An Optimized Web Feed Aggregation Approach for Generic Feed Types

David Urbansky, Sandro Reichert, Klemens Muthmann, Daniel Schuster, Alexander Schill
Dresden University of Technology
Faculty of Computer Science
Institute of Systems Architecture
01062 Dresden, Germany
{david.urbansky, sandro.reichert, klemens.muthmann, daniel.schuster, alexander.schill}@tu-dresden.de

Abstract

Web feeds are a popular way to access updates for content in the World Wide Web. Unfortunately, the technology behind web feeds is based on polling. Thus, clients ask the feed server regularly for updates. There are two concurrent problems with this approach. First, many times a client asks for updates, there is no new item and second, if the client's update interval is too large it might be notified too late or even miss items.

In this work we present adaptive feed polling algorithms. The algorithms learn from the previous behaviors of feeds and predict their future behaviors. To evaluate these algorithms we created a *real* set of over 180,000 diversified feeds and collected a dataset of their updates for a time of three weeks. We tested our adaptive algorithms on this set and show that adaptive feed polling reduces traffic significantly and provides near-real-time updates.

1 Introduction

The central objects of interest in this paper are web feeds. Feeds are a technology used in the World Wide Web to notify interested users of recent updates and changes to a web site's content. They are transferred as XML messages in one of the two standard formats *RSS* or *Atom*. Usually they are fetched by a program called feed reader. Feeds are used by news sites, blogs, and social media portals to announce new content to everyone interested.

To reduce traffic, most feeds—83 % of the ones we used in our research—provide some means to return feed content only if it has changed since the last request. One is the support of HTTP *LastModified*, another the support for the *ETag* header field. Additional methods are provided by the feed standards directly. The *Time to live* (ttl) element provides information on the time the next update will occur. To prevent polls on seldom updated times such as weekends or during night hours, some feeds support *skipHours* and *skipDays* elements. In addition, there are blogs streaming their content through an endless HTTP stream removing the load imposed by feed readers. This technique is unfortunately not adopted by many blog providers (Hurst and Maykov 2009). Furthermore, there are patch systems such as *pubsubhubbub* (Google 2010) which can be seen as a moderator between feed readers and servers. All of these solutions only work if both, client and server support the extensions, which is rarely the case. Pubsubhubbub, for example, is only supported by 2 % of the feeds in our dataset. In fact, many clients still support only a fixed update interval, usually set to one hour by default and never changed by most users (Liu, Ramasubramanian, and Sirer 2005). Thus, it is obvious that the current situation of a very heterogeneous landscape of feed technologies will persist in the medium term and the predominant way to exchange feeds remains inefficient polling.

The goal of our research is to lower the problems mentioned so far without changing the current technology. We show that it is possible to make a feed reader learn about the update behaviors of different feeds. An optimal strategy has to minimize the number of polls to find a new item and the delay between the publish time of the item and the poll. We call this approach the *min delay policy*.

We make the following contributions in this paper:

- A set of algorithms to predict a feed's future update behavior based on its update history.
- An evaluation of these algorithms on a corpus of 180,000 *real* feeds from which we got an update history over a time of three weeks yielding over 19 million items.

The rest of this paper is structured as follows. First, we briefly introduce our dataset followed by describing algorithms to adapt a feed reader to the update behaviors of different feeds. Then, we present the evaluation of our approaches. Next, we present an overview of related work in the area of improving feed update algorithms and finish with conclusions and an outlook on future work.

2 Dataset

To evaluate feed reading algorithms and their update strategies we need a dataset of feeds and their items. Our most important requirement on the dataset was the diversity of the feeds to represent a *real world snapshot* of the feeds on the web. We made the dataset publicly available at the research platform Areca[1]. To compile the dataset, we created keyword lists with tags and categories from web pages such

[1] http://www.areca.co/1/Feed-Item-Dataset-TUDCS1

Copyright © 2011, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

as delicious.com and dmoz.org. We used more than 70,000 keywords and combinations of them to query Yahoo and utilized the autodiscovery[2] mechanism on the first top 1,000 web pages that were returned for each query. This way we gathered 240,000 feeds for a wide variety of topics from 220,000 different domains. After polling each feed during a time span of three weeks and storing each item along with its timestamp, title, link, number of items, and size, we had a *real* dataset of over 19 million items.

