9-2012

On the Analysis and Management of Cache Networks

Elisha Rosensweig
University of Massachusetts Amherst

ON THE ANALYSIS AND MANAGEMENT OF CACHE NETWORKS

A Dissertation Presented

by

ELISHA J. ROSENSWEIG

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2012

Computer Science

ON THE ANALYSIS AND MANAGEMENT OF CACHE NETWORKS

A Dissertation Presented

by

ELISHA J. ROSENSWEIG

Approved as to style and content by:

Jim Kurose, Chair

Don Towsley, Member

David Jensen, Member

Lixin Gao, Member

Lori Clarke, Department Chair
Computer Science

To my wife,
the wind beneath my wings,
and to my parents,
an overflowing river from a live spring.

ACKNOWLEDGMENTS

The statement that this dissertation could never have come to be without the support of many people requires no formal proof, though if one would try to construct such a proof there would be ample evidence for it in all the avenues of my life. On both the intellectual and personal planes, I would never have been able to produce the work presented here if not for my biological and academic families standing by my side from the very beginning.

I would like to thank my advisor, Prof. Jim Kurose, who was not only a teacher and an advisor, but a true mentor to me. The mixture of constructive criticism and encouragement to follow my instincts he bestowed upon me has truly molded my research style. Especially, I would like to thank him for teaching me the importance not only of good research, but also of good writing, and I hope that he sees in me a successful product in both.

I owe a great deal as well to two other members of my committee, Prof. Don Towsley and Prof. David Jensen. Each, in their own field, have taught me the importance of asking the right questions and the right way, and to let the evidence lead the way. Throughout these past few years, their doors and minds were always open for discussion of research ideas I wished to share, which was a critical step on the way to producing this document. I would also like to thank each of the members of my committee for agreeing to serve on my committee. A special thanks in this is due to Prof. Lixin Gao, who agreed to help me in this final step of reviewing my dissertation despite our lack of previous acquaintance.

At this point I would also like to thank my Master's Advisor, Prof. Hanoch Levy from Tel-Aviv University, without whom I would never have made it this far. My

Master's Thesis did not have an Acknowledgement section, and so he never got his due. I hope this corrects that mistake, if belatedly.

Next, I would like to thank my labmates in the CNRG lab, who sat there through my dry-runs and gave me comments, and who were there for brainstorming sessions despite having plenty on their plate already. Thank you all, for making our lab a place where nobody ever needed to feel alone. I would also like, at the same time, to thank the amazing and caring staff of the UMass CS department, whose doors and ears were always open and happy to help with any problem I had.

Moving on the the personal sphere, I would like to thank the wonderful community in West Hartford, CT, who were our family for these past five years. Of these, I must single out Dr. Adam Gamzon, a recent graduate from the UMass school of Mathematics. It is not an exaggeration to say that he probably saved my life numerous times by being my reliable car-pooling partner on the long drives to and from campus, and that in addition to being a wonderful conversation partner for hours on-end.

And finally, I would like to thank my family. My parents, who supported us in all things material and spiritual, and are responsible for making this amazing journey more than just a young man's dream. My children, who made sure to interrupt my academic activities just enough to remind me that there is life worth living beyond these walls. Most of all, though, I thank my wife - for making sure that our family was taken care of through all the paper deadlines and exams, and for never letting me give up even when I doubted my own worth. As the renown Jewish sage of Antiquity, Rabbi Akiva, said to his students upon his return home from years of study and teaching, "All that is mine and yours - belongs to her".

ABSTRACT

ON THE ANALYSIS AND MANAGEMENT OF CACHE NETWORKS

SEPTEMBER 2012

ELISHA J. ROSENSWEIG

B.Sc., HEBREW UNIVERSITY OF JERUSALEM

M.Sc., TEL-AVIV UNIVERSITY

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Jim Kurose

Over the past few years Information-Centric Networking, a networking architecture in which host-to-content communication protocols are introduced, has been gaining much attention. A central component of such an architecture is a large-scale interconnected caching system. To date, the modeling of these cache networks, as well as understanding of how they should be managed, are both in their infancy.

This dissertation sets out to consider both of these challenges. We consider approximate and bounding analysis of cache network performance, the convergence of such systems to steady-state, and the manner in which content should be searched for in a cache network. Taken as a whole, the work presented here constitutes an array of fundamental tools for addressing the challenges posed by this new and exciting field.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION AND OVERVIEW

1.1 Introduction

Since the development of its earliest technical foundations more than 40 years ago, the Internet's dominant communication paradigm has been packet-based, host-to-host communication. Indeed, this view is deeply embedded in today's layered Internet architecture, with host-to-host segment-transport service (as embodied in TCP and UDP) serving as the communication abstraction provided to all applications. The idea, as developed over the past five decades and as encapsulated in the TCP/IP protocols, was to separate the question of "who/what do I wish to communicate with" from the implementation question of "how does a message get from here to there." While circuit-switching required global knowledge of the network topology and continuously maintaining a path between the two points by some central entity (e.g. a human operator), packet-switching required knowledge only of the address of the destination, and at each junction along the way (i.e., routers) a local decision was made regarding the next hop. The implications of this change in design were far-reaching, allowing the Internet to grow while retaining its basic properties of efficiency (via multiplexing sessions over the same links) and robustness (no single-point-of-failure).

Fast-forwarding ahead almost fifty years, the same basic architecture is still in place, surviving through the years with astounding resilience. However, with the passing of time the world surrounding and connecting to the Internet has changed in significant ways. Mobile devices now connect to the Internet, upsetting the topological stability that the network once had. As the Internet became more ubiquitously available, individuals and businesses began to move their services to the network. To deal with the increase in demand for content, suppliers first replicated content across servers, then moved to leasing Content Delivery Network (CDN) services, as well as leveraging Peer-to-Peer (P2P) technology. Packet-switching was designed to help computers locate each other; Internet users now needed systems to help locate content, the need growing as the rate of content production continued to increase.

To address this challenge, the past decade has seen the emergence of Information Centric Networking (ICN), in works such as TRIAD and DONA, and more recently in the CCN architecture [1, 25–27, 32, 52, 55, 68]. In these proposals, each piece of content is given a unique identifier (a "name") which can then be referenced in a host-to-content communication protocol. Content consumers state the name of the content they wish to retrieve, and the request is routed within the network, searching for the content according to some search policy. Just as TCP/IP separated the "what" from the "how" with regard to inter-host communication, these architecture propose to do the same for host-to-content communication.

Since, in ICN architecture, routers are aware of which content they store via content name, this creates an opportunity for content reuse: a router can send the same content to several content consumers that have expressed an interest in this content, instead of having each consumer access the content server individually. A central part of all leading ICN proposals thus involves universal caching: refitting routers with large caches that allow each router to store content that passes through it. When requests for content arrive at such a router-cache element, the cache can be checked for a copy of the content. If the content is available, it can then be downloaded from that cache, thus avoiding redundant delay and access to the content servers.

This change in the architecture is transformative: while previously the network could be thought of, broadly, as a large system of "bit-pipes," content is now stored at multiple locations inside the network, raising issues of scalability and privacy. Furthermore, with respect to content requests, the system can be viewed as a series of filters, allowing only a fraction of requests arriving to be forwarded on to the next hop. While networked caches have been researched in the past, these works have considered only small-scale systems and simple topologies (e.g., hierarchies). This dissertation considers a set of key challenges presented by the ICN architecture: the design, analysis and management of widely-deployed, tightly-connected, heterogenous Internet-scale networks of caches, of arbitrary topologies. As we will see, this is still a relatively uncharted field.

1.2 Goals and Contributions

In this dissertation, we address two distinct goals. The first is that of analyzing the behavior of Cache Networks (abbreviated as CNs). To understand the challenges involved in achieving this goal, it is instructive to consider the difference between models of caching networks, models of packet-switched networks, and models of circuit-switched networks. Unlike queuing networks for packet-switched networks, requests (workload) in caching networks are not queued - instead, they are either immediately satisfied locally or forwarded upstream. In this latter case, assuming negligibly small request-forwarding times, an exogenous arrival effectively generates simultaneous request arrivals at each node along the path in the routing tree, from the cache at which it first exogenously arrived to the node at which the file is found. According to leading ICN proposals (e.g., [27]), the file is then cached at each of these intermediate nodes. Thus caching network models should perhaps more closely resemble loss network models [34, 35] of circuit-switched networks, where exogenous calls are allocated resources (e.g., trunk lines between switches) at each node from source to destination. However, unlike loss networks, where a departing call releases resources simultaneously, resources in a caching network (cache storage) are separately released at each node as a result of subsequent arrivals and the node's cache-replacement policy. Also, unlike loss networks, a request results in resources (cache storage) being allocated only from the point of exogenous arrival to the point at which the file is found, rather than on the entire path from the arrival node to the destination node where the request was initially sent.

Cache networks differ in significant ways also from the small-scale cache hierarchies that have been analyzed previously with some success. In these smaller systems, content requests are all forwarded upstream, towards a common content server, which we refer to here as a *content custodian*, while caches are inspected along the way. Once content is located, it is sent downstream, back to where the request originated from. As a result, requests at one level in the tree are comprised solely of request misses one level below, while requests at the upper levels have no impact on lower-level caches. This special structure is heavily relied upon in previous work, allowing a bottom-up analysis of the network from the lower levels up towards the root. In cache networks of arbitrary topologies, on the other hand, multiple content custodians are spread throughout the network, and requests for different content are forwarded along different paths. As a result, two neighboring nodes might both send requests to, and receive requests from, one another. This introduces two layers of complexity into the model that do not exist in hierarchies. First, one cannot define an ordering on the nodes in the system, such that solving the state of one node depends only on those preceding it in this ordering; the modeling method mentioned before is difficult therefore to apply here. Second, since requests can flow in both directions across a given link, the miss streams of neighboring nodes can have a reinforcing effect on one another, a fact which does not exist in models for hierarchical systems. Finally, an additional challenge arises from the large scale of the network, which can make small node-wise modeling errors (that were acceptable in small networks) grow significantly over multiple hops.

Just as queueing theory was central for understanding the behavior of packet-switching networks, we believe that developing such a new set of analytical tools will be crucial for understanding cache networks. In this dissertation, we make the following contributions to the study of cache network modeling:

1. We develop an *approximation algorithm* for cache network performance, called a-NET. a-NET takes existing approximation algorithms that estimate the performance of a single cache in isolation, and uses these models to compute an approximation for an entire network. a-NET can deal with any network topology, and heterogenous networks where caches use different replacement policies.

2. We conduct an analysis of the factors affecting the accuracy of a-NET, when using a specific approximation algorithm for stand-alone LRU caches, and demonstrate that for this version of a-NET the dependencies within the cache miss streams are the major cause for inaccuracies. Based on this observations, we identify topological properties of a network that affect the accuracy of a-NET when using this LRU approximation algorithm.

3. We develop a *network calculus* for bounding request flows passing through LRU and FIFO caches. This calculus produces several key analytical results regarding LRU and FIFO worst-case performance. We demonstrate via simulation that in cache networks the worst-case bounds are indicative of actual network performance.

4. We also consider the factors that impact the steady-state behavior of a cache network. We demonstrate that, counter-intuitively, some networks can be greatly influenced by the initial state of the system (i.e., the initial contents in each cache). We then prove three independent conditions that ensure the steady-state of the system is not impacted by the initial state. In the course of

this, we present the concept of *equivalence classes* among replacement policies, such that proving properties for one proves them for all others in the class.

The second goal we set out to achieve is improved management of cache networks, specifically in the realm of content search - how to route content requests within the network. In the original proposal for Content Centric Networking (CCN) [27], requests are routed to the content custodian, which is known in advance via some mechanism (e.g., a search engine), and caches are inspected along this path. Routing directly to the custodian might reach a copy quicker, but it does so at the expense of creating bottlenecks at or nearby the custodian, and with possibly missing opportunities at caches off this direct path. Diametrically opposed to this would be exhaustive search or random walks, as in earlier versions of Gnutella P2P networks, which can spread the load but can incur long delays. A third option would be for caches to collaborate among themselves, determining where content is stored and where to route requests; these approaches can have high computational complexity and communication overhead which might make them unfeasible for ICNs [51]. A DNS-like system that explicitly keeps track of content location might suffer from similar problems, in addition to introducing a single point of failure. In light of the existing options and their limitations, we set out to develop a simple method for content search, that is both light-weight in terms of stored state information and coordination, while at the same time adaptive to system state.

In this dissertation, we make the following additional contributions, focusing on the field of content search in CNs:

5. We describe *Breadcrumbs*, a best-effort content search policy, in which each cache routes requests dynamically, based solely on local information. *Breadcrumbs* achieves this by using past traffic to set up breadcrumb entries - short-term routing hints that eventually expire *Breadcrumbs* is tunable, striking a balance between the route-to-custodian and exhaustive search policies. *Bread-*

crumbs also fosters an implicit inter-cache coordination of routing, without involving any inter-cache control overhead.

6. For a certain version of *Breadcrumbs*, called BECONS, we prove several properties regarding the efficiency of breadcrumb management. We show that BECONS creates a perimeter surrounding each content custodian, such that requests are routed to the custodian when they originate within this perimeter, thus reducing the load at custodians.

7. We present an analysis of causal relationships within the network, specifically between cache state and request routing tables. From this analysis, we devise and execute experiments to demonstrate the impact that *Breadcrumbs*-based search has on custodian load reduction.

1.3 Thesis Outline

The rest of this dissertation is organized as follows. Chapter 2 presents our algorithm for approximating the behavior of CNs. We also introduce here the model and notation used throughout this proposal, as well as much of the related work on CNs. Chapter 3 presents a network calculus for cache networks, and Chapter 4 discusses the impact (or lack thereof) of the initial state of a CN on its steady-state. Chapter 5 presents *Breadcrumbs*, our best-effort content-search policy, with extensive experimental results as well as causal analysis, focused on determining the efficiency of the *Breadcrumbs* content search. We conclude in chapter 6 with a summary of the thesis contributions and discuss future research directions.

CHAPTER 2

ANET - APPROXIMATING CACHE NETWORK PERFORMANCE

2.1 Introduction

Caches are an integral part of many computing systems, and consequently their policies and resulting performance have been the focus of much research. Earlier works have considered caches in isolation; more recent research has considered hierarchical (i.e., tree-like) cache network architectures [7,9,12,44]. Aside from the work presented here, interest in modeling cache networks of arbitrary topology has only recently started to appear; in addition to the work presented in this thesis there are several works-in-progress for analyzing such systems [33, 60, 61].

Caches are notoriously difficult to analyze, even when the policies employed to control which content is stored and which removed from the cache, known as *replacement policies*, are seemingly simple. For example, when considering the single, isolated cache running the popular LRU replacement policy, the complexity of exact models of cache state and performance grow exponentially as a function of cache size and the number of files in the system [14,37]. The challenges only increase as one considers *networks* of such caches. As a result, research on analyzing cache networks has been limited to simulation studies on the one hand and modeling a limited range of topologies and replacement policies on the other. With the increasing interest in ICN, there is a need for tools that can estimate the behavior of large-scale inter-connected *networks* of caches, arranged in an arbitrary topology and running a variety of cache replacement policies.

Networked caches in arbitrary topologies have several aspects that contribute to their complexity, and here we mention two of them. First, in all such systems the output process (known as the *miss stream*) of one cache becomes a part of the input process to another cache. Computing the state of a flow after passing through several such caches is thus a complex process. Second, when the topology allows requests and content to flow in both directions between neighboring caches, referred to here as *cross flows*, the analysis becomes more complex as each neighbor can affect the other simultaneously. While the first challenge has been discussed when considering hierarchical systems, to the best of our knowledge the work presented here is the first to address the second challenge.

In this chapter we present a novel **m**ulti-**c**ache **a**pproximation (MCA) algorithm, denoted as a-NET, that approximates the performance of a cache network of any topology and scale, and which can deal with heterogenous mixes of replacement policies. As a basic building block, we assume that for each individual cache we have a method for evaluating its performance under a given load. Due to the computational complexity of exact models for several common replacement policies, such as the LRU replacement policy we focus on here, we specifically consider the case where the performance of the cache is only approximated. Algorithms that compute such an approximation are referred to here as **s**ingle-**c**ache **a**pproximation (SCA) algorithms. Given SCA algorithms for the replacement policies used by caches in the network, the approach taken by a-NET is to **(a)** compute an SCA for each individual cache in the network, given its arrival process; **(b)** recompute the arrival process at each cache based on the misses computed in the previous step; and **(c)** repeat steps (a)-(b) multiple times until the solutions converge to a fixed point. In addition to presenting a-NET, we also consider its convergence properties and its accuracy for multiple topologies. In the process of this analysis, we identify several key parameters that affect this accuracy.

The structure of this chapter is as follows. Section 2.2 describes the model of cache networks adopted throughout this thesis, and Section 2.3 surveys work on cache network modeling, which is the focus of Chapters 2-4. Section 2.4 presents a-NET, and discusses some initial observations regarding its output. We follow with a survey of a-NET performance in Section 2.5. Motivated by several of the observations made in this section, in Section 2.6 we develop a method for determining the impact of several key factors on the precision of a-NET, and use it to determine the significance of inter-dependencies within the request stream on a-NET inaccuracies. We conclude with a summary of our results in this chapter in Section 2.7.

2.2 Model and Notation

2.2.1 System Components and Operation

We begin by describing the model used in this dissertation for cache networks (CNs). A summary of the notation that follows is presented in Table 2.1.

Content and Caches. Let $G = \langle V, E \rangle$ be a finite network comprised of nodes $V = \{v_1, ...v_N\}$ and edges $E \subseteq V \times V$. Each node corresponds to a *cache-router* element — a router augmented with short-term storage capabilities, such that content forwarded by the router can also be stored locally. Edges in this network indicate neighbor relationships among the cache-routers, such that cache-router v_i can forward an unsatisfied request — a cache *miss* — only via its neighbors. In related work, these are sometimes referred to as *Transparent En-Route Caches,* TERC for short [29,39,40]. For the sake of readability, the terms “cache”, “router” and “node” will be used interchangeably in what follows, each indicating such a cache-router entity.

Let $F = \{f_1, ..., f_L\}$ be the set of unique items of content, termed here *files,* that can be requested in the network. The *state* of a node v_i at time t is the *sequence* of files stored at it, where the order of files in the sequence determines the next file to be evicted upon a cache miss and subsequent download of new content. For replacement

Table 2.1: Table of System Notation

Notation	Meaning
v_i	A cache-router entity
c	The number of files a cache can store
f	A content entity (file)
N, L	The number of nodes and files, respectively
$cust(j) \subseteq V$	List of custodians for f_j
q_{ij}	A request for f_j at cache v_i
e_{ij}	Probability that $f_j \in v_i$
λ_{ij}	Exogenous arrival rate for f_j at v_i
r_{ij}	Combined arrival rate for f_j at v_i
s_{ij}	Miss rate for f_j at v_i
$\mathcal{R}_i$	Request routing matrix at node v_i

policies in which the file order within the cache is unimportant, we shall represent the cache state as a *set* of files instead of a sequence. The state of the network is the state of all its nodes. $f_j \in v_i$ denotes that f_j is stored at the cache of v_i, and $c_i = |v_i|$ is the size of the ith cache. By default, we will assume that all files in the system are of identical size, and consequently the units of cache sizes will be the number of files the cache can store[1]. This assumption is common practice in the field, and we adopt it here since we focus on replacement policies that are agnostic to content size. For simplicity of presentation, we shall assume all caches have identical size, and accordingly we denote the size of each cache as c.

Content Custodians. In addition to the short-term storage provided by caches, we assume the each piece of content is also stored permanently at one or more content *custodians* in the network, such as public content servers [23, 49, 67]. Each custodian connects to the network at a specific location — a cache-router element — and so we will denote these custodians as a set $C \subseteq V$. Note that for each $v_i \in C$, the storage required for maintaining these permanent copies is not included in the specified cache

[1]These files can also be thought of as named *chunks* of data, such that each file with variable size is broken into chunks of uniform size for dissemination [48].

size, as the content is not stored at the cache-router but at a device connected to it. We use $v_i \in cust(j)$ to denote that (a custodian connected to) node v_i has a permanent copy of f_j. All of our results in this dissertation hold for content stored at multiple custodians, and it is only for expositional purposes that we assume for all $1 \leq j \leq L$, $|cust(j)| = 1$, both in our discussion and in our simulations.

Request Routing. Requests for content can arrive at a cache either *exogenously* from a user directly connected to the cache-router, or *endogenously* when a cache miss occurs at a neighboring cache (Fig. 2.1). Exogenous requests flow through the network, passed endogenously from one cache to another until they are satisfied by locating a copy of the requested content. When a cache cannot satisfy a request, it generates a *cache miss*, and forwards the request along a *path* in the network in search of a copy. In this work we assume there exists a static routing matrix for each v_i denoted as $\mathcal{R}_i$, such that a cache miss for f_j at v_i will be forwarded to v_k with probability $\mathcal{R}_i(j,k)$. Denote $\mathcal{R} := \{\mathcal{R}_i\}_{1 \leq i \leq N}$ as the set of all routing matrices. In this work we assume each file has at least one custodian and that the request path ends at a node $v \in cust(j)$, so all requests are satisfied in finite time. A common example for a set of static paths is that of *shortest path* routing (used, for example, in [9]), in which a request for f_j is routed along the shortest path to the closest node in $cust(j)$. Dynamic routing matrices that change over time are addressed in Chapter 5.

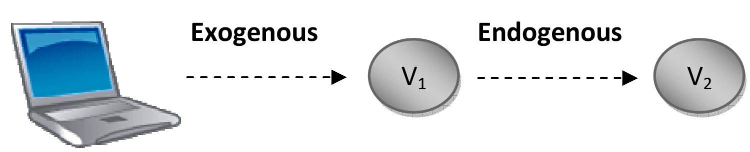


Figure 2.1: Example for endogenous and exogenous arrivals.

Request Handling. A request for f_j is denoted as q_j, and such a request arriving at node v_i is denoted as q_{ij}. For all $1 \leq i \leq N$ and $1 \leq j \leq L$, λ_{ij} is the exogenous rate of q_{ij}, where by "rate" we mean the *average* number of requests per unit time. We further denote

$$\boldsymbol{\lambda} = \{\lambda_{ij}\}_{1 \le j \le L, 1 \le i \le N} \tag{2.1}$$

When a request q_j arrives at v_i, the treatment of the request depends on the cache state:

- If $f_j \in v_i$, a cache *hit* occurs, and the file is forwarded back to the origin node where the request first entered the network. Unless otherwise stated, the file follows the reverse path traversed by the request.

- Otherwise, a cache *miss* occurs. If $v_i \in cust(j)$ then f_j is retrieved from this custodian; otherwise, the request is forwarded according to $\mathcal{R}_i$.

The *hit probability* at node v_i for f_j, denoted as h_{ij}, is the fraction of requests for f_j at v_i that result in a hit. The *miss probability* is then simply $m_{ij} := 1 - h_{ij}$.

We will consider two approaches for handling miss forwarding, as shown in Figure 2.2. The first (Fig. 2.2(a)-(b), referred to here as the *CCN* approach following its introduction in [27], allows for only a single miss for each file to be forwarded until that file is retrieved. Additional misses that arrive between the first miss (since the last download) and the resulting eventual download are registered at the node; when the requested content arrives at the node, it is forwarded to all nodes that requested it in that duration. The second, referred to here as the *baseline* approach and shown in Fig. 2.2(c)-(d), forwards each cache miss regardless of past events.

We denote $e_{ij} = Pr(f_j \in v_i)$. Also, let r_{ij} be the combined incoming rate of q_{ij}, and let s_{ij} be the rate of requests for f_j in the miss stream at node v_i. The rate of q_{ij} is then

$$r_{ij} = \lambda_{ij} + \sum_{v_k \in V} \mathcal{R}_k(j, i) s_{kj} \tag{2.2}$$

and a depiction of this is shown in Figure 2.3.

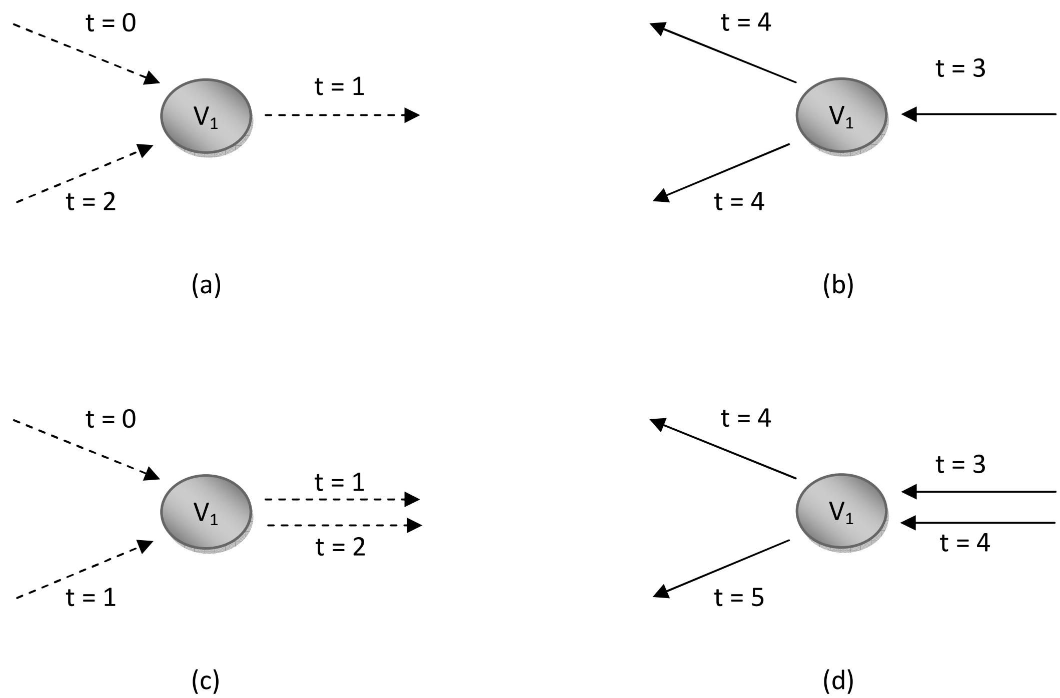


Figure 2.2: Request forwarding options. Dotted lines indicate requests, while solid lines indicate content downloads. A single node v_1 is shown. Case (a)-(b) depicts the CCN protocol, where requests are aggregated, while (c)-(d) depicts the baseline protocol where requests are not aggregated.

As with the exogenous rates, we denote

$$\boldsymbol{r} = \{r_{ij}\}_{1 \leq j \leq L, 1 \leq i \leq N}\,, \quad \boldsymbol{s} = \{s_{ij}\}_{1 \leq j \leq L, 1 \leq i \leq N}$$

When a file f_j is downloaded and passes through a node v_i whose cache is full and $f_j \notin v_i$, one of the files in the cache will be *evicted* to make room for f_j. A *replacement policy* at each cache determines which file is evicted. The caching literature is filled with many policies for such cache replacement, and in this dissertation we limit ourselves to considering the following policies, commonly found in the caching literature:

- Random (RND) - when removing a file, select a file uniformly at random from the available cached content.

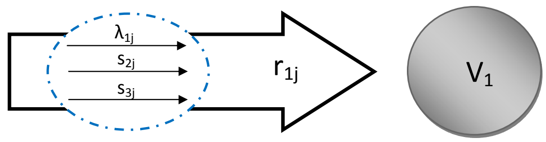


Figure 2.3: Request aggregation, for the scenario where nodes v_2, v_3 forward all misses for f_j to v_1.

- First-In, First-Out (FIFO) - when removing a file, select the file least recently *stored.* This policy is conveniently implemented as a queue of content, where the item most recently stored is placed at the tail of the queue and files are removed from the head of the queue.

- Least Recently Used (LRU) - when removing a file, select the file least recently *requested.* This policy is conveniently implemented as a queue of content, where an item is placed at the tail of the queue when it is first stored, and moved to the tail whenever it is requested. This is also the replacement policy of choice for many systems, including leading proposals for ICN [27].

2.2.2 Model Assumptions

In this dissertation, we frequently adopt two significant modeling assumptions that are common in the caching literature. In places where we relax or change these assumptions, we state this explicitly.

The first assumption concerns the properties of the **exogenous arrival processes.** We model the arrival process of exogenous requests using the Independent Reference Model (IRM) (as in, for example, [14,23]). According to IRM, the next file requested exogenously at a given cache is independent of the earlier requests. Formally, let X_h be a random variable representing the hth file exogenously requested at some cache v, then with IRM we have for all $1 \leq j \leq L$

$$Pr(X_h = f_j | x_1, \dots x_{h-1}) = Pr(X_h = f_j) \tag{2.3}$$

This assumption is considered valid when we assume that the exogenous requests are generated by a large number of independent users [19].

The second assumption relates to **content download delay**. When modeling the behavior of a cache or a cache network, a common assumption in the literature is that the download time of content is negligible [12, 14, 23, 24], which we term here the *zero download-delay* (ZDD) assumption. The main implication of this assumption is that whenever a cache miss occurs, the requested content is assumed to be instantaneously retrieved and stored at the cache. This makes the system more tractable for modeling, since as a result the order of content arrivals at the node is identical to the order of request arrivals, and cache state between request and subsequent download need not be considered. In Chapters 3-5 this assumption is not required, though we adopt it at times to make the exposition clearer, as is explained later. In this chapter we assume that ZDD holds, since the SCA algorithms we consider make this assumption for the single cache, and indeed a-NET would require modification in its design to explicitly address this delay. Addressing this is beyond the scope of this work.

The ZDD assumption has an additional convenient implication: with ZDD, one can ignore which miss forwarding policy we adopt, whether the CCN or baseline policy described above. This is due to the fact that these differ from one another only when there is a non-negligible window of time between when a cache miss occurs and its corresponding content download. Since we assume ZDD by default, we ignore this distinction unless otherwise stated.

2.2.3 Simulation and experimental methodology

The experiments conducted for this dissertation were conducted on an event-driven simulator written in Python (using the numpy and pylab libraries) for this purpose. Each experiment was constructed in two phases. First, the exogenous arrivals were generated and stored in a file, and then a simulation was conducted using this file. By reusing these arrival stream files, we could compare the performance of the different policies over different network instances for an identical request stream.

Since caches are always initialized empty and get populated via experienced content flows, we ignored the state of the system during the transient period at the beginning of each simulation. We defined a transient period in our system as the time until the distribution of exogenous requests at each node becomes close to the request distribution in the underlying generative model of these requests. Similarity between distributions was measured by using the *Kolmogorov-Smirnoff (K-S) Statistic*, also known as *K-S distance*. Given two random variables, X_1, X_2, the K-S statistic of these is defined as the maximal distance between their CDFs, i.e.

$$\max_x |F_{X_1}(x) - F_{X_2}(x)|$$

In all the simulations here, where X_1 was the generative model and X_2 was the actual load experienced at each node, we set the transient period to end when for each node the K-S distance for the exogenous request distribution was less than 0.05. In practice, in each simulation approximately 10,000 *exogenous* requests arrived at each node, and approximately 20% of these were attributed to the transient period.

Distance in the network was defined as the number of hops between two points. For constructing $\mathcal{R}_i$ we used shortest path routing, and when several paths existed with the same distance, each path was selected uniformly at random from all those with the shortest distance.

2.3 Related Work

The first three chapters of this work consider different aspects of cache network modeling, and thus share much of the related work. This section thus outlines past research related to the first three chapters. Our survey will focus on the LRU replacement policy, the replacement policy of choice in leading ICN proposals, although we will consider the Random and FIFO policies as well in some of the subsequent chapters.

2.3.1 Results for stand-alone caches

Research on analyzing the performance of single-cache systems abounds, and surveying it is beyond the scope of this section. A partial survey of common replacement policies can be found in [2,73]. In general, it is accepted that for many classic replacement policies (e.g.,LRU,FIFO), exact modeling of a stand-alone cache is intractable due to state explosion as c and L grow [37]. As a result, fast approximations have been proposed for these caches [14, 17].

A description of a CN includes, among other things, the policies used by each cache. The a-NET algorithm we present in this chapter assumes that there are algorithms for approximating performance of stand-alone caches using these policies, termed SCA (Single Cache Approximation) algorithms. In this work we use an IRM SCA algorithm developed by Dan and Towsley [14] for LRU caches, which we denote here a-LRU. a-LRU computes e_{ij} for v_i and f_j pair, and under IRM this is equal to the file hit probabilities [59]. See Appendix A for the formal description of this algorithm.

In addition to a-LRU other researchers have presented algorithms that compute an SCA of the hit probability at an isolated LRU cache under IRM assumptions. For example, Flajolet et. al. [17] present an integral solution for the cache approximation problem, which can be solved numerically to produce the hit probability. However, there is no straight-forward manner by which to observe the behavior of each file with this approach. Levy and Morris [47] compute the hit probabilities of an LRU cache given the *stack-depth distribution* of the cache — the distribution of which slot in an infinite cache will be referenced by a random request. Che et. al. [12] use a *mean field approximation* to approximate the behavior of individual caches. Their approximation assumes each file spends a constant time in the cache before it is evicted if not requested; they claim this assumption becomes more appropriate as the number of files in the system goes to infinity. In addition to the limitations of this approach for analyzing arbitrary topologies (see below), a-NET is a framework that can deal with policies for which files spend variable time in the cache.

Most of the cache analysis research to date has focused on the steady-state behavior of these systems, but there has also been interest in the behavior during the transient period of the system. In [6] the authors discuss the warmup phase of LRU, when starting from either an empty or non-empty cache, and use this to understand better how LRU behaves under traffic surges. Our work in Chapter 4 also considers the effects of the initial state, but differs in that it considers entire networks of such caches, and in that we focus on the resulting steady state of the system as a function of the initial state. To the best of our knowledge, this issue has not been addressed before.

2.3.2 Results for networked caches

We next consider models for *networks* of caches. There has been substantial work regarding cache *hierarchies* or *trees* [7–9, 12, 22, 44, 53, 54, 57, 73]. These systems are characterized by the existence of a single content custodian at the root for all content (e.g., slow memory for file-systems, the Internet for web proxy caches) and shortest-path request routing.

Rodriguez [57] considers cache hierarchies of multiple layers, but assumes the cache hit probabilities are *given,* and focuses instead on optimizing performance for a given system. Che et. al. [12] model a 2-tier cache hierarchy using the aforementioned *mean field approximation* (MFA). In subsequent work [44], they use this modeling technique to analyze cache coordination policies for cache hierarchies. Neither of these two papers provides much simulation support for this model. In recent work, Fricker et. al. [19] strengthen the analytical justification for using MFA in the context of Content-Centric Networking, and specifically suggest using MFA as the SCA within the a-NET algorithm we present here. Exploring this connection is left for future work.

The models mentioned above rely heavily upon the special properties of the hierarchical topology. First, several efforts make use of the symmetry of tree structures [7, 12]. When this symmetry is combined with uniformity assumptions on node policy and exogenous load, nodes at the same tree level behave in an identical manner. Second, with only one custodian and shortest path routing, requests flow only upstream, from caches towards the custodian, and content flows only in the opposite direction, essentially making this network a *feed forward* network. This feature (combined with ZDD assumptions) allows for analysis to be done from the bottom up. In contrast to these works, all four chapters in this dissertation are applicable networks of *arbitrary* topologies, where content and requests can flow in both directions on network links.

2.3.3 The P2P connection

A large body of work that bears much resemblance to networks of caches is that of P2P networks, especially *hybrid unstructured P2P networks* [23,24,49,67], abbreviated here as HP2P. In these systems, peers form an overlay network of arbitrary topology, search for content among peers in this network and then download the located content. The system is "hybrid" as it assumes there is always an accessible publisher entity for each content item, to distinguish from "pure" P2P systems where content might become unavailable when a set of peers leaves the swarm. Thus, publishers and peers have similar roles to custodians and caches respectively, suggesting that results from one field might be applicable to the other [2].

In the field of HP2P, Kleinrock & Tewari [67] show that using LRU at all peers achieves near-optimal replication of content in terms of load distribution at peers and distance to content. They assume copies are distributed uniformly in the network, and ignore the question of how content is found. Ioannidis & Marbach [23] consider the performance of Random Walk and expanding ring query propagation in HP2P systems. They also assume content is uniformly distributed in the system, and ignore the storage limitations of each node (thus not considering replacement policies).

While these results contribute to our understanding of cache networks, there are some important differences between the two fields. One important difference is that with most P2P and HP2P systems the overlay topology is relevant only for content search, while in CNs content download and content search take place over the same topology, with content populating the caches during download. While there are P2P systems where content populates the peers along the download path (e.g., Gnutella [42]), this is not required for P2P to function, and little analysis has been conducted for such systems. In this sense, cache networks are a generalization of P2P/HP2P systems, in that the download path plays a central role in how and where content is stored within the system. Thus, a richer set of tools is needed to understand their behavior.

2.3.4 Modeling Assumptions

All the components of the model we adopt and describe here are used elsewhere in the literature. These include ZDD [12, 23, 24, 44], IRM exogenous requests [12, 14, 23, 24,44,67], and storing content at nodes that did not request it [49]. In our simulations,

[2] In fact, some work on hybrid P2P networks might be even more applicable to CNs than to P2P systems. For example, in [23, 24, 49, 67] the analysis of HP2P networks relies on the network having a static topology. While this assumption is violated in P2P, it is more well-founded in CNs.

we in general assume identical exogenous request distribution at all users [7, 24, 67]. We now explore in more detail some of our main modeling assumptions — IRM exogenous arrivals, cache coordination and homogeneous cache policies.

