COST-AWARE RESOURCE MANAGEMENT FOR DECENTRALIZED INTERNET SERVICES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Venugopalan Saraswati Ramasubramanian

January 2007

COST-AWARE RESOURCE MANAGEMENT FOR DECENTRALIZED INTERNET SERVICES

Venugopalan Saraswati Ramasubramanian, Ph.D.

Cornell University 2007

Decentralized network services, such as naming systems, content distribution networks, and publish-subscribe systems, play an increasingly critical role and are required to provide high performance, low latency service, achieve high availability in the presence of network and node failures, and handle a large volume of users. Judicious utilization of expensive system resources, such as memory space, network bandwidth, and number of machines, is fundamental to achieving the above properties. Yet, current network services typically rely on less-informed, heuristic-based techniques to manage scarce resources, and often fall short of expectations.

This thesis presents a principled approach for building high performance, robust, and scalable network services. The key contribution of this thesis is to show that resolving the fundamental cost-benefit tradeoff between resource consumption and performance through mathematical optimization is practical in large-scale distributed systems, and enables decentralized network services to meet efficiently system-wide performance goals. This thesis presents a practical approach for resource management in three stages: analytically model the cost-benefit tradeoff as a constrained optimization problem, determine a near-optimal resource allocation strategy on the fly, and enforce the derived strategy through light-weight, decentralized mechanisms. It builds on self-organizing structured overlays, which

provide failure resilience and scalability, and complements them with stronger performance guarantees and robustness under sudden changes in workload. This work enables applications to meet system-wide performance targets, such as low average response times, high cache hit rates, and small update dissemination times with low resource consumption. Alternatively, applications can make the maximum use of available resources, such as storage and bandwidth, and derive large gains in performance.

I have implemented an extensible framework called Honeycomb to perform cost-aware resource management on structured overlays based on the above approach and built three critical network services using it. These services consist of a new name system for the Internet called CoDoNS that distributes data associated with domain names, an open-access content distribution network called CobWeb that caches web content for faster access by users, and an online information monitoring system called Corona that notifies users about changes to web pages. Simulations and performance measurements from a planetary-scale deployment show that these services provide unprecedented performance improvement over the current state of the art.

BIOGRAPHICAL SKETCH

Venugopalan Ramasubramanian (Rama) hails from a small town in India called Salem. He completed his under-graduate education in Computer Science and Engineering at the Indian Institute of Technology Madras in 1999. After that, he moved to the United States of America to pursue his doctorate in Computer Science at Cornell University. Cornell stimulated his interests in building practical yet sound networks and systems. He received a special Master's in 2002 for his research on wireless networking protocols. Later, his interests shifted to building systems for wide area network applications, whose contributions are summarized in this doctoral dissertation. At the time of writing this thesis, Rama was working in Microsoft Research Silicon Valley.

To my parents, Venugopalan and Swarna, on whose sacrifice this thesis rests.

ACKNOWLEDGEMENTS

I started the doctorate program with a funny feeling that is similar to what one feels while riding a massively contorted roller coaster for the first time. After countless free falls and throw outs of disappointment and joy, I'm finally at the threshold of writing this note of appreciation for all who have followed the bumpy ride, cheered and supported my successes and failures, and wished me well all throughout. To understate, I am thrilled to be in this position; as unlike roller-coaster riders, not all who start out on a Ph.D. reach this point.

The foremost supporter of this dissertation is my adviser, Emin Gün Sirer. His contributions extend far beyond giving technical advice on my research; he played an instrumental role in developing my oral and written presentation skills, served as a cheer leader during the times of disappointment, and provided ecstatic encouragement to pursue wacky ideas that eventually turned into this thesis. I'm deeply indebted to him for turning me into a polished researcher.

In addition to my adviser, my thanks are due to several others who played a valuable role in my educational experience: my former adviser and thesis committee member, Ken Birman, with whom I worked for two years, other thesis committee members, Andrew Myers and Zygmunt Haas, collaborators, Daniel Mossé and Doug Terry, and several well-wishers from whom I have benefited, sometimes without my own knowledge. I owe them greatly for their generous support, guidance, and mentorship.

A large part of this thesis would not have been possible without the contributions of the following collaborators. I thank Yee Jiun Song for building and maintaining the CobWeb content distribution network, Ryan Peterson for implementing and running Corona, the web news monitoring system, and Hongzhou

Liu for collecting RSS workload traces. My thanks are also due to Yegnaswamy Sermadevi and Amar Sapra for patiently explaining the nuances of mathematical optimization.

Finally, I cannot understate the role played by my friends, whose presence was a great source of inspiration, fulfillment, and entertainment. While I refrain from listing their names for the sole fear of missing out a few, I owe greatly to those who brought me out of deep personal crisis, transformed my life, and made my stay at Cornell the most enriching part of my life. I also thank my long-standing friends, teachers, and family members for their valuable support and wishes.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Decentralized network services play an increasingly important role. Decentralization offers three significant benefits for running network services, namely, improved performance as geographic distribution enables clients to contact closer servers, high availability in the presence of network and node failures as there is no single point of vulnerability, and ability to handle increases in workload and number of clients by adding more servers. These advantages have made Internet services to move more and more towards decentralized architectures. These services are typically deployed in two ways: as servers distributed in a wide area network at geographically distant locations or as a cluster of servers that are hosted in a single location. Prominent examples of decentralized Internet services include name systems, such as the Domain Name System (DNS), which enable users to locate resources identified by names, Content Distribution Networks (CDNs) such as Akamai and Digital Island, which replicate Web content around the globe so that users can fetch content faster, and publish-subscribe systems, which facilitate users to subscribe for content of their interest and receive notifications when new content matching their interest is published.

In practice, resource limitations pose fundamental challenges for decentralized network services. Allocating more resources such as server machines, storage units, and network bandwidth can improve performance; however, the performance improvement comes at a cost as resources tend to be scarce or expensive. For example, a content distribution network trying to consistently replicate all web content at every server in the world would require substantial amount of storage space

and network bandwidth. Such naive approaches to resource allocation driven by over-provisioning of resources quickly become impractical at large scales.

Judicious utilization of resources is critical for achieving large gains in performance in a distributed system. Attaining high performance using minimal resources translates to huge savings in monetary terms by decreasing equipment required, energy consumption, as well as administrative costs. Similarly, when system resources are scarce, deriving maximum performance from the limited resources can serve the users significantly better.

Most decentralized services today depend on less-informed, heuristic-based techniques for resource allocation. Heuristics typically make decisions about allocating resources using limited knowledge about resource availability and workload characteristics. Consequently, heuristic-based techniques often have fundamental limitations. Their effectiveness is driven by the workload and deviations from assumed workload characteristics may lead to poor performance. They provide no assurance or guarantees about performance.

The limitations of less-informed heuristics are best illustrated in *caching*, the most widely-used heuristic in distributed systems. Caching in decentralized services such as CDNs is typically governed by the following heuristic: nodes opportunistically store objects they fetch anticipating that future requests for the stored objects can be served from the cache. This form of *opportunistic caching* works well when the workload satisfies three conditions: a large number of requests are destined for a small number of highly popular objects, objects have similar sizes, and objects are static or do not change often.

Such assumptions rarely hold in practice for a large class of applications. For instance, the workloads for name systems, CDNs, and publish-subscribe systems are

characterized by heavy-tailed popularity distributions, where only a small number of lookups go for very popular objects and the remainder of the lookups go for a large number of unpopular objects forming a long, heavy tail in the popularity distribution. Content sizes can vary from a few bytes to gigabytes and update times may range from seconds to no updates at all. Consequently, previous attempts to apply opportunistic caching for these applications provides less-than -adequate improvement in performance. Moreover, opportunistic caching is driven by the workload and cannot make use of any additional network or storage resources available to further improve performance. These observations have led researchers to believe that opportunistic caching is not well-suited for the Web [18, 155].

The key contribution of this thesis is a principled approach to perform informed, cost-aware resource allocation in large-scale, decentralized systems. This approach analytically models resource-performance tradeoffs as mathematical optimization problems and computes near-optimal solutions on the fly. The analytical model captures the tradeoff between resources consumed and performance derived based on characteristics of the application workload. Mathematical optimization, then, finds an efficient resource allocation strategy to meet system-wide performance goals, which take the form of constraints in the optimization problem.

The approach presented in this thesis provides applications with fine-grained control over cost and performance. An application can set and attain system-wide performance goals. Such performance goals can take two forms; 'achieve a target level of performance using the minimum amount of resources' or 'achieve the best performance using the limited amount of resources available.' The former enables applications to meet performance targets, such as those specified in service agreements, at low cost. The latter enables applications to derive large performance

gains out of available resources.

The presented approach is well-suited for a wide variety of applications with different workload characteristics and performance requirements. For example, a lookup service may use this approach to achieve low lookup latency while reducing network bandwidth required for maintaining replicated copies of data. A CDN system may use this approach to achieve high rate of cache hits without exceeding storage space available at each server. And, a publish-subscribe system may apply this approach for an entirely different purpose of detection events and sending timely update notifications to users.

Clearly, this thesis is not the first to apply mathematical optimization for resource allocation. Several researchers in the past few decades have explored analytical models to express cost-performance tradeoffs and devised techniques to perform resource allocation for different applications, both in theory and in practice [35, 48, 75, 87, 88, 107, 108, 116, 145, 146]. The primary contribution of this thesis lies in demonstrating the effectiveness of optimization-based resource allocation in distributed, Internet-scale, real-world systems. While prior approaches for optimization-based resource allocation typically offer centralized solutions or involve algorithms with high run-time complexity, this thesis presents fully decentralized, scalable, and adaptable techniques to perform informed resource allocation on the fly.

This thesis achieves high performance, decentralization, and scalability through the use of *structured overlays*. A structured overlay [71, 96, 97, 121, 128, 138, 161] is a self-organizing network that has a regular, well-defined topology (such as a ring, hyper cube, or torus) with uniform node degree (number of neighbors) and bounded diameter (maximum distance between two nodes). It provides two key

features that are well-suited for decentralized services: namely the ability to detect failures and take remedial actions automatically so that the service is constantly available to the users and to increase the capacity of the system seamlessly without disruptions. This thesis complements these two properties by enabling structured overlays to meet application specific performance targets efficiently.

The rest of the chapter describes the contributions of this thesis in greater detail.

1.1 Contributions

This thesis makes three major contributions. First, it presents a principled approach to resolve cost-performance tradeoffs in large scale decentralized services by posing them as concise optimization problems. Second, it describes light-weight, distributed mechanisms to determine and enforce near-optimal resource allocation strategies. Finally, it shows how this approach of optimal resource management improves the state of the art in three different decentralized applications, namely, name services, content distribution networks, and online information monitoring systems.

1.1.1 Analytical Modeling

Cost-performance tradeoffs in distributed systems arise in the form of resource allocation problems. For example, consider the practice of caching objects in lookup services. The key question in caching is which object should be cached at which node. Clearly, caching an object at all the nodes provides the best response time to clients requesting that object. However, limited capacity in the system to store and update cached objects prohibits all objects from being cached at all nodes.

Consequently, system resources need to be allocated carefully between the objects.

This thesis takes the fundamental approach that the tradeoff between cost and performance can be expressed as a mathematical optimization problem. The desired system-wide performance targets can be expressed as constraints to the optimization problem. Deriving the solution to this optimization problem, that is, the optimal decision to determine which object should be hosted at which node, can lead to efficient, well-informed resource allocation in practice. Such cost-performance optimization problems can take two forms depending on application goals: 'minimize total cost such that global performance meets a target' or 'maximize global performance such that total resource consumption is within a bound.'

The cost-performance tradeoffs depends on the characteristics of the objects in the system. For instance, content size determines the amount of storage or memory space required for an object; update rate and content size determine the amount of network bandwidth required to keep copies of an object consistent. Similarly, the popularity of an object determines the performance benefit obtained by caching it. Caching popular objects with small sizes and low update rates provides more benefits than caching large, unpopular objects that change often.

These complex tradeoffs between objects with wide ranging characteristics renders cost-performance optimization difficult. The optimization problems described earlier depend on the number of nodes N and number of objects M with a complexity of $O(MN)$ to express the problem. In a large scale network service with thousands of nodes and millions of objects, the $O(MN)$ complexity just to pose the problem is unmanageable.

The key insight behind this thesis is that coarse-grained yet structured resource

allocation renders cost-performance optimization practical. This thesis takes advantage of *structured overlays* to reduce the complexity of resource allocation problems. The intuition is that instead of making resource allocation decisions for each and every node in the system, decisions can be made for groups of nodes in well-defined regions of the structured overlay. To start with, each object is hosted at one single node called *owner*. If it is desirable to increase resources for an object, it is allocated to all the neighboring nodes of the owner. This process is extended so that resource allocation is performed in groups of nodes defined by the distance, or the number of hops, from the owner node.

The above process of structured, systematic resource allocation leads to concise and elegant analytical models for resource-performance tradeoffs. Resource allocation is now reduced to deciding at which level or distance from the owner node an object should be hosted. Thus the complexity of the optimization problem is reduced from $O(MN)$ to $O(MK)$, where K is the diameter of the overlay. Since the diameter of structured overlays is sub-linear in N, typically logarithmic or even lower, the complexity is considerably lower and easier to manage. Moreover, uniform distribution of node degree or number of neighbors makes analytical modeling tractable. For instance, the cost of caching an object at level d depends on the number of nodes at distance d from the owner and can be modeled as b^d, where b is the node degree. Similarly, the lookup latency for locating an object cached at level d depends on the number of hops required to reach a node with a replica and can be modeled as $K - d$, where K is the diameter.

This thesis presents analytical models to express different kinds of resource-performance tradeoffs. For lookup services, it considers average response times for lookups, for CDNs, average cache hit rates, and average update detection time for

online data monitoring systems. For each of these applications, it models cost for different types of resources such as network bandwidth, memory consumption, and computational load.

1.1.2 Decentralized Optimization

Deriving the optimal resource allocation strategy based on the above models to meet system-wide performance goals poses three major challenges. First, resource-performance optimization problems are typically NP-hard rendering them computationally infeasible in large systems. Second, since parameters of the optimization problem such as workload and object characteristics are distributed throughout the system, it is difficult for any node to compute the globally optimal resource allocation strategy using local information. Finally, workload characteristics such as popularity may change continuously rendering the previously computed resource allocation strategies outdated.

This thesis develops fast, decentralized, and adaptable techniques to determine and enforce efficient resource allocation strategies in practice. First, it presents two techniques to determine near-optimal solutions for typical cost-performance optimization problems at a single node. Second, it presents low-overhead mechanisms to make the optimization process completely decentralized. Third, it presents efficient methods to detect and adapt to changes in workload and object characteristics. These techniques enable the system to meet global performance targets through independent and local decisions.

The first optimization technique derives the solution to the optimization problem as a closed-form mathematical formula. This analytical derivation is facilitated by making simplifying assumptions on workload and object characteristics. This

technique models popularity as a power-law or Zipf distribution, an analytically tractable probability distribution with a single parameter called *exponent*; several Internet applications, such as DNS and Web, follow Zipf popularity distributions. It assumes that object characteristics are assumed to be uniform ignoring the effects of size and update rate on resource consumption. The advantage of this fully analytical technique is that computing the solution requires only one parameter to be estimated, namely the exponent of the Zipf distribution.

The second optimization technique is based on a fast and accurate numerical algorithm developed in this thesis and generalizes the approach to a broader class of applications, which may have any distribution of workload and object characteristics. It explicitly takes into account all workload and object characteristics and provides substantial decrease in resource consumption compared to the analytical technique through better-informed resource allocation. The algorithm has a small run time complexity of $O(MK \log(MK))$ for a system with M objects and diameter K and enables the optimization to be reapplied frequently so that the system can adapt itself to changes in workload characteristics. It is near-optimal and provides solutions accurate to one object per node. That is, resource consumption at each node exceeds from the optimum by at most one object.

The optimization techniques proposed above require characteristics of the global system-wide workload and objects. Since it is expensive to make the entire information available at each node, this thesis develops techniques for coarse-grained aggregation without significant loss of accuracy. This technique treats objects with similar characteristics as a single clustered entity, averages their characteristics, and performs resource allocation on the clustered entity. Through aggregation of characteristics of clustered entities across the system, each node gathers coarse-

grained information about entities not hosted by itself. Aggregating coarse-grained characteristics enables each node to apply the solution techniques independently and still determine a globally consistent resource allocation strategy.

Simultaneously, nodes keep track of changes to workload and object characteristics. This thesis presents a unique method for monitoring changes to object popularity when the object is distributed on several nodes. This method seamlessly combines two different ways to estimate popularity, counting number of accesses in a time interval and measuring inter-arrival time between accesses. The result is a mechanism that is fast and efficient; it quickly detects sudden increases in popularity, which may occur during a flash crowd or denial of service attack; and incurs an overhead proportional to object popularity, thus requiring little resources to monitor the large number of unpopular objects.

The key property of the techniques and mechanisms presented in this thesis is decentralization. No single node in the system takes sole responsibility for resolving tradeoffs and becomes a central point of failure. Each node makes decisions independently and enforces the decisions only on the locally hosted objects; expensive techniques for distributed consensus and agreement are not employed. Moreover, the network overhead for system-wide aggregation of global information is small and bounded. Nodes only communicate with their neighbors for aggregation and the size of messages exchanged is bounded by using only a constant number of clustered entities for each level. Finally, the techniques for monitoring workload characteristics are light-weight and seamlessly handle both highly popular and less popular objects efficiently.

I have implemented the above techniques in the form of an framework called *Honeycomb*. Honeycomb provides applications the ability to resolve resource-

performance tradeoffs on structured overlays. It makes minimal assumptions about the underlying overlay, adds little additional overhead, and provides an extensible interface that accommodates many new applications.

1.1.3 Applications

The combination of optimization-based resource management and structured overlays provides new opportunities to build robust and scalable Internet services with unprecedented performance. This thesis presents three different applications built using this principled approach: first, a new naming system for the Internet called *CoDoNS* that provides a safety net and a possible replacement for the Domain Name System, the current service for looking up host names, second, a new content distribution network called *CobWeb* that provides high cache hit rates and low latency access to web content, and finally, an online information monitoring system called *Corona* that monitors web sites for changes and notifies users about updates that happen in web sites of interest to the users.

This thesis examines the legacy systems that currently provide the above services, illustrates how their use of heuristics for resource allocation leads to fundamental drawbacks, and shows through deployed prototypes that the approach developed in this thesis significantly improves the state of the art.

CoDoNS: Naming System

Identifying the location of a network resource, that is, a named entity such as an Internet host, is an essential predecessor to communication. Currently, this critical task of translating human-friendly resource names to network addresses is provided by the Domain Name System (DNS). In addition to name-address

translation, DNS also serves as a general-purpose database for mapping names to many different kinds of associated data, including mail servers, public keys, and service configurations.

DNS operates through static, hierarchical partitioning of the namespace and decentralized management. The DNS namespace is partitioned into several domains, each of which may be further divided into subdomains. The owner of a subdomain holds the responsibility for both managing the namespace and resolving names under that subdomain. Thus a name lookup in DNS involves traversing this hierarchical namespace until a server for the queried subdomain is reached. DNS reduces the lookup latency of this iterative process through wide-spread use of opportunistic caching.

While DNS sustained the growth of the Internet for two decades, recent increases in malicious behavior, explosion in client population, and the need for fast service relocation has exposed fundamental drawbacks in the design of DNS. These drawbacks arise in the form of long latencies for resolving queries and propagating updates to DNS bindings, poor resilience to server and network failures, and high vulnerability to malicious attacks. This thesis examines these drawbacks though a large-scale survey of DNS based on over half a million unique domain names and hundred thousand namesevers.

Poor Performance: The hierarchical architecture of DNS leads to large latencies for name resolution and update propagation. Recent studies [153, 73, 79] show that DNS lookup time contributes more than one second for up to 30% of web object retrievals. The heuristic-driven, opportunistic caching employed by DNS is not effective for the DNS workload, which is characterized by heavy-tailed Zipf distributions. The use of short expiry times for cached mappings, to facilitate easy

service relocation, further reduces the effectiveness of caching in DNS. Moreover, manual configuration errors introduce bad delegations to non-existent servers and create latent performance problems [110, 103].

Unanticipated changes to DNS bindings do not propagate quickly to clients as ad hoc, opportunistic caching prohibits fast propagation of updates. DNS relies on timeout-based invalidations of cached copies since it is expensive to keep track of the locations of replicas in an opportunistic cache. This weak cache consistency implies that changes may not be visible to clients for long durations, effectively preventing quick service relocations in response to emergencies.

Low Availability: Despite the large amount of collective resources in DNS, the number of servers hosting the bindings of any given name is typically small. The failure, compromise, or overload of these servers that act as a bottleneck can lead to failed lookups, malicious take overs, and Denial of Service (DoS) attacks. Approximately 80% of the domain names are hosted by just two servers, and a small 0.8% by only one. At the network level, all servers for 32% of the domain names are connected to the Internet through a single gateway.

Moreover, the hierarchical architecture of DNS inherently poses a load imbalance between the servers at the top level of the hierarchy and the servers at the leaves. Consequently, the root and top level servers are frequent targets of DoS attacks, where a recent DoS attack on the root DNS servers crippled nine of thirteen root servers [166].

Security Risks: DNS is highly susceptible to malicious attacks due to imprudent delegation of authority for serving name address bindings. Name owners in the DNS designate certain servers as *authoritative* for hosting their name bindings. These delegations are based on names rather than network addresses and may

trigger extraneous lookups during name resolution. This name-based delegation of authority leads to subtle and complex dependencies between DNS servers. For example, the authority for hosting *cornell.edu* is delegated to servers in the domain *rochester.edu*, which is delegated to servers in the *wisc.edu* domain and in turn delegated to *umich.edu*. These dependencies lead to complex trust relationships and impose a high risk of compromise.

The large-scale measurement survey mentioned above quantifies risks posed by transitive delegation of authority. It shows that the extent of dependencies extend to as high as 46 servers on average and well over hundred servers for a significant 8% of the domains. About 125 servers control a disproportionate 10% of the namespace, where 25 of these critical servers are operated by educational institutions, which may not have adequate incentives and resources to enforce security.

This thesis presents a new architecture called Cooperative Domain Name System (CoDoNS) to replace DNS. The key principle behind CoDoNS is to separate namespace management from name resolution. This separation enables CoDoNS to retain the successful hierarchical and decentralized legacy DNS namespace, while providing name resolution service through a flat, peer-to-peer architecture layered on Honeycomb. This architecture based on the structured overlays and cost-aware resource allocation leads to high performance, availability, and data integrity as follows:

- **High Performance**: CoDoNS provides low latency name resolution and fast propagation of updates through structured, proactive caching driven by the cost-aware resource allocation approach outlined earlier. CoDoNS targets low average lookup latency while minimizing the total network bandwidth

consumed. It pushes updates proactively to all the cached replicas taking advantage of structured caching to determine their locations and eliminates the need for soft, timeout-based mechanisms.

- **Availability**: Any node in CoDoNS can serve any binding. Consequently, CoDoNS can tolerate large number of failures in the system. Self-organizing overlays ensure that CoDoNS transparently heals around network and node failures and no single node becomes a bottleneck in the system.

- **Data Integrity**: CoDoNS preserves data integrity and alleviates the impact of node compromises by supporting the DNSSEC [50] standard. Signed cryptographic certificates provided by name owners enables clients to verify the authenticity of bindings fetched from CoDoNS.

A prototype of CoDoNS is deployed on the planetary-scale test bed called PlanetLab. Performance measurements from this deployment show that CoDoNS provides a median response time of a few milliseconds, a factor of two decrease in the average response time compared to legacy DNS, and can adapt itself quickly to meet sudden surges in workloads, including flash crowds and DoS attacks.

CobWeb: Content Distribution Network

The World Wide Web has emerged as the primary means of information sharing in the Internet. Naturally, considerable effort has been spent to improve the user perceived latency for accessing web content. The primary technique used to improve lookup performance on the web has been caching. Web caching solutions to date have been deployed in two different settings: passive caching driven by client workload and active caching managed by content distribution networks.

Passive web caches are opportunistic; similar to DNS caches, they are driven entirely by the client workload. That is, they fetch objects from the Web on behalf of clients, cache them locally, and reply with cached copies when available. Passive web caches may reside on Web browsers at the clients exploiting temporal locality within the clickstream of a single user or at a common point of access to many users, such as near network gateways that connect institutions to the Internet. These caches may also cooperate with each other to exploit common interests of independent users. In contrast to client-driven passive caches, content distribution networks perform active replication, where copies of web objects are placed a priori in geographically distributed nodes to enable quicker access to the content for their clients.

Resource limitations pose fundamental challenges to both active and passive web caches. The caching solutions advocated to date are predominantly driven by heuristics to decide which object to cache where. Passive caches sidestep the problem of resource management by relying on the client access pattern to dictate where objects are cached. Active caches, on the other hand, use more sophisticated heuristics that incorporate server load as well as object size and update rate into caching decisions. Overall, the literature on web caching is replete with several different heuristics [152, 2, 124, 25, 159, 106, 85, 53, 14], which use different intuition, make different assumptions, and consider different factors to resolve the resource-performance tradeoff.

However, several measurement and simulation studies have shown that less-informed, heuristic-based caching for the web provided limited performance gains. While some heuristics perform marginally better than others depending on the circumstances, the overall cache hit rate is typically lower than 40% [18, 156]. This

limitation has been shown to stem from the heavy-tailed nature of web popularity distribution, where a large percentage of queries go for less popular objects. Thus, the conventional wisdom about web caching has been negative.

This thesis shows that high hit rates can be achieved in web caches at low overhead. It does so through a deployed content distribution network called CobWeb based on the principles of cost-aware resource management outlined earlier. CobWeb is a distributed cooperative cache, in the sense that it is composed of a peer-to-peer network of nodes that cooperate and share cache data to provide a low latency CDN to the users. However, unlike passive client-driven cooperative caches, it uses proactive caching, where objects are cached to nodes in anticipation of future demand. Similar to CoDoNS, CobWeb is layered on a self-organizing structured overlay, and uses the Honeycomb framework to manage resources judiciously.

CobWeb provides unprecedented performance improvement over heuristics by explicitly handling heavy-tailed distributions. CobWeb can meet a high target for cache hit rate so that it a large number of requests can be answered locally with little delay. CobWeb achieves the target using less memory and bandwidth resources. Alternatively, CobWeb can be set to maximize the utility of caching with a limit on memory and bandwidth consumption. In both configurations, CobWeb automatically learns the workload and object characteristics and manages resources judiciously to achieve high performance at low cost.

CobWeb is deployed on PlanetLab, a global infrastructure for distributed systems research, and operated as an open-access CDN. It provides the same interface to clients as existing Web caches and CDNs. It uses DNS redirection to dynamically redirect a client to the closest CobWeb server [158]. It serves about 10

million requests a day at the time of writing of this thesis. Simulations and measurements show that it provides significant improvement in latency over heuristic based caching solutions.

Corona: Online Data Monitoring System

The naming system and the CDN discussed in the previous sections support a pull mode of content dissemination, where clients issue requests to fetch content from servers. However, there is an emerging need for systems that directly deliver content to the users. Such asynchronous event notifications are provided by publish-subscribe systems, which enable clients called subscribers to register for events of their interest on the Internet and receive notifications when the events occur. Publish-subscribe systems are crucial for monitoring the large volume of data sources, such as web pages, online databases, and sensor systems.

The fundamental challenge in monitoring such online data sources is that they typically only provide a pull-based interface. Thus, clients are forced to explicitly query the data source to find new content or updates to old content. This need to provide asynchronous content delivery for traditional data sources has led to a new industry standard where data is published as *feeds* in well-defined XML-based formats, such as RSS and Atom, which are parsed by automated tools called *feed readers*. Feed readers poll the content servers periodically on behalf of the user and report updates detected to the user.

Publish-subscribe through uncoordinated polling, as in the current feed readers, suffers from poor performance and scalability. Subscribers do not receive updates quickly, as the polling period poses a fundamental limit to the update detection time. Clients are tempted to poll at faster rates in order to detect updates quickly.

Consequently, content providers have to handle the high bandwidth load imposed by clients, each polling independently and multiple times for the same content. Moreover, the workload tends to be "sticky;" that is, users subscribed to popular content do not unsubscribe after their interest diminishes, causing a large amount of wasted bandwidth.

This thesis applies the cost-aware resource allocation framework outlined earlier for building an online data monitoring system. This system, called Corona, provides a high-performance update notification service for the web without requiring any changes to the existing infrastructure, such as web servers. Instead of relying on naive, independent polling, Corona allocates multiple nodes to poll for a feed cooperatively so that updates can be detected faster and shared with clients.

The key resource tradeoff in Corona involves bandwidth and update latency. Clearly, polling data sources more frequently will enable the system to detect and disseminate updates earlier. Yet polling every data source constantly would place a large burden on publishers, congest the network, and potentially run afoul of server-imposed polling limits that would ban the system from monitoring feeds or web pages. The goal of Corona, then, is to maximize the effective benefit of the aggregate bandwidth available to the system, while remaining within server-imposed bandwidth limits. This thesis explores three different modes of operation for Corona: how to minimize update latency while ensuring that the average load on publishers is no more than what it would have been without Corona, how to minimize bandwidth consumption in order to achieve a targeted update latency, and how to ensure that the load is more fairly balanced across channels with different update characteristics.

Corona, just like CoDoNS and CobWeb, is deployed on PlanetLab and made

available for public use. Evaluation of this deployment shows that Corona achieves an order of magnitude improvement in update performance. In experiments parameterized by real RSS workload collected at Cornell [92] and spanning 80 PlanetLab nodes and involving 150,000 subscriptions for 7500 different channels, Corona clients see fresh updates in intervals of 45 sec on average compared to legacy RSS clients, which see a mean update interval of 15 min. At the same time, Corona issues no more polling requests to the content servers than issued by the legacy RSS clients.

Summary

The above applications indicate that a principled approach for resource allocation based on mathematical optimization can be practical, efficient, and well-suited for a large class of distributed applications. Each application describe above employed this general approach to meet different performance goals, namely, lookup latency, cache hit rate, and update detection time. And in each case, the approach led to substantial improvement in performance over the state of the art, which relies on ad-hoc heuristics for resource allocation. Overall, the thesis shows that a well-informed, cost-aware approach to resource allocation leads to high performance decentralized network services.

1.2 Outline

The rest of this thesis describes these contributions in greater detail. Chapter 2 presents the optimization-based approach for resource allocation and describes the analytical models, optimization algorithms, and distributed mechanisms developed for near-optimal resource allocation on structured overlays. Chapter 3, 4, and 5

respectively describe the three services, namely CoDoNS, CoBWeb, and Corona, built using this approach. For each service, the chapters describe the current state of the art, present a new architecture based on structured overlays and near-optimal resource allocation, and provide a detailed evaluation of the new architecture in comparison to the state of the art through simulations and real-world deployment. Chapter 6 provides a summary of other research work related to the topics discussed in this thesis. And, finally, Chapter 7 summarizes the contributions and discusses the implications of the contributions of this thesis.

Chapter 2

Resource Allocation through Mathematical Optimization

This chapter presents the principled approach outlined earlier for resolving resource-performance tradeoffs in distributed systems. This principled approach is based on mathematical optimization, where the key cost-performance tradeoff is posed as an optimization problem with constraints to represent system-wide performance targets of the application. The application-specific performance targets can then be satisfied efficiently by finding near-optimal solutions to the constrained-optimization problem. This chapter derives analytical models to capture resource-performance tradeoffs, presents techniques for finding near-optimal solutions to the resource-performance optimization problems, and shows how these techniques can be implemented efficiently in practice.

This thesis primarily focuses on critical, performance-demanding decentralized services. The unifying feature of these applications is that expensive resources such as bandwidth, memory, and computational power distributed on multiple nodes need to be allocated between application-level entities such as name-data mappings, web objects, or data feeds. The techniques developed in this chapter enables decentralized applications to make judicious resource allocation decisions. While this chapter focuses on the driving applications introduced in Chapter 1, the techniques developed here are general and can be applied to other distributed applications with similar tradeoffs between resource utilization and performance.

Resource allocation in decentralized network services is governed by decisions to a fundamental question: 'which nodes host which application-level entities?' As

mentioned earlier, the total number of decision variables in the above optimization problems is MN for a system with M objects and N nodes and can be intractable in large-scale decentralized systems. While vast amount of literature on theoretical analysis and solution techniques for decision making problems exists [35, 48, 75, 87, 88, 107, 108, 116, 145], the scale and complexity of decision making problems in real-world applications creates a need for exploring new techniques that work efficiently in practice.

This thesis takes advantage of *structured overlays* as an underlying substrate to facilitate efficient resource-allocation in large-scale decentralized systems. The next section provides a brief overview of structured overlays before subsequent sections describe the core approach of optimization-based resource allocation.

2.1 Overview of Structured Overlays

Overlays are just distributed systems composed of nodes called *peers* connected by a communication network. They are called overlays because they form a superimposed virtual network on top of the underlying network. That is, the communication path between two nodes is not always the direct path provided by the underlying network, but can be an indirect path passing through other peers in the overlay. Overall, the peer nodes in a overlay system form a topology, where each node communicates directly to a few nodes called *neighbors* and reaches any other node by following multi-hop indirect paths in the topology.

Structured overlays are a class of overlays that have a regular, well-defined topological structure. The neighbors of a node are carefully chosen in order to meet certain criteria so that the topology has predictable properties. For example, nodes can be organized to form a two-dimensional torus so that the maximum length of

an overlay path between any two nodes can be analytically bounded by the square root of the number of nodes. Similarly, each node has a uniform *node degree*, the number of neighbors, so that the average node degree in a structured overlay can be analytically bounded. Analytical tractability of node degree and diameter in structured overlays facilitates bounded delays for basic overlay operations such as locating a node, adding a new node, and repairing the overlay during a node failure.

The key property that makes structured overlays an attractive substrate for distributed systems is self organization. The overlay automatically forms a regular topology during node failures and joins without external, manual intervention. Thus when a new node joins the system, it can find its place in the topology, add new neighbors, and become connected to existing nodes automatically. Similarly, when a node fails, its neighbors detect the failure automatically and find new neighbors without disrupting the applications layered on the overlay. The regular topology enables these protocols to be lightweight. Thus, structured overlays provide high failure resilience and scalability through self organization and analytically tractable topology.

Decentralized services are layered on a structured overlay as follows. Each application-level entity, called *object*, has a unique identifier and a unique location in the system called the *home node*. Structured overlays perform *routing* to locate the home node of an object. Routing is the process of tracing a path from any node in the system to the home node by following direct paths along neighbors of intermediate nodes.

The resource allocation techniques presented in this chapter is applicable to a wide range of structured overlays that have been proposed to date. While the

suitable overlays include CAN [121], Chord [138], Kademlia [97], Pastry [128], SkipNet [71], Tapestry [161], and Viceroy [96], this chapter describes optimization using Pastry as an example. Chapter 6 provides a detailed overview of recent advances in structured overlays.

2.1.1 Pastry

Pastry organizes the system in a ring topology by assigning each node a unique identifier drawn from a large circular space of numbers. Each identifier can be expressed as a string of digits and the number of digits in each identifier is fixed. The ring is defined based on proximity in the identifier space, that is, each node is connected to neighbors with the next highest and next lowest identifier. Modular arithmetic for computing proximity ensures that the identifier space wraps around itself and forms a ring.

Objects are also assigned unique identifiers from the same identifier space. The home node of an object in Pastry is the unique live node whose identifier is the closest to the object, closeness being measured in the identifier space. Object identifiers are typically assigned through consistent hashing [81], that is, a one-way hash function is used to map the name of the object to the identifier space. A large identifier space, typically of 128 or 160 bits, is used to ensure that two different objects do not hash to the same identifier. Assigning identifiers through consistent hashing provides two advantages. First, the identifier of an object can be independently generated by any node without communicating with other nodes. Second, it balances the number of objects managed by each home node, by spreading the objects uniformly in the system.

Any node in the overlay can forward lookup messages for an object to its home

node using the above mapping between objects and home nodes. A naive way to route such messages is to forward it along the neighbors in the ring. However, this approach takes $O(N)$ hops in the average case in a system of N nodes and is consequently inefficient. Instead, Pastry routes messages using a technique called *prefix matching* [114]. With prefix matching, each node forwards a message to a node whose identifier has one more matching prefix digit with the object identifier than the current node. Prefix matching reduces the search space in the system by a factor of B, the base of the identifier space, during each iteration and finds the home node in $O(\log_B N)$ hops on average.

Pastry nodes maintains additional overlay neighbors to support prefix-matched routing. Pastry distinguishes between two kinds of neighbors, namely *ring neighbors*, which are the L successor and predecessor nodes along the ring, and *prefix neighbors*, which are overlay nodes with different number of matching prefix digits with the node. The former nodes are represented in a data structure called the *leaf set*, while the prefix neighbors are represented in a tabular structure called the *routing table*. The entry in the l^{th} row and b^{th} column of the routing table points to a node whose identifier has the same l prefix digits as this node's identifier and b as the $(l+1)^{th}$ prefix digit. The average neighborhood state maintained by a Pastry node is $O(L)$ for the leaf set and $O(B \log_B N)$ for the routing table.

The routing table enables Pastry node to forward queries using prefix matching. When a Pastry node needs to route a message, it forwards it to the home node directly if present in the leaf set. Else, it picks the node with one more matching prefix digit from the routing table and forwards the message to it. In exact terms, if the object identifier has l matching prefix digits with the node's identifier and has b as the $(l+1)^{th}$ prefix digit, then Pastry picks the node in the l^{th} row and b^{th}

column in the routing table to forward the message. Sometimes, no node may exist in the l^{th} row and b^{th} column of the routing table. In that case, Pastry forwards the message to the closest node in the l^{th} row of the routing table. Figure 2.1 illustrates the process of routing in Pastry. An analysis of the complexity of the routing protocol in [128] proves that the worst case latency is $\lceil \log_B N \rceil$ hops.

Pastry nodes automatically update their leaf set and routing table during node failures and joins. A new node joining the network initiates the join by contacting a *bootstrap node*, which can be any node in the current Pastry ring known to the joining node out of band. The bootstrap node then routes a *join message* in the ring to locate the node closest to the joining node. The joining node then learns its own position in the ring and fills its leaf set based on the leaf set of the closest node found by routing the join message. The nodes in the leaf set of the joining node also update their leaf set to include the new node. The new node fills its routing table based on the routing tables of the intermediate nodes that routed the join message for the new node. More precisely, the new node fills the l^{th} row in its routing table from the l^{th} row of the intermediate node with l matching prefix digits.

For failure management, each node checks liveness of its neighbors by periodically probing all nodes in their leaf set and routing table. When a neighbor in the leaf set fails, the node adds a new neighbor to the leaf set based on the leaf sets of other neighbors in the leaf set. That is, if one of the L successors of a node fails, its new L^{th} successor is the immediate successor of the old L^{th} successor. When a node in the routing table fails, a new node to fill that position is chosen from nodes in the same row and column of the routing tables of other nodes in the same row. Overall, the failure management protocols ensure that message routing rarely fails

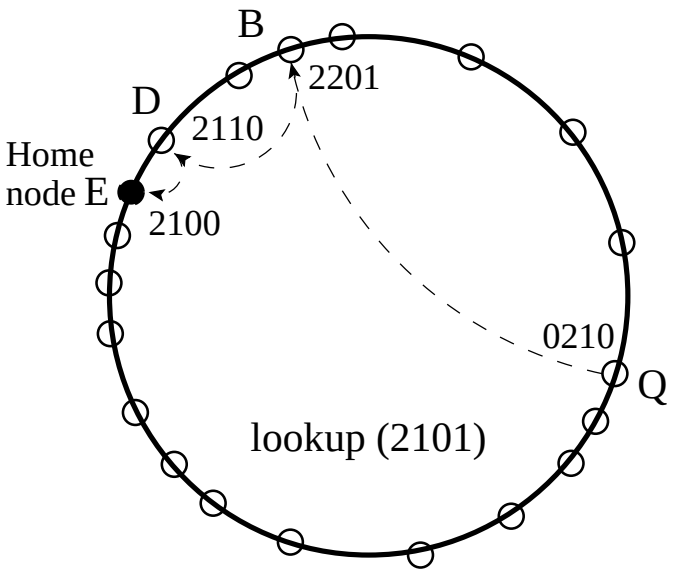

Figure 2.1: Prefix Routing in Pastry: The object 2101 is hosted by the home node E, which has the closest identifier 2100. A query for 2101 is routed towards the home node by iteratively matching prefix digits. In the figure, the query traverses through intermediate nodes B and D, which share one more prefix digit with the object than the previous node in the path.

even in the presence of high churn, that is, frequent joins and failures of nodes.

2.2 Analytical Modeling

This thesis presents a systematic approach to allocate nodes in a structured overlay so that the complexity of resource-allocation decisions is manageable. This approach takes advantage of the regular topology of structured overlays, which induces a Directed Acyclic Graph (DAG) rooted at each node in the system. This DAG is formed by the intermediate nodes through which messages are routed for objects hosted by that home node. Such a DAG in a structured overlay has a uniform node degree at each level. For example, the node degree in Pastry is its base b.

The systematic approach allocates nodes in the DAG based on distances from the home node. For example, all nodes at distance l hops from the home node may be allocated to host and object. Such an allocation provides an intuitive handle to track resource performance tradeoff. In the preceding example, the number of hops to locate a node hosting the object decreases by h hops, while the number of nodes hosting the object is b^h, where b is the branching factor of the DAG. Thus, the resource performance tradeoff of an object is modeled using just a single variable.

