Cobra: Content-based Filtering and Aggregation of Blogs and RSS Feeds

Ian Rose, Rohan Murty, Peter Pietzuch, Jonathan Ledlie,
Mema Roussopoulos, and Matt Welsh
School of Engineering and Applied Sciences
Harvard University
{ianrose,rohan,prp,jonathan,mema,mdw}@eecs.harvard.edu

Abstract

Blogs and RSS feeds are becoming increasingly popular. The blogging site LiveJournal has over 11 million user accounts, and according to one report, over 1.6 million postings are made to blogs every day. The "Blogosphere" is a new hotbed of Internet-based media that represents a shift from mostly static content to dynamic, continuously-updated discussions. The problem is that finding and tracking blogs with interesting content is an extremely cumbersome process.

In this paper, we present Cobra (Content-Based RSS Aggregator), a system that crawls, filters, and aggregates vast numbers of RSS feeds, delivering to each user a personalized feed based on their interests. Cobra consists of a three-tiered network of *crawlers* that scan web feeds, *filters* that match crawled articles to user subscriptions, and *reflectors* that provide recently-matching articles on each subscription as an RSS feed, which can be browsed using a standard RSS reader. We present the design, implementation, and evaluation of Cobra in three settings: a dedicated cluster, the Emulab testbed, and on PlanetLab. We present a detailed performance study of the Cobra system, demonstrating that the system is able to scale well to support a large number of source feeds and users; that the mean update detection latency is low (bounded by the crawler rate); and that an offline service provisioning step combined with several performance optimizations are effective at reducing memory usage and network load.

1 Introduction

Weblogs, RSS feeds, and other sources of "live" Internet content have been undergoing a period of explosive growth. The popular blogging cite LiveJournal reports over 11 million accounts, just over 1 million of which are active over the last month [2]. Technorati, a blog tracking site, reports a total of 50 million blogs worldwide; this number is currently doubling every 6 months [38]. In July 2006, there were over 1.6 million blog postings every day. These numbers are staggering and suggest a significant shift in the nature of Web content from mostly static pages to continuously updated conversations.

The problem is that finding interesting content in this burgeoning blogosphere is extremely difficult. It is unclear that conventional Web search technology is well-suited to tracking and indexing such rapidly-changing content. Many users make use of RSS feeds, which in conjunction with an appropriate reader, allow users to receive rapid updates to sites of interest. However, existing RSS protocols require each client to periodically poll to receive new updates. In addition, a conventional RSS feed only covers an individual site, such as a blog. The current approach used by many users is to rely on RSS aggregators, such as SharpReader and FeedDemon, that collect stories from multiple sites along thematic lines (e.g., news or sports).

Our vision is to provide users with the ability to perform *content-based filtering and aggregation* across millions of Web feeds, obtaining a *personalized* feed containing only those articles that match the user's interests. Rather than requiring users to keep tabs on a multitude of interesting sites, a user would receive near-real-time updates on their personalized RSS feed when matching articles are posted. Indeed, a number of "blog search" sites have recently sprung up, including Feedster, Blogdigger, and Bloglines. However, due to their proprietary architecture, it is unclear how well these sites scale to handle large numbers of feeds, vast numbers of users, and maintain low latency for pushing matching articles to users. Conventional search engines, such as Google, have recently added support for searching blogs as well but also without any evaluation.

This paper describes Cobra (Content-Based RSS Aggregator), a distributed, scalable system that provides users with a personalized view of articles culled from potentially millions of RSS feeds. Cobra consists of a three-tiered network of *crawlers* that pull data from web feeds, *filters* that match articles against user subscriptions, and *reflectors* that serve matching articles on each subscription as an RSS feed, that can be browsed using a standard RSS reader. Each of the three tiers of the Cobra network is distributed over multiple hosts in the Internet, allowing network and computational load to be balanced, and permitting locality optimizations when placing services.

The core contributions in this paper are as follows. First, Cobra makes use of a novel offline *service provisioning* technique that determines the minimal amount of physical resources required to host a Cobra network capable of supporting a given number of source feeds and users. The technique determines the configuration of the network in terms of the number of crawlers, filters, and reflectors, and the interconnectivity between these services. The provisioner takes into account a number of characteristics including models of the bandwidth and memory requirements for each service and models of the feed content and query keyword distribution.

Our second contribution is a set of optimizations designed to improve scalability and performance of Cobra under heavy load. First, crawlers are designed to intelligently filter source feeds using a combination of HTTP header information, whole-document and per-article hashing. These optimizations result in a 99.8% reduction in bandwidth usage. Second, the filter service makes use of an efficient text matching algorithm [18, 29] allowing over 1 million subscriptions to match an incoming article in less than 20 milliseconds. Third, Cobra makes use of a novel approach for assigning source feeds to crawler instances to improve network locality. We perform network latency measurements using the King [24] method to assign feeds to crawlers, improving the latency for crawling operations and reducing overall network load.

The third contribution is a full-scale experimental evaluation of Cobra, using a cluster of machines at Harvard, on the Emulab network emulation testbed, and on PlanetLab. Our results are based on measurements of 102,446 RSS feeds retrieved from `syndic8.com` and up to 40 million emulated user queries. We present a detailed performance study of the Cobra system, demonstrating that the system is able to scale well to support a large number of source feeds and users; that the mean update detection latency is low (bounded by the crawler rate); and that our offline provisioning step combined with the various performance optimizations are effective at reducing overall memory usage and network load.

2 Related Work

Our design of Cobra is motivated by the rapid expansion of blogs and RSS feeds as a new source of real-time content on the Internet. Cobra is a form of *content-based publish-subscribe system* that is specifically designed to handle vast numbers of RSS feeds and a large user population. Here, we review previous work in pub-sub systems, both traditional and peer-to-peer designs. The database community has also developed systems for querying large numbers of real-time streams, some of which are relevant to Cobra.

Traditional Distributed Pub-Sub Systems: A number of distributed *topic-based* pub-sub systems have been proposed where subscribers register interest in a set of specific topics. Producers that generate content related to those topics publish the content on the corresponding *topic channels* [12, 22, 27, 5, 6, 7] to which the users are subscribed and users receive asynchronous updates via these channels. Such systems require publishers and subscribers to agree up front about the set of topics covered by each channel, and do not permit arbitrary topics to be defined based on a user's specific interests.

The alternative to the topic-based systems are *content-based* pub-sub systems [39, 37, 13, 40]. In these systems, subscribers describe content attributes of interest using an expressive query language and the system filters and matches content generated by the publishers to the subscribers' queries. Some systems support both topic-based and content-based subscriptions [32]. For a detailed survey of pub-sub middleware literature, see [31].

Cobra differentiates itself from other pub-sub systems in two ways. First, distributed content-based pub-sub systems such as Siena [13] leave it up to the network administrator to choose an appropriate overlay topology of filtering nodes. As a result, the selected topology and the number of filter nodes may or may not perform well with a given workload and distribution of publishers and subscribers in the network. By providing a separate provisioning component that outputs a custom-tailored topology of processing services, we ensure that Cobra can support a targeted work load. Our approach to pub-sub system provisioning is independent from our application domain of RSS filtering and could be used to provision a general-purpose pub-sub system like Siena, as long as appropriate processing and I/O models are added to the service provisioner.

Second, Cobra integrates directly with existing protocols for delivering real-time streams on the Web — namely, HTTP and RSS. Most other pub-sub systems such as Siena do not interoperate well with the current Web infrastructure, for example, requiring publishers to change the way they generate and serve content, and requiring subscribers to register interest using private subscription formats. In addition, the filtering model is targeted at structured data conforming to a well-known schema, in contrast to Cobra's text-based queries on (relatively unstructured) RSS-based web feeds.

