# ON THE ANALYSIS AND MANAGEMENT OF CACHE NETWORKS

A Dissertation Outline 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

June 2011

Computer Science

# ON THE ANALYSIS AND MANAGEMENT OF CACHE NETWORKS

A Dissertation Outline 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,*
*who stood by me through it all*

# ACKNOWLEDGMENTS

TOBEDECIDEDLATER

# ABSTRACT

## ON THE ANALYSIS AND MANAGEMENT OF CACHE NETWORKS

JUNE 2011

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, a number of researchers have begun rethinking the fundamental communication model underlying the Internet. In addition to the traditional host-to-host communication model that has endured for more than four decades, many researchers have begun to focus on Content Networking - a networking model in which content is addressable and host-to-content (rather than host-to-host) interaction is the norm. With content accessibility in the spotlight, caches are poised to become central components of any Content Networking architecture. While many traditional systems employ caching as a means to improve performance, the design, analysis and management of widely deployed, tightly-connected, heterogenous Internet-scale *networks* of caches, termed here *Cache Networks*, is a relatively uncharted field.

In this thesis we will develop modeling techniques for analyzing tightly-interconnected cache networks, and design efficient policies for locating cached content. The first part

of our work addresses the modeling and analysis challenge, where we consider the behavior of these systems from a theoretical standpoint. The second part of our work considers the challenge of managing these systems, both in finding content and load balancing content requests.

# TABLE OF CONTENTS

# INTRODUCTION

Over the past few years, a number of researchers have begun rethinking the fundamental communication model underlying the Internet. In addition to the traditional host-to-host communication model that has endured for more than four decades, many researchers have begun to focus on Content Networking - a networking model in which content is addressable and host-to-content (rather than host-to-host) interaction is the norm [1, 22, 23, 26, 43, 47, 57]. This transition is a culmination of many changes in end-user demands (such as the astounding growth of P2P systems), a focus on content distribution, and the increasing mobility of network elements.

With content accessibility in the spotlight, caches are poised to become central components of any next generation Content Networking architecture. Storing large chunks of popular content at multiple network locations can greatly reduce server load, network congestion and content access delays experienced by end users. While many traditional systems employ caching as a means to improve performance (see [8, 9, 36, 42, 46] for a small sample of such discussions), the design, analysis and management of widely-deployed, tightly-connected, heterogenous Internet-scale *networks* of caches, termed here *Cache Networks* and abbreviated "CNs", is a relatively uncharted field. In this thesis we will develop modeling techniques for analyzing tightly-connected cache networks, and design efficient policies for locating and serving cached content. In the five chapters that follow we discuss results to date, as well as future work to be done as part of this dissertation.

The first part of our work addresses modeling and analysis challenges. The widespread deployment of caches in the network introduces considerable changes to the endogenous flows of content requests in the system [17], by applying a *selective*

*filter* on these these flows as cache hits occur [9, 58]. Existing analytical work in this field does not deal with large networks of arbitrary topologies and heterogenous cache management policies. Our research here begins to build an "analytical toolkit" for analyzing the performance of cache networks. 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 The first three chapters of this work present our contributions in this area.

In Chapter 1 we approximately model the performance of a CN; the performance metrics of interest are cache hit probabilities and the load experienced at each cache and at each "custodian" responsible for storing a permanent copy of a piece of content. Recognizing that precise models for CNs are computationally intractable, we develop an algorithm named *a-NET* that approximates CN performance. *a-NET* uses known algorithms that approximate the behavior of *stand-alone* caches to iteratively compute an approximate solution for a *network* of such caches. In addition to presenting this algorithm and analyzing its performance via simulation, we also develop a method for determining parameters that affect its accuracy. We apply these methods to conclude that dependencies within the miss streams are the main cause of approximation errors. This work has appeared in [50]. As future work, we will explore the run-time efficiency of *a-NET* as a function of system parameters such as the size of the network.

In addition to approximating the performance of CNs, *a-NET* has additional uses which Chapter 1 explores. Using the model proposed by Cruz [12] in his network calculus, we develop an algorithm that computes a worst-case bound on the performance of the deterministic LRU and FIFO replacement policies. Using this algorithm within *a-NET*, we plan to compute worst-case bounds on the performance of CNs that use these replacement policies, and compare these bounds to performance in practice. Depending on the relationship between performance in practice and these worst-case

bounds, we will draw conclusions regarding the selection of suitable cache replacement policies for CNs.

In Chapter 2 we consider the impact of the initial state of a CN on its long-term, steady-state behavior. We present examples that demonstrate how this initial state can, in some cases, impact the steady-state of the system. We state and prove several sufficient conditions for a CN that ensure the system has a single steady-state solution[1] for a given user demand, independent of the initial state. Our results show, among other things, that replacement policies can be grouped into equivalence classes, such that the ergodicity of a system in which caches use one policy from this class implies the ergodicity of the same system in which caches use any policy from within this class (or mixed use of different class policies at different CN nodes). These results can be found in [49], which is currently in submission.

We conclude the analytical part of this thesis proposal in Chapter 3, where we consider a cache inference question. In CNs cache misses are routed within the network, and information about content availability might be used to improve these routing decisions. In this chapter we present a method for estimating the eviction rate at a cache based on (parts of) its miss stream. Knowing the eviction rate of network caches can help instruct where to route requests, as we discuss in Chapter 4. To date, we have formulated an inference algorithm, and we will implement it and study its utility and timeliness via simulation.

The second part of this proposal discusses request routing, i.e., how to search for content in a CN. In a CN, different content search policies will result in different search overhead and load distribution (i.e., cache hits) over the network caches and custodians. Consider, for example, three "standard" approaches for locating content: Exhaustive search of the entire network (as in earlier versions of Gnutella P2P net-

[1]The "solution" of the system is the steady-state distribution of content in the caches of the network.

works) will always turn up a cached copy (if one exists), but with potentially high delay experienced at the requesting end-user and considerable communication overhead. Alternately, forwarding requests directly to publicly-known content custodians, and returning requested content from caches only if they happen to be on this route, could result in missed opportunities (e.g., content copies just off this route are not accessed) and an increased load at the custodian. Collaborative protocols in which caches coordinate storage and request routing, can have high computational complexity and communication overhead [42]. 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.

Our contributions to date in this area are presented in in Chapter 4, where we describe Breadcrumbs - a best-effort content search policy. Breadcrumbs uses short-term, soft-state routing entries, termed here "breadcrumbs", at each cache-router to direct future requests. When content $f$ passes through a cache-router element $v$, $v$ registers the source and destination of $f$. If a future request for $f$ arrives at $v$ and $f$ is not cached at $v$, $v$ uses a corresponding breadcrumb entry to route the request in the direction where content is most likely to be found. For a specific implementation of Breadcrumbs, called BECONS, we formally establish properties of this system in uncongested networks, and via simulation compare its performance to that of more stateful systems. These results can be found in [48]. We plan on extending this work in several ways. Further simulation is needed to validate the preliminary results we present in [48]; we will also employ additional metrics to better evaluate the search performance of Breadcrumbs; and consider additional implementations of Breadcrumbs besides BECONS.

The last problem we will consider in this dissertation is that of distributing load among the caches in the network, focusing on the *utility of timely information*; our work here is just the beginning. In Chapter 5, we examine a system in which all

nodes have perfect information about where content is stored at any given time $t$, but only partial information regarding the load at each node. Load here is defined as the number of downloads taking place at a content source, whether custodian or cache. Assuming each cache is aware of the *average* load at all nodes over a recent period of time and forwards a request to the node with minimal load, we explore how well the load is balanced as a function of the window over which the known mean load is taken, as well as the download path taken by the file.

The remainder of this thesis proposal is structured as follows. Chapter 1 presents our algorithm for approximating the behavior of CNs. We also introduce here the model and notation we use throughout this proposal, as well as much of the related work on CNs. Chapter 2 discusses the impact (or lack thereof) of the initial state of a CN on its steady-state, and Chapter 3 considers how to infer eviction rates of a neighboring cache based on its miss stream. Chapter 4 presents *Breadcrumbs*, our best-effort content-search policy, and we conclude with Chapter 5 with a discussion of the importance of timely information when attempting to balance load across the system.

# CHAPTER 1

# APPROXIMATE MODELS FOR CACHE NETWORKS

## 1.1 Introduction

Caches are an integral part of many computing systems, and consequently their policies and resulting performance has been the focus of much research. Earlier works considered caches in isolation; more recent research has considered hierarchical (i.e., tree-like) cache network architectures [7, 11, 18, 36]. As a rule, caching policies are notoriously difficult to analyze even when their operation is quite simple. Even for 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 [13, 28]. 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 and modeling a limited range of topologies and policies. Consequently, there is a need for tools that can predict the behavior of large-scale CNs comprised of caches arranged in an arbitrary topology.

In this chapter we present a novel **m**ulti-**c**ache **a**pproximation (MCA) algorithm, denoted *a-NET*, that can be applied to cache networks of any topology or scale. The approach taken by *a-NET* is to compute a **s**ingle-**c**ache **a**pproximation (SCA) for each individual cache in the network, and combine these approximations into a solution for the entire system. To account for cache interactions *a-NET* repeats this process, in each round feeding the output of the previous round as input to the next, until the solutions converge to a fixed point. We demonstrate the performance of *a-NET* for multiple topologies, and identify key parameters that affect its performance.

*a-NET* can also be used to help understand how individual cache performance is affected within a network of such caches. Previous work on cache hierarchies has shown that simply chaining two LRU caches together, with cache A forwarding its miss stream to cache B, yields less than double the benefits [58]. In order to understand to what degree this holds for cache networks, we will compare the performance of LRU caches in a network to their theoretical worst-case performance in the same setting. To this end, we develop a bounding SCA algorithm for the LRU replacement policy, using the well-known $(\rho, \sigma)$ notation of Network Calculus, and use *a-NET* to compute worst case bounds an performance. We plan to compare these bounds to actual behavior, and based on our results discuss the benefits and limitations of LRU as a replacement policy in such networks.

The structure of this chapter is as follows. Section 1.2 describes the network of interconnected caches that will be adopted throughout this thesis, and Section 1.3 presents related work for the first two chapters. Section 1.4 then presents *a-NET* and discusses its performance when using the LRU SCA proposed in [13]. Our analysis here indicates that the non-IRM nature of the endogenous request streams has significant impact on the performance of *a-NET*. To further understand the impact of non-IRM request streams on performance, in Section 1.5 we construct an algorithm that deterministically bounds the misses of a stand-alone LRU cache. We propose to use it with *a-NET* to compute worst-case bounds on LRU CNs, and compare these bounds with simulated behavior. We conclude with section 1.6 where we summarize our work and discuss remaining work for this chapter.

