Corona: A High Performance Publish-Subscribe System for the World Wide Web

Venugopalan Ramasubramanian Ryan Peterson Emin Gün Sirer
Cornell University, Ithaca, NY 14853
{*ramasv,ryanp,egs*}*@cs.cornell.edu*

Abstract

Despite the abundance of frequently changing information, the Web lacks a publish-subscribe interface for delivering updates to clients. The use of naïve polling for detecting updates leads to poor performance and limited scalability as clients do not detect updates quickly and servers face high loads imposed by active polling. This paper describes a novel publish-subscribe system for the Web called Corona, which provides high performance and scalability through optimal resource allocation. Users register interest in Web pages through existing instant messaging services. Corona monitors the subscribed Web pages, detects updates efficiently by allocating polling load among cooperating peers, and disseminates updates quickly to users. Allocation of resources for polling is driven by a distributed optimization engine that achieves the best update performance without exceeding load limits on content servers. Large-scale simulations and measurements from PlanetLab deployment demonstrate that Corona achieves orders of magnitude improvement in update performance at a modest cost.

1 Introduction

Even though Web content changes rapidly, existing Web protocols do not provide a mechanism for automatically notifying users of updates. The growing popularity of frequently updated content, such as Weblogs, collaboratively authored Web pages (wikis), and news sites, motivates a *publish-subscribe* mechanism that can deliver updates to users quickly and efficiently. This need for asynchronous update notification has led to the emergence of *micronews syndication* tools based on naïve, repeated polling. The wide acceptance of such micronews syndication tools indicates that backwards compatibility with existing Web tools and protocols is critical for rapid adoption.

However, publish-subscribe through uncoordinated polling, similar to the current micronews syndication, suffers from poor performance and scalability. Subscribers do not receive updates quickly due to the fundamental limit posed by the polling period, and 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 subscribers, each polling independently and repeatedly for the same content. Moreover, such a workload tends to be "sticky;" that is, users subscribed to popular content tend not to unsubscribe after their interest diminishes, causing a large amount of wasted bandwidth. Existing micronews syndication systems provide ad hoc, stop-gap measures to alleviate these problems. Content providers currently impose hard rate-limits based on IP addresses, which render the system inoperable for users sharing an IP address, or they provide hints for when not to poll, which are discretionary and imprecise. The fundamental problem is that an architecture based on naïve, uncoordinated polling leads to ineffective use of server bandwidth.

This paper describes a novel, decentralized system for detecting and disseminating Web page updates. Our 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. Corona enables users to subscribe for updates to any existing Web page or micronews feed, and delivers updates asynchronously. The key contribution that enables such a general and backwards-compatible system is a distributed, peer-to-peer, cooperative resource management framework that can determine the optimal resources to devote to polling data sources in order to meet system-wide goals.

The central resource-performance tradeoff in a publish-subscribe system in which publishers 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 micronews feed or Web page. 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 a mathematical optimization problem. The system then computes the optimal way to allocate bandwidth to monitored Web objects using a decentralized algorithm that works on top of a distributed peer-to-peer overlay. This allocation takes object popularity, update rate, content size, and internal system overhead stemming from accounting and dissemination of meta-information into account, and yields a polling schedule for different objects that will achieve global performance goals, subject to resource constraints. Corona can optimize the system for different performance goals and resource limits. In this paper, we examine two relevant goals: how to minimize update latency while ensuring that the average load on publishers is no more than what it would have been without Corona, and how to achieve a targeted update latency while minimizing bandwidth consumption. We also examine variants of these two main approaches where the load is more fairly balanced across objects.

The front-end client interface to Corona is through existing instant messaging (IM) services. Users subscribe for content by sending instant messages to a registered Corona IM handle, and receive update notifications asynchronously. Internally, Corona consists of a cloud of nodes that monitor the set of active feeds or Web pages called *channels*. The Corona resource allocation algorithm determines the number of nodes designated to monitor each channel. Cooperative polling ensures that the system can detect updates quickly while no single node exceeds server-designated limits on polling frequency. Each node dedicated to monitoring a channel has a copy of the latest version of the channel contents. A feed-specific *difference engine* determines whether detected changes are germane by filtering out superficial differences such as timestamps and advertisements, extracts the relevant portions that have changed, and distributes the delta-encoded changes to all internal nodes assigned to monitor the channel, which in turn distribute them to subscribed clients via IM.

We have implemented a prototype of Corona and deployed it on PlanetLab. Evaluation of this deployment shows that Corona achieves more than an order of magnitude improvement in update performance. In experiments parameterized by real RSS workload collected at Cornell [19] and spanning 60 PlanetLab nodes and involving 150,000 subscriptions for 7500 different channels, Corona clients see fresh updates in intervals of 45 seconds on average compared to legacy RSS clients, which see a mean update interval of 15 minutes. At all times during the experiment, Corona limits the total polling load on the content servers within the load imposed by the legacy RSS clients.

Overall, Corona is a new overlay-based publish-subscribe system for the Web that provides asynchronous notifications, fast update detection, and optimal bandwidth utilization. This paper makes three contributions: (i) it outlines the general design of a publish-subscribe system that does not require any changes to content sources, (ii) formalizes the tradeoffs as an optimization problem and presents a novel distributed numerical solution technique for determining the allocation of bandwidth that will achieve globally targeted goals while respecting resource limits, and (iii) presents results from extensive simulations and a live deployment that demonstrate that the system is practical.

The rest of the paper is organized as follows. The next section provides background on publish-subscribe systems and discusses other related work. Section 3 describes the architecture of Corona in detail. Implementation details are presented in Section 4 and experimental results based on simulations and deployment are described in Section 5. Finally, Section 6 summarizes our contributions and concludes.

2 Background and Related Work

Publish-subscribe systems have raised considerable interest in the research community over the years. In this section, we provide background on publish-subscribe based content distribution and summarize the current state of the art.

Publish-Subscribe Systems: The 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, pub-sub systems can be classified as *topic-based* or *content-based*. In topic-based systems, publishers and subscribers are connected together by predefined topics, called *channels*; content is published on well-advertised channels to which users subscribe to 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.

