Engineering Push-Based Web Services

Lars Brenna
Dept. of Computer Science
University of Tromsø, Norway
larsb@cs.uit.no

Dag Johansen
Dept. of Computer Science
University of Tromsø, Norway
dag@cs.uit.no

Abstract

Much of the content of popular Internet information sources is highly dynamic: urgent in nature and sometimes relevant only for a short time. The typical approach to querying such dynamic sources is polling for updates often.[1] This strains the traditional pull-based Internet and wastes network resources on transmitting redundant information.

This paper focuses on how to structure the Internet to avoid the unnecessary client-server interactions dominating the Internet. To that end, we extend the API of popular existing Internet services trough Web service wrappers. These wrappers use the API of, for instance, Google, but provide functionality that is richer. Initial experience shows that major performance gains can be achieved through this approach.

1 Introduction

The existing Internet has serious scaling limitations. The main reason for this is related to the widely deployed client-server model where users pull data down from remote web servers. Typically, a user issues a request to some remote web server through a browser. Next, he waits for a response, before he can parse and validate the received data. This is a simple, but adequate model for pulling down a few static HTML pages. Nevertheless, Internet data are now much more dynamic, and it might have relevance for just a short time window upon publication. A user can not know in advance when an important remote data item is changing. One obvious example is, for instance, a stock value that exceeds a certain threshold.

To alleviate this problem, a more frequent pulling scheme can be applied. However, RSS feeds demonstrate how this poses a scaling problem due to more traffic. RSS suffers from the inability to express individual user interests, often referred to as subscriptions, close to the data sources. For every pull, the entire feed is sent even if there are none or few changes since the previous pull. Hence, a huge amount of RSS data is unnecessarily transmitted over the wire [10, 13].

Added pulling also comes with a high cost for the end user because he has to validate the additional incoming data. He is *in* the client-server loop, while our goal is to place him above the loop.

We conjecture that a push-based interaction scheme is more appropriate in this situation. That is, if data are validated close to its source, less data has to be moved over the wire due to less pulling. It is only when data changes, that it is potentially transmitted. Thus, less data needs to be evaluated by the end user.

In the WAIF [9] project[2], we build extensible mediator structures [16] between existing Internet services and clients. The idea is to extend existing services with push-based intermediaries. We provide new APIs to existing services, in this case push-based interfaces validating data close to its source. This way, we transform existing Internet services into publishers through an expressive interface. Similarly, we turn traditional browsing clients, into asynchronous subscribers.

The rest of the paper is organized as follows. In section 2, we discuss our architectural goals for push-based web service wrappers. Next, in section 3, we present our wrapper implementation, the WAIF Proxy. Section 4 is a case study of two proxy deployments. Experiments with these two deployments are described in section 5. Before we conclude, section 6 discusses the overall concept and section 7 presents related work.

2 Architectural Goals for Push-based Web Service Wrappers

The component model presented by Web service technologies offers RPC API across widely available protocols, such as HTTP. This might not be appropriate for all structuring needs. Hence, we conjecture that a more push-based Web service interface should be

---

[1]We define *pulling* as an unconditional get operation. Similar, we define *polling* as a check for updated content, without actually getting the data.

complementing the traditional pull-based Web service model. This will potentially reduce the amount of unnecessary pulling. Our conjecture is that we can build systems that minimize pulling by wrapping Internet services with a push-based event notification component.

The fundamental problem is that remote data changes. Unless such updates are produced in regular or well-known intervals, it is impossible to guarantee that a pull request is made once and only once for every data update. This is illustrated in Figure 1, where a client must repeat the pull request arbitrarily often. Frequent pulling turns into a scaling problem when many concurrent users try to capture all updates when they occur. Consequently, popular Internet services such as Slashdot[3], punish users performing excessive polling by temporarily blocking the originating IP. Data filters shared by many users and placed close to remote sources can reduce the amount of unnecessary pulling drastically. Although upstream evaluation with shared data filters can be computationally intensive, its use can be justified if a significant amount of the data to be sent over the wire is blocked.

Our approach is through Internet service wrappers interposed on the traditional client-server communication path. This is a design that Web services fit well. To specify complex parameters as part of a Web service request is efficient, and potentially more precise than content adaptation in mediators. Semantic enhancements to Web services [14] can help wrappers use ontological and semantic information to provide better service to their users. The ability to correlate events enables a Web service wrapper to further enhance precision. By correlation, we mean to combine data from a series of events into one, be it to track remote data changing over time, or to collect and combine event data from different sources. Performing correlation on the mediator level and not on the client side makes sense in combination with upstream evaluation, because they share the need for event dissemination. However, performing event correlation across different remote services requires the ability to push events to a service other than the one issuing the subscription.