3 Update Strategies

In this section we describe four update strategies, three of them take update behaviors of feeds into account. In the *min delay policy*, we try to read a feed as soon as it has at least one new item and use the following notation for modeling the problem. u is the update interval, that is, the predicted time until a new item is posted in the feed. w is the window size of the feed, that is, the number of items that can be collected at once when looking up the feed. i_n is the publish time of the nth item and p is the time of the poll. For all strategies, we limit the poll interval to [2 min, 31 d] to prevent constant polling and not polling at all.

3.1 Fix

The simplest algorithm is to poll the feeds in fixed intervals. We chose one hour and one day as the baseline intervals Fix 1h and Fix 1d, as these are common change intervals of websites (Grimes and O'Brien 2008). Furthermore, one hour is a common default interval in feed readers[3].

3.2 Fix Learned

We learn the time between two new items and use it as a fixed interval for future polls. At the very first poll, we collect all publish dates for the items, calculate the average interval length as shown in Equation 1 and do not change this interval anymore in the future. If the interval can not be calculated by Equation 1, we set a fixed interval to one hour.

$$u = \frac{1}{w-1} * \sum_{n=1}^{w-1} (i_{n+1} - i_n) \tag{1}$$

3.3 Moving Average

The moving average strategy does almost the same as Fix Learned but updates the predicted interval u continuously. The idea is that the last intervals between new items are a good predictor for the next item. About 30 % of the feeds in our dataset change their post frequencies quite often. By continuously averaging the intervals, we can pick up those frequency changes.

There are two possible observations at each time of poll p. Either there are new items in the window of the feed or not. If there are new items, we apply the moving average formula as shown in Equation 1. If there are no new items we have polled too early and should increase u. Calculating the update interval with Equation 1 would yield in no difference because we still have the same items in the window as the last time we polled. We therefore add a *virtual item* on the timeline at the time p so that $i_{virtual} = p$ and remove the oldest one from the window. If we now calculate the update interval again, we have the chance to increase it and skip gaps of no postings in the feed.

We can therefore update the old interval by removing the last item and adding the new virtual item as shown in Equation 2 where u_{n-1} is the current update interval.

$$u_n = u_{n-1} - \frac{i_1 - i_0}{w-1} + \frac{p - i_{w-1}}{w-1} \tag{2}$$

Consider the case shown in Figure 1. The feed has a fix window size $w = 5$ and periods of more frequent updates (*period 1 and 3*) and periods of rare updates (*period 2*). We now poll the feed for the very first time at p_0 and calculate the update interval u_0 by averaging the update intervals of the five items in our window w_0. The next poll p_1 is therefore $p_0 + u_0$. At p_1, we have again five items in our window and the new update interval will not change much since the one new item that we found at p_1 and the one that was the last in our window have similar intervals. After p_1, *period 2* of less frequent updates starts. At p_2, we still have the same items in our window that we had at p_1, so calculating the new update interval u_1 using Equation 1 would not change anything. We want to increase u, however, and therefore we drop the oldest item in the window and replace it with the virtual item (light gray circle). The following polls are farther apart from each other since the update intervals u_1 to u_8 slightly increase due to fewer posted items. At p_9, *period 3* of more frequent updates has started and we must decrease the update interval in order to lower the delay to new items. We find five new items in our window and the new, smaller update interval stays about the same until p_{13} having only short delays to each new posted item in the feed.

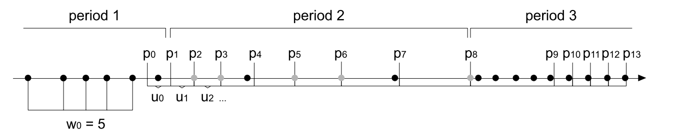


Figure 1: Changing Update Intervals in the Moving Average Strategy.