## 1.2 Model and Notation

### 1.2.1 System Components and Operation

**Basic model.** We begin by describing the system of interest in this paper - cache networks (CNs). A summary of the notation that follows is presented in Table 1.1. Our model follows common practices (see for example [18, 20, 36, 55]).

Let $G = \langle V, E \rangle$ be a finite network comprised of nodes $V = \{v_1, ...v_m\}$ 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 that is forwarded by the router can also be stored locally. Edges in this network indicate neighbor relationships among the cache-routers, such that cache-router $v$ can forward an unsatisfied request - a cache *miss* - only via its neighbors. For the sake of readability, the terms "caches" and "nodes" will be used interchangeably here to indicate these cache-router entities. In related work, these are sometimes referred to as *Transparent En-Route Caches,* TERC for short [24, 30, 31].

Let $F = \{f_1, ..., f_n\}$ be the set of unique items of content, termed here *files,* that can be requested in the network. The *state* of a node $v$ at time $t$ is the set (or sequence[1]) of elements stored at $v$ at $t$, and the state of the network is the state of all its nodes. $f_i \in v_j$ denotes that $f_i$ is stored at the cache of $v_j$, $|v_j|$ is the size of the $j$th cache. By default, we will assume that all files in the system are of identical size, as are all caches. As a result, the size of a cache can be expressed in terms of the number of files it can store, and each cache is of size $|v|$. It important to note that these two assumptions are not required for most of the results presented here, and are used mostly for clarity of exposition. Whenever possible, we shall discuss how to extend our results to cases with variable cache and file sizes.

[1]Random replacement policies ignore the position of a cached file, while LRU and FIFO are examples of replacement policies where the order is significant.

**Table 1.1.** Table of System Notation

| Notation | Meaning |
|---|---|
| $v$ | A cache-router entity |
| $\|v\|$ | The number of files a cache can store |
| $f$ | A content entity (file) |
| $m, n$ | The number of nodes and files, respectively |
| $cust(i) \subseteq V$ | List of custodians for $f_i$ |
| $(v_j, f_i)$ | A request for $f_i$ at cache $v_j$ |
| $e_{ij}$ | Probability that $f_i \in v_j$ |
| $\mathcal{R}$ | Request routing matrix |

**Permanent Copies.** In addition to the short-term storage provided by caches, we assume content is also stored permanently at one or more content *custodians* in the network, such as public content servers [20, 39, 55]. Each custodian connects to the network at a specific location - a cache-router element - and so we will denote them as a set $C \subseteq V$. Note that for each $v \in C$, the storage required for maintaining these permanent copies is not included in the specified cache size. We use $v \in cust(i)$ to denote that (a custodian connected to) node $v$ has a permanent copy of $f_i$ . For exposition purposes here we assume that for all $1 \leq i \leq n$, $|cust(i)| = 1$, though our results hold as long as $|cust(i)| \geq 1$.

**Request Routing.** Requests for content can arrive at a cache in two ways: directly via exogenous request from a user directly attached to the cache-router, or as a cache miss at a neighboring cache. Initially, requests originate exogenously from users seeking content, who send requests into the network. Each request is then forwarded endogenously within the network until it is satisfied. When a cache cannot satisfy a request, the request is forwarded along a *path* in the network in search of a copy. In chapters 1-3 we assume there exists a static routing matrix $\mathcal{R} \in V^{m \times n}$, such that $\mathcal{R}(v_j, f_i)$ indicates the next-hop to forward unresolved requests, termed *cache misses*, for $f_i$ at $v_j$. To ensure all requests for $f_i$ are eventually satisfied, we assume the path for every request ends at a node $v \in cust(i)$. A common example

for a set of static paths is that of *shortest path* routing (used, for example, in [18]), in which a request for $f_i$ is routed along the shortest path to the closest node in $cust(i)$. Dynamic routing matrices that change over time are addressed in the second part of this thesis proposal (Chapters 4-5).

**Request Handling.** A request for $f_i$ arriving at node $v_j$ is denoted $(v_j, f_i)$. For all $f_i \in F$, $v_j \in V$, $\lambda_{ij}$ is the exogenous rate of $(v_j, f_i)$, where by "rate" we mean the *average* number of requests per time unit. When a request for $f_i$ arrives at $v_j$, one of the following occurs:

- If $f_i \in v_j$, 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 that the request traversed.

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

Unless otherwise stated, when a file $f_i$ is being downloaded and it passes through a node $v_j$ with a full cache, one of the files in the cache will be *evicted* to make room for $f_i$ (assuming $f_i \notin v_j$). A *replacement policy* at each cache determines which file is evicted.

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

$$r_{ij} = \lambda_{ij} + \sum_{h:\mathcal{R}(v_h,f_i)=v_j} m_{ih} \tag{1.1}$$

### 1.2.2 Model Assumptions

In this work we adopt the following common modeling assumptions.

- **Arrival Process.** We model the arrival process of exogenous requests according to the Independent Reference Model (IRM) (e.g., [13, 20] and others), which

states that the probability that the next *exogenous* request at a cache is independent of the earlier requests. Formally, let $X_k \in F$ be the $k$th file exogenously requested at some cache $v$, with IRM we have

$$Pr(X_k = f_i | X_1, \ldots X_{k-1}) = Pr(X_k = f_i) \tag{1.2}$$

- **Download Delay.** We will assume by default that the time between a cache miss and the time when the file arrives at the cache is much smaller than the inter-request timescale and can therefore be ignored [11, 13, 20, 21]. Thus, once a cache miss occurs, the file is assumed to be instantaneously downloaded into the cache, and at caches along the download path as well. We refer to this assumption as the *Zero Download Delay* (ZDD) property. In several places in this work assuming ZDD is not essential other than making the exposition clearer, and we point this out where relevant.

## 1.3 Related Work

This chapter and the next both consider different aspects of cache network modeling, and thus have much related past research in common. This section outlines research related to the topics addressed in both chapters.

### 1.3.1 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, 60]. 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 $|v|$ and $n$ grow [28]. As a result, fast approximations have been proposed for these caches [13].

The description of a CN includes, among other things, the policies each cache uses. The *a-NET* algorithm we present in this chapter assumes that there are algorithms for approximating performance of stand-alone caches using these policies. Such an algorithm is termed an SCA (Single Cache Approximation) algorithm. In this work we use an IRM SCA algorithm developed by Dan and Towsley [13] for LRU caches, which we denote here *a-LRU*. *a-LRU* predicts for each cache *mynode* and $f \in F$ $Pr(f \in v)$, and under IRM this is equal to the file hit probabilities [50].

In addition to *a-LRU* there are other algorithms that compute an LRU SCA of the hit probability at an isolated cache with IRM assumptions. For example, Flajolet et. al. [16] present an integral solution for the cache approximation problem, which can be solved numerically to produce the hit ratio. However, there is no straight-forward manner by which to observe the behavior of each file with this approach. Che et. al. [11] use a *mean field approximation* to approximate the behavior of individual caches. Their approximation assumes a constant time for a file to spend in a cache before it is evicted if not requested, and 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 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 2 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.

### 1.3.2 Results for networked caches

We next consider models for *networks* of caches. There has been substantial work regarding cache *hierarchies* or *trees* [7, 9, 11, 18, 19, 36, 44, 45, 60]. 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. These models rely heavily upon the special properties of this topology. First, some papers make use of the symmetry of tree structures; 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. This feature (combined with ZDD assumptions) allows for analysis to be done from the bottom up. In contrast to these works, we focus here on networks of arbitrary topologies, where content and requests can flow in both directions on network links.

Che et. al. [11] model a 2-tier cache hierarchy using the aforementioned *mean field approximation*. In subsequent work [36] 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. Also, the mean field approximation they use becomes difficult to apply to hierarchies with more levels, and cannot be easily extended to arbitrary topologies.

### 1.3.3 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* [20, 21, 39, 55], 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 it. The system is termed "hybrid" to indicate that it assumes there is always a publisher

entity available in the system, to which clients can turn to if content is not available in the P2P "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 [55] 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 [20] 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 occurs over the same topology the search used, with content populating the caches enroute as content is returned to the requesting node. While there are P2P systems where content populates the peers along the download path (e.g., Gnutella [33]), this is not required for P2P to function, and little in terms of 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] In fact, some work on hybrid P2P networks might be even more applicable to CNs than to P2P systems. For example, in the previously-cited papers 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.

### 1.3.4 Modeling Assumptions

All the components of the model we adopt and describe here are used elsewhere in the literature. These include ZDD [11, 20, 21, 36], IRM requests [11, 13, 20, 21, 36, 55], identical exogenous access patterns at all users [7, 21, 55] and storing content at nodes that did not request it [39]. We now explore in more detail some of our main modeling assumptions - IRM exogenous arrivals, cache coordination and homogeneous cache policies.

#### 1.3.4.1 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. Since CNs exhibit the opposite effect, with content requested recently *less* likely to be requested next, [41] is less useful here. An approach that deviates sharply from IRM is that of Stack Depth Distribution (SDD) [38]. 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, and so is not used here.

#### 1.3.4.2 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 [29, 54, 32], while at the other end of the spectrum some have considered systems where caches are oblivious to the state of other caches [11, **?**, 48, 50]. In this work we consider this second type of architecture, though the tools presented here could be used for some coordinated systems as well.

#### 1.3.4.3 System Architecture - Homogeneous vs. Heterogeneous systems

The results presented in the next two chapters apply equally to homogenous and heterogenous cache networks. With *homogenous* CNs 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 $v_j$ uses Random replacement). We conjecture that this second class of networks is likely to occur especially when network management is not in the hand of a single controlling authority, and indeed might improve performance in some cases [34]. 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 with regards to cache hierarchies, where some have suggested that upper-level caches should employ different replacement policies than those in lower levels [9, 19, 60]. Extending these ideas to networks with arbitrary topologies is non-trivial and is beyond the scope of this work.