2.3.4.1 IRM Exogenous Request Streams

The model we use here for exogenous request traffic is the Independent Reference Model (IRM). However, there are alternative models for request patterns at single caches. Panagakis et. al. present approximate analysis for streams that have short term correlations for requests [50]. As we shall see below, CNs exhibit the opposite effect, with content requested recently *less* likely to be requested next.

An approach that deviates sharply from IRM is that of Stack Depth Distribution (SDD) [47]. With this model, the stream of requests is characterized as a distribution $\vec{h} = (h_i)_{i=1}^{\infty}$ over the cache slots in a cache of infinite capacity, where h_i is the probability that the next cache hit will be at slot i. In this model, all information regarding the individual files being requested is ignored or unavailable.

2.3.4.2 System Architecture - Cache Coordination

When considering cache interaction, different approaches have been suggested for coordinating caches. Some have proposed systems where caches explicitly coordinate what to store and where [38, 41, 65], while at the other end of the spectrum some have considered systems where caches are oblivious to the state of other caches [12, 58, 59]. In this dissertation we consider only architectures of the second type, though some of the modeling tools presented here could be used for some coordinated systems as well. We discuss the differences between these approaches in more detail in Chapter 5.

2.3.4.3 System Architecture - Replacement Policies

It is standard practice to select LRU replacement policy as the policy used by caches in the network. This selection is justified by the benefits that LRU has been shown to give in smaller caching systems. As such, most of the experiments and discussion in this dissertation assumes LRU is used at all nodes. However, recently [20] it has been suggested that RND replacement would have comparable performance. This might make RND a better choice for CNs, as it is much easier to implement and its performance is easier to understand and characterize via modeling. These results correspond to observations we have made over the course of our work. A detailed comparison of replacement policies for CN is beyond the scope of this work.

The results presented in Chapters 2 and 4 apply equally to homogenous and heterogenous cache networks. *Homogenous* CNs are networks in which all caches employ the same replacement policy, while in *heterogenous* CNs each cache might use a different policy or policy combination (e.g., v_i uses LRU and v_k uses RND replacement). We believe that this second class of networks is likely to occur especially when network management is not under a single controlling authority, and indeed might improve performance in some cases [43]. It is worth noting that very little is known about the performance of heterogenous networks with an arbitrary topology. Heterogenous replacement policies have been discussed previously in the context of cache hierarchies, where some have suggested that upper-level caches should employ different replacement policies than those in lower levels [8, 22, 73]. Extending these ideas to networks with arbitrary topologies is non-trivial and is beyond the scope of this work.

2.4 The a-NET Algorithm

In this section, we describe the a-NET algorithm and discuss some of its properties. After presenting preliminary concepts (§2.4.1) we formally present the algorithm

(§2.4.2), and prove that a-NET always converges to a solution for FIFO and RND (§2.4.3). We followup In Section 2.5 with a survey of the accuracy of a-NET over multiple scenarios.

2.4.1 Preliminaries

For a given cache network and exogenous request load, our goal is to compute the load experienced at each cache in the network, as well as the performance of each cache under that load.

We characterize the exogenous arrival stream as the set of *request rates* $\boldsymbol{\lambda}$, where arrivals follow IRM as discussed above. We adopt this model here — one which is widely used in the literature — to conform to the assumptions made regarding the input to isolated-cache approximation algorithms we use here. However, it is important to note that a-NET can be applied more generally to any flow characterization, as discussed in Section 2.7.

a-NET takes as input a cache network G, the size of each cache, the exogenous rates $\boldsymbol{\lambda}$, the custodian location for each file, and the routing tables for each node. It produces an estimate for both $\boldsymbol{r}$ and $\boldsymbol{s}$, and then the miss probability for f_j at v_i is s_{ij}/r_{ij}. Note that for the case of a single node, by construction $\boldsymbol{\lambda} = \boldsymbol{r}$. In what follows we shall refer to computing an estimate for $\boldsymbol{r}$ and $\boldsymbol{s}$ as estimating the *performance* of the network.

In order to compute the performance of the network, a-NET requires an algorithm for computing the performance of a single cache (in isolation, with no surrounding network), given the arrival rates at the cache. We refer to these as Single Cache Approximation (SCA) algorithms. The input to the SCA algorithms we consider here consists of the cache size and IRM arrival rates, and the output is the miss-rate at the cache for each file. A detailed definition of the algorithms we use here can be found in Appendix A.

2.4.2 Algorithm Description

a-NET is an iterative, fixed-point algorithm, as shown in the flow-diagram in Figure 2.4. We begin with assigning $\boldsymbol{r}$ the values of the exogenous arrival flows (top left box), and using the SCA algorithm we compute the miss stream per cache. We then repeat, until convergence, the following two-step process: (a) Recompute the arrival streams at each node from the miss stream and the routing matrices; (b) Recomputing the miss streams using the SCA algorithms and the new arrival streams. The converged-to values are returned as the estimate for arrival and miss rates (bottom right box).

Algorithm 1 presents the pseudo-code for this process, which we review now in more detail. After initializing the estimation variables to zero (line 1) we enter the WHILE loop (lines 2-3). In each iteration over this loop, we assign the arrival rates at each node according to Eq. 2.2, which in the first iteration equals the exogenous rates (lines 5-9). We then compute the resulting miss rates (lines 10-12). The difference between the computed values from the previous round and this round is computed (line 13), and the loop condition is checked. We repeat this process until the system converges to a fixed point, which is returned as the estimate for the network performance.

To compute the distance between iteration computations, we use the *Kolmogorov-Smirnoff (K-S) Statistic*, also known as *K-S distance*. Given two random variables, X_1, X_2, the K-S statistic of these is defined as the maximal distance between their CDFs, i.e.

$$\max_x |F_{X_1}(x) - F_{X_2}(x)|.$$

In our experiments, we computed the K-S distance for each node between one iteration and the next. Formally, let $(p_{i1}^{(k)}, ..., p_{iL}^{(k)})$ be the popularity distribution at node v_i as computed for the kth iteration, where $p_{ij}^{(k)} := \frac{\hat{r}_{ij}^{(k)}}{\sum_h \hat{r}_{ih}^{(k)}}$. The K-S distance is computed for these distributions, between one iteration and the next, for each node. In our

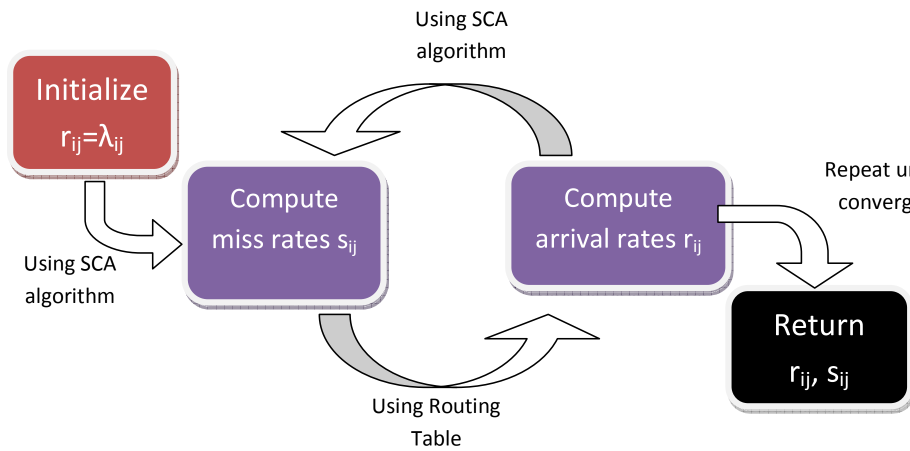


Figure 2.4: Flow-Diagram of a-NET.

experiments, we halted the algorithm when this distance went below 10^{-4} for all caches. In all of our experiments, the system converged to such a fixed point.

The number of iterations a-NET required until convergence depended on the topology under consideration. For tree topologies, the number of iterations required was equal to the height of the tree, since after the kth iteration there is no change to the arrival streams at the bottom k levels of the tree, as can be proven by induction[3]. For other, arbitrary topologies with multiple custodians, the number of iterations required for convergence was bounded by twice the radius of the network. These results highlight the benefits of using a-NET over simulation of a cache network, where the simulation length required for meaningful results can be very long.

In Section 2.5 we survey the accuracy of a-NET in detail, and for now we consider a single example to gain some intuition as to what a-NET produces as output, shown in Figure 1. Here we consider a 10x10 torus (see Fig. 2.6 below) with 500 files distributed among four custodians. File popularity follows a Zipfian distribution

[3]Proof sketch: For $k = 1$, the only arrivals at the leaf nodes are exogenous, and these do not change at any iteration. For the induction step, the kth level from the bottom receives requests only from lower levels, and since there is no change in these levels from this iteration on per our induction assumption, the claim is proven.

Algorithm 1 The a-NET algorithm.

Input: $\boldsymbol{\lambda}, c, \mathcal{R}, \epsilon, alg$ // alg is the SCA algorithm

$\forall i, j\ \hat{s}_{ij} := 0,\ \hat{r}_{ij} := 0$
$\Delta = 2\epsilon$ // Dummy value, to ensure entering loop.
while $\Delta > \epsilon$ **do**
  $\hat{\boldsymbol{r}}_{prev} := \hat{\boldsymbol{r}}$ // Store for convergence check
  for i=1 to N **do**
    for j=1 to L **do**
      $\hat{r}_{ij} = \lambda_{ij} + \sum_{v_k \in V} \mathcal{R}_k(j,i)\hat{s}_{kj}$
    end for
  end for
  for i=1 to N **do**
    $\hat{s}_{i1}, ..., \hat{s}_{iL} = alg(c, \hat{r}_{i1}, ..., \hat{r}_{iL})$
  end for
  $\Delta = \text{computeDiff}(\hat{\boldsymbol{r}}_{prev}, \hat{\boldsymbol{r}})$
end while
RETURN $\hat{\boldsymbol{r}}, \hat{\boldsymbol{s}}$

with parameter 0.8. The figure shows the actual and estimated $\boldsymbol{s}$. As we can see in this example, a-NET consistently under-estimates the miss-rates in this example, with the approximation usually within $80 - 85\%$ accuracy.

2.4.3 a-NET convergence

Before delving into the output of a-NET, we address the issue of algorithm convergence. As is clear from Algorithm 1, a-NET can only return a solution if the iterative procedure converges to some fixed point solution. In this section we prove such convergence is guaranteed for FIFO and RND. Showing this for LRU and other policies is left for future work. We note, however, that in all of our LRU experiments, the algorithm converged.

We begin with a qualification of our claim in this section. The convergence (or lack thereof) of an a-NET implementation depends on the SCA algorithm, which is affected in part by the replacement policy whose performance is being approximated. Since all such algorithms attempt to reflect the behavior of the actual replacement policy, we prove here that a *perfect* SCA algorithm will ensure convergence. Since the SCAs

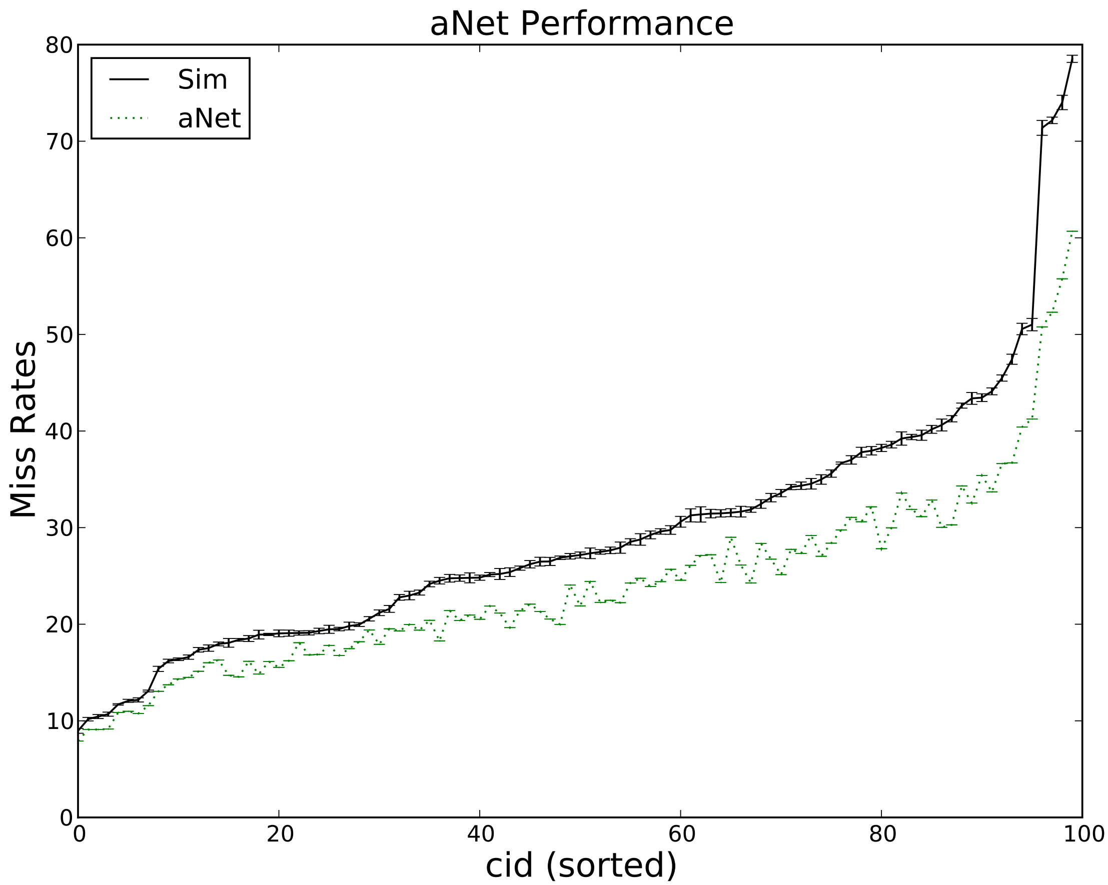


Figure 2.5: Example of a-NET performance, where data points are sorted according to increasing miss rates in the simulation. Shows the miss rates of a 10-by-10 Torus topology with four custodians, each holding a quarter of 500 files, as computed via simulation and a-NET. Values are shown for each cache (x-axis) and sorted in ascending order of simulation values. Requests arriving at each node are distributed according to Zipf distribution. 90% confidence intervals shown.

we consider here assume IRM for the arrival streams, this perfect algorithm computes the correct miss rates given an IRM set of arrival rates. For specific approximation algorithms, this proof approach might be applicable depending on the approximation properties.

Since we assume IRM, we use the following important property that relates the existence probability of f_j at a cache to the hit probability for f_j:

Lemma 1. *If the request arrival process is IRM, $e_j = h_j$ for all $1 \leq j \leq L$.*

Proof: Let X_k be a random variable for the identity of the file requested by the kth request to arrive at v, such that $X_k = f_j$ indicates the kth request was for f_j. The hit probability is then, according to Bayes' Theorem,

$$h_j = Pr(f_j \in v \mid X_k = f_j) = \frac{Pr(X_k = f_j \mid f_j \in v)Pr(f_j \in v)}{Pr(X_k = f_j)} \tag{2.4}$$

Since the arrival process follows IRM, X_k is independent of earlier requests; also note that whether $f_j \in v$ is determined only by the previous requests. Thus, $Pr(X_k = f_j \mid f_j \in v) = Pr(X_k = f_j)$, and continuing from where Eq. 2.4 left off we conclude our proof:

$$\frac{Pr(X_k = f_j \mid f_j \in v)Pr(f_j \in v)}{Pr(X_k = f_j)} = \frac{Pr(X_k = f_j)Pr(f_j \in v)}{Pr(X_k = f_j)} = Pr(f_j \in v) = e_j$$

■

From this lemma we can compute $m_j := 1 - h_j = 1 - e_j$, i.e., the miss probability can be determined from the existence probability; combined with the arrival rate, the miss rate can be computed. Thus in what follows we will focus on properties of the existence probability of individual files in FIFO and RND.

2.4.3.1 FIFO and RANDOM replacement

We next state and prove properties of FIFO and RND that, if reflected in the SCA algorithms used, will ensure a-NET convergence. Our proofs hold also for heterogenous networks, where each cache selects one of these policies independently. To this end, we make use of the following property of these replacement policies:

Lemma 2. *Let v be some cache using FIFO (or RND) replacement. Then whenever $f_j \in v$, requests for f_j do not impact cache state.*

Proof: With RND there is no meaning to internal ordering of content in the cache, and requesting content that is in the cache does not generate any changes in the contents of the node. With FIFO the internal ordering of content reflects the order of when content was last *inserted* into the cache, but additional content references do not impact cache state. ■

Lemma 3. *Let v be some cache using FIFO (or RND) replacement. Let $\tau_{j,in}$ be the mean time f_j spends in v before eviction. Then $\tau_{j,in}$ is independent of r_j.*

Proof: From Lemma 2 we know that for both FIFO and RND, cache hits do not affect the cache state. Since cache hits occur iff the requested content is in the cache, during $\tau_{j,in}$ all requests for f_j will generate hits, with no impact on cache state. Thus, we specifically conclude that the value of $\tau_{j,in}$ is not impacted by requests for f_j. ■

Theorem 4. *When each cache independently uses either FIFO or RND, a-NET converges to a fixed-point solution.*

Proof: We show that each individual cache will converge in a-NET, thus leading to the convergence for the entire network. Recall that we are assuming the arrival process at each cache is IRM, which allows us to use Lemma 1.

First we note that, for each file, the sum of all exogenous request rates arriving into the system is a bound on the arrival and miss rates for each cache individually. This is a bound on the arrival rates since the routing table is assumed to lack cycles, so requests never move through a cache more than once on their way to a custodian. Furthermore, by definition the miss stream rates at a cache are bound by the arrival stream rates, so by transitivity the miss stream of each cache is bound from above as well.

Next, we now show that for a cache using RND or FIFO, the miss rate for f_j increases monotonically with an increase to the arrival rate of f_k requests. We consider two scenarios:

- $j \neq k$: From Lemmas 2-3 we know that the added requests for other content only impact the cache state if they generate a cache miss, and cache misses for $f_k \neq f_j$ generate evictions at the cache, which can cause f_j to be evicted sooner. Thus, an increase in requests for $f_k \neq f_j$ can only decrease $\tau_{j,in}$. Let δ be the decrease in $\tau_{j,in}$, h_j be the original hit probability and $h_j^{(\delta)}$ the new hit probability. With IRM an eviction is independent of past and future requests, so $\tau_{j,out} = 1/r_j$; From Lemma 1 we know the hit probabilities are equivalent to the existence probability, so we get

  $$h_j = \frac{\tau_{j,in}}{\tau_{j,in} + 1/r_j} > \frac{\tau_{j,in} - \delta}{\tau_{j,in} - \delta + 1/r_j} = h_j^{(\delta)}$$

  The miss rate is then increased, since the arrival rate for f_j remained unchanged: $s_j = r_j(1 - h_j) < r_j(1 - h_j(\delta)) = s_j^{(\delta)}$.

- $j = k$: With the increase in requests for f_j the hit probability increases as well. We show now that despite this rise, the miss rate continues to increase.

  Let $s_r(r_j)$ be the miss rate for f_j given that r is a vector of the arrival rates of each $f_h \neq f_j$, and r_j the arrival rate for f_j. Similarly we use $e_r(r_j), h_r(r_j)$ to denote the existence and hit probabilities as a function of r and r_j. So,

  $$\begin{aligned} s_r(r_j) &= r_j(1 - h_r(r_j)) = r_j(1 - e_r(r_j)) \\ &= r_j\left(1 - \frac{\tau_{j,in}}{\tau_{j,in} + 1/r_j}\right) \\ &= r_j \frac{1/r_j}{\tau_{j,in} + 1/r_j} \\ &= \frac{1}{\tau_{j,in} + 1/r_j} \end{aligned}$$

  Taking the derivative of this expression w.r.t. r_j we get (keeping in mind that $\tau_{j,in}$ is independent of r_j, as shown in Lemma 3)

$$s_r(r_j)' = \frac{1}{\tau_{j,in} + 1/r_j} \cdot \frac{1}{r_j^2} > 0$$

This expression is always positive, and so we know that the miss rate is monotonically increasing, despite the rise in hit probability.

From this property, we prove using induction that in each a-NET iteration the arrival and miss rates for each file monotonically increase. In the first iteration, the arrivals are the exogenous arrivals and the miss rates are set to zero. At the end of the first iteration, the miss-rates are non-zero, and the arrivals are now a combination of the exogenous arrivals and the endogenous miss flows. For the induction step, we know from the claim above that if there is an increase in the arrival rates, there will be an increase in the miss rates, so the the claim is proven.

We conclude our proof by stating that since the arrival and miss rates are monotonically increasing but bounded (per cache) from above, the system converges to a fixed point. ■

2.4.3.2 Convergence for LRU

In proving the convergence of a-NET for FIFO and RND replacement, we leveraged the monotonicity of the miss stream in these caches. Based on our experience, such an approach is not suitable for proving convergence with LRU. Our experiments have shown that, with LRU, increasing the arrival rate for a given file can actually *reduce* the miss rate for that file. Specifically, consider the miss rate for file f_j at some cache v using LRU, as a function of r_j. Our experiments indicate that the miss rate for this file is *unimodal:* s_j first increases with r_j, then decreases. Assuming that this observed behavior is indeed a property of LRU, it can be explained by the fact that with LRU a popular file can remain almost indefinitely in the cache, thus having a near-zero misses. Thus, other proof techniques should be considered when proving convergence of a-NET where caches use LRU as a replacement policy.

Table 2.2: Default values in simulations

Parameter	Value (default)
Topology	Tree / 10x10 Torus
num. of files (L)	500
Popularity distribution	Zipfian with parameter 0.8
Confidence intervals	90%

2.5 Performance Evaluation

In this section we evaluate the performance of a-NET, varying multiple parameters: network connectivity, custodian placement, request distribution, cache dimensions and the number of unique files in the system. While most of our experiments relate to LRU caches, we demonstrate a-NET also for RND replacement. Finally, we also consider the impact of download delay on the approximation accuracy. Some of these results are used to motivate our discussion of the properties of a-NET in the following sections of this chapter.

In this chapter, the default network parameters are specified in Table 2.2. We consider two basic topologies:

Trees. These are complete k-ary trees, with a single content custodian at the root. These topologies are studied in much of the related work, though usually for small-scale, two-level scenarios.

Toruses. See Fig. 2.6 for a 2D depiction of this topology. As shown there, this topology can be viewed as a grid where nodes directly opposite at the grid rim have edges between them. Unless otherwise specified, we divide the files among four custodians. These custodians that are placed in the network so as to maximize the minimal distance between any two custodians, thus generating much traffic in both directions across all links in the network. To the best of our knowledge, these topologies have not been addressed to-date in the cache networks modeling literature, and thus we present here more torus than tree topology simulations.

Our selection of the Torus topology (instead of more realistic topologies) was motivated by the significance of *custodian placement* in these networks. The location of each content custodian in the network determines the direction in which requests are routed. In order to avoid simulation artifacts that are strongly affected by this custodian placement within the topology, we selected the torus topology due to its symmetric structure.

In order to measure approximation accuracy, the metric we focus on here is the *miss probability ratio*, abbreviated MPR. The MPR is the ratio, for each node, between the actual miss-probability at that node and the approximated miss probability.

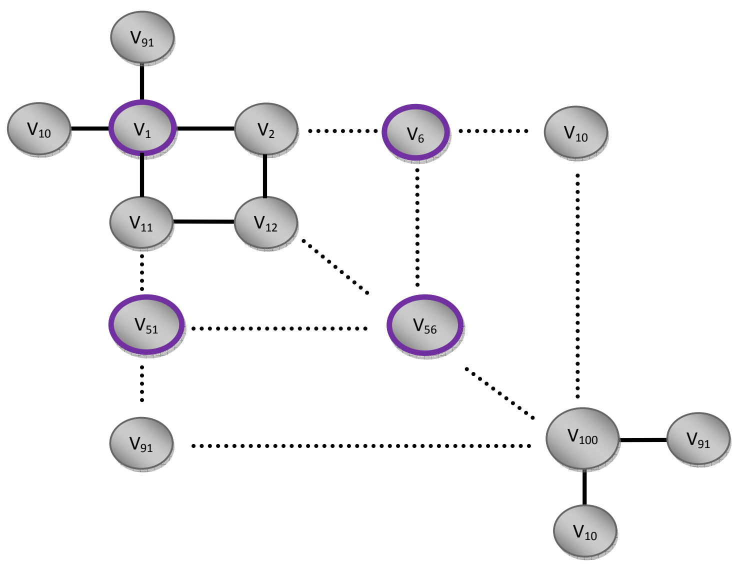


Figure 2.6: Torus topology used throughout this dissertation. Four custodians are indicated (bold borders) at nodes 1,6,51,56. The torus property is explicitly denoted for nodes v_1, v_{100}, but apply across all the border.

Impact of number of files, cache size and exogenous distribution. We consider the impact of changes to the number of files L and the cache size c on a-NET accuracy for the torus topology. In Figure 2.7 we plot the MPR for each node, for varying combinations of L and c. As can be seen in this figure, as the L/c ratio increases, so does a-NET prediction improve its precision. Figure 2.8 shows, for these cases, the correlation between the simulation miss probability and the MPR. We can see here that the MPR increases as the miss probability grows, though this effect is weaker within each scenario. The fact that a-NET performs better when the L/c ratio increases is an important feature for practical uses of a-NET, as in real systems this ratio is expected to be very large. We also can see in Fig. 2.9 that a-NET performs better as the exogenous distribution is closer to uniform.

Impact of node degree. Figure 2.10 shows the MPR in a complete k-ary tree as a function of the tree branching factor (i.e., the value of k). Our results show clearly that as the branch factor increases, so too does the accuracy of a-NET improve. This same behavior is exhibited with other exogenous popularity distributions, such as uniform and geometric. Thus, it seems that a-NET is more precise as the node degree increases. Additional corroboration for this can be seen in Fig. 2.11, which shows the precision of a-NET over a random graph where each pair of nodes has an edge with probability p, shown in the x-axis. As p grows, the performance of a-NET also improves, matching our findings in trees. However, in Random graphs and unlike the tree scenario, increasing p also decreases the path length between nodes, which causes a-NET to require fewer iterations. Thus, the decrease in error here might be related to a different factor. We provide some analytical support for the importance of node degree Section 2.6.

Impact of inter-custodian distance. In a torus topology where custodians are close together, we expect the network to behave in a similar manner to that of a Tree with a single custodian. Specifically, on most edges in the network, requests will flow in one direction and content in the opposite direction. By distancing custodians from one another we generate *cross-flows* on edges.

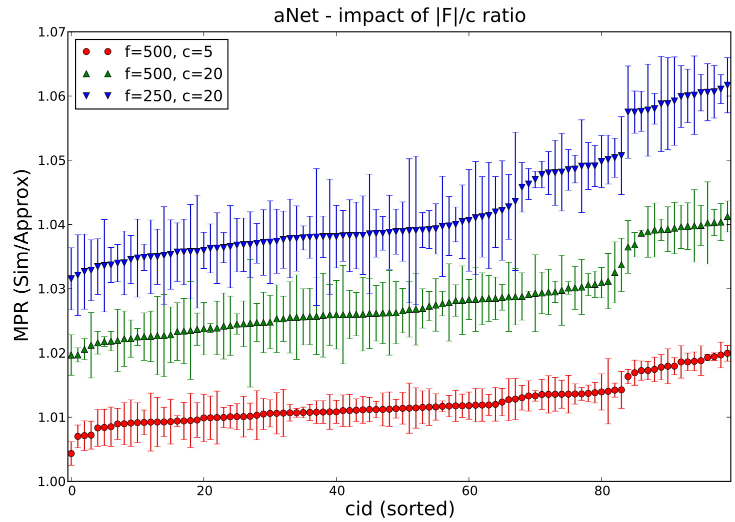


Figure 2.7: Per-node MPR for 10-by-10 torus networks, as a function of the L/c ratio. The values were sorted in ascending order. As we can see here, as the ratio grows the performance of a-NET improves.

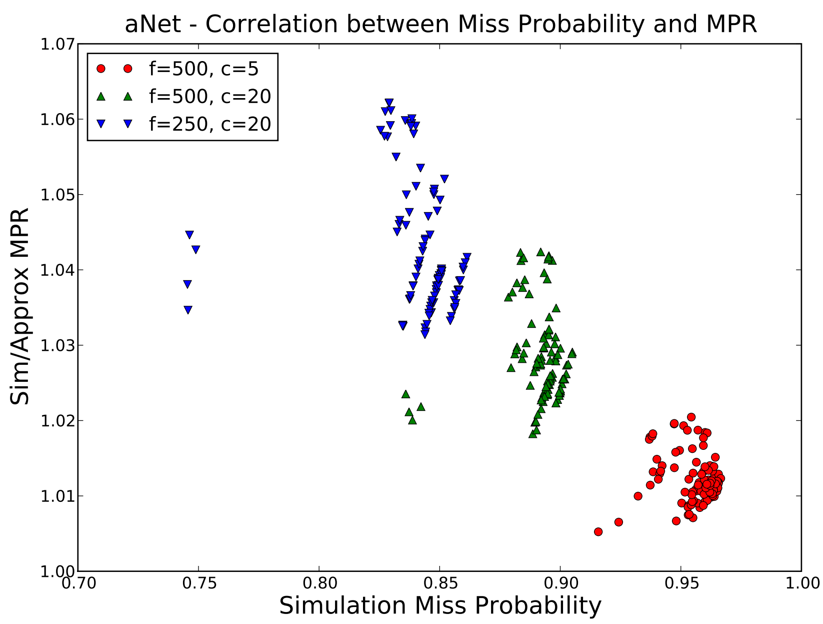


Figure 2.8: For the same scenario shown in Figure 2.7, the correlation between MPR and miss probability in the simulation. Each point represents the miss probability and MPR for a cache in the network. For each of the three scenarios we show the correlation for a single simulation. We can see here that between scenarios, the MPR decreases as the miss probability increases.

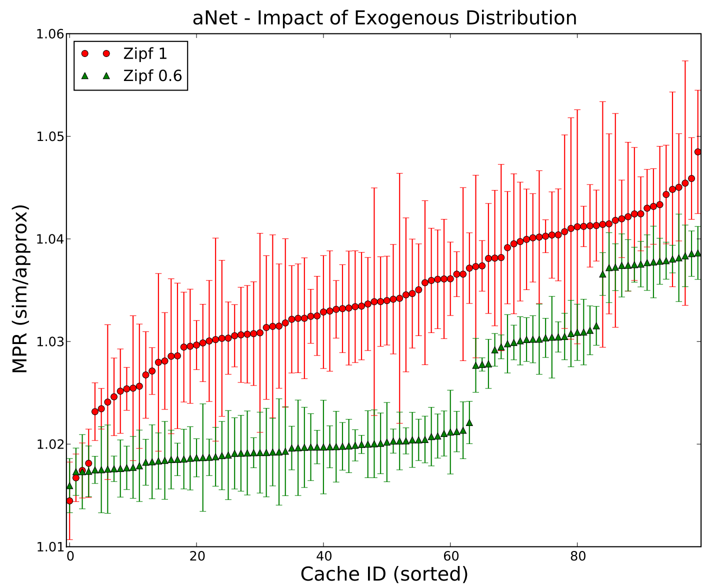


Figure 2.9: Per-node MPR for 10-by-10 torus networks, as a function of the arrival distribution (Zipfian with parameters 1.0 and 0.6). The values were sorted in ascending order. As we can see here, as the distribution becomes less skewed (0.6), the performance of a-NET improves.

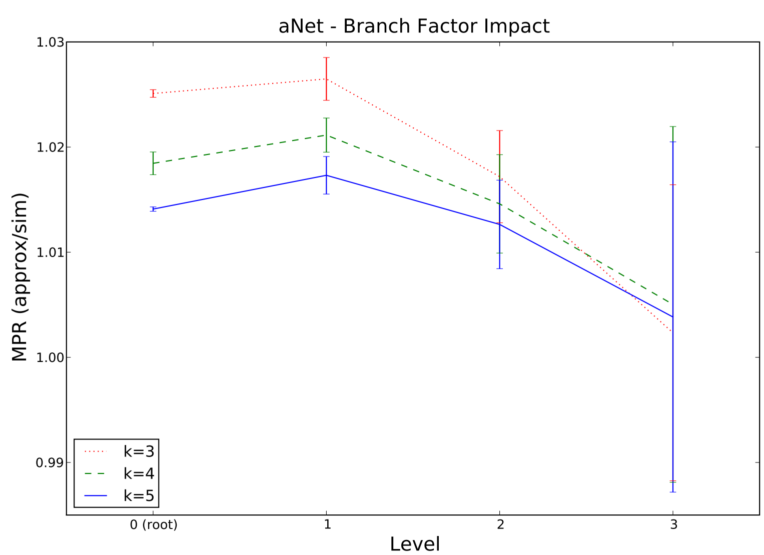


Figure 2.10: The impact of a tree branch factor on a-NET performance. Due to symmetry within the tree, values for each level are aggregated. We can see that as the branch factor grows, so does the approximation become more accurate.

Our results (Fig. 2.12) indicate that performance degrades as the inter-custodian distance grows. While the exact mechanism that explains this is currently unknown, this result highlights the complications inherent in non-hierarchical systems.

a-NET for Random replacement. To demonstrate the flexibility of a-NET, we present here in Fig. 2.13 the MPR for a different replacement policy — Random replacement, using the SCA algorithm defined in Appendix A. As can be seen here, a-NET demonstrates very high accuracy for this scenario. We leave an extensive review of this accuracy under different conditions to future work.

Impact of delay. a-NET was designed to compute estimates under the ZDD assumption. In Figures 2.15-2.14 we show the impact of adding propagation delay to the system on its precision. We add a small amount of *constant* delay to the system: the mean exogenous arrival rate at each node was 10 requests per time unit, and so we let query propagation be of length 1/20 and content propagation 1/10, to reflect that content is larger and thus might take longer move in the network. We also consider the impact when these values are doubled. Note that since a-NET does not take delay into account, adding delay only changes the simulation data, not the approximated values.

Fig. 2.14 shows the per-node performance, and Fig. 2.15 shows this after sorting the values in ascending order, independently for each delay. In Fig. 2.14 we can see that on a per-node basis, the delay at most of the nodes has not impacted performance to a large degree. Also, we take note that content custodians are placed at nodes $0, 5, 50, 55$[4], and that most of the large deviations as a function of delay take place at these nodes and their immediate surroundings. In Fig. 2.15 we can see a clear trend, that per-node approximation accuracy improves as the delay increases, lowering the miss probability of the nodes. While a majority of nodes still have an MPR above 1.0,

[4]These indices are off by one compared to Fig. 2.6, due to programming indices starting from 0.

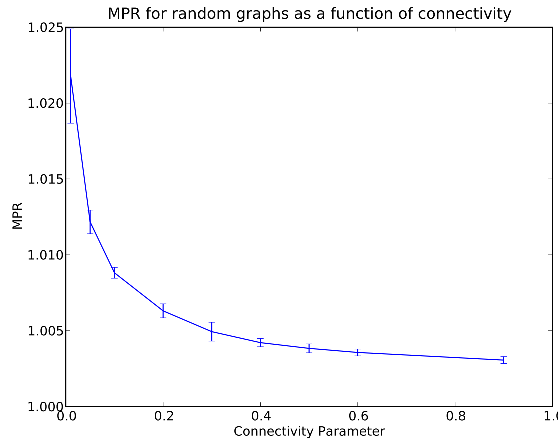


Figure 2.11: Mean MPR for random graphs over 400 nodes, as a function of p, the probability that each edge is in the network. The mean is taken over 10 simulations for each p, with 95% confidence intervals showing.

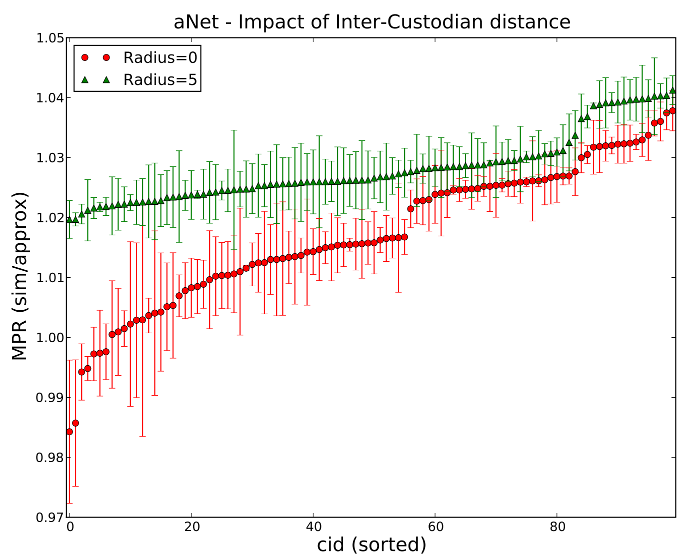


Figure 2.12: The impact of inter-custodian distance on a-NET, for 10x10 torus topologies and four custodians. Radius indicated the minimal distance between custodians, and values are sorted in ascending order. As seen here, the increased distance makes performance degrade.

and for these a-NET under-estimates the miss probabilities, it seems that with large delays a-NET might eventually become an upper bound on the number of misses in practice.