3.4 Post Rate

The Post Rate strategy learns the update pattern for each feed and uses this data to predict future updates. A very similar strategy has been described in (Adam, Bouras, and Poulopoulos 2010; Bright, Gal, and Raschid 2006) where it was called *individual-based history adaptive strategy*.

The goal is to find the update interval u where we can expect that a new item has been posted. We use a meticulous post distribution of the feed, that is, we have 1440 time frames with item post rates for each minute of the day. We can now calculate the number of expected new items over

[2] Autodiscovery mechanism searches the web page's header for explicitly stated feed URLs.

[3] http://www.therssweblog.com/?guid=20070408105324

a certain time span using Equation 3, that is, summing up the probabilities for a new item in each minute. Our update interval u is $t_{goal} - t_{now}$ with t_{goal} being a future point in time so that the number of expected new items is 1 or higher.

$$\text{newItems}(t_{goal}) = \sum_{t=t_{now}}^{t_{goal}} \text{postRate}(t) \tag{3}$$

4 Evaluation

We evaluate the update algorithms for the *min delay policy*, where the goal is to minimize lookups but get the next new item as soon as possible. A good strategy should therefore minimize the number of polls (and by that also the network traffic) as well as the delay to each new item. As baselines we compare the Fix strategies Fix 1h and Fix 1d with the Fix Learned, the Post Rate, and the Moving Average strategy.

4.1 Network Traffic

First, we compared the network traffic of the five strategies during a time frame of three weeks. Figure 2 shows the total transferred amount of gigabytes for each strategy. Polling a feed once every hour, regardless of its real update behavior, leads to the highest traffic (Fix 1h). The learning strategies Moving Average and Post Rate lead to a saving of 90 % compared to the Fix 1h strategy. Interestingly, the Fix Learned strategy performs a bit better than Fix 1h although it takes the initial update interval into consideration. We can therefore conclude that feeds change their update behaviors frequently and the update strategy benefits from picking up on the change. Fix 1d performes slightly better than the learning strategies and leads to the fewest network traffic.

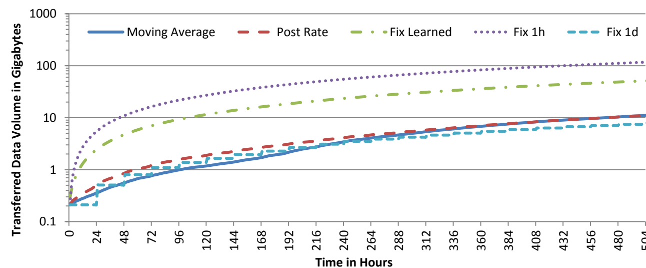


Figure 2: Transferred data volume during three weeks.

4.2 Timeliness

Second, we compared the delays for discovering newly posted items. In Figure 3, we averaged the delays over all feeds that had at least 300 updates during the creation of our dataset. This subset contained 1,541 feeds. As expected, Fix 1h is about 30 minutes too late on average, while Fix 1d (not shown) reads each new item about 11 hours too late. Fix Learned performs best with a "constant" average delay of seven minutes, followed by Moving Average with eight minutes after 25 polls that have at least one new item. Post Rate continuously learns from the growing history, but even after 300 polls, it performs still worse than Moving Average and Fix Learned. In this discipline, the learning strategies outperform Fix 1h and Fix 1d which was best in Sec. 4.1.

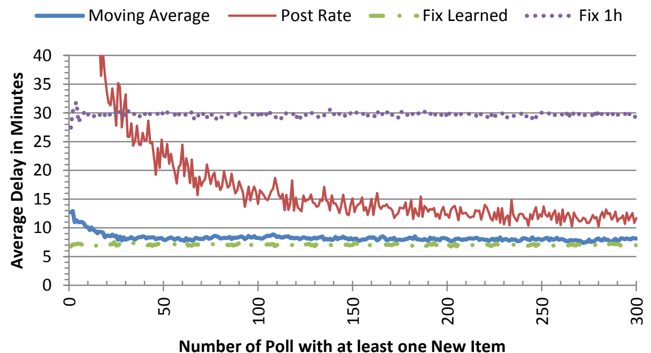


Figure 3: Timeliness of the update strategies, 300 polls.

