CUP: CONTROLLED UPDATE PROPAGATION IN PEER-TO-PEER NETWORKS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Mema Dimitra-Isidora Roussopoulos
July 2002

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Dr. Mary Baker
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Dr. Armando Fox

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Dr. Rajeev Motwani

Approved for the University Committee on Graduate Studies:

Abstract

A peer-to-peer system is a self-organizing distributed system where a group of Internet hosts both provide and receive services from each other in a cooperative effort without distinguished roles as pure clients or pure servers. A fundamental problem in peer-to-peer networks is that of locating content. Given the name or a set of keyword attributes (metadata) of an object of interest, how do you locate the object within the peer-to-peer network? Most peer-to-peer networks return a set of index entries in response to a search query. These index entries point to the locations of replica nodes in the network that serve the content whose metadata satisfies the search query.

Once the location algorithm is decided upon, two basic problems emerge. First, can we improve upon the time it takes to resolve a search query? Second, after we resolve the search query to obtain a set of index entries, how do we choose from amongst them?

To address the first problem of improving search query time, many peer-to-peer designers suggest caching index entries at intermediate nodes that lie on the path taken by a search query. However, little attention has been given to how to maintain these intermediate caches. To this end, we propose a new protocol, CUP, for performing Controlled Update Propagation in a peer-to-peer network. CUP asynchronously builds and maintains caches of index entries while answering search queries. CUP controls and confines propagation to updates whose cost is likely to be recovered by subsequent queries. CUP allows peer nodes to use their own incentive-based policies to determine when to receive and when to propagate updates. We develop CUP and compare its performance against caching with expiration at intermediate nodes. We show that CUP significantly reduces average query latency and that CUP update overhead is compensated for many times over by its savings in cache misses.

The second part of the thesis focuses on how to choose amongst the returned index entries. This choice is important, because the demand for particular content should be balanced across the set of replica nodes serving that content. Previous decentralized load balancing techniques in distributed systems base their decisions on periodic updates containing information about load or available capacity. We show that these techniques do not work well in the peer-to-peer context; either they do not handle peer node heterogeneity, or they suffer from significant load oscillations. We propose a new decentralized algorithm, Max-Cap, based on the maximum inherent capacities of the replica nodes and show that unlike previous algorithms, it is not tied to the timeliness or frequency of updates. Yet, Max-Cap can handle the heterogeneity of a peer-to-peer environment without suffering from load oscillations.

Dedication

Στην οικογενεια μου.

(To my family.)

Acknowledgments

This thesis would not be possible without the support of a number of colleagues and friends. First, I would like to thank my adviser, Mary Baker, for her kindness, her guidance, and her continuous encouragement throughout my graduate school life. Mary taught me to be an independent researcher, gave me the freedom to work on any research topic of my choice, and taught me to present my work clearly and convincingly. Moreover, Mary has extraordinary grace when interacting with colleagues. I hope I can follow her example.

Thanks to the other two members of my reading committee, Armando Fox and Rajeev Motwani. I could not have asked for more helpful and cooperative readers. Their invaluable insights and enthusiastic feedback throughout the latter portion of my PhD career greatly helped improve the work presented here.

Thanks to Doug Terry and Andrea Goldsmith, who served on my orals committee. Both posed questions that helped me think about my work in a different way. Thanks to all the above folks for serving on my orals committee. I could not have asked for a smoother scheduling of the defense.

Thanks to all the past and present members of the MosquitoNet research group for their support, their insights, and their good cheer throughout the years.

Thanks to Alexandros Labrinidis for the use of his “report-maker” Perl script, which was a huge time-saver. Without it, I’d still be looking at graphs.

Thanks to the following generous sources of funding that supported this work. These include: a National Science Foundation Graduate Fellowship, Daimler-Chrysler, a gift from NTT Mobile Communications Network, Inc. (NTT DoCoMo), a grant from the Keio Research Institute at SFC, Keio University and the Information-Technology Promotion Agency of Japan, a grant from the Okawa Foundation, the Stanford Networking Research Center, and a DARPA (contract N66001-00-C-8015),

Finally I would like to thank the special group of friends who made the sometimes arduous task of getting a PhD more pleasant:

Thanks to Petros Maniatis and Edward Swierk, two of the dearest friends a person could have, for listening to my jokes (multiple times each) and more importantly, for just plain listening. In Petros, my unforgettable officemate, I found someone willing to let me bug him at any time, whether for technical advice, sys-admin pointers, or moral support, and a cohort with which to go through the PhD process. In Edward, I found not only a "best friend", but also one of the sweetest and most pleasant souls around.

I owe the preservation of my sanity during my first year here to Jennifer Hughes and Luis Gravano. I was very lucky to be assigned to be Jen's roommate in EV 50 B when I first arrived. This graduate housing assignment brought me wonderful conversations and a friend I will cherish for life. Thanks to Luigi for always being willing to listen even when he was running late, and for laughing at ALL of my jokes, even the "rare" non-funny ones. :-)

Thanks to Aris Gionis, for his unique perspective and the many long enjoyable conversations, to Kevin Lai for his great kindness and generosity, to Katerina Argyraki for the many wonderful dinner conversations, to Antonis Hondroulis and Christine Jesser for cheerfully feeding me on many an occasion, to Sergio Marti and TJ Giuli, for their boundless good spirits, and to the Hellas Greek Dancing Group for the pleasant I-Fest rehearsals which were great stress-busters for me.

Finally I dedicate this thesis to the members of my family. I cannot begin to explain how much they mean to me. My father and academic inspiration, Nikos. During how many difficult times in the whole process did you say to me "Everyone goes through this phase, don't worry"? I needed to hear it every time. You believed in me, supported me, and convinced me I had the strength to keep going.

My loving mother, Nina, who countless times supported me and reminded me that my health and trying my best are the two most important things and everything else is “*σταχτη και μπουρμπϵρη*.” As I get older, I only now begin to fathom the immeasurable love and support my parents have given me. I can only hope to be like them with my children.

My dearest sister and best friend Evie, whose cheer, clever humor, and nicknames for me are always a great source of happiness. I can only hope to one day return the “care package.”

Finally, I would like to thank Pavlos Christofilos, for giving me the ultimate motivation to finish my PhD. Seven is indeed a magic number.

My heartfelt thanks to you all, it was worth it...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter motivates two problems observed in peer-to-peer networks, lists the major contributions of this thesis, and presents the outline for this dissertation.

1.1 Problem Statement

Peer-to-peer networks are self-organizing distributed systems where participating nodes both provide and receive services from each other in a cooperative effort without distinguished roles as pure clients or pure servers. Peer-to-peer networks are receiving much attention primarily because of the great number of features they offer applications that are built on top of them. One of the primary features is that they allow applications to exploit the aggregate processing power and storage capabilities of participating peer nodes [4]. For example, distributing content has been a dominant use of peer-to-peer networks [18, 3, 1].

Other features of peer-to-peer networks include:

- *Decentralized administration*: Participating nodes are widely-distributed hosts (typically PCs) that live at the edge of the Internet. These nodes form and maintain an overlay network without depending on a particular global entity or set of entities being in place to administer changes in network membership.

- *Scalability*: A peer node joining or leaving the network is a local operation involving only a few nodes. This gives peer-to-peer networks the potential to scale to thousands of nodes.

- *Availability*: Peer nodes are equal in functionality. This means that any one of a set of "replica" nodes can provide the requested content, increasing the availability of content without requiring the presence of any particular serving node.
- *Fault tolerance*: The self-organizing, decentralized nature of peer-to-peer networks allows them to continue to function even as nodes fail or depart the network.
- *Autonomy from DNS servers*: Since peer nodes can be home PCs with unstable connectivity and unpredictable IP addresses, peer-to-peer networks do not depend on the presence of the DNS system to function [4].
- *Anonymity*: Peer-to-peer networks allow users to publish or request content anonymously or pseudonymously to protect their privacy [18, 59]. This is because in a peer-to-peer network, content-distribution does not depend on having a well-known set of nodes to which users are obligated to submit requests to publish or receive content. Instead, any peer node can handle such requests or forward them on to other peer nodes. In a large peer-to-peer network with thousands of nodes, tracing the traversal of a particular request to the source of the request can be quite difficult.

Along with these features has come an array of technical challenges. For example, a fundamental problem in peer-to-peer networks is that of locating content. Given the name or a set of keyword attributes (metadata) of an object of interest, how do you locate the object within the peer-to-peer network? Most peer-to-peer networks push a set of index entries in response to a search query. These index entries point to the locations of replica nodes in the network that serve the content whose metadata satisfies the search query.

The search algorithm used to retrieve these index entries differs from system to system. The algorithm can be unstructured as in Gnutella [3] where there is no organization in where index entries are stored. In such networks, search queries travel haphazardly through the network via flooding or random walks. The algorithm can also be structured (e.g., as in CAN [45]). This type of algorithm uses a distributed hash table where index entries are dynamically allocated across nodes in a very organized manner such that every node is the authority node for a specific set of index entries. Search queries travel along a well-defined path between the querying node and the authority node responsible for the index entries pertaining to this query.

Once the search algorithm is decided upon, there arise two fundamental problems. First, how can we improve upon the time it takes to resolve a search query? That is, how can we reduce search query latency? Second, after we resolve the search query to obtain a set of index entries, how do we choose from amongst them? This choice is important for load-balancing. Ideally, you want to choose a replica node such that the overall demand for content is distributed fairly across the set of replicas serving this particular content. We motivate these problems in turn.

1.1.1 Reducing Query Latency

Recent work suggests that metadata-based search queries for index entries can be a significant performance bottleneck in peer-to-peer systems [17]. As a result, to make search query resolution more timely, designers of peer-to-peer systems suggest caching index entries at intermediate nodes that lie on the path taken by a search query [3, 54, 45, 57]. We refer to this as *Path Caching with Expiration* (PCX) because index entries typically have expiration times after which they are considered stale and require a new search.

PCX is desirable because it balances query load for popular entries across multiple nodes, it reduces latency, and it alleviates hot spots. However, little attention has been given to how to maintain these intermediate caches. The cache maintenance problem is interesting because the peer-to-peer model assumes that the index will change constantly as peer nodes continuously join and leave the network, content is continuously added to and deleted from the network, and replicas of existing content are continuously added to alleviate bandwidth congestion at nodes holding the content. Each of these events causes a change in the current global set of valid index entries. Keeping cached index entries up-to-date therefore requires tracking which entries need to be updated, as well as tracking when interest in updating particular entries at each cache has died down so as to avoid unnecessary update messages.

In this thesis we propose a new protocol, CUP, for performing Controlled Update Propagation to maintain caches in a peer-to-peer network. CUP asynchronously builds caches of index entries while answering search queries. It then propagates updates of index entries to maintain these caches. CUP controls and confines propagation to updates whose cost is likely to be recovered by subsequent queries. CUP allows peer nodes to use their own incentive-based policies to determine when to receive and when to propagate updates.

We develop the CUP protocol in detail and perform extensive experiments comparing CUP against PCX under typical workloads that have been observed in measurements of real peer-to-peer networks. We show that CUP often reduces average search query latency manifold. Moreover, CUP overhead is more than compensated for by its savings in cache misses. The cost of saved misses can be two orders of magnitude greater than the cost of updates pushed.

1.1.2 Choosing Amongst Returned Index Entries

After retrieving the set of index entries in response to a local search query, the peer node must choose which of the index entries to use to forward the client content request. This is the second problem we focus on in this thesis. One reason for considering this choice is load balancing. Some replica nodes may have more capacity to answer queries for content than others, and the system can serve content in a more timely manner by directing queries to lightly loaded replica nodes. Until now, however, there has been little focus on how to make this choice.

In this thesis we study the problem of load-balancing demand for content in a peer-to-peer network. This problem is challenging for several reasons. First, in the peer-to-peer scenario there is no centralized dispatcher that performs the load-balancing of requests; each peer node individually makes its own decision on how to allocate incoming requests to replicas. Second, nodes do not typically know the identities of all other peer nodes in the network, and therefore they cannot coordinate this decision with those other nodes. Finally, replica nodes in peer-to-peer networks are not necessarily homogeneous. Some replica nodes may be very powerful with great connectivity, whereas others may have limited inherent capacity to handle content requests.

Previous load-balancing techniques in the literature base their decisions on periodic or continuous updates containing information on load or available capacity. We refer to this information as load-balancing information (LBI). These techniques have not been designed with peer-to-peer networks in mind and thus

- do not take into account the heterogeneity of peer nodes (e.g., [32, 43]), or
- use techniques such as migration or handoff of tasks that cannot be used in a peer-to-peer environment (e.g., [35]), or
- can suffer from significant load oscillations, or "herd behavior" [43], where peer nodes simultaneously forward an unpredictable number of requests to replicas with low reported load or high reported available capacity causing them to become overloaded. This herd behavior defeats the attempt to provide load-balancing.

Most of these techniques also depend on the timeliness of LBI updates. The wide-area nature of peer-to-peer networks and the variation in transfer delays among peer nodes makes guaranteeing the timeliness of LBI updates difficult. Peer nodes will experience varying degrees of staleness in the LBI updates they receive depending on their distance from the source of updates. Moreover, guaranteeing timeliness of LBI updates is also costly, since all updates must travel across the Internet to reach interested peer nodes. The smaller the inter-update period and the larger the overlay peer network, the greater the network traffic overhead incurred by LBI updates. Therefore, in a peer-to-peer environment, an effective load-balancing algorithm should not be critically dependent on the timeliness of updates.

To this end we propose a practical load-balancing algorithm, Max-Cap, based on the inherent maximum capacities of the replica nodes. We define maximum capacity as the maximum number of requests per time unit a replica allocates for answering requests for its content. The maximum capacity is like a contract by which the replica agrees to abide. If the replica cannot sustain its advertised rate, then it may choose to advertise a new maximum capacity.

We evaluate load-based and availability-based load-balancing algorithms and compare them with Max-Cap in the context of CUP. Specifically, we leverage the update propagation mechanism of the CUP protocol by piggybacking load-balancing information onto the updates CUP propagates. We demonstrate that Max-Cap is a practical algorithm for load-balancing in a peer-to-peer environment. Max-Cap handles heterogeneity, is not tied to the timeliness or frequency of updates as algorithms based on load or available capacity are, and does not suffer from oscillations when the workload rate is within 100 percent the total maximum capacities of the replicas. Moreover, Max-Cap adjusts better to constantly changing replicas sets and to very large fluctuations in the workload where the rate of requests suddenly increases to several times the total maximum capacities one instant and then drops dramatically the next.

1.2 Primary Contributions of this PhD Thesis

The primary contributions of this thesis are:

1. The development of a new protocol, CUP, for maintaining caches of index entries in a peer-to-peer network through Controlled Update Propagation.

2. A comprehensive suite of experiments demonstrating that CUP greatly reduces average search query latency with overhead that is many times recovered by its savings in cache misses.

3. A demonstration through experimental evidence and detailed explanation of why previous load-balancing techniques can suffer when applied in a peer-to-peer environment.

4. A new and practical algorithm, Max-Cap, for load-balancing in a peer-to-peer network.

1.3 Outline of Dissertation

The remainder of this dissertation is organized as follows:

Chapter 2 defines some background information on search in peer-to-peer networks and introduces some terminology used throughout this dissertation. This chapter also describes previous work related to the CUP protocol as well as previous load-balancing work in distributed systems.

Chapter 3 describes the design of the CUP protocol and provides an extensive experimental evaluation of the protocol.

Chapter 4 introduces the Max-Cap load-balancing algorithm and shows how this algorithm compares to previous load-balancing algorithms in a peer-to-peer environment. This chapter also provides a detailed explanation of why previous load-balancing algorithms can suffer in the peer-to-peer environment.

Chapter 5 concludes the dissertation.

Chapter 2

Background and Related Work

In this chapter we give some background information about peer-to-peer networks and the indexing and routing mechanisms they use to support search queries. We introduce some terminology we use throughout the thesis which will be helpful in Chapter 3 in understanding the CUP protocol and how it works on top of the routing and indexing mechanisms in place. We then describe previous work related to the CUP protocol, as well as previous load-balancing work in distributed systems.

2.1 Terminology

A peer-to-peer system is a self-organizing distributed system where a group of Internet hosts both provide and receive services from each other in a cooperative effort without distinguished roles as pure clients or pure servers. The hosts form and maintain an overlay network that may constantly change as new hosts enter and existing hosts leave the network. Hosts join the peer-to-peer network to partake in a specific application: for example, to share content (e.g., songs or documents) or to share computation.

Peer-to-peer systems are receiving much attention primarily because of the great number of features they offer applications that are built on top them. These features include: scalability, availability, fault tolerance, decentralized administration, and anonymity.

We next introduce some terminology we use throughout the thesis and explain the basic search mechanism used in peer-to-peer networks:

Node: This is a host participating in the peer-to-peer network. Each node periodically exchanges "keep-alive" messages with its neighbors to confirm their existence and to trigger recovery mechanisms should one of the neighbors fail.

Structured vs. Unstructured Search: Peer-to-peer search algorithms have been classified into three categories: structured, unstructured and hybrid [37, 63]. In a structured search, search queries follow a well-defined path from the querying node to an authority node that holds the index entries pertaining to the query. Examples of structured networks include: CAN, Chord, Pastry, and Tapestry [45, 57, 48, 64].

In an unstructured search, search queries haphazardly travel through the network via flooding or random walks in search of index entries. Examples of such networks include the original Gnutella specification [3] as well as modifications proposed by Lv et al [37].

In structured networks, the number of overlay hops traversed by a query is bounded and each hop traversed by a query brings the query closer to the answer. Structured networks are best suited to handle exact-match search queries where the name or identifier of the content is known in advance. Keyword-based search is possible, but requires several parallel searches for each keyword. Unstructured networks are more flexible in the type of search queries they can handle. Essentially, they allow any kind of query including exact-match queries, keyword-based queries as well as queries over the actual content stored. This flexibility comes at the cost of less efficient search since queries do not have a well-defined path they can take to reach the answer and travel without direction.

In this thesis we primarily focus on structured peer-to-peer networks. We define the remaining terms for structured networks only.

Search Query: A fundamental operation in a peer-to-peer network is search. A search query posted at a node N is a request to locate a peer node in the network that holds particular content of interest. The search query typically consists of the name of the content or a set of keyword attributes describing the content of interest. The response to such a search query is a set of index entries pointing to replica nodes that serve content whose metadata satisfies the query.

Global Index: In structured peer-to-peer networks a hashing scheme maps keys (names of content files or keywords) onto a virtual coordinate space using a uniform hash function that evenly distributes the keys to the space. The coordinate space serves as a global index that stores index entries which are *(key, value)* pairs. The value in an index entry is a pointer (typically an IP address) to the location of a replica that stores the content file associated with the entry's key. There can be several index entries for the same key, one for each replica of the content.

Authority Node: Each node N in a structured peer-to-peer system is dynamically allocated a subspace of the coordinate space (i.e., a partition of the global index) and all index entries mapped into its subspace are owned by N. We refer to N as the authority node of these entries. *Replicas* of content whose key corresponds to an authority node N send birth messages to N to announce they are willing to serve the content. Depending on the application supported, replicas might periodically send refresh messages to indicate they are still serving a piece of content. They might also send deletion messages that explicitly indicate they are no longer serving the content. These deletion messages trigger the authority node to delete the corresponding index entry from its local index directory.

Search/Routing Mechanism: In structured networks, when a node issues a query for key K, the query will be routed along a well-defined path with a bounded number of hops from the querying node to the authority node for K. The routing mechanism is designed so that each node on the path hashes K using the same hash function to choose deterministically which of its neighbors will serve as the next hop.

The CUP protocol, described in Chapter 3 is not tied to the underlying routing mechanism and can be applied in both structured and unstructured networks.

Query Path for Key K: This is the path a search query for key K takes. In a structured network, each hop on the query path is in the direction of the authority node that owns K. If an intermediate node on this path has unexpired entries cached, the path ends at the intermediate node; otherwise the path ends at the authority node. The reverse of this path is the *Reverse Query Path* for key K.

Local index directory: This is the subset of global index entries owned by a node in a structured network.

Cached index entries: This is the set of index entries cached by a node N in the process of handling queries and propagating updates for keys for which N is not the authority. The set of cached index entries and the local index directory are disjoint sets.

Lifetime of index entries: Each index entry cached at a node typically has associated with it a lifetime during which it is considered fresh and after which it is considered expired.

2.2 Previous Index Caching Work

To our knowledge, CUP is the first protocol aimed at maintaining caches of index entries to improve search queries in peer-to-peer networks. While designers of peer-to-peer systems [3, 45, 57, 48, 64] advocate caching index entries to improve performance, there has been little follow-up work studying when and where to cache the entries and how to maintain these cached entries.

Cox et al. study providing DNS service over a peer-to-peer network [23]. They cache index entries which are DNS mappings along search query paths. Similarly, the TerraDir Distributed Directory caching scheme has nodes along the search query path cache pointers to other nodes previously traversed by the query [54]. In each of these examples, cached index entries have expiration times and are not refreshed or maintained until a miss or failure occurs.

