CONTENT SYNDICATION PLATFORM

BACKGROUND

RSS, which stands for ***R***eally ***S***imple ***S***yndication, is one type of web content syndication format. RSS web feeds have become more and more popular on the web and numerous software applications with RSS support are being developed. These numerous applications can have many varied features and can lead users to install several different RSS-enabled applications. Each RSS application will typically have its own list of subscriptions. When the list of subscriptions is small, it is fairly easy for a user to enter and manage those subscriptions across the different applications. As the list of subscriptions grows, however, management of the subscriptions in connection with each of these different RSS-enabled applications becomes very difficult. Thus, it is very easy for subscription lists to become unsynchronized.

In addition, web feeds come in several different file formats, with the popular ones being RSS 0.91, 0.92, 1.0, 2.0 and Atom. Each RSS-enabled application has to support most of these formats and possibly even more in the future. Implementing parsers for use in the RSS context for some applications is more difficult than for others. Given that not all application developers are RSS experts who possess experience and knowledge with regard to the intricacies of each format, it is unlikely that all application developers will implement the parsers correctly. Hence, it is likely given the rich number of file formats that some application developers will opt to not develop applications in this space or, if they do, the applications will not be configured to fully exploit all of the features that are available across the different file formats.

Another aspect of RSS and web feeds pertains to the publishing of content. For example, the number of users with blogs (weblogs) is increasing. There are many publicly available services that provide free blog services. Publishing content to a blog service, however, can be rather cumbersome since it might involve opening a browser, navigating to the blog service, signing in, and then typing the entry and submitting it.

Many application developers would prefer to be able to publish from within their particular application, without breaking the user flow by having to go to a website. In addition, there are many different types of protocols that can be used to communicate between a client device and a particular service. Given this, it is unlikely that application developers will implement all protocols. As such, the user experience will not be all that it could be.

SUMMARY

A content syndication platform, such as a web content syndication platform, manages, organizes and makes available for consumption content that is acquired from a source, such as the Internet, an intranet, a private network or other computing device, to name just a few. In some embodiments, the platform can acquire and organize web content, and make such content available for consumption by many different types of applications. These applications may or may not necessarily understand the particular syndication format. An application program interface (API) exposes an object model which allows applications and users to easily accomplish many different tasks such as creating, reading, updating, deleting feeds and the like.

In addition, the platform can abstract away a particular feed format to provide a common format which promotes the useability of feed data that comes into the platform. Further, the platform processes and manages enclosures that might be received via a web feed in a manner that can make the enclosures available for consumption to both syndication-aware applications and applications that are not syndication-aware.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a high level block diagram that illustrates a system that includes a web content syndication platform in accordance with one embodiment.

Fig. 2 is a block diagram illustrates aspects of an object model in accordance with one embodiment.

Fig. 3 is a block diagram that illustrates a feed synchronization engine in accordance with one embodiment.

Fig. 4 illustrates an exemplary feed store in accordance with one embodiment.

Fig. 5 illustrates an exemplary user's profile in accordance with one embodiment.

Fig. 6 illustrates exemplary objects in accordance with one embodiment.

Fig. 7 illustrates exemplary objects in accordance with one embodiment.

DETAILED DESCRIPTION

Overview

A content syndication platform, such as a web content syndication platform, is described which is utilized to manage, organize and make available for consumption content that is acquired from a source, such as the Internet, an intranet, a private network or other computing device, to name just a few. In the context of this document, the platform is described in the context of an RSS platform that is designed to be used in the context of RSS web feeds. It is to be appreciated and understood that the RSS context constitutes but one example and is not intended to limit application of the claimed subject matter to only RSS contexts. The description below assumes some familiarity on the part of the reader with RSS. For background on RSS, there are a number of publicly available specifications that provide information that may be of interest to the reader.

In this document, certain terminology will be used in the context of the RSS embodiment that is described. An *item* is a basic unit of a feed. Typically, an item represents a blog entry or a news article/abstract, with a link to the actual article on the website. An *enclosure* is similar to an email attachment, except that there is a link to actual content. A *feed* is a list of items in a resource, usually only the most recent

additions. A *system feed list* is a list of feeds to which a user is subscribed. A *subscription* refers to the act of signing up to receive notifications of new feed items.

In the various embodiments described in this document, the platform can acquire and organize web content, and make such content available for consumption by many different types of applications. These applications may or may not necessarily understand the particular syndication format. Thus, in the implementation example, applications that do not understand the RSS format can nonetheless, through the platform, acquire and consume content, such as enclosures, acquired by the platform through an RSS feed.

The platform comprises an application program interface (API) that exposes an object model which allows applications and users to easily accomplish many different tasks such as creating, reading, updating, deleting feeds and the like. For example, using the API, many different types of applications can access, manage and consume feedlists which includes a list of feeds.

In at least one embodiment, the platform provides multiple different feed parsers each of which can parse a particular format in which a web feed may be received. The parsed format is then converted into a *common format* which can then be leveraged by applications and users. The common format is utilized to abstract away specific notions embodied by any one particular format in favor of a more universal, understandable format.

Further, the platform processes and manages enclosures that might be received via a web feed in a manner that can make the enclosures available for consumption to both syndication-aware applications and applications that are not syndication-aware. In at least some embodiments, the APIs allow for discovery of the relationship between an enclosure and its associated feed item.

