FeedEx: Collaborative Exchange of News Feeds[*]

Seung Jun
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280, U.S.A.
jun@cc.gatech.edu

Mustaque Ahamad
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280, U.S.A.
mustaq@cc.gatech.edu

ABSTRACT

As most blogs and traditional media support RSS or Atom feeds, the news feed technology becomes increasingly prevalent. Taking advantage of ubiquitous news feeds, we design FeedEx, a news feed exchange system. Forming a distribution overlay network, nodes in FeedEx not only fetch feed documents from the servers but also exchange them with neighbors. Among many benefits of collaborative feed exchange, we focus on the low-overhead, scalable delivery mechanism that increases the availability of news feeds. Our design of FeedEx is incentive-compatible so that nodes are encouraged into cooperating rather than free riding. In addition, for a better design of FeedEx, we analyze the data collected from 245 feeds for 10 days and present relevant statistics about news feed publishing, including the distributions of feed size, entry lifetime, and publishing rate.

Our experimental evaluation using 189 PlanetLab machines, which fetch from real-world feed servers, shows that FeedEx is an efficient system in many respects. Even when a node fetches feed documents as infrequently as every 16 hours, it captures more than 90% of the total entries published, and those captured entries are available within 22 minutes on average after published at the servers. By contrast, stand-alone applications in the same condition show 36% of entry coverage and 5.7 hours of time lag. The efficient delivery of FeedEx is achieved with low communication overhead as each node receives only 0.9 document exchange calls and 6.3 document checking calls per minute on average.

Categories and Subject Descriptors

C.2.4 [**Computer-communication networks**]: Distributed systems—*Distributed applications*

General Terms

Design, Performance

Keywords

FeedEx, RSS, Atom, news feeds, collaborative exchange

[*]This paper was supported in part by NSF grant ITR-0121643.

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.
WWW 2006, May 23–26, 2006, Edinburgh, Scotland.
ACM 1-59593-323-9/06/0005.

1. INTRODUCTION

The advent of the web and more recently blogs introduce an unprecedented opportunity for information sharing in that anyone can write their knowledge and opinion for everyone to read. According to the BBC, one blog is created every second these days, reaching more than 14.2 million blogs [11]. Although the increased level of information flow possibly evolves our society forward, such advancement necessitates an efficient way of exchanging information.

As a response to the need of efficient information exchange, the standards such as RSS (really simple syndication or rich site summary) and later Atom have been introduced. They specify document formats that are used to contain a list of entries summarizing recent changes in a web site or a blog. These RSS or Atom feeds, referred to as *news feeds* throughout the paper, are used by end users as well as other web sites. Currently, most traditional mass media and personal blogs publish their articles in news feeds. However, the standards around this technology have paid little attention to an efficient delivery of news feeds. In fact, there is currently no distinction between news feeds and regular web pages from a web server's perspective. Thus, if users are to check whether new entries are published, they only have to fetch the feed documents as frequently as they want. The lack of effective notification of updates can lead to the aggressive probing, which not only wastes clients' network bandwidth but more importantly overloads the servers.

In this paper, we design and evaluate FeedEx, a news feed exchange system. Its nodes form a distribution overlay network over which news feeds are exchanged. Since this exchange allows nodes to reduce the frequency of fetching documents from servers, it can decrease the server load. In a sense, FeedEx builds an effective notification system that the current news feed technology lacks. Due to the effective notification, nodes benefit from timely delivery and high availability of feeds. We design an incentive mechanism for FeedEx so that nodes are encouraged into being collaborators rather than free riders. Since FeedEx does not require any modification of current feed servers or document formats, it can be readily deployed. Our Internet experimental results show that it achieves high availability and quick delivery time with low communication overhead, thus helping the feed servers scale well.

The rest of this paper is organized as follows. The remainder of this section gives the background on the news feed technology and points out the benefits of FeedEx. In Section 2, we present relevant statistics about news feed publishing by analyzing the data collected from 245 feeds

```
<feed>
  <title>NYT Technology</title>
  <!-- other elements -->
  <entry>
    <title>Google Wants to Dominate ...</title>
    <link>http://www.nytimes.com/2005/...</link>
    <summary>This year Google will ...</summary>
    <!-- other elements -->
  </entry>
  <entry>
    ...
  </entry>
  <!-- more entries -->
</feed>
```

Figure 1: A simplified sample of news feed

for 10 days. Section 3 describes the design and the protocol of FeedEx. We evaluate the system and show the experimental results in Section 4. Related work is presented in Section 5, and the conclusions in Section 6.

1.1 Background

We briefly introduce the standards and terminology about news feeds. Although several versions of news feeds specify formats that are compatible to a varying degree [33], they convey more or less the same content at a high level. A simplified sample of news feed is shown in Figure 1. A *feed* in Atom terminology (or channel in RSS terminology) is a place at which related entries are published and is identified by a URL from which feed documents are fetched. A feed *document* contains a list of entries (or items) as well as metadata about the feed itself such as the feed title and the published date. Each *entry* in turn contains a list of elements including the title of the entry, the link from which detailed information can be obtained, and the summary (or description) of the entry.

The news feed standards are concerned only with the document format. From a web server's perspective, fetching a feed document is the same as fetching a regular web document, using the unmodified HTTP. Thus, *subscribing* to a news feed does not mean that feed documents are delivered automatically upon a change. It merely means that subscribers fetch the corresponding URL repeatedly, either manually or through a client-side setup. Likewise, *publishing* a feed does not mean that publishers actually "push" documents to subscribers. It is subscribers and their applications that should ensure the timely update of news feeds. Nevertheless, these terms are used conventionally, and in this paper as well, to emphasize the dynamics of contents and the persistent behavior of readers regarding news feeds.

Starting as a means of syndicating web sites, the news feed technology has evolved so much as to be used in various ways. For example, Mozilla web browsers provide Live Bookmarks [1], which treat a feed as a folder and the contained entries as bookmarks in it, while Microsoft's new operating system, code-named "Longhorn," supports this technology from a broader perspective [7]. In this paper, we focus on its primary functionality, that is, delivering news summaries. In particular, we explore the potential of sharing news feeds among peers to expedite the dissemination and reduce the server loads.

1.2 Benefits of FeedEx

Currently, there are two ways of consuming news feeds. One way is using stand-alone applications, which look and work like traditional news readers or email clients except that posting is not possible. In fact, some email clients such as Mozilla Thunderbird support this functionality. Such applications, thus far, interact with nothing but the feed servers. Another way is using web-based services such as My Yahoo. If users register news feeds of their interests, they read them in one place provided by the web service. Allowing nodes to exchange news feeds with other nodes, FeedEx has several advantages over stand-alone and web-based aggregators:[1]

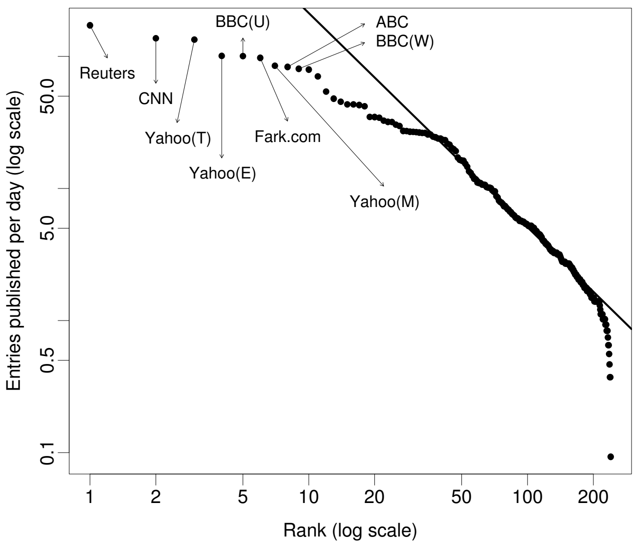
Server scalability. Since checking whether a feed is up to date costs no more than fetching a web document, nodes may well tend to do so frequently. However, fetching at a high rate from many subscribers can easily overload highly popular servers. In FeedEx, as nodes can receive new feed entries from their neighbors as well as directly from the servers, they can decrease the rate of checking, which relieves server load. FeedEx liberates the resource-constrained servers from being a victim of its own popularity. Although feeds forwarded through nodes may incur more traffic on the client side, our experiments show that the increased cost is minimal due to several techniques we use to reduce the flooded traffic.

Archivability. Since a feed document can contain only a limited, and often fixed, number of entries with new ones constantly published, the lifetime of an individual entry is also limited. Thus, subscribers that only have sporadic connections to the network for various reasons may wish to fetch the lost entries that cannot be obtained from the original server. FeedEx essentially forms a network of feed archives in that participating nodes store relevant entries locally for later reference, which allows users to retrieve the archived entries even when they are no longer available at the original servers.

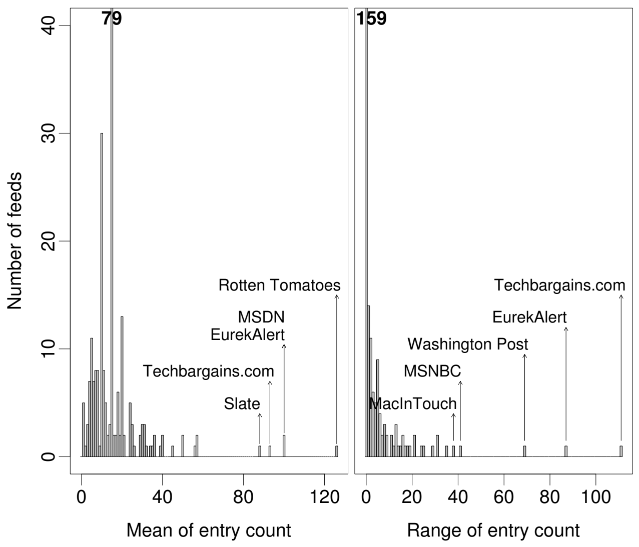
Controllability. Web-based services do not provide enough control or flexibility to users, often for the sake of their own scalability. For example, they usually forbid users to adjust the fetching intervals, restrict the number of entries to display or the length of each entry, and rarely provide the archive of past entries.

Filtering and recommendation. Users in FeedEx can tag their opinions on the entries they relay for filtering and recommendation. Recommendation can be done explicitly (e.g., rating or voting) or implicitly (e.g., user's reading can be interpreted as endorsement). In either way, they can help each other sift through the information flood in a grassroots way. In addition to entry recommendation, peers can recommend feeds to neighbors by comparing its subscription set with that of its neighbor. That is, if a peer finds that its neighbor has similar interests based on their subscription

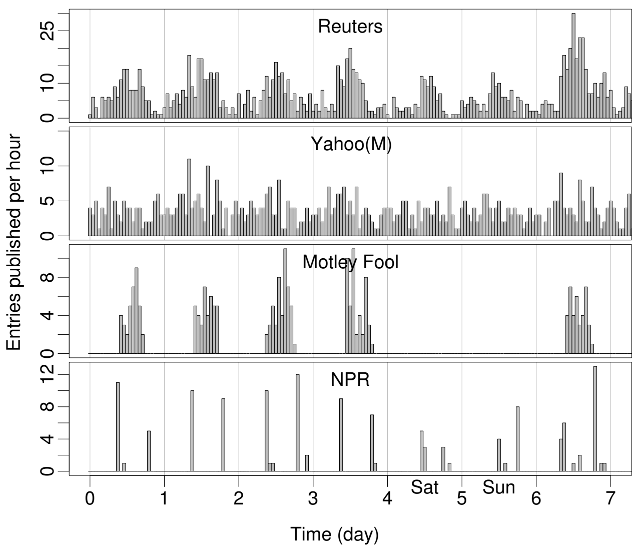
[1]In principle, both stand-alone applications and feed service providers can participate in the FeedEx network. In other words, FeedEx complements, rather than competes with, both ways.



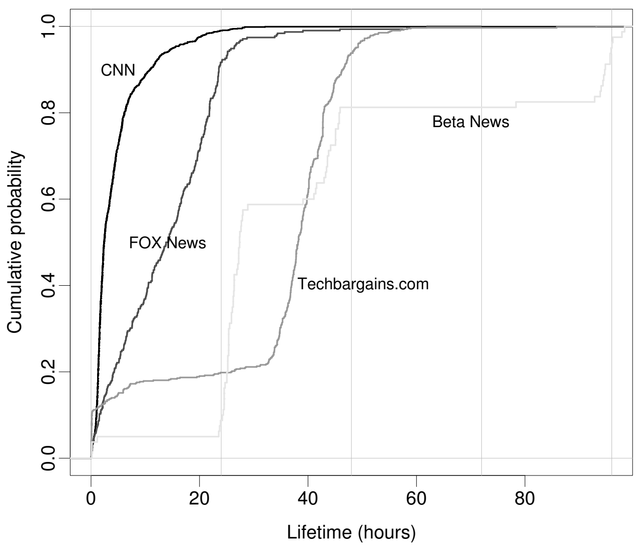
(a) **Distribution of publishing rate. The line is fitted between ranks 30 and 200.**



(b) **Histograms about entry count. Two long bars go up to 79 and 159, respectively.**



(c) **Change of publishing rate. Each bar is one hour wide. Notice different y-axis scales.**



(d) **Cumulative distribution of entry lifetime. Vertical guidelines indicate days.**

Figure 2: Statistics of news feed publishing

sets, it can try neighbor's other feeds that are not in its subscription set.

Privacy. Some users may not want to make public their subscriptions to certain feeds. While stand-alone applications cannot avoid exposing users to the servers, web-based services are even more vulnerable because all subscription and activity information is stored on the service providers. FeedEx can provide a framework for enhancing privacy through plausible deniability, somewhat analogous to onion routing or mixer-based request delivery [34, 16]. Plausible deniability is achieved by allowing users to fetch documents for others even if they may not need them. Enough indirection can anonymize actual consumers and obfuscate usage patterns, protecting from powerful adversaries that may breach user privacy.

As a first step towards exploring this new application, we focus this paper on the server scalability and the news entry availability.

2. ANALYSIS OF FEED PUBLISHING

We analyze various aspects of the current practice of news feed publishing. Understanding the current practice is not only interesting by itself but also helpful for a better design of FeedEx. Based on the popularity in Gmail's "web clips" and Bloglines' "most popular feeds," a total of 245 news feeds are chosen for monitoring. We fetch a document from each feed every two minutes in order to continually observe the activity. Entries in a document are identified by their identifiers (e.g., the `id` element for Atom or `guid` for RSS 2.0). If no identifier is provided by the feed, we use a pair of the `title` and the `link` elements as an identifier. We find 6 different versions[2] from 245 news feeds. With the data collected for 10 days, we analyze the following aspects of news feed publishing.

First, we characterize the distribution of publishing rates. The characterization into a certain distribution is useful, for example, to generate a synthetic model. As some feeds publish at a high rate while many others at a low rate, we hypothesize that the distribution follows Zipf's law, which states that the size or frequency q of an event of rank r is inversely proportional to r^b, where b is a positive constant, and r is an index (starting from one) into the list sorted in the non-increasing order [8]. Many quantities on the web seem to follow the Zipf distribution: the number of visits to a site, the number of visits to a page [14], and the number of links to a page [9], to name a few. Note that since $q = ar^{-b}$, for some constants a and b, if q is drawn against r on a log-log plot, a straight line should be observed. Thus, putting on a log-log plot is an easy, graphical way to see whether a distribution follows Zipf's law.

Figure 2(a) shows the distribution of publishing rates. Feeds are ranked in the descending order of publishing rate and then put on the log-log plot. While the middle range (from rank 30 to rank 200) is well fitted on the line, both ends deviate from the line. The upper left corner (head) is crowded by such sites as Reuters, CNN, BBC, and Yahoo's Top Stories, which tend to publish news articles similar in both amount and contents. It is also due to multiple active feeds from a single site. For example, Yahoo publishes Top Stories, Entertainment, and Most Emailed Stories, all at high rates. Given that we only plot popular feeds, which are highly likely to publish at high rates, the deviation in the lower right corner (tail) will be compensated for by a huge amount of less popular feeds.

Next, we show the distributions of entry counts for the news feeds. An entry count is referred to as the number of entries in a document. Instead of showing 245 histograms, we use two summarizing histograms shown in Figure 2(b). The histogram on the left shows the distribution of the arithmetic means of entry counts, indicating that many feeds contain 10 (in 30 feeds) or 15 (in 79 feeds) entries in a document on average. One feed, in contrast, contains as many as 126 entries on average. The histogram on the right shows the distribution of the ranges of entry counts. A range is defined as the difference between the maximum and the minimum entry counts for a feed. The histogram indicates that 65% of feeds have a fixed entry count, as their ranges are 0. Other feeds use variable length presumably to cope with the bursts of entries generated. Although not shown by a figure, there exists little correlation between the publishing rate and the entry count as the coefficient of determination R^2 from the linear regression is only 0.06.[3] The lack of correlation between the two variables implies that the entries from the feeds with high publishing rates should live shorter.

Third, Figure 2(c) shows the change of publishing rate over the first week of monitoring. For a better presentation, we select four feeds that represent distinct patterns. The vertical guidelines indicate midnights in EDT (4 hours behind UTC/GMT). In most feeds publishing at high rates, represented by the Reuters graph, the publishing rate is affected by the time of day. That is, publishing is more active during day than in night. By contrast, Yahoo's Most Emailed Stories does not show such periodicity because, we suspect, the feeds are generated by the feedback of users, who live in different time zones and whose activity is more dispersed. We also find that many feeds, like Motley Fool, do not show any activity during the weekends, which fall on days $[4, 6)$ in the graph. Another observation is that feeds such as NPR update feeds only at certain times of day. Although the predictability may be a good hint for subscribers, it is likely to cause "flash crowd" on such servers.

Last, Figure 2(d) shows cumulative distributions of entry lifetime from four selected feeds. The vertical guidelines mark the lengths of days. Most feeds publishing at a high rate show a similar distribution to that of CNN. Fox News and some other feeds exhibit linear progression up to a certain time. Note that this means a uniform distribution (flat range in a probability density function). We believe that it occurs as a result of the publisher's policy that controls entry lifetime. For example, the publisher may assign importance to each entry so that it lives as long as its importance. With variable-size documents as in FoxNews and TechBargains, it is easier to implement such a policy about the lifetime of entries. Beta News shows an interesting distribution in which entries' lifetimes are discrete with three levels (one day, two days, and four days long), which also seems to result from some policy.

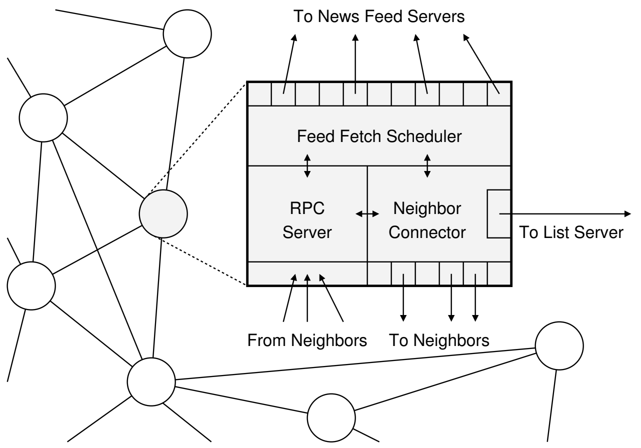


Figure 3: Architecture of FeedEx

3. DESIGN AND PROTOCOL

We describe the design and the operation of FeedEx, a news feed exchange system. As Figure 3 shows, a node is connected to a small number of neighbors. It occasionally, depending on the scheduler policy, fetches a document from a feed server to which it subscribes. If it finds new entries from the fetched document, it propagates them to the neighbors that are interested in them. At the same time, new entries may become available from a neighbor. In this case, it forwards new entries to other relevant neighbors.

[2]RSS 0.91 (two variants), 0.92, 1.0, and 2.0; and Atom 0.3

[3]A value, $0 \leq R^2 \leq 1$, close to 1 indicates a better fit. In our case, R^2 is the same as the square of correlation coefficient [36].

FUNCTION	ARGUMENTS	RETURN VALUE	SEC.
`get_neighbors`	`nid`	`(ip,port)` list	3.1
`connect`	`nid,subset`	`subset`	3.1
`update_subset`	`nid,subset`	*ok* or *error*	3.1
`check_did`	`nid,did`	*seen* or *unseen*	3.4
`put_entries`	`nid,bundle`	*ok* or *error*	3.4
`query_feed`	`nid,query`	`answer`	3.6

Table 1: XML-RPC function set. The argument `nid` stands for node identifier, `subset` for subscription set, and `did` for document identifier.

A *node* is identified by a pair of IP address and port number. The IP address of a node is taken implicitly by the neighbor from the connection, HTTP and hence TCP, information while the port number is stated explicitly by the requester. The port number indicates the listening port of the XML-RPC server. The implicitly taken address (using, for example, a `getpeername` system call on UNIX systems) helps prevent the Sybil attack [21] as it is difficult for an adversary to establish a connection through the 3-way handshaking of TCP with its identity (i.e., IP address) spoofed. Secure binding of identity is required for reliable accountability, which in turn helps build a robust system. We do not address issues caused by network address translation or other mechanisms that hamper the network transparency [15].

A *feed* is identified by the URL with which it is associated. Occasionally, more than one URL maps to the same feed. Such aliases are resolved externally as it is difficult to distinguish without human intervention. An *entry*, as discussed in Section 2, is identified by the `id` or `guid` element along with the URL of the feed. In the absence of such an identifier, a pair of the `title` and the `link` elements determines an entry.

A node advertises a *subscription set* to the neighbors. The subscription set contains a list of feeds in which the node is interested. Thus, the neighbors forward only the entries whose feed is in the advertised subscription set. Since the subscription set can contain feeds that are not in its interest but in its neighbors' (explained later), each feed in the set is tagged by an integer called *hop count* of interest. For example, hop count 0 means that the feed is in the advertising node's direct interest, hop count 1 means that the feed is not in its interest but in at least one of its neighbors', and so forth. A subset of a subscription set that contains only the feeds of hop count 0 is referred to as *direct subscription set*.

In the rest of this section, we describe the detailed protocol and operation of FeedEx. The protocol uses XML-RPC [37], and we refer to the relevant functions in Table 1 as we proceed.

3.1 Bootstrapping

Bootstrapping is a procedure through which a node becomes part of the system. It proceeds in two steps. The first step of bootstrapping is to acquire a list of nodes that are currently running in the system. The list can be obtained from three sources. First, a node can contact a well-known server that provides the list (using the `get_neighbors` call in Table 1). This list server is similar to the host cache service in Gnutella [17] and the tracker in BitTorrent [18]. The contacting node includes its information so that it can be included later in the lists for other nodes. Second, the feed servers may append to a feed document a list of clients that have recently retrieved the same feed. This idea, similar to Pseudoserving [28] and CoopNet [31], appeals to the servers because they are able to reduce the workloads in the long run by redirecting clients. Third, if a node has participated in the system before, it may reuse some of its earlier neighbors that are available at the moment. Reconnecting to old neighbors is advantageous in that they can trust each other from the past experience (see Section 3.5 for the incentive issue).

The second step of bootstrapping is to connect to the nodes whose locations are obtained in the first step. The `connect` call, shown in Table 1, carries the subscription set as an argument. Each element of the subscription set is a pair of the URL of a feed and the hop count of interest, as explained earlier. If the connected node Q decides to accept the connecting node P, Q returns its subscription set to P. Then, Q updates its subscription set including hop counts. Q's returning set must be computed excluding P's advertisement. Otherwise, Q would falsely advertise back the feeds in which only P, and none of Q's other neighbors, is interested. If Q decides not to accept P (for example, because Q's neighbor capacity is full), it returns an empty set, which is interpreted accordingly by P. Other reasons for rejection are discussed in Section 3.2.

As a node has its neighbors come and go, the subscription set changes over time. Thus, to update neighbors, the node advertises (using `update_subset`) its subscription set periodically and not immediately upon change. The immediate response to set changes may cause considerable traffic due to feedback loops. Since the direct subscription sets are always exchanged from the start and never change throughout the connection, the advertisement period can be set to a large value to reduce the overhead. Although feeds associated with large hop counts are more subject to change, they affect the overall performance to a less degree. To further reduce the overhead, the advertisement is incremental. That is, only the difference from the previous advertisement is transmitted. For the same reason as computing the returning subscription set, the advertised set must be computed for each neighbor excluding the neighbor in question. In the prototype implementation, this computation is done by a SQL query as we store the advertisements into a table of relational database. With proper indexing, the computation is efficient.

3.2 Neighbor Selection

Although having more neighbors may bring more information or bring it faster, it also causes higher overhead in communication and processing. Thus, a node should restrict the number of neighbors within a sustainable level. Since a node may have more neighbor candidates than it can sustain, it needs to select "good" neighbors. As a primary selection metric, we use the degree of overlap in subscription sets.[4]

Initially, a node P assigns a weight w_i of preference to each feed i in its direct subscription set. Periodically, P

[4]It can be supplemented with secondary metrics such as topological proximity and neighboring duration.

assigns each neighbor Q the degree of usefulness $u(Q)$:

$$u(Q) = \sum_{i \in (S_P \cap S'_Q)} w_i d^{-h_i}$$

where $S_P \cap S'_Q$ denotes the intersection of P's direct subscription set and Q's entire subscription set; d is a positive constant; and h_i is a hop count, with respect to Q, for feed i. The purpose of a factor d^{-h_i} is to depreciate the value of overlap in inverse proportion to the hop count because a feed with a large hop count is more prone to disappear.

The assigned usefulness values are used to decide which neighbors to keep or drop. A node may need to drop some of current neighbors, for example, when it encounters shortage of network bandwidth or processing power or when a newly connecting node has a higher degree of usefulness.

3.3 Adaptive Fetching

If nodes do not fetch feeds frequently enough, they may obtain entries too late or even miss them. On the other hand, if fetching is too frequent, it may well waste the network bandwidth and overload the servers. Thus, it is important to balance the frequency of fetching that is appropriate for both subscribers and publishers. However, the coordination among nodes is difficult because it involves a large number of nodes that join and leave the system at a possibly high rate. Another challenge is that publishers give little explicit hint about the publishing rate or entry lifetime. Some hints may even be misleading. In our analysis, for example, some feeds fill the `pubDate` element with the fetching time. That is, although the actual contents remain unchanged, the element keeps changing each time a document is fetched.

We develop an adaptive algorithm that adjusts the fetching intervals without explicit coordination nor any hints from the servers. The intuition behind the algorithm is that if fetching is too frequent, a fetched document contains few new entries. On the other hand, if fetching is too infrequent, most of the entries are likely to be new. Thus, we keep track of the fraction of new entries in a fetched document and use this feedback to adapt the fetching interval.

We define a *freshness rate* of a fetched document as the fraction of new entries contained in it. Each feed needs a separate fetching interval because the publishing rate and pattern vary greatly as shown in Figure 2. For each feed, a node assigns a value to the target parameter f_t for freshness rate ($0 \leq f_t \leq 1$). A value f_t that is close to 1 means that a node wishes new entries as early as possible. Two parameters, $T_{\min}$ and $T_{\max}$, denote the minimum and maximum possible intervals, respectively. We use a simple algorithm to adjust the fetching interval t of a feed. Each time a document is fetched, the freshness rate f is computed as the ratio of the new entries to the total entries in the document. If $f < f_t$, then t becomes halved or set to $T_{\min}$, whichever is the greater. Or if $f > f_t$, then t becomes doubled or set to $T_{\max}$, whichever is the less.

3.4 Entry Dissemination

After a node fetches a document from a feed server, it filters and stores new entries from the document. These new entries are bundled and forwarded to the neighbors that subscribe to the corresponding feed. The *entry bundle* is assigned a globally unique identifier `did`. To assign `did`, we use the 20-byte SHA-1 digest [30] of a bundle. Although there is a very slight chance of collision, the cost of the collision is missing an entry bundle whose entries can be available later from other bundles. For the small cost, we can avoid coordination for a unique identifier. The entry bundle also attaches a path attribute, which keeps records of a growing list of forwarders. It is a receiver, rather than a sender, that puts the forwarder on the path list in order to reduce the chance of undesirable modification of the path. Thus, the sender, although it may alter the past path, cannot avoid appearing on the list in the absence of collusion. Even with collusion, at least one of such a forwarder must appear on the list.

Forwarding is triggered either from inside (the feed fetch scheduler in Figure 3) or from outside (the RPC server). If a node detects a loop (that is, its identifier is already included in the path attribute), the entry bundle is discarded. The bundle is forwarded to a neighbor only if the neighbor is interested in the bundle, which means that the neighbor's subscription set includes the bundle's originating URL and that the hop count for the matching feed is not greater than a system parameter `max_subset_hops`. Forwarded bundle may contain entries that have already been stored locally. Those old entries are removed from the bundle before it is further forwarded. Forwarding involves two remote calls that helps reduce the wasted traffic. First, using a `check_did` call, small enough to fit in one packet as shown in Section 4.4, the forwarding node checks whether a neighbor has already seen this entry bundle. For this purpose, nodes store the bundle identifiers `did` that have been seen recently (e.g., 1,000 latest `did`s in our prototype). While simple hashing is sufficient for searching a small amount of `did` cache, Bloom filter [13] may be used for a large cache [23]. Second, if the neighbor returns "unseen," the entry bundle is forwarded via a `put_entries` call.

3.5 Incentive Mechanism

Due to resource constraints and the lack of centralized administration, nodes may manifest selfish behavior. For example, they may want to save the network bandwidth by only receiving the entry bundles without forwarding them. As another example, they may lie about the subscription set to become a preferred neighbor. Without proper incentive mechanisms and the detection of misbehavior, the system may become full of free riders and suffer from the "tragedy of the commons" [26] in the long run.

To ensure the mutual contribution, we measure the degree of contribution from a neighbor. When a node i receives a new entry from a neighbor j, i updates the contribution metric $c_{j,i}$ (contribution from j to i):

$$c_{j,i} \leftarrow c_{j,i} + w_f \alpha^{-h_f}$$

where α is a system-wide constant, f is the feed to which the entry belongs, w_f is a weight of preference for feed f, and h_f is a hop count for feed f. That is, a node gives the most credit when it receives an entry from a feed belonging to the direct subscription set, the second most credit for an entry from a feed of hop count 1, and so forth. Then, we define the *deficit of contribution* $d_{i,j}$ of node i against node j as follows:

$$d_{i,j} = c_{i,j} - c_{j,i}$$

Note that i can maintain both $c_{i,j}$ and $c_{j,i}$ without relying on j. To prevent free riding, node i ensures that $d_{i,j}$ does not exceed a constant parameter D. The effectiveness of this deficit bounding is shown in our previous work [27]. Although it is discussed in the context of bulk transfer, the same principles apply to this case as well.

3.6 Entry Pulling

Since nodes store entries into their database, FeedEx forms a distributed archive system. If a node is to retrieve past entries that are no longer available from the feed servers, it can rely on other nodes that have those entries stored. The `query_feed` call serves the purpose of finding such nodes. It works similarly to the Gnutella's `query` and `query_hit` pair [17]. The requester specifies in the query what feed entries it is looking for in terms of feed titles, entry titles, published dates, and others. The query is propagated recursively rather than iteratively. That is, analogous to the recursive DNS query, once a neighbor receives a query, it propagates the query to its neighbors on behalf of the original requester. A possible drawback of recursive mode is excessive traffic caused by query flooding, which can be similarly controlled using the unique query identifier and the maximum number of hops. On the other hand, recursive queries have three advantages over iterative queries. First, the results can be aggregated along back to the original requester. Second, results can be cached, which may be useful for popular queries. Third, and most important, we can put this query relaying under the incentive mechanism discussed earlier. If iterative queries are used, nodes do not have incentive to answer the requests from non-neighbors because answering is unlikely to be rewarded.

The original requester C selects a node S from the query results for downloading the entries. Unlike the `query_feed` call, because of potentially large traffic, C and S make a direct connection to retrieve entries rather than communicating through a chain of neighbors. However, S has no incentive to upload entries to C as it only costs S its resources (i.e., network bandwidth). Thus, C needs to find either an altruistic S that unconditionally uploads to C or a circular dependency of requests. For example, if S also wants to download some other entries from C, S and C form a circular dependency of length 2. Or if there exists a node P such that P wants to download from C, and S from P, they form a circular dependency of length 3. If a circular dependency is found, they are likely to agree to serving one in exchange of being served by another. Anagnostakis and Greenwald [10] discuss how to detect circular dependencies and perform n-way exchanges.

4. EVALUATION

We evaluate FeedEx using various metrics in comparison with stand-alone applications. We also measure the overhead of FeedEx that is caused mainly by forwarding entry bundles.

4.1 Prototype Implementation

For the proof of concept and the evaluation, we have implemented a prototype in Python [4]. As mentioned earlier, the protocol is implemented using XML-RPC for interoperability in the future as well as fast prototyping. The communication and the concurrency are due to Twisted, an event-driven networking framework [6]. The feed entries and the subscription sets are stored into tables of relational database. For this purpose, we use SQLite [5], a lightweight embedded relational database engine, which is accessed via Python's DB-API 2.0 [3]. Feed documents are parsed by Universal Feed Parser [32].

Parameter	Value
Subscription set size	20
Maximum subset hops	3
Advertisement interval	300 seconds
Maximum neighbors	10
Minimum neighbors	8

Table 2: Experimental parameters

4.2 Metrics

To measure the performance of news feed delivery systems, we define the following performance metrics:

Time lag. We define the time lag for an entry as the time difference between when the entry is published at a feed server and when it becomes available to a node (either directly from the server or from a forwarding neighbor).

Missing entries. We refer to a missing entry as the one that has been published at a feed server but that has never been available at a node. Only entries from the subscribed feeds are considered.

Communication cost. Communication cost for a node includes the cost of fetching feed documents from servers and, in exchange mode, the cost of the XML-RPC calls between neighbors.

In a sense, a missing entry is an entry with an infinite time lag. However, we treat two metrics separately for easy summarization because mean and variance are computable only without infinite numbers.

Measuring both time lag and missing entries requires the publishing logs from servers. Since such data are difficult to obtain, we instead run a *reference node* that monitors the servers' activity during the experiment. The reference node runs in stand-alone mode, fetching all monitored feeds every two minutes. Thus, in a failure-free environment, the maximum possible error in time lag is two minutes while entries that live shorter than two minutes long may even miss at the reference node. The effect of failures in the reference node is discussed in Section 4.4.

In stand-alone mode and, to a lesser degree, exchange mode, a node can reduce both time lag and missing entries by fetching documents more frequently. On the other hand, fetching document too often increases the communication cost as well as the server load. In exchange mode, the feed exchange involves controlled flooding of entry bundles, it is essential to keep the communication cost to a sustainable level. Thus, a good system should "score high" on these metrics in a balanced manner.

4.3 Experimental Setup

We used PlanetLab [2] for evaluation as it provides a platform for experimenting with machines distributed worldwide. Specifically, we ran our prototype on 189 PlanetLab machines for about 22 hours on a weekday. PlanetLab machines tended to shut down or reboot frequently not least

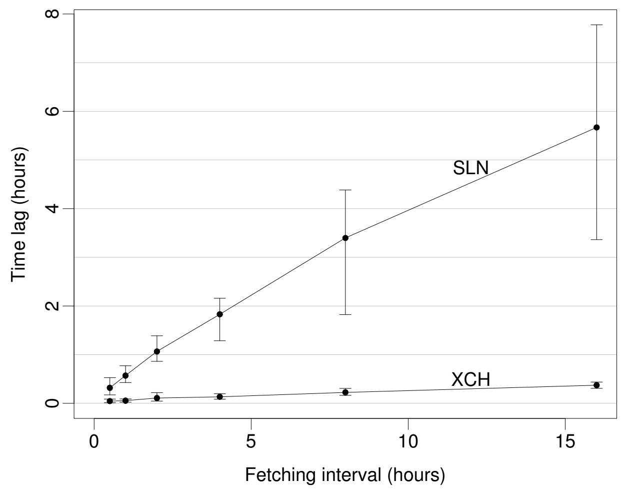


Figure 4: Time lag

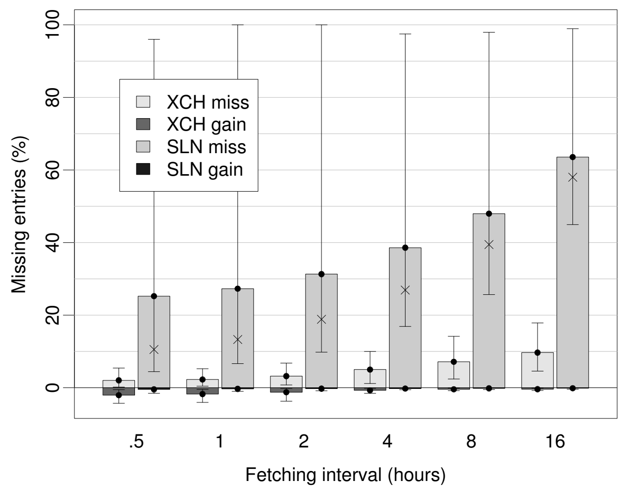


Figure 5: Missing entries

because they were shared and overloaded by many users. As a result, during those 22 hours, 28 machines were shut down or rebooted, and we report the results from the remaining 161 machines that ran during the whole experiments. Nodes from the rebooted machines did not re-join the network. As a FeedEx node is able to detect neighbor failures and act accordingly (e.g., remove unreachable nodes and replenish new neighbors if there remain less than `min_peers` neighbors), we believe that the failed machines did not affect the results in any significant way.

The primary experimental factor is the fetching interval, which most affects all three performance metrics. We choose 6 intervals from 30 minutes up to 16 hours with each subsequent interval doubled. To factor out the effect of fetching interval, one FeedEx network consists of the nodes having the same interval. For the same reason, we do not apply adaptive fetching interval algorithm discussed in Section 3.3. We run 6 networks, each with a different interval, in parallel. That is, each *machine* runs 6 *nodes* of different intervals during the experiment. We strictly distinguish the two terms, machine and node, in this section. While different networks have no direct interaction with each other, we believe that they should not interfere much because they do not consume much processing power or network bandwidth. As the experiment runs about 22 hours long, we select 70 feeds, out of the same 245 feeds as shown in Section 2, that publish at least 5 entries per day. Out of these 70 feeds, each node puts 20 feeds independently and randomly into its direct subscription set. The direct subscription set does not change during the experiment.

For each fetching interval, two delivery modes, SLN (for stand-alone) and XCH (for exchange, representing FeedEx), are compared. The results of SLN are simulated from those of XCH. That is, no SLN nodes are actually run, but the results such as time lag and missing entries are computed from those of XCH by excluding the effects from neighbors. Note that since the feed exchange does not interfere with the document fetch, the simulated results are exactly the same as if they were run in stand-alone mode. Since the SLN and the corresponding XCH nodes have the same subscription sets (i.e., the factor, subscription set, is blocked), the results from the two modes are directly (i.e., pairwise) comparable.

4.4 Results

Figure 4 shows how time lags change for different modes and fetching intervals. Classified by mode and interval, each group consists of 161 nodes. Each node collects some number of entries during the experiment. For each entry collected, we compute its time lag as defined in Section 4.2. Then, the average time lag of entries is computed for each node. The distributions of the node averages are shown in the figure. That is, each point in the figure is the average of the node averages while error bars indicate the 90% range of the node averages with the lower bars corresponding to the 5th percentile of node averages, and the upper bars to the 95th percentile.

The gap between SLN and XCH grows as the interval increases. The time lag of SLN is 7.0 times that of XCH for the 30-minute interval while the ratio jumps to 15.2 times for the 16-hour interval. Wide error bars in SLN indicate that it has much more variation in time lag than XCH and that the time lag variation of SLN increases as the fetching interval increases. As such an increase does not occur to XCH, it can be said that its performance is more stable.

For stand-alone mode, the expected time lag is half the fetching interval if entries are published uniformly. The expectations are close to the actual results for smaller intervals whereas they are greater for larger intervals. For example, the expected time lag of 8 hours for the 16-hour fetching interval is greater than the node average 5.7 hours. This discrepancy occurs because as the experiment period (22 hours) is not a multiple of the fetching interval (16 hours), some feeds fetched twice during the experiment lead to the time lag shorter than expected.

Figure 5 shows the percentage of entries that each group misses on average. To compute the missing entry rate, we compare, for each node, the set of entries stored in its database with that of entries stored in the reference node for the same subscription set. A group average is computed as an average of the node missing rates in the same group. As in the previous figure, each point indicates the average of the node averages for the group. As all the SLN miss groups include outliers with high missing rates (the reason is discussed shortly), for them, medians are indicated by the "x" marks. Error bars are drawn from the 5th percentile to the

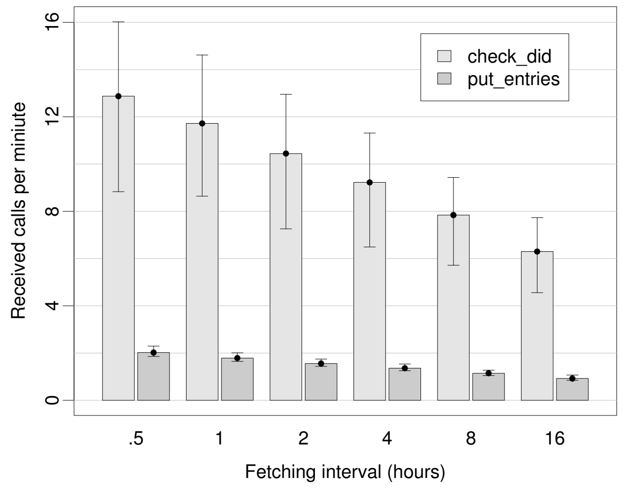


Figure 6: Frequency of remote call invocations

95th percentile of the node missing rates. For each fetching interval, two "miss" bars grow above while two "gain" bars grow below, both starting from 0%. An entry that appears in the reference node but not in a XCH node accounts for XCH miss while an entry that appears in a XCH node but not in the reference node accounts for XCH gain (and similarly for SLN miss and SLN gain).

While miss rates increase along the increased fetching interval for both modes, XCH remains much lower rate than SLN. For the interval of 30 minutes, XCH nodes even manage to obtain entries that the reference node has missed (about 2% of the total number of entries collected by the reference node). Some of these entries may have lived less than two minutes, the fetching interval of the reference node, while other entries are missed because of the temporary failures in the reference node. As the failures were mostly timeouts and not temporally clustered, we believe that they were due to the servers, most likely overloaded servers, rather than the reference node. In any case, the failures in the reference node were so few and far between that they do not affect time lags and missing rates in any significant way.

There are 27 machines (hence all 162 nodes running on those machines) showing more than 80% SLN missing rate because most fetching attempts failed during the experiment. All those troubling nodes belong to the next-generation Internet. Specifically, 15 machines belong to the `internet2.planet-lab.org` domain (Internet2), 9 machines to `canet4.nodes.planet-lab.org` (CANARIE), and 3 machines to `hpl.hp.com` (HP Labs' Internet2 machines). At the same time, no machines that fetch documents properly belong to these domains except for some `hpl.hp.com` machines that are not part of Internet2. Thus, we deduce that the particular network as a whole had a problem in name resolution or routing during the experiment.[5] Still, since such machines were able to communicate with neighbors, the XCH miss rates on those machines were not affected as much. This event, although anecdotal, illustrates that FeedEx is more resilient to certain types of failures than stand-alone clients as it can obtain information not only from the feed servers but also from neighbors.

[5]In the log files, all fetching failures were logged as timeout. Since they did not have a problem in communicating other machines, which did not require the name resolution, we suspect that name resolution was a more likely cause.

Fetching interval	.5	1	2	4	8	16
Bundle size	2.67	2.63	2.83	2.87	3.29	3.65

Table 3: Average entries in a bundle

Figure 6 shows the rate of XML-RPC calls as a measure of communication cost of FeedEx. Out of 6 functions in Table 1, only `check_did` and `put_entries` are shown. The invocations of the other functions are so infrequent (less than a total of 50 times per node for 22 hours) that we do not include them in the figure. The `check_did` call, which is an HTTP request, is 344 bytes long, including the HTTP header, and the call return, which is an HTTP response, is about the same size. Since IPv4 specifies the maximum transfer unit to be at least 576 bytes (and 1,280 bytes for IPv6) except for few special cases, both the request and the response should fit in one IP packet. Thus, the overhead from the `check_did` is small. The rate of `put_entries` calls is as low as two calls per minute even when nodes fetch documents every 30 minutes. Since each call contains only 2.67 entries on average, shown in Table 3, we conclude that the communication cost is low.

Table 3 shows average bundle size by fetching interval. The bundle size is measured as the number of entries in an entry bundle. From the table, we see the average bundle size increase as fetching interval increases because it is likely to have more new entries available for the increased interval. Compared with the entry count distributions shown in Figure 2(b), entry bundles are smaller than actual feed documents because only new entries are included before forwarded.

Putting together the experimental results, we see that FeedEx has low communication overhead while achieving short time lag and low missing rates.

5. RELATED WORK

Web caching and content distribution network address the similar goals of relieving the server load and reducing the latency for clients to retrieve web pages. Various approaches have been researched, including recent peer-to-peer flavors [24, 35]. FeedEx is different from web caching or content distribution networks in that there is no distinction between clients and proxies or content networks. That is, a peer in FeedEx plays dual roles as a consumer and as a cache. Such duality creates bidirectional service and gives an advantage to making it incentive-compatible.

FeedEx can be considered a gossip-based protocol in that a peer delivers the information it has learned to the neighbors. Gossip-based or epidemic protocols generally achieve robustness and scalability due to their distributed nature of dissemination [12, 25, 22]. Unlike some epidemic protocols, a FeedEx peer adheres to its neighbors, rather than changes them for each retransmission, because repeated transactions increase the chance to establish trust with the neighbors. With each pair of neighbors trying to be "fair" to each other, the system becomes robust to free riders.

The LOCKSS system [29] preserves digital contents by periodic voting. That is, nodes ensure the integrity of contents they own by comparing their fingerprints with those from neighbors, with possible human intervention in case of no definite voting result. Developed in the context of digital library, LOCKSS addresses copyright issue by requiring that a peer must own the contents before participating in voting. Thus, while it is not concerned with the propagation of new contents, it proposes a way of protecting integrity, which can apply to FeedEx.

Since peer-to-peer systems are prone to free riding, it is important to ensure the contribution of peers. The techniques to ensure the fairness and provide the incentives are developed in such various contexts as storage [20], bulk transfer [18, 27], and data archiving [19]. As FeedEx also requires the cooperation of peers in propagating news feeds, it provides an incentive mechanism as shown in Section 3.5.

6. CONCLUSIONS

We make a case for collaborative exchange of news feeds by presenting FeedEx. By enabling nodes to exchange news feeds, it reduces the time lag and increases the entry coverage, helping the server scalability. We also emphasize that FeedEx is incentive-compatible so that cooperation is elicited. Without proper incentive mechanisms, the system becomes unsustainable due to the unwantedly induced free riders.

While we demonstrate the scalability and efficiency of FeedEx, it also has potential for other benefits such as anonymous subscription and collaborative filtering and recommendation. We plan to investigate further so as to augment FeedEx, which will help exchange information that grows increasingly unwieldy.

7. REFERENCES

[1] Live bookmarks. `http://www.mozilla.org/products/firefox/live-bookmarks`.

[2] PlanetLab. `http://planet-lab.org`.

[3] Pysqlite. `http://pysqlite.org`.

[4] Python. `http://python.org`.

[5] SQLite. `http://sqlite.org`.

[6] Twisted. `http://twistedmatrix.com`.

[7] Microsoft to deliver RSS support to end users and developers in windows "Longhorn". `http://www.microsoft.com/presspass/press/2005/jun05/06-24RSSIntegration%PR.mspx`, June 2005.

[8] L. A. Adamic. Zipf, power-law, Pareto—a ranking tutorial. `http://www.hpl.hp.com/research/idl/papers/ranking/ranking.html`, Oct. 2000.

[9] R. Albert, H. Jeong, and A.-L. Barabasi. Diameter of the world-wide web. *Nature*, 401, 1999.

[10] K. G. Anagnostakis and M. B. Greenwald. Exchange-based incentive mechanisms for peer-to-peer file sharing. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 524–533, 2004.

[11] BBC News. One blog created 'every second'. `http://news.bbc.co.uk/1/hi/technology/4737671.stm`, Aug. 2005.

[12] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Science*, 17(2), May 1999.

[13] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[14] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evedence and implications. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, Mar. 1999.

[15] B. Carpenter. RFC 2775. internet transparency, Feb. 2000.

[16] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, Feb. 1981.

[17] Clip2 Distributed Search Services. The Gnutella protocol specification v0.4.

[18] B. Cohen. Incentives build robustness in BitTorrent. In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems (P2PECON)*, June 2003.

[19] B. F. Cooper and H. Garcia-Molina. Peer-to-peer data trading to preserve information. *ACM Trans. Inf. Syst.*, 20(2):133–170, 2002.

[20] L. P. Cox and B. D. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[21] J. R. Douceur. The Sybil attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[22] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermacrrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Science*, 21(4):341–374, Nov. 2003.

[23] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[24] M. J. Freedman, E. Freudenthal, and D. Mazieres. Decomcratizing content publication with Coral. In *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2004.

[25] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2):139–149, Feb. 2003.

[26] G. Hardin. The tragedy of the commons. *Science*, 162:1243–1248, 1968.

[27] S. Jun and M. Ahamad. Incentives in BitTorrent induce free riding. In *Proceedings of the ACM SIGCOMM Workshop on Economics of Peer-to-Peer Systems (P2PECON)*, pages 116–121. ACM Press, Aug. 2005.

[28] K. Kong and D. Ghosal. Mitigating server-side congestion in the Internet through pseudoserving. *IEEE/ACM Transactions on Networking*, 7(4):530–544, 1999.

[29] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. H. Rosenthal, and M. Baker. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Science*, 23(1):2–50, 2005.

[30] National Institute of Standards and Technology. Secure hash standard. FIPS PUB 180-1, May 1993.

[31] V. N. Padmanabhan and K. Sripanidkulchai. The case for cooperative networking. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[32] M. Pilgrim. Universal feed parser. `http://feedparser.org`.

[33] M. Pilgrim. The myth of RSS compatibility. `http://diveintomark.org/archives/2004/02/04/incompatible-rss`, Feb. 2004.

[34] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, May 1998.

[35] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Proceedings of the USENIX Annual Technical Conference*, pages 171–184, 2004.

[36] E. W. Weisstein. Correlation coefficient. From MathWorld at `http://mathworld.wolfram.com/CorrelationCoefficient.html`.

[37] D. Winer. XML-RPC specification. `http://xmlrpc.com/spec`.