Data consumers can now be given a rich, expressive interface for data filtering.

The push-based architecture in Figure 2 illustrates our approach. First, an initial `subscribe` request is issued. This contains a query and a (client) owner address. The query is equivalent to a pull request, but can contain push-specific preferences such as pull-frequency or delivery at specific times or rates. Next, the wrapper is activated, which pushes updates whenever they
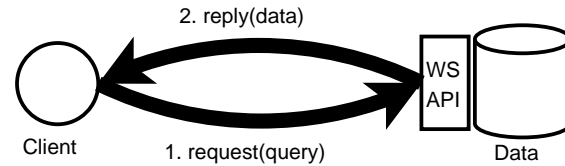

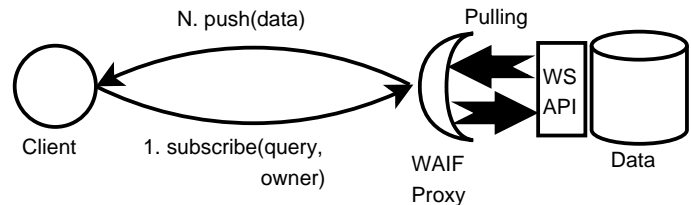**Figure 1.** Traditional pull-based client-server architecture.

Figure 2. A push-based architecture where the wrapper does all the pulling.

happen or exactly when the user wants them delivered. If a wrapper has many subscribers, publish-subscribe substrates can be used between the wrapper and its clients for efficient event delivery.

We can now present the design goals we strive for in engineering push-based Web service wrappers. The goals are not limited to any current technology, like .Net[4], J2EE[5], WebLogic[6] or Python for Web services[7].

Evaluate Persistent Requests Close to Data
To potentially reduce the amount of unnecessary data sent over the network, a wrapper should store and evaluate requests close to the data source. This is upstream from the client on the traditional client-server path.

Improve Service Expressiveness
Wrapper parameters should equal or supersede the parameters of the underlying Web service. A wrapper should not give its users less expressiveness than the underlying service. Adding a push-interface enables a wrapped service to offer new functionality, like timed delivery or personalized filtering.

---

[3]http://slashdot.org

[4]http://www.microsoft.com/net/

[5]http://java.sun.com/j2ee

[6]http://www.bea.com/content/products/weblogic/

[7]http://pywebsvcs.sourceforge.net/

Enable Event Correlation
To further increase clients ability to precisely define their interests, a Web service wrapper should enable correlation between events. This enables new events to be triggered by previous events, where the produced event contains, or is based on, data or state aggregated over time or by different sources.

Upstream evaluation also potentially gives better precision. Publishers can now perform individual comparison of an event to subscriptions and possibly reduce the number of messages sent. The alternative is a push-based wrapper that pushes every data update to its subscriber, without evaluating whether the data should actually be sent.

3 The WAIF Proxy

Guided by the goals stated in Section 2, we have designed and implemented a prototype Internet service wrapper, the *WAIF Proxy*. Designed to run close to popular Internet services, it can be easily customized to wrap any pull-based resource. This includes Web services, but also regular web pages, databases, and file systems. It is called a proxy because it acts as a mediator between push- and pull-based software protocols, and because it performs computations on behalf of individuals or groups of users. Our design is built on a general structure for extensible servers we previously presented in [1].

The basic functionality of a WAIF Proxy is to wrap some pull-based resource, and offer subscriptions to data changes generated by regular pulling. Deploying the proxy requires adding application specific logic to load data from an external resource, to parse incoming events, and to use loaded or incoming data to create new events. Hence, it meets our first design goal by storing and evaluating persistent requests close to the data source. Personalized event subscriptions can contain push-specific parameters on, for instance, delivery. Our wrapper will thereby supersede the expressiveness of the underlying data source, satisfying our second design goal.

