OWeB: A Framework for Offline Web Browsing

Ganesh Ananthanarayanan[1]
ganeshan@microsoft.com

Sean Blagsvedt[1]
seanb@microsoft.com

Kentaro Toyama[1]
kentoy@microsoft.com

[1]Microsoft Research India

Abstract

Internet browsing is highly dependent on the real-time network availability and speed. This becomes a significant constraint when browsing over slow and intermittent networks. In this paper, we describe a readily deployable system designed for web browsing over slow, intermittent networks – OWeB, that is minimally dependent on the real-time network availability and requires no changes on the part of the web servers. The system subscribes to Really Simple Syndication (RSS) feeds from web servers and pre-fetches all new content as specified in the feed. Since the RSS feeds are published by web servers they give accurate information about the new and updated content. Efficiency of network usage is achieved by employing standard techniques to handle intermittent networks, and near-complete utilization of all downloaded content results in better resilience in case of interrupted data downloads.

We observed a co-relation between the items in an RSS feed and the homepage of the corresponding website (i.e.) the feed items essentially define the content section of the homepage. As part of OWeB, we developed an algorithm for automatically extracting the template of home pages and then locally stitching the feed items into the template. This results in websites being up-to-date and fully available offline and bandwidth savings, as we only need to download the RSS feed to construct the homepage.

1. Introduction

Internet browsing has traditionally been dependent on the real-time network availability and speed. This dependency becomes a significant constraint when accessing the web over slow and intermittent networks. There is an inordinate delay in downloading the web pages and files and this results in an unpleasant browsing experience. Accessing large chunks of data, or streaming content across the web often incurs problems with latency, interruptions, and poor quality, at least where network connectivity is questionable.

Successful models for offline data access include SMS and E-mail synchronization over mobile phones. The important characteristic of this class of applications is that the end-user never needs to wait for access to data. The content is always available and offline. This results in a positive user experience as the user is insulated from the vagaries of the network. The essential point is to provide a "best-effort" guarantee to ensure that the latest content is always downloaded and present on the device.

In this paper, we focus on improving the Internet experience for websites with associated RSS [14] feeds. To this end, we have designed and built an offline web-browsing system, OWeB. The salient features of the system are (1) intelligent pre-fetching, (2) robust and resilient measures for intermittent network handling, (3) Template Identifier - automatic identification of the core content area of a home page and template extraction, and (4) local stitching of the dynamic content into the template.

OWeB combines the advantages of both pull-based and push-based techniques. It queries the web server for the RSS feeds to obtain information about the new content. These queries are inexpensive as the RSS feeds are small in size. The RSS feeds alleviate the problem of successfully predicting the importance of the new content since they are defined by the website authors and hence are accurate. Unlike traditional push-based models this mechanism is scalable, as the web server's role is limited to just publishing the feed with interested clients downloading the feed and the content. After getting the RSS feed, OWeB fetches the new content as specified in the feed and stores it locally. The system handles intermittent networks by employing well-known techniques like opportunistically downloading the content when the network is available and queuing interrupted downloads for re-trials.

While RSS feeds give information about the new links and web pages on a server, it does not convey any information about the homepage. We believe that the homepage is necessary for a complete and fully offline browsing experience. Based on the observation of a co-relation between the homepage and the RSS feed items, we have developed an algorithm that automatically identifies the core content area of a homepage and extracts the template, and then locally stitches the feed items into the template. This enables our system to construct the homepage given only the RSS feed and since the feeds are significantly smaller than the homepages, is bandwidth efficient.

The important research contributions of this paper are – (1) a robust and resilient system to improve web browsing over slow and intermittent networks by opportunistic and intelligent usage of network resources, (2) template identification algorithm to automatically identify the core content area of a webpage, and (3) stitching algorithm to collate the incoming items from the RSS feeds into the template. Points (2) and (3) together introduce a novel mechanism for latency reduction and data savings by getting to a level of granularity less than a webpage.

It is to be noted that OWeB is ideally designed for content-based websites where there is considerable data to be downloaded and not for websites that require interaction like web-search, uploads etc.

The remaining parts of the paper are organized as follows: Section 2 describes the offline web-browsing system, as well as the algorithms required to identify the template of a web page and stitch the RSS feed items into the template. Section 3 details our experiments and provides an analysis of the results. In Section 4, we talk about related research in intermittent network handling and data extraction from web pages.

We conclude with a summary and future directions in Section 5.