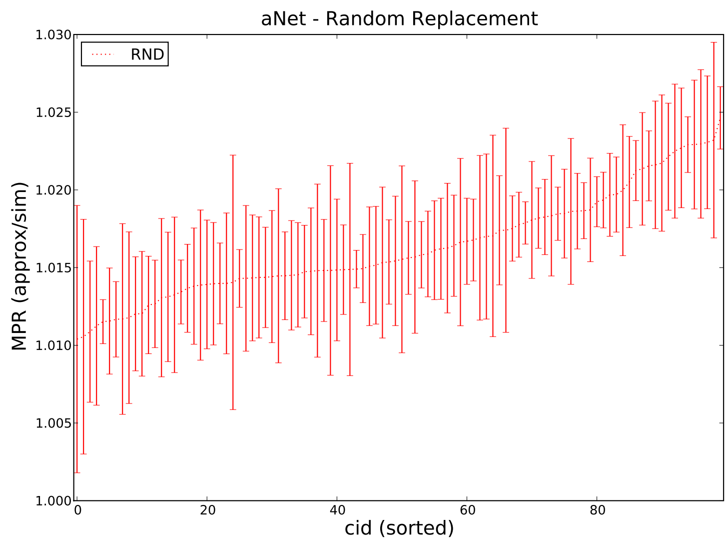


Figure 2.13: a-NET performance for Random Replacement, using the SCA Algorithm defined in Appendix A. As can be seen here, precision here seems to be higher than for LRU.

2.6 Analysis of performance-affecting factors

A close review of all the examples shown in the previous section reveals that, under the ZDD assumption, a-NET usually under-estimates the MPR per cache. In the next two sections we investigate the cause or causes for approximation error in a-NET (§2.6). By determining these causes, we can identify features of a cache network that can affect the performance of a-NET. Such analysis can also help determine the aspects to focus on when developing improved approximation tools in the future.

We argue that for any implementation of a-NET as shown in Algorithm 1, there are three[5] possible error-causing factors to consider for a-NET:

Prediction error of the SCA algorithm. The precision of the SCA algorithm(s) clearly impacts the precision of a-NET for the entire network.

Non-IRM flows. The SCA algorithms we use here were developed for IRM arrival streams, while the actual arrival stream at a cache contains dependencies. Since the arrival stream does not match the model for which the SCA algorithm was designed, this can cause approximation errors.

Error propagation / Input error. In each iteration of a-NET, the computation of the arrival streams is based on the misses computed in the previous iteration. Thus, errors in the preceding iteration (due to one or both of the causes mentioned previously) cause the input to the next iteration to be inexact, which in turn causes the prediction in the next iteration to be inexact as well. Thus, we can see that errors to the input of a cache propagate through the system with each iteration, which might cause small deviations at one cache to have considerable impact on the final system-wide estimate.

[5]Note that since we assume ZDD for this chapter, we ignore aspects that involve the download delay.

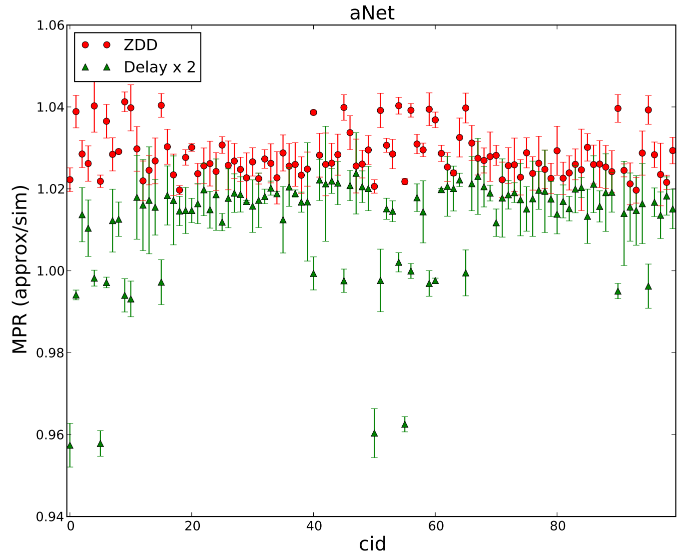


Figure 2.14: Per-node MPR for 10-by-10 torus networks, as a function of propagation delay. Delay for query propagation was set to be half the request arrival rate per-node, and content propagation double that. For the increased delay, the values are doubled. The nodes that are impacted the most by the introduction of delay are those close to the custodians (nodes 0, 5, 50, 55).

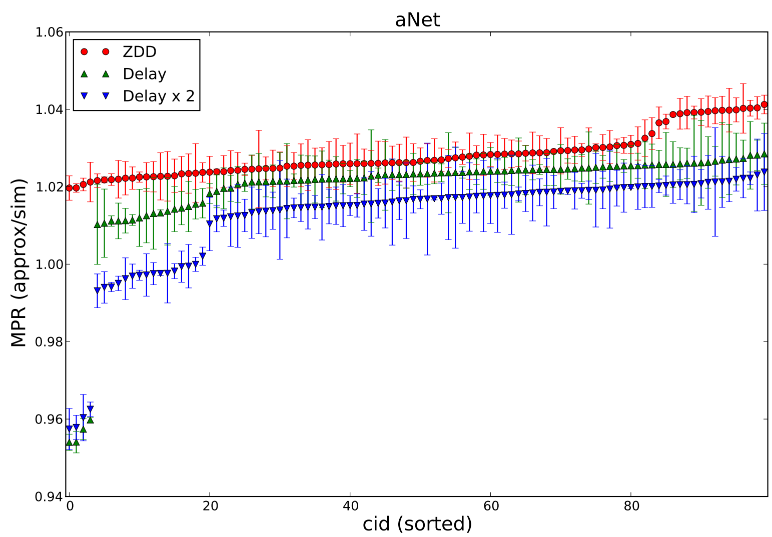


Figure 2.15: Per-node MPR for 10-by-10 torus networks, as a function of propagation delay. The arrival rates at each node were 10 requests/unit time; propagation delay of requests was 0.05 time units; and content download delay is 0.1 time units. For the increased delay, the propagation delay values are doubled. Values are sorted in ascending order to emphasize the fact that as the delay grows, the per-cache performance of the network seems to improve, as the miss-probability decreases.

We next present an analysis of these factors, isolating the effect of each factor on a-NET accuracy. For system performance, we again select the miss probability at a cache, though this methodology can be equally applied for other performance metrics, such as miss rates. Denote the miss probability of a given system by *sim*, and similarly let *aprx* denote the corresponding values for the a-NET approximation. The ratio *sim*/*aprx* is the prediction error. We then conduct the following three additional experiments to determine the impact of each of these factors:

Quantifying the impact of non-IRM traffic. Recall that our SCA algorithm (and hence a-NET) assume IRM arrivals at each cache. To evaluate the impact of this assumption we do the following. From the true (simulated) system we determine the distribution of requests $\boldsymbol{r}$ at all nodes. Note that no IRM assumptions were imposed on these request flows, which are a mixture of IRM exogenous flows and (possibly) non-IRM endogenous flows. We then perform a second simulation of each of the individual nodes in isolation, using $\boldsymbol{r}$ as input but generating arrivals in accordance with IRM. Thus, this second simulation (termed a *quasi-simulation*, and formally defined in Algorithm 2) is similar to the first, except that we have introduced IRM-ness into the combined arrival stream at each node. We then compare the miss probabilities of this second simulation to the original simulation. Both of these simulate the performance of the cache using the same arrival rates $\boldsymbol{r}$, only the first simulation makes no assumptions regarding dependencies in the stream, while the quasi-simulation receives IRM flows as input per cache. Thus, the only difference between these is the impact of IRM.

Quantifying the effect of error propagation. Our approach here is similar to that of our factor analysis of IRM. We once again determine $\boldsymbol{r}$, but this time give it as input to the SCA at each node. We call the result of this approximation a *quasi-approximation*, which is formally defined in Algorithm 3. We then compare the miss-probability of this approximation to that of a-NET. Both of these use the same SCA algorithm per node, which assumes IRM in the arrival stream. They differ only in the propagating error — a-NET uses the estimated arrival rates $\hat{\boldsymbol{r}}$ as input while the quasi-approximation uses $\boldsymbol{r}$.

Quantifying the impact of the SCA algorithm. Depending on the method of computing the misses at a cache given the arrivals, this method might introduce additional inaccuracies into the estimates of a-NET, especially when using an approximation algorithm for this purpose. Thus, we seek to quantify the effects/magnitude of the error introduced by the SCA algorithm we used in our work. From the true (simulated) system we again determine the request rates $\boldsymbol{r}$. We then adopt the IRM assumption to drive both a simulation of, and an SCA computation of, each node in isolation and examine the ratio of these performance computed via these two techniques. In other words, we compare the performance of the quasi-simulation to that of the quasi-approximation. In this case, the only difference at each node is the manner in which individual cache performance is computed (by the approximate SCA or by simulation, both assuming IRM), as the inputs are identical.

Algorithm 2 Computing Quasi-simulation

Input: $G, \boldsymbol{r}, c, simIRM$

for i=1 to N **do**
  $(\hat{s}_{i1}, ..., \hat{s}_{iL}) := simIRM(c, r_{i1}, ..., r_{iL})$
end for
RETURN $\hat{\boldsymbol{s}}$ // With $\boldsymbol{r}$ and $\hat{\boldsymbol{s}}$, compute performance

---

Formally, for node v_i $sim(v_i)$ denotes the simulated miss probability at v_i, and denote similarly for a-NET, the quasi-simulation and quasi approximation with $aprx(v_i)$, $q\text{-}sim(v_i)$ and $q\text{-}aprx(v_i)$ respectively. Above we explained how

- $\frac{sim(v_i)}{aprx(v_i)}$ quantifies the approximation error;

Algorithm 3 Computing Quasi-approximation

Input: $G, \boldsymbol{r}, c, alg$

for i=1 to N **do**
  $(\hat{s}_{i1}, ..., \hat{s}_{iL}) := alg(c, r_{i1}, ..., r_{iL})$
end for
RETURN $\hat{\boldsymbol{s}}$ // With $\boldsymbol{r}$ and $\hat{\boldsymbol{s}}$, compute performance

- $\frac{q\text{-}sim(v_i)}{q\text{-}aprx(v_i)}$ quantifies the SCA error;
- $\frac{q\text{-}aprx(v_i)}{aprx(v_i)}$ quantifies the propagating error;
- $\frac{sim(v_i)}{q\text{-}sim(v_i)}$ quantifies the non-IRM error;

and we can easily see

$$\frac{sim(v_i)}{aprx(v_i)} = \frac{sim(v_i)}{q\text{-}sim(v_i)} \times \frac{q\text{-}sim(v_i)}{q\text{-}aprx(v_i)} \times \frac{q\text{-}aprx(v_i)}{aprx(v_i)} \tag{2.5}$$

Which can be interpreted as

$$\text{a-NET error} = \text{IRM error} \times \text{SCA error} \times \text{Propagating error.}$$

Let us next consider a few examples of this methodology at work. Recall that we are focused on the *miss probability ratio*, abbreviated MPR, the ratio of the miss-probability of each cache according to the different simulations and approximations.

The results for both torus and tree topologies, when using the SCA algorithm presented in [14] and described in Appendix A, are shown in Figures 2.16 and 2.17, respectively. In both of these we can see that, once removing propagating and IRM-based errors, the error all but disappears (i.e., the ratio is close to 1.0). More importantly, the strongest impact on approximation error is that of non-IRM traffic — the dependencies within the miss stream.

One implication of this observation is that in networks where the dependencies in the endogenous flow are lessened, the approximation error will decrease. Recall that

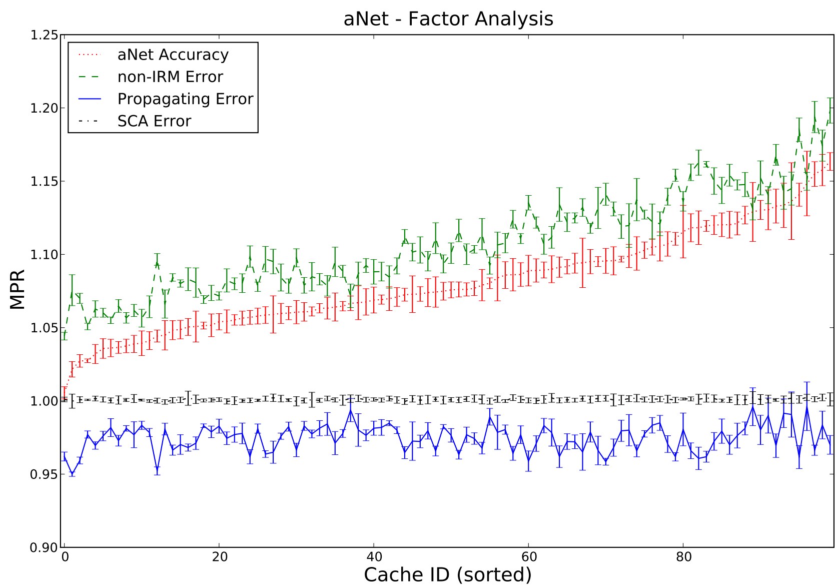


Figure 2.16: Example of analyzing the impact of error factors on a-NET, for the plot shown in Fig. 2.5. As in said figure, we consider performance of a 10-by-10 Torus topology with four custodians, each holding a quarter of 500 files. Requests arriving at each node are distributed according to Zipf distribution. 90% confidence intervals show. The results are plotted in ascending sim-to-approx order. As can be seen here, the non-IRM traffic is the major contributor to approximation error

we saw earlier that the performance of a-NET improves for trees as we increase the branch factor k of the tree. The incoming flow at each node in a tree is a mixture of the miss flows from lower down in the tree, and specifically note that these flows are independent of one another[6]. As k grows, this incoming flow is a mixture of more mutually-independent flows, which reduces the intra-flow dependencies, and as k goes to infinity, we get closer to a purely IRM request stream at v_i. This hypothesis is supported by the results shown in Figure 2.10 for k-ary trees. The results show clearly that as the branch factor increases, so too does the accuracy of a-NET.

[6]This is true since each of v_i's children does not share any descendants with any other child of v_i, and we are assuming ZDD.

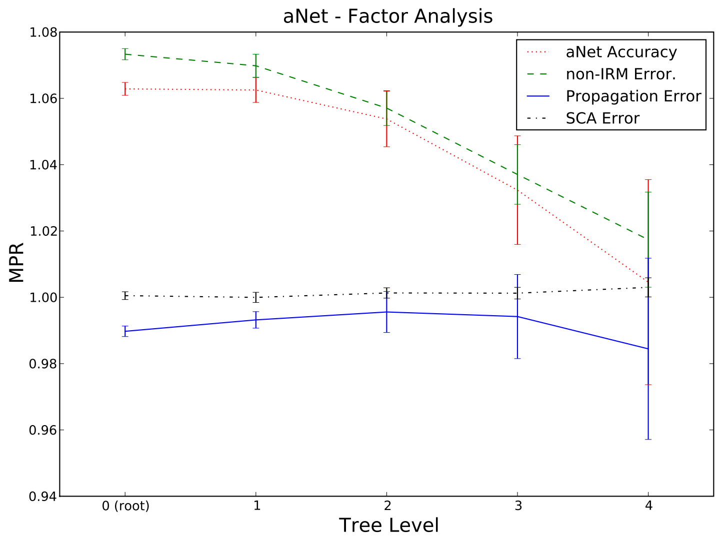


Figure 2.17: Example of analyzing the impact of error factors on a-NET, for a cache hierarchy - a 4-level binary tree. As can be seen here, the non-IRM traffic is the major contributor to approximation error

2.7 summary

In this chapter, we presented a-NET, an algorithm for approximating the performance of cache networks of arbitrary topology and scale. Given an SCA algorithm for each of the replacement policies used in the network, a-NET can also compute performance estimates for heterogenous networks.

An important note to make is that a-NET can support additional stream representations, and these might greatly affect the accuracy of a-NET. In the implementation we considered here, the SCA accepted as input the arrival rates at each cache and gave miss rates as output, thus implicitly assuming IRM. If, however, an SCA were to be developed that considered parameters that expressed the locality of reference within the arrival stream in addition to the rates, a-NET could be used with this as well, and might indeed produce more accurate results than shown here.

Since, in our experiments, the inherent error in the SCA algorithms we used (See appendix A) was very small for IRM traffic, our experiments can also be viewed as indicating the impact of non-IRM flows on the performance of LRU caches within a network. Our results thus join observations made by others, that the miss stream of an LRU cache does is not suitable for efficient caching as is. This phenomenon has been widely observed in the literature, leading some to propose *heterogenous* architectures, where neighboring caches use different replacement policies [8, 22, 71, 73]. This highlights the importance of mixing the miss flows, as we have shown in addressing the impact of node degree on cache performance.

As for future work, we have constructed a Markov model for characterizing the miss stream of a cache, to further understand the difficulty posed for caching by the miss stream of a cache. While some progress has been made in characterizing the miss stream for Zipfian arrival streams [28], little is understood about the miss stream under arbitrary arrival loads. Our model can compute the *inter-arrival distance* between requests in the miss stream for any IRM arrival stream for both LRU and FIFO replacement, from which we can determine certain properties of the next-hop cache performance. We believe that with the insights gained from this model, better understanding (and, hopefully, better cache network architectures) could be developed.

CHAPTER 3

A NETWORK CALCULUS FOR CACHE NETWORKS

3.1 Introduction

The previous chapter highlights the difficulty in approximating efficiently the performance of a cache network. The endogenous flows that flow within the network have complex dependencies, and this mismatch with the simple IRM flow model expected by the SCA algorithm causes approximation error. In this chapter, we address this challenge by developing an alternate approach — a *network calculus* for computing deterministic bounds on the request flows that move through the cache network. We show how these flow bounds can then be used to calculate performance bounds for metrics such as the cache miss rate for a given piece of content at a given network cache. Our work is inspired by Cruz's pioneering network delay calculus [13] for deterministically bounding flows in general queueing networks, which later led to new bounding techniques [46,63,64,72] and found use in fields beyond classic queueing networks, including sensor networks [31,62], smart grids [45] and anomaly detection [56]. While flows in a network delay calculus represent units of work routed among queues, flows in a cache network represent content requests routed among caches. Here, a request may either be satisfied at a cache (and the content subsequently stored at downstream caches as requested content is returned to the requestor) or forwarded upstream to another cache in the event of a cache miss. Queueing networks and cache networks thus have many fundamental differences.

Our work makes several important contributions.

1. We define an upper-bound characterization of a stream of requests at a cache, and highlight differences between cache networks and queueing networks.

2. We develop a calculus for computing bounds on the miss stream of an LRU cache, given bounds on the incoming request stream. We show that these bounds are tight and consistent, i.e., that the upper bound can be realized for all files simultaneously.

3. We use this calculus to gain analytical insights into the behavior of LRU caches in isolation, and in networks. We identify the uniformizing effect of LRU on the request stream, and the impact of cache and topology diversity on system performance.

4. Using an iterative fixed-point approach similar to a-NET, we use this calculus to study LRU replacement in cache networks with arbitrary topology. Our results indicate that these bounds can be close-to-tight for realistic network scenarios.

More generally, we believe our work represents an important step forward in developing performance models for emerging content-centric networks, as well as other systems in which an interconnected network of caches provides efficient, scalable content distribution.

The remainder of this chapter is organized as follows. In Section 3.2 we present related work. In Section 3.3 we present our flow model, and define the notion of bound *tightness*. In Section 3.4 we present theorems on bounding the number of cache misses over a *finite window*, and formulate theorems bounding the miss stream *flow* in Section 3.5. These theorems reveal analytical properties of LRU's impact on request flows. In Section 3.6 we use our calculus to study the performance of cache networks and evaluate bound tightness. We conclude with a summary of our results and discussion of future work.

3.2 Related Work

Beginning with Cruz's pioneering network delay calculus [13], numerous researchers have developed both deterministic and stochastic calculi for bounding the performance of networks of queues [46]. Networks of queues, where units of work proceed from one queue to another are quite different from networks of caches, which perform a filtering function, only forwarding cache misses on to the next hop towards a custodian.

A number of efforts have adopted a bounding approach, similar to network delay calculus for analyzing systems with complex, time-varying, stochastic flows. These efforts have analyzed energy flows in smart grid systems [45, 70] and energy harvesting/expenditure in wireless sensor networks [31]. While the flows in these systems are characterized by (σ, ρ) bounds, the behavior of individual components through which these flows pass, and the manner in which the bonded flows are transformed, are quite different from cache networks.

In our bounding model, the rate component is used to indicate the popularity of content while burstiness is used to bound the variation in arrival rates. Other models for flows in the network can be considered. Fonseca et. al. [18] proposed characterizing a stream using two different measures: the popularity *distribution* was characterized as a whole by its entropy, with lower entropy indicating skewed distributions, and the inter-arrival distribution for the entire stream was expressed via coefficient of variation. While their approach can be helpful in analyzing existing streams (as proposed in [18]), in its current form it does not differentiate between files, nor does it readily lend itself for computing the impact of LRU on the flow.

3.3 A (ρ, σ) Model for Cache Networks

In this chapter we continue using the model presented above in Section 2.2 for CNs.. In this section we present the model we use here for flow bounding (§3.3.1).

We define the concept of *tight bounds* as used in this chapter (§3.3.2), and conclude with an expositional example of bounds on the miss stream (§3.3.3).

3.3.1 Bounding Model

In this work we adopt the flow model proposed by Cruz [13]. For a stream of events over time let $R(t)$ be the number of events that took place at time slot t. These events can be jobs or packets in queueing networks, or content requests in cache networks. In this work, we consider the latter. For a stream $R(t)$, (ρ, σ) is a deterministic bounding representation of a stream, if for any interval $[t_1, t_2)$, $0 \leq t_1 \leq t_2 \in \mathbb{R}$,

$$\int_{t_1}^{t_2} R(t) \leq \lceil \rho(t_2 - t_1) + \sigma \rceil \tag{3.1}$$

Note that we take the ceiling of the bound since arrivals are binary in nature — a request either arrives or does not arrive during some window, yet $\rho(t_2 - t_1) + \sigma$ can be any real number.

In queueing networks, the standard interpretation of ρ is the average arrival rate per time unit, and σ indicates the "burstiness" component of the stream, as the bounds allow σ packets to arrive irrespective of the size of the window. In Cruz's work [13] this bounding property was denoted as $R \sim (\sigma, \rho)$. However, as we shall see later on, cache networks differ from queueing networks in that the rate component dominates the impact on the miss stream. We express this by modifying the notation slightly, and use instead $R \sim (\rho, \sigma)$.

The key result in [13] for queueing networks is that if each input flow j (corresponding to a source-destination node pair in a queueing network) has a $(\rho_{j,in}, \sigma_{j,in})$ characterization, then its output flow has a $(\rho_{j,out}, \sigma_{j,out})$ characterization that can be computed as a function of the input characterizations at that node, $\{(\rho_{k,in}, \sigma_{k,in})\}_{k=1}^{L}$ for L input flows (see Figure 3.1). These output flows are then the input flows at the subsequent network nodes, and in this manner, per-flow bounding characterizations can be "pushed" through feed-forward networks. For non-feedforward networks, a system of simultaneous equations can be established and solved. Here we develop a calculus for cache networks, specifically those employing LRU caches, which is the policy of choice for many ICN architectures.

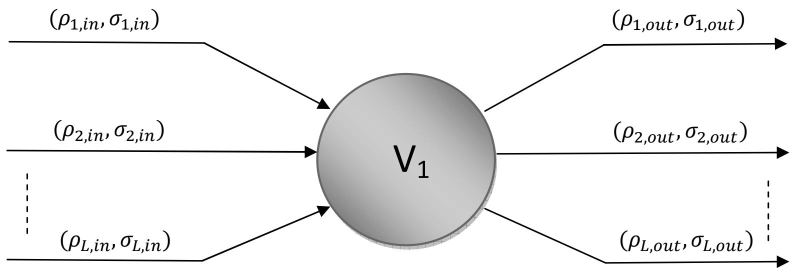


Figure 3.1: Network calculus - high-level depiction of flow-bounds "entering" the cache and miss-flow bounds "leaving" the cache.

3.3.2 Bound tightness

Next, we address how to select the bound for a given stream. Since (ρ, σ) is only an upper bound, there are an infinite number of bounds for any given stream: for example, if (ρ, σ) is a bound for $R(t)$, then for any positive $\Delta_\rho, \Delta_\sigma$ $(\rho + \Delta_\rho, \sigma + \Delta_\sigma)$ is also a bound for $R(t)$. Thus, we define the following concept of bound tightness:

Definition 5. For a given (ρ_j, σ_j) bound for f_j requests we will say that it is *globally-tight* if (a) $\lim_{t\to\infty} \frac{1}{t} \int_{t'=0}^{t} R_j(t')dt' = \rho_j$, i.e., if ρ_j is the average rate of requests, and (b) if $\lceil \sigma_j \rceil \geq 0$ is minimized given that ρ_j.

Lemma 6. *ρ is minimized over all bounds when the bound is globally-tight.*

Proof: Let (ρ_g, σ_g) be the globally-tight bound, and (ρ, σ) be any other bound for $R(t)$. By construction, $\lim_{t\to\infty} \frac{1}{t} \int_{t'=0}^{t} R(t')dt' = \rho_g$. Additionally, using equation 3.1 we conclude our proof:

$$\begin{aligned} \int_{t'=0}^{t} R(t') &\leq \lceil \rho t + \sigma \rceil \leq \rho t + \sigma + 1 \\ \lim_{t\to\infty} \frac{1}{t} \int_{t'=0}^{t} R(t') &\leq \lim_{t\to\infty} \rho + \frac{\sigma + 1}{t} \\ \rho_G &\leq \rho \end{aligned}$$

■

In what follows we shall prove that globally-tight bounds always exists for boundable flows, and these are the bounds we shall compute. We select these bounds because they support the interpretation of ρ as the long-term mean arrival rate, which is convenient in several contexts. For example, for a given arrival stream characterization, we can compute ρ by computing the mean arrival rate.

3.3.3 Bounds at work: an example

We conclude this section with a simple example of bounding a miss stream, to give the reader some intuition regarding the impact of caches on request streams, specifically regarding how these streams are characterized using the (ρ, σ) model. We use this example to draw distinctions between the manner in which queueing networks and cache networks behave.

For a given cache v, let $R_{j,in}(t)$ be the arrival rate for q_j requests at v, and let $R_{j,out}$ be the miss rate for q_j at v, for all $1 \leq j \leq L$. Assume that these flows are bounded as follows:

$$\begin{aligned} R_{1,in} &\sim (\rho_{1,in}, 0),\ \ \rho_{1,in} > 0 \\ \forall 2 \leq j \leq L\ \ R_{j,in} &\sim (0, \sigma_{j,in}),\ \ \sigma_{j,in} > 0 \end{aligned}$$

Aside from the f_1 flow, the remaining streams will consist of a finite number of requests over an infinite window of time. Consider now the case where the cache size is $c = 1$. The miss stream is maximized for all request streams when the arrival stream is an alternating sequence of requests, i.e.,

$$f_1, f_{\neq 1}, f_1, f_{\neq 1}, f_1, f_{\neq 1}, ...$$

In this sequence, all q_j for $j \neq 1$ will generate a miss, and we will also have $\sum_{j=2}^{L} \sigma_{j,in} + 1$ misses for f_j (we add the 1 for the first request of f_1). After all these misses take place, all requests for f_1 will generate cache hits. Thus, we move from arrivals $R_{1,in} \sim (\rho_{1,in}, 0)$ to misses $R_{1,out} \sim (0, \sum_{j=2}^{L} \sigma_{j,in} + 1)$.

We glean several insights from this example. First, note that the ρ component has disappeared in the miss stream, and that a σ component (that did not exist in the input stream) has appeared instead. Thus, there is no conservation of flows in this model, nor is there no conservation of ρ or σ individually.

A second observation is that the miss stream of f_1 is bounded by the combined arrival streams for the other files. A cache miss for f_1 occurs only if requests for other files caused f_1 to be evicted from v before the next f_1 request arrived. This underscores the difference between queueing and cache networks. In the former, an increase in traffic of flow i might decrease the rate of flow j by causing flow j packets to be dropped; in the latter, an increase in flow i can have the opposite effect, by causing evictions of f_j and subsequent additional misses.

One final and critical insight here is regarding the interpretation of σ. In the context of a queue, the worst case usually occurs when a large *burst* of jobs arrives at the queue at the same time. Thus, in queueing networks, the σ component is commonly referred to as the *burstiness* component. However, in the example we show here, the worst case is when the $\sigma_{j,in}$ requests for f_j arrive spaced out, to generate maximum misses at the cache w.r.t. f_1 and f_j. Additionally, note that the miss stream of f_1 has a positive σ component, despite the fact that the miss stream is only a thinning of the non-bursty arrival process. Thus, a more convenient way to think of the σ component in cache networks is as a set of requests, each of which can arrive *at any time* for any given window, without positioning constraints. Despite this change, in what follows we shall stick with the conventional terms and refer to ρ and σ as the rate and burstiness components, respectively.

3.4 Computing Worst-Case Bounds for finite windows

In this section, we describe how to bound the miss stream for each file over a window $w = [s, t)$. For each file we are given the number of requests that arrive during w, and then we compute a bound on the number of misses per file during w.

w	a window of time
$\mathcal{T}_j$	f_j request stream
$\mathcal{T}$	combined arrival stream
$I(w,j)$	num. of arriving q_js during window w for $\mathcal{T}$
$O(w,j)$	num. of q_j misses during window w for $\mathcal{T}$
$M_{w,j}$	max. num. of miss sets for f_j during w
M_w	max. num. of miss sets during w
$\hat{M}_j$	max. miss rate for f_j

Table 3.1: Table of Notation

3.4.1 Notation and Preliminaries

We begin with notation, as summarized in Table 3.1:

- $\mathcal{T}_j = (t_{j,1}, t_{j,3}, t_{j,3}...)$ is a (possibly infinite) monotonically increasing sequence of times for f_j requests.

- $\mathcal{T}$ denotes a sequence of file requests arriving at a specific cache. Formally, $\mathcal{T} = \{\mathcal{T}_j\}_{j=1}^{N}$.

- The *outcome* of a request for f_j at time t is a *hit* if at that time $f_j \in v$, and a *miss* otherwise.

- For a window w and request sequence $\mathcal{T}$, let $I(w, j)$ be the number of q_js in w, and $O(w, j)$ the number of misses. Note that for deterministic cache replacement policies, the misses can be computed as a function of cache contents at the outset of w and $\mathcal{T}$.

Definition 7. For any file f_j, requests for f_i where $j \neq i$ are said to be *interfering* with respect to f_j.

Definition 8. A *miss set* $s \subseteq F$ for a cache of size c is a multi-set of requests for at least $c + 1$ unique files, where we omit the dependence of s on c for notational convenience. A *miss sequence* $\overrightarrow{s}$ is any ordered miss set. A *miss sequence for* f_j $\overrightarrow{s_j}$ is a miss sequence containing one or more requests q_j, such that these requests make up the sequence suffix. Formally, let X_i be the ith request in miss sequence $\overrightarrow{s}$, and let $k > c$ be the length of the sequence. $\overrightarrow{s}$ is a miss sequence for f_j iff there exists an index $c < h < k$ s.t.

$$\forall \quad i \leq h \quad X_i \neq q_j$$
$$\forall \quad i > h \quad X_i = q_j$$

For example, (q_1, q_2, q_3, q_3) is a miss sequence for f_3 when $c = 2$, but not (q_1, q_3, q_2, q_3).

A miss sequence for f_j is so named for the following reason. For a window w, if the arrivals during w form a miss sequence for f_j, there is a single miss for f_j which occurs at the first q_j in w. Since a miss set is a multi-set, it has the following property:

Property 9. *Let s be a miss set s.t. $q_j \in s$. Then*

- $s' := s \cup \{q_j\}$ *is a miss set.*

- $s' := s \setminus \{q_j\}$ *is a miss set if* $q_j \in s'$.

Finally, a note about ZDD. As in the previous chapter, we assume in our discussion here that ZDD applies. As a result, the order and time of requests is the same as the order and time of the corresponding content arriving at the cache. However, we demonstrate in Section 3.4.3 that (at least when we adopt the CCN approach for miss forwarding[1]), the bounds computed for ZDD apply also when ZDD is not assumed and download delay can be positive. Thus, the bounds computed here are applicable also to realistic systems.

3.4.2 Bounds over window w

We begin with bounding $O(w, j)$, given $\mathcal{T}$. For a given window w, denote with W a partition of w, $W = \{w_1, ..., w_l\}$ s.t. $w = w_1|w_2|...|w_l$. (| indicates concatenation).

Lemma 10. *For a given request sequence $\mathcal{T}$ and window w, and assume f_j is the most recently requested file at the beginning of w, $O(w, j)$ equals the maximal number of $w_k \in W$ for any partition W of w s.t. the requests in w_k form a miss sequence for f_j.*

Proof: With LRU, a cache miss occurs for f_j iff c interfering requests for unique files arrive at the cache between two consecutive requests for f_j. Including the q_j request at the end of this sequence, we get a miss sequence. A any additional requests for f_j at the end of the sequence retain the definition as a miss sequence, yet generate only hits as the file is still in the cache. Note that since we assume f_j was the most recently requested file when w starts, the first cache miss for this file will also occur only after it is evicted by the first c requests. ■

Note that any partition of w defines also a partition of the arrivals over w into disjoint sets. In what follows, we will say that miss-sets are disjoint if each is contained

[1] Recall the CCN approach states that if several requests for the same content arrive at a node and are misses, only the first is routed on, and when the content is downloaded to this node, it is then forwarded in all the directions from which requests came in.

in a different window in some partition of w. We next consider the case where the exact sequence $\mathcal{T}$ is unknown, and we are only given the *number* of arrivals $I(w,j)$ over w for all $1 \leq j \leq L$. Then, for each arrangement of these arrivals, the miss sequence can be different. Denote

- $M_{w,j}$ as the maximum number of disjoint miss-sets for q_j in any arrangement and partition of arrivals over w.
- M_w as he maximum number of disjoint miss-sets in any arrangement and partition of arrivals over w.

Note that M_w is not necessarily equal to $\max_j M_{w,j}$. To see this, consider a case where $c = 2$ and a sequence of requests over w consisting of a single request for each of $f_1, ...f_9$. In this scenario, $M_w = 3$, while for all j $M_{w,j} = 1$.

Using these definitions, the following corollary of Lemma 10 follows immediately:

Corollary 11. *Given $I(w,j)$ for all $1 \leq j \leq L$, $O(w,j) \leq M_{w,j} + 1$, and there exists a sequence $\mathcal{T}$ for which this bound is reached.*

Proof: The first request for f_j might be a miss regardless of preceding requests in w, for example if it is not in the cache at the beginning of w. After this request, from Lemma 10 we have an equality between the number of misses and the number of miss sequences for the given arrival sequence. Since $M_{w,j}$ is the maximal number of miss sequences for f_j in any arrival sequence, $M_{w,j} + 1$ thus bounds the misses for f_j in this scenario, and is reachable for some sequence. ■

Next, in the main result of this section, we quantify $M_{w,j}$:

Theorem 12.

$$M_{w,j} = \min\{I(w,j), M_w\} \tag{3.2}$$

Proof: Assume there is an arrangement and partition for which there are M_w miss-sets. If $I(w,j) \leq M_w$, from the pigeonhole principle we can move q_js so that each is in a different miss-set. If there is a miss-set with duplicates, from Property 9 it can be moved to a set with no q_j without changing the number of miss-sets. Thus we get $M_{w,j} = I(w,j)$. Otherwise $I(w,j) > M_w$, and using the same argument we can move q_js between sets until each has at least one q_j, in which case $M_{w,j} = M_w$, which concludes our proof. ∎

3.4.3 Bounds and Download Delay

We complete this section by proving that our bounds apply also for the non-ZDD case.

Theorem 13. *The upper-bound on the miss stream for a ZDD system is also a bound on the miss stream for non-ZDD systems, for the CCN approach to request forwarding*[2].

Proof: When assuming ZDD, $\mathcal{T}$ represents the arrival stream for both requests and content. In a non-ZDD system, however, $\mathcal{T}$ represents the arrival process for requests, while the content arrives at some later point in time. Consider then a sequence of requests $\mathcal{T}$ arriving over a window w_1, and let $w = w_1|w_2$ be the window during which all the requests from w_1 are satisfied at this node.

First, we note that requests q_j that arrive between a request q_j and corresponding download of f_j have no impact on the miss stream (as they are not forwarded in the CCN approach), so we ignore these intermediate requests temporarily. We therefore associate each forwarded miss with a corresponding subsequent file download.

Next, we note that with LRU, cache state is only affected by file arrivals and cache hits. As a result, the time between a cache miss and its corresponding content has no impact on cache behavior. For example, if we kept the file arrivals as is and changed

[2]Recall the CCN approach states that if several requests for the same content arrive at a node and are misses, only the first is routed on, and when the content is downloaded to this node, it is then forwarded in all the directions from which requests came in.

the arrival time of the corresponding request, the number of misses over w would not change.

Consider therefore the scenario where we make the arrival time of these requests equal to the download time of content. This new configuration generates the same number of misses for the requests originally arriving in w_1, while abiding by ZDD. From Lemma 14 (see below) we know that including the misses we ignored earlier cannot decrease the number of misses over w. Thus, while the misses are spread over a larger window (w instead of w_1) in this new configuration, the total number of misses remains the same, so the mean arrival rate does as well.

Next, we return to the requests we ignored earlier. These misses would be forwarded on to the next cache if ZDD were assumed. By increasing the number of misses at the next cache, this will increase the number of arrivals at neighboring caches, which again by Lemma 14 does not result in lower miss bounds, which concludes our proof.