Our implementation is a Python[8] package. We chose Python as our development platform because it allows easy and flexible prototyping, with rich support for Internet applications. Our package contains a set of basic modules optimized for a range of situations. This enables monitoring of items in RSS streams, objects on web services and even low-level file system events. Not
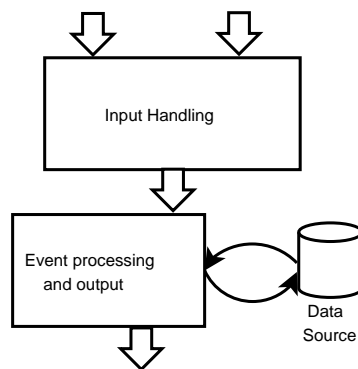

Figure 3. Overall view of a WAIF Proxy pulling an external data source.

only does the proxy supply a push-based resource interface, it allows users to build personalized network services by combining proxies to form custom applications. Examples include personalized alarms for people who want bus routes for work only in cold weather or when traffic is congested. The WAIF Proxy handles two types of events: *internal* events triggered by data updates on a monitored source and *external* events received from other WAIF proxies. Event handling may change the state of a proxy, or it can trigger new events. Connecting WAIF proxies to form a personal network service implies event correlation, our third design goal for push-based Web service wrappers.

Adding a push-based interface to a pull-based resource is motivated by the assumption that both users and resource suppliers gain from it. On the publisher side, it depends on the cost of handling millions of users pulling versus the cost of matching events with millions of subscriptions. Pushing does indeed pose a significant cost, and this is also why hierarchical publish-subscribe networks [4, 15, 5] were designed with focus on the matching and delivery of events in wide-area environments. In this setting, a WAIF Proxy can be used as a top-end publisher in hierarchical networks. Filtering networks can also be created using WAIF Proxy instances without adding custom logic and thereby only using their subscription and filtering functionality. Filters must however be placed directly at every proxy, and they must include target addresses since we do not apply hierarchical filter forwarding.

3.1 Generic Structure

The WAIF Proxy is completely event-driven and handles all events asynchronously. Implemented as an object-oriented structure, it makes extending and adding custom event handlers easy. The WAIF Proxy
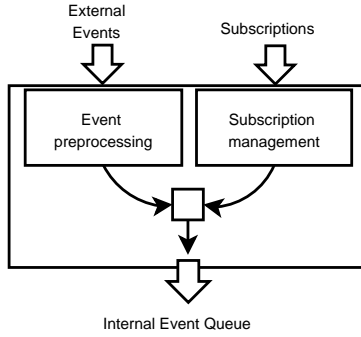
Figure 4. The input handling component.

Figure 5. The event processing component and its data source.

is divided into two threads communicating through a shared, synchronized Python Queue object. The division enables asynchron event handling, since one thread can receive, verify and queue new events while the other is handling them. The Python thread model does not offer true parallelism, so performance will not improve if we add more threads. The separation will, however, decouple incoming event deliveries from outgoing deliveries. Figure 3 shows the proxy with its two sub-components and an external, pull-based data source. Both components are customizable and extendible to fit any pull-based data resource. New customized proxies can build upon our Python module to cooperate with other WAIF Proxies. However the communication protocol is built on SOAP, so implementations in other languages are possible.

The input component in Figure 4 handles incoming events and subscriptions, both delivered via the same SOAP interface, but via separate SOAP-RPC calls. Even though subscription updates could be treated as special-case events, we chose to separate them to streamline common-case performance. Using synchronous RPC to accept incoming events and subscription updates gives WAIF proxies a chance to communicate status and error messages. Event validation and pre-processing, and subscription updates are handled synchronously. This means incoming RPC calls can be returned quickly. Valid events are then put on an internal event queue for asynchronous event handling. On Figure 4 this queue is shown as the bottom arrow. New or updated subscriptions can also produce events, for instance if a new user can instantly be delivered events from the proxy data cache. Such events are also put on the internal event queue.

The event processing component, shown in Figure 5 has an internal event scheduler. The thread waits for incoming events on the shared event queue, and the scheduler delegates control to a suitable handler for the event. Handlers are registered under aliases in a
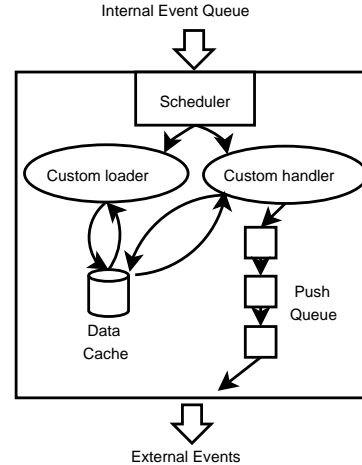
hash-table on initialization, and events should contain a reference to one of these aliases. Otherwise, they will be fed to the built-in default handler. If the default alias is not taken by other handlers in the hash-table, the built-in default handler will log an error message and return. The output of an event handler is either nothing, a new internal event, or a new event to be pushed to a remote WAIF Proxy.