## 1.4 *a-NET*

### 1.4.1 *a-NET*- Design

We now outline the *a-NET* algorithm. As noted above, *a-NET* is an iterative algorithm that computes the hit probability and the request load at each node in the network. In each iteration it estimates the performance of each individual network cache, and uses this estimate as input to the next iteration. We assume that the performance of a single cache in isolation (i.e., a stand-alone cache) can be approximated via some existing single cache approximation (SCA) algorithm. In what follows we describe the system for LRU caches, though it applies equally for any replacement policy. Also, in our experiments with LRU caches we used an SCA algorithm proposed by Towsley and Dan [13], termed here *a-LRU*.

For a single cache $v_j$, let $LRU^*$ be an SCA algorithm for LRU. With request rates $\vec{r}_j = (r_{1j}, ..., r_{nj})$ arriving at $v_j$, the algorithm computes the miss stream of the cache, i.e., $LRU^*(\vec{r}_j) := (\hat{m_{1j}}, ..., \hat{m_{nj}})$, with $\hat{m_{ij}}$ the estimated miss rate for $f_i$ at $v_j$. From this we can compute the *hit probability* for each file as $1 - \hat{m_{ij}}/r_{ij}$. For the networked setting, Eq. 1.1 expresses the arrival rates at a node as

$$r_{ij} = \lambda_{ij} + \sum_{h:\mathcal{R}(v_h, f_i)=v_j} m_{ih}$$

*a-NET* approximates a given system by decomposition, repeatedly approximating the performance of individual nodes for their input request load, and based on where cache misses are forwarded recomputes the input request load at all nodes. Formally, for each $i, j$, *a-NET* initializes $r_{ij} = \lambda_{ij}$ and computes $LRU^*$ for this input. Next, for each iteration we compute

$$\forall i \in [n], j \in [m], \quad \hat{r_{ij}} = \lambda_{ij} + \sum_{h:\mathcal{R}(v_h, f_i)=v_j} \hat{m_{ih}} \tag{1.3}$$

$$\forall j \in [m], \quad (\hat{m_{1j}}, ..., \hat{m_{nj}}) = LRU^*(\hat{r_{1j}}, ..., \hat{r_{nj}}) \tag{1.4}$$

*a-NET* repeats this process, until the entire network converges to a fixed point for $\hat{r_{ij}}$[3]. In principle, *a-NET* can compute in such a manner an MCA for any topology, as well as heterogenous systems where each cache uses different management policies.

We simulated the behavior of *a-NET* on many different topologies, and present here some sample results for its performance on a 10-by-10 torus cache network to provide the flavor of our results. The exogenous request distribution and rate is the same for all caches, and requests are distributed according to Zipf distribution

[3]In Chapter 2 we will leverage Theorem 1 to prove that when the system uses only non-protective replacement policies, as defined there, *a-NET* has a single fixed point to which it converges.

with parameter 1 (i.e., the $i$-th most popular file arrives with probability $\frac{1/i}{\sum_{j=1}^{N} 1/j}$), as has been observed in real web access traces [8]. There are $|F| = 500$ files in the system, cache sizes are $|v| = 50$, and there are 4 custodians amongst which content is partitioned. Figure 1.1 demonstrates the ratio between the predicted hit probability at each cache and the actual hit probability, with the mean ratio showing *a-NET* overestimating the hit probability by 8%. Further support for the good agreement between *a-NET* analysis and simulation can be found in Figure 1.2, which presents the standard-deviation of the arrival process at each cache. The similarity between the approximated and actual versions is a preliminary indication that even when hit probabilities are overestimated, the algorithm accurately models system behavior in terms of arrival distribution per node. We will explore this last point with additional distribution-comparing metrics in this thesis.

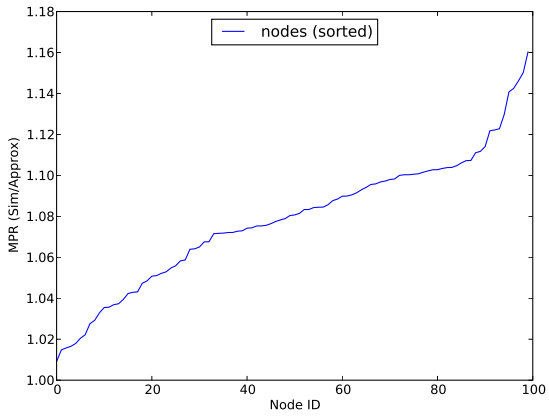


**Figure 1.1.** The ratio of hit probabilities at each cache for 10-by-10 torus, with file requests distributed using Zipf distribution, $|F| = 500$, and all caches of size $|v| = 50$. Nodes are sorted by increasing ratio.

In addition to presenting this algorithm in detail, this chapter will include the following contributions (reported in [50]):

**Factor Analysis of Approximation Errors** *a-NET* is an approximation only, and for the SCA we used appears to consistently over-estimate the performance of

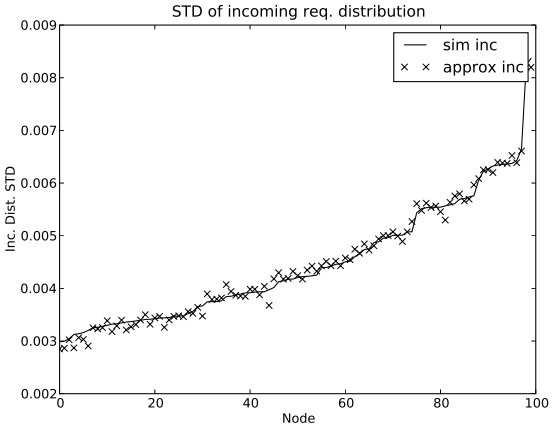


**Figure 1.2.** Standard deviation of incoming request distribution at each node. The solid line (sim inc) is for STD in the simulation, while the x's (approx inc) are for the approximation. Nodes are sorted according to increasing STD in the simulation.

CNs in terms of cache hit probability. We perform a factor analysis of this approximation error, and determine that the cause of this error is the fact that the endogenous request streams are non-IRM.

**Implication of Factor Analysis** As a result of this discovery, we consider the question of where *a-NET* is more likely to perform well. We determine that networks with many high-degree nodes (e.g., trees with large branching factors) are better approximated.

**Analysis of non-IRM traffic** The non-IRM nature of endogenous traffic is also studied here. We conjecture, based on related work, that the large inter-arrival distance between requests for a file in the miss stream is the cause of the over-estimating nature of *a-NET*. While this property of the miss stream is intuitive, and has been demonstrated empirically for many scenarios, it has had little analytical justification prior to this work. We use a Markov model for request arrivals to demonstrate and study the effects of IRM-violation on the miss stream.

**Extensive Experimentation** We conduct extensive experimentation to explore the performance of *a-NET* under a wide range of conditions.

## 1.5 Towards a Network Calculus for LRU Cache Networks

A well-known tool for analyzing flow bounds in packet-switching networks is that of Network Calculus. First presented by Cruz [12] for deterministic bounds, it has since been expanded upon to include a wider variety of bounding methods and address many topologies [52, 53, 59]. Due to the success of this approach to bound the performance of queueing networks with complex inter-nodal flows, we have taken initial steps towards developing a network calculus suitable for analyzing cache networks, specifically those using LRU as a replacement policy.

Let us consider the well-known $(\sigma, \rho)$ deterministic bounding representation of a stream, where $\rho$ indicates the average arrival rate per time unit and $\sigma$ indicates the burstiness component of the stream. For a stream with traffic rate $R(t)$ and for any interval $[t_1, t_2]$, the traffic during this interval conforms to

$$\int_{t_1}^{t_2} R(t)dt \leq \rho(t_1 - t_2) + \sigma \tag{1.5}$$

The key result in [12] for queueing networks is that if each input flow $i$ (corresponding to a source-destination node pair in a queueing network) has a $(\sigma_{i,in}, \rho_{i,in})$ characterization, then each output flow has a $(\sigma_{i,out}, \rho_{i,out})$ characterization that can be computed as a function of the input characterizations at that node, $\{(\sigma_{j,in}, \rho_{j,in}), 1 \leq j \leq N\}$ for $N$ input flows. 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 cache networks, the input stream to a cache consists of a set of incoming streams $\{R_i(t)\}_{f_i \in F}$, each stream consisting of requests for $f_i$. Our goal here is to compute, given the exogenous incoming streams at each cache, $(\sigma, \rho)$ worst-case bounds

on the arrival and miss rates at each node as well as the hit probability at these nodes. Our research in this area will be as follows:

**Bounding Algorithm for a single LRU/FIFO cache.** We develop an algorithm for bounding the worst-case performance of a *single* LRU cache (Note that LRU and FIFO worst-case are the same). Given a set of incoming streams $\{R_i(t)\}_{f_i \in F}$, each stream in a $(\sigma, \rho)$ characterization, the algorithm computes $(\sigma, \rho)$ bounds on the miss stream. An outline of this algorithm is presented below.

**Bounds on Feed-Forward Networks.** Just as with traditional network calculus, we will use this bounding algorithm to "push" bounds on flows through the network. Once calculated, we will compare these results to actual cache performance to conclude how well LRU performs in these settings, as opposed to single-cache performance.

**Bounds on networks with arbitrary topology.** We will extend our evaluation to non-feed-forward networks, a known challenge for standard network calculus [52]. We will do this using *a-NET*, using our bounding algorithm in place of the SCA algorithm component, and once again compare these bounds to actual performance.

In what remains of this chapter we present an outline of our bounding algorithm for LRU. We make use of the following Lemma:

**Lemma 1** *Given a finite number of requests for each $f_i \in F$, the highest number of misses for $f_i$ when using an LRU cache occurs by maximizing the pairs of sequential $f_i$ requests that are separated by exactly $|v|$ requests for different files.*

Based on this Lemma, we compute the worst case performance of LRU w.r.t. $f_i$ by solving a bin-packing problem. The *items* in this comparison are requests for

file, with the items filling the $j$th bin including the $j$th request for $f_i$ as well as the requests that arrived between the $(j-1)$th and $j$th request for $f_i$. Since, given a piece of content to be evicted we need to have requests for $|v|$ different pieces of content between two requests for the content to be evicted, we get the following problem:

**Input:** An infinite number of empty bins, all of the same capacity $|v|+1$. We also have a set of pairwise-disjoint sets $\mathcal{A} = \{A_1, ..., A_n\}$, such that $|A_j| = a_j$.

**Output:** The maximum number of *filled* bins. Let $\mathcal{B} = \{b_1, ..., b_k\}$ be the filled bins, then we require that for each bin $|b_h \cap A_i| = 1$, and for all $j \neq i$ $|b_h \cap A_j| \geq 1$.

