News-feed Subscription Management for Intermittently Connected Environments

Jason LeBrun[1], Chen-Nee Chuah[1], Supratik Bhattacharyya[3], and Jean Bolot[2]

[1] University of California, Davis
[2] Sprint ATL
[3] SnapTell, Inc.

Abstract. Polling-based subscription technologies (like RSS and Atom) have quickly become popular forms of information summary, due to the ease of their creation, and their use of already widespread technologies, XML and HTTP. However, for mobile devices, where extraneous data transfer is particularly undesirable due to power concerns and bandwidth constraints, current newsfeed management methods are insufficient. In this work we present the NOOS system, a proxy-based newsfeed management system that allows mobile devices to monitor a large number of feeds without over-taxing system resources.

1 Introduction

XML-based newsfeed technology has quickly become a popular form of content summary publication on the Internet because of the ease of use of the formats for both content providers and content subscribers. There are a number of different feed formats, like RSS[9] and Atom[1]. However, they all convey data in a similar fashion: content is presented in a headline-summary fashion, with optional metadata, using an XML schema of some sort.

At the same time, data-capable mobile devices are moving from the realm of specialized business tool into the realm of ubiquitous consumer devices. The ability to carry a networked computing device with us anywhere opens up new opportunities for data consumption—we no longer have to wait until we are sitting at our desks, or near some sort of WiFi/Ethernet connectivity. We can take advantage of short bursts of free time (waiting for the elevator, riding the bus, waiting for a latte) to read and process small bits of information.

Unfortunately, the combination of these two trends is not co-optimized. While the constant polling of newsfeed updates on a system with constant, high-speed connection to the Internet works sufficiently, small power- and resource-constrained mobile devices are much more noticeably strained by the polling and fetching behaviors required by newsfeed-style data management. There are a number of factors that contribute to this. First, each of the interfaces on a mobile device will gain and lose connectivity as a user moves. Second, many mobile devices are equipped with more than one wireless interface, and these interfaces each exhibit different connection availability patterns. The task of

monitoring and managing all of these interfaces for all applications can strain resource-limited mobile wireless devices. Furthermore, there is no way to tell whether new data is available in a feed without the device actively polling the feed. While this issue is mitigated slightly by using techniques such as HTTP "If-Modified-Since" header, not all hosts implement it. Even if they do, the wireless radio still needs to wake up and query the remote host, which wastes energy. Finally, newsfeeds from the same source contain redundant information (old article summaries) if they are close together in time. Based on these observations, we believe that newsfeed management systems should take an opportunistic approach rather than a brute-force polling approach, and should minimize the sending of redundant information.

In this paper, we present a system called NOOS: Newsfeeds Obtained OpportuniStically. NOOS is a system that solves the problems mentioned above for applications that require periodic information retrieval—like RSS feeds. It is designed to perform well in the disruptive environment of mobile communications. The high-level goal of NOOS is to make newsfeeds of interest available for reading on-demand, without requiring intervention from the user to fetch the feeds, and without placing undue strain on energy and bandwidth resources of the mobile device. NOOS adapts client activity to the publication rate of the newsfeeds, rather the desired polling rate of the client. This means eliminating unnecessary connections to internet hosts, and reducing the transfer of redundant information. Furthermore, we want the system to be able to easily take advantage of all interfaces on the system. Finally, the design of NOOS also ensures that existing newsfeed aggregation software can be used, and that the newsfeed providers do not need to be changed.

The paper is structured as follows: first, we look at related work in newsfeed management on mobile devices. Next, in section 3 we present a detailed design overview of the newsfeed management system, the primary contribution of this work. Then, section 4 explains our prototyping and experimentation , and shows the gain that can be achieved by using the system. Finally we present a conclusion in section 5 with future directions for the work.

2 Background and Related Work

The world wide web has created a venue for publishing that allows huge amounts of information to be generated every day. As the number of sources of information grew, it became clear that the number of sources was too large to be efficiently managed without a compact and standard form of presentation. As a result, formats were created that allow information providers to present a summary of the information on their page in a consistent, low-overhead format. The most popular of these summary technologies is known as RSS. All popular newsfeed formats today are structured XML files that contain a series of items. Each item consists of a title, a link to a main story, and usually a summary.

