Efficient extraction of news articles based on RSS crawling

George Adam, Christos Bouras and Vassilis Poulopoulos
Research Academic Computer Technology Institute, and
Computer and Informatics Engineer Department, University of Patras
Rion, Greece, GR26500
adam@cti.gr, bouras@cti.gr and poulop@cti.gr

Abstract

The expansion of the World Wide Web has led to a state where a vast amount of Internet users face and have to overcome the major problem of discovering desired information. It is inevitable that hundreds of web pages and weblogs are generated daily or changing on a daily basis. The main problem that arises from the continuous generation and alteration of web pages is the discovery of useful information, a task that becomes difficult even for the experienced internet users. Many mechanisms have been constructed and presented in order to overcome the puzzle of information discovery on the Internet and they are mostly based on crawlers which are browsing the WWW, downloading pages and collect the information that might be of user interest. In this manuscript we describe a mechanism that fetches web pages that include news articles from major news portals and blogs. This mechanism is constructed in order to support tools that are used to acquire news articles from all over the world, process them and present them back to the end users in a personalized manner.

1 Introduction

The World Wide Web has grown from a few thousand pages in 1993 to more than billions of pages at present. Moreover, all these pages that exist on the WWW are neither static nor stable, but they form a continuously changing system. The results from the studies of Cho and Garcia - Molina [7] and Fetterly et al. [8] prove that a huge number of web pages change on a weekly or even on a daily base. The mechanisms that were invented to make Web seem less chaotic need information and waste a great amount of time in order to collect it. These mechanisms are defined as Web Crawlers, bots or spiders, and their main scope is to create an offline copy of the web pages in order to support engines that need these pages as feeds. Web crawlers are an essential component of all search engines and are increasingly becoming more and more important in data mining and other indexing applications.
Much research has been done for constructing crawlers that will have ”fresh” collection of web pages. The basic algorithm of most of the existing crawlers is; keep a collection of uniform resource locators (URLs) and, starting from a root page (starting point), they begin to visit and download all the pages that derive from the root page in a periodical manner. However, web pages are changing at different rates which mean that an intelligent crawler should be able to decide how often and which pages of the collection have to be revisited by using an efficient method [9]. This leads to creation of crawlers that have at least two basic modules, one for periodical crawling (scheduled) and another for incremental crawling (update the most frequent changing pages). In [2] and [3] is denoted that most web pages in the US are modified during the US working hours a statement that is extremely logical and useful for crawling mechanisms. In [7], Cho and Garcia-Molina introduce the extreme difference in ”page change” rates for each unique domain. Arasu et al. in [1] report a half-life of 10 days for web pages in order to create an algorithm for maintaining the freshness of the ”offline collection” of the WWW.
In [12], [13], [17] and [10] some specific strategies are introduced for effective crawling and for parallel crawling. The basic idea that lies behind parallel crawling is a manager which is assigned with the task of

organizing the set of terminals-crawlers that access and download pages. The manager should read and update accordingly the distributed databases with data collected from every terminal-crawler in order to prevent situations of duplicate entries.

In this manuscript we describe advaRSS, a crawling mechanism, which is created in order to support ”peRSSonal” [5] [4], a mechanism that produces personalized RSS feeds. As the mechanism intends to be base utility for systems offering collections of news articles in real time to internet users, it has to maintain a fresh collection of the latest news. In order to achieve this we utilize the RSS feeds which are widely known and supported by almost every news portal and weblog. In contrast to the common crawling mechanisms our system is focused on fetching only news articles from the major and minor portals worldwide (multilingual), in order to deliver personalized content to users. The news is produced in a random order any time of the day and thus the crawling mechanism must periodically poll the sources and check for changes many times per day. To make this resource intensive task more efficient, the system has to learn to predict the rate that an RSS is publishing articles, based both on the static features and the complete posting history. Finally, the system is able to adapt its internal procedure to the time that an RSS changes and thus it is possible to learn the temporal behaviour of a feed. The system consists of two phases, a training phase and the normal execution phase. When a new RSS is added to the feed URLs of advaRSS the system is running in training phase for the specific RSS. During the training phase we utilize simple algorithms for fetching the articles from the RSS feed and in parallel we construct the posting history of the RSS per hour and per day. After four weeks, the system has enough information about the RSS in order to start the normal execution phase. During the normal execution phase the posting history of the RSS and by applying advanced algorithms we are able to fetch articles more efficiently.

