in a form of load balancing among neighbors caches.

The rest of the paper is structured as follows. In Section II we discuss related work. In section III we describe 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. In section IV we take a look at the reaction of a cache's neighbors to changes in it's incoming query distribution, and present guidelines for understanding the interaction between these neighboring caches. Section V presents simulation results, that evaluate the performance of BECONS and compare it to several, both explicitly-coordinated and implicitly-coordinated, cache networks.

II. Related Work

There have been several proposals for developing ubiquitous global caching systems [5][10], and the reader is referred to the technical report [12] for a more complete survey of these. 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. Our approach uses 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 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. In our work here we utilize TERCs that have been augmented to store a minimal amount of caching and routing history.

A framework similar to the one we discuss here is discussed extensively in [5]. Here, the authors 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. BECONS 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.

It has been shown [6][13] that, in 2-level hierarchical cache systems, performance can be improved by using different cache replacement techniques at different levels. This observation becomes less useful in our model, as the parent-child structure of the hierarchy changes w.r.t. different nodes in the network [12]. In our work, we assume therefore that all caches are using the commonly chosen LRU policy.

III. Breadcrumbs, and where they lead

A. Basic architecture

We consider 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 $5-tuple$ entry, indexed by a global file ID (FID), containing the following information:

- ID of node from which the file arrived.
- ID of node to which the file was forwarded.
- Most recent time the file passed through the node.
- Most recent time the file was requested at the node.

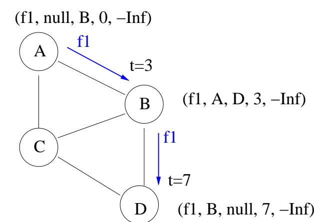

Fig. 1. Breadcrumbs example

In the simple case portrayed in Figure 1, file f_1 was sent from the source to node A and along the route $A-B-D$, and finally delivered to the user that is connected to node D. The source (server) and destination (user) of the file do not use such caches, signified by entering *null* for entries in the tuple that refer to these non-router locations. When the file is first cached, we enter *-Inf* as the most recent request time. Thus, as the file is downloaded, it leaves 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 for now that a BC for each file can be maintained at each cache indefinitely, though in practice they can be dropped once they become invalid, as explained later.

To begin our discussion of the breadcrumbs architecture, let's consider a request for a file. This request will be routed towards the publicly-known source of the file. En-route to the source, 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 router's 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 to the source, but that the file may also be found (possibly faster) downstream, as discussed below. 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 [14].

An 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 what follows we focus on the scenario in which the file originated from some public source, and the trail was discovered by first routing towards the source. Heading upstream is thus equivalent to routing towards the source, where the file is always available.

Cache replacement schemes play a large part in determining where queries should be routed. In this work we assume that the replacement policy is LRU, which rewards frequent requests for a file by keeping it in the cache longer (see [12] for a discussion of other replacement policies). For such policies, nodes downstream have seen the file more recently and thus the file has a higher probability of being located there, but only when routing upstream is there a certainty that the file will be found (perhaps at the source).

A useful content-search algorithm in such a system would take into consideration, at each node, the time that elapsed since two critical events: most recent caching of the file at the node, and most recent request for the file at the node. When a file is cached en-route to some destination, it creates a new trail of cached copies. When a file is requested at the cache and the request is forwarded down a breadcrumb trail, it will refresh the file at the node where it is found (*if* it is found), and extend it's time-to-live in that cache. Both of these factors, therefore, can serve as good indicators of the probability of finding the file downstream.

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, though T_f and T_{q_f} can be different for each node and/or file. In the following section we demonstrate, however, that even a simple version of this policy can be quite useful in locating content in the network.

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 attractive properties: *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. A node becomes *invalid*, when using BECONS, if the interval between file/query arrivals is too long, in which case the BC *times out*; or if 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 is invalid.

Definition 3: The downstream trail $c_1, ..., c_j$ is said to be *stable* if it does not become broken *during* a downstream search. A query starting a search along a stable downstream trail will therefore end its search at an invalid BC only if all BCs further down are 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.

Consider now a simple instance of BECONS, termed S-BECONS (Simple BECONS), that for each file f uses the same T_f and T_{q_f} at all caches. With S-BECONS, if a query reaches a node with an invalid BC, the query is (re)routed towards the source. We require that $T_f \geq T_{q_f}$, since a new file download causes the file to be cached at every cache in the trail, whereas a new query may or may not refresh it's presence in some nodes. Finally, we make the following assumptions about the network properties:

- The propagation 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.

Based on the assumptions mentioned above, we make the following two claims, proven in the technical report [12].