This approach lowers the decision-making complexity from $O(MN)$ to $O(MK)$, where K is the diameter of the structured overlay. Since structured overlays have small diameters, typically ranging from a constant to a root ($O(N^{1/d})$) of the number of nodes, this approach reduces the complexity of modeling resource allocation problems substantially. Obviously, the coarse granularity at which allocation decisions are made could lead to sub-optimal solutions. However, as shown

in the subsequent chapters, this coarse-granularity resource allocation leads to a fundamentally superior level of performance compared to ad-hoc, heuristic based techniques.

The systematic node allocation strategy can be formalized for a typical prefix-matching structured overlay as follows. Each object is allocated nodes at a level l called the *allocation level*. The level l corresponds to a wedge of nodes that have l or more matching prefix digits with the object. Thus an l level object has lookup latency of l hops and is replicated at $\frac{N}{b^l}$ nodes in the system. The allocation level for an object can range from 0, where all the nodes in the system host the object to $\lceil \log_b N \rceil$, where only the home node hosts the object. Figure 2.2 illustrates the concept of allocation levels in Pastry.

The central question, then, is to determine the best allocation levels for each object in the system. The optimal allocation strategy is determined by posing the resource-performance tradeoff as constrained optimization problems as follows:

$$\text{Find } L^* = \text{arg. min. } \sum_{1}^{M} c_m(l_m) \text{ s.t. } \sum_{1}^{M} p_m(l_m) = T_P \tag{2.2.1}$$

$$\text{Find } L^* = \text{arg. max. } \sum_{1}^{M} p_m(l_m) \text{ s.t. } \sum_{1}^{M} c_m(l_m) = T_C \tag{2.2.2}$$

In the above expressions, l_m represents the allocation level of object m and functions $c_m(l)$ and $p_m(l)$ represent cost and performance for each object as a function of their allocation level. The expressions consider total cost as a summation of individual resources allocated to each object, and treat performance as a metric averaged over each object.

Expression 2.2.1 poses an optimization problem to minimize the cost required to achieve a performance target T_P, while expression 2.2.2 represents the converse problem of maximizing performance without exceeding a bound on cost T_C. In

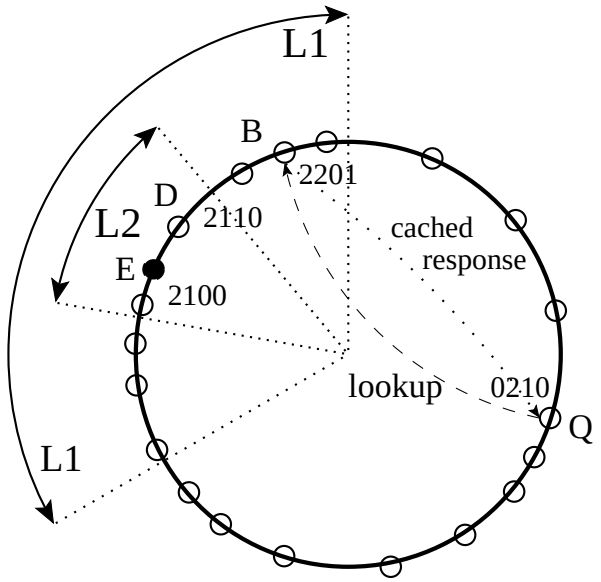

Figure 2.2: Structured Allocation: Objects are allocated to well-defined wedges of nodes defined by distances from the home node. This structured allocation facilitates analysis of cost and performance. In the figure, allocating the object to all nodes with 1 matching prefix digit provides one-hop lookup latency for that object.

either cases, the goal of the optimization is to find the optimal allocation levels of all the objects denoted by the vector $L^* = \{l_1^*, l_2^*, \cdots, l_m^*\}$. The allocation level of each object takes integral values between 0 and K, the diameter of the structured overlay, $\lceil \log_b N \rceil$ for Pastry.

The next section presents analytical models for two different performance metrics that arise in the context of real-world applications, namely, lookup latency and update detection times.

2.2.1 Lookup Latency

Lookup latency is the primary performance factor perceived by users of lookup services, such as naming systems and content distribution networks. Layering these services on top of structured overlays alone does not provide adequate lookup performance as lookups may incur long latencies as routing involves multiple hops. Since each overlay hop may traverse long distances in the Internet, the overall lookup latency in structured overlay could be quite high [43, 30], sometimes much longer than what naming systems and CDNs currently provide [39].

A naive way to improve lookup performance is by caching objects opportunistically at intermediate nodes in the lookup path. While opportunistic caching provides some improvement in lookup latency, it suffers from fundamental drawbacks as outlined in Chapter 1. This chapter presents a fundamentally different approach to caching, where the extent of caching for an object is determined through analysis of cost-performance tradeoffs rather than opportunistic decisions.

The average lookup performance for an object at level l is given by $p_m(l) = q_m D(l)$, where q_m is the popularity of the object in terms of the number of queries it receives in unit time, and $D(l)$ is the network latency in the underlying overlay to

Table 2.1: Notation: Symbols used in this thesis and their meanings.

N	number of nodes
b	base of underlying overlay
K	diameter of underlying overlay
M	number of objects
l_m	allocation level of object m
q_m	popularity of object m
s_m	size of object m
u_m	update rate of object m
$p(l)$	performance function
$c(l)$	cost function
T	performance target
τ	polling period for detecting updates
α	exponent of Zipf distribution
λ	Lagrange multiplier

traverse l hops. For a structured overlay that does not take into account network proximity while choosing neighbors, the average latency is the same at all levels and $D(l) = l$.

However, structured overlays sometimes fill positions in the routing table based on network proximity [30, 43]. For such proximity-aware overlays, the latency $D(l)$ can be modeled as $\sum_0^l d_j$, that is, the sum of the average latencies of first l hops. Proximity aware overlays tend to have lower average latencies in initial hops than hops close to the home node because a node typically has more choices to fill a position at lower levels of the routing table than at higher levels. For example, in Pastry, there are more nodes with 1 matching prefix digits than 2 matching prefix digits, hence the average latency at level 1 tends to be lower than at level 2. In practice, the values of d_j can be determined experimentally.

The cost tradeoff for an object depends on the key type of resource the application desires to conserve. For storage cost or memory consumption, the cost function for object m is $c_m(l) = s_m \frac{N}{b^l}$, where s_m is the size of the object. The bandwidth consumption for managing an object consists of three components: the update cost required to keep the object up to date, allocation cost to host the object at a node, and maintenance cost required to manage resource allocation over time. The cost to update object m at a single node is $s_m u_m$, where s_m is the size of the object and u_m is the update rate of the object, that is, the number of updates seen by the object in unit time. The maintenance cost is a constant A for each object at each node in the system.

The cost to change the allocation level of an object depends on both the current and the new levels of allocation. When increasing the number of nodes allocated to an object, that is, decreasing its allocation level, a bandwidth cost is incurred only

for additional nodes as the state needs to be copied only to the newly allocated nodes. On the other hand, increasing an object's allocation level (reducing the amount of allocation) incurs negligible cost as state is only deleted and not copied over the network. Thus, the allocation cost $a_m(l)$ for object m currently at level l' is:

$$a_m(l, l') = \begin{cases} s_m \left(\frac{N}{b^l} - \frac{N}{b^{l'}} \right) & \forall\ l < l' \\ 0 & \forall\ l \geq l' \end{cases}$$

The overall bandwidth overhead for changing the allocation level of an object m from l' to l is $c_m(l, l') = (A + s_m u_m)\frac{N}{b^l} + a_m(l, l')$.

Optimization problems to capture the application-specific performance requirements can be posed using the above expressions to model cost and lookup performance. An example optimization problem, that achieves a target average lookup latency of T_L hops with minimal bandwidth consumption is expressed as:

$$\text{Find } L^* = \text{arg. min. } \sum_{1}^{M} (A + s_m u_m)\frac{N}{b^l} + a_m(l, l') \text{ s.t. } \sum_{1}^{M} q_m l_m \leq T_L \quad (2.2.3)$$

The above optimization problem takes into account the size, update rate, and popularity of objects with minimizing bandwidth cost. The lookup latency is expressed as a weighted average over the number of queries received by different objects. While Equation 2.2.3 models a typical lookup latency versus bandwidth tradeoff that arises in a decentralized service, Chapters 3 and 4 present optimization problems specific to performance requirements of two practical applications.

2.2.2 Update Detection Time

This section presents an analytical model for another key performance metric, namely update latency. Update latency, the time taken for users of an application

to learn about updates to objects, is a critical performance metric for online data monitoring and event detection systems. These systems typically monitor an online data source from different vantage points by polling the data source periodically. The key resource allocation problem is to determine the number of nodes required to monitor each data source keeping in mind that monitoring a data source incurs bandwidth overhead.

This update latency versus bandwidth tradeoff can be modeled in a structured overlay as follows. The average update detection time at a single node polling periodically at an interval τ is estimated as $\frac{\tau}{2}$. An object at level l has, on average, $\frac{N}{b^l}$ nodes polling it and the average time for detecting updates cooperatively is $\frac{\tau}{2}\frac{b^l}{N}$. The total network load for polling an object at level l is $\tau s_m \frac{N}{b^l}$, where s_m is the size of the object.

Based on the above expressions for update detection time and network load, a typical update latency versus bandwidth optimization problem can be posed as follows:

$$\text{Find } L^* = \text{arg. min. } \sum_{1}^{M} s_m \frac{N}{b^{l_m}} \text{ s.t. } \sum_{1}^{M} q_m \frac{b^{l_m}}{N}\frac{\tau}{2} \leq T_U \tag{2.2.4}$$

This constrained optimization problem targets an average update detection time of T_U in the system while minimizing total bandwidth required for polling. The variables q_m represent the number of users interested in object m and s_m the size of object m. The average update detection time is measured as a weighted average over number users interested in each object. Chapter 5 presents other optimization problems to capture different performance requirements that arise in the context of Corona.

2.3 Optimization Techniques

The optimization problems formulated in the preceding section are NP-hard. When the allocation levels take only integral values, these problems are equivalent to integer linear programming and multiple knapsack problems [82], which are well-known NP-hard problems. Since the number of objects in the system is expected to be of the order of millions or higher, finding the exact optimal solution to the optimization problem takes exponential run-time complexity and is not practical. Instead, this thesis develops techniques to obtain near-optimal solutions in real time. It presents two techniques, namely an analytical technique and a numerical technique, that are accurate and fast.

2.3.1 Analytical Optimization

The analytical approach obtains closed-form solutions to the optimization problem through mathematical derivation. Closed-form solutions imply that the optimal replication strategy can be quickly obtained by evaluating a simple mathematical formula. To facilitate mathematical derivation, this approach approximates the characteristics of the objects such as popularity, size, and update rate, with well-known, analytically tractable distributions. Analytical modeling of tradeoff parameters ensures that the resulting closed-form solutions depend on few variables and can be computed locally by each node.

This section derives closed-form solutions for a typical optimization problem that achieves a target average lookup performance T_L expressed in overlay hops while minimizing overhead. To facilitate analysis, it makes the following approximations for the tradeoff parameters: the size and update rate are assumed to be

uniformly same for all objects and the popularity of the objects are assumed to follow a power-law or Zipf distribution.

A Zipf distribution [163] has the characteristic that the number of queries received by the i^{th} most popular object is proportional to $i^{-\alpha}$, where α is a parameter of the Zipf distribution called the *exponent*. The Zipf distribution is a heavy-tailed distribution, where the less popular objects in the tail of the distribution contribute substantially to the total number of queries. The exponent gives a quantitative indication of the contribution of the tail. The smaller the exponent, the heavier the tail, that is, more queries go for less-popular objects. Measurement studies show that several Internet applications, including DNS, web content distribution, and RSS syndication, are characterized by Zipf distributions [79, 18, 92] with varying exponents.

With the above approximations, the optimization problem to achieve target lookup latency with minimal overhead can be expressed as follows:

$$\text{Find } L^* = \text{arg. min. } \sum_{1}^{M} \frac{N}{b^l} \text{ s.t. } \frac{\sum_1^M m^{-\alpha} l_m}{\sum_1^M m^{-\alpha}} \leq T_L \tag{2.3.1}$$

The above expression assumes that the objects are sorted in the reverse order of popularity, that is, object m is the m^{th} popular object in the system. Since the object size and update rate are not incorporated, the overhead only depends on the number of nodes allocated to the object. Thus, the goal of the optimization problem reduces to minimizing the total number of allocations while meeting a target lookup latency.

Note that analytical function optimization techniques operate on continuous real functions. Performing such a function optimization on Equation 2.3.1 would violate the integrality requirements of L^* and provide real number values for optimal allocation levels with fractional components. Rounding off the real solution to

obtain an integral solution may lead to large deviations from the integral optimal solution. In order to deal with round-off errors without sacrificing the accuracy of solutions, the problem 2.3.1 is transformed to a problem with different decision variables.

Instead of deciding the optimal allocation levels of objects, the technique reverses the problem to decide how many objects should be allocated to a certain level. Thus, instead of determining $L^* = \{l_1^*, l_2^*, \cdots, l_m^*\}$, the transformed problem determines the optimal $X^* = \{x_0^*, x_1^*, \cdots, x_{K-1}^*\}$, where x_l indicates the number of objects allocated at level l or lower. Also, x_K equals M as all the objects are allocated at level K or lower. Since the number of objects in the system is expected to be large, rounding off a real value x_l^* leads to a deviation of one object at most. Thus, the rounded-off solution results in increased allocation by at most one object compared to the optimal.

Once the optimal solution is derived in terms of x_l^*, it is quite trivial to decide which objects are allocated at level l; the most popular x_0^* objects are allocated at level 0, the next popular $x_1^* - x_0^*$ objects are allocated at level 1, and so on until any remaining objects are allocated to level K, the diameter of the overlay. Basically, an object is always allocated at a level lower or equal to a less popular object. Otherwise, swapping their allocation levels provides lower average latency, indicating that the original solution is not optimal and can be improved. The transformed optimization problem can then be expressed as follows:

$$\text{Find } X^* = \text{arg. min. } \sum_{0}^{K} x_l \frac{N}{b^l} \tag{2.3.2}$$

$$\text{s.t. } \frac{\sum_1^K l[Q(x_l) - Q(x_{l-1})]}{Q(x_K)} \leq T_L \tag{2.3.3}$$

$$\text{and } 0 \leq x_l \leq M \quad \forall \quad 0 \leq l < K \tag{2.3.4}$$

Here, $Q(x) = \sum_1^x q_m$ gives the total number of queries to the popular x objects in the system. There are additional constraints to box the values of x_l^* within acceptable limits.

This constrained optimization problem can be converted to an unconstrained optimization problem by introducing a Lagrange multiplier λ. In general, the Lagrange multiplier technique converts the constrained optimization problem Min. $f(X)$ s.t. $g(X) = 0$ to the unconstrained optimization Min. $f(X) + \lambda g(X)$. The optimal solution to the latter is then obtained by setting partial derivatives with respect to the variables X and λ to zero. The transformed optimization problem is as follows:

$$\text{Find } X^* = \text{arg. min. } \sum_0^K x_l \frac{N}{b^l} \quad + \quad \lambda[\frac{\sum_1^K l[Q(x_l) - Q(x_{l-1})]}{Q(x_K)} - T_L] \quad (2.3.5)$$

For a Zipf distribution with exponent α, the summation $Q(x) = \sum_1^x m^{-\alpha}$ can be approximated as follows:

$$Q(x) \quad = \quad \begin{cases} \ln(x) & \forall\ \alpha = 1 \\ \frac{x^{1-\alpha}-1}{1-\alpha} & \forall\ \alpha \neq 1 \end{cases}$$

The above approximations are derived by converting the discrete summation into an integral.

Using the above approximation for $Q(x)$ and analytically finding the roots of the partial derivatives of 2.3.5 produces the following solution.

$$x_l^* \quad = \quad \begin{cases} \frac{M\frac{K-T_L}{K}b^l}{b^{\frac{K-1}{2}}} & , \alpha = 1 \\ \frac{Md^l[K-T_L(1-\frac{1}{M^{1-\alpha}-1})]}{1+d+\cdots+d^{K-1}} & , \alpha \neq 1 \end{cases}$$

$$\text{where, } d \quad = \quad b^{\frac{1-\alpha}{\alpha}}$$

For a detailed derivation of the above formulas see Tables 2.2 and 2.3. The expression 2.3.6 gives the closed-form solution to the optimization problem 2.3.1 for

different values of the Zipf exponent α.

The Lagrange multiplier applied above does not consider the bounding constraints 2.3.4 and consequently, the computed values of x_l^* may exceed M the number of objects in the system. It is, however, quite easy to incorporate the bounding constraints into the analytical solution. If the closed-form solution provides a value greater than M for x_{K-1}^*, then the analysis can be repeated by forcing the value of x_{K-1}^* to M. Forcefully setting x_{K-1}^* does not change the overall form of the optimization problem 2.3.1, but only changes the number of levels to determine allocation from K to $K-1$ and the target T_L to $T_L - 1$. Thus, the optimal solution to the problem 2.3.1 can be derived quickly by iteratively applying the closed-form formulas until the solution meets the bounding constraints.

Thus, the analytical approach enables nodes in the decentralized system to quickly and independently compute the optimal allocation levels. The closed-form formulas take minimal parameters as input, namely, the number of objects in the system M and the exponent α of the Zipf distribution. Section 2.4 describes how these parameters can be estimated efficiently in a structured overlay with low communication overhead.

The closed-form solution provides a mechanism to understand cost-performance tradeoffs that arise in practice. For example, consider a resource location service such as DNS with $\alpha = 0.9$, $10,000$ nodes, and $1,000,000$ mappings, layered on a structured overlay with base 32. Applying this analytical method to achieve an average lookup time, T_L, of one hop yields $k' = 2$, $x_0 = 1102$, $x_1 = 51900$, and $x_2 = 10000$. Thus, the most popular 1102 objects would be replicated at level 0, the next most popular 50814 mappings would be hosted at level 1, and all the remaining mappings at level 2. The average per node storage requirement of this system

would be 3700 mappings.

A more detailed analysis of resource-performance tradeoffs is shown in Figure 2.3. The figure shows the total number of objects cached in the system in order to optimally meet target lookup performance in a Pastry-like overlay of $100,000$ nodes, base 32, and $10,000,000$ bindings. It shows that overhead increases very rapidly as more and more aggressive, that is, close to zero, lookup latency is targeted. Nevertheless, Figure 2.3 indicates that significantly low average lookup latencies, such as a small fractional value of 0.5 hops, can be achieved with modest overhead of caching around 4% of objects at each node.

Overall, the analytical approach provides an elegant, light-weight technique to resolve performance-overhead tradeoffs. Chapters 3 and 4 shows that this approach provides substantial performance improvement over heuristic-based caching. Yet, this approach depends on assumptions about the workload of the applications and has drawbacks. First, application objects such as web objects have orders-of-magnitude differences in their size and update rates [47]; sizes can range from a few kilobytes to several megabytes and update intervals from a few seconds to no updates whatsoever. The analytical approach often ends up allocating more nodes to a large or frequently updated object than a small static object of slightly less popularity. Consequently, it can consume significantly more bandwidth than necessary. Second, it optimizes only for total number of allocations and cannot handle fine-grained overhead consumption in terms of network bandwidth or storage. Finally, the solutions determined by the analytical approach can be far from the optimal when the popularity distribution deviates from Zipf; this deviation may happen even for workloads that typically satisfy Zipf behavior as a result of sudden increases popularity during flash crowds.

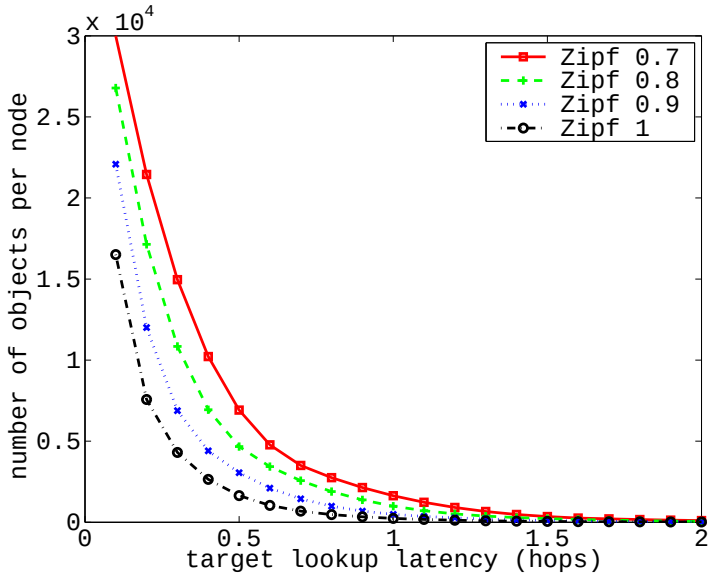

Figure 2.3: Optimal Overhead for Target Lookup Latency: The number of allocations required to meet a target lookup latency increases rapidly as lower and lower performance is targeted. Nevertheless, significantly low average lookup latencies (0.25 to 0.5 hops) can be achieved with moderate overhead.

Table 2.2: Analytical Derivation: The closed-form optimal solution for problem 2.3.1 when $\alpha \neq 1$.

The resource-performance optimization problem considered here is the following:

$$\text{Find } X^* = \text{arg. min. } \sum_{0}^{K} x_l \frac{N}{b^l}, \text{ s.t. } \frac{\sum_1^K l[Q(x_l) - Q(x_{l-1})]}{Q(x_K)} \leq T_L$$

The constraint can be simplified as follows:

$$K - \sum_{0}^{K-1} \frac{Q(x_l)}{Q(x_K)} \quad \leq \quad T_L$$

Substituting the approximation for $Q(x) = \sum_1^x x^{-\alpha} \approx \frac{x^{1-\alpha}-1}{1-\alpha}$, gives the following expression for the constraint:

$$K - \sum_{0}^{K-1} \frac{x_l^{1-\alpha} - 1}{M^{1-\alpha} - 1} \leq T_L \quad \Rightarrow \quad \sum_{0}^{K-1} x_l^{1-\alpha} \geq K + (M^{1-\alpha} - 1)(K - T_L)$$

The unconstrained optimization problem with Lagrange multiplier λ is:

$$\sum_{0}^{K} x_l \frac{N}{b^l} \quad - \quad \lambda[\sum_{0}^{K-1} x_l^{1-\alpha} - K - (M^{1-\alpha} - 1)(K - T_L)]$$

Taking the partial derivatives with respect to each x_l and λ and equating them to 0 gives the following set of equations:

$$\begin{aligned} \lambda(1-\alpha)x_l^{-\alpha} &= \tfrac{N}{b^l}, & \forall\ 0 \leq l < K \\ \textstyle\sum_0^{K-1} x_l^{1-\alpha} &= K + (M^{1-\alpha} - 1)(K - T_L) \end{aligned}$$

Solving the above set of equations provides the following closed-form formulas for x_l^*s:

$$x_l^* = \frac{Md^l[K - T_L(1 - \frac{1}{M^{1-\alpha}-1})]}{1 + d + \cdots + d^{K-1}}, \qquad d = b^{\frac{1-\alpha}{\alpha}}$$

Table 2.3: Analytical Derivation: The closed-form optimal solution for problem 2.3.1 when $\alpha = 1$.

The resource-performance optimization problem considered here is the following:

$$\text{Find } X^* = \text{arg. min. } \sum_{0}^{K} x_l \frac{N}{b^l}, \text{ s.t. } \frac{\sum_1^K l[Q(x_l) - Q(x_{l-1})]}{Q(x_K)} \leq T_L$$

The constraint can be simplified as follows:

$$K - \sum_{0}^{K-1} \frac{Q(x_l)}{Q(x_K)} \quad \leq \quad T_L$$

Substituting the approximation for $Q(x) = \sum_1^x x^{-\alpha} \approx \ln(x)$, gives the following expression for the constraint:

$$K - \sum_{0}^{K-1} \frac{\ln(x_l)}{\ln(M)} \leq T_L \quad \Rightarrow \quad \sum_{0}^{K-1} \ln(x_l) \geq \ln(M)(K - T_L)$$

The unconstrained optimization problem with Lagrange multiplier λ is:

$$\sum_{0}^{K} x_l \frac{N}{b^l} \quad - \quad \lambda[\sum_{0}^{K-1} \ln(x_l) - \ln(M)(K - T_L)]$$

Taking the partial derivatives with respect to each x_l and λ and equating them to 0 gives the following set of equations:

$$\begin{aligned} \tfrac{\lambda}{x_l} \quad &= \quad \tfrac{N}{b^l}, \qquad\qquad \forall\ 0 \leq l < K \\ \textstyle\sum_0^{K-1} \ln(x_l) \quad &= \quad \ln(M)(K - T_L) \end{aligned}$$

Solving the above set of equations provides the following closed-form formulas for x_l^*s:

$$x_l^* \quad = \quad \frac{M^{\frac{K-T_L}{K}} b^l}{b^{\frac{K-1}{2}}}$$

2.3.2 Numerical Optimization

This chapter presents a general optimization technique to address the above drawbacks and support a broad class of applications. This techniques employs numerical optimization algorithms to solve the general resource-performance tradeoff problems 2.2.1 and 2.2.2 posed earlier. Unlike the analytical technique described earlier, this technique makes no assumptions about the popularity distribution of the objects, explicitly takes into account fine-grained object-characteristics such as size and update rate, and handles detailed cost-performance models such as the expression 2.2.3 for network bandwidth consumption.

The numerical algorithm presented here is an approximation algorithm to solve general optimization problems of the following kind:

$$\text{Opt. } F(L) = \sum_{1}^{M} f_m(l_m) \text{ s.t. } G(L) = \sum_{1}^{M} g_m(l_m) \leq T$$

Here, $f_m(l)$ and $g_m(l)$ can define the performance or the resource consumption for object m as a function of the allocation level l. The numerical algorithm assumes that the functions $f_m(l)$ and $g_m(l)$ are monotonic in l. Both performance and resource consumption satisfy monotonicity.

As highlighted earlier, the challenge in solving the optimization problem 2.3.6 is that it is NP-hard. The numerical algorithm should therefore find a good balance between accuracy of solution and run-time complexity. The algorithm proposed here has a run-time complexity of $O(MK \log(MK))$ and accuracy within the granularity of one object per node. The numerical algorithm achieves high accuracy by finding upper and lower bounds for the optimal value of the objective function differing in allocation levels for at most one object. These upper and lower bounds are exact optimal solutions to problem 2.3.6 with slightly different constraints; one

with a constraint $T_L \leq T$ and another with constraint $T_U \geq T$. The solutions L_L^* and L_U^* differ in at most one object, that is, there may be one object that has a different allocation level in L_L^* and in L_U^*. The optimal solution L^* for the original problem 2.3.6 with constraint T may actually have vast differences in the levels allocated to each object compared to both L_L^* and L_U^*. Yet, the optimal value of the objective function $F(L^*)$ is bounded by $F(L_L^*)$ and $F(L_U^*)$.

The lower and upper bounds are computed by using the Lagrange multiplier technique to convert the constrained optimization problem into an unconstrained optimization problem. The Lagrange multiplier λ converts the problem 2.3.6 into the problem:

$$\text{Opt. } F'(L,\lambda) = \sum_{1}^{M} f_m(l_m) - \lambda[\sum_{1}^{M} g_m(l_m) - T]$$

The monotonicity of $f_m(l)$ and $g_m(l)$ ensures that there is a single optimum over the space of λ.

A simple approach to obtain the bounding solutions L_L^* and L_U^* that bracket the optimum is to iterate over the space of λ. Such an iterative algorithm can be performed using a standard bisection or bracketing technique [115] as follows. The algorithm starts with the bounds $L(\lambda_L)$ and $L(\lambda_U)$ which optimize the respective functions $F'(L,\lambda_L)$ and $F'(L,\lambda_U)$ and iteratively updates the bound by determining the optimal solution L_M for an intermediate value of $\lambda_M = \frac{\lambda_U+\lambda_L}{2}$. If the intermediate allocation meets the constraint, that is, $G(L_M) \leq T$ then the lower bound L_L is set to L_M, otherwise the upper bound L_U is set to L_M. The above iteration is repeated until there is no change in the bounds L_L and L_U.

The above algorithm resembles binary search over the space of λ. Each iteration for finding the optimal $F'(L,\lambda_M)$ can be accomplished in $O(MK)$ time as the optimal allocation level l_m of object m can be found by optimizing $f_m(l) - \lambda_M g_m(l)$

independent of other objects. Yet, the number of iterations required to converge to the lower and upper bounds is indeterminate as the binary search is performed over the unbounded space of real numbers. Although in practice, the number of iterations is limited by the precision of the floating point number representation. Other alternatives to this algorithm, such as the Secant Method [115], are also not guaranteed to converge within a bounded number of iterations.

This chapter presents a technique to bound the number of iterations by making the following observation. For any object m there are at most K values of λ at which $\arg\operatorname{opt} f_m(l) - \lambda g_m(l)$ changes. These critical values are essentially $\frac{\Delta f_m(l)}{\Delta g_m(l)}$, where $\Delta f(l) = f(l) - f(l-1)$, $\forall 0 < l \leq K$. This is because exactly at $\lambda = \frac{\Delta f_m(l)}{\Delta g_m(l)}$ the optimal argument changes from $l-1$ to l.

Pre-computing the critical values of λ for each object restricts the search space to $O(MK)$ discrete values of λ. By performing a binary search using a sorted list of the discrete λ values, the number of iterations of the numerical algorithm can be bounded by $O(\log(MK))$. Overall, the run-time complexity of the optimization algorithm is $O(KM \log(MK))$ including precomputations, sorting, and searching. Table 2.4 presents the complete numerical algorithm.

2.4 Distributed Resource Allocation

The optimization techniques described in the previous section provide to efficiently resolve the resource-performance tradeoff at single, centralized node. This section extends these techniques to determine a global allocation strategy in a decentralized manner. The mechanisms outlined here are lightweight and do not rely on protocols such as global consensus, which are difficult and expensive to achieve in large wide-area systems.

Table 2.4: Numerical Optimization Algorithm

Input:
$f(M,K)$: objective function for object m
$g(M,K)$: constraint function for object m
T: target on the constraint
Output:
$L_{low}(M)$: lower bound solution
$L_{up}(M)$: upper bound solution

Solve (**Input:** $f(M,K)$, $g(M,K)$, T; **Output:** $L_{low}(M)$, $L_{up}(M)$) {
variable $\Lambda := \{-\infty, \infty\}$
/* Pre-compute Lambda Values */
for $m = 1$ to M
for $l = 1$ to K
$\Lambda := \Lambda \cup \{\frac{f(m,l)-f(m,l-1)}{g(m,l)-g(m,l-1)}\}$
end for
end for
$\Lambda := sort(\Lambda)$ /* Sort Lambda Values */
/* Bisection on Lambda Values */
variable $low := 1$
variable $up := M * K + 2$
for $m = 1$ to M
$L_{low} :=$ arg. min. $f(m,l) - \Lambda(low)g(m,l)$
$L_{up} :=$ arg. min. $f(m,l) - \Lambda(up)g(m,l)$
end for
variable $T_{low} := \sum_1^M g(m, L_{low})$
variable $T_{up} := \sum_1^M g(m, L_{up})$
while$(low < up)$
variable $mid := \frac{low+up}{2}$
for $m = 1$ to M
$L_{mid} :=$ arg. min. $f(m,l) - \Lambda(mid)g(m,l)$
end for
variable $T_{mid} := \sum_1^M g(m, L_{mid})$
if $((T_{low} \leq T$ and $T_{mid} \leq T)$ or $(T_{low} \geq T$ and $T_{mid} \geq T))$
$L_{low} := L_{mid}$; $T_{low} := T_{mid}$
else
$L_{up} := L_{mid}$; $T_{up} := T_{mid}$
end if
end while
end

Instead, the presented techniques rely independent decision making and limited, local communication. Each node applies the closed-form formula or executes the numerical algorithm locally to determine the optimal allocation levels of objects. However, the analytical and the numerical techniques outlined in the previous section rely on global information about characteristics of all objects in the system. These global object-characteristics are aggregated by the system at a coarse-granularity through periodic communication between overlay neighbors. Together, local computation and limited aggregation ensure that the system achieves stable resource allocation close to the global optimum.

This thesis presents a periodic three-phase protocol to perform resource allocation efficiently. The three phases consist of an *optimization phase*, a *maintenance phase*, and an *aggregation phase*. In the optimization phase, each node applies the selected optimization technique using fine-grained characteristics for locally hosted objects and coarse-grained characteristics obtained from overlay neighbors during the previous aggregation phase. In the maintenance phase, changes to allocation levels are communicated to peer nodes enabling them to host an object or stop hosting an object. Finally, the aggregation phase enables nodes to receive new aggregates of object-characteristics for the next optimization phase. In practice, the three phases occur concurrently at each node with aggregation data piggy-backed on *maintenance messages* sent during the maintenance phase.

In the optimization phase, nodes operate independently and make decisions to increase or decrease allocation levels of each locally hosted object. Initially, only the home node at level K hosts an object. If the home node decides to lower the allocation level to $K-1$ (based on local optimization), it sends a message to the contacts at row $K-1$ of its routing table in the next maintenance phase. As a

result, a small wedge of level $K-1$ nodes start hosting that object. Subsequently, each of these nodes may independently decide to further lower the allocation level of that object. Similarly, if the home node decides to raise the allocation level from $K-1$ to K it asks its contacts in the $K-1$ wedge to stop hosting the object. In general, the responsibility of deciding to host or not host an object at a node is delegated to the parent in the DAG rooted at the home node of the object. These decisions are made periodically during each optimization phase.

The decisions made during the optimization phase are communicated to neighboring nodes during the maintenance phase. When a level l node lowers the level to $l-1$ or raises the level from $l-1$ back to l, it instructs row $l-1$ in its routing table contacts to start or stop hosting that object by sending a *maintenance message*. This control path, illustrated in Figure 2.4, is closely related to the DAG rooted at the home node. The maintenance phase proceeds periodically following every optimization phase.

Nodes aggregate characteristics of objects during the aggregation phase. Each node requires a snapshot of the characteristics of all the objects in the system in order to apply the optimization technique and derive the globally optimal allocation strategy. For the analytical optimization technique, the node requires the popularity estimates of all objects in order to compute the relative popularity rank of objects and the Zipf exponent α of the global popularity distribution. For the numerical technique, each node requires other object characteristics such as size and update rate in addition to popularity. The next sections describe techniques to estimate the popularity of an object in a distributed system and aggregate global workload characteristics.

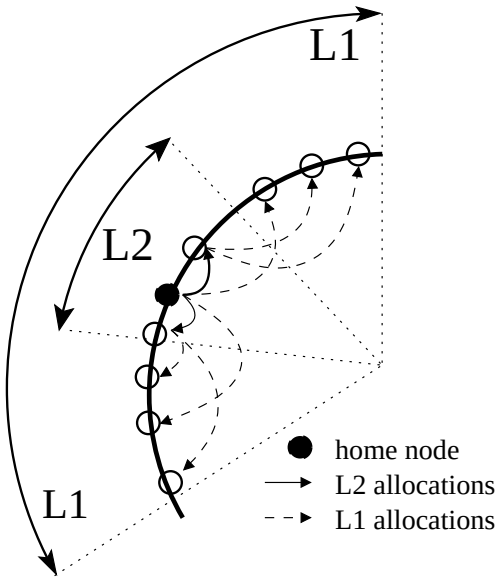

Figure 2.4: Distributed Allocation: Nodes allocate or deallocate objects their one-hop neighbors. Initially the home node allocates an object to its one-hop neighbors, shown in the figure with thick lines for L2. Subsequently, these nodes control allocations for their one-hop neighbors, shown as dashed lines in the figure.

2.4.1 Popularity Estimation

Estimating the popularity of an object, that is, the total number of queries in the system for that object per unit time, is non-trivial as the object may be hosted by several nodes. Each node receives a fraction of the queries destined for that object. Estimating the popularity of an object at a certain time involves gathering details of query arrival for that object at each hosting node.

A naive way to compute the query rate of an object, is to have each node periodically measure, in some *aggregation interval*, the number of queries an object receives in a given period and send the number to the home node, which can add up the numbers reported by different nodes. However, if the query distribution is heavy-tailed there may be orders of magnitude of difference between the query rates of popular and unpopular objects. Hence, no single aggregation interval is large enough to accurately estimate the query rates of all objects and small enough to allow the system to detect rapid changes in the object popularity as may occur during a flash crowd.

An alternative is to measure the inter-arrival time between queries for each object independently at each hosting node and use those measurements to determine the query rate. However, deriving the query rate by aggregating the inter-arrival times reported by several nodes is expensive as the inter-arrival times do not aggregate, unlike the number of queries.

This chapter presents a hybrid of the above two approaches, namely query-rate estimation and inter-arrival time estimation. Nodes hosting an object measure the number of queries for that object in each aggregation interval. They periodically transmit the data collected for each object towards the home node of the object along the DAG. Parent nodes in the DAG aggregate the data they receive and

continue to route the data towards the home node. Ultimately, the home node receives a count of queries for the object.

Home nodes then estimate the inter-arrival time using the aggregate query-rates. For unpopular objects, which may receive no queries in many aggregation intervals, the home node estimates the query inter-arrival time in terms of the number of aggregation intervals before a query is seen. That is, if an object receives one query every i^{th} aggregation interval, it has a query inter-arrival time of i. For popular objects, which receive many queries in the same aggregation interval, it estimates their query inter-arrival time as $1/j$, where j is the number of queries seen in a single aggregation interval.

The advantage of the hybrid technique is that small values of aggregation interval can be chosen without decreasing the accuracy of query rate estimates. Increased aggregation overhead can be partially reduced by sending aggregation messages only when the aggregated values are non zero. Overall, this technique enables quick detection of sudden increases in popularity of an object such as during flash crowds or denial of service attacks. At the same time, popularity estimation for both popular and unpopular objects are handled uniformly with a common aggregation interval without using different aggregation intervals for each object.

The home node distributes the latest popularity estimate of the object along with the maintenance messages. Thus, popularity aggregation includes information flow from the hosting nodes in the system to the home node and back from the home node to the hosting nodes. This process takes at most K rounds of aggregation for detecting changes in the popularity of an object and another K rounds of maintenance for any remedial action to take effect based on the new popularity estimates.

2.4.2 Distributed Tradeoff Aggregation

As noted earlier, the optimization relies on information about all objects in the system to compute a global solution. It is clearly impractical to make information about every object in the system available to every node. At the same time, computing allocation levels based solely on locally hosted objects leads to large deviations from the global optimum.

This chapter presents a technique to approximate objects with similar characteristics as coarse-grained clusters in order to achieve scalable data aggregation. Objects with similar cost performance tradeoffs are combined into larger units called *clusters*. These clusters are formed by comparing the ratios of the dominant factors in cost and performance functions, that is, $\frac{f_m}{g_m}$. For achieving lookup-latency targets, objects with comparable values for $\frac{q_m}{s_m u_m}$ ratios are clustered and treated as a single unit.

The accuracy of the clustering technique depends upon the number of clusters chosen. While using more clusters improves accuracy, it also imposes additional bandwidth overhead for aggregation. The presented technique uses a constant C number of clusters for each level of allocation, that is, all objects allocated to the same level are divided into C clusters. This division is performed by taking the $\frac{q_m}{s_m u_m}$ values of objects at a particular level, dividing the space of these values into C equal units, and then combining the objects that fall into each range together. Combining objects in the this manner has a subtle advantage over just putting equal number of objects into each cluster. The method described here ensures that objects in the head of the popularity distribution are clustered at a finer grain compared to objects at the tail of the popularity distribution. Figure 2.5 illustrates the above clustering approach.

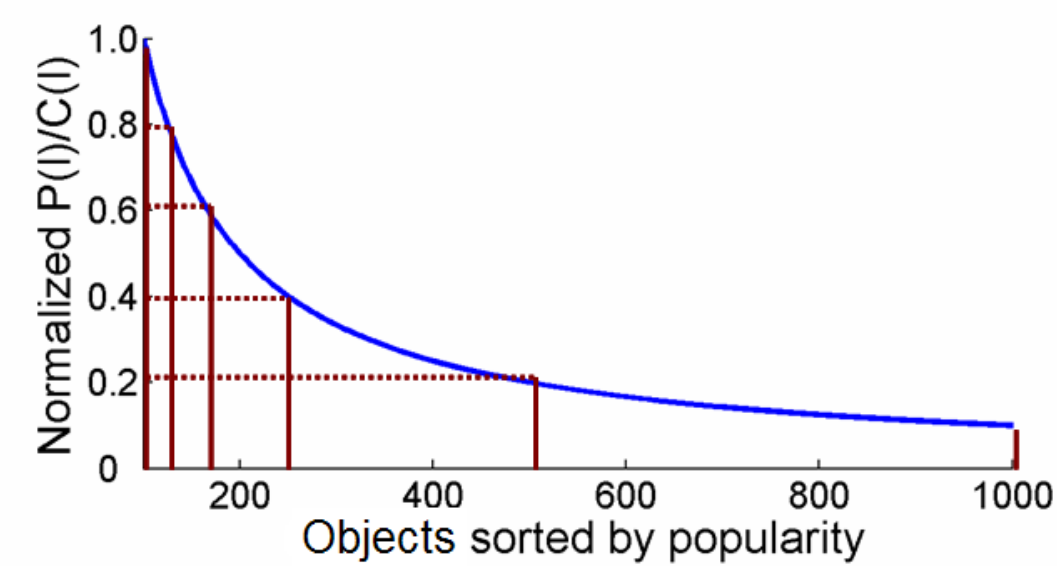

Figure 2.5: Clustering Objects with Similar Characteristics: Objects with similar characteristics are combined to form clusters. This clustering is done in a non-uniform way; for instance, highly popular objects are clustered at fine-grain while less popular objects are clustered at coarse-grain.