This algorithm returns the maximum number of misses that $f_i$ can experience as caused by the rest of the request stream, with each bin reflecting a sequence of requests that will evict $f_i$ prior to the next request for it, generating a cache miss. When we assign $a_i := \rho_i$, we get the total number of misses w.r.t. the average request rate, and when $a_i := \sigma_i$ we get the same for the burstiness component.

## 1.6 Remaining Work

Most of our research on *a-NET* (Section 1.4) is done and can be found in [50]. We have already analyzed the performance of *a-NET* with respect to LRU extensively, considering hierarchy networks as well as other topologies. We plan on testing how well *a-NET* works with other SCA algorithms, for Random replacement for example. We also plan on estimating the speed of convergence to a fixed point of *a-NET*, as a function of key parameters such as the size of the network. Finally, we will further investigate the implication of requests flowing in both directions across each hop, a phenomenon that does not occur in hierarchical systems and for which preliminary findings were presented in [48].

Regarding the bounding analysis, we need to implement the bounding algorithm described §1.5, formally establish how bounds on the input process are transformed

to bounds on the output process in both feed-forward and feedback networks, and evaluate the closeness of the bounds to actual performance.

# CHAPTER 2

# ON THE ERGODICITY OF CACHE NETWORKS

## 2.1 Introduction

In this chapter we focus on factors that impact the steady-state of content occupancy, which directly impacts the performance of CNs. Unlike previous work in this field, our results are applicable to networks of arbitrary topologies, and allow also heterogenous systems where caches use different management policies.

To date, analytical models for caching systems usually consider factors such as the cache capacity (i.e., how much storage is at its disposal), the topology of the cache network (e.g., hierarchical), the cache-management policies and the exogenous request arrival rates per-file at each cache [11, 20, 36, 50]. 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 these systems are *ergodic*, i.e., once the caches have undergone a "warmup" phase in which they are populated based on user demand, the initial state becomes irrelevant to the system behavior in the long run. In this chapter we will consider this assumption and its validity as a function of various system properties.

The major contributions of this chapter will be the following:

- We present two examples of non-ergodic CNs, in the sense that the content placed initially at the caches impacts the steady-state of the system and, as a result, its performance. In both examples, the observed behavior expresses itself only when the caches are networked.

- We establish several important properties of CNs, in the form of three sufficient conditions for a CN to be ergodic. Each property targets a different aspect of the system - topology, admission control and cache replacement policies.

- With regard to replacement policies, we demonstrate that 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. Section 2.2 presents our formal Markov Model for CNs, as well as definitions used throughout this chapter. In Section 2.3 we present two examples of non-ergodic CNs. In Section 2.4 we present three theorems regarding ergodicity CNs, and we conclude the chapter with Section 2.5 that details remaining work for this chapter. The material discussed in this chapter is presented in detail in [49].

## 2.2 A Markov Model for CNs

In this section we formally define the Markov Model of a CN, with notation summarized in Table 2.1. We model a Cache Network as a discrete-time Markov chain with state space $\Omega_0$. The system state $s$, $s = (s[1], s[2], \ldots, s[m])$, is a concatenation of $m$ vectors of size $|v_j|$ each, $1 \leq j \leq m$. The $i$th element in vector $v_j$ corresponds to the content in the $i$th position of $v_j$. When all caches have the same size, the state space $\Omega_0$ has cardinality $\left(\binom{n}{|v|} \cdot |v|!\right)^m$. When the order of the elements in the caches is irrelevant, states which 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 $\binom{n}{|v|}^m$.

A sample path $\boldsymbol{\tau}_a = (\boldsymbol{\sigma}, \boldsymbol{\pi})$ of a CN is determined by its initial state, $a$, a sequence of file requests $\boldsymbol{\sigma}$, and the resulting set of evicted files for each request $\boldsymbol{\pi}$. Each request in $\boldsymbol{\sigma}$ consists of the file requested and the node at which it exogenously

arrived. Let $\boldsymbol{\sigma} = (\sigma_k)_{1 \leq k \leq K}$ be a sequence of $K$ file requests. Each request has an associated set of files evicted from caches when the requested content was downloaded; let $\boldsymbol{\pi} = (\pi_k)_{1 \leq k \leq K}$ be a sequence of $K$ file eviction *sets*. For each request $\sigma_k$, let $\pi_k(h; \boldsymbol{\sigma}) \in F \cup \{\star\}$ be the file evicted from cache $h$, $1 \leq h \leq m$, while request $\sigma_k$ is satisfied. $\pi_k(h; \boldsymbol{\sigma}) = \star$ means that no file is evicted from $h$ when request $\sigma_k$ is satisfied. Note that a single request can cause, via file download path, changes at multiple caches. Specifically, when we assume ZDD, we can ignore intermediate stages reflecting system states while content is being forwarded along its download path, as the probability that a request arrives during this time is modeled as being negligible. We assume ZDD here only for presentation clarity, but the results presented in this chapter all hold for when ZDD is not assumed.

Given initial state $s$, let $s_k$ be the state resulting from the service of the $k$-th request in $\boldsymbol{\tau}_s$. Let $s_k[h]$ be the state of cache $h$ 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 $s_1 = s$ and $\varphi(\boldsymbol{\tau}_s) := s_K$ is the last state of $\Gamma(\boldsymbol{\tau}_s)$.

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

$$a_{c,d} = \begin{cases} 1, & \exists \boldsymbol{\tau}_c = (\boldsymbol{\sigma}, \boldsymbol{\pi}) : d = \varphi(\boldsymbol{\tau}_c) \wedge |\boldsymbol{\sigma}| = 1 \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

State $d$ can be reached from state $c$ if there is a sample path $\boldsymbol{\tau}_c = (\boldsymbol{\sigma}, \boldsymbol{\pi})$ such that $d = \varphi(\boldsymbol{\tau}_c)$. Let $a_{c,d}^{(n)}$ be an element of $\mathbf{A}^n$, $1 \leq c, d \leq m$. State $d$ can be reached from state $c$ if there exists an integer $n$ such that $a_{c,d}^{(n)} = 1$.

We conclude with some terminology used throughout this chapter:

**Definition 1** *A* transient state *of a CN is a state that can be left and never reached again.*

**Table 2.1.** Table of Markov Model Notation

| Notation | Meaning |
|---|---|
| $a, b \in \Omega$ | States in the Markov Chain system representation |
| $a[j]$ | The content of $v_j$ when system at state $a$ |
| $\boldsymbol{\tau}_a$ | Sample path, starting from state $a \in \Omega$ (or $\Omega_0$) |
| $\boldsymbol{\sigma}$ | Sequence of requests |
| $\boldsymbol{\pi}$ | Sequence of (sets of) file evictions |

**Definition 2** *A* recurrent state *of a CN is a state that is not transient.*

**Definition 3** *An* ergodic set *of a CN is a set in which every state can be reached from every other state, and which cannot be left once it is entered.*

**Definition 4** *An* ergodic CN *is a CN that comprises a single ergodic set.*

Note that the last definition slightly differs from the traditional definition of an ergodic Markov chain, whose states form a single ergodic set [27].

## 2.3 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.

### 2.3.1 Example 1

In our first example we consider the topology in Figure 2.1. Let $A, B$ be two disjoint sets of files, s.t. $|A| = |B| = |v|$, and assume LRU replacement is used at both caches. User 1 requests only files of type $A$, and user 2 only files of type $B$. Now, we consider two initial states: (I) when both caches are empty, and (II) when $v_1 = A$, $v_2 = B$. For scenario II the system will remain in the initial state indefinitely, with each cache storing only files from a single set, and no cache misses. For scenario I, on the other hand, cache misses will occur indefinitely, and both caches will store over time files from both sets.

While this example is outwardly simple, it contains several interesting lessons, beyond the impact on performance. First, this phenomenon is not possible with a single LRU cache, which is ergodic. Second, with this user request pattern, the state space is disjoint: the initial state described in II *cannot be reached* from any other state. Finally, slight changes in the request patterns of users can lead to drastically different cache behaviors. For example, if each user requests a set of files $C_1, C_2$, s.t. $|C_i| = |v|$ and $C_i \cap A \neq \emptyset, C_i \cap B \neq \emptyset$ for users $i = 1, 2$, the system would eventually converge to a single state, $v_i = C_i$, mirroring the user demand. Thus, we can see dependencies form in the network in such a way that seemingly small changes in user demand can sometimes have a very significant impact on the CNs behavior.

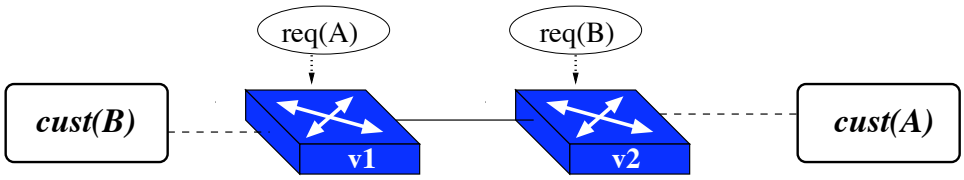


**Figure 2.1.** Example scenario in which the solution of the MC is dependent on initial state. Cube-shaped routers indicate cache-routers, rectangles denote custodians (with the stored FIDs indicated within each), and elipses indicate user demand. Solid lines indicate an edge between neighboring cache-routers and dotted lines indicate a cache-custodian or cache-user association.