A large body of related work in crawling, content identification and information retrieval that attempts to solve similar problems using various techniques exists and we will present the most representative of it. The crawlers are widely used today and as it was expected, the last few years, many crawlers were designed and constructed. Ubicrawler [14] is a distributed crawler written in Java, and it has no central process. It is composed of a number of identical ”agents”; and the assignment function is calculated using consistent hashing of the host names. The crawler is designed to achieve high scalability and to be tolerant to failures. WebRACE [18] is a crawling and caching module implemented in Java. The system receives requests from users for downloading Web pages. The most outstanding feature of WebRACE is that it is continuously receiving new starting URLs to crawl from (”seed”). PolyBot [15] is a distributed crawler written in C++ and Python. It is composed of a ”crawl manager”, one or more ”downloaders” and one or more ”DNS resolvers”. Collected URLs are added to a queue on disk, and processed later to search for seen URLs in batch mode. In contrast to the common crawling mechanisms, advaRSS is focused on fetching only news articles from the major and minor portals and blogs worldwide (multilingual). The difference between advaRSS and a usual crawler is the fact that the news is produced in a random order any time of the day and thus the crawling mechanism has to be efficient in order to obtain each news article that occurs in a portal immediately after its publishing. Another difference is that advaRSS intends to feed peRSSonal with information and thus its output should be ”easily readable” and accessed. Another approach utilizes the fact that different domains have very different ”page change” rates and consists of a scheduling mechanism which estimates the next update timing of web pages based on their update history using the Poisson process [16]. In [6] is proposed an additional feature, which includes the politeness constraint which indicates that we may only probe the source at most n times and that no two probes may be spaced less than delta time units apart. This policy is intended to minimize the required bandwidth and to prevent the crawler from being blocked from the source. Finally, in [11] Sia et al. study how the RSS aggregation services should monitor the data sources to retrieve new content quickly using minimal resources and to provide its subscribers with fast news alerts. Their experiments prove that, with proper resource allocation and scheduling, an ”RSS aggregator” can provide useful content

significantly fast. The rest of the manuscript is structured as follows. In the next section we present the architecture of advaRSS and then the flow of information. In the fourth section we analyze the algorithms and following we present some experimental evaluation that was done in order to present the accuracy of the sub-systems. We conclude with remarks and future work.

2 Architecture

The architecture of the advaRSS, consists of multiple sub-systems which are assigned with specific roles in order to achieve high speeds of between them and between the crawling sub-system and the peRSSonal mechanism. The basic parts of the system are (a) the centralized database (using peRSSonal's database), (b) the crawler's controller and (c) the terminals that execute the fetching and analysis. The database is used for storing permanent information, the controller is used in order to organize and distribute the procedure and finally the terminals are used in order to fetch the HTML pages from the internet. A