In the discussion that follows, an exemplary platform and its components are first described under the heading "Web Content Syndication Platform". Following this

discussion, an implementation example (under the heading "Implementation Example") is provided and describes a set of APIs that expose an object model that enables applications and users to interact with the platform in a meaningful and robust way.

Web Content Syndication Platform

Fig. 1 shows an exemplary system in accordance with one embodiment, generally at 100. Aspects of system 100 can be implemented in connection with any suitable hardware, software, firmware or combination thereof. In at least one embodiment, aspects of the system are implemented as computer-readable instructions that reside on some type of computer-readable medium.

In this example, system 100 comprises a content syndication platform 102 and a collection of applications 104 individual ones of which can be configured to utilize the platform in different ways, as will become apparent below. In at least some embodiments, the content syndication platform comprises a web content syndication platform. In the discussion that follows, the platform 102 is described in the context of an RSS platform. It is to be appreciated and understood that this is intended as but an example and is not intended to limit application of the claimed subject matter to only RSS environments. Rather, principles of the described embodiments can be utilized in other syndication environments without departing from the spirit and scope of the claimed subject matter.

In this example, platform 102 comprises an object model 106 that is exposed by a set of APIs that enable applications 104 to interact with the platform. A synchronization engine 108 is provided and is configured to, among other things, acquire web content and, in at least some embodiments, convert the web content into a so-called *common format*, which is described in more detail below.

A publishing engine 110 permits users to publish content, such as blogs, in a manner that abstracts away, via the APIs, the communication protocol that is utilized to

communicate between the user's application or computing device and the server or destination software that is to receive the content.

In addition, in at least one embodiment, platform 102 includes a feed store 112 that stores both feed lists 114 and feed data 116. Further, platform 102 utilizes, in at least one embodiment, file system 118 to store and maintain enclosures 120. Using the file system carries with it advantages among which include enabling applications that do not necessarily understand the syndication format to nonetheless consume enclosures that may be of interest. Further, platform 102 includes a post queue 122 that holds post data 124 that is to be posted to a particular web-accessible location.

As noted above, platform 102 can enable applications to access, consume and publish web content. Accordingly, the collection of applications 104 can include many different types of applications. In at least some embodiments, the types of applications can include those that are syndication-aware and those that are not syndication-aware. By "syndication-aware" is meant that the application is at least somewhat familiar with the syndication format that is utilized. Thus, in the RSS context, a syndication-aware application is one that may be configured to process data or otherwise interact with content that is represented in an RSS format. This can include having the ability to parse and meaningfully interact with RSS-formatted data. Similarly, an application that is not syndication-aware is typically not configured to understand the syndication format. Yet, through the platform, as will become apparent below, applications that are not syndication aware can still access and consume content that arrives at the platform in a syndication format.

Looking more specifically at the different types of applications that can interact with the platform, collection 104 includes a web browser application 122, an RSS reader application 124, a digital image library application 126, a media player application 128 and a blog service 130. In this example, RSS reader application 124 is a syndication-aware application, while media player 128 may not necessarily be a syndication-aware

application. Further, web browser application 122 may or may not be a syndication-aware application. Of course, these applications constitute but examples of the different types of applications that can interact with the platform. As such, other types of applications that are the same or different from those illustrated can be utilized without departing from the spirit and scope of the claimed subject matter. By way of example and not limitation, these other types of applications can include calendar applications for event feeds, social networking and email applications for contact feeds, screen saver applications for picture feeds, CRM for document feeds, and the like.

In the discussion that follows, aspects of the individual components of the platform 102 are described in more detail, each under its own heading.

Object Model

Fig. 2 illustrates individual objects of object model 106 in accordance with one embodiment. The object model about to be described constitutes but one example of an object model that can be utilized and is not intended to limit application of the claimed subject matter to only the object model that is described below. As noted above, the object model is exposed by an API, an example of which is described below.

In this particular object model, a top level object 200 called *feeds* is provided. The feeds object 200 has a property called *subscriptions* of the type folder. Subscription or folder objects 202 are modeled as a hierarchy of folders. Thus, in this particular example, subscription or folder objects have properties that include *subfolders* 204 of the type folder and *feeds* 206 of the type feed. Underneath the feeds object 206 is an item object 208 of the type item, and underneath the item object 206 is an enclosure object 210 of the type object.

The individual objects of the object model have properties, methods and, in some instances, events that can be utilized to manage web content that is received by the platform. The above-described object model permits a hierarchical structure to be

utilized to do such things as manage feedlists and the like. For example, using a folder structure, the platform can execute against a set of feeds. As will be appreciated by the skilled artisan, this makes it easier for the application developer. For example, executing against a set of feeds provides the ability to refresh all of the "news" feeds, located within the news folder.

As an example, consider the following. Assume that a user wishes to interact with or consume data associated with a feed to which they are not actually subscribed. For feeds that are subscribed to, i.e. those that are represented inside the root level subscription folder, the synchronization engine 108 (Fig. 1) will pick up the feed and start to, on an appropriate interval, fetch data associated with the feed. There are cases, however, when an application that uses the platform does not wish to be subscribed to a particular feed. Rather, the application just wants to use the functionality of the platform to access data from a feed. In this case, in this particular embodiment, subscriptions object 202 supports a method that allows a feed to be downloaded without subscribing to the feed. In this particular example, the application calls the method and provides it with a URL associated with the feed. The platform then utilizes the URL to fetch the data of interest to the application. In this manner, the application can acquire data associated with a feed in an adhoc fashion without ever having to subscribe to the feed.