### 2.3.2 Example 2

Another example is a network comprised of caches using the FIFO replacement policy and $n = |v| + 1$ files in the network, with a non-zero request rate for each of these files at each node. For this case, the set of states for which the cache is full can be partitioned into $(n-1)!$ ergodic 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 which repeats itself indefinitely. Thus, the initial state (or, if we start with an

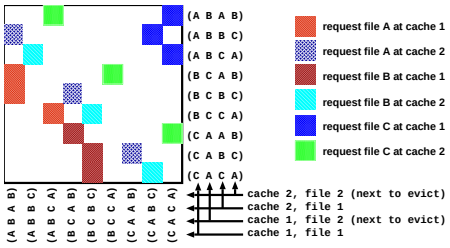


**Figure 2.2.** Transition matrix of Example 2 (diagonal elements not shown). The system state is $(w, x, y, z)$ where $w$ and $x$ (resp., $y$ and $z$) are the two files at cache 1 (resp., 2). Let $A = f_1$, $B = f_2$, $C = f_3$ when the initial state is $(f_1, f_2)$. Let $A = f_1$, $B = f_3$, $C = f_2$ when the initial state is $(f_1, f_3)$.

empty cache, the first $|v|$ unique files to populate the cache) determines the steady-state of the cache, resulting in a non-ergodic system. Despite this fact, $e_i = Pr(f_i \in v)$ is *independent* of the initial state, as can be determined from the balance equations, which are

$$(1 - e_i)\lambda_i = (1 - e_j)\lambda_j, \qquad 1 \leq i < j \leq n. \tag{2.2}$$

The system of equations (2.2) admits a single solution, independent of the initial state,

$$e_i = 1 - \left( \lambda_i \sum_{j=1}^{n} \frac{1}{\lambda_j} \right)^{-1} \tag{2.3}$$

Since the arrival process is IRM, the occupancy probability is also the hit probability [50], and so performance is unaffected by the initial state. When we interconnect these caches, however, performance can be greatly influenced by the initial state. Consider the CN shown in Figure 2.3. This network has three files and two caches arranged in a line, $|v| = n - 1 = 2$. Upon a cache miss, the request is forwarded in the direction of the custodian. Figure 2.2 shows the transition probability matrix for this case, obtained using Tangram II [14].

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 $\lambda_{1,1} = 0.35$, $\lambda_{2,1} = 0.55$, $\lambda_{3,1} = 0.1$, $\lambda_{1,2} = 0.05$, $\lambda_{2,2} = 0.15$, $\lambda_{3,2} = 0.8$. Table II 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.

**Table 2.2.** Example of the impact of initial state on system solution for the topology in Fig. 2.3 and transition matrix shown in Fig. 2.2.

| Initial State | $e_{12}$ | $e_{22}$ | $e_{32}$ |
|---|---|---|---|
| $(f_1, f_2)$ | 0.46651 | 0.63134 | 0.90214 |
| $(f_1, f_3)$ | 0.33054 | 0.76861 | 0.90083 |


**Figure 2.3.** Topology for scenario in which the solution of the MC is dependent on initial state. Cube-shaped routers indicate cache-routers, while rectangles with rounded edges denote custodians, with the stored FIDs. Solid lines indicate an edge in $E$ and dotted lines indicate a cache-custodian association.

Let us consider the behavior of FIFO caches with $n = |v|+1$. For a both networked and stand-alone caches the system is non-ergodic. However, for the stand-alone cache there is a symmetry among the states in the Markov Model, causing performance to be unaffected by the initial state (or initial requests, if the cache does not start full). Once the caches are networked, this symmetry no longer holds, and the non-ergodicity of the system impacts performance in a noticeable manner.

## 2.4 Conditions for Ergodicity: Topology, Admission Control and Replacement policy

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

**Definition 5** *The topology of a cache network is a* hierarchy *or* tree *if it has a single custodian and requests are forwarded along the shortest path to this custodian.*

**Definition 6** *An exogenous request stream for files at $v_j$ is said to be* positive *iff $\forall f_i \in F$, $\lambda_{ij} > 0$. If this condition holds for all $v \in V$, we say the exogenous request stream at the (cache) network is positive.*

**Definition 7** *A CN is said to be* individually ergodic *if its components are ergodic in isolation. This means that for each cache $v \in V$, when $v$ functions as a stand-alone cache and is subject to a positive exogenous request stream, $v$ is ergodic.*

**Theorem 1** *An individually-ergodic CN with positive exogenous request streams is ergodic if it is a hierarchy.*

**Theorem 2** *Consider an individually-ergodic CN where $v_j$ caches file $f_i$ (if and when $f_i$ passes through $v_j$) with probability $0 < \theta_{ij} < 1$ for all $1 \leq i \leq n$ and $1 \leq j \leq m$. Then this system is ergodic when subject to positive exogenous request streams.*

These two theorems are proven by demonstrating that there is a sample path between any two states $a, b \in \Omega$. 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 of finite size, this implies ergodicity of the entire system.

The proof for Theorem 1 relies on the following observation regarding cache trees:

**Lemma 2** *When ZDD is assumed, caches at the $i$th level of a hierarchy are unaffected by the state or request stream experienced at caches of level $i' \leq i$ (where level 0 is the root/custodian).*

Using this Lemma, we can start by constructing a sequence of requests and evictions that modify leaf caches from their state in $a$ to that in $b$. Next, we repeat this process one level up in the tree, and so on till we reach the root node. Based on Lemma 2

the changes in cache content at higher-level nodes will not impact the state at lower levels, thus concluding our proof. In a similar way for Theorem 2, we can find a sample path from one state to the next by using the probabilistic storage of caches to ensure each download only impacts the caches needed to move from $a$ to $b$.

The main contribution of this work is a theorem regarding the impact of a replacement policy on the ergodicity of the system. We begin with a narrow formulation considering only Random replacement, and then expand this result to a broad class of replacement policies by demonstrating structural similarities between the Random replacement Markov model and that of policies in this class.

Clearly, a CN in which all caches use random replacement is individually ergodic. In general, whether individually ergodic CNs are ergodic is an open question. Nevertheless, when establish the following results for CNs where all caches use the Random replacement policy.

**Theorem 3** *A CN that uses Random replacement is ergodic when subject to positive exogenous request streams.*

*Proof Method:* Let $a, b \in \Omega$ be two *recurrent* states. We prove our claim by showing there exists a state $c$ that is reachable from both $a$ and $b$. Since we assume $a$ and $b$ are recurrent, there must exist a reverse path from $c$ to each of them, and so $a$ and $b$ communicate with each other. This concludes our proof, since it shows there is a single ergodic set of states in this system.

*Proof Sketch:* We consider two CNs that are identical in all aspects (topology, routing, cache size and replacement policy, custodian location) except in their initial state - $\text{CN}_a$ begins in state $a$ and $\text{CN}_b$ in state $b$. Given their states, we generate a sequence of exogenous requests $\boldsymbol{\sigma}$, which will 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 $\gamma_{a,b}(j) = |a[j] \cap b[j]|$ be the agreement index of two network states at $v_h$. We say that two caches *agree* iff $\gamma_{a,b}(j) = |v|$. As we will demonstrate for the sequence we construct, for all $1 \leq k < h \leq K$ and all $1 \leq j \leq m$, $\gamma_{a_k,b_k}(j) \leq \gamma_{a_h,b_h}(j)$, and after $\boldsymbol{\sigma}$ is served for all $1 \leq j \leq m$, $\gamma_{a_K,b_K}(j) = |v|$.

After proving this theorem, we broaden our claims to cover a wide range of replacement policies that are similar, in the specific sense discussed below, to Random replacement. Specifically, we consider the following class of policies:

**Definition 8** *A replacement policy for an isolated cache is said to be* non-protective *if (under positive exogenous request streams) 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 $f_i \in v_j$ that arrives at $v_j$ 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 $|v| > 1$. We prove the following theorem:

**Theorem 4** *A CN with positive exogenous request streams is ergodic using Random replacement iff it is ergodic when each cache uses a non-protective replacement policy.*

At a high level, our proof for Theorem 4 demonstrates that given a path in the Markov chain of a Random replacement CN, we can add exogenous requests at each cache to cause a reordering of evictions, in order to maintain correlation with the path for Random replacement. From Theorems 3 and 4 we conclude:

**Corollary 1** *A CN with positive exogenous request streams is ergodic if the replacement policy used in each cache is non-protective.*

Note that this theorem applies also to *heterogenous* systems where each cache independently selects a non-protective replacement policy.

## 2.5 Existing and Remaining Work

A paper detailing the results described above is currently in submission to INFOCOM 2012, and can be found at [49]. In addition, we will attempt to locate more cases where the initial state impacts the steady-state the system converges to. We will also generalize Theorem 1 to apply to all feed-forward networks.

# CHAPTER 3

# INFERRING EVICTION RATES FROM CACHE MISS STREAMS

## 3.1 Introduction

In the previous two chapters considered networks with caches that operate without explicit consideration of the state of their neighbors: items are cached using single-cache replacement policies, and requests are forwarded using a static routing matrix. However, as we shall show in Chapters 4 and 5, we can improve routing and caching policies by using information about other caches and the flows between them. In the research described in this chapter, we explore an approach for *collecting* state information without any added communication between caches.

As has been noted in the literature [35, 37], the miss stream of a cache contains information about the operation of that cache. In a network of caches, $v$ receives portions of the miss stream of its neighbor $v'$, from which it might be able to recover information about request load and management policies running at $v'$. The ability to recover this information is a double-edged sword: if $v$ is benign it can use this information to better manage its own traffic [35], while a malicious $v$ might be able to exploit this information to negatively impact system performance [37].

In this chapter we consider the following challenge: given a *partial* miss stream of a cache, determine the rate of eviction at that cache, which is related to the *characteristic time* of a cache as defined in [11]. Knowing the eviction rate can be useful, since $v$ can then use this information in estimating the hit-probability of a request for $f \in F$ at neighboring caches which previously stored $f$ and decide where

to route the request. In the next chapter (Chapter 4) we develop a request routing framework that can make use of this type of information.

## 3.2 Related Work

The cache inference problem (CIP) was first formulated as such by Laoutaris et. al. [35]. In this work, the authors cast the problem as being the opposite of traditional cache questions: whereas research on caches has focused on how the output (miss stream) of the cache changes as a function of the input (arrival stream, replacement policies, cache size...), the cache inference problem tries to deduce the input based on the output. In their work, Laoutaris et. al. [35] consider the following specific problem: "Given the miss stream of a caching agent, infer the replacement policy, the capacity, and the characteristics of the input request stream". They constrain their goal to solving this problem for IRM and Generalized Power Law request streams. Our goal here differs in three important respects. First, we are interested in determining the eviction rate, which is not addressed by [35]. Second, we consider networked caches in arbitrary topology, while they consider a single-cache model. The implication of this assumption is that in [35] the entire miss stream is available to the inference algorithm, while within an arbitrary network request routing can result in the algorithm observing only a partial miss stream. Finally, we make no assumptions regarding the arrival process.

To the best of our knowledge, [35] is the only work on *passive* inference from a cache. More recently, as part of the effort to develop CCN, Lauinger [37] pointed out several *active* inference techniques for caches in a Cache Network, in which an attacker uses specially-designed "probe" requests to determine properties of a specific cache or the entire network, such as the network topology. While Lauinger considers active inference over a hierarchical topology with LRU replacement, we address passive inference within a general topology of RANDOM replacement caches.