2. OWeB Framework

Our aim was to build a readily-deployable offline web-browsing system that would ensure that the most recent content is always available on the client and require no changes on the part of the web servers. Our approach was to explore the existing protocols which worked towards this end and leverage their benefits to build a robust framework on top of it. One such protocol was Really Simple Syndication (RSS) [14]. RSS is a Web Content Syndication format through which web servers can publish information about new or updated content on their websites. RSS feeds are in XML 1.0 format. The RSS feeds can be visualized as a collection of items with every item describing a new content on the web server. The web server can publish information about *anything* that is new on its web site – links, images, audio/video, etc. It is significant that the site-author defines the items and so this eliminates the problem of the client having to predict the importance of the new content. Also, the fact that RSS is popular makes our framework easily and readily deployable without expecting any significant changes on the part of web site authors.

OWeB queries the server for the RSS feeds and then downloads the content as specified in the feed. Obtaining new/updated content over intermittent connections is a significant challenge. We address the issue through our offline web-browsing system by employing standard techniques in literature like queuing and periodic re-trials of interrupted downloads. The aggregation of the benefits provided by RSS, queuing and re-trials greatly improves the bandwidth efficiency of the system and adds resilience under varying connection speeds.

While RSS feeds give information about the new links on a server, they do not convey any information about the homepage. We observed an inherent relation between the homepages and their corresponding RSS feeds. The RSS feed items make up the core content area of the homepage and the remaining parts of a page namely the directory structure, search bar, copyright information, etc. can be effectively assumed to be static. To exploit this, we developed an algorithm to identify the core content area of a homepage and essentially extract its template. We leverage the inherent tree structure of web pages and compare the corresponding sub-trees for this purpose. The RSS feed items can be filled into this template to construct the homepage and since the feeds are smaller in size than the homepages themselves, we achieve a reduction in the amount of data needed to be downloaded and also a complete offline browsing experience.

2.1. Component Interaction and Control Flow

The primary components involved in our system are:

- User Interface
- Download Manager
- Local repository of downloaded content
- Web Servers *(external to the system)*

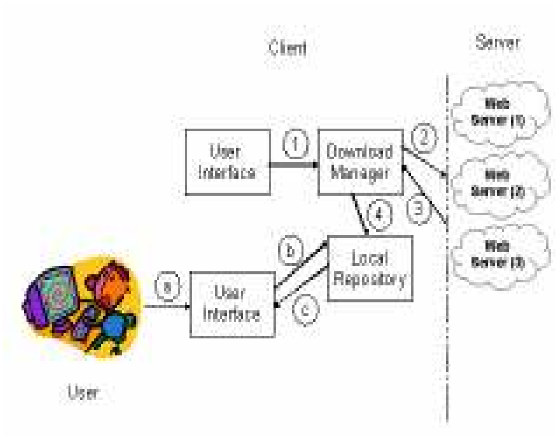

Fig.1. Component Interaction

The components interact in the following manner in a typical synchronization sequence. Note that these set of steps are performed periodically, with a user-settable time period *(Fig 1)*.

1. The application gives the list of feeds to be synchronized to the download manager.
2. The download manager contacts the appropriate Web Servers and requests for the feed and possible new content.
3. The download manager downloads all new content from the Web Servers (Steps 2 and 3 are explained in detail in the following section).
4. The download manager places the content in the local repository.

Now that the data is available offline, the user can browse through them unconcerned about the network availability.

a. The user requests the application for the viewing the content.
b. The user interface requests the local repository for the content.
c. The repository hands over the content to the interface for rendering.

2.2. Intermittent Network Handling

Intermittent network handling is an important mechanism built into the system. Data transfers over intermittent networks are often likely to be interrupted and we overcome this problem by integrating standard techniques like queuing and periodic re-trials into our system. OWeB also provides a highly robust and seamless mechanism to recover and resume downloads in case of interrupted data transfers.
Fig.2 illustrates the series of steps in a download sequence. The first decision box determines whether the current download is a fresh request or a resumption of a previously interrupted download. The backing off between re-trials ensures that our requests do not clog an already congested network. After all feeds have been downloaded and synchronized (or at least an attempt has been made), then we start servicing the files slotted for later trials. For every file listed for re-trial, the same series of steps as given in Fig.2 is followed. If we are unable to download a file even in a re-trial, we use an exponential back-off algorithm and perform periodic re-trials to obtain it. This re-trial goes till the earlier of the two times – *min (time of next synchronization attempt, no more files to re-try)*.
The synchronization is periodic and happens in the background. Periodically, the web servers are queried for the feeds and the latest content thereby synchronizing the local storage with the latest available content on the server.