Considering the object model further, consider item and enclosure objects 208, 210 respectively. Here, these objects very much reflect how RSS is structured itself. That is, each RSS feed has individual items inside of which can optionally appear an enclosure. Thus, the structure of the object model is configured to reflect the structure of the syndication format.

From an object model perspective, there are basically two different types of methods and properties on an item. A first type of method/property pertains to data

which is read only, and a second type of method/property pertains to data which can be both read and written.

As an example of the first type of method property, consider the following. Each feed can have data associated with it that is represented in an XML structure. This data includes such things as the title, author, language and the like. Data such as this is treated by the object model as read only. For example, the data that is received by a feed and associated with individual items is typically treated as read only. This prevents applications from manipulating this data. Using an XML structure to represent the feed data also carries with it advantages as follows. Assume that the synchronization engine does not understand a new XML element that has been added. Nonetheless, the synchronization engine can still store the element and its associated data as part of the feed item data. For those applications that do understand the element, this element and its associated data are still available for the application to discover and consume.

On the other hand, there is data that is treated as read/write data, such as the name of a particular feed. That is, the user may wish to personalize a particular feed for their particular user interface. In this case, the object model has properties that are read/write. For example, a user may wish to change the name of a feed from "New York Times" to "NYT". In this situation, the name property may be readable and writable.

Feed Synchronization Engine

In the illustrated and described embodiment, feed synchronization engine 108 (Fig. 1) is responsible for downloading RSS feeds from a source. A source can comprise any suitable source for a feed, such as a web site, a feed publishing site and the like. In at least one embodiment, any suitable valid URL or resource identifier can comprise the source of a feed. The synchronization engine receives feeds and processes the various feed formats, takes care of scheduling, handles content and enclosure downloads, as well as organizes archiving activities.

Fig. 3 shows an exemplary feed synchronization engine 108 in a little more detail in accordance with one embodiment. In this embodiment, synchronization engine includes a feed format module 300, a feed schedule module 302, a feed content download module 304, an enclosure download module 306 and an archiving module 308. It is to be appreciated and understood that these module are shown as logically separate modules for purposes of clearly describing their particular functionalities. The logically separate modules are not intended to limit the claimed subject matter to only the particular structures or architectures described herein.

Feed Format Module -- 300

In the illustrated and described embodiment, feeds are capable of being received in a number of different feed formats. By way of example and not limitation, these feed formats can include RSS 1.0, 1.1, .9x, 2.0, Atom .3, and so on. The synchronization engine, via the feed format module, receives these feeds in the various formats, parses the format and transforms the format into a normalized format referred to as the *common format*. The common format is essentially a superset of all supported formats. One of the benefits of using a common format is that applications that are format-aware now need to only be aware of one format—the common format. In addition, managing content that has been converted into the common format is much easier as the platform need only be concerned with one format, rather than several. Further, as additional syndication formats are developed in the future, the feed format module can be adapted to handle the format, while at the same time permit applications that are completely unaware of the new format to nonetheless leverage and use content that arrives at the platform via the new format.

With regard to the common format, consider the following. From a format standpoint, the common format is represented by an XML schema that is common between the different formats. In a different format, certain elements may have different

names, different locations within the hierarchy of the XML format and the like. Accordingly, the common format is directed to presenting a common structure and syntax that is derived collectively from all of the different formats that are possible. Thus, in some instances, elements from one format may be mapped into elements of the common format.

Feed Schedule Module -- 302

Each feed can have its own schedule of when the synchronization engine 108 should check to ascertain whether there is new content available. Accordingly, the synchronization engine, through the feed schedule module 302, manages such schedules to respect a site's as well as a user's or a system's requirements and limitations.

As an example, consider the following. When a feed is first downloaded, an update schedule (i.e. a schedule of when the feed is updated) may be included in the feed's header. In this case, the feed schedule module 302 maintains the update schedule for this particular feed and checks for new content in accordance with the update schedule. If, however, no schedule information is included, then the feed schedule module can utilize a *default schedule* to check for new content. Any suitable default schedule can be used such as, for example, re-downloading the feed content every 24 hours. In at least some embodiments, the user may specify a different default work schedule.

In addition, in at least some embodiments, the feed schedule module can support what is referred to as a *minimum schedule. The minimum schedule* refers to a minimum update time that defines a period of time between updates. That is, the platform will not update a feed more often than what the minimum schedule defines. In at least some embodiments, the user can change the minimum time. In addition, the user can also initiate a manual refresh of any, or all feeds.

In addition to supporting default and minimum schedules, in at least some embodiments, the feed schedule module can support *publisher-specified schedules*. As the name implies, a publisher-specified schedule is a schedule that is specified by a particular publisher. For example, the publisher-specified schedule can typically specify how many minutes until the client should next update the feed. This can be specified using the RSS 0.9x/2.0 "ttl" element. The synchronization engine should not fetch a new copy of the feed until at least that number of minutes has passed. The publisher-specified schedule can also be specified at different levels of granularity such as hourly, daily, weekly, etc.

It should be noted that each copy of a feed document can have a different publisher-specified schedule. For example, during the day, the publisher may provide a schedule of 15 minutes, and then during the night, the publisher may provide a schedule of 1 hour. In this case, the synchronization engine updates its behavior every time the feed is downloaded.