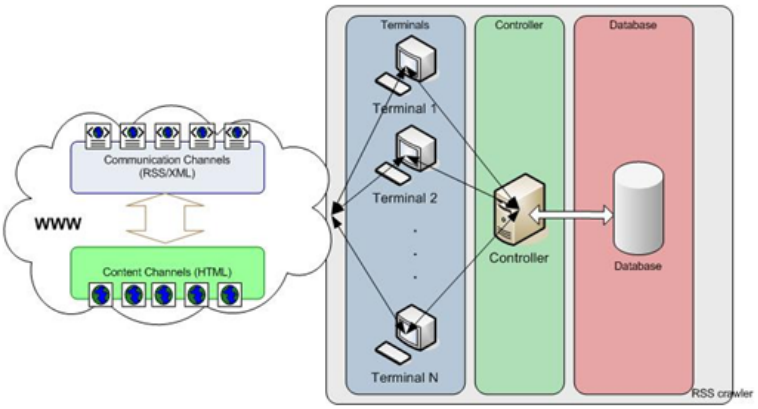

Figure 1: advaRSS's Architecture

single database is used in order to store the starting URL, which is links to XML files, and the results of the parsing procedure. The XML files are RSS feeds which means communication channels provided by news portals. Additionally, the database stores information concerning the articles that are fetched from the advaRSS crawler. The information that is needed about an RSS feed is its URL and some meta-data while for an article we need information like the title, the HTML code, its URL, the language in which it is written, the date that it was fetched and the category (business, entertainment, politics, etc) in which it is pre-classified by the website from which it derives. The second sub-system of the RSS crawler mechanism is the controller of the whole procedure. The controller is assigned with two major tasks. The first is the direct communication with the database (only the controller can interact with the database) and the second is the job assignment and checking of the terminals. The controller is the part of the mechanism that includes the main procedures and feeds the terminals with URLs from which to download information. Two kinds of information are usually forwarded for download: (a) URL to XML file and (b) URL to plain HTML file which has to be downloaded. In parallel, the controller examines the outputs of the terminals' analysis and stores any information to the database.

3 Algorithm Analysis

The algorithmic procedure is divided into two phases. The first phase is the training phase of the system which utilizes simple metrics in order to fetch articles from an RSS but in parallel it construct a change

rate history of each article per hour and per day.

3.1 Training Phase

For every new RSS in the database the system maintains three different variables, which are stored to a central database and two files that are stored locally to each terminal. The variables are used in order to create a system that is able to adapt on the RSS changing behavior. By changing behavior we define the time period which implies changes to an RSS (added articles). The files are used in order to quickly check if an RSS had changed since the last time it was parsed. The first file includes the hash code of the XML file and the date that was fetched in the last execution while the second file includes all the titles of the articles that were added in the latest fetched XML file (RSS feed).
The hash code is sensitive to minor changes to the file and thus we can easily obtain information if the file has changed while the second file includes all the titles of the articles contained in the last downloaded RSS and help us realize if the new RSS file that is fetched includes any new articles. This prevents communicating with the database in order to check for any minor or major RSS change. We utilize both hash codes and dates in order to check if an RSS has changed as both of them are needed to cover all the possible situations of a changing web files. The possible situations are: a) a file has not changed and its modification date has not changed, b) a file is unchanged but for a reason its modification date is changed and c) a file is changed and thus its modification date is changed.
It is inevitable that the state of unchanged modification date and changed hash code cannot possibly occur unless somebody is changing the date of the file manually which is not supposed to be done when dealing with web files. By utilizing the modification date we are able to understand if a file has changed without even downloading it. If the modification date is changed we need to download it in order to check its hash code.
An algorithm is utilized in order to update the execution time of an RSS in order not to check every RSS in every execution of the system. This algorithm is applied as the peRSSonal system, which is supported by the crawler, can have hundreds of RSSs to be parsed and it is not possible to check every single RSS every ten minutes that is the time limit of the system's execution.

```
feeds[]=Fetch_rss_having_zero_ToE();
Foreach(feeds[] as url)
If(not modified)
newTimer=T+M;
M = M + 30%T
T = newTimer
Else
T=1
M = 20%M+80%T
End If
ToE = M
End For
```

Where: M (median): a number that indicates how often an RSS has to be parsed. T (timer): a number representing how many executions, at least, have passed since the RSS was last changed. ToE : a counter that is reduced in every execution of the system. When it reaches the value zero, the RSS has to be parsed.
This algorithm is utilized in order to change the ToE of each RSS according to its rate of change which is represented by the timer (T). The median (M) variable is used in order to store the rate of change, which is a value that changes according to the rate of change of an RSS and can learn the behavior of an article.

One of the most basic parts of the system is the execution time updater. It is a subsystem of the mechanism that is able to change the execution time of each RSS which indicates how often an RSS has to be parsed. We have designed this algorithm by observing how an RSS changed during the 24 hours of a day. The following diagram shows the number of the articles published in an RSS at a given hour, divided by the overall number of articles published by this RSS during the whole day.