Currently, there are two popular ways to manage and read newsfeed subscribtions. The first uses a newsfeed aggregater on the user's machine. There are

many of these applications, a popular example being the built-in feed aggregation capability of Mozilla Thunderbird[3]. An aggregater periodically checks the remote newsfeed sites for updates, and stores and presents the articles to the user. The other option is to use an online reader like Google reader[6]. In this case, the feeds are managed on a remote machine, and the user reads the feed via an online interface, e.g., through the web. The NOOS system combines desirable features from both of these options: feeds are managed by a resource-rich remote device, but the feed data is available for viewing on the client's device regardless of connectivity status at the time that the user would like to read.

A system for managing intermittent connectivity was recently published[2]. In this system, arbitrary web snippings can be monitored and sent to a mobile device using a special client-side application. However, our system works with legacy newsfeed aggregation applications, and requires only minimal setup.

The Haggle project, a joint endeavour between Intel Research, Cambridge and the University of Cambridge, addresses the problem of content distribution in environments which exhibit intermittent connectivity[4, 7]. However, the Haggle project is a longer-term project, and does not apply to currently deployed wireless systems without major modifications.

The OCMP project[8], born from joint work between the University of Waterloo and Sprint ATL, provides a framework for mobile devices that allows management of multiple wireless interfaces with different characteristics and intermittent connectivity. We use OCMP as a basis for a prototype implementation. It is described in more detail in section 3.1.

3 Design of NOOS

The NOOS system provides seamless information retrieval for devices that have multiple wireless network interfaces with an inconsistent internet connection. The plugin splits the management of feeds into two asynchronous tasks. A remote proxy fetches feeds from remote hosts, and the client daemon fetches feeds from the remote proxy. The client proxy holds the feeds in local storage until they can be fetched by a news aggregation application. The goal of the system is to retrieve news feed items in a timely fashion, without requiring excessive polling or user interaction on the mobile client side. Figure 1 shows an overview of the system architecture of NOOS.

NOOS works using a configuration file that contains information about feeds to be monitored. This configuration file can be created/modified in a number of ways. The creation of this file is beyond the scope of this report; we simply assume that it already exists. The file simply contains a lists of feeds and the frequency at which they should be polled.

3.1 The Opportunistic Communications Management Protocol - OCMP

Our reference implementation of the NOOS system uses OCMP—the Opportunistic Communications Management Protocol—framework as a substrate. OCMP

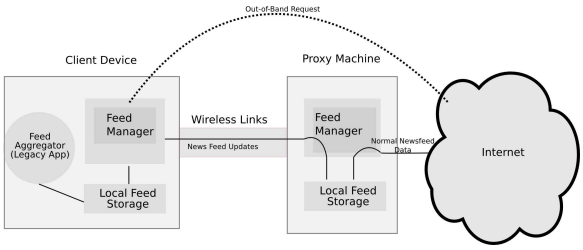

Fig. 1. NOOS & OCMP Architecture

provides a proxy-based architecture for management of disconnection and efficient use of multiple interfaces. OCMP also works for legacy applications. These features make it ideal for the implementation of NOOS. It should be noted that OCMP is *not* required to implement NOOS, we have just chosen it to help us with prototyping. In this section, we describe characteristics of OCMP that are relevant to the NOOS system.

System Capabilities The OCMP framework works by augmenting a state-management and connection-management system with a number of application-specific plugins that control the flow of data between a remote host and a proxy, as well as between the proxy and mobile client. The application plugins allow legacy applications on the client device to co-exist with the OCMP framework.

The system works by passing all client application data through a client-side proxy, via a local socket. This local proxy can perform arbitrary processing on the data, if it is required for the application in question. The data is passed on to the remote proxy, via a single logical connection that is maintained between the mobile client and the proxy (if the mobile device is not currently connected to the proxy, then a connection is established). The remote proxy fetches the requested data on behalf of the mobile client, and can also perform arbitrary processing on the data if necessary, via a proxy-side application plugin. The proxy then sends the requested data back to the mobile client's local proxy, which forwards the data on to the application program.