In addition, in at least some embodiments, the synchronization engine, via the feed schedule module 302, supports the notion of skipping hours and/or days. Specifically, RSS 0.9 and 2.0 enable a server to block out certain days and hours during which the client should not conduct an update. In this case, the synchronization engine respects these settings, if provided by the server, and does not update the feed during those times.

In addition to the default, minimum and publisher-specified schedules, in at least some embodiments, the synchronization engine supports the notion of user-specified schedules and manual updates. More specifically, on a per-feed basis, the user can specify a schedule of their choice. From a platform perspective, the user-specified schedule can be as complex as specified by a server. In this instance, the platform, via the feed schedule module, maintains the most recent schedule extracted from the feed as well as the user schedule. In at least some embodiments, the user schedule always

overrides the publisher's schedule. In addition, at any time, an application can initiate a forced update of all feeds or individual feeds.

With regard to bandwidth and server considerations, consider the following. In accordance with one embodiment, the synchronization engine can be designed in view of two related issues. First, the synchronization should be considerate of the user's bandwidth and CPU. Second, because of widespread use of the RSS platform, the synchronization engine should be considerate of its impact on servers. These two issues have an impact on both *when* and *how* feeds are downloaded.

From the perspective of *when* a feed is downloaded, synchronization engine can be designed with the following considerations in mind. In the absence of a schedule from the server, and any other instructions from the user, the synchronization engine should be very conservative in how often it updates. Hence, in at least some embodiments, the default schedule is set to 24 hours. Further, to protect the user's resources from being adversely impacted by an inefficient server, a minimum schedule can be enforced to keep the synchronization engine from updating too often, even if the server specifies otherwise. In addition, updates at login time (and at common intervals, e.g. each hour from the startup time) should be carefully managed. Feed updates should be delayed until a specified period of time after user login has completed, and should be staggered slightly to avoid large update hits each hour, on the hour. This can be balanced against a user's desire to have all of the updates happen at once. Further, when a server uses the skip hours or skip days feature described above, the client should not immediately fetch an update as soon as the moratorium period is over. Instead, the client should wait a random interval ranging up to 15 minutes before fetching the content.

To assist the synchronization engine in this regard, the feed schedule module 302 can maintain a state for each feed, such as fresh or stale. A "fresh" state means that, based on the publisher schedule, the feed is fresh. A "stale" state means that the publisher's schedule has indicated an update, but the synchronization engine has not yet

completed the update. Clients with an interest in the freshest content can request an immediate update, and be notified when it is available. If this expectation is set, then the synchronization engine can implement arbitrary delays in updating the content, rather than rigorously following the schedule to the detriment of the user and the server.

With regard to *how* a feed is downloaded, consider the following. In one embodiment, the synchronization engine can use a task scheduler to launch a synchronization engine process at a pre-defined time. After the synchronization engine has completed, it updates a task schedule with the next time it should launch the synchronization engine again (i.e., NextSyncEngineLaunchTime).

When the synchronization engine launches, it queues up all "pending" feeds whose NextUpdateTime is less or equal to the currentTime and then processes them as follows. For each feed, the following properties are tracked: LastUpdateTime, NextUpdateTime, Interval (specified in minutes) and LastErrorInterval.

At the end of successfully synching a feed, the feed's LastUpdateTime is set to the current time and NextUpdateTime is set to LastUpdateTime plus an interval plus randomness (1/10th of the interval). Specifically:

```
LastUpdateTime = currentTime
NextUpdateTime = currentTime + Interval + Random(Interval * 0.1)
ErrorInterval = 0
```

Random(argument) is defined to be a positive value between 0 and its argument. For example Random(10) returns a float between 0..10.

If synching of a feed failed for one of the following reasons:

```
HTTP 4xx response code;
HTTP 5xx response code;
Winsock/network error; or
HTTP 200, but response body has a parsing error (not a recognized feed format)
```

then an exponential back off algorithm is applied as follows:

```
LastUpdateTime = <unchanged>
ErrorInterval = min( max(ErrorInterval * 2 , 1min), Interval)
NextUpdateTime = currentTime + ErrorInterval + Random(ErrorInterval *
0.1)
```

After synchronization of all "pending" feeds has completed, the synchronization engine determines if there are any feeds whose NextUpdateTime has passed (NextUpdateTime <= currentTime). If there are, then those "pending" feeds are queued and processed as if the synchronization engine just launched.

If there are no outstanding "pending" feeds, then the synchronization engine determines if there are any "soon-to-sync" feeds whose NextUpdateTime is within two minutes of the current time (currentTime + 2min >= NextUpdateTime). If there are any "soon-to-sync" feeds then the synchronization engine process continues to run, and it sets a timer to "wake up" at NextUpdateTime and process "pending" feeds.

If there are no "soon-to-sync" feeds then the NextSyncEngineLaunch is set to the NextUpdateTime of the feed with the soonest NextUpdateTime. Then the task scheduler is set to NextSyncEngineLaunchTime and the synchronization engine process ends.

In accordance with one embodiment, if there are several "pending" feeds in the queue, the synchronization engine can synchronize multiple feeds in parallel. However, the number of parallel synchronizations should be limited, as well as how many synchronizations are performed in a certain time period in order to not saturate network bandwidth and processor utilization. In accordance with one embodiment, feed synchronization shaping is provided via a token-bucket. Conceptually, the token bucket works as follows.