Path caching of content in peer-to-peer systems has received more attention. Freenet [18], CFS [24], PAST [47], and Lv et al. [37] each perform path caching, or caching of content along the search path of a query. These studies do not focus on cache maintenance, but rather depend on expiration or cache size constraints to limit the use of stale content.

To maintain caches, CUP periodically sends updates down a conceptual tree whose branches are formed by queries traveling up toward the node (or nodes) holding the answer. CUP trees are similar to application-level multicast trees, particularly those built on peer-to-peer networks. (For a survey of application-level multicast work, see

[16].) These include Scribe [49] built on top of Pastry [48] and Bayeaux built on top of Tapestry [64]. Here, we focus on Scribe. For a survey of application-level multicast work, see [16].

Scribe is a publish-subscribe infrastructure where subscribers interested in a topic join its corresponding multicast group. Scribe creates a multicast tree rooted at the rendez-vous point of each multicast group. Publishers send a message to the rendez-vous point which then transmits the message to the entire group by sending it down the multicast tree. The multicast tree is formed by joining the Pastry routes from each subscriber node to the rendez-vous point. Scribe could apply the ideas that CUP introduces to provide update propagation for cache maintenance in Pastry.

Cohen and Kaplan study the effect that aging through cascaded caches has on the miss rates of web client caches [19]. For each object an intermediate cache refreshes its copy of the object when its age exceeds a fraction v of the lifetime duration. The intermediate cache does not push this refresh to the client; instead, the client waits until its own copy has expired at which point it fetches the intermediate cache's copy with the remaining lifetime. For certain values of v and request inter-arrival distributions, the client cache can suffer from a higher miss rate than if the intermediate cache only refreshed on expiration. A CUP tree could be viewed as a series of cascaded caches in that each node depends on the previous node in the tree for updates to an index entry. The key difference is that in CUP, refreshes are pushed down the entire tree of interested nodes. Therefore, barring communication delays, whenever a parent cache gets a refresh so does the interested child node. In such situations, the miss rate at the child node actually improves.

2.3 Previous Load-Balancing Work

Load-balancing has been the focus of many studies described in the distributed systems literature. We first describe load-balancing techniques that could be applied in a peer-to-peer context. We classify these into two categories, those algorithms where the allocation decision is based on load and those where the allocation decision is based on available capacity. We then describe other load-balancing techniques (such as process migration) that cannot be directly applied in a peer-to-peer context.

2.3.1 Load-Based Algorithms

Of the load-balancing algorithms based on load, a very common approach to performing load-balancing has been to choose the server with the least reported load from among a set of servers. This approach performs well in a homogeneous system where the task allocation is performed by a single centralized entity (dispatcher) which has complete up-to-date load information [61, 62]. In a system where multiple dispatchers are independently performing the allocation of tasks, this approach however has been shown to behave badly, especially if load information used is stale [30, 40, 31, 43, 52]. Mitzenmacher talks about the "herd behavior" that can occur when servers that have reported low load are inundated with requests from dispatchers until new load information is reported [43].

Dahlin proposes *load interpretation* algorithms [25]. These algorithms take into account the age (staleness) of the load information reported by each of a set of distributed homogeneous servers as well as an estimate of the rate at which new requests arrive at the whole system to determine to which server to allocate a request.

Many studies have focused on the strategy of using a subset of the load information available. This involves first randomly choosing a small number, k, of homogeneous servers and then choosing the least loaded server from within that set [42, 30, 58, 10, 33]. In particular, for homogenous systems, Mitzenmacher [42] studies the tradeoffs of various choices of k and various degrees of staleness of load information reported. As the degree of staleness increases, smaller values of k are preferable.

Genova et al. [32] propose an algorithm that first randomly selects k servers. The algorithm then weighs the servers by load information and chooses a server with probability that is inversely proportional to the load reported by that server. When $k = n$, where n is the total number of servers, the algorithm is shown to perform better than previous load-based algorithms and for this reason we focus on this algorithm in this thesis.

As we will see in Chapter 4, algorithms that base the decision on load do not handle heterogeneity.

2.3.2 Available-Capacity-Based Algorithms

Of the load-balancing algorithms based on available capacity, one common approach has been to exclude servers that fail some utilization threshold and to choose from the remaining servers. Mirchandaney et al. [41] and Shivaratri et al. [52] classify machines as lightly-utilized or heavily-utilized and then choose randomly from the lightly-utilized servers. This work focuses on local-area distributed systems. Colajanni et al. use this approach to enhance round-robin DNS load-balancing across a set of widely distributed heterogeneous web servers [22], Specifically, when a web server surpasses a utilization threshold it sends an alarm signal to the DNS system indicating it is out of commission. The server is excluded from the DNS resolution until it sends another signal indicating it is below threshold and free to service requests again. In this work, the maximum capacities of the most capable servers are at most a factor of three that of the least capable servers.

In Chapter 4 we show that when applied in the context of a peer-to-peer network where many nodes are making the allocation decision and where the maximum capacities of the replica nodes can differ by two orders of magnitude, excluding a serving node temporarily from the allocation decision can result in load oscillation.

Another approach is to choose amongst a set of servers based on the available capacity of each server [65] or the available bandwidth in the network to each server [14]. The server with the highest available capacity/bandwidth is chosen by a client with a request. The assumption here is that the reported available capacity/bandwidth will continue to be valid until the chosen server has finished servicing the client's request. This assumption does not always hold; external traffic caused by other applications can invalidate the assumption, but more surprisingly the traffic caused by the application whose workload is being balanced can also invalidate the assumption. We demonstrate this in Chapter 4.

2.3.3 Other Load-balancing Techniques

We now describe load-balancing techniques that appear in the literature but cannot be directly applied in a peer-to-peer context.

There has been a large body of work devoted to the problem of load-balancing across a set of servers residing within a cluster. In some cluster systems there is one centralized dispatcher that has full control over the allocation of requests to servers [31, 27, 2]. In other systems there are multiple dispatchers that make the allocation decision. One common approach is to have front-end servers sit at the entrance of the cluster intercepting incoming requests and allocating requests to the back-end servers within the cluster that actually satisfy the requests [15]. Still others have requests be evenly routed to servers within the cluster via DNS rotation (described below) or via a single IP-switch sitting at the front of the cluster (e.g., [5]). Upon receiving a request each server then decides whether to satisfy the request or to dispatch it to another server [65]. Some cluster systems have the dispatchers(s) poll each server or a random set of servers for load/availability information just before each allocation decision [9, 51]. Others have the dispatcher(s) periodically poll servers, while still others have servers periodically broadcast their load-balancing information. Studies that compare the tradeoffs among these information dissemination options within a cluster include [65, 51].

Regardless of the way this information is exchanged, cluster-based algorithms take advantage of the local-area nature of the cluster network to deliver timely load-balancing updates. This characteristic does not apply in a peer-to-peer network where load-balancing updates may have to travel across the Internet.

Most cluster algorithms assume that servers are homogenous. The exceptions to this rule include work by Castro et al. [15]. This work assumes that servers will have different processing capabilities and allows each server to stipulate a *maximum desirable utilization* that is incorporated into the load-balancing algorithm. The algorithm they use assumes that servers are synchronized and send their load updates at the same time. This is not true in a peer-to-peer network where replicas cannot be synchronized. Zhu et al. [65] assume servers are heterogeneous and use a metric that combines available disk capacity and CUP cycles to choose a server within the cluster to handle a task [65]. Their algorithm uses a combination of random polling before selection and random multicasting of load-balancing information to a select few servers. Both are techniques that would not scale in a large peer-to-peer network.

Another well-studied load-balancing cluster approach is to have the heavily loaded servers handoff requests they receive to other servers within the cluster that are less loaded or to have lightly loaded servers attempt to get tasks from heavily loaded servers (e.g., [26, 53]). This can be achieved through techniques such as HTTP redirection (e.g., [12, 7, 13]) or packet header rewriting (e.g., [8]) or remote script execution [65]. HTTP redirection adds additional client round-trip latency for every rescheduled request. TCP/IP hand-off and packet header rewriting require changes in the OS kernel or network interface drivers. Remote script execution requires trust between the serving entities.

Similar to task handoff is the technique of process migration. Process migration to spread job load across a set of servers in a local-area distributed system has been widely studied both in the theoretical literature as well as the systems literature (e.g., [28, 36, 29, 44, 35]). In these systems overloaded servers migrate some of their currently running processes to lighter loaded servers in an attempt to achieve more equitable distribution of work across the servers.

Both task handoff and process migration require close coordination amongst serving entities that can be afforded in a tightly-coupled communication environment such as a cluster or local-area distributed system. In a peer-to-peer network where the replica nodes serving the content may be widely distributed across the Internet, these techniques are not possible.

A lot of work has looked at balancing load across multi-server homogeneous web sites by leveraging the DNS service used to provide the mapping between a web page's URL and the IP address of a web server serving the URL. Round-robin DNS was proposed, where the DNS system maps requests to web servers in a round-robin fashion [34, 6]. Because DNS mappings have a Time-to-Live (TTL) field associated with them and tend to be cached at the local name server in each domain, this approach can lead to a large number of client requests from a particular domain getting mapped to the same web server during the TTL period. Thus, round-robin

DNS achieves good balance only so long as each domain has the same client request rate. Moreover, round-robin load-balancing does not work in a heterogeneous peer-to-peer context because each serving replica gets a uniform rate of requests regardless of whether it can handle this rate. Work that takes into account domain request rate improves upon round-robin DNS and is described by Colajanni et al. [21].

Colajanni et al. later extend this work to balance load across a set of widely distributed heterogeneous web servers [22]. This work proposes the use of adaptive TTLs, where the TTL for a DNS mapping is set inversely proportional to the domain's local client request rate for the mapping of interest (as reported by the domain's local name server). The TTL is at the same time set to be proportional to the chosen web server's maximum capacity. So web servers with high maximum capacity will have DNS mappings with longer TTLs, and domains with low request rates will receive mappings with longer TTLs. Max-Cap, the algorithm proposed in this thesis, also uses the maximum capacities of the serving replica nodes to allocate requests proportionally. The main difference is that in the work by Colajanni et al., the root DNS scheduler acts as a centralized dispatcher setting all DNS mappings and is assumed to know what the request rate in the requesting domain is like. In the peer-to-peer case the authority node has no idea what the request rate throughout the network is like, nor how large is the set of requesting nodes.

Lottery scheduling is another technique that, like Max-Cap, uses proportional allocation. This approach has been proposed in the context of resource allocation within an operating system (the Mach microkernel) [60]. Client processes hold tickets that give them access to particular resources in the operating system. Clients are allocated resources by a centralized lottery scheduler proportionally to the number of tickets they own and can donate their tickets to other clients in exchange for tickets at a later point. Max-Cap is similar in that it allocates requests to a replica node proportionally to the maximum capacity of the replica node. The main difference is that in Max-Cap the allocation decision is completely distributed with no opportunity for exchange of resources across replica nodes.

Chapter 3

The CUP Protocol

In this chapter we develop in detail a new protocol, CUP, for Controlled Update Propagation in peer-to-peer networks to maintain caches of index entries. We show through extensive experiments that CUP greatly reduces average search query latency with recoverable propagation overhead when compared with Path Caching with eXpiration (PCX).

First, we give an overview of the protocol. Second, we describe the protocol semantics in detail. Third we describe experiments demonstrating the benefits of CUP, and finally we conclude the chapter with summary results.

3.1 CUP Overview

3.1.1 Basic Idea

As described in Chapter 1, CUP is a protocol for Controlled Update Propagation to maintain caches in a peer-to-peer network. CUP asynchronously builds caches of index entries while answering search queries. It then propagates updates of index entries to maintain these caches. CUP is not tied to any particular search mechanism and therefore can be applied in both networks that perform structured search as well as networks that perform unstructured search. In this chapter we extensively study how CUP works on structured peer-to-peer networks.

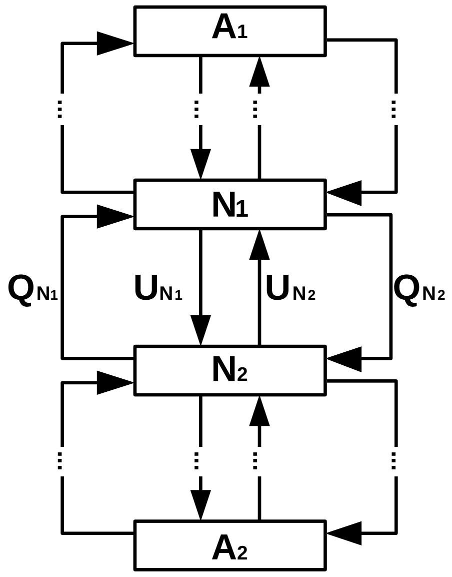

Figure 3.1: CUP Query & Update Channels. A_1 and A_2 are authority nodes. A query arriving at node N_2 for an item for which A_1 is the authority is pushed onto query channel Q_{N_1} to N_1. If N_1 has a cached unexpired entry for the item, it returns it to N_2 through U_{N_1}. Otherwise, it forwards the query towards A_1. Any update for the item originating from authority node A_1 flows downstream to N_1 which may forward it onto N_2 through U_{N_1}. The analogous process holds for queries at N_1 for items for which A_2 is one of the authority nodes.

The basic idea is that every node in the peer-to-peer network maintains two logical channels per neighbor: a query channel and an update channel. The query channel is used to forward search queries for objects of interest to the neighbor that is closest to the authority node holding the entries for those objects. The update channel is used to forward query responses asynchronously to a neighbor and to update index entries that are cached at the neighbor.

Queries for an item travel "up" the query channels of nodes along the path toward the authority node for that item. Updates travel "down" the update channels along the reverse path taken by a query. Figure 3.1 shows this process.

The query and update channels provide several benefits. The query channel enables "query coalescing". If a node receives two or more queries for an item for which it does not have a fresh response, the node pushes only one instance of the query for that item up its query channel. This approach can have significant savings in traffic, because bursts of requests for an item are coalesced into a single request. Second, coalescing multiple queries for the same item solves the "open connection" problem suffered by some peer-to-peer systems. The asynchronous nature of the query channel relieves nodes from having to maintain many separate open connections while waiting for query responses; instead all responses return through the update channel. Through simple bookkeeping (setting an interest bit) the node registers the interest of its neighbors so it knows which of its neighbors to push the query response to when it arrives.

The cascaded propagation of updates from authority nodes down the reverse paths of search queries has many advantages. First, updates extend the lifetime of cached entries allowing intermediate nodes to continue serving queries from their caches without re-issuing new queries. It has been shown that up to fifty percent of content hits at caches are instances where the content is valid but stale and therefore cannot be used without first being re-validated [20]. These occurrences are called *freshness misses.* Second, a node that proactively pushes updates to interested neighbors reduces its load of queries generated by those neighbors. The cost of pushing the update down is recovered by the first query for the same item following the update. Third, the further down an update gets pushed, the shorter the distance subsequent queries need to travel to reach a fresh cached answer. As a result, query response latency is reduced. Finally, updates can help prevent errors. For example, an update to delete an index entry prevents a node from erroneously answering queries using the invalid entry before it expires.

In CUP, nodes decide individually when to receive updates. A node only receives updates for an item if the node has registered interest in that item. Furthermore, each node uses its own incentive-based policy to determine when to cut off its incoming supply of updates for an item. This way the propagation of updates is controlled and does not flood the network.

Similarly, nodes decide individually when to propagate updates to interested neighbors. This is necessary because a node may not always be able or willing to forward updates to interested neighbors. In fact, a node's ability or willingness to propagate updates may vary with its workload. CUP addresses this by introducing an adaptive mechanism each node uses to regulate the rate of updates it propagates downstream. A salient feature of CUP is that even if a node's capacity to push updates becomes zero, nodes dependent on the node for updates fall back to the case of PCX and incur no overhead.

In this chapter, we compare CUP against PCX with coalescing under typical workloads that have been observed in measurements of real peer-to-peer networks. In PCX, when a search query is posted at a node N, the set of index entries returned in response to the query will be cached with an expiration time along the reverse query path down to N. We show that CUP significantly reduces the average query latency over PCX by as much as an order of magnitude. Moreover, CUP overhead is more than compensated for by its savings in cache misses. The cost of saved misses can be two to 200 times the cost of updates pushed.

3.1.2 CUP Node Bookkeeping

At each node, index entries are grouped together by key. For each key K, the node stores a flag that indicates whether the node is waiting to receive an update for K in response to a query, and an interest bit vector. Each bit in the vector corresponds to a neighbor and is set or clear depending on whether that neighbor is or is not interested in receiving updates for K.

Each node tracks the popularity or request frequency of each non-local key K for which it receives queries. The popularity measure for a key K can be the number of queries for K a node receives between arrivals of consecutive updates for K or a rate of queries of a larger moving window. On an update arrival for K, a node uses its popularity measure to re-evaluate whether it is beneficial to continue caching and receiving updates for K. We elaborate on this cut-off decision in Section 3.3.4.

Node bookkeeping in CUP involves no network overhead and a few tens of megabytes for hundreds of thousands of entries. With increasing CPU speeds and memory sizes, this bookkeeping is negligible when we consider the reduction in query latency achieved.

3.2 CUP Protocol Specification

3.2.1 Handling Queries

Upon receipt of a query for a key K, there are three basic cases to consider. In each of the cases, the node updates its popularity measure for K and sets the appropriate bit in the interest bit vector for K if the query originates from a neighbor. Otherwise, if the query is from a local client, the node maintains the connection until it can return a fresh answer to the client. To simplify the protocol description we use the phrase "push the query" to indicate that a node pushes a query upstream toward the authority node. We use the phrase "push the update" to indicate that a node pushes an update downstream in the direction of the reverse query path.

Case 1: Fresh Entries for key K are cached. The node uses its cached entries for K to push the response to the querying neighbor or local client.

Case 2: Key K is not in cache. The node adds K to its cache and marks it with a *Pending-Response* flag. The flag's purpose is to coalesce bursts of queries for K into one query. A subsequent query for K will be suppressed since the node is already waiting the response for the first query of the burst. Query coalescing results in significant network savings, for both PCX and CUP. In some of the workloads, coalesced queries can form up to 90 percent of the total number of queries that miss.

With every query push, a timer is set so that if the query response is lost, the node pushes up another query.

Case 3: All cached entries for key K have expired. The node must obtain the fresh index entries for K. If the *Pending-Response* flag is set, the node does not need to push the query; otherwise, the node sets the flag and pushes the query.

3.2.2 Update Types

We classify updates into three categories: deletes, refreshes, and appends. Deletes, refreshes, and appends originate from the replicas of a piece of content and are directed toward the authority node that owns the index entries for that content.

Deletes are directives to remove a cached index entry. Deletes can be triggered by two events: 1) a replica sends a message indicating it no longer serves a piece of content to the authority node that owns the index entry pointing to that replica. 2) the authority node notices a replica has stopped sending "keep-alive" messages and assumes the replica has failed. In either case, the authority node deletes the corresponding index entry from its local index directory and propagates the delete to interested neighbors.

Refreshes are directive messages that extend the lifetimes of cached index entries. Refreshes that arrive at a cache do not prevent errors as deletes do, but help prevent freshness misses.

Finally, appends are directives to add index entries for new replicas of content. These updates help alleviate the demand for content from the existing set of replicas since they add to the number of replicas from which clients can download content.

3.2.3 Handling Updates

A key feature of CUP is that a node does not forward an update for K to its neighbors unless those neighbors have registered interest in K. Therefore, with some light bookkeeping, we prevent unwanted updates from wasting network bandwidth.

Upon receipt of an update for key K there are three cases to consider.

Case 1: Pending-Response flag is set. This means that the update is a query response carrying a set of index entries in response to a query, the node stores the index entries in its cache, clears the *Pending-Response* flag, and pushes the update to neighbors whose interest bits are set and to local client connections open at the node.

Case 2: Pending-Response flag is clear. If all the interest bits for K are clear, the node decides whether it wants to continue receiving updates for K. The node bases its decision on K's popularity measure. Each node uses its own policy for deciding whether the popularity of a key is high enough to warrant receiving further updates for it. If the node decides K's popularity is low, it pushes a *Clear-Bit* control message to the sender of the update to notify it that is no longer interested in K's updates. Otherwise, if the popularity is high or some of the neighbor's interest bits are set, the node applies the update to its cache and pushes the update to those neighbors.

Note that a greedy or selfish node can choose not to push updates for a key K to interested neighbors. This forces downstream nodes to fall back to PCX for K. However, by choosing to cut off downstream propagation, a node runs the risk of receiving subsequent queries from its neighbors costing it twice as much, since it must both receive and respond to the queries. Therefore, although each node has the choice of stopping the update propagation at any time, it is in its best interest to push updates for which there are interested neighbors.