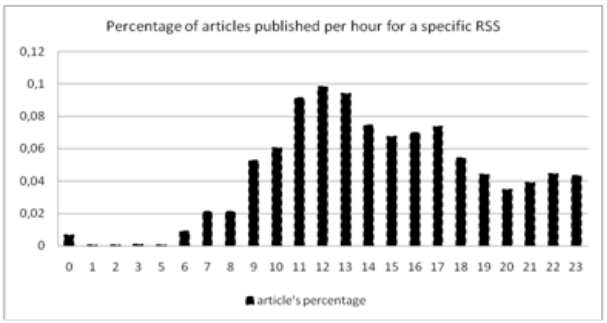

Figure 2: Percentage of articles published by a specific RSS per hour

As it is obvious from Figure 2 the system should check the specific RSS very frequently between 11 and 14 o'clock as almost 30 percent of the articles of the specific RSS are published during these hours while advaRSS should not check very frequently this RSS from 1 to 6 o'clock as less than 2 percent of the articles are published during these hours. We should find thus a way in order to change the time an RSS is processed by the system. We define the ToE, M and T variables already mentioned in the previous paragraphs. We follow the next equations in order to change their values. If ToE greater than 0

$$ToE = ToE - 1 \tag{1}$$

If ToE = 0, there are two different cases which are: (a) no new articles occur in the RSS feed compared to the latest time the RSS was parsed and (b) new articles (one or more) occur. In the first case we apply the next equations in the presented order:

$$\begin{aligned} Temp &= T + M \\ M &= M + 0.3T \\ T &= Temp \end{aligned} \tag{2}$$

In the second case we apply the following equations:

$$\begin{aligned} T &= 1 \\ M &= 0.2M + 0.8T = 0.2M + 0.8(becauseT = 1) \end{aligned} \tag{3}$$

After each execution (ToE = 0) we set the value of ToE to M:

$$ToE = ceil[M] \tag{4}$$

From eq. 1 it is obvious that for every execution of the system the ToE is decreased in order to reach the value 0, which indicates that the RSS has to be parsed. When the ToE is zero, then we check the current RSS feed for changes. If no changes are observed then the T variable is increased by M (the times that the RSS was unchanged) and the M variable is changed according to Eq. 3. If the file has not changed (no fresh articles) then M increases slightly, while when the file has changed (indicating fresh articles) M decreases dramatically. We concluded to these changes to the Median (M variable) in order to

achieve two basic goals: (a) when an RSS is not changing it is checked fewer times per day and (b) when an RSS starts to be updated the Median variable is rapidly decreasing and we manage to adapt the ToE to the rate of change of the article. The maximum value for M is 80. Using this value, if we check the RSSs every X minutes then the RSS will be checked every 80*x minutes which means at least 18/x times per day in the worst case. This seems to be quite a lot for RSSs that change once a week but still, we have to maintain a fresh collection. On the other hand, the minimum value for M could be 1 indicating check for fresh articles every X minutes.
The initial values of M and T are 4 and 1 consequently. T is set to 1 in order to point an initial unchanged state for each RSS. We set the value of M to 4 in order to pre-set a small rate of change of the page and then adapt to the actual temporal behavior of the page. ToE is set with an integer random value between 0 and 3 for each RSS in the database. This is translated to: when the system begins its first execution, it checks every RSS with ToE equal to zero and processes it. The RSSs will be found to be changed (as we do not have any previous instance of the RSS during the first execution of the system) and thus the median will be set directly to 1.6 (M=0.2M+0.8T). While all the ToEs of the RSSs will be already reduced, the ToE of the RSS that were just checked will have the value 2 (ceil[M] where M is set to 1.6). By doing this we assure that the first time that the RSSs are going to be parsed they will be checked at least one time in subsets and after that they will start to adapt on their rate of change.
The basic part of the algorithm is eq. 2 and eq. 3 which update the M variable that represents the rate of change of an RSS within a specific time limit. As the rate of change of an RSS may vary during the day we maintain a history of M and we base our system on a specific pattern.

3.2 Crawling using the posting history

During the normal execution phase advaRSS can retrieve the posting history of an RSS by utilizing the information that was recorded during the training phase and, more specifically, the hourly posting rate of the articles. One of the most basic parts of the system is the scheduling mechanism. It is a subsystem of the crawler that, by using the aforementioned posting pattern, manages to schedule the next visit to the RSS feed. During each subsequently visit, the historic information is updated and new predictions are made, leading to a system that is able to adapt on the RSS updating behavior. We concluded to this algorithm by observing how new articles are published by an RSS during the 24 hours of a day. The following diagram shows the average number of articles posted per hour, for a random RSS in our database.