Peer-to-Peer Overlay Networks: A number of content-delivery and event notification systems have been developed using peer-to-peer overlay networks, where the premise is that these systems are highly dynamic and can contain a large number of nodes exhibiting high rates of churn. As we explain in Section 3, we do not envision running Cobra in a peer-to-peer setting where nodes are contributed by volunteers, but instead

assume the use of well-provisioned hosting centers, as is currently the norm for commercial Internet services. Nonetheless, it is worth describing some of the other design differences with these systems.

Corona [34] is a pub-sub system for the Web built on top of Pastry [35]. Users specify interest in specific URLs and updates are sent to users using instant message notifications. The main goal of Corona is to mitigate the *polling overhead* placed on monitored URLs, which is accomplished by spreading polling load among cooperating peers and amortizing the overhead of crawling across many users interested in the same URL. An informed algorithm determines the optimal assignment of polling tasks to peers to meet system-wide goals such as minimizing update detection time or minimizing load on content servers.

Corona is strictly concerned with allowing users to monitor an individual URL and focuses on the efficiency of the polling operation. Unlike Cobra, Corona does not permit an individual user to monitor a large number of web feeds simultaneously, nor specify content-based predicates on which content should be pushed to the user. Like Cobra, Corona can interoperate seamlessly with the current pull-based Web architecture.

A number of application-level multicast systems [36, 43] have been built using DHTs that construct an information dissemination tree with the multicast group members as leaf nodes. The resulting application-level multicast service is similar to the aforementioned topic-based pub-sub systems without content-based filtering. The multicast tree is formed by joining the routes from each subscriber node to a root node. In contrast, Cobra constructs an overlay topology using a service provisioning technique, taking into account the required resources to support a target number of source feeds and users.

Real-time Stream Querying: There has been much recent work from the database community on continuous querying of real-time data streams located at geographically dispersed data sources. These include Medusa [15], PIER [25], IrisNet [21], Borealis [14], and Stream-Based Overlay Networks [33]. These systems provide a general-purpose service for distributed querying of data streams, tend to assume a relational data model, and provide an elaborate set of operators to applications. Cobra, on the other hand, is specifically designed to filter and deliver relatively unstructured RSS feeds, provides a simple keyword-based query format to the user, and has models of the resource consumption of its crawler, filter, and reflector services used for provisioning.

Blog Search Engines: Recently, a number of "blog search engines" have come online, including Feedster, Blogdigger, Bloglines, IceRocket, and Technorati. Apart from Google and MSN's blog search services, most of these sites appear to be backed by small startup companies and little is known about how they operate. In particular, their ability to scale to large numbers of feeds and users, use network resources efficiently, and maintain low update latencies is unknown. In Section 4 we attempt to measure the update latency of several of these sites. As for MSN and Google, we expect these sites leverage the vast numbers of server resources distributed across many data centers to rapidly index updates to blogs. Although an academic research group cannot hope to garner these kinds of resources, by developing Cobra we hope to shed light on important design considerations and tradeoffs for this interesting application.

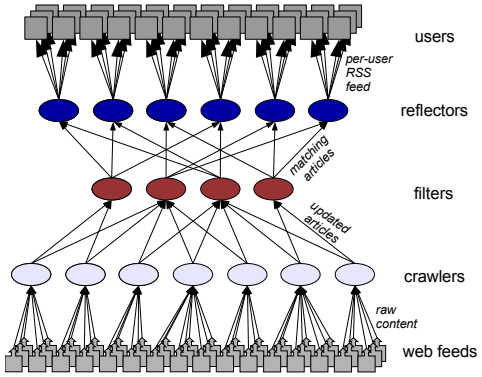


Figure 1: **The Cobra content-based RSS aggregation network.**

3 Cobra System Design

The overall architecture of Cobra is shown in Figure 1. Cobra consists of a three-tiered network of *crawlers*, *filters*, and *reflectors*. Crawlers are responsible for periodically crawling web feeds, such as blogs, news sites, and other RSS feeds, which we collectively call *source feeds*. A source feed consists of a series of *articles*. The number of articles provided by a source feed at any time depends on how it is configured; a typical blog or news feed will only report the most recent 10 articles or so. As described below, Cobra crawlers employ various techniques to reduce polling load by checking for updates in the source feeds in a lightweight manner.

Crawlers send new articles to the filters, which match the content of those articles against the set of user *subscriptions*, using a case-insensitive, index-based matching algorithm. Articles matching a given subscription are pushed to the appropriate reflector, which presents to the end user a personalized RSS feed that can be browsed using a standard RSS reader. The reflector caches the last k matching articles for the feed (where k is typically 10), requiring that the user poll the feed periodically to ensure that all matching articles will be detected. This behavior matches that of many existing RSS feeds that limit the number of articles included in the feed. Although the reflector must be polled by the user (as required by current

RSS standards), this polling traffic is far less than requiring users to poll many thousands of source feeds. Also, it is possible to replace or augment the reflector with push-based notification mechanisms using email, instant messaging, or SMS; we leave this to future work.

The Cobra architecture uses a simple congestion control scheme that applies backpressure when a service is unable to keep up with the incoming rate of data from upstream services. Each service maintains a 1MB data buffer for each upstream service. If an upstream service sends data faster than it can be processed, the data buffer will fill and any further send attempts will block until the downstream service can catch up, draining the buffer and allowing incoming data again. This ensures that the crawlers do not send new articles faster than the filters can process them, and likewise that the filters do not pass on articles faster than the reflectors can process them. The provisioner (detailed in section 3.5) takes this throttling behavior into account when verifying that each crawler will be able to finish crawling its entire list of feeds every 15 minutes (or whatever the *crawl-rate* is specified to be).

3.1 Crawler service

The crawler service takes in a list of source feeds (given as URLs) and periodically crawls the list to detect new articles. A naive crawler would periodically download the contents of each source feed and push all articles contained therein to the filters. However, this approach can consume a considerable amount of bandwidth, both for downloading the source data and sending updates to the filters. In a conventional usage of RSS, many users periodically polling a popular feed can have serious network impact [34]. Although Cobra amortizes the cost of crawling each feed across all users, the sheer number of feeds demands that we are careful about the amount of network bandwidth we consume.

The Cobra crawler includes a number of optimizations designed to reduce bandwidth usage. First, crawlers attempt to use the HTTP `Last-Modified` and `ETag` headers to check whether a feed has been updated since the last polling interval. Second, the crawler makes use of HTTP delta encoding for those feeds that support it.

When it is necessary to download the content for a feed (because its HTTP headers indicate it has changed, or if the server does not provide modification information), the crawler filters out articles that have been previously pushed to filters, reducing bandwidth requirements further and preventing users from seeing duplicate results. We make use of two techniques. First, a whole-document hash using Java's hashCode function is computed; if it matches the previous hash for this feed, the entire document is dropped. Second, each feed that has changed is broken up into its individual articles (or *entries*), which are henceforth processed individually. A hash is computed on each of the individual articles, and those matching a previously-hashed article are filtered out. As we show in Section 4, these techniques greatly reduce the amount of traffic between source feeds and crawlers and between crawlers and filters.

3.2 Filter service

The filter service receives updated articles from crawlers and matches those articles against a set of *subscriptions*. Each subscription is a tuple consisting of a *subscription ID*, *reflector ID*, and list of *keywords*. The subscription ID uniquely identifies the subscription and the reflector ID is the address of the corresponding reflector for that user. Subscription IDs are allocated by the reflectors when users inject subscriptions into the system. Each subscription has a list of keywords that may be related by either conjunctions (e.g. "law AND internet"), disjunctions (e.g. "copyright OR patent"), or a combination of both (e.g. "(law AND internet) OR (privacy AND internet)"). When an article is matched against a given subscription, each word of the subscription is marked either *true* or *false* based on whether it appears anywhere in the article; if the resulting boolean expression evaluates to *true* then the article is considered to have matched the subscription.