To produce events based on pulling some web resource, a programmer can implement a custom data loader function that adds customized objects to the internal data cache. If a function called `loader` is present in a class descending from the WAIF Proxy class, it is automatically discovered. Data returned from a loader is checked for validity, and marked with a timestamp. Cache loading is scheduled like regular events using built-in timers that sleep in intervals and only wake up to queue new loading events. Likewise, built-in handlers are invoked by timers to make sure the content of the data cache and other state information is written to persistent storage. Timer intervals are adjustable to fit the application and the desired event rate. If the cache has changed, the update can trigger events to users. Timers can also be used to, for instance, batch events in daily deliveries.

3.2 Extensibility and Configuration

To support the full range of pull-based web resources, the features of our generalized proxy service are easy to adapt and customize. Since the WAIF Proxy contains only what we regard as minimal functionality, we allow extensibility in both the input com-

ponent and the event processing component. However, extensions are not allowed during runtime. Making runtime extensions safe without limiting functionality and performance is a difficult task.

The WAIF Proxy is distributed in Python source-code, and is designed to be easy to extend and customize for programmers with knowledge of object oriented and event-driven programming methodologies. A customized wrapper should extract key data from the wrapped service by fine-grained pulling.

Our generalized proxy framework is open for *any* arbitrarily rich query and configuration language. This is possible because the proxy framework only defines the parameters it actually needs, and all others are accepted and passed on to the internal event handlers. Associative arrays, such as Python dictionaries, suit this purpose well.

A subscription is initiated with a SOAP call to the `subscribe` function with the mandatory parameters shown in Table 1.

waifID	User ID of subscription owner.
taddr	Target URI for event notifications.
params	Optional parameters. See Table 2.

Table 1. Mandatory Subscription Parameters.

The `taddr` URI should point to another WAIF Proxy. Subscriptions issued with an empty `taddr` parameter will not fail, only yield a warning. A target address is not necessary for proxies set up to deliver data via a GUI or alternative protocols, like email or SMS. In that case, an alternative address must be given. The `params` dictionary can carry optional parameters that will be used by the proxy, as shown in Table 2.

localID	ID of existing subscription.
DataType	ID or alias of event handler.
interval	Repeated timer every `interval` secs.
countdown	Timer counting `countdown` secs.
email	Alternative delivery address.

Table 2. Optional Subscription Parameters.

The `interval` and `countdown` parameters will initiate timers that, after the specified amount of time, will trigger an internal event to this subscription.

If the subscription parameters do not contain the `localID` parameter, the call will return a new subscription identifier. The (`waifID`, `subID`) tuple is considered a unique key, and is later used to validate external events delivered directly to individual users. Thus, a subscription creates a permanent proxy that keeps state for each user. A user can have multiple subscrip-

tions, to, for instance, use multiple different filters on the same data.

Communication between proxies or between a proxy and a user is established by subscriptions, and events are pushed accordingly. External events from other proxies are delivered via the `notify(waifID, subID, event)` call, which triggers an internal event for the specified subscription. If no such subscription is found, the proxy will return an error message to the caller. However, a proxy can be configured to accept all events, and events carrying data can define whether they are available for all subscribers or just one specific subscription. Incoming data can be correlated with data from the receiving proxy, given that the receiver understands the data format and content. Our message format for communication between WAIF proxies lets filtering applications define their own data formats. For topic-based filtering, this implies we do not specify a certain topic space, and there is no Web service-style name-space specification scheme. Hence, collaborating proxies will need a common data format.

A subscription is active until an `unsubscribe(waifID, subID)` call is received. This will not delete the user profile by default, only deactivate it.

3.3 Event Format

We distinguish between internal and external events, but they have similar formats. The format of an external event that gets delivered to the WAIF proxy via the `notify` function is shown in Table 3.

waifID	User ID of subscription owner.
subID	Subscription ID.
message	An associative array for event data.

Table 3. External Event Format.

Both external events delivered via the `notify` function and internal events are added to the same event queue. External events are preprocessed, type conversions made by the SOAP infrastructure are undone, and a handle is added to identify the event handler for this event. At this point, the two event types are not distinguished, and they have the same format, as shown in Table 4. The type conversion from SOAP associative arrays to Python dictionaries and lists is the reason the `message` parameter in Table 3 changes name to `payload` in Table 4.