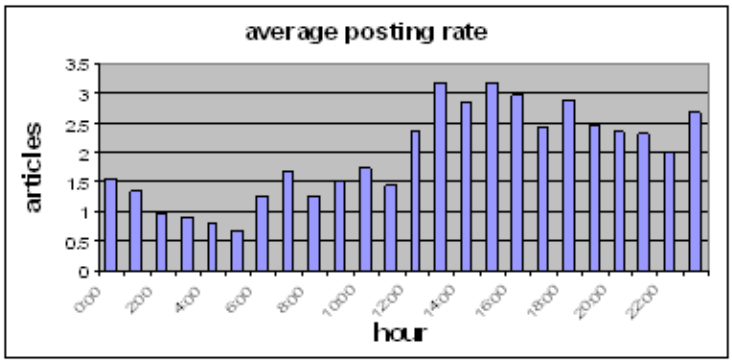

Figure 3: average number of new published articles per hour for a specific RSS

As it is obvious from Figure 3, the mechanism should schedule more frequent visits at hours with high posting rate. Having the hourly past posting pattern of an RSS and the last time that it was retrieved,

we can use the following equation to estimate the expected number of new posted articles since the last retrieval, using the precision of 1 second:

$$articles(t_{now}) = \int_{last}^{t_{now}} \frac{postingRate(t_{now} - t)}{3600} \quad (5)$$

The above equation utilizes posting rate per second by dividing the hourly rate by the total number of seconds in one hour. It is obvious that we expect a high number of new articles from an RSS feed with high posting rate. Due to resource constraints, the mechanism is able to perform only limited retrievals per time period. Thus, the crawler has to decide which RSS sources to contact in order to fetch as many articles as it can. A simple monitoring algorithm that utilizes 5 to schedule a total of k retrievals for each execution should estimate the expected new articles and select the first k with the higher number of articles. Utilizing the fact that the advaRSS crawler can be used as a part of a system offering collections of news in real time to internet users, we can increase its efficiency by including information about the users. The number of subscribers of each feed and their activity on the system can be used in order to modify the above monitoring algorithm. We assume that if a source has more subscribers than others, it should be retrieved with higher priority, in case that all of them have similar posting rates. Putting the above together on a unique ranking metric, we have for an RSS feed f:

$$rank(f,t) = articles_f(t) \cdot (1 + c \cdot subscribers(f)) \quad (6)$$

Parameter c in the eq. 6 can be adjusted to reflect how important the information about the number of subscribers is. In systems that the number of subscribers for each source is unknown, this constant must be set to zero, which means that the ranking metric will not use this information at all. Additionally, the scheduling mechanism that we present, takes into account the politeness constraint, which means that no two subsequent retrievals may be performed in less than x time units apart. We assume that a user can tolerate a delay of 10 minutes for an article, thus we set the minimum time between two subsequent retrievals to this time period. Finally, the mechanism is be able to update the posting pattern for each RSS based on the result of the next retrieval, which is the number of new articles that have been published. The updating process of the posting rates, distributes the number of new articles to each hourly rate, making the mechanism able to adapt to each source. However, a source may have not published a new article for a day, which means that it's posting pattern and ranking metric will be equal to zero. Thus the above algorithm will not retrieve this source ever again. To overcome this problem, the mechanism uses a minimum value greater than zero for the posting rates.

4 Experimental Evaluation

In this section we provide experimental evaluation of the advaRSS crawler performing with or without the posting rate history.

4.1 Evaluation during the training phase

According to our opinion, a crawler has to be adaptive on each URL that it is searching and the workload has to be distributed in order to access parallel a huge amount of data. In that means we have conducted experiments in order to observe the different possible solutions for our crawler and select the most suitable for our case. It is expected that a crawler that is processing multiple RSS at the same time (parallelism) will be faster than a crawler that is accessing its feed URL in a serial manner, though we have to observe if the adaptation algorithmic procedure (file checking for duplicate entries and RSS

changes) consumes too much time. We conducted an experiment with three different systems and measured the execution time for each of them. The first system was without any adaptation (files) and run on a single computer. The second system distributed the procedure over multiple computers, without any adaptation while the third system was utilizing the adaptation algorithm with multiple terminals.
The time of execution by utilizing files for duplicate entries or RSS changes with distributed procedure is the most suitable for our system as it takes only 10 seconds to search for new articles in the RSSs and add the new articles in the database for every time the crawler is running. In other cases the Time of Execution is more than 2 minutes in average. The time measured is the average time needed for each time the system is executed on hundreds of RSS feeds in order to analyze them and fetch all the new articles that occur. In average the system is able to add more than 2500 articles daily to its database.