2.2.1. Resilience under Failed Downloads

OWeB is highly resilient in handling interrupted downloads. It uses the standard browser's technique of storing all partially downloaded data on secondary storage and re-uses them. Interrupted downloads are always resumed and not re-started. HTTP's *byte-range request format* [13] is utilized for this purpose. Interrupted downloads of large audio and video files can be resumed without duplicating data downloads.

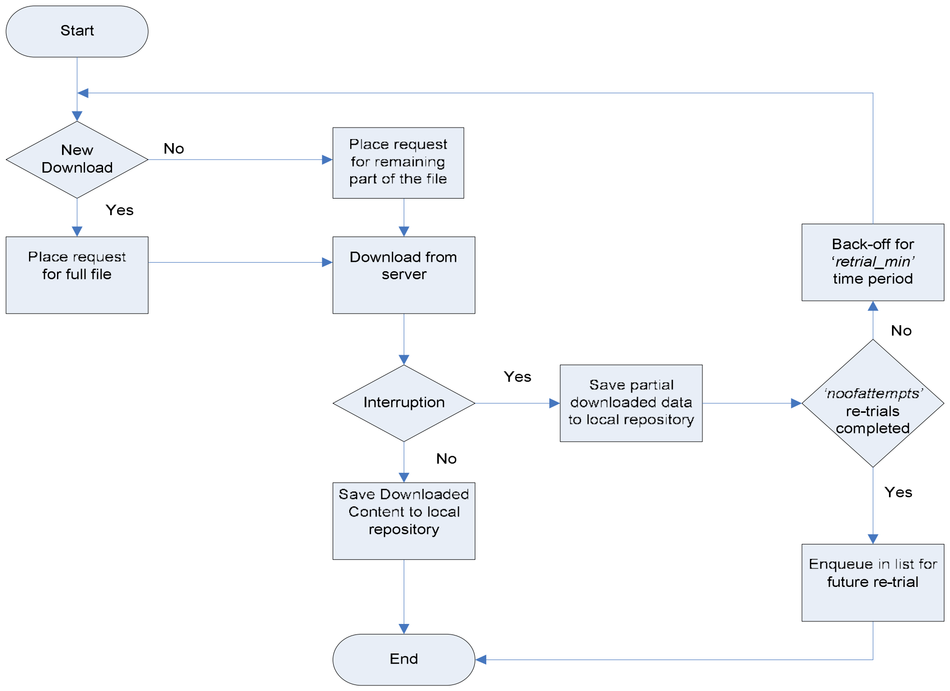
Start
New Download
No
Yes
Place request for remaining part of the file
Place request for full file
Download from server
Back-off for 'retrial_min' time period
No
Interruption
Yes
Save partial downloaded data to local repository
'noofattempts' re-trials completed
No
Save Downloaded Content to local repository
Yes
Enqueue in list for future re-trial
End

Fig.2. Flowchart for a typical download sequence

2.3. Page Stitching Mechanism

The RSS feed items do not directly convey any information about the homepages of web servers. We believe that the homepage is necessary for a complete and fully offline browsing experience. The components present in a homepage can be broadly classified as directory structure, search bar, core content, advertisements and footnotes. We developed an algorithm to identify the core content area of a homepage and essentially extract its template. The algorithm was based on the observation that the core content area is the one housing the items present in the RSS feed and also the most dynamic part of the page. This template can be used to collate the items from the RSS feed to construct the homepage locally. Fig. 3 illustrates the working of the template identifier and the feed stitching algorithm. Since the RSS feed and since the feeds are at least an order of magnitude smaller than the homepages, it results in bandwidth efficiency. Most efforts at reducing latency take a boolean decision on whether to download a page in full or not. The fact that OWeB gets to a level of atomicity below a web page increases its bandwidth efficiency.

2.3.1. Algorithm Overview

Document Object Model (DOM) is a form of representation of structured documents as an object-oriented model. DOM is the official World Wide Web Consortium (W3C) standard for representing structured documents in a platform- and language-neutral manner [1]. A Document Object Model (DOM) can be obtained from every well-formed web page.
The DOM structure of a web page is in the form of a well-defined tree. Our observation was that the most dynamic part of the DOM tree houses the items present in the RSS feed or the core content, and the goal is to identify this sub-tree. The other parts of the page contain information like the directory structure, search bar, copyright, etc. which we assume to be effectively static. Our approach is to compare the DOM trees of the page at two distinct points of time. The times are chosen such that the webpage has had a change in the area housing the dynamic content and has got a new RSS feed published. We associate novelty values with every sub-tree and the most dynamic region of the page housing the core content will have a significantly higher novelty value compared to the others.