## 3.3 Overview of proposed research

As in the previous chapters, we assume here that content, once found, is forwarded to the request origin along the reverse request path, and that it is stored at each cache along this path. With such a system, there is a 1-to-1 correlation between a cache *miss* and a cache *eviction*, since each cache miss results in new content being downloaded to the cache, which results in content eviction. In other words,

$$miss\ rate = eviction\ rate \tag{3.1}$$

Therefore, we can use the miss stream to estimate the probability that a recently-cached file is still in the cache. We consider two scopes of the data available for inference:

**Complete Miss Stream.** All misses are observed by the inference algorithm.

**Partial Miss Stream.** The miss stream for *a subset of file IDs* is available to the inference algorithm.

The first case relates to systems such as hierarchical networks, in which the entire miss stream is forwarded to a single neighbor. Since we are given the entire miss stream, the eviction rate can be determined exactly.

The second case relates to cases where every request for the same file is routed along the same path, such as when the routing matrix is static, or with dynamic routing protocols during periods of stability in the routing. Since we are given only part of the miss stream, determining the eviction rate is more difficult in the general case. However, if certain static properties of the neighboring cache are known we believe this information can be used to help improve inference. Specifically, we will address the case where the neighbor is known to use Random Replacement and that it has a cache size $|v|$. If the rate of eviction is $x$, the rate of eviction for any specific

file *currently in the cache* is $x/|v|$. However, $v$ observes only the cache misses. Let $\mu_{miss}$ be the mean time between two misses for the same file, then

$$\mu_{miss} = \mu_{ins,ev} + \mu_{ev,req} = \frac{|v|}{x} + \mu_{ev,req}, \tag{3.2}$$

where $\mu_{ins,ev}$ is the mean time between when the file was last inserted to the cache and when it was evicted, and is the same for all files, and $\mu_{ev,req}$ is the mean time between when the file was evicted and when it was requested next, causing a miss. We are interested in determining the eviction rate $x$, so we get

$$x = |v| \left(\mu_{miss} - \mu_{ev,req}\right)^{-1},$$

but we only use the miss stream we can only approximate this value with

$$\hat{x} = |v|\mu_{miss}^{-1} < x,$$

and so this approach gives us a lower bound on the eviction rate. The tightness of this bound decreases as $\mu_{ev,req}$ decreases, as might happen with popular requests. Thus, if the cache running the inference algorithm sees misses for files in $F' \subseteq F$, by computing bounds for each $f \in F'$ and taking the maximum on these, we can improve our bound, using tools such as Maximum Likelihood.

While the method just described focused on the observed miss rate per file $f_i$, additional tools can be used that take into consideration also the observed cache misses between each two misses for $f_i$. We propose to employ this information in several ways. First, we conjecture that we can use the miss stream for other files that also arrive at the observing node to more accurately estimate $\mu_{ev,req}$, since each of these cache misses will evict the file in question with probability $1/|v|$. Second, the number of observed misses is correlated with the total number of misses at the cache.

By using regression techniques and MLE we can better determine the eviction rate at the observed cache.

Our research here is still in its earliest stages. Initially, we will implement these ideas and test how well this inference approach works, as a function of traffic characteristics and cache size. Additionally, we will investigate how estimate precision increases with time. We will also develop a more rigorous mathematical formulation of this problem, possibly relying on a maximum likelihood approach as well as regression techniques.

# CHAPTER 4

# BREADCRUMBS: BEST-EFFORT CONTENT SEARCH IN CACHE NETWORKS

## 4.1 Introduction

In these next two chapters we drop our assumption in Chapters 1-2 that the routing matrix $\mathcal{R}$ is static, and consider *dynamic* request routing policies that adapt as a function of time and/or system state. We present an adaptive content search scheme, named Breadcrumbs, in which caches use only local information to determine where a copy of the content is *likely to be found*, and route requests (i.e., search for content) accordingly.

In general, caches can collect information to assist in content-request routing decisions, and can coordinate their actions with the rest of the network, in two ways - explicitly or implicitly. Explicit coordination includes any protocol that involves sending messages to other caches to announce their state (or state summary) [32, 36, 54]. These messages are then used by their recipients in managing their cache and routing requests. While explicit coordination can be useful, it comes at a cost of increased communication overhead and, at times, computationally-expensive coordination algorithms. Also, many times such systems also imply a degree of uniformity in cache operation, which can result in undesired effects [34].

An alternative to this approach is to rely on *implicit coordination,* where each cache acts based solely on its local view, i.e., the stream of requests and the content that passes through it. One example of implicit coordination is cache hierarchies, where the *position* of the cache in the hierarchy imposes a form of coordination, with

caches lower down serving one type of request pattern and shaping the miss stream for the next level up; this has led some to suggest using different policies at different levels in the hierarchy [9].

Breadcrumbs is a *best effort* content search policy of the second type. By "best effort" we mean that if content is stored at a cache, we expect a forwarded request may locate a cached copy, but there are no guarantees, and in the event that this dynamic search fails the content is eventually retrieved at a custodian node. The search we propose is made possible by keeping a minimal amount of per-file information (which we refer to as a "breadcrumb") at a cache, which is used in content search. We find that although our system promises best-effort only, it performs well even when compared to several classic, and more stateful, explicit-cooperation cache systems.

The structure of this chapter is as follows. In Section 4.2 we present related past research on content location within a cache network. In Section 4.3 we describe the Breadcrumbs system and consider its performance. In Section 4.4 we discuss plans for future work.

## 4.2 Related Work

For small scale caching systems, previous work has examined systems where a small number of caches are placed within a (possibly large) network. These works consider where in the network to place the caches [31] and where to cache specific objects [54], and have generally addressed the question in a limited number of topologies. Our work here assumes that content is cached blindly (or probabilistically) as it passes through a cache-router, and that all routers are equipped with caches, though both of these assumptions can be easily relaxed.

Work on improving the performance of a large-scale CN can be found in [2, 3], where the authors consider cache *replacement* in a CN and present an adaptive *caching* system named ACME. ACME uses machine learning techniques to determine when

and what to cache locally, without explicit communication among caches. With ACME each cache uses a pool of local virtual caches, each managed by a different static policy, and each simulating the behavior of the cache had it been using that replacement policy. ACME assigns performance-based weights to the virtual caches and, using ML algorithms, selects the best current policies from the virtual cache pool 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 uses adaptive *request routing*, instead of adaptive caching, in order to improve performance. 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 work.

Efficient content search has been addressed in the (Hybrid) P2P literature as well. In the Gnutella P2P system [4], content search is primarily done via exhaustive search (in the form of broadcasting requests to all neighbors), while others have proposed to limit the search cost by using Random Walks [10]. To mitigate communication overhead, peers using more recent versions of Gnutella implement the Query Routing Protocol (QRP) to notify their close neighbors of content they have; statistical versions of this method have been considered as well [32]. In [61], network nodes are organized into *semantic groups* such that only nodes with the same type of content requested are flooded with requests, while in [33] caching content along the download path within the P2P network is considered (when request distribution is zipfian) and used to reduce flooding. [20, 39] discusses search via expanding ring search and random walks, with [39] using simulations for evaluation while [20] provides theoretical bounds. Our Breadcrumbs system differs from all of these in that (a) content location changes dynamically, and (b) requests are routed towards likely locations of content without *any* explicit coordination among caches in the search process.

In [21], the authors address the problem of determining that certain content is *not available* anywhere in the P2P overlay network. Preliminarily, content is searched for via random walk, and after a bounded amount of search time the search ends if the content is not found. To help bound the search time of future random walks, a peer that took part in a failed search keeps a log of this occurrence, and any future search for this file that passes through this peer is halted. As time passes, peers with this information leave the network, allowing newer peers to search for content once more in case the content state of the system has changed. While in [21] 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 find where copies *may* be found.

The problem of locating content within a CN, which we address here, is related to several classic AI search challenges. Network traversal (using DFS/BFS, A* and other such algorithms) in a large network as we are considering here would be costly, both in time and in increased traffic in the network, making it inefficient. Other tools, such as Markov Decision Processes (MDPs), traditionally assume global knowledge to solve, and can also be very costly. Instead, the Breadcrumbs system we propose here can be framed, perhaps, as a form of *reinforcement learning*, in which each action taken is given a "reward" to indicate how "good" the action was. With Breadcrumbs, a request is rewarded with the (negative reward of the) time difference between when it arrived at a node and when content was last seen there, and each breadcrumb entry serves as a hint as to where good rewards can be found. In a similar way, Breadcrumbs could also be seen as a form of Simulated Annealing. The literature on this topic is vast, and we plan to investigate this connection.

## 4.3 Breadcrumbss Overview

### 4.3.1 The *breadcrumb* entry

In the Breadcrumbs system, when a file passes through a node in the network, that node stores a short-term routing entry known as a *breadcrumb,* denoted *bc.* Each *bc* entry is a 5-tuple of the form

$$(fid,\ t_{seen},\ t_{requested},\ v_{source},\ v_{dest}),$$

where $1 \leq fid \leq n$ is the unique ID of file in question; the next two entries indicate the last time the file was either seen or requested at this node, respectively; and the last two entries are the source and destination nodes of the path this file took at time $t_{seen}$. For a given cache, we sometimes add a subscript to indicate the file to which the *bc* relates, such as $t_{i,requested}$ for $f_i \in F$.

When each node keeps such a breadcrumb entry for each download it sees, a file being downloaded (sent) to the requesting node leaves a *trail of breadcrumbs* along its download path. Due to the small size of *bc* entries relative to the size of content stored at the cache, we assume here that an entry for each file can be maintained, per cache, indefinitely. The information in such an entry is, however, useful only for a limited time, and so the entry can be removed eventually, as we discuss shortly.

### 4.3.2 Using the *bcs*

Now we describe how we use these breadcrumbs. If a request $(v_j, f_i)$ arrives and $f_i \notin v_j$, by default the request is sent to $\mathcal{R}_{sp}(v_j, f_i)$, where $\mathcal{R}_{sp}$ is the routing matrix of the shortest path to content custodians. However, if there is a "recent" *bc* entry for this *fid*, the request can be routed either upstream towards $v_{source}$ or downstream towards $v_{dest}$ along the breadcrumb trail[1]. If this leads eventually to a *dead-end* node

[1]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].