The second adaptation of the system is on the rate of change of each RSS. It is expecting that thousands of RSS URLs will be added on the database when in production. This means that the system will have to check every ten minutes thousands of URLs for new articles (hundreds of RSS X ¿10 URLs in each RSS feed) every time the crawler is executed. By adapting on each RSS feed rate of change we are able to access the RSS feeds not every time that the crawler is running but every time that the system ”believes” that there may be a change to the RSS’s content.

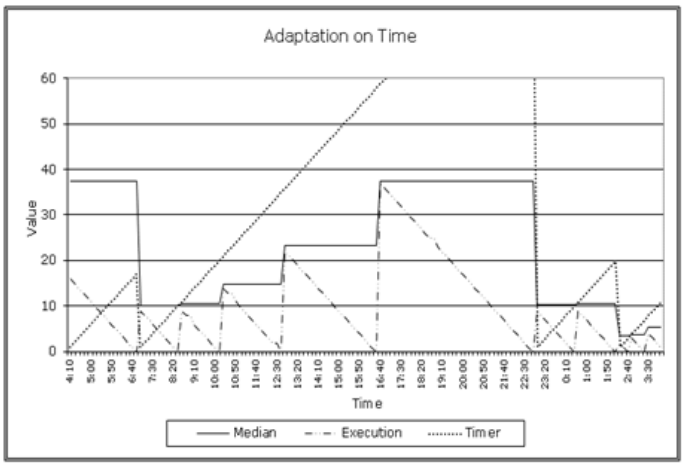

Figure 4: Adaptation of the variable M, ToE and T on Time (1)

As it is obvious from the experiments an RSS is not checked every six to ten minutes that the crawler is executed but it is checked every time that the system ”believes” that a change may have occurred.

By applying the adaptation we check an RSS according to the learning algorithm for the system, which is in average 20 times per day for the RSSs that change too much in a day and less than 10 times per day for RSSs that do not change frequently. An issue that arises from the previous adaptation is how fresh the articles that are entering the system are. This could be translated into a simple assumption: if we are not checking the RSSs periodically for new articles, then there is a possibility to ”lose a change”. As long as we want to provide a real time service to the end users, this means that we must have a new article added to the database within a time limit if this article is published within the same time limit. The time limit that we would like to achieve is at most 30 minutes. We conducted a simple experiment in order to observe how ”fresh” are the articles in our database which means that we have cross-checked the time that the articles were added in our database compared to the time they were published on their original website. The latency of fetching the articles compared to the original time that they were posted on their official website is 35 minutes in the worst case and 27 in the best of the worst cases. It is expected that the time difference of the rest of the articles will be less than 27 minutes as we are investigating the

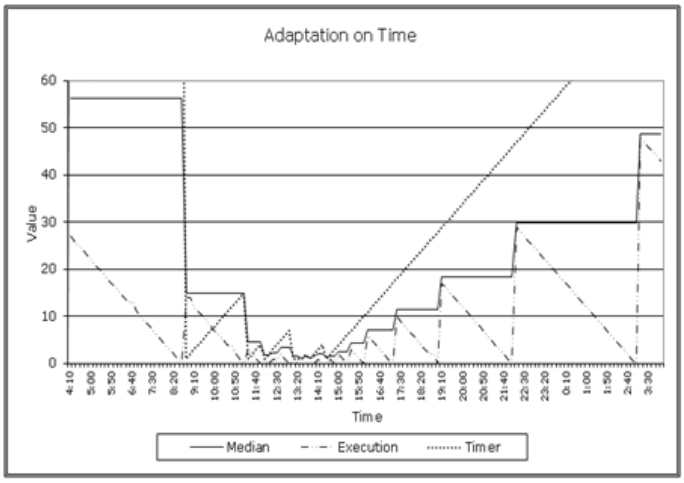

Figure 5: Adaptation of the variable M, ToE and T on Time (2)

10 worst cases. The system has an average of 14 minutes latency which is acceptable compared to the rate that internet users are checking for news.