`waifID`	User ID of receiving subscriber.
`subID`	Subscription ID.
`handle`	Name of chosen event handler.
`payload`	Dictionary for application data.

Table 4. Internal Event Format.

4 Case Studies and Applicability

Our WAIF Proxy has been tested in numerous application areas, ranging from news recommendations, fine-grained file system eventing, persistent Web search wrapping Google, weather alerts, commuter information, stock quote updates and RSS feed alerts. We will describe the latter two here.

4.1 The WAIF Feed Proxy

Web syndication[9] using RSS, Atom or similar protocols has become a popular, although resource straining, Web application. The idea behind syndication feeds for Web resources is that frequent users can efficiently get a brief overview of the latest updates without explicitly visiting the feed source with a Web browser. The user registers feeds in a feed reader application on his computer, often called a news aggregator, and lets this reader pull selected feeds arbitrarily often. Although relieving the user from unnecessary manual pulling, there is little or no evaluation on the feed source of whether a user should be notified of a feed update. Thus, the user relies on frequent pulling and client-side evaluation to stay updated. This puts a high strain on the feed source and the network, having to serve many users checking for updates. These characteristics of RSS feeds and client behaviour are investigated in [10]. Their conclusion is that a better update notification scheme is needed, because much of the bandwidth is consumed by re-fetching feeds. If publishers notify subscribers about when they should pull for updates, less bandwidth would be consumed. *rssCloud* is an RPC upcall system where subscribers ask for a lightweight SOAP notification when new feed updates are available. However, this feature has rarely been implemented by feed publishers since its introduction with RSS 0.92 in 2001[10].

Our feed proxy is designed to alleviate problems both for the user and the feed source. By reducing the number of messages sent over the wire, we reduce both network strains for RSS publishers and the need to evaluate data on the client side. Our goal is keeping bandwidth consumption to a minimum, while optimizing timely delivery of feed updates.

A well-known technique to help scale high-traffic servers is to insert mediators on the client-server path and replicate data or functionality across nodes. A WAIF Proxy acting as a mediator on the client-server path allows us to relieve strain from busy news servers. However, simply moving computation from the server to a mediator will only serve to move the scaling problem to another node in the system. Thus, a mediator structure must have added functionality to scale the entire system.

Our mediator is a WAIF Proxy that can pull feeds updates on behalf of its users. It can pull as often as necessary to catch updates in a timely manner, and will push only the differences to its subscribers when new data becomes available. This is similar to the functionality of the FeedTree[13] peer-to-peer micronews distribution network.

The implementation uses the Python feedparser[11] and feedfinder[12] libraries to find, fetch and parse feeds. These libraries support all known feed protocols and formats, like RSS and Atom, enabling the feed proxy to extract and forward feed entries on a simple, common format. Users can subscribe by specifying a feed by base URL, direct URI or feed title, and optionally indicating how often they want notifications. The default behaviour is to pull every feed not more than every thirty minutes, to avoid getting blocked from busy sites like Slashdot.org, who will block IPs of users that pull too often. Subscriptions giving only a title of a feed will fail unless that feed has already been added to the feed cache by another subscriber. To save bandwidth, the feed proxy does not fetch data from feeds without subscribers. When a feed has been updated, every subscriber to this feed is notified. If a feed no longer has subscribers to it, the feed proxy will stop updating it, but it will stay in the cache.

Feeds with subscribers will by default be pulled once every 30 minutes. This seems to be a reasonable interval, since some sites block users pulling more often than that. Setting individual update intervals for each feed is not implemented, even though [10] suggests it is more appropriate.

An example subscription to the BBC News Service is given in Table 5. It contains a delivery URI and an email address in case a notification is undeliverable via the SOAP interface.

A feed is identified by a URI where an XML file is published and republished when it changes, and each version of this file has several entry items with ti-

---

[9]http://en.wikipedia.org/wiki/Web_syndication

[10]http://www.thetwowayweb.com/soapmeetsrss

[11]http://www.feedparser.org

[12]http://diveintomark.org/projects/feed_finder/

waifID	larswaif
taddr	`http://waif.cs.uit.no/feedclient:` `7878`
params	{feed:bbc, email:larsb@cs.uit.no}

Table 5. Feed Subscription.

tles, summaries, and links. An updated feed may not mean that every entry is new, experiments show that merely changing some of the summary text or even ordering of entries will trigger a feed update. Some sites use `HTTP/modified` headers or ETags on both the main feed file and each entry. These can be used for conditional `GET` operations. Servers responding with `HTTP/304` messages make it easier to check for modified feeds, but it is not always possible to know exactly what data has been modified and how. Therefore, headers cannot be fully trusted to tell whether a feed should be pushed to subscribers.