■

3.5 Computing (ρ, σ) bounds on the miss stream

In this section we leverage the bounding techniques for finite windows to generate (ρ, σ) bounds on the miss stream, given bounds on the incoming stream. We begin with the following lemma, which states that increasing the number of arrivals over w for file f_j does not lower the bounds on the number of misses for file f_k:

Lemma 14. *For all $1 \leq i, j \leq L$, $M_{w,i}$ monotonically increases with $I(w, j)$.*

Proof: Increasing $I(w, j)$ can only increase the number of disjoint miss sets that can be constructed. So, from Equation 3.2 we get that $M_{w,i} = \min\{I(w, i), M_w\}$ monotonically increases. ■

We will therefore assume that $I(w, j)$ *equals* the arrival bounds over w, and say that the bounds are *tight over w*. Let $\hat{M}_w$ be M_w when for all $1 \leq j \leq L$, $I(w, j)$ is tight over window w, and similarly regarding $\hat{M}_{w,j}$. Since we assume the bounds are tight, and from Equation 3.1 the only parameter that impacts the bounds is the window size, the following lemma directly follows:

Lemma 15. *If* $|w| = |w'|$, $\hat{M}_w = \hat{M}_{w'}$ *and* $\hat{M}_{w,j} = \hat{M}_{w',j}$.

3.5.1 Bounding the miss rate

To compute bounds on $\rho_{j,out}$, we first show that the "burstiness" parameters $\{\sigma_{i,in}\}_{i=1}^{L}$ do not impact $\rho_{j,out}$, as they constitute only a finite number of requests:

Theorem 16. *Let* $\mathcal{T}, \mathcal{T}'$ *be two request streams with corresponding sets of globally-tight bounds,* $\{(\rho_{j,in}, \sigma_{j,in})\}_{j=1}^{N}$ *and* $\{(\rho'_{j,in}, \sigma'_{j,in})\}_{j=1}^{N}$*, such that for all* $1 \leq j \leq L$ $\rho_{j,in} = \rho'_{j,in}$*. Let* $\{(\rho_{j,out}, \sigma_{j,out})\}_{j=1}^{N}$ *and* $\{(\rho'_{j,out}, \sigma'_{j,out})\}_{j=1}^{N}$ *be the corresponding globally-tight bounds on these miss streams. Then, for all* $1 \leq j \leq L$, $\rho_{j,out} = \rho'_{j,out}$.

Proof: W.l.o.g., assume $\sigma_{j,in} = 0$ for all $1 \leq j \leq L$, and consider the q_j miss stream over some window w. We use the subscripts $\mathcal{T}$ and $\mathcal{T}'$ to distinguish between the different streams. Since we assume that the bounds are tight over w, we set $I_{\mathcal{T}}(w, j) = \rho_{j,in}(t - s) + \sigma_{j,in}$ and $I_{\mathcal{T}'}(w, j) = \rho'_{j,in}(t - s) + \sigma'_{j,in}$. W.l.o.g. assume that these values are integers, so we drop the rounding operation. Then, the difference in the total volume of the arrival streams is

$$\begin{aligned}\Delta &= \sum_{j=1}^{L} I_{\mathcal{T}'}(w, j) - \sum_{j=1}^{L} I_{\mathcal{T}}(w, j) \\ &= \sum_{j=1}^{L} (\rho'_{j,in} - \rho_{j,in})(t - s) + \sigma'_{j,in} - \sigma_{j,in} = \sum_{j=1}^{L} \sigma'_{j,in}\end{aligned}$$

For a given sequence $\mathcal{T}$, each request can belong to at most a single miss-sequence w.r.t. f_j. Thus, if we maximize the number of miss-sequences for both streams, Δ

bounds the difference between the number of misses — $\left|O_{\mathcal{T}'}(w,j) - O_{\mathcal{T}}(w,j)\right| \leq \Delta$. Since Δ is independent of the window size and $O_{\mathcal{T}}(w,j) = \int_{u=s}^{t} R_{j,out}(u)$ we get

$$\left|\int_{u=s}^{t} R'_{j,out}(u) - \int_{u=s}^{t} R_{j,out}(u)\right| \leq \Delta$$

$$\left|\frac{1}{t-s}\left(\int_{u=s}^{t} R'_{j,out}(u) - \int_{u=s}^{t} R_{j,out}(u)\right)\right| \leq \frac{\Delta}{t-s}$$

We take the limit for $t \to \infty$, and recall the definition of globally-tight bounds,

$$\left|\rho'_{j,out} - \rho_{j,out}\right| \leq 0$$

$$\rho'_{j,out} = \rho_{j,out}$$

■

Based on Theorem 16, we shall thus assume in this section that the σ components of all arrival streams are 0. We next turn to bound the rate of the miss stream — the (ρ, σ) version of Theorem 12.

Theorem 17 (ρ bounds)**.** *Let $\hat{M} = \lim_{|w|\to\infty} \hat{M}_w/|w|$. Then the globally-tight bound on the mean miss rate for f_j is*

$$\rho_{j,out} := \min\{\rho_{j,in}, \hat{M}\}$$

and there exists a $\sigma_{j,out}$ for which $(\rho_{j,out}, \sigma_{j,out})$ is a bound for the miss stream.

Proof: From Theorem 12 we know $M_{w,j} = \min\{I(w,j), M_w\}$. For tight bounds and $\sigma = 0$, $I(w,j) = \lceil |w|\rho_{j,in}\rceil$, which results in

$$\hat{M}_{w,j} = \min\{\lceil |w|\rho_{j,in}\rceil, \hat{M}_w\},$$

$$\min\{|w|\rho_{j,in}, \hat{M}_w\} \leq \hat{M}_{w,j} \leq \min\{|w|\rho_{j,in} + 1, \hat{M}_w\}.$$

Dividing by $|w|$ and taking the limit as the window size approaches to infinity we get

$$\min\{\rho_{j,in}, \hat{M}\} \leq \lim_{|w|\to\infty} \frac{\hat{M}_{w,j}}{|w|} \leq \min\{\rho_{j,in} + \lim_{|w|\to\infty} \frac{1}{|w|}, \hat{M}\}$$

and using this sandwich argument we get

$$\lim_{|w|\to\infty} \frac{\hat{M}_{w,j}}{|w|} = \min\{\rho_{j,in}, \hat{M}\} \tag{3.3}$$

Finally, if we maximize the number of misses per window, we get that regarding the miss process

$$\begin{aligned}
\int_w R_{j,out}(t) &= \hat{M}_{w,j} + 1 \\
\frac{1}{|w|}\int_w R_{j,out}(t) &= \frac{\hat{M}_{w,j} + 1}{|w|} \\
\lim_{|w|\to\infty} \frac{1}{|w|}\int_w R_{j,out}(t) &= \lim_{|w|\to\infty} \left(\frac{\hat{M}_{w,j}}{|w|} + \frac{1}{|w|}\right) = \lim_{|w|\to\infty} \frac{\hat{M}_{w,j}}{|w|}
\end{aligned}$$

and by the definition of global tightness we get

$$\rho_{j,out} = \lim_{|w|\to\infty} \hat{M}_{w,j}/|w| \tag{3.4}$$

and from Eq. 3.3 and 3.4 we conclude that $\rho_{j,out} = \min\{\rho_{j,in}, \hat{M}\}$.

We have shown here how to compute the per-stream miss rate over an infinite horizon. However, our bounds must hold as well for *all* windows, and so we must demonstrate next that this bound can be used with a *finite* burstiness component for all windows. To show this, we prove next that $\hat{M}_{w,j}$ monotonically increase as $|w|$ grows, and since $\rho_{j,out}$ is computed for an infinite window the argument is proven.

Let w, w' be two windows s.t. $|w| = k \cdot |w'|$ for some integer $k > 1$. Since we can assume the burstiness component is zero in the arrival streams, the number of requests arriving in w is exactly k times that of what arrives in w'. Denote $|w'| = \delta$ and $w = [s, t)$. Then,

$$\begin{aligned}
\tfrac{1}{|w|}\hat{M}_{w,j} \quad &\geq_{(*)} \frac{1}{|w|}\sum_{h=0}^{k-1}\hat{M}_{[s+h\delta, s+(h+1)\delta),j} \\
=_{(**)} \quad &\frac{1}{k|w'|}\sum_{h=0}^{k-1}\hat{M}_{w',j} = \frac{1}{k|w'|}k\hat{M}_{w',j} = \frac{1}{|w'|}\hat{M}_{w',j}
\end{aligned}$$

Inequality (*) is a result of the fact that we can construct miss sets for w by iteratively doing so for each $[s + h\delta, s + (h + 1)\delta)$ separately. Equality (**) is based on Lemma 15. From this derivation we see that $\frac{1}{|w|}\hat{M}_{w,j}$ is monotonic with the increase of the window size. Thus we know that $\rho_{j,out}$ can be applied to every window for some constant σ, which concludes our proof. ■

Discussion. Theorem 17 reveals some interesting properties of LRU caches. As this theorem states, the bound $\hat{M}$ is the same for *all* files — in the worst case, LRU acts as a capping mechanism on the arrival flow, enforcing a *cutoff point* at $\hat{M}$. Arrival rates are only affected by the cache if they go above a certain value. The literature on LRU contains observations that in practice LRU conducts a sort of "low-pass filtering" [12] or "Majorization" [69]. Previous work on this subject showed this for actual behavior but was limited to simulation-based conclusions, specific topologies or analytical models for a limited range of arrival distributions. Our results here prove this to be the case for worst-case *bounds* over *arbitrary* boundable flows and arbitrary network topologies.

3.5.2 Computing $\hat{M}$ as a function of input bounds

In the previous section we demonstrated how the per-flow bounds are a function of $\hat{M}$. In this section we present an algorithm for computing M_w and $\hat{M}$.

We begin with the case of a finite window w. With Algorithm 4 we can compute the value of M_w when the input is $x_j := I(w, j)$ for all $1 \leq j \leq L$.

Theorem 18. *Algorithm 4 returns M s.t. $\lfloor M \rfloor = M_w$.*

Proof sketch: The algorithm consists mainly of iterating over two steps, shown in lines 8 and 10 of Algorithm 4. Line 10 (and initially 5) bound the number of miss sets by dividing the number of requests by $c+1$, the size of a miss-set. Since duplicate requests in such a set do not increase the number of misses, in Line 8 we remove such requests from our accounting. this ensures we get an upper bound. When the algorithm concludes, from the pigeonhole principle we show that each request can be a part of a miss-set, so the bound is tight. Next, we prove this claim formally.

Algorithm 4 *Bounds($x_1, ..., x_L$, c).*

1: // For all the following, assume $1 \leq k \leq L$
2: **for** $1 \leq k \leq N$ **do**
3:     $y_k := x_k$
4: **end for**
5: $M := \frac{1}{c+1} \sum_k y_k$
6: **while** $\max_k y_k > M$ **do**
7:     **for** $1 \leq k \leq L$ **do**
8:         $y_k := \min\{x_k, M\}$
9:     **end for**
10:     $M := \frac{1}{c+1} \sum_k y_k$
11: **end while**
12: RETURN M

Lemma 19. *If with an arrival of $\{x_k\}_{1 \leq k \leq L}$ we can construct M_w miss sets, then with $y_k := \min\{x_k, M_w\}$ requests for each k we can construct M_w miss sets as well.*

Proof: From Property 9 we know that for any miss set, after removing duplicate requests this set remains a miss sequence w.r.t. the same file. Thus, considering only the cases where all miss-sets are strict sets and not multi-sets is sufficient. To generate M_w strict miss-sets, at most one request for each file can appear in each such set, so M_w bounds the number of requests for each file, which concludes our proof. ■

Lemma 20. *If $M \geq M_w$ and $y_k := \min\{x_k, M\}$, then $M_w \leq \frac{1}{c+1} \sum_k y_k$.*

Proof: Since $M \geq M_w$, then from Lemma 19 we know that by using only y_k requests for f_k does not reduce the number of miss sets we can construct. Next, since the minimal size of a miss set is $c+1$ and the sets are disjoint, the lemma is proven. ■

Theorem 21. *Algorithm 4 returns M s.t. $\lfloor M \rfloor = M_w$.*

Proof: Denote the output of the algorithm as M. First we show that in each stage of the algorithm, $M \geq M_w$. At the initialization of the algorithm we have $y_k = I(w,k)$, and from Lemma 20 we know that in Line 5 $M \geq M_w$.

If we enter the "while" loop, in each iteration we reduce y_k in a manner which, according to Lemma 19, does not reduce the maximal number of miss-sets that can be constructed. We then update the value of M in line 10, which according to Lemma 20 bounds the number of miss-sets that can be constructed. Repeated application of these two steps will therefore not violate the condition $M \geq M_w$. Thus, regardless of entering the loop, we always get $M \geq M_w$, and since $M_w \in \mathbb{N}$, this implies $\lfloor M \rfloor \geq M_w$.

Next we show that $\lfloor M \rfloor \leq M_w$ by proving that when the algorithm halts $\lfloor M \rfloor$ miss sets can be constructed. By the loop exit condition (line 6) we know that $y_k \leq M$ for all $1 \leq k \leq L$ when the algorithm halts, and that (from line 10) $M = \frac{1}{c+1}\sum_k y_k$. From line 10 we know that $y_k = M$ for at most $c+1$ files, and since for all $1 \leq k \leq L$ $x_k \in \mathbb{N}$, taking the maximum in line 8 ensures at most $c+1$ files have non-integer y_k. Thus, rounding down all y_k will result in enough requests to construct $\lfloor M \rfloor$ miss sets. By the pigeonhole principle, since for all files $\lfloor y_k \rfloor \leq M$, we can construct M miss sets, which concludes our proof. ■

Next, we use Algorithm 4 to compute $\hat{M}$. This is done by applying the algorithm with ρ components as inputs, and no rounding operation.

Lemma 22.

$$\frac{1}{t}Bounds(\rho_{1,in}t, ..., \rho_{L,in}t, c) = Bounds(\rho_{1,in}, ..., \rho_{L,in}, c),$$

where Bounds() *is specified in Algorithm 4.*

Proof: To show this, we note that in both lines $8, 10$ the t parameter has linear impact. In the first iteration:

$$\begin{aligned} \textbf{line 8: } y_k &= \min\{\rho_{k,in}t, \frac{t}{c+1}\sum_k \rho_{k,in}\} \\ &= t\min\{\rho_{k,in}, \frac{1}{c+1}\sum_k \rho_{k,in}\} \end{aligned}$$

$$\textbf{line 10: } M = \frac{1}{c+1}\sum_k \rho_{k,in}t = \frac{t}{c+1}\sum_k \rho_{k,in}$$

and in all subsequent iterations, this phenomenon repeats itself, and so we can extract the t variable from the input.

■

Theorem 23. $\hat{M} = Bounds(\rho_{1,in}, ..., \rho_{L,in}, c)$

Proof: Let t be the length of window w. We can get bounds on the miss rate with

$$\frac{1}{t}M_w = \frac{1}{t}Bounds(I(w,1), ..., I(w,L), c)$$

For the case where the arrival streams are tight over the window w we get

$$\frac{1}{t}\hat{M}_w = \frac{1}{t}Bounds(\lceil \rho_{1,in}t + \sigma_{1,in}\rceil, ..., \lceil \rho_{L,in}t + \sigma_{L,in}\rceil, c)$$

From Theorem 16 we assume the burstiness is zero, and using a sandwich argument as in Theorem 17 the rounding operation can be ignored as $t \to \infty$. Thus, from Lemma 22 we conclude

$$\begin{aligned}\hat{M} &= \lim_{t\to\infty} \frac{1}{t} Bounds(\rho_{1,in}t, ..., \rho_{L,in}t, c) \\ &= Bounds(\rho_{1,in}, ..., \rho_{L,in}, c)\end{aligned}$$

■

The following theorem is also a result of this algorithm:

Theorem 24. *Consider two adjacent caches A, B such that the arrival stream at B consists totally of the entire miss stream of A, and B is smaller or equal in size to A. Then the bounds on the miss stream in A are identical to the bounds on the miss stream in B.*

Proof: This can be determined from Algorithm 4. For equal sized caches, the value $\lfloor M \rfloor$ computed for cache A will be computed in Line 5, and the loop will not be entered, so the same cap will be used. This the miss stream is a result of this capping, the cache B miss stream is unaffected. For smaller caches, the cap will be higher, once again having no impact on the miss stream of B. ■

Theorem 24 emphasizes the importance of cache and flow diversity in the network: In order for the next hop cache to lower the bounds on the arrival flows it experiences, it must be of a larger size, use different replacement policies or accept miss flows from a multitude of neighboring caches.

3.5.3 Achieving bounds simultaneously

Until this point, our discussion has focused on the upper bounds per individual file, rearranging the arrival order of the interfering requests to generate the worst case for some f_j. In this section, we show that in fact these bounds are tight also in combination — the worst case can be reached for *all* files simultaneously. We do so using a constructive proof: Algorithm 5 provides an arrangement of requests that generates the worst-case for all files.

In Algorithm 5, which considers a window w, we take as input the number of miss sets M and, for $1 \leq k \leq L$, $y_k = \min\{I(w,k), M\}$ as used in Algorithm 4. We show now that the arrangement the algorithm produces will generate misses for all y_k requests, for all k, in the case where the cache was empty at the beginning of w.

Algorithm 5 *GetMissSets*($y_1, ..., y_L$, c, M)

$S = \emptyset$
for k=1 to M **do**
    $\vec{s_k} = \emptyset$ // Initialize empty sequence
    $S := S \cup s_k$
end for
$j = 0$
for k=1 to L **do**
    for $h = 1$ to y_k **do**
        $s_j := s_j | q_k$ // "|" indicates concatenation
        $j := (j+1) \mod M$ // Next q_k request will be in a different sequence
    end for
end for
RETURN S

Theorem 25. *All y_k requests for f_k will be cache misses, for all $1 \leq k \leq L$.*

Proof: First note that each q_k is in a different miss sequence, by the pigeonhole principle and the fact that $y_k \leq M$. Next, denote the position of q_k in s_i as *index(i,k)* . If $q_k \in s_i \cap s_j$ and $i < j$, the algorithm ensures $index(i,k) \in \{index(j,k), index(j,k)+1\}$. Thus, if we concatenate the sequences in the reverse index order, i.e.,

$$\vec{s_M}.\vec{s_{M-1}}.....\vec{s_2}.\vec{s_1},$$

then all requests for the same file will be spaced out by at least c interfering requests. The first requests are all misses since we assume an empty cache, which concludes our proof. ■

The algorithm just described arranges exactly y_k requests per file to generate the worst case, when in practice there are $I(w,k) \geq y_k$ arrivals during w. To address this, we can place each of the excess $I(w,k) - y_k$ requests for f_k adjacent to a q_k in the sequence produced by the algorithm. This will not change the number of cache misses for any file, as a miss sequence for f_j can have an arbitrarily-long suffix consisting of requests for f_j.

3.5.4 Bounding the miss burstiness

We next consider the burstiness components of the miss streams, given the arrival stream bounds and $\rho_{j,out}$ computed in the previous sections. In a slight variation of our earlier definition, we define an eviction set (previously - *miss set*) and eviction sequence as follows:

Definition 26. An *eviction set* $e \subseteq F$ is a multi-set of at least c requests for unique files. A *eviction sequence* $\overrightarrow{e}$ is an ordered eviction set. We say this is an eviction set (sequence) for file j if $q_j \notin e$ ($q_j \notin \overrightarrow{e}$).

We further define similar concepts for eviction sets as we did for miss sets. $E_{w,j}$ denotes the number of eviction sets for j over w; $\hat{E}_{w,j}$ is $E_{w,j}$ when the arrivals are tight with the arrival bounds; and $\hat{E}_j = \lim_{|w|\to\infty} \hat{E}_{w,j}/|w|$. Since appending an eviction sequence w.r.t. j with a request q_j yields a miss sequence, it can be shown from Theorems 12 and 17 that

$$M_{w,j} = \min\{I(w,j), E_{w,j}\} \tag{3.5}$$

$$\rho_{j,out} = \min\{\rho_{j,in}, \hat{E}_j\} \tag{3.6}$$

In what follows we also use the following two sets for each j: $X_j = \{k \neq j : \rho_{k,in} < \hat{M}\}$, and $Y_j = \{1, ..., L\} \setminus (X \cup j)$.

Theorem 27 (σ bounds)**.** **(a)** *If* $\rho_{j,in} < \hat{E}_j$*, then* $\sigma_{j,out} = \sigma_{j,in}$*.*

(b) *If* $\rho_{j,in} > \hat{E}_j$*, then* $\sigma_{j,out} = \mathit{Bounds}(\{\sigma_{k,in}\}_k \in X_j, c - |Y_j| - 1)$*, where the* Bounds() *function is defined in Algorithm 4.*

(c) *If* $\rho_{j,in} = \hat{E}_j$*, then* $\sigma_{j,out} = \min\{\sigma_{j,in}, \mathit{Bounds}(\{\sigma_{k,in}\}_k \in X_j, c - |Y_j| - 1)\}$

As with the rate component, we see here once again that the less-popular files are unaffected by the cache (as shown in part (a) of the theorem), contrary to the popular files.

Proof: We adopt an amortized analysis approach here: for purposes of computing the bounds, we first associate requests to the rate component and then associate the remaining requests to the burstiness component of a given bound. We say that the first group are *rate related* while the second is *burstiness related.* We note that for any file such that $\rho_{j,in} < \hat{M}$, the entire rate component is accounted for in computing $\hat{M}$. We can see this by observing that in Algorithm 4 increasing this $\rho_{j,in}$ slightly (e.g. to anything less than $\hat{M}$) will result in an increase of $\hat{M}$. On the other hand, if $\rho_{j,in} > \hat{M}$, parts of the rate component will not be associated with any rate-related miss-set.

(a) Assume $\rho_{j,in} < \hat{E}_j$, then we know $\rho_{j,out} < \hat{E}_j$ from Eq. 3.6. Thus, there is a large enough window [s,t) over which $(\hat{E}_j - \rho_{j,out})(t - s) \geq \sigma_{j,in}$, where we can place each of $\sigma_{j,in}$ requests for q_j after a eviction-sequence w.r.t. j, resulting in additional $\sigma_{j,in}$ miss-sets for j. This yields a total number of misses of $\rho_{j,out}t+\sigma_{j,in} = \rho_{j,in}t+\sigma_{j,in}$, which is clearly bounded by the input, so it is tight.

(b) Assume $\rho_{j,in} > \hat{E}_j$, then from Eq. 3.6 we know $\rho_{j,out} < \rho_{j,in}$, so we have an infinite number of requests for f_j that are not in a rate-related miss-set. We now construct additional miss-sets for f_j by using the burstiness components. For each $k \in Y_j$ we have an infinite number of q_k arrivals also not in any miss-set, and all we require to complete a miss set is to add a request q_j and an additional $c - |Y_j|$ unique requests from X_j. The number of these is at most *Bounds(*$\{\sigma_{k,in}\}_k \in X_j$*,* $c - |Y_j| - 1$*).*

(c) If $\rho_{j,in} = \hat{E}_j$, both bounds from the previous sections hold using the same arguments above. Since one bounds the potential of interfering requests and the other the requests for f_j, taking the minimum of both gives us the bound on the miss stream burstiness. ■

What is left is to compute $\hat{E}_j$. To this end, note that eviction sets for f_j are identical to miss sets, except that (a) they do not include q_js and (b) they are of size c, not $c+1$. Thus, to compute $\hat{E}_j$ we once again use Algorithm 4, but with two changes:

- The input given is only for files $k \neq i$.
- In line 10 we substitute $1/(c+1)$ with $1/c$.

The arguments and proofs are identical to those shown for the proof regarding $\hat{M}$ and so are not detailed here.

3.6 Evaluation of Worst-Case Bounds

3.6.1 Extracting bounds from Trace Data

When evaluating how close our bounds come to predicting actual performance, we compare them to simulator-generated traces. Here we briefly discuss how to compute the (ρ, σ) bounds for flows in the simulation, both exogenous (user-to-router) and endogenous (router-to-router).

For a given trace, and since we compute here *globally-tight* bounds, this can be computed in linear time with the length (in terms of the number of requests) of the simulation:

- ρ_j is the mean request rate for f_j.
- To compute σ_j, compute first $\sigma' = \max_{k \in \mathbb{N}} \frac{1}{t_{j,k+1} - t_{j,k}}$, where $t_{j,k}$ is the arrival time of the kth request for f_j. σ' is the highest observed arrival rate. We then

compute σ by canceling out the mean rate component for that same time slot, so we get $\sigma = \sigma' - \rho(t_{j,k+1} - t_{j,k})$.

In addition to computing bounds based on trace data, we may want to compute the bounds on an arrival process based on its stochastic properties. Computing the ρ component is once again identical to the mean arrival rate of the process. Regarding the burstiness component, some processes do not have a deterministic bound (e.g., exponential distribution). In such cases, we can use a statistical bounding point: for some α, let σ_α be such that

$$Pr(t_{j,k} \leq t + \sigma_\alpha | t_{j,k-1} = t) = \alpha$$

for all $k \in \mathbb{N}$. Then, $\sigma_\alpha - \rho$ is the burstiness bound.

Due to the negligible or zero impact that the burstiness component has on performance (e.g., when considering the miss *rates*), we do not present simulation results regarding it in this work.

3.6.2 Bound tightness in practice

We next present several results concerning the performance of our calculus. As in the analytical sections, we focus on the ρ component, due to its centrality for system performance. As proven in this chapter, the bounds hold for all the experiments we conducted.

For Figures 3.3-3.5, the topology we consider is a complete binary tree of depth 4, where level 0 is the root node, shown in Figure 3.2. By default, we assume 600 unique files can be requested exogenously. One of the benefits of our calculus is the ability to compute performance for non-hierarchical systems. Thus, we place two custodians at nodes v_7, v_{14}, and split the files between them. As a result, the path from v_7 to v_{14} experiences *cross-flows* — flows of requests going in both directions. We consider the number of cross-flows to be the minimum of rates in either direction across a link, so for hierarchical systems this number is zero.

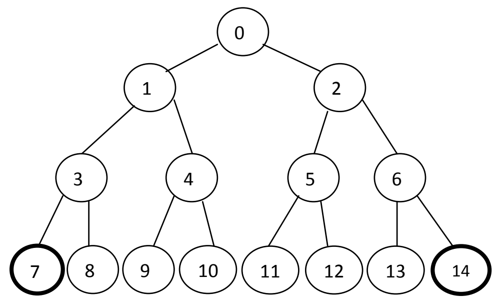


Figure 3.2: Topology for simulations. Custodians are at nodes $7, 14$.

Since we are interested in assessing the impact of cross-flows on the bound tightness, we consider the case where files are distributed according to a multi-zipf distribution. The approach here is to divide the files into sets of equal size, give each set an equal probability, and then have the popularity within each set be distributed according to zipf. In the examples shown here we divide the files into eight sets of 75 files. The benefit of using this distribution is that it is uniform across the sets, so we can move sets between custodians and know that each set carries the same probability, while still modeling the realistic scenario of non-uniform request patterns. Note that as the number of sets increases to L we get closer to uniform distribution, while as the number decreases to 1 we get the zipf distribution.

We consider two uses of our calculus. The first is for computing bounds on a network of arbitrary topology. We begin by computing per-node bounds when the exogenous rates are the arrivals per node. These arrival rates are then recomputed by combining the exogenous rates with the bounds on the miss stream that are forwarded to that node. This process is then repeated until the system converges to a fixed point. Essentially, we are using a-NET where our bounding algorithm acts as the SCA. We then compare the computed bounds to the actual performance of the system using simulations. As the bound-to-simulation ratio goes to 1, the bounds become more reflective of actual performance, indicating LRU performing close to its worst case.

The second use of our calculus is for specifically studying the performance of LRU in cache network scenarios. In this context, we simulate the performance of a cache network and then extract the simulated arrival rates at each node. We feed these arrival rates to the calculus and compare the actual (simulated) miss rates with the bounds. The same interpretation of bound-to-simulation ratio applies here as well.

Figures 3.3-3.5 consider the first use case of evaluating the tightness of these bounds. In Fig. 3.3 we gradually shift content from the custodian at v_{14} to the one at v_7, which generates more cross-flows. We see in this figure how this increase in cross flows causes the bounds to be tighter, especially near the root of the tree (nodes 0-2) where the flows are largest.

In Fig. 3.4 we see how when decreasing the cache size the bounds become tighter, and that the same phenomenon occurs when increasing the number of files. These results are especially relevant to cache networks, where the file-to-cache size ratio is expected to be high, making the bounding calculus a useful tool in estimating an upper bound on performance in practice.

We now turn briefly to applying our calculus in the second manner noted above — to determine how well LRU performs in a cache network. We once again consider the tree topology as before, but this time place a single custodian for all files at the root node, thus eliminating cross flows. The results are shown in Figure 3.6. They demonstrate that the bounds became tighter as we progress up the tree, indicating that cache hierarchies using LRU at all levels are inefficient as they increase in scale.

The importance of this calculus is highlighted when we consider a wider variety of arrival processes. Most models for caches consider only cases where the exogenous arrival process follows the Independent Reference Model (IRM). In Figure 3.5 we show how varying the inter-arrival time distribution can generate worse performance for

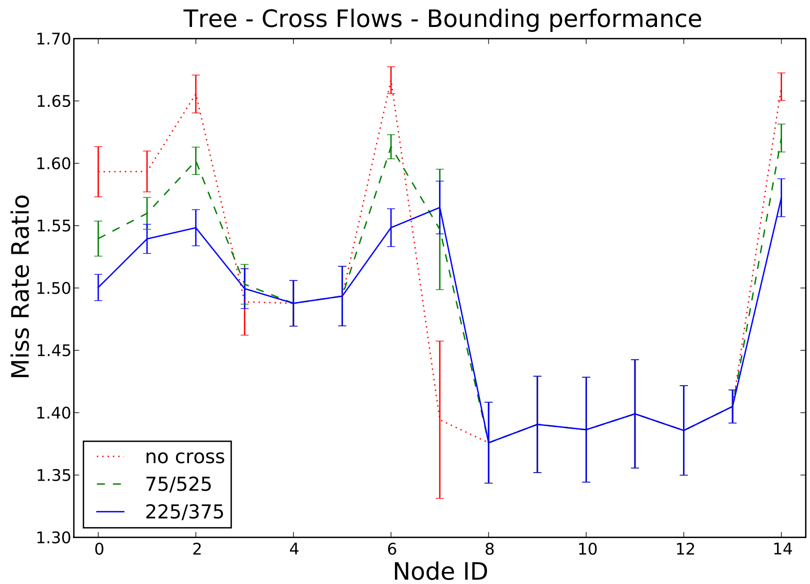


Figure 3.3: Impact of cross-flows on the bound tightness. cache size on bound tightness, with 90% confidence intervals shown. Setup is identical to Fig. 3.4. X/Y indicates X files at v_7 and Y files at v_{14}.

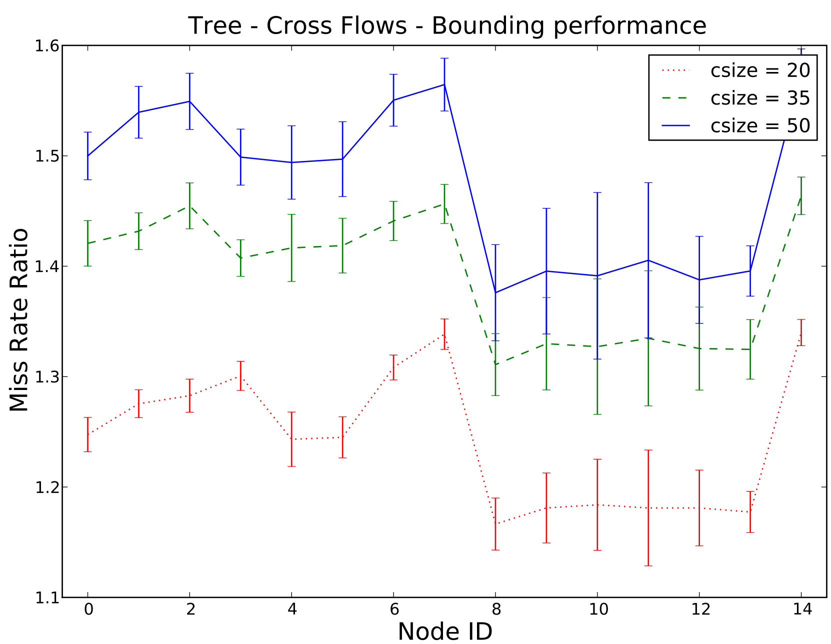


Figure 3.4: Impact of cache size on bound tightness, with 90% confidence intervals shown. Requests arrive at all nodes following a multi-zipf distribution. Files are divided between custodians at nodes $7, 14$, with 225 files at the first and 375 at the second. As cache sizes decrease, bounds become more tight.

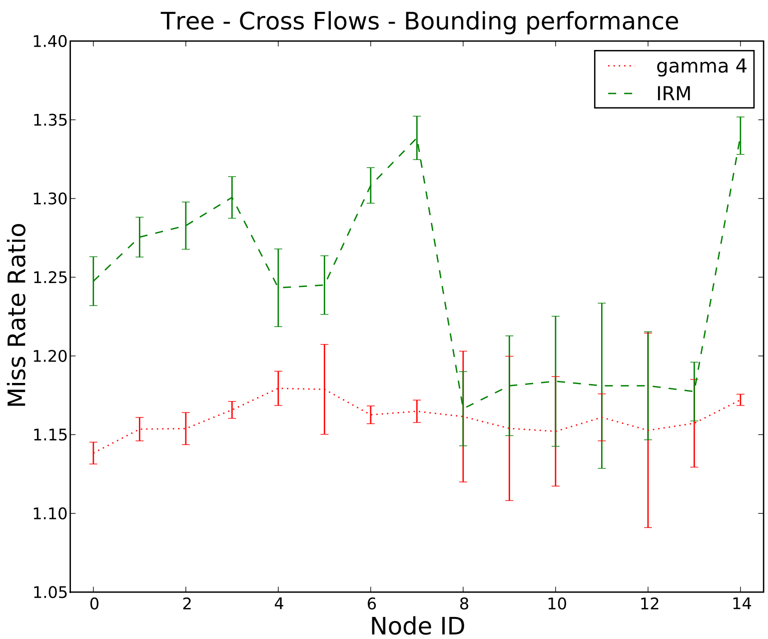


Figure 3.5: Impact of non-IRM traffic on bound tightness, with 90% confidence intervals shown. Setup is identical to Fig. 3.4. As we see here, with inter-arrival distances following the Gamma distribution with a scale parameter 4, bounds become more tight relative to with IRM.

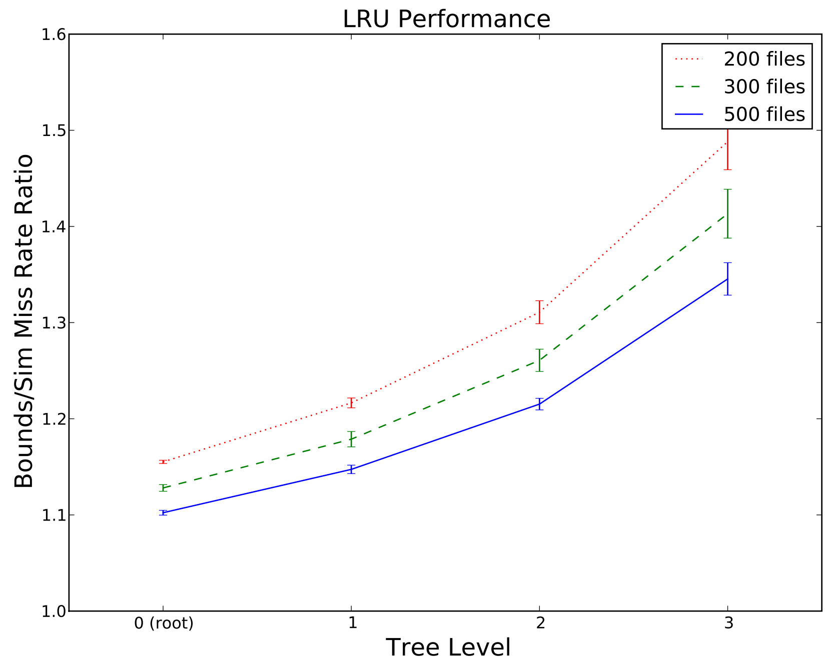


Figure 3.6: Performance of LRU as compared to LRU worst-case. 90% confidence intervals shown.

LRU. In this plot, we use the Gamma distribution to model exogenous arrivals: the time until the next arrival of a request for f_j with popularity $p_j = \lambda_j / \sum_i \lambda_i$ is modeled according to Gamma(p_j, β), where β is a scaling parameter ($\beta = 1$ generates the exponential distribution). As we can see in this figure, as the scaling parameter grows, the bounds become tighter. Thus, our bounding calculus is suitable for generating upper bounds in cases where the arrival process is not known in advance.

3.7 Discussion and Future Work

In this work we presented a Network Calculus for boundable flows in an LRU cache network, and demonstrated its performance for non-hierarchical topologies — scenarios that have not been addressed to date. Our bounds reveal that in the worst-case LRU acts as a cutoff point on the arrival process, providing analytical support to similar observations made earlier [12,69] regarding actual behavior in the network.

The results presented here can be extended in several directions. The bounds here can be shown to hold equally well for FIFO, whose worst-case is very similar to that of LRU. Also, due to the limited impact of burstiness on the miss rate, similar bounds on the rate can be shown for non-deterministic bounding models, such as Exponentially Bounded Burstiness [64]. As for extending beyond deterministic replacement policies and addressing policies such as random replacement, a bound on the mean behavior would be more suitable, for which a different set of analytical tools will be needed.