4.2 Crawling using the posting history

In this section, we compare the proposed monitoring algorithm that utilizes the posting rate history, to other retrieval policies. The experiment procedure lasted 90 days and was conducted using RSS feeds from major and minor portals and weblogs. At the first experiment we put focus on the maximum number of pending articles that a source can have. As pending articles we define the articles that are published but have not been retrieved. The comparison is made using other two monitoring policies. The first is a round-robin policy, which places the RSS feeds in a queue and schedule the retrievals using the FIFO method which means that a source will be revisited after all others have been processed. The second policy uses the posting pattern in order to minimize the total delay of the fetched articles. The delay is defined as the time period between the publishing and the retrieval time.

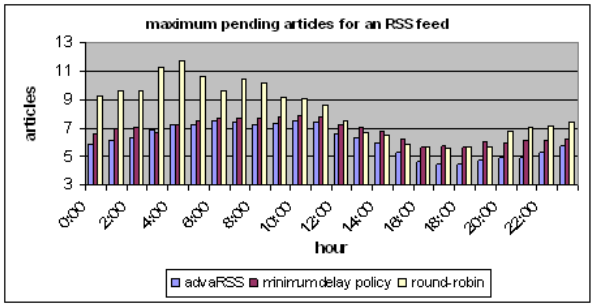

Figure 6: average number of articles, of the RSS with the maximum pending articles

As it is obvious from Figure 10, the policy that minimizes the total delay increases by 11.2 percent the maximum pending articles on the source. Using the round-robin policy we notice an increment of 33.4 percent. The number of articles is an average of the daily measurements. Apart from the above metric, it is interesting to estimate the total pending articles on the system. Thus, the second experiment was conducted using 460 sources and the objective was to calculate the summary of the articles that have

not been retrieved yet, for all RSS sources. Both experiments were made by collecting information about these feeds for a period of three months and applying the aforementioned policies.

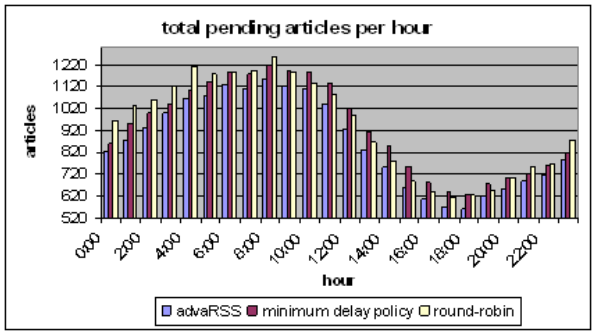

Figure 7: : total pending articles on system, per hour

Figure 11 shows that the pending articles are depending on the posting rate for each hour. We can see that with the "minimum-delay" policy, the total number of pending articles is 7.5 percent more than the proposed policy. Finally, the result of the round-robin policy is approximately 8.5 percent more articles. For the experiments, the average retrieval rate is 15

5 Conclusion and Future Work

In this paper, we described the architecture and implementation details of our crawling system, and also presented some experiments. We showed the importance of adaptation on each domain as it is obvious that the web pages of different domains have different behaviour (change in a different manner). We also highlighted the importance of utilizing RSS feeds in order to retrieve useful content from the Web and how this can be efficiently implemented on a system even with limited resources. In a World Wide Web that has grown enough from the time of its invention, the personalization issue seems to make the difference, and seems to be one of the most important of our era. The advaRSS intends to be the base utility for systems offering collections of news in real time to internet user such as peRSSonal [5], [4] which is a single web place that offers, in a unified way, personalized and dynamically created views of news deriving from RSS feeds. There are obviously many improvements to the system that can be made. A major open issue for future work is a detailed study of how the advaRSS mechanism could become even more distributed, retaining though quality of the content of the crawled pages. When a system is distributed, it is possible to use only one of its components or easily add a new one to it. Additionally what we have to do is to compare the results of our crawler with the implementations of other incremental crawlers that selectively chose which pages to crawl. Though, we believe that our system consists of something more than just a crawler. Our intention is to create a clever system that would be able to collect "fresh" content from the web in order to support, with data, mechanisms specialized on data mining, information extraction and categorization.