The OCMP proxy and client are connected through one logical connection. However, this connection can consist of multiple underlying transport layer connections. If the mobile device has multiple interfaces, then a connection is established between the proxy and mobile client on each interface, one transport per interface. Application data is then distributed appropriately over these multiple interfaces, based on various policies. If the mobile device loses connectivity on one pipe, there is no disruption of service, just a change in quality of service. If connectivity is lost completely, the client and remote proxies retain enough state to continue data transfer once connectivity is re-established.

The Filesystem Plugin Framework A recent addition to the OCMP toolset is the filesystem-based plugin creation system. With the new system, it is not necessary to write complicated Java application handlers for every new application that you want to write. Instead, the proxy passes internet data to and from arbitrary handlers in the form of file objects.

In order to implement a plugin in this system, you simply create a directory structure which designates an 'inbound' and 'outbound' directory for both the proxy and the client, associated with a particular application. The OCMP daemon monitors the 'outbound' directories, and when new files appear, it sends them to the proxy on the other side. When a proxy receives a file, it places it in the 'inbound' directory, and calls an arbitrary handler, which can be any program or script that will run on the machine.

3.2 NOOS Proxy Operation

The proxy will perform the following functions:

- Periodically scanning the configuration file for new entries.
- Polling news feeds at the frequency specified in the configuration file
- Caching and managing news items in the proxy database
- Notifying mobile clients via various out-of-band methods, when new news arrives that the user is interested in.

Proxy Implementation The proxy communicates with the client by placing files in an outbound OCMP Directory-API location. The action to perform is specified by adding one header line to the file. The header contains one of three commands:

- `CONFIG` - Indicates that the files is a new feed configuration file. The client should update its feed configuration list as necessary. The configuration file has a very simple format. The first line is the word `CONFIG`. The subsequent lines start are of the format `FEED <url> <polling rate> <keyword>.`
- `SEED <URL>` - Indicates that the file is a complete newsfeed file. The URL is a unique indicator for the feed.
- `FEED <URL>` - Indicates that the file is a diff against the previous feed file that the client received (or the aggregation of a `SEED` and a number of previous diffs). The URL is a unique indicator for the feed.

The feed polling component is implemented in python. The first time a feed is downloaded, the entire file is sent with a `SEED` instruction header via the outbound directory. If the difference option is not used, then all feed reports from the proxy to the client will be `SEED` messages, containing the entire text of the feed. If the difference option is enabled, then updates are sent to the client in a file that begins with `FEED`. This file is a unified diff file representing the differences between the latest feed file and the last feed sent to the client.

The difference-only option only makes sense in very controlled feed-retrieval environments, so that the differences do not get out of sync. The feed differences are in "unified difference" format[10], so if a difference does not apply correctly, the client can repair the situation by requesting and entire feed file.

3.3 Client Operation

The client-side proxy performs the following functions:

- Opportunistic download in the background.
- Listening for out-of-band messages to trigger data retrieval.
- Local storage of feed items to serve to applications.
- Legacy application compatibility

Client Implementation The client plugin for OCMP is a simple shell script that parses files that appear in the inbound client-side directory associated with the newsfeed plugin. The shell script reads the header of the file that appears, and parses it as described in the previous section. When new feed information is received, it is copied into a local repository, where it is held until a feed reader application requests it. If a `FEED` diff is received, the diff is applied to the feed that exists in the repository directory. If the diff fails for any reason, a `SYNC` command is sent to the proxy (via a file containing one line: `SYNC <URL>`), which triggers the proxy to re-send the entire feed in question (using a `SEED` file).

The actual fetching of the files is triggered by messages received by some out-of-band means. This could be an SMS message, an email, or a notification piggybacked on other application data, if such support is available.

4 Prototyping and Experiment

In this section, we present our analysis of the NOOS approach to newsfeed management, as compared to an approach in which the mobile client must manage the newsfeed itself[4]. We examine two metrics: bandwidth consumption, and power. There is a pretty strong correlation between bandwidth usage and power, however when we model power, we try to also take into account the overhead associated with failed connection attempts, out of band messages, and the need to wake up the interface from sleep.