Prior research on pub-sub systems has primarily focused on the design and implementation of content filtering and event delivery mechanisms. Topic-based publish-subscribe systems have been built based on several decentralized mechanisms, such as group communication in Isis [13], shared object spaces in Linda [5], TSpace [36], and Java Spaces [16] and rendezvous points in TIBCO [35] and Herald [4]. Content-based publish-subscribe systems that use in-network content filtering and aggregation include SIENA [6], Gryphon [34], Elvin [32], and Astrolabe [37]. While the above publish-subscribe systems impose well-defined structures for the content, few systems have been proposed for semi-structured and unstructured content. YFilter [8], Quark [3], XTrie [7], and XTreeNet [11] are recent architectures for supporting complex content-based queries on semi-structured XML data. Conquer [21] and WebCQ [20] support unstructured Web content.

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 layout middle-boxes in the core of the Internet. On the other hand, Corona interoperates with the current pull-based Web architecture, requires no changes to legacy Web servers, and provides an easy-to-use IM based interface to the users. Optimal resource management in Corona aimed at bounding network load insulates Web servers from high load during flash-crowds.

Micronews Systems: Micronews feeds are short descriptions of frequently updated information, such as news stories and blog updates, in XML-based formats such as RSS [30] and Atom [1]. 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 model of content dissemination and define XML tags such as *cloud* that tell clients how to receive asynchronous updates, as well as *TTL*, *SkipHours*, and *SkipDays* that inform clients when not to poll. Yet few content providers currently use the *cloud* tag to deliver asynchronous updates.

Recently, commercial services such as Bloglines, NewsGator, and Queoo have started disseminating micronews updates to users. Corona differs fundamentally from these commercial services, which use fragile centralized servers and relentless polling to detect updates. Corona is layered on a self-organizing overlay comprised of cooperative peers that share updates, judiciously determine the amount of bandwidth consumed by polling, and can provide strong bandwidth guarantees.

FeedTree [31] is a recently proposed system for disseminating micronews that also uses a structured overlay and shares updates between peers. FeedTree nodes perform cooperative update detection in order to reduce update dissemination latencies, and Corona shares the insight with FeedTree that cooperative polling can drastically reduce update latencies. FeedTree decides on the number of nodes to dedicate to polling each channel based on heuristics. Corona's key contribution is the use of informed tradeoffs to optimal resource management. 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.

CAM [26] and WIC [25] are two techniques for allocating bandwidth for polling data sources on a single node. Similar to Corona, they perform resource allocation using analytical models for the tradeoff and numerical techniques to find near-optimal allocations. However, these techniques are limited to a single node. Corona performs resource allocation in a decentralized, cooperative environment and targets globally optimal update performance.

Overlay Networks: Corona is layered on structured overlays and leverages the underlying structure to facilitate optimal resource management. Recent years have seen a large number of structured overlays that organize the network based on rings [33, 29, 39, 23], hyper-dimensional cubes [28], butterfly structures [22], de-Bruijn graphs [17, 38], or skip-lists [14]. Corona is agnostic about the choice of the overlay and can be easily layered on any overlay with uniform node degree, including the ones listed here.

Corona's approach to a peer-to-peer resource management problem has a similar flavor to that of Beehive [27], a structured replication framework that resolves space-time tradeoffs optimizations in structured overlays. Corona differs fundamentally from Beehive in three ways. First, the Beehive problem domain is limited to object replication in systems where objects have homogeneous popularity, size, and update rate properties, whereas Corona is designed for the Web environment where such properties can vary by several orders of magnitude between objects [10, 19]. Thus, Corona takes object characteristics into account during optimization. Second, the more complex optimization problem renders the Beehive solution technique, based on mathematical derivation, fundamentally unsuitable for the problem tackled by Corona. Hence, Corona employs a more general and sophisticated numerical algorithm to perform its optimizations. Finally, the resource-performance tradeoffs that arise in Corona are fundamentally different from the tradeoffs that Beehive addresses.

3 Corona

Corona (Cornell Online News Aggregator) is a topic-based publish-subscribe system for the Web. 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

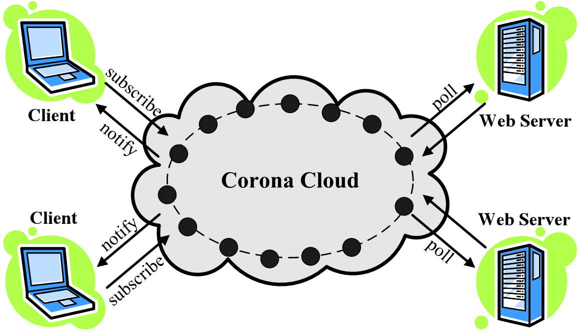

Figure 1: **Corona Architecture: Corona is a distributed publish-subscribe system for the Web. It detects Web updates by polling cooperatively and notifies clients through instant messaging.**

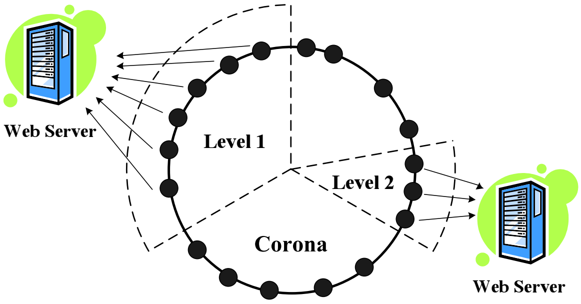

Figure 2: **Cooperative Polling in Corona: Each channel is assigned a wedge of nodes to poll the content servers and detect updates. Corona determines the optimal wedge size for each channel through analysis of the global performance-overhead tradeoff.**

on registered channels by cooperatively polling the content servers from geographically distributed nodes.

We envision 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 Akamai, or consist of server-class nodes contributed by participating institutions. By participating in Corona, institutions can significantly reduce the network bandwidth consumed by frequent redundant polling for content updates, as well as reduce the peak loads seen at content providers that they themselves may host. Corona nodes self-organize to form a structured overlay system. We use structured overlays to organize the system, as they provide decentralization, good failure resilience, and high scalability [33, 29, 39, 28, 9, 14, 17, 23, 24, 38]. Figure 1 illustrates the overall architecture of Corona.