- A token is added to the bucket every 1/r seconds;

- The bucket can hold at most *b* tokens; if a token arrives when the bucket is full, it is discarded;
- When a feed needs to be synchronized, a token is removed from the bucket and the feed is synchronized;
- If no tokens are available, the feed stays in the queue and waits until a token becomes available.

This approach allows for bursts of feed synchronizations of up to *b* feeds. Over the long run, however, the synchronizations are limited to a constant rate *r*. In an implementation example, the synchronization engine uses the following values for b and r: b = 4 and r = 2.

Feed Content Download Module -- 304

In accordance with one embodiment, feed content download module 304 handles the process of downloading a feed and merging the new feed items with the existing feed data.

As an example of how one can implement a feed content download module, consider the following. At the appropriate time, the synchronization engine, via the feed content download module, connects to a server and downloads the appropriate content.

In accordance with one embodiment, the platform is configured to support different protocols for downloading content. For example, the synchronization engine can support downloading the feed document over HTTP. In addition, the synchronization engine can support encrypted HTTP URLs (e.g., SSL, https and the like). Likewise, the synchronization engine can also support compression using the HTTP gzip support, as well as support feed downloads from Universal Naming Convention (UNC) shares.

In addition, the synchronization engine via the feed content download module can support various types of authentication. For example, the synchronization engine

can store a username/password for each feed, and can use this username/password for HTTP Basic authentication to retrieve the feed document.

With regard to updating a feed, consider the following. To determine if a feed has new content, the synchronization engine keeps the following pieces of information, for each feed:

- The last time the feed was updated as reported by the Last-modified header on the HTTP response;
- The value of the Etag header in the last HTTP response; and
- The most recent pubDate value for the feed (i.e. the feed-level publication date and time).

If the site supports Etag or Last-modified, then the synchronization engine can use these to check if there is new content. The site can respond with an HTTP response code 304 to indicate that there is no new content. Otherwise, the content is downloaded. For example, if the site supports RFC 3229-for-feeds, the site can return only the new content, based on the Etag passed by the client. Either way, the client then merges the new content with the stored content.

As a more detailed description of how feed content can be downloaded in but one implementation example, consider the following. To determine if a particular site has changed, the synchronization engine will submit a request with:

- The *If-None-Match* header, if the client has a saved Etag;
  - The header A-IM with the values: feed, gzip (used for RFC 3229-for-feeds);
- The *If-Modified-Since* header, if the client has a saved Last-modified value.

If the server responds with an HTTP Response code 304, then the content has not changed and the process may end here. If the server responds with content (i.e. HTTP codes 200 or 206), then the downloaded content is merged with the local content (note:

code 206 means that the server supports RFC3229-for-feeds, and the content downloaded is only the new content).

If there is content available and if the synchronization engine has a pubDate stored, and the downloaded feed document contains a channel-level pubDate element, the two dates are compared. If the local pubDate is the same as the downloaded pubDate, then the content has not been updated. The downloaded feed document can then be discarded.

If the synchronization engine processes each item one at a time, each item's pubDate is compared against the pubDate that the synchronization engine has stored (if any) and older items are discarded. Each item is then compared against the items in the store. The comparison should use the guid element, if present, or the link element, if guid is not present. If a match is found, then the content of the new item replaces that of the old item (if both have a pubDate, then it is used to determine which is newer, otherwise, the most recently downloaded is new). If no match is found, then the new item is pre-pended to the stored feed content (maintaining a "most recent at the top" semantic). If any item is added or updated in the local feed, the feed is considered updated, and clients of the RSS platform are notified.

For error cases, consider the following. If the server responds with a code 500 or most 400 errors, the synchronization schedule is reset and the server tries again later. The HTTP error 410, however, should be treated as an indication to reset the update schedule to "no more updates."

HTTP-level redirects should be followed, but no changes should be made to the client configuration (there are several pathological scenarios where redirects are given accidentally).

If the server responds with an XML redirect, then the feed should be redirected, and the stored URL to the feed should be automatically updated. This is the only case where the client updates the feed URL automatically.

With regard to downloading the feed, the download should not interrupt ordinary usage of the machine (e.g., bandwidth or CPU) when the user is engaged in other tasks. In addition, the user should be able to get the content as fast as possible when in an interactive application that relies on the content.

Enclosure Download Module -- 306

In accordance with one embodiment, enclosure download module 306 is responsible for downloading enclosure files for a feed and applying the appropriate security zone. At the time of downloading the feed content, the enclosures are downloaded as well.

Downloading enclosures can be handled in a couple of different ways. First, a *basic enclosure* is considered to be an RSS 2.0-style enclosure. For basic enclosures, the synchronization engine, via the enclosure download module 306, will automatically parse the downloaded feeds for enclosure links. The synchronization engine is configured to support multiple basic enclosures. Using the enclosure link, the enclosure download module can then download the enclosure. In at least some embodiments, for any new feed, the default action is not to download basic enclosures. Using the API which exposes the above-described object model, client can do such things as change the behavior on a per-feed basis to, for example, always download enclosures or force the download of a specific enclosure of a specific item in a specific feed.

Enhanced enclosure handling can be provided through the use of the common format described above. Specifically, in at least one embodiment, the common format defines additional functionality for enclosures. Specifically, the common format enables multiple representations of a particular piece of content. This includes, for example, including standard definitions of preview content and default content, as well as the ability to indicate whether an enclosure should be downloaded or streamed. In addition, the common format permits arbitrary metadata on an enclosure, and on representations

of the content. For any new feed, the default action is to download the "preview" version of any enclosure, subject to a default size limit of, for example, 10k per item.