Given a high volume of traffic from crawlers and a large number of users, it is essential that the filter be able to match articles against subscriptions efficiently. Cobra uses the matching algorithm proposed by Fabret *et al.* [18, 29]. This algorithm operates in two phases. In the first phase, the filter service uses an index to determine the set of all words (across all subscriptions) that are matched by any article. This has the advantage that words that are mentioned in multiple subscriptions are only evaluated once. In the second phase, the filter determines the set of subscriptions in which all words have a match. This is accomplished by ordering subscriptions according to overlap and by ordering words within subscriptions according to selectivity, to test the most selective words first. If a word was not found in the first phase, all subscriptions that include that word can be discarded without further consideration. As a result, only a fraction of the subscriptions are considered if there is much overlap between them.

Due to its sub-linear complexity, the matching algorithm is extremely efficient: matching a single article against 1 million user subscriptions has a 90th percentile latency of just 10 ms (using data from real Web feeds and synthesized subscriptions, as discussed in Section 4). In contrast, a naive algorithm (using a linear search across the subscription word lists) requires more than 10 sec across the same 1 million subscriptions, a difference of

four orders of magnitude.

3.3 Reflector service

The final component of the Cobra design is the reflector service, which receives matching articles from filters and reflects them as a personalized RSS feed for each user. In designing the reflector, several questions arose. First, should the filter service send the complete article body, a summary of the article, or only a link to the article? Clearly, this has implications for bandwidth usage. Second, how should filters inform each reflector of the set of matching subscriptions for each article? As the number of matching subscriptions increases, sending a list of subscription IDs could consume far more bandwidth than the article contents themselves.

In our initial design, for each matching article, the filter would send the reflector a summary consisting of the title, URL, and first 1 KB of the article body, along with a list of matching subscription IDs. This simplifies the reflector's design as it must simply link the received article summary to the personalized RSS feed of each of the matching subscription IDs. Article summaries are shared across subscriptions, meaning if one article matches multiple subscriptions, only one copy is kept in memory on the reflector.

However, with many active subscriptions, the user list could grow to be very large: with 100,000 matching subscriptions on an article and 32-bit subscription IDs, this translates into 400KB of overhead *per article* being sent to the reflectors. One alternative is to use a bloom filter to represent the set of matching users; we estimate that a 12KB filter could capture a list of 100,000 user IDs with a false positive rate of 0.08%. However, this would require the reflector to test each user ID against the filter on reception, involving a large amount of additional computation.

In our final design, the filter sends the complete article body to the reflector without a user list, and the reflector *re-runs* the matching algorithm against the list of active subscriptions it stores for the users it is serving. Since the matching algorithm is so efficient (taking 10ms for 1 million subscriptions), this appears to be the right trade-off between bandwidth consumption and CPU overhead. Instead of sending the complete article, we could instead send only the union of matching words across all matching subscriptions, which in the worst case reduces to sending the full text.

For each subscription, Cobra caches the last k matching articles, providing a personalized feed which the user can access using a standard RSS reader. The value of k must be chosen to bound memory usage while providing enough content that a user is satisfied with the "hits" using infrequent polling; typical RSS readers poll every 15-60 minutes [28]. In our current design, we set $k = 10$, a value that is typical for many popular RSS feeds (see Figure 5). Another approach might be to dynamically set the value of k based on the user's individual polling rate or the expected popularity of a given subscription. We leave these extensions to future work.

In the worst case, this model of user feeds leads to a memory usage of $k * subscriptions * 1KB$ (assuming articles are capped at 1KB of size). However, in practice the memory usage is generally much lower since articles can be shared across multiple subscriptions. In the event that memory does become scarce, a reflector will begin dropping the content of new articles that are received, saving to users' feeds only the articles' titles and URLs. This greatly slows the rate of memory consumption, but if memory continues to dwindle then reflectors will begin dropping *all* incoming articles (while logging a warning that this is happening). This process ensures a graceful degradation in service quality when required.

A user subscribes to Cobra by visiting a web site that allows the user to establish an account and submit subscription requests in the form of keywords. The web server coordinates with the reflectors and filters to *instantiate* a subscription, by performing two actions: (1) associating the user with a specific reflector node; and (2) injecting the subscription details into the reflector node and the filter node(s) that feed data into that reflector. The response to the user's subscription request is a URL for a private RSS feed hosted by the chosen reflector node. In our current prototype, reflector nodes are assigned randomly to users by the Web server, but a locality-aware mechanism such as Meridian [42] or OASIS [20] could easily be used instead.

3.4 Hosting model

Although "peer to peer," self-organizing systems based on shared resources contributed by volunteers are currently en vogue, we are not sure that this model is the best for provisioning and running a service like Cobra. Rather, we choose to exploit conventional approaches to distributed systems deployment, making use of well-provisioned hosting centers, which is the norm for commercial Internet services. Each of the three tiers of Cobra can be distributed over multiple hosts in the Internet, allowing computational and network load to be balanced across servers and hosting centers. Distribution also allows the placement of Cobra services to take advantage of improved locality when crawling blogs or pushing updates to users.

The use of a hosting center model allows us to make certain assumptions to simplify Cobra's design. First, we assume that physical resources in a hosting center can be dedicated to running Cobra services, or at least that hosting centers can provide adequate virtualization [4, 10] and resource containment [9] to provide this illusion.

Second, we assume that Cobra services can be replicated within a hosting center for increased reliability. Third, we assume that hosting centers are generally well-maintained and that catastrophic outages of an entire hosting center will be rare. Cobra can tolerate outages of entire hosting centers, albeit with reduced harvest and yield [19]. Finally, we assume that allocating resources to Cobra services and monitoring their performance at runtime can be performed centrally. These assumptions strongly influence our approach to service provisioning as we are less concerned with tolerating unexpected variations in CPU and network load and intermittent link and node failures, as is commonly seen on open experimental testbeds such as PlanetLab [30].

3.5 Service provisioning

As the number of source feeds and users grows, there is a significant challenge in how to provision the service in terms of computational horsepower and network bandwidth. Server and network resources cost money; additionally, a system may have limitations on the amount of physical resources available. Our goal is to determine the minimal amount of physical resources required to host a Cobra network capable of supporting a given number of source feeds and users. For this purpose, we make use of an offline *service provisioning* technique that determines the configuration of the Cobra network in terms of the number of crawlers, filters, and reflectors, as well as the interconnectivity between these services. Due to space constraints, we only provide an informal description of the service provisioning algorithm.

The provisioner takes as inputs the target number of source feeds and users, a model of the memory, CPU and bandwidth requirements for each service, as well as other parameters such as distribution of feed sizes and the per-user polling rate. The provisioner also takes as input a set of node constraints, consisting of limits on inbound and outbound bandwidth, maximum memory available to the JVM, and CPU processing power. Note that this last value is difficult to measure directly and thus we model it simply as a dimensionless parameter relative to the processing performance observed on Emulab's pc3000 machines [1]. For example, a CPU constraint of 0.75 implies that the provisioner should assume that nodes will process messages only 75% as fast as the pc3000s. The provisioner's output is a graph representing the topology of the Cobra network graph, including the number of feeds assigned to each crawler and the number of subscriptions assigned to each reflector and each filter.

The provisioner models each Cobra service as running on a separate physical host with independent memory, CPU and bandwidth constraints. This results in a conservative estimate of resource requirements as it does not permit multiple services within a hosting center to share resources (e.g., bandwidth). A more sophisticated algorithm could take such resource sharing into account.