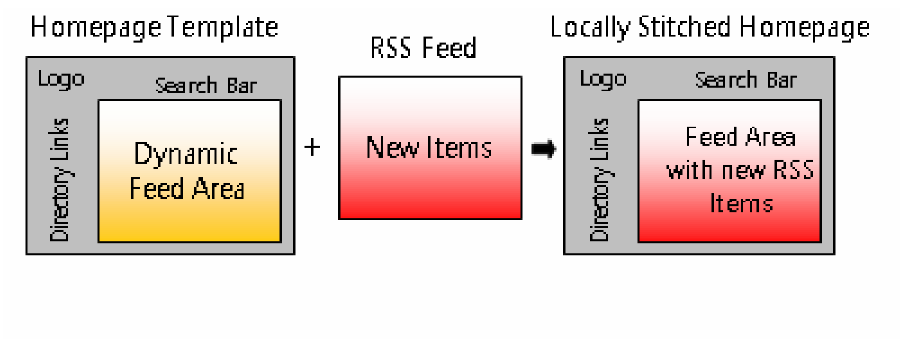

Fig. 3. Template Identifier and Feed Items Stitching

2.3.2. Identification of the Most Dynamic Sub-tree

We observed that the most dynamic sub-tree of the homepage housed the items from the RSS feed or the core content, and our algorithm identifies the most dynamic sub-tree.

We adopt the following definitions on DOM trees from *Shiren Ye et al. [11]*.

1. The similarity of two nodes n_1 and n_2, with attribute sets a_1 and a_2 and tokens t_1 and t_2 is defined as

$$sim(n_1, n_2) = s(t_1, t_2) + |(a_1 \cap a_2)| \qquad (1)$$

where $s(t_1, t_2) = \begin{Bmatrix} 1 \; if \; t_1 = t_2 \\ 0 \; if \; t_1 \neq t_2 \end{Bmatrix}$

2. The repeatability of node $n_1 \in T_1$ with respect to tree T_2 is

$$R(n_1, T_2) = \max_{\forall n \in nodes(T_2)} (sim(n_1, n) + sim(parent(n_1), parent(n))) \qquad (2)$$

3. Normalized repeatability of a node $n_1 \in T_1$ with respect to a T_2 is

$$\overline{R(n_1)} = R(n_1, T_2) / R(n_1, T_1) \qquad (3)$$

4. Novelty of a node n_1 is

$$N(n_1) = 1 - \overline{R(n_1)} \qquad (4)$$

5. Novelty of a sub-tree ST is

$$N(ST) = N(root(ST)) + \sum N(ST_x) \qquad (5)$$

where ST_x is the sub-tree rooted at child-node of *root (ST)*

6. *ST (n)* is the sub-tree rooted at node *n*.

Now we define the algorithm to identify the most dynamic sub-tree, FA (feed area) housing the items from the feed or the core content.

T_1 and T_2 are the two DOM trees of a homepage at two distinct points in time. We need to choose the times such that the webpage has had a change in the area housing the dynamic content and has got a new RSS feed published.

```
For all $n_1 \in$ nodes (T $_1$)
{
    Calculate $R(n_1, T_1)$ using (2)
    Calculate $R(n_1, T_2)$ using (2)
    $\overline{R(n_1)} = R(n_1, T_2) / R(n_1, T_1)$ using (3)
    $N(n_1) = 1 - \overline{R(n_1)}$ using (4)
    Calculate $N(ST(n_1))$ using (5)
}
```

Quite obviously, the novelty value of the sub-tree rooted at the root of T_1 would be the maximum.

```
For all $n_1 \in$ nodes (T $_1$)
{
    maxc ( $n_1$ ) = $\max_{x \in child(n_1)} (N(ST_x))$
    diff ( $n_1$ ) = $N(ST(n_1))$ - maxc ( $n_1$ )
}
```

Hence, the node n_1 with the maximum *diff* value between itself and the child with the maximum novelty is the root of the most novel sub-tree *FA*. *FA* is that sub-tree in which the difference between its own novelty and the novelty of the most novel sub-tree of its children is the maximum. In other words, this is the point in the tree which has the most novel children and so we can deduce that this is the sub-tree that we need to fill with the incoming data from the RSS feed. *(Note that this algorithm has to be run periodically to ensure that the page is consistent with the one at the server. Generally, these changes are not very frequent and happen only with a re-design of a site layout).*

2.3.3. Stitching the feed items into the template

Now, that we have identified the most dynamic sub-tree our job is to fill in the items from the RSS feeds into this sub-tree. Simply put, we replace this particular sub-tree with the items from the RSS feed every time, leaving the other parts intact, to construct the homepage locally.