We compare three different scenarios:

- No-proxy: The management of feeds is handled by the mobile device.
- Proxy: The NOOS system is used, but full feeds are sent with every update.
- Proxy-Diff: The NOOS system is used with the difference option enabled.

4.1 Experimental Method

To analyze the behavior of the NOOS system, we use two methods. First, the system is validated functionally on a short-term scale. During the functional validation, we record the amount of data sent and received at each endpoint in the cases above. We can use the data collected in this stage to more accurately model components of the simulation that we use in larger-scale simulation.

We performed the small-scale validation using the implementation based on OCMP as discussed in the previous section. During the validation, information was collected about the amount of data sent and received to the client, via both wireless and "out of band" interfaces, as feeds were managed. The data collected in this step was used to adjust parameters of the simulator.

Based on the data gathered from the short-term experiments, we run simulations that generate data based on expected long-term performance. In order to simulate long-term performance, a number of feeds were collected over the period of about 3 days (68 hours). We then wrote a simulator in Python that

[4] Supplementary information, including simulation/implementation code is available for viewing: http://www.jasonlebrun.info/Research/OCMPFeeds

simulates the fetching of the collected feeds at different polling rates, and using each of the different systems discussed above. The simulator allows us to test over many different parameters spaces quickly.

We model the wakeup and failure probabilities using a simple model. Since the feed connection attempts are spaced fairly far apart in time, and occur in the background we make the assumption that they are more or less independent of any other events. So, we can decide whether or not an interface was asleep or awake when a feed fetch attempt occured by simply randomly deciding that the interface with probability P_{sl} each time a fetch occurs. Similarly, we just randomly decide that the connection was available or not with P_c probability each time a fetch occurs.

Our simulator is implemented in Python, using the SimPy simulation framework. Each interface (net, wireless and out-of-band), as well as the proxy, the client, and the internet, are represented as a SimPy process. The process that drives the simulation is the Proxy process, since it does the periodic polling of the remote newsfeeds. Connections between the proxy, client, and internet are modeled using an interface process. These interface objects provide a simple model of connectivity, and the ability to monitor the amount of data sent and received, as well as the number of time the interface was forced to wake up, and the number of failed connection attempts. The situation in which there is no proxy is modeled by treating the proxy as a mobile device, and using wireless interface parameters on the network interface object. For the proxy based experiment, we assume that the land-line network is always on and always available.

As the simulation executes, each interface keeps information about the amount of data sent, the amount of data received, the number of time that the interface was woken up on behalf of the feed application, and the number of times that a failed connection attempt occurred on behalf of the feed application.

4.2 Results: Bandwidth

The first exploration examines the improvement in wireless interface usage gained by using a proxy, and a proxy that sends only feed differences. We fix the probability of connection at 1, and the probability of a sleeping interface at 0, and assume that the user is monitoring 10 feeds. The polling rate is varied with the values 10,30,60,180, and 360. Figure 3 shows how many connection attempts are made by the wireless client over the course of the experiment. The number of connection attempts influences the number of times that the interface will need to wake up, as well as the number of failed connection attempts. At any polling rate, the proxy-based method consistently reduces the number of times that the mobile client needs to attempt to wake up to receive data.

Figure 2 shows similar results, and also motivates the addition of the differencing feature. While there is a consistent improvement for the non-differencing proxy, the use of feed differences adds up to substantial savings in bandwidth, even at low polling rates. For example, at a polling rate of 30 minutes per check, the proxy method provides a 62% reduction in bandwidth compared to the client managing the feeds on its own. The proxy with feed differencing provides

a slightly better reduction of 89%. However, at a slower polling rate of 6 hours, the proxy method only provides a 12% reduction in bandwidth usage, while the proxy method with differencing provides a 59% reduction in bandwidth usage.