The clusters are then aggregated system-wide by exchanging aggregate characteristics for each cluster between the neighbors in the overlay network. Each node receives C clusters for every allocation level from each neighbor. Once a cluster is received from a neighbor the cluster is treated just as an object during optimization and for further aggregation. The cluster aggregation overhead in terms of memory state as well as network bandwidth is limited by the node degree of the system and the diameter of the underlying overlay. For Pastry, this overhead amounts to $CB \log^2 N$ clusters at each node.

Overall, each node utilizes the precise characteristics for the locally cached objects, and cluster-level, coarse-grained characteristics for other objects. Optimization is then performed based on these characteristics. For analytical optimization, the global Zipf exponent is estimated using the cluster-level popularity estimates and the precise popularity estimates for local objects. Finally, nodes use the derived optimal allocation levels only for local objects and ignore the allocation levels for clusters.

2.5 Implementation

The algorithms and mechanisms outlined in the preceding sections have been implemented in the form of a tool kit called **Honeycomb**. Honeycomb is a near-optimal resource management framework layered on Pastry, the prefix-matching structured overlay system described in Section 2.1.1. This combination of near-optimal resource allocation and structured overlays enables Honeycomb to supporting high performance, robust, and scalable network services. This section provides details behind Honeycomb implementation.

Honeycomb uses 128-bit SHA-1 hash function to generate identifiers for both

the nodes and objects. The node identifiers are obtained by hashing the IP address of the node. Using IP addresses ensures that when a node joins the system after a transient failure, it gets to host the same objects as before. Thus, state associated with an object can be stored on the disk and loaded back from it instead of copying the object from the network at the time of joining. Object identifiers are obtained by hashing application-specific names such as domain names or web URLs.

Assigning identifiers based on static IP addresses alone is not sufficient to ensure that objects survive node failures. Honeycomb prevents objects from being lost from the system by always hosting the object on multiple nodes called *owners*. Each Honeycomb object has $2O + 1$ owners, which include the home node and the O closest nodes in the ring on either side of the home node. Upon the failure of the home node, the closest of the remaining owner nodes take over as the new home node. When a owner node fails, it is placed by a new owner chosen from the leaf set of the underlying overlay and allocated to host the object. Permanent loss of an object may still occur when all the owners of the object fail, for instance, during a massive system failure involving a large number of nodes. Ensuring durability of objects through such low-probability events is expensive and beyond the scope of this thesis.

In addition to application-specific state, Honeycomb, also associates modest amount of meta state with each object in order to manage its resources. This meta state includes the following:

- Object ID: A 128-bit identifier.
- Version ID: A 32-bit version number.
- Allocation Level: A 8-bit integer specifies the current allocation level.

- Size: A 32-bit integer gives current size of object.
- Update Rate: A 32-bit real gives number of changes to the object in unit time.
- Popularity: A 32-bit real gives current popularity estimate.
- Popularity-Aggregate: A 32-bit real used to aggregate popularity.

The Version ID mentioned above is used to track changes to the object. When an object changes, the update needs to be propagated to all the nodes hosting the object. Honeycomb supports mutable objects by proactively disseminating updates to all nodes hosting that object. Honeycomb takes advantage of the structure of the overlay to efficiently disseminate object updates to all the nodes hosting an object. Just the allocation level of an object indicates the set of nodes that host the object. Consequently, Honeycomb does not require expensive mechanisms to keep track of locations of objects.

Update dissemination in Honeycomb works as follows. The home node initiates the update through the nodes in its routing table. If the object is replicated at level l, the home node sends a copy of the update to each node in the l^{th} level of the routing table. The nodes receiving the update subsequently propagate the update using their routing table. For example, the nodes at level l of the home node's routing table further propagate the update to nodes in the $(l+1)^{th}$ level of their routing tables. This update propagation process takes $O(K)$ propagation delay to reach all nodes in the system and ensures that no node is reached more than once.

The proactive update propagation protocol provides best-effort delivery guarantees. In order to ensure that any node that may have missed an update gets

eventually updated, Honeycomb also supports a lazy update propagation mechanism. Nodes send the current version numbers of the objects they host to their parent nodes as part of the aggregation message. The parent nodes check the version numbers and sends an update to those nodes with older versions of the object.

Prefix matching overlays occasionally create *orphans*, that is, objects with no nodes having $K-1$ matching prefixes. Orphans are created because there may be no nodes with enough matching prefix digits in the system and the wedge corresponding to level $K-1$ may be empty. A consequence of this for the prefix-based resource allocation technique is that it cannot assign additional nodes to host an orphan. Left unhandled, this problem can have an adverse impact on the performance tradeoff as performance for orphan objects cannot be improved. Honeycomb compensates for the loss of performance due to orphans by separately aggregating the characteristics of all orphans into a *slack cluster*, which is used to correct the performance target prior to optimization.

Chapter 3

CoDoNS: Cooperative Domain Name System

Internet communication such as access to web sites, file transfers, or remote login sessions, begin with the critical task of translating the name of the remote host or web service to a network address. The Domain Name System (DNS) provides this translation from human-friendly names of network hosts to their network addresses. In addition to this critical functionality, it also acts as an extensible database for storing and retrieving data associated with names, including names of email servers, geographic location of hosts, and information about web services.

DNS [101, 102] manages domain names by hierarchically partitioning the namespace into non-overlapping regions called *domains*. This hierarchical partitioning provides a decentralized approach for managing the namespace efficiently by dividing a domain into *subdomains*. For example, *cs.cornell.edu* is a sub-domain of the domain *cornell.edu*, which in turn is a sub-domain of the top-level domain *edu*. Top-level domains are sub-domains of a global root domain. Each domain name belongs to a *nameowner*, who has complete freedom for further partitioning the name into subdomains controlled by other nameowners. This hierarchical approach for managing the DNS namespace has been critical to the scalability of DNS. Moreover, each nameowner can manage its part of the namespace without coming into conflict with others.

Domain names have associated information identified by well-defined types and represented using extensible data structures, called *resource records*. A domain name may have many resource records of each type. These types identify specific

kinds of resources, such as IP address (type A), mail server (type MX), service attributes (type SRV), geographic location (type LOC), among others. DNS resource records are served by Internet hosts called *nameservers*.

In order to facilitate decentralized name resolution, DNS follows a *delegation-based architecture*. It works by delegating the responsibility of serving resource records for each domain to a set of replicated nameservers called *authoritative nameservers*. The authoritative nameservers of a domain manage all information for names in that domain and keep track of authoritative nameservers of the sub-domains of that domain. At the top of the legacy DNS hierarchy are *root nameservers*, which keep track of the authoritative nameservers for the *top-level domains* (TLDs). The top-level domain namespace consists of generic TLDs (gTLD), such as *.com, .edu, and .net*, and country-code TLDs (ccTLD), such as *.uk, .tr, and .in*. Nameservers are statically configured with thirteen IP addresses for the root servers. BGP-level anycast is used in parts of the Internet to reroute queries destined for these thirteen IP addresses to a larger number of root servers.

Clients rely on *resolvers* to lookup for DNS resource records. Clients typically issue DNS queries to local resolvers within their own administrative domain. Resolvers follow a chain of authoritative nameservers in order to resolve the query. The local resolver contacts a root nameserver to find the top-level domain nameserver. It then issues the query to the TLD nameserver and obtains the authoritative nameserver of the next sub-domain. The authoritative nameserver of the sub-domain replies with the response for the query. Figure 3.1 illustrates the different stages in the resolution of an example domain name *www.cs.cornell.edu*.

The DNS name resolution protocol defines a multi-hop procedure where one nameserver after the other in the authority chain is contacted for resolving a query.

However, delegations to authoritative nameservers are based on names rather than network addresses. Hence, the names of the intermediate nameservers have to be in turn resolved to their addresses. Thus, following the chain of delegations requires additional name resolutions to be performed in order to obtain the addresses of intermediate nameservers. Each additional name resolution, in turn, depends on a chain of delegations. Overall, these delegations induce complex, non-obvious dependencies among nameservers.

Figure 3.2 illustrates the nameserver dependencies for *www.cs.cornell.edu*. In addition to the top-level domain nameservers, the resolution of this name depends on twenty other nameservers, of which only nine belong to the *cornell.edu* domain. Several nameservers that are outside the administrative domain of Cornell have indirect control over Cornell's namespace. In this case, *cornell.edu* depends on *rochester.edu*, which depends on *wisc.edu*, which in turn depends on *umich.edu*. While Cornell directly delegates *cayuga.cs.rochester.edu* to serve its namespace, it has no control over the nameservers that *rochester.edu* delegates to.

Pursuing the complicated chain of delegations to resolve a query, naturally, incurs significant delay. DNS employs passive caching of resource records in order to reduce the latency of query resolution. The resolvers and nameservers cache responses to queries they issue, and use the cached responses to answer future queries. Since records may change dynamically, legacy DNS provides a weak form of cache coherency through a *time-to-live* (TTL) field. Each record carries a TTL assigned by the authoritative nameserver, and is cached by a nameserver or resolver until the TTL expires.

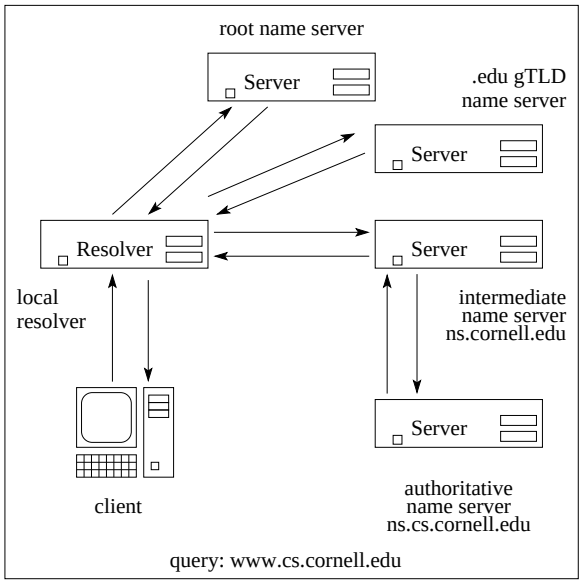

Figure 3.1: Name Resolution in Legacy DNS: Resolvers translate names to addresses by following a chain of delegations iteratively (2-5) or recursively (6-9).

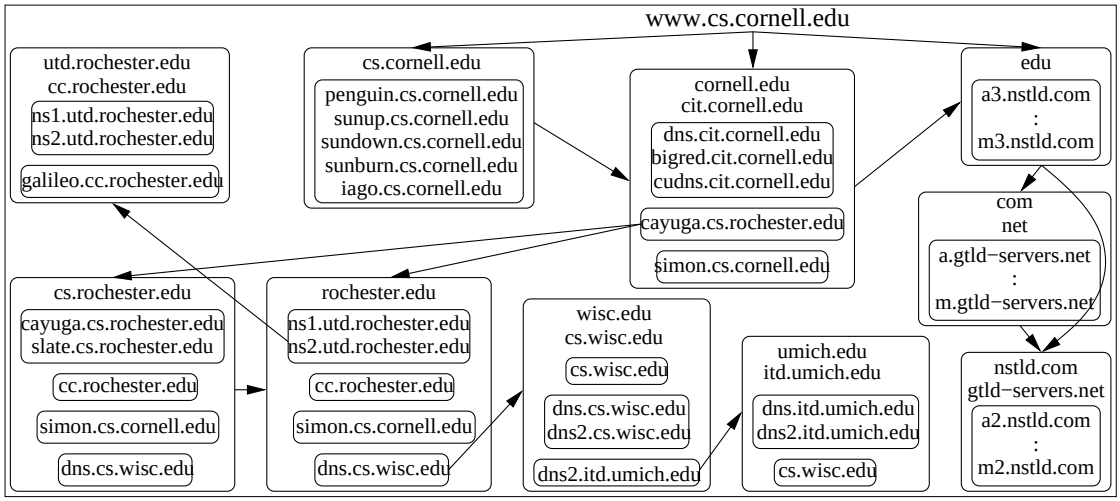

Figure 3.2: Delegation Graph: DNS exhibits complex inter-dependencies among nameservers due to its delegation based architecture. For example, the domain name *www.cs.cornell.edu* depends indirectly on a nameserver in *umich.edu*. Arrows in the figure indicate dependencies. Self loops and redundant dependencies have been omitted for clarity.

3.1 Problems with Legacy DNS

The current use and scale of the Internet has exposed several shortcomings in the functioning of the legacy DNS. We performed a large scale survey to analyze and quantify its vulnerabilities [120, 119]. This survey exposes problems three important dimensions, namely, high vulnerability to malicious attacks, poor resilience to failures, and low latencies. This sections describes these problems and shows that they fundamentally stem from the hierarchical, delegation-based architecture of DNS.

3.1.1 Survey Methodology

We collected 593160 unique names by crawling the Yahoo! and DMOZ.org directories. These names are distributed among 196 distinct top-level domains. Since the names were extracted from Web directories, these names are representative of the sites visited by users. We then queried DNS for these names and recorded the chain of nameservers that were involved in their resolution. A total of 166771 nameservers were discovered in this process. We thus obtained a snapshot of the DNS dependencies on July 22, 2004. In addition, we also separately examined the 500 most popular domains, as determined by the Alexa ranking service.

We examined the DNS delegation information to study three important characteristics of the legacy DNS architecture. First, we examined the vulnerability of DNS to malicious attacks. We studied this by looking at the overall volume of the dependencies, that is, the total number of nameservers that are involved in the resolution of a domain name. We further explored the impact of known security loopholes in DNS nameservers on the overall vulnerability of domain names. Sec-

ond, we focus on the failure resilience of the legacy DNS architecture. Our survey quantifies the minimum number of nameserver failures that DNS can tolerate before a domain name can no longer be resolved. Finally, we examine the lookup and update latencies provided by legacy DNS. In particular, we focus on the choice of TTL values for resource records and how this choice impacts the lookup and the update performance of legacy DNS.

3.1.2 Vulnerability to Malicious Attacks

The delegation based architecture of the DNS induces complex non-obvious dependencies among nameservers, and can cause unexpected nodes to exert great control over remote domains. The compromise of any one of them may lead to a domain hijack as the compromised nameserver can divert DNS requests to malicious nameservers, which could provide false IP addresses for the queried host; clients can thus be misdirected to servers controlled by attackers.

A domain hijack accomplished by exploiting DNS dependencies can be partial or complete. We distinguish between a *partial hijack*, where an attacker compromises a few nameservers and diverts some queries for the targeted name, and a *complete hijack*, where an attacker compromises enough nameservers to guarantee the misdirection of all queries for that name. A domain name is said to *depend* on a nameserver if the nameserver could be involved in the resolution of that name. We represent the dependencies among nameservers that directly or indirectly affect a domain name as a *delegation graph*. The delegation graph consists of the transitive closure of all nameservers involved in the resolution of a given name. The nameservers in the delegation graph of a domain name form the *trusted computing base* (TCB) of that name.

The vulnerability of a DNS name is tied to the number of servers in its trusted computing base, whose compromise could potentially misdirect clients seeking to contact that server. Larger TCBs provide attackers with a wider choice of targets to attack. Further, larger TCBs also imply more complex and deeper dependencies among nameservers making it more difficult for the nameowner to control the integrity of the servers it depends on.

In this section, we characterize the TCB size of the surveyed names. Figure 3.3 plots the cumulative distribution of TCB sizes not including the root nameservers, which belong to the TCBs of all the domain names. Our survey shows that TCB size follows a heavy-tailed distribution with a median of 26 nameservers, and an average of 46 nameservers; about 6.5% of the names has a TCB of greater than 200 nameservers. We computed the TCB by counting the number of distinct server names in the delegation graph. Since distinct names referring to the same machine may cause the TCB to appear larger, we also computed the number of distinct IP addresses in the delegation graphs. TCB size based on IP addresses has the same median (26), while the average decreases marginally to 44.

One might expect that the administrators of the popular websites would be better aware of the security risks and keep their DNS dependencies small. To test this hypothesis, we separately studied the TCB sizes for the 500 most popular websites reported by *alexa.org*. Figure 3.3 shows that these names are more vulnerable; they depend on 69 nameservers on average, and 15% of them depend on more than 200 nameservers.

Next, we study the TCB sizes for names belonging to different TLDs. Figures 3.4 and 3.5 plot in decreasing order the TCB sizes for names in the generic TLDs, and the fifteen most vulnerable country-code TLDs, respectively. Overall,

ccTLD names have a much higher average TCB size of 209 nameservers than gTLD names, whose average is 87 nameservers. GTLDs *aero* and *int* have considerably larger TCBs than other gTLDs, and, among the ccTLDs, Ukraine, Belarus, San Marino, Malta, Malaysia, Poland and Italy, in that order, are the most vulnerable.

We examined the dependencies to determine why names in certain domain (e.g., *aero* and *int* TLDs and several ccTLDs) have much larger TCBs than others. We find that names with larger TCBs typically have authoritative nameservers distributed across distant domains. Improving availability in the presence of network outages is one of the primary reasons why administrators delegate to, and implicitly trust, nameservers outside their control. Extending trust to a small number of nameservers that are geographically distributed may provide high resilience against failures. However, DNS forces nameowners to trust the entire transitive closure of the all names that appear in the physical delegation chains.

Sometimes even top-level domains are set up such that it is impossible to own a name in that subdomain and not depend on hundreds of nameservers. Ukrainian names seem to suffer from many such dependencies including nameservers in the US at Berkeley, NYU, UCLA, as well as many locations spanning the globe: Russia, Poland, Sweden, Norway, Germany, Austria, France, England, Canada, Israel, and Australia. It is likely that the Ukrainian authorities do not realize their dependency on servers outside their control. A cracker that controls a nameserver at Monash University in Australia can end up hijacking any Ukranian name.

Next, we examine the feasibility of malicious attacks through known vulnerabilities in commonly deployed nameservers. Early studies [44, 86, 103] identified several implementation errors in legacy DNS servers that can lead to compromise. While many of these have been fixed, a significant percentage of nameservers con-

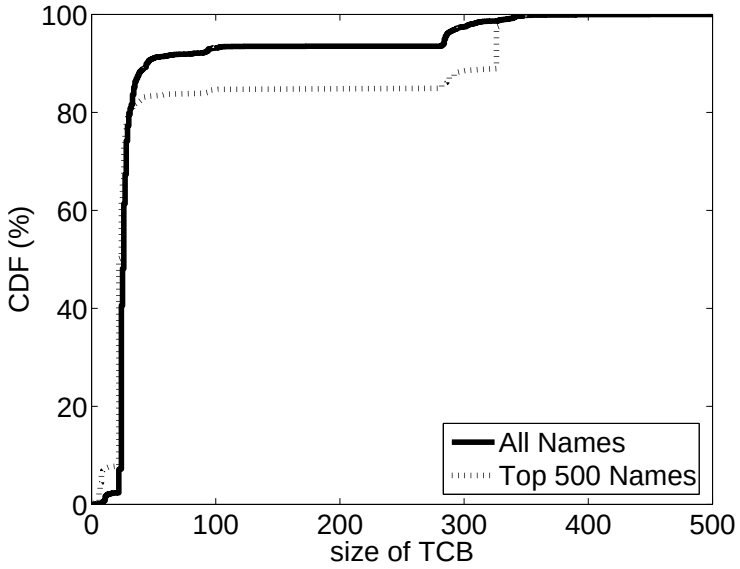

Figure 3.3: Size of TCB: DNS Name resolution depends on a large number of nameservers. On average, name resolution involves 46 nameservers, while a sizable fraction of names depend on more than 100 nameservers.

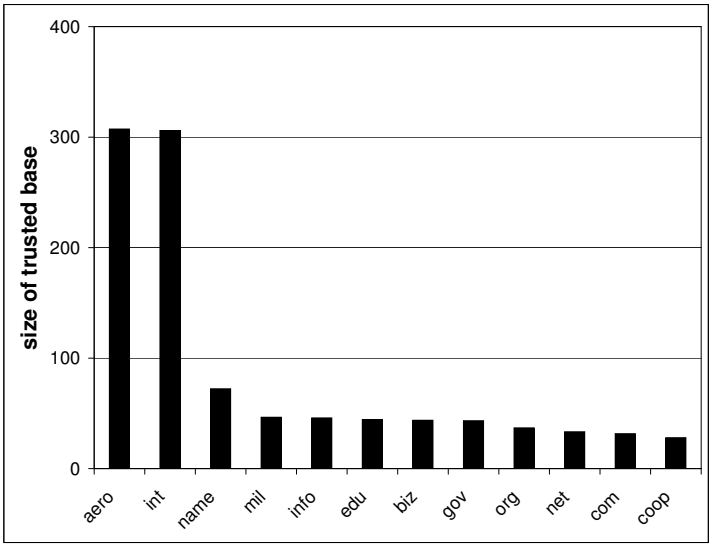

Figure 3.4: Average TCB Size for gTLD Names: Names in *.aero* and *.int* have significantly larger TCBs.

tinue to use buggy implementations. We surveyed 150,000 nameservers to determine if they contain any known vulnerabilities, based on the Berkeley Internet Name Daemon (BIND) exploit list maintained by the Internet Systems Consortium (ISC) [168]. Table 3.1 summarizes the results of this survey. Approximately 18% of servers do not respond to version queries, and about 14% do not report valid BIND versions. About 1.4% of nameserves have the *tsig* bug, which permits a buffer overflow that can enable malicious agents to gain access to the system. 13% of nameserves have the *negcache* problem that can be exploited to launch a DoS attack by providing negative responses with large TTL value from a malicious nameserver.

We combine these known vulnerabilities with the delegation graphs of domain names to explore which names are easily subjected to compromise. For nameservers whose vulnerabilities we do not know, we simply assume that they are *non-vulnerable*; hence, the results presented here are optimistic. Of the 166771 nameservers we surveyed, 27141 have known vulnerabilities. A naive expectation might be that, with 17% vulnerable nameservers, only 17% of the names would be affected. Instead, these vulnerabilities affect 264599 names, approximately 45%, because transitive trust relationships "poison" every path that passes through an insecure nameserver.

For example, *www.fbi.gov* was vulnerable to being hijacked at the time we performed our survey , along with all other names in the *fbi.gov* domain. The *fbi.gov* domain was served by two machines named *dns.sprintip.com* and *dns2.sprintip.com*. The *sprintip.com* domain was in turn served by three machines named *reston-ns[123].telemail.net*. Of these machines, *reston-ns2.telemail.net* was running an old nameserver (BIND 8.2.4), with four different known exploits against it (lib-

Table 3.1: Vulnerabilities in BIND: A significant percentage of nameservers use BIND versions with known security problems.

problem	**severity**	**affected nameservers**	
		all domains	**top 500**
tsig	critical	1.4 %	0.59 %
nxt	critical	0.04%	0.15 %
negcache	serious	13.44 %	2.57 %
sigrec	serious	9.72 %	1.32 %
DoS multi	serious	7.98 %	1.32 %
DoS findtype	serious	1.84%	0.59 %
srv	serious	1.31 %	0.59 %
zxfr	serious	1.24 %	0.44 %
libresolv	serious	1.06 %	0 %
complain	serious	0.92 %	0 %
so-linger	serious	0.78 %	0.15 %
fdmax	serious	0.78 %	0.15 %
sig	serious	0.49 %	0.15 %
infoleak	moderate	3.12 %	0.59 %
sigdiv0	moderate	1.25 %	0.59 %
openssl	medium	1.22 %	0.37 %
naptr	minor	1.81 %	0.15 %
maxdname	minor	1.81 %	0.15 %

bind, negcache, sigrec, and DoS_multi) [168]. Having compromised *reston-ns2*, an attacker could divert a query for *dns.sprintip.com* to a malicious nameserver, which could then divert queries for *www.fbi.gov* to any other address, hijacking the FBI's website and services.

Figure 3.6 shows the cumulative distribution of the number of vulnerable nameservers in the TCBs of surveyed names. 45% of DNS names depend on at least one vulnerable nameserver, and can be compromised by launching well-known, scripted attacks. Figure 3.7 shows the percentage of nodes with no known bugs in the TCBs of surveyed names. Surprisingly, a few names do not have any non-vulnerable nameservers in their TCB; these names belong to the ccTLD *ws*, which relies on older buggy versions of BIND. Overall, the average number of vulnerable servers is 4.1, about 9% of the average TCB size. The extent of vulnerability in the TCBs of the 500 most popular names is also high (7.6), about 11% of the average TCB size.

3.1.3 Failure Resilience

The legacy DNS is highly vulnerable to network failures, compromise by malicious agents, and denial of service attacks, because domains are typically served by a very small number of nameservers. We next examine the *delegation bottlenecks* in DNS; a delegation bottleneck is the minimum number of nameservers in the delegation graph of each domain that need to be compromised in order to completely hijack that domain. Table 3.2 shows the percentage of domains that are bottlenecked on different numbers of nameservers. 78.63% of domains are restricted by two nameservers, the minimum recommended by the standard [101]. Surprisingly, 0.82% of domains are served by only one nameserver. Even the highly popular

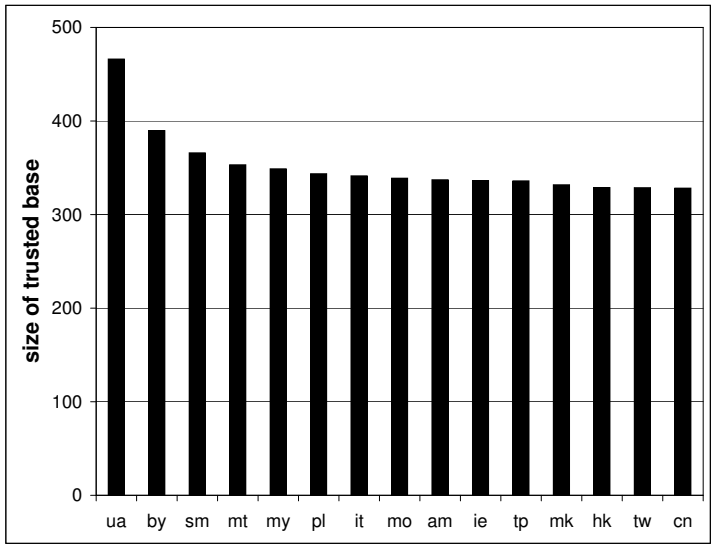

Figure 3.5: Average TCB Size for ccTLD Names: Some ccTLDs rely on, and are vulnerable to compromises in, a large number of servers.

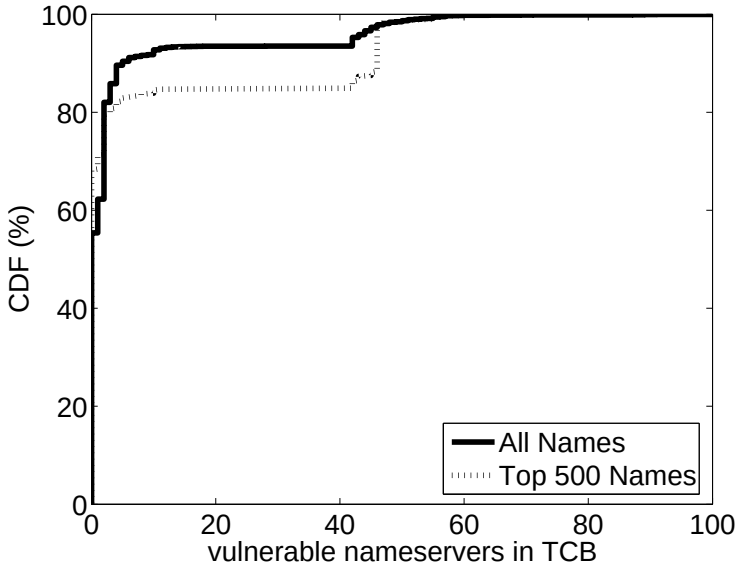

Figure 3.6: Vulnerable Nameservers in TCB: 45% of the names depend on at least one nameserverver with known vulnerability.

domains are not exempt from severe bottlenecks in their delegation chains. Some domains (0.43%) spoof the minimum requirement by having two nameservers map to the same IP address. Overall, over 90% of domain names are served by three or fewer nameservers and can be rendered inaccessible by relatively small-scale DoS attacks.

Failure and attack resilience of the legacy DNS is even more limited at the network level. We examined *physical bottlenecks*, that is, the minimum number of network gateways or routers between clients and nameservers that need to be compromised in order to control that domain. We measured the physical bottlenecks by performing traceroutes to 10,000 different nameservers, which serve about 5,000 randomly chosen domain names, from fifty globally distributed sites on PlanetLab [12]. Figure 3.8 plots the percentage of domains that have different numbers of bottlenecks at the network level, and shows that about 33% of domains are bottlenecked at a single gateway or router. While this number is not surprising as domains are typically served by a few nameservers located in the same subnetwork, it highlights that a large number of domains are vulnerable to network outages. Recently, Microsoft's services became unavailable for a substantial period of time due to a misconfiguration in their network gateway. The primary reason for the success of this attack was that all of Microsoft's DNS servers were in the same part of the network [165].

We next quantify to what extent the known vulnerabilities in the DNS nameservers affect the overall availability of a domain. Figure 3.9 shows the number of non-vulnerable nameservers in the min-cut of the delegation graphs. Surprisingly, about 30% of domain names have a min-cut consisting entirely of vulnerable nameservers. The average size of a min-cut is 2.5 nameservers. This implies that

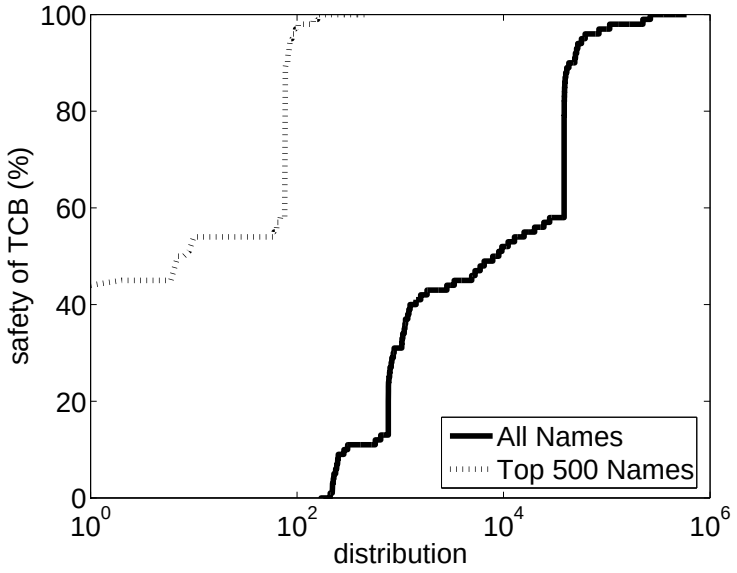

Figure 3.7: Percentage of Non-Vulnerable Nodes in TCB: A few names have their entire TCB vulnerable to known exploits.

Table 3.2: Delegation Bottlenecks in Name Resolution: A significant number of names are served by two or fewer nameservers, even for the most popular 500 sites.

Bottlenecks	**All Domains**	**Top 500**
1	0.82 %	0.80 %
2	78.44 %	62.80 %
3	9.96 %	13.20 %
4	4.64 %	13.00 %
5	1.43 %	6.40 %
13	4.12 %	0 %

these domain names can be completely hijacked by compromising less than three machines on average. Moreover, another 10% of domain names have only one non-vulnerable nameserver in their min-cut. A denial of service attack on the non-vulnerable nameserver, coupled with the compromise of the other vulnerable bottleneck nameservers, is sufficient to completely hijack these domains.

The severely limited resilience to failures is common in DNS and affect many top level domains and popular web sites. Naturally, DNS is an easy target for both malicious attacks. While few instances of phishing attacks through a domain hijack has been reported to date, the DNS is often subjected to denial of service (DoS) attacks and failures. DNS measurements at root and TLD nameservers show that these servers are frequently subjected to denial of service attacks [21, 23]. A massive distributed DoS attack [166] in November 2002 rendered nine of the thirteen root servers unresponsive. Partly as a result of this attack, the root is now served by more than sixty nameservers and is served through special-case support for BGP-level anycast. While this approach fixes the superficial problem at the topmost level, domains below the TLD level find it difficult to take advantage of this special-case approach to defend themselves against DoS attacks.

3.1.4 Performance Latencies

Name resolution latency is a significant component of the time required to access web services. Wills and Shang [153] have found, based on NLANR proxy logs, that DNS lookup time contributes more than one second to 20% of web object retrievals, Huitema et al. [73] report that 29% of queries take longer than two seconds, and Jung et al. [79] show that more than 10% of queries take longer than two seconds. Bent and Voelker show that the average DNS lookup latency per web

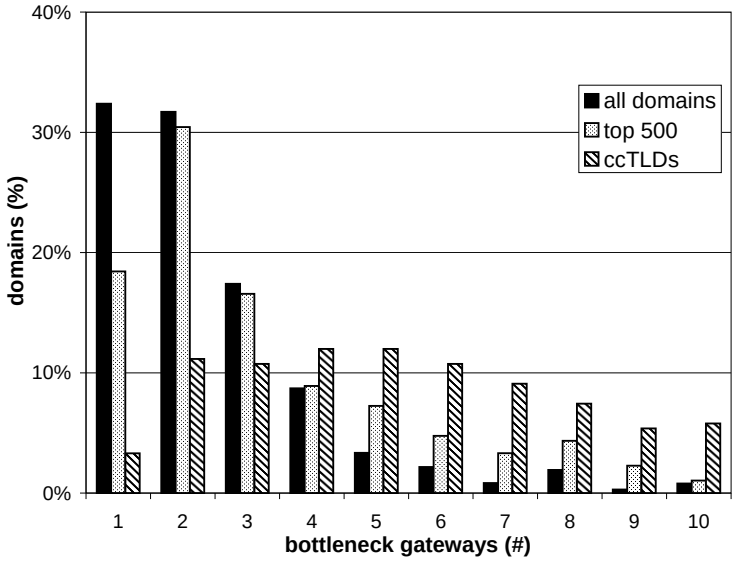

Figure 3.8: Physical Bottlenecks in Name Resolution: A significant number of domains, including top-level domains, depend on a small number of gateways for their resolution.

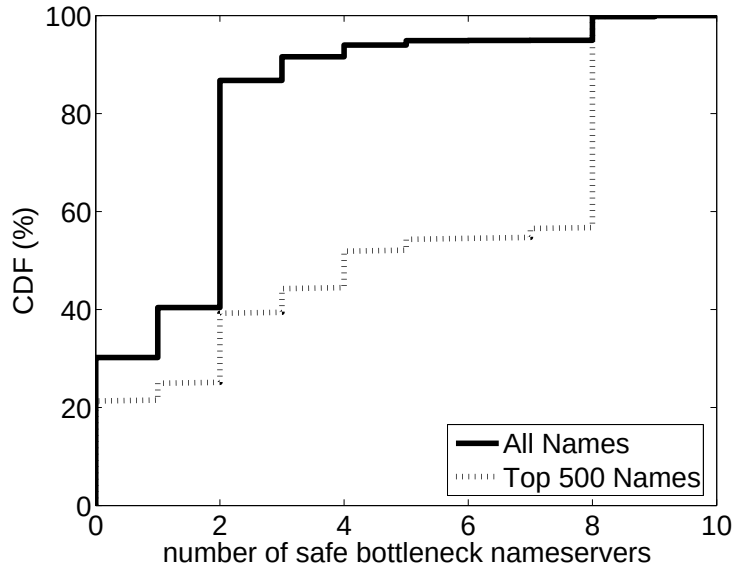

Figure 3.9: DNS Nameserver Bottlenecks: 30% percentage of names can be completely hijacked by compromising a critical set of vulnerable bottleneck nameservers.

page download is 529.7 ms, or about 12.2% of the overall latency to fetch a web page with parallel connections turned on [13].

The results from the above studies show that DNS incurs high lookup latencies and forms a significant bottleneck for accessing the Web. The low performance is due mainly to ineffectiveness of passive caching in DNS. Two fundamental reasons contribute to the ineffectiveness of caching. First is the heavy-tailed, Zipf-like query distribution in DNS; a study by Jung et al. measures a low exponent of 0.91 for popularity of DNS domains [79]. It is well known from studies on Web caching [18] that heavy-tailed query distributions severely limit cache hit rates.

The second is the use of timeout based mechanisms to manage cache consistency. Selection of a suitable value for the TTL involves a fundamental tradeoff between lookup latency and update latency. Short TTLs adversely affect the lookup performance and increase network load [79, 78], while long TTLs interfere with service relocation. For instance, a popular online brokerage firm uses a TTL of thirty minutes. Its users do not incur DNS latencies when accessing the brokerage for thirty minutes at a time, but they may experience outages of up to half an hour if the brokerage firm needs to relocate its services in response to an emergency.

We examined the TTL values associated with each resource record encountered in the survey. Figure 3.10 shows a cumulative distribution of TTL values. It shows that a majority of domains, nearly 63% of domain names, use TTLs of one hour or higher. The most common value of TTL is a day; about 30% of all domain names have one day for their TTL. This use of long TTLs prohibits fast dissemination of unanticipated changes to records. Surprisingly, the chosen TTLs are also too small compared to the actual rate of change of DNS data. We performed an active polling study, where we fetched resource records for the surveyed domain names

every day for a week and measured the average rate of change of DNS records. We found that only 0.08% of records change every day. Thus, the choice of TTLs for DNS records is entirely arbitrary and does not favorably affect either the lookup or the update performance.

Our study also showed that a small number of domain names (about 5%) have a very small TTL value of 30 seconds. These aggressively low TTL values are typically set by content distribution networks. These services, such as Akamai and Digital Island, use DNS lookups to direct clients to nearby servers of Web content. They typically use very short TTLs (on the order of 30 seconds) in order to perform fine grain load balancing and respond rapidly to changes in server or network load. But, this mechanism, called *server selection*, virtually eliminates the effectiveness of caching and imposes enormous overhead on DNS. A study on impact of short TTLs on caching [78] shows that cache hit rates decrease significantly for TTLs lower than fifteen minutes. Another study on the adverse effect of server selection [134] reports that name resolution latency can increase by two orders of magnitude.

In addition to the impact of badly chosen TTL values, DNS lookup performance is also affected by the presence of a large number of broken (lame) or inconsistent delegations. In our survey, address resolution failed for about 1.1% of nameservers due to timeouts or non-existent records, mostly stemming from spelling errors. For 14% of domains, authoritative nameservers returned inconsistent responses; a few authoritative nameservers reported that the domain does not exist, while others provided valid records. Failures stemming from lame delegations and timeouts can translate into significant delays for the end user. Since these failures and inconsistencies largely stem from human errors [103], it is clear that manual configuration

and administration of such a large scale system is expensive and leads to a fragile structure.

3.2 CoDoNS: System Design

The use and scale of today's Internet is drastically different from the time of the design of the DNS. Even though the legacy DNS anticipated the explosive growth and handled it by partitioning the namespace, delegating the queries, and widely caching the responses, this architecture contains inherent limitations. This section presents an overview of CoDoNS, describes its implementation, and highlights how it addresses the problems of the legacy DNS.

The key feature of CoDoNS is the separation of namespace management and resolution, the twin functionalities provided by legacy DNS. Through this separation, CoDoNS retains the successful aspects of DNS, namely the decentralized and scalable namespace management, while replacing its hierarchical, delegation-based resolution process with a flat, peer-to-peer architecture.

CoDoNS consists of globally distributed nodes that self organize to form a peer-to-peer network. We envision that each institution would contribute one or more servers to CoDoNS, forming a large-scale, cooperative, globally shared DNS cache. These servers could come from the same resources supporting the DNS today, namely the DNS nameservers. CoDoNS provides query resolution services to clients using the same wire format and protocol as legacy DNS, and thus requires no changes to client resolvers. Nameowners need only to purchase certificates for names from namespace operators and introduce them into CoDoNS; to the nameowners, CoDoNS provides an interface consisting of insert, delete and update. CoDoNS places no restrictions on the orgaization of the namespace and

is agnostic about the administrative policies of the nameowners.

3.2.1 Architecture

CoDoNS is layered as an application on top of Honeycomb, the near-optimal resource management framework described in Chapter 2. Consequently, it inherits and benefits from all the properties provided by Honeycomb. These properties include high resilience to failures, low, configurable lookup latency, as well as proactive propagation of updates.

Each domain name in CoDoNS is associated with the home node, whose identifier is closest to the consistent hash [81] of the domain name. The home node stores a permanent copy of the resource records owned by that domain name and manages replication for that domain. CoDoNS uses multiple owners for each domain, so that the records of that domain are replicated on all owners and data loss due to node failures is mitigated.

Replacing the DNS entirely with CoDoNS is an ambitious plan, and we do not expect nameowners to immediately use CoDoNS for propagating their information. In order to gradually grow into a globally recognized system, CoDoNS provides backwards compatibility with the legacy DNS. CoDoNS uses the legacy DNS to resolve queries for records not explicitly inserted by nameowners. The home node retrieves resource records from the legacy DNS upon the first query for those records. The additional redirection latency only affects the first query issued in the entire system for a domain name.

Overall, query resolution in CoDoNS takes place as follows. Client sends a query in the wire format of the legacy DNS to the local CoDoNS server in the same administrative domain. The local CoDoNS server replies immediately if it

has a cached copy of the requested records. Otherwise, it routes the query internally in the CoDoNS network using the underlying overlay. The routing terminates either at an intermediate CoDoNS node that has a cached copy of the record or at the home node of the domain name. The home node retrieves the records from the legacy DNS, if it does not already have it, and sends a response to the first contacted CoDoNS server, which replies to the client. In the background, CoDoNS nodes proactively replicate the records in based on the measured popularity. Figure 3.11 shows a typical deployment of CoDoNS and illustrates the process of query resolution.