Let T_1 be the DOM tree at time t_1. Let the RSS feed at time t_1 be f_1. We use the algorithm described in the previous section to identify DT_1 in T_1, the most dynamic sub-tree.

$$f_1 = \{i_1 \mid i_1 \text{ is an item in the RSS feed}\}$$

where every item $i = \{t, l, d, m\}$. *t, l, d* and *m* are the title, link, description and other miscellaneous items associated with every item *i*. DT_1 is a collection of sub-trees with one sub-tree for every item in the RSS feed.

$$\mathrm{DT}_1 = \{ST_i \mid \forall \text{ an } ST_i \text{ for every } i \in f_1\}$$

and $ST_i = \{n_t, n_l, n_d, n_m\}$ where n_t, n_l, n_d, n_m are nodes of the sub-tree ST_i. These nodes have attribute values *t, l, d* and *m* obtained from the corresponding item *i* in f_1. Essentially we consider these nodes as place-holders into which attribute values can be filled.
Now, let t_c be the current time and the feed available now is $f_c = \{i_c \mid i_c \text{ is an item in the RSS feed}\}$ and we need to construct the page locally with this feed.
For every element $i_c = \{t_c, l_c, d_c, m_c\} \in f_c$ we fill in the t_c, l_c, d_c, m_c values into the attribute values of $n_t, n_l, n_d, n_m \in ST_i$ in DT_1.
The number of items in the feed at time t_1 and t_c are not necessarily the same and we need to account for this when we construct the tree DT_1. Hence, in the event of $N(f_c) \neq N(f_1)$, then we need to make the following changes in the set DT_1.

1. If $N(f_c) > N(f_1)$, then we need to add $(N(f_c) - N(f_1))$ members to DT_1. Every member is of type ST_i and the corresponding nodes get their values from the elements of f_c.
2. If $N(f_c) < N(f_1)$, the remove the excess $(N(f_1) - N(f_c))$ elements from DT_1.

Therefore the set DT_1 now consists of elements that have nodes with attribute values filled in from the new feed f_c. So, the tree T_1 will now represent the page for the new feed f_c. We leave the other parts of T_1 unchanged and replace only DT_1 every time.
We note that is a deliberate attempt to update only the core content in the page; we assume that inconsistencies in the non-static elements in or above the most novel sub-tree in the stitched page are tolerable (frequently, they contain advertisements, etc., and other minor changes which do not affect the core content).

2.3.4. Homepages with Multiple Feeds

There are a few homepages composed of multiple sections with each section populated by a different feed. Examples of such pages are general news websites that have a section and a corresponding feed, each for politics, health, sports, finance etc. We need to make minor modifications to our feed stitching algorithm to handle such cases.
The structure of these pages is such that the multiple content sections aggregate under a common parent at some level in the DOM tree of the homepage. So, the template identifier would identify that node as the root of the most dynamic sub-tree *DT*.
Now, we need to find the individual trees DT_1, DT_2... DT_f in *DT* corresponding to the various sections for the different RSS feeds. Every DT_i would have the items from a single feed as its children. Hence there is a direct co-relation between the children of these trees and the items from a feed. By making a direct comparison between the values (title, link and description) present in the child nodes of every DT_i and the values of the items from every feed, we can find the corresponding DT_i for every feed and hence obtain the mapping between the various feeds and the DT_i*'s*. Once we have the corresponding DT_i for every feed, we can apply the feed stitching algorithm as described in the previous section.

3. Experiments and Results

We implemented OWeB on a smartphone running Windows Mobile 2003 SE. We used the General Packet Radio Service (GPRS) as the channel to connect to the Internet over the cellular network. GPRS connectivity is intermittent and unreliable and hence is a good test case for our system. We focused on evaluating the three most important criteria – the accuracy of the template identifier and page stitching algorithm, amount of savings in the data downloaded because of employing the stitching algorithm and the performance improvement because of the robustness and resilience in the system.