Case 3: Incoming update has expired. This could occur when the network path has long delays and the update does not arrive in time. The node does not apply the update and does not push it downstream. If the *Pending-Response* flag is set then the node re-issues another query for K and pushes it upstream.

3.2.4 Handling Clear-Bit Messages

A *Clear-Bit* control message is pushed by a node to indicate to its neighbor that it is no longer interested in receiving updates for a particular key from that neighbor.

When a node receives a *Clear-Bit* message for key K, it clears the interest bit for the neighbor from which the message was sent. If the node's popularity measure for K is low and all of its interest bits are clear, the node also pushes a *Clear-Bit* message for K. This propagation of *Clear-Bit* messages toward the authority node for K continues until a node is reached where the popularity of K is high or where at least one interest bit is set.

Clear-Bit messages can be piggybacked onto queries or updates intended for the neighbor, or if there are no pending queries or updates, they can be pushed separately.

3.2.5 Adaptive Control of Update Push

Ideally every node would propagate all updates to interested neighbors to save itself from having to handle future downstream misses. However, from time to time, nodes are likely to be limited in their capacity to push updates downstream. Therefore, we introduce an adaptive control mechanism that a node uses to regulate the rate of pushed updates.

We assume each node N has a capacity U for pushing updates that varies with N's workload, network bandwidth, and/or network connectivity. N divides U among its outgoing update channels such that each channel gets a share that is proportional to the length of its queue. This allocation maintains queues of roughly equal size. The queues are guaranteed to be bounded by the expiration times of the entries in the queues. So even if a node has its update channels completely shut down for a long period, all entries will expire and be removed from the queues.

Under a limited capacity and while updates are waiting in the queues, each node can re-order the updates in its outgoing update channels by pushing ahead updates that are likely to have greater benefit. For example, a node can re-order refreshes and appends so that entries that are closer to expiring are given higher priority. Such entries are more likely to cause freshness misses which in turn trigger a new search query.

The strategy for re-ordering depends on the application. For example, in an application where query latency and accuracy are of the most importance, one can push updates in the following order: deletes, refreshes, and appends. In an application subject to flash crowds [56] that query for a particular item, appends might be given higher priority over the other updates. This would help distribute the load faster across a larger set of replicas. For all strategies, during the re-ordering any expired updates are eliminated.

3.2.6 Node Arrivals and Departures

The peer-to-peer model assumes that participating nodes will continuously join and leave the network. CUP must be able to handle both node arrivals and departures seamlessly.

Arrivals. When a new node N enters a structured peer-to-peer network, it becomes responsible for a portion of another node M's share of the global index and becomes the authority node for those index entries mapped into that portion. N, M, and all surrounding affected nodes (old neighbors of M) update the bookkeeping structures they maintain for indexing and routing purposes. This is a necessary part of maintaining the connectivity of any structured peer-to-peer network when the set of nodes in the network changes. management.

For CUP, the issues at hand are updating the interest bit vectors of the affected nodes and deciding what to do with the index entries stored at M. This may require bit vector translation. For example, if a node that previously had M as its neighbor now has N as its neighbor, the node must make the bit ID that pointed to M now point to N.

To deal with its stored index entries, M could simply not hand over any of its entries to N. This would cause entries at some of M's previous neighbors to expire and subsequent queries from those nodes will restart update propagations from N. Alternatively, M could give a copy of its stored index entries to N. Both N and M would then go through each entry and patch its bit vector. Both solutions are viable. The first solution requires no bit translation but temporarily loses the CUP update benefits and behaves like PCX for the untransferred entries. The second solution gets the benefits of transferring the entries, at the expense of transferring the index entries and performing the bit vector patching. The metadata and bit vectors for thousands of index entries can be compressed into a few kilobytes and can be piggybacked onto messages that are already being exchanged to reconfigure the topology. Once the transfer occurs, the bit vector patching is an in-memory, local operation that with today's CPU and memory capacities takes only a few seconds for a few million entries.

Departures. Node departures can be either graceful (planned) or ungraceful (due to sudden failure of a node or its communication link). In either case the index mechanism in place dictates that a neighboring node M take over the departing node N's portion of the global index. To support CUP, the interest bit vectors of all affected nodes must be patched to reflect N's departure.

If N leaves gracefully, N can choose not to hand over to M its index entries. Any entries at surrounding nodes that were dependent on N to be updated will simply expire and subsequent queries will restart update propagations. Again, alternatively N may give M its set of entries. M must then merge its own set of index entries with N's, by eliminating duplicate entries and patching the interest bit vectors as necessary. If N's departure is due to a failure, there can be no hand-over of entries and all entries in the affected neighboring nodes will expire as in PCX.

3.3 Evaluation

The goal of CUP is to extend and multiply the benefits of PCX. In doing so, there are two key performance questions to address. First, by how much does CUP reduce the average query latency? Second, how much overhead does CUP incur in providing this reduction?

We first define the notion of a CUP tree. We use this definition to present a cost model based on economic incentive used by each node to determine when to cut off the propagation of updates for a particular key. We give a simple analysis of how the cost per query is reduced (or eliminated) through CUP. We then describe our experimental results comparing the performance of CUP with that of PCX.

3.3.1 CUP Trees

We define two kinds of CUP trees: *Real CUP Trees* and *Spanning CUP Trees*.

Figure 3.2 shows a snapshot of CUP in progress. The left half of each node shows the set of keys for which the node is the authority. The right half shows the set of keys for which the node has cached index entries as a result of handling queries. For example, node C owns K1 and K2 and has cached entries for K3, K4, and K5.

The process of querying for a key K and updating cached index entries pertaining to K forms a tree which we refer to as the *Real CUP Tree*. This tree, denoted R(A,K), is similar to an application-level multicast tree and has as its root the authority node A for K. The exact structure of R(A,K) depends on the actual workload of queries for K. The branches of the tree are formed by the paths traveled by queries from other nodes in the network. For example, in Figure 3.2, the tree R(C,K1) has grown branch

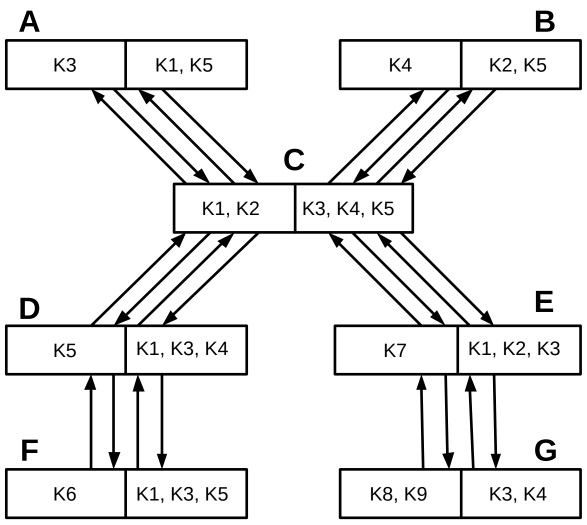

Figure 3.2: CUP Trees

{F, D, C} as the result of a query for K1 at node F. Updates for K1 originate at the root (authority node) C and travel down the tree to interested nodes A, D, E, and F. The entire workload of queries for all keys results in a collection of criss-crossing Real CUP Trees with overlapping branches.

We define the *Spanning CUP Tree* for key K, S(A,K) as the tree that contains all possible query paths for K. This is the tree that would be generated by issuing a query for K from every node in the peer-to-peer network. For example, in Figure 3.2, S(C,K1) is rooted at C (level 0), has nodes A, B, D, E at level 1, and nodes F and G at level 2.

3.3.2 Cost Model

Consider a node N within spanning tree S(A,K) that is at distance D from A. We define the cost per query for K at N as the number of hops in the peer-to-peer network that must be traversed to return an answer to N. When a query for K is posted at N for the first time, it travels toward A. If none of the nodes between N and A have a fresh response cached, the cost of the query at N is $2D$: D hops up and D hops for the response to travel down. If a node on the query path has a fresh answer cached, the cost is less than $2D$. Subsequent queries for K at N that occur within the lifetime of the entries now cached at N have a cost of zero. As a result, caching at intermediate nodes can significantly lower average query latency.

We can gauge the performance of CUP by calculating the percentage of updates CUP propagates that are "justified", i.e., those whose cost is recovered by a subsequent query. Updates for popular keys are likely to be justified more often than updates for less popular keys.

A refresh update is justified if a query arrives sometime between the previous expiration of the cached entry and the new expiration time supplied by the refresh update. An append update is justified if at least one query arrives between the time the append is performed and the time of its expiration. Finally, a deletion update is considered justified if at least one query arrives between the time the deletion is performed and the expiration time of the entry to be deleted.

For each update, let T be the critical time interval described above during which a query must arrive in order for the update to be justified. Consider a node N at distance D from A in R(A,K). An update propagated down to N is justified if at least one query Q is posted within T time units at any of the nodes of the spanning subtree S(N,K).

Given the distribution of query arrivals at each node in the tree S(N,K), we can find the probability that the update at N is justified by calculating the probability that a query will arrive at some node in S(N,K). If the queries for K arrive at each node N_i in S(N,K) according to a Poisson process with parameter λ_i, then it follows that queries for K arrive at S(N,K) according to a Poisson process with parameter Λ equal to the sum of all $\lambda's$. Therefore, the probability that a query for K will arrive within T time units is $1 - e^{-\Lambda T}$ and equals the probability that the update pushed to N is justified. The closer to the authority N is, the higher the Λ and thus the higher the probability for an update pushed to N to be justified. For $\Lambda = 1$ query arrival per second and $T = 6$ seconds, the probability that an update arriving at N is justified is 99 percent.

The benefit of a justified CUP update goes beyond just recovery of its cost. For each hop a justified update u is pushed down to the root N of subtree S(N,K), exactly one hop is saved since without the propagation, the first subsequent query landing at a node N_i in S(N,K) within T time units will cause two hops, from N to its parent and back. This halves the number of hops traveled between N and its parent which in turn reduces query latency. In fact all subsequent queries posted somewhere in S(N,K) within T time units will benefit from N receiving u. The cumulative benefit an update u brings to subtree S(N,K) increases when N is closer to the authority nodes since there is a higher probability that queries will be posted within S(N,K). We define "investment return" as the cumulative savings in hops achieved by pushing a justified update to node N. The experiments show that the return is large even when CUP's reduction in latency is modest and is substantially large when the latency reduction is high.

High investment return in some nodes, especially those close to the authority nodes may provide enough benefit margin for more aggressive CUP strategies. For example, a more aggressive strategy would be to push some updates even if they are not justified. As long as the number of justified updates is at least fifty percent the total number of updates pushed, the overall update overhead is completely recovered. Therefore, if network load is not the prime concern, an "all-out" push strategy achieves minimum latency.

3.3.3 Experiment Setup and Metrics

CUP can be viewed as extending the caching benefits of PCX. Therefore, we evaluate CUP by comparing it with PCX. We perform our simulation experiments using a wide range of parameters based on measurements of real peer-to-peer workloads [39, 50, 11, 37, 55].

For our experiments, we simulate a content-addressable network (CAN) [45] using the Stanford Narses simulator [38]. Again, we stress that CUP is independent of the specific search mechanism used by the peer-to-peer network and can be used as a cache maintenance protocol in any peer-to-peer network.

As in previous studies (e.g., [45, 57, 47, 17, 49, 48]), we measure both CUP and PCX performance in terms of the number of hops traversed in the overlay network.

We justify this metric shortly. *Miss cost* is the total number of hops incurred by all misses, i.e. freshness and first-time misses. CUP overhead is the total number of hops traveled by all updates sent downstream plus the total number of hops traveled by all clear-bit messages upstream. (We assume clear-bit messages are not piggybacked onto updates. This somewhat inflates the overhead measure.) *Total cost* is the sum of the *miss cost* and all overhead hops incurred. Note that in PCX, the *total cost* is equal to the *miss cost. Average query latency* is the average number of hops a query must travel to reach a fresh answer plus the number of hops the answer must travel downstream to reach the node where the query was posted. (For coalesced queries we count the number of hops each coalesced query waits until the answer arrives.) Thus, the average latency is over all queries, including hits, coalesced and non-coalesced misses.

Measuring average query latency in overlay hops allows us to fairly evaluate CUP relative to PCX. A hop traversal avoided by CUP clearly translates into savings in actual traversal time. Since peer nodes vary in their network connectivities and thus network link delays, the question is how does CUP's reduction in overlay hops traversed translate to reduction in actual-clock query latency? We claim that for increasingly large network sizes, an X percent CUP reduction in average hops traveled per query translates into an X percent CUP reduction in average actual traversal time per query. The reasoning for this is as follows. Each overlay hop corresponds to some amount of network delay. By examining all pairs of adjacent neighbors in the overlay network, we can find the distribution of delays between adjacent neighbors. For increasingly large networks, if we reduce the overlay hop traversal by X percent, the distribution of neighbor delays for the remaining hops will continue to be the same. Therefore, the average query latency in actual clock time should be roughly reduced by X percent as well.

We compute investment return (IR) as the overall ratio of saved miss cost to overhead incurred by CUP:

$$IR = \frac{MissCost_{PCX} - MissCost_{CUP}}{OverheadCost_{CUP}}$$

Thus, as long as IR is greater than 1, CUP fully recovers its cost.

The simulation takes as input the number of nodes in the overlay peer-to-peer network, the number of keys owned per node, the distribution of queries for keys, the distribution of query inter-arrival times, the number of replicas per key, the lifetime of replicas in the system, and the fraction of the replica lifetime remaining at which refreshes are pushed out from the authority node. We present experiments for n = 2^k nodes where k ranges from 7 to 14. After a warm-up period for allowing the peer-to-peer network to connect, the measured simulation time is 3000 seconds. We present results for experiments with index entry lifetimes of five minutes to reflect the dynamic nature of peer-to-peer networks where it is prudent to assume nodes might only serve content for a few minutes at a time [50]. Refreshes of index entries occur one minute before expiration. Since both Poisson and Pareto query inter-arrival distributions have been observed in peer-to-peer environments [37, 39], we present experiments for both distributions. Nodes are randomly selected to post queries. We also present experiments where queries are posted at particular "hot spots" in the network.

We present seven sets of experiments. First, we compare the effect on CUP performance of different incentive-based cut-off policies and compare the performance of these policies to those of PCX. Second, using the best cut-off policy of the first experiment, we study how CUP performs as we scale the network. Third, we study the effect on CUP performance of varying the topology of the network by increasing the average node degree, thus decreasing the diameter of the network. Fourth, we study the effect on CUP performance of limiting the outgoing update capacities of nodes. Fifth, we study how CUP performs when queries arrive in bursts, as observed with Pareto inter-arrivals. Sixth, we study how CUP performs when there are hot spots of querying nodes in the network. These six experiments show the per-key benefits of CUP according to the query rates observed by each key. In the last experiment, we show the overall benefits of CUP when keys are queried for according to a Zipf-like distribution.

3.3.4 Varying the Cut-Off Policies

As discussed in Section 3.3.2, the propagation of updates is beneficial only if the updates are justified; when a node's incentive to receive updates for a particular key fades, continuing to propagate updates to that node simply wastes network bandwidth. Therefore, each node needs an independent and decentralized way of controlling its intake of updates.

We base a node's incentive to receive updates for a key on the *popularity* of the key at the node. The more popular a key is, the more incentive there is to receive updates for that key, because the more likely it is that updates for that key will be justified. For a key K, the popularity is the number of queries a node has received for K since the last update for K arrived at the node. (Note that the popularity metric is node-dependent and could be defined in another way such as with a moving average of query arrivals for K.)

We examine two types of thresholds against which to test a key's popularity when making the cut-off decision: probability-based and log-based.

A probability-based threshold uses the distance of a node N from the authority node A to approximate the probability that an update pushed to N is justified. Per our cost model of section 3.3.2, the further N is from A, the less likely an update at N will be justified. We examine two such thresholds, a linear one and a logarithmic one. With a linear threshold, if an update for key K arrives at a node at distance D and the node has received at least αD queries for K since the last update, then K is considered popular and the node continues to receive updates for K. Otherwise, the node cuts off its intake of updates for K by pushing up a clear-bit message. The logarithmic popularity threshold is similar. A key K is popular if the node has received $\alpha \lg(D)$ queries since the last update. The logarithmic threshold is more lenient than the linear in that it increases at a slower rate as we move away from the root.

A log-based threshold is one that is based on the recent history of the last n update arrivals at the node. If within n updates, the node has not received any queries, then the key is not popular and the node pushes up a clear-bit message. A specific example of a log-based policy is the "second-chance policy", $n = 2$. When an update arrives, if no queries have arrived since the last update, the policy gives

Table 3.1: Total cost per key per query rate for varying cut-off policies.

Policy	1 q/s Total Cost	10 q/s Total Cost	100 q/s Total Cost	1000 q/s Total Cost
PCX	61568 (1.00)	154502 (1.00)	476420 (1.00)	2296869 (1.00)
Linear, $\alpha = 0.10$	41281 (0.67)	34311 (0.22)	47132 (0.10)	196650 (0.09)
Linear, $\alpha = 0.01$	31110 (0.51)	24697 (0.16)	48330 (0.10)	196797 (0.09)
Logarithmic, $\alpha = 0.25$	30683 (0.50)	24695 (0.16)	48330 (0.10)	196797 (0.09)
Logarithmic, $\alpha = 0.10$	30683 (0.50)	24695 (0.16)	48330 (0.10)	196797 (0.09)
Second-chance	16958 (0.28)	23702 (0.15)	48330 (0.10)	196797 (0.09)
Optimal push level	15746 (0.26)	23696 (0.15)	45325 (0.095)	153309 (0.07)

the key a "second chance" and waits for the next update. If at the next update, still no queries for K have been received, the node pushes a clear-bit message. The philosophy behind this policy is that pushing these two updates down from the node's parent costs the same as one query miss occurring at the node, since a query miss incurs one hop up to the parent and one hop down. This means that just one query arriving at the node between the two updates is enough to recover the propagation cost.

Table 3.1 compares the total cost of PCX with CUP using the linear and logarithmic polices for various α values, with CUP using second chance, and with a version of CUP that does not use any cut-off policy but instead pushes updates until the optimal push level is reached. To determine the optimal push level we make CUP propagate updates to all querying nodes that are at most p hops from the authority node. By varying the push level p, we determine the level which achieves minimum total cost. This is shown by the row labeled optimal push level and used as a baseline against which to compare PCX and CUP with the cut-off policies described.

In Table 3.1 we show cut-off policy results for a network of 2^{10} nodes and Poisson λ rates of 1, 10, 100 and 1000 queries per second. In each table entry, the first number is the total cost and the number in the parentheses is the total cost normalized by the total cost for PCX. First, we see that regardless of the cut-off policy used, CUP outperforms PCX. Second, for the lower query rates, the performance of the linear and the logarithmic policies is greatly affected by the choice of parameter α, whereas for the higher query rates, the choice of α is less dramatic. These results show that choosing a priori an α value for the linear and logarithmic policies that will perform

Table 3.2: Per-key Comparison of CUP with PCX for varying network sizes, Poisson arrivals of 1 query/second.

Network Size	**128**	**256**	**512**	**1024**
CUP / PCX MissCost	0.10	0.10	0.15	0.17
PCX AvgLat (σ)	1.51 (2.8)	2.67 (4.0)	4.49 (5.9)	6.74 (8.3)
CUP AvgLat (σ)	0.21 (1.1)	0.46 (1.6)	1.25 (3.2)	2.17 (4.4)
IR	4.15	4.88	6.29	7.83

Table 3.3: Per-key Comparison of CUP with PCX for varying network sizes, Poisson arrivals of 1 query/second.

Network Size	**2048**	**4096**	**8192**	**16384**
CUP / PCX MissCost	0.19	0.22	0.20	0.21
PCX AvgLat (σ)	11.01 (12.1)	17.47 (17.5)	29.29 (27.8)	45.56 (40.3)
CUP AvgLat (σ)	4.18 (7.1)	7.70 (11.3)	11.48 (15.1)	19.17 (23.7)
IR	11.43	16.14	24.85	35.98

well across all workloads is difficult.

For the higher query rates, the log-based second-chance policy performs comparably to the probability-based policies, and for the lower query rates outperforms the probability-based policies. In fact, across all rates, the second-chance policy achieves a total cost very near the optimal push level total cost. This is because, unlike the probability-based policies, the second-chance policy adapts to the timing of the queries within the workload in a manner that is independent of the distance of the node. In all remaining experiments, we use second-chance as the cut-off policy.

3.3.5 Scaling the Network

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

Tables 3.2 and 3.3 compare CUP and PCX for network sizes between 128 and 16384 nodes for a Poisson λ rate of 1 query per second. The first row shows the CUP miss cost as a fraction of the PCX miss cost. The second and third rows show

Table 3.4: Per-key Comparison of CUP with PCX for varying network sizes, Poisson arrivals of 10 queries/second.