Using the API, clients can do such things as change the behavior on the per-feed basis. For example, the behavior can be changed to always download the "default" version of the items in a feed or to always download any specific version that has a metadata element of a particular value. This can be done, for example, with a client callback which provides the "download this?" logic for each enclosure. In addition, using the API, clients can force immediate download of any specific representation of any specific enclosure of any specific item (or all items) in a specific feed.

With regard to providing security in the enclosure download process, consider the following.

In accordance with one embodiment, downloaded enclosures use the Windows XP SP2 Attachment Execution Service (SP2 AES) functionality. This functionality can provide file-type and zone based security. For example, provided with a file name and zone information (i.e. where an enclosure came from), AES can indicate whether to block, allow or prompt.

With regard to zone persistence, when saving a file, AES can persist the zone information so that, when it is subsequently opened, the user can be prompted.

The table just below describes AES risk-level/zone to action mapping:

Risk Levels	**Restricted**	**Internet**	**Intranet**	**Local**	**Trusted**
Dangerous, e.g. EXE	Block	Prompt	Allow	Allow	Allow
Moderate/Unknown, e.g. DOC or FOO	Prompt	Prompt	Allow	Allow	Allow
Low, e.g. TXT or JPG	Allow	Allow	Allow	Allow	Allow

In the illustrated and described embodiment, the synchronization engine will call a method, for example ::CheckPolicy, for each enclosure that it downloads. Based on the response, the synchronization engine can do one of the following:

- Block: Don't save (mark it as failed in the feed file);
- Allow: Save the enclosure
- Prompt: Save, but persist zone information. This means that if the user double-clicks on the file, they'll get a "Run/Don't Run" prompt.

In accordance with one embodiment, the synchronization engine will first save an enclosure to disk and will not download the enclosure in memory. Saving to disk triggers filter-based antivirus applications and gives these applications an opportunity to quarantine the enclosure if they choose.

Archiving Module -- 308

In accordance with one embodiment, archiving module 308 is responsible for dealing with old feed data. By default, a feed will hold a maximum of 200 items. When a feed exceeds the specified maximum, the older feed items are deleted by the archiving module. The associated enclosures are not, however, deleted.

Feed Store

In accordance with one embodiment, feed store 112 (Fig. 1) holds two types of information – a feed list 114 and feed data 116. As an example, consider Fig. 4. There, feed list 114 is embodied as a hierarchical tree structure 400 of the list of feeds. The feed data 116 comprises the data associated with a particular feed. In this example, the feed data 116 is arranged on a per-feed basis to include a collection 402 of items and enclosures.

There are many different ways that one might implement a feed store. In this particular embodiment, the feed store comprises part of the file system. One reason for this pertains to simplicity. That is, in this embodiment, the feed list is represented simply as a regular directory under which there can be sub-directories and files. The hierarchy is reflected as a normal file system hierarchy. Thus, each folder such as "News" and "Blogs" is essentially a regular directory in the file system with subdirectories and files.

In this particular example, there is a special file type that represents a feed subscription. By way of example only, consider that this type of file has the following format: "xyz.stg". The .stg file stores all of the data for a feed. Thus, you have a feed list, such as the list embodied in tree structure 400, and inside each feed (or file) is the feed data.

In the illustrated and described embodiment, the .stg files are implemented using *structured storage* technology. Structure storage techniques are known and will be appreciated by the skilled artisan. As brief background, however, consider the following.

Structured storage provides file and data persistence in COM by handling a single file as a structured collection of objects known as storages and streams. The purpose of structured storage is to reduce the performance penalties and overhead associated with storing separate object parts in different files. Structured storage provides a solution by defining how to handle a single file entity as a structured collection of two types of objects—storages and streams—through a standard implementation called *compound files*. This enables the user to interact with, and manage, a compound file as if it were a single file rather than a nested hierarchy of separate objects. The storage objects and stream objects function as a file system within a file, as will be appreciated by the skilled artisan. Structured storage solves performance problems by eliminating the need to totally rewrite a file to storage whenever a new object is added to a compound file, or

an existing object increases in size. The new data is written to the next available location in permanent storage, and the storage object updates the table of pointers it maintains to track the locations of its storage objects and stream objects.

Thus, in the illustrated and described embodiment, the .stg files are implemented using structured storage techniques and an API on top of the feed store allows access to the different streams and storages. In this particular example, each RSS item is written into one stream. Additionally, a header stream contains information associated with a particular feed such as the title, subscription, feed URL and the like. Further, another stream stores index-type metadata that allows quick and efficient access to contents in the file for purposes that include quickly marking something as read/unread, deleting an item and the like.

File System – Enclosures

In the illustrated and described embodiment, enclosures are not stored in structured storage or as part of the feed data, as indicated in Fig. 1. Rather, enclosures are recognized as being items, such as a picture or pictures, that other applications and the user may want to access and manipulate.

Thus, in the illustrated and described embodiment, enclosures are written into a user's particular profile. A link, however, is maintained between the enclosure and the associated feed item.

As an example, consider Fig. 5. Once a user starts subscribing to a feed, the feed content is stored locally under the user's profile, either in Application Data or in a Knownfolder "feeds".