CHAPTER 4

STEADY-STATE OF CACHE NETWORKS

4.1 Introduction

In this chapter, we continue our analysis of cache networks and focus on factors that impact the steady-state behavior of content occupancy, which directly impacts performance.

Analytical models for caching systems, such as those discussed in the previous two chapters usually take into account the cache capacity (i.e., how much storage is available), the topology of the cache network (e.g., hierarchical), the cache-management policies and the exogenous request arrival rates per-file at each cache [44] [23] [59] [12]. Absent from this list is the *initial state* of the system — the files stored in each cache when the system is initialized. The fact that the initial state is ignored reflects an (explicit or implicit) intuition that the steady-state performance of the system is unaffected in the long-term by the initial content stored in the cache. In this chapter we consider this assumption and its validity as a function of various system properties.

The major contributions of this chapter are the following:

- We present two examples of non-ergodic CNs, in the sense that different content placed initially at the caches will lead to different steady-state behavior. In both examples, the observed behavior arises only when the caches are interconnected.
- We establish several important properties of CNs, in the form of three independently sufficient conditions for the CN system to be ergodic. Each property addresses a different aspect of the system — topology, admission control and cache replacement policies.

Table 4.1: Table of notation for Markov model representation

Notation	Meaning
$a, b \in \Omega$	States in the Markov Chain system representation
$a[j]$	The content of v_i when system at state a

- We demonstrate that the replacement policies can be grouped into "equivalence classes," such that the ergodicity (or lack-thereof) of one policy implies the same property holds for all replacement policies in the class.

The structure of this chapter is as follows. We begin in Section 4.2 by presenting the Markov model used throughout this chapter. Then, in Section 4.3, we present two examples of non-ergodic cache networks, in that the initial state determines the steady-state that the system converges to, with a consequent impact on system performance. In Section 4.4 we formulate and prove two theorems regarding system ergodicity, which relate to network topology and admission control. Then, in Section 4.5 we outline a class of cache replacement policies for which the system is always ergodic. We also identify equivalence classes of replacement policies such that the ergodicity (or lack-thereof) of one policy implies the same holds for other policies in that class. We conclude the chapter with a summary and discussion of future work in Section 4.6.

4.2 Model and Notation

We adopt here the same model described in Section 2.2, and only add to it now a description of a Markov Model for the behavior of a cache network. We do this using a discrete-time Markov chain with state space Ω_0. The system state s, $s = (s[1], s[2], \ldots, s[N])$, is a concatenation of N vectors, each of length equal to the cache size — c_i $\forall 1 \leq i \leq N$. The kth element in vector v_i corresponds to the content (e.g. file identifier) in the kth position of v_i. When all caches have the same size, the state space Ω_0 has cardinality $\left(\binom{L}{c} \cdot c!\right)^N$. When the order of the elements in the caches is irrelevant, states that differ only through such ordering are lumped together. In this case, when all caches have the same size, the state space $\Omega \subset \Omega_0$ has cardinality $\binom{L}{c}^N$. In what follows, we denote the size of the state space as u.

A sample path $\boldsymbol{\tau}_s$ of a CN is determined by its initial state, s, and a sequence of file requests and consequent evictions, $\boldsymbol{\sigma}$ and $\boldsymbol{\pi}$, respectively, $\boldsymbol{\tau}_s = (\boldsymbol{\sigma}, \boldsymbol{\pi})$. Let $\boldsymbol{\sigma} = (\sigma_k)_{1 \le k \le K}$ be a sequence of K file requests. Let $\boldsymbol{\pi} = (\pi_k)_{1 \le k \le K}$ be a sequence of K sets of files, each set indicating the files evicted in the network as a result of σ_k. For each request σ_k, let $\pi_k(i; \boldsymbol{\sigma}) \in F \cup \{\star\}$ be the file evicted from v_i, $1 \le i \le N$, while request σ_k is served. $\pi_k(i; \boldsymbol{\sigma}) = \star$ means that no file is evicted from v_i when request σ_k is served. Note that a single request can cause, via file download path, changes at multiple caches. Recall that we assume ZDD, so we can ignore intermediate system states that would exist while content is being forwarded along its download path.

Given initial state s, let s_k be the state resulting from the service of the k-th request in $\boldsymbol{\tau}_s$. Let $s_k[i]$ be the state of v_i at system state s_k. Let $\Gamma(\boldsymbol{\tau}_s)$ be the sequence of states of the sample path $\boldsymbol{\tau}_s$, $\Gamma(\boldsymbol{\tau}_s) = (s_1, \ldots, s_k)$, where $s_1 = s$. Let $\varphi(\boldsymbol{\tau}_s)$ denote the last state of $\Gamma(\boldsymbol{\tau}_s)$, $\varphi(\boldsymbol{\tau}_s) = s_k$.

Let $\mathbf{A} = (\alpha_{d,e})_{1 \le d,e \le u}$ be the adjacency matrix of a CN. $\mathbf{A}$ is a binary matrix, where $\alpha_{d,e} = 1$ if it is possible to reach state e from state d through one transition,

$$\alpha_{d,e} = \begin{cases} 1, & \exists \boldsymbol{\tau}_d = (\boldsymbol{\sigma}, \boldsymbol{\pi}) : e = \varphi(\boldsymbol{\tau}_d) \wedge |\boldsymbol{\sigma}| = 1 \\ 0, & \text{otherwise} \end{cases} \tag{4.1}$$

State e can be reached from state d if there is a sample path $\boldsymbol{\tau}_d = (\boldsymbol{\sigma}, \boldsymbol{\pi})$ such that $e = \varphi(\boldsymbol{\tau}_d)$. Let $\alpha_{d,e}^{(n)}$ be an element of $\mathbf{A}^n$, $1 \le d, e \le u$. State e can be reached from state d if there exists an integer n such that $\alpha_{d,e}^{(n)} = 1$. We conclude with terminology that will be used throughout the chapter:

Definition 28. A *recurrent state* of a CN is a state d such that for any state e for which there exists an $n \in \mathbb{N}$, $\alpha_{d,e}^{(n)} = 1$ there exists an $n' \in \mathbb{N}$ s.t. $\alpha_{e,d}^{n'} = 1$.

Definition 29. A *transient state* of a CN is any state that is not recurrent.

Definition 30. An *ergodic set* of a CN is a set of recurrent states in which every state can be reached from every other state, and which cannot be left once it is entered. Formally, it is a set S s.t. for all $d, e \in S$ there exists an $n \in \mathbb{N}$ s.t. $\alpha_{d,e}^{(n)} = 1$, and for all pairs of states s.t. $d \in S, e \notin S$, $\alpha_{d,e}^{(n)} = 0$ for all $n \in \mathbb{N}$. [Note that for the second case there still might be some $n \in \mathbb{N}$ s.t. $\alpha_{e,d}^{(n)} = 1$.]

Definition 31. A *quasi-ergodic CN* is a CN that comprises a single ergodic set. Formally, it is a CN such that if two states $d, e \in \Omega_0$ (Ω) each belong to some ergodic set S_d, S_e respectively, then $S_d = S_e$. For such a CN, we say that its Markov chain is quasi-ergodic.

According to Definition 31 a quasi-ergodic CN is a CN whose state space consists of a single ergodic set after the removal of all transient states. Note that according to this definition, a quasi-ergodic Markov chain differs from the classic ergodic Markov chain, whose states form a single ergodic set but transient states are not allowed [36]. In the remainder of this paper, except otherwise noted, we will ignore transient states and refer to a quasi-ergodic CN simply as an ergodic CN.

4.3 Sensitivity to the initial state: examples

To motivate the need for considering the initial state of the CN, we present here two scenarios in which the initial conditions of the CN determine its steady-state behavior, and consequently the performance of the CN system.

4.3.1 Example 1

In our first example we consider the topology in Figure 4.1, consisting of two caches of size c each. Let A, B be two disjoint sets of files, such that $|A| = |B| = c$, and assume LRU replacement is used at both caches. The file set that user i requests from is denoted by Y_i, and consider the case where $Y_1 = A, Y_2 = B$. Now, we consider two initial states: (I) when v_1 (v_2) contains exactly the set of files A (B), and (II) when both caches are empty. For scenario I the system will remain in the initial state indefinitely, with each cache storing only files from a single set, and will experience no cache misses. For scenario II, on the other hand, cache misses will occur indefinitely, and both caches will store files from both sets over time.

While this example is outwardly simple, it offers several interesting lessons, beyond the impact on performance. First, it is important to note that, in this example, the state space is disjoint: the initial state described in I *cannot be reached* from any other state. Second, slight changes in the request patterns of users can lead to drastically different cache behaviors. For example, one can show that if $|Y_1| = |Y_2| = c$ but each Y_i has elements from both A and B, the system would eventually converge to a single state, $v_i = Y_i$ (i.e., that each cache stores the content requested by the exogenous stream of requests arriving at it), mirroring the user demand, independently of the initial state. This is true since once in this state it will never be changed, and it can be shown that this state is reachable from any other state for LRU caches. Thus, we can see dependencies form in the network in such a way that, at times, small changes in user demand can have a very significant impact on cache behavior.

4.3.2 Example 2

As a second example consider a network comprised of caches using the FIFO replacement policy and $L = c + 1$ files in the network, with a non-zero request rate for each of these files at each node. In §4.3.2.1 we will first show that although these caches are non-ergodic in isolation, their performance is independent of the initial state. Next, in §4.3.2.2, we show that once interconnected, performance depends crucially on the initial state.

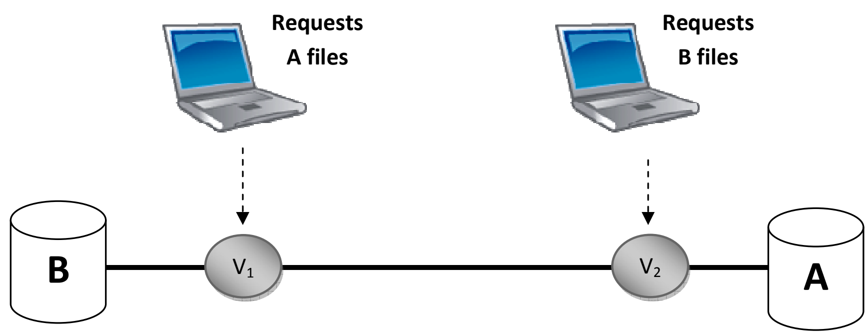


Figure 4.1: Example scenario in which the solution of the MC is dependent on initial state.

4.3.2.1 A single FIFO cache in isolation

Here we show that a single FIFO cache with the given $L = c + 1$ ratio is non-ergodic. The set of states for which the cache is full can be partitioned into $(n-1)!$ disjoint sets of states, such that a state is reachable from another only if they are within the same set. Each of these sets of states corresponds to a cyclical ordering of the files, indicating the order in which they are evicted — an order that repeats itself indefinitely. Thus, the initial state (or, if we start with an empty cache, the first c unique files requested), determines the steady-state of the cache. Despite this fact the probability that $f_j \in v$ is *independent* of the initial state. This can be determined from the balance equations for the Markov chain:

$$(1 - e_j)\lambda_j = (1 - e_k)\lambda_k, \qquad \forall\ 1 \leq j < k \leq L \tag{4.2}$$

where e_j is the probability $f_j \in v$. The system of equations (4.2) admits a single solution, independent of the initial state,

$$e_j = 1 - \left(\lambda_j \sum_{k=1}^{L} \frac{1}{\lambda_k}\right)^{-1} \tag{4.3}$$

Since the arrival process is IRM, the occupancy probability is also the hit probability [59].

4.3.2.2 Dependencies in networks

Next, we show that the interconnection of the two isolated FIFO caches described in the previous section results in a CN in which the initial state impacts steady state performance. Consider the CN shown in Figure 4.2. This network has three files and two caches arranged in a line, and $c_1 = c_2 = 2$. Upon a cache miss, the request is forwarded in the direction of the custodian. Figure 4.3 shows the transition probability matrix, obtained using Tangram II [15].

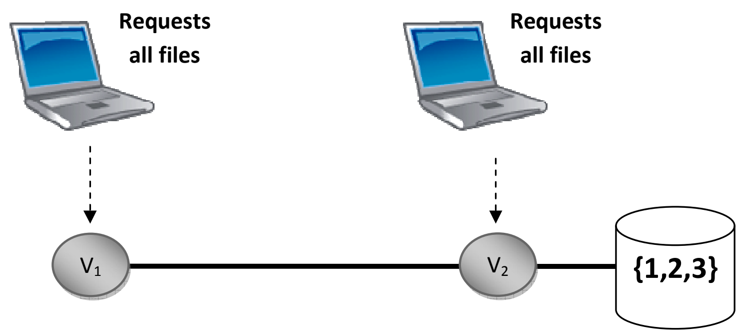


Figure 4.2: Topology for second scenario in which the solution of the MC is dependent on initial state. Caches here are assumed to be using the FIFO replacement policy.

To illustrate the impact of the initial state on the steady state solution, we initialize both caches at the same state, and consider two different initial conditions, $v_1 = v_2 = (f_1, f_2)$ and $v_1 = v_2 = (f_1, f_3)$. Let $\lambda_{11} = 0.35$, $\lambda_{12} = 0.55$, $\lambda_{13} = 0.1$, $\lambda_{21} = 0.05$, $\lambda_{22} = 0.15$, $\lambda_{23} = 0.8$. Table 4.2 shows the steady state file occupancy probabilities at cache 2, as a function of the initial state. It is clear from these results that, for this system, the initial state has substantial impact on the overall steady state performance.

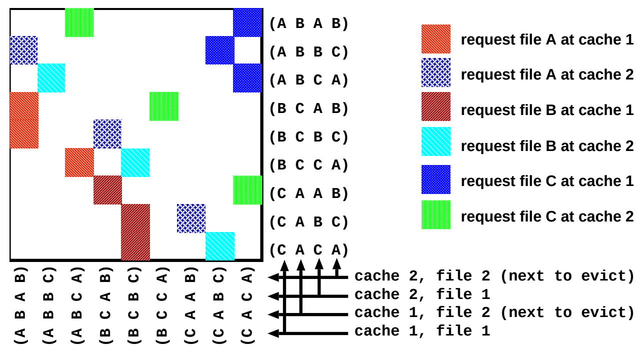


Figure 4.3: Transition matrix of Example 2 (diagonal elements not shown). The system state is (w, x, y, z) where w and x (resp., y and z) are the two files at cache 1 (resp., 2). Let $A = f_1$, $B = f_2$, $C = f_3$ when the initial state is (f_1, f_2). Let $A = f_1$, $B = f_3$, $C = f_2$ when the initial state is (f_1, f_3).

Table 4.2: Example of the impact of initial state on system solution for the topology in Fig. 4.2 and transition matrix shown in Fig. 4.3.

Initial State	e_{21}	e_{22}	e_{23}
(f_1, f_2)	0.46651	0.63134	0.90214
(f_1, f_3)	0.33054	0.76861	0.90083

Let us provide some intuition about what is happening here. In the case of a single cache, the system is non-ergodic but, due to a symmetry among the states in the Markov model, the performance of this cache is unaffected by the initial state or requests. Once the caches are networked, however, this symmetry no longer holds, and a different steady-state distribution of files is obtained, depending on the initial conditions. Once again we see that the interconnecting (networking) of caches introduces unexpected behaviors.

4.4 Conditions for Ergodicity: Topology and Admission Control

In light of the examples presented in the previous section, we present here several theorems with regards to the ergodicity (or lack thereof) of a CN. Each theorem presents an independently-sufficient condition for ergodicity. We begin with several definitions.

Definition 32. The topology of a cache network is *feed-forward* if on every link requests flow only in one direction and content is downloaded only in the other direction.

A classic example of a feed-forward network is a cache hierarchy (i.e., a tree), with a single custodian at the root.

Definition 33. An exogenous request stream for files at v_i is said to be *positive* iff $\forall f_j \in F$, $\lambda_{ij} > 0$. If this condition holds for all $v \in V$, we say the exogenous request stream at the (cache) network is positive.

Recall that a cache in isolation is a single-cache system (see §4.3.2).

Definition 34. A CN is said to be *individually ergodic* if its components are ergodic in isolation for a positive request stream. This means that, for each cache $v \in V$, when v functions as a cache in isolation, v is ergodic, given the exogenous request stream is positive.

The example in §4.3.1 used a non-positive request stream, while the example in §4.3.2 uses a cache hierarchy that is not individually ergodic. We are currently unaware of any system where request streams are positive and caches are individually-ergodic, but the system as a whole is non-ergodic. We discuss the possibility that such a system exists in §4.6. We now state our first two theorems.

Theorem 35. *An individually-ergodic CN with positive exogenous request streams is ergodic if it is feed-forward.*

Theorem 36. *Consider an individually-ergodic CN where v_i caches file f_j (if and when f_j passes through v_i) with probability $0 < \theta_{ij} < 1$ for all $1 \leq j \leq L$ and $1 \leq i \leq N$. Then this system is ergodic when subject to positive exogenous request streams.*

These two theorems are proven using the same general approach: We begin by selecting a pair of states $a, b \in \Omega$ and demonstrate that there exists a sample path between them. In other words, we show that there is a series of requests and evictions that change the system state from a to b. Since the system is finite in size, this implies ergodicity of the entire system.

We begin with Theorem 35. For a feed-forward network we define the direction of requests as *upstream* and the reverse as *downstream*. We rely on the following observation regarding feed-forward networks:

Lemma 37. *When ZDD is assumed, caches are not affected by the state or request stream experienced at caches further upstream.*

Proof: A cache v is only affected by the requests and content that pass through it. Requests only travel upstream, so upstream nodes do not impact v via the requests it experiences directly. Content flows downstream, but only for requests that were forwarded by v, which are not affected by upstream nodes. Finally, with ZDD the download delay is negligible, so state along the download path of upstream node has no impact on the state at v. ■

Proof: [Proof of Theorem 35] To transition from a to b, we iterate over all caches, marking them as we proceed. We start from those with no downstream nodes, and gradually move up to the next node that has no unmarked children downstream. Recall from Table 4.1 that for a system state a, $a[i]$ is the state (i.e., stored contents) at node v_i at that state. At each node v_i, we generate a sequence of exogenous requests at each cache that will modify it from state $a[i]$ to $b[i]$. This can be done, since we assume the request stream is positive and IRM, and that the nodes are individually ergodic. From Lemma 37 we also know that this process has no impact on the state of caches downstream. ■

Proof: [Proof of Theorem 36] We assume that for each cache v_i $\theta_{ij} < 1$ for all $1 \leq j \leq L$. Thus, for any finite sequence of files σ that pass through this node and any sub-sequence σ' of σ, there is a positive probability that only the files in σ' will be admitted to the cache for storage. Thus, we iterate over the caches in arbitrary order, and at each cache let σ_i be a sequence of requests originating from v_i, s.t. applying these requests yields $a[i] \rightsquigarrow b[i]$ — since the caches are individually ergodic, such a sequence exists. In addition, there is a non-zero probability that only v_i will store the files requested in σ_i. This series of events will therefore result in the path $a \rightsquigarrow b$. ■

4.5 Conditions for Ergodicity: Replacement Policy

We now proceed to present the main contribution of this chapter — a theorem regarding the impact of a replacement policy on the ergodicity of the system. We shall begin in Section 4.5.1 considering only Random replacement, and then in Section 4.5.2 expand this result to a broad class of replacement policies.

4.5.1 Theorem for Random Replacement

In this section we consider the Random replacement policy. A CN in which all caches use the Random replacement policy is individually ergodic. In general, whether individually ergodic CNs are ergodic is an open question. Nevertheless, when caches use the Random replacement policy the answer to the question is *yes*, as stated in the following theorem.

Theorem 38. *A CN that uses Random replacement is ergodic when subject to positive exogenous request streams.*

Before we present our proof, we give a short overview of our method of proof. Let $a, b \in \Omega$ be two *recurrent* states. We will prove our claim by showing there exists a state d that is reachable from both a and b. Since we assume a and b are recurrent, there exists a reverse path from d to each of them, and so a and b belong in the same ergodic set. Since a, b are any two recurrent states, this proves there is a single ergodic set of states in this system, which concludes our proof.

Our proof will proceed by considering two CNs that are identical in all aspects (topology, routing, cache size and replacement policy, custodian location) except in their initial state — CN_a begins in state a and CN_b in state b. Given their states, we generate a sequence of exogenous requests $\boldsymbol{\sigma}$, (denote $K := |\boldsymbol{\sigma}|$), that arrive at both CNs. To accommodate the differences in the initial state, we will also design two sequences of evictions $\boldsymbol{\pi}_a$ and $\boldsymbol{\pi}_b$ to match $\boldsymbol{\sigma}$, and demonstrate that the sample path in both networks leads to the same state. In fact, we will design these paths so that both networks are monotonically becoming more similar to one another. To quantify this, we use the following definition: Let $\gamma_{a,b}(i) = |a[i] \cap b[i]|$ be the agreement index of two networks at v_i. We say that two caches *agree* iff $\gamma_{a,b}(i) = c$. As we will demonstrate for the sequence we construct, for all $1 \leq h < k \leq K$ and all $1 \leq i \leq N$, $\gamma_{a_k,b_k}(i) \geq \gamma_{a_h,b_h}(i)$, and after $\boldsymbol{\sigma}$ is served $1 \leq i \leq N$, $\gamma_{a_K,b_K}(i) = c$. In what follows, we formalize this intuition by showing how to construct the sequences of file requests and evictions.

Requests. Consider two CNs, CN_a and CN_b, which differ only through their initial states, a and b, respectively. Algorithm 6 describes how to construct the sequence of requests $\boldsymbol{\sigma}$ to be applied in each of these networks. Our approach will be to iterate over all the caches (line 2), and for each cache ensure that its state is the same for both CNs. Specifically, for cache v_i, Δ_i is the set of files in the cache in CN_b but not in CN_a (line 3). Then, for each file f_j we generate requests for f_j at each cache along the path from $cust(j)$ to v_i according to the routing matrix. We do so by injecting an exogenous request at each node along this path (lines 5-11).

Algorithm 6 SigmaConstruct(a, b).

Input: $a, b \in \Omega$ recurrent states in the Markov chain representing the CN
1: $\boldsymbol{\sigma} \leftarrow ()$ // Empty sequence
2: **for** $i = 1 \to N$ **do**
3:     $\Delta_i \leftarrow b[i] \setminus a[i]$
4:     **for** $f_j \in \Delta_i$ **do**
5:         $\boldsymbol{\sigma'} \leftarrow ()$
6:         $h \leftarrow i$
7:         **while** $v_h \neq cust(j)$ **do**
8:             $\boldsymbol{\sigma'} \leftarrow q_{hj}.\boldsymbol{\sigma'}$ // "." indicates concatenation
9:             $h \leftarrow$ next hop according to $\mathcal{R}_h$
10:         **end while**
11:         $\boldsymbol{\sigma} \leftarrow \boldsymbol{\sigma} | \boldsymbol{\sigma'}$
12:     **end for**
13: **end for**
14: return $\boldsymbol{\sigma}$

Evictions. Next, our goal is to determine the evictions that will take place in both networks. Assume during execution of $\sigma_k \in \boldsymbol{\sigma}$ file f_j arrives at v_h. The file that we evict (as part of the process of bringing the state of the two networks together) will depend on the state at each of the networks at this stage:

- $f_j \in a_k[h] \cap b_k[h]$ - no evictions are needed, since in both networks the file is already cached at v_h.

- $f_j \notin a_k[h] \cup b_k[h]$ - in both networks, the cache does not have the file. We cache f_j in each, and evict some file from each.
  - If $a_k[h] \cap b_k[h] = \emptyset$, select a random file from each to evict. This is called a *random two-sided eviction.*
  - Otherwise, select a file $f \in a_k[h] \cap b_k[h]$ to evict. This is called an *identical two-sided eviction*
- Otherwise, w.l.o.g. $f_j \in b_k[h] \setminus a_k[h]$. Then we change nothing at $b_k[h]$, and evict from $a_k[h]$ some file $f' \in a_k[h] \setminus b_k[h]$. We call this a *one-sided eviction.*

Lemma 39. *Following the eviction rules above, $b_k[i] \setminus a_k[i] \subseteq b_h[i] \setminus a_h[i]$ for all $h < k$.*

Proof: Let us consider a file f_j, requested at σ_l, $h < l \leq k$, causing an eviction at node v_i. A one-sided eviction increases agreement of caches, since a non-matching file was evicted to make room for a matching file. A random two-sided eviction increases agreement, since beforehand the caches were disjoint, and now they share f_j. Finally, with an identical two-sided eviction, file $f_q \in b_h[i] \cap a_h[i]$ was evicted from both caches to make room for f_j, which does not decrease the cache agreement. ■

Lemma 40. *After the files in Δ_i were requested in $\boldsymbol{\sigma}$, both networks agree on v_i.*

Proof: Assume Δ_{i-1} ended with request σ_k. From Lemma 39 we know that $b_k[i] \setminus a_k[i] \subseteq \Delta_i$. Every file that is in $b_k[i] \setminus a_k[i]$ will therefore cause a one-sided eviction, increasing agreement by 1, and every other file does not decrease agreement. Thus at the end of Δ_i the networks agree at v_i. ■

We use this construction to prove our Theorem.

Proof: [Proof of Theorem 38] We prove Theorem 38 using an inductive argument. From Lemma 40 we know that after requesting Δ_i both networks agree on the state of v_i. Furthermore, we know that no download can negatively impact cache agreement from Lemma 39, so once caches agree they continue to agree. Thus, after requesting all the mismatched files at all caches, the networks agree. ■

4.5.2 From Random Replacement to non-protective policies

A review of our proof for Theorem 38 reveals that the only place in which it relied on using Random replacement was in assuming that, given that an eviction is taking place, there is a transition in the Markov chain for evicting each of the files at that node. This allowed us more freedom in designing a sample path between two designated states. With this insight, we consider the following class of policies:

Definition 41. A replacement policy for an isolated cache is said to be *non-protective* if for any file $f \in v$ there is a positive probability that f will be the next file to be evicted (if another eviction takes place). A replacement policy for which this property does not hold will be termed *protective.*

Note that a cache using a non-protective replacement policy is individually ergodic. While with Random replacement the next eviction could be any file, this definition is broader. It covers policies in which requests can change the order of evictions at a cache without changing its contents. LRU is an example of such a policy, since a request for $f_j \in v_i$ that arrives at v_i can change the eviction order. The same holds for other policies such as LRU-K and LFU. FIFO, on the other hand, is a protective policy when $c > 1$. In this section, we prove the following extension of theorem 38:

Theorem 42. *A CN with positive exogenous request streams is ergodic if the replacement policy used in each cache is non-protective.*

Note that this theorem allows for *heterogenous* systems where each cache selects a replacement policy that might be different than the policy selected at another node in the network. Our approach will be to demonstrate that ergodicity of Random replacement can be used to prove the ergodicity of all cache networks with non-protective caches. At a high level, given a path in the Markov chain of a Random replacement network, we add exogenous requests at each cache where a reordering of evictions must take place so that the sequence of states of this non-Random replacement network continues to match the sequence of states in the Random-replacement network. in order to maintain correlation with the path for Random replacement. For exposition purposes, examples shall be presented using the LRU replacement policy, though the proof applies to all non-protective policies.

Unlike Random replacement, other policies use an (explicit or implicit) ordering of the files in the cache, from which the eviction order can be determined. With LRU, for example, items are ordered according to last reference. We begin here by "lumping" together, in the LRU model, all the states that differ only in the internal ordering of content, and mapping these to the state in the Random model. Figure 4.4 depicts an example of such a mapping for a 2-node cache, while Figure 4.5 does the same for FIFO.

These examples demonstrate that changes in eviction order without impacting the content of any cache are possible within LRU networks but not within FIFO networks. For the former, the closure of each such "lump" of states is a clique, with every pair of states communicating with one another only via other states with the same content. For the latter, even for an isolated cache, a change in order can only be achieved by changing the content of the cache, and in networked scenarios the situation is complicated by the fact that during content download other caches might be impacted as well. This makes determining ergodicity for FIFO and other protective policies more challenging, since a state-request pair fully or partially determines the files to be evicted, limiting us in finding a path between two recurrent states.

Proof: [Proof of Theorem 42] Let M_{alg} be the Markov chain of a CN using a replacement algorithm *alg*. Furthermore, let M^*_{alg} be the same graph, but after contracting all nodes representing identical cache contents. After the contraction, edges between states with the same content are eliminated, and all edges with other states are attached to the contraction node. We first demonstrate that $M^*_{alg} = M_{rnd}$

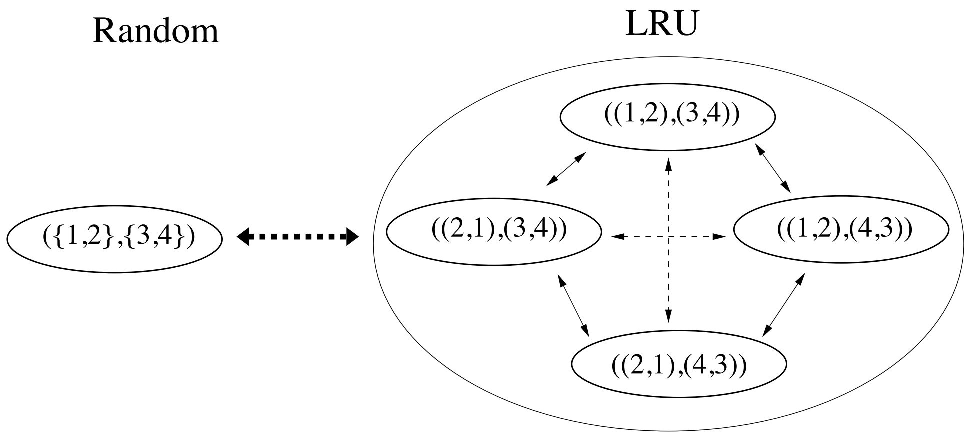


Figure 4.4: RND-to-LRU state mapping and edge contractions example. Edges indicate transitions in the markov model. As can be seen here, the closure of the indicated transitions results in a clique (broken edges mark the added connectivity), so it is possible to move from any state to any other without influencing the set of files stored in any cache.

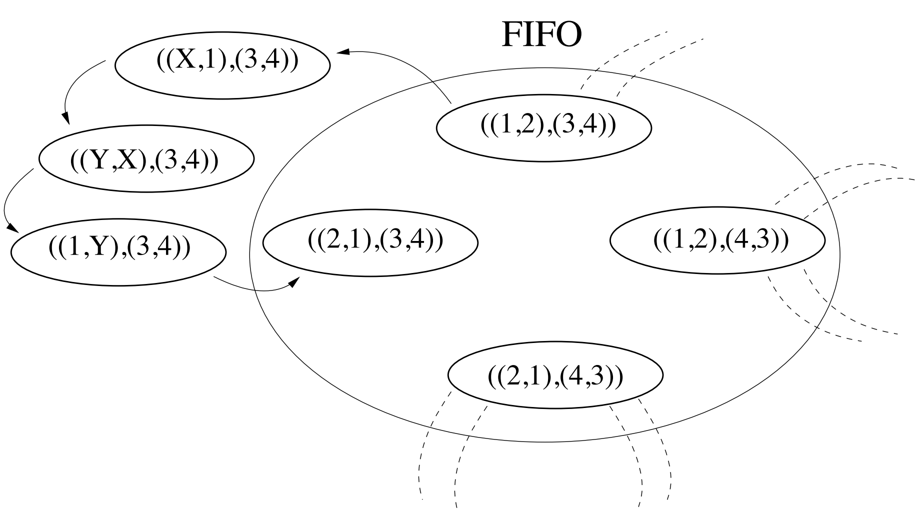


Figure 4.5: An example for the situation with FIFO replacement. $X, Y \in F \setminus \{1, 2\}$. As can be seen here, with FIFO there are no edges between states with the same content in all the caches, and all paths between such states require changing the content of some caches. In fact, there is no way to change the order of eviction in a cache with FIFO.

if *alg* is non-protective. Next, we show that when *alg* is non-protective, if M^*_{alg} is ergodic, so is M_{alg}. Since we know M_{rnd} is ergodic, the theorem is proven.

As discussed above, each node in M_{rnd} is mapped to the a node in M^*_{alg} representing all nodes in M_{alg} where caches have the same content. By construction, this is a one-to-one mapping. Next, consider two states in M_{rnd} and their mapped counterparts, denoted as $s_{rnd}, s'_{rnd}, s_{alg*}, s'_{alg*}$. We want to prove there is an edge (s_{rnd}, s'_{rnd}) iff there exists an edge (s_{alg*}, s'_{alg*}).

- First, we note that in s_{rnd} and s_{alg*} each cache holds the same content. Thus, a request will traverse the same caches in both networks regardless of replacement algorithm, and be stored at the same caches.

- Second, we show that the same evictions can take place in both. Consider a specific cache v_i. With Random replacement all content in v_i is up for eviction. Similarly, since *alg* is non-protective, for every $f_j \in v_i$ there is a state in the set of contracted states that evicts this file. Thus, an edge reflecting this eviction will exist.

Thus, the identity of these graphs is proven, and one is ergodic iff the other is as well. We now demonstrate that since M^*_{alg} is ergodic, M_{alg} is as well. Since with non-protective systems there is a path between any two states that share the same cache contents, without moving outside this set of states, this claim is true. For each request σ_k and eviction set π_k for random replacement, we inject requests between σ_{k-1} and σ_k that cause no eviction, but rearrange the content in the caches such that when σ_k is served the same π_k are evicted. This is possible due to being non-protective. Thus, since all states within the contracted states communicate, and all recurrent states in M^*_{alg} communicate (following ergodicity), we conclude that M_{alg} is ergodic. ■

4.5.3 Generalizing the Model

Throughout the chapter we assumed ZDD and that files and caches were all of constant size. While these make the exposition simpler, the proofs we present here apply equally when these restrictions are removed. Recall that our proof technique was to demonstrate that there exists a path within the Markov chain that leads from one state to another. Even when we relax the ZDD assumption, it is still *possible* that every request was satisfied before the next one was generated. Thus, the same path we constructed in each of the proofs will exist in this finer-granularity environment as well.

Regarding cache sizes, the proofs apply as-is to variable cache sizes, by changing each c with $|v_i|$ for the jth cache. Similarly, allowing for files to have variable size does not interfere with our proof concept, as long as (when needed) it is possible to evict a set of smaller files to make room for a single large file. The proofs for Theorems 35 and 36 do not rely on file sizes. For Theorem 42, the request sequence described in the proof is constructed in the same manner, and when evictions take place, each eviction type can be shown to maintain or improve cache agreement.

4.6 Summary and Future Work

In this chapter we continued our theme of cache network analysis from the previous two chapters, establishing here several properties regarding the ergodicity of cache networks. While solving a Markov model of a cache network is intractable for any Internet-scale system, we have shown here that one can still use these models to make *structural* arguments that lead to interesting insights.

The significance of our results are threefold. From a theoretical standpoint, our analysis provides tools for determining the ergodicity of cache networks. Furthermore, since we considered only the structural topology of the Markov chain while ignoring edge weights, our results show that for a non-ergodic system there are certain recurrent states that are unreachable even during fluctuations in the Markov chain edge weights. From an experimental standpoint, ergodicity determines whether or not the initial state of the system must be varied for valid system evaluation.

The examples presented in §4.3 did not include cases with positive request streams *and* individually ergodic systems. We pose as an open question if there exists such a system that is not ergodic. Theoretically, such a system could exist, with non-ergodic behavior on a system-wide level caused by dependencies among caches, formed by the file download paths. At this time, however, we are unaware of such a system. We hope that future investigations will shed light on this question.

CHAPTER 5

BREADCRUMBS - BEST-EFFORT CONTENT SEARCH IN CACHE NETWORKS

5.1 Introduction

In this final technical chapter, we discuss best-effort content search methods in a cache network. To this end, we relax the assumption that we use only static routing matrices $\mathcal{R}_i$ for request routing, and consider *dynamic* request routing policies, where nodes adapt their routing decisions as a function of time and/or system state. We present an adaptive content search scheme named *Breadcrumbs*, in which caches only use local information to determine where a copy of the content is *likely to be found*, and route requests accordingly.

When we allow for dynamic request routing, the miss routing decisions can be viewed as part of a *content search* process. In many ways, a cache network can be thought of as a large distributed cache; several classic performance metrics, such as hit probabilities and miss rates, are equally applicable to individual caches as well as to the network as a whole (i.e., considering the network of caches as a single, distributed entity). The fact that caching functionality is distributed can impact certain measures; for example, the download delay might be variable even when content was located at one of the networked caches. Perhaps the most significant manner in which a cache network differs from a single cache is, however, the fact that a copy of content might be present at some networked cache and still a request might not locate this copy, and instead retrieve the content from the content's custodian. The search process defined by the request routing policy is therefore a major differentiator of cache networks from standard caches, and demands special attention.

In general, caches can collect information about network state and then collaborate with other caches to improve content search. Such a collaboration can take the form of determining where content is stored and where to search for it. One defining characteristic of this collaboration is whether caches coordinate explicitly or implicitly. With *explicit coordination,* caches send messages to other caches, announcing their state (or state summary) [41, 44, 65]. Upon receiving these messages, a cache can then use the information in the messages to make decisions regarding what to store or evict, as well as where to route misses (i.e., where to search for content). While explicit coordination may be useful, it comes at the cost of increased communication overhead and possibly computationally-expensive coordination algorithms as well.

An alternative approach to explicit coordination is *implicit coordination* among network caches. With implicit coordination, each cache acts based solely on its limited, local view, and does not notify other caches of its state. When such coordination is constructed effectively, the actions taken by caches based on this local perspective can result in favorable results on a system-wide scale. The local view of the network consists, in our case, of the stream of requests received at the cache and the content that have passed through the cache.