Network Size	**128**	**256**	**512**	**1024**
CUP / PCX MissCost	0.08	0.09	0.09	0.08
PCX AvgLat (σ)	0.37 (1.58)	0.87 (2.89)	2.28 (5.72)	4.21 (8.78)
CUP AvgLat (σ)	0.03 (0.44)	0.09 (0.96)	0.26 (2.14)	0.47 (3.17)
IR	6.79	8.30	11.76	13.00

Table 3.5: Per-key Comparison of CUP with PCX for varying network sizes, Poisson arrivals of 10 queries/second.

Network Size	**2048**	**4096**	**8192**	**16384**
CUP / PCX MissCost	0.09	0.09	0.10	0.11
PCX AvgLat (σ)	7.48 (13.35)	14.42 (21.37)	25.87 (32.73)	43.85 (48.33)
CUP AvgLat (σ)	1.14 (5.74)	2.53 (9.93)	5.26 (16.23)	9.97 (25.09)
IR	14.89	21.52	30.88	50.52

Table 3.6: Per-key Comparison of CUP with PCX for varying network sizes, Poisson arrivals of 100 queries/second.

Network Size	**128**	**256**	**512**	**1024**
CUP / PCX MissCost	0.08	0.08	0.09	0.08
PCX AvgLat (σ)	0.16 (1.07)	0.33 (1.87)	0.91 (3.65)	1.77 (5.99)
CUP AvgLat (σ)	0.01 (0.28)	0.03 (0.53)	0.08 (1.14)	0.14 (1.73)
IR	28.41	29.05	39.80	39.96

Table 3.7: Per-key Comparison of CUP with PCX for varying network sizes, Poisson arrivals of 100 queries/second.

Network Size	**2048**	**4096**	**8192**	**16384**
CUP / PCX MissCost	0.10	0.08	0.09	0.10
PCX AvgLat (σ)	3.91 (10.79)	8.60 (18.65)	17.94 (30.63)	35.96 (49.28)
CUP AvgLat (σ)	0.36 (3.51)	0.76 (5.92)	2.06 (11.98)	5.50 (21.89)
IR	44.08	52.62	60.48	83.34

Table 3.8: Per-key Comparison of CUP with PCX for varying network sizes, Poisson arrivals of 1000 query/second.

Network Size	**128**	**256**	**512**	**1024**
CUP / PCX MissCost	0.08	0.08	0.08	0.08
PCX AvgLat (σ)	0.12 (0.88)	0.23 (1.42)	0.54 (2.54)	0.92 (3.89)
CUP AvgLat (σ)	0.01 (0.26)	0.02 (0.41)	0.05 (0.75)	0.07 (1.10)
IR	202.45	185.99	226.84	192.11

Table 3.9: Per-key Comparison of CUP with PCX for varying network sizes, Poisson arrivals of 1000 query/second.

Network Size	**2048**	**4096**	**8192**	**16384**
CUP / PCX MissCost	0.12	0.08	0.09	0.09
PCX AvgLat (σ)	1.90 (6.77)	4.01 (11.67)	8.55 (19.93)	18.36 (34.40)
CUP AvgLat (σ)	0.16 (2.05)	0.34 (3.59)	0.79 (6.55)	2.22 (12.29)
IR	192.01	207.12	221.55	264.80

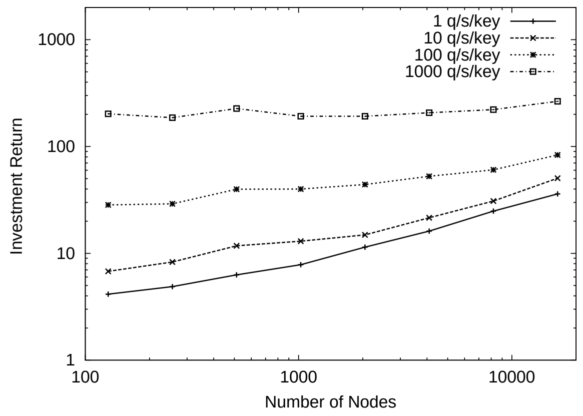

Figure 3.3: IR vs. net size. (Log-scale axes.)

the average query latency in hops for PCX and CUP respectively. The number in parentheses is the standard deviation. As can be observed, CUP reduces average query latency respectively by 9.77, and 17.81, and 26.39 hops for the 4096, 8192, and 16384 node networks. This shows a substantial reduction in average query latency that improves with increasing network size. Comparing the standard deviations of CUP and PCX we see that PCX also has more variability around its average query latency than CUP does.

The fourth row in the tables shows the IR per overhead push performed by CUP. We observe a growth in the rate of return with returns of 16.14, 24.85, and 35.98 for the last three network sizes. These numbers are significant, especially considering the overhead investment is completely recovered.

Tables 3.4-3.9 show the corresponding tables for λ rates of 10, 100, and 1000 queries per second.

To get a quick idea of how CUP performs across all four query rates, Figure 3.3 shows the IR of CUP versus network size for Poisson with $\lambda = 1$, 10, 100, and 1000 queries per second. From the figure we see that for a particular network size, if we increase the query rate the IR increases, and for a particular query rate, if we increase the network size, the IR also increases. This demonstrates that CUP scales to higher query rates and higher network sizes.

3.3.6 Varying the Network Topology

In general, different peer-to-peer networks exhibit different topologies and thus different network diameters. The particular topology created depends on the protocol the peer nodes use to join the network and to keep it connected. The CAN (Content-Addressable Network) design is based on a d-dimensional coordinate space, with our experiments thus far having been for $d = 2$. Increasing the number of dimensions results in a topology where nodes have higher degree and the network has smaller diameter. Smaller diameter implies that the average path length of a query on a miss is shorter for both PCX and CUP. Shorter query paths means that PCX miss latencies decrease, which implies that the benefits of CUP may be less pronounced. On the other hand, CUP total update cost also decreases since there will be shorter distances for updates to travel. As a result, we find that CUP continues to provide significant savings in terms of both overall total cost, latency reduction, and IR per overhead push.

In this set of experiments we study the effect of increasing the number of CAN dimensions on a network with 1024 nodes. The dimensions chosen for this experiment are 2, 3, 5, and 10. These dimensions result in network diameters of 24, 12, 8, and 8 respectively. (For a network of 1024 nodes, increasing beyond five dimensions does not reduce the network diameter any further.) The queries arrive according to a Poisson process with λ rate of 1, 10, 100, and 1000 queries per second for a network of 1024 nodes. Figure 3.4 shows the IR versus the query rate for each dimension. From the figure we see that the curves for dimensions 5 and 10 are very similar because they have equal network diameters. We also see that dimension 2 achieves the highest IR across all query rates, and that the IR decreases with dimension. However, even for the higher dimensions (5 and 10), the IR per overhead hop is at least 2.1 per overhead hop for 1 q/s and increases to 36.6 per overhead hop for 1000 q/s.

In Tables 3.10-3.13, for these dimensions, we show the CUP miss cost as a fraction of PCX miss cost, the CUP miss latency, the PCX miss latency, and the ratio of saved misses to CUP update cost.

Table 3.10: Comparison of CUP with PCX for varying dimensions

Average Rate (q/s)	**1**	**1**	**1**	**1**
Number of dimensions)	**10**	**5**	**3**	**2**
CUP / PCX MissCost	0.33	0.33	0.26	0.17
PCX AvgLat (σ)	2.09 (2.27)	2.21 (2.38)	3.14 (3.49)	6.74 (8.25)
CUP AvgLat (σ)	1.04 (1.53)	1.10 (1.59)	1.37 (2.22)	2.17 (4.37)
IR	2.01	2.08	3.26	7.83

Table 3.11: Comparison of CUP with PCX for varying dimensions

Average Rate (q/s)	**10**	**10**	**10**	**10**
Number of dimensions)	**10**	**5**	**3**	**2**
CUP / PCX MissCost	0.09	0.09	0.09	0.08
PCX AvgLat (σ)	0.94 (1.80)	0.95 (1.82)	1.51 (3.15)	4.21 (8.78)
CUP AvgLat (σ)	0.11 (0.68)	0.11 (0.67)	0.17 (1.14)	0.47 (3.17)
IR	3.03	3.06	4.78	13.00

Table 3.12: Comparison of CUP with PCX for varying dimensions

Average Rate (q/s)	**100**	**100**	**100**	**100**
Number of dimensions)	**10**	**5**	**3**	**2**
CUP / PCX MissCost	0.08	0.08	0.09	0.08
PCX AvgLat (σ)	0.31 (1.41)	0.30 (1.37)	0.61 (2.50)	1.77 (5.99)
CUP AvgLat (σ)	0.03 (0.40)	0.03 (0.42)	0.06 (0.78)	0.14 (1.73)
IR	7.16	6.75	13.82	39.96

Table 3.13: Comparison of CUP with PCX for varying dimensions

Average Rate (q/s)	**1000**	**1000**	**1000**	**1000**
Number of dimensions)	**10**	**5**	**3**	**2**
CUP / PCX MissCost	0.08	0.08	0.08	0.08
PCX AvgLat (σ)	0.16 (0.97)	0.14 (0.90)	0.30 (1.59)	0.92 (3.89)
CUP AvgLat (σ)	0.01 (0.28)	0.01 (0.25)	0.02 (0.46)	0.07 (1.10)
IR	36.68	31.46	66.81	192.11

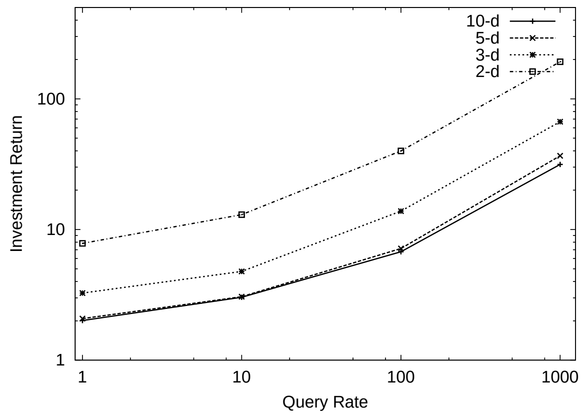

Figure 3.4: IR vs. query rate, varying dimensions. (Log-scale axes.)

3.3.7 Varying Outgoing Update Capacity

Our experiments thus far show that CUP outperforms PCX under conditions where all nodes have full outgoing update capacity. A node with full outgoing capacity is a node that can and does propagate all updates for which there are interested neighbors. In reality, an individual node's outgoing capacity will vary with its workload, network connectivity, and willingness to propagate updates. In this section we study the effect on CUP performance of reducing the outgoing update capacity of nodes.

We present an experiment run on a network of 1024 nodes. In this experiment, after a five minute warm up period, we randomly select twenty percent of the nodes and reduce their outgoing capacity to a fraction of their full capacity. These nodes operate at reduced capacity for ten minutes after which they return to full capacity. After another five minutes for stabilization, we randomly select another set of twenty percent of the nodes and reduce their capacity for ten minutes. We proceed this way for the entire 3000 seconds during which queries are posted, so capacity loss occurs three times during the simulation.

Figure 3.5 shows the ratio of CUP total cost to PCX total cost versus reduced capacity c for this experiment and for four different Poisson query rates λ. The capacity reduction c ranges from 0, implying that no updates are propagated to 1 in which nodes have full outgoing capacity. $c = .25$ means that a node is only capable/willing of pushing out one-fourth the updates it receives.

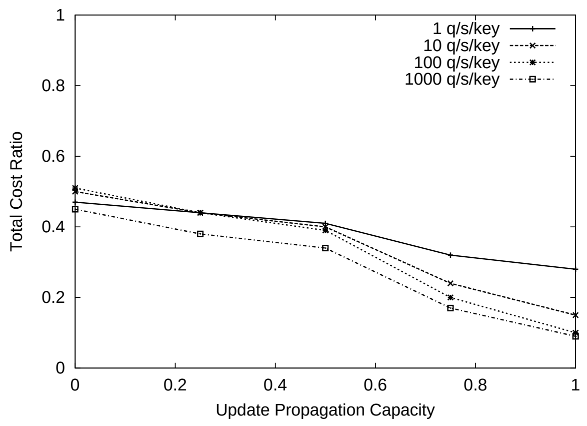

Figure 3.5: Total cost ratio vs. update propagation capacity

Note that even when one fifth of the nodes do not propagate any updates, the total cost incurred by CUP is about half that of PCX. As the outgoing capacity increases, the total cost decreases smoothly until $c = 1$ where CUP achieves its full potential. A key observation from these experiments is that CUP's performance degrades gracefully as the capacity c decreases. This is because reduction in update propagation also results in reduction of its associated overhead. Therefore, the capacity reduction should be seen as a missed opportunity for higher returns rather than as an overall loss. Clearly though, CUP achieves its full potential when all nodes have maximum propagation capacity.

3.3.8 Pareto Query Arrivals

Recent work has observed that in some peer-to-peer networks, query inter-arrivals exhibit burstiness on several time scales [39], making the Pareto distribution a good candidate for modeling these inter-arrival times. Therefore, in this section we compare CUP with PCX under Pareto inter-arrivals.

The Pareto distribution has two parameters associated with it: the shape parameter $\alpha > 0$ and the scale parameter $\kappa > 0$. The cumulative distribution function of inter-arrival time durations is $F(x) = 1 - (\frac{\kappa}{(x+\kappa)})^{\alpha}$. This distribution is heavy-tailed

Table 3.14: Per-key, per-query rate comparison of CUP with PCX for Pareto arrivals.

Average Rate (q/s)	**1**	**1**	**10**	**10**
Pareto rate (a)	**1.25**	**1.1**	**1.25**	**1.1**
CUP / PCX MissCost	0.24	0.14	0.08	0.07
PCX AvgLat (σ)	7.77 (9.3)	6.99 (9.4)	3.84 (8.4)	4.01 (8.8)
CUP AvgLat (σ)	3.16 (5.7)	1.71 (4.4)	0.42 (3.0)	0.37 (2.8)
IR	6.41	7.49	13.09	16.03

with unbounded variance when $\alpha < 2$. For $\alpha > 1$, the average number of query arrivals per time unit is equal to $\frac{(\alpha-1)}{\kappa}$. For $\alpha <= 1$, the expectation of an inter-arrival duration is unbounded and therefore the average number of query arrivals per time unit is 0.

We ran experiments for a range of α and κ values and present representative results here. Tables 3.14 and 3.15 compare CUP with PCX for α equal to 1.1 and 1.25 respectively for a network of 1024 nodes. We set the value of κ in each run so that the average rate of arrivals $\frac{(\alpha-1)}{\kappa}$ equals 1, 10, 100, and 1000 queries per second to match the λ rate of the Poisson experiments in previous sections.

As α decreases toward 1, query interarrivals become more bursty. Queries arrive in more frequent and more intense bursts, followed by idle periods of varying lengths. If an idle period occasionally falls in the heavy-tail portion of the Pareto distribution (i.e. very long idle period), then second chance CUP propagation cost could be unrecoverable, since the next query may arrive long after the cached entry has expired. However, CUP does well under bursty conditions because when it is able to refresh a cache before a burst of queries, it saves a large penalty which by far outweighs any unrecovered overhead that occurs during the occasional, very long idle period. Therefore, refreshing the cache in time provides greater benefits with increasing burstiness. The table results confirm this. In going from $\alpha = 1.25$ to $\alpha = 1.1$, we see that the average query latency reduction CUP achieves generally improves and the IR increases for all query rates.

Table 3.15: Per-key, per-query rate comparison of CUP with PCX for Pareto arrivals.

Average Rate (q/s)	**100**	**100**	**1000**	**1000**
Pareto rate (a)	**1.25**	**1.1**	**1.25**	**1.1**
CUP / PCX MissCost	0.07	0.09	0.08	0.08
PCX AvgLat (σ)	1.75 (5.9)	1.61 (5.6)	1.00 (4.0)	1.10 (4.2)
CUP AvgLat (σ)	0.13 (1.7)	0.15 (1.7)	0.08 (1.2)	0.09 (1.2)
IR	43.25	53.57	223.97	293.30

3.3.9 Query Hot Spots

In this experiment we show how CUP performs when there are hot spots in the network, that is when a small portion of the network is posting queries for particular keys. It is possible that some portions of the network will have high interest whereas others will show only low interest for a particular key. We examine how CUP performs in such a scenario.

We show results for an experiment for a 1024 node network in which queries for a particular key were generated according to a Poisson process with 1, 10, 100, and 1000 queries per second. Nodes to post the queries were chosen according to a zipf distribution of the node IDs. This has the effect of having a small number of active querying nodes and a large number of nodes which issue very few queries for the key throughout the simulation.

Table 3.16 compares CUP with PCX. From the table we see that the results of the hot spot experiment are similar to those where nodes are randomly chosen to post queries. This is because the CUP tree simply grows branches in the direction of nodes that have interest in the key. Each node decides individually when to cut off its intake of updates and this decision is independent of the query rates or propagation behavior of other nodes that lie in a different part of the network.

Table 3.17 compares CUP with PCX in a network with hot spots and Pareto arrivals, with $\alpha = 1.1$ and average rate of arrivals of 1, 10, 100, and 1000 queries per second. Again the results here are similar to those observed when nodes are randomly selected to post queries.

Table 3.16: Comparison of CUP with PCX for varying query rates and Hot Spot, Poisson Arrivals

Average Rate (q/s)	**1**	**10**	**100**	**1000**
CUP / PCX MissCost	0.16	0.08	0.08	0.09
PCX AvgLat (σ)	7.07 (8.85)	3.58 (7.76)	1.62 (5.57)	0.96 (3.95)
CUP AvgLat (σ)	2.06 (4.29)	0.39 (2.40)	0.13 (1.59)	0.08 (1.22)
IR	8.87	11.47	32.35	193.89

Table 3.17: Comparison of CUP with PCX for varying query rates and Hot Spot, Pareto Arrivals

Average Rate (q/s)	**1**	**10**	**100**	**1000**
CUP / PCX MissCost	0.16	0.08	0.08	0.09
PCX AvgLat (σ)	6.87 (9.41)	3.56 (7.93)	1.59 (5.42)	1.13 (4.23)
CUP AvgLat (σ)	1.68 (4.39)	0.35 (2.40)	0.14 (1.63)	0.10 (1.32)
IR	7.98	14.20	47.56	294.89

3.3.10 Zipf-like Key Distributions

A recent study has shown that queries for multiple keys in a peer-to-peer network can follow a Zipf-like distribution, with a small portion of the keys getting the most queries [55]. That is, the number of queries received by the *i'th* most popular key is proportional to $\frac{1}{i^\alpha}$ for constant α.

In this section we compare CUP with PCX in a network of 1024 nodes, where each node owns one key. The query distribution among the 1024 keys follows a Zipf-like distribution with parameter $\alpha = 1.2$. Table 3.18 shows results for Poisson arrivals where the λ rates are 100, 1000, 10000, and 100000 queries per second. (We also ran simulations with $\alpha = 0.80$ and 2.40 and with Pareto arrivals with equivalent average rates and the results were similar.)

From the table we see that CUP outperforms PCX with IR ranging from 6.57 to 30.02. The latency reduction ranges from 3.2 (for 100 q/s) to an order of magnitude reduction (for 100000 q/s, latency dropped from 1.53 to 0.13). The Zipf-like distribution causes some of the keys to get a large percentage of the queries, leaving others to

Table 3.18: Cross-Key comparison of CUP with PCX, for Poisson arrivals and Zipf-like key distribution.

Avg Rate (q/s)	**100**	**1000**	**10000**	**100000**
CUP / PCX MissCost	0.45	0.23	0.10	0.08
PCX AvgLat (σ)	10.6 (9.9)	6.9 (8.9)	3.4 (7.5)	1.53 (5.47)
CUP AvgLat (σ)	7.4 (8.5)	2.6 (5.2)	0.4 (2.7)	0.13 (1.67)
IR	6.57	8.52	10.98	30.02

be asked for quite rarely. For such keys, caching does not help since the entry expires by the time the key is queried for again, and the query rate for these keys is not high enough to recover the update propagation. However, the IR for the very hot keys is high enough to by far offset the unrecoverable cost of other less popular keys. As a result, CUP achieves an overall IR of at least 6.57 for 100 q/s and as much as 30.02 for 100000 q/s.

3.4 Summary

CUP is a protocol for maintaining caches of index entries in peer-to-peer networks. CUP query channels coalesce bursts of queries for the same item into a single query. CUP update channels asynchronously transport query responses and refresh intermediate caches. Through light book-keeping and incentive-based propagation cut-off policies, CUP controls and confines propagations to updates that are likely to be justified. In fact, CUP's overhead is compensated for by a factor of 2 to 300 times in terms of savings in cache misses.

When compared with path caching with expiration (PCX), CUP significantly reduces the average query latency over a wide variety of workloads, including Poisson and Pareto query arrivals, networks of increasing size and various topologies, and uniform and Zipf-like multi-key query distributions. We have also shown that even with limited update propagation, CUP continues to outperform PCX. The overall conclusion is that CUP benefits increase with network size and query rates. Although

Poisson query interarrivals may result in less unrecovered overhead than Pareto arrivals, the sheer force of Pareto bursts results in higher CUP benefit (investment return).