$u$ - i.e., a node $u$ s.t. $f_i \notin u$ with no recent breadcrumb entry - $(v_j, f_i)$ is rerouted towards a content custodian, so as to ensure the end-user receives $f_i$ eventually.

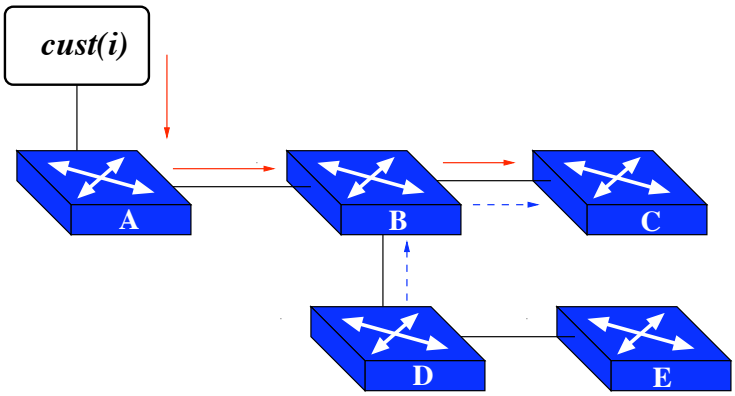


**Figure 4.1.** Breadcrumbs example. Red solid arrows denote the path of a file download, and blue broken arrows mark the request routing path that takes place at a later time. As can be seen, the request path in this example follows the downstream of the breadcrumb trail instead of heading towards the custodian.

Fig. 4.1 illustrates the basic operation of the protocol. In this example, at time $t$ file $f_i$ was downloaded from $cust(i)$ along the path $A \rightarrow B \rightarrow C$, leaving breadcrumb entries at each of these caches. At time $t' > t$ a request arrives at node $D$ for $f_i$. For the first hop $D$ sends the request towards the custodian, to node $B$. However, at node $B$ (where content was not found) the request is routed away from the custodian, along the downstream direction of the trail of breadcrumbs to node $C$. If $C$ has the content, it will send it towards $D$. If not, it will either forward the request further along the breadcrumb trail, or reroute it towards $cust(i)$.

### 4.3.3 Searching for Content with Breadcrumbss

Within the general outline described above, there are multiple decisions to be made when routing requests. These include the direction to follow when a *bc* entry is found, under what circumstances to follow the breadcrumb trail, and selecting the download path once content is found. In [48] we considered a version of Breadcrumbs called BECONS (Best Effort CONtent Search), with the following properties:

**When to follow.** BECONS uses a soft-state timeout mechanism that removes *bcs* after a preset time has passed since the file was last seen or requested. Specifically, let $\Delta_{ij}^{(seen)} \geq \Delta_{ij}^{(req)} \geq 0$ be two TTL parameters set at cache $v_j$ for $f_i$. A *bc* for $f_i$ is used until both $t - t_{i,seen} > \Delta_{ij}^{(seen)}$ and $t - t_{i,requested} > \Delta_{ij}^{(req)}$, at which point the entry is removed (also termed here *timed out* or *invalidated*). The intuition behind this approach is that content recently stored at a cache or requested at that cache is likely to be available there in the near future[2].

**Forward direction.** When valid *bc* entries are available at a node, the requests are forwarded *downstream* towards $v_{dest}$. This decision is motivated by the intuition that the downstream copies were placed there more recently, and thus a copy is more likely to be found there.

**Download path.** Finally, we consider two download policies: downloading along the reverse request/search path, and downloading along the shortest path from where content was found to the request origin. These are termed DFQ (Download Follows Query) and DFSP (Download Follows Shortest Path).

In [48] we have proven several properties for a simple version of BECONS, called S-BECONS (Simple BECONS). S-BECONS is BECONS where $\forall i \in [n], j \in [m]$ $\Delta_{ij}^{(seen)} = \Delta_{ij}^{(req)} = \Delta$ for some contstant $\Delta > 0$. We use the following definitions in stating our results:

**Definition 9** *Consider a trail of* bc *along the trail* $v_{\pi(1)}, ..., v_{\pi(k)}$ *for some permutation* $\pi$ *over node indices* $1, ..., m$ [3]. *Such a trail is said to be broken if there exist indices*

[2]With LRU requests can directly extend the time a file spends in the cache. For other policies such as FIFO and RANDOM, where a request does not have this effect, still more requests indicate that the content will be stored along new download trails extending the original trail.

[3]Assume here that this sequence of nodes is a valid path, i.e., for all $i \in [1, k-1]$ there is an edge $(v_{\pi(i)}, v_{\pi(i+1)}) \in E$.

$1 < h < i < j < k$ *s.t. the breadcrumbs at* $v_{\pi(h)}$ *and* $v_{\pi(j)}$ *are valid while the breadcrumb at* $v_{\pi(i)}$ *is invalid.*

**Definition 10** *A* stable *trail is a trail that for every query at node* $v_{\pi(j)}$ *that is following the trail, the trail suffix* $v_{\pi(j)}, ..., v_{\pi(k)}$ *will not become broken until the request traverses the trail or finds content along it.*

Note that from this last definition, if a request starts a search along a stable downstream trail and ends at an invalid *bc*, all *bcs* further down are invalid as well.

**Definition 11** *A policy is said to have trail invalidation built into it, if there is some mechanism such that a node* $v$ *can determine, for any file* $f$*, that all the caches along the suffix of the downstream trail (starting from* $v$*) have evicted* $f$ *since the trail was set up.*

There are many advantages to a system that ensures stability and has the capacity for trail-invalidation. Stability ensures that there will be no break in the trail, and so we know a search downstream will reach all nodes in the trail unless content is found beforehand. Trail invalidation, on the other hand, helps prevent sending a request down paths where the content copies that set up the breadcrumb trail, via content download and subsequent requests, have since been evicted. Intuitively, once a node is aware of this information it should be less inclined to follow that trail. In [48] we demonstrate that S-BECONS provides trail invalidation using only local information, and ensures all trails as stable when propagation delays and queueing delays are constant, such as with uncongested networks.

With S-BECONS, a content request always follows valid *bcs* downstream towards where content was more recently placed. When trail stability holds, nodes downstream will tend to have valid breadcrumbs even when the breadcrumbs upstream have already timed-out. Combining these two results with the emergence of a *border*

*node* along every trail (associated with a specific file $f_i$) - a node $v_{border(i)}$ such that a request for $f_i$ arriving at nodes further downstream will follow the trail, while requests arriving at caches upstream will be directed to $cust(i)$. Thus, nodes outside a certain vicinity (determined by the $\Delta$ values on the breadcrumbs) of $cust(i)$ will forward requests downstream along the breadcrumb trail, while those within its vicinity will continue to forward requests to $cust(i)$.

In addition to these findings, we have also simulated and evaluated S-BECONS, comparing its performance to that of CNs, where requests follow the shortest path to a custodian, as well as to a CN where a cache explicitly coordinates its contents with the next-hop cache along $\mathcal{R}_{sp}$[4]. Our work has focused to-date mainly on how Breadcrumbs reduces load on custodians; an example result is presented in Figure 4.2, which shows performance, broken down by FID, for a 10-by-10 torus network with four custodians and $|v| = 20$, a zipfian exogenous request load arriving at each LRU cache and DFSP used for download. We have also found that the relative performance of Breadcrumbs improves as the network size increases.

When compared to the cache-coordinating system mentioned above, we have found that the relative performance of Breadcrumbs also improves as the value of $|v|/n$ decreases. The intuition behind these results is that any cooperation among a bounded number of caches in the network is limited in the total storage capacity of this set of cooperating caches, while Breadcrumbs can potentially route requests to content anywhere in the network.

### 4.3.4 Metrics

Another focus of our work is that of appropriate metrics for evaluating the performance of search in cache networks. Traditionally, performance metrics consist of

[4]Specifically, this means (1) a file passing through will be cached only if it is not in the next-hop cache, and (2) we prefer to evict from a cache content that is in the next hop.

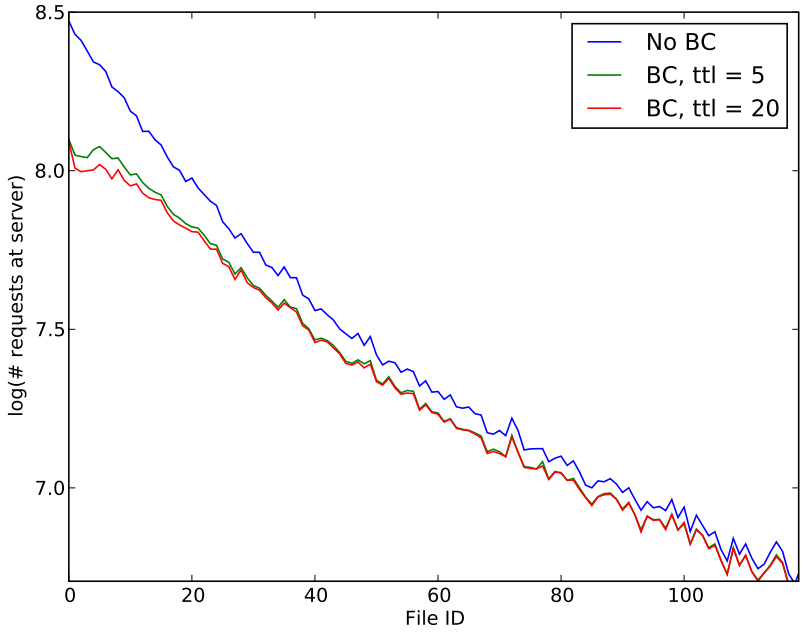


**Figure 4.2.** Load at custodians with and without Breadcrumbs used. File IDs ($x$-axis) listed in order of decreasing popularity, and performance ($y$-axis) measured by total number of requests at custodians. In this example popular file requests show the most benefit, with S-BECONS reducing the custodian load for the most popular file (FID=0) by 1/3.

cache hit probability, user delay and network congestion and load; these are yardsticks against which the effectiveness of a search policy must be measured. However, performance according to these metrics is not solely the result of the search policy, but rather a combination of the search policy and the state of the caches in the network (i.e., what is cached and where). The performance metrics listed above are not informative as to *why* the given policy resulted in that performance.