4.3 Update Interval Prediction

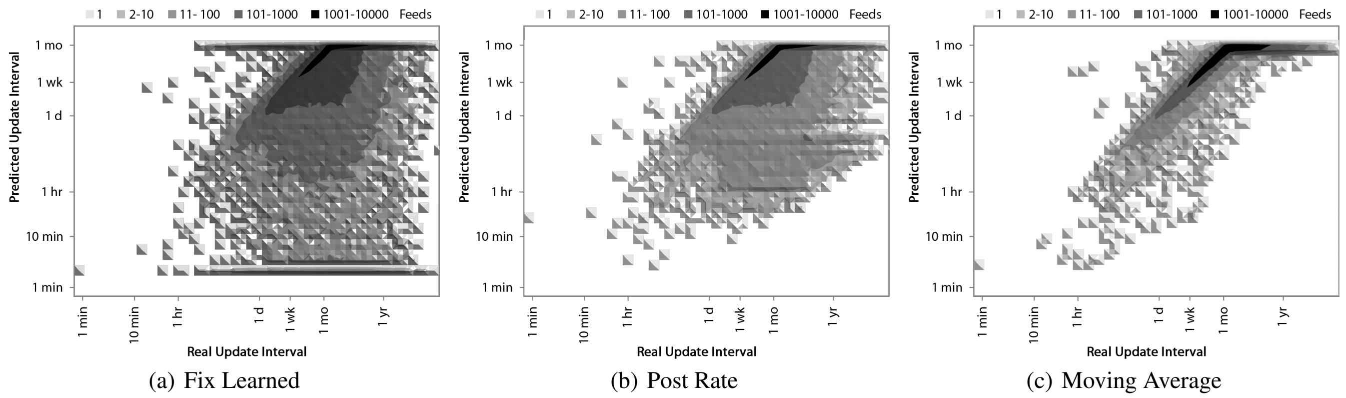
So far, we analyzed that some strategies adapt their polling intervals and decrease their average delays. Infrequently updated feeds negatively influence the delay in minutes when averaging over all feeds. Therefore, Figure 4 compares the feeds' real and predicted update intervals, using heatmaps with log_{10} scales for all three axis to show how precise the strategies are for a variety of update intervals. Each dot represents one feed. A perfect strategy would be a straight line with $f(x) = x$. Polling with $f(x) < x$ means too often and $f(x) > x$ results in a delay. We omitted plotting the baselines Fix 1h and Fix 1d as they were horizontal lines at 1 hr and 1 d respectively. For a big proportion of the feeds in our dataset, we see real update intervals of more than three weeks because these feeds were updated infrequently such as once a month or year.

All figures have 2 minutes / 31 days as the polling intervals' lower / upper bound in common. Fix Learned tends to poll too often since it uses the feed's first window to calculate the interval. The bottom line at two minutes is caused by feeds that once had a high update frequency but relapsed into silence. Post Rate performs better than Fix Learned. It has a lower variance and does not stick at the lower bound. Moving Average's performance is best. It shows a small variance at the whole range of real update intervals.

4.4 Overall Comparison

Table 1 shows a comparison of the five update strategies regarding the average delay to finding a newly posted item and the number of polls to discover *one* new item. We used two different averaging modes *Feeds* and *Polls*. In the *Feeds* averaging mode we averaged the measures for all polls that each strategy made for each feed and averaged them per strategy. In the *Poll* averaging mode, we averaged over all polls of all feeds so feeds with many polls dominate the final result. We can see that, while the Fix 1h strategy has the smallest delay on average, it also polls most often. The Post Rate and Moving Average strategy have similar delays, but the Post Rate strategy polls more often on average.

The best strategy can only be determined in the context of a real application. If plenty of resources are given and it is more important to get updates on time, the Fix 1h (or in even smaller intervals) could be used. We can, however,



(a) Fix Learned (b) Post Rate (c) Moving Average

Figure 4: Comparing real and predicted update intervals.