The key feature that enables Corona to achieve fast update detection is *cooperative polling*. Corona assigns multiple nodes to periodically poll 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. While it is tempting to take the maximum advantage of cooperative polling by having every Corona node poll for every feed, such a naïve approach is clearly unscalable and would impose substantial network load on both Corona and content servers.

Corona makes informed decisions on distributing polling tasks among nodes. The number of nodes that poll for each channel is determined based on an analysis of the fundamental tradeoff between update performance and network load. Corona poses this tradeoff as an optimization problem and obtains the optimal solution using Honeycomb, a light-weight toolkit for computing optimal performance-overhead tradeoffs in structured distributed systems. This principled approach enables Corona to efficiently resolve the tradeoff between performance and scalability.

In this section, we provide detailed descriptions of the components of Corona's architecture, including the analytical models, optimization framework, update detection and notification mechanisms, and user interface.

3.1 Analytical Modeling

Our analysis-driven approach can be easily applied on any distributed system organized as a structured overlay with uniform node degree. In this paper, we describe Corona using Pastry [29] as the underlying substrate.

Pastry organizes the network into a ring by assigning identifiers from a circular numeric space to each node. The identifiers are treated as a sequence of digits of base b. In addition to neighbors along the ring, each node maintains contact with nodes that have matching prefix digits. These long-distance contacts are represented in a tabular structure called a *routing table*. The entry in the i^{th} row and j^{th} column of a node's routing table points to a node whose identifier shares i prefix digits and whose $(i+1)^{th}$ digit is j. Essentially, the routing table defines a directed acyclic graph (DAG) rooted at each node, enabling a node to reach any other node in $\log_b N$ hops.

Corona assigns nodes in well-defined wedges of the Pastry ring for polling each channel. Each channel is assigned an identifier from the same circular numeric space. A wedge is as set of nodes sharing a common number of prefix digits with a channel's identifier. A channel with *polling level* l is polled by all nodes with at least l matching prefix digits in their identifiers. Thus, polling level 0 indicates that all the nodes in the system poll for the channel. Figure 2 illustrates the concept of polling levels in Corona.

Assigning well-defined portions of the ring to each channel enables Corona to manage polling efficiently with little overhead. The set of nodes polling for a channel can be represented by just a single number, the polling level, eliminating the expensive $O(n)$ complexity for managing state about cooperating nodes. Moreover, this also facilitates efficient update sharing, as a wedge is a subset of the DAG rooted at each node, and all the nodes in a wedge can be reached quickly using the contacts in the routing table.

The polling level of a channel quantifies its performance-overhead tradeoff. A channel at level l has, on average, $\frac{N}{b^l}$ nodes polling it, which can cooperatively detect updates in $\frac{\tau}{2}\frac{b^l}{N}$ time on average, where τ is the polling interval. We estimate the average update detection time at a single node polling periodically at an interval τ to be $\frac{\tau}{2}$. Simultaneously, the collective load placed on the content server of the channel is $\tau\frac{N}{b^l}$. Note that we do not include the propagation delay for sharing updates in this analysis because updates can be detected by comparing against any old version of the content. Hence, even if an update detected at a different node in the system is received late, the time to detect the next update at the current node does not change.

An easy way to set polling levels is to independently choose a level for each channel based on these estimates. However, such an approach involves investigating heuristics for determining the appropriate performance requirement for each channel and for dividing the total load between different channels. Moreover, it does not provide fine-grained control over the performance of the system, often causing it to operate far from optimally. The rest of this section describes how the tradeoffs can be posed as mathematical optimization problems to achieve different performance requirements.

Corona-Lite: The first performance goal we explore is minimizing 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 they would handle from the clients if the clients fetched the objects directly from the servers.

The optimization problem for Corona-Lite is defined in Table 1. The overall update performance is measured by taking an average of the update detection time of each channel weighted by the number of clients subscribed to the channels. 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 optimization is simply the total number of subscriptions in 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 does not suffer from "diminishing returns," using its surplus polling capacity on less popular channels where the extra bandwidth yields higher marginal benefit. Since improvement in update performance is inversely proportional to the number of polling nodes, a naïve 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, improving their update detection times and achieving a better global average.

Corona-Lite:

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

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 target average update detection time, while minimizing the load placed on content servers.

Corona-Fair:

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

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

Corona-Fair-Sqrt:

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

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

Corona-Fair-Log:

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

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

Notation

τ	polling interval
M	number of channels
N	number of nodes
b	base of structured overlay
T	performance target
l_i	polling level of channel i
q_i	number of clients for channel i
s_i	content size for channel i
u_i	update interval for channel i

Table 1: **Performance-Overhead Tradeoffs: This table summarizes the optimization problems for different performance goals in Corona.**

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 previous 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 choose a target of 30 seconds to quickly detect changes to stock prices.

Corona-Fast shields legacy Web servers from sudden increases in load. A sharp 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. Moreover, the increased load typically continues unabated in legacy RSS as subscribers forget to unsubscribe, creating "sticky" traffic. Corona-Fast protects content servers from flash-crowds and sticky traffic.

Corona-Fair: Corona-Fast and Corona-Lite do not consider the actual rate of change of content in a channel. While some Web objects are updated every few minutes, others do not change for days at a time [10, 19]. 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 a modified update performance metric as the ratio of the update detection time and the update interval of the channel, which it minimizes to achieve a target load.

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 many subscribers for the channel. We compensate for this bias by exploring other update performance metrics based on square root and logarithmic functions, which grow sub-linearly. A sub-linear metric dampens the tendency of the optimization algorithm to punish slow-changing yet popular feeds. Table 1 summarizes the optimization problems for different versions of Corona.

3.2 Decentralized Optimization

Corona determines the optimal polling levels using the Honeycomb optimization toolkit. Honeycomb provides numerical algorithms and decentralized mechanisms for solving optimization problems that can be expressed as follows:

$$\text{min.} \sum_{1}^{M} f_i(l_i) \quad \text{s.t. } \textstyle\sum_1^M g_i(l_i) \leq T.$$

Here, $f_i(l)$ and $g_i(l)$ can define the performance or the cost for channel i as a function of the polling level l.