Clients generate a large number of queries for names in their local administrative domain. Since the home node of a name may be located in a different domain, local queries can incur extra latency and impose load on wide-area network links. CoDoNS supports efficient resolution of local names through *direct caching*. Name-owners can directly insert, update, and delete their records at CoDoNS servers in their administrative domain, and configure the local CoDoNS servers to use the direct cache for replying to local queries.

3.2.2 Analysis-driven Optimization

CoDoNS derives its performance characteristics from the resource management framework described in Chapter 2. The key resource-performance tradeoff that arises in CoDoNS is the tradeoff between lookup latency and the resources required to store and maintain duplicate copies of the resource records. Lookup latency can be improved by caching resource resource records on more and more nodes in the system, where as, keeping the records consistently updated when resource records change consumes network bandwidth.

CoDoNS finds the right balance between lookup latency and bandwidth consumption by modeling the tradeoff as an optimization problem and using the resource management framework to determine and enforce the optimal tradeoff in the system. CoDoNS uses the following analytical model to express its resource-performance tradeoff.

$$\text{Min.} \sum_{1}^{M} s_m u_m \frac{N}{b^l} \quad \text{s.t.} \ \frac{\sum_1^M q_m l_m}{\sum_1^M q_m} \ \leq T_L \tag{3.2.1}$$

All the notations used in the preceding expression are as used in Chapter 2 and defined in Table 2.2.

3.2.3 Proactive Update Propagation

The resource management framework also enables CoDoNS to rapidly push updates to all the replicas in the system. CoDoNS uses just a small integer, the allocation level of a resource record, to determine the range of nodes hosting the record. Proactive update propagation obviates the need for timeout-based caching. Thus, CoDoNS does not make use of TTLs in resource records. Instead, it ensures through proactive update dissemination that all copies of resource records are promptly updated and the changes reach the clients quickly. With CoDoNS, network administrators can relocate services at any time without losing availability.

3.2.4 Implementation

Each CoDoNS server implements a complete, recursive, caching DNS resolver and supports all requirements described in the specification [101, 102]. CoDoNS also supports inverse queries that map IP addresses to a domain name by inserting reverse address-name records into the DHT when name-address records are intro-

duced.

Domain names in CoDoNS have unique 128 bit identifiers obtained through the SHA-1 hashing algorithm. The home node, the closest node in the identifier space, stores permanent copies of the resource records of the domain name and takes responsible for detecting and propagating updates for the record. Since CoDoNS does not associate TTLs with the records, the home nodes push the updates to all replicas, which retain them until the replication level of the record is downgraded, or until an update is received. Nameowners insert updated resource records into CoDoNS, and the home nodes proactively propagate the updates.

CoDoNS ensures the consistency of records obtained from the legacy DNS by proactively refetching them. The home node uses the TTL specified by the legacy DNS as the duration to store the records. It refetches the records from legacy DNS after TTL duration, and propagates the updated records to all the replicas if the records change. Since CoDoNS polls for updates in the background, its lookup performance is not affected. The TTL values are rounded up to a minimum of thirty seconds; records with lower TTL values are not placed into the system. Such low TTL values typically indicate dynamic server selection in legacy DNS.

The legacy DNS relies on error responses, called *NXDOMAIN* responses, to detect names that do not exist. Since clients reissue a request several times when they do not receive prompt replies, the DNS specification recommends that resolvers cache NXDOMAIN responses. CoDoNS provides complete support for negative caching as described in [7]. However, permanently storing NXDOMAIN responses could exhaust the capacity of the system, since an unlimited number of queries can be generated for non-existent domains. Hence, CoDoNS nodes cache NXDOMAIN responses temporarily and do not refresh them upon expiry.

3.2.5 Issues and Implications

CoDoNS decouples namespace management from the physical location of name-servers in the network. Instead of relying on physical delegations to trusted hosts and assuming that Internet routing is secure, CoDoNS uses cryptographic delegations and self-verifying records based on the DNSSEC [50] standard.

DNSSEC uses public key cryptography to enable authentication of resource records. Every namespace operator has a public-private key pair; the private key is used to digitally sign DNS records managed by that operator, and the corresponding public key is in turn certified by a signature from a domain higher up in the hierarchy. This process creates a chain of certificates, terminating at a small number of well-known public keys for globally trusted authorities. Since records are signed at the time of issue, the private keys need not be kept online. The signature and the public key are stored in DNS as resource records of type *sig* and *key* respectively. Clients can verify the authenticity of a resource record by fetching the sig record and the key record from the DNS.

The use of cryptographic certificates enables any client to check the verity of a record independently, and keeps peers in the network from forging certificates. To speed up certificate verification, CoDoNS servers cache the certificates along with the resource records and provide them to the clients. Existing clients that are not DNSSEC compliant need to trust only the local CoDoNS servers within their administrative domain, since CoDoNS servers internally verify data fetched from other nodes.

CoDoNS authenticates nameowners directly through certificates provided for every insertion, delete, and update. Insertions simply require a signed resource record with a corresponding well-formed certificate. An increasing version number

associated with each record, signed by the owner and checked by every server, ensures that old records cannot be reintroduced into the system. Deletions require a signed request that identifies the record to be expunged, while updates introduce a new signed, self-verifying record that replaces the now-stale version.

Since DNSSEC has not yet been widely deployed in the Internet, CoDoNS cannot rely on the legacy DNS to provide certificates for resource records fetched from legacy DNS. Consequently, CoDoNS uses its own centralized authority to sign resource records fetched from the legacy DNS. Queries to the legacy DNS are directed to a small pool of *certifying resolvers*, which fetch authoritative resource records from the legacy DNS, sign them, and append the sig records to the legacy DNS response. This approach requires trust to be placed in the certifying resolvers. Threshold cryptography [162] can be used to limit the impact of adversaries on these resolvers until CoDoNS takes over completely. The certifying name resolvers ensure that CoDoNS participants cannot inject corrupted records into the system.

Malicious participants may also disrupt the system by corrupting the routing tables of peers and misrouting or dropping queries. Castro et al. [29] propose a method to handle routing table corruptions in DHTs. This scheme augments the regular routing table with a secure routing table where the entries need to satisfy strict constraints on node identifiers that limit the impact of corrupt nodes. Since nodes in the secure routing table are not picked based on short network latencies, this scheme may increase the lookup delay. Setting a lower target latency at the Beehive layer can compensate for the increase in lookup latency at the cost of bandwidth and storage.

CoDoNS thus acts as a large cache for stored, self-verifying records. This design, which separates namespace management from the physical servers, prohibits

dynamic name resolution techniques where the mapping is determined as a result of a complex function, evaluated at run time. In the general case, such functions take arbitrary inputs and have confidentiality requirements that may prohibit them from being shipped into the system. For instance, content distribution networks, such as Akamai, use proprietary techniques to direct clients to servers [20, 134]. To support such dynamic mapping techniques, CoDoNS enables nameowners to stipulate redirections of queries for certain names using a special *redirection record*. High lookup performance during redirections is ensured through proactive replication and update of the redirection record in the same manner as regular resource records.

As with any peer-to-peer system, CoDoNS relies on its participants to contribute resources on behalf of others. While it may seem, at first, that rational actors might be averse to participating in the system for fear of having to serve as home nodes for highly popular records, proactive replication ensures that the load perceived by all nodes is comparable. A highly popular record will be replicated until the load it projects on its home node is comparable to the query load for other records.

3.3 Evaluation

This section provides a detailed evaluation of CoDoNS through a combination of simulations and measurements on an experimental deployment. The simulations show that underlying resource management framework finds near-optimal solutions to the latency-bandwidth tradeoff problem posed in expression 3.2.1. The experimental evaluations demonstrates how the analysis-driven resource management approach translates into substantial performance improvement in real life.

Both the simulation and the measurement study were performed for real DNS workloads. We used a DNS workload from traces collected at MIT between the 4^{th} and 11^{th} of December 2000 [79]. We extracted the first 12 hours of this trace, which consisted of 265,111 total queries for 30,397 distinct DNS records. This workload closely resembles a Zipf distribution with exponent 0.91.

3.3.1 Simulations

The simulation results discussed here are drawn from experiments on a 1024 node Pastry network of base 16. For all our experiments, we started the simulation with an empty cache; DNS records were cached as queries from the trace were injected into the system. The workload was uniformly divided and queries were made to each node at an uniform rate. The total query rate to the system was approximately 6 queries per second. The aggregation interval was set to 12 minutes, the optimization interval to 120 minutes, and the target latency was set to 0.5 hops. We compare the resulting lookup performance, measured as average overlay hops, with the network bandwidth required for replication and update propagation.

We compared the tradeoffs between lookup latency and bandwidth consumption for four different configurations. The first configuration uses the analytical solution technique, which finds closed-form solutions to the tradeoff problem. This technique models the popularity of domain names as a Zipf distribution and assumes uniform update rates and sizes for all records. We call this configuration Beehive-DNS, after Beehive, the initial version of the resource management framework that only used the analytical solution technique [118]. The second configuration called Honeycomb-DNS uses the numerical technique to solve the tradeoff problem 3.2.1 taking into account the popularity, the size, and the update rate of

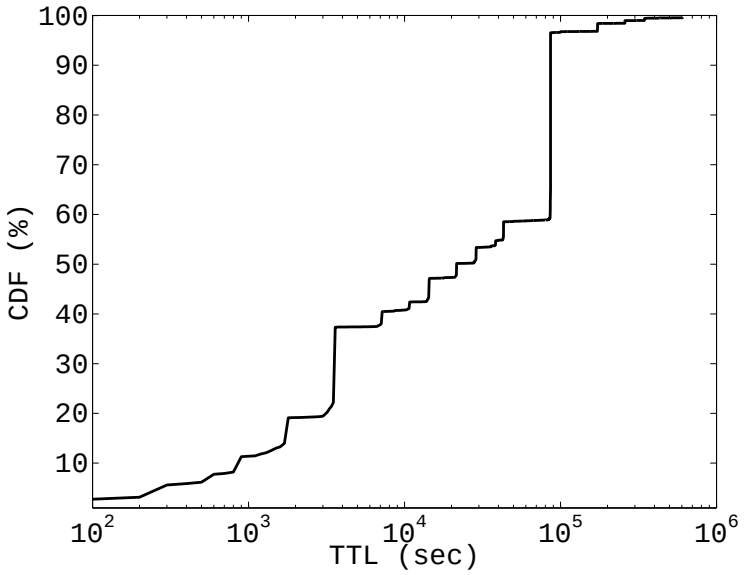

Figure 3.10: Distribution of TTLs of DNS Records: More than 67% of domains set high values for TTLs (> one hour) thereby prohibiting quick service relocation during emergencies.

Table 3.3: Parameters used in CoDoNS Deployment

Parameter	**Value**
base	16
leaf set size	24
aggregation interval	6 min
analysis interval	60 min

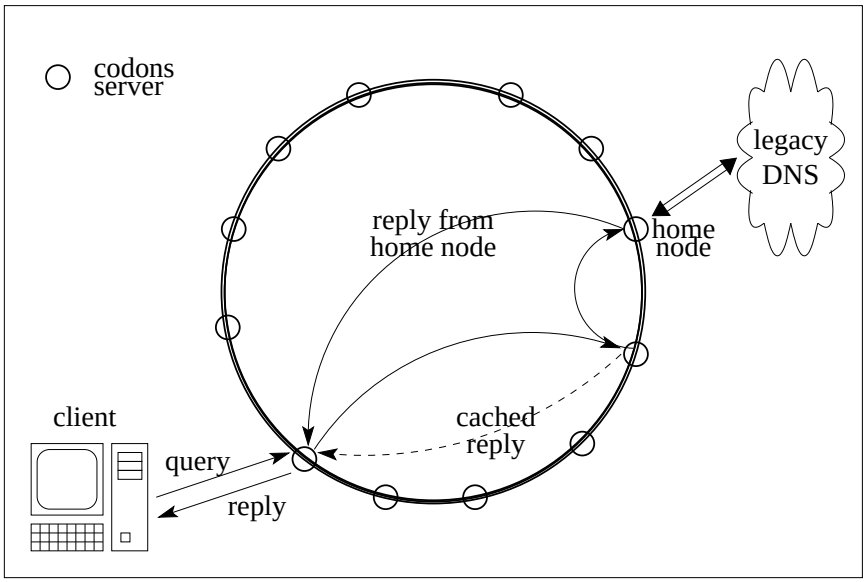

Figure 3.11: CoDoNS Architecture: CoDoNS servers self-organize to form a peer-to-peer network. Clients send DNS requests to a local CoDoNS server, which obtains the records from the home node or an intermediate node, and responds to the client. In the background, the home nodes interact with the legacy DNS to keep records fresh and propagate updates to cached copies.

each record. Third, in order to highlight the difference between the analytical and numerical technique, we also evaluate the performance of Honeycomb-DNS using uniform update rate and size. We call this crippled version HoneycombNoUpdate-DNS. Finally, we simulate a DNS system based on passive caching. This configuration called PCPastry-DNS does not use the resource management framework at all. Instead, it merely caches the record at all intermediate nodes on the lookup path. PCPastry-DNS essentially simulates a typical timeout-based passive caching approach widely used by DNS servers today.

Lookup Performance

The lookup performance of the four DNS configurations in a 12 hour simulation is shown in Figure 3.12. As expected, Beehive-DNS and both Honeycomb based systems achieve their target lookup performance, 0.5 hops, when they reach steady state. For each of these configurations, achieving the steady state takes about four hours, which corresponds to two rounds of optimization. Since the diameter of the underlying network is $log_{16}1024 \sim 2.5$, incremental allocation of levels to the records takes the expected time of two optimization rounds, to decrease the level from 2 initially to 1 and then to 0.

PCPastry, on the other hand, is only able to achieve a lookup latency of 1.6 hops. While this poor performance of PCPastry is surprising given the wide use of passive caching in the legacy DNS, the inability of PCPastry to achieve big gains in performance can be easily explained. There are two factors of the DNS workload that works against PCPastry. First is the heavy-tailed popularity distribution which means enough repeated queries do not go to the highly popular and therefore widely cached resource records. Relying on heuristics, PCPastry

cannot allocate resources adequately to improve the lookup performance further. The second reason is the use of timeout based mechanism to handle mutable objects. Cached resource records simply expire after a period of time and have to be refetched upon the next query. Together, heavy-tailed popularity distribution and short timeouts ensure that passive caching based mechanisms cannot provide adequate improvement to DNS lookup latency.

Network Bandwidth

Figure 3.13 shows the network overhead incurred by each configuration. We observe that both Honeycomb setups incurred far less network bandwidth than Beehive-DNS. Even without considering update rates, HoneycombNoUpdate-DNS was able to outperform Beehive in terms of minimizing network overhead. This is due the fact that Beehive assumes a Zipf distribution in the query load and needs to estimate the Zipf parameter of the query distribution before its replication solution becomes optimal. Inaccurate estimation of the Zipf parameter, due to insufficient global popularity information available at each node initially, causes Beehive to underestimate the Zipf exponent, resulting in increased replication and high network overhead.

Comparing Honeycomb-DNS with HoneycombNoUpdate-DNS, the former is able to reduce the network bandwidth consumption by a factor of two compared to the latter. This shows the importance of taking into account object specific characteristics. In this case, by taking advantage of the large variance in the update rate of DNS records, Honeycomb-DNS is able to resolve the tradeoff more efficiently. This comparison also highlights the performance difference between the analytical and numerical techniques.

In comparison to PCPastry-DNS, the network overhead of PCPastry-DNS is much lower than that of the other three DNS systems. This is not surprising as PCPastry-DNS, being a passive caching system, is incapable of proactively utilizing bandwidth to improve its lookup performance. Thus, while its overhead is low, its performance gains are also limited.

3.3.2 Deployment

We have deployed CoDoNS on PlanetLab [12], an open platform for developing, deploying, and accessing planetary-scale services. PlanetLab enables us to deploy CoDoNS on servers around the world and evaluate it against the background of real Internet with congestion, losses, and unpredictable failures. In this section, we present performance measurements from the PlanetLab deployment for the same DNS workload. Our experiments highlight three important properties of CoDoNS. First, they show that CoDoNS provides a low latency name resolution service. Second, they demonstrate CoDoNS' ability to resist flash crowds by quickly spreading the load across multiple servers. Finally, they evaluate CoDoNS' support for fast update propagation.

We setup a peer-to-peer network of CoDoNS servers on globally distributed PlanetLab nodes. The values used for different parameters of Pastry and Honeycomb are listed in Table 3.3. We start the CoDoNS servers with no initial DNS records. After an initial quiescent period to stabilize Pastry, we issue DNS requests from a real workload to the CoDoNS server at each node. During the experiment, we measure the lookup latency of CoDoNS, and periodically record the load handled and overhead incurred by each node. We also apply the same workload to the legacy DNS, and measure its performance. The measurements reported in this

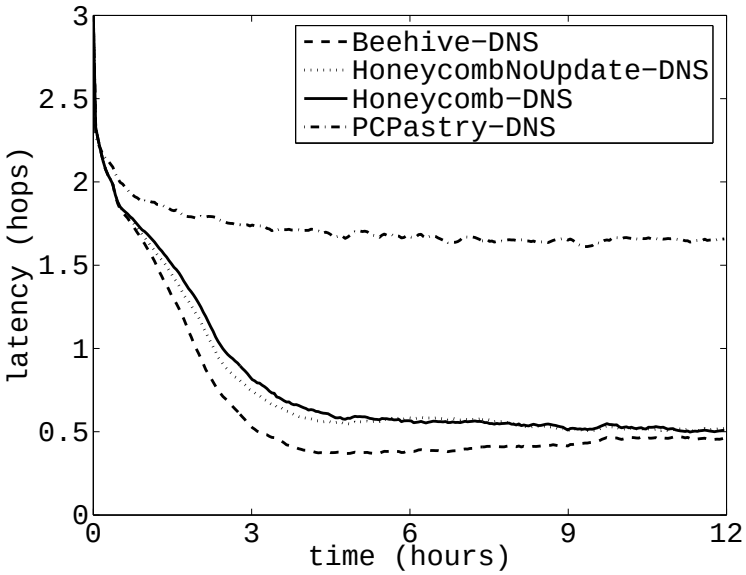

Figure 3.12: Average DNS Lookup Latency for Simulated Workload: Both Honeycomb and Beehive quickly converge to the target latency of 0.5 hops.

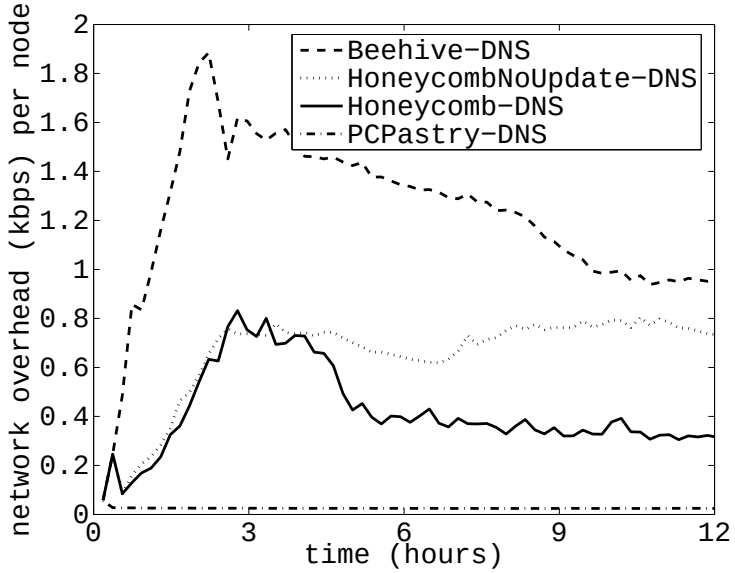

Figure 3.13: Per Node Network Overhead for DNS Simulations: Honeycomb-DNS consumes substantially lower bandwidth than Beehive-DNS by including update overhead in the analysis.

paper were taken from a deployment on 75 geographically distributed PlanetLab nodes.

Lookup Latency

Figure 3.14 shows the cumulative distribution of lookup latencies incurred by CoDoNS and the legacy DNS. Table 3.4 summarizes the results of Figure 3.14 by providing the median, mean, and the 90^{th} percentile of the latency distribution. We aggregate the latency during the second half of the workload, allowing the first half to warm the caches of both CoDoNS and the legacy DNS. The second half of the workload also contains DNS requests for domain names not present in the cache, and CoDoNS incurs the extra latency of fetching the queried records from the legacy DNS. In order to study the impact of contacting the legacy DNS, we separately evaluate the lookup performance of CoDoNS by inserting the records at their home nodes before applying the work load. This study essentially evaluates the scenario after a complete take over of the legacy DNS by CoDoNS.

50% of the queries in CoDoNS are answered immediately by the local CoDoNS server without incurring network delay, since proactive replication pushes responses for the most popular domain names to all CoDoNS servers. Consequently, CoDoNS provides a significant decrease in median latency to about 2 milliseconds compared to about 39 milliseconds for the legacy DNS. The tail of the latency distribution indicates that cache misses leading to legacy DNS lookups have an impact on the worst-case lookup performance of CoDoNS. However, a complete take over from the legacy DNS would obviate the extra latency overhead. Overall, CoDoNS achieves low latencies in the mean, median, and the 90^{th} percentile, for both deployment scenarios, with and without dependence on the legacy DNS.

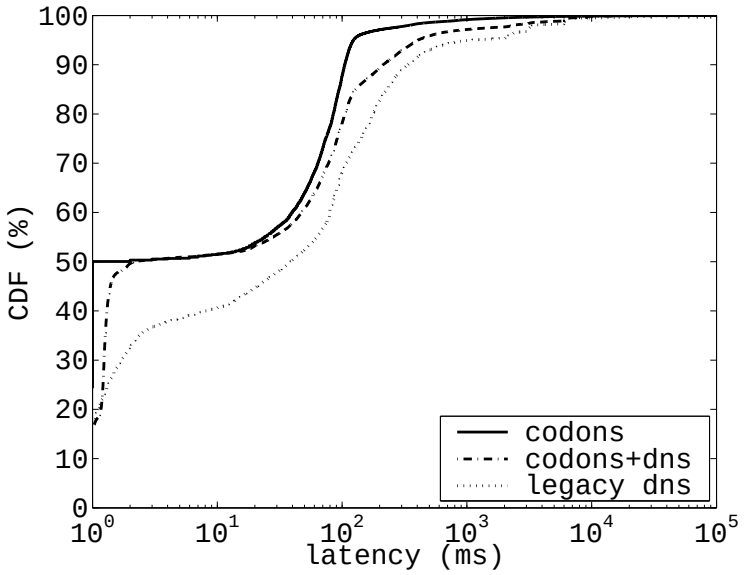

Figure 3.14: Cumulative Distribution of Latency: CoDoNS achieves low latencies for name resolution. More than 50% of queries incur no network delay as they are answered from the local CoDoNS cache.

Table 3.4: Query Resolution Latency: CoDoNS provides low latency name resolution through analytically informed proactive caching.

Latency	**Mean**	**Median**	**90^{th} %**
CoDoNS	106 ms	1 ms	105 ms
CoDoNS+DNS	199 ms	2 ms	213 ms
Legacy DNS	382 ms	39 ms	337 ms
PlanetLab RTT	121 ms	82 ms	202 ms

Figure 3.15 shows the median latency of CoDoNS and the legacy DNS over time. The fluctuations in the graph stem from the changing relative popularities of names in the workload over time. CoDoNS reacts to these changes by continuously adjusting the extent of proactive caching. Initially, CoDoNS servers have an empty cache and relies on legacy DNS for most of the queries. Consequently, they incur higher latencies than the legacy DNS. But as resource records are fetched from legacy DNS and replication in the background pushes records to other CoDoNS servers, the latency decreases significantly. The initial surge in latency can be easily avoided by bootstrapping the system with records for well known domain names.

Flash-crowd Effect

Next, we examine the resilience of CoDoNS to sudden upheavals in the popularity of domain names. To model a flash-crowd effect, we take the DNS workload and modify the second half to reflect large scale changes in the popularity of all domain names. We achieve this by completely reversing the popularities of all the domain names in the workload. That is, the least popular name becomes the most popular name, the second least popular name becomes the second most popular name, and so on. This represents a worst case scenario for CoDoNS because records that are replicated the least suddenly need to be replicated widely, and vice versa, simulating, in essence, a set of flash crowds for the least popular records.

Figure 3.16 shows the median resolution latencies in CoDoNS during the flash-crowd effect introduced at the six hour mark. There is a temporary increase in the median latency of CoDoNS when the flash crowd is introduced. But, proactive replication in the background detects the changes in popularity, adjusts the

number of replicas, and decreases the lookup latency. The latency of CoDoNS after popularity reversal quickly reaches the low values in Figure 3.15, indicating that CoDoNS has recovered completely from the worst-case, large scale changes in popularity.

Load Balance

We evaluate the automatic load balancing provided by proactive replication in CoDoNS by quantifying load balance using the *coefficient of variation*, defined as the ratio of the standard deviation of the load across all the nodes to the mean load. The overall average of query load is about 6.5 per second for the system.

Figure 3.17 shows the load balance in queries handled by CoDoNS servers, either from their internal cache or by querying the legacy DNS, for the duration of the workload. At the start of the experiment, the query load is highly unbalanced, since home nodes of popular domain names receive far greater number of queries than average. The imbalance is significantly reduced as the records for popular domains get replicated in the system. Even when a flash crowd is introduced at the six hour mark, dynamic changes in caching keep the load balanced after a temporary increase in load variance. Overall, continuous monitoring and adaptation of proactive caching enable CoDoNS to respond to drastic changes in the popularity of names and handle flash crowds.

The network bandwidth and per-node storage costs incurred by proactive caching are modest. The average bandwidth consumed over the entire experiment was 12.2 KB/s per node (std. dev. 2.26 KB/s) for all network activities. The average number of records per node was 4217 (std. dev. 348), a mere 10% of the total number of records. These records require, on average, 13 MB per node. These measure-

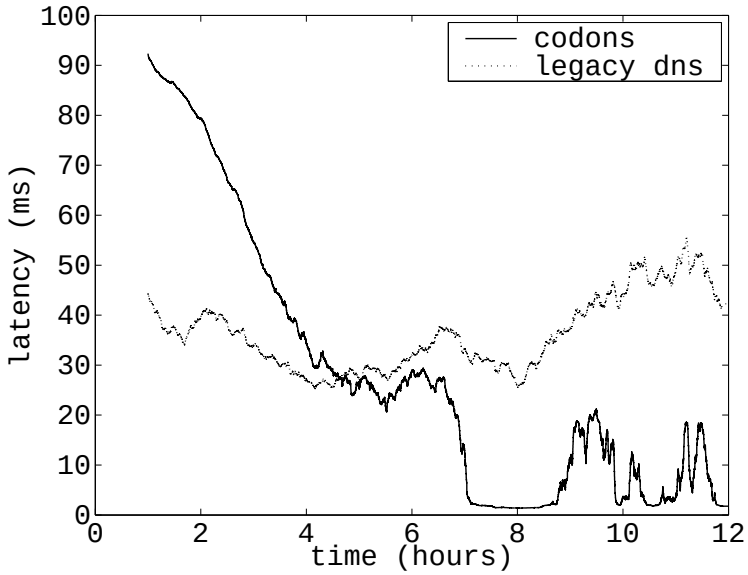

Figure 3.15: Median Latency vs Time: Lookup latency of CoDoNS decreases significantly as proactive caching takes effect in the background.

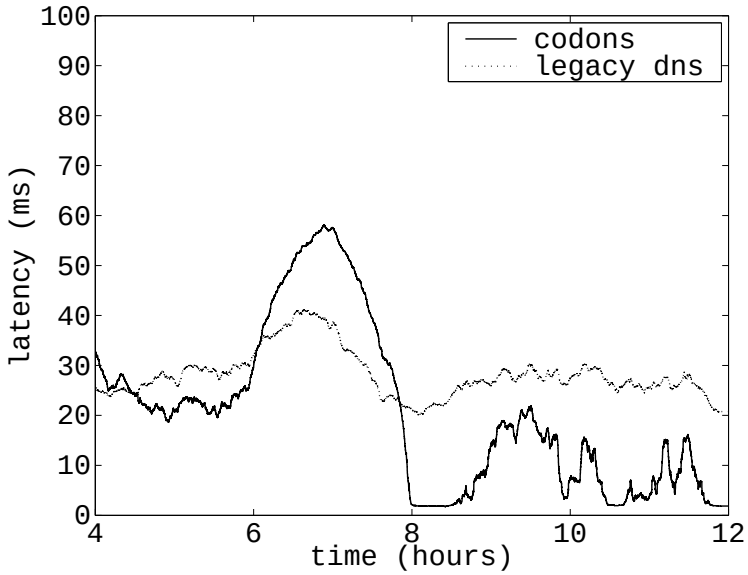

Figure 3.16: Median Latency vs Time as a flash crowd is introduced at 6 hours: CoDoNS detects the flash crowd quickly and adapts the amount of caching to counter it, while continuing to provide high performance.

ments indicate that CoDoNS distributes the load evenly across the system and incurs low uniform bandwidth and storage overhead at each node.

Update Propagation

Finally, we examine the latencies incurred by CoDoNS for proactive update propagation. Figure 3.18 shows the delay incurred for disseminating updates to resource records replicated at different levels. 98% of the replicas are updated within one second even for level-0 records, which are replicated at all nodes in the system. It takes a few seconds longer to update some replicas due to high variance in network delays and loads at some hosts. The latency to update 99% of replicas one hop from the home node is about one second. Overall, update propagation latency in CoDoNS depends on the extent of replication of records. In the worst case, it takes $\log N$ hops to update all the nodes in the network. For a million node CoDoNS network, updating 99% of replicas would take far less than a minute for even the most popular domain names replicated throughout. This enables nameowners to relocate their services without noticeable disruptions to their clients.

3.3.3 Summary

Performance measurements from a planetary-scale deployment against a real workload indicate that CoDoNS can provide low latencies for query resolution. Massive replication for the most popular records, but a modest number of replicas per server, achieves high performance with low overhead. Eliminating the static query processing hierarchy and shedding load dynamically onto peer nodes greatly decreases the vulnerability of CoDoNS to denial of service attacks. Self organization and continuous adaptation of replication avoids bottlenecks in the presence of flash

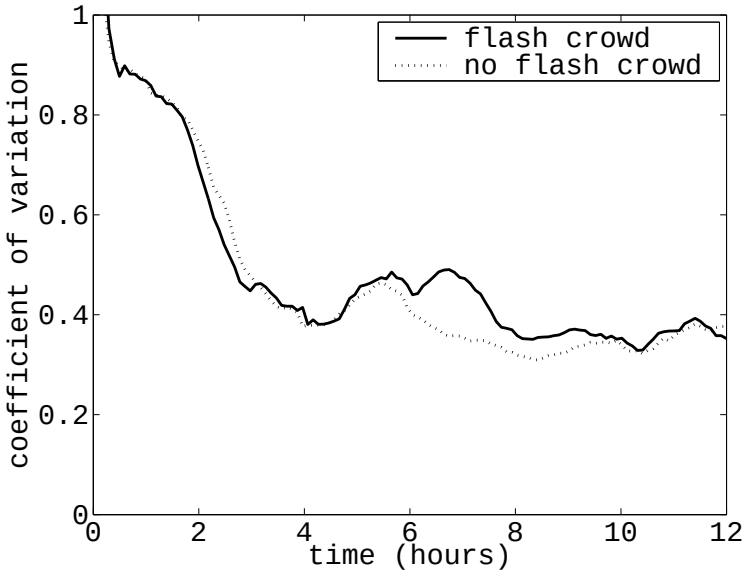

Figure 3.17: Load Balance vs Time: CoDoNS handles flash crowds by balancing the query load uniformly across nodes. The graph shows load balance as a ratio of the standard deviation to the mean across all nodes.

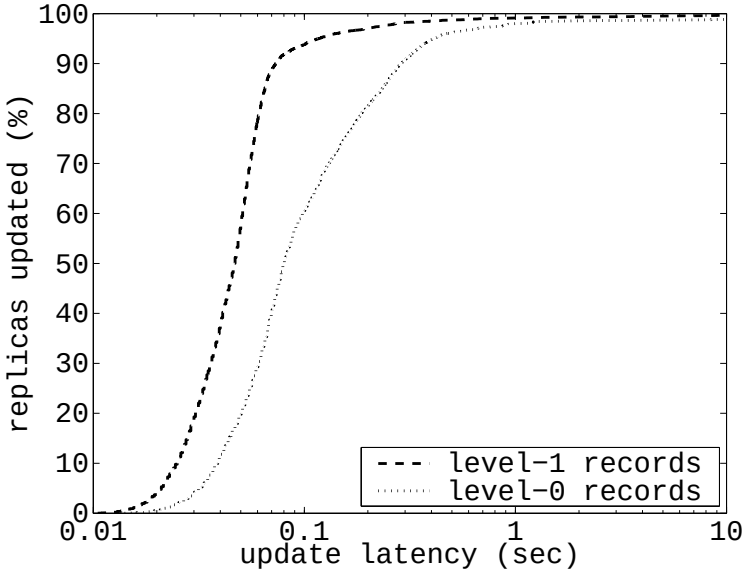

Figure 3.18: Update Propagation Time: CoDoNS incurs low latencies for propagating updates. 98% of replicas get updated within one second.

crowds. Proactive update propagation ensures that unanticipated changes can be quickly disseminated and cached in the system.

CoDoNS provides a new platform for nameowners to efficiently publish and manage their data. Our current implementation and deployment provides a simple incremental migration path from legacy DNS towards the performance and functionality offered by CoDoNS. During this process CoDoNS can serve as a safety net alongside legacy DNS.

Chapter 4

CobWeb Content Distribution Network

The Web has become increasingly important as it enables users to access information and services located throughout the world. Naturally, significant amount of effort has been spent to improve client performance, reduce server load, and minimize network traffic. The fundamental technique employed in improving web performance has been caching.

Web caches to date have been deployed in two different settings, one driven by clients and one by content providers. Web caches that are placed close to the clients exploit temporal locality within the request stream of a single user as well as spatial locality stemming from the common interests of independent users of the same proxy. These web caches, which are common to users located in several independent institutions, are called cooperative caches as they typically consist of a distributed system of caches that exchange information with each other to achieve a better overall cache hit rate. However, they depend on passive monitoring and opportunistic caching, where each proxy only caches objects that have been requested by a client that is directly connected to it. This form of passive, opportunistic caching severely limits potential benefits because web traffic is well-known to follow a Zipf distribution, with a heavy tail [19, 41, 5]. The heavy tail means that a large number of web requests go for unpopular objects making it difficult to achieve high cache hit ratios.

Web caches can also be placed within the network to aid content distribution. In particular, companies such as Akamai and Digital Island provide content distribution services to web site operators by placing servers in strategic locations to

cache and replicate content. Such networks of servers are commonly known as *content distribution networks* (CDNs), and are driven by content providers rather than content consumers. In contrast to the demand-driven nature of web proxies, most CDNs replicate web objects proactively throughout the network using heuristics aimed at load balancing and improving performance [57, 148]. These heuristics aim to maximize the effective benefit from the bandwidth spent on proactive content distribution, but typically do not provide any hard performance guarantees.

The fundamental challenge faced by any web cache is to decide which objects to replicate and to what extent. Proxy web caches sidestep this problem by passively caching objects that local clients have requested. In doing so, they limit the benefits that can be realized through caching to only those objects that have been fetched by the client population. CDNs often utilize heuristics which offer little control over the performance characteristics and resource consumption of the resulting system. Such heuristics do not provide a guaranteed way to achieve a certain hit rate or to control bandwidth consumption optimally.

Several measurement and simulation studies of different heuristics for web caching show that cache performance is limited. Breslau et al. compare cache performance of four widely-used heuristics for size different web workloads and show that no single heuristic is clearly superior to others for all workloads and the cache hit rate typically does not exceed 30% [18]. Similar negative observations for cooperative caches are reported by Wolman et al., whose study of web workloads in two large institutions show that typical cache hit rates do not exceed 40% [156].

This chapter presents a content distribution network called CobWeb [136]. CobWeb employs the optimal resource management approach described in this thesis to provide unprecedented control over the performance of a web cache. CobWeb

can be set a predetermined target cache hit rate, which it achieves with minimal resource consumption or it can provide the best cache hit rate using a limited amount of memory or bandwidth resources. The rest of this chapter describes the architecture of CobWeb and shows how well CobWeb meets its performance targets.

4.1 System Architecture

CobWeb operates as a globally distributed ring of cooperating nodes. Each CobWeb node acts as a Web proxy capable of serving HTTP requests. The techniques underlying CobWeb can be used to drive both client driven as well as server driven caches. Institutions that currently have large web caches at their gateway to the Internet may let the caches join the global CobWeb ring and share cache content intelligently and optimally. Other publicly available Web caches, such as Squid, may also be part of the CobWeb system. Alternatively, CobWeb could be deployed as a CDN under a single administrative domain such as Akamai. This chapter describes CobWeb as a cooperative web cache under a single administrative domain. Figure 4.1 illustrates this architecture of CobWeb.

CobWeb distributes objects uniformly between its nodes through consistent hashing [81]. Each web object is assigned a unique identifier that is a SHA-1 hash of its URL. When a CobWeb proxy receives a request from a client, it routes the request through the underlying overlay, directing the query toward the object's home node, the node whose identifier is numerically closest to the object's identifier. The first node along the routing path which has a copy of the object returns the object to the origin CobWeb proxy, which is responsible for delivering it to the client.

Web objects are not loaded into CobWeb unless requested. When a URL is first requested, its home node is responsible for fetching the object from the origin web server and inserting it into the system. Subsequently, the home node is also responsible for renewing the object when it expires and propagating changes to other nodes. Non-cacheable web objects are simply delivered to the client but not stored within the CobWeb system. Home nodes also delete objects from the system if they do not receive any queries over a long period of time.

CobWeb inherits high failure resilience from the overlay substrate. When a home node fails, the next closest node in the identifier automatically becomes the home node of an object. An objects for which the home node has the sole copy, simply disappear from the system. This behavior is acceptable because CobWeb serves merely as a performance enhancing soft cache, rather than a permanent store.

Users access CobWeb in a transparent way without requiring any extensions or reconfigurations to the browser. In order take advantage of CobWeb, a user merely needs to append ".cobweb.org:8888" to the main URL of a web site. The HTTP request is diverted to the closest CobWeb server through DNS-redirection. Subsequently, all web pages accessed through links on the main URL are automatically redirected through CobWeb. Alternatively, CobWeb is also available as a conventional proxy service, which can be accessed by setting the proxy options in the browser to point to the closest CobWeb node.

The above architecture facilitates deployment under a single administrative domain, such as in the Akamai model or our current deployment on PlanetLab. However, in a collaborative deployment, where nodes under different administrative domains are part of the CobWeb network, some nodes may be malicious and either

attack the overlay or corrupt the content cached in the system. This problem can be easily solved if web servers provide digitally signed certificates along with content. An alternative solution that does not require changes to servers is to use threshold cryptography to generate self-certifying content [162, 77]. When new content is to be inserted into the ring, the object can be fetched and partially-signed by a quorum of ring members. If the quorum size exceeds a threshold, partial signatures may be combined into a single signature that attests that t out of n nodes in a wedge on the CobWeb ring agree on the content. Such a scheme can ensure that rogue nodes below a threshold level cannot corrupt the system with bad content and other measures [29] can protect the underlying substrate from malicious nodes. However, the design and implementation of such a threshold-cryptographic scheme for a non-collaborative environment is beyond the scope of this thesis.

The rest of this section describes in detail the different components of the CobWeb architecture.

4.1.1 Optimal Resource Management

CobWeb is layered Honeycomb, the optimal resource management framework described in Chapter 2. Each object is cached at a *replication level* that corresponds to a well-defined region of the overlay system at a certain fixed distance from the home node of the object. The numerical optimization algorithm proposed earlier then enables CobWeb to find the optimal replication levels for each object in the system, taking into account popularity, size, and update rate.

CobWeb optimizes the average lookup performance since content distribution networks are primarily concerned with providing users with low latencies through a high hit rate. At the same time, it aims to make the best use of the resources

available. Given that disk storage is cheap and disk capacity is rapidly increasing, storage cost is unlikely to be of much interest in practice. Network bandwidth, on the other hand, is expensive and often the bottleneck in distributing large objects. Hence CobWeb primarily focuses on network bandwidth as the resource to be optimized.

CobWeb uses the analytical models derived in Chapter 2 for expressing lookup performance and bandwidth overhead. These expressions are as follows:

$$\text{Avg. Latency} = \frac{\sum_1^M q_m l_m}{\sum_1^M q_m} \tag{4.1.1}$$

$$\text{Total Bandwidth} = \sum_1^M (A + s_m u_m)\frac{N}{b^l} + a_m(l, l') \tag{4.1.2}$$

The above expressions represent average lookup latency in overlay hops and total network bandwidth in bytes per unit time including the bandwidth consumed for management, update propagation, and replication.

CobWeb computes the optimal replica strategy in two possible configurations. In the first configuration, it sets a target lookup latency, T_L, and computes the replica placement strategy that will achieve this target with the minimum cost. In this configuration, CobWeb provides a knob that allows system administrators to tune the performance of the system. For example, a target of 0.5 ensures that at least 50% of all queries do not require a network hop and results in 50% cache hit rate. In the second configuration, CobWeb minimizes the average lookup latency subject to a limit on resource consumption. Given that our measure for cost is bandwidth overhead, a system administrator can set the amount of bandwidth, T_B that CobWeb can consume over a time interval. CobWeb then computes the replica placement strategy that will produce the best lookup performance within these limits.