Implicit coordination schemes can rely on and leverage the network topology [10], cache management policies [3], or other system parameters. For example, in a cache hierarchy [10], where all requests are forwarded towards the root, the position of a cache w.r.t. the root can define its function within the network; the caches lower down serve one type of request pattern, and shape the miss stream for the next level up. For such hierarchies, it has been suggested that using different replacement policies at different levels in the hierarchy would be beneficial [8]. Others have proposed hierarchy-specific eviction and caching policies [16, 44].

This chapter presents *Breadcrumbs* - a best-effort content search approach that uses *only* implicit coordination among caches[1]. *Breadcrumbs* is "best-effort" - there are no guarantees that cached content will be found and downloaded from a network cache. Therefore, in the event that content search has failed, the request is re-routed directly to the content custodian, where the requested content can always be found. However, we demonstrate in this chapter that, despite the lack of explicit coordination, *Breadcrumbs* can match and even improve upon performance compared to caching architectures that are based on explicit coordination. Indeed, one of our goals in developing *Breadcrumbs* was to investigate how well a simple, implicitly-coordinated caching system would compare to its more stateful, and more complex, explicitly-coordinated counterparts.

The main contributions of this chapter are:

- We describe *Breadcrumbs*, a best-effort content search policy for cache networks, in which each cache determines the next hop to route a request dynamically, based solely on local information (in addition to knowledge of the location of the content custodians). *Breadcrumbs* achieves this by using past traffic to set up breadcrumb entries - short-term routing hints that eventually expire. *Breadcrumbs* is tunable, striking a balance between the route-to-custodian and exhaustive search policies. As we will see, *Breadcrumbs* also fosters implicit inter-cache routing coordination, without involving any inter-cache control overhead.

- For a particular version of *Breadcrumbs*, called BECONS, we prove several properties regarding the efficiency of breadcrumb management. We show that BECONS creates a perimeter surrounding each content custodian, such that

[1]It is important to clarify that our use of the term "Breadcrumbs" is for *request routing* hints, not to be confused with the same term used in [27] for *content forwarding* hints in CCN.

requests originating outside this perimeter are routed, with high probability, away from the custodian. In such a manner, BECONS reduces the load at custodians.

- We present an analysis of causal relationships within the network, specifically between cache state and request routing tables. From this analysis, we devise experiments to demonstrate the impact of *Breadcrumbs*-based search on custodian load reduction.

The remainder of this chapter is structured as follows. In Section 5.2 we discuss related work. In Section 5.3 we present *Breadcrumbs*. As this approach has multiple variations, we focus our discussion on one such version, which we name the *Best-Effort Content Search* (BECONS) policy (Section 5.4). For this version, we prove several properties regarding the efficiency of breadcrumb management. In Section 5.5 we present extensive simulation results of that demonstrate the performance of BECONS, and compare it two other content-search methods - shortest-path routing and an explicitly-coordinated cache-management system defined below. In Section 5.6 we delve deeper into understanding the manner in which *Breadcrumbs* achieves its performance. We use causality analysis to determine the degree to which request routing, and not content distribution, is responsible for the performance observed for *Breadcrumbs*.

5.2 Related Work

5.2.1 Optimizing Cache Networks

Research on optimizing the performance of caching systems touches upon many fields. For small scale caching systems, previous works have examined systems where a small number of caches are placed within a (possibly large) network. These works consider the question of where in the network to place the caches [40], where to cache specific objects [65], and generally address the question in a limited number of topologies. Indeed, it has been recently [21] noted that existing work on networked caching is insufficient for addressing the cache networks proposed for ICN architectures. In this chapter we address the ICN architecture, of caches distributed on a large scale throughout the network, and study how to improve content search within such a network.

Work on improving the performance of large-scale cache networks can be found in [2, 3], where the authors consider how to optimize system performance via an adaptive cache *replacement* policy named ACME. ACME uses machine learning techniques to determine when and what to cache locally, without explicit communication among caches. Specifically, each network cache manages a pool of *virtual* caches. Each virtual cache is assigned a different (static) replacement policy, and simulates the behavior of a network cache had it been using this replacement policy. ACME assigns performance-based weights to each virtual cache and, using Machine Learning algorithms, selects from the virtual cache pool the best replacement policy to apply in the near future. Using this approach, ACME achieves improved performance compared to specific static policies. The *Breadcrumbs* system we present here differs from ACME in that *Breadcrumbs* improves performance via adaptive *request routing*, instead of adaptive caching. In this sense, the two architectures are complementary, making it possible to potentially combine these approaches. Such a task, however, is beyond the scope of this thesis.

5.2.2 Content Search in Cache Networks

Efficient content search has been addressed in the (Hybrid) P2P literature. In the Gnutella P2P system [4], content is located via exhaustive search (in the form of broadcasting requests to all neighbors); others have proposed to limit the search cost by having requests move through the network using random walks [11]. To mitigate communication overhead, peers using more recent versions of Gnutella implement the Query Routing Protocol (QRP) to notify their close neighbors of their cached content; statistical versions of QRP have been considered as well [41]. In [74], network nodes are organized into *semantic groups,* and request flooding is constrained to occur within these groups. Specifically, a node will receive a request for specific content only if it is associated with this type of content. In [42], request flooding is controlled by caching content along the download path within the P2P network. [23, 49] discuss search via expanding ring search and random walks, with [49] using simulations for evaluation while [23] provides theoretical bounds. Our *Breadcrumbs* system differs from all of these in that (a) we consider systems where content location changes dynamically, and (b) requests are routed towards likely locations of content without explicit coordination among caches prior to or during the search process.

In [24], the authors develop a method for ascertaining if certain content is *unavailable* anywhere in the P2P overlay network. Initially, content is searched for via a random walk for a bounded amount of time, after which the search ends if the content was not found. To help reduce the search time of future random walks, a peer that received such a content request logs this occurrence, and any future search for this content that passes through this peer is dropped. As time passes, peers with request blocking information leave the network, allowing new arrivals to support search for this content once more. This enables the system to adapt to changes over time. While in [24] past requests are used to learn what is not available in the network, *Breadcrumbs* nodes keep logs of past requests and downloads, and these are used to determine where copies *might* be found.

Content search in a cache network is also related to some degree to several classic search algorithms proposed within the field of Artificial Intelligence. Network traversal algorithms, such as DFS, BFS, and A*, are not suitable in this context due both to the large network scale and to the dynamic nature of content position. Other tools, such as Markov Decision Processes (MDPs), traditionally assume global knowledge in order to solve the system, and can also have very high time complexity.

5.2.3 Breadcrumbs expansions

Since the publication of *Breadcrumbs* in [58], there have been several followup projects that considered different aspects of this content search approach. In [30] Kakida et. al considered how to ensure that no breadcrumb cycles form within the network. In [66], Tsutsui et. al. considered the performance of a cache network in which *Breadcrumbs* is deployed only at a portion of network caches. They demonstrated that such deployment improves the performance of the network, especially if an overlay network is constructed between the breadcrumbs-supporting nodes. Recently, researchers from NEC and Kansei University have demonstrated a *Breadcrumbs* implementation.

In [16], the authors propose a combination of *Breadcrumbs* with the LCD caching approach presented in [44]. In this system, content is stored in only *one* location along a download path, the position of which is determined by its popularity. Specifically, more popular content is stored further downstream. This differs from the *Breadcrumbs* approach discussed here, in which content is cached at all nodes along the downstream path, though the claims regarding *Breadcrumbs* presented here are equally valid if content is cached at only a subset of nodes along the download path.

5.3 The *Breadcrumbs* Architecture

The *Breadcrumbs* system builds upon the cache network architecture described in Section 2.2, by adding a dynamic request-routing element. Each node maintains two routing tables - the static table $\mathcal{R}_i$ and a dynamic routing table $\mathcal{R}_i^{bc}$. The dynamic table is populated by *breadcrumb* entries, which are logs of recent activity for each file. Specifically, each breadcrumb at v_i is a 5-tuple entry, with at most a single entry per file, containing the following information:

- The file identifier.
- v_{prev} - The node from which the file arrived at v_i.
- v_{next} - The node to which the file was sent from v_i.
- t_j^f - the time when the file passed through v_i.
- t_j^q - the time when the file was last requested at v_i.

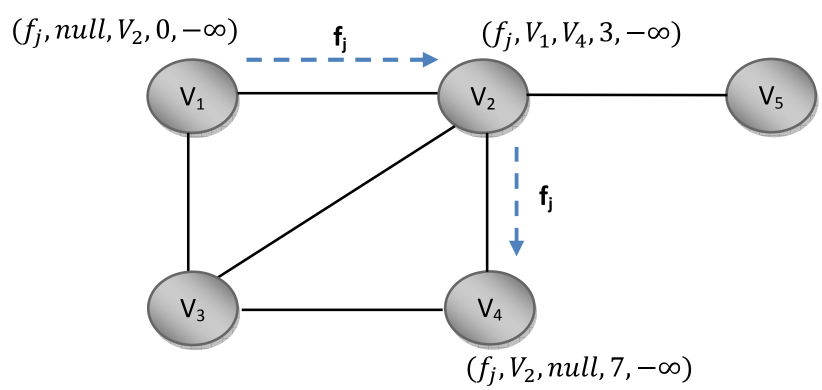


Figure 5.1: Breadcrumbs example

We denote by bc_{ij} the breadcrumb for file f_j at cache v_i; when the cache is known, we use the simpler notation bc_j. In the case portrayed in Figure 5.1, file f_j was sent from a custodian, attached to v_1, to node v_2, and the file arrived at v_2 at time $t = 3$. From there the file was forwarded to the destination at v_4, arriving at v_4 at time $t = 7$. The first and last hops have "null" entries where no cache identifier is relevant. There were no requests for f_j in the scenario shown, so the time of the last request for f_j is set to $t_j^q = -\infty$.

As in the example above, when each file is downloaded it leaves behind a *trail of breadcrumbs* at the caches along its download path, where a *trail* is simply a non-cyclic path in the graph. Each entry along this trail can be thought of as a bi-directional pointer, indicating the upstream and downstream caches along a trail, and therefore where such content might still be currently located. As each breadcrumb is very small, we assume for now that there is no limit on the number of such entries that a node can store at any given time. In practice, the length of time that a breadcrumb is kept in the table depends on both the popularity of the corresponding content and a timeout parameter that can be tuned by the cache (or network) operator. We discuss this timeout parameter below.

We next describe a simple use-case of these breadcrumb entries. Consider again the scenario in Fig 5.1, followed by a request q_j arriving at node v_5. This request is initially routed towards the custodian, to v_2, using the standard routing tables $\mathcal{R}_5$. En-route to the custodian, at node v_2, the cache contents are inspected as in the standard operation of a cache network. In the event that $f_j \notin v_2$, $\mathcal{R}_2^{bc}$ is inspected. If there is a breadcrumb entry for f_j in $\mathcal{R}_2^{bc}$, as is the case in Fig. 5.1, we say that the request has *intercepted* a trail of breadcrumbs for f_j. Using this entry, q_j can be routed either upstream (v_1) or downstream (v_4) towards a cache with a recent copy of the content might be found. Once the content is located, it is forwarded to the requesting user as in the standard operation of a cache network. We emphasize that file downloads *set* the pointers in a breadcrumb, while requests *follow* the pointers. 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 [5].

While arguments can be made for using either of the pointers of a breadcrumb, in this work we limit ourselves to forwarding requests *downstream*, in the direction of v_{next}. The motivation for such a heuristic policy is twofold. First, the request will typically move farther away from the content custodian when routed downstream, thus encouraging load distribution in the network. Second, during the last download content was cached at downstream nodes more recently than at those upstream (due to download delay).

Before we continue to analyze the behavior of *Breadcrumbs*, we note that trails can be *extended* multiple times by repeated downloads. This basic property is shown in Figure 5.2. Here we see that the initial download of f_j generated a trail of breadcrumbs that ended at v_3, and that a future request from v_4 to v_2 (which still had a copy of f_j) changed the direction of the initial path and extended the trail to node v_4, replacing the breadcrumb pointer to v_3. Note that such trail extensions or changes are possible only via content download, which can set the breadcrumb entries, but not via request routing. Also, the times of t_j^f, t_j^q are always monotonically increasing as one proceeds downstream along the *updated* trail.

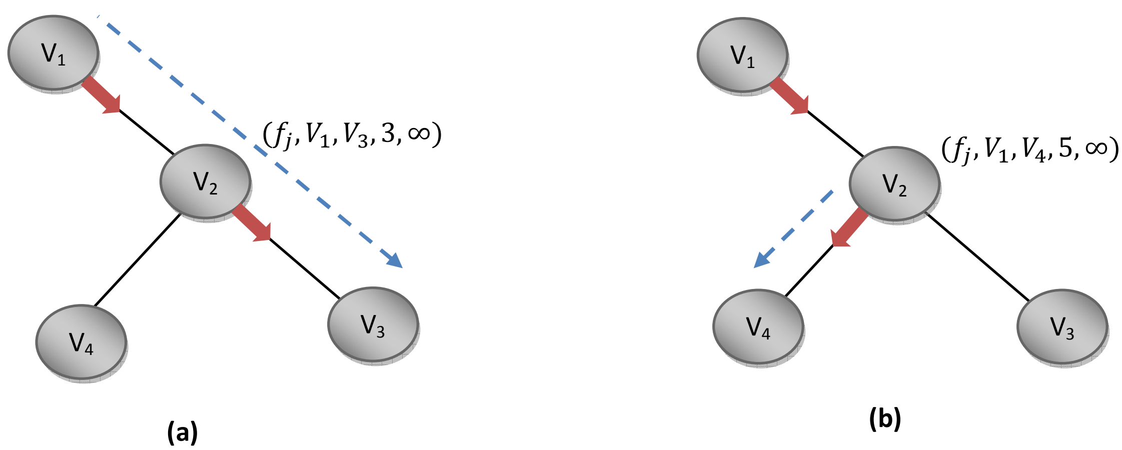


Figure 5.2: Example of trail extension. Broken lines indicate file download, and red arrows indicate the direction of breadcrumb pointers.

Due to the fact that the location of content in the cache network is dynamic, the information in a breadcrumb loses its relevance over time. When it is determined that the information in a breadcrumb is stale (a stale breadcrumb, if you will), it is said to be *invalidated.* An invalidated breadcrumb bc_{ij} is removed from $\mathcal{R}_i^{bc}$, and until f_j passes through v_i once more, all requests for f_j will be routed according to $\mathcal{R}_i$.

In our work, we consider two conditions that, each independently, cause breadcrumb invalidation:

Soft-state timeout. Recall that each breadcrumb logs t_j^f and t_j^q, which are used to determine when an entry contains stale information. Here, a breadcrumb for f_j is invalidated at time t if $t - t_j^f$ and $t - t_j^q$ are each greater than some threshold. The reason for this policy is that (a) when f_j passes through a node v, it will be cached along the trail, and thus it is likely that to remain cached along the trail shortly after t_j^f; and (b) when a request q_j passes through and is routed according to the entry, it might locate the content f_j, and there refresh it (as in LRU) or download it elsewhere. Either way, following the downstream trail can be seen as searching in the direction where content, according to the local view of v, is likely to be found, but only as long as the breadcrumb is fresh.

Different methods can be proposed for how to select this timeout threshold[2], as we discuss in later sections.

Reverse request traffic. A breadcrumb for f_j is invalidated if a request for f_j arrives at the node from the direction of v_{next}, i.e., from the immediate downstream node. The reason for this is twofold. First, the content is not at v_{next}, as indicated by the forwarding of the request from v_{next} to v. Second, a request arriving from v_{next} indicates that the breadcrumb at v_{next} is no longer valid, so a request sent to it will have no trail of breadcrumbs to follow.

For a *Breadcrumbs* network (BCN) as just described, it is possible for a breadcrumb trail to form a directed cycle. A request moving in such a cycle can continuously update the value of t_j^q at each node in the cycle, and thus continue to move in the cycle indefinitely, never locating the content. There are several ways to avoid such cycles,

[2]In this manner, breadcrumbs is a generalization of the standard routing policy, as we can set the threshold to be zero.

for example those proposed in [30]. In this work we do not concern ourselves with a method for cycle detection. Instead, we assume that cycles can be detected, and that when they are detected, $\mathcal{R}$ is used instead of $\mathcal{R}^{bc}$ from that point and on, until the request is satisfied. The same protocol is used if a request follows a breadcrumb trail and reaches a dead end, i.e., a cache without the content or a valid breadcrumb. In Section 5.7 we discuss the impact of allowing breadcrumb entries to be used more persistently, even once a cycle is detected or a dead-end is reached.

5.3.1 File download path

Once a request reaches a cache with a copy of the content, it is downloaded to the requesting user. When requests are routed along the shortest path to the custodian, the download path is identical to the shortest path from where content was found. With *Breadcrumbs*, on the other hand, the request and download path might be different. As such, we consider two options for download policy, depicted in Figure 5.3:

Download Follows Query (DFQ). When the file is downloaded from v_i, it follows the reverse search path that the request traversed.

Download Follows Shortest Path (DFSP). When the file is downloaded from v_i, it is sent along the shortest path to the requesting user.

DFSP has a straightforward advantage: it ensures that the content arrives in the most efficient manner to the requesting user. However, note that the content download path affects the places where content is cached, and so this property does not imply that DFSP also generates the best performance globally, since it might generate sub-optimal content placement that will affect future searches.

Using the DFQ approach has several advantages as well. First, it allows for request aggregation - other requests that are currently following the same breadcrumb trail will be satisfied at nodes along the download path, since the download is moving along the same path but in the opposite direction to these requests. Second, it maintains some of the properties of standard cache networks, allowing analysis tools for standard networks (including those developed in earlier chapters of this dissertation) to be applied to those using *Breadcrumbs*. For example, with DFQ there is still a single f_j download at v_i for every q_j sent by v_i.

DFQ ensures additional properties for the breadcrumb trails as well. Specifically, when we consider the upstream path we find that it retains some properties of shortest-path routing to the custodian, as we state now:

Theorem 43. *The upstream path of a breadcrumb trail when using DFQ for content download is always the shortest path to the custodian, if the initial request routing for each file request follows the shortest path.*

Proof: Consider a request q_j. The initial request routing, until a trail is located, follows the shortest path to the f_j custodian, and once content is located it is downloaded along the reverse query path, which is the shortest path. Since this holds for every request, each upstream path consists of a concatenation of (partial) shortest paths to the custodian. ∎

DFQ has, however, an important drawback. If we follow the *Breadcrumbs* policy in a strict manner, a breadcrumbs path would be reversed upon each successful search, since it reverses the upstream/downstream directions. In Fig. 5.3, the breadcrumb at v_2 will change its pointer direction towards v_5, instead of continuing to point to where content was just found. To negate this continuous change of pointers, we propose the following change in *Breadcrumbs* when using DFQ: *the direction of a breadcrumb is not modified at v when a file arrives at v from the direction of v_{next}.*[3] While this means that the breadcrumb is not always pointing to the most recently-cached copy, it can extend the length of time a trail is maintained.

[3] This is not to be confused with the arrival of a *request* for f_j from v_{next}. A file arrival indicates the content is somewhere downstream, while a request indicates the trail is no longer useful.

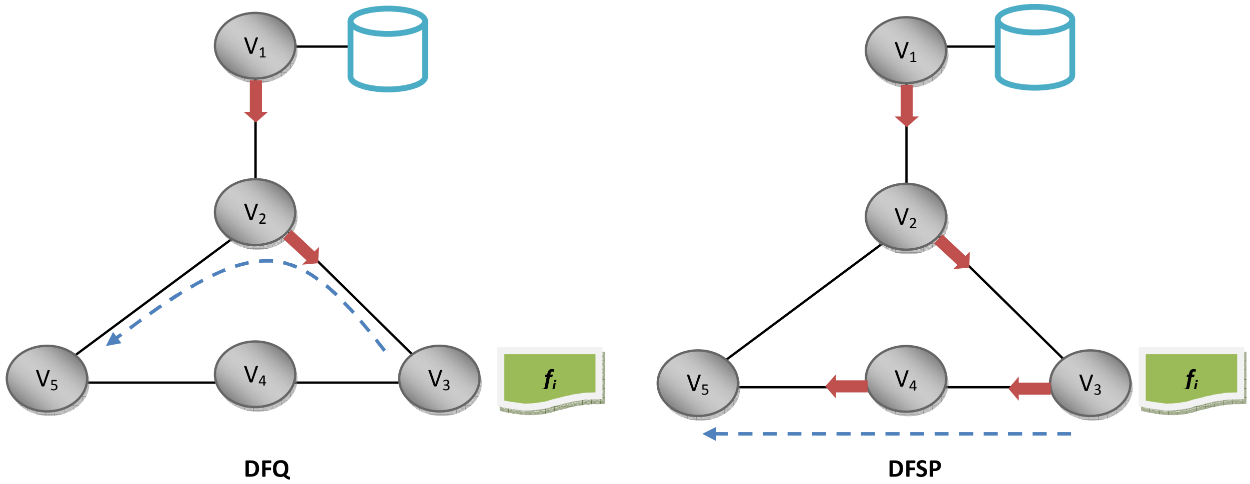


Figure 5.3: Download policies depiction. Initially, the content f_j was downloaded to v_3 via v_1, v_2. Later, a request for this content originated at node v_5, passed through v_2 which did not have the content but did have a valid breadcrumb, pointing to v_3.

5.4 Best-Effort Content Search (BECONS)

In Section 5.3 we presented the basic architecture of *Breadcrumbs*. In this section we propose the following specific instantiation of *Breadcrumbs*, termed the **B**est **E**ffort **CON**tent **S**earch (BECONS) query routing policy. In BECONS, we make the following two decisions regarding *Breadcrumbs*:

Invalidation at content destination. If a node v is the last hop of a specific file f_j, it invalidates any existing breadcrumb for that file once it received the file. We use this property later on. For now, we note that this invalidation policy ensures that a node does not have a breadcrumb that was created earlier than when the content was last at the node. Consequently, if v does not have the content, it does not consider any specific neighbor as especially likely to be storing the content.

Identical Thresholds. For each file f_j there are two thresholds that are identical across all caches, Δ_j^f and Δ_j^q, such that a breadcrumb bc_{ij} is timed-out at time t iff

$$t - t^f_{ij} > \Delta^f_j \text{ and } t - t^q_{ij} > \Delta^q_j. \tag{5.1}$$

That is, a breadcrumb is invalidated and removed from $\mathcal{R}^{bc}_i$ if both Δ^f_j time has passed since the content was last forwarded by this node, *and* Δ^q_j time passed since the content was last requested at this node. Since the passing of a file through v_i is a stronger indication of the file's presence along the trail than the passing of a request for the file, we shall set thresholds such that $\Delta^f_j > \Delta^q_j$ for all $1 \leq j \leq L$. Note that by assuming a common threshold for each file, we are introducing a degree of explicit coordination among caches. This coordination is very limited, though, and is independent of the traffic flowing in the network. Additionally, once network caches share the same threshold values, new caches connecting to the network can easily discover them by querying their neighbors once for these threshold values. We discuss alternatives to this approach in Section 5.7.

We next prove that BECONS has two important properties: trail stability and trail invalidation.

Definition 44. A trail $v'_1, ..., v'_k$ for f_j is said to be *broken* if there exist indices $1 \leq h < i < l \leq k$ s.t. bc_{hj}, bc_{lj} are valid yet bc_{ij} has been invalidated. An example is shown in Fig. 5.4.

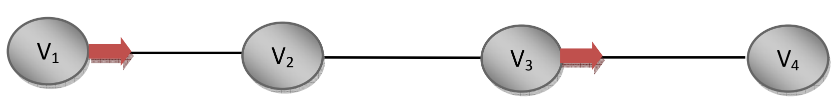


Figure 5.4: Example of a broken trail. A breadcrumb entry is valid at nodes v_1, v_3, but not at the intermediate node v_2.

Definition 45. Consider a trail $v'_1, ..., v'_k$ and a request starting to follow this trail downstream. The trail is said to be *stable* if it does not become broken during the request's traversal of the trail. Thus, if a query starting a search along a stable downstream trail reaches an invalid breadcrumb, this implies that all breadcrumbs further downstream are also invalid.

A policy that ensures stability is advantageous: it ensures that a search downstream will cover all valid breadcrumbs in the trail while searching for the file, and that a dead-end will not be reached due to a single invalidated breadcrumb mid-stream in the trail.

Definition 46. A policy is said to support the *trail obsoleteness* property, if a node v can determine that all nodes along its downstream trail for f_j do not have a copy of f_j *as a result of this trail download.*

There is an important clarification to be made regarding this property. With trail obsoleteness, it is still possible that f_j is in some node downstream when the trail is obsolete. However, this downstream copy was cached as a result of a download along a *different* download trail (i.e., a trail that has some nodes not in the obsolete trail). As a result, there is no reason for v to forward its requests along this trail, as (a) there are no breadcrumbs along this trail that were installed when the file last passed along this download trail; and (b) all copies that were downloaded during that last download have since been evicted. The property of detecting trail obsoleteness thus helps reduce searches according to breadcrumbs that have lost their semantic meaning.

Theorem 47. *Let v_1, v_2 be two nodes such that v_2 is the downstream neighbor of v_1 w.r.t. f_j. If v_1 receives a request for f_j from v_2, v_1 can consider the downstream trail to be obsolete.*

Proof: We prove this by induction on the length of the trail. For the base case of a single link (2 nodes), if v_2 sends a query upstream to v_1, this means that the file is not cached at v_2 and that it does not have a breadcrumb for f_j, so the downstream trail is obsolete. Note that here we rely on the fact that the destination node of a download removes its breadcrumb entry for that node, so v_2 cannot have a breadcrumb from a previous download pointing to v_1.

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. v_2 forwarded a query upstream to v_1, so obviously v_2 does not contain the file. In addition, the query was forwarded upstream instead of downstream, so the breadcrumb at v_2 is invalid. Hypothetically, there can be two possible causes for this:

- The breadcrumb at v_2 has timed out. This, however, is not possible: the breadcrumb at v_1 is valid, and thus the breadcrumb at v_2 must have been refreshed at a later point in time. Since in BECONS we use the same Δ_j^f, Δ_j^q for all nodes, the breadcrumb at v_2 can time out only after the breadcrumb at v_1 has timed-out.

- Node v_2 has received a request for f_j from v_3 downstream. By the induction step, this means the entire trail downstream w.r.t. v_2 is obsolete, and since the breadcrumb at v_2 is invalid and v_2 does not have the content, this property holds for the entire downstream trail from v_1.

■

We next prove that BECONS also has the property of trail stability. For any neighboring nodes v_i, v_k, let $Y_f(i,k)$ and $Y_q(i,k)$ be random variables representing the delays associated with transmitting a file and a request, respectively, from v_i to v_k.

Theorem 48. *Assume that q_j arrived at v_1 at time t when the breadcrumb bc_{1j} is still valid, and assume a (valid) breadcrumb trail exists along nodes $v_1, ..., v_k$. Then*

1. *If $t - t_j^f < \Delta_j^f$, the probability that the remaining trail is stable is bounded from below by*

$$\prod_{i=1}^{k-1} P\left[Y_f(i,i+1) \geq Y_q(i,i+1)\right] \tag{5.2}$$

2. If $Y_q(i,h)$ for all h are constant, this bound holds as long as there is a valid breadcrumb at v_1.

Proof: **1.** Let q_j be a request, and w.l.o.g. $t=0$ be the time at which it starts the search downstream along the trail. At time $t=0$, we know that v_1 has a valid breadcrumb and $t - t_j^f < \Delta_j^f$. This means f_j passed through v_1 within the last Δ_j^f time, the earliest time being $t_j^f = -\Delta_j^f$. Consequently, the content was cached at each downstream node v_h at the earliest at $-\Delta_j^f + \sum_{i=1}^{h-1} Y_f(i,i+1)$. The breadcrumb at v_h will thus timeout at $\sum_{i=1}^{h-1} Y_f(i,i+1)$. Since the request arrives at time $t=0$, the time for it to reach node v_h if no hits occur along the way is $\sum_{i=1}^{h-1} Y_q(i,i+1)$, and we then directly conclude

$$P\left(\sum_{i=1}^{h-1} Y_f(i,i+1) \geq \sum_{i=1}^{h-1} Y_q(i,i+1)\right) \geq \prod_{i=1}^{k-1} P\left(Y_f(i,i+1) \geq Y_q(i,i+1)\right) \qquad (5.3)$$

2. Next we consider the case in which the breadcrumb at v_1 was refreshed by a request at time $-\Delta_j^q$ at the earliest. If this earlier request reached node v_h, then, using the same reasoning as above, we know that this earlier request reached v_h no earlier than time $-\Delta_j^q + \sum_{i=1}^{h-1} Y_q(i,i+1)$, and the breadcrumb will not timeout before $\sum_{i=1}^{h-1} Y_q(i,i+1)$, by which time the new request will reach this node.

Consequently, there can be no breaks in the trail *due to timeouts*. What remains to address is the possibility of a break in the trail due to other types of invalidations — namely, a request backtracking up the trail. However, as we saw in Theorem 47, if this happens than all the nodes from node v_h until the end of the trail have been invalidated, so there is no break in the trail, and stability is maintained. ■

Since requests are likely to be smaller than files, we expect that $Y_f(i,k) \geq Y_q(i,k)$ with high probability, which allows BECONS to enjoy the benefits of trail stability.

A consequence of trail stability is that at every point in time t, each trail has a single *border node*. A border node $v_{border(j,t)}$ is a node on the trail such that:

- Requests for f_j arriving upstream of $v_{border(j,t)}$ will be routed towards the content custodian, as their breadcrumb entries will time-out prior to t.

- Requests for f_j arriving downstream of $v_{border(j,t)}$ will be routed downstream along the trail, as their breadcrumb entries will be valid at t and remain so throughout the search.

Stability ensures these properties since otherwise the trail would be broken at some node along the trail. Note as well that the location of the border node is monotonically-dependant on the threshold values: the longer it takes for a breadcrumb to timeout, the closer this border-node is to the custodian, diverting more traffic away from it to search downstream. Thus, these values can be used as a tuning mechanism for load distribution within the network.

5.5 *Breadcrumbs* Evaluation

5.5.1 Comparison Benchmarks

We compare the performance of *Breadcrumbs* to two alternative cache network management policies. The first is the standard content search policy of cache networks, in which a request is routed along the shortest path to the content custodian, with caches being inspected along the way. The second is a more stateful caching system, which relies on explicit coordination among caches, both for content caching and request routing. This policy, which we refer to here simply as *coordinated caching,* and abbreviated as CC, allows each cache to store content and route requests based on the state of its direct (one-hop) neighbors. For each node v_i and file f_j, let $\eta(i,j)$ be the set of all next-hop nodes when routing q_j according to $\mathcal{R}$. Formally,

$$\eta(i,j) := \{v_k : \mathcal{R}_i(j,k) > 0\}. \tag{5.4}$$

Each node v_i keeps track of each f_j whether or not it is present at nodes in $\eta(i,j)$. This can be achieved either by v_i requesting this information periodically or by each $v_k \in \eta(i,j)$ broadcasting state updates. With this state information available, the following rules are followed:

- If a request q_j arrives at v_i and $f_j \notin v_i$, v_i will check if there is a $v_k \in \eta(i,j)$ s.t. $f_j \in v_k$. If no such node exists, routing follows according to $\mathcal{R}_i$ as usual. If a single such node exists, q_j is routed to it. If there is a set of such nodes $\eta^*(i,j)$, $|\eta^*(i,j)| > 1$, q_j is routed according to $v_k \in \eta^*(i,j)$ with probability proportional to $\mathcal{R}_i(j,k)$.

- If $f_j \notin v_i$ passes through v_i, it is only cached at v_i if f_j is not in any next-hop nodes (i.e., if $f_j \notin v_k$ for all $v_k \in \eta(i,j)$). In this manner, content replication in neighboring nodes is avoided (to some degree[4]).

- If $f_j \notin v_i$ passes through v_i and is cached at v_i, we evict content at v_i that is already present in next-hop neighbors if possible. Specifically, we select for eviction a $f_h \in v_i$ s.t. $f_h \in v_k \in \eta(i,h)$, if such a file exists in v_i. When several such files exist, we evict according to the replacement policy of the cache. For example, with LRU we will evict the least-recently used file $f_h \in v_i$ that can be found in a neighbor in $\eta(i,h)$.

5.5.2 Simulation Setup

As *Breadcrumbs* does not assume ZDD, the experiments we ran allowed for constant request and content propagation delay. The request rate was set to 10 requests per time unit arriving at each node, the rate of content propagation was set to be equal to this rate, and query propagation was set at double the rate. Also, with tree

[4]The same content can still be stored in two neighboring nodes since the relationship is directional - v_i does not store a file f_j that is in $v_k \in \eta(i,j)$, but there is nothing to stop v_k from storing content that is already present in v_i.

topologies being less interesting in the context of content search, we used the torus topologies as in Chapter 2. Specifically, we used the same custodian placement and exogenous request generating statistic as in Section 2.5.

We experiment here using BECONS, and it is to this version of *Breadcrumbs* that we refer to in the discussion below when talking about the performance of *Breadcrumbs*. We set, in all our experiments, $\Delta_j^f = \Delta_j^q$, and the values shown in the figures below are in time-units.

5.5.3 Performance Metrics

We consider here several metrics for evaluating *Breadcrumbs* Cache Networks (BCN):

Network-wide hit probability (or: reduced custodian load). An improved search policy is expected to locate a copy of content in *some* network cache with higher probability than with alternative policies. We refer to these as *cache network hit probability*. As a result of this increase, the load at custodians is reduced.

Search and download distance. We are also interested in the quality of the search process. One method for evaluating this is to count the number of hops it took to locate the content copy; the shorter these paths are, the smaller the delay experienced by a content consumer. Similarly, the distance that the content must traverse during download should also be minimized.

Search efficiency. We also consider the ratio between the search and download distance. A low ratio implies that the search was focused, directed early in the search towards where content was eventually located. Note that with shortest-path routing as well as *Breadcrumbs* with DFQ this ratio is always 1.0, as the search and download paths are identical. This metric is thus of interest only for *Breadcrumbs* with DFSP.

In what follows, we considered these values both on a per-file basis as well as globally, for the entire set of requests combined.

5.5.4 Performance Evaluation

We now investigate the performance of *Breadcrumbs*, beginning with the **network-wide hit probability**. Figures 5.5a-5.5b show the fraction of requests satisfied at some cache (and not a custodian) for *Breadcrumbs* using both DFSP and DFQ, shortest path routing (labeled "Greedy" in said figures) and (most interestingly) CC. In these and all results shown in this section, 90% confidence intervals are shown. Figures 5.6a-5.6b show these same results, but only for the most popular files. These results indicate that *Breadcrumbs* improves upon both shortest path and CC, especially for the most popular files. The total hit probability for all files combined is also better with *Breadcrumbs*, as shown in Fig. 5.7a. We also see in these figures that DFSP demonstrates slightly better performance than that of DFQ, and so in what follows we present, at times, simulation results for DFSP only.

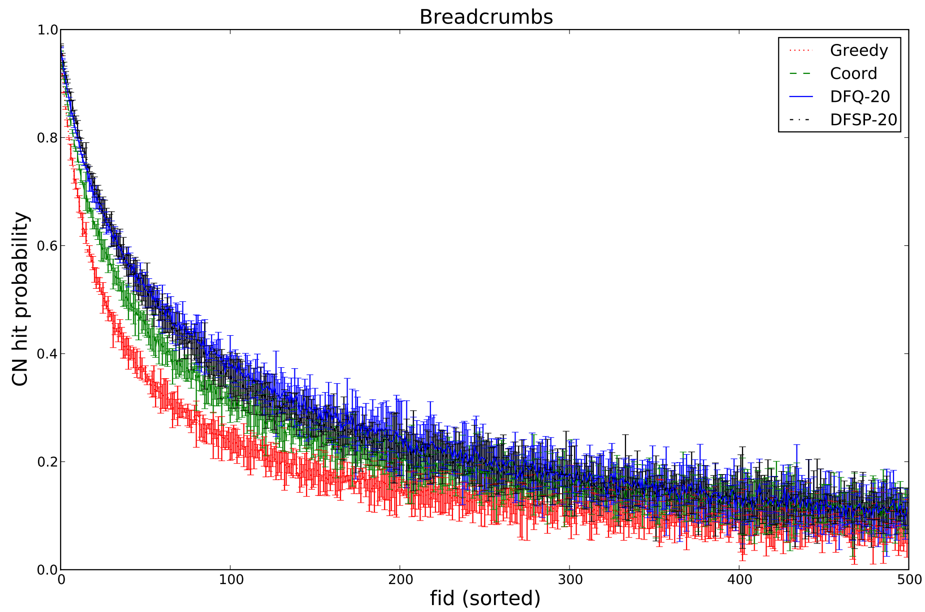
These figures show how the margin of improved performance relative to CC increases when the cache size is smaller (compared to the number of files, as we saw with a-NET). In the examples shown here, *Breadcrumbs* has 3% more hits than CC when cache size is 20, but 8% more when cache size is halved to 10. One possible explanation for this phenomenon is that CC improves performance mainly by having neighboring caches act as a single cache under central control. When the radius of such cooperation is finite and does not scale with network size, the gains are thus bounded by the combined cache size. With *Breadcrumbs*, on the other hand, there is no limit to the length of the breadcrumb trail, and so the number of network hits does not decrease as fast.