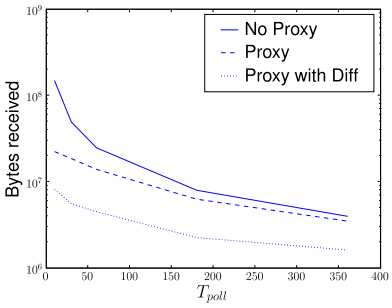

Fig. 2. Bandwidth vs. Polling Rate

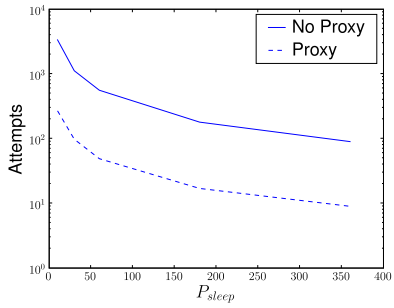

Fig. 3. Number of connection attempts

Next, in Figure 4 we look at how many out-of-band (SMS) messages are sent as the polling rate is increased. We consider the case where $T_{poll} = 60$, because a 1 hour feed polling rate is a common default setting in desktop aggregators. The out-of-band messages add a small amount of overhead to the proxy-based system. However, it is insignificant, compared to the savings gained. Since the out-of-band messages are typically sent through a different channel than the data itself, we do not compare this metric directly to the bandwidth metric.

Figure 5 shows scaling of bandwidth usage as the number of feeds increases. For these experiments, we set $T_{poll} = 30$ again, and and $P_{avail} = 1$ to avoid the affect of a failed interface. Although only 1-10 feeds were tested, we can see that using a proxy significantly improves the scaling of newsfeed management as the number of feeds increases. The bandwidth requirements of both systems grows linearly as the number of feeds increases, but when using the proxy system, the growth is only 1.9 megabytes per feed, as opposed to 5 megabytes per feed for the client-managed system. Using the proxy with the feed differencing feature adds only 0.5 megabytes per feed, an even bigger savings.

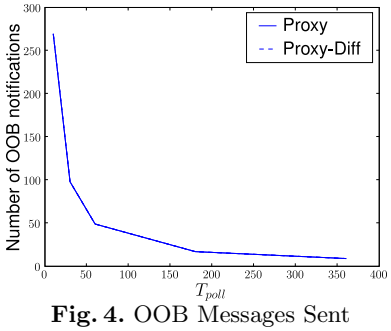

Fig. 4. OOB Messages Sent

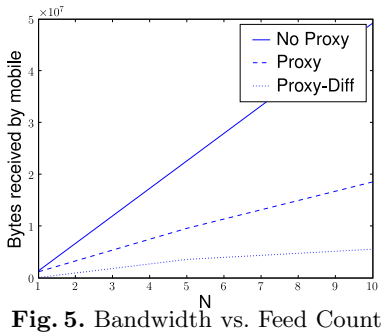

Fig. 5. Bandwidth vs. Feed Count

4.3 Results: Energy Consumption

There are a number of factors that contribution to the energy consumption of newsfeed management. These factors are shown together in table 1.

P_{wu}	The power needed to activate the interface if it was in a sleep mode.
P_{rx}	The power needed to receive a given number of bytes.
P_{tx}	The power needed to send a given number of bytes.
P_{fail}	The power used when the device tries to connect to a remote host, but fails.
P_{sms}	The power used by short-message services used for out-of-band communication.
p_{sleep}	Probability that the interface is asleep when it's time to fetch data.
p_{avail}	Probability of connection available when a connection attempt is made.
N_{rx}	Number of bytes received.
N_{tx}	Number of bytes sent.
N_{oob}	Number of out of band messages sent.

Table 1. Modeling Parameters

Using these parameters, we can calculate the energy required for a given system over the duration of the simulation using equation 1

$$E = P_{rx} * N_{rx} + P_{tx} * N_{tx} + p_{sleep} * P_{wu} + (1 - p_{avail}) * P_{fail} + P_{sms} * N_{oob} \quad (1)$$

Using this equation, we examine the energy usage of newsfeed management for the various systems, as a function of the polling rate, as well as of the number of feeds managed. The actual energy consumption of the various systems will vary depending on the hardware platform that is used. For our analysis, we base the energy results on the Atheros 5001x 802.11b chipset[5].