4.1.2 Cache Consistency Management

A common concern in maintaining replicas at multiple locations is the issue of maintaining consistency. Due to the structure of its overlay network, CobWeb is capable of efficiently maintaining consistency among objects. When a web object expires, its home node is responsible for fetching a new copy from the origin web server. This new copy is then propagated proactively to all nodes with cached copies of the object. Given the allocation level of an object, each node can determine exactly the set of nodes it needs to deliver the updates to, allowing this process to be fast and efficient.

4.1.3 User Interface

As mentioned earlier, CobWeb provides two different interfaces for different classes of uses. Users may change the proxy settings in their browser and designate a CobWeb node as a web proxy. In designating the proxy node, users can either specify the explicit address of a CobWeb node close to them, or instead use the generic proxy address "cobweb.closestnode.com". As described in Section 4.1.3 below, CobWeb uses the Meridian mechanism [158, 157] based on active measurements to locate the CobWeb node closest to the client.

Although the proxy interface is fast and relatively easy to use, it is not always possible for users to change the proxy settings of their browsers. Further, content providers, such as Slashdot, who wish to take advantage of the load shedding and performance improvement provided by the CobWeb cache may not be in a position to force their clients to modify their proxy designations. In these cases, clients can be redirected to use the CobWeb cache by appending the suffix "cob-web.org:8888" to the host name of any URL. For instance, *cnn.com* can be accessed via the URL

"http://www.cnn.com.cob-web.org:8888". Rewriting the host name suffix forces client browsers to look up the name with the CobWeb DNS server, which again uses the Meridian mechanism to route the client's request to the closest CobWeb node.

URL Rewriting

CobWeb performs URL rewriting on the fly in order to provide clients with a seamless experience, where all resources on a "cobwebbed" URL are fetched from the CobWeb cache instead of the origin server. This enables CobWeb to support high-volume sites such as Slashdot. Consider a HTML page hosted on one web server that includes many images hosted on another server with a different host name. URL rewriting ensures that when the page is requested through CobWeb, all the images will be accessed through CobWeb as well, alleviating the load on both the HTML server and the image server. URL rewriting occurs only once when a page is first fetched by a CobWeb node from the origin server. Subsequent accesses incur no overhead since the resultant page is then cached in the system.

DNS Redirection

Latency between clients and cobweb servers may form a significant portion of the overall lookup performance. To keep this latency low, it is important that users are directed to the CobWeb proxy that is closest to them. CobWeb accomplishes this by using the Meridian algorithm for closest node selection [158, 157]. When a user first queries for a cobwebbed URL, a DNS request is sent to CobWeb's DNS server, which initiates a recursive Meridian lookup. Meridian is a network service that enables clients to locate the closest node from a network of nodes. Meridian finds

the closest node by organizing the network into concentric, non-overlapping rings with exponentially increasing radii, based on the node's distance from each other. Upon a query to find the closest node, a Meridian node determines its distance d to the client using a reverse DNS query or an ICMP ping, examines its rings in the range $d/2$ to $3d/2$ to find suitable nodes, and asks those nodes to measure their distances to the client. If a suitable node is found, the query is forwarded to that node and the process continues recursively; otherwise, the current node is designated as the closest proxy for that client. The Meridian algorithm reduces the distance between the candidate proxy and the client node exponentially at each hop, has been proven to succeed with very high probability under general models for the Internet latency space, and achieves low error rates in practice.

To mask the latency of proximity detection from the client, CobWeb caches closest node information reported by Meridian. Internally, CobWeb caches measurements taken during the Meridian routing process that are used to determine inter-node distances. In addition, when the closest node to a client is found, the identity of that node is cached at the DNS server for a relatively long period of 10 minutes, allowing subsequent queries from that client to be satisfied instantly.

4.2 Evaluation

In this section, we evaluate the performance of CobWeb through extensive simulations and measurements from a real world deployment of our system.

4.2.1 Simulations

We first compare the performance of CobWeb, in its two different configurations, with Beehive, which uses the analytical approach for solving the optimization

problem. In addition, we compare CobWeb with PCPastry, a caching mechanism built on top of Pastry that passively caches objects on the intermediate nodes in the routing path, to show the difference in the characteristics of a proactive replication system such as CobWeb and that of a passive, opportunistic caching system. As a baseline for comparison, we also include plain Pastry in our simulations, with no caching at all. Finally, we examine the performance of CobWeb in the face of flash crowds and show that it is capable of quickly adapting to rapid changes in the popularity of objects.

In the experiments below, we run CobWeb in two different configurations. In the first configuration, CobWeb-TL, CobWeb is configured to achieve a target latency to guarantee high performance. In our experiments, we set this target latency to 0.5 hops, which seeks to satisfy more than 50% of queries at the local CobWeb proxy. In the second configuration, CobWeb-TB, CobWeb is set to meet a target bandwidth limit. This emulates the situation where a CDN needs to provide optimal performance subject to a resource constraint. In our experiment, we set the bandwidth limit to 0.25 KB/s. In both cases, CobWeb-TL and CobWeb-TB are configured with an aggregation interval of 12 minutes. We configure Beehive to meet the same latency target of 0.5 hops.

For each of these systems, our simulations model a 1024 node network. We inject queries to these servers based on a workload extracted from a week-long trace from a busy proxy server that is part of the IRCache project at the National Laboratory for Applied Network Research [172] in October 2004. The workload consists of a total of 409,600 queries for 10,000 objects. The workload distribution follows a Zipf distribution with parameter 0.82. The queries are uniformly divided among the clients, which send queries into the system at a steady rate. The total

query rate seen by the system is about 6 queries per second. Wherever the graphs are plotted with error bars, the experiments were repeated five times with different seeds to the random number generator and the standard deviation is plotted as the error.

Proactive Caching

Since the Beehive system and the CobWeb-TL system are both configured to meet the performance targets, we expect both systems to perform similarly once they have converged to their performance targets. CobWeb-TB, on the other hand, is expected maximize its performance while keeping to a bandwidth limit.

Figure 4.2 shows the latency average latency of CobWeb and Beehive systems over the duration of the experiment. As expected, CobWeb-TL and Beehive both converge to the target latency within the first few hours. CobWeb-TB, given its aggressive bandwidth constraints, experiences a slower improvement in latency because it has to stay within its bandwidth limit. CobWeb-TB's performance stabilizes after about 5 hours, at a steady average lookup latency of about 0.68, because its bandwidth constraints do not allow it to maintain a sufficient number of replicas to match CobWeb-TL and Beehive's performance.

While performance is an important goal for these systems, keeping network and storage overhead at a minimum is also important. Figure 4.3 shows that analytically informed caching can achieve high performance while keeping bandwidth consumption modest. Not surprisingly, CobWeb-TB converges to its target bandwidth limit of 0.25 KBps very quickly, and its bandwidth consumption remains at this level in the steady state. Both Beehive and CobWeb-TL, which target lookup performance instead of bandwidth consumption, meet their targets with a

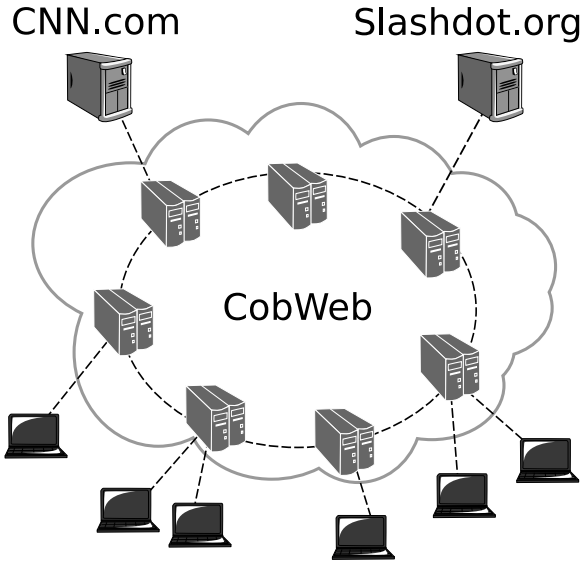

Figure 4.1: CobWeb Architecture

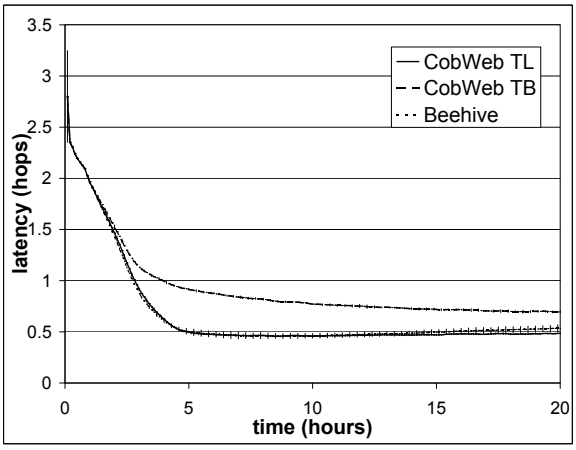

Figure 4.2: Average Lookup Latency: Beehive and CobWeb-TL quickly converge to their target latency of 0.5; CobWeb-TB achieves lower performance.

bandwidth consumption of 0.5 KBps.

Unlike CobWeb-TB, Beehive and CobWeb-TL experience an initial bandwidth spike. This is a result of the aggressive replication that occurs at the beginning of the experiment, as both systems try to rapidly improve their hit rates to meet their performance goals. Between the two, CobWeb-TL consumes much lower network bandwidth as it converges to its performance target. The reason for this lower overhead is two-fold. First, CobWeb does not require an accurate estimate of the Zipf parameter of the workload, in fact, it does not even assume a Zipf distribution, allowing it to converge to an optimal solution much faster than Beehive. Second, because CobWeb-TL takes object sizes into account when computing its replication solution, it is able to minimize network usage.

Figure 4.4 shows the storage overhead of each node during the experiment. We observe that the storage overhead of the systems corresponds closely to that of the network overhead. Beehive's storage overhead initially overshoots its steady state value before gradually settling on its steady-state value. This is a result is again due to intitial underestimation of the zipf parameter in Beehive. The CobWeb systems do not make the assumption of a Zipf distribution of the query load and are inherently not subject to this estimation error. In addition, CobWeb-TL, by preferentially replicating smaller objects, incurs only about half the overhead of Beehive while achieving the same performance.

The storage overhead of CobWeb-TB is lower because it is indirectly limited by its bandwidth constraint. Although CobWeb-TB's resource consumption limit is defined in terms of network overhead, this creates an indirect limit on storage consumption due to the fact that each replicated object consumes network bandwidth for update propagation and aggregation overhead. When the system reaches

a state where all available network bandwidth is being consumed by maintenance overhead in this fashion, CobWeb-TB can no longer cache additional objects.

Comparison to Passive Caching

We next compare CobWeb-TL to PCPastry and Pastry. For this experiment, we increased the target latency of CobWeb-TL to 1.0. This allows CobWeb-TL to match PCPastry's performance so that we can make reasonable comparisons of the two systems' resource consumption.

Being a passive caching system, PCPastry is indifferent to the popularity of objects. Therefore, we expect it to incur a significantly higher storage overhead than CobWeb, and to converge to a steady state at a much slower rate. Our simulations confirm this. Figure 4.5 shows the latency performance of PCPastry and CobWeb-TL. We observe that the latency performance of PCPastry converges very slowly to a lookup latency of about 1 hop, as cached copies of objects are slowly created throughout the network in response to the workload. CobWeb-TL converges rapidly to the targeted performance level. Being a passive system, PCPastry is incapable of providing any means of trading off more resources for performance gains.

Also, as expected, CobWeb-TL is able to accomplish the same performance target as PCPastry at much lower cost. Figure 4.6 shows the storage overhead of the two systems. The storage overhead of PCPastry increases steadily over the course of the experiment. As more queries are injected into the system, the passive caching mechanism of PCPastry indiscriminately caches every object that passes through every node. In sharp contrast, CobWeb-TL computes an optimal replication strategy and stores a much smaller set of objects at each node. Once

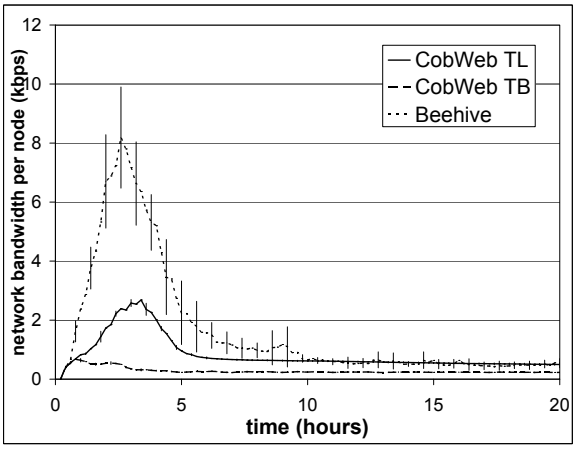

Figure 4.3: Per Node Network Overhead: CobWeb-TL incurs significantly lower network overhead than Beehive, while CobWeb-TB uses the least network overhead, being able to stay below its allotted bandwidth limit

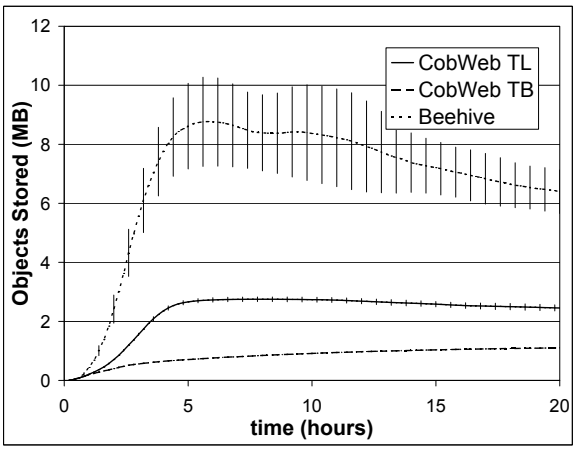

Figure 4.4: Per Node Storage Overhead: CobWeb-TL incurs significantly lower storage overhead than Beehive, while CobWeb-TB, because of its bandwidth limit constraint, incurs the least storage overhead.

CobWeb-TL achieves a steady state where it is able to meet its performance target, its storage overhead remains constant.

In the above experiments, PCPastry was set an unbounded cache size. It may appear that constraining the cache size of PCPastry may help reduce its resource consumption without a significant impact on its lookup performance. In order to test this hypothesis, we simulated PCPastry with its cache size limited to the steady state storage consumption of CobWeb-TL and used the least recently used (LRU) and least frequently used (LFU) heuristics for cache replacement. Unfortunately, under this scenario, PCPastry provided almost negligible performance improvement compared to Pastry with no caching. This experiments further highlights the fundamentally superior level of performance of principled, well-informed approaches over blind heuristics.

Flash Crowds

One of the goals of the CobWeb system is to alleviate the "Slashdot effect," also known as "flash crowds." We simulate the conditions of a flash crowd and show that CobWeb adapts rapidly to such situations. In this experiment, the workload consists of 409,600 queries for a total of 5000 unique objects. The query distribution follows a Zipf distribution with exponent 0.9 and the aggregation interval for CobWeb is set to 45 seconds. The two systems, CobWeb-TL and CobWeb-TB, are configured with a target latency of 1 hop, and a target bandwidth limit of 2 KBps respectively. In order to simulate a flash crowd, the popularities of the 10 least popular objects in the system are increased by three orders of magnitude, after 10 hours, making them the most popular objects in the system.

Figure 4.7 shows the average latency observed by clients over the course of the

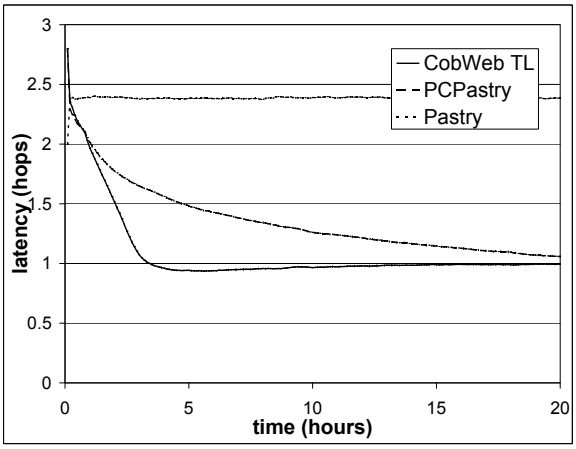

Figure 4.5: CobWeb-TL converges to the target latency of 1 rapidly, while PCPastry converges much more slowly.

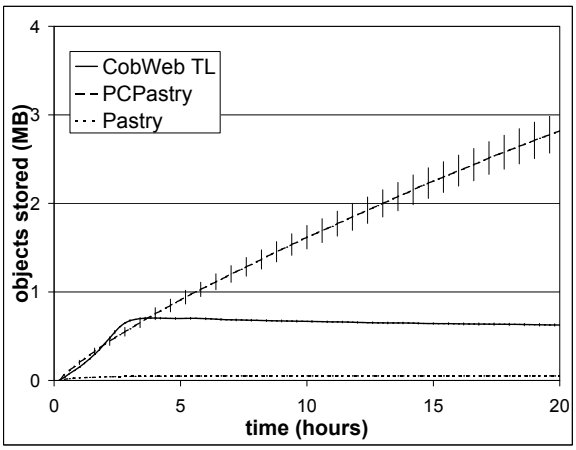

Figure 4.6: Per Node Storage Overhead: CobWeb-TL's storage overhead reaches a low, steady-state value rapidly, while PCPastry's storage overhead increases steadily overtime.

experiment. At the 10th hour, when the flash crowd occurs, both CobWeb-TL and CobWeb-TB experience a sudden increase in the average latency. However, both systems quickly recover to their steady state average latency within a matter of minutes as the systems learn the new popularity distribution and take remedial actions.

The corresponding network bandwidth consumption is shown in Figure 4.8. When the flash crowd occurs, CobWeb-TL's network bandwidth consumption increases rapidly because CobWeb-TL aggressively replicates the newly popular objects in order to meet its performance targets. CobWeb-TB, on the other hand, sees an almost steady bandwidth consumption in the face of a flash crowd. Yet, slower background replication ensures that CobWeb-TB converges back to its steady state latency.

Our results show that CobWeb performs well under flash crowd conditions. CobWeb's fast aggregation techniques allowed the system to detect changes to object popularity quickly and change replication strategy accordingly. As a result, both CobWeb-TL and CobWeb-TB were able to recover to their steady state performance within minutes. CobWeb-TB was able to accomplish this while staying within its target bandwidth limit.

4.2.2 Deployment

We next show results from a live deployment of CobWeb on PlanetLab to demonstrate that the performance benefits seen in simulations are achievable in practice.

Our deployment consists of a set of 90 widely distributed PlanetLab [12] nodes, each acting as a CobWeb server. Each CobWeb server is configured in CobWeb-TL mode, minimizing network overhead while aiming a target lookup latency of 0.5

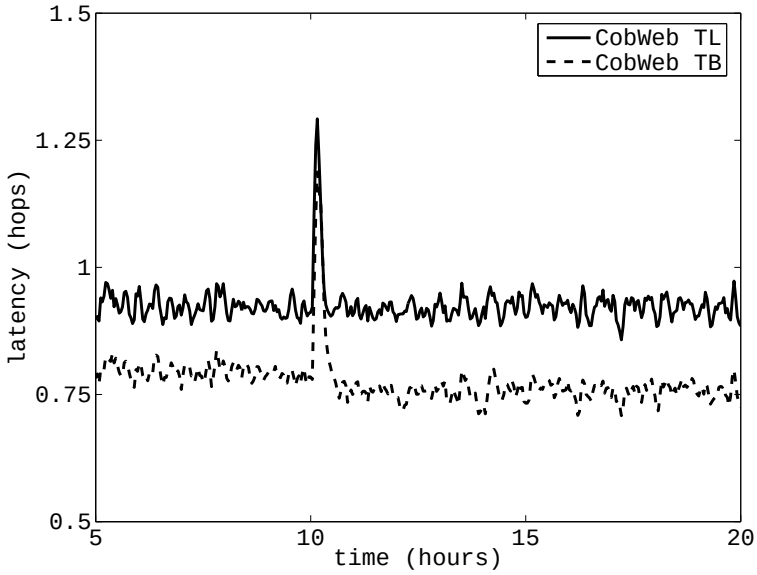

Figure 4.7: Network bandwidth consumed during a flash crowd: CobWeb-TL sees a sudden increase in network bandwidth usage which rapidly returns to its previous steady state; CobWeb-TB shows little change in network bandwidth usage.

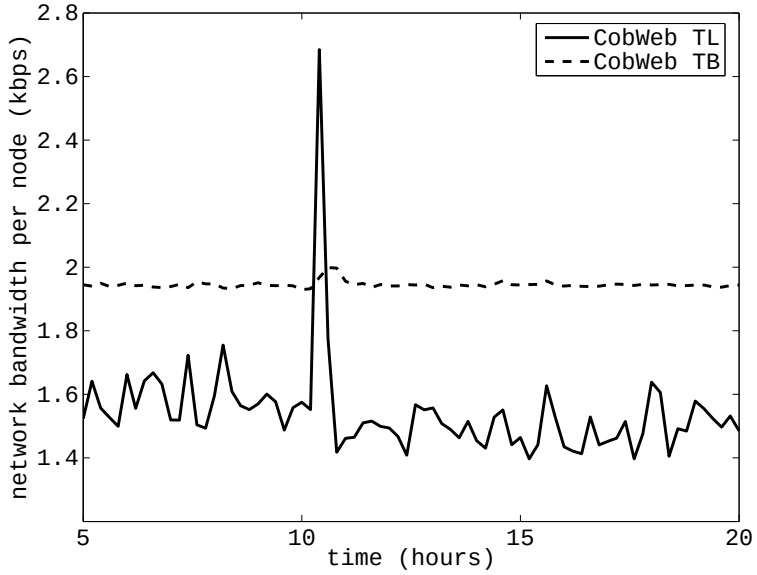

Figure 4.8: Average lookup latency during a flash crowd: Both systems see a small increase in latency, but quickly recovers to the steady state latency.

hops. The workload we use for testing our deployment is extracted from the same NLANR trace that we used for our simulations and contains a total of 100,000 queries for 24,822 unique objects. The query distribution closely follows a Zipf-like distribution with parameter 0.83. We divide this workload uniformly and issue HTTP

We divide this workload uniformly and issue HTTP requests from 20 PlanetLab nodes. The aggregate rate of queries sent into the system is about 240 queries per second. We measure the time taken to complete each query as seen by each of these clients, as well as the network overhead and cache hit rates seen by each CobWeb server. Then, we measure the latency seen by each of the clients when they fetched web objects directly from the origin servers without the use of any web proxies.

Our experimental results show that CobWeb provides a significant performance improvement over fetching objects directly from the origin server. Figure 4.9 shows the cumulative distribution of lookup latencies for fetching objects through CobWeb and directly from the origin server. Note that the horizontal axis of the graph is plotted on a log scale. We observe that the cumulative distribution graph for CobWeb rises steeply to about 0.58. This steep rise corresponds to the large portion of queries that were satisfied by a hit in the local cache. Approximately 60% of queries were satisfied in less than 30 milliseconds. In contrast, less than 5% of direct fetches were completed in that time. The graph shows that the median time to fetch an object through CobWeb was 27 milliseconds, while the median time to fetch an object directly from the origin web server was 200 milliseconds. Our measurements shows that the network overhead incurred was modest, never exceeding 500 bytes per second (Figure 4.10).

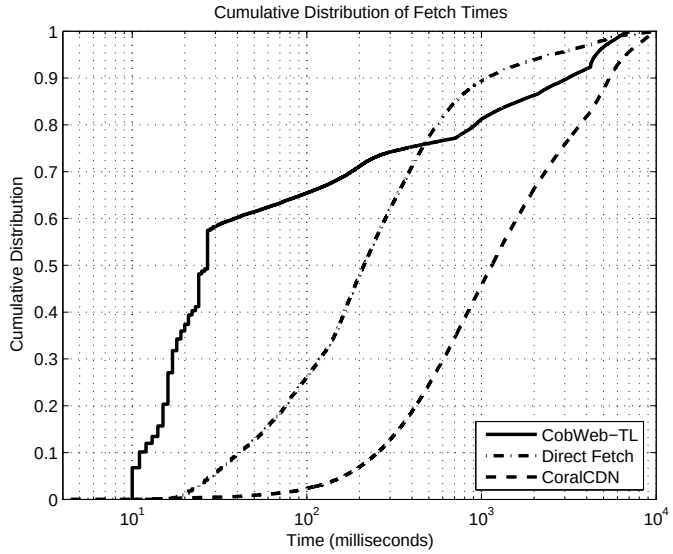

Figure 4.9: CDF of latency to fetch web objects: clients using CobWeb observed a large performance increase over clients fetching web objects directly from web servers.

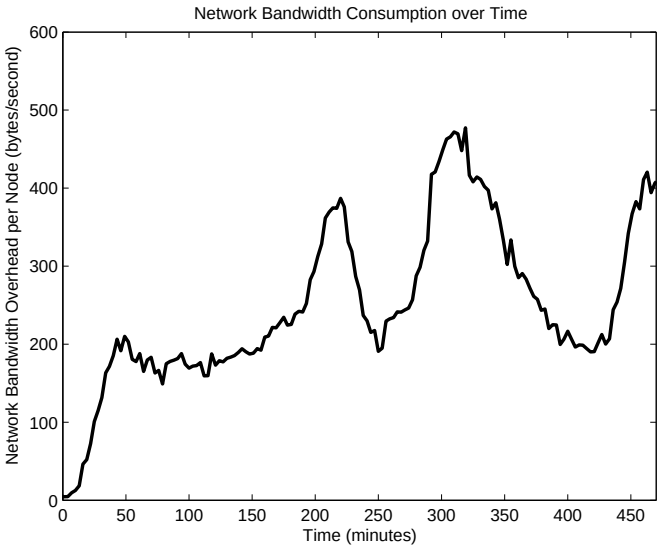

Figure 4.10: Network Overhead Per Node: CobWeb incurred a modest network overhead.

Since the implementation of CobWeb in May 2005, we have deployed the system on PlanetLab and made it available for public use. Our PlanetLab deployment runs on the same 90 nodes that were used for the measurements described above, and is configured in CobWeb-TL mode to meet a target latency of 0.5 hops. The number of requests served by CobWeb has increased steadily ever since our initial deployment. CobWeb currently serving more than 10 million requests daily. This has allowed us to observe CobWeb's behavior under a real world workload. Figure 4.11 shows the global hit rate seen by the CobWeb deployment over a one week period. The graph shows that CobWeb is capable of meeting its performance target under a real workload.

CobWeb demonstrates that informed, proactive replication is capable of supporting a high-performance content distribution network that minimizes resource overhead by taking into account object popularity, sizes, and update rate when computing the optimal replication solution. The modest network overhead incurred suggests that CobWeb can scale to support a large population of clients with a high query rate. Our experience with the deployment of CobWeb as a publicly available service on PlanetLab confirms this.

4.3 Summary

This chapter described CobWeb, a globally distributed content distribution network that applies the approach of optimization based resource management to the problem of web object caching. Simulations and real world experiments clearly demonstrated the superiority of a principled approach to a heuristic-driven one. CobWeb is able to make better use of its resources and provide a significantly better performance to its users compared to a passive web cache.

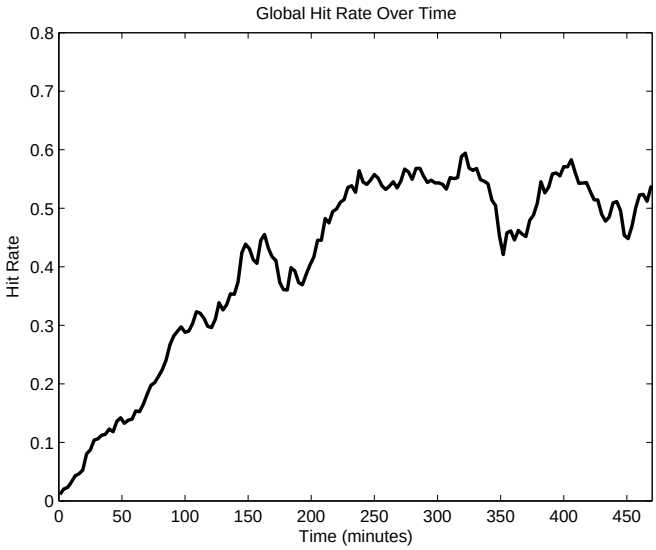

Figure 4.11: Hit Rates over time: CobWeb-TL converges to a target hit rate of 0.5

Chapter 5

Corona: Online Data Monitoring System

Online data sources have become increasingly prevalent. The growing popularity of frequently updated online content, including weblogs, collaboratively authored web pages (wikis), and news articles, motivates a *publish-subscribe* mechanism that can deliver updates to users quickly and efficiently, with low aggregate load on the network and content providers. Yet, existing web protocols do not provide a mechanism for automatically notifying users of updates.

In the research community, publish-subscribe systems have raised considerable interest over the years. A typical publish-subscribe paradigm consists of three components: *publishers*, who generate and feed the content into the system, *subscribers*, who specify content of their interest, and an infrastructure for matching subscriber interests with published content and delivering matched content to the subscribers. Based on the expressiveness of subscriber interests, two types of publish-subscribe systems have been proposed, namely *topic-based* or *content-based*. In topic-based systems, publishers and subscribers are connected together by pre-defined topics, called *channels*; content is published on well-advertised channels to which users subscribe to and receive asynchronous updates. Content-based systems enable subscribers to express elaborate queries on the content and use sophisticated content filtering techniques to match subscriber interests with published content.

The fundamental drawback of the preceding publish-subscribe systems is their non-compatibility with the current Web architecture. Publish-subscribe systems so far have primarily focused on the design and implementation of content filtering and event delivery mechanisms. Such mechanisms require substantial changes in

the way publishers serve content, expect subscribers to learn sophisticated query languages, or propose to lay out middle boxes in the core of the Internet. However, the growing popularity of a new application called *micronews syndication*, which monitors web content through naive, repeated polling, indicates that backwards compatibility with existing web tools and protocols is critical for rapid adoption.

Micronews feeds are short descriptions of frequently updated information, such as news stories and blog updates, in XML based formats such as RSS [131] and Atom [167]. They are accessed via HTTP through URLs and supported by client applications and browser plug ins called *feed readers*, which check the contents of micronews feeds periodically and automatically on the user's behalf and display the returned results. The micronews standards envision a publish-subscribe mode of content dissemination and define XML tags that tell clients how to receive asynchronous updates as well as special tags that inform clients when not to poll. Yet, few content providers currently use these tag to enable asynchronous updates. The current state of the art in micronews syndication continues to rely on repeated polling.

The current publish-subscribe architecture based on uncoordinated polling suffers from poor performance and scalability. Subscribers do not receive updates quickly, as the polling period poses a fundamental limit to the update detection time. Clients are tempted to poll at faster rates in order to detect updates quickly. Consequently, content providers have to handle the high bandwidth load imposed by clients, each polling independently and multiple times for the same content. Moreover, the workload tends to be "sticky;" that is, users subscribed to popular content do not unsubscribe after their interest diminishes, causing a large amount of wasted bandwidth.

Existing micronews syndication systems provide ad hoc, stop-gap measures to handle the performance and scalability problems. Content providers currently impose hard rate limits based on IP addresses, which render the system inoperable for users sharing an IP address (such as clients behind NATs and Firewalls), or they provide hints for when not to poll, which are discretionary and imprecise. The fundamental problem is that the server bandwidth is used inefficiently, and stems from an architecture based on naive, uncoordinated polling.

This chapter describes a novel distributed system for monitoring changes to online data sources. The presented system, called Corona, provides a high-performance update notification service without requiring any changes to the existing infrastructure, such as web servers. Corona enables any client to subscribe for updates to any existing web page or micronews feed, and asynchronously and efficiently delivers updates. Corona derives its superior performance from the optimal resource management framework described in Chapter 2. The resource management framework determines the optimal amount of bandwidth to devote to polling data sources in order to meet system-wide goals.

This chapter is organized as follows. First, it provides insights into the characteristics of micronews feeds derived through a large-scale measurement study [92]. Second, it describes in detail the architecture of the Corona system [117, 113]. Finally, it evaluates the performance of the proposed architecture based on simulations and real-life deployment.

5.1 Characteristics of Micronews Feeds

Understanding the workload characteristics of a publish-subscribe application is essential for designing a high performance pub-sub system. Currently no published

measurement study of real-life publish-subscribe application exists. We fill this breach by examining RSS syndication, the first widely deployed publish-subscribe system, which is used for disseminating Web micronews.

This section studies the feed characteristics and client behavior in the RSS system using data collected through a combination of passive logging and active polling. First, we recorded a 45-day trace from the Department of Computer Science at Cornell University. We use this trace to examine the characteristics of RSS workload, such as the popularity of RSS feeds, and user behavior, including polling rate and subscription patterns. Second, we collected snapshots of RSS content by actively polling a large number of RSS feeds.

We report on three broad aspects of the RSS system using the trace data and periodic snapshots. First, we analyze the characteristics of RSS feeds, such as the popularity distribution, content size, format, and version of RSS used. Second, we investigate how RSS feeds are updated; in particular, we focus on the update intervals of RSS feeds, the amount of change involved in updates, and correlations between updates and feed size. Finally, we examine how clients use RSS by studying their polling behavior and subscription patterns.

5.1.1 Measurement Methodology

Passive Logging: We built a software tool for tracing RSS traffic and installed it at the network border of our department. Our department is a medium-sized academic organization with about 600 graduate students, faculty, and staff. The network is topologically separated from transient users, such as undergraduates in computer labs, who do not have dedicated computers for long-running programs. We traced user activity over a 45 day period, spanning from 22 March to 3 May

2005, and recorded all RSS related traffic. The trace consists of 158 different RSS users, who subscribe to 667 feeds in total.

Our tracer software operates by capturing every TCP packet, reassembling full TCP flows, and logging the flows that contain an RSS request or response. For anonymity, we obfuscate client IP addresses using a one-way hash salted with a secret; this enables us to identify unique IP addresses without being able to map them back onto hosts. Although DHCP is used in our department, the assignment of IP addresses is decided by the physical network port used, and is therefore quite static. Laptop users that connect to public network ports may have different IPs over time, but we estimate the number of laptop users to be low compared to users with fixed IPs. The tracer tool ran on a Dell dual processor 4650 workstation, which was able to keep up with packet capture at Gigabit line speed on the link from our department to the campus backbone. We made flow assembly non performance critical by performing it offline on the captured packet stream and observed no packet drops during the whole trace period.

Active Polling: We obtained a list of 99,714 RSS feeds from *syndic8.com*, a directory that acts as a vast repository of RSS feeds. We actively polled 1000 randomly chosen feeds every ten minutes for 131 hours (more than five days) and recorded the results. While picking feeds for active polling, we ensured that no two feeds are belong to the same web site in order to make sure the results are not biased by the update behavior of any single web site. During the polls, download timeout was set to 20 seconds and a request was retried 4 times if the response was not received within the timeout period. A successful download of the RSS content gives a snapshot of the RSS feed at that time. A download may fail due to high instantaneous load on the server, network congestion, or stringent polling

limits imposed by servers. We fetched 769813 snapshots in total; that is, only 2% of snapshots were lost per feed on average.

5.1.2 Feed Characteristics

We first present statistics on RSS workload and content. We compute the popularity of RSS feeds based on the user activity traces and derive content characteristics from the snapshots of RSS feeds. We measure popularity in two ways: based on the number of requests received for each RSS feed and based on the number of clients who subscribed to each RSS feed.

Feed Popularity: Figure 5.1 shows the popularity of RSS feeds ranked by the number of requests received. The popularity follows roughly a Zipf distribution with exponent $\alpha = 1.37$. The most popular feed (BBC news) receives 12,203 requests, while there is a long tail of many feeds that receive only a single request. Figure 5.2 plots the popularity of RSS feeds based on number of subscribers observed in the trace. The distribution of subscribers also follows a Zipf distribution ($\alpha = 0.5$). The small number of clients in our trace makes the log-log plot diverge a little from the Zipf line. Overall, RSS workload has characteristics similar to Web workloads, which are also known to follow heavy-tailed power-law distributions [18].

Feed Size: RSS feeds typically consist of Web content encapsulated in XML format. Therefore, we expect the majority of RSS feeds to have size close to most Web objects. This is confirmed by Figure 5.3, which plots the distribution of feed size. The feed size is calculated as the average of all the snapshots of the feed; the variance is very small for the feed snapshots. More than 80% of the RSS feeds are relatively small at less than 10KB. The minimum observed feed size is 356 bytes,

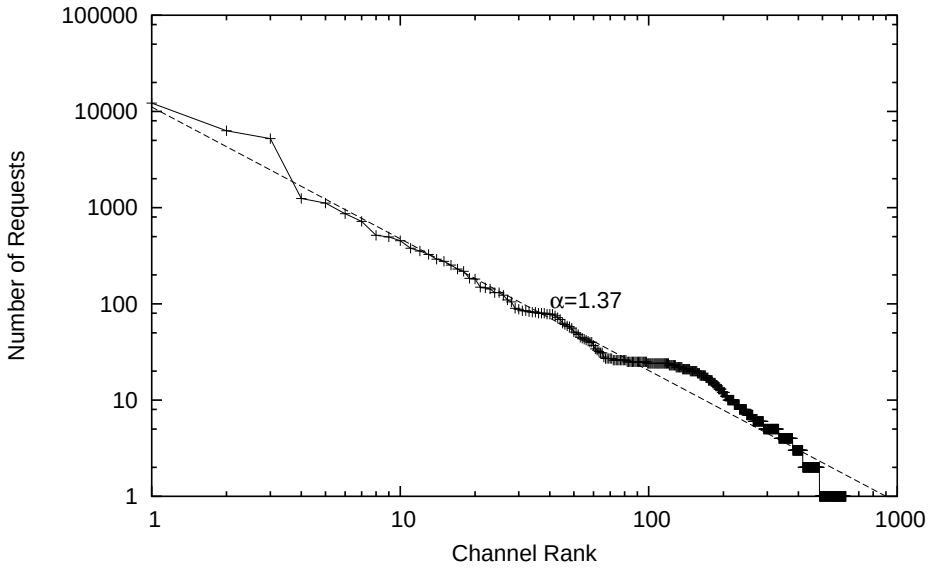

Figure 5.1: Feeds Ranked by the Number of Requests: RSS popularity follows a Zipf distribution.

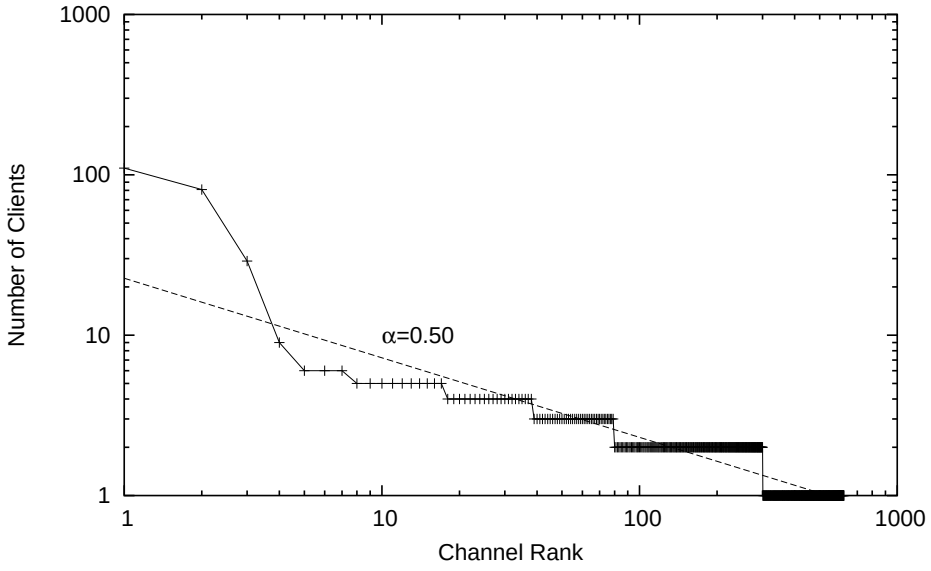

Figure 5.2: Feeds Ranked by the Number of Subscribers: RSS popularity based on subscriptions also follows a Zipf distribution.

median is 5.8KB, and the average is 10KB. While, 99.9% of feeds are smaller than 100KB, the feed size distribution is heavy tailed with the largest feed at 876,836 bytes (not shown in the graph).

Extremely large RSS feeds, however, are rare, unlike some Web objects that can be of several megabytes or more. RSS feeds are expected to be more concise than web pages because RSS is meant for the quick dissemination of news updates, often only carrying links to the more elaborate news articles. Moreover, the current architecture of RSS, where clients need to fetch the whole feed for checking updates, poses a high bandwidth load on content servers. This discourages content providers from supporting large feeds and biases towards small feed sizes.

5.1.3 Update Characteristics

Updates are the main driving force of the RSS system. We examine the nature of RSS updates using the series of hourly snapshots gathered through active polling. We ensure that missing snapshots do not affect the calculations of update interval by only counting the intervals between *valid updates*; an update is valid only if there is a valid snapshot preceding the update, and that preceding snapshot matches the last recorded update.

Update Interval: Figure 5.4 shows the average update interval of RSS feeds, calculated by averaging the valid update intervals measured for each feed. We see that feed update intervals fall in two extremes: they either update very frequently or very rarely. Over 50% of the RSS feeds do not change at all during the entire polling period of 5 days. At the same time, other RSS feeds change at a fairly rapid rate. About 9% of the feeds change at least once every hour. A significant 5% get updated every ten minutes. Since we gathered snapshots every ten minutes, our

data do not show updates that happen within that time. Nevertheless, we find that RSS feeds have widely varying update intervals. This result suggests that RSS readers should use different polling periods for different feeds. However, most RSS readers poll all the feeds at a uniform rate.