For this reason, in our future work we will consider metrics that evaluate the effectiveness of the search policy given the state of the network when search commenced. Our approach to this will be to compare three distances: (a) the length of the search path, (b) the shortest distance from request origin to where content was found, and (c) and the distance to the closest available content copy when search commenced.

We hope these metrics may provide insight into the responsiveness of the search policy is to the system state.

## 4.4 Future work

In the previous section we discussed our findings from our initial simulations. Still, additional experiments are needed to support these preliminary findings. In addition strengthening what we have found, we will also explore the following avenues in future work:

**Breadcrumbs and Simulated Annealing.** Just as simulated annealing incorporates time into its decision making process in a continuous fashion, so we believe a similar approach might be suitable for decaying (rather than timing out) the relevance of breadcrumb entries, i.e., as time elapses we should use the breadcrumbs with decaying probability. We will explore this method and compare it to the timeout approach used to-date.

**Search metrics.** As mentioned above, we will evaluate the effectiveness of Breadcrumbs search w.r.t. the best possible search. This measures how responsive our search is to the state of the system.

**Eviction-rate aware timeout.** Using the tools developed in Chapter 3 to compute eviction rates from cache misses, we will investigate how to set the timeout period for a breadcrumb entry as a function of the eviction rate at the node to which the breadcrumb points. This portion of our work will be conducted for Random replacement, in accordance to Chapter 3.

# CHAPTER 5

# THE VALUE OF INFORMATION: LOAD-BALANCING WITH GLOBAL INFORMATION

## 5.1 Introduction

One of the main purposes of adding caching to the network is to reduce the time between when content is requested and when content arrives at the requestor. By distributing copies of content in many locations around the network, customers can locate content close to them, reducing the propagation delay as well as the congestion in the network. Additionally, queueing delay for content service/transmission at the caches and custodians can be much reduced as the number of content copies increases. This last property relies, naturally, on the efficient balancing of load across custodians and caches.

In this chapter we will discuss early thoughts on investigating how sensitive a cache network is to short-term fluctuations in the load distribution, as well as to the download path. We consider a hypothetical system in which, at time $t$, each cache has up-to-date global knowledge of the state of the system - what content is available at each cache at $t$. In addition, each cache has some knowledge of the recent load at each cache in the network. As we specify below, we are interested in how load-balancing is affected as this load information becomes less and less recent, and as a function of content download path which affects content distribution in the network.

## 5.2 Related Work

In the literature there is much discussion of balancing load among network elements. We begin in listing some results for load balancing in general, and then present some results for Content Centric Networks.

### 5.2.1 Load Balancing - General

Load balancing has been considered over homogenous [15] systems, where service time for all jobs is modeled identically and load is measured by queue length, and heterogenous systems where jobs vary in their service time [40]. In this chapter we will begin with the simpler, homogenous model, assuming that at each node (cache or custodian) the service time of each serviceable request is the same.

The method of load balancing we consider here is *adaptive,* assigning each request to a cache with the content based on system state. We assume that each node routes requests based on exact information regarding each node in the system. Clearly, and as has been noted in [25], a policy that would rely on such information will not scale. This is especially significant to us since here we consider large-scale CNs. However, in this chapter we are primarily interested in understanding the utility of timely information regarding load, and so considering a system where nodes have this amount of information about the system can give us a benchmark for comparing the behavior of practical policies developed in the future.

### 5.2.2 Load Balancing in CNs

Load balancing in a Cache Network differs from the general formulation of the problem in that each node in the network can, at a given point in time, service only a subset of requests (=jobs) that arrive at the system. Furthermore, this subset changes dynamically with the download process, as caches store and evict files over time. These dynamic changes in system state are a major focus of our research here. An additional level of complexity in these systems is caused by the fact that requests

usually inspect caches along the request path, and download content from the first cache hit. With such a system, some node $v'$ might not be reachable from a requesting node $v$, since any path from $v$ to $v'$ has a node in between that stores the content. As stated in the next section, we drop this requirement in this chapter and allow for requests to selectively inspect caches based on load balancing policy.

In addressing load balancing, one component can be to coordinate the placement of content in caches. [51] considers how this might be done within Content Centric Networks, and surveys related work using this approach. In keeping with the basic philosophy of Breadcrumbs we allow content placement to take place "blindly" as it is downloaded, and aim at load balancing only via adapting the request routing.

When considering *hierarchical* CNs, Borst et al [7] formulate a linear programming problem, the solution of which minimizes the "cost" of service, where cost is a function of bandwidth usage. For symmetric scenarios, in which the cache hierarchy is symmetric and caches in the same tree-level have the same properties, they solve this linear program. In addition to this, they use insights from this solution to construct both intra-level and inter-level cache coordination algorithms to come close to this optimal solution.

Load distribution among peers in HP2P systems as a function of content search is discussed in [56]. Here the authors consider the optimal replication strategy, and show that uniformly replicating in the network (i.e., given a copy is in the network, place it at each peer with equal probability) will yield the best results when request patterns are also uniformly distributed among peers. [56] does not consider how to achieve this distribution in practice, and relies on several independence assumptions that are violated in real systems.

Our work here departs from all the above, in several ways. First we consider general topologies, and take into account the download path of content and consequent caching along that path. Additionally, we do not consider the optimal strategy for

load balancing, but rather strive to understand how a given load balancing policy (e.g., download from least-loaded node with content) performs as a function of the quality of its information/input.

## 5.3 Proposed Research Outline

We begin by considering a system where caches have partial global knowledge about the state of the system. Specifically, caches are aware at any given moment of the content stored at each cache in the network, as well as the mean load at caches in the system. Load is defined as the mean number of downloads at a node per unit time, though we will also consider weighted systems, which will assign a different cost for load at caches and load at custodians, and aim at cost minimization.

We will consider, at first, the performance of a simple adaptive shortest expected delay (SED) policy of servicing a request at the node with the lowest (known) load. To this end we will assume ZDD, also known as negligible *transfer delay* in the load-balancing literature. This is essential for a CNin which we want to deterministically locate the available node with the lowest current load, since the list of request types a node can serve changes over time, and delay in the request transfer could result in reaching a node that no longer holds the content.

In order to focus on the utility of information, we will limit our consideration of network topology in this chapter in one notable way. For a cache to be able to route a request to a preselected location (e.g., a cache temporarily holding the content, or a custodian), we must ensure a request en-route to node $v$ does not download content from a different node $v'$ along the path to $v$. Thus, in this chapter we will first consider the case where a request *does not inspect* caches en-route to $v$ and ignores content stored along the way.

In addition to knowing the contents of all caches in the network, we assume nodes are also aware at time $t$ of the average load on each cache over the window

$\tau_{x,y}(t) = [t - x, t - y]$ (where $x \geq y \geq 0$). Given this information, a routing policy which we shall initially consider will have node $v$ route a request for file $f_i$ to the cache $v'$ s.t. the load at $v'$ is minimal for all nodes where $f_i$ is available at time $t$.

In addressing load balancing in this system, we will consider two performance-impacting variables. The first is the observed window of time $\tau_{x,y}(t)$ at the disposal of a cache for all $t$ when making the routing decision. While different windows will result in different load balancing, for a given model of exogenous request arrivals it is unclear what the optimal window might be, how fast or slow performance degrades from that optimal point, and what is the tradeoff between recent information (the size of $x$) and the window size $(y - x)$. The second variable that might have great impact on performance is the *content download path.* The download path determines the distribution of the number of copies for each file, which can result in great changes to load variance.

The work proposed above is only the beginning of an outline of what will be investigated in this chapter. Further work is needed to define the exact scope of research to be done here, its relationship to earlier research and the methodologies to be employed. Depending on our findings, we shall modify and expand the scope of our proposed work here.

# 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] Breslau, Lee, Cue, Pei, Cao, Pei, Fan, Li, Phillips, Graham, and Shenker, Scott. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM* (1999), pp. 126–134.

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

[10] 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.

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

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

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

[14] 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.

[15] Eager, D.L., Lazowska, E.D., and Zahorjan, J. Adaptive load sharing in homogeneous distributed systems. *IEEE transactions on software engineering 12*, 5 (1986), 662–675.

[16] 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.

[17] 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).

[18] Giovanna Carogli, Massimo Gallo, Muscariello, Luca, and Perino, Diego. Modeling data transfer in content-centric networking. Tech. rep., France Telecom, 2011.

[19] 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.

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

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

[22] 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.

[23] 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.

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

[25] Kabalan, K.Y., Smari, W.W., and Hakimian, J.Y. Adaptive load sharing in heterogeneous systems: Policies, modifications, and simulation. *International Journal of Simulation, Systems, Science and Technology 3*, 1-2 (2002), 89–100.

[26] 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.

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

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

[29] 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.

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

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

[32] 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.

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

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

[35] Laoutaris, N., Zervas, G., Bestavros, A., and Kollios, G. The cache inference problem and its application to content and request routing. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE* (may 2007), pp. 848 –856.

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

[37] Lauinger, Tobias. Security and scalability of content-centric networking. Master's thesis, TU Darmstadt, Darmstadt, Germany, and Eurcom, Sophia-Antipolis, France, September 2010.

[38] 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.

[39] 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.

[40] Mirchandaney, R., Towsley, D., and Stankovic, J.A. Adaptive load sharing in heterogeneous systems. In *Distributed Computing Systems, 1989., 9th International Conference on* (1989), IEEE, pp. 298–306.

[41] 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.

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

[43] 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.

[44] 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.

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

[46] Qiu, Lili, Padmanabhan, V.N., and Voelker, G.M. On the placement of web server replicas. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (2001), vol. 3, pp. 1587 –1596 vol.3.

[47] 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.

[48] 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.

[49] Rosensweig, Elisha J., Menasche, Daniel S., and Kurose, Jim. On the steady state of cache networks. In *IEEE INFOCOM (in submission)* (2012). http://gaia.cs.umass.edu/networks/papers/INFOCOM12-CN-SteadyState.pdf.

[50] 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.

[51] Soulas, Vasilis, Gkatzikis, Lazaros, and Tassiulas, Leandros. Online storage management with distributed decision making for content-centric networks. http://sites.google.com/site/vsourlas/NGI_2011_CR.pdf.

[52] 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.

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

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

[55] 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.

[56] Tewari, Saurabh, and Kleinrock, Leonard. Analysis of search and replication in unstructured peer-to-peer networks. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (New York, NY, USA, 2005), SIGMETRICS '05, ACM, pp. 404–405.

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

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

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

[60] 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.

[61] 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.