We believe that CUP provides a general purpose framework for maintaining metadata in peer-to-peer networks. We demonstrate this in the next chapter where we leverage the CUP protocol to deliver metadata required for effective load-balancing of content demand across replica nodes.

Chapter 4

Practical Load Balancing in P2P Networks

In this chapter we study the problem of load-balancing demand for content in a peer-to-peer network. We propose a new algorithm, Max-Cap, and demonstrate through experiments that Max-Cap avoids the problems observed by current load-balancing algorithms when applied in a peer-to-peer environment.

First, we give an overview of the Max-Cap algorithm. Second, we explain how we leverage the CUP protocol to enable load-balancing. Third we describe the three algorithms compared in our load-balancing study, a load-based algorithm, an algorithm based on available capacity, and Max-Cap. Fourth, we describe experiments that demonstrate the practicality of Max-Cap for load-balancing in a peer-to-peer environment and finally, we conclude the chapter with summary results.

4.1 Max-Cap Overview

Max-Cap is a load-balancing algorithm based on the inherent maximum capacities of the replica nodes. We define maximum capacity as the maximum number of requests per time unit a replica allocates for answering requests for its content. Alternative measures such as maximum (allowed) connections can also be used. The maximum capacity is like a contract by which the replica agrees to abide. If the replica cannot sustain its advertised rate, then it may choose to advertise a new maximum capacity. Max-Cap is not critically tied to the timeliness or frequency of LBI updates, and as a result, when applied in a peer-to-peer environment, outperforms algorithms based on load or available capacity, whose benefits are heavily dependent on the timeliness of the updates.

In this chapter, we show that Max-Cap takes peer node heterogeneity into account unlike algorithms based purely on load. While algorithms based on available capacity also take heterogeneity into account, we show that they can suffer from load oscillations in a peer-to-peer network in the presence of small fluctuations in the workload, even when the workload rate is well within the total maximum capacities of the replicas. On the other hand, Max-Cap avoids overloading replicas in such scenarios and is more resilient to very large fluctuations in workload. This is because a key advantage of Max-Cap is that it uses information that is not affected by changes in the workload.

In a peer-to-peer environment the expectation is that the set of participating nodes will change constantly. Since replica arrivals to and departures from the peer network can affect the information carried in load-balancing updates, we also compare Max-Cap against availability-based algorithms when the set of replicas continuously changes. We show that, Max-Cap is less affected by changes in the replica set than the other algorithms.

Since it is most probable that each replica node in a peer-to-peer environment will run other applications besides the content distribution application, Max-Cap must also be able to handle fluctuations in "extraneous load" observed at the replicas. This is load caused by external factors such as other applications the users of the replica node are running or network conditions occurring at the replica node.

We modify Max-Cap to perform load-balancing using the "honored maximum capacity" of each replica. This is the maximum capacity minus the extraneous load observed at the replica. Although the honored maximum capacities may change frequently, the changes are independent of fluctuations in the content request workload. As a result, Max-Cap continues to provide better load-balancing than availability-based algorithms even when there are large fluctuations in the extraneous load.

We evaluate load-based and availability-based algorithms and compare them with Max-Cap in the context of CUP. We describe how we leverage CUP's update propagation mechanism to transport load-balancing information (LBI) in the next section.

4.2 Leveraging CUP for Load-Balancing

As described in Chapter 3, CUP periodically sends updates down a conceptual tree (similar to an application-level multicast tree) whose branches are formed by queries traveling up toward the authority node holding the answer. Replica nodes send birth, refresh (i.e., renewal), and invalidation messages to the authority node to indicate they have started serving, continue to serve, or are no longer serving particular content. The authority node then propagates these updates down the CUP tree to all interested nodes.

To enable load-balancing, we leverage this propagation mechanism to transport load-balancing information such as replica load or available capacity down the CUP tree. Peer nodes then use the load-balancing information when choosing to which replica a client request should be forwarded.

4.3 The Algorithms

We evaluate two different algorithms, Inv-Load and Avail-Cap. Each is representative of a different class of algorithms that have been proposed in the distributed systems literature. We study how these algorithms perform when applied in a peer-to-peer context and compare them with our proposed algorithm, Max-Cap. These three algorithms depend on different LBI being propagated, but their overall goal is the same: to balance the demand for content fairly across the set of replicas providing the content. In particular, the algorithm should avoid overloading some replicas while underloading others, especially when the aggregate capacity of all replicas is enough to handle the content request workload. Moreover, the algorithm should prevent individual replicas from oscillating between being overloaded and underloaded.

Oscillation is undesirable for two reasons. First, many applications limit the number of requests a host can have outstanding. This means that when a replica node is overloaded, it will drop any request it receives. This forces the requesting client to resend its request which has a negative impact on response time. Even for applications that allow requests to be queued while a replica node is overloaded the queueing delay incurred will also increase the average response time. Second, in a peer-to-peer network, the issue of fairness is sensitive. The owners of replica nodes are likely not to want their nodes to be overloaded while other nodes in the network are underloaded. An algorithm that can fairly distribute the request workload without causing replicas to oscillate between being overloaded and underloaded is preferable.

We describe each of the algorithms we evaluate in turn:

Allocation Proportional to Inverse Load (Inv-Load). There are many load-balancing algorithms that base the allocation decision on the load observed at and reported by each of the serving entities (see Related Work Section 2.3). The representative load-based algorithm we examine in this thesis is Inv-Load, based on the algorithm presented by Genova et al. [32]. In this algorithm, each peer node in the network chooses to forward a request to a replica with probability inversely proportional to the load reported by the replica. This means that the replica with the smallest reported load (as of the last report received) will receive the most requests from the node. Load is defined as the number of request arrivals at the replica per time unit. Other possible load metrics include the number of request connections open at the replica at reporting time [8] or the request queue length at the replica [25].

The Inv-Load algorithm has been shown to perform as well as or better than other proposed algorithms in a homogeneous environment and for this reason we focus on this algorithm in this study. As we will see in Section 4.4.1, Inv-Load, as is, is not designed to handle replica node heterogeneity. We can slightly modify Inv-Load so that replicas are chosen with probability inversely proportional to the load and at the same time proportional to the maximum capacity of the replicas. This results in an algorithm that is essentially equivalent to Avail-Cap, which we describe next.

Allocation Proportional to Available Capacity (Avail-Cap). In this algorithm, each peer node chooses to forward a request to a replica with probability proportional to the available capacity reported by the replica. Available capacity is the maximum request rate a replica can handle minus the load (actual request rate) experienced at the replica. This algorithm is based on the algorithm proposed by Zhu et al. [65] for load sharing in a cluster of heterogeneous servers. Avail-Cap takes into account heterogeneity because it distinguishes between nodes that experience the same load but have different maximum capacities.

Intuitively, Avail-Cap seems like it should work; it handles heterogeneity by sending more requests to the replicas that are currently more capable. Replicas that are overloaded report an available capacity of zero and are excluded from the allocation decision until they once more report a positive available capacity. Unfortunately, as we will show in Section 4.4.2, this exclusion can cause Avail-Cap to suffer from wild load oscillations.

Both Inv-Load and Avail-Cap implicitly assume that the load or available capacity reported by a replica remains roughly constant until the next report. Since both these metrics are directly affected by changes in the request workload, both algorithms require that replicas periodically update their LBI. (We assume replicas are not synchronized in when they send reports.) Decreasing the period between two consecutive LBI updates increases the timeliness of the LBI at a cost of higher overhead (in number of updates pushed through the peer-to-peer network). In large peer-to-peer networks, there may be several levels in the CUP tree down which updates will have to travel, and the time to do so could be on the order of seconds.

Allocation Proportional to Maximum Capacity (Max-Cap). This is the algorithm we propose. In this algorithm, each peer node chooses to forward a request to a replica with probability proportional to the maximum capacity of the replica. The maximum capacity is a contract each replica advertises indicating the number of requests the replica claims to handle per time unit. Unlike load and available capacity, the maximum capacity of a replica is not affected by changes in the content request workload. Therefore, Max-Cap does not depend on the timeliness of the LBI updates. In fact, replicas only push updates down the CUP tree when they choose to advertise a new maximum capacity. This choice depends on extraneous factors that are unrelated to and independent of the workload (see Section 4.4.4). If replicas rarely choose to change contracts, Max-Cap incurs near-zero overhead. We believe that this independence of the timeliness and frequency of updates makes Max-Cap practical and elegant for use in peer-to-peer networks.

4.4 Experiments

In this section we describe experiments that measure the ability of the Inv-Load, Avail-Cap and Max-Cap algorithms to balance requests for content fairly across the replicas holding the content. We simulate a content-addressable network (CAN) [45] using the Stanford Narses simulator [38]. A CAN is an example of a structured peer-to-peer network, defined in Section 2.1. In each of these experiments, requests for a specific piece of content are posted at nodes throughout the CAN network for 3000 seconds. Using the CUP protocol described in Section 2.1, a node that receives a content request from a local client retrieves a set of index entries pointing to replica nodes that serve the content. The node applies a load-balancing algorithm to choose one of the replica nodes. It then points the local client making the content request at the chosen replica.

The simulation input parameters include: the number of nodes in the overlay peer-to-peer network, the number of replica nodes holding the content of interest, the maximum capacities of the replica nodes, the distribution of content request inter-arrival times, a seed to feed the random number generators that drive the content request arrivals and the allocation decisions of the individual nodes, and the LBI update period, which is the amount of time each replica waits before sending the next LBI update for the Inv-Load and Avail-Cap algorithms.

We assign maximum capacities to replica nodes by applying results from recent work that measures the upload capabilities of nodes in Gnutella networks [50]. This work has found that for the Gnutella network measured, around 10% of nodes are connected through dial-up modems, 60% are connected through broadband connections such as cable modem or DSL where the upload speed is about ten times that of dial-up modems, and the remaining 30% have high-end connections with upload speed at least 100 times that of dial-up modems. Therefore we assign maximum capacities of 1, 10, and 100 requests per second to nodes with probability of 0.1, 0.6, and 0.3, respectively.

In all the experiments we present in this chapter, the number of nodes in the network is 1024, each individually deciding how to distribute its incoming content requests across the replica nodes. We use both Poisson and Pareto request inter-arrival distributions, both of which have been found to hold in peer-to-peer networks [11, 39].

We present five experiments. First we show that Inv-Load cannot handle heterogeneity. We then show that while Avail-Cap takes replica heterogeneity into account, it can suffer from significant load oscillations caused by even small fluctuations in the workload. We compare Max-Cap with Avail-Cap for both Poisson and bursty Pareto arrivals. We also compare the effect on the performances of Avail-Cap and Max-Cap when replicas continuously enter and leave the system. Finally, we study the effect on Max-Cap when replicas cannot always honor their advertised maximum capacities because of significant extraneous load.

4.4.1 Inv-Load and Heterogeneity

It should be intuitive that algorithms based purely on load do not handle heterogeneity. To confirm this intuition, in this experiment, we examine the performance of Inv-Load in a heterogeneous peer-to-peer environment. We use a fairly short inter-update period of one second, which is quite aggressive in a large peer-to-peer network. We have ten replica nodes that serve the content item of interest, and we generate request rates for that item according to a Poisson process with an arrival rate that is 80% of the total maximum capacities of the replicas. Under such a workload, a good load-balancing algorithm should be able to avoid overloading some replicas while underloading others. Figure 4.1 shows a scatterplot of how the utilization of each replica proceeds with time when using Inv-Load. We define utilization as the request arrival rate observed by the replica divided by the maximum capacity of the replica. In this graph, we do not distinguish among points of different replicas. We see that throughout the simulation at any point in time, some replicas are severely overutilized (over 250%) while others are lightly underutilized (around 25%).

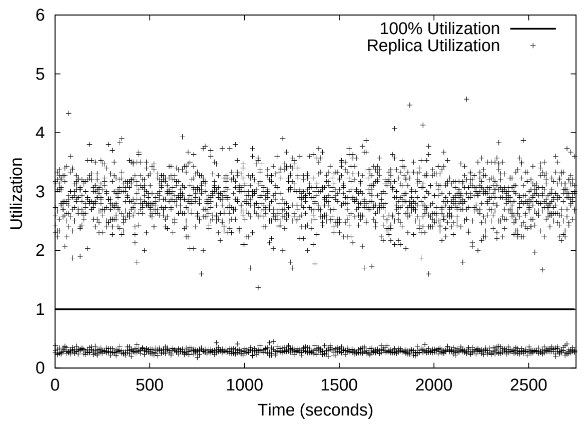

Figure 4.1: Replica Utilization versus Time for InvLoad with heterogeneous replicas.

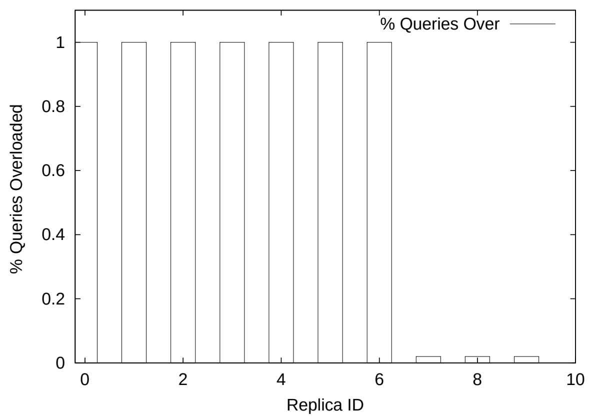

Figure 4.2: Percentage Overloaded Queries versus Replica ID for Inv-Load with heterogeneous replicas.

Figure 4.2 shows for each replica, the percentage of all received requests that arrive while the replica is overloaded. This measurement gives a true picture of how well a load-balancing algorithm works for each replica. In Figure 4.2, the replicas that receive almost 100% of their requests while overloaded (i.e., replicas 0-6) are the low and middle-end replicas. The replicas that receive almost no requests while overloaded (i.e., replicas 7-9) are the high-end replicas. We see that Inv-Load penalizes the less capable replicas while giving the high-end replicas an easy time.

Inv-Load is designed to perform well in a homogeneous environment. When applied in a heterogeneous environment such as a peer-to-peer network, it fails. As we will see in the next section Max-Cap is much better suited. Apart from showing that Max-Cap has comparable load balancing capability with no overhead in a homogeneous environment (see Appendix), we do not consider Inv-Load in the remaining experiments as our focus here is on heterogeneous environments.

4.4.2 Avail-Cap versus Max-Cap

In this set of experiments we examine the performance of Avail-Cap and compare it with Max-Cap.

Poisson Request Arrivals

In Figures 4.3 and 4.4 we show the replica utilization versus time for an experiment with ten replicas with a Poisson request arrival rate of 80% the total maximum capacities of the replicas. For Avail-Cap, we use an inter-update period of one second. For Max-Cap, this parameter is inapplicable since replica nodes do not send updates unless they experience extraneous load (see Section 4.4.4). We see that Avail-Cap consistently overloads some replicas while underloading others. In contrast, Max-Cap tends to cluster replica utilization at around 80%. We ran this experiment with a range of Poisson lambda rates and found similar results for rates that were 60-100% the total maximum capacities of the replicas. Avail-Cap consistently overloads some replicas while underloading others whereas Max-Cap clusters replica utilization at around X% utilization, where X is the overall request rate divided by the total maximum capacities of the replicas.

It turns out that in Avail-Cap, unlike Inv-Load, it is not the same replicas that are consistently overloaded or underloaded throughout the experiment. Instead, from one instant to the next, individual replicas oscillate between being overloaded and severely underloaded.

We can see a sampling of this oscillation by looking at the utilizations of some individual replicas over time. In Figures 4.5-4.10, we plot the utilization over a one minute period in the experiment for a representative replica from each of the replica classes (low, medium, and high maximum capacity). We also plot the ratio of the

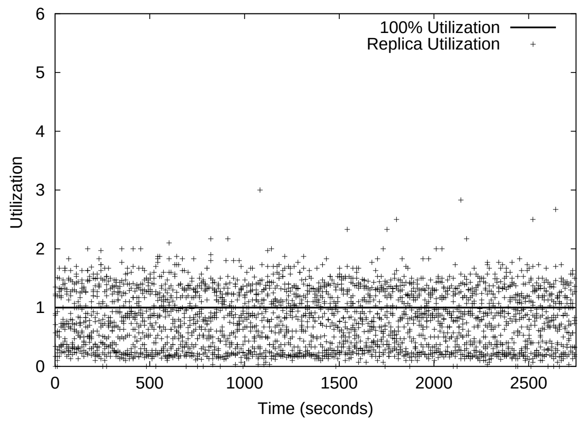

Figure 4.3: Replica Utilization versus Time for Avail-Cap with heterogeneous replicas.

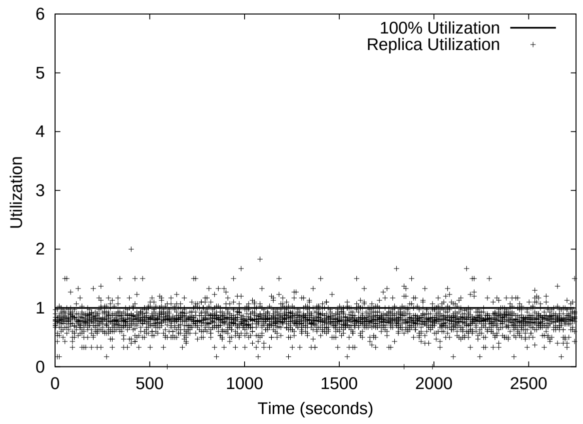

Figure 4.4: Replica Utilization v. Time for Max-Cap with heterogeneous replicas.

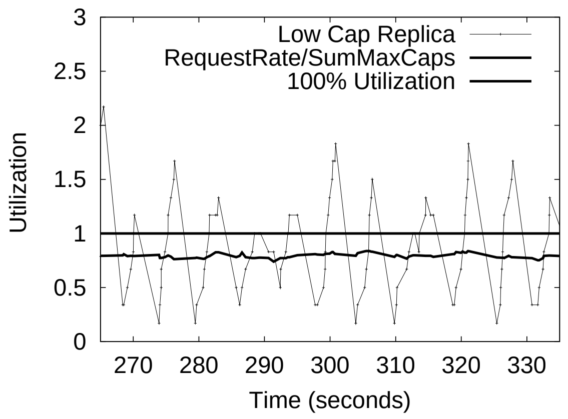

Figure 4.5: Low-end Replica Utilization versus Time for Avail-Cap, Poisson arrivals.

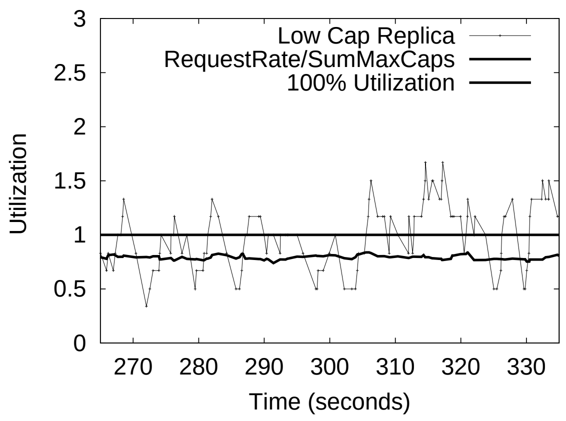

Figure 4.6: Low-end Replica Utilization versus Time for Max-Cap, Poisson arrivals.

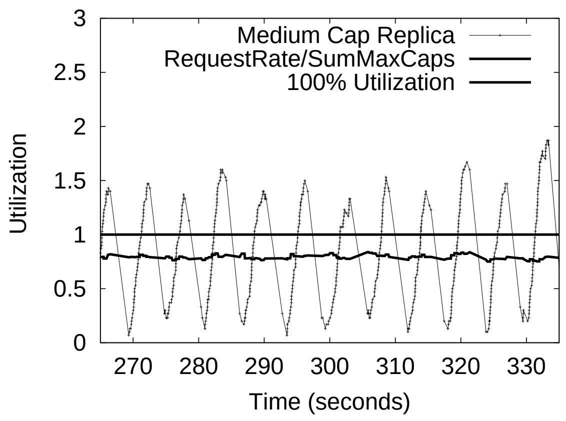

Figure 4.7: Medium-end Replica Utilization versus Time for Avail-Cap, Poisson arrivals.

overall request rate to the total maximum capacities of the replicas and the line $y = 1$ showing 100% utilization. We see that for all replica classes, Avail-Cap suffers from significant oscillation when compared with Max-Cap which causes little or no oscillation. This behavior occurs throughout the experiment.

Figures 4.11 and 4.12 show the percentage of requests that arrive at each replica while the replica is overloaded for Avail-Cap and Max-Cap respectively. We see that Max-Cap achieves much lower percentages than Avail-Cap.