Update Size: We quantify update sizes using the minimum edit distance ("diff") between two consecutive snapshots. Figure 5.5 shows the cumulative distribution of update sizes. 64% of all updates involve no more than two lines of changes. The average change in the number of lines is 16.7 (6.8% of feed size) and the maximum is 16,542. The feed that changes most is hosted by a weather service website that provides weather forecast for many areas.

The major criticism against RSS has centered around its scalability. The constant polling by clients poses a significant bandwidth challenge on RSS servers. This study indicates that it is highly desirable to send clients only the "delta," that is, the portion of data that actually changes. Our measurement shows that the feed updates only 6.8% of its content on average, which suggests that this optimization can reduce bandwidth consumption by as much as 93.2%.

5.1.4 Client Behavior

Finally, we analyze how clients use the RSS system from the user activity trace we collected.

Polling Frequency: We divide the clients into two categories, namely *auto and manual*, according to their polling behavior. Auto clients poll feeds at a fixed rate, usually by running RSS readers in the background, while manual clients use RSS in the same way as they browse the Web, that is, launch RSS readers when they really want to read the news, and close the program after reading it. We

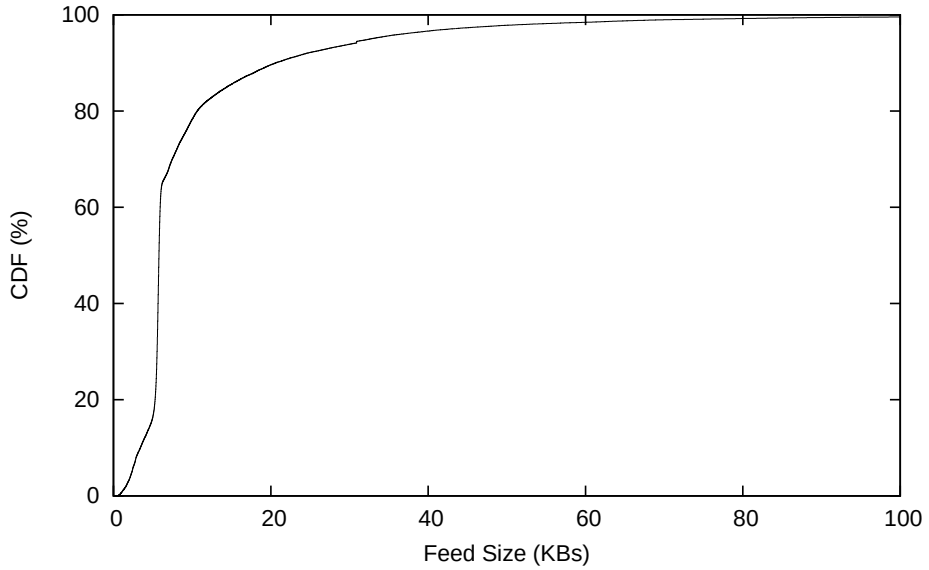

Figure 5.3: CDF of Feed Size: RSS feeds are typically small (less than 10 KB) with a median of about 5.8 KB.

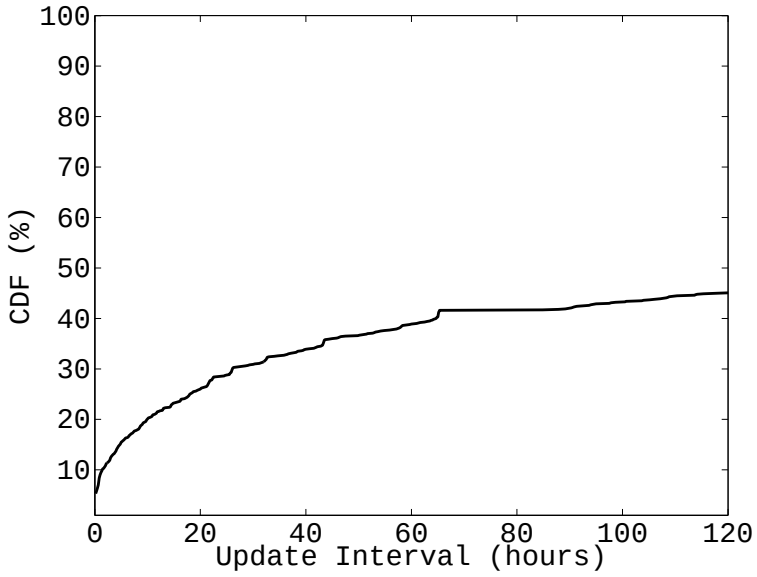

Figure 5.4: Average Update Time: 9% of feeds have average update interval of less than one hour, while 50% of feeds do not change for more than five days.

consider clients who poll a feed for less than 3 times a day or with irregular polling intervals as manual clients. We find that 36% of clients in our department fall in this category. For auto clients, who poll at periodic intervals, we show the polling rate in Figure 5.6. 58% of them poll feeds hourly, suggesting that most users do not change the default setting of their RSS readers. A small number of aggressive clients poll as often as every ten minutes.

Number of Subscriptions: Figure 5.7 shows the number of feeds subscribed by each client in sorted order. This distribution also follows a Zipf distribution with exponent $\alpha \sim 1.13$. While most clients subscribe to less than five feeds, there are several clients that subscribe to more than 100 feeds.

5.1.5 Summary of RSS Characteristics

The measurement study of RSS provides insights about how a publish-subscribe system is utilized in practice and what issues need to be addressed while designing publish-subscribe systems.

The main focus of our study is to analyze how feeds are updated, a fundamental aspect of pub-sub systems. This study shows that update rates of RSS feeds are distributed in extremes; many feeds (9%) update every hour, while a large number of feeds (50%) do not change for days. Hence, significant bandwidth savings can be obtained by using the optimal polling period for each feed instead of a single common polling rate for all feeds. End users of RSS, however, cannot be relied on to set the optimal polling rate, as this study shows that clients predominantly do not change the default settings of RSS readers. A better solution is for content providers to indicate when and at what rate to poll a particular feed. The version 2.0 of RSS already provides support for customized polling, although many readers

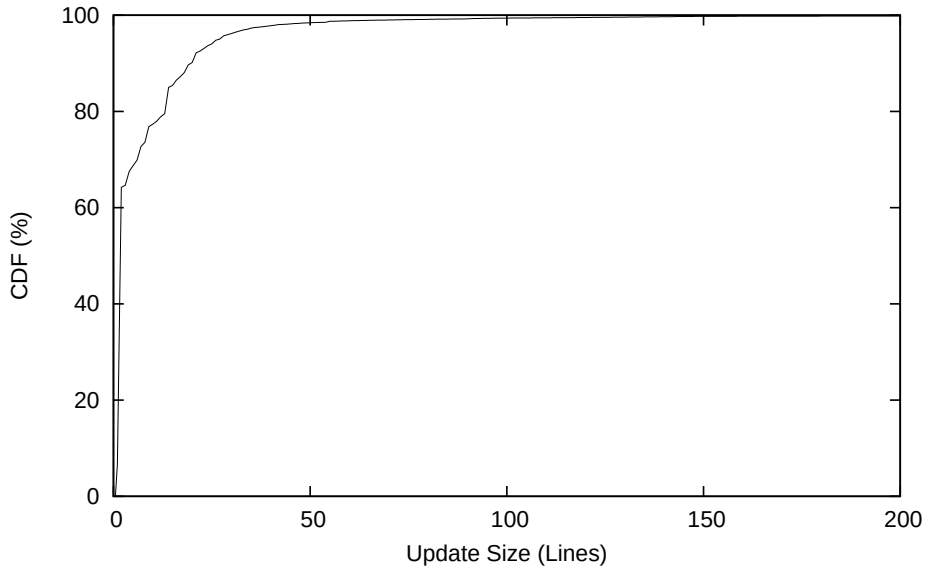

Figure 5.5: Number of Changed Lines in Updates: 64% of updates involve no more than two lines of change.

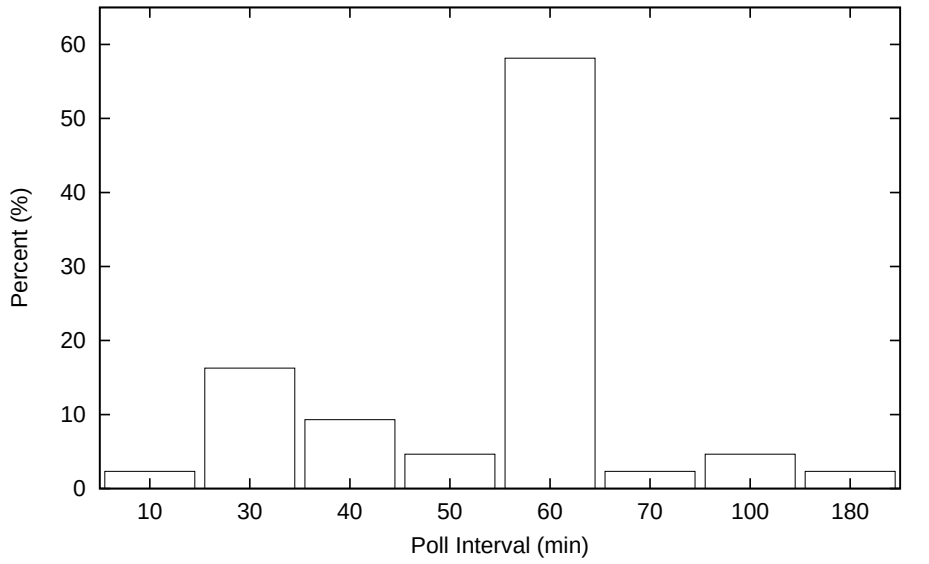

Figure 5.6: Polling Rate of Clients: About 58% of clients use the default setting of one hour as the polling period.

are yet to support this feature.

Much of the bandwidth in RSS goes towards refetching feeds in order to check for updates because the current RSS architecture does not employ asynchronous notifications. This study indicates that delta encoding is a major opportunity for improving bandwidth usage in RSS, as updates are often made only to a tiny portion of the content (about 7% of the feed on average). Finally, clients subscribed to the same feed poll the content servers independently, imposing a high load on the servers of popular feeds.

5.2 Corona: Architecture

Corona (Cornell Online News Aggregator) is a topic-based publish-subscribe system for monitoring online data sources. It provides asynchronous update notifications to clients, while interoperating with the current pull-based architecture of the Web. URLs of Web content serve as topics or *channels* in Corona; users register their interest in some Web content by providing its URL and receive updates asynchronously about changes posted to that URL. Any web object identifiable by a URL can be monitored with Corona. In the background, Corona checks for updates on registered channels by cooperatively polling the content servers from geographically distributed nodes.

We envisage Corona as an infrastructure service offered by a set of widely-distributed nodes. These nodes may be all part of the same administrative domain, such as Google, or consist of server-class nodes contributed by participating institutions. By participating in Corona, institutions can significantly reduce the network bandwidth consumed in frequent redundant polling for content updates, as well as reduce the peak loads seen at content providers that they themselves

may host.

The central feature that enables Corona to achieve fast update detection is *cooperative polling*. Corona assigns multiple nodes to periodically poll for the same channel and shares updates detected by any polling node. In general, n nodes polling with the same polling interval and randomly distributed polling times can detect updates n times faster if they share updates with each other. Corona makes informed decisions on allocating polling tasks among nodes by using the optimal resource management framework proposed earlier. Figure 5.8 illustrates the overall architecture of Corona.

This section provides detailed descriptions of the components of Corona's architecture, including the analytical models, update detection and notification mechanisms, and the user interface.

5.2.1 Analytical Models

The key resource tradeoff in a publish-subscribe system where publishers are exogenous entities that serve content only when polled involves bandwidth versus update latency. Clearly, polling data sources more frequently will enable the system to detect and disseminate updates earlier. Yet polling every data source constantly would place a large burden on publishers, congest the network, and potentially run afoul of server-imposed polling limits that would ban the system from monitoring the data source. The goal of Corona, then, is to maximize the effective benefit of the aggregate bandwidth available to the system, while remaining within server-imposed bandwidth limits.

Corona resolves the fundamental tradeoff between bandwidth and update latency by expressing it formally as an optimization problem with performance re-

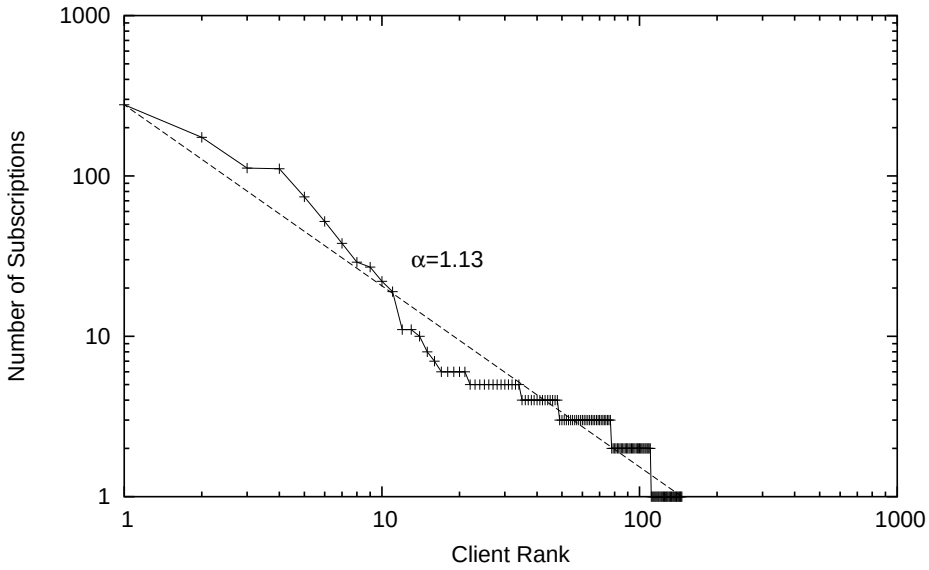

Figure 5.7: Number of Subscriptions made by Clients: The number of channels subscribed by clients follows a Zipf distribution.

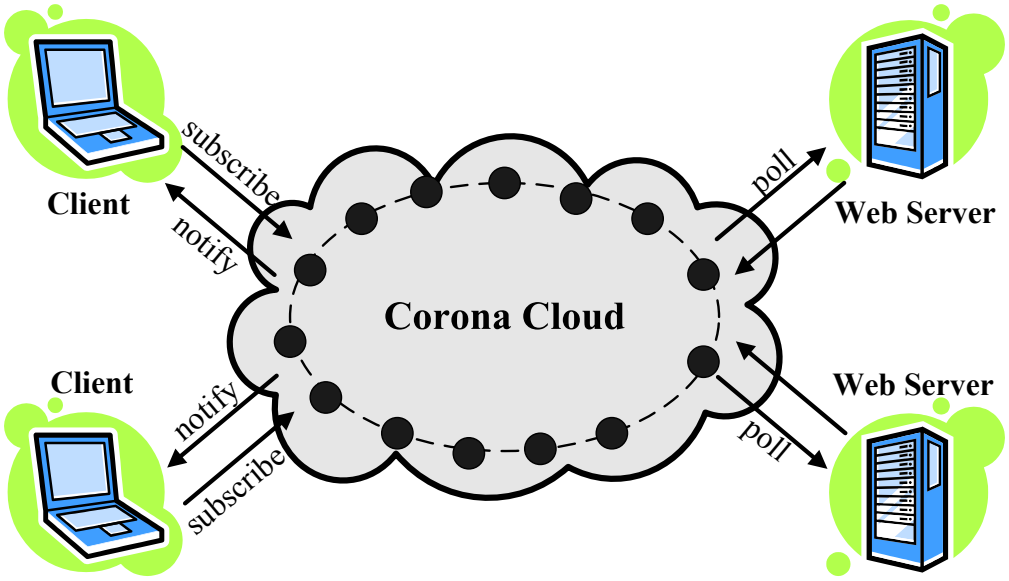

Figure 5.8: Corona Architecture: Corona is a distributed publish-subscribe system for the Web. It detects Web updates by polling cooperatively and notifies clients through instant messaging.

quirements expressed as constraints. We explore three different performance goals that are important for a publish-subscribe system.

Corona-Lite: The first performance goal we set is to minimize the average update detection time while bounding the total network load placed on content servers. Corona-Lite improves the update performance seen by the clients while ensuring that the content servers handle a light load, no more than what they would handle from the clients if the clients fetched objects directly from the servers.

Table 5.1 shows the optimization problem for Corona-Lite. The overall update performance is measured by taking an average of the update detection time for each channel weighed by the number of clients subscribed to that channel. We weigh the average using the number of subscriptions because update performance is an end user experience and each client counts as a separate unit in the average. The target network load for this case is simply the total number of subscriptions seen by the system.

Corona-Lite clients experience the maximum benefits of cooperation. Clients of popular channels gain greater benefits than clients of less popular channels. Yet, Corona-Lite avoids suffering from diminishing returns and uses its surplus polling capacity on less popular channels where the extra bandwidth yields higher marginal benefit. Since improvement in update performance is inversely related to the number of polling nodes, there is little benefit in increasing the number of polling nodes beyond a point. A heuristic based scheme that assigns polling nodes in proportion to number of subscribers would clearly suffer from diminishing returns. Corona, on the other hand, distributes the surplus load to other, less popular channels, achieving a better global average update detection time. Consequently, a less popular channel also gains substantial performance improvement

compared to what cooperation between that channel's clients alone can achieve.

Corona-Fast: While Corona-Lite bounds the network load on the content servers and minimizes update latency, the update performance it provides can vary depending on the current workload. Corona-Fast provides stable update performance, which can be maintained steadily at a desired level through changes in the workload. Corona-Fast solves the converse of the above optimization problem; that is, it minimizes the total network load on the content servers while meeting a target average update detection time. Corona-Fast enables us to tune the update performance of the system according to application needs. For example, a stock-tracker application may pick a target of 30 seconds to quickly detect changes to stock prices.

Corona-Fast shields legacy web servers from sudden increases in load. Sudden increase in the number of subscribers for a channel does not trigger a corresponding increase in network load on the web server, since Corona-Fast does not increase polling after diminishing returns sets in. In contrast, in legacy RSS, popularity spikes cause a significant increase in network load on content providers. Corona-Fast protects content servers from flash crowds and sticky traffic.

Corona-Fair: Both Corona-Fast and Corona-Lite do not consider the actual rate of change of content in a channel. As observed in section 5.1, update intervals for Web objects are known to vary considerably from a few minutes to no change over several days. Corona-Fair incorporates the update rate of channels into the performance tradeoff in order to achieve a fairer distribution of update performance between channels. It defines update performance as a ratio of the update detection time and the update interval of the channel and aims to minimize the modified metric for a load target.

While the new metric accounts for the wide difference in update characteristics, it biases the performance unfavorably against channels with large update interval times. A channel that does not change for several days experiences long update detection times, even if there are are many subscribers for the channel. Corona addresses this imbalance by exploring other update performance metrics based on square root and logarithm functions, which grow sub-linearly. A sub-linear metric dampens the tendency of the optimization algorithm to punish slow-changing yet popular feeds. Table 5.1 summarizes the optimization problems for different versions of Corona.

5.2.2 System Management

Corona is a completely decentralized system, where nodes act independently, share load, and achieve globally optimal performance through mutual cooperation. Corona spreads load uniformly among the nodes through consistent hashing [81]. Each channel in Corona has a unique identifier and one or more *owner nodes* managing it. The identifier is a content hash of the channel's URL and the *primary owner* of the channel is the Corona node with the closest identifier to the channel, that is, it's home node. Corona sets additional owners for a channel in order to tolerate failures. These owners are the f-closest neighbors of the primary owner along the ring. In the event an owner fails, a new neighbor automatically replaces it.

Owners take responsibility for managing subscriptions, polling, and updates for a channel. They keep state about the subscribers of a channel and send notifications to them when fresh updates are detected. In addition, owners also keep track of channel specific factors that affect the performance tradeoffs, namely the number of subscribers, the size of the content, and the interval at which servers

Table 5.1: Performance-Overhead Tradeoffs: This figure summarizes the optimization problems for different versions of Corona.

Corona-Lite:

$$\text{min. } \sum_1^M q_i \frac{b^l{}_i}{N} \qquad \text{s.t. } \sum_1^M s_i \frac{N}{b^l{}_i} \;\leq\; \sum_1^M s_i q_i$$

Minimize average update detection time, while bounding the load placed on content servers.

Corona-Fast:

$$\text{min. } \sum_1^M s_i \frac{N}{b^l{}_i} \qquad \text{s.t. } \sum_1^M q_i \frac{b^l{}_i}{N} \;\leq\; T \sum_1^M q_i$$

Achieve a targeted average update detection time, while minimizing the load placed on content servers.

Corona-Fair:

$$\text{min. } \sum_1^M q_i \frac{\tau}{u_i} \frac{b^l{}_i}{N} \qquad \text{s.t. } \sum_1^M s_i \frac{N}{b^l{}_i} \;\leq\; \sum_1^M s_i q_i$$

Minimize average update detection time w.r.t. expected update frequency, bounding load on content servers.

Corona-Fair-Sqrt:

$$\text{min. } \sum_1^M q_i \frac{\sqrt{\tau}}{\sqrt{u_i}} \frac{b^l{}_i}{N} \qquad \text{s.t. } \sum_1^M s_i \frac{N}{b^l{}_i} \;\leq\; \sum_1^M s_i q_i$$

Corona-Fair with sqrt weight on the latency ratio to emphasize infrequently changing channels.

Corona-Fair-Log:

$$\text{min. } \sum_1^M q_i \frac{\log \tau}{\log u_i} \frac{b^l{}_i}{N} \qquad \text{s.t. } \sum_1^M s_i \frac{N}{b^l{}_i} \;\leq\; \sum_1^M s_i q_i$$

Corona-Fair with log weight on the latency ratio to emphasize infrequently changing channels.

update channel content. The latter is estimated based on time between updates detected by Corona.

Corona relies on the mechanisms provided by the Honeycomb resource management framework for managing its polling allocation. As described in Chapter 2, Honeycomb manages polling using light-weight mechanisms that impose a small, predictable overhead on the nodes and network. Its decision making does not rely on expensive constructs such as consensus, leader election, or clock synchronization. All networking activity is local and limited to contacts in the routing table.

5.2.3 Update Dissemination

Updates are central to the operation of Corona; hence, we ensure that they are detected and disseminated efficiently. Corona uses monotonically increasing numbers to identify versions of content. The version numbers are based on content modification times whenever the content carries such a timestamp. For other channels, the primary owner assigns version numbers in increasing order based on the updates received by it.

Corona nodes share updates only as *diffs*, the difference between old and new content, rather than the entire content. A *difference engine* enables Corona to identify when a channel carries new information that needs to be disseminated to subscribed clients. The difference engine parses the HTML or XML content to discover the core content in the channel, ignoring frequently changing elements such as timestamps, counters, and advertisements. The difference engine generates a diff if it detects an update after isolating the core content. The data in a diff resembles the typical output of the POSIX 'diff' command; it carries the line

numbers where the change occurs, the changed content, an indication whether it is an addition, omission or replacement, and a version number of the old content to compare against.

When a diff is generated by a node, it shares the update with all other nodes at the same polling level as the channel. To achieve this, the node simply disseminates the diff along the DAG rooted at it up to a depth equal to the polling level of the channel. The dissemination along the DAG takes place using contacts in the routing table of the underlying overlay. For channels that cannot obtain a reliable modification timestamp from the server, the node detecting the update sends the diff to the primary owner, which assigns a new version number and initiates the dissemination to other nodes polling that channel. Two different nodes may detect a change "simultaneously" and send diffs to the primary owner. The primary owner always checks the current diff with the latest updated version of the content and ignores redundant diffs.

5.2.4 User Interface

Corona employs instant messaging (IM) as its user interface. Users add Corona as a "buddy" in their favorite instant messaging system; both subscriptions and update notifications are then transported as instant messages between the users and Corona. Users send request messages of the form "subscribe url" and "unsubscribe url" to subscribe and unsubscribe for a channel. A subscribe or unsubscribe message delivered by the IM system to Corona is routed to all the owner nodes of the channel, which update their subscription state. When a new update is detected by Corona, the current primary owner sends an instant message with the diff to all the subscribers through the IM system. If a subscriber is off-line at the time

an update is generated, the IM system buffers the update and delivers it when the subscriber subsequently joins the network.

Delivering updates through instant messaging systems may incur some additional latency, but this latency is typically modest. Instant messaging systems are already designed to reduce such latencies during two-way communication. Moreover, IM systems that allow peer-to-peer communication between their users, such as Skype, do not suffer from the additional latency of tunnelling through a centralized service.

Instant messaging enables Corona to be easily accessible to a large user population, as little computer skill is required. It is freely accessible for users behind public-access computers, which restrict users from reconfiguring the system, as well as users behind firewalls since instant messaging connections are moderated by centralized services on well-defined ports. Moreover, instant messages also guarantee the authenticity of the source of update messages to the clients, as instant messaging systems pre-authenticate Corona as the source through password verification.

5.2.5 Issues and Implications

Corona interacts with IM systems using GAIM [61], an open source instant messaging client for Unix based platforms. Our current version supports multiple IM systems including Yahoo Instant Messenger, AOL Instant Messenger, and MSN Messenger. Some of these IM systems pose a limitation that only one instance of a user can be logged on at a time, preventing Corona nodes to be all logged on at the same time. While we hope that IM systems will support simultaneous logins from automated users such as Corona in the near future, as they have for

certain chat robots, our current implementation uses a centralized server to talk to IM systems as a stop-gap measure. This server acts as an intermediary for all update diffs sent to clients as well as subscription messages sent by clients. Also, a few IM systems, such as Yahoo, rate limit instant messages sent by unprivileged clients. Corona's implementation limits the rate of updates sent to clients and avoids sending updates in bursts.

Corona trusts the nodes in the system to behave correctly and generate authentic updates. However, it is possible that in a collaborative deployment, where nodes under different administrative domains are part of the Corona network, some nodes may be malicious and generate spurious updates. This problem can be easily solved if content providers are willing to publish digitally signed certificates along with the content. An alternative solution that does not require changes to servers is to use threshold cryptography to generate a certificate for content [162, 77]. The responsibility for generating partial signatures can be shared among the owners on a node ensuring that rogue nodes below the threshold level cannot corrupt the system.

5.3 Evaluation

We evaluate the performance of Corona through large-scale simulations and wide-area experiments on PlanetLab [12]. In all our evaluations, we compare the performance of Corona with the performance of legacy RSS, a widely-used micronews syndication system. The simulations and experiments are driven by real-life RSS traces described in Section 5.1.

5.3.1 Simulations

In order to scale the workload to the larger scale of our simulations, we extrapolate the distribution of feed popularity from the workload traces and set the popularity to follow a Zipf distribution with exponent 0.5. For update rate of channels, we use distribution obtained through active polling, setting the update interval of the channels that do not see any updates to one week.

We perform simulations for a system of 1024 nodes, 80,000 channels, and 4,000,000 subscriptions. We start each simulation with an empty state and issue all subscriptions at once before collecting performance data. We run the simulations for six hours with a polling interval of 30 minutes and maintenance interval of one hour. We study the performance of the three schemes, Corona-Lite, Corona-Fast, and Corona-Fair, and compare the performance with that of legacy RSS clients polling at the same rate of 30 minutes.

Corona-Lite

Figures 5.9 and 5.10 respectively show the network load and update performance for Corona-Lite, which minimizes average update detection time while bounding the total load on content servers. The figures plot the network load, in terms of the average bandwidth load placed on content servers, and update performance, in terms of the average update detection time. Figure 5.9 shows that Corona-Lite stabilizes at its target load equal to that imposed by legacy RSS clients. Starting from a clean slate, where only owner nodes poll for each channel, Corona-Lite quickly converges to its target in two maintenance phases. The average load exceeds the target for a brief period before stabilization. This slight delay is due to nodes not having complete information about characteristics of other channels in

the system. However, the discrepancy is corrected automatically when aggregated global channel characteristics are available to each node.

At the same time, Figure 5.10 shows that Corona-Lite achieves an average update detection time of about one minute. The update performance of Corona-Lite represents an order of magnitude improvement over the average update detection time of 15 minutes provided by legacy RSS clients. This substantial difference in performance is achieved through judicious distribution of polling load between cooperating nodes, while imposing no more load on the servers than the legacy clients.

Figures 5.11 and 5.12 show the number of polling nodes assigned by Corona-Lite to different channels and the resulting distribution of update detection times. The X axis shows channels in reverse order of popularity. We only plot 20,000 channels for clarity. The load imposed by legacy RSS is equal to the number of clients. For Corona-Lite, three levels of polling can be identified in Figure 5.11, channels clustered around 100 at level 1, channels with less than 10 clients at level 2, and orphan channels close to the X axis with just one owner node polling them. The sharp change in the distribution after 60,000 channels indicates the point where the optimal solution changes polling levels.

Figure 5.11 shows that Corona-Lite favors popular channels over unpopular ones when assigning polling levels. Yet, it significantly reduces the load on servers of popular content compared to legacy clients, which impose a highly skewed load on content servers and overload servers of popular content. Corona-Lite reduces the load at the over-loaded servers, and transfers the extra load to servers of less popular content to improve their update performance.

The favorable behavior of Corona-Lite is due to diminishing returns caused by

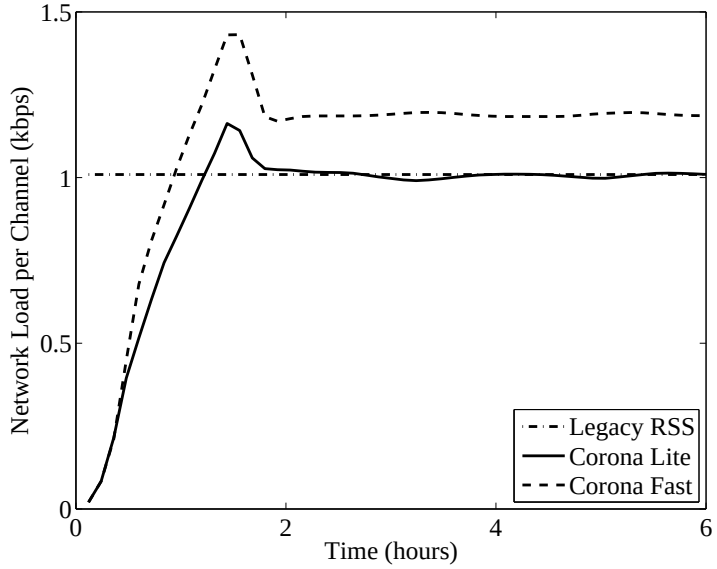

Figure 5.9: Network Load on Content Servers: Corona-Lite settles down quickly to match the network load imposed by legacy RSS clients.

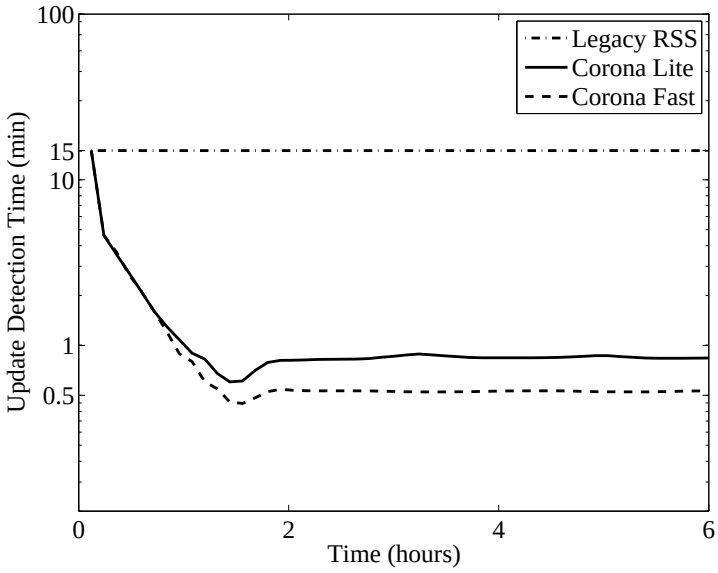

Figure 5.10: Average Update Detection Time: Corona-Lite provides 15-fold improvement in update detection time compared to legacy RSS clients for the same network load.

the inverse relation between the update detection time and the number of polling nodes. It is more beneficial to distribute the polling across many channels than devote a large percentage of the bandwidth to polling the most popular channel. Nevertheless, load distribution in Corona-Lite respects the popularity distribution of channels; popular channels are polled by more nodes than less popular channels (see Figure 5.11). The upshot is that popular channels gain an order of magnitude improvement in update performance than less popular ones (see Figure 5.12).

Corona-Fast

Unlike Corona-Lite, Corona-Fast minimizes the total load on servers while aiming to achieve a target update detection latency. Figures 5.9 and 5.10 show the network load and update performance, respectively, for Corona-Fast. Figure 5.10 confirms that Corona-Fast closely meets the desired target of 30 seconds. This improvement in update detection time entails an increase in server load over Corona-Lite. Unlike Corona-Lite, whose update performance may vary depending on the workload seen by the system, Corona-Fast provides a stable average update performance. Moreover, it enables us to set the performance depending on the requirements of the application or users and ensures that the targeted performance is achieved with minimal load on content servers.

Corona-Fair

Finally, we examine the performance of Corona-Fair, which uses update rate of channels to further fine-tune the distribution of load. It takes advantage of the fact that channels with long update intervals need not be polled as often as rapidly updated channels. Figure 5.13 shows the distribution of update detection times

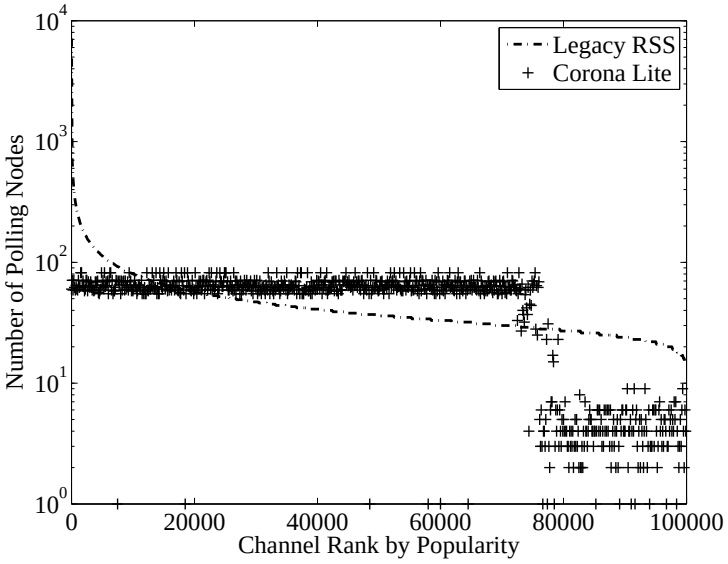

Figure 5.11: Number of Pollers per Channel: Corona trades off network load from popular channels to decrease update detection time of less popular channels and achieve a lower system-wide average.

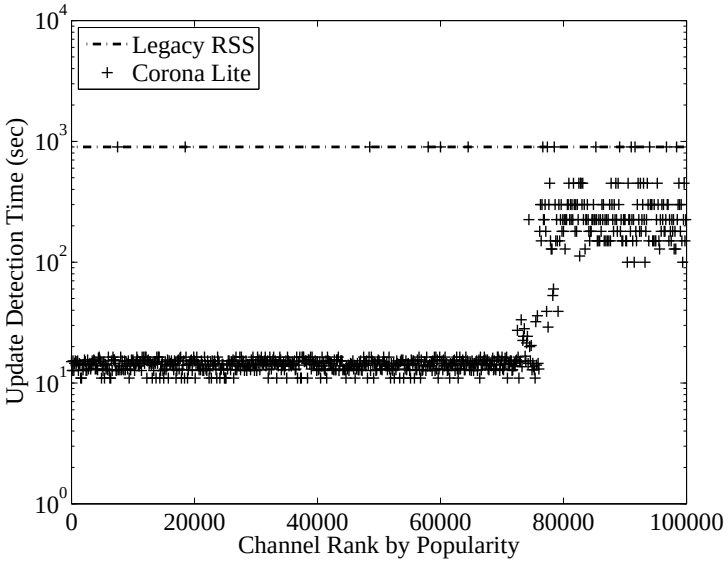

Figure 5.12: Update Detection Time per Channel: Popular channels gain greater decrease in update detection time than less popular channels.

achieved by Corona-Lite and Corona-Fair for different channels ranked by their update intervals. Channels with same update intervals are further ranked by popularity. For clarity of presentation, we plot the distribution for 200 randomly picked channels.

Figure 5.13 shows the impact of not using update interval information while assigning polling levels in Corona-Lite. Some channels with large update intervals may have short update detection times (shown in the lower right part of the graph), while some rapidly changing channels end up with long update detection times (shown in the upper left part of the graph). Corona-Fair fixes the bias by using update intervals of channels to influence polling level assignment. Figure 5.13 shows that Corona-Fair has a fairer distribution of update detection times with update intervals, that is, channels with shorter update intervals have faster update detection time and vice versa.

Corona-Fair optimizes for update performance measured as the ratio of update detection time and update interval. Thus, channels with long update intervals may also have proportionate update detection times leading to long wait times for clients. Section 2.2 proposed to correct this bias using two metrics with sub-linear growth based on the square root and logarithm of the update interval. Figure 5.14 shows that Corona-Fair-Sqrt and Corona-Fair-Log achieve update detection times that are fairer and lower than Corona-Fair. Between the two metrics, Corona-Fair-Sqrt is better than Corona-Fair-Log where a few channels with small update interval have long update detection times.

Overall, the Corona-Fair schemes provide fair distribution of polling between channels with low average update detection times and without exceeding bandwidth load on the servers. The average update detection time and load for dif-

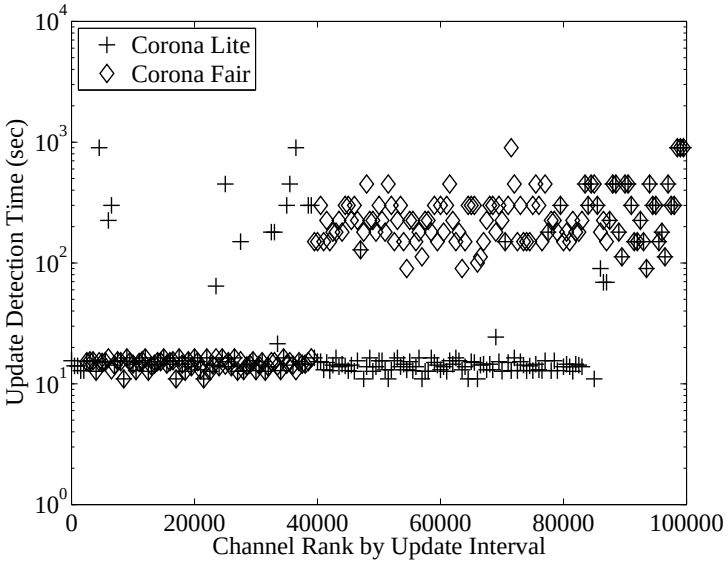

Figure 5.13: Update Detection Time per Channel: Corona-Fair provides greater decrease in update detection time for channels that change rapidly than channels that change rarely.

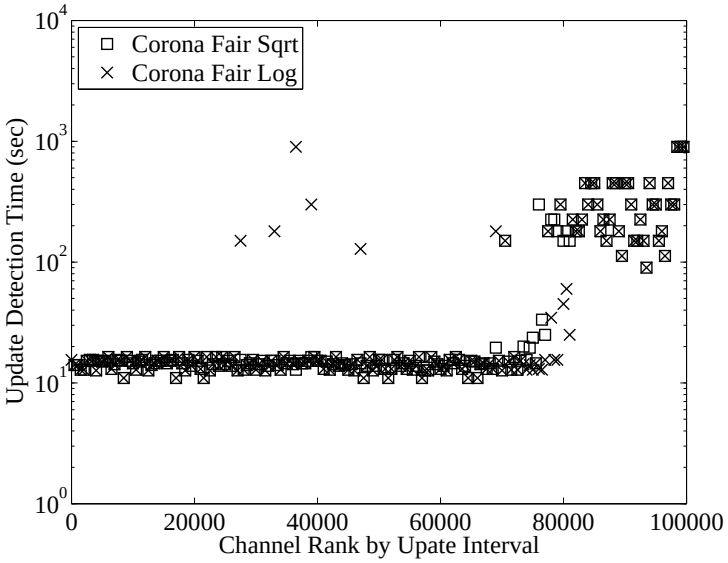

Figure 5.14: Update Detection Time per Channel: Corona-Fair-Sqrt and Corona-Fair-Log fix the bias against channels that change rarely and provide better update detection time for them than Corona-Fair.

Table 5.2: Performance Summary: This table provides a summary of average update detection time and network load for different versions of Corona. Overall, Corona provides significant improvement in update detection time compared to Legacy RSS, while consuming the same load.

Scheme	**Average Update Detection Time** (sec)	**Average Load** (polls per 30 min per channel)
Legacy-RSS	900	50.00
Corona-Lite	54	49.00
Corona-Fair	142	50.24
Corona-Fair-Sqrt	56	49.25
Corona-Fair-Log	53	49.32
Corona-Fast	32	58.92

ferent Corona-Fair schemes is shown in Table 5.2. The average update detection time suffers a little in Corona-Fair compared to Corona-Lite, but the modified Corona-Fair schemes provide an average performance close to that of Corona-Lite.

5.3.2 Corona versus Heuristics

Next, we compare the update performance of Corona with two commonly two heuristic-based schemes for assigning polling load between nodes. The first heuristics called *Proportional* allocates nodes in proportion to the channel popularity, that is, the number of clients subscribed to a channel. This heuristic represents a realistic scenario where all the users interested in a channel cooperatively detect and share updates. The second heuristic called *Square Root* is a simple modifi-

cation of the first and allocates polling load in proportion to the square root of channel popularity. We choose the square root heuristic because it is known to work better than proportional in other domains such as replication in unstructured overlays [38].