3.1. Accuracy of Template Identifier and Page Stitching Algorithm

We tested our template identification and feed stitching algorithm on the homepages of 22 different websites covering diverse classes like news, sports, technology, entertainment and education including prominent websites like *Microsoft Watch, Slashdot, Smartmobs*, *Engadget, Reuters* and *Google News*. Table 2 *(Appendix-A)* has the complete list of sites and their corresponding feeds used to test the accuracy and effectiveness of the template detection algorithm and the stitching algorithm. Of these 22 websites, five of them had homepages which were composed of multiple sections with an RSS feed for every section.

In all the test cases, the most dynamic sub-tree was accurately identified and corresponded to the core content section of the homepage populated by the items from the RSS feed. We take this result as a validation of our observation that these web pages have a core *content* section that is the most dynamic and described by the items in the RSS feeds, and also of the effectiveness and accuracy of our template identification algorithm.

The stitching algorithm functioned successfully in 19 of the 22 test cases including all the five instances where the homepages were fed by multiple RSS feeds. We considered the stitching algorithm to be *successful* if the stitched webpage and the original webpage from the server have no difference in the content section and the static portions. Inconsistencies in other non-static portions like advertisements etc. were considered to be tolerable and hence ignored. The cases where the stitching algorithm failed were when the content section of the homepage was described by multiple RSS feeds or when a single RSS feed defined multiple sections of the page with no clear co-relations. Note that even in these cases, the template identifier accurately identified the most dynamic portion of the webpage. In Table 2 *(Appendix-A)*, while *Slate* had a single feed describing multiple sections, *Movies.Com* and *BBC News* had no clear relation between the feed and the content section.

3.2. Data Savings

The reduction in the amount of data downloaded because of the local construction of the homepage was also analyzed. We considered a sample of four websites (*Microsoft Watch, Slashdot, The Hindu,* and *Smartmobs*) for this purpose. Please refer to Table 3 *(Appendix-A)* for details about the individual web sites. The size of the homepages of the four test web sites together was 217 kilobytes and the size of their RSS feeds put together was 56 kilobytes. The testing was done for a period of three days during which the sites updated their contents and hence their homepages 76 times. So, every time there was a change in the page, we could save on the amount of data downloaded by just getting the feed and not the complete page. We could reduce the amount of data downloaded by 3032 kilobytes which was an appreciable 70% reduction. This value would obviously increase if the testing is done for a longer time including more websites. This particular set of four websites is purely random and we do not expect significant changes in results on a different set.

3.3. Resilience

We used feeds from sources listed in Table 1 *(Appendix-A)* for evaluating the robustness and resilience of the framework. These news feeds provide audio and video files of the telecasts and so are good test cases for the application. The robustness and resilience of the system was very significant when downloading huge audio and video files. The ability to resume downloads and re-try in case of interruptions considerably reduced the time taken and the net amount of data downloaded.

4. Related Work

4.1. Intermittent Network Handling

The problem of handling intermittent networks has been looked into from multiple angles. Client-pull based techniques [2] try to intelligently predict the time of change of data at the server and pull the data. Also, intelligence is needed to pull only the "relevant and useful" data. Studies have been conducted to see the effects of trade-off in the coherency of the data versus the network overload in obtaining it.

Push-based techniques shifted the onus on the server to push relevant data to the clients/proxies. This clearly reduced the problems of coherency and prediction of data change on the client side. Also, this ensured that there wasn't a huge pile of unnecessary data transfers due to faults in the prediction. Notification systems [3] were part of this work. But push-based techniques faced an important issue of not being scalable. Both push and pull based techniques faced the important problem determining the importance of the content [4].

Really Simple Syndication (RSS) [5] is a synchronization mechanism through which web servers can publish information about the new and updated content on their web sites. OWeB combines the advantages of both pull-based and push-based techniques by following a two-step approach – first, it gets the RSS feed from the website and hence gets information about the new content on the server and in the second step, downloads the content as specified by the feed. Since the RSS feeds are very small in size, the first step is a very inexpensive operation and by virtue of the fact that the feeds are site-authored, unambiguously lists the new content. This mechanism is scalable as the web server's role is limited to just publishing the feed with interested clients downloading it when required.

RSS aggregators are applications that take in RSS feeds from the web servers and notify the clients of the availability of new data. While one of the components of OWeB performs the same function as any standard RSS aggregator, our system differs from the RSS aggregators on two important counts – (1) In addition to obtaining the RSS feeds, OWeB also fetches the new content specified in the RSS feed in a network resilient manner, and (2) OWeB has an in-built automatic template identification and stitching algorithm for homepages of the websites.