Consequently, RSS users must pull frequently to make sure they get all updates, and also need to either have a smart client able to detect duplicate updates or perform manual evaluation, i.e. keep reading RSS news all the time. Thus, RSS users may not experience saving much time compared to regular Web browsing.

To detect duplicates, a feed is broken down to entries, and for every new feed pull, each entry is compared to every other entry we have cached for that feed. Currently, we only look at the link to the full article, however we are currently experimenting with a fuzzy duplicate detection scheme to apply on the summary text. It is a hard task to determine whether a story has been updated with breaking news, or simply edited. A problem with using the link as a key occurs when feed servers use `HTTP/302` forwarding schemes for load balancing. We have seen servers change links for every pull operation, and in combination with lack of ETags or modified headers this makes it nearly impossible to perform duplicate detection without fuzzy content analysis.

An example feed entry being pushed to a user subscribing to the BBC UK News service is given in Table 6.

Title	*Secrets of largest fish revealed.*
Summary	Researchers use satellite tags to gain unprecedented insight into the life of the whale shark, the world's largest fish.

URI	`http://news.bbc.co.uk/go/rss/-/`
	`1/hi/world/asia-pacific/4273844.`
	`stm.`

Table 6. Feed Entry Format.

Updated or new feed entries with new links are declared true positives and written to the feed cache, existing entries are declared false positives and thrown away. The feed cache will only keep as many entries as the feed contains at any given time, and does not support chronological queries for outdated feed entries. After a successful update pull, the proxy will push every new entry to subscribers of this feed. Typically, when signing up, a user receives the complete content of the feed cache for that feed, and after that only receives new or updated entries.

To receive feeds from the WAIF Feed Proxy, you need a special client application. It is also built on the WAIF Proxy framework, and features a GUI do display the news entries. Currently, a Trillian[13] plugin is being developed.

4.2 The WAIF Stock Quote Proxy

Buyers and sellers of publicly traded stocks usually want to be kept up to date on the movements in stock prices. Price movements, both the direction and the amount, are hard to predict: they depend on complex factors and we usually do not know in advance when they will occur. Hence, stock quote alerts are a very common application for push-based systems.

The WAIF Stock Quote Proxy is designed to pull stock quotes from a public Web service, and generate alerts upon price movements. Users subscribe by giving a list of stocks to watch, and how much a certain stock should move before an alert is pushed. The reason why we expect significant bandwidth savings using a push-based approach, is that the most actively traded stocks, measured both in dollar and share volume, rarely make radical movements. During a trading day, such stocks are not expected to move more than a little bit, as for large trading volumes, a change of only 1% is worth a lot of money. Hence, if stocks such as Google or Microsoft moves more than 0.5%, it may be considered unusual and thus a subscriber to these stocks would want to know about it. Smaller movements should not disturb the user. A sample subscription is shown in Table 7.

waifID	larswaif
taddr	`http://waif.cs.uit.no/` `stockclient:8787`
params	{tickers:[orcl,goog,msft,sunw], threshold:0.5}

Table 7. Stock Quotes Subscription.

---

[13]`http://www.ceruleanstudios.com/`

The threshold level sets how much a stock has to move before an update is considered a true positive. Each new update is compared to the previously stored true positive. If a stock quote subscriber has a 0.5% threshold setting, he will receive two notifications if in the course of one day his stocks moved up 0.5%, and then 0.5% down again. A sample stock quote alert is shown in Table 8

ticker	goog
price	$315.68
previous	$314.10

Table 8. Stock Quote Notification.

When a stock quote update is declared a true positive, the new value replaces the previously cached quote, and gets pushed to its subscribers.

5 Experiments

To demonstrate the potential advantages of using push-based WAIF proxies as wrappers for pull-based Web services, we have experimented with the two applications we described in Section 4.

A requirement for optimal pulling is that every pull request should return a true update, i.e. new data. The unpredictable nature of dynamic and volatile systems like stock trading or news feeds, implies that data are not updated on a planned schedule. In a pull-based system, the only practical solution to avoid missing updates, is to increase the pull frequency of the clients. However, this means that some pull requests will return new, changed data (true positives), and others will return unchanged data (false positives). Hence, the ratio between true and false positives is a measurement indicating the amount of unnecessary network traffic and server resources.

The motivation for both experiments is to investigate applications typically implemented using frequent pