RICE UNIVERSITY

FeedTree: Scalable and prompt delivery for Web feeds

by

Daniel R. Sandler

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

Approved, Thesis Committee:

Peter Druschel
Professor of Computer Science

Dan S. Wallach
Associate Professor of Computer Science

T. S. Eugene Ng
Assistant Professor of Computer Science

Houston, Texas

April 2007

ABSTRACT

FeedTree: Scalable and prompt delivery for Web feeds

by

Daniel R. Sandler

An increasing number of Internet users now use *Web feeds* (or RSS feeds) to get their news, hear music and audio programs, and keep in touch. The result for website owners, however, is known as the "RSS bandwidth problem": because each feed reader polls every subscribed feed repeatedly for updates, the bandwidth demands of hosting a popular feed can be extreme. Our FeedTree system replaces this polling architecture with efficient and scalable application-level multicast based on the Pastry peer-to-peer overlay network. Instead of independently polling feed resources, FeedTree users cooperate to distribute feed updates; they substantially reduce the bandwidth burden placed on feed publishers, while updates arrive *faster* than with polling. In this thesis we explore the existing problems with Web feeds and describe the design and implementation of the FeedTree solution. We also share our experiences deploying FeedTree as a public Internet service.

ACKNOWLEDGEMENTS

Foremost, this work owes its existence to the ideas, encouragement, and critical eye of my advisor, Dr. Peter Druschel. Under Peter's influence I quickly developed a fascination with peer-to-peer systems; within a couple of months in his seminar and working with his team, I began to see applications of Pastry everywhere I looked. When I proposed using it to combat the problems with RSS, he prodded me to develop and sophisticate my ideas, resulting directly in this work. Under his careful supervision I received my first opportunity to conduct serious research, and for that I am tremendously grateful.

I would also like to thank Alan Mislove and Ansley Post, co-authors on the original FeedTree workshop paper, who helped me acclimate to graduate school, speak in the voice of research papers, and survive my first paper submission (re-running `pdflatex`, `cvs commit`, and `HTTP POST` right up until midnight).

Thanks to Dr. Dan Wallach, who has generously offered feedback and help during the writing of this document, even though it is only tangentially related to his current research. I also owe thanks to Dr. Eugene Ng, the final member of my committee, who graciously agreed to review this work.

To my close friends, Drs. Dennis Geels, Kevin Hwang, and Jeremy Templeton, who cheerfully encouraged me to enter graduate school: Thanks a lot.

I owe sincere thanks to my family, who—through genuine interest or kind indulgence, and they need not confess which—read through multiple drafts, ferreting out typos and confusion.

Finally I must attempt to sufficiently thank my kind and brilliant wife Erin. I strive to write as clearly and fluidly, to argue as concisely and convincingly, and to use adverbs as sparingly as she.

CONTENTS

1 Introduction **1**

2 Background **4**

2.1 Publish and subscribe 4
2.2 Early Internet pub-sub 5
2.2.1 Usenet 5
2.2.2 PointCast 6
2.3 Web feeds: RSS and Atom 7
2.4 Feed bandwidth 8
2.5 Weblog pings 10
2.6 Peer-to-peer overlay networks 10

3 Related work **13**

3.1 Current feed bandwidth remedies 13
3.1.1 Optimizing HTTP requests 13
3.1.2 Make it somebody else's problem 14
3.2 Alternative solutions 14
3.2.1 Bandwidth savings with Web caches 14
3.2.2 Cooperative feed polling 15
3.3 Other applications of structured p2p overlays 17

4 Design and Implementation **19**

4.1 FeedTree participants 19
4.1.1 Publishing updates 19
4.1.2 Receiving updates 21
4.2 Implementation 21
4.2.1 The FeedTree Core 22
4.2.2 FeedTree Web Proxy 24
4.2.3 Cooperative polling in the proxy 25
4.2.4 FeedTree Publisher 27
4.3 Benefits for participants 27
4.4 Overhead 28
4.4.1 Maintenance 28
4.4.2 Bandwidth and incentives 28

4.5 The importance of open membership . . . 29
4.6 Security . . . 29
4.6.1 Signed feed updates . . . 29
4.6.2 Recovery of lost data . . . 30
4.6.3 Additional possible defenses . . . 30

5 Evaluation **33**
5.1 Synthetic experiments . . . 33
5.1.1 CPU . . . 34
5.1.2 Network . . . 35
5.2 feedtree.net: The global FeedTree network . . . 39
5.2.1 Centralized components . . . 39
5.2.2 Distributed components . . . 39
5.3 Data collection . . . 40
5.4 Analysis . . . 42
5.4.1 Subscription popularity . . . 42
5.4.2 Network traffic . . . 42
5.5 Summary . . . 47

6 Conclusions **48**

LIST OF FIGURES

2.1 Timeline of feedtree.net HTTP bandwidth. 9

4.1 FeedTree system overview. 20
4.2 FeedTree software architecture.. 22
4.3 FeedTree messages.. 24
4.4 FeedTree Web Proxy: web-based interface. 26

5.1 Experiment: CPU usage under synthetic load. 34
5.2 Experiment: CPU usage (detail). 35
5.3 CPU usage vs. update rate. 36
5.4 Experiment: Bandwidth of node under synthetic load. 37
5.5 Experiment: Bandwidth vs. update rate.. 37
5.6 Experiment: Network growth.. 38
5.7 FeedTree adoption. 40
5.8 A FeedTree Scribe tree.. 42
5.9 Subscription popularity. 43
5.10 Network traffic in the public FeedTree deployment. 45
5.11 Bandwidth of polling nodes.. 46
5.12 Slashdot RSS traffic vs. FeedTree.. 47

LIST OF TABLES

5.1 Sample feed workload. 44

1 INTRODUCTION

The World Wide Web, as a platform for distribution of timely news and information, has evolved greatly from the static HTML of its earliest days. Initially, a handful of news-oriented Web sites of broad appeal updated their content once or twice a day; readers were by and large able to get all the news they needed by surfing one-by-one to their favorite sites (and pressing `Reload`). However, the Web today has experienced an explosion of *micronews:* highly focused chunks of content, appearing frequently and irregularly, scattered across news websites and personal weblogs. The problem, for readers, is harnessing this flow of timely information without having to reload scores of sites many times a day, searching for new and updated items.

This surge of content has spurred the adoption of Web feeds, which marshal micronews into a common, convenient format. Instead of downloading entire web pages, specialized feed-reading applications download a XML-based "feed" (in a format like RSS or Atom) containing a list of recent articles. An increasing number of users now rely on their feed readers to track breaking news, keep in touch with friends, and even listen to podcasts (periodic radio-style audio programs).

The technology is still quite young, and in the early stages of adoption. Nine out of ten Internet users don't know what RSS is [44], including many who are actually already users (thanks to the transparent involvement of feeds in personalized portal pages like my.yahoo.com and downloadable "news ticker" software programs). Use of Web feeds is steadily increasing, however, thanks to the efforts of website operators and authors, who are eager to offer up-to-the-minute content to their readers. Orange “ ” icons and “RSS” links, now featured prominently on most major news websites and almost every weblog in existence, announce the availability of feeds for the casual reader and news junkie alike.

These same publishers have discovered that adoption of this particular technology carries with it a substantial price in terms of networking resources. Because feed reading applications check for new entries by repeatedly requesting each feed from its canonical URL, the bandwidth requirements of hosting a feed can be quite large—disproportionately so when compared with other popular Web resources of similar file size. At the same time, end users have every incentive to exacerbate network stress on publishers. They have chosen to use RSS because it promises convenient and timely updates, so users who elect to poll feeds even more frequently than the *de facto* standard of once per hour will only make the situation worse. [56]

We see an opportunity to address this problem now, while feed scalability is merely an annoyance and not yet a strong deterrent for publishers considering Web feed service for their readers. In this thesis we describe a solution called FeedTree that improves service for all parties, delivering updates faster to users while *reducing* network strain for publishers. This document will explore in detail the design and implementation of that system.

FeedTree users participate in a peer-to-peer overlay network in which they cooperate to redistribute micronews among themselves rather than independently requesting updates from a feed’s Web server. A small subset of any one feed’s subscribers will undertake the burden of polling that feed for the group; when feed updates are discovered, they are forwarded to interested parties using an efficient distributed multicast system. The cooperation of multiple such volunteers reduces the polling update delay for all subscribers.

We have developed software to provide the FeedTree service to end users. The implementation builds on current Web feed standards and software; it works with all existing RSS feeds and runs alongside all existing desktop feed reading applications. By creating an open-membership distributed system in which each user’s own computer participates, we ensure that our system’s capacity grows along with the actual user base without need for a similarly increasing investment in server infrastructure.

Publishers may also elect to participate in FeedTree by "pushing" new feed data to FeedTree subscribers at the time of publication. In this way publishers can eliminate all client polling and further reduce the update delay to seconds. We have also developed software to provide this service to current publishers, requiring nothing more than an existing RSS or Atom feed.

We will structure the remainder of this thesis as follows. Chapter 2 places FeedTree in context of other publish-subscribe systems on the Internet and of feeds in particular. Chapter 3 investigates contemporary approaches to improving Web feed distribution. We describe the design of the FeedTree system in detail in Chapter 4, and share our observations and measurements of the system under the real workload of Internet users in Chapter 5. Finally, we offer our conclusions and future explorations in Chapter 6.

2

BACKGROUND

Web feeds are a platform for one-to-many communication, after a model known as "publish and subscribe" (or "pub-sub"). They are not the first pub-sub solution for the Internet, but they have achieved remarkable adoption (despite the technical complexity of creating and consuming them) due to their remarkable ability to combine fragments of timely information from disparate sources in a common format suitable for streamlined presentation to the user. FeedTree represents an evolution of this distribution system, bringing techniques from the research community to bear on the problems with feeds. To understand its contribution, we will explore the space of pub-sub systems in the literature and news distribution systems on the Internet. Finally, we will describe fully the problems with today's Web feeds as well as the limitations in current proposals to address those problems.

2.1 Publish and subscribe

The distributed systems literature features a wealth of group communication systems fitting the publish-and-subscribe (or "pub-sub") model. The ISIS [3] toolkit is an early example of group communication in distributed systems[1], and introduced the notion of subscription to all events matching a fixed "topic." Pub-sub systems have since come to be subdivided into as *topic-*

[1]The guarantees offered by ISIS are stronger—and costlier—than we have need of when dealing with Internet news delivery. We introduce it here to establish ancestry for publish and subscribe research systems.

based and *content-based* systems [19]. Whereas topic-based systems, such as the Information Bus [39] and Usenet news (see Section 2.2.1), inherit the ISIS model of a fixed topic identifier, the content-based approach allows subscribers to restrict the events they would like to receive to those satisfying certain rules. An example is the Linda [22] generative communication model, in which programs interoperate by creating and consuming tuples (data objects of heterogeneous type) in a shared store. Linda programs subscribe to events by waiting for the appearance of tuples matching a given pattern. This pattern can constrain tuples to those of a given type signature, or those containing particular data values, or a combination of rules of these types.

2.2 Early Internet pub-sub

2.2.1 Usenet

The Usenet system is perhaps the most widely-adopted pub-sub mechanism in use on the Internet. Usenet is a global "bulletin board" style system organized around *newsgroups*, which are identified by memorable strings in a global hierarchy (e.g. `comp.os.linux.misc`, `sci.math`). End users employ *newsreader* software to post individual *articles* to these newsgroups, as well as to read new articles.

Articles are transmitted across Usenet as follows: a properly-formatted [26] message is submitted by the newsreader to a local NNTP [30] server, which in turn forwards the data to other neighbor servers (its *newsfeeds*, in Usenet parlance). Each server keeps a list of newsfeeds for this purpose; the result is a connected graph of servers comprising the global Usenet.[2] Data is simply flooded along the server graph; servers reject, and cease to retransmit, articles they have already seen [55].

Subscription control in Usenet is entirely local: Each server in the network can choose which newsgroups to accept, store, and propagate to other servers. Each end user can further

[2]These connections originally included UUCP, FidoNet, and other non-Internet links, but by now almost all Usenet traffic is carried over the Internet.

select, from among those groups her local server carries, groups she is interested in reading in her newsreader.

Scalability is a major concern for operators of news servers; they must accept gigabytes of new data per day (terabytes if the "binaries" groups are included), as well as retaining a local store many times as large (to make recent articles available to users at will, and to prevent retransmitting old articles). In order for news to be successfully transmitted across the Usenet cloud, every server must carry (accept, store, and forward) most major groups, even if there are no local subscribing users. Recently, peer-to-peer storage techniques have been proposed to ameliorate some of these storage and bandwidth requirements [59].

Increasingly, users are turning to simple Web-based Usenet access (such as Google Groups[3]), rather than using local newsreader software, in part because of the difficulty of finding a reliable NNTP server willing to provide a newsfeed. More generally, end-user participation in Usenet is now in decline, in favor of modern Web-based information sources such as websites, web-based forums, and RSS feeds, as well as older (but more accessible) two-way discussion tools such as e-mail lists.

2.2.2 PointCast

PointCast was launched in 1996 as a way for users to receive news on their idle desktops. The system, called PointCast Network, claimed the ability to deliver news in a timely fashion to client hosts all over the Internet; its appearance prompted a surge in interest and hype around "push" applications [20]. Push technologies became part of the Windows operating system (in the form of Active Desktop) and the Netscape Communicator product (Netcaster) [12].

System administrators quickly noticed that PointCast's push system used a large amount of their network capacity [11]; to achieve prompt updates, PointCast clients were actually polling central servers rapidly and repeatedly to download new content [16]. Before long, "push" went from being a buzzword to an anti-buzzword, and companies working to develop push software

[3]http://groups.google.com/

began to distance themselves from the term (and the technology). The PointCast Network ceased operations in 2000 [29].

Despite its failure, the PointCast product spurred the development of what would become the foundation of modern Web feeds. Microsoft's XML-based *Channel Definition Format* [18], used in Active Desktop, bears a strong resemblance to RSS, and when Netscape designed a similar XML-based format for news channels on its My Netscape portal service, the result was in fact the first version of RSS.

2.3 Web feeds: RSS and Atom

In 1999, Netscape released a specification called RDF Site Summary (RSS) version 0.90 [36], which allowed any news-oriented website to publish headlines in a format that could be integrated into the My Netscape portal then in operation. Users of the portal could pick and choose their favorite news sources, and the headlines from each source's RSS *feed* would be aggregated on the portal page for convenient reading.

In many ways, today's Web feeds are quite similar. Any website can publish a feed (as compared with the proprietary channel agreements favored by PointCast and other push networks of that era); each feed has a unique URL where it can be fetched by a feed *reader* or *aggregator* and presented to the user. These readers take the form either of software applications running on the user's computer or Web-based services providing this function.

There exist a number of descendants of the original RSS specification, each offering varying levels of compatibility and feature parity with the other. The most common formats in use today are the RSS 1.0 [54] and 2.0 [69] specifications[4], as well as the new IETF-standardized Atom syndication format [38].

[4] It would be reasonable to suppose that RSS 2.0 is a newer version of 1.0, yet they are related only in purpose and in name; one is not descended from the other, nor has RSS 1.0 ceased evolution now that 2.0 is available. They are independent specifications, so much so that they even disagree on the proper expansion of the RSS acronym ("RDF Site Summary" vs. "Really Simple Syndication" for 1.0 and 2.0, respectively). The formats also have various disjoint features and other incompatibilities; any software wishing to parse feeds must include code to deal with each of them separately, as it appears unlikely that the feed community will settle upon a single standard in the near future.

Each of these Web feed formats share the following general structure: a feed contains a set of items, each of which contains a URL, a title, and possibly other meta-data such as HTML text and publication date. The feed itself may have additional meta-data describing the feed and the Web resource it is intended to summarize. New items may appear in feeds at any time, so feed readers must periodically request the feed from its canonical URL in order to learn of updates. It is this polling behavior that causes bandwidth problems for publishers.

2.4 Feed bandwidth

At the time of publication of [56], there was considerable concern among feed publishers about the "RSS bandwidth problem". They observed that feeds are requested more often and for a longer period of time than other Web resources, and so they require more network capacity to serve.

This situation is most likely the effect of many behaviors working in concert:

Polling. As described in Chapter 1, each feed reader must issue periodic HTTP requests in order to collect updates from each subscribed feed. Though some optimizations exist (GZip compression, or If-Modified-Since/ETag content hashes), in general an active feed will be downloaded in its entirety by each reader many times a day.

Superfluity. The data format of feeds is essentially static; all entries are returned every time the feed is polled. By convention, feeds are limited to some N most recent entries, but those N entries are emitted for every request, regardless of which of them may be "new" to a client. A recent measurement study [33] found that 95% of RSS transfer bandwidth was spent on superfluous, unchanged data. This particular aspect of the bandwidth problem could be helped by introducing a delta-based transfer mechanism (in which a client might request all entries updated after some previous poll time t_0), exchanging some bandwidth for some processing load at the server. No such scheme is currently standard or widely supported for Web feeds.

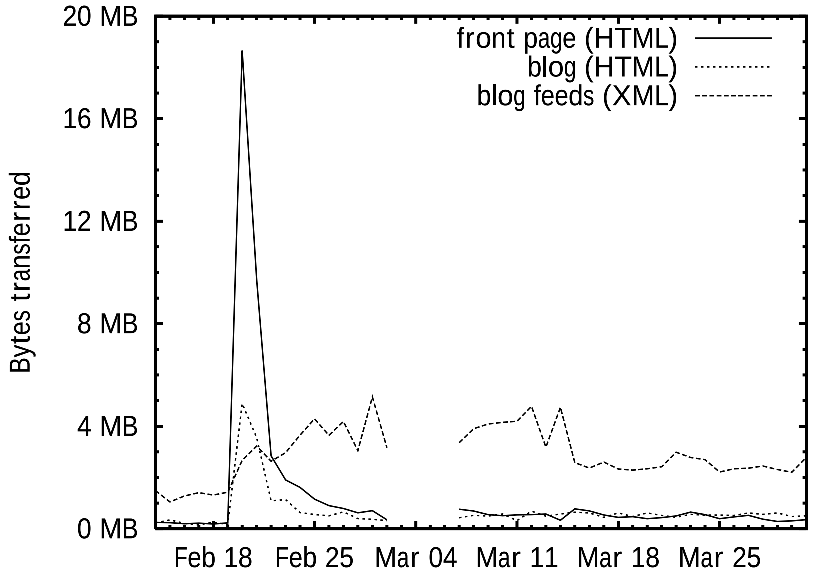


Figure 2.1: Timeline of feedtree.net HTTP bandwidth. The figure shows bytes transferred in response to HTTP requests for three resources: the front page (HTML only, no images), the first page of the FeedTree weblog (HTML only), and the main RSS and Atom feeds (combined, XML data only). HTTP server logs for March 3–6 are unavailable.

Stickiness. A single Web page may become briefly popular and therefore cause a traffic spike; a suddenly-popular RSS feed, however, will result in an ongoing bandwidth demand from the endless polling of new subscribers. [58]

Twenty-four-hour traffic. RSS retrieval processes (in the form of end-user applications) tend to run persistently on personal computers, in much the same way email readers tend to remain running at all times. The constant polling of these idle hosts generates a persistent and unnecessary traffic load on publishers. By contrast, typical Web requests can be expected to follow a diurnal pattern, with readers browsing during waking hours. While a site's users may be distributed around the globe, nominally creating round-the-clock demand on the server, it is certainly a fraction of the demand associated with constant requests from *every* potential reader at all hours.

To the extensive measurement of these characteristics in [33], we offer an additional, anecdotal data point: the request logs from the feedtree.net web server, before and after a mention [63] on the popular technology news site Slashdot[5]. Figure 2.4 shows the bandwidth required to serve requests from outside the Rice network. The spike on March 20, corresponding to the Slashdot article, is hard to miss; the front page (about 5 KB in size) received 3291 hits that day, corresponding to 18 MB of traffic on HTML alone. (The total Web bandwidth for that day was 1.7 GB.) Within a couple of days this traffic slackened nearly to pre-Slashdot levels; over the same time period, however, feed subscription (as indicated by request bandwidth) increased steadily. As of April 1, the bandwidth dedicated to feeds is now ten times that of the site's main HTML page, and is waning only gradually.

2.5 Weblog pings

Related to the feed bandwidth problem is a new issue referred to as the "ping crisis". Many weblogs send a message to *blog ping services* (such as weblogs.com, blo.gs, and pingomatic.com) to announce new content. These ping services in turn use the information to list recently-updated weblogs, send emails to interested parties, notify search engines of new content, etc.

The sustainability of such services, which require tremendous amounts of bandwidth and processing power but have little hope of profitability, has recently come into question [6, 43]. There is substantial overlap between the scalability issues of such services and those of Web feeds. A solution to the Web feed distribution problem which results in timely and efficient notification of updates can be used to replace the existing ping infrastructure as well.

2.6 Peer-to-peer overlay networks

Recent interest in peer-to-peer (p2p) systems can be traced to the surge in popularity of file-sharing applications beginning with Napster in 1999. These systems subverted the conventional client-server structure of most existing Internet services, leveraging instead the available resources of end users—network bandwidth, storage capacity, processing power—to provide a level of service difficult for a central server to provide for a reasonable cost. We

[5]http://slashdot.org/

use the epithet "peer-to-peer" here to refer to any distributed system in which resources are chiefly provided by users of the system. Participants in peer-to-peer systems become nodes in an *overlay network* superimposed on the underlying Internet network topology.

In peer-to-peer systems such as Napster and Gnutella, there exists no underlying structure or organization to the network. Any data object may reside unpredictably on any node, making the task of locating objects problematic. In Gnutella, an exhaustive search is performed by flooding queries through the overlay; the original Napster system maintains a central index of all object locations that nodes may query. [57] In BitTorrent [14], a dedicated service called a *tracker* maintains a list of nodes which hold portions of each file.

By contrast, *structured* overlay networks use recursive routing and constrained object placement to achieve sub-linear complexity. These decentralized, self-organizing systems—including Pastry [52], Bamboo [49], Chord [64], and Tapestry [70]—assign a unique identifier (in a large address space, typically 2^{160}) to each node in the network. Identifiers in this space are also assigned to data objects or rendezvous points. The overlay allows any participant to route messages to the node nearest to any identifier in a logarithmic (relative to network size) number of steps and with logarithmic local space requirements for the routing infrastructure. Applications commonly built on these overlay networks are distributed hash tables (DHTs) and multicast messaging systems.

Scribe [8] is one such application. It is a scalable group communication system built on top of the Pastry p2p overlay. Each Scribe group is assigned a *groupId* on $[0, 2^{160} - 1)$ which serves as the address of the group. The nodes subscribed to each group form a multicast tree, consisting of the union of Pastry routes from all group members to the node with nodeId numerically closest to the groupId. Membership management is decentralized and requires less than $\log n$ messages on average, where n is the number of nodes in the overlay.

Scribe has been shown to provide cooperative multicast service that is efficient and low-overhead [8]. The delay stretch is approximately double that of IP multicast and comparable to other end system multicast systems such as ESM [10] and Overcast [28]. Link stress is also low and less than twice that of IP multicast. When there are a large number of groups in the system, the load is naturally balanced among the participating nodes. Scribe uses a periodic heartbeat mechanism to detect broken edges in the tree; this mechanism is lightweight and is only invoked when there are no messages being published to a group. It has been shown to scale well to both large groups and to a large number of groups. These properties make it a good fit for building large scale event notification systems like FeedTree.

3

RELATED WORK

While there are currently no significant proposals to improve the timeliness of updates for end users, the Web feed community has developed a few best practices for ameliorating the “bandwidth problem.” In this chapter we will consider these existing solutions, as well as examine a few alternative proposals for addressing feed bandwidth. Finally, we will explore the landscape of other related peer-to-peer technologies and applications before examining the FeedTree system in the following chapter.

3.1 Current feed bandwidth remedies

The current “best practices” for publishers concerned about feed bandwidth fall into two categories: reduce the burden of each individual HTTP request on the server, or send those requests somewhere else entirely.

3.1.1 Optimizing HTTP requests

`gzip` [17] compression is part of the original HTTP specification [2] and is supported by most HTTP user-agents (including feed readers), so the bandwidth demands of feeds can be abated somewhat by compressing the feed data on the server side and decompressing at the client. Additionally, servers may elect to send `ETag` and `Last-Modified` cache control headers;

when clients return these values to the server during a request, the server may elect to respond with a small *304 Not Modified* message instead of the full (unchanged) content. This mechanism may also be applied to Web feeds [35], but it will fail to save bandwidth for feeds which change frequently or have dynamic content (e.g. advertisements). More generally, these bandwidth reduction techniques can never reduce the number of requests for a feed, but merely the size of the response.

3.1.2 Make it somebody else's problem

Concern over feed bandwidth has generated something of a third-party business opportunity. Companies like FeedBurner[1] host feeds on behalf of their customers, taking over all bandwidth demands of the feed. A publisher who signs up for such a service provides a private feed URL to the third party, which in turn polls that URL for updates. The third party then uses its larger network capacity to re-publish the feed contents at a new, public URL. Payment can be rendered in a number of ways; in the case of FeedBurner, customers must promote the service (in the form of a feedburner.com component to the feed's URL), or accept in-feed advertising (ads placed in the feed data by FeedBurner), or pay recurring service fees. These costs, when passed onto the publisher, present a barrier for small websites looking to reach a wide audience. Furthermore, these services represent a new outside dependency for their customers; should a feed hosting company change its terms or service or disappear entirely, its customers will scramble to restore service to their users (especially if every feed URLs refers to the third party host).

3.2 Alternative solutions

3.2.1 Bandwidth savings with Web caches

Distributed Web caches, such as Akamai[2], CoDeeN [68], CoBlitz [41], CobWeb [13], and the Coral content distribution network [21], solve the general problem of bandwidth-

[1] http://feedburner.com/
[2] http://www.akamai.com/

hungry Web resources by replicating them at many nodes around the world. Some of these systems (CoDeeN, CobWeb) require end users to adjust their HTTP proxy settings in order to leverage the cache. In others (Akamai, Coral, CoBlitz), operation is transparent for end users: the Web site operator must (purchase service, in the case of Akamai, and) replace URLs to heavily-requested or large objects with URLs to those objects as hosted by the caching service. Several of these services make clever use of DNS to route requests to geographically nearer caches [9, 21].

These services (in particular Akamai) enjoy great popularity on the current Internet, but they are principally intended to cache static content. They are able to achieve reasonable performance by replicating slowly-changing Web objects across participating nodes. However, the update frequency of Web feeds is often much higher and more irregular than the cache revalidation intervals of these services.

Furthermore, a peer-to-peer caching system relying on the contributed bandwidth, storage, and CPU of a small (relative to the user base) number of infrastructure nodes is fundamentally limited in capacity by the size of that infrastructure. In the case of the services deployed on PlanetLab, each of these services is competing for the *same* resources, which are finite.

To truly scale with the user base, these systems would need to leverage resources contributed by all, or some fraction, of their users. This "open membership" scenario requires either that all nodes be assumed trustworthy, or that the distributed system implement some sort of defense against malicious nodes wishing to alter or withhold data or inject spurious data into the system. Akamai does not allow end user nodes to store and forward cached data; the research systems (CoDeeN, CoBlitz, CobWeb, Coral) are similarly limited to their centrally-controlled infrastructure on PlanetLab and do not currently permit end user nodes to contribute resources. We explore the challenges associated with open-membership peer-to-peer systems further in Section 4.5.

3.2.2 Cooperative feed polling

A very recent application of distributed systems research to the Web feed domain is Corona [45], which uses a decentralized network of nodes to detect updates in arbitrary Web resources. Nodes in the Corona poll URLs cooperatively to spread the request burden among multiple clients and to avoid server-imposed polling frequency limits (a technique shared with FeedTree; see Section 4.2.3). Corona treats the problem of efficient update detection as an optimization problem, and so nodes coordinate their polling schedules (and the size of the pool of pollers) to achieve varying tradeoffs between update resolution and server load.

The Corona algorithms currently rely on correct, cooperating nodes in order to offer these benefits. A malicious polling node could inject invalid updates into Corona, or could fail to cooperate in the chosen polling strategy. Therefore, Corona does not currently support open membership, and so it too is limited in capacity by the size of the installed infrastructure (also PlanetLab in this case). FeedTree avoids these problems by providing publishers with a way to inject verifiably authentic events into the system, thereby offering both authenticity and rapid updates to end users. In the absence of cooperating publishers, FeedTree nodes make polling decisions based entirely on local observations; if a node is getting poor service from other (malicious or simply faulty) nodes in the network, it will begin to poll the resource itself in order to reliably offer updates to the end user. These mechanisms are described further in Section 4.6.

The end user experience of Corona and FeedTree is also markedly different. Users interact with Corona by sending subscription instructions to it via IM (instant messaging). The Corona IM "bot" will then begin forwarding Web diffs to the user's IM account. FeedTree differs in that it makes every effort to leverage the user experience of existing feed reading software, which is specifically designed to help users manage many streams of micronews without unduly taxing the user's attention. In addition, we feel that the restriction of FeedTree to Web feeds (rather than offering updates on arbitrary Web resources) is not a limitation; most sources of micronews on the Web are either already offered in feed form, or trivially convertible to feeds. Finally, the task of automatically isolating dynamic but irrelevant content (such as advertisements or randomized presentation)—and distinguishing it from legitimate news—is difficult to accomplish for all websites without a priori knowledge of the organization of the page. Put another way, Web pages are presentational documents, requiring heuristic extraction of news items, while Web feeds are semantic documents that describe news items explicitly and unambiguously.

3.3 Other applications of structured p2p overlays

The number of users at the edge of the Internet continues to outstrip growth in server capacity and provisioning, prompting great interest in using peer-to-peer principles to solve many problems of cooperation, distributed computation, and resource sharing. Structured overlay networks [52, 64, 70] (defined in Section 2.6) provide a substrate upon which these kinds of p2p applications can be built to scale to large numbers of participants and to survive various modes of failure.

The canonical application of structured overlays (that is, the application that inspired their development) is the distributed hash table (DHT), in which storage objects are spread evenly among cooperating peers and are accessible in $O(\log n)$ hops. Many variations on this theme have been developed and are the subject of active research, including PAST [53], CAN [46], CFS [15], OceanStore [31], OpenDHT [48], and Kademlia [34].

DHTs have also found use as an accelerant in a number of otherwise unstructured peer-to-peer file-sharing applications. The eMule file-sharing application [32], originally developed to work with the eDonkey2000 [25] ("ED2K") peer-to-peer network, now also employs a network called *Kad*, which is based on Kademlia. [65] The inclusion of the DHT allows eMule users to find one another and to find files without relying on the dedicated servers required by ED2K.

Similarly, recent implementations of the BitTorrent protocol [5, 4, 1, 66], employ a Kademlia-based DHT to track object location and availability in a decentralized fashion. While BitTorrent is a peer-to-peer file transfer protocol, in conventional use each torrent relies upon a stable "tracker" service to maintain a real-time map of the nodes in that torrent's swarm (that is, nodes holding chunks of data included in the torrent). The tracker is not global, as was the case with the original Napster, but it is a centralized service subject to failure. By distributing this information among participating nodes, so-called "trackerless" operation is possible.

Overlays and DHTs are also helping to distribute the costs (CPU, network bandwidth, etc.) of applications that are not naturally peer-to-peer in nature. As described in Section 3.2.1, Web caching is one such problem domain: it is fundamentally a one-to-many kind of communication, yet by cooperating peers may alleviate choke points such as server bandwidth. In some cases client bandwidth can be improved as well; Squirrel [27] allows the readers of a website under a common administrative domain (such as a single workplace) to pool their cached Web resources, reducing overall utilization of the group's shared Internet link.

Another popular one-to-many communication model is streaming media; this mode of content distribution is synchronous, so a DHT-based storage system will not suffice. Many peer-to-peer multicast systems have grown to fill this niche, among them Scribe ([8] and Section 2.6), Narada [10], Overcast [28], and CAN multicast [47]. These systems allow participants in such an application to participate in the forwarding load of a media stream (similar to Scribe's role in FeedTree, as detailed in Section 4).

More recent overlay-based multicast systems, such as SplitStream [7], are more sophisticated; in SplitStream, a total stream is divided into *stripes*, each of which is sent along a different tree to the same set of subscribers. Nodes that are interior in one tree are leaves in every other, limiting the damage they can do if they fail to operate correctly. Additionally, carefully-designed streams may degrade gracefully if a subset of stripes are received, allowing graceful "peeling" of the overall stream into useful (albeit lower-fidelity) subsets. Research continues into ways to overcome the fragility of multicast trees and support faster start-up and recovery operations [40, 67].

4

DESIGN AND IMPLEMENTATION

FeedTree is a peer-to-peer system that uses distributed group communication to distribute Web feed updates without polling. It is self-organizing, efficient, and scalable. FeedTree is fault-tolerant and resistant to attacks by malicious or self-interested users. This chapter details the design and implementation of the FeedTree system and describes its properties and benefits.

4.1 FeedTree participants

Feed publishers and subscribers each have a role to play in the FeedTree network. The functions performed by these parties are illustrated in Figure 4.1 and described in the following sections.

4.1.1 Publishing updates

In FeedTree, publishers notify their subscribers of new content using Scribe[1] multicast. The entire contents of the feed update are carried in each Scribe message; subscribers no longer need to poll the original source for updates.

Participants in the system, including end users and publishers, collectively share the burden of transmitting multicast messages to interested parties. The structure of the Scribe multicast

[1]Although we chose to base this design on Scribe, there is no reason it could not be deployed on any group communication system that provides similar semantics and performance characteristics.

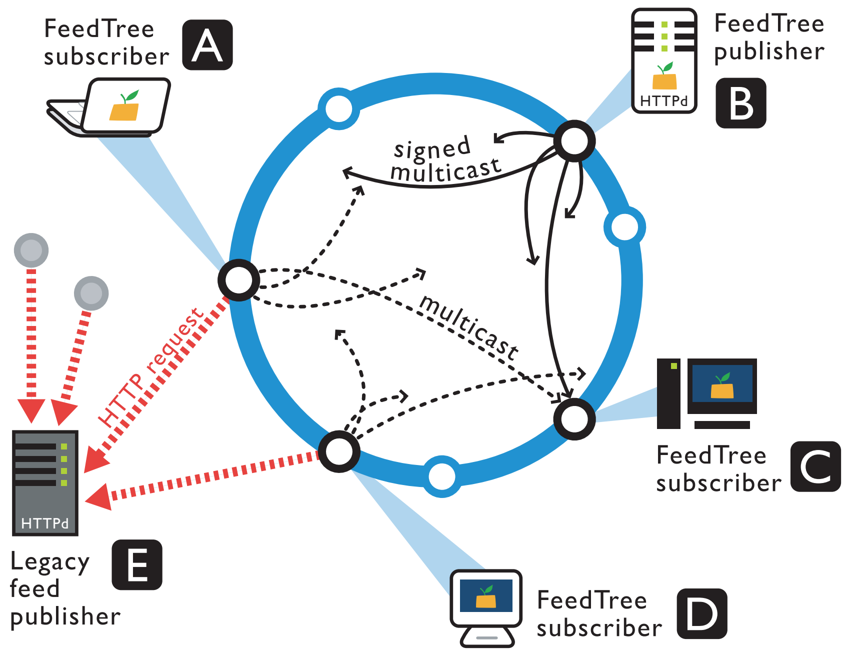


Figure 4.1: FeedTree system overview. Participating nodes form an overlay network in order to efficiently distribute update messages. Subscriber A receives feed updates via multicast. She also notices that one of her favorite feeds isn't being multicast, so she volunteers to poll that feed occasionally and share the news with everyone. Publisher B signs each new feed entry as it is created and multicasts it to all subscribers immediately. Subscriber C gets all the news he needs via multicast, and so never has to poll. Another subscriber, D, volunteers to poll A's feed as well, increasing the effective update frequency. The legacy publisher E is unaware of FeedTree, and publishes RSS and Atom feeds as usual, yet sees an aggregate improvement in his bandwidth costs.

tree ensures that any single node or network link, even those at the source of a new event, are not overly stressed; for more on overhead, see Section 4.4.

Each feed (distinguished by a unique, canonical URL, typically the same URL where it can be downloaded by conventional feed readers) is assigned its own Scribe group for the purposes of sharing updates. Publishers using FeedTree to deliver updates to subscribers take the following steps for each update:

- **A complete XML feed document is created** to contain the new or updated feed entry. Each item may further be annotated with a timestamp and a sequence number, to aid clients in the detection of omitted or delayed events.

- The feed data is then **signed with the publisher's private key.** This is essential to establishing the authenticity of each published item.

- The signed feed update is **multicast in the overlay** to peers in the feed's Scribe group. (Each group in Scribe is uniquely identified by a 160-bit *topicId*; for FeedTree, a feed's group has a topic equal to the SHA-1 hash of the feed's canonical URL.)

4.1.2 Receiving updates

End users of FeedTree join Scribe groups corresponding to the feeds they are interested in. Upon receipt of a FeedTree multicast message, a client will:

- **Verify the message's signature.** The recipient will use the public key of the publisher to authenticate the message. Distribution of key material is described in Section 4.6.

- **Extract feed entries** from the message. Each FeedTree message includes a complete valid feed, in which entries are clearly identified by feed format standards such as Atom and RSS.

- **Process the new entries** like any other feed update. This might involve sending the entry to a graphical feed reading application, and notifying the user.

In situations where no authoritative publisher is available for a feed, some small subset of the feed's subscribers will "volunteer" to poll the feed and distribute news to other subscribers. In this case, users (such as the nodes labeled [A] and [D] in Figure 4.1) may behave both as publishers and subscribers. Exactly how these volunteering nodes coordinate with one another is explained in detail in Section 4.2.3.

4.2 Implementation

The FeedTree software, freely available in source and binary form from feedtree.net, is comprised of two different packages: an end-user application which provides FeedTree benefits

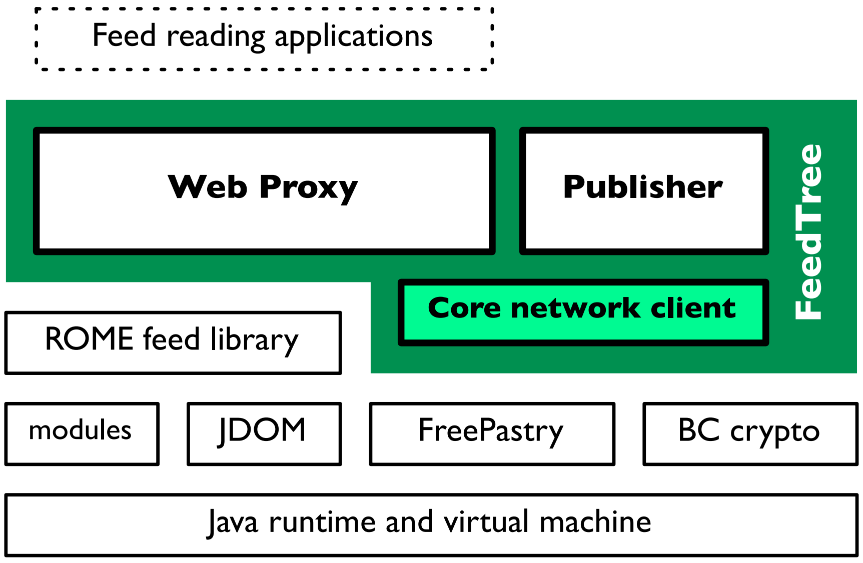


Figure 4.2: FeedTree software architecture. The FeedTree Web Proxy and Publisher applications are built on top of the general-purpose FeedTree Core. The ROME toolkit [50] (which in turn depends on plug-in modules and the JDOM XML library) offers a uniform feed parsing and generation API to the Proxy and Publisher. The BouncyCastle library provides implementations of cryptographic primitives to the Core. FreePastry provides the Pastry overlay network and Scribe multicast/anycast service to the Core.

to existing feed readers, and an application allowing feed owners to push updates to FeedTree subscribers. The applications are provided as compiled Java bytecodes and are usable on any platform with a Java 1.4 runtime environment (J2RE 1.4). Within each of these packages is a core FeedTree module which can be embedded into any Java application, as well as a number of support libraries. The end-user package also includes the FeedTreeProxy application, described in Section 4.2.2; the package for publishers includes the FeedTreePublisher, described in Section 4.2.4. The software architecture is summarized in Figure 4.2. We now explore these components in detail.

4.2.1 The FeedTree Core

The core client module provides the FeedTree system abstraction on top of the Scribe multicast/anycast system. The basic operations available to an application using the core are:

Connect to the FeedTree network. The application may specify a network port on which to run the Pastry node underlying FeedTree. In addition, a *bootstrap host* may be supplied: the IP address or DNS name of a host that is already a member of a FeedTree network. By connecting to the bootstrap, the application joins that network. (Alternatively, the client may omit the bootstrap information and start a new network.) In practice, users will connect to the public FeedTree bootstrap, `bootstrap.feedtree.net` (see Section 5.2.1).

Subscribe or unsubscribe a feed. The client will join the Scribe group whose topicId is equal to the SHA-1 hash of the given feed URL. When subscribing to a feed, the client will also use Scribe *anycast* to send a "catchup" message to the feed's group, soliciting other subscribers to send a few recent feed items back. This is a convenient way to recover a small amount of recent history on a newly-joined feed.

Publish a feed event. The event (which may be signed or unsigned, or may be a "no news" heartbeat message) will be sent to the appropriate feed's Scribe group.

Applications using the core client can register several callbacks (by overriding the appropriate Java virtual methods) to be notified when new FeedTree *events* are received. The recognized events are listed in Figure 4.2.1.

We envision the FeedTree core client library as an integrated part of feed reading and feed publishing software alike:

- By including the core client, **feed reading applications** become part of the global FeedTree system, distributing the network and processing loads of RSS event forwarding. The user interface for such a FeedTree-enabled application would remain unchanged; the user subscribes to feeds as she would do ordinarily, and the software takes care of subscribing to each feed in FeedTree. New items are made available to users as soon as the FeedTree events are received by the application.
- **Web publishing tools** (such as content management systems or weblog publishing packages) can incorporate the FeedTree core to automatically send FeedTree events for

MESSAGE TYPE	SCOPE	PURPOSE
Feed update	Multicast	A new entry is available for a given feed; the update contains the XML representation of the entry and the URL of the feed it pertains to.
Feed update (Signed)	Multicast	An authoritative version of the above. To be sent only by the authoritative publisher (the owner of the feed).
Heartbeat	Multicast	No news is available for a given feed. This is used for coordination: it indicates the presence of a source of FeedTree messages for this feed.
Heartbeat (Signed)	Multicast	An authoritative version of the above. Receipt of a signed heartbeat message indicates the presence of the authoritative publisher in FeedTree.
Catchup request	Anycast	A node is requesting recent messages for a feed. (Response is optional.)
Catchup response	Anycast	A node is responding to a catchup request.

Figure 4.3: FeedTree messages. Each message type corresponds to some kind of feed event, or, in the case of catchup messages, a joining node wishing to hear about recent feed events.

new content. Readers of feeds generated by such publishing software will receive updates promptly via Scribe.

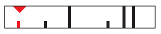
4.2.2 FeedTree Web Proxy

Because integrated support for FeedTree is not currently a part of existing feed reading applications, we must offer a way for end users to participate in FeedTree today, so they can relieve bandwidth stress on publishers (and receive rapid updates themselves).

Rather than develop a fully-featured feed reading application as a front-end for FeedTree, we chose to build an HTTP proxy which integrates FeedTree with any desktop feed reader application. The proxy is available as a Java archive (JAR) as well as Windows and Mac OS X packages which cause the software to resemble a stand-alone desktop application (though it still requires Java to be present).

The proxy application, once launched by the user, joins the FeedTree network and waits for a local feed reader to request a feed via HTTP. In response to such a request, the proxy instead substitutes the feed data from its local cache, which includes all updates received via multicast. Local applications can then be configured to poll (via the proxy) arbitrarily often, to ensure that new updates are seen by the user as quickly as possible.

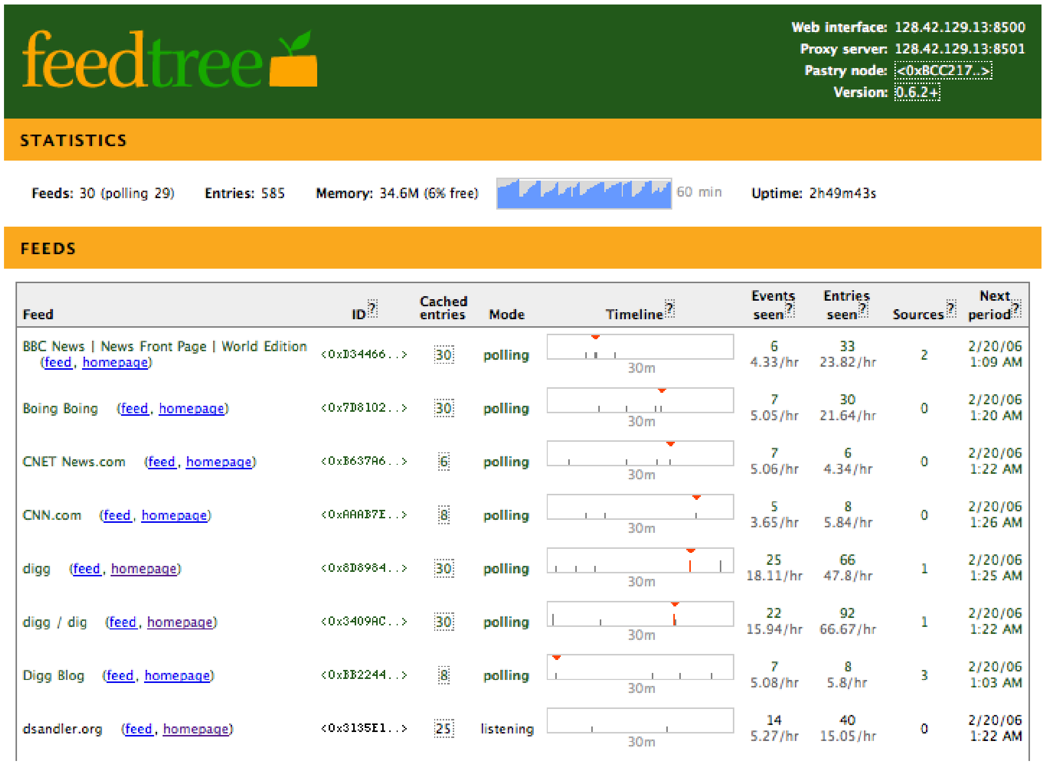
The proxy observes all HTTP requests and attempts to intercept the retrieval of feeds. Anything that looks like a feed is returned to the user, while the proxy subscribes to the FeedTree multicast group for that URL; subsequent proxied requests for that URL will return cached feed data from the proxy. This cached data is updated any time a FeedTree event (either a new entry in the feed, or a "nothing new" heartbeat message) is received by the proxy. Clients are therefore free to poll feeds as frequently as they want (for example, every 5 minutes) because these requests are intercepted by the proxy. The more aggressively the user's feed reader software polls through the proxy, the sooner the user will see news that has arrived via FeedTree multicast.

The end user can investigate the operation of the proxy application by way of its Web-based interface, shown in Figure 4.4. Recent events for each subscribed feed are portrayed in the form of a small timeline graphic. These timelines contain symbols for each feed event, with newer events toward the right: a poll invoked by the local node (triangle), a heartbeat (short line), an update (long line). Red symbols represent events originating at the local proxy node. For example, the timeline  indicates that, first, the user's proxy polled, discovering nothing new (thus generating a heartbeat); later it saw, in order, a heartbeat from some other node, an update (new data) from another node, another heartbeat, and then two new updates.

4.2.3 Cooperative polling in the proxy

If a FeedTree user subscribes to a feed that isn't being pushed by its original publisher, that user's proxy will fall back to a polling regimen for that feed. The proxy now enters "volunteer mode": any new entries it discovers are then multicast to that feed's Scribe group, satisfying any other FeedTree users who might later subscribe to the same feed.

By polling when necessary, the first FeedTree user to subscribe to a given feed—or even the first user of FeedTree itself—will continue to see feed updates just as without FeedTree. As



feedtree

Web interface: 128.42.129.13:8500
Proxy server: 128.42.129.13:8501
Pastry node: <0xBCC217..>
Version: 0.6.2+

STATISTICS

Feeds: 30 (polling 29) Entries: 585 Memory: 34.6M (6% free) 60 min Uptime: 2h49m43s

FEEDS

Feed	ID	Cached entries	Mode	Timeline	Events seen	Entries seen	Sources	Next period
BBC News | News Front Page | World Edition (feed, homepage)	<0xD34466..>	30	polling	30m	6 4.33/hr	33 23.82/hr	2	2/20/06 1:09 AM
Boing Boing (feed, homepage)	<0x7D8102..>	30	polling	30m	7 5.05/hr	30 21.64/hr	0	2/20/06 1:20 AM
CNET News.com (feed, homepage)	<0xB637A6..>	6	polling	30m	7 5.06/hr	6 4.34/hr	0	2/20/06 1:22 AM
CNN.com (feed, homepage)	<0xAAAB7E..>	8	polling	30m	5 3.65/hr	8 5.84/hr	0	2/20/06 1:26 AM
digg (feed, homepage)	<0x8D8984..>	30	polling	30m	25 18.11/hr	66 47.8/hr	1	2/20/06 1:25 AM
digg / dig (feed, homepage)	<0x3409AC..>	30	polling	30m	22 15.94/hr	92 66.67/hr	1	2/20/06 1:22 AM
Digg Blog (feed, homepage)	<0xBB2244..>	8	polling	30m	7 5.08/hr	8 5.8/hr	3	2/20/06 1:03 AM
dsandler.org (feed, homepage)	<0x3135E1..>	25	listening	30m	14 5.27/hr	40 15.05/hr	0	2/20/06 1:22 AM

Figure 4.4: FeedTree Web Proxy: web-based interface. End users can observe the state of all feeds currently being tracked by the local proxy application, including those that are being polled and those that are received entirely by multicast.

soon as a second FeedTree user joins a feed's Scribe group, the feed's publisher will begin to see bandwidth savings as two subscribers are satisfied by the requests of one.

So far, we have described a system that provides service equivalent to conventional polling for feeds whose publishers are not FeedTree participants. We can leverage this volunteering mechanism to improve this service for users of the FeedTreeProxy, bringing updates more rapidly than a typical polling schedule will allow.

The heuristic used by FeedTreeProxy to decide to enter volunteer mode is as follows: If the overall rate of FeedTree events received by the local node is lower than the *desired* update rate, the node begins to poll. It does so after a randomly-chosen initial delay, resulting (on average) in a staggered workload spread across all polling nodes.

In the current FeedTreeProxy implementation, this desired update rate is set to 10 minutes; combined with a 30 minute polling period, this results in a threefold improvement in timeliness of updates for all feeds in FeedTree. This additional polling overhead on the publisher is small (still constant with respect to the number of subscribers), yet provides a substantial benefit to subscribers. We note in Section 4.6 that this technique also offers some defense against faults and malicious nodes.

4.2.4 FeedTree Publisher

Legacy publishing software that currently emits valid RSS or Atom can be adapted to FeedTree with the FeedTreePublisher. This application is actually a *republisher:* it is a long-running process which polls a set of conventional Web feeds on an aggressive schedule (perhaps every minute or two), sifting out new content and distributing it via signed FeedTree events. By signing these events, a feed owner can ensure that content is not tampered with during distribution. The FeedTreePublisher need not run on the same host as the web server offering the conventional feed, which offers some much-needed flexibility to sites wishing to slowly roll out FeedTree support.

4.3 Benefits for participants

The FeedTree system offers substantial benefits for both producers and consumers of Web feed data. The chief incentive for content providers is the lower cost associated with publishing feeds: large Web sites with many readers may offer large volumes of timely content to FeedTree clients without fear of saturating their network links, and a smaller Web site need not fear sudden popularity when publishing a FeedTree feed. FeedTree also offers publishers an opportunity to provide differentiated feed services, perhaps by publishing simple (low-bandwidth) headlines in a conventional feed, while delivering full HTML stories in FeedTree.

End users will receive even *better* news service with FeedTree than is currently possible. While users currently punish Web sites with increasingly aggressive polling schedules in order to get fresh news, no such schedule will match the timeliness of FeedTree, in which users will see new items within seconds rather than minutes or hours. If publishers begin to offer richer micronews through FeedTree, we believe users will be even more likely to use the system.

Finally, since feed reading applications are generally long-running processes, building FeedTree into these applications will likely result in a stable overlay network with low churn.

4.4 Overhead

4.4.1 Maintenance

The Scribe multicast operation is highly efficient; Scribe trees have an expected height of $O(\log n)$, with the logarithmic base being 16 in the case of Pastry. A subscriber in a network of one million nodes should expect, for example, to receive published messages in 5 hops on average. Each forwarding node should expect to send messages to only 16 children; due to Pastry's proximity-based routing tables, each of these children is expected to be a good choice in terms of network proximity (each child should be one of the closest nodes to the parent among all the parent's descendants). Finally, because Scribe tree maintenance is performed in a distributed fashion, multicast trees are resilient to dynamic groups and node failures.

4.4.2 Bandwidth and incentives

The bandwidth demands made on any individual participant in each multicast tree are quite innocuous. For example, an RSS feed generating 4 KB/hour of updates (a very reasonable figure) will cause an interior tree node with sixteen children to forward less than 20 bytes per second of outbound traffic. Most nodes will have substantially fewer children, as the number of highly-loaded interior nodes is quite small, and nodes will have no children at all in most of their trees [8].

Due to the extremely low forwarding overhead, we believe that nodes will have little motivation to demand service without providing service to other participants. Should freeloading become a problem, for example in the presence of very high-bandwidth feeds, we may need to add constraints to the system to force users to contribute resources. Incentives-compatible mechanisms to ensure fair sharing of bandwidth [37] can be applied if most users subscribe to several feeds and therefore depend upon one another for service, which we expect to be the case for most users (and which we observe in Section 5 to be the case among FeedTree users). No such mechanisms are currently implemented in FeedTree, and we are unaware of any participants for whom the overhead of FeedTree is onerous.

4.5 The importance of open membership

In order for the FeedTree system to scale to support large numbers of end users, it is essential that every end user participates in the Pastry network and assists in carrying the forwarding load. If, on the other hand, FeedTree were to rely on a subset of "supernodes" (as is common in p2p filesharing networks) or a fixed infrastructure (e.g. PlanetLab [42]), the capacity of the network would be limited by that infrastructure. In a system with I infrastructure nodes (supernodes) and N end users not contributing to the network, each node $i \in I$ can expect to provide service to on average $\frac{N}{I}$ end nodes,[2] rather than a constant outdegree (16 in the case of FeedTree). In the case where $N \gg I$ (which must be expected for a growing end-user readership of Web feeds and finite infrastructure resources), the logarithmic scaling properties of the network devolve to linear.

4.6 Security

4.6.1 Signed feed updates

For all its scalability benefits, open membership also presents a substantial security challenge: untrusted nodes in the interior of multicast trees may alter messages they are asked to forward. To protect the authenticity of published updates in FeedTree, it is therefore desirable for all publishers to cryptographically sign their data.

But how to distribute public key material? To avoid the issues (availability, fault-tolerance, trust, etc.) commonly associated with PKI or certificate authority solutions, we rely on the presumed authenticity of each conventional (i.e. HTTP-fetched) Web feed. In particular, we

[2]That is, assuming that node i is a leaf node on the multicast trees within the infrastructure. An interior node, however, will also need to shoulder the overhead of multicast within the infrastructure network.

assume that the client trusts any feed data it downloads via HTTP from the feed's canonical URL.[3]

We therefore use the conventional feed to publish both a URL and a fingerprint for the publisher's signing certificate. A FeedTree client, when first subscribing to a feed, will retrieve the conventional feed and examine it for this certificate information; if found, the client will in turn retrieve (and cache) the credentials necessary to validate the integrity of any future updates received for that feed via Scribe multicast.

In situations where publishers do not participate in FeedTree, signed updates are not available, and nodes must therefore trust their peers to poll resources and re-distribute any updates. FeedTree still offers limited resistance to malice in this situation, as discussed in the following sections.

4.6.2 Recovery of lost data

Even though Scribe multicast trees will heal themselves in the event of node departures, Scribe is still a best-effort service. Departures and failures within the multicast tree may result in missed events for some subscribers. FeedTree's periodic heartbeat messages help clients detect such failures; if a client ceases to hear heartbeats for a given feed, there may be a problem—a partition in the network, a general network failure, or a malicious node suppressing events—or the existing publisher(s) for that feed may simply have ceased sending updates.

The current policy of the FeedTreeProxy is to mask this problem: as described in Section 4.2.3, if the number of heartbeats from distinct sources drops below a threshold, the client will take action by starting to poll the feed on its own. The proxy's goal, at all times, is to ensure that the end user never misses feed data, so if the FeedTree network fails for any reason to deliver that data, it falls back to conventional means (i.e., polling) to satisfy the user's needs.

4.6.3 Additional possible defenses

[3] This seems a safe assumption, given that feed readers trust feed data they *currently* download via HTTP. Users concerned about man-in-the-middle attacks (e.g., DNS hijacking) may elect only to trust public keys received via `https:` URLs using valid SSL certificates (that is, those matching the URL domain and containing a valid chain of trust from a trusted certificate authority).

Stronger security measures may be added to FeedTree, though they are not currently implemented in the FeedTreeProxy. By adding sequence numbers to all (signed) Scribe messages, we can offer clients the ability to detect a careful denial of service attack in which individual events are dropped or suppressed but heartbeats continue to arrive. In this situation, a client might query its parent to recover the missing items (a variation on the "catchup" messages described in Section 4.2.1). Combined with the digital signatures already present in FeedTree, sequence numbers permit clients to detect any tampering with the flow of data from the authentic publisher.

Malicious interior nodes may still cause trouble for participants. They may suppress messages bound for their descendants, and even though those nodes can detect this (given the additional defenses described above), they may have difficulty recovering the missed messages. A node looking to fill a gap in its sequence numbers could successively query each ancestor in the tree, hoping to find a node "above" the attacker which can satisfy the request. Upon finding such an ancestor, the querying node could then elect to re-join the Scribe tree at this point, thereby exiting the "shadow" of the attacker in the tree.

Another approach is to distribute the responsibility of the Scribe root among several nodes. Multiple redundant Scribe trees, with different roots, may be employed to dramatically reduce the likelihood that a single malicious node is able to suppress messages to a given listener in all trees. In a fashion similar to SplitStream [7], these trees can be established with different rendezvous points predictably derived from the feed's URL, resulting in "crisscrossing" trees in which any single forwarding node in one tree is unlikely to have any of his descendants (in that tree) as descendants in all other trees.

Finally, we consider what utility can be derived from FeedTree by a node that believes absolutely nothing it hears via Scribe. For such a node, no peers can be trusted to poll external resources or forward messages correctly; furthermore, let us assume that no publishers can be counted upon to multicast regular signed messages (the absence of which could be detected). It turns out that there is yet some hope for the truly paranoid in this case. Such a participant can treat FeedTree updates purely as *hints,* prompting it to request an updated copy of the possibly-updated feed via HTTP; updates retrieved this way are considered to be authoritative. Obviously our paranoid node can choose not to *take* those hints if they arrive too quickly, thus placing an upper bound on his polling frequency and preventing his load on the server from exceeding that of a reasonable conventional client. His polling frequency is also bounded from below by the same heuristic described in Section 4.2.3. In any case his bandwidth burdern is no greater than that of a conventional RSS reader, and in the general case (in which hints received via Scribe are accurate and arrive less frequently than the conventional polling interval) his burden is lower.

5 EVALUATION

The FeedTree system is designed to reduce a the burden that feed service places on publishers even as it distributes feed updates faster than polled RSS. Its design, based on Pastry and Scribe, allows it to offer these benefits without unreasonable stress (network, CPU) on the nodes in the system. To validate this claim, this chapter evaluates the performance of FeedTree in synthetic experimental conditions and in a real-world deployment in the public Internet.

5.1 Synthetic experiments

We created a 16-node private FeedTree network to measure CPU and bandwidth demands under varying feed forwarding load (as simulated by adjusting the rate at which updates were sent through the network). We experimented with update periods of 1, 2, 4, 8, 15, 30, 60, 120, 240, and 480 seconds; each period was tested three times. To generate the desired update interval, we launched a FeedTree publisher node configured to refresh an always-changing Atom feed at the desired resolution. Thus, every time the publisher polled the feed, new data would be found and immediately multicast to the network. In each experiment, every node was subscribed to this constantly-updating feed. We describe the CPU and bandwidth results in the following sections.

5.1.1 CPU

In Figure 5.1, each experiment is shown from the perspective of a single node. (For clarity, experiments corresponding to 4, 8, and 30 second update periods are omitted from this graph.) The experiment progresses through three phases, clearly visible in the graph: startup (in which the network is grown to the desired size), quiescence (in which we observe the bandwidth demands of the steady-state network), and teardown. Early on in the bootstrapping phase, most nodes are involved in each join event, but as the network grows, bootstrapping becomes less traumatic. (The teardown phase is brutal: nodes are killed abruptly until only the one under test remains.)

We are most interested in the ten-minute steady-state period, which is shown in detail in Figure 5.2. CPU consumption clearly increases as a function of the update rate. To uncover the exact relationship, we plot the average CPU burden against update rate in Figure 5.3; from this graph we can see that the burden of participating in FeedTree varies roughly with the square root of the update frequency.

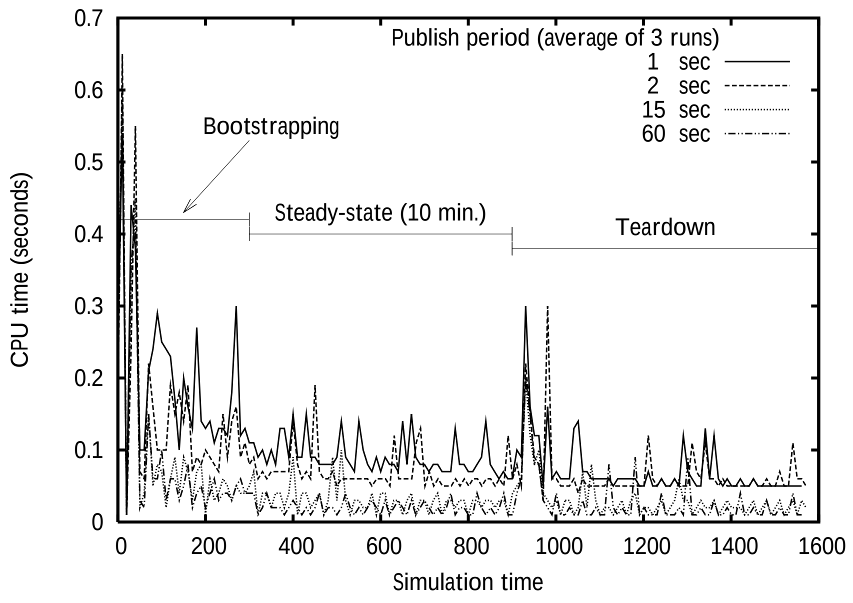


Figure 5.1: Experiment: CPU usage under synthetic load. Each 25-minute experiment, from the perspective of the first participating node, is graphed in its entirety. Clearly visible are the startup and teardown phases, as well as the ten-minute quiet period shown in Figure 5.2.

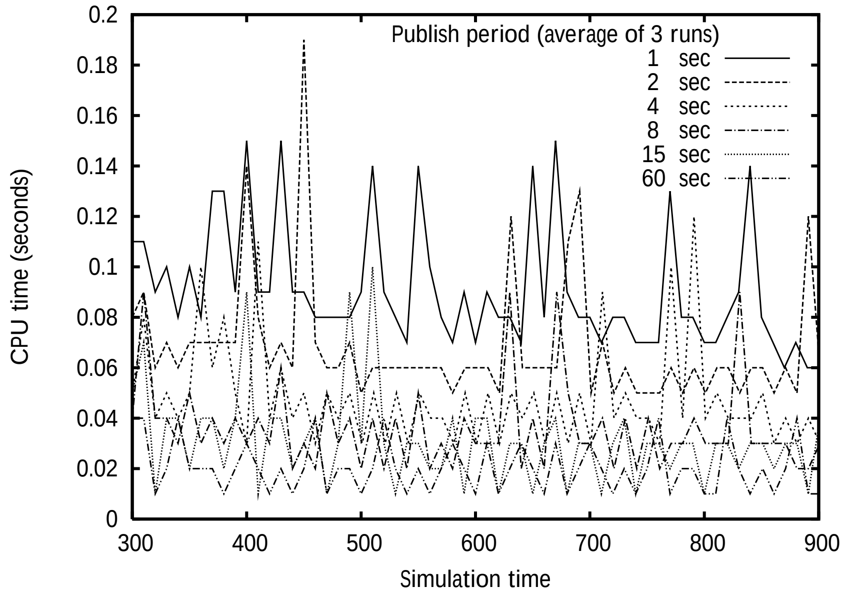


Figure 5.2: Experiment: CPU usage (detail). The "quiet period" is shown. Update rates omitted from Figure 5.1 for clarity can now be shown. The node whose CPU is being measured is a lightly-loaded dual-processor Apple Macintosh G5; it is subscribed to all feeds in the load, so it is involved in every single update.

5.1.2 Network

Our bandwidth measurements during these same 16-node experiments are shown in Figure 5.4. Network utilization was measured by capturing packets on all FeedTree (Pastry) ports using `tcpdump` on the node under test. Bandwidth seems to vary more greatly between runs than did CPU utilization, which is a result of the many random factors influencing the particular overlay network created for each run. (In particular, since *nodeId*s are chosen randomly, the shape of the Pastry ring is radically different, resulting in a different set of forwarding and rendezvous nodes for each Scribe tree in each run.)

As was done for CPU, we summarize our bandwidth data by plotting in Figure 5.5 the average network utilization against the period between updates. Traffic varies inversely with update period (and linearly with update rate), which is what we would expect for a conventional feed reader. This tells us that FeedTree is not applying any undue burden (beyond a constant hum of maintenance traffic) on each node.

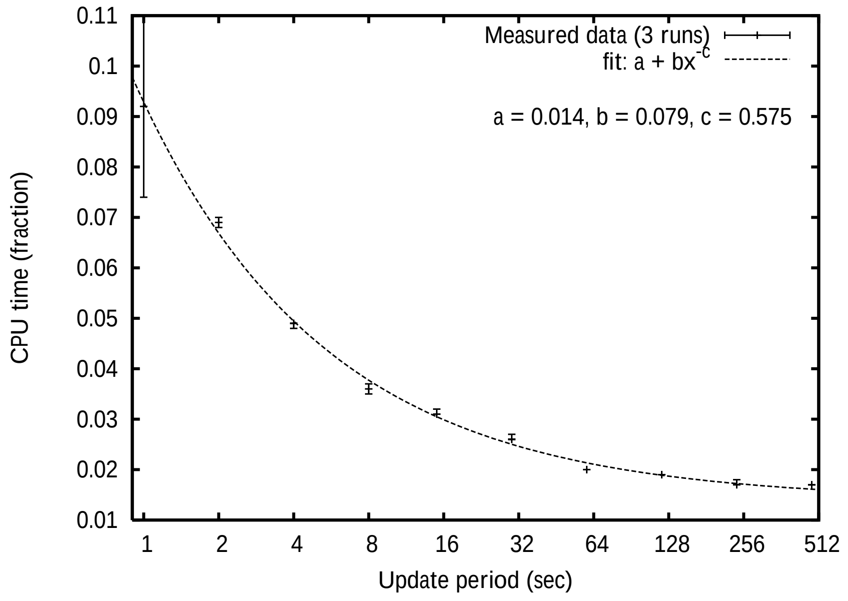


Figure 5.3: CPU usage vs. update rate. Summarizing the findings of Figure 5.2, the average CPU load for each update rate is shown. Error bars give the spread across three experiments. The burden of receiving (and possibly further disseminating) events varies roughly with the square root of the update frequency f (more exactly, with $f^{0.575}$).

We also created a larger, 64-node network, introducing nodes one at a time, so we could observe how a growing network impacts an individual node's bandwidth. These results are shown in Figure 5.6. Of note is the fact that the traffic does not increase appreciably with network size; this is because nodes joining a Pastry ring cause only local disruption, effectively keeping the impact of a single join from reverberating across the entire network.

As with CPU overhead, we generally found that FeedTree imposed a very small network overhead during normal operation, and a larger burden during major network maintenance events (such as massive joins and departures).

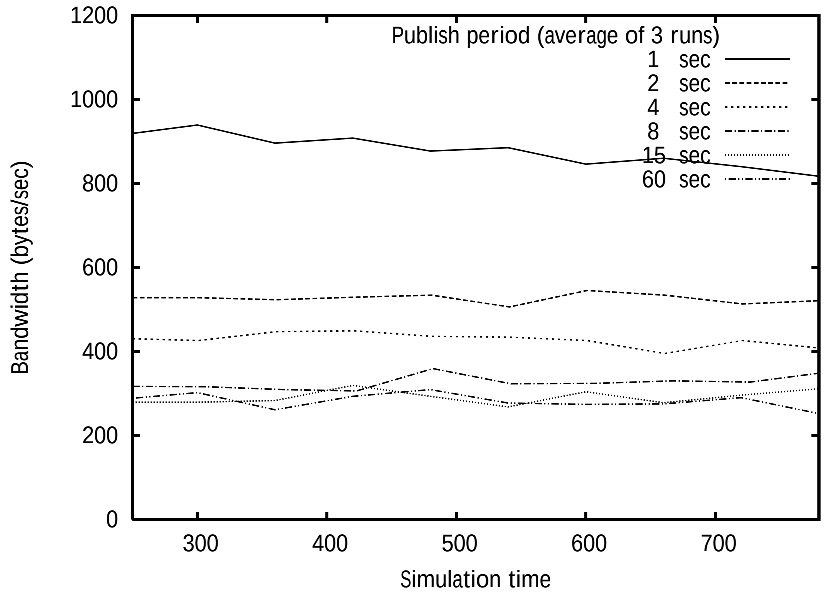


Figure 5.4: Experiment: Bandwidth of node under synthetic load. In the experiment described in Figure 5.1, we also measured FeedTree network traffic, which in this graph (showing a subset of our experiments) is seen to vary with update frequency. This relationship is plotted for all our experiments in Figure 5.5.

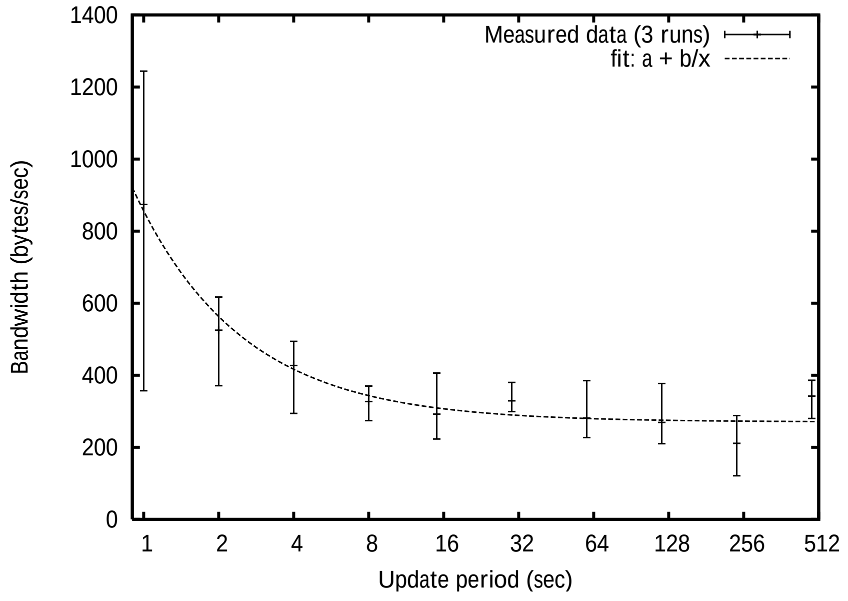


Figure 5.5: Experiment: Bandwidth vs. update rate. As for CPU in Figure 5.3, we plot the average network utilization against the period between updates. Traffic can be seen to vary linearly with feed update frequency.

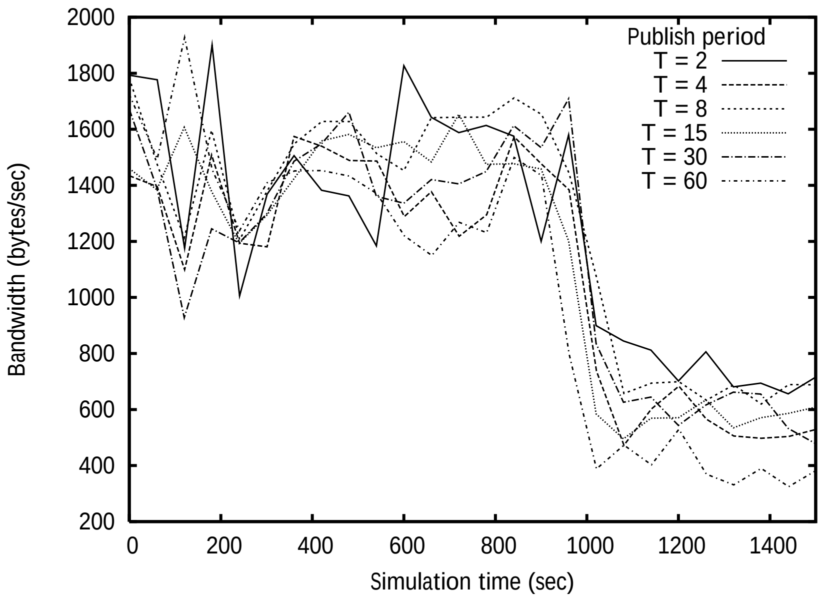


Figure 5.6: Experiment: Network growth. 64 nodes were started one-by-one, spaced about 15 seconds apart; the first node's FeedTree traffic over this period is plotted over time. The bootstrap phase (the first 1000 seconds of simulation) is clearly seen here as a period of increased network activity, but that activity does not increase with network size, reflecting Pastry's scalability.

5.2 feedtree.net: The global FeedTree network

We turn now to the public deployment of FeedTree. The FeedTree software was released in August 2005; the network has operated roughly continuously since then and is still operational as of early 2007.

5.2.1 Centralized components

In addition to the feedtree.net website, where users can download the FeedTree software and get information about the project, we provide `bootstrap.feedtree.net`, a round-robin DNS address (A) record which points to four official bootstrap nodes.[1] A FeedTree node can use any other node as a bootstrap, but the hosts referenced by `bootstrap.feedtree.net` are guaranteed to be available as rendezvous points for joining nodes. The public FeedTree network is thus defined as the Pastry ring which includes these four nodes. Finally, there is a database server where nodes can periodically report statistics about their operation, for later analysis.

5.2.2 Distributed components

Beyond these few services, the FeedTree network has no persistent infrastructure. The operation of the network depends entirely on the contributed capacity of the Pastry nodes operated by each individual user, in the form of a FeedTree proxy or publisher application (as described in Section 4.2). Indeed, even if the FeedTree website and bootstrap services were to disappear, clients could easily create a new and independent public network in which to share feed updates.

While many current peer-to-peer systems rely on PlanetLab [42] to host central infrastructure (if not the entire distributed system), FeedTree does not currently employ PlanetLab nodes. If we are to consider FeedTree an application that thrives in the "wild" of the Internet, we must not rely on the PlanetLab "zoo," whose nodes—monitored by researchers, and, at the time of this work, well-provisioned—are not representative of general Internet hosts [24].

[1]Our previous experiences with peer-to-peer systems programming has contraindicated using hardcoded IP addresses in client software.

We have instead focused our efforts on growing a network of real Internet users. Figure 5.7 shows the size of the FeedTree network (number of nodes, omitting hosts at Rice) as a function of time.

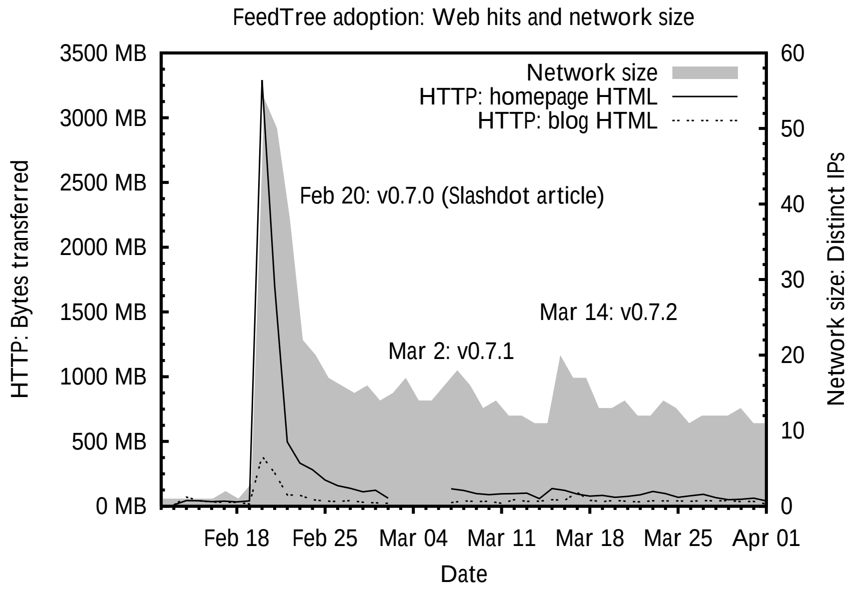


Figure 5.7: FeedTree adoption. The size of the FeedTree network, as estimated by the number of distinct (non-Rice) IPs reporting statistics on a given day, is plotted against the same Web server statistics shown in Figure 2.4. New users are clearly attracted by publicity and, to a lesser extent, new releases. Many new users join the network at first (peaking at 54 on 20 Feb), but the overall network size settles quickly (in the course of just a few days) to a steady community of 12 or 13 nodes.

5.3 Data collection

Users running FeedTree client software—that is, the Proxy or Publisher—report data periodically to a central database for research. In particular, the following data is submitted by all clients via HTTP `POST`:

- The Pastry *nodeId* of the client.
- The type of application; this is currently either `proxy` or `publisher`.
- The version number of the software in use (in typical dot-delimited format, e.g. `0.7.2`).

- ▶ The software's build number (defined to be the latest source-control change number as of the build date) of the software in use.
- ▶ The current Java virtual machine heap size; that is, the result of a call to `Runtime.getRuntime().totalMemory()`.
- ▶ Information about each Scribe tree in which this node participates:
  - ▷ The Scribe *topicId*.
  - ▷ The *nodeId* of the node's immediate parent in the tree (or an empty string if the node is the root of the tree and therefore has no parent).
- ▶ Publisher nodes submit the following additional information for each feed they are responsible for publishing:
  - ▷ The feed's canonical URL.
  - ▷ The feed's *topicId* (equivalent to the SHA-1 hash of the feed's URL).

These reports, sent every hour, allow us to reconstruct the approximate state of the network at that time. Because the reports are not synchronized across all nodes, this is hardly a consistent snapshot, but it does give us a good idea of the size of the overall network and the shape of the individual Scribe trees present. (Note that because its membership algorithm is entirely distributed, it is not possible to directly query a Scribe tree for its size or structure.)

An example of a Scribe tree constructed from these reports is shown in Figure 5.8. The graph (showing 11 subscribers and 3 forwarding-only nodes participating in a multicast tree for a single feed) was reconstructed from reports sent by participating nodes on February 21, 2006, between 18:30 and 21:30 CST; it therefore represents not a consistent snapshot at any single point in time, but a composite of local observations over that period. This data is drawn from a global FeedTree network of 40 nodes (10 at Rice, 30 outside), as measured by the number of distinct IP addresses in the network during the same period.[2]

[2] While this represents one of FeedTree's most popular days (in terms of network size), it is still not a particularly large system. In Chapter 6, we consider some reasons why FeedTree might not have taken off the way we had hoped, and discuss what we learned from our experience.

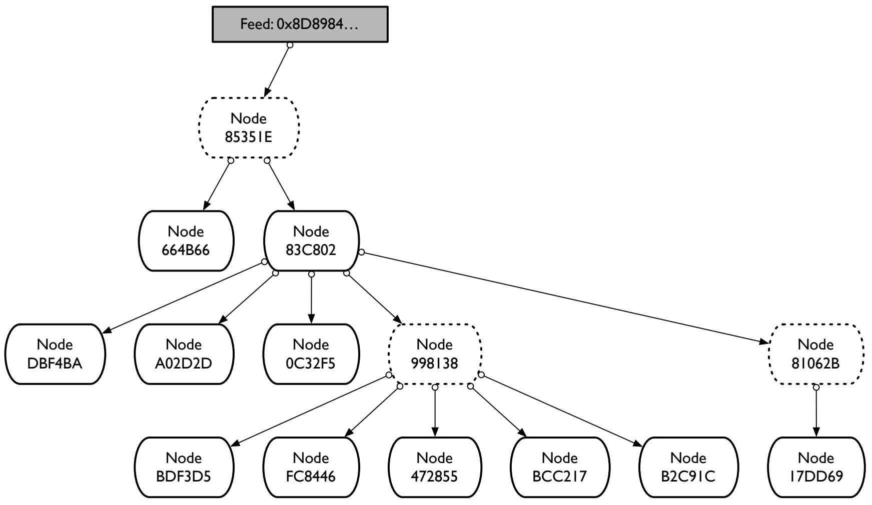


Figure 5.8: A FeedTree Scribe tree. The tree shown here, with *topicId* prefix `0x8D8984`, represents the Scribe event distribution tree for the Slashdot RSS feed, http://rss.slashdot.org/Slashdot/slashdot. Nodes with dashed borders are forwarders only; nodes with solid borders are subscribers (some of which also forward traffic). The node `0x85351E`, whose *nodeId* is closest to the *topicId*, is the tree root; new feed data is routed first to the root and then propagated to all descendants.

5.4 Analysis

5.4.1 Subscription popularity

Figure 5.9 ranks feeds by popularity across all FeedTree users over a 4-day period. The data shows a Zipf distribution, which is consistent with observations made by Liu and Sirer [33]. The outlier (plotted with a box in Figure 5.9) is the most popular feed among FeedTree users: the Slashdot.org RSS feed. It is more popular (with 22 subscribers) than the exponential curve fitting the rest of the data would indicate (13 or so). We suggest that this might be easily explained by the fact that most of our users discovered FeedTree through a Slashdot article [63] about the project, and so Slashdot is this likely to be represented disproportionately in our users' subscriptions.

5.4.2 Network traffic

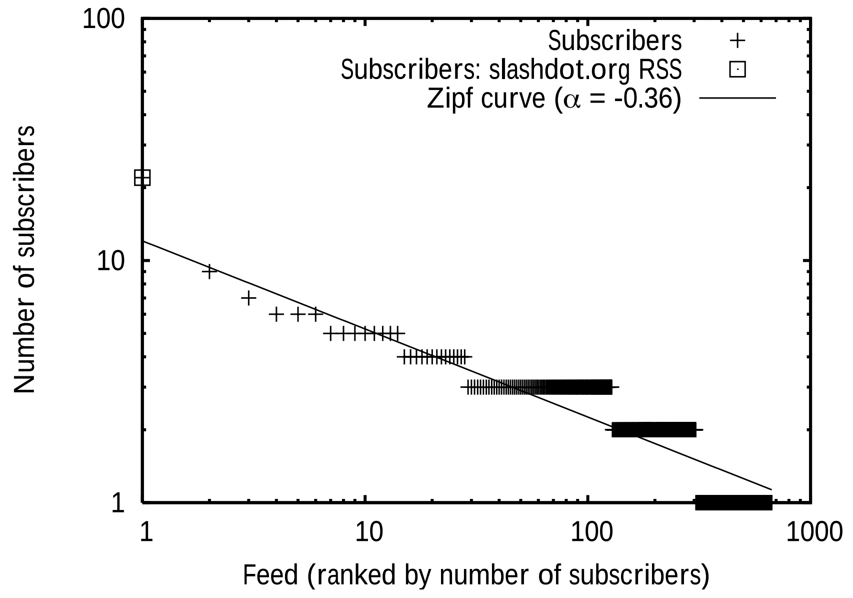


Figure 5.9: Subscription popularity. The data, collected from nodes in the live FeedTree network over a 4-day period between 22-Feb and 26-Feb 2006, show a clear Zipf distribution (fitting the curve kx^{α} with $k = 12.05$ and $\alpha = -0.36$). The most popular feed has 22 subscribers; the least popular feeds have just one. 670 feeds are represented in this plot.

We consider now the network traffic incurred by nodes in the FeedTree network. Figure 5.10 shows raw packet data at several nodes located at Rice. We observe that traffic is higher during the initial spike in users, corresponding to the additional maintenance traffic involved in supporting such a dynamic network. Traffic settles down to average about 40 MB/day, or about 3 Kbps.[3]

To place this bandwidth in perspective, consider that each node in the network contributes a roughly comparable amount of bandwidth to the overall problem of maintaining the Pastry ring and distributing feed updates. (This is an essential benefit of a totally symmetric distributed system.) Figure 5.11 shows the individual bandwidth usage of two FeedTree nodes with predetermined workloads (the URLs for which are shown in Table 5.1). Drawn for comparison is the approximate daily bandwidth (as computed in Table 5.1) required to poll each of these feeds using conventional HTTP requests.

[3]The network protocol in the version of FreePastry used by FeedTree relies upon Java object serialization to format messages between nodes. This encoding is quite verbose, and so the bandwidth required by FeedTree should decrease with a more optimized network encoding, of the sort that can be found in more recent FreePastry releases.

URL	**Bytes**
http://digg.com/rss/index.xml	33962
http://digg.com/rss/indexdig.xml	31364
http://diggtheblog.blogspot.com/atom.xml	16094
http://feeds.chron.com/houstonchronicle/topheadlines	13871
http://feeds.feedburner.com/boingboing/iBag	71287
http://feeds.feedburner.com/thericethresher/front/	11277
http://feeds.feedburner.com/thericethresher/news/	8785
http://feeds.feedburner.com/thericethresher/opinion/	7225
http://feeds.feedburner.com/thericethresher/sports/	5322
http://feeds.gawker.com/gizmodo/excerpts.xml	23801
http://feeds.gawker.com/gizmodo/full	81883
http://google.blogspace.com/rss	9045
http://jwz.livejournal.com/data/rss	50999
http://news.com.com/2547-1_3-0-5.xml	5007
http://newsrss.bbc.co.uk/rss/newsonline_world_edition/front_page/rss.xml	16517
http://reddit.com/.rss	11636
http://reddit.com/new.rss	11410
http://rss.cnn.com/rss/cnn_topstories.rss	3944
http://rss.news.yahoo.com/rss/topstories	43047
http://rss.slashdot.org/Slashdot/slashdot	16338
http://scobleizer.wordpress.com/feed/	138961
http://www.engadget.com/rss.xml	181934
http://www.joelonsoftware.com/rss.xml	18457
http://www.nytimes.com/services/xml/rss/nyt/HomePage.xml	4261
http://www.salon.com/rss/salon.rss	18810
http://www.scripting.com/rss.xml	2267
http://www.slate.com/rss/	52667
http://www.techdirt.com/techdirt_rss.xml	37903
http://www.theregister.co.uk/tonys/slashdot.rdf	5018
http://www.wired.com/news/feeds/rss2/0,2610,,00.xml	16070
Total	949162
... $\times$ *2 polls per day* $\times$ *24 hours*	45559776

Table 5.1: Sample feed workload. Nodes "3" and "4" in Figure 5.11 are FeedTreeProxy instances through which HTTP requests are made for these URLs. Thus, these nodes simulate proxies run by users who subscribe to these feeds in their feed reading applications. The feeds chosen include the most popular on the Internet (as of early 2006) as well as several feeds of more local interest. The total download bandwidth required to fetch all these feeds conventionally is computed to be approximately 43 MB/day; this bandwidth is compared against the actual measured bandwidth of proxies "3" and "4" in Figure 5.11. (Feed sizes were sampled on 3 Apr 2006.)

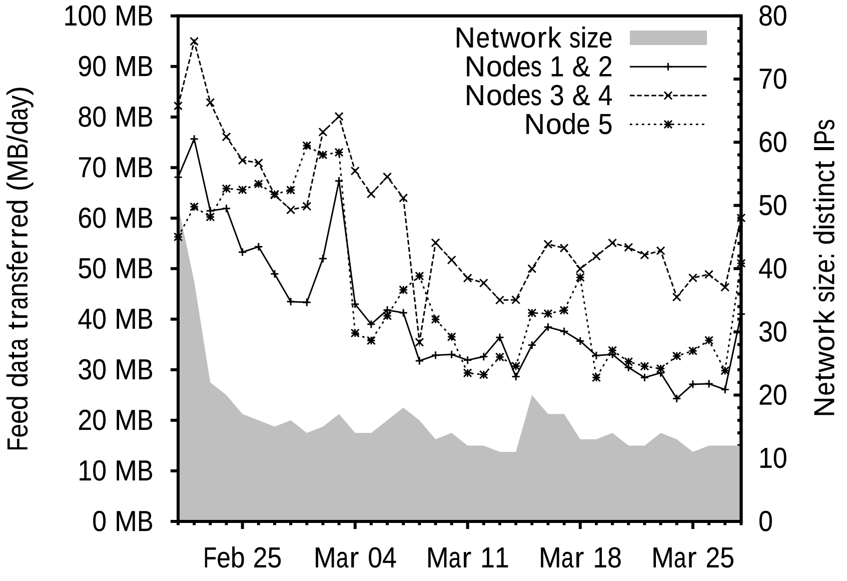


Figure 5.10: Network traffic in the public FeedTree deployment. The daily Pastry message traffic of several FeedTree nodes (located at Rice) is shown over a two-month period. Traffic is measured by a long-running `tcpdump` process on each host under observation, capturing all inbound and outbound TCP and UDP packets on FeedTree ports. The line labelled **Nodes 1 & 2** is the average bandwidth of two nodes running the FeedTreeProxy but subscribed to no feeds; these nodes merely forward traffic. **Nodes 3 & 4** (also averaged) are FeedTreeProxies responsible for polling 30 popular feeds; they both publish to and receive published events from the Scribe groups for these feeds. **Node 5** runs the FeedTreePublisher; it is subscribed to no feeds, but publishes 8 feeds (including heartbeats every 10 minutes) and forwards traffic. The overall network size, as approximated by the number of distinct IP addresses observed each day, is plotted in the solid background region for comparison.

To put these observed bandwidth figures in perspective, we consider the same bandwidth (symmetric across nodes) against the bandwidth required to host a given feed. We choose Slashdot's feed because its bandwidth demands are quite large, and yet the traffic observed at each current FeedTree node is sufficient to propagate all Slashdot update events and many, many others (cf. the 670 feeds represented by Figure 5.9).

In Figure 5.12, the RSS bandwidth demands of Slashdot's users is compared with the bandwidth incurred by one of our FeedTreePublisher nodes (the same "Node 5" from Figure 5.10). The difference is remarkable; four orders of magnitude separate Slashdot's feed needs from the modest bandwidth requirements of participating in FeedTree.

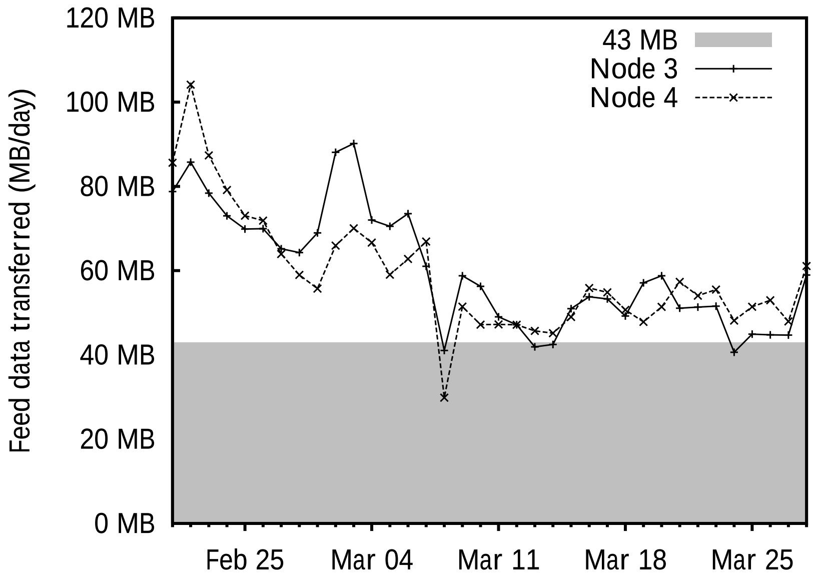


Figure 5.11: Bandwidth of polling nodes. The bandwidth of nodes "3" and "4" from figure 5.10 is broken out here. These nodes are each responsible for polling 30 predetermined feeds, including several very popular feeds (*e.g.* Slashdot, CNN) and a few less popular ones (*e.g.* Rice University's student newspaper). Drawn for comparison is 53 MB/day, the daily bandwidth required to request each of these feeds every half-hour conventionally via HTTP (as computed based on the average size of the feeds; see Table 5.1). [The dip in bandwidth on March 8 corresponds to downtime on the affected nodes.]

The small size of the FeedTree network does not invalidate this comparison; Pastry and Scribe are specifically designed such that their demands on each node increase only logarithmically with the overall size of the network. Even if every subscriber to Slashdot's feed (about 400,000 over the period from which this data is drawn) participated in FeedTree, each node will only be responsible for forwarding to a few nodes (if any at all).

The actual bandwidth figures for Slashdot may be somewhat lower; available data includes only the number of "hits" on the feed, so we use an average feed size of 16K (reasonable for Slashdot) to approximate the daily traffic. We expect that many clients are able to reduce the bandwidth of many of their requests using the HTTP optimizations described in Section 3.1, but this savings is unlikely to account for more than an order of magnitude. The FeedTree bandwidth requirements remain substantially smaller.

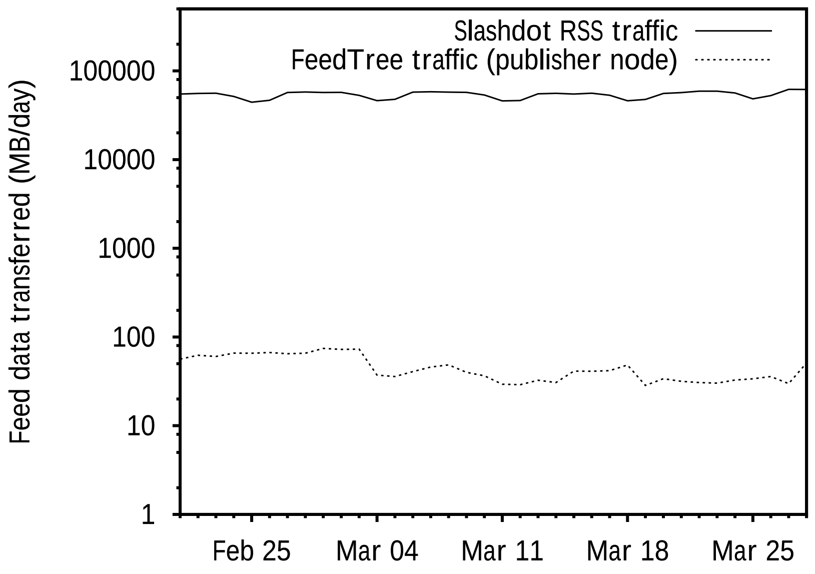


Figure 5.12: Slashdot RSS traffic vs. FeedTree. The actual bandwidth of a FeedTreePublisher node is compared (in a semilog plot) with the approximate bandwidth of the Slashdot.org main RSS feed between 21 February and 28 March. Slashdot figures, provided by email by Jamie McCarthy and Patrick Schoonveld of Slashdot and OSDI, included the number of hits per day for this period; while they did not include actual bytes transferred, an approximate feed size of 16K (typical for Slashdot) is used to generate the above graph.

5.5 Summary

We find that the overhead imposed on FeedTree nodes by the Pastry network is modest, and well within the available CPU available to modern personal computers and the available network bandwidth enjoyed by residential broadband subscribers. Our experiments show that FeedTree scales well, thanks to its Pastry underpinnings; we observe that nodes experience negligible overhead as network size increases. The deployed public FeedTree network, by contrast, gives us an opportunity to observe the network dynamics of a "wild" peer-to-peer application. This data shows a FeedTree community that is able to sustain its demand for popular and obscure feeds with a modicum of additional bandwidth investment, lending credibility to the argument that publishers would do well to adopt peer-to-peer techniques such as ours.

6

CONCLUSIONS

We are pleased to observe, based on our deployment experiences, that the FeedTree concept is fundamentally sound. A global system composed entirely of cooperating peers (modulo some modest rendezvous services) is able to cooperatively shoulder the burden of distributing feed updates. Moreover, feed service (as defined by timely delivery) is strictly better for participants in FeedTree, which is the opposite effect users expect from conventional Web-based bandwidth-saving solutions (*viz.*, Web caches).

FeedTree is thus an early example of a public, *self-sustaining* distributed system based on FreePastry. The authors have in fact performed very little maintenance on the network since its launch: our only ongoing services to the community are a Web site and a Pastry rendezvous address (described in Section 5.2.1). Despite our occasional (benign) negligence, FeedTree continues to provide service to its users.

But how *many* users, in the end? Despite all our efforts to make FeedTree interoperate with existing software and run on any platform, end user adoption was weaker than we had hoped. We speculate that this was chiefly the result of two factors: a problem of incentive and a problem of technology. The incentive problem we observed is as follows: our "pitch" to users was first, to get feed updates faster, with a secondary benefit of decreased strain for publishers. On the whole, users seemed uninterested in saving bandwidth, which was the original motivation for this work, and only slightly interested in prompt feed updates. It is possible (if not likely) that any users who are deeply concerned about feed update delay are also *unconcerned* with publisher bandwidth, and so choose to poll feeds as rapidly as their software (or the publisher) will allow.

It stands to reason, then, that publishers should have had the most incentive to adopt FeedTree, encouraging their readers to do the same. Several major websites showed interest in the concept but little willingness to investigate actual FeedTree deployment; it is probable that the existence of inexpensive, easy-to-use third-party feed hosts (namely those described in Section 3.1.2) has caused publishers to consider the "RSS bandwidth problem" to be solved. We believe the problem is still quite open, though it may be temporarily suppressed by the benevolence of third parties.

As for the *consumers* of Web feeds, those early adopters who were intrigued by FeedTree—whether due to its promise of rapid updates or because the concept looked interesting—faced technical hurdles that often proved insuperable. Early versions of FeedTree had only a command-line interface, excluding interested but nontechnical users. Some users objected to running a Java virtual machine alongside their usual suite of applications.

The most significant challenge to prospective users, however, came in simply trying to connect to the FeedTree network. Like most peer-to-peer systems, FreePastry demands that participating nodes be reachable at a stable IP address and port. This poses difficulties for a public Internet deployment in which the majority of potential users are trapped behind network address translation and firewalls not under their control. Few users were able to configure their routers to allow them to participate in the Pastry ring at the heart of FeedTree.

Those users who were able to use FeedTree, however, found that it delivered on its promises. Beyond timely and efficient feed delivery, one of the most powerful aspects of FeedTree is the way in which it builds upon existing implementations and applications. We observe that asking all existing users of a popular system to abandon it for a proposed "better" solution is typically a losing proposition; FeedTree therefore works with existing feed formats and existing software tools. By providing an HTTP proxy, we allow end users to leverage existing feed reading applications, whose user interfaces are optimized for the unique problem of presenting news feeds to users. Similarly, the FeedTree publishing tool will sign and publish any existing feed—no matter which blogging package or custom content management system created it—allowing publishers to join the system with ease. Finally, by trading at every step in existing feed formats like RSS and Atom, FeedTree is prepared both to take advantage of improvements in these feed formats and to allow compatible implementations in the future.

Having successfully developed and deployed FeedTree, we confirm that it provides better feed service to every one of its users while decreasing those users' bandwidth impact on the feeds they read. Even feed publishers unaware of FeedTree benefit from its existence; FeedTree users pool their polling resources, decreasing both update detection time and bandwidth consumption. At any time, any publisher can decide to begin pushing signed updates to FeedTree subscribers, resulting in immediate delivery of authentic feed data. We conclude that peer-to-peer distribution of Web feeds is both a marked improvement over current feed distribution protocols and a novel application of peer-to-peer research systems.

Postscript. After the original public announcement of FeedTree (and the ensuing Slashdot coverage [63]), the author was contacted by Skinkers [60], a London-based technology firm specializing in desktop-based notification applications. This so-called "skinker" software behaves a bit like an RSS reader, but is typically dedicated for use with a single information source (for example, an internal corporate communications feed, or a stock ticker, or sporting schedule).

Early deployments of these tools for Internet-scale distribution of news had already begun to reveal bandwidth as a serious factor threatening the scalability of the approach, especially when timeliness of updates is a priority as well. Technology like FeedTree was seen as a way to dramatically improve the future prospects for Skinkers, who decided to commission a custom FeedTree client to explore how Scribe could be integrated with the existing Skinkers software. The 2005 collaboration between Skinkers and myself also included several technical lectures on structured overlays, Pastry, Scribe, and the FeedTree architecture.

While Skinkers did not end up incorporating FeedTree into their product, it has partnered with Microsoft to develop a new Pastry-based peer-to-peer micronews distribution system called Skinkers Live Delivery Network. [62] It represents the first technology transfer from the Microsoft Cambridge Research Labs [23, 61, 51], the site of the original Pastry research.

BIBLIOGRAPHY

[1] Azureus distributed hash table (version 21 Feb 2006). http://www.azureuswiki.com/index.php/DistributedTrackerAndDatabase.

[2] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: Hypertext Transfer Protocol – HTTP/1.0, May 1996.

[3] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993.

[4] DHT network in BitComet (version 8 June 2005). http://wiki.bitcomet.com/help/DHT_Network_in_BitComet.

[5] BitTorrent DHT protocol. http://www.bittorrent.org/Draft_DHT_protocol.html.

[6] K. Burton. A pending ping crisis? http://www.feedblog.org/2005/09/a_pending_ping_.html, Sept. 2005.

[7] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: high-bandwidth multicast in cooperative environments. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Operating Systems Review, pages 298–313. ACM Press, Oct. 2003.

[8] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 20(8), Oct. 2002.

[9] J. Challenger, A. Iyengar, P. Dantzig, D. Dias, and N. Mills. Engineering highly accessed Web sites for performance. *Lecture Notes in Computer Science*, 2016:247–??, 2001.

[10] Y. Chu, S. Rao, and H. Zhang. A case for end system multicast. In *ACM Sigmetrics*, pages 1–12, June 2000.

[11] Low-bandwidth push is on. *CNET News.com*, Apr. 1997. http://news.com.com/2100-1001_3-279269.html.

[12] Netscape pushes Netcaster. *CNET News.com*, Apr. 1997. http://news.com.com/2100-1001_3-278849.html.

[13] Cobweb: A high-performance content distribution network. http://www.cs.cornell.edu/People/egs/beehive/cobweb/.

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

[15] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, Oct. 2001.

[16] E. X. DeJesus. The pull of push. *BYTE*, Aug. 1997. http://www.byte.com/art/9708/sec6/art4.htm.

[17] P. Deutsch. RFC 1951: GZIP file format specification version 4.3, May 1996.

[18] C. Ellerman. Channel definition format. http://www.w3.org/TR/NOTE-CDFsubmit.html.

[19] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[20] M. Franklin and S. Zdonik. "Data in your face": push technology in perspective. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD98).*, pages 516–519, June 1998.

[21] M. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, California, Mar. 2004.

[22] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[23] N. Gohring. Microsoft shares technology with two more start-ups, June 2006. http://www.networkworld.com/news/2006/060606-microsoft-shares-technology-with-two.html.

[24] A. Haeberlen, A. Mislove, A. Post, and P. Druschel. Fallacies in evaluating decentralized systems. In *Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS'06)*, Feb 2006.

[25] O. Heckmann and A. Bock. The eDonkey 2000 protocol. Technical Report KOM-TR-08-2002, Multimedia Communications Lab, Darmstadt University of Technology, Dec 2002.

[26] M. Horton. RFC 1036: Standard for interchange of USENET messages, Dec. 1987.

[27] S. Iyer, A. Rowstron, and P. Druschel. SQUIRREL: A decentralized, peer-to-peer web cache. In *Proceedings of the 21st annual symposium on Principles of Distributed Computing (PODC'02)*, Monterey, CA, July 2002.

[28] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *4th Symposium on Operating System Design & Implementation (OSDI'00)*, San Diego, CA, Oct. 2000.

[29] S. Junnarkar. PointCast sends its final push broadcast. *CNET News.com*, Feb. 2000. http://news.com.com/2100-1023-237059.html.

[30] B. Kantor and P. Lapsley. RFC 977: Network News Transfer Protocol, Feb. 1986.

[31] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent store. In *Proc. ASPLOS'2000*, Cambridge, MA, November 2000.

[32] Y. Kulbak and D. Bickson. The eMule protocol specification. Technical Report TR-2005-03, the Hebrew University of Jerusalem, 2005.

[33] H. Liu, V. Ramasubramanian, and E. G. Sirer. Client and feed characteristics of RSS, a publish-subscribe system for web micronews. In *Proceedings of the Internet Measurement Conference (IMC)*, Oct. 2005.

[34] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, Berkeley, CA, Mar. 2002.

[35] R. C. Morin. HowTo RSS Feed State. http://www.kbcafe.com/rss/rssfeedstate.html, Sept. 2004.

[36] Netscape Communications Corp. *My Netscape Network*, Mar. 1999. http://www.purplepages.ie/RSS/netscape/rss0.90.html.

[37] T.-W. J. Ngan, A. Nandi, A. Singh, D. S. Wallach, and P. Druschel. On designing incentives-compatible peer-to-peer systems. In *Proceedings of the 2nd International Workshop on Future Directions in Distributed Computing (FuDiCo'04)*, Bertinoro, Italy, June 2004.

[38] M. Nottingham and R. Sayre. RFC 4287: the Atom syndication format, Dec. 2005. http://www.ietf.org/rfc/rfc4287.txt.

[39] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus: an architecture for extensible distributed systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 58–68, New York, NY, USA, 1993. ACM Press.

[40] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. E. Mohr. Chainsaw: Eliminating trees from overlay multicast. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS'05)*, Ithaca, New York, Feb 2005.

[41] K. Park and V. S. Pai. Deploying large file transfer on an HTTP content distribution network. In *Proceedings of the First Workshop on Real, Large, Distributed Systems (WORLDS '04)*, San Francisco, CA, December 2004.

[42] PlanetLab. http://www.planet-lab.org/.

[43] P. Querna. Blog pings are stupid. http://paul.querna.org/journal/articles/2005/09/28/blog-pings-are-stupid, Sept. 2005.

[44] L. Rainie. Pew Internet & American Life Project: Data memo: Public awareness of Internet terms, July 2005. http://www.pewinternet.org/PPF/r/161/report_display.asp.

[45] V. Ramasubramanian, R. Peterson, and E. G. Sirer. Corona: A high performance publish-subscribe system for the world wide web. *Proceedings of Networked System Design and Implementation (NSDI '06)*, May 2006.

[46] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.

[47] S. Ratnasamy, M. Handley, R. Karp, , and S. Shenker. Application-level multicast using content-addressable networks. In *Proceedings of Third International Workshop on Networked Group Communication (NGC '01)*, London, England, 2001.

[48] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A Public DHT Service and Its Uses. In *Proceedings of ACM SIGCOMM'05*, Philadelphia, Pennsylvania, Aug. 2005.

[49] S. C. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, pages 127–140. USENIX, June 2004.

[50] ROME: RSS and Atom Utilities for Java. https://rome.dev.java.net/.

[51] D. Rowe. Skinkers: tech transfer deal accelerates market leader. Microsoft Startup Zone, Jan. 2007. http://microsoftstartupzone.com/blogs/david_rowe/archive/2007/01/04/1000.aspx.

[52] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov. 2001.

[53] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, Oct. 2001.

[54] RSS-DEV Working Group. *RDF Site Summary (RSS) 1.0*, Dec. 2000. http://purl.org/rss/1.0/spec.

[55] R. Salz. InterNetNews: Usenet transport for Internet sites. In *Proceedings of the USENIX Summer 1992 Technical Conference*, pages 93–98, Summer 1992.

[56] D. Sandler, A. Mislove, A. Post, and P. Druschel. FeedTree: Sharing Web micronews with peer-to-peer event notification. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS'05)*, Ithaca, New York, Feb. 2005.

[57] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking*, 2002.

[58] R. Scoble. A theory on why RSS traffic is growing out of control. http://radio.weblogs.com/0001011/2004/09/08.html#a8200, Sept. 2004.

[59] E. Sit, F. Dabek, and J. Robertson. UsenetDHT: A low overhead Usenet server. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS'04)*, Feb. 2004.

[60] Skinkers. http://www.skinkers.com/.

[61] Skinkers strikes equity for technology deal with Microsoft. Press release, June 2006. http://www.skinkers.com/index.php?id=32{&}tx_mininews_pi1[showUid]=10{&}cHash=ab6159b7f8.

[62] Skinkers raises UKP2m to fund the launch of innovative peer-to-peer technology for live content delivery. Press release, Jan. 2007. http://www.skinkers.com/index.php?id=32{&}tx_mininews_pi1[showUid]=53{&}cHash=46f0916e9e.

[63] Slashdot.org. Faster Feeds Using FeedTree Peer-To-Peer, Feb 2006. http://slashdot.org/article.pl?sid=06/02/20/1719241.

[64] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.

[65] D. Stutzbach and R. Rejaie. Improving lookup performance over a widely-deployed DHT. In *Proc. IEEE INFOCOM 2006*, Barcelona, Spain, April 2006.

[66] μTorrent FAQ (accessed 1 jan 2007). http://www.utorrent.com/faq.php.

[67] V. Venkataraman, P. Francis, and J. Calandrino. Chunkyspread: Multi-tree unstructured peer-to-peer multicast. In *Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS'06)*, Santa Barbara, California, Feb 2006.

[68] L. Wang, K. Park, R. Pang, V. S. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Proceedings of the USENIX 2004 Annual Technical Conference*, Boston, MA, June 2004.

[69] D. Winer. *RSS 2.0 Specification*, Sept. 2002. http://blogs.law.harvard.edu/tech/rss.

[70] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.