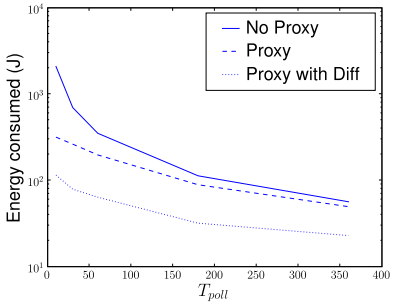

Fig. 6. Energy vs. Polling Rate

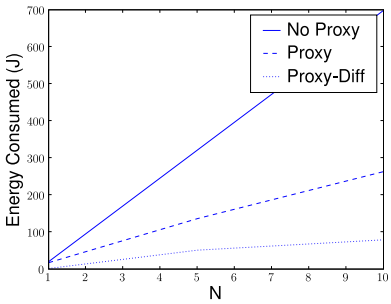

Fig. 7. Energy vs. Feed Count

Figures 6 and 7 show us the energy consumed by the feed management system over the course of the experiment, plotted against polling rate, and number of feeds monitored, respectively. The shape of the energy consumption curves looks similar to the bandwidth curves, because bandwidth is the dominating term of the energy calculation. Based on our energy consumption result, we see that users of the proxy-based system with feed differencing will be able to monitor more feeds, more often. For example, for the energy cost of monitoring 10 feeds every 3 hours with no proxy, we could monitor 10 feeds every 10 minutes with the proxy system with differencing.

5 Conclusion

Newsfeed publication technologies such as RSS provide a succinct and consistent way for content providers to summarize the information that they provide. Because of the ease of content providing and summarizing, users often discover a large number of feeds that they'd like to follow. As the number of feeds grows, it is difficult for a resource-constrained mobile device to manage the feeds efficiently.

In this paper, we present the design and evaluation of a newsfeed management system, NOOS, for mobile devices. We build the system on the OCMP framework which provides basic support for managing multiple interfaces and intermittent connectivity. As a result, the system allows a mobile client to fully utilize multiple network interfaces, and combat the problems of intermittent connectivity, to make information feeds available promptly to the user. Through simulation studies based on real-world data, we show that the NOOS system can help a user manage newsfeeds in a much more efficient way than a naive polling scheme.

The explorations in this system only scratch the surface of the flexibility provided by the NOOS system. A number of future directions exist for this system. The feed configuration file creates much potential for user-programmable feed management. In addition to the polling rate examined in this work, users might specify keywords to monitor or relative importance of feeds. Users might also prioritize feeds such that certain feeds are only sent when low-cost connections are available. Using a common proxy also presents an opportunity for optimization because the proxy can put all newsfeeds (RSS, Atom, etc.) into the same format, so that the mobile client's feed aggregator can be simpler. We plan on exploring a number of these possibilities in future work, as well as testing actual deployments of the system to verify its real-world performance.

References

1. RFC 4287: The Atom Syndication Format. http://www.ietf.org/rfc/rfc4287.txt.
2. T. Armstrong, O. Trescases, C. Amza, and E. de Lara. Efficient and transparent dynamic content updates for mobile clients. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, 2006.
3. Mozilla Thunderbird. http://www.mozilla.com/en-US/thunderbird.
4. A. Chaintreau, P. Hui, J. Crowcroft, C. Diot, R. Gass, and J. Scott. Impact of human mobility on the design of opportunistic forwarding algorithms. In *Proceedings of 25th IEEE Conference on Computer Communications*, 2006.
5. Atheros Communications. Power Consumption and Energy Efficient Comparisons of WLAN Products. 2003.
6. Google Reader. http://google.com/reader.
7. P. Hui, A. Chaintreau, J. Scott, R. Gass, J. Crowcroft, and C. Diot. Pocket switched networks and the consequences of human mobility in conference environments. In *Proceedings of the SIGCOMM 2005 Workshop on Delay Tolerant Networking*, 2005.
8. OCMP - Opportunistic Communications Management Protocol. http://blizzard.cs.uwaterloo.ca/tetherless/index.php/OCMP.
9. RSS 2.0 Specification. http://validator.w3.org/feed/docs/rss2.html.
10. GNU Diff Manual. http://www.gnu.org/software/diffutils/manual.