The feedlist and feeds are stored in Application Data to better be able to control the format of the feedlist and the feeds. APIs are exposed (as will be described below) such that applications can access and manage the feeds.

The feedlist is the set of feeds that the user is subscribed to. In this example, the file that comprises the Feedlist is located at:

C:\Users\<Username>\AppData\Roaming\Microsoft\RSS\

The file contains the feed's properties, as well as items and enclosure properties (a URL to the file that is associated to the item). For example, the file for feed "NYT" is located at:

C:\Users\<Username>\AppData\Roaming\Microsoft\RSS\NYT.stg

In this example, the enclosures are grouped by feed and stored in the Knownfolder "feeds". This enables the user and other applications to easily access and use downloaded files.

For example, a user subscribes to the NPR feed and wants to make sure that their media player application can automatically add those files. Making this a Knownfolder enables the user to browse to it from the media player and set it as a monitored folder. Enclosures have the appropriate metadata of the feed and post such that applications can access the associated post and feed. Enclosures are located as follows:

C:\Users\<Username>\Feeds\<Feedname>\

Each enclosure that is written to the user's hard disk will have a secondary stream (e.g., a NTFS stream) which contains metadata about this enclosure. The metadata can include by way of example and not limitation, the feed that enclosure is from, author, link to feed item, description, title, publish date, and download date as well as other meta data as appropriate.

Publishing Engine/Post Queue

Many times when one writes a regular blog post, essentially what is being written is an RSS item. This RSS item is typically sent to some type of server, and this server maintains account information, the location of the blog, and the like. In this context, publishing engine 110 (Fig. 1) is configured to enable an application to make a posting or publish content, while at the same time abstract from the application the communication protocol that is utilized to communicate with the server. Hence, the application need only provide the data or content that is to be posted, and the publishing engine will handle the remaining task of formatting and communicating the content to the appropriate server.

As there can be several different protocols that are used, abstracting the protocols away from the applications provides a great deal of flexibility insofar as enabling many different types of applications to leverage the publishing functionality. In the illustrated and described embodiment, the publishing engine's functionality is implemented as an API that allows an application to post a blog without having to be knowledgable of the protocol used to communicate with the server.

Hence, in this example, the API has a method to create a new post which, when called, creates an RSSItem object. This RSSItem object has a post method which, when called, stores the content—in this case a blog—in a temporary store, i.e. post queue 122 (Fig. 1). The content is stored in a temporary store because the user may not be on line at the time the blog is created. Then, when the user makes an on line connection, publishing engine 110 makes a connection to the appropriate server and uses the server-appropriate protocol to upload the blog to the server.

Implementation Example

In the description that follows, an exemplary set of APIs is described to provide but one example of how one might implement and structure APIs to implement the above-described functionality. It is to be appreciated and understood that other APIs can be utilized without departing from the spirit and scope of the claimed subject matter. The described APIs are typically embodied as computer-readable instructions and data that reside on some type of computer-readable medium.

The APIs that are described below can be used to manipulate the set of feeds that a user is subscribed to (System Feed List) and the properties on the feeds. In addition, feed data APIs (i.e., item and enclosures) provide access to feeds that are stored in the feed store, as well as ad-hoc download of feeds. Using the Feed APIs, applications such as web browsers, media players, digital image library applications and the like can then expose the feed data within their experience.

In the example about to be described, the APIs are implemented as COM dual interfaces which also makes the APIs useable from scripting languages, managed code as well as native Win32 (C++) code.

Fig. 6 illustrates a top level object or interface IFeeds and an IFeedFolder object or interface together with their associated properties, methods and events in accordance with one embodiment.

In this example, IFeeds has one property—subscriptions which is an IFeedFolder. This is a root folder for all subscriptions. There are a number of methods on the root object such as DeleteFeed(), DeleteFeedByGuid(), DeleteFolder() and the like.

Of interest in this example is the GetFeedByGuid() method. This method can be called by applications to access a particular feed by, for example, the feed's GUID. Thus, the application need not be knowledgeable of the hierarchical ordering of the feeds. Rather, the feed's GUID can be used by the application to enable the platform to fetch the feed.

In addition, the ExistFeed() method checks for the existence of a feed by name, and the ExistFeedByGuid() check for a feed's existence by GUID. The GetFeed() method gets a feed by name or by GUID. The IsSubscribed() method enables an application or caller to ascertain whether a particular feed has been subscribed to.

In addition, the IFeeds object also has a SubscriptionsNotifications event which allows for registration for notifications for changes on the system feed list.

As noted above, Subscriptions are of the type IFeedFolder. The IFeedFolder object or interface essentially provides a directory and has similar kinds of properties such as the Name, Parent, Path and the like. In addition, the IFeedFolder object has a Feeds property of the type IFeed and a Subfolders property of the type IFeedFolder. The Subfolders property pertains to a collection of the folders underneath the instant folder (e.g., this is where the hierarchical structure derives) and Feeds property pertains to the actual feeds in a particular folder. In addition, the IFeedFolder has a LastWriteTime property which indicates the last time that anything was written to inside the folder. This property is useful for applications that may not have been running for a while, but yet need to look at the feed platform and ascertain its state so that it can synchronize if necessary.

There are a number of methods on the IFeedFolder, at some of which pertain to creating a feed (which creates a feed that the system does not have and adds it to a particular folder), creating a subfolder, deleting a folder or subfolder and the like.