Theorem 1: Let c_2 be a downstream neighbor of c_1. With S-BECONS, c_1 can determine trail invalidation for f if it receives a query q_f from c_2.

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

There are many advantages to a forwarding policy that ensures stability and has the capacity for trail-invalidation. Stability ensures that a search downstream will cover all valid breadcrumbs in the trail while searching for the file, while trail invalidation ensures that queries are not sent along a trail with no cached content.

Additionally, we observe that a consequence of trail stability is that every trail has a single *border node* - a node on the trail such that a query ariving to a node that is downstream from the border node will have the least expected cost when forwarded downstream, while a query arriving upstream will have the least expected cost when forwarded upstream. Stability ensures this since otherwise the trail would be broken at some point. The existence of such a node causes all queries q_f intercepting the trail lower than the border node to be forwarded down until the file is located. If it is, the cache containing this file will enjoy a considerable stream of queries q_f, which in turn will allow it to keep f stored for an extended period of time.

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. Let c_0 be the node where the trail was first intercepted. Here we argue that DFSP is the preferred policy, by observing the effect of each on node c_0: DFQ will have the file cached there on its way to the user, while DFSP will plot a new path to the user, possibly not containing c_0. The importance of this difference is that with DFSP c_0 has refreshed its BC with the query and will thus continue to forward future queries downstream in the same direction. As we discussed in the previous section, if the flow of queries is high enough, this can ensure with high probability that a copy shall remain cached downstream. With DFQ, on the other hand, the file passes through c_0 on its way to the user, creating a new breadcrumb trail and changing the direction of future query forwarding, thus preventing a critical mass of queries to persist over time at a specific node. 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, 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 q_j monotonically increases w.r.t. the arrival rate of q_i $(j \neq i)$. As r_i increases, f_i takes over a cache slot for longer stretches of time and in such a manner forces other files to be dropped more frequently. Let $I_x(i)$ $(O_x(i))$ be the incoming (miss) rate of queries q_i at node x. Also, for any parameter y, let $+[y]$ $(-[y])$ express an increase (decrease) in the parameter. Using this notation, we can write that for any cache x,

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

Reducing the incoming rate of requests for f_i will have the opposite effect as well. In what follows we demonstrate that this simple observation leads to a sophisticated type of implicit coordination between neighboring caches. For lack of room we present here the analysis for the case of 2 nodes only. A detailed discussion of a 3-node network is presented in the technical report [12]. We conjecture there that the 3-node case is sufficient to express the basic behavior of large networks, and leave an extensive discussion of the problems and methodologies presented here for future investigation.

Consider, then, the case of two neighboring nodes, x, y. 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 refer to such miss streams moving in opposite directions as *opposing streams*. We observe the following behavior:

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

Proof: An increase in the output of node x (wlog) 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 *reciprocity* - if x increases the query load on node y, y in return increases the query load on node x, but always using queries for a different set of files. Reciprocity can also suggest 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 important property as well:

Theorem 3: $+[I_y(1)] \Longleftrightarrow +[O_x(1)]$

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

The importance of Theorem 3 is that x increases the rate of q_1 being sent to y as a result of the original increase at y, even though the increase might have originated from a node other than x. 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.

Another important observation is that the reciprocating behavior of y's neighbors does not depend on the state at these neighbors. To understand this, note that the query miss rate for a given file is *not* monotonic with the incoming rate of queries. When $p_i \to 0$, the miss rate is bounded by the incoming rate which is going to 0, and when $p_i \to 1$, cache misses become rare for LRU caches and eventually go to 0. This makes the behavior of such systems difficult to predict. However, from

Lemma 1 we see that the miss rate for opposing miss streams are a monotonic function of each other. This property can help in modeling the behavior of a network in many ways. For instance, it implies that it is enough to know the changes in the miss stream of a single cache, in order to predict the type of change this will cause in the opposing streams from its neighbor, without having knowledge of the specific state at this neighbor.

In [12] we discuss the specific states that LRU caches can be in, and present additional support for the argument that with BECONS and similar policies node will tend to specialize in certain files, keeping them in the cache for longer and lending the system more stability than otherwise available. Further development of this high-level approach to cache-system analysis is left for future research.

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 cache-network simulator that generates queries at every node, modeled as a Poisson process. Both the location of the public file sources and the number of files assigned to each were chosen at random (uniformly). We evaluated performance over a non-congested system, allowing delays in the system to be constant, and leave a more load-dependant analysis for future work. Details of the simulation parameters can be found in the technical report [12].

Requested files from each node were selected at random, using the same distribution at each node, for which we chose both uniform and zipf. For every request sequence, we simulated the system behavior using different combinations of routing policies and cache replacement algorithms. Specifically we considered the simple case of routing to the source, as well as when using S-BECONS with either DFQ and DFSP. We also compared these to two cases of explicitly coordinated caching systems, without S-BECONS:

- A file enters a cache only if it is not located in any direct neighbor. A file is dropped from the cache using LRU, with preference to drop a file that is also cached in one of the neighbors.
- Same policy, but considering only the next cache en-route to the source of the file considered for caching/dropping.

We measured performance here in terms of reduction in query load at the file sources. We found that DFQ performed approximately the same as the simple, route-to-source policy, and so we present here our results only with regards to DFSP. Figure 2 depicts the *relative* aggregate number of downloads from all sources 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 relatively small. This can be explained by noting that explicitly-cooperating caches show performance gains mainly due to the fact that a group of caches acts as a single larger cache. When the cache size is relatively small compared to the number of different files, small cache groups cannot show large performance gains. Using Breadcrumbs, a decrease in cache size will mostly affect the time to locate and download content, rather than making it unavailable. In a real system the cache size will be considerably smaller than the number of files in the system, so Breadcrumbs should prove to be a more effective and scalable solution than explicit cooperation. Additional results can be found in the technical report [12].

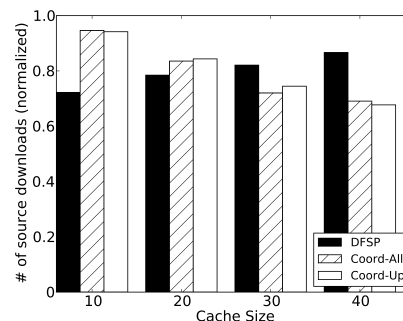

Fig. 2. Relative number of downloads from sources, using DFSP and two types of coordinated LRU caching. All values are normalized by the number of downloads from source experienced when routing to the source with standard LRU. The smaller the value, the less load experienced at the sources.

References

[1] A. Datta, K. Dutta, H. Thomas, and D. VanderMeer, "World wide wait: A study of internet scalability and cache-based approaches to alleviate it," *Management Science*, vol. 49, no. 10, pp. 1425 – 1444, Oct. 2003.

[2] V. Jacobson. (2007) A new way to look at networking. Internet video. [Online]. Available: http://video.google.com/videoplay?docid=-6972678839686672840

[3] S. Paul, R. Yates, D. Raychaudhuri, and J. Kurose, "The cache-and-forward network architecture for efficient mobile content delivery services in the future internet," in *Innovations in NGN: Future Network and Services*, May 2008, pp. 367 – 347.

[4] X. Tang and S. T. Chanson, "Coordinated en-route web caching," *IEEE Transactions on Computers*, vol. 51, no. 6, pp. 595 – 607, 2002.

[5] I. Ari, A. Amer, R. Gramacy, E. L. Miller, S. A. Brandt, and D. D. E. Long, "Acme: Adaptive caching using multiple experts," in *Proceedings in Informatics*, 2002, pp. 143–158.

[6] H. Che, Z. Wang, and Y. Tung, "Analysis and design of hierarchical web caching systems," in *IEEE INFOCOM*, 2001, pp. 1416–1424.

[7] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell, "A hierarchical internet object cache," in *Proceedings of the USENIX Annual Technical Conference*. Berkeley: Usenix Association, Jan. 1996, pp. 153–164.

[8] P. Krishnan, D. Raz, and Y. Shavitt, "The cache location problem," *IEEE/ACM Trans. on Networking*, vol. 8, no. 5, pp. 568–582, 2000.

[9] Y. Jin, W. Qu, and K. Li, "A survey of cache/proxy for transparent data replication," in *SKG*. IEEE Computer Society, 2006, p. 35.

[10] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura, "Self-organizing wide-area network caches," in *INFOCOM*, 1998, pp. 600–608.

[11] E. J. Rosensweig and J. Kurose, "Breadcrumbs: efficient, best-effort content location in cache networks," UMass Amherst, MA, Tech. Rep. UM-CS-2009-005, Aug. 2008.

[12] M. Busari and C. L. Williamson, "Simulation evaluation of a heterogeneous web proxy caching hierarchy," in *MASCOTS*. IEEE Computer Society, 2001, pp. 379–388.

[13] T. Ballardie, P. Francis, and J. Crowcroft, "Core based trees (CBT)," in *SIGCOMM*, 1993, pp. 85–95. [Online]. Available: http://doi.acm.org/10.1145/166237.166246