The provisioner attempts to configure the network to meet the target number of source feeds and users while minimizing the number of services. The algorithm operates as follows. It starts with a simple 3-node topology with one crawler, one filter, and one reflector. In each iteration, the algorithm identifies any constraint violations in the current configuration, and greedily resolves them by *decomposing* services as described below. When no more violations exist, the algorithm terminates and reports success, or if a violation is found that cannot be resolved, the algorithm terminates and reports failure.

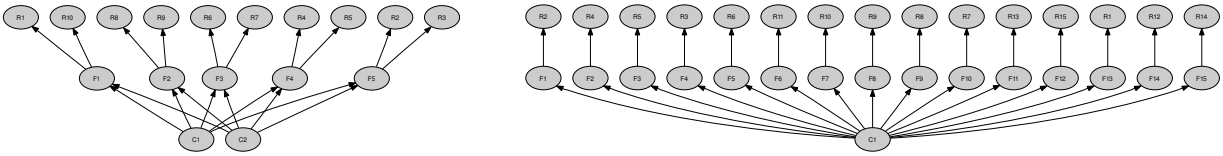
An *out-decomposition* resolves violations by replacing a single service with n replicas such that all incoming links from the original service are replicated across the replicas, whereas the outgoing links from the original services are load balanced across the replicas. An *in-decomposition* does the opposite: a single service is replaced by n replicas such that all outgoing links from the original service are replicated across the replicas, whereas the incoming links from the original services are load balanced across the replicas.

In resolving a violation on a service, the choice of decomposition type (in- or out-) depends both on the type of violation (in-bandwidth, out-bandwidth, CPU, or memory) and the type of service (crawler, filter or reflector). Figure 3 shows which decomposition is used in each situation.

When faced with multiple violations, the algorithm uses a few simple heuristics to choose the order in which to resolve them. Some violations have the potential to be resolved indirectly in the course of resolving other violations. For example, if a crawler service has both in-bandwidth and out-bandwidth violations, resolving the in-bandwidth violation is likely to also resolve the out-bandwidth violation (by reducing the number of feeds crawled, we also implicitly reduce the number of feed updates that are found and output by the crawler). Thus it is preferable in this case to resolve the in-bandwidth violation first as it may solve both violations with one decomposition. In general, when choosing which of multiple violations to resolve first, the algorithm will choose the violations with the least potential to be resolved indirectly, thus saving the violations with higher potential until as late as possible (in the hopes that they will "happen to be resolved" in the mean time).

Although this greedy approach might lead to local minima and may in fact fail to find a topology that satisfies the input constraints when such a configuration does exist, in practice the algorithm produces network topologies with a modest number of nodes to handle large loads. We chose this greedy iterative approach because it

[1] 3.0 GHz 64-bit Xeon processors



(a) Configuration for 4x CPU and 25 Mbps bandwidth

(b) Configuration for 1x CPU and 100 Mbps bandwidth

Figure 2: **Operation of the Cobra network provisioner.** *These figures show how provisioner results can vary for different constraint combinations; in both of these cases the network is provisioned for 800,000 feeds, 8 million subscriptions, and 1024 MB of memory, but the CPU and bandwidth constraints differ. (a) Shows the resulting configuration when the CPU constraint is 4x the default value (see text) and the bandwidth constraint is 25 Mbps. (b) Shows the resulting configuration when the CPU constraint is the default value (1x) and the bandwidth constraint is 100 Mbps. Compared to (a), this configuration requires half the number of crawlers, 50% more reflectors and three times as many filters (on account of the much greater processing needs).*

Service	Violation	Decomposition	Reason
Crawler	In-BW	In	Reduces the number of feeds crawled.
	Out-BW	In	Reduces the rate that updates are found and output to filters.
	CPU	None	Not modeled
	Memory	None	Not modeled
Filter	In-BW	In	Reduces the number of crawlers that send updates to each filter.
	Out-BW	In	Reduces the rate that articles are received, and thus also the rate that articles are output to reflectors.
	CPU	Out	Reduces the number of subscriptions that articles must match against.
	Memory	Out	Reduces the number of subscriptions that must be stored on the filter.
Reflector	In-BW	None	Not resolvable because reflectors must receive updates from all feeds (otherwise users will not receive all articles that match their subscription).
	Out-BW	Out	Reduces the subscriptions held by each reflector, which reduces the expected frequency of web queries by users.
	CPU	Out	Reduces the number of subscriptions that each incoming articles must be matched against and the number of article-queues that must be updated.
	Memory	Out	Reduces the number of subscriptions and article lists that must be stored.

Figure 3: **Provisioner choice of decomposition for each service/violation combination.**

was conceptually simple and easy to implement. Figure 2 shows two provisioner topologies produced for different input constraints.

3.6 Service instantiation and monitoring

The output of the provisioner is a *virtual graph* (see Figure 2) representing the number and connectivity of the services in the Cobra network. Of course, these services must be instantiated on physical hosts. A wide range of instantiation policies could be used, depending on the physical resources available. For example, a small startup might use a single hosting center for all of the services, while a larger company might distribute services across multiple hosting centers to achieve locality gains. Both approaches permit incremental scalability by growing the number of machines dedicated to the service.

The Cobra design is largely independent of the mechanism used for service instantiation. In our experiments described in Section 4, we use different strategies based on the nature of the testbed environment. In our dedicated cluster and Emulab experiments, services are mapped one-to-one with physical hosts in a round-robin fashion. In our PlanetLab experiments, services are distributed randomly to achieve good coverage in terms of locality gains for crawling and reflecting (described below). An alternate mechanism could make use of previous work on network-aware service placement to minimize bandwidth usage [8, 33].

After deployment, it is essential that the performance of the Cobra network be monitored to validate that it is meeting targets in terms of user-perceived latency as well as bandwidth and memory constraints. Also, as the user population and number of source feeds grow it will be essential to re-provision Cobra over time. We envision this process occurring over fairly coarse-grained time periods, such as once a month or quarter. Each Cobra node is instrumented to collect statistics on memory usage, CPU load, and inbound and outbound bandwidth consumption. These statistics can be collected periodically to ascertain whether re-provisioning is necessary.

3.7 Source feed mapping

Once crawler services have been instantiated, the final step in running the Cobra network is to assign source feeds to crawlers. In choosing this assignment, we are concerned not only with spreading load across multiple crawlers, but also reducing the total *network load* that the crawlers will induce on the network. A good way of

reducing this load is to optimize the locality of crawlers and their corresponding source feeds. Apart from being good network citizens, improving locality also reduces the latency for crawling operations, thereby reducing the update detection latency as perceived by users. Because the crawlers use fairly aggressive timeouts (5 sec) to avoid stalling on slow feeds, reducing crawler-feed latency also increases the overall yield of a crawling cycle.

In Cobra, we assign source feeds to crawlers in a latency-aware fashion. One approach is to have each crawler measure the latency to all of the source feeds, and use this information to perform a coordinated allocation of the source feed list across the crawlers. Alternately, we could make use of network coordinate systems, such as Vivaldi [17], which greatly reduces ping load by mapping each node into a low-dimensional coordinate space, allowing an estimate of the latency between any two hosts to be measured as the Euclidean distance in the coordinate space. However, such schemes require end hosts to run the network coordinate software, which is not possible in the case of oblivious source feeds.

Instead, we perform an offline measurement of the latency between each of the source feeds and crawler nodes using King [24]. King estimates the latency between any two Internet hosts by performing an external measurement of the latency between their corresponding DNS servers; King has been reported to have a 75th percentile error of 20% of the true latency value. It is worth noting that many source feeds are hosted by the same IP address, so we achieve a significant reduction in the measurement overhead by limiting probes to those nodes with unique IP addresses. In our sample of 102,446 RSS feeds, there are only 591 unique IP addresses.

Given the latency matrix between feeds and crawlers, we perform assignment using a simple first-fit bin-packing algorithm. The algorithm iterates through each crawler C_j and calculates $i^\star = \arg\min l(F_i, C_j)$, where $l(\cdot)$ is the latency between F_i and C_j. $F_{i^\star}$ is then assigned to C_j. Given F feeds and C crawlers, we assign F/C feeds to each crawler (assuming $F > C$). We have considered assigning varying number of feeds to crawlers, for example, based on the posting activity of each feed, but have not yet implemented this technique.

Figure 4 shows an example of the source feed mapping from one of our experiments. To reduce clutter in the map we show only 3 crawlers (one in the US, one in Switzerland, and one in Japan) and the 5 nearest crawlers, according to estimated latency, for each. The mapping process is clearly effective at achieving good locality and naturally minimizes traffic over transoceanic links.

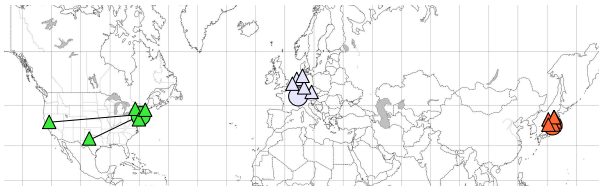


Figure 4: **An example of locality-aware source feed mapping.** *Three crawlers are shown as circles and the 5 nearest source feeds, according to estimated latency, are shown as triangles. Colors indicate the mapping from feeds to crawlers, which is also evident from the geographic layout.*

3.8 Implementation

Our prototype of Cobra is implemented in Java, and makes use of our substrate for *stream-based overlay networks* (SBONs) [33] for setting up and managing data flows between services. Note, however, that the placement of Cobra services onto physical hosts is determined statically, at instantiation time, rather than dynamically as described in our previous work [33]. A central *controller node* handles provisioning and instantiation of the Cobra network. The provisioner outputs a logical graph which is then instantiated on physical hosts using a (currently random) allocation of services to hosts. The instantiation mechanism depends on the specific deployment environment.

Our implementation of Cobra consists of 29178 lines of Java code in total. The crawler service is 2445 lines, the filter service is 1258 lines, and the reflector is 622 lines. The controller code is 377 lines, while the remaining 24476 consists of our underlying SBON substrate for managing the overlay network.

4 Evaluation

We have several goals in our evaluation of Cobra. First, we show that Cobra can scale well to handle a large number of source feeds and user subscriptions. Scalability is limited by service resource requirements (CPU and memory usage) as well as network bandwidth requirements. However, a modestly-sized Cobra network (Figure 2) can handle 8M users and 800,000 source feeds. Second, we show that Cobra offers low latencies for discovering matching articles and pushing those updates to users. The limiting factor for update latency is the rate at which source feeds can be crawled, as well as the user's own polling interval. We also present data comparing these update latencies with three existing blog search engines: Google Blog Search, Feedster, and Blogdigger.

We present results from experiments on three platforms: a local cluster, the Utah Emulab testbed [41], and PlanetLab. The local cluster allows us to measure service-level performance in a controlled setting, al-

	median	90th percentile
Size of feed (bytes)	7606	22890
Size of feed (articles)	10	17
Size of article (bytes)	768	2426
Size of article (words)	61	637

Figure 5: **Properties of Web feeds used in our study.**

though scalability is limited. Our Emulab results allow us to scale out to larger configurations. The PlanetLab experiments are intended to highlight the value of source feed clustering and the impact of improved locality.

We use a combination of real and synthesized web feeds to measure Cobra's performance. The real feeds consist of a list of 102,446 RSS feeds from syndic8.com, an RSS directory site. The properties of these feeds were studied in detail by Liu et al. in [28]. To scale up to larger numbers, we implemented an artificial *feed generator*. Each generated feed consists of 10 articles with words chosen randomly from a distribution of English words based on popularity rank from the Brown corpus [3]. Generated feed content changes dynamically with update intervals similar to those of real feeds, based on data from [28]. The feed generator is integrated into the crawler service and is enabled by a runtime flag.

Simulated user subscriptions are similarly generated with a keyword list consisting of the same distribution as that used to generate feeds. We exclude the top 29 most popular words, which are considered excessively general and would match essentially any article. (We assume that these words would normally be ignored by the subscription web portal when a user initially submits a subscription request.) The number of words in each query is chosen from a distribution based on a Yahoo study [11] of the number of words used in web searches; the median subscription length is 3 words with a maximum of 8. All simulated user subscriptions contain only conjunctions between words (no disjunctions). In Cobra, we expect that users will typically submit subscription requests with many keywords to ensure that the subscription is as specific as possible and does not return a large number of irrelevant articles. Given the large number of simulated users, we do not actively poll Cobra reflectors, but rather estimate the additional network load that this process would generate.

4.1 Properties of Web feeds

Liu et al. [28] present a detailed evaluation of the properties of RSS feeds, using the same list of 102,446 RSS feeds used in our study. Figure 5 summarizes the size of the feeds and individual articles observed in a typical crawl of this set of feeds between October 1–5, 2006. The median feed size is under 8 KB and the median number of articles per feed is 10.

Figure 6 shows a scatterplot of the size of each feed compared to its crawl time from a PlanetLab node running at Princeton. The figure shows a wide variation in the size and crawl time of each feed, with no clear relationship between the two. The large spike around size 8000 bytes represents a batch of 36,321 RSS feeds hosted by *topix.net*. It turns out these are not static feeds but dynamically-generated aggregation feeds across a wide range of topics, which explains the large variation in the crawl time.

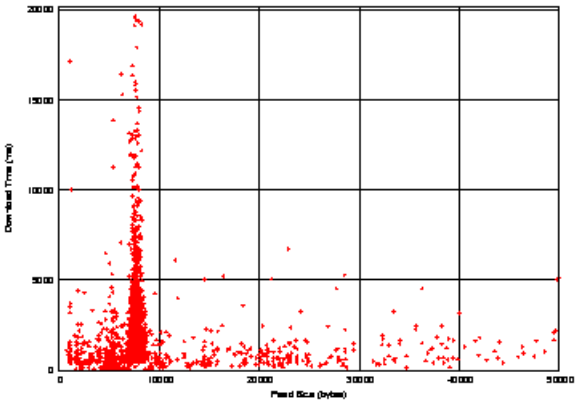


Figure 6: **Relationship between feed size and crawl time.**

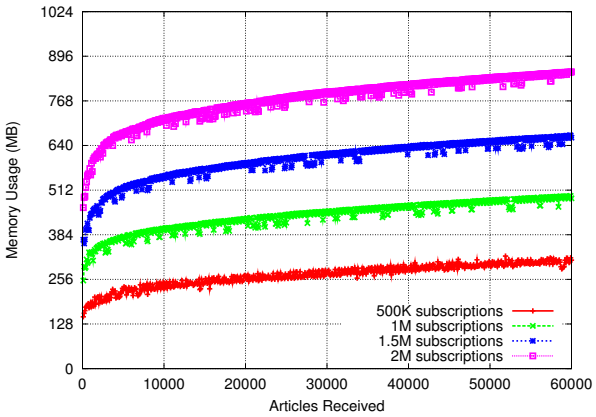


Figure 7: **Memory usage of the reflector service over time.** *The x-axis of this figure is the total number of articles received by the reflector. For context, we estimate that a set of 1 million feeds can be expected to produce an average of ∼48.5 updated articles every second, or ∼2910 each minute.*

4.2 Microbenchmarks

Our first set of experiments measure the performance of the individual Cobra services.