Work was also done around distribution of data from the content server to geographically distributed caches. These caches reduced the access time from the server to the clients resulting in lower latency and better experiences. Caches needed to maintain coherency and also and relevant" data. Replication in web content also needed to conserve bandwidth by "intelligently" pulling only the "useful has problems and issues associated with it [6].

4.2. Data Extraction and Template Generation

The problem of segregating the dynamic parts of a web page is heavily significant in the context of intermittent networks as it reduces the amount of data needed to be downloaded. This work draws significantly from the existing work done in extracting information from web pages.

The work of extracting information from web pages can broadly be classified into two categories – semantic based techniques and structure based techniques. The semantic techniques are not quite mature enough and also not generically applicable. The structural techniques try to exploit the tree arrangement of the data blocks and are, in general, more deterministic. In our case, we have decided to use structure based techniques because (a) the accuracy requirement is very stringent (b) the diversity of pages we are dealing with is not enormous and (c) the problem can be modeled in a deterministic fashion such that we can apply structure based techniques with a good level of assurance.

Semantic based techniques are generally applicable in scenarios where the accuracy requirement is reasonably soft. Input pages are treated as a list of tokens, and regular expressions are used as the templates [7]. Research also went into formulating template generation algorithms [8]. It used the tag trees of web pages to generate explicit regular tree templates and hence took advantage of the hierarchical structure information contained in trees.

The structure based techniques for information extraction are more deterministic and work with relatively better accuracy levels. Techniques were designed to extract data from lists and tables in pages [9] [10]. But these made assumptions about the presence of well formed tables and lists in the pages. Also, we have to deal with more complicated and generic cases, where the assumptions about the presence of certain components are not acceptable.

Most methods of data extraction, also, do not quite lead to reducing network latency. It is more about manipulating the data extracted and presenting it in a more regulated and concise information [11] [12]. Here we are trying to use the data extraction algorithm in a novel fashion which results in better user experience in terms of network latency and visualization of pages.

5. Conclusion

In this research, we proposed and implemented a system, OWeB to improve the Internet browsing experience over slow and intermittent networks. OWeB made the browsing experience significantly independent of the network availability. The OWeB framework was made robust and resilient by employing standard techniques like queuing and re-trials. We observed a co-relation between the core content section of homepages and the items in the RSS feeds and devised an algorithm to automatically extract the template of a web page and stitch the incoming RSS feeds into the template locally, thereby achieving significant savings in the data downloaded. Our system implementation results validated the correctness of our observation and also illustrated the accuracy of our template identifier and stitching algorithm in addition to appreciable data savings.
Going forward, we plan to integrate OWeB with standard browsers so as to effect a seamless migration from the present-day web browsing experience.
Also, the current semantics available with RSS and HTML do not allow for specifying the static and dynamic parts of the web page. We plan to explore if the specifications can be extended to represent these details and other important details necessary for intermittent networks. An example in this regard is for the web page to specify the importance level of the objects in its page.
As mentioned in Section 3, our algorithm works only when every content section is defined by a single RSS feed. We plan to extend our algorithm to work in scenarios where a single content section is defined by multiple RSS feeds and when a single RSS feed defines multiple sections of the page.

6. References

[1] DOM, http://www.w3.org/TR/DOM-Level-2-HTML/

[2] Q. Yang and H. H. Zhang. Integrating web prefetching and caching using prediction models, *World Wide Web*, 4(4: 299-321), 2001.

[3] Roberto S. Silva Filho et al. The design of a configurable, extensible and dynamic notification service, *Proceedings of the 2nd international workshop on Distributed event-based systems*, 2003.

[4] J. Beaver et al. Scalable Dissemination: what's hot and what's not, *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS*, 2004.

[5] RSS, http://blogs.law.harvard.edu/tech/rss

[6] S. Sivasubramanian et al. Replication for Web hosting systems, ACM *Computing Surveys (CSUR)*, 36(3: 291-334), 2004.

[7] V. Crescenzi et al. RoadRunner: Towards automatic data extraction from large Web sites, *Proceedings of the 2001 International Conference on Very Large Data Bases*, 2001.

[8] S. L. Chuang. Automatic generation of tree-structured templates for information extraction from html documents. Master's thesis, *National Taiwan University*, 1999.

[9] B. Liu et al. Mining data records in Web pages, *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2003.

[10] K.Lerman et al. Automatic data extraction from lists and tables in Web sources, IJCAI-01, *Workshop on Adaptive Text Extraction and Mining*, 2001.

[11] Shiren Ye and Tat-Seng Chua. Detecting and Partitioning Data Objects in Complex Web Pages, *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence (WI'04)*, 2004.

[12] A. Arasu and H. Garcia-Molina. Extracting structured data from Web pages. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages, 2003.

[13] http://www.w3.org/Protocols/rfc2616/rfc2616.html. Hypertext Transfer Protocol -- HTTP/1.1: RFC 2616.

[14] http://web.resource.org/rss/1.0/spec - RDF Site Summary (RSS) 1.0.

.

Appendix – A

Table 1 – Feeds to test the robustness and resilience of the framework

S. No.	Feed Name	Feed URL
1	Science Friday	http://www.sciencefriday.com/audio/scifriaudio.xml
2	Microsoft Watch – Mary Jo Foley	http://feeds.ziffdavis.com/ziffdavis/MicrosoftWatch
3	Slashdot	http://rss.slashdot.org/Slashdot/slashdot
4	Ick Music	http://feeds.feedburner.com/Ickmusic
5	ESPN - Sports	http://sports.espn.go.com/espn/rss/news
6	The Hindu	http://www.hindu.com/rss/01hdline.xml
7	Smartmobs	http://www.smartmobs.com/archive/feeds/index.xml
8	Slate News	http://www.slate.com/rss/

Table 2 – Pages and their feeds to test the template identifier and stitching algorithm

S. No.	Feed Name	Page URL	Feed URL
1	Microsoft Watch – Mary Jo Foley	http://www.microsoft-watch.com/	http://feeds.ziffdavis.com/ziffdavis/MicrosoftWatch
2	Slashdot	http://slashdot.org/	http://rss.slashdot.org/Slashdot/slashdot
3	The Hindu	http://www.thehindu.com	http://www.hindu.com/rss/01hdline.xml
4	Smartmobs	http://www.smartmobs.com	http://www.smartmobs.com/archive/feeds/index.xml
5	Google News – US	http://news.google.com/?ned=us&topic=n	http://news.google.com/?ned=us&topic=n&output=rss
6	CNN – US	http://www.cnn.com/US/	http://rss.cnn.com/rss/cnn_us.rss
7	Slate News $^{\perp}$	http://www.slate.com/	http://www.slate.com/rss/
8	Google News †	http://www.news.google.com/	http://news.google.com/?output=rss
9	Grand Prix †	http://www.grandprix.com/	http://www.grandprix.com/rss.xml
10	Mac Daily News	http://www.macdailynews.com/	http://macdailynews.com/index.php/weblog/rss_2.0
11	Wired News	http://www.wirednews.com/	http://feeds.wired.com/wired/topheadlines
12	Movies.Com $^{\perp}$	http://movies.go.com/	http://movies.go.com/xml/rss/reviews.xml
13	Engadget	http://www.engadget.com/	http://www.engadget.com/rss.xml
14	Wall Street Journal	http://www.wsj.com/	http://online.wsj.com/xml/rss/3_7011.xml
15	New York Times	http://www.nytimes.com/	http://www.nytimes.com/services/xml/rss/nyt/HomePage.xml
16	Rediff †	http://www.rediff.com	http://ia.rediff.com/push/rss.htm
17	Reuters †	http://www.reuters.com	http://today.reuters.com/rss/newsrss.aspx
18	Gadling	http://www.gadling.com/	http://www.gadling.com/rss.xml
19	Yahoo News †	http://news.yahoo.com/	http://news.yahoo.com/rss;_ylt=AqF4O8azzo51uU3nn0kVwCWs0NUE;_ylu=X3oDMTA2cXY4cTA0BHNlYwNudA--
20	BBC News $^{\perp}$	http://www.bbc.co.uk/?ok	http://newsrss.bbc.co.uk/rss/newsonline_world_edition/front_page/rss.xml
21	AutoBlog	http://www.autoblog.com/	http://www.autoblog.com/rss.xml
22	Guardian Football	http://football.guardian.co.uk/	http://www.guardian.co.uk/rssfeed/0,,5,00.xml

† - Web Pages with multiple associated RSS feeds
$^{\perp}$ - Web Pages where the feed stitching algorithm failed

Table 3 – Data Savings by using the stitching algorithm

Website	Homepage Size (KB)	RSS Feed Size (KB)	Number of updates	Data Savings (KB)
The Hindu	27	10	6	102
Smartmobs	57	22	30	1050
Slashdot	59	17	32	1344
Microsoft Watch – Mary Jo Foley	74	7	8	536