Figure 5.15 shows the average update detection time for Corona-Lite and Corona-Fast in comparison to Proportional and Square Root heuristics. First, both heuristics provide a substantial improvement in update performance compared to the naive, uncoordinated polling of legacy feed readers by more than a factor of two, with Square Root performing slightly better than Proportional. However, Corona-Lite based on a well-informed, principled approach is able to out perform the heuristics to a significant extent. In this simulation, Corona-Lite achieves an average update detection time of 53 ms compared to 357 ms for Proportional and 319 for Square Root. At the same time, the network load imposed on content servers remains the same for all the schemes compared in Figure 5.15 except Corona-Fast.

5.3.3 Deployment

We deployed Corona on a set of 60 PlanetLab nodes and measured its performance. The deployment is based on the Corona-Lite scheme, which minimizes update detection time while bounding network load. For this experiment, we use 2600 real channels providing RSS feeds obtained from *www.syndic8.com*. We issue 50,000 subscriptions for them based on a Zipf popularity distribution with exponent 0.5. Subscriptions are issued at a uniform rate during the first one hour of the experiment. The maintenance interval and the polling interval are both set to 30 minutes. We collected data for a period of eight hours.

Figure 5.16 shows the average update detection time for Corona deployment compared to legacy RSS. Corona decreases the average update time to about 38 seconds compared from 15 minutes for legacy RSS. Figure 5.17 shows the corresponding polling load imposed by Corona on content servers. Corona gradually increases the number nodes polling for objects and reaches a load limit of around 1500 polls per minute. Corona's total network load is bounded by the load imposed by legacy RSS, which averages to just under 1600 polls per minute. These graphs highlight that while imposing comparable load as legacy RSS, Corona achieves a substantial improvement in update detection time.

5.4 Summary

This chapter proposes a novel publish-subscribe architecture that is compatible with the existing pull-based architecture of the Web. Motivated by the growing demand for micronews feeds and the absence of any infrastructure to provide asynchronous notifications, we develop a unique solution that addresses the shortcomings of pull based content dissemination and delivers on the promise of a real, deployable, easy-to-use publish-subscribe system.

Corona's unique contribution is the optimal resolution of performance-overhead tradeoffs. Any pull-based content dissemination system has a fundamental tension between the amount of polling required to achieve good update performance and the corresponding network load imposed on content providers. Corona resolves this dilemma by posing the tradeoff as an optimization problem and derives the optimal tradeoff through decentralized, low-overhead mechanisms. Moreover, it provides a "knob" to control the overall performance of the system at fine granularity by setting application-specific performance targets.

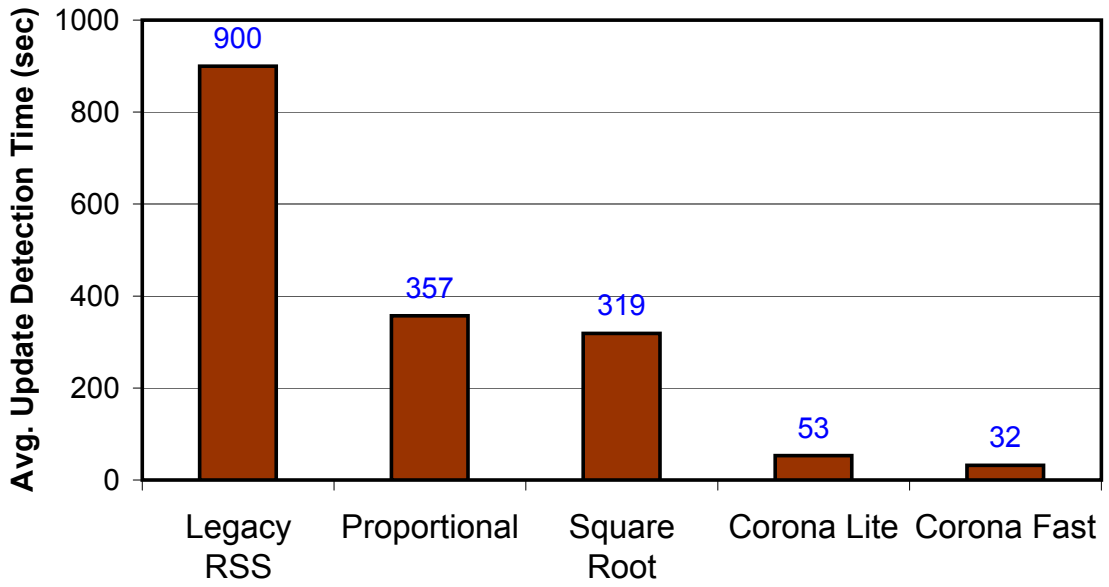

Figure 5.15: Corona vs. Heuristics: Corona performs significantly better than commonly used heuristics.

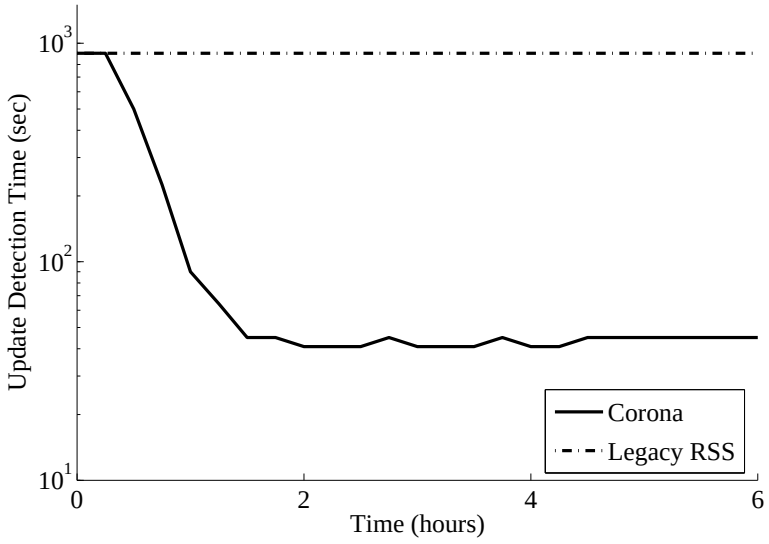

Figure 5.16: Average Update Detection Time: Corona provides an order of magnitude lower update detection time compared to legacy RSS.

Corona's principled approach achieves large gains in performance and scalability. Performance measurements based on simulations and real-life deployment show that Corona clients can achieve several orders of magnitude improvement in update latency. At the same time, Corona bounds the total network load experienced by web servers. Finally, Corona acts as a buffer between clients and servers, shielding servers from the impact of flash crowds and sticky traffic. Overall, Corona alleviates the two fundamental problems of pull-based systems, namely bad update latencies for clients and high network load on servers, with a single, unified approach.

The results from simulations and wide-area experiments confirm that Corona achieves a balance between update latency and network load. It dynamically learns the parameters of the system such as number of nodes, number of subscriptions, and channel-characteristics, and uses the new parameters to periodically adjust the optimal polling levels of channels and meets performance and load targets. Corona offers considerable flexibility in the kind of performance goals it can achieve. We showed three specific schemes targeting update detection time, network load, and fair distribution of load under different metrics of fairness. Measurements from the deployment showed that achieving globally optimal performance in a distributed wide-area system is practical and efficient. Overall, Corona proves to be a high performance, scalable publish-subscribe system.

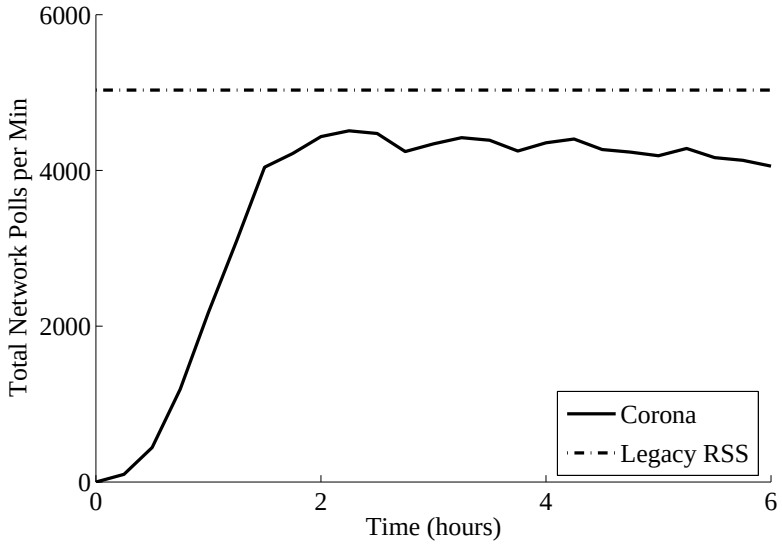

Figure 5.17: Total Polling Load on Servers: The total load generated by Corona is well below the load generated by clients using legacy RSS

Chapter 6

Related Work

This chapter provides an overview of current research and development in areas related to this thesis. In particular, it gives a summary of prior research on resource allocation problems, describes recent advances in distributed overlay systems, and summarizes work related to the three decentralized services explored in this thesis, namely naming systems, content distribution networks, and publish-subscribe systems.

6.1 Resource Allocation Problems

Resource allocation in distributed systems is a classical problem, which has drawn the attention of researchers for decades. The problem of deciding which node hosts which object or file has been classically called the File Assignment Problem (FAP). Dowdy and Foster [48] and Wah [145] provide a comprehensive survey of analytical models developed for posing the file assignment problem in a distributed network. The models they discuss include the kind of analytical models posed in this thesis to meet performance goals given resource constraints. More recently, Li et al. [88], Jamin et al. [75], and Qiu et al. [116] pose similar file assignment problems in the context of content distribution networks. While, Cho et al. [35] and Pandey et al. [108, 107] explore resource allocation problems in the context of monitoring online data through polling.

Several solution techniques that work well for small sized systems have also been proposed in the past. Solution techniques applied to these problems include techniques such as dynamic programming [88] and branch and bound [145], which

are efficient for small problems but have worst-case exponential time complexity, as well as, approximation algorithms to solve the allocation problem expressed as K-center problem [75], facility location problem [116], and constrained optimization problem [35]. However, the above techniques depend on a single, centralized server to perform the computations. Kurose and Simha [87] proposed a fully decentralized technique that solves resource allocation problems, but requires all-to-all communication between the nodes.

In addition to resource management on distributed systems, a vast amount of literature also exists on resource management in a single system, often described as *scheduling*. Prior research in scheduling is not discussed in this chapter; a good survey of different techniques for scheduling resource usage on a single node can be found in Waldspurger's Ph.D. thesis [146].

In contrast to the extensive prior work in resource allocation, the techniques presented in this thesis enable resource allocation for large-scale, distributed systems, which are crucial for hosting services in the current Internet.

6.2 Peer-to-Peer Overlays

Peer-to-peer overlay systems have emerged in recent times as a powerful alternative to the traditional client-server architecture. Instead of depending on centrally administered server nodes, a peer-to-peer overlay consists of commodity, privately owned computers that form a communication network between themselves. Such self-organizing, peer-to-peer overlays became widely-used due to the popularity of content sharing applications, such as Freenet [36], Gnutella [169], and Kazaa [171], which enable users to search and download content that is not publicly hosted on the Web.

The success of file-sharing systems showed that peer-to-peer systems can achieve high scalability and failure resilience and provided the impetus for their extensive study in the research community. This section provides an overview of current research in the field of peer-to-peer systems.

6.2.1 Unstructured Overlays

Peer-to-peer overlays can be characterized as *unstructured* or *structured* depending on how they organize themselves into an overlay network. Unstructured overlays resemble an irregular graph where any node may pick any other peer node as neighbor. Early peer-to-peer file sharing systems, such as Freenet and Gnutella are examples of unstructured overlays. Searching in unstructured overlays is performed through controlled flooding, where a query originating at a node may eventually spread to all the nodes in the system. Such floods resemble typical graph-traversal algorithms, such as depth-first and breadth-first traversal. Freenet employs a depth-first search algorithm, whereas Gnutella employs a breadth-first algorithm.

While the unstructured overlays could easily scale to millions of peer nodes and handle the high rate of churn caused by nodes leaving and joining the system frequently they do not provide good lookup performance. An average query visits a large fraction of nodes in the system and consumes a large amount of time to fetch results. In the worst case, a query spreads to every live node and may still not invoke any response from a single node in the system.

Several techniques have been proposed to improve the performance of unstructured overlays. Lv et al. [95] propose to replace flooding-based search with random walks. Their technique substantially decreases the network overhead for searching,

but does not decrease lookup performance. Chawathe et al. [34] combine random walks with topology adaptation based on lifetime and bandwidth availability of nodes to improve lookup and download performance. These improvements, however, do not make a qualitative difference to the lookup performance of unstructured overlays.

6.2.2 Structured Overlays

Structured overlays provide bounded lookup performance through clever organization of the overlay network into a well-defined, regular topology. The regular topology ensures that the maximum distance between any two nodes in the system, the *diameter* of the topology, can be analytically bounded. Structured overlays can provide superior lookup performance by providing worst-case and average-case bounds on lookup latency that are significantly better than unstructured overlays.

To achieve better lookup performance, structured overlays restrict the type of lookups. Unlike unstructured overlays, which support search based on arbitrary keywords, structured overlays typically provide lookups only on globally-defined unique keys. Such unique keys are usually derived by hashing the content or a well-defined attribute of the content, such as its universal name (URN), into an integral key space [81]. The overlay system maps each key to a specific node or set of nodes and routes queries to the node or nodes responsible for the queried key.

A large number of structured overlays, which differ in their topology and lookup algorithm, have been proposed so far. The choice of topology and lookup algorithm determines the worst-case lookup performance of the system as well as the number of neighbors monitored by each node.

CAN [121] organizes the network into a hyper-dimensional torus of constant

dimension and routes lookups from one dimension to the next until the mapping node for the key is reached. The CAN network has a diameter of $O(dN^{\frac{1}{d}})$ and node degree of $O(d)$ for a network of N nodes and dimension d. It is possible to configure the CAN network with a dimension $d = \log N$ and obtain a worst-case lookup performance of constant $O(\log N)$ hops, if a reasonable bound on the size of the network can be estimated in advance.

Pastry [128], Tapestry [161], and Kademlia [97] organize the topology based on digits of integral identifiers assigned to each node. In these overlays, each node has neighbors that differ in exactly one prefix digit in the identifiers. They support keys drawn from the same space as node identifiers and map the key to the "nearest" node in the identifier space. While Pastry and Tapestry use modular difference between two identifiers to measure nearness, Kademlia uses the XOR operation to determine the distance between two identifiers. All three overlays employ a routing algorithm proposed by Plaxton et al., which routes queries iteratively from one node to another with increasing number of prefix digits between node identifier and the queried key [114]. Overall, these overlays have a diameter of $O(\log_b N)$ and node degree $O(b \log_b N)$, for a network of N nodes and a base b for representing identifier digits.

Chord [138] uses the same network organization as the above prefix-matching overlays, but use a slightly different algorithm for routing queries. Chord nodes maintain neighbor links with nodes at distances in geometric sequence. That is, each node has a link to the closest nodes at distances between 2^{i-1} and 2^i. By iteratively routing queries to the farthest neighbor with identifer less than the key, Chord achieves a worst case lookup performance of of $O(\log N)$ with node degree $O(\log N)$, for a network of N nodes.

A few other overlay systems make use of well-known data structures to organize network topology. Viceroy [96] organizes the network into a butterfly data structure, commonly used in numerical algorithms for computing Fast Fourier Transforms (FFT). The advantage of the butterfly structure is that while it has the same $O(\log N)$ diameter as the previously described overlays, it requires only a constant seven neighbors per node. However, the butterfly structure is organized into hierarchical layers, where nodes in the top layers of the hierarchy serve as intermediaries to a significantly greater number of queries than the nodes in the lower layers.

SkipNet [71] provides the same properties of $O(\log N)$ diameter and constant node degree as Viceroy while also ensuring uniformly balanced routing load for each node. SkipNet achieves these properties by using a probabilistic data structure called Skip List to organize the network. In addition, SkipNet also facilitates *range queries*, that is, queries for a range of keys rather than a single key.

The varying topological properties of the previously described structured overlays indicate a fundamental trade off between the diameter and the node degree; fewer neighbors per node implies longer diameter, and vice versa. Koorde [80] explores this tradeoff using a data structure called de Bruijn graphs. It shows that the best worst-case bound on lookup performance for a $O(\log N)$ node-degree network is $O(\frac{\log N}{\log \log N})$ and for a network with constant node degree d is $O(\log_d N)$. Independently, Naor and Wieder [151] provide an alternative method to construct structured overlays with the same tradeoff bounds as Koorde.

6.2.3 Techniques to Improve Lookup Performance

While structured overlays provide analytical bounds on the worst case lookup performance, the actual lookup latencies are still large; a typical query makes several hops and each hop may span the Internet, incurring a large delay overall.

Several techniques have been proposed to improve the lookup performance of a structured overlay. Typically, these techniques are of three types: a) altering the structure of the overlay to reduce the number of hops to a small constant, b) exploiting physical latencies between peer nodes when constructing overlays to reduce lookup times, and c) caching or replication of objects to terminate queries earlier in the lookup path.

Constant Diameter Topologies

Douceur et al. [46] propose a structured overlay called SALAD that has a constant diameter d, which can be configured to a suitable integral value. SALAD achieves d hop lookup performance using $\mathrm{O}(dN^{\frac{1}{d}})$ neighbors per node.

Mizrak et al. [100] propose a two-level hierarchical architecture to achieve a three-hop overlay. The upper level consists of specially designated nodes called *super peers*, which partition a circular identifier space between themselves in a manner similar to Chord, but are fully connected. The lower level nodes are connected to a super peer. A query is answered in three hops consisting of the super peer of the originating node, the super peer of the destination node, and the destination node.

Gupta et al. [66, 67] propose a one-hop overlay by using a fully connected graph. They argue that maintaining $\mathrm{O}(N)$ neighborhood at each node is feasible even for moderately large networks of thousands of nodes.

Kelips [68] provides a more scalable approach for achieving one-hop lookup performance with only $O(\sqrt{N})$ node degree. Kelips divides the nodes into $O(\sqrt{N})$ groups of $O(\sqrt{N})$ nodes on average. Each node is a neighbor of all the nodes in its group and at least one node in every other group. An object is mapped to a group and replicated on all nodes in its group. A query is resolved within one hop by simply forwarding it to any node in the group responsible for the queried object.

Proximity-based Neighbor Selection

While the preceding approaches propose alternative topologies for organizing an overlay efficiently, other approaches exploit the variance in network latencies between nodes to improve lookup performance without altering the underlying topology. These latency-based techniques typically operate by picking the closest eligible node to perform the role of an overlay neighbor.

Castro et al. [30] explore proximity based neighbor selection in Pastry. They show that by picking the closest node among all nodes with same prefix digits as neighbors, Pastry can effectively achieve O(1) lookup performance. Their experiments show an expected lookup performance equivalent to 1.4 average Internet hops. Dabek et al. [43] explore the same technique in the context of Chord. Zhang et al. [64] propose techniques to adaptively chose proximal neighbors by piggybacking information on lookup messages.

Coral [59] takes advantage of the variance in latencies between Internet hosts by grouping nodes with similar latencies into clusters. Each node belongs to clusters of different latency ranges and each cluster is organized as a separate structured overlay. Coral achieves improved lookup latency by always looking up a key in a lower latency cluster before a higher latency cluster.

6.2.4 Caching and Replication in Overlays

The third technique commonly used to improve lookup performance in overlays is caching or replication. Caching in overlays is typically used in the passive, opportunistic form described earlier in this thesis. That is, search results are cached at the intermediate nodes traversed by the query and timeouts are used to expunge entries from caches. This form of passive caching is used in CFS [42] and PAST [129], two file systems layered on top of Chord and Pastry respectively.

OpenDHT [122] employs a limited amount replication to achieve low lookup latency. OpenDHT is a publicly deployed structured overlay similar to Pastry. It reduces lookup latency by replicating each object at a fixed number of geographically distributed nodes. Sending parallel lookups to the replica holding nodes and using the fastest response ensures that the overall lookup performance is high.

In addition to improve lookup performance, limited replication is also commonly used to provide persistence of storage on overlays. Here, objects are always replicated on a certain fixed k number of nodes, so that object would continue to reside in the system despite node failures. There is a cost-performance tradeoff governing the choice of k; a large value of k increases the number of failures the system can tolerate , but also incurs overhead to replicate the object k times and preserve consistency for mutable objects. Current systems do not explore this tradeoff, but pick an arbitrary value for k. Both CFS and PAST use this form of replication for data persistence.

6.2.5 Overlays in Practice

This section describes the mechanisms proposed to handle issues that arise when overlays are deployed in practice. In particular, it describes techniques to handle

churn, improve security, and overcome lack of transitivity in the network.

Churn

Churn refers to the frequent joining and leaving of nodes in a peer-to-peer to system. Rhea et al. [123] explore a combination of proactive and reactive failure recovery to handle high churn in an overlay system. Kelips employs replication and gossip-based epidemic failure management to handle churn [68]; it maintains $O(\sqrt{N})$ copies of each object to ensure that a query can always be answered with a high probability. Replication happens in the background through periodic gossip.

Security

Security is an important concern for overlay networks as they are typically deployed in a distributed environment, where nodes are not administered centrally. Hence, it is essential to ensure that the effect of infiltration and compromise of hosts by malicious agents is limited and does not affect the integrity of the system as a whole. A malicious agent should not be able to insert arbitrarily large number of hosts into the system and malicious or compromised nodes should not be able to induce other benign nodes to perform bad things.

Castro et al. [29] propose techniques to limit the impact of compromised and malicious nodes in an overlay. They use statistical bounds of node density when node identifiers are assigned randomly to restrict the number of identities a malicious agent can create. In their scheme, an overlay neighbors are picked from a narrow portion of the identifier space so that malicious agents only have an impact proportional to the amount of identifier space they control. Other researchers have proposed mechanisms to limit the amount of infiltration into the overlays by using

out-of-band mechanisms to authenticate the identity of the hosts [129, 42].

Non Transitivity

In addition to churn and security, several other engineering issues arise while overlays are deployed in practice. One such issue is the absence of transitive connectivity in network connections in the Internet. While structured overlays assume that a node A can communicate to C if A can communicate to B and B to C, such transitivity does not always exist in the Internet. Freedman et al. [58] analyze the problems cause by transitivity violations and propose several techniques to mitigate the problems.

6.2.6 Applications of Structured Overlays

Several applications have been layered on top of structured overlays taking advantage of its high failure resilience and scalability. Most applications of structured overlays are based on the key-based lookup primitive. These applications include file systems, cooperative web caches, name services, and other ad hoc lookup services.

CFS and PAST, mentioned earlier, are two overlay based file systems proposed for archival storage of data. Ivy [104], also layered on top of Chord like CFS, provides a full-fledged read-write interface that PAST and CFS do not provide. Farsite [3] is a serverless, peer-to-peer filesystem designed to reclaim extra disk space available on hosts within an institution. It uses the SALAD overlay for managing file location information and meta data.

Structured overlays have also been used to build cooperative web caches, which enable users to improve web performance by taking advantage of common interests

of other users within the institutions. Structured overlays server the purpose of locating web objects among other participating hosts in the institution before the object is fetched from the web server. Examples of cooperative web caches using structured overlays include Squirrel [74], Kache [91], and CoralCDN [59]. These cooperative web caches are layered on Pastry, Kelips, and Coral respectively.

Structured overlays also serve as building blocks for lookup services. New name services using overlays have been proposed to replace the DNS. DDNS [39] and Overlook [143] are two name services built using Chord and Pastry respectively. OverCite [139] is a system to lookup bibliography information for academic publications. UsenetDHT [135] serves as a storage system for articles posted on Usenet, a widely-used network newsgroup.

The second class of applications on structured overlays take advantage of its topology for multicast, that is, simultaneous data dissemination to multiple hosts. Scribe [130] is application-level multicast protocol that provides data dissemination using the Pastry overlay. SplitStream [31] is a system for disseminating large content, such as multi media and software binaries, using Scribe as an underlying framework. POST [99] is a system for email layered on Pastry. And, FeedTree uses Scribe multicast to disseminate update for web micronews.

Surprisingly, peer-to-peer file sharing applications, which motivated Freenet and Gnutella, have not been the driving applications for structured overlays. The key reason for this is the difficulty of performing key-word searches on structured overlays. However, Castro et al. [28] show that flooding based key-word searches can be efficiently designed on structured overlays with bounded worst case lookup performance. Overnet [173], based on Kademlia, is the only widely-used peer-to-peer file sharing system layered on a structured overlay. Its algorithm for perform-

ing keyword based search is, however, not public.

The rest of this chapter describes work related to the three applications discussed in this thesis.

6.3 Domain Name System

The Domain Name System (DNS) was designed in the early eighties to provide the fundamental service of translating host names to IP addresses. Prior to the deployment of DNS in 1982, the name-address mappings were stored and transferred around in a single file, which had limited scalability as the number of end hosts grew. DNS enables name-address translation to scale to the millions of hosts that comprise the Internet today and serves different types of data including IP addresses, names of mail servers, public keys, etc. The DNS standard and protocols is elaborately detailed in Internet RFCs [101, 102, 51], while a good reference for operational issues is the book [4]. DNS hierarchically partitions name spaces into domains and uses delegations to transfer control from a domain to its sub domains. This basic design of DNS has not changed since its initial deployment in 1982.

6.3.1 DNS Performance Studies

In 1988, the designers of DNS, Mockapetris and Dunlap, published a retrospective analysis of the successful features and shortcomings of DNS [103]. Their study identified the decentralized and hierarchical namespace as the biggest success and the delegation of authority required to serve a namespace, that is, to provide DNS mappings, as the biggest shortcoming.

The first major measurement study of DNS was performed by Danzig et al.,

who studied the characteristics of DNS traffic [44]. Their study identified several software errors in nameserver implementations that trigger a large amount of DNS traffic. Several measurement studies since then have provided good insight into the advantages and the drawbacks of the system.

Lookup Performance: Jung et al. performed a large scale survey of DNS performance by tracing active workload of users at MIT [79, 78]. Their study analyzes the client-perceived lookup performance of DNS, characteristics of DNS query workload, and the effectiveness of caching in DNS. It shows that DNS queries follow a heavy-tailed Zipf distribution. Bent and Voelker [13], Huitema et al. [73], and Wills et al. [153] focus on the impact of DNS latencies to web download times. All three studies show that DNS lookup latency forms a significant portion of web latency.

Failure Resilience: Pappas et al. study the impact of broken delegations, cyclic dependencies, and nameserver redundancies on the availability of DNS [110]. Their study shows that DNS has surprisingly low tolerance to failures, predominantly less than 2 node failures, and that broken delegations contribute substantially to its lookup latency. Park et al. [111] study the robustness of DNS implementations and show that poor implementations of DNS resolvers contribute substantially to unresolved DNS queries and long lookup latencies. Finally, several studies on the DNS traffic seen by root servers show that DNS root servers are frequently subject to denial of service attacks [21, 23, 22].

DNS-based Redirection: A few studies have focused on the impact of DNS-based network control, that is, the practice of dynamically redirecting web requests to lightly-loaded or proximal servers during IP address resolution. Shaikh et al. [134], Johnson et al. [76], Krishnamurthy et al. [83], and Pang et al. [109]

show that DNS-based server selection is highly coarse-grained and the chosen server may deviate considerably from the best server, in terms of both network distance and server load. Moreover, Shaikh et al. show that server selection increases DNS lookup latency considerably because dynamically generated DNS mappings have very small timeout values.

Trust in DNS: Our measurement study on DNS focuses on an important and previously unexplored aspect of DNS security; namely the impact of delegating trust to other administrative domains for serving DNS mappings. Our study, described in Chapter 3 and [120], shows that a typical DNS name depends on a large number of servers for its resolution and the high prevalence of security vulnerabilities in DNS software amplifies the security risks.

6.3.2 DNS Security

Protecting the integrity of DNS has always been a major concern that led to the initial proposal of DNSSEC as a standardization effort [50]. DNSSEC associates each mapping with a chain of cryptographic certificates that start from the owner domain and end at the root server. DNSSEC has received very little acceptance to date primarily due to three drawbacks: first, the DNSSEC standard does not clearly demarcate ownership of certificates across domain boundaries, second, authenticated denial of existence of DNS domains or records for a name was not described clearly, and finally, depends on a centralized public key infrastructure that prevents domains to secure themselves independent of their parents.

A new standard for DNSSEC has been recently proposed that provides clean security for delegation of authorities and denial of existence [8]. However, the third problem related to incremental deployment continues to hinder the wide adoption

of DNSSEC.

6.3.3 Design Proposals

Few design alternatives for DNS have been proposed since its initial deployment. Here we give a brief overview of the proposals.

Initial proposals involved minor changes to some aspects of DNS operation. Cohen and Kaplan [37] propose a proactive caching scheme for DNS records. In their scheme, expired DNS records in the cache are proactively fetched from the authoritative nameservers. They analyze several heuristics-based prefetching algorithms, and evaluate their performance. This scheme entails prefetching at intermediate caches, which generates substantial amount of background traffic to update DNS records.

CoDNS proposes to mask delays in the legacy DNS caused by failures in local resolvers by diverting queries to healthy resolvers in other nearby administrative domains [111]. However, it does not consider the security risk involved in trusting servers outside the administrative domain. Overall, CoDNS provides resilience against temporary problems with the legacy DNS, but is not intended as a replacement.

Several recently proposed DNS architectures leverage the failure resilience and scalability properties of structured overlays. DDNS [39] and Overlook [143] are name service architectures layered on Chord and Pastry respectively. However, both Chord and Pastry are overlays with $O(\log N)$ diameter and provide poor lookup performance, with latencies much longer than what the DNS achieves currently.

Handley and Greenhalgh [69] have recently proposed to actively push critical

DNS mappings such as the NS records that represent nameserver delegations and the glue records that carry addresses of nameservers, to all the DNS servers in the world. However, they only show that pushing the critical records to all servers is feasible in terms of bandwidth consumption and do not propose any mechanisms to actually perform and manage the replication.

Finally, Balakrishnan et al. propose to replace the hierarchical DNS and URL namespace with flat global identifiers and introduce additional layers of translation to facilitate a namespace that does not tie resource names to host names [147, 10]. Currently, a URL for a resource includes the name of the host serving the resource. While their proposal alters the namespaces, it is complementary and would benfit from CoDoNS, the high performance resource location service proposed in this thesis.

CoDoNS proposed is intended a safety net and a possible replacement for DNS. By separating namespace management from name lookups, it provides an efficient platform for name resolution, irrespective of the namespace. It combines proactive, push-based caching and structured overlays to provide low lookup latency, high failure resilience, and scalability. Optimal resolution of performance-overhead tradeoffs ensures that CoDoNS makes the best use of limited resources such as network bandwidth, while providing significantly improved lookup performance over legacy DNS.

6.4 Web Caching and CDNs

The Web has always been an important topic of research and several methods to improve the Web have been proposed. The primary technique employed to improve Web performance is caching, both passive as well as active. Rabinovich

and Spatscheck [154] provide a detailed survey of web caching and replication. This section provides an overview of the most representative performance studies and caching techniques to improve Web performance.

6.4.1 Web Performance Studies

Measurement studies of the Web have focused on understanding its performance as well as workload characteristics. Some well-known traces of live web activity include the DEC traces [84] collected at Digital Equipment Corporation, the UCB traces [65] collected at the University of Berkeley, the UPisa traces [125] collected at Universita di Pisa in Italy, and the NLANR traces [172] collected by requests seen on Squid caches. Bent and Voelker[13] study the lookup performance of web page downloads and the impact of several browser options such as the use of persistent connections, number of parallel connections, etc.

Other studies have focused on characteristics of Web workload, such as the object popularity, size, and update rate. Analyzing the above traces, Breslao et al. [18] show that Web workload is heavy-tailed and follows the Zipf distribution. Almeida et al. [5] study the locality of reference, both spatial and temporal, in web workloads. Crovella et al. [41] study the correlations between document size and popularity. Douglis et al. [47] focus their study on the update rate of web objects and show that web objects have a wide range of update rates. Barford et al. [11] compare the popularity and size of content in traces collected three years apart and report that the characteristics are qualitatively similar. Several other studies by Abdulla [1], Arlitt and Williamson [9], Bray [17], Glassman [63], as well as Gribble and Brewer [65] have reported on characteristics of the Web workload based on user activity traces.

6.4.2 Caching Algorithms

Initial research on web caching was largely focused on heuristics for managing the local cache on a web browser, that is, for deciding which objects in the cache should be removed when the cache is full. Two popular choices for cache replacement has been the least recently used (LRU) and the least frequently used (LFU) algorithms, which remove the oldest cache entry and the least popular cache entry respectively. However, both these strategies do not recognize the high variance in the characteristics of web objects such as size, update rate, latency to the server, etc.

By far, size of the web object is the most commonly used characteristic to influence cache replacement algorithms. The Largest Size First heuristic proposes to remove the biggest object from a full cache [152]. Another heuristic uses the logarithm of the object size and removes the object with the largest $\log(size)$ while breaking ties using LRU [2]. The Lowest Relative Value (LRV) uses the ratio of size and popularity and removes the object with the lowest ratio [124].

A few algorithms use the latency for downloading the object during a cache miss to influence cache replacement. The Lowest Latency First heuristic evicts the object with smallest download latency so that the latency impact of a cache miss can be reduced [159]. The Greedy Dual Size (GD-Size) algorithm combines both the cost for a cache miss as well as the size of the object to decide cache replacement. It picks the object with the smallest ratio of the downloading latency and the object size [25].

Cao and Irani [25] study the effectiveness of the cache replacement heuristics described above through simulations. Their study shows that the performance of the different heuristics are comparable and no single heuristic is well-suited for

all scenarios. The overall cache hit rate varies of the heuristics between 20% to 40%. Brelau et al. [18] show that the low cache hit rate is primarily due to the heavy-tailed nature of popularity distribution. Their study on the performance of LRU, LFU, and GD-Size on different real-life workload traces agrees with the findings of Cao and Irani.

In addition to cache replacement algorithms, research has also focused on handling stale objects; that is, objects whose cache lifetime has expired. Several researchers have studied the usefulness of proactively pre-fetching expired objects from the web servers instead of expunging them from the cache. Padmanabhan et al. [106], Kroeger et al. [85] and Fan et al [53] show that pre-fetching can provide substantial reduction in web latency, especially for clients with low-bandwidth Internet access such as dial-up modems. Their studies also observe that while pre-fetching can provide substantial improvement in the performance of web caches, there is a corresponding increase in the bandwidth required to achieve the improvement. They propose different heuristics to decide which object should be pre-fetched taking into account the cost-performance tradeoffs.

A few web caching strategies use sophisticated techniques to predict access patterns, that is, temporal and spatial correlations between object access, in order to enhance cache performance. Bestavros [14], Padmanabhan and Mogul [106], and Zukerman et al. [164] have proposed modeling techniques based on Markov chains to deduce probability of object access.

Several studies have analyzed the performance limits of caching. The fundamental limitation stems from the fact that web queries may contain unforgeable entities such as cookies, may be meant for dynamically generated data, or real-time data for which caching does not make sense. Studies by Feldmann et al. [54] and

Wolman et al. [156] show that up to 40% of web queries may not be cacheable for the above reasons. The number of queries to dynamically generated objects appears to be growing from 1% in the mid nineties [152] to about 10-20% in the late nineties [54, 156]. The study by Feldmann et al. [54] also shows that about 20-30% of web queries carry user-specific cookies in them.

6.4.3 Cooperative Caching

While the previously described studies focus on caching at a single client, several techniques have been proposed to pool caches together and cooperatively share content across the caches. The key intuition behind this approach is that the workload generated by a large number of clients can be exploited for redundancy as popular objects fetched by a few clients may be of interest to several others as well. Research in this form of caching called *cooperative caching* has primarily focused on how to organize the network of caches and how to exchange cache content between caches.

Several researchers have analyzed traces of Web activity and shown that cooperative caching can provide substantial improvement over the performance of independent caches. Duska et al. [49] show that cooperative caching can achieve 85% cache hit rate. Wolman et al. [154, 156] show that while cache hit rate increases with population of users using the cache, the benefits diminish as the population base grows very large. Gribble and Brewer [65] deduce based on client traces that cache hit rate increases logarithmically with the size of client population.

Cooperative caches can be classified into hierarchical and peer-to-peer based on their organization. Harvest is one of the earliest proposed hierarchical cooperative cache [33]. It organizes the caches into a hierarchy based on the location and

scale of the cache. For example, a browser cache is lower than an institution-level cache, which is lower than a regional or national cache. The Harvest architecture is employed by Squid, which is a globally deployed and widely used cooperative cache [150].

In the hierarchical caching system, a cache miss at a lower-level cache is forwarded to its parent at the next higher level. Therefore, hierarchical caches incur substantial latency cost during cache misses. Peer-to-peer caches eliminate this problem by organizing the system into a flat structure with no hierarchy. Such peer-to-peer caches typically form an overlay network of caches were each cache has a certain number of neighbor caches with which it exchanges information about cache content.

Summary Cache proposed by Fan et al. [52] proposes a cooperative cache architecture where each cache knows about every other cache and periodically exchange their cache content. It uses Bloom filters, a compression scheme for representing a set of objects, to achieve cache information sharing with less network bandwidth. Since a web object may reside on any web cache, all-to-all communication is required for caches to locate another cache with content if there is a local miss. Tewari et al. [141] propose a similar flat organization for caches, but retain the hierarchy for communicating cache content. In their system exchange of cache summaries is restricted to the parent and siblings. Yet, both schemes require each cache to store complete cache summaries of all the caches in the system.

A few researchers avoid the need for all-to-all exchange of cache information by allocating each web object to one or more well-defined caches for managing that web object. Ross et al. [126] propose a scheme for distributing responsibility for a web object among a non-changing, static set of nodes through hashing. Thaler

and Ravishankar [142] explore the issue of adding and removing web servers efficiently in a hash-based cooperative cache. Karger et al. [81] propose a technique called consistent hashing that can allocate objects to nodes without requiring prior information about the nodes and can accommodate frequent changes in the set of nodes.

A few cooperative caches take advantage of structured overlays to provide an efficient and scalable architecture for cooperative caching. Squirrel [74] and Kache [91] are overlay-based cooperative web caches. Squirrel uses consistent hashing to map objects to nodes and the Pastry overlay to locate cached objects. Kache is layered on top of Kelips and takes advantage of the replication in Kelips to provide low latency and high failure resilience.

In addition to the purely hierarchical and purely flat cache organizations, several web caching systems combine hierarchy as well as peer-to-peer communication to maximize benefits. The Internet Cache Protocol (ICP) is a modification of the Harvest hierarchy that enables caches to exchange content with other caches at the same level [149]. Michael et al. [98] propose an adaptive cache where caches are organized into a hierarchy of clusters. Each cluster is is a self-organized peer-to-peer network of caches that are proximal to each other. Caches within a cluster exchange cache content with each other. Gadde et al. [60] develop a cooperative web cache that is aided by a centralized directory service providing meta data about locations of objects.

The previously described cooperative caching schemes mandate a specific amount of network communication on each node. Roussoulos and Baker [127] have recently proposed an incentive-based mechanism for nodes to decide when to propagate updates about cache summaries. The decisions made by a node to participate in the

propagation of cache locations for a particular object depends on the local popularity of that object and the rate of change of caching information for that object.

6.4.4 Content Distribution Networks

The caching schemes described earlier all rely on passive caching, where caches are filled in response to queries generated by clients. The alternative approach is to proactively push copies of web objects to selected sets of caches or content servers so that clients looking for those objects can download them from a close-by node. This push-based approach to caching has been popularized by the success of commercial content distribution networks such as Akamai and Digital Island.

Push-based caching has also been extensively studied in the research community. Bestavros [14] proposes proactive cooperative web caches where objects are pushed to different caches based on their demand. CoDeen is a widely used content distribution network developed by Park et al. [148, 112]. While the exact algorithm used by CoDeen for distributing objects is not publicized, it employs a combination of heuristics to place objects based on the demand, latency, and load level of each node.

CoralCDN [57] and Backslash [137] are CDNs built using structured overlays. CoralCDN uses the Coral structured overlay to cache objects. CoralCDN provides quick responses using the latency-based overlay architecture of Coral and withstands flash crowds due to rapid, dynamic increase in the number of nodes caching an object as the popularity of the object increases. Backslash also focuses on handling flash crowds efficiently by creating additional copies of objects in the system. It uses deterministic, heuristics to decide when to create more copies for an object.

Overall, previous cooperative caching schemes and CDNs rely on ad-hoc heuristics to decide which node should cache which objects. In contrast, the key contribution of this thesis is to show that a far more efficient way of making the above key decision is through mathematical optimization. CobWeb, the CDN we built based on this principled approach, there by, achieves superior performance in terms of latency while making the best use of available storage and bandwidth resources.

6.5 Publish-Subscribe Systems

The publish-subscribe (pub-sub) paradigm consists of three components: *publishers*, who generate and feed the content into the system, *subscribers*, who specify content of their interest, and an infrastructure for matching subscriber interests with published content and delivering matched content to the subscribers. The field of publish-subscribe had its origins in early systems for group communication. The first publish-subscribe systems were based on the Isis group communication tool kit [62].