Memory usage

Figure 7 shows the memory usage of a single Reflector service as articles are received over time. In each case, the usage follows a logarithmic trend. However, the curves' obvious offsets make it clear that the number of subscriptions stored on each reflector strongly influences its memory usage. For example, with a half-million subscriptions, the memory usage reaches ∼310 MB after re-

ceiving 60,000 articles, whereas with 1 million subscriptions the memory usage reaches nearly 500 MB. This is not surprising; not only must reflectors store each actual subscription (for matching), but also each user's list of articles.

However, after the initial burst of article storage, the rate of memory consumption slows dramatically due to the cap (of $k = 10$) on each user's list of stored articles. This cap prevents users with particularly general subscriptions (that frequently match articles) from continually using up memory. Note that in this experiment no articles (or article contents) were dropped by the reflectors' scarce memory handling logic (as described in section 3.3). The only time that articles were dropped was when a user's list of stored articles exceeded the size cap.

This experiment assumes that articles are never expired from memory (except when a user's feed grows beyond length k). It is easy to envision an alternative design in which a user's article list is cleared whenever it is polled (by the user's RSS reader) from a reflector. Depending on the frequency of user polling, this may decrease overall memory usage on reflectors but an analysis of the precise benefits is left to future work.

In contrast, the memory usage of the crawler and filter services does not change as articles are processed. For crawlers, the memory usage while running is essentially constant since crawlers are unaffected by the number of subscriptions. For filters, the memory usage was found to vary linearly with the number of subscriptions (∼0.16 MB per 1000 subscriptions held) and thus changes only when subscriptions are added or removed.

Crawler performance

Figure 8 shows the bandwidth reduction resulting from optimizations in the crawler to avoid crawling feeds that have not been updated. As the figure shows, using last-modified checks for reading data from feeds reduces the inbound bandwidth by 57%. The combination of techniques for avoiding pushing updates to the filters results in a 99.8% reduction in the bandwidth generated by the crawlers, a total of 2.2 KB/sec for 102,446 feeds. We note that none of the feeds in our study supported the use of HTTP delta encoding, so while this technique is implemented in Cobra it does not yield any additional bandwidth savings.

The use of locality-aware clustering should reduce the time to crawl a set of source feeds, as well as reduce overall network load. From our initial set of 102,446 feeds, we filtered out those that appeared to be down as well as feeds from two aggregator sites, *topix.net* and *izynews.de*, that together constituted 50,953 feeds. These two sites host a large number of dynamically-generated feeds that exhibit a wide variation in crawl times, making it difficult to differentiate network effects.

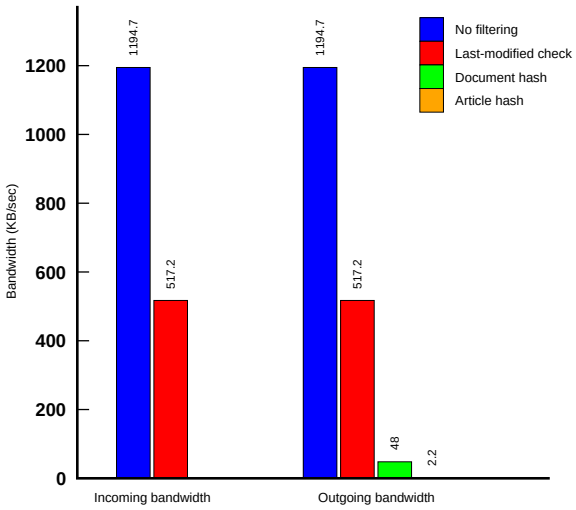


Figure 8: **Bandwidth reduction due to intelligent crawling.** *This graph shows the amount of data generated by the crawler using different techniques: (a) crawl all feeds; (b) filter based on last-modified header; (c) filter based on whole-document hash; and (d) filter based on per-article hashes.*

Figure 9 shows the time to crawl the remaining 34,092 RSS feeds distributed across 481 unique IP addresses. 11 crawlers were run on PlanetLab distributed across North America, Europe, and Asia. With locality aware mapping, the median crawl time per feed drops from 197 ms to 160 ms, a reduction of 18%.

Filter performance

Figure 10 shows the median time for the filter's matching algorithm to compare a single article against an increasing number of user subscriptions. The matching algorithm is very fast, requiring less than 20 ms to match an article of 2000 words against 1 million user subscriptions. Keep in mind that according to Figure 5 that the median article size is just 61 words, so in practice the matching time is much faster: we see a 90th percentile of just 2 ms per article against 1 million subscriptions. Of course, as the number of incoming articles increases, the overall matching time may become a performance bottleneck, although this process is readily distributed across multiple filters.

4.3 Scalability measurements

To demonstrate the scalability of Cobra with a large number of feeds and user subscriptions, we ran additional experiments using the Utah Emulab testbed. Here, we are interested in two key metrics: (1) The *bandwidth consumption* of each tier of the Cobra network, and (2) The

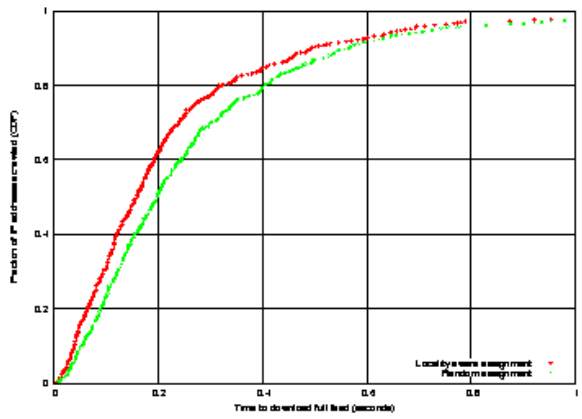


Figure 9: **Effect of locality-aware clustering.** *This is a CDF of the time to crawl 34092 RSS feeds across 481 separate IP addresses from 11 PlanetLab hosts, with and without the locality aware clustering.*

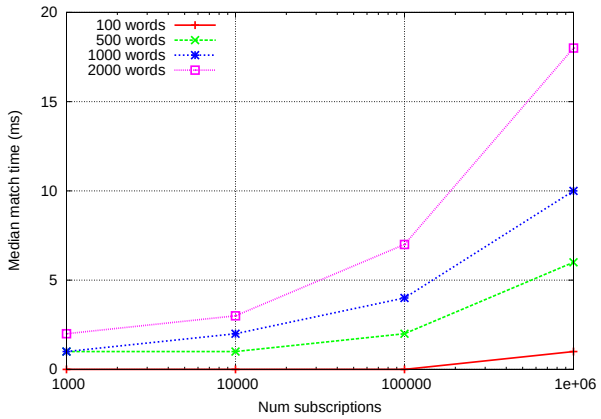


Figure 10: **Article match time versus number of subscriptions and number of words per article.** *The median time to match an article is a function of the number of subscriptions and the number of words per article.*

latency for an updated article from a source feed to propagate through the three tiers of the network. In total, we evaluated four different topologies, summarized in Figure 11.

Each topology was generated by the provisioner with a bandwidth constraint of 100 Mbps [2], a memory constraint of 1024 MB, and a CPU constraint of the default value (1x). In addition, we explicitly over-provisioned by 10% as a guard against bursty traffic or unanticipated bottlenecks when scaling up, but it appears that this was an largely unnecessary precaution. Each topology was run for four crawling intervals of 15 minutes each and the logs were checked at the end of every experiment to confirm that none of the reflectors dropped any articles (or article contents) to save memory (a mechanism invoked when available memory runs low, as discussed in section 3.3).

[2]We feel that the 100 Mbps bandwidth figure is not unreasonable; bandwidth measurements from PlanetLab indicate that the median inter-node bandwidth across the Internet is at least this large [26].