We also see in Figure 4.12 that Max-Cap exhibits a step-like behavior where the

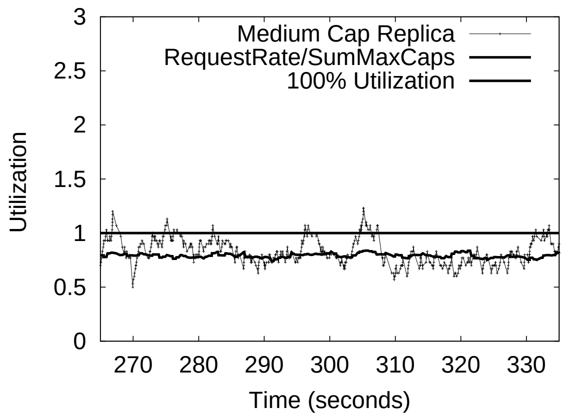

Figure 4.8: Medium-end Replica Utilization versus Time for Max-Cap, Poisson arrivals.

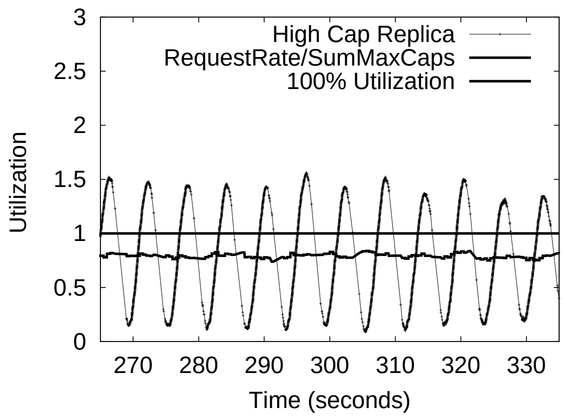

Figure 4.9: High-end Replica Utilization versus Time for Avail-Cap, Poisson arrivals.

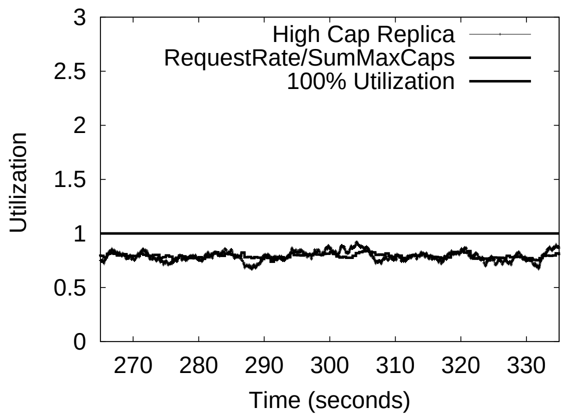

Figure 4.10: High-end Replica Utilization versus Time for Max-Cap, Poisson arrivals.

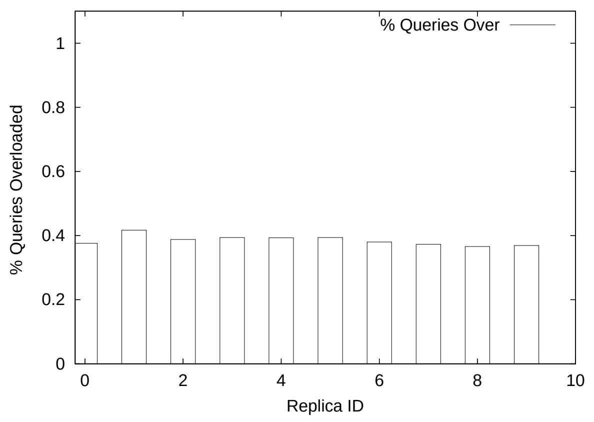

Figure 4.11: Percentage Overloaded Queries versus Replica ID for Avail-Cap, with inter-update period of 1 second.

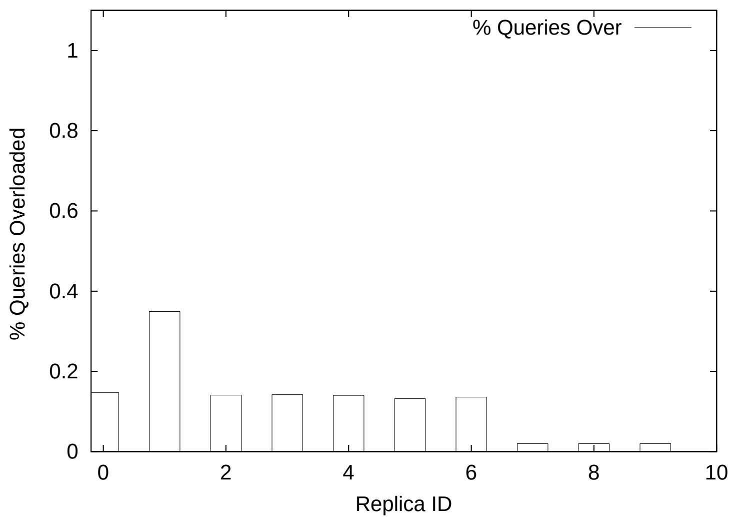

Figure 4.12: Percentage Overloaded Queries versus Replica ID for Max-Cap.

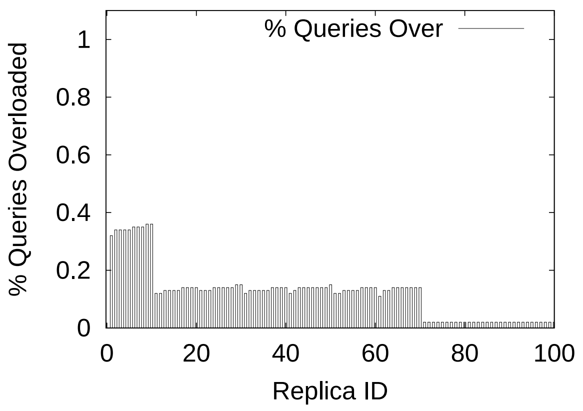

Figure 4.13: Percentage Overloaded Queries versus Replica ID for Max-Cap for ten experiments.

low-capacity replica (replica 1) is overloaded for about 35% of its queries, the middle-capacity replicas (replicas 0 and 2-6) are each overloaded for about 14% of their queries, and the high-capacity replicas (replicas 7-9) are each overloaded for about 0.1% of their queries. To verify that this step effect is not a random coincidence, we ran a series of experiments, with ten replicas per experiment, and Poisson arrivals of 80% the total maximum capacity, each time varying the seed fed to the simulator. In Figure 4.13, we show the overloaded percentages for ten of these experiments. On the x-axis we order replicas according to maximum capacity, with the low-capacity replicas plotted first (replica IDs 1 through 10), followed by the middle-capacity replicas (replica IDs 11-70), followed by the high-capacity replicas (replica IDs 71-100). From the figure we see that the step behavior consistently occurs. This step behavior occurs because the lower-capacity replicas have less tolerance for noise in the random coin tosses the nodes perform while assigning requests. They also have less tolerance for small fluctuations in the request rate. As a result, lower-capacity replicas are overloaded more easily than higher-capacity replicas.

Figure 4.11 shows that Avail-Cap with an inter-update period of one second causes much higher percentages than Max-Cap (more than twice as high for the medium and high-end replicas). Avail-Cap also causes fairly even overloaded percentages at around 40%. Again, to verify this evenness, in Figure 4.14, we show for a series of ten

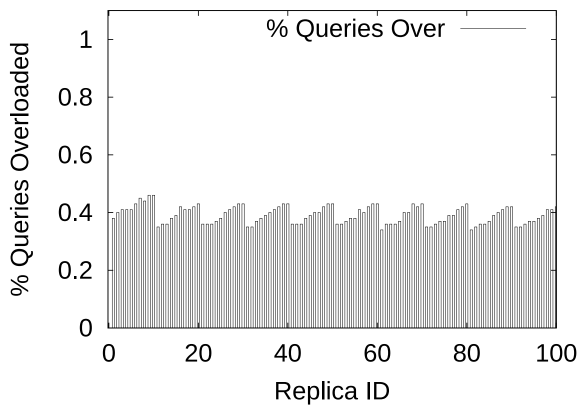

Figure 4.14: Percentage Overload Queries versus Replica ID for Avail-Cap with inter-update period of 1 second, for ten experiments.

experiments, the percentage of requests that arrive at each replica while the replica is overloaded. We see that Avail-Cap consistently achieves roughly even percentages (at around 40%) across all replica types in contrast to the step effect observed by Max-Cap. This can be explained by looking at the oscillations observed by replicas in Figures 4.5-4.10. In Avail-Cap, each replica is overloaded for roughly the same amount of time regardless of whether it is a low, medium or high-capacity replica. This means that while each replica is getting the correct proportion of requests, it is receiving them at the wrong time and as a result all the replicas experience roughly the same overloaded percentages. In Max-Cap, we see that replicas with lower maximum capacity are overloaded for more time that higher-capacity replicas. Consequently, higher-capacity replicas tend to have smaller overload percentages than lower-capacity replicas.

The performance of Avail-Cap is highly dependent on the inter-update period used. We find that as we increase the period and available capacity updates grow more stale, the performance of Avail-Cap suffers more. As an example, in Figure 4.15, we show the overloaded query percentages in the same series of ten experiments for Avail-Cap with a period of ten seconds. The overloaded percentages jump up to about 80% across the replicas.

In a peer-to-peer environment, we argue that Max-Cap is a more practical choice

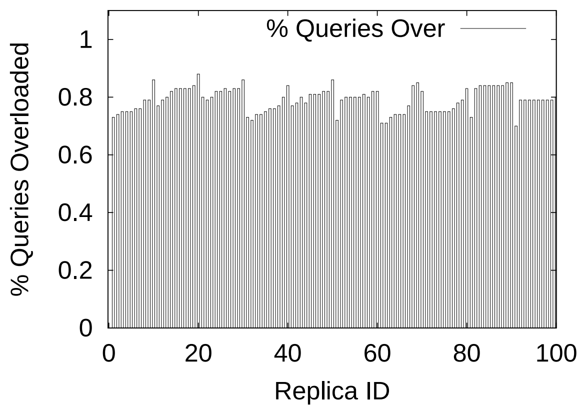

Figure 4.15: Percentage Overload Queries versus Replica ID for Avail-Cap with inter-update period of 10 seconds, for ten experiments.

than Avail-Cap. First, Max-Cap typically incurs no overhead. Second, Max-Cap can handle workload rates that are below 100% the total maximum capacities and can handle small fluctuations in the workload as are typical in Poisson arrivals.

A question remaining is how do Avail-Cap and Max-Cap compare when workload rates fluctuate beyond the total maximum capacities of the replicas? Such a scenario can occur for example when requests are bursty, as when inter-request arrival times follow a Pareto distribution. We examine Pareto arrivals next.

Pareto Request Arrivals

Recent work has observed that in some peer-to-peer networks, request inter-arrivals exhibit burstiness on several time scales [39], making the Pareto distribution a good candidate for modeling these inter-arrival times.

The Pareto distribution has two parameters associated with it: the shape parameter $\alpha > 0$ and the scale parameter $\kappa > 0$. The cumulative distribution function of inter-arrival time durations is $F(x) = 1 - (\frac{\kappa}{(x+\kappa)})^{\alpha}$. This distribution is heavy-tailed with unbounded variance when $\alpha < 2$. For $\alpha > 1$, the average number of query arrivals per time unit is equal to $\frac{(\alpha-1)}{\kappa}$. For $\alpha <= 1$, the expectation of an inter-arrival duration is unbounded and therefore the average number of query arrivals per time unit is 0.

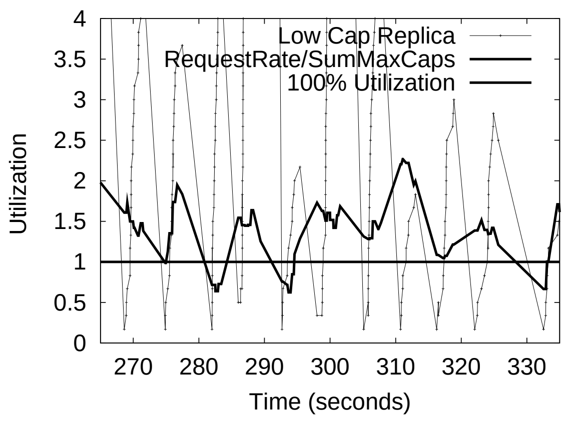

Figure 4.16: Low-capacity Replica Utilization versus Time for Avail-Cap, Pareto arrivals.

Typically, Pareto request arrivals are characterized by frequent and intense bursts of requests followed by idle periods of varying lengths. During the bursts, the average request arrival rate can be many times the total maximum capacities of the replicas. We present a representative experiment in which α and κ are 1.1 and 0.000346 respectively. These particular settings cause bursts of up to 230% the total maximum capacities of the replicas. With such intense bursts, no load-balancing algorithm can be expected to keep replicas underloaded. Instead the best an algorithm can do is to have the oscillation observed by each replica's utilization match the oscillation of the ratio of overall request rate to total maximum capacities.

In Figures 4.16-4.21 we plot the same representative replica utilizations over a one minute period in the experiment. We also plot the ratio of the overall request rate to the total maximum capacities as well as the $y = 100\%$ utilization line. From the figures we see that Avail-Cap suffers from much wilder oscillation than Max-Cap, causing much higher peaks and lower valleys in replica utilization than Max-Cap. Moreover, Max-Cap adjusts better to the fluctuations in the request rate; the utilization curves for Max-Cap tend to follow the ratio curve more closely than those for Avail-Cap.

(Note that idle periods contribute to the drops in utilization of replicas in this experiment. For example, an idle period occurs between times 324 and 332 at which point we see a decrease in both the ratio and the replica utilization.)

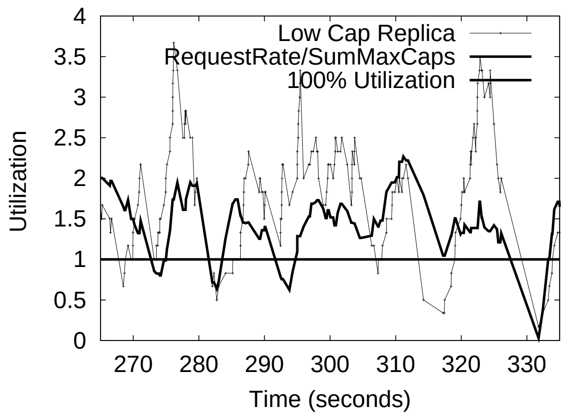

Figure 4.17: Low-capacity Replica Utilization versus Time for Max-Cap, Pareto arrivals.

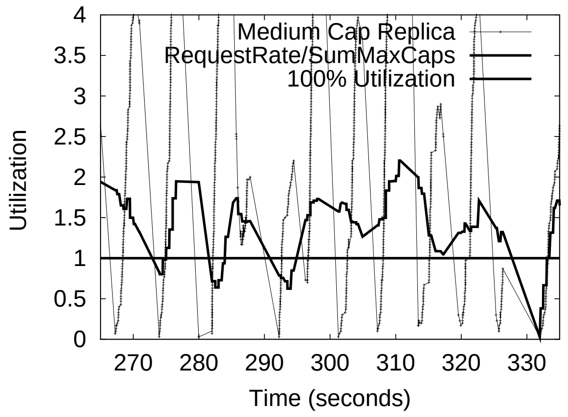

Figure 4.18: Medium-capacity Replica Utilization versus Time for Avail-Cap, Pareto arrivals.

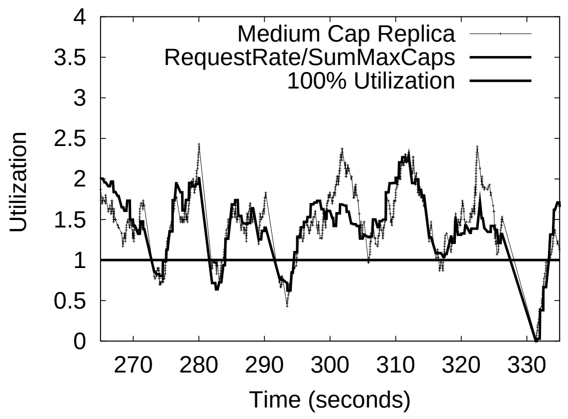

Figure 4.19: Medium-capacity Replica Utilization versus Time for Max-Cap, Pareto arrivals.

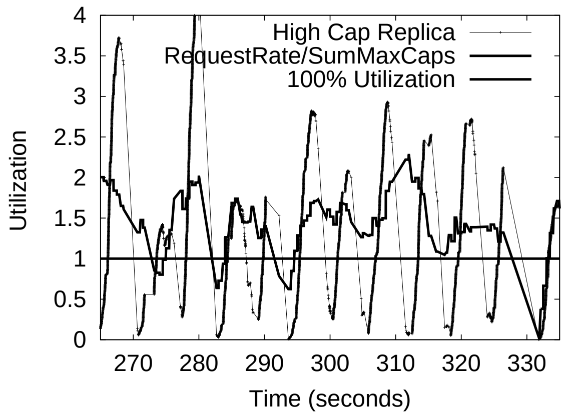

Figure 4.20: High-capacity Replica Utilization versus Time for Avail-Cap, Pareto arrivals.

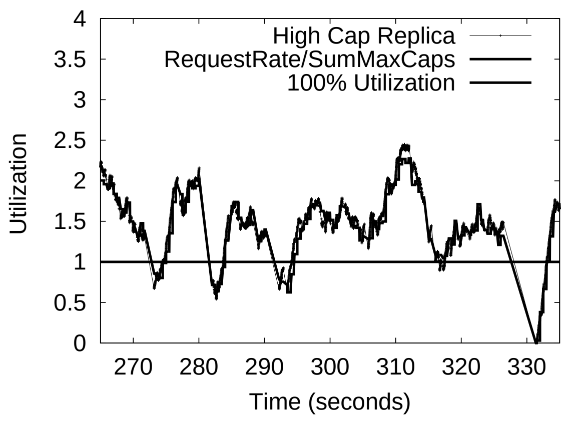

Figure 4.21: High-capacity Replica Utilization versus Time for Max-Cap, Pareto arrivals.

Why Avail-Cap Can Suffer

From the experiments above we see that Avail-Cap can suffer from severe oscillation even when the overall request rate is well below (e.g., 80%) the total maximum capacities of the replicas. The reason why Avail-Cap does not balance load well here is that a vicious cycle is created where the available capacity update of one replica affects a subsequent update of another replica. This in turn affects later allocation decisions made by nodes which in turn affects later replica updates. This description becomes more concrete if we consider what happens when a replica is overloaded.

In Avail-Cap, if a replica becomes overloaded, it reports an available capacity of zero. This report eventually reaches all peer nodes, causing them to stop redirecting requests to the replica. The exclusion of the overloaded replica from the allocation decision shifts the entire burden of the workload to the other replicas. This can cause other replicas to overload and report zero available capacity while the excluded replica experiences a sharp decrease in its utilization. This sharp decrease causes the replica to begin reporting positive available capacity which begins to attract requests again. Since in the meantime other replicas have become overloaded and excluded from the allocation decision, the replica receives a flock of requests which cause it to become overloaded again. As we observed in previous sections, a replica can experience wild and periodic oscillation where its utilization continuously rises above its maximum capacity and falls sharply.

In Max-Cap, if a replica becomes overloaded, the overload condition is confined to that replica. The same is true in the case of underloaded replicas. Since the overload/underload situations of the replicas are not reported, they do not influence follow-up LBI updates of other replicas. It is this key property that allows Max-Cap to avoid herd behavior.

There are situations however where Avail-Cap performs well without suffering from oscillation (see Section 4.4.3). We next describe the factors that affect the performance of Avail-Cap to get a clearer picture of when the reactive nature of Avail-Cap is beneficial (or at least not harmful) and when it causes oscillation.

Factors Affecting Avail-Cap

There are four factors that affect the performance of Avail-Cap: the inter-update period U, the inter-request period R, the amount of time T it takes for all nodes in the network to receive the latest update from a replica, and the ratio of the overall request rate to the total maximum capacities of the replicas. We examine these factors by considering three cases:

Case 1: U is much smaller than R ($U << R$), and T is sufficiently small so that when a replica pushes an update, all nodes in the CUP tree receive the update before the next request arrival in the network. In this case, Avail-Cap performs well since all nodes have the latest load-balancing information whenever they receive a request.

Case 2: U is long relative to R ($U > R$) and the overall request rate is less than about 60% the total maximum capacities of the replicas. (This 60% threshold is specific to the particular configuration of replicas we use: 10% low, 60% medium, 30% high. Other configurations have different threshold percentages that are typically well below the total maximum capacities of the replicas.) In this case, when a particular replica overloads, the remaining replicas are able to cover the proportion of requests intended for the overloaded replica because there is a lot of extra capacity in the system. As a result, Avail-Cap avoids oscillations. We see experimental evidence for this in Section 4.4.3. However, over-provisioning to have enough extra capacity in the system so that Avail-Cap can avoid oscillation in this particular case seems a high price to pay for load stability.