References

[1] A. Arasu, J. Cho, H. Garcia-Molina, A Paepcke, and S. Raghavan. *The evolution of the Web and implications for an incremental crawler.* ACM Transactions on Internet Technology, Vol. 1, No. 1, August 2001.

[2] B. E. Brewington and G. Cybenko. *How dynamic is the web?* Computer Networks, Volume 33, Issues 1-6, June 2000.

[3] B. E. Brewington and G. Cybenko. *Keeping up with the Changing Web*. IEEE Computer, vol. 33, no. 5, May 2000.

[4] V. Poulopoulos C. Bouras and V. Tsogkas. *Efficient Summarization Based On Categorized Keywords*. The 2007 International Conference on Data Mining DMIN07, Las Vegas, Nevada, USA, June 2007.

[5] V. Poulopoulos C. Bouras and V. Tsogkas. *PeRSSonal's core functionality evaluation: Enhancing text labeling through personalized summaries*. Data and Knowledge Engineering Journal, Elsevier Science, Vol. 64, Issue 1, 2008.

[6] C. Valentim C. Souza, E. Laber and E. Cardoso. *A Polite Policy for Revisiting Web Pages*. Latin American Web Conference (LA-WEB 2007), 2007.

[7] J. Cho and H. Garcia-Molina. *The evolution of the Web and implications for an incremental crawler*. Proceedings of the 26th International Conference on Very Large Databases, Morgan Kaufmann 2000, September 2000.

[8] M. Najork D. Fetterly, M. Manasse and J. L. Wiener. *A large-scale study of the evolution of Web pages*. Software: Practice and Experience, Special Issue: Web Technologies, Wiley Interscience, Volume 34 Issue 2, June 2004.

[9] S. K. Gupta H. Bullot and M. K. Mohania. *A Data-Mining Approach for Optimizing Performance of an Incremental Crawler*. In Proceedings of the IEEE/WIC International Conference on Web Intelligence, 2003.

[10] H. Garcia-Molina J. Cho and L. Page. *Efficient Crawling through URL ordering*. Computer Networks and ISDN Systems, Volume 30, Number 1, April 1998.

[11] H.-K. Cho K. Cheung Sia, J. Cho. *Efficient Monitoring Algorithm for Fast News Alerts*. IEEE Transactions on Knowledge and Data Engineering, vol. 19, no. 7, July 2007.

[12] M. Najork and J. L. Wiener. *Breadth-first search crawling yields high quality pages*. In Proceedings of the 10th World Wide Web Conference (WWW10), 2001.

[13] M. Najork and J. L. Wiener. *High-Performance web crawling*. SRC Research Report 173, Compaq Systems Research, Compaq Systems Research Center, 2001.

[14] M. Santini P. Boldi, B. Codenotti and S. Vigna. *UbiCrawler: a scalable fully distributed Web crawler*. Software: Practice and Experience, Wiley Interscience, 2004.

[15] V. Shkapenyuk and T. Suel. *Design and Implementation of a High-Performance Distributed Web Crawler*. Data Engineering, 2002. Proceedings. 18th International Conference, 2002.

[16] T. Tamura and M. Kitsuregawad. *Evaluation of Scheduling Methods of an Incremental Crawler for Large Scale Web Archives*. Abstracts of IEICE transactions on Information and Systems (Japanese) Vol.J91-D No.3, 2008.

[17] J. Wolf, M. Squillante, P. Yu, J. Sethuraman, and L. Ozsen. *Optimal Crawling strategies for web search engines*. Proceedings of the 11th international conference on World Wide Web, 2002.

[18] D. Zeinalipour-Yatzi and M. Dikaiakos. *Design and Implementation of a Distributed Crawler and Filtering Processor*. Lecture Notes In Computer Science. Vol. 2382, Proceedings of the 5th International Workshop on Next Generation Information Technologies and Systems, 2002.