Subs	Feeds	Crawlers	Filters	Reflectors
10M	1M	1	28	28
20M	500,000	1	25	25
40M	250,000	1	28	28
1M	100,000	1	1	1

Figure 11: **Topologies used in scalability measurements.** *The last topology (100K feeds, 1M subscriptions) is meant to emulate a topology using the live set of 102,446 feeds.*

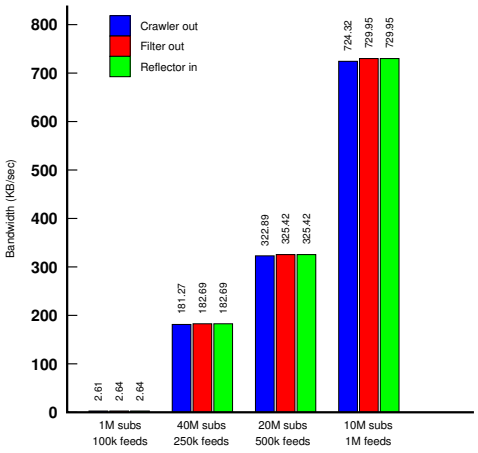


Figure 12: **Bandwidth consumption of each tier.** *The bandwidth of each tier is a function both of the number of feeds that are crawled and of the fan-out from each crawler to the filters.*

Figure 12 shows the total bandwidth consumption of each tier of the Cobra network for each of the four topologies evaluated. As the figure shows, total bandwidth consumption remains fairly low despite the large number of users and feeds, owing mainly to the effective use of intelligent crawling. Note that due to the relatively large number of subscriptions in each topology, the *selectivity* of the filter tier is nearly 1; every article will match some user subscription, so there is no noticeable reduction in bandwidth from the filter tier (the very slight increase in bandwidth is due to the addition of header fields to each article). One potential area for future work is finding ways to reduce the selectivity of the filter tier. If the filters' selectivity can be reduced, that will reduce not only the filters' bandwidth consumption, but also the number of reflectors needed to process and store the (fewer) articles sent from the filters. One way to lower filter selectivity may be to assign subscriptions to filters based on similarity (rather than the current random assignment); if all of the subscriptions on a filter tend towards a single, related set of topics, then more articles may fail to match any those subscriptions.

We are also interested in the *intra-network latency* for an updated article passing through the three tiers of the

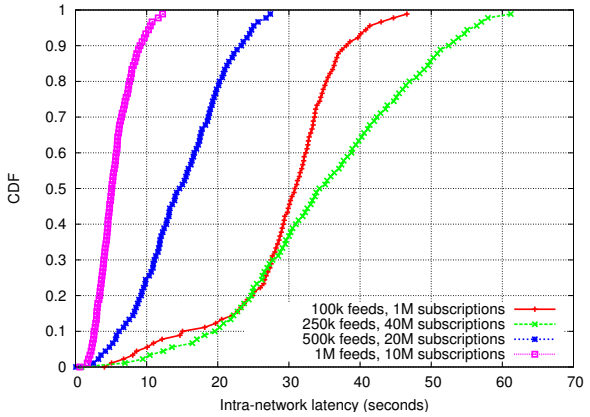


Figure 13: **CDF of intra-network latency for various topologies.** *This experiment shows that the intra-network latency is largely a factor of the processing load on filter and reflectors.*

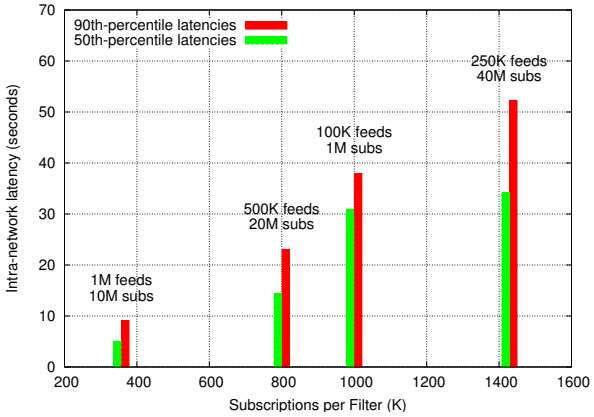


Figure 14: **Intra-network latency as a function of subscriptions per Filter.** *This figure shows the relationship between intra-network latency and the number of subscriptions stored on each Filter (note that in each of these topologies, the number of filters equals the number of reflectors, and thus the x-axis is equivalent to "Subscriptions per Reflector (K)").*

Cobra network. To gather this data, we instrumented the crawler, filter, and reflector to send a packet to a central logging host every time a given article was (1) generated by the crawler, (2) received at a filter, (2) matched by the filter, and (3) delivered to the reflector. Although network latency between the logging host and the Cobra nodes can affect these results, we believe these latencies to be small compared to the Cobra overhead.

Figure 13 shows a CDF of the latency for each of the four topologies. As the figure shows, the fastest update times were observed on the 1M feeds / 10M subs topology, with a median latency of 5.06 sec, whereas the slowest update times were exhibited by the 250K feeds / 40M subs topology, with a median latency of 34.22 sec. However, the relationship is not simply that intra-network latency increases with the number of users; the median latency of the 100K feeds / 1M subs topology was 30.81 sec - nearly as slow as the 250K feeds / 40M subs topology. Instead, latency appears more closely related to the number of subscriptions stored *per node* (rather than in total), as shown in Figure 14.

As mentioned at the end of section 3.2, nodes are able to throttle the rate at which they are passed data from other nodes. This is the primary source of intra-network latency; article updates detected by crawlers are delayed in reaching reflectors because of processing congestion on filters and/or reflectors. Since the time for a filter (or reflector) to process an article is related to the number of subscriptions that must be checked (see figure 10), topologies with larger numbers of subscriptions *per node* exhibit longer processing times, leading to rate-throttling of upstream services and thus larger intra-network latencies. Figure 14 shows a clear relationship between the number of subscriptions per node and the intra-network latencies. However, even in the worst of these cases, the latencies are still fairly low overall. As the system is scaled to handle more subscriptions and more users, Cobra will naturally load-balance across multiple hosts in each tier, keeping latencies low.

Note that the user's *perceived update latency* is bounded by the sum of the *intra-network latency* once an article is crawled by Cobra, and the *crawling interval*, that is, the rate at which source feeds are crawled. In our current system, we set the crawling interval to 15 minutes, which dominates the intra-network latencies shown in Figure 13. The intra-network latency is in effect the minimum latency that Cobra can support, if updates to feeds could be detected instantaneously.

4.4 Comparison to other search engines

Given the number of other blog search engines on the Internet, we were curious to determine how well Cobra's update latency compared to these sites. We created blogs on two popular blogging sites, LiveJournal and Blogger.com, and posted articles containing a sentence of several randomly-chosen words to each of these blogs.[3] We then searched for our blog postings on three sites: Feedster, Blogdigger, and Google Blog Search, polling each site at 5 sec intervals.

We created our blogs at least 24 hours prior to posting, to give the search engines enough time to index them. Neither Feedster or Blogdigger detected *any* of our postings to these blogs, even after a period of over four months (from the initial paper submission to the final camera-ready). We surmise that our blog was not indexed by these engines, or that our artificial postings were screened out by spam filters used by these sites.

Google Blog Search performed incredibly well, with a detection latency as low as 83 seconds. In two out

[3] An example posting was "certified venezuela gribble spork." Unsurprisingly, no extant blog entries matched a query for these terms.

of five cases, however, the latency was 87 minutes and 6.6 hours, respectively, suggesting that the performance may not be predictable. The low update latencies are likely the result of Google using a *ping service*, which receives updates from the blog site whenever a blog is updated [1]. The variability in update times could be due to crawler throttling: Google's blog indexing engine attempts to throttle its crawl rate to avoid overloading [16]. As part of future work, Cobra could be extended to provide support for a ping service and to tune the crawl rate on a per-site basis.