With the same reasoning in mind, we consider another feature that can impact performance - network size. As the scale of the network increases, there are more opportunities for locating content en-route to the custodian, and more opportunities as well for *Breadcrumbs*. Fig. 5.7b demonstrates this when scaling from a 10x10 to a 15x15 torus topology. As we can see, the ratio between *Breadcrumbs* and shortest path routing remains at a steady 20%, but compared to CC there is an increase in the relative gain - from 8% for the smaller network to 10% for the larger one. Thus, as the network size grows so to do the benefits from *Breadcrumbs* become more pronounced compared to coordinated methods with fixed radius.

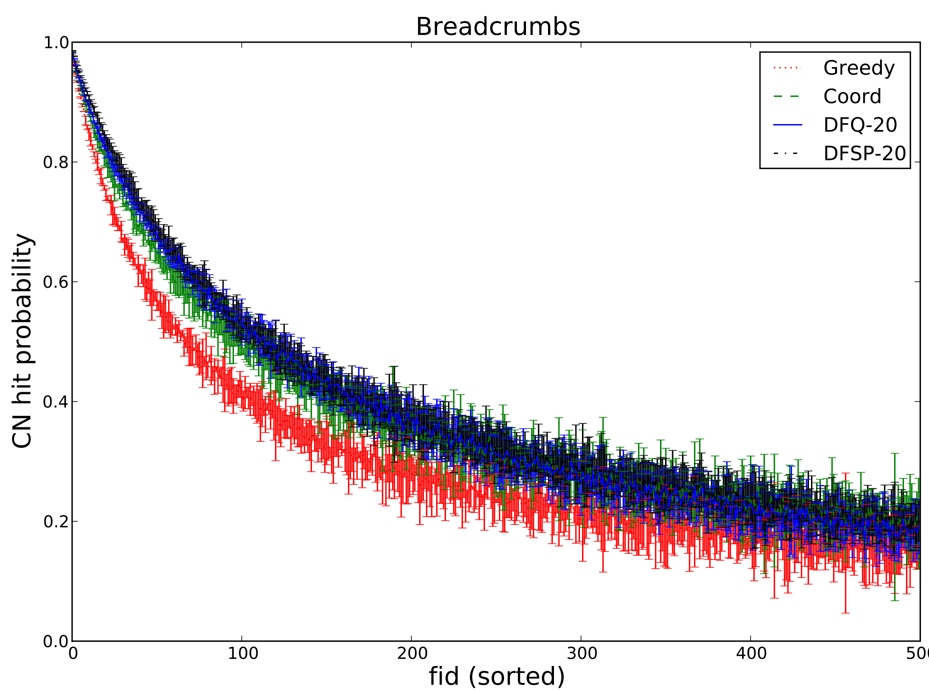
The effects of the timeout threshold were also investigated. As these thresholds are increased, *Breadcrumbs* are allowed to remain valid longer and so the search can be extended along longer paths, providing additional opportunities to locate content. The results in Figure 5.8 show that increasing the TTL from 5 to 20 time units increases the hit probability of *Breadcrumbs*. We consider the impact of this increase on the search efficiency below.

We next consider performance regarding the search and download paths. Regarding download paths, in Figures 5.9a-5.9b we find that *Breadcrumbs* does not only locate more content within the network, it does so at a location closer to where the request originated from, when comparing to shortest-path routing. This result illustrates that some degree of *load-balancing* among the nodes is taking place with *Breadcrumbs*, allowing each node to find a content copy closer to it. CC still outperforms *Breadcrumbs* in this regard, and additional experimentation is required to determine how this property scales with network size.

This improved search comes at a price, however, of increased search cost. Figures 5.10a-5.10b show the number of search hops as a function of file popularity. Several properties can be observed from these results. First, we see that increasing the TTL threshold of breadcrumb timeout extends the search path length. Of special interest is the search length increase for semi-popular files in the range $f_{20} - f_{40}$. In Fig. 5.10a we can see that, for the benchmark algorithms, the search length is monotonically

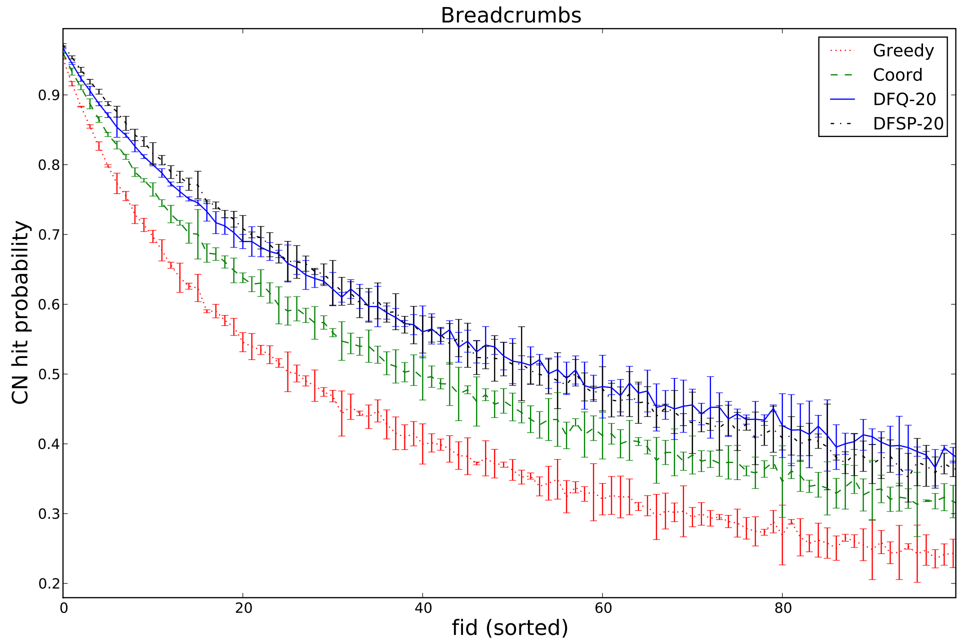


(a) c=10.

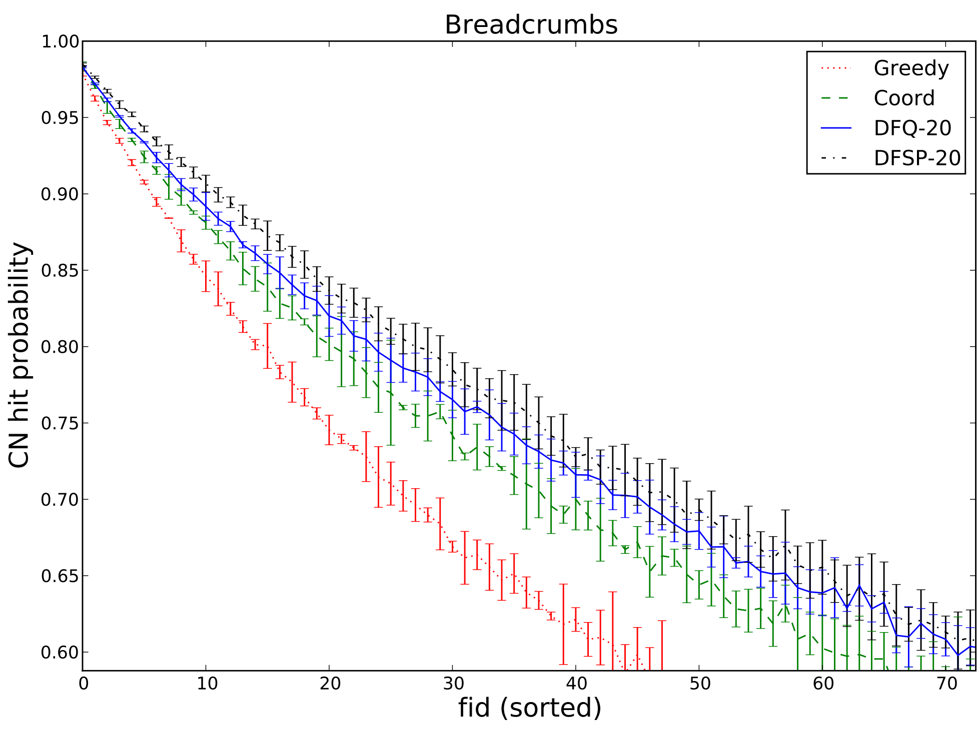


(b) c=20.

Figure 5.5: CN hit probabilities, broken down according to file IDs. Popular files have lower indices. The impact of *Breadcrumbs* is mainly on the popular files. 90% confidence intervals shown. See Figure 5.6 for these results but focusing on the popular files.

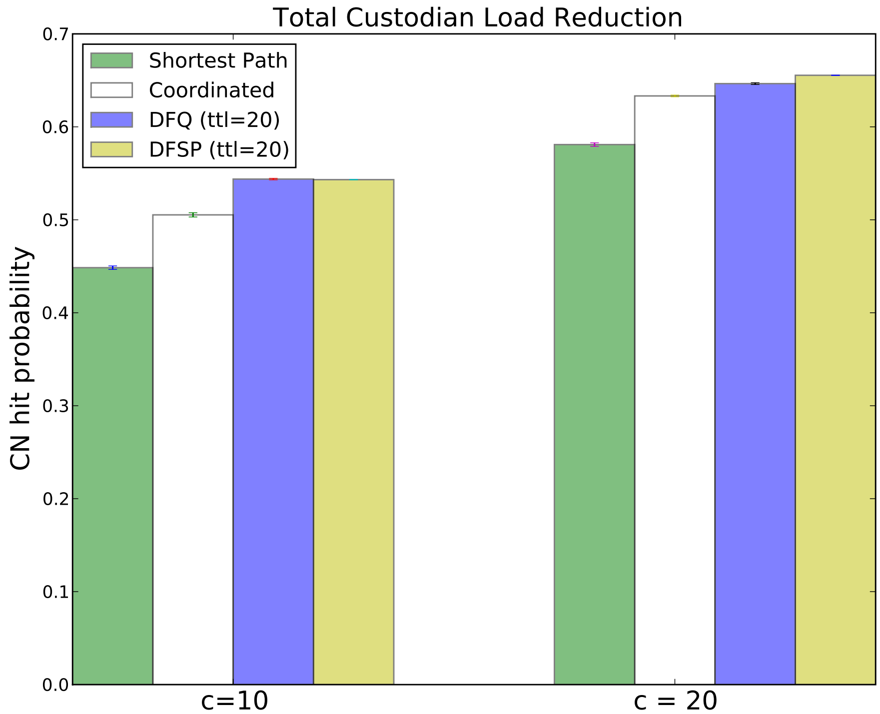


(a) c=10

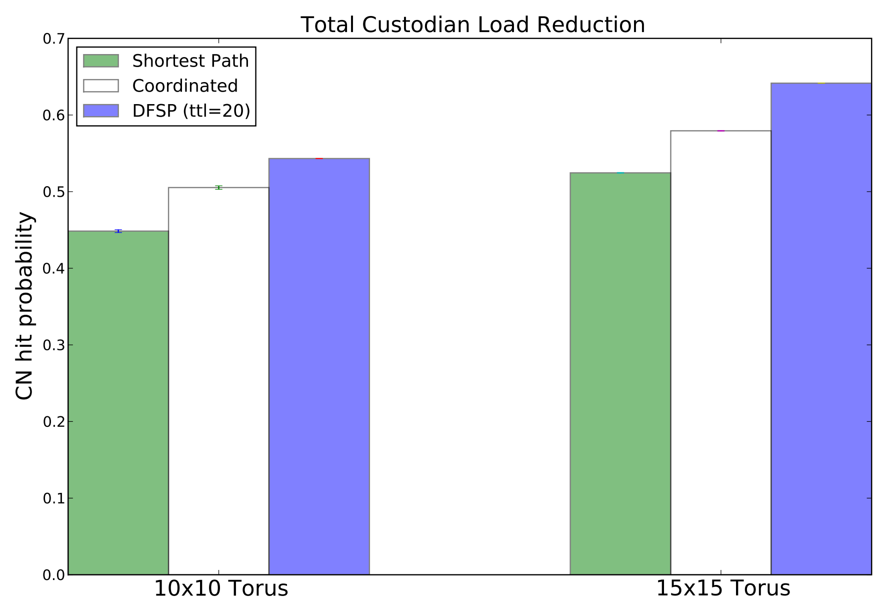


(b) c=20

Figure 5.6: CN Hit probabilities, broken down according to file IDs, and showing popular files. Popular files have lower indices. 90% confidence intervals shown.



(a) Impact of cache size. For c=10, the values shown are: Shortest path = .448, Coordinated = .503, DFSP = .543. For c=20, the values shown are: Shortest path = .580, Coordinated = .633, DFSP = .655



(b) Impact of network scale. For 10x10 torus, the values shown are: Shortest path = .448, Coordinated = .503, DFSP = .543. For 15x15 torus, the values shown are: Shortest path = .524, Coordinated = .579, DFSP = .641.

Figure 5.7: CN Hit probabilities, as impacted by cache and network scale. 90% confidence intervals shown.

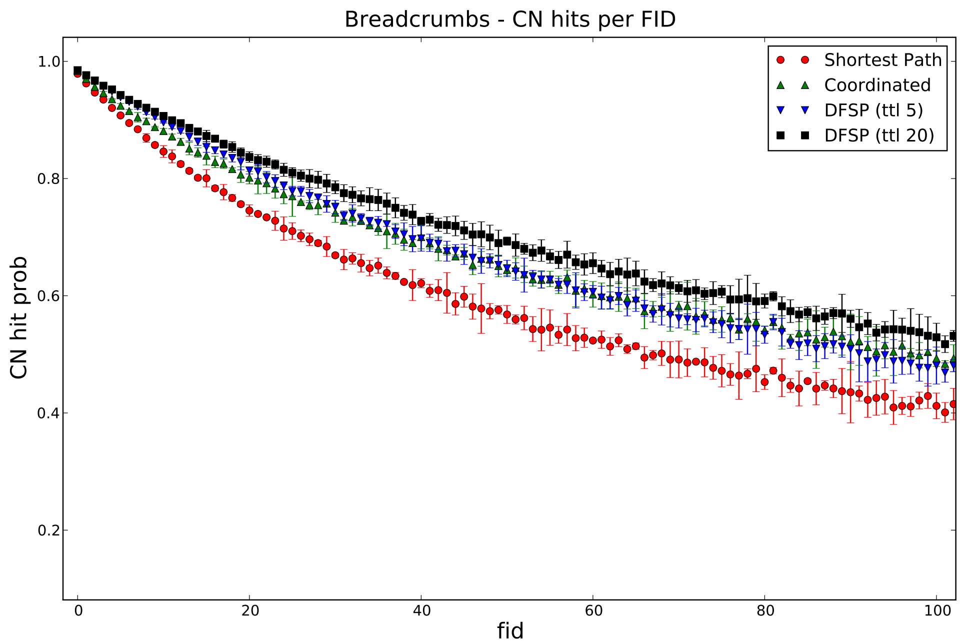
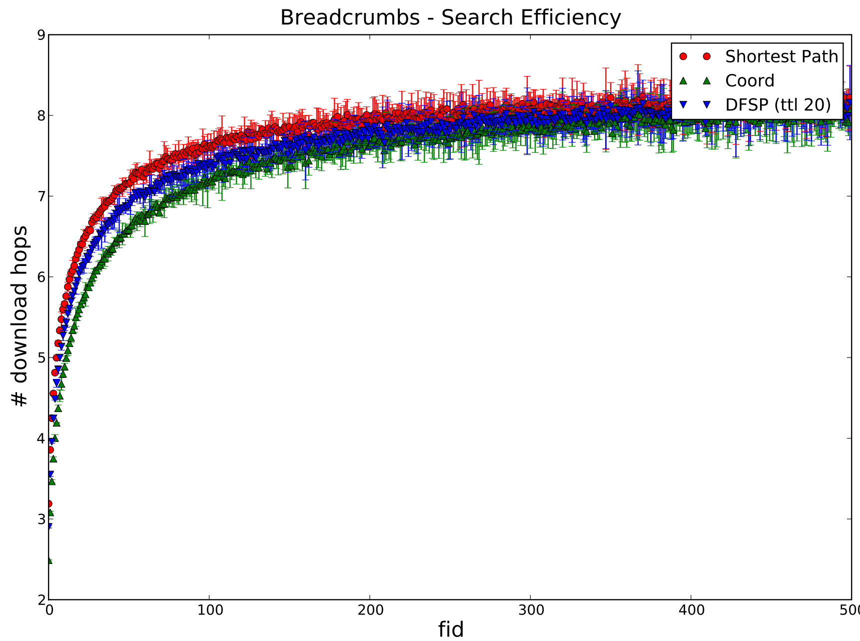


Figure 5.8: The impact of the timeout threshold, or *time to live (TTL)*, on performance. As we can see, with longer TTL the hit probabilities increase. Popular files shown.

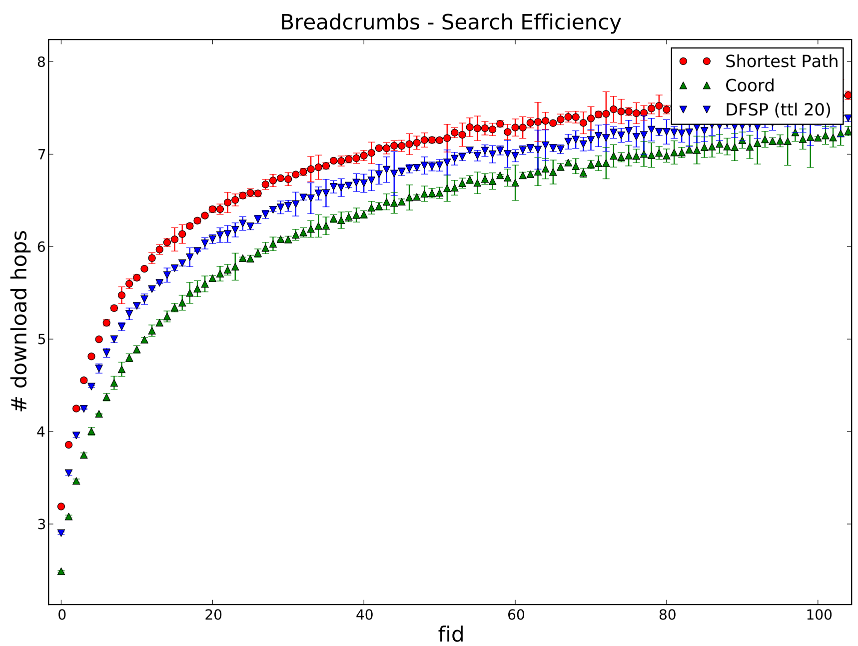
increasing for all files as their popularity decreases. For *Breadcrumbs*, however, we can see that the search length for these semi-popular files is actually longer than for the unpopular files.

One possible explanation for this phenomenon can be found in Figures 5.11a - 5.11b, which show the ratio of the number of search hops and download hops. Here we see once again that the search efficiency is worst for the semi-popular files, indicating a long search path relative to where content is actually found. We conjecture that this is because popular files are located quickly along breadcrumb trails; unpopular content is found less often but its breadcrumb trails are also shorter, ending in dead-ends much quicker; but semi-popular content falls in a middle category, where there are long breadcrumb trails as a result of multiple downloads but which nonetheless eventually end in a dead-end.

Another important factor that emerges from the increase in both the cache-network hit probability and the search path length is what might be considered a local/global tradeoff: while the number of hits within the cache network grows, thus increasing the network hit probability, the *individual* caches experience a *decrease* in their hit probability (Fig. 5.12).

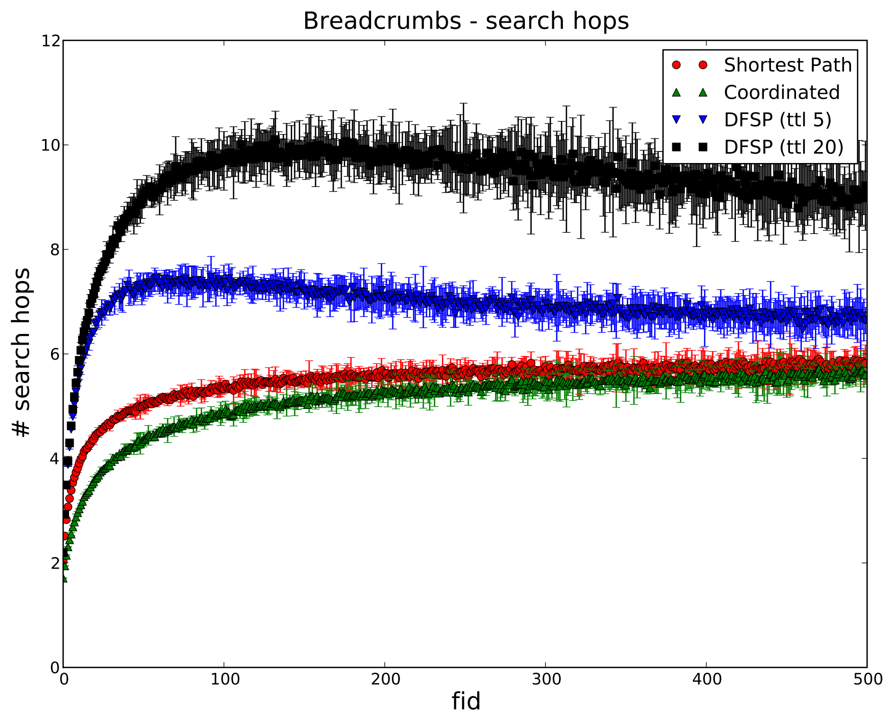


(a) Download hops for all files

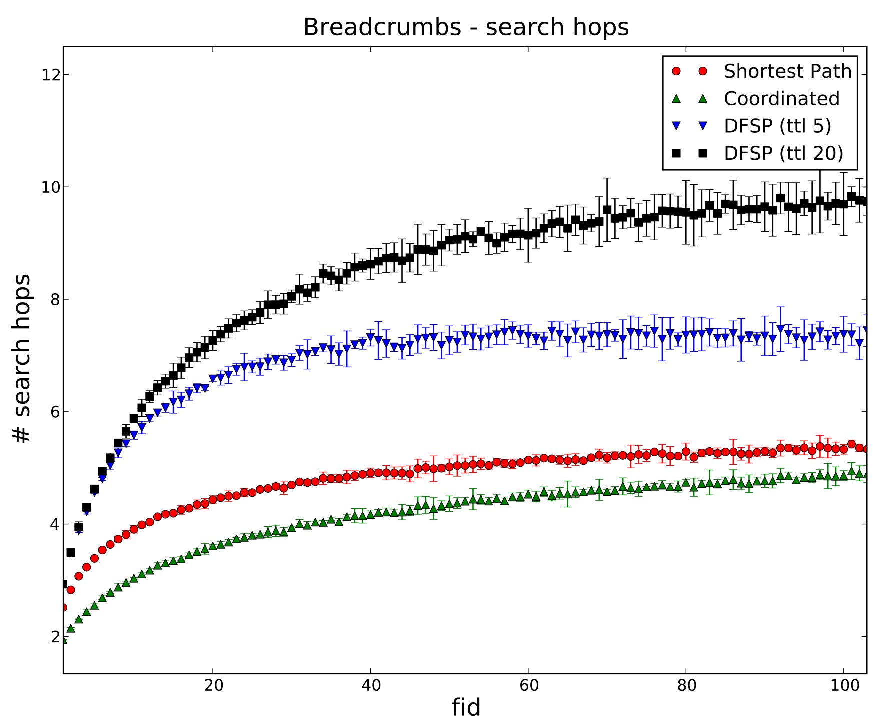


(b) Search hops for most popular files

Figure 5.9: Mean search hops, broken down according to file IDs, for a 15x15 torus. Popular files have lower indices. 90% confidence intervals shown.

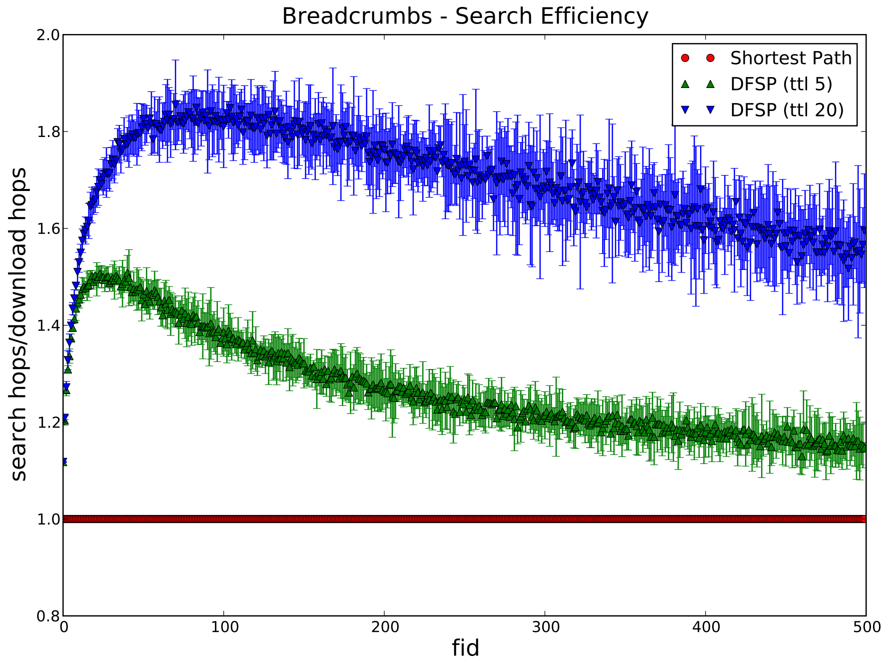


(a) Search hops for all files

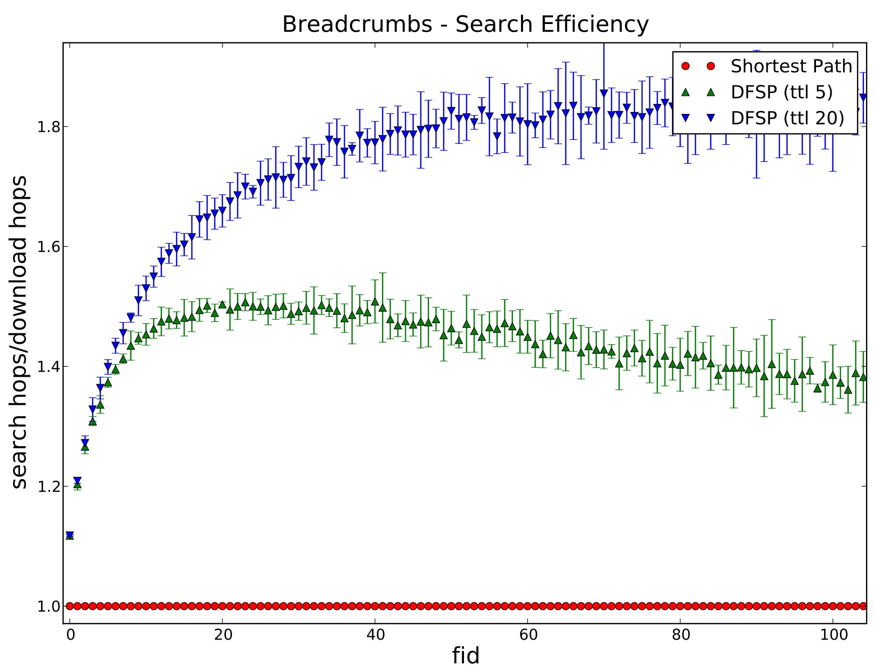


(b) Search hops for most popular files

Figure 5.10: Mean search hops, broken down according to file IDs. Popular files have lower indices. 90% confidence intervals shown.



(a) Ratio for all files



(b) Ratio for most popular files

Figure 5.11: Ratio between search and download hops, broken down according to file IDs. Popular files have lower indices. 90% confidence intervals shown.

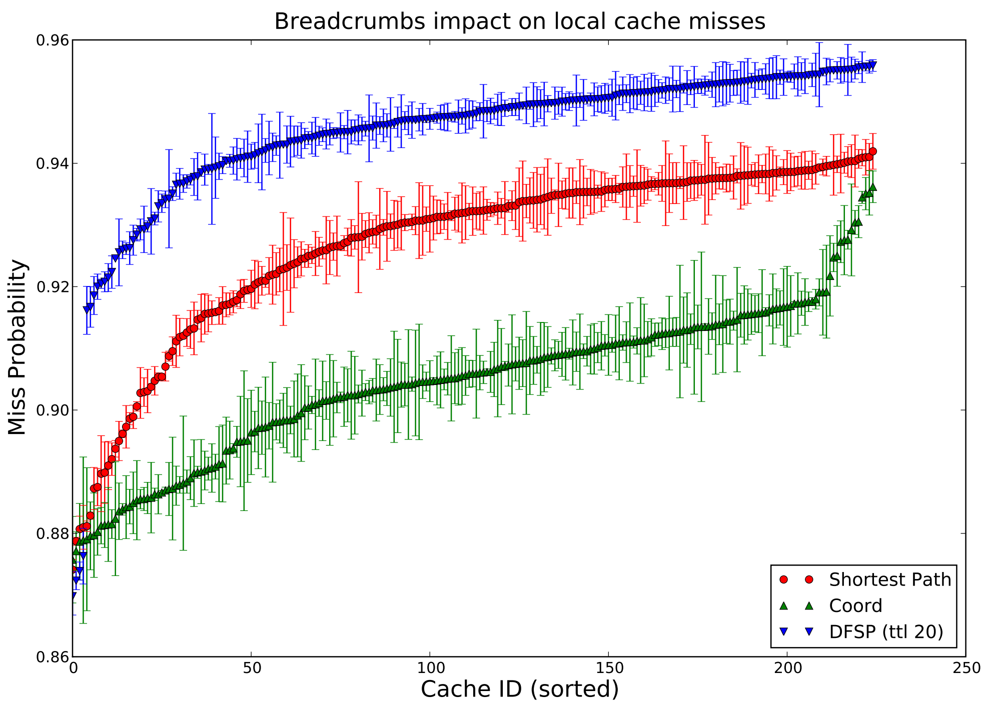


Figure 5.12: The impact of using *Breadcrumbs* on local cache miss probabilities. As we can see, with *Breadcrumbs* the miss probabilities per cache grow, even though globally the network satisfies more requests.

5.6 Causality analysis - cache contents vs. search policy

As observed in Section 5.5, *Breadcrumbs* reduces the load on custodians while reducing the download distance, compared to shortest-path download policies traditionally proposed for ICNs. The cause for this improvement in performance, however, has yet to be isolated. In a *Breadcrumbs* Cache Network (BCN), both routing and cache contents affect the system behavior. Indeed, each of these two factors - caching and routing - impacts the state of the other. On the one hand, request routing impacts the eventual content download path, which determines where content will be stored. On the other hand, the distribution of content determines at what node a request will be satisfied; with *Breadcrumbs*, the request path determines those nodes that will have their breadcrumb entries refreshed during content search.

In this section we are interested in the impact that each of these two factors - caching and routing - has on the system behavior. We are specifically interested in answering the question: to what degree does *Breadcrumbs* reduce custodian load via effective content search, and to what degree is this a result of improved content placement?

Taken as phrased here, this question can be construed as meaningless: the performance of any cache network is not the outcome of one of these two factors, but of *interaction* between them. We therefore consider two *well-defined* variations of this question:

- In the *strong* version, we limit our focus to network scenarios where content state is not impacted by the request routing policy. In such scenarios we can manipulate the routing tables without affecting cache contents, and by observing the outcome we can determine the impact of the routing policy. We discuss this approach in Section 5.6.2.

- In the *weak* version, we limit our investigation into the degree to which *Breadcrumbs* takes advantage of the cache state it creates. We do so by considering a system where content caching is governed by *Breadcrumbs* but content search follows a static policy such as shortest-path search. We discuss this approach in Section 5.6.3.

5.6.1 *Breadcrumbs* Causality Model

Before we move on to considering the strong and weak versions of *Breadcrumbs* impact evaluation, let us define the challenge and the solution approaches formally. To this end we consider the partial DAPER model presented in Fig. 5.13. DAPER (Direct Acyclic Probabilistic Entity Relationship) models are used to express the relationships between entities in a system. In the format we adopt here, the entities are denoted using rectangles and the variable nodes (denoted with ellipses) that are related to each entity are placed within its rectangle. Directed edges in this model indicate *causal* relationships: an arrow from variable node A to B indicates that manipulating the state of variable A will probabilistically affect the state of variable B. The model presented here is *partial* in the sense that it does not show all the entities and attributes that make up a cache network, only those of interest to us.

One important feature of the model shown in Fig. 5.13, and which sets it apart from standard DAPER models, is that it replicates variables to reflect the impact of time on the system. Thus we have two variables for the routing tables and two for cache contents, to reflect the impact of past states on future states.

Let us now consider Fig. 5.13 in detail. It shows three entity classes - caches, routers and servers (custodians), and directed links that represent the causal relationships between the classes. Each of the entity classes shown represents all the entities of that type in the network, and causal edges indicate that each entity of the "cause" variable affects one or several entities of the "effect" variable.

In order to reflect the differences between a standard cache network (CN) and a BCN, we use black directed links to represent the causal relationships in a standard

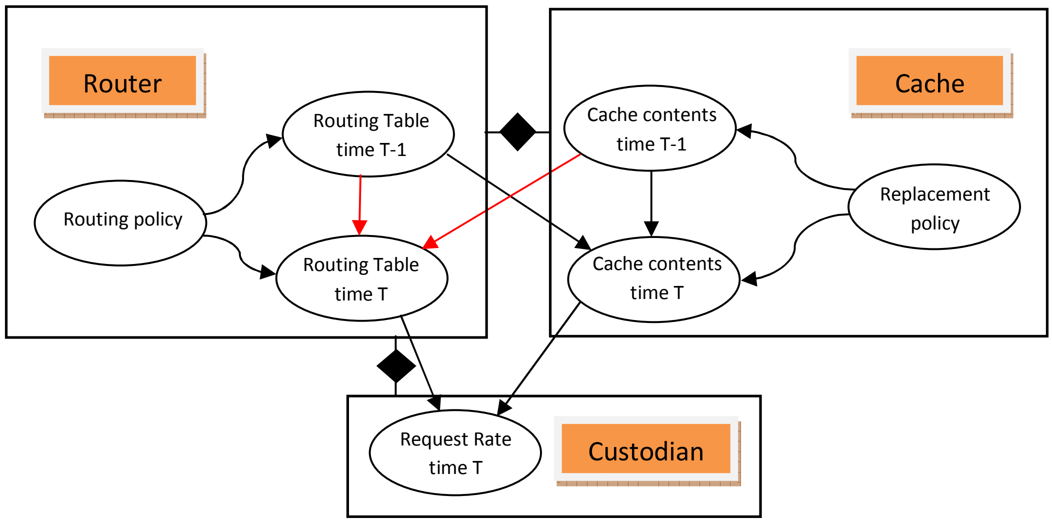


Figure 5.13: Partial DAPER model of Breadcrumbs system, focusing on custodian load as affected by routing and cache contents. Each logical entity represents possibly multiple physical entities in the network.

CN, where content is routed directly to the custodian; for the case of a BCN, the directed edges in red should also be added to the model.

This model demonstrates the argument presented informally above. For the case of standard cache networks:

- The routing remains static over time (assuming no changes in the network structure), and thus there is no causal link between the routing table at different times, given the routing *policy*. This can be seen from the lack of a directed edge from the routing table state at time $T-\epsilon$ and to that at time T. Note that both are affected by the same routing policy, which remains static over time.

- Cache contents at time T is determined by the state of the cache at earlier points in time, as well as the request stream arriving at it, which is controlled in part by the routing tables. Thus we have directed edges from both routing tables and caches from earlier points in time to the state of the cache at the current time.

- Both request routing and cache contents will affect which requests arrive at the content custodian.

With *Breadcrumbs*, two additional links are added, shown in red in Fig. 5.13:

- Routing policy supports changes to the routing tables over time. Thus, the effective routing tables - the combination of both static and breadcrumbs routing tables - are dependent on those from the past. Thus, we have an edge from past routing tables to the current ones at time T.

- Cache state can impact routing tables, as the cache state determines where a request for content will be satisfied, which in turn impacts which breadcrumb entries are refreshed.

When all the links, both black and red, are present in the model, it is clear that both cache state and routing tables impact the eventual load at custodians. However, if we were to conceive of a system where some of these causal links could be removed, we might be able to make a distinction between caching and routing impact on performance; and this is what we set out to do in Sections 5.6.2 and 5.6.3.

5.6.2 The impact of *Breadcrumbs* routing in Random replacement network with limited caching.

In Figure 5.14 we specify the causal model for the effects routing can have on cache contents. As is evident here, there are two such effects:

- Depending on the replacement policy, when a request arrives at a cache and generates a cache hit, this can affect the order of future file evictions. For example, with LRU caches a cache hit impacts the file ordering within the cache, impacting future evictions and hits.

- Along the download path of f_j, the file is stored in caches along the way, changing their state.

Parameter	Value
Topology	Torus
Dimensions	10-by-10
# files	500
File request distribution (each user)	Zipfian
File request rate (each user)	10 requests per unit time
File placement in network	4 sources, equally distanced
Propagation delay	0

Table 5.1: List parameter values used for causality investigation.

An example of these effects is demonstrated using the scenario shown in Fig. 5.15, depicting a portion of a cache network. A request q_j originating from node v_1 can be routed along the shortest path to the custodian (path $v_2 - v_4$) or to follow the breadcrumb trail ($v_5 - v_7$). Assume that f_j can be found either in node v_4 or v_7, but not in any intermediate nodes along these paths. Along the path the request follows, it will affect the state of all nodes along the path either via cache hits (as with LRU) or by content download and evictions.