Fig. 7 illustrates additional objects and their associated methods in accordance with one embodiment. Specifically illustrated are the IFeed, Item and IEnclosure objects.

Starting first with the IFeed object, consider the following. Many of the properties associated with this object come from the RSS feed itself, e.g, Title, Url, Webmaster, SkipHours, SkipDays, ManagingEditor, Homepage, ImageURL and the like, as will be appreciated by the skilled artisan. In addition, there is another set of

properties of interest, i.e. the *Items property* which is a collection that has all of the items that are part of a feed and the *LocalEnclosurePath* property which provides the actual directory to which all of the enclosures are written. Thus, for an application, the latter property makes it very easy for an application to access the enclosures.

In addition, this object supports a small set of methods such as Delete() and Download() which are used to manage particular feeds. Further, this object supports a method XML(), which returns a feed's XML in the *common format*. The XML data can be used for such things as creating a newpaper view of a feed. Clone() returns a copy of the feed that is not subscribed to.

Moving to the Item object, this object has a set of properties that represent regular RSS elements, e.g. Description, Url, Title, Author and the like. In addition, there is a Parent property that points back to the associated actual feed, and an Id property so that an application can manipulate the Id versus having to iterate over all items. In addition, there is an Enclosures property which is the collection of the item's enclosures of the type IEnclosure. Further, an IsRead property enables an application to indicate whether a particular item has been read.

Moving to the Enclosure object, consider the following. This object has properties that include a Type property (e.g. mp3) and Length property that describes the length of a particular enclosure. There is also the LocalAbsolutePath to a particular enclosure. The Download() method allows individual enclosures to be downloaded and used by applications.

Conclusion

The web content syndication platform described above can be utilized to manage, organize and make available for consumption content that is acquired from the Internet. The platform can acquire and organize web content, and make such content available for consumption by many different types of applications. These applications may or may

not necessarily understand the particular syndication format. An application program interface (API) exposes an object model which allows applications and users to easily accomplish many different tasks such as creating, reading, updating, deleting feeds and the like. In addition, the platform can abstract away a particular feed format to provide a common format which promotes the useability of feed data that comes into the platform. Further, the platform processes and manages enclosures that might be received via a web feed in a manner that can make the enclosures available for consumption to both syndication-aware applications and applications that are not syndication-aware.

Although the invention has been described in language specific to structural features and/or methodological steps, it is to be understood that the invention defined in the appended claims is not necessarily limited to the specific features or steps described. Rather, the specific features and steps are disclosed as preferred forms of implementing the claimed invention.

CLAIMS

1. A system comprising:

one or more computer-readable media;

computer-readable instructions on the media which implement a content syndication platform comprising:

a feed synchronization engine configured to acquire content from a source and make the content available to applications that are both syndication-aware and applications that are not syndication-aware; and

a feed store configured to store feed lists and feed data.

2. The system of claim 1, wherein the feed synchronization engine is configured to convert content in different formats into a common format.

3. The system of claim 1, wherein the feed synchronization engine is configured to support multiple different types of schedules, at least one of which comprises a minimum schedule that defines a minimum update time that defines a period of time between updates.

4. The system of claim 1, wherein the feed synchronization engine is configured to download feed data and update previously downloaded feed data.

5. The system of claim 1, wherein the feed synchronization engine is configured to download enclosures and provide such enclosures into a file system, wherein the enclosures can be accessed by applications that are not syndication-aware.

6. The system of claim 5, wherein the platform is configured to write enclosures into a user's profile.

7. The system of claim 6, wherein the platform is configured to maintain a link between an enclosure and an associated feed item.

8. The system of claim 1, wherein the feed store is implemented as part of a file system.

9. The system of claim 8, wherein the feed lists are represented as directories that can have sub-directories.

10. The system of claim 8, wherein the feed lists and feed data are managed in the file system using structured storage techniques.

11. The system of claim 1, wherein the feed synchronization engine is configured to enable a user to publish content in a manner that abstracts away the communication protocol between the user's application and a publishing location.

12. The system of claim 1, wherein the syndication platform is configured to conduct feed synchronization.

13. The system of claim 1, wherein the content syndication platform comprises an RSS platform.

14. A system comprising:

one or more computer-readable media embodying a set of application program interfaces comprising:

a first interface that serves as a root folder for subscriptions and which has one or more methods that pertain to feeds;

a second interface that serves as a folder for feeds, wherein the second interface has a feed property and a subfolder property and one or more methods that pertain to folders;

a third interface that includes properties associated with an individual feed and one or more methods that pertain to individual feeds;

a fourth interface that includes properties associated with individual items and at least one method that pertains to individual items;

a fifth interface that includes properties associated with individual enclosures and one or more methods associated with individual enclosures at least one method of which permits an enclosure to be downloaded by an application.

15. The system of claim 14, wherein the first interface has a method for downloading a feed without requiring a subscription to the feed.

16. The system of claim 14, wherein the first interface includes a notification event that allows an application to register for notifications that pertain to a system feed list.

17. The system of claim 14, wherein the second interface has a property that indicates the last time data was written to an associated folder.

18. The system of claim 14, wherein the third interface includes an item property which is a collection of items associated with a particular feed and a local enclosure path property that provides a directory to which individual enclosures are written.

19. The system of claim 14, wherein the third interface includes a method that returns a feed's XML to a caller.

20. The system of claim 14, wherein the fourth interface includes a parent property that points to an associated feed and an enclosure property associated with an item's enclosures.