We also uncovered what appears to be a bug in Google's blog indexer: setting our unique search term as the title of the blog posting with no article body would cause Google's site to return a bogus results page (with no link to the matching blog), although it appears to have indexed the search term. Our latency figures ignore this bug, giving Google the benefit of the doubt although the correct result was not returned.

In contrast, Cobra's average update latency is a function of the crawler period, which we set at 15 minutes. With a larger number of crawler daemons operating in parallel, we believe that we could bring this interval down to match Google's performance. To our knowledge, there are no published details on how Google's blog search is implemented, such as whether it simply leverages Google's static web page indexer.

5 Conclusions and Future Work

We have presented Cobra, a system that offers real-time content-based search and aggregation on Web feeds. Cobra is designed to be incrementally scalable, as well as to make careful use of network resources through a combination of offline provisioning, intelligent crawling and content filtering, and network-aware clustering of services. Our prototype of Cobra scales well with modest resource requirements and exhibits low latencies for detecting and pushing updates to users.

Mining and searching the dynamically varying blogosphere offers many exciting directions for future research. We plan to host Cobra as a long-running service on a local cluster and perform a measurement study generated from real subscription activity. We expect our experience to inform our choice of parameters, such as how often to re-provision the system and how to set the number of articles cached for users (perhaps adaptively, depending on individual user activity). We also plan to investigate whether more sophisticated filtering techniques are desirable. Our current matching algorithm does not rank results by relevance, but rather only by date. Likewise, the algorithm is unconcerned with positional characteristics of matched keywords; as long as all keywords match an article, it is delivered to the user.

Unlike Web search engines, it is unclear what constitutes a good ranking function for search results on RSS feeds. For example, the link-based context such as that used by PageRank [23] may need to be modified to be relevant to Web feeds such as blog postings, which have few inbound links but often link to other (static) Web pages. Incorporating this contextual information is extremely challenging given the rapidly-changing nature of Web feeds.

Another open question is how to rapidly discover new Web feeds and include them into the crawling cycle. According to one report [38], over 176,000 blogs were created *every day* in July 2006. Finding new blogs on popular sites such as Blogger and LiveJournal may be easier than more generally across the Internet. While the crawler could collect lists of RSS and Atom URLs seen on crawled pages, incorporating these into the crawling process may require frequent rebalancing of crawler load. Finally, exploiting the wide distribution of update rates across Web feeds offers new opportunities for optimization. If the crawler services could learn which feeds are likely to be updated frequently, the crawling rate could be tuned on a per-feed basis.

References

[1] Google Blog Search FAQ. `http://www.google.com/help/blogsearch/about_pinging.html`.

[2] Livejournal. `http://www.livejournal.com`.

[3] Net dictionary index – brown corpus frequent word lising. `http://www.edict.com.hk/lexiconindex/`.

[4] Vmware esx server. `http://www.vmware.com/products/vi/esx/`.

[5] Js-javaspaces service specification, 2002. `http://www.jini.org/nonav/standards/davis/doc/specs/html/js-spec.html`.

[6] Tibco publish-subscribe, 2005. `http://www.tibcom.com`.

[7] Tspaces, 2005. `http://www.almaden.ibm.com/cs/TSpaces/`.

[8] Y. Ahmad and U. Çetintemel. Network-Aware Query Processing for Stream-based Applications. In *Proc. of VLDB'04*, Toronto, Canada, Aug. 2004.

[9] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. the Third OSDI (OSDI '99)*, February 1999.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP*, Oct. 2003.

[11] D. Bogatin. Yahoo Searches More Sophisticated and Specific. `http://blogs.zdnet.com/micro-markets/index.php?p=27`, May 18 2006.

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

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

[14] U. Centintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. ag S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Z. k. The Design of the Borealis Stream Processing Engine. In *Proc. of CIDR*, Asilomar, CA, Jan. 2005.

[15] M. Cherniack, H. Balakrishnan, M. Balazinska, et al. Scalable Distributed Stream Processing. In *Proc. of CIDR*, Asilomar, CA, Jan. 2003.

[16] M. Cutts. More webmaster console goodness. Matt Cutts: Gadgets, Google, and SEO (blog), `http://www.mattcutts.com/blog/more-webmaster-console-goodness/`, October 20 2006.

[17] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *SIGCOMM*, Aug. 2004.

[18] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, and K. A. Ross. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *Proceedings of ACM SIGMOD*, 2001.

[19] A. Fox and E. A. Brewer. Harvest, yield and scalable tolerant systems. In *Proc. the 1999 Workshop on Hot Topics in Operating Systems*, Rio Rico, Arizona, March 1999.

[20] M. J. Freedman, K. Lakshminarayanan, and D. Mazieres. OASIS: Anycast for Any Service. In *Proc. USENIX/ACM NSDI*, San Jose, CA, 2006.

[21] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An Architecture for a World-Wide Sensor Web. *IEEE Pervasive Computing*, 2(4), Oct. 2003.

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

[23] S. Grin and L. Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *WWW7*, 1998.

[24] K. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating Latency between Arbitrary Internet End Hosts. In *Second Usenix/ACM SIGCOMM Internet Measurement Workshop*, Nov. 2002.

[25] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. tt Shenker, and I. Stoica. Querying the Internet with PIER. In *Proc. of VLDB*, Berlin, Germany, Sept. 2003.

[26] S.-J. Lee, P. Sharma, S. Banerjee, S. Basu, and R. Fonseca. Measuring Bandwidth between PlanetLab Nodes. In *Proc. Passive and Active Measurement Workshop*, Boston, MA, March 2005.

[27] L.F.Cabera, M. Jones, and M. Theimer. Herald: Achieving a Global Event Notification Service. In *Workshop on Hot Topics in Operating Systems*, 2001.

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

[29] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient Matching for Web-Based Publish-Subscribe Systems. In *Proceedings of the 7th International Conference on Cooperative Information Systems (CoopIS)*, 2000.

[30] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Mui. Experiences Building PlanetLab. *OSDI*, Nov. 2006.

[31] P. Pietzuch. *Hermes: A Scalable Event-Based Middleware*. PhD thesis, University of Cambridge, Feb. 2004.

[32] P. Pietzuch and J. M. Bacon. Hermes: A Distributed Event-Based Middelware Architecture. In *1st Workshop on Distributed Event-Based Systems*, jul 2002.

[33] P. Pietzuch, J. Ledlie, J. Shneidman, M. R. M. Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *ICDE*, Apr. 2006.

[34] V. Ramasubramanian, R. Peterson, and G. Sirer. Corona: A High Performance Publish-Subscribe System for the World Wide Web. In *NSDI*, 2006.

[35] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware)*, Nov. 2001.

[36] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *NGC*, 2001.

[37] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin. In *Proceedingsof AUUG2K*, 2000.

[38] D. Sifry. State of the Blogosphere, August 2006. `http://www.sifry.com/alerts/archives/000436.html`.

[39] R. Strom, G. Banavar, T. Chandra, M. Kaplan, and K. Miller. Gryphon: An Information Flow Based Approach to Message Brokering. In *Proceedings of the International Symposium on Software Reliability Engineering*, 1998.

[40] R. van Renesse, K. Birman, and W. Vogels. A Robust and Scalable Technology for Distributed Systems Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21(2):164–206, 2003.

[41] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, Dec. 2002.

[42] B. Wong, A. Slivkins, and G. Sirer. Meridian: A Lightweight Network Location Service without Virtual Coordinates. In *SIGCOMM*, Aug. 2005.

[43] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. In *NOSSDAV*, June 2001.