Under the assumption of causal completeness - that the model in Fig. 5.14 includes all the directed causal paths from routing to cache contents - we propose the following BCN scenario where we show that routing does not affect cache contents:

Replacement Policy. Caches use RND instead of LRU as the replacement policy. As discussed earlier in this dissertation, cache hits have no impact on the cache state of a Random replacement cache.

Admission control. Caches only store the contents for requests that arrived at the cache *exogenously* - we call this *limited* placement. In Fig. 5.15, this would correspond to caching f_j only at node v_1 and not in any of the intermediate nodes along the download path. With this limited placement policy, the download path does not affect cache state.

Download Delay. As we have through much of this work, we assume ZDD.

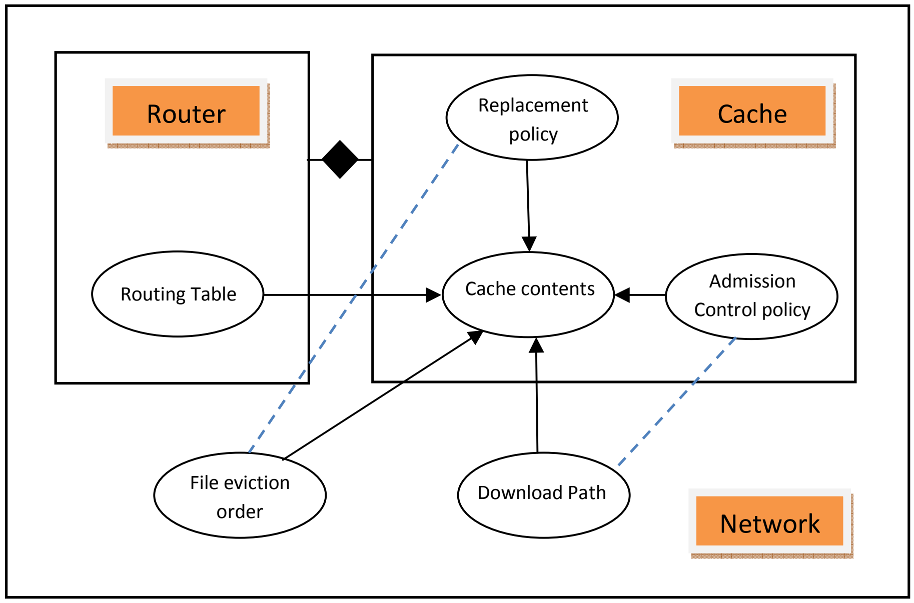


Figure 5.14: DAPER model for the causal links from routing to cache content. Blue dotted lines connect one policy variable to one other variable. This (non-standard) notation indicates that whether or not the attached variable will have any impact on cache contents will depend on the policy variable.

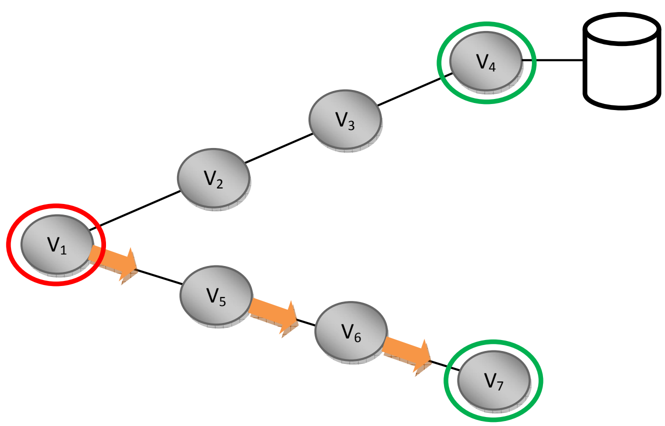


Figure 5.15: Topology portion, depicting the different paths affected when following a breadcrumb trail vs. the shortest path to the custodian. The origin of the request is v_1, and f_j can be found at both v_4, v_7, which are circled in green.

For such a system, cache state is agnostic to search policy, and a corresponding causal model would not have an edge from former routing tables to cache contents. As such the only differentiator between a CN and a BCN is the search policy. We can therefore compare CNs to BCNs in such systems to determine the impact of content search policies on custodian load (or any other metric). We simulated such systems, for 10x10 torus topologies with file popularity following Zipfian distribution, and content distributed among four custodians as above, with varying cache sizes of $5, 20$ and 40, and to see the impact of this search policy routing on performance. The results are presented in Figure 5.16, using two metrics. Let Q be the total number of requests that were generated by users in the simulation, $hits(CN)$ and $hits(BCN)$ are the number of these requests served by the cache network and not by custodians, for CNs and BCNs respectively.

1. The relative increase in cache hits *compared to standard CNs* is shown in the white bars (left), which is computed

$$\frac{hits(BCN) - hits(CN)}{hits(CN)}.$$

2. The increase in *hit probability* is shown on the right, in the yellow bars, and computed

$$\frac{hits(BCN) - hits(CN)}{Q}.$$

For example, for the case of caches of size 20 BCN served approximately 10% more requests than standard CN, which constitutes of an increase of 6% of total requests that were sent into the system.

Figure 5.16 shows that, as the cache size becomes smaller, the added performance of *Breadcrumbs* goes up. This can be explained by noting that as caches become smaller, the probability of a cache miss goes up, and so more requests follow breadcrumbs than with bigger caches. This corresponds to what we saw in our evaluation of

Breadcrumbs, where a rise in L/c ratio makes the *Breadcrumbs* improvement margin grow compared to both shortest-path and CC caching policies.

5.6.3 The utility of *Breadcrumbs* routing in general BCNs

The approach outlined in Section 5.6.2 is clearly limited to specific instances of *Breadcrumbs* systems. For the general case, we limit ourselves to the weaker version of our analysis, and focus on gaining some insight into the degree to which the search policy utilizes the content distribution in the network. To this end, we construct an experiment that will distribute content according to *Breadcrumbs*, while content search will be conducted according to a different policy - in our case, by routing requests according to $\mathcal{R}_i$ for all nodes. The resulting simulation is termed here a *quasi-BCN*. Since a quasi-BCN shares the same content distribution as a BCN, the only distinguishing feature is the content search. By comparing the performance of *Breadcrumbs* to that of the quasi-BCN, we can thus gain insight into the added benefit *Breadcrumbs* request routing brings to the system.

To generate this quasi-BCN, we take each exogenous request and represent it as *two* requests that flow through the system, affecting it in distinct ways:

- The request $q_j^{(state)}$, and the eventual download of f_j, affect the state of the caches w.r.t. f_j. The request follows breadcrumb trails when available, refreshes content in caches with f_j according to the replacement policy, and downloads content from caches or custodians, wherever found. However, when content is located, we do not log this event as a custodian or cache download. We refer to these as *state-requests.*

- The request $q_j^{(log)}$ logs the performance of the system in terms of custodian request rates. This request is routed to the content custodian along the shortest path. If content is found at some cache, it does not affect cache state, and when downloading f_j from where it was found it does not affect cache or router state.

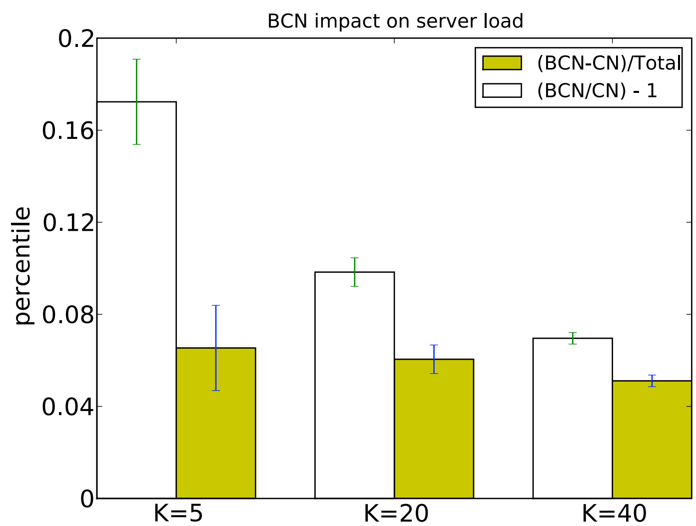


Figure 5.16: The performance increase due to efficient routing with RANDOM replacement and limited placement, for caches sizes $k = 5, 20, 40$. The white bars (left) represent $(hits(BCN) - hits(CN))/hits(CN)$, the fractional reduction in custodian load when moving to BCN. The yellow bars (right) show the fraction of requests sent int o the system that were served by the network due to BCN routing. 95% confidence intervals are shown.

The only impact this request has is on the logged events - cache hits and misses, as well as custodian request rates, which are logged. We refer to these as *log-requests.*

From the definitions above, we know that (a) only state-requests affect cache contents, and (b) at the caches are populated just as they would be for the corresponding BCN. Log-requests, on the other hand, are routed along the shortest path to the custodians, and thus are impacted by cache state but not by the $\mathcal{R}^{bc}$ routing tables. Thus, in this quasi-BCN, there are no links between logged routing (which is static) and each cache content populated using *Breadcrumbs*.

We now compare the load on the custodian for a BCN and quasi-BCN. The cache state in both is identical for each point in time, but the request routing differs: BCN uses *Breadcrumbs* while the quasi-BCN uses shortest path routing to the custodian. We can thus observe in comparison how much *Breadcrumbs* takes advantage of the specific placement of content generated by *Breadcrumbs*.

Note that the selection of shortest-path routing is simply one of convenience, and any static routing scheme can be used for comparison. Indeed, this evaluation process can be repeated several times with different static routing policies, to determine with higher certainty the contribution to performance of content search.

We ran several quasi-simulations and compared their performance to that of standard CNs and to BCNs. Figure 5.17 shows the results for caches of size 20 and Zipfian popularity distribution. We found that these results held also when making caches larger and smaller. As can be seen here, the quasi-BCN yields performance slightly worse than that of a standard CN. These results would indicate that the content search policy has a strong impact on performance.

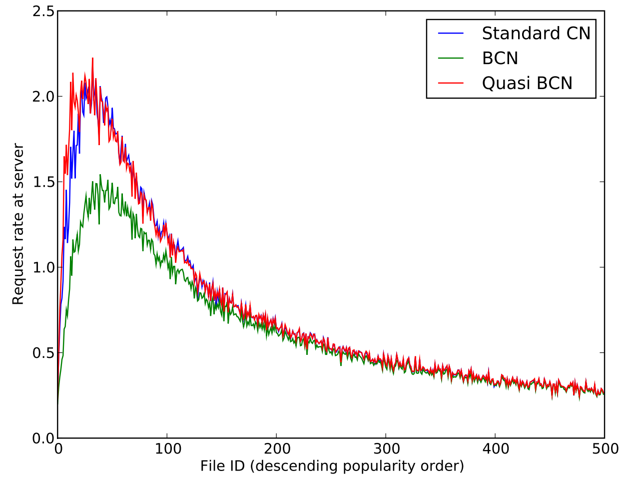


Figure 5.17: Custodian request rate - comparison with quasi-simulation of BCNs. $k = 20$.

5.7 Discussion

In this chapter we presented *Breadcrumbs* — a method for efficient best-effort content search within a cache network. We demonstrated its utility for a specific instance (BECONS) via extensive experimentation, and proved several useful properties for this system. Our results indicate that much can be achieved with systems of implicit coordination, and that these can at times match the performance of more stateful systems using explicit coordination.

The *Breadcrumbs* architecture we presented here is quite flexible, and offers many variations to explore. We briefly survey a few notable examples for such directions, and leave detailed analysis of their properties to future work.

Dynamic Networks and Partial Deployments. The breadcrumb entry as defined above is suitable for networks with topologies that change on a long time-scale. For these, we can assume that the next and previous hops of a node remain the same over the period of time that a specific breadcrumb trail is used. *Breadcrumbs* can be easily extended to other systems where this assumptions does not hold. By substituting the next and previous hops with the *source* and *destination* of the content download, *Breadcrumbs* can be used as well in networks where topologies change more frequently, such as mobile networks. This approach would also allow support for networks where only a sub-set of nodes supports the *Breadcrumbs* protocol. In both cases, having the first and last node in the path can help guide requests even when passing through nodes that have no breadcrumb entries.

Persistent following of breadcrumbs. In our work here, we allowed a request to follow a breadcrumb path only as long as no dead-end was reached and no cycle detected. Once this occurs, the request is routed to the custodian, checking caches along the way but ignoring $\mathcal{R}^{bc}$. Alternative policies might allow for $\mathcal{R}^{bc}$ tables to still be consulted along the way after such an occurrence, once the risk of repeating the cycle or reaching the same dead-end has been avoided.

Adaptive Thresholds. BECONS relies on a small amount of state-exchange to determine the threshold values for timing out breadcrumb entries. It is worth considering, though, implementations where these thresholds are determined at each cache dynamically. For example, by observing the miss streams of neighboring caches, v might be able to determine the rate of content eviction from these neighbors and use this information in determining the threshold after which content is not likely to be at each neighbor. We have developed an outline for such a method with Random Replacement caches, and hope to address this in future work.

In addition to this adaptation, one might want to bound the cost of following a breadcrumb trail in advance. This is possible, to take one example, when we assume the distance to the custodian is known from every node in the network. Assume we wish to bound the increase in search length, compared to routing to the custodian along the shortest path, by a performance parameter $\alpha \geq 1$. Denote the distance from each node v_i to the custodian as $d(i)$, then the goal is to bound the search path at $\alpha d(i)$. This can be done by allowing a request arriving exogenously at v_i to follow a breadcrumb trail for k hops that end at node v_h if $\alpha d(i) \geq k+d(h)$. The analysis of this approach is left for future work, and is brought here only to demonstrate the wide range of options available for using *Breadcrumbs*.

CHAPTER 6

SUMMARY AND FUTURE DIRECTIONS

In this dissertation, we examined the emerging architecture of cache networks. We considered this architecture from multiple perspectives, touching upon both modeling and management of these new systems. The tools we developed here, especially a-NET and *Breadcrumbs*, are easily extendable and can accommodate many variations, which can be tailored to the interests of the researcher and the additional tools at his disposal (e.g., an SCA algorithm).

In summary, made the following contributions to the study of cache networks in this dissertation:

1. We developed an *approximation algorithm* for cache network performance, called a-NET, that leverages SCA algorithms to compute an approximation for an entire network. a-NET can deal with any network topology, and heterogenous networks where caches use different replacement policies.

2. We conducted an analysis of performance-affecting factors on the approximation error of a-NET using a specific SCA algorithm for LRU, and demonstrated the significance of dependencies within the cache miss streams on approximation precision when using this SCA algorithm.

3. We developed a *network calculus* for bounding request flows passing through LRU caches, demonstrating that these bounds are tight in theory, and experimentally explored when the bounds are tight in practice.

4. We considered factors that impact the steady-state behavior of a cache network, specifically showing the possible impact of the initial cache state on long-term behavior. We also proved that for many cache networks, and specifically for a class of replacement policies, the initial state does not influence the steady-state distribution.

5. We described *Breadcrumbs*, a best-effort content search policy, in which each cache routes requests dynamically, based solely on local information. *Breadcrumbs* fosters an implicit inter-cache coordination of routing, without involving any (or only a negligible amount of) inter-cache control overhead.

6. For a certain version of *Breadcrumbs*, called BECONS, we proved the properties of trail stability and trail obsoleteness detection, and the emergence of a border node along the downstream trail. We investigated the performance on *Breadcrumbs* via simulation, showing that in many cases, *Breadcrumbs* outperforms more stateful (and more complex) approaches.

7. We presented an analysis of causal relationships within the network, specifically between cache state and request routing tables. From this analysis, we devised experiments to demonstrate the impact that *Breadcrumbs*-based search has on custodian load reduction.

The work presented here constitutes one of the first to address directly the challenges presented by cache networks of *arbitrary* topologies. While ICN architectures clearly involve such systems on a large scale, the research community has only just begun to grapple with the modeling and management complications for these systems. We hope that, beyond the direct contributions present here, this work can raise awareness of the need for new methods for designing, modeling and analyzing such systems.

In the spirit of directing such future research, we would like to end by pointing out two recurring phenomenon that we observed over the course of our work. The first relates to the impact of *cross flows* in the network on *load balancing* between direct neighbors. The fact that links experience requests for content flowing in both directions on the link is a feature that distinguishes arbitrary cache networks from their classic hierarchical counterparts. In our work, we have observed that these cross flows also generate an implicit form of load balancing between them. This is evidenced in the following behavior: (a) cache misses from A to B cause B to download files A requested; (b) B evicts files from its cache, resulting in more cache misses at B; (c) some of these misses are forwarded to A, where the same process takes place. The result of this back-and-forth is that neighboring caches store different files. While this behavior has been observed in the course of our work, we have yet to discover its exact impact and patterns in large networks.

The second insight we share here is regarding the manner in which cache misses should be considered. In single-cache systems, a cache miss is generally considered a negative event; the goal of optimal caching is to reduce these to a minimum. However, as we have observed in a-NET, when a cache is large, the next hop cache of similar size will have much lower hit probabilities. In other words, *local* benefit at one cache can result in negatively impacting the performance at future hops, perhaps impacting *global* behavior to the worse. The reverse was observed with *Breadcrumbs*, where an increase in cache misses can result in an overall reduction in custodian load and download distance. Our interpretation of this phenomenon is currently that, in a cache network, cache misses are also *information streams* sent from one node to the next. Since each cache determines what to cache based on the arrival stream properties, and the arrival stream includes the miss streams of the neighbors. It can thus be interesting to consider a system that allows cache misses to propagate for purposes of information flow through the network. The exact manner in which this could be done effectively is left for future work.

APPENDIX

APPROXIMATION ALGORITHMS FOR INDIVIDUAL CACHES

In this appendix, we present the SCA approximation algorithms referenced in Chapter 2.

A.1 LRU

The SCA algorithm for LRU we used was developed by Towsley and Dan [14], which we shall denote a-LRU. a-LRU is designed to compute the probability that a file exists in a cache at a random point in time, which for IRM requests is the same as the hit probability, as proven in Lemma 1.

We review briefly some terminology. Let a k-prefix be the top k slots in an LRU cache, which store the k most recently used files. a-LRU leverages a useful property of LRU — that the state of the cache at its k-prefix slots can be computed for a given miss stream without considering the rest of the cache slots. As a result, the content of the cache can be computed incrementally: given the state of the k-prefix for some k, we compute for the $(k+1)$th slot, conditioning on the file not being in the k-prefix. See the full algorithm here (Algorithm 7).

A.2 RND

We next consider an SCA algorithm for Random replacement. We start with reviewing some notation:

Algorithm 7 Approx-LRU($\lambda_1, ..., \lambda_L, L, c$).

For all $1 \le j \le L$, $p_j \leftarrow \frac{\lambda_j}{\sum_k \lambda_k}$
$pdf_1 \leftarrow (p_1, ..., p_L)$
$cdf_1 \leftarrow pdf_1$
for $2 \le i \le c$ **do**
    $cdfRemainder \leftarrow (max\{0, 1 - cdf_{i-1,j}\})_{1 \le j \le L}$ // Probability content not in $i-1$-prefix
    $weights \leftarrow (cdfRemainder_j * p_j)_{1 \le j \le L}$
    $pdf_i \leftarrow \left(\frac{weights_j}{\sum_h weights_h}\right)_{1 \le j \le L}$ // pdf for the contents of slot i
    $cdf_i \leftarrow (cdf_{i-1,j} + pdf_{i,j})_{1 \le j \le L}$ // Probability that content is in i-prefix
end for
RETURN cdf_L // Probability that content is in the cache

- $e_j = Pr(exists_j)$ - probability that f_j can be found in the cache at a random point in time. When the request process is IRM, this is also the probability for a cache hit.

- λ - combined exogenous request rate at the cache.

- $\mu_{ev,j}$ - rate of evictions (= cache misses) from cache given that content f_j is in the cache.

- μ_{ev} - mean rate of evictions at the cache.

- $p_1, \ldots, p_L$ - request distribution at the cache. Assume IRM.

- c - cache size

- $\tau_{j,in}$ - mean time that file i spends in the cache before eviction.

- $\tau_{j,out}$ - mean time that file i spends outside the cache after eviction, before it is cached again.

The solution for a random replacement cache with IRM request probabilities $p_1 \dots p_L$ and a overall request rate of $\boldsymbol{\lambda}$ can be computed from the following equations:

$$\begin{aligned}
\sum_{j=1}^{L} e_j &= c && \text{(A.1)}\\
\forall j \in [n] \quad e_j &= \frac{\tau_{j,in}}{\tau_{j,in} + \tau_{j,out}} && \text{(A.2)}\\
\forall j \in [n] \quad \tau_{j,out} &= 1/\lambda_j := 1/\lambda p_j && \text{(A.3)}\\
\forall j \in [n] \quad \tau_{j,in} &= c/\mu_{ev,j} && \text{(A.4)}\\
\forall j \in [n] \quad \mu_{ev,j} &= \lambda \sum_{k \neq j} Pr(f_k \notin v | f_j \in v) \cdot p_k && \text{(A.5)}
\end{aligned}$$

We briefly explain the meaning of each equation, in order:

1. The existence probabilities sum up the the cache size, as these probabilities can be thought of as the mean cache space taken up by each file.

2. The existence probability for f_j is the fraction of time it spends in the cache.

3. With IRM, the time spent outside the cache is the inverse of the arrival rate.

4. Since which file is evicted is selected uniformly at random from the content in the cache, the rate at which f_j is evicted is $\frac{1}{c} \times \mu_{ev,j}$. The mean time spent in the cache before eviction is the inverse of this value.

5. The eviction rate when f_j is in the cache is the rate of arrivals times the probability of a miss, given that f_j is in the cache. Note that in the last equation, if $j = k$ we get 0 so we do not need to explicitly denote this case.

We approximate this set of equations by substituting $\mu_{ev,i} := \mu_{ev}$, which can be computed with greater simplicity thus:

$$\mu_{ev} = \lambda \sum_{j=1}^{n} p_j(1 - e_j) \tag{A.6}$$

Next, we note that any solution to the first four equations must conform to

$$\sum_j \frac{1}{c + \mu_{ev,j}/(p_j\lambda_j)} = 1 \tag{A.7}$$

and substituting as stated we get

$$\sum_j \frac{1}{c + \mu_{ev}/(p_j\lambda)} = \sum_j \frac{p_j\lambda}{p_j\lambda c + \mu_{ev}} = 1 \tag{A.8}$$

which we solve for μ_{ev}. Once this value is known, solving the set of equations above is straightforward, from which we derive the existence probabilities. Our implementation did this via binary search, using the fact that the value of the sum is monotonically decreasing with μ_{ev}.

BIBLIOGRAPHY

[1] Ahlgren, Bengt, Dannewitz, Christian, Imbrenda, Claudio, Kutscher, Dirk, and Ohlman, Börje. A Survey of Information-Centric Networking (Draft). In *Information-Centric Networking* (Dagstuhl, Germany, 2011), Bengt Ahlgren, Holger Karl, Dirk Kutscher, Börje Ohlman, Sara Oueslati, and Ignacio Solis, Eds., no. 10492 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[2] Ari, Ismail. *Design And Management Of Globally Distributed Network Caches*. PhD thesis, UC Santa Cruz, 2004. http://www.soe.ucsc.edu/ ari/Ari-PhD-Thesis.pdf.

[3] Ari, Ismail, Amer, Ahmed, Gramacy, Robert, Miller, Ethan L., Brandt, Scott A., and Long, Darrell D. E. Acme: Adaptive caching using multiple experts. In *Proceedings in Informatics* (2002), pp. 143–158.

[4] Babenhauserheide, Arne. *GnuFU - Gnututella For Users*, 2004. http://draketo.de/inhalt/krude-ideen/gnufu-en.pdf.

[5] Ballardie, Tony, Francis, Paul, and Crowcroft, Jon. Core based trees (CBT). In *SIGCOMM* (1993), pp. 85–95.

[6] Bhide, A.K., Dan, A., and Dias, D.M. A simple analysis of the lru buffer policy and its relationship to buffer warm-up transient. In *Data Engineering, 1993. Proceedings. Ninth International Conference on* (apr 1993), pp. 125 –133.

[7] Borst, S., Gupta, V., and Walid, A. Distributed caching algorithms for content distribution networks. In *INFOCOM, 2010 Proceedings IEEE* (march 2010), pp. 1 –9.

[8] Busari, Mudashiru, and Williamson, Carey L. Simulation evaluation of a heterogeneous web proxy caching hierarchy. In *MASCOTS* (2001), IEEE Computer Society, pp. 379–388.

[9] Carofiglio, G., Gallo, M., Muscariello, L., and Perino, D. Modeling data transfer in content-centric networking. In *Proceedings of the 23rd International Teletraffic Congress* (2011), ITCP, pp. 111–118.

[10] Chankhunthod, Anawat, Danzig, Peter B., Neerdaels, Chuck, Schwartz, Michael F., and Worrell, Kurt J. A hierarchical internet object cache. In *Proceedings of the USENIX Annual Technical Conference* (Berkeley, Jan. 1996), Usenix Association, pp. 153–164.

[11] Chawathe, Yatin, Ratnasamy, Sylvia, Breslau, Lee, Lanham, Nick, and Shenker, Scott. Making gnutella-like p2p systems scalable. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2003), SIGCOMM '03, ACM, pp. 407–418.

[12] Che, Hao, Wang, Zhijung, and Tung, Ye. Analysis and design of hierarchical web caching systems. In *IEEE INFOCOM* (2001), pp. 1416–1424.

[13] Cruz, R.L. A calculus for network delay. i. network elements in isolation. *Information Theory, IEEE Transactions on 37*, 1 (jan 1991), 114 –131.

[14] Dan, Asit, and Towsley, Donald F. An approximate analysis of the lru and fifo buffer replacement schemes. In *SIGMETRICS* (1990), pp. 143–152.

[15] de Souza e Silva, E., Leao, R. M. M., and Figueiredo, D. R. An integrated modeling environment for computer systems and networks. *Performance Evaluation Review 36*, 4 (2009), 64–69.

[16] Eum, S., Nakauchi, K., Murata, M., Shoji, Y., and Nishinaga, N. Catt: Potential based routing with content caching for icn. *ICN Sigcomm*.

[17] Flajolet, Philippe, Gardy, Danièle, and Thimonier, Loÿs. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Appl. Math. 39*, 3 (1992), 207–229.

[18] Fonseca, Rodrigo, Almeida, Virgilio, Crovella, Mark, and Abrahao, Bruno. On the intrinsic locality properties of web reference streams. In *In Proceedings of the IEEE INFOCOM* (2003).

[19] Fricker, C., Robert, P., and Roberts, J. A versatile and accurate approximation for lru cache performance. *Arxiv preprint arXiv:1202.3974* (2012).

[20] Gallo, M., Kauffmann, B., Muscariello, L., Simonian, A., and Tanguy, C. Performance evaluation of the random replacement policy for networks of caches. *Arxiv preprint arXiv:1202.4880* (2012).

[21] Ghodsi, A., Shenker, S., Koponen, T., Singla, A., Raghavan, B., and Wilcox, J. Information-centric networking: seeing the forest for the trees. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks* (2011), ACM, p. 1.

[22] Gupta, R., Tokekar, S., and Mishra, D.K. A paramount pair of cache replacement algorithms on l1 and l2 using multiple databases with security. In *Emerging Trends in Engineering and Technology (ICETET), 2009 2nd International Conference on* (dec. 2009), pp. 346 –351.

[23] Ioannidis, Stratis, and Marbach, Peter. On the design of hybrid peer-to-peer systems. In *SIGMETRICS* (2008).

[24] Ioannidis, Stratis, and Marbach, Peter. Absence of evidence as evidence of absence: A simple mechanism for scalable p2p search. In *IEEE INFOCOM* (2009).

[25] Jacobson, Van. A new way to look at networking. Internet video, 2007.

[26] Jacobson, Van, Smetters, Diana K., Briggs, Nicholas H., Plass, Michael F., Stewart, Paul, Thornton, James D., and Braynard, Rebecca L. Voccn: voice-over content-centric networks. In *Proceedings of the 2009 workshop on Re-architecting the internet* (New York, NY, USA, 2009), ReArch '09, ACM, pp. 1–6.

[27] Jacobson, Van, Smetters, Diana K., Thornton, James D., Plass, Michael F., Briggs, Nicholas H., and Braynard, Rebecca L. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies* (New York, NY, USA, 2009), CoNEXT '09, ACM, pp. 1–12.

[28] Jelenković, P.R., and Kang, X. Characterizing the miss sequence of the lru cache. *ACM SIGMETRICS Performance Evaluation Review 36*, 2 (2008), 119–121.

[29] Jin, Yingwei, Qu, Wenyu, and Li, Keqiu. A survey of cache/proxy for transparent data replication. In *SKG* (2006), IEEE Computer Society, p. 35.

[30] Kakida, M., Tanigawa, Y., and Tode, H. Breadcrumbs+: Some extensions of naive breadcrumbs for in-network guidance in content centric networks. In *Applications and the Internet (SAINT), 2011 IEEE/IPSJ 11th International Symposium on* (2011), IEEE, pp. 376–381.

[31] Kansal, Aman, Hsu, Jason, Zahedi, Sadaf, and Srivastava, Mani B. Power management in energy harvesting sensor networks. *ACM Trans. Embed. Comput. Syst. 6*, 4 (Sept. 2007).

[32] Katsaros, K., Xylomenos, G., and Polyzos, G.C. A hybrid overlay multicast and caching scheme for information-centric networking. In *INFOCOM IEEE Conference on Computer Communications Workshops , 2010* (march 2010), pp. 1 –6.

[33] Katsaros, K., Xylomenos, G., and Polyzos, G.C. Multicache: An overlay architecture for information-centric networking. *Computer Networks 55*, 4 (2011), 936–947.

[34] Kelly, F.P. Blocking probabilities in large circuit-switched networks. *Advances in Applied Probability* (1986), 473–505.

[35] Kelly, F.P. Loss networks. *The annals of applied probability* (1991), 319–378.

[36] Kemeny, John, and Snell, J. *Finite Markov Chains*. Springer, 1976.

[37] King, W. F. Analysis of paging algorithms. In *IFIP Congress* (1971), pp. 485–490.

[38] Korupolu, Madhukar R., and Dahlin, Michael. Coordinated placement and replacement for large-scale distributed caches. *IEEE Transactions on Knowledge and Data Engineering 14*, 6 (2002), 1317–1329.

[39] Krishnan, P., Raz, Danny, and Shavitt, Yuval. Transparent en-route caching in wans?, 1999.

[40] Krishnan, P., Raz, Danny, and Shavitt, Yuval. The cache location problem. *IEEE/ACM Trans. on Networking 8*, 5 (2000), 568–582.

[41] Kumar, A., Xu, J., and Zegura, E.W. Efficient and scalable query routing for unstructured peer-to-peer networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE* (march 2005), vol. 2, pp. 1162 – 1173 vol. 2.

[42] Kunwadee, Sripanidkulchai. The popularity of gnutella queries and its implications on scalability. *http://www.cs.cmu.edu/ kunwadee/research/p2p/paper.html* (2001).

[43] Laoutaris, N., Smaragdakis, G., Bestavros, A., Matta, I., and Stavrakakis, I. Distributed selfish caching. *Parallel and Distributed Systems, IEEE Transactions on 18*, 10 (oct. 2007), 1361 –1376.

[44] Laoutaris, Nikolaos, Che, Hao, and Stavrakakis, Ioannis. The lcd interconnection of lru caches and its analysis. *Performance Evaluation 63* (2006), 609–634.

[45] Le Boudec, J.-Y., and Tomozei, D.-C. Demand response using service curves. In *Innovative Smart Grid Technologies (ISGT Europe), 2011 2nd IEEE PES International Conference and Exhibition on* (dec. 2011).

[46] Le Boudec, J.Y., and Thiran, P. *Network calculus: a theory of deterministic queuing systems for the internet.* springer-Verlag, 2001.

[47] Levy, Hanoch, and Morris, Robert J. T. Exact analysis of bernoulli superposition of streams into a least recently used cache. *IEEE Trans. Softw. Eng. 21*, 8 (1995), 682–688.

[48] Liu, Y., Guo, Y., and Liang, C. A survey on peer-to-peer video streaming systems. *Peer-to-peer Networking and Applications 1*, 1 (2008), 18–28.

[49] Lv, Qin, Cao, Pei, Cohen, Edith, Li, Kai, and Shenker, Scott. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th international conference on Supercomputing* (New York, NY, USA, 2002), ICS '02, ACM, pp. 84–95.

[50] Panagakis, Antonis, Vaios, Athanasios, and Stavrakakis, Ioannis. Approximate analysis of lru in the case of short term correlations. *Comput. Netw. 52*, 6 (2008), 1142–1152.

[51] Peng, Gang. Cdn: Content distribution network. http://www.scientificcommons.org/21241169.

[52] Pentikousis, Kostas, and Rautio, Teemu. A multiaccess network of information. In *World of Wireless Mobile and Multimedia Networks (WoWMoM), 2010 IEEE International Symposium on a* (june 2010), pp. 1 –9.

[53] Psaras, Ioannis, Clegg, Richard, Landa, Raul, Chai, Wei, and Pavlou, George. Modelling and evaluation of ccn-caching trees. In *NETWORKING 2011*, Jordi Domingo-Pascual, Pietro Manzoni, Sergio Palazzo, Ana Pont, and Caterina Scoglio, Eds., vol. 6640 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2011, pp. 78–91.

[54] Psaras, Ioannis, Clegg, Richard G., Landa, Raul, Chai, Wei K., and Pavlou, George. Modeling and evaluation of ccn-caching trees. In *IFIP Networking* (2011).

[55] Ratnasamy, Sylvia, Francis, Paul, Handley, Mark, Karp, Richard, and Schenker, Scott. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev. 31*, 4 (2001), 161–172.

[56] Raza, M.H., Robertson, B., Phillips, W.J., and Ilow, J. Network calculus based modeling of anomaly detection. In *Performance Evaluation of Computer and Telecommunication Systems (SPECTS), 2010 International Symposium on* (2010), IEEE, pp. 416–421.

[57] Rodriguez, P.R. *Scalable Content Distribution in the Internet*. PhD thesis, Universidad Publica de Navarra, 2000.

[58] Rosensweig, Elisha, and Kurose, Jim. Breadcrumbs: efficient, best-effort content location in cache networks. In *IEEE INFOCOM Mini-Conference* (2009). http://gaia.cs.umass.edu/networks/papers/INFOCOM09-mini_Breadcrumbs.pdf. An extended version can be found in the technical report UM-CS-2009-005 at http://www.cs.umass.edu/publication/docs/2009/UM-CS-2009-005.pdf.

[59] Rosnsweig, Elisha J., Kurose, Jim, and Towsley, Don. Approximate models for general cache networks. In *INFOCOM, 2010 Proceedings IEEE* (march 2010), pp. 1 –9. http://gaia.cs.umass.edu/networks/papers/CacheModels_INFOCOM10.pdf.

[60] Rossi, D., and Rossini, G. Caching performance of content centric networks under multi-path routing (and more). Tech. rep.

[61] Rossini, DRG, and Rossi, D. A dive into the caching performance of content centric networking. Tech. rep., Technical report, Telecom ParisTech, 2011.

[62] Schmitt, J., and Roedig, U. Sensor network calculus–a framework for worst case analysis. *Distributed Computing in Sensor Systems* (2005), 467–467.

[63] Starobinski, David, Karpovsky, Mark, and Zakrevski, Lev A. Application of network calculus to general topologies using turn-prohibition. *IEEE/ACM Trans. Netw. 11* (June 2003), 411–421.

[64] Starobinski, David, and Sidi, Moshe. Stochastically bounded burstiness for communication networks. *IEEE Transactions on Information Theory 46* (1999), 206–212.

[65] Tang, X., and Chanson, S. T. Coordinated en-route web caching. *IEEE Transactions on Computers 51*, 6 (2002), 595 – 607.

[66] Tatsuhiro Tsutsui, Hiroyuki Urabayashi, Miki Yamamoto Elisha J. Rosensweig, and Kurose, Jim. Performance evaluation of partial deployment of breadcrumbs in content oriented networks. In *IEEE ICC Workshop* (2012).

[67] Tewari, S., and Kleinrock, L. Proportional replication in peer-to-peer networks. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings* (april 2006), pp. 1 –12.

[68] Trossen, Dirk, Sarela, Mikko, and Sollins, Karen. Arguments for an information-centric internetworking architecture. *SIGCOMM Comput. Commun. Rev. 40* (April 2010), 26–33.

[69] Vanichpun, S., and Makowski, A.M. Comparing strength of locality of reference-popularity, majorization, and some folk theorems. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies* (2004), vol. 2, IEEE, pp. 838–849.

[70] Wang, K., Ciucu, F., Lin, C., and Low, S.H. A stochastic power network calculus for integrating renewable energy sources into the power grid. *Selected Areas in Communications, IEEE Journal on 30*, 6 (2012), 1037–1048.

[71] Williamson, Carey. On filter effects in web caching hierarchies. *ACM Trans. Internet Technol. 2* (February 2002), 47–77.

[72] Yaron, Opher, and Sidi, Moshe. Generalized processor sharing networks with exponentially bounded burstiness arrivals. In *Journal of High Speed Networks* (1994), pp. 628–634.

[73] Zhou, Y., Chen, Z., and Li, K. Second-level buffer cache management. *Parallel and Distributed Systems, IEEE Transactions on 15*, 6 (june 2004), 505 – 519.

[74] Zhu, Yingwu, Yang, Xiaoyu, and Hu, Yiming. Making search efficient on gnutella-like p2p systems. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International* (april 2005), p. 56a.