The preceding optimization problem is NP-Hard, as the polling levels only take integral values. Hence, instead of using computationally intensive techniques to find the exact solution, Honeycomb finds an approximate solution quickly in time comparable to a sorting algorithm. Honeycomb's optimization algorithm runs in $O(M \log M \log N)$ time.

The solution provided by Honeycomb is accurate and deviates from the optimal in at most one channel. Honeycomb achieves this accuracy by finding two solutions that optimize the problem with slightly altered constraints: one with a constraint $T_d \leq T$ and another with constraint $T_u \geq T$. The corresponding solutions L_u^* and L_d^* are exactly optimal for the optimization problems with constraints T_u and T_d respectively, and differ in at most one channel. That is, one channel has a different polling level in L_u^* than in L_d^*. Note that the optimal solution L^* for the original problem with constraint T may actually decide to allocate channels differently from L_d^* and L_u^*. Yet, the minimum determined by L^* will be bounded by the minima determined by L_d^* and L_u^* due to monotonicity. Honeycomb then chooses L_d^* as the final solution because it satisfies the constraint T strictly.

Honeycomb computes L_d^* and L_u^* using a Lagrange multiplier to transform the optimization problem as follows:

$$L^* \quad = \quad \text{arg min.} \sum_{1}^{M} f_i(l_i) - \lambda[\sum_{1}^{M} g_i(l_i) - T].$$

Honeycomb iterates over λ and obtains the two solutions L_d^* and L_u^* that bracket the minimum using standard bracketing methods for function optimization in one dimension.

Two observations enable Honeycomb to speed up the optimization algorithm. First, $L^*(\lambda')$ for a single iteration can be computed by finding arg min.$f_i(l_i) - \lambda' g_i(l_i)$ independently for each channel. This takes $O(M \log N)$ time as the number of levels is bounded by $\lceil \log N \rceil$. Second, for each channel there are only $\log N$ values of λ that change arg min.$f_i(l_i) - \lambda' g_i(l_i)$. Pre-computing these λ values for each object provides a discrete iteration space of $M \log N$ λ values. By keeping a sorted list of the λ values, Honeycomb computes the optimal solution in $O(\log M)$ iterations. Overall, the run-time complexity of the optimization algorithm is $O(M \log M \log N)$ time, including the time spent in pre-computation, sorting, and iterations.

The preceding algorithm requires the tradeoff functions $f_i(l)$ and $g_i(l)$ of all channels in the system in order to compute the global optimum. Solving the optimization problem using limited data available locally can produce highly inaccurate solutions. On the other hand, collecting the tradeoff factors for all the channels at each node is clearly expensive and impractical. It is possible to gather the tradeoff data at a central node, run the optimization algorithm at a single location, and then distribute the optimal levels to peers from the central location. We avoid using a centralized infrastructure as it introduces a single point of failure in the system and has limited scalability.

Instead, Honeycomb internally aggregates coarse grained information about global tradeoff factors. It combines channels with similar tradeoff factors into a *tradeoff cluster*. Each cluster summarizes the tradeoff factors for multiple channels and provides coarse-grained tradeoff information. A ratio of performance and cost factors, f_i/g_i, is used as a metric to combine channels. For example, channels with comparable values for $\frac{q_i}{u_i s_i}$ are combined into a cluster in Corona-Fair.

Honeycomb nodes periodically exchange the clusters with contacts in the routing table and aggregate the clusters received from the contacts. Honeycomb keeps the overhead for cluster aggregation low by limiting the number of clusters for each polling level to a constant *Tradeoff_Bins*. Each node receives Tradeoff_Bins clusters for every polling level from each contact in the routing table. Combined, these clusters summarize the tradeoff characteristics of all the channels in the system. The cluster aggregation overhead in terms of memory state as well as network bandwidth is limited by the size of the routing table, and scales with the logarithm of the system size.

3.3 System Management

Corona is a completely decentralized system, in which nodes act independently, share load, and achieve globally optimal performance through mutual cooperation. Corona spreads load uniformly among the nodes through consistent-hashing [18]. 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 a channel is the Corona node with the numerically closest identifier to the channel's identifier. Corona adds 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. Owners receive subscriptions through the underlying overlay, which automatically routes all subscription requests of a channel to its owner. The owners 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 update channel content. The latter is estimated based on the time between updates detected by Corona.

Corona manages cooperative polling through a periodic protocol consisting of an *optimization phase*, a *maintenance phase*, and an *aggregation phase*. In the optimization phase, Corona nodes apply the optimization algorithm on fine-grained tradeoff data for locally polled channels and coarse-grained tradeoff clusters obtained from overlay contacts. In the maintenance phase, changes to polling levels are communicated to peer nodes in the routing table through *maintenance messages*. Finally, the aggregation phase enables nodes to receive new aggregates of tradeoff factors. In practice, the three phases occur concurrently at a node with aggregation data piggy-backed on maintenance messages.

Corona nodes operate independently and make decisions to increase or decrease polling levels locally. Initially, only the owner nodes at level $K = \lceil \log N \rceil$ poll for the channels. If an owner decides to lower the polling level to $K - 1$ (based on local optimization), it sends a message to the contacts in its routing table at row $K-1$ in the next *maintenance phase*. As a result, a small wedge of level $K - 1$ nodes start polling for that channel. Subsequently, each of these nodes may independently decide to further lower the polling level of that channel. Similarly, if an owner node decides to raise the level from $K - 1$ to K, it asks its contact in the $K - 1$ wedge to stop polling.

In general, when a level i node lowers the level to $i-1$ or raises the level from $i - 1$ back to i, it instructs its contact in row $i - 1$ of its routing table to start or stop polling for that channel. This control path closely follows the DAG rooted at the owner node. Nodes at level i (depth $K - i$) in this DAG decide whether their children at level $i - 1$ should poll a channel and convey these decisions periodically every *maintenance interval*. When a node is instructed to begin polling for a channel, it waits for a random interval of time between 0 and the polling interval before the first poll. This ensures that polls for a channel at different nodes are randomly distributed over time.