Mode	Strategy	Delay	PPI	Error
Feeds	Fix 1h	**30m**	222.06	**6,662**
	Fix 1d	12h:9m	9.52	6,931
	Fix Learned	54h:33m	41.90	137,139
	Post Rate	56h:51m	3.57	12,174
	MAV	74h:37m	**1.96**	8,773
Polls	Fix 1h	30m	21.02	179
	Fix 1d	12h:25m	1.91	1,423
	Fix Learned	1h:14m	7.77	570
	Post Rate	51m	5.88	300
	MAV	**4m**	**1.61**	**6**
PPI = Polls Per Item; MAV = Moving Average.				

Table 1: Comparison of Update Strategies.

conclude that the strategy which satisfies our two requirements of a short delay and few polls is the Moving Average. We calculate the *Error* as shown in Equation 4. The lower the error the better the strategy. Also, compared to the Post Rate strategy which has been proposed in literature before, the Moving Average is simpler as it does not require a post history to be stored.

$$\text{Error} = \text{Avg. Delay} * \text{Polls Per Item} \tag{4}$$

5 Related Work

Several solutions have already been proposed to the polling problem. Most similar to our approach are the algorithms described in (Adam, Bouras, and Poulopoulos 2010; Han et al. 2008; Bright, Gal, and Raschid 2006; Lee et al. 2008; 2009). All four propose adaptive feed polling algorithms to address the problems mentioned so far. However, all of them have different optimization goals and focus on a specific subset of feeds. Many of them can not adapt to a change in a feed's update behavior dynamically. Most are following the probabilistic Post Rate algorithm, which needs a large previous update history.

6 Conclusion and Outlook

This paper shows the details of four web feed polling strategies applied to a large corpus of feed update histories. We collected these histories from more than 180,000 *real world* web feeds over a timespan of three weeks and showed the improvements of our adaptive strategies in comparison to classical and state-of-the-art adaptive feed polling. The algorithms used are Fixed, Fixed Learned, Post Rate, and Moving Average. The Moving Average strategy satisfies these two requirements best. Efficient polling is an important issue as long as push mechanisms are not used on a large scale.

In the near future, we will analyse enhancements such as weighting of items, different history sizes for Moving Average and Post Rate, combinations of algorithms and usage of feed-specific features such as *skipHours* and *skipDays*.

References

Adam, G.; Bouras, C.; and Poulopoulos, V. 2010. Efficient extraction of news articles based on RSS crawling. In *International Conference on Machine and Web Intelligence (ICMWI)*.

Bright, L.; Gal, A.; and Raschid, L. 2006. Adaptive pull-based policies for wide area data delivery. *ACM Trans. Database Syst.* 31(2):631–671.

Google. 2010. pubsubhubbub - Project Hosting on Google Code. http://code.google.com/p/pubsubhubbub/.

Grimes, C., and O'Brien, S. 2008. Microscale evolution of web pages. In *WWW '08: Proc. of the 17th international conference on World Wide Web*, 1149–1150. New York, NY, USA: ACM.

Han, Y. G.; Lee, S. H.; Kim, J. H.; and Kim, Y. 2008. A new aggregation policy for RSS services. In *CSSSIA '08: Proc. of the 2008 international workshop on Context enabled source and service selection, integration and adaptation*, 1–7. New York, NY, USA: ACM.

Hurst, M., and Maykov, A. 2009. Social Streams Blog Crawler. In *ICDE*, 1615–1618. IEEE.

Lee, B. S.; Im, J. W.; Hwang, B.; and Zhang, D. 2008. Design of an RSS Crawler with Adaptive Revisit Manager. In *Proc. of 20th International Conference on Software Engineering and Knowledge Engineering–SEKE'08.*

Lee, B.; Hwang, B.; Mastorakis, N.; Martin, O.; and Zheng, X. 2009. An Efficient Method Predicting Update Probability on Blogs. In *WSEAS International Conference. Proc. Mathematics and Computers in Science and Engineering*. WSEAS.

Liu, H.; Ramasubramanian, V.; and Sirer, E. G. 2005. Client behavior and feed characteristics of RSS, a publish- subscribe system for web micronews. In *Proc. of he 5th ACM SIGCOMM conference on Internet Measurement*, 29–34. USENIX Association.