Case 3: U is long relative to R ($U > R$) and the overall request rate is more than about 60% the total maximum capacities of the replicas. In this case, as we observe in the experiments above, Avail-Cap can suffer from oscillation. This is because every request that arrives directly affects the available capacity of one of the replicas. Since the request rate is greater than the update rate, an update becomes stale shortly after a replica has pushed it out. However, the replica does not inform the nodes of its changing available capacity until the end of its current update period. By that point many requests have arrived and have been allocated using the previous, stale available capacity information.

In Case 3, Avail-Cap can suffer even if $T = 0$ and updates were to arrive at all nodes immediately after being issued. This is because all nodes would simultaneously exclude an overloaded replica from the allocation decision until the next update is issued. As T increases, the staleness of the report only exacerbates the performance of Avail-Cap.

In a large peer-to-peer network (more than 1000 nodes) we expect that T will be on the order of seconds since current peer-to-peer networks with more than 1000 nodes have diameters ranging from a handful to several hops [46]. We consider $U = 1$ second to be as small (and aggressive) an inter-update period as is practical in a peer-to-peer network. In fact even one second may be too aggressive due to the overhead it generates. This means that when particular content experiences high popularity, we expect that typically $U + T >> R$. Under such circumstances Avail-Cap is not a good load-balancing choice. For less popular content, where $U + T < R$, Avail-Cap is a feasible choice, although it is unclear whether load-balancing across the replicas is as urgent here, since the request rate is low.

The performance of Max-Cap is independent of the values of U, R, and T. More importantly, Max-Cap does not require continuous updates; replicas issue updates only if they choose to re-issue new contracts to report changes in their maximum capacities. (See Section 4.4.4). Therefore, we believe that Max-Cap is a more practical choice in a peer-to-peer context than Avail-Cap.

4.4.3 Dynamic Replica Set

A key characteristic of peer-to-peer networks is that they are subject to constant change; peer nodes continuously enter and leave the system. In this experiment we compare Max-Cap with Avail-Cap when replicas enter and leave the system. We present results here for a Poisson request arrival rate that is 80% the total maximum capacities of the replicas.

We present two dynamic experiments. In both experiments, the network starts with ten replicas and after a period of 600 seconds, movement into and out of the network begins. In the first experiment, one replica leaves and one replica enters the network every 60 seconds. In the second and much more dynamic experiment, five replicas leave and five replicas enter the network every 60 time units. The replicas that leave are randomly chosen. The replicas that enter the network enter with maximum capacities of 1, 10, and 100 with probability of 0.10, 0.60, and 0.30 respectively as in the initial allocation. This means that the total maximum capacities of the active replicas in the network varies throughout the experiment, depending on the capacities of the entering replicas.

Figures 4.22 and 4.23 show for the first dynamic experiment the utilization of active replicas throughout time as observed for Avail-Cap and Max-Cap. Note that points with zero utilization indicate newly entering replicas. The jagged line plots the ratio of the current sum of maximum capacities in the network, S_{curr}, to the original sum of maximum capacities, S_{orig}. With each change in the replica set, the replica utilizations for both Avail-Cap and Max-Cap change. Replica utilizations rise when S_{curr} falls and vice versa.

From the figure we see that between times 1000 and 1820, S_{curr} is between 1.75 and 2 times S_{orig}, and is more than double the overall workload rate of $0.8 * S_{orig}$. During this time period, Avail-Cap performs quite well because the workload rate is not very demanding and there is plenty of extra capacity in the system (Case 2 above). However, when at time 1940 S_{curr} falls back to S_{orig}, we see that both algorithms exhibit the same behavior as they do at the start, between times 0 and 600. Max-Cap readjusts nicely and clusters replica utilization at around 80%, while Avail-Cap starts to suffer again.

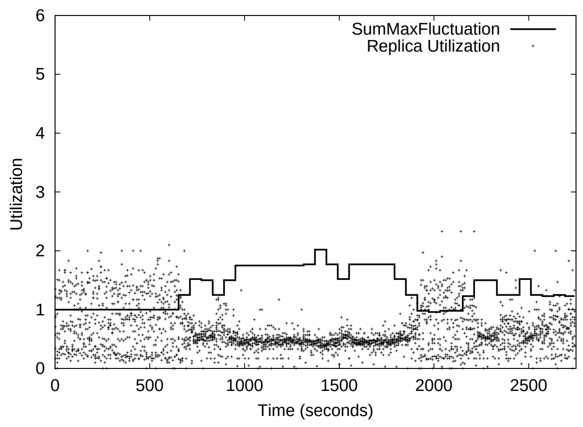

Figure 4.22: Replica Utilization versus Time for Avail-Cap with a dynamic replica set. One replica enters and leaves every 60 seconds.

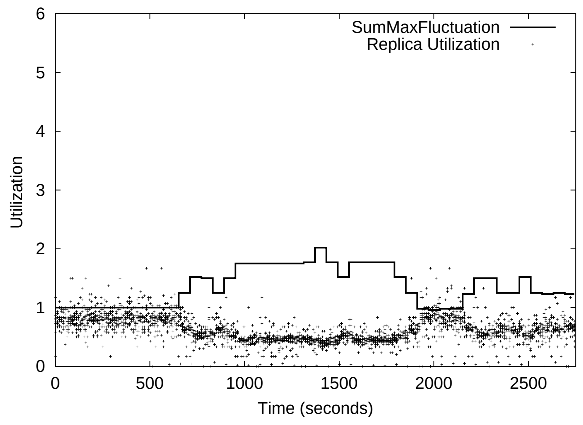

Figure 4.23: Replica Utilization versus Time for Max-Cap with a dynamic replica set. One replica enters and leaves every 60 seconds.

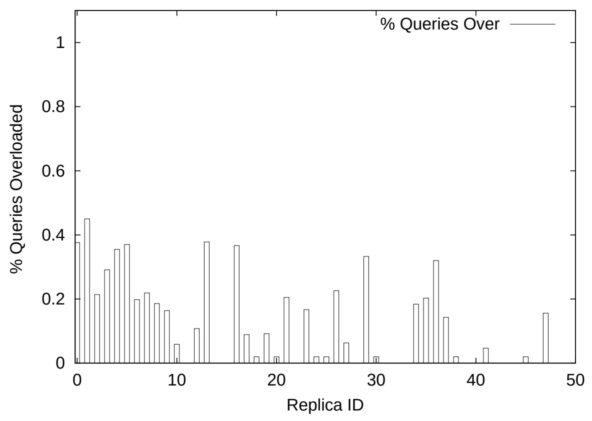

Figure 4.24: Percentage Overloaded Queries versus Replica ID for Avail-Cap with a dynamic replica set. One replica enters and leaves every 60 seconds.

Figures 4.24 and 4.25 show for the first dynamic experiment the percentage of queries that were received by each replica while the replica was overloaded for Avail-Cap and Max-Cap. Replicas that entered and departed the network throughout the simulation were chosen from a pool of 50 replicas. Those replicas in the pool which did not participate in this experiment do not have a bar associated with their ID in the figure. From the figure, we see that Max-Cap achieves smaller overload query percentages across all replica IDs.

Figures 4.26 and 4.27 show the utilization scatterplot and Figures 4.28 and 4.29 show the overloaded query percentage for the second dynamic experiment. We see that changing half the replicas every 60 seconds can dramatically affect S_{curr}. For example, when S_{curr} drops to $0.2S_{orig}$ at time 2161, we see the utilizations rise dramatically for both Avail-Cap and Max-Cap. This is because during this period the workload rate is four times that of S_{curr}. However by time 2401, S_{curr} has risen to $1.2S_{orig}$ which allows for both Avail-Cap and Max-Cap to adjust and decrease the replica utilization. At the next replica set change at time 2461, S_{curr} equals S_{orig}. During the next minute we see that Max-Cap overloads very few replicas whereas Avail-Cap does not recuperate as well. Similarly, when examining the overloaded query percentage we see that Max-Cap achieves smaller percentages when compared with Avail-Cap.

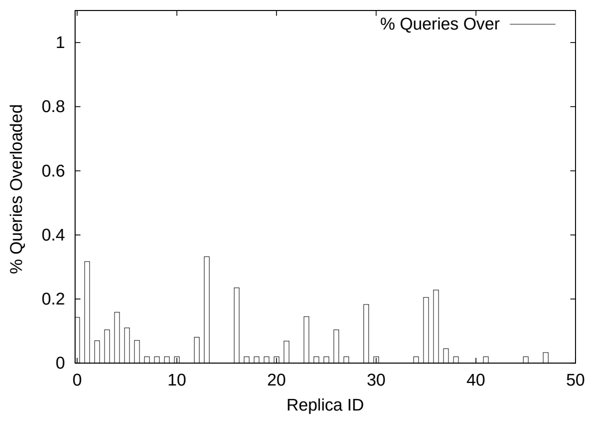

Figure 4.25: Percentage Overloaded Queries versus Replica ID for Max-Cap with a dynamic replica set. One replica enters and leaves every 60 seconds.

The two dynamic experiments we have described above show two things; first, when the workload is not very demanding and there is unused capacity, the behaviors of Avail-Cap and Max-Cap are similar However, Avail-Cap suffers more as overall available capacity decreases. Second, Avail-Cap is affected more by short-lived fluctuations (in particular, decreases) in total maximum capacity than Max-Cap. This is because the reactive nature of Avail-Cap causes it to adapt abruptly to changes in capacities, even when these changes are short-lived.

4.4.4 Extraneous Load

When replicas can honor their maximum capacities, Max-Cap avoids the oscillation that Avail-Cap can suffer, and does so with no update overhead. Occasionally, some replicas may not be able to honor their maximum capacities because of *extraneous load* caused by other applications running on the replicas or network conditions unrelated to the content request workload.

To deal with the possibility of extraneous load, we modify the Max-Cap algorithm slightly to work with honored maximum capacities. A replica's honored maximum capacity is its maximum capacity minus the extraneous load it is experiencing. The algorithm changes slightly; a peer node chooses a replica to which to forward a content request with probability proportional to the honored maximum capacity advertised by

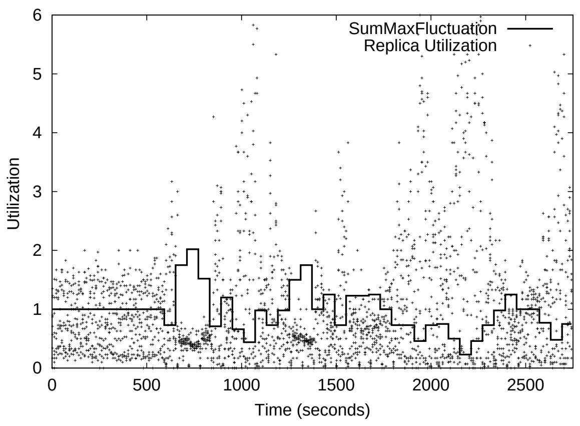

Figure 4.26: Replica Utilization versus Time for Avail-Cap with a dynamic replica set. Half the replicas enter and leave every 60 seconds.

the replica. This means that replicas may choose to send updates to indicate changes in their honored maximum capacities. However, as we will show, the behavior of Max-Cap is not tied to the timeliness of updates in the way Avail-Cap is.

We view the honored maximum capacity reported by a replica as a contract. If the replica cannot adhere to the contract or has extra capacity to give, but does not report the deficit or surplus, then that replica alone will be affected and may be overloaded or underloaded since it will be receiving a request share that is proportional to its previous advertised honored maximum capacity.

If, on the other hand, a replica chooses to issue a new contract with the new honored maximum capacity, then this new update can affect the load balancing decisions of the nodes in the peer network and the workload could shift to the other replicas. This shift in workload is quite different from that experienced by Avail-Cap when a replica reports overload and is excluded. The contracts of any other replica will not be affected by this workload shift. Instead, the contract is are solely affected by the extraneous load that replica experiences which is independent of the extraneous load experienced by the replica issuing the new contract. This is unlike Avail-Cap where the available capacity reported by one replica directly affects the available capacities of the others.

In this section we study the performance of Max-Cap in an experiment where

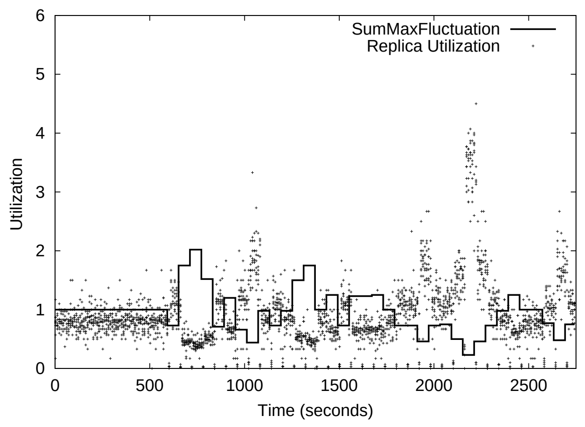

Figure 4.27: Replica Utilization versus Time for Max-Cap with a dynamic replica set. Half the replicas enter and leave every 60 seconds.

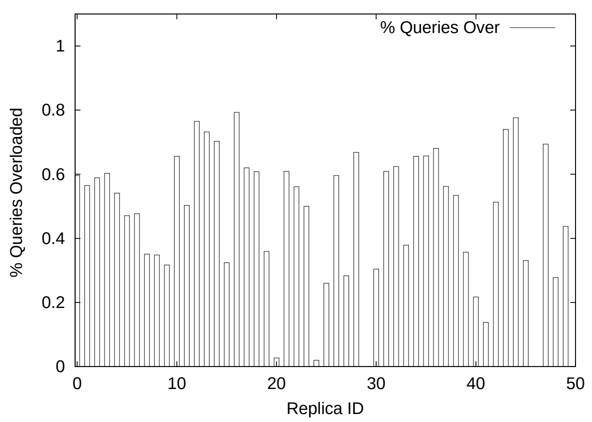

Figure 4.28: Percentage Overloaded Queries versus Replica ID for Avail-Cap with a dynamic replica set. Half the replicas enter and leave every 60 seconds.

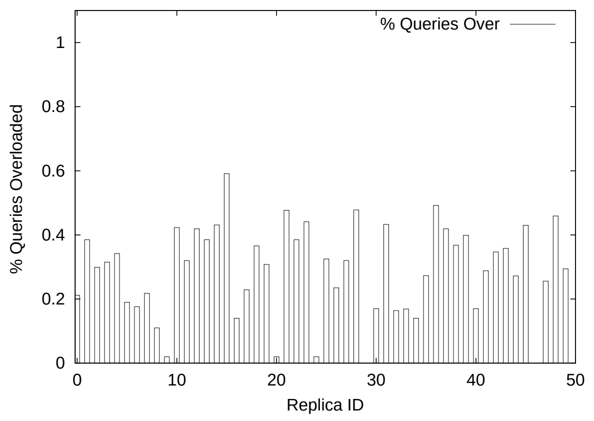

Figure 4.29: Percentage Overloaded Queries versus Replica ID for Max-Cap with a dynamic replica set. Half the replicas enter and leave every 60 seconds.

all replica nodes are continuously issuing new contracts. Specifically, for each of ten replicas, we inject extraneous load into the replica once a second. The extraneous load injected is randomly chosen to be anywhere between 0% and 50% of the replica's original maximum capacity. Figures 4.30 and 4.31 show the replica utilization versus time and the overloaded query percentages for Max-Cap with an inter-update period of 1 second. The jagged line in Figure 4.30 shows the total honored maximum capacities over time. Since throughout the experiment each replica's honored maximum capacity varies between 50% and 100% its original maximum capacity, the total maximum capacity is expected to hover at around 75% the original total maximum capacity and we see that the jagged line hovers around this value. We therefore generate Poisson request arrivals with an average rate that is 80% of this value to keep consistent with our running example of 80% workload rates.

From the figures, we see that Max-Cap continues to cluster replica utilization at around 80%, but there are more overloaded replicas throughout time than when compared with the experiment in which all replicas adhere to their contracts all the time (Figure 4.4). We also see that the overloaded percentages are higher than before (Figure 4.12). The reason for this performance degradation is that the randomly injected load (of 0% to 50%) can cause sharp rises and falls in the reported contract of each replica from one second to the next. Since the change is so rapid, and updates

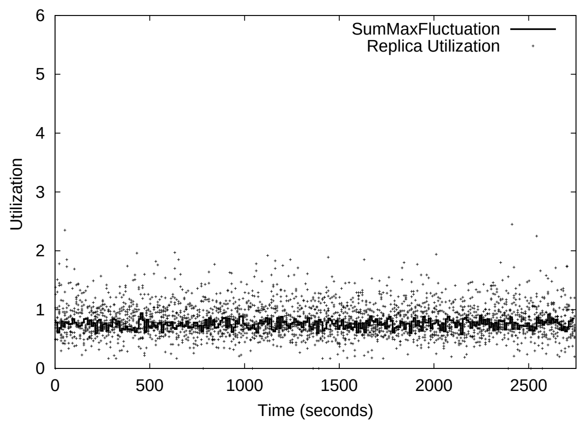

Figure 4.30: Replication Utilization versus Time for Max-Cap with extraneous load and an inter-update period of one second.

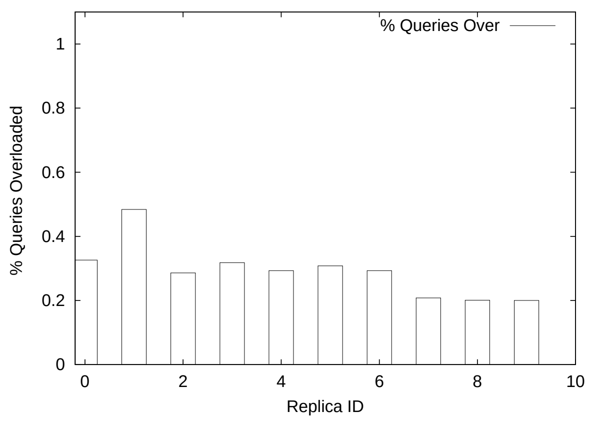

Figure 4.31: Percentage Overloaded Queries versus Replica ID, Max-Cap with extraneous load and an inter-update period of one second.

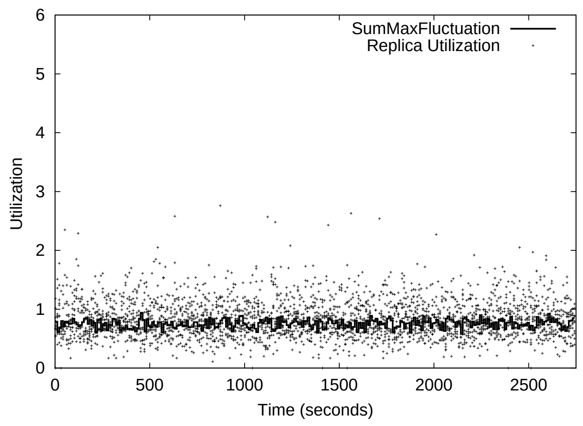

Figure 4.32: Replication Utilization versus Time for Max-Cap with extraneous load and an inter-update period of ten seconds.

take on the order of seconds to reach all allocating nodes, allocation decisions are continuously being made using stale information.

In the next experiment we use the same parameters as above but we change the update period to 10 seconds. Figures 4.32 and 4.33 show the utilization and overloaded percentages for this experiment. We see that the overloaded percentages increase only slightly while the overhead of pushing the updates decreases by a factor of ten. In contrast, when we perform the same experiment for Avail-Cap, we find that the overloaded query percentages for Avail-Cap increase from about 55 to more than 80% across all the replicas when the inter-update period changes from 1 to 10 seconds. However, this performance degradation is not so much due to the fluctuation of the extraneous load as it is due to Avail-Cap's tendency to oscillate when the request rate is greater than the update rate.

We purposely choose this scenario to test how Max-Cap performs under widely fluctuating extraneous load on every replica. We generally expect that extraneous load will not fluctuate so wildly, nor will all replicas issue new contracts every second. Moreover, we expect the inter-update period to be on the order of several seconds or even minutes, which further reduces overhead.

We can view the effect of extraneous load on the performance of Max-Cap as similar to that seen in the dynamic replica experiments. When a replica advertises

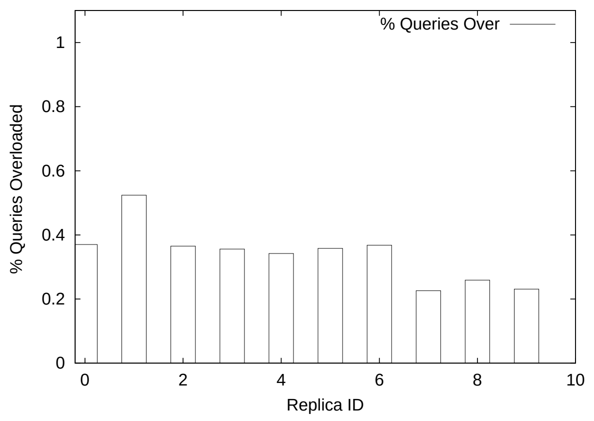

Figure 4.33: Percentage Overloaded Queries versus Replica ID for Max-Cap with extraneous load and an inter-update period of ten seconds.

a new honored maximum capacity, it is as if that replica were leaving and being replaced by a new replica with a different maximum capacity.

4.5 Summary

In this chapter we examine the problem of load-balancing in a peer-to-peer network where the goal is to distribute the demand for a particular content fairly across the set of replica nodes that serve that content. Existing load-balancing algorithms proposed in the distributed systems literature are not appropriate for a peer-to-peer network. We find that load-based algorithms do not handle the heterogeneity that is typical in a peer-to-peer network. We also find that algorithms based on available capacity reports can suffer from load oscillations even when the workload rate is as low as 60% of the total maximum capacities of replicas.

We propose and evaluate Max-Cap, a practical algorithm for load-balancing. Max-Cap handles heterogeneity, yet does not suffer from oscillations when the workload rate is within 100% of the total maximum capacities of the replicas, adjusts better to very large fluctuations in the workload and constantly changing replica sets, and incurs less overhead than algorithms based on available capacity since its reports are affected only by extraneous load on the replicas. We believe this makes Max-Cap a practical and elegant algorithm to apply in peer-to-peer networks.

Chapter 5

Conclusions

With the ever-increasing proliferation of Internet-connected hosts, the concept of peer-to-peer networks has emerged as a popular technology paradigm. Peer-to-peer networks allow users to perform useful computation or share content by exploiting the aggregate processing power and storage capabilities of widely-distributed hosts that live at the edge of the network. In particular, content distribution is a dominant application taking advantage of such networks.

A fundamental problem in supporting content distribution over a peer-to-peer network is that of locating content. Most peer-to-peer networks push index entries in response to a content search query. These index entries point to replica nodes in the network that serve the content of interest.

This thesis tackles two basic problems that arise from the content location problem. First, how can we improve upon the time it takes to resolve a search query once the location algorithm has been set? Second, how do we choose from amongst the index entries returned by a search query such that the demand for content is distributed fairly across the set of replica nodes serving the content?

To address the first problem, we develop CUP, a protocol for Controlled Update Propagation that maintains caches of index entries in peer-to-peer networks. CUP query channels coalesce bursts of queries for the same item, resulting in significant network traffic savings. CUP update channels asynchronously transport query responses and refresh intermediate caches. A key advantage of CUP is that it allows nodes to decide individually when to receive and when to propagate updates. To this end, we present a cost model based on the notion that a node has economic incentive to receive an update that is "justified". Using this cost model, we develop and compare two kinds of propagation cut-off policies used by nodes to determine when to stop receiving updates.

Through extensive experiments, we compare CUP against path caching with expiration using typical workloads that have been observed in measurements of real peer-to-peer networks. We demonstrate that through light book-keeping and incentive-based propagation cut-off policies, CUP controls and confines propagation to updates that are likely to be justified. We show that CUP greatly reduces average search query latency over path caching with expiration by as much as an order of magnitude. Moreover, we show that CUP overhead is typically many times compensated for by its savings in cache misses. The cost of saved misses can be two to 200 times the cost of updates pushed.

To address the second problem, we leverage CUP to study the problem of fairly balancing the demand for content across the replica nodes serving the content. Load-balancing is a problem that has been widely studied in the distributed systems literature. We argue that in the peer-to-peer context, this problem is unique for three reasons: 1) the allocation decision is completely decentralized with each peer node in the network making its own individual decisions, 2) peer-to-peer networks scale to very large sizes with thousands of nodes, precluding global coordination amongst the nodes as they make their allocation decisions, and 3) content replica nodes tend to be heterogeneous.

We show that previous load-balancing algorithms proposed in the distributed systems literature do not work well in the peer-to-peer network. In particular, we show that algorithms where the allocation decision is based on load do not handle the heterogeneity of content replica nodes. We also show that algorithms based on available capacity do handle heterogeneity, but can suffer from load oscillations as the workload rate approaches the total maximum capacities of replicas.

To overcome these problems, we propose and evaluate Max-Cap, an algorithm based on maximum capacities. We argue that Max-Cap is a practical load-balancing algorithm for a peer-to-peer network because it does not require frequent and timely updates carrying load-balancing information as previous algorithms do. We demonstrate the benefits of Max-Cap by comparing it against algorithms based on load and available capacity. We show that Max-Cap handles heterogeneity, yet does not suffer from oscillations when the workload rate is within 100 percent the total maximum capacities of the replicas. Finally we show that Max-Cap adjusts better to constantly changing replica sets and incurs less overhead than previous algorithms.

5.1 Future Work

There are several directions in which to expand this work. First, since CUP is not tied to any particular search algorithm and can be applied in both structured and unstructured networks, the most obvious direction would be to perform an extensive study of CUP over unstructured networks such as Gnutella [3]. The difference is that in an unstructured network, there is no single authority node per content item that holds the set of index entries pointing to the replica nodes that serve that item. Queries typically travel haphazardly or via random walks until they reach a replica node that serves the content of interest. The replica node responds with an index entry pointing to itself. This index entry travels along the reverse query path to the querying node. This means that a node issuing a search query may receive a set of index entries from more than one neighbor, each index entry having traversed the reverse query path from a content replica node to the issuing node. A node can therefore partake in several CUP trees per content item, each rooted at a replica node serving the content item.

Applying CUP in an unstructured network requires some additional bookkeeping. The issuing node needs to detect when it receives duplicate updates for the same index entry from two different neighbors so it can choose to cut-off its intake of updates from one of them. In essence, this is an extension to the detection mechanism already built into unstructured search protocols to detect duplicate query responses. Rather than dropping responses (as currently done), a node would push a clear-bit message to one of its neighbors (parents) to prevent wasting future bandwidth with redundant updates. Also, different incentive-based propagation cut-off policies would need to be explored since a node might be participating in multiple trees per content item.

Second, a natural extension to our load-balancing work would be to use CUP to support techniques for dynamic content replication and removal. This is particularly useful when the demand for a content item exceeds the total maximum capacities of the current set of replicas that serve that item. This would happen in the case of a flash crowd. In such a scenario, we can compare different techniques for creating new content replicas (e.g., as in [56]), and leverage CUP to propagate updates carrying index entries for new content replicas in the network quickly. Similarly, when the demand for a particular content item dies down and content replicas are removed, CUP can notify interested nodes as well.

Finally, we believe that CUP provides a general purpose framework for maintaining metadata (besides index entries) in peer-to-peer networks. We demonstrate an application of this in Chapter 4 where we leverage CUP to deliver metadata required for load-balancing of content demand across the replica nodes. We could leverage CUP to transport other kinds of metadata such as price information to enhance service negotiation between peer nodes.

Appendix A

Inv-Load and Max-Cap in a Homogenous Context

It should not surprise the reader that Inv-Load does not handle heterogeneity since the same load at one replica may have a different effect on another with a different maximum capacity. However, surprisingly it turns out that when replicas are homogenous, the performance of Inv-Load and Max-Cap are comparable.

In this set of experiments, there are ten replicas, each of whose maximum capacity we set at 10 requests per second for a total maximum capacity of 100 requests per second. Queries are generated according to a Poisson process with a lambda rate that is 80 percent the total maximum capacities of the replicas.

Figures A.1 and A.2 show a scatterplot of how the utilization of each replica proceeds with time when using Inv-Load with a refresh period of one time unit and Max-Cap respectively. Inv-Load and Max-Cap have similar scatterplots.

Figures A.3 and A.4 show for each replica, the percentage of queries that arrived at the replica while the replica was overloaded. Again, we see that Inv-Load and Max-Cap have comparable performance.

The difference is that Inv-Load incurs the extra overhead of 1 load report per replica per second. In a CUP tree of 100 nodes this translates to 1000 updates per second being pushed down the CUP tree. In a tree of 1000 nodes this translates to 10000 update per second being pushed. Thus, the larger the CUP tree, the larger the

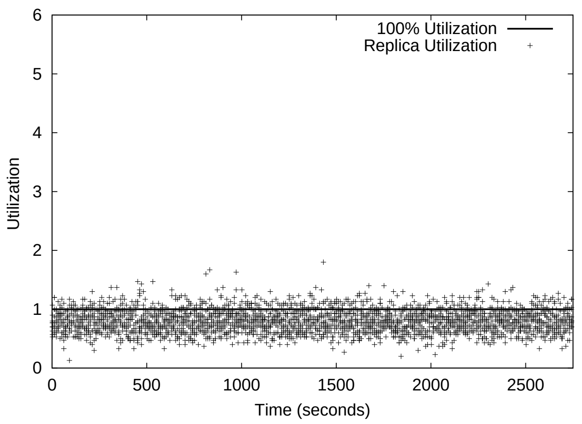

Figure A.1: Replica Utilization versus Time for Inv-Load with an inter-update period of one second and homogenous replicas.

overall network overhead. The overhead incurred by Inv-Load could be reduced by increasing the period between two consecutive reports at each replica. Increasing the period results in staler load reports. We find that when experimenting with a range of periods (one to sixty seconds), we confirm earlier studies [43] that have found that as load information becomes more stale with increasing periods, the performance of load-based balancing algorithms decreases.

We ran experiments with Pareto(α, κ) query interarrivals with a wide range of α and κ values (the Pareto distribution shape and scale parameters) and found that with homogeneous replicas, Inv-Load with a period of one and Max-Cap continue to be comparable. However, Max-Cap is preferable in these cases because it incurs no overhead.

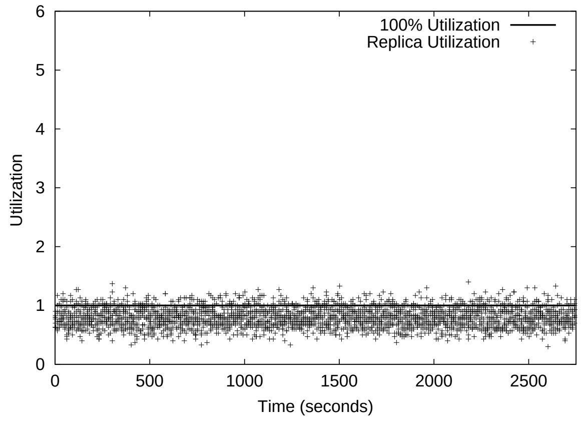

Figure A.2: Replica Utilization versus Time for Max-Cap with homogenous replicas.

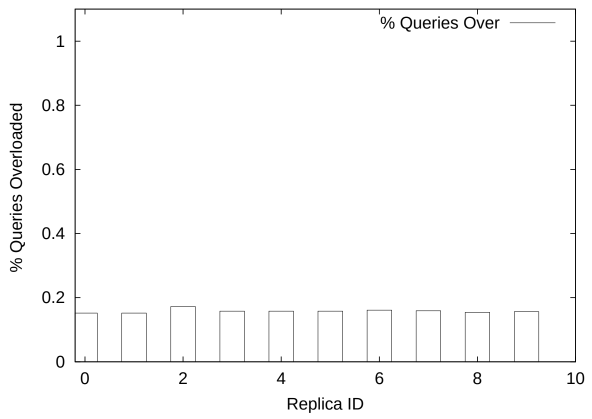

Figure A.3: Percentage Overload Queries versus Replica ID for Inv-Load with an inter-update period of one second and homogenous replicas.

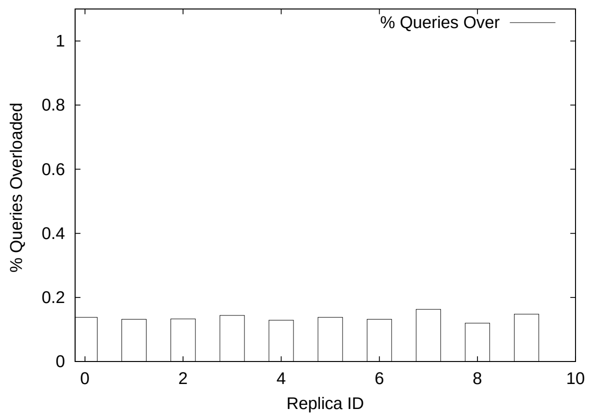

Figure A.4: Percentage Overload Queries versus Replica ID for Max-Cap with homogenous replicas.

Bibliography

[1] MojoNation. `http://www.mojonation.net/`.

[2] Scaling the Internet Web Servers. Cisco Systems Whitepaper, November 1997.

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

[4] What is P2P... And What Isn't. `http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html`.

[5] Foundry Networks ServerIron Server Load Balancing Switch, 1998. `http://www.foundrynet.com`.

[6] D. Andresen, T. Yang, V. Holmedahl, and O.H. Ibarra. SWEB: Towards a Scalable WWW Server on MultiComputers. In *IEEE International Symposium on Parallel Processing*, April 1996.

[7] D. Andresen, T. Yang, and O.H. Ibarra. Towards a Scalable Distributed WWW Server on Networked Workstations. *Journal of Parallel and Distributed Computing*, 42:91–100, 1996.

[8] Luis Aversa and Azer Bestavros. Load Balancing a Cluster of Web Servers Using Distributed Packet Rewriting. In *IEEE International Performance, Computing, and Communications Conference*, February 2000.

[9] Baruch Awerbuch, Yossi Azar, Amos Fiat, and Tom Leighton. Making Commitments in the Face of Uncertainty: How to Pick a Winner Almost Every Time. In *Twenty-eighth ACM Symposium on Theory of Computing*, 1996.

[10] Yossi Azar, Andrei Broder, Anna Karlin, and Eli Upfal. Balanced Allocations. In *Twenty-sixth ACM Symposium on Theory of Computing*, 1994.

[11] Pei Cao. Search and Replication in Unstructured Peer-to-Peer Networks, February 2002. Talk at `http://netseminar.stanford.edu/sessions/2002-01-31.html`.

[12] V. Cardellini, M. Colajanni, and P.S. Yu. Redirection Algorithms for Load Sharing in Distributed Web Server Systems. In *Proceeding of IEEE 19th International Conference on Distributed Computing Systems*, June 1999.

[13] V. Cardellini, M. Colajanni, and P.S. Yu. Geographic Load Balancing for Scalable Distributed Web Systems. In *Proceedings of Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Mascots)*, August 2000.

[14] R. Carter and M. Crovella. Server Selection Using Dynamic Path Characterization in Wide-Area Networks. In *INFOCOM*, 1997.

[15] Maurice Castro, Michael Dwyer, and Michael Rumsewicz. Castro, Maurice and Dwyer, Michael and Rumsewicz, Michael. In *Procedings of the IEEE International Conference on Control Applications*, August 1999.

[16] Yatin Chawathe. *Scattercast: An Architecture for Internet Broadcast Distribution as an Infrastructure Service.* PhD thesis, UC Berkeley, December 2000.

[17] Yatin Chawathe, Sylvia Ratnasamy, Scott Shenker, and Lee Breslau. Can Heterogeneity Make Gnutella Scale? May 2002. `http://www.research.att.com/~yatin/publications/`.

[18] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *DIAU*, July 2000.

[19] Edith Cohen and Haim Kaplan. Aging Through Cascaded Caches: Performance Issues in the Distribution of Web Content. In *SIGCOMM*, 2001.

[20] Edith Cohen and Haim Kaplan. Refreshment Policies for Web Content Caches. In *INFOCOM*, 2001.

[21] M. Colajanni, P.S. Yu, and D.M. Dias. Scheduling Algorithms for Distributed Web Servers. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 1997.

[22] Michele Colajanni, Philip S. Yu, and Valeria Cardellini. Dynamic Load Balancing in Geographically Distributed Heterogeneous Web Servers. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 1998.

[23] R. Cox, A. Muthitacharoen, and R. T. Morris. Serving DNS using a Peer-to-Peer Lookup Service. In *1st International Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002.

[24] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area Cooperative Storage with CFS. In *SOSP*, 2001.

[25] M. Dahlin. Interpreting Stale Load Information. In *International Conference on Distributed Computing Systems*, 1999.

[26] S. Dandamudi. Performance Impact of Scheduling Discipline on Adaptive Load Sharing in Homogeneous Distributed Systems. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 1995.

[27] Daniel M. Dias, William Kish, Rajat Mukherjee, and Renu Tewari. A Scalable and Highly Available Web Server. In *Proceedings of IEEE COMPCON'96*, 1996.

[28] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Spite Implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[29] A. Downey and M. Harchol-Balter. A Note on 'The Limited Performance Benefits of Migrating Active Processes for Load Sharing'. Technical Report UCB/CSD-95-888, UC Berkeley, November 1995.

[30] D. Eager, E. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, 1986.

[31] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Symposium on Operating Systems Principles*, 1997.

[32] Zornitza Genova and Kenneth J. Christensen. Challenges in URL Switching for Implementing Globally Distributed Web Sites. In *Workshop on Scalable Web Services*, 2000.

[33] R. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM Simulation on a Distributed Memory Machine. In *Twenty-fourth ACM Symposium on Theory of Computing*, 1992.

[34] E.D. Katz, M. Butler, and R. McGrath. A Scalable HTTP server: the NCSA prototype. *Computer Networks and ISDN Systems*, 27:155–164, 1994.

[35] C. Lu and S.M. Lau. An Adaptive Load Balancing Algorithm for Heterogeneous Distributed Systems with Multiple Task Classes. In *International Conference on Distributed Computing Systems*, 1996.

[36] Reinhard Luling and Burkhard Monien. A Dynamic Distributed Load Balancing Algorithm with Provable Good Performance. In *ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[37] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *16th ACM International Conference on Supercomputing (ICS)*, June 2002.

[38] Petros Maniatis, T.J. Giuli, and Mary Baker. Enabling the Long-Term Archival of Signed Documents through Time Stamping. Technical Report cs.DC/0106058, Stanford University, June 2001. `http://www.arxiv.org/abs/cs.DC/0106058`.

[39] Evangelos P. Markatos. Tracing a large-scale Peer-to-Peer System: an hour in the life of Gnutella. In *Second IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

[40] R. Mirchandaney, D. Towsley, and J. Stankovic. Analysis of the Effects of Delays on Load Sharing. *IEEE Transactions on Computers*, 38:1513–1525, 1989.

[41] R. Mirchandaney, D. Towsley, and J. Stankovic. Adaptive Load Sharing in Heterogeneous Distributed Systems. *Journal of Parallel and Distributed Computing*, 9:331–346, 1990.

[42] Michael Mitzenmacher. *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, UC Berkeley, September 1996.

[43] Michael Mitzenmacher. How Useful is Old Information? In *Sixteenth Symposium on the Principles of Distributed Computing*, 1997.

[44] S. Petri and H. Langendorfer. Load Balancing and Fault Tolerance in Workstation Clusters - Migrating Groups of Communicating Processes. *Operating Systems Review*, 29(4):25–36, Oct 1995.

[45] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *SIGCOMM*, 2001.

[46] Matei Ripeanu and Ian Foster. Mapping the Gnutella Network: Macroscopic Properties of Large-Scale Peer-to-Peer Systems. In *First International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

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

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

[49] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *NGC*, 2001.

[50] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of Multimedia Computing and Networking (MMCN)*, 2002.

[51] Kai Shen, Tao Yang, and Lingkun Chu. Cluster Load Balancing for Fine-Grain Network Services. In *International Parallel and Distributed Processing Symposium*, 2002.

[52] N. Shivaratri, P. Krueger, and M. Singhal. Load Distributing for Locally Distributed Systems. *IEEE Computer*, pages 33–44, Dec 1992.

[53] N.G. Shivaratri and P. Krueger. Two Adaptive Location Policies for Global Scheduling Algorithms. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 1990.

[54] Bujor Silaghi, Bobby Bhattacharjee, and Pete Keleher. Routing in the TerraDir Directory Service, 2002. `http://motefs.cs.umd.edu/terradir/`.

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

[56] Tyron Stading, Petros Maniatis, and Mary Baker. Peer-to-Peer Caching Schemes to Address Flash Crowds. In *1st International Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002.

[57] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM*, 2001.

[58] N. Vvedenskaya, R. Dobrushin, and F. Karpelevich. Queuing Systems with Selection of the Shortest of Two Queues: an Asymptotic Approach. *Problems of Information Transmission*, 32:15–27, 1996.

[59] Marc Waldman, Aviel D. Rubin, and Lorrie F. Cranor. Publius, A Robust, Tamper-Evident and Censorship-Resistant Web Publishing System. In *9th USENIX Security Symposium*, 2000.

[60] C.A. Waldspurger and W.E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov 1994.

[61] R. Weber. On the Optimal Assignment of Customers to Parallel Servers. *Journal of Applied Probability*, 15:406–413, 1978.

[62] W Winston. Optimality of the Shortest Line Discipline. *Journal of Applied Probability*, 14:181–189, 1977.

[63] Beverly Yan and Hector Garcia-Molina. Efficient Search in Peer-to-peer Networks. In *International Conference on Distributed Computing Systems*, 2002.

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

[65] Huican Zhu, Tao Yang, Qi Zheng, David Watson, Oscar H. Ibarra, and Terrence Smith. Adaptive load sharing for clustered digital library services. In *Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 1998.