Corona nodes gather current estimates of tradeoff factors in the aggregation phase. Owners monitor the number of subscribers and send out fresh estimates along with the maintenance message. Subsequent maintenance messages sent out by descendant nodes in the DAG propagate these estimates to all the nodes in the wedge. The update interval and size of a feed only change during updates and are therefore sent along with updates. Tradeoff clusters are also sent by contacts in the routing table in response to maintenance messages.

Corona inherits robustness and failure-resilience from the underlying structured overlay. If the current contact in the routing table fails, the underlying overlay automatically replaces it with another contact. When new nodes join the system or nodes fail, Corona ensures the transfer of subscription state to new owners. A node that is no longer an owner simply erases its subscription state, and a node that becomes a new owner receives the state from other owners of the channel. Simultaneous failure of more than F adjacent nodes poses a problem for Corona, as well as to many other peer-to-peer systems; we assume that F is chosen to make such an occurrence rare. Note that clients can easily renew subscriptions should a catastrophic failure lose some subscription state.

Overall, Corona manages polling using light-weight mechanisms that impose a small, predictable overhead on the nodes and network. Its algorithms do not rely on expensive constructs such as consensus, leader election, or clock synchronization. Networking activity is limited to contacts in the nodes' routing tables.

3.4 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 node assigns version numbers in increasing order based on the updates it receives.

Corona nodes share updates only as *deltas*, the differences between old and new content, rather than the entire content. A measurement study on micronews feeds conducted at Cornell shows that the amount of change in content during an update is typically tiny. The study reports that the average update consists of 17 lines of XML, or 6.8% of the content size [19], which implies that a significant amount of bandwidth can be saved through delta-encoding.

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 delta if it detects an update after isolating the core content. The data in a delta resembles the typical output of the POSIX 'diff' command: it carries the line numbers where the change occurs, the changed content, an indication of whether it is an addition, omission, or replacement, and a version number of the old content to compare against.

When a delta is generated by a node, it shares the update with all other nodes in the channel's polling wedge. To achieve this, the node simply disseminates the delta 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 delta 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 deltas to the primary owner. The primary owner always checks the current delta with the latest updated version of the content and ignores redundant deltas.

3.5 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 communicated 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 delta to all the subscribers through the IM system. If a subscriber is offline 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 incurs additional latency since messages are sent through a centralized service. However, the additional latency is modest as IM systems are designed to reduce such latencies during two-way communication. Moreover, IM systems that allow peer-to-peer communication between their users, such as Skype, deliver messages in quick time.

Instant messaging enables Corona to be easily accessible to a large user population, as no computer skills other than an ability to "chat" are required, and ubiquitous IM deployment ensures that hosts behind NATs and firewalls are supported. 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.

4 Implementation

We have implemented a prototype of Corona as an application layered on Pastry [29], a prefix-matching structured overlay system. The implementation uses a 160-bit SHA-1 hash function to generate identifiers for both the nodes (based on their IP addresses) and channels (based on their URLs). Both the base of Pastry and the number of tradeoff clusters per polling level are set to 16.

Prefix matching overlays occasionally create *orphans*, that is, channels with no owners having enough matching prefix digits. Orphans are created when the required wedge of the identifier space, corresponding to level $\lceil \log N \rceil - 1$, is empty. Corona cannot assign additional nodes to poll an orphan channel to improve its update performance. Moreover, orphans adversely affect the computation of globally optimal allocation. Corona handles orphan channels by adjusting the tradeoffs appropriately. The tradeoff factors of orphan channels are aggregated into a *slack cluster*, which is used to adjust the performance target prior to optimization.

Corona's reliance on IM as an asynchronous communication mechanism poses some operational challenges. Corona interacts with IM systems using GAIM [12], an open source instant messaging client for Unix-based platforms that supports multiple IM systems including Yahoo Instant Messenger, AOL Instant Messenger, and MSN Messenger. Several IM systems have a limitation that only one instance of a user can be logged on at a time, preventing all Corona nodes from being logged on simultaneously. While we hope that IM systems will support simultaneous logins from automated users such as Corona in the near future, as they have for several chat robots, our implementation uses a centralized server to talk to IM systems as a stop-gap measure. This server acts as an intermediary for all updates sent to clients as well as subscription messages sent by clients. Also, IM systems such as Yahoo rate-limit instant messages sent by unprivileged clients. Corona's implementation limits the rate of updates sent to clients to avoid 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 their content. An alternative solution that does not require changes to servers is to use threshold-cryptography to generate a certificate for content [40, 15]. The responsibility for generating partial signatures can be shared among the owners of a node ensuring that rogue nodes below the threshold level cannot corrupt the system. Designing and implementing such a threshold-cryptographic scheme is, however, beyond the scope of this paper.

5 Evaluation

We evaluate the performance of Corona through large-scale simulations and wide-area experiments on PlanetLab [2]. In all our evaluations, we compare the performance of Corona to the performance of legacy RSS, a widely-used micronews syndication system. The simulations and experiments are driven by real-life RSS traces.

We collected characteristics of micronews workloads and content by passively logging user activity and actively polling RSS feeds [19]. User activity recorded between March 22 and May 3 of 2005 at the gateway of the Cornell University Computer Science Department provided a workload of 158 clients making approximately 62,000 requests for 667 different feeds. The channel popularity closely follows a Zipf distribution with exponent 0.5. The survey analyzes the update rate of micronews content by actively polling approximately 100,000 RSS feeds obtained from *syndic8.com*. We poll these feeds at one hour intervals for 84 hours, and subsequently select a subset of 1000 feeds and poll them at a finer granularity of 10 minutes for five days. Comparing periodic snapshots of the feeds shows that the update interval of micronews content is widely distributed: about 10% of channels changed within an hour, while 50% of channels did not change at all during the five days of polling.

5.1 Simulations

We use tradeoff parameters based on the RSS survey in our 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. We use a distribution for the update rates of channels 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, 100,000 channels, and 5,000,000 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 3 and 4 show the network load and update performance, respectively, 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 3 shows that Corona-Lite stabilizes at the load imposed by legacy RSS clients. Starting from a clean slate where only owner

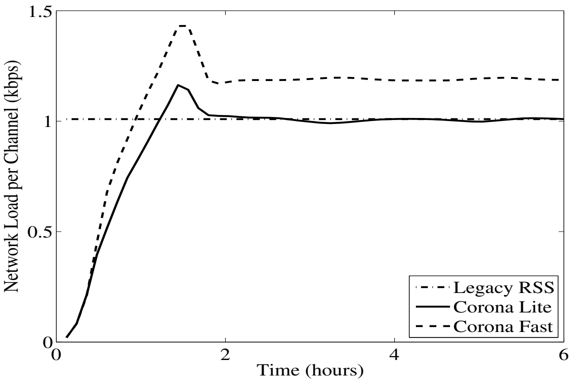

Figure 3: **Network Load on Content Servers: Corona-Lite converges quickly to match the network load imposed by legacy RSS clients.**

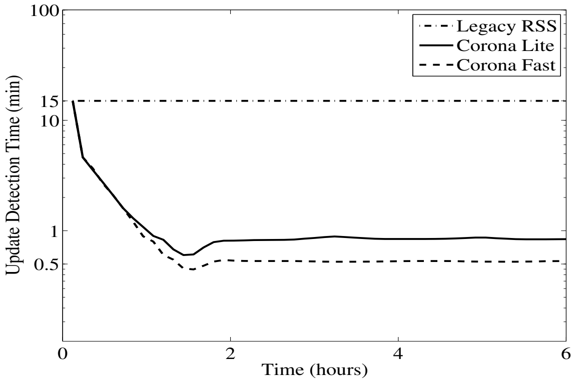

Figure 4: **Average Update Detection Time: Corona-Lite provides 15-fold improvement in update detection time compared to legacy RSS clients for the same network load.**

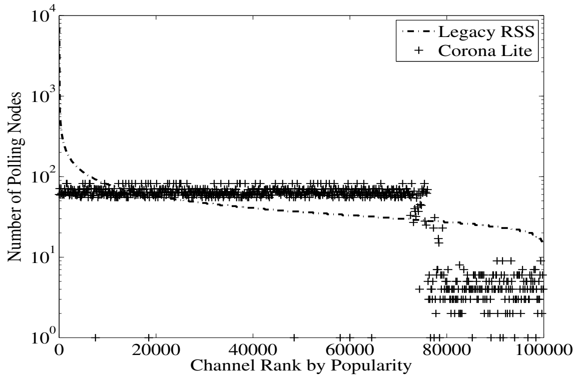

Figure 5: **Number of Pollers per Channel: Corona trades off network load from popular channels to decrease update detection time of less popular channels and achieve a lower system-wide average.**

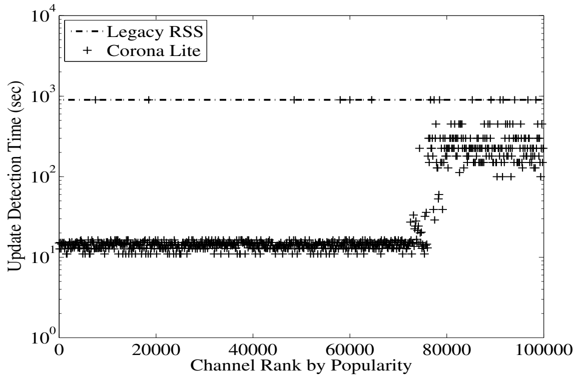

Figure 6: **Update Detection Time per Channel: Popular channels gain greater decrease in update detection time than less popular channels.**

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 tradeoff factors of other channels in the system. However, the discrepancy is corrected automatically when aggregated global tradeoff factors are available to each node.

At the same time, Figure 4 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 and 6 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: channels clustered around 100 at level 1, channels with fewer 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 75,000 channels indicates the point where the optimal solution changes polling levels.

Figure 5 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 of the over-loaded servers and transfers the extra load to servers of less popular content to improve update performance.

Scheme	Average Update Detection Time (sec)	Average Load (polls per 30 min per channel)
Legacy-RSS	900	50.00
Corona-Lite	53	48.97
Corona-Fair	142	50.14
Corona-Fair-Sqrt	55	49.46
Corona-Fair-Log	53	49.43
Corona-Fast	32	58.75

Table 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 placing the same load on servers.**

The favorable behavior of Corona-Lite is due to diminishing returns caused by 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 to devote a large percentage of the bandwidth to polling the most popular channels. 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). The upshot is that popular channels gain an order of magnitude improvement in update performance over less popular ones (see Figure 6).

Corona-Fast

Unlike Corona-Lite, Corona-Fast minimizes the total load on servers while aiming to achieve a target update detection latency. Figures 3 and 4 show the network load and update performance, respectively, for Corona-Fast. Figure 4 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 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 the update rates of channels to fine-tune the distribution of load. It takes advantage of the wide distribution of update intervals among channels and aims to poll frequently updated channels at a higher rate than channels with long update intervals. Figure 7 shows the distribution of update detection times achieved by Corona-Lite and Corona-Fair for different channels ranked by their update intervals. Channels with the same update intervals are further ranked by popularity. For clarity of presentation, we plot the distribution for 200 uniformly chosen channels.

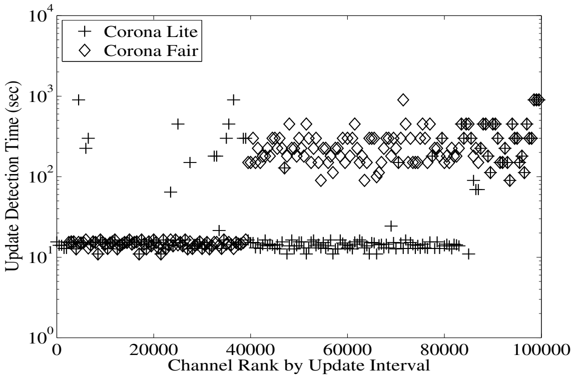

Figure 7: **Update Detection Time per Channel: Corona-Fair provides better update detection time for channels that change rapidly than for channels that change rarely.**

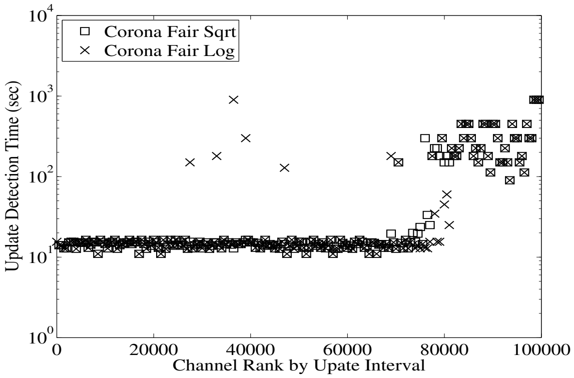

Figure 8: **Update Detection Time per Channel: Corona-Fair-Sqrt and Corona-Fair-Log fix the bias against channels that change rarely and provide better update detection time for them than Corona-Fair does.**

Figure 7 shows that Corona-Lite achieves an unfair distribution of update detection times by ignoring update interval information. Many channels with large update intervals have short update detection times (shown in the lower-right of the graph), while some rapidly changing channels have long update detection times (shown in the upper-left of the graph). Corona-Fair fixes this unfair distribution of update detection time by using update intervals of channels to influence the choice of polling levels. Figure 7 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 times 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 proportionally long update detection times, leading to long wait times for clients. Section 3.1 proposed to compensate for this bias using two metrics with sub-linear growth based on the square root and logarithm of the update interval. Figure 8 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, which has a few channels with small update intervals but long update detection times.

Overall, the Corona-Fair schemes provide fair distributions of polling between channels with low average update detection times without exceeding bandwidth load on the servers. The average update detection time and load for different Corona-Fair schemes is shown in Table 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.

5.2 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 7500 real channels providing RSS feeds obtained from *www.syndic8.com*. We issue 150,000 subscriptions for them based on a Zipf popularity distribution with exponent 0.5. Subscriptions are issued at a uniform rate during the first hour and a half of the experiment. The maintenance interval and the polling interval are both set to 30 minutes. We collected data for a period of six hours.

Figure 9 shows the average update detection time for Corona deployment compared to legacy RSS. Corona decreases the average update time to about 45 seconds compared to 15 minutes for legacy RSS. Figure 10 shows the corresponding polling load imposed by Corona on content servers. Corona gradually increases the number of nodes polling each channel and reaches a load limit of around 4500 polls per minute. Corona's total network load is bounded by the load imposed by legacy RSS, which averages to just above 5000 polls per minute. These graphs highlight that while imposing comparable load as legacy RSS, Corona achieves a substantial improvement in update detection time.

5.3 Summary

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 tradeoff factors of all channels, and uses the new parameters to periodically adjust the optimal polling levels of channels and meet performance and load targets. Corona offers considerable flexibility in the kind of performance goals it can achieve. In this section, 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.

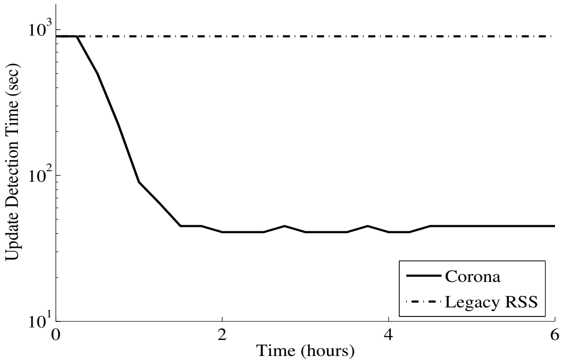

Figure 9: **Average Update Detection Time: Corona provides an order of magnitude lower update detection time compared to legacy RSS.**

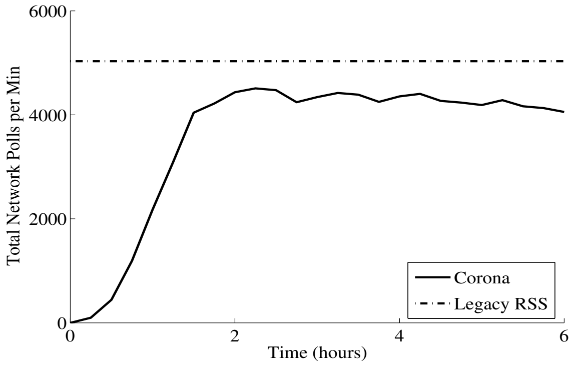

Figure 10: **Total Polling Load on Servers: The total load generated by Corona is well below the load generated by clients using legacy RSS**

6 Conclusions

This paper 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 paucity of infrastructure to provide asynchronous notifications, we develop a unique solution that addresses the shortcomings of pull based content dissemination and delivers a real, deployable, easy-to-use publish-subscribe system.

Many real-world applications require quick and efficient dissemination of information from data sources to clients. It is quite common for legacy data sources, such as Web pages, sensors, stock feeds, event trackers and so forth, to be deployed piecemeal, and thus to force clients to poll them manually and explicitly to receive the latest updates. As the numbers of data sources increase, the task of monitoring so many event sources quickly becomes overwhelming for humans. At sufficiently large scales, the task of allocating bandwidth is difficult even for computers. We can see examples of such applications in large scale sensor networks, in investment management systems that track commodity prices, and in many adaptive distributed systems for detecting events. All of these applications pose a fundamental tension between the polling bandwidth required to achieve fast event detection and the corresponding load imposed by periodic polling.

Our unique contribution is the optimal resolution of performance-overhead tradeoffs in such event detection systems. This paper provides a general approach based on analytical modeling of the cost-performance tradeoff and mathematical optimization that enables applications to make informed, near-optimal decisions on which data sources to monitor, and with what frequency. We develop techniques to solve typical resource allocation problems that arise in distributed systems through decentralized, low-overhead mechanisms.

The techniques at the core of this system are easily applicable to any domain where a set of nodes monitor exogenous events. The Corona approach to monitoring oblivious, pull-based data sources makes it unnecessary to change the data publishing workflow, agree on new dissemination protocols, or deploy new software on data sources. This is particularly relevant when the sources to be monitored are large in number, and deploying new software is logistically difficult. For instance, large scale Web spiders that monitor changes to Websites to incrementally update a Web index could benefit from the principled approach developed here.

Corona applies this general approach to disseminating updates to the Web, where the resource-performance tradeoff is affected by the popularity, size, and update rate of Web content and the network capacities of clients and content servers. Performance measurements based on simulations and real-life deployment show that Corona clients can achieve several orders of magnitude improvement in update latency without an increase in average load. Corona acts as a buffer between clients and servers, shielding servers from the impact of flash-crowds and sticky traffic. Our implementation is currently deployed on PlanetLab and available for public use. We hope that a backwards-compatible, high-performance, efficient publish-subscribe system will make it possible for people to easily track frequently changing content on the Web.

Acknowledgements

We would like to thank Rohan Murty for implementing an earlier prototype of the Corona system, Yee Jiun Song for his help with the system, and our shepherd Dave Andersen for his guidance and feedback. This work was supported in part by NSF CAREER grant 0546568, and TRUST (The Team for Research in Ubiquitous Secure Technology), which receives support from the National Science Foundation (CCF-0424422) and the following organizations: Cisco, ESCHER, HP, IBM, Intel, Microsoft, ORNL, Qualcomm, Pirelli, Sun, and Symantec.

References

[1] Atom. Atom Syndication Format. http://www.-atomenabled.org/developers/syndication, Oct. 2005.

[2] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proc. of Symposium on Networked Systems Design and Implementation*, Boston, MA, Mar. 2004.

[3] C. Botev and J. Shanmugasundaram. Context Sensitive Keyword Search and Ranking for XML. In *Proc. of International Workshop on Web and Databases*, Baltimore, MD, June 2005.

[4] L. F. Cabera, M. B. Jones, and M. Theimer. Herald: Achieving a Global Event Notification Service. In *Proc. of Workshop on Hot Topics in Operating Systems*, Elmau, Germany, May 2001.

[5] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, Apr. 1989.

[6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.

[7] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proc. of International Conference on Data Engineering*, San Jose, CA, Feb. 2002.

[8] Y. Diao, S. Rizvi, and M. Franklin. Towards an Internet-Scale XML Dissemination Service. In *Proc. of International Conference on Very Large Databases (VLDB)*, Toronto, Canada, Aug. 2004.

[9] J. R. Douceur, A. Adya, W. J.Bolosky, and D. 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.

[10] F. Douglis, A. Feldman, B. Krishnamurthy, and J. 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, Dec. 1997.

[11] W. Fenner, M. Rabinovich, K. K. Ramakrishnan, D. Srivastava, and Y. 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, Sept. 2005.

[12] GAIM. A Multi-Protocol Instant Messaging Client. http://gaim.sourceforge.net, Oct. 2005.

[13] B. Glade, K. P. Birman, R. Cooper, and R. van Renesse. Light-Weight Process Groups in the ISIS System. *Distributed Systems Engineering*, 1(1):29–36, Sept. 1993.

[14] N. Harvey, M. Jons, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, Mar. 2003.

[15] W. K. Josephson, E. G. Sirer, and F. 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, Feb. 2004.

[16] JS-JavaSpaces Service Specification. http://www.jini.-org/nonav/standards/davis/doc/specs/html/js-spec.html, 2002.

[17] F. Kaashoek and D. Karger. Koorde: A Simple Degree-Optimal Distributed Hash Table. In *Proc. of International Workshop on Peer-to-Peer Systems*, Berkeley, CA, Feb. 2003.

[18] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. 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, Apr. 1997.

[19] H. Liu, V. Ramasubramanian, and E. G. Sirer. Client Behavior and Feed Characteristics of RSS, a Publish-Subscribe System for Web Micronews. In *Proc. of ACM Internet Measurement Conference*, Berkeley, CA, Oct. 2005.

[20] L. Liu, C. Pu, and W. Tang. WebCQ: Detecting and Delivering Information Changes on the Web. In *Proc. of the International Conference on Information and Knowledge Management*, McLean, VA, Nov. 2000.

[21] L. Liu, C. Pu, W. Tang, and W. 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.

[22] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proc. of ACM Symposium on Principles of Distributed Computing*, Monterey, CA, Aug. 2002.

[23] P. Maymounkov and D. 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, Mar. 2002.

[24] A. Mizrak, Y. Cheng, V. Kumar, and S. Savage. Structured Superpeer: Leveraging Heterogeneity to Provide Constant-time Lookup. In *Proc. of IEEE Workshop on Internet Applications*, San Francisco, CA, Apr. 2003.

[25] S. Pandey, K. Dhamdere, and C. Olston. WIC: A General-Purpose Algorithm for Monitoring Web Information Sources. In *Proc. of the Conference on Very Large Data Bases (VLDB)*, Toronto, Canada, Aug. 2004.

[26] S. Pandey, K. Ramamritham, and S. Chakraborti. Monitoring the Dynamic Web to Respond to Continuous Queries. In *Proc. of the International World Wide Web Conference*, Budapest, Hungary, May 2003.

[27] V. Ramasubramanian and E. G. 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, Mar. 2004.

[28] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. of ACM SIGCOMM*, San Diego, CA, Aug. 2001.

[29] A. Rowstron and P. 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, Nov. 2001.

[30] RSS 2.0 Specifications. http://blogs.law.harvard.edu/-tech/rss, Oct. 2005.

[31] D. Sandler, A. Mislove, A. Post, and P. Druschel. FeedTree: Sharing Web Micronews with Peer-to-Peer Event Notification. In *Proc. of International Workshop on Peer-to-Peer Systems*, Ithaca, NY, Feb. 2005.

[32] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin. In *Proc. of AUUG2K*, Canberra, Australia, June 2000.

[33] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM*, San Diego, CA, Aug. 2001.

[34] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An Information Flow Based Approach to Message Brokering. In *Proc. of International Symposium on Software Reliability Engineering*, Paderborn, Germany, Nov. 1998.

[35] TIBCO Publish-Subscribe. http://www.tibco.com, Oct. 2005.

[36] TSpaces. http://www.almaden.ibm.com/cs/TSpaces/, Oct. 2005.

[37] R. van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology For Distributed Systems Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21(3), May 2003.

[38] U. Wieder and M. Naor. A Simple Fault Tolerant Distributed Hash Table. In *Proc. of International Workshop on Peer-to-Peer Systems*, Berkeley, CA, Feb. 2003.

[39] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan. 2004.

[40] L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A Secure Distributed On-line Certification Authority. *ACM Transactions on Computer Systems*, 20(4):329–368, Nov. 2002.