Publish-subscribe systems can be classified as *topic-based* or *content-based* depending on the expressiveness of subscriber interests. In topic-based systems, publishers and subscribers are connected together by pre-defined topics, called *channels*; content is published on well-advertised channels to which users subscribe and from which they receive asynchronous updates. Content-based systems enable subscribers to express elaborate queries and use sophisticated content filtering techniques to match subscriber interests with published content. This section provides a background on content distribution based on the publish-subscribe and summarizes the current state of the art.

6.5.1 Topic-based Publish Subscribe

Topic-based publish-subscribe systems provide event or update notification service on well-defined topics that subscribers can register for. Isis [62], TIBCO [174], FeedTree [132] and Herald [24] are a few well-known topic-based pub-sub systems. The topics represent well-defined entities such as multicast groups as Isis [62], or web URLs as in FeedTree [132], or keywords as in TIBCO [174].

A few topic-based publish-subscribe systems use type systems to identify topics. In these systems each topic is represented as a typed tuple and all events that match the same type of tuple as associated to the same topic. Tuple space based systems include Linda [26], T-spaces [175], and Java Spaces [170].

Much of the research on topic-based publish-subscribe systems have focused on building efficient infrastructures for event dissemination. Group communication or multi-cast protocols, which simultaneously transport a network message to multiple participants is a key component of several topic-based publish-subscribe systems. The group communication protocols used in publish-subscribe include virtual synchrony in Isis, reliable multicast protocols such as Scalable Reliable Multicast [56] (SRM), Reliable Multicast Transport Protocol [89] (RMTP), and bimodal multicast [16], and application-level multicast protocols such as Scribe [130] in FeedTree.

An alternative technique used for event notification involves allocating the notification load for each topic to nodes called *rendezvous points* in the distributed system. In these systems, each topic has one or more rendezvous points which manage subscription state and forward events and updates to the subscribers. TIBCO was the first system to use the rendezvous approach for event notification. A recent system, Herald [24], also proposes to use distributed rendezvous points

for event notification.

6.5.2 Content-based Publish Subscribe

Content-based publish-subscribe systems, unlike topic-based ones, provide a much richer interface for users to select content that matches their interest. Consequently, research in content-based publish-subscribe systems has focused primarily on languages for expressing complex subscriber interests and on techniques for matching published content with queries expressed in these domain specific languages.

Content-based publish-subscribe systems have either designed their own query languages or used off the shelf standard query languages. These languages allow users to specify their interests in simple terms through relational algebra or in fairly complex terms through user-written scripts in Turing-complete languages. Similarly the content handled by the publish-subscribe systems ranges from structured content with well-defined attributes, to semi-structured content such as XML, and unstructured content such as web objects

Several publish-subscribe systems assume that the published content has well-defined meta-data or attributes against which queries can be issued. Examples of these systems include, Gryphon [140], Siena [27], Elvin [133], Jedi [40], Java Messaging Services [70] (JMS), Astrolable [144], SDIMS [160], Cone [15], Pepper [90], and PIER [72]. A few systems such as Pepper and Cone only support equality and range queries. Several including Gryphon, Siena, and Elvin, invent their own query language, while others such as Astrolabe, PIER, and JMS support queries expressed using SQL, a widely used query language for relational databases.

Several publish-subscribe systems have been designed for content published

in XML, an industrial standard for representing semi-structured data of ad hoc structure. The key reason for adopting XML is that it forms an intermediate between structured and unstructured content. Since having universal, globally-defined attributes in published content is difficult to implement in practice, self-describing content in XML provides an alternative. XML queries are described using XPath expressions for simple queries and XQL (XML Query Language) for complex queries. XPath and XQL are part of the XML standard accepted by a collaboration of leading enterprizes.

Several researchers have proposed efficient content-filtering algorithms for XML. Altinel and Franklin have proposed a mechanism called XFilter based on finite state machines for filtering XML documents [6]. Yanlei et al. have proposed an improved mechanism called Yfilter based on non-deterministic finite state machines [45]. Other approaches for XML content filtering include the use of a trie-based data structure called XTrie by Chan and Rastogi [32] and enhancing XML filtering with keyword search in Niagara [105].

Few publish-subscribe systems are designed for unstructured web content. Conquer [94] and WebCQ [93] support keyword queries on web objects and provide asynchronous event notifications. These systems apply database techniques for processing streaming data to detect perform keyword searches. However, similar to other publish-subscribe systems described earlier, these systems do not focus on an important aspect in the design of publish-subscribe systems for the Web, namely detecting changes made to Web content. This important aspect of update detection is reviewed later in this section.

Content-based publish-subscribe systems also organize the network of participating nodes to ensure efficient content matching and event dissemination. As-

trolabe and SDIMS organize the network into a hierarchy and take advantage of aggregating query results from children at the parent nodes to improve the efficiency of content filtering. Few systems, instead organize the network into a flat peer-to-peer overlays to achieve better load balance. Examples of publish-subscribe systems layered on structured overlays include Cone, Pepper, and PIER. XTreeNet [55], a publish-subscribe system for XML content, uses unstructured overlay to organize its network.

The fundamental drawback of the preceding publish-subscribe systems is their non-compatibility with the current Web architecture. They require substantial changes in the way publishers serve content, expect subscribers to learn sophisticated query languages, or propose to lay out middle boxes in the core of the Internet.

6.5.3 Detecting Changes in the Web

Few publish-subscribe have focused on supporting the Web as is without requiring Web content providers to change the way they provide content. The key problem faced by these publish-subscribe systems is detecting changes in web content using the pull-based mechanisms of the Web.

FeedTree [132], is a recently proposed system topic-based publish-subscribe system for detecting changes in web pages and disseminating the updates to the users. FeedTree participants cooperatively poll the web to detect changes and share updates with each other. FeedTree organizes the nodes using Pastry structured overlay, allocates polling tasks to nodes in the system, and disseminates updates using the Scribe application level multicast protocol. However, FeedTree nodes decide to poll for a feed and share updates based in an ad hoc manner based on

heuristics.

A principled approach that takes into account tradeoffs between resource availability and performance, have been explored for optimizing update detection in centralized systems. CAM [108] is an algorithm for allocating polling resources in a centralized server for achieving different application specific performance goals. WIC [107] is another algorithm for allocating polling resources in a centralized server that focuses on the between rate of polling and update detection rate. Cho and Garcia-Molina [35] propose techniques to derive optimal polling schedules to optimize average freshness and age of objects in a web cache. While these algorithms employ optimization based approach, similar to Corona, the above proposed techniques cannot be applied for optimizing polling in a decentralized setting. Moreover, the effectiveness of the above algorithms has not been demonstrated in a deployed system.

Corona, the publish-subscribe system presented in this thesis, uses optimal resource allocation to poll web pages and detect updates in a distributed system. This principled approach enables Corona to provide the best update performance for its users, while ensuring that content servers are lightly loaded and do not get overwhelmed due to flash crowds or sticky traffic. Moreover, Corona requires no additional tools to be installed by the clients, but employs an easy-to-use IM based interface to the users.

Chapter 7

Conclusions

This thesis presented a principled approach to building decentralized network services that offer high performance guarantees. In this domain, judicious use of scarce or expensive resources is critical to achieving high performance. The thesis formulated this fundamental resource-performance tradeoff mathematically as constrained optimization problems, derived near-optimal solutions to the modeled problems, and used these solutions to achieve desired performance goals in real world applications. The result was an ability to achieve unprecedented improvement in performance compared to ad hoc, less-informed techniques widely employed in today's applications.

This chapter summarizes the key contributions of this thesis, places the contributions in context, and provides directions for future work.

7.1 Summary

Distributed systems today predominantly employ techniques that are unaware or less conscious of resource consumption. This trend continues despite the fact that performance of network services critically depends on the available quantity of resources such as storage space, network bandwidth, and computing servers. The scale of today's network applications can be so large that the resources required to achieve high levels of performance is prohibitively expensive. For example, the amount of space and bandwidth required to replicate even a modest fraction of the web is beyond the reach of most institutions. Thus, careful utilization of resources is crucial to obtain large gains in performance.

The approach presented in this thesis enables applications to control their resource utilization at a fine grain. It provides a tunable knob that applications can use to meet system wide performance goals. Accurate analytical models of how individual object parameters such as size, popularity, and update rate influence resource consumption ensure that the system achieves the targeted performance at low cost. Alternatively, instead of specifying explicit performance targets, applications can just obtain high performance by making judicious use of the available resources.

This analysis-driven approach provides a fundamentally different level of performance compared to commonly used heuristic-driven techniques. For instance, it is common knowledge that passive, opportunistic web caches provide low performance as web objects follow heavy-tailed popularity distributions [18, 155]. This thesis showed a way to surpass these limitations and achieve high performance in the presence of heavy tailed popularity distributions. Unlike heuristics tailored to specific workloads, the approach presented in this thesis is general and supports a broad class of applications as it handles any distribution of object characteristics.

The above theoretically sound approach lends itself to a scalable, adaptable, and robust system. This thesis presented decentralized, light-weight techniques to implement a practical resource management framework. This framework, called Honeycomb, scales logarithmically with the size of the system, continuously learns and adapts itself to changes in workload, and efficiently tolerates network and node failures.

Honeycomb currently supports three real network services. These services described in the thesis, namely the CoDoNS naming system, the CobWeb content distribution network, and the Corona data monitoring system, are deployed on a plan-

etary scale test bed and are available for public use. Evaluation of these deployed systems show that Honeycomb can be well-suited for building such performance-demanding applications.

7.2 Limitations and Future Work

While this thesis was largely self-contained and explored important directions to study resource management in decentralized network services, operational experience in running the above services exposed a few limitations in our approach and opened up several new and interesting avenues to explore further.

This thesis does not address locality of access for objects. While highly popular objects tend to be popular throughout the system, there are also objects that have high popularity only in a particular locality. Our current replication protocol assumes that objects are uniformly popular in all locations. However, the independent, decentralized decision making approach presented in this thesis can be easily extended to handle locally popular objects efficiently. In order to replicate objects locally, the network can be organized into a locality-aware overlay through proximity-based neighbor selection [30, 43], which ensures that a node's one-hop neighbors are located in close proximity to the node. Further more, using the locally estimated popularity of objects, instead of the globally aggregated popularity, can make object replicas reside close to their locality of access.

Second, this thesis treats all nodes in the system uniformly. That is, it assumes that all nodes have the same amount of resources such as bandwidth and memory. While the kind of infrastructure services explored in this thesis are expected to run on sufficiently well-provisioned hardware, it is practical to expect a limited degree of heterogeneity in the amount of resources available to different nodes. Heteroge-

neous nodes can be easily handled by using the notion of *virtual nodes* [42], where a real node is split into several virtual nodes with constant bandwidth and storage or memory capacities. Splitting nodes into virtual nodes at a coarse granularity ensures that the system appears homogeneous while few virtual nodes are introduced in the system.

Finally, the analytical models presented in this thesis predominantly focused only on the average values of performance and cost. While average is a useful metric to quantify resource consumption or level of performance, additional metrics such as percentiles and variance may be better-suited for some applications. For example, Internet Service Providers (ISPs) often charge for bandwidth based on the 95^{th} percentile rather than the average, while service level agreements (SLAs) typically specify performance requirements in percentiles. Similarly, using the variance enables applications to bound the performance difference across objects. Extending the resource allocation techniques to handle percentiles and variance would substantially expand the domain of suitable applications.

7.3 Impact

Despite a few limitations, the work presented in this thesis has already attracted substantial attention. The three public services we maintain continues to gain new users. CoDoNS is our first deployed service, now in operation since August 2004. In addition to several anonymous users of our DNS service, we have received interest from companies and organizations that want to host their own CoDoNS network. In particular, CNNIC (China Internet Network Information Center) has expressed a desire to use CoDoNS to host the names under the *.cn* top level domain. CobWeb, our open-access CDN, is our largest used service. Since deployment in

May 2005, CobWeb has been serving about 10-12 million queries per day. Finally, Corona, deployed in February 2006, has already attracted over a hundred users, who have rarely withdrawn from the system.

Overall, this thesis demonstrated that a well-informed, optimization-based approach to resource allocation is practical even for planetary-scale decentralized systems and leads to unprecedented performance improvement over conventional, heuristic-based techniques.

BIBLIOGRAPHY

[1] Ghaleb Abdulla. *Analysis and Modeling of World Wide Web Traffic*. PhD thesis, Virginia Polytechnic Institute and State University, 1998.

[2] Marc Abrams, Charles Standbridge, Ghaleb Abdulla, Stephen Williams, and Edward Fox. Cost-Aware WWW Proxy Caching Algorithms. In *Proc. of the International World Wide Web Conference (WWW)*, Boston, MA, December 1995.

[3] Atul Adya, William Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John Douceur, Jon Jowell, Jacob Lorch, Marvin Theimer, and Roger Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proc. on Symposium of Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[4] Paul Albitz and Cricket Liu. *DNS and BIND*. O'Reilly, 2001.

[5] Virgilio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. Characterizing Reference Locality in the WWW. In *Proc. of IEEE International Conference in Parallel and Distributed Information Systems*, Tokyo, Japan, June 1996.

[6] Mehmet Altinel and Michael Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. of International Conference on Very Large Databases (VLDB)*, Cairo, Egypt, September 2000.

[7] Mark Andrews. Negative Caching of DNS Queries. Request for Comments (RFC) 2308, March 1998.

[8] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol Modifications for the Domain Name System Security Extensions. Request for Comments 4035, March 2005.

[9] Martin Arlitt and Carey Williamson. Web Server Workload Characterization: the Search for Invariants. Philadelphia, PA, May 1996.

[10] Hari Balakrishnan, Karthik Lakshminarayanan, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Michael Walfish. A Layered Naming Architecture for the Internet. In *Proceedings of the ACM SIGCOMM Conference*, Porland, OR, August 2004.

[11] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in Web Client Access Patterns: Characteristics and Caching Implications. *World Wide Web*, 2(1-2):15–28, 1999.

[12] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proc. of Symposium on Networked Systems Design and Implementation*, Boston, MA, March 2004.

[13] Leann Bent and Goeffrey Voelker. Whole Page Performance. In *Proc. of the International Workshop on Web Content Caching and Distribution*, Boulder, CO, August 2002.

[14] Azer Bestavros. Demand-based Document Dissemination to Reduce Traffic and Balance Load in Distributed Information Systems. In *Proceedings of the Symposium on Parallel and Distributed Processing*, San Anotonio, TX, February 1995.

[15] Ranjita Bhagwan, George Varghese, and Geoff M Voelker. Cone: Augmenting DHTs to Support Distributed Resource Discovery. Technical Report CS2003-0755, University of California at San Diego, July 2003.

[16] Kenneth Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, 1999.

[17] Tim Bray. Measuring the Web. Paris, France, May 1996.

[18] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. of IEEE International Conference on Computer Communications*, New York, NY, March 1999.

[19] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. of IEEE International Conference on Computer Communications*, New York, NY, March 1999.

[20] Thomas Brisco. DNS Support for Load Balancing. Request for Comments (RFC) 1794, April 1995.

[21] N. Brownlee, kc claffy, and E. Nemeth. DNS Measurements at a Root Server. In *Proc. of IEEE Global Telecommunications Conference*, San Antonio, TX, November 2001.

[22] N. Brownlee, kc claffy, and E. Nemeth. DNS Measurements at a Root Server. In *Proc. of IEEE Global Telecommunications Conference*, San Antonio, TX, November 2001.

[23] Nevil Brownlee, kc Claffy, and Evi Nemeth. DNS Root/gTLD Performance Measurements. In *Proc. of Usenix Systems Administration Conference*, San Diego, CA, December 2001.

[24] Luis Felipe Cabera, Michael B Jones, and Marvin Theimer. Herald: Achieving a Global Event Notification Service. In *Proc. of Workshop on Hot Topics in Operating Systems*, Elmau, Germany, May 2001.

[25] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proc. of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, December 1997.

[26] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, April 1989.

[27] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[28] Miguel Castro, Manuel Costa, and Antony Rowstron. Debunking some Myths about Structured and Unstructured Overlays. In *Proceedings of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, CA, May 2005.

[29] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan Wallach. Secure Routing for Structured Peer-to-Peer Overlay Networks. In *Proc. of Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[30] Miguel Castro, Peter Druschel, Charlie Hu, and Antony Rowstron. Proximity Neighbor Selection in Tree-Based Structured Peer-to-Peer Overlays. Technical Report MSR-TR-2003-52, Microsoft Research, September 2003.

[31] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: High-Bandwidth Multicast in a Cooperative Environment. In *Proc. of ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003.

[32] Chan, Fan, Felber, M Garofalakis, and Rajiv Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proc. of International Conference on Data Engineering*, San Jose, CA, February 2002.

[33] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A Hierarchical Internet Object Cache. In *Proc. of USENIX Annual Technical Conference*, pages 153–164, San Diego, CA, January 1996.

[34] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making Gnutella-like P2P Systems Scalable. Karlsruhe, Germany, August 2003.

[35] Junghoo Cho and Hector Garcia-Molina. Effective Page Refresh Policies. *ACM Transactions on Database Systems*, 28(4), 2003.

[36] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. *Lecture Notes in Computer Science*, 2009, 2001.

[37] Edith Cohen and Haim Kaplan. Proactive Caching of DNS Records: Addressing a Performance Bottleneck. In *Proc. of Symposium on Applications in the Internet*, San Diego-Mission Valley, CA, January 2001.

[38] Edith Cohen and Scott Shenker. Replication Strategies in Unstructured Peer-to-Peer Networks. In *Proc. of ACM SIGCOMM*, Pittsburgh, PA, August 2002.

[39] Russ Cox, Athicha Mutitacharoen, and Robert Morris. Serving DNS using a Peer-to-Peer Lookup Service. In *Proc. of International Workshop on Peer-to-Peer Systems*, Cambridge, MA, March 2002.

[40] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–850, 2001.

[41] Carlos Cunha, Azer Bestavros, and Mark Crovella. Characteristics of WWW Client-Based Traces. Technical Report TR-95-010, Boston University, 1995.

[42] Frank Dabek, Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-Area Cooperative Storage with CFS. In *Proc. of ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[43] Frank Dabek, Jinyang Li, Emil Sit, M Frans Kaashoek, Robert Morris, and Chuck Blake. Designing a DHT for Low Latency and High Throughput. In *Proc. of USENIX Symposium on Networked Systems Design and Implementation*, San Francisco, CA, March 2004.

[44] Peter Danzig, Katia Obraczka, and Anant Kumar. An Analysis of Wide-Area Name Server Traffic: A Study of the Internet Domain Name System. In *Proc. of ACM SIGCOMM*, Baltimore, MD, August 1992.

[45] Yanlei Diao, Shariq Rizvi, and Michael Franklin. Towards an Internet-Scale XML Dissemination Service. In *Proc. of International Conference on Very Large Databases (VLDB)*, Toronto, Canada, August 2004.

[46] John R. Douceur, Atul Adya, William J.Bolosky, and Dan Simon. Reclaiming Space from Duplicate Files in a Serverless Distributed File System. In *Proc. of International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.

[47] Fred Douglis, Anja Feldman, Balachander Krishnamurthy, and Jeffrey Mogul. Rate of Change and Other Metrics: a Live Study of the World Wide Web. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997.

[48] Lawrence Dowdy and Derrell Foster. Comparative Models of the File Assignement Problem. *ACM Computer Surveys*, 14:287–314, 1982.

[49] Bradley M. Duska, David Marwood, and Michael Feeley. The Measured Access Characteristics of World Wide Web Client Proxy Caches. In *Proc. of USENIX Symposium on Internet Technology and Systems*, Monterey, CA, December 1997.

[50] Donald Eastlake. Domain Name System Security Extensions. Request for Comments 2335, March 1999.

[51] R. Elz and R. Bush. Clarifications to the DNS Specifications. Request for Comments 2181, July 1997.

[52] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[53] Li Fan, Quinn Jacobson, Pei Cao, and Wei Lin. Web Prefetching between Low-Bandwidth Clients and Proxies: Potential and Performance. In *Proc. of the ACM Conference on the Measurement and Modeling of Computer Systems (SigMetrics)*, Atlanta, GA, May 1999.

[54] Anja Feldmann, Ramon Caceres, Fred Douglis, Gideon Glass, and Michael Rabinovich. Performance of Web Proxy Caching in Heterogeneous Bandwidth Environments. New York, NY, 1999.

[55] William Fenner, Michael Rabinovich, K K Ramakrishnan, Divesh Srivastava, and Yin Zhang. XTreeNet: Scalable Overlay Networks for XML Content Dissemination and Querying (Synopsis). In *Proc. of International Workshop on Web Content Caching and Distribution*, Sophia Antipolis, France, September 2005.

[56] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing. *IEEE/ACM Transaction on Networking*, pages 784–803, December 1997.

[57] Michael Freedman, Eric Freudenthal, and David Mazières. Democratizing Content Publication with Coral. In *Proc. of Symposium on Networked Systems Design and Implementation*, San Francisco, CA, March 2004.

[58] Michael Freedman, Karthik Lakshminarayanan, Sean Rhea, and Ion Stoica. Non-Transitive Connectivity and DHTs. In *Proc. of the Usenix Workshop on Real Large Distributed Systems (WORLDS)*, San Francisco, CA, December 2005.

[59] Michael Freedman and David Mazières. Sloppy Hashing and Self-Organizing Clusters. In *Proc. of International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, February 2003.

[60] Syam Gadde, Michael Rabinovich, and Jeffrey S. Chase. Reduce, Reuse, Recycle: An Approach to Building Large Internet Caches. In *Proc. of Workshop on Hot Topics in Operating Systems*, Cape Cod, MA, May 1997.

[61] GAIM. A Multi-Protocol Instant Messaging Client. http://gaim.sourceforge.net.

[62] Bradford Glade, Kenneth P. Birman Robert Cooper, and Robert van Renesse. Light-Weight Process Groups in the ISIS System. *Distributed Systems Engineering*, 1(1):29–36, sep 1993.

[63] Steven Glassman. A Caching Relay for the World Wide Web. *Computer Networks and ISDN Systems*, 27(2):165–173, 1994.

[64] Ashish. Goal, Hui Zhang, and Ramesh Govindan. Incrementally Improving Lookup Latency in Distributed Hash Table Systems. In *Proc. of the ACM International Conference on Measurement and Modeling of Computer Systems (SigMetrics)*, San Diego, CA, June 2003.

[65] Steven Gribble and Eric Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997.

[66] Anjali Gupta, Barabara Liskov, and Rodrigo Rodrigues. One Hop Lookups for Peer-to-Peer Overlays. In *Proc. of Workshop on Hot Topics in Operating Systems*, Lihue, HI, May 2003.

[67] Anjali Gupta, Barabara Liskov, and Rodrigo Rodrigues. Efficient Routing for Peer-to-Peer Overlays. In *Proc. of Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, March 2004.

[68] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robert van Rennesse. Kelips: Building an Efficient and Stable P2P DHT Through Increased Memory and Background Overhead. In *Proc. of International Workshop on Peer-to-Peer Systems*, Berkeley, CA, February 2003.

[69] Mark Handley and Adam Greenhalgh. The Case for Pushing DNS. In *Proceedings of the Workshop on Hot Topics in Networks (HotNets)*, College Park, MD, November 2005.

[70] Mark Hapner, Rich burridge, Rahul sharma, and Joseph Fialli. Java Message Service. http://java.sun.com/products/jms/docs.html, April 2002.

[71] Nicholas Harvey, Michael Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, March 2003.

[72] Ryan Huebsch, Joseph M Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER. In *Proc. of International Conference on Very Large Databases*, Berlin, Germany, September 2003.

[73] C. Huitema and S. Weerahandi. Internet measurements: The rising tide and the DNS Snag. In *Proc. of ITC Specialist Seminar on Internet Traffic Measurement and Modeling*, Monterey, CA, September 2000.

[74] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: A Decentralized Peer-to-Peer Web Cache. In *Proc. ACM Symposium on Principles of Distributed Computing*, Monterey, CA, July 2002.

[75] Sugih Jamin, Cheng Jin, Anthony Kurc, Danny Raz, and Yuval Shavitt. Constrained Mirror Placement on the Internet. In *Proc. of INFOCOM Conference*, Anchorage, AL, April 2001.

[76] Kirk L. Johnson, John F. Carr, Mark S. Day, and M. Frans Kaashoek. The Measured Performance of Content Distribution Networks. In *Proc. of International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[77] William K. Josephson, Emin Gün Sirer, and Fred B. Schneider. Peer-to-Peer Authentication With a Distributed Single Sign-On Service. In *Proc. of International Workshop on Peer-to-Peer Systems*, San Diego, CA, February 2004.

[78] Jaeyon Jung, Arthur Berger, and Hari Balakrishnan. Modeling TTL-based Internet Caches. In *Proc. of IEEE International Conference on Computer Communications*, San Francisco, CA, March 2003.

[79] Jaeyon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. DNS Performance and Effectiveness of Caching. In *Proc. of SIGCOMM Internet Measurement Workshop*, San Francisco, CA, November 2001.

[80] Frans Kaashoek and David Karger. Koorde: A Simple Degree-Optimal Distributed Hash Table. In *Proc. of International Workshop on Peer-to-Peer Systems*, Berkeley, CA, February 2003.

[81] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigraphy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. of ACM Symposium on Theory of Computing*, El Paso, TX, April 1997.

[82] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer Verlag, 2005.

[83] Balachander Krishnamurthy, Craig Wills, and Yin Zhang. On the Use and Performance of Content Distribution Networks. In *Proc. of ACM SIGCOMM Workshop on Internet Measurement*, San Francisco, CA, November 2001.

[84] Thomas Kroeger, Jeff Mogul, and Carlos Maltzahn. Digital's Web Proxy Traces, August 1996.

[85] Tom M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997.

[86] Anant Kumar, Jon Postel, Clifford Neuman, Paul Danzig, and Steve Miller. Common DNS Implementation Errors and Suggested Fixes. Request for Comments 1536, October 1993.

[87] James Kurose and Rahul Simha. A Microeconomic Approach to Optimal Resource Allocation in Distributed Computer Systems. *IEEE Transactions on Computers*, 38(5):705–717, May 1989.

[88] Bo Li, Mordecai Golin, Guiseppe Ialiano, and Xin Deng. On the Optimal Placement of Web Proxies in the Internet. In *Proc. of INFOCOM Conference*, New York, NY, March 1999.

[89] John C. Lin and Sanjoy Paul. RMTP: A Reliable Multicast Transport Protocol. In *Proc. of the IEEE INFOCOM*, San Francisco, CA, March 1996.

[90] Prakash Linga, Adina Crainiceanu, Johannes Gehrke, and Jayavel Shanmugasundaram. Guaranteeing Correctness and Availability in P2P Range Indices. In *Proc. of ACM SIGMOD Confernece*, Baltimore, MD, June 2005.

[91] Prakash Linga, Indranil Gupta, and Ken Birman. A Churn-Resistant Peer-to-Peer Web Caching System. In *Proc. of the Workshop on Survivable and Self-Regenerative Systems (SSRS)*, Fairfax, VA, February 2003.

[92] Hongzhou Liu, Venugopalan Ramasubramanian, and Emin Gün Sirer. Client Behavior and Feed Characteristics of RSS, a Publish-Subscribe System for Web Micronews. In *Proc. of ACM Internet Measurement Conference*, Berkeley, CA, October 2005.

[93] Ling Liu, Calton Pu, and Wei Tang. WebCQ: Detecting and Delivering Information Changes on the Web. In *Proc. of the International Conference on Information and Knowledge Management*, McLean, VA, November 2000.

[94] Ling Liu, Calton Pu, Wei Tang, and Wei Han. CONQUER: A Continual Query System for Update Monitoring in the WWW. *International Journal of Computer Systems, Science and Engineering*, 14(2):99–112, 1999.

[95] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *Proc. of International Conference on Supercomputing*, New York, NY, June 2002.

[96] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proc. of ACM Symposium on Principles of Distributed Computing*, Monterey, CA, August 2002.

[97] Petar Maymounkov and David Mazieres. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *Proc. of International Workshop on Peer-to-Peer Systems*, Cambridge, CA, March 2002.

[98] Scott Michel, Khoi Nguyen, Adam Rosenstein, Lixia Zhang, Sally Floyd, and Van Jacobson. Adaptive Web Caching: Towards a New Global Caching Architecture. In *Proc. of International WWW Caching Workshop*, Manchester, UK, June 1997.

[99] Alan Mislove, Ansley Post, Charles Reis, Paul Willmann, Peter Druschel, Dan S Wallach, Xavier Bonnaire, Pierre Sens, Jean-Michel Busca, and Luciana Arantes-Bezerra. POST: A Secure, Resilient, Cooperative Messaging System. In *Proc. of International Workshop on Peer-to-Peer Systems*, Berkeley, CA, February 2003.

[100] Alper Mizrak, Yuchung Cheng, Vineet Kumar, and Stefan Savage. Structured Superpeer: Leveraging Heterogeneity to Provide Constant-time Lookup. In *Proc. of IEEE Workshop on Internet Applications*, San Francisco, CA, April 2003.

[101] Paul Mockapetris. Domain Names: Concepts and Facilities. Request for Comments 1034, November 1987.

[102] Paul Mockapetris. Domain Names: Implementation and Specification. Request for Comments 1035, November 1987.

[103] Paul Mockapetris and Kevin Dunlop. Development of the Domain Name System. In *Proc. of ACM SIGCOMM*, Stanford, CA, August 1988.

[104] Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen. IVY: A Read/Write Peer-to-Peer File System. In *Proc of Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[105] Jeffrey Naughton, David DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Gupta, and Rushan Chen. The Niagara Internet Query System. *The IEEE Data Engineering Bulletin*, 24(2):27–33, 2001.

[106] Venkat Padmanabhan and Jeffrey Mogul. Using Predictive Prefetcing to Improve World Wide Web Latency. In *Proc. of the ACM SIGCOMM Conference*, Stanford, CA, August 1996.

[107] Sandeep Pandey, Kedar Dhamdere, and Christpoher Olston. WIC: A General-Purpose Algorithm for Monitoring Web Information Sources. In *Proc. of the Conference on Very Large Data Bases (VLDB)*, Toronto, Canada, August 2004.

[108] Sandeep Pandey, Krithi Ramamritham, and Soumen Chakraborti. Monitoring the Dynamic Web to Respond to Continuous Queries. In *Proc. of the International World Wide Web Conference*, Budapest, Hungary, May 2003.

[109] Jeffrey Pang, Aditya Akella, Anees Shaikh, Balachander Krishnamurthy, and Srinivasan Seshan. On the Responsiveness of DNS-based Network Control. In *Proc of the ACM SIGCOMM Conference on Internet Measurement*, Taormina, Italy, October 2004.

[110] Vasileios Pappas, Zhiguo Xu, Songwu Lu, Daniel Massey, Andreas Terzis, and Lixia Zhang. Impact of Configuration Errors on DNS Robustness. In *Proc. of ACM SIGCOMM*, Portland, OR, August 2004.

[111] KyoungSoo Park, Vivek Pai, and Larry Peterson. CoDNS: Improving DNS Performance and Reliability via Cooperative Lookups. In *Proc. of Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[112] Limin Park, KyoungSoo Park, Ruoming Pang, Vivek Pai, and Larry Peterson. Reliability and Security in the CoDeeN Content Distribution Network. In *Proc. of USENIX Annual Technical Conference*, Boston, MA, June 2004.

[113] Ryan Peterson, Venugopalan Ramasubramanian, and Emin Gün Sirer. A Practical Approach to Peer-to-Peer Publish-Subscribe. *Usenix ;login:*, 31(4):42–46, August 2006.

[114] Greg Plaxton, Rajmohan Rajaraman, and Andrea Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. *Theory of Computing Systems*, 32:241–280, 1999.

[115] William Press, Brian Flannery, Saul Teukolsky, and William Vetterling. *Numerical Recipies in C: The Art of Scientific Computing*. Cambridge University Press, 2 edition, 1992.

[116] Lili Qiu, Venkat Padmanabhan, and Geoff Voelker. On the Placement of Web Server Replicas. In *Proc. of INFOCOM Conference*, Anchorage, AL, April 2001.

[117] Venugopalan Ramasubramanian, Ryan Peterson, and Emin Gün Sirer. Corona: A High Performance Publish-Subscribe System for the World Wide Web. In *Proc. of Symposium on Networked Systems Design and Implementation*, San Jose, CA, May 2006.

[118] Venugopalan Ramasubramanian and Emin Gün Sirer. Beehive: Exploiting Power Law Query Distributions for 0(1) Lookup Performance in Peer-to-Peer Overlays. In *Proc. of Symposium on Networked Systems Design and Implementation*, San Francisco, CA, March 2004.

[119] Venugopalan Ramasubramanian and Emin Gün Sirer. The Design and Implementation of a Next Generation Name Service for the Internet. In *Proc. of ACM SIGCOMM*, Portland, OR, August 2004.

[120] Venugopalan Ramasubramanian and Emin Gün Sirer. Perils of Transitive Trust in the Domain Name System. In *Proc. of ACM Internet Measurement Conference*, Berkeley, CA, October 2005.

[121] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *Proc. of ACM SIGCOMM*, San Diego, CA, August 2001.

[122] Sean Rhea, Byung-Gon Chun, John Kubiatowicz, and Scott Shenker. Fixing the Embarrassing Slowness of OpenDHT on PlanetLab. In *Proc. of the Usenix Workshop on Real Large Distributed Systems (WORLDS)*, San Francisco, CA, December 2005.

[123] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *Proc. of USENIX Annual Technical Conference*, Boston, MA, June 2004.

[124] Liugi Rizzo and Lorenzo Vicisano. Replacement Policies for a Proxy Cache. *IEEE/ACM Transactions on Networking (ToN)*, 8(2):158–170, 2000.

[125] Luigi Rizzo. Web Proxy Traces, May 1997.

[126] Keith W. Ross. Hash-Routing for Collections of Shared Web Caches. *IEEE Network Magazine*, 1997.

[127] Mema Roussopoulos and Mary Baker. CUP: Controlled Update Propagation in Peer-to-Peer Networks. In *Proc. of USENIX Annual Technical Conference*, San Antonio, TX, June 2003.

[128] Antony Rowstorn and Peter Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Proc. of IFIP/ACM International Conference on Distributed Systems Platforms*, Heidelberg, Germany, November 2001.

[129] Antony Rowstorn and Peter Druschel. Storage Management and Caching in PAST, a Large-scale, Persistent Peer-to-Peer Storage Utility. In *Proc. of ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[130] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The Design of a Large-scale Event Notification Infrastructure. In *Proc. of Workshop on Networked Group Communications*, London, UK, November 2001.

[131] RSS 2.0 Specifications. http://blogs.law.harvard.edu/tech/rss, 2005.

[132] Dan Sandler, Alan Mislove, Ansley Post, and Peter Druschel. FeedTree: Sharing Web Micronews with Peer-to-Peer Event Notification. In *Proc. of International Workshop on Peer-to-Peer Systems*, Ithaca, NY, February 2005.

[133] Bill Segall, David Arnold, Julian Boot, Michael Henderson, and Ted Phelps. Content Based Routing with Elvin. In *Proc. of AUUG2K*, Canberra, Australia, June 2000.

[134] Anees Shaikh, Renu Tewari, and Mukesh Agarwal. On the Effectiveness of DNS-based Server Selection. In *Proc. of IEEE International Conference on Computer Communications*, Anchorage, AK, April 2001.

[135] Emil Sit, Frank Dabek, and James Robertson. UsenetDHT: A Low Overhead Usenet Server. In *Proc. of the International Workshop on Peer-to-Peer Systems (IPTPS)*, San Diego, CA, February 2004.

[136] Yee Jiun Song, Venugopalan Ramasubramanian, and Emin Gün Sirer. Optimal Resource Utilization in Content Distribution Networks. Technical Report TR-2005-2004, Cornell University, Computing and Information Science, November 2005.

[137] Tyron Stading, Petros Maniatis, and Mary Baker. Peer-to-Peer Caching Schemes to Address Flash Crowds. In *Proc. of International Workshop on Peer To Peer Systems*, Cambridge, MA, March 2002.

[138] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM*, San Diego, CA, August 2001.

[139] Jeremy Stribling, Isaac Councill, Jinyang Li andFrans Kaashoek, David Karger, Robert Morris, and Scott Shenker. OverCite: A Cooperative Digital Research Library. In *Proc. of the International Workshop on Peer-to-Peer Systems (IPTPS)*, Ithaca, NY, February 2005.

[140] Robert Strom, Guruduth Banavar, Tushar Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman, and Michael Ward. Gryphon: An Information Flow Based Approach to Message Brokering. In *Proc. of International Symposium on Software Reliability Engineering*, Paderborn, Germany, November 1998.

[141] Renu Tewari, Michael Dahlin, Harrick Vin, and John Kay. Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet. Technical Report TR98-0, University of Texas at Austin, 1998.

[142] David Thaler and Chiniya Ravishankar. A Name-based Mapping Scheme for Rendezvous. *ACM/IEEE Transactions on Networking*, 6(1):1–14, February 1998.

[143] Marvin Theimer and Michael Jones. Overlook: Scalable Name Service on an Overlay Network. In *Proc. of the International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002.

[144] Robbert van Renesse, Kenneth P Birman, and Werner Vogels. Astrolabe: A Robust and Scalable Technology For Distributed Systems Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21(3), May 2003.

[145] Benjamin Wah. File Placement in Distributed Computer Systems. *IEEE Computer*, 17:22–33, 1984.

[146] Carl Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, 1995.

[147] Michael Walfish, Hari Balakrishnan, and Scott Shenker. Untangling the Web from DNS. In *Proceedings of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, March 2004.

[148] Limin Wang, Vivek Pai, and Larry Peterson. The Effectiveness of Request Redirection on CDN Robustness. In *Proc. of Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[149] D. Wessels and K. Claffy. Internet Cache Protocol (ICP), version 2. Request For Comments (RFC) 2186, 1997.

[150] Duane Wessels and kc Claffy. ICP and the Squid Web Cache. *IEEE Journal on Selected Areas in Communications*, 16(3):345–357, 1998.

[151] Udi Wieder and Moni Naor. A Simple Fault Tolerant Distributed Hash Table. In *Proc. of International Workshop on Peer-to-Peer Systems*, Berkeley, CA, February 2003.

[152] Stephen Williams, Marc Abrams, Charles Standbridge, Ghaleb Abdulla, and Edward Fox. Removal Policies in Network Caches for World Wide Web Documents. In *Proc. of the ACM SIGCOMM Conference*, Stanford, CA, August 1996.

[153] Craig E. Wills and Hao Shang. The Contribution of DNS Lookup Costs to Web Object Retrieval. Technical Report TR-00-12, Worcester Polytechnic Institute, July 2000.

[154] Alec Wolman, Geoff Voelker, Nitin Sharma, Neal Cardwell, Molly Brown, Tashana Landray, Denise Pinnel, Anna Karlin, and Henry Levy. Organization-Based Analysis of Web-Object Sharing and Caching. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, October 1999.

[155] Alec Wolman, Goeffrey Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry Levy. On the Scale and Performance of Cooperative Web Proxy Caching. In *Proc. of ACM Symposium on Operating Systems Principles*, Kiawah Island, CA, December 1999.

[156] Alec Wolman, Goeffrey Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry Levy. On the Scale and Performance of Cooperative Web Proxy Caching. In *Proc. of ACM Symposium on Operating Systems Principles*, Kiawah Island, CA, December 1999.

[157] Bernard Wong and Emin Gün Sirer. ClosestNode.com: An Open-Access, Scalable, Shared Geocast Service for Distributed Systems. *SIGOPS Operating Systems Review*, 40(1), January 2006.

[158] Bernard Wong, Aleksandrs Slivkins, and Emin Gün Sirer. Meridian: a Lightweight Network Location Service without Virtual Coordinates. In *Proc. of ACM SIGCOMM*, Philadelphia, PA, August 2005.

[159] Roland Wooster and Marc Abrams. Proxy Caching that Estimates Page Load Delays. In *Proc. of the International World Wide Web Conference (WWW)*, Santa Clara, CA, April 1997.

[160] Praveen Yalagandula and Mike Dahlin. A Scalable Distributed Information Management System. In *Proc. of ACM SIGCOMM*, Porland, OR, August 2004.

[161] Ben Zhao, Ling Huang, Jeremy Stribling, Sean Rhea, Anthony Joseph, and John Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[162] Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. COCA: A Secure Distributed On-line Certification Authority. *ACM Transactions on Computer Systems*, 20(4):329–368, November 2002.

[163] George K. Zipf. *Human Behavior and the Principle of Least-Effort*. Addison-Wesley, Cambridge, MA, 1949.

[164] Ingrid Zukerman, David Albrecht, and Ann Nicholson. Predicting users' requests on the WWW. In *Proc. of the International Conference on User Modeling*, Banff, Canada, June 1999.

[165] Microsoft Suffers another DoS Attack. http://www.winnetmag.com /WindowsSecurity/Article/ArticleID/19770/WindowsSecurity_19770.html, January 2001.

[166] Massive DDoS Attack Hit DNS Root Servers. http://www.internetnews.com/dev-news/article.php/1486981, October 2002.

[167] Atom Syndication Format. http://www.atomenabled.org/developers /syndication.

[168] BIND Vulnerabilities. http://www.isc.org/sw/bind/bind-security.php, February 2004.

[169] The Gnutella 0.4 Protocol Specification. http://dss.clip2.com/GnutellaProtocol0.4.pdf, 2000.

[170] JS-JavaSpaces Service Specification. http://www.jini.org/nonav/standards /davis/doc/specs/html/js-spec.html, 2002.

[171] Kazaa. http://www.kazaa.com.

[172] National Laboratory of Applied Network Research. http://www.nlanr.org, July 1997.

[173] Overnet. http://www.edonkey2000.com/documentation/how_on.html.

[174] TIBCO Publish-Subscribe. http://www.tibco.com.

[175] TSpaces. http://www.almaden.ibm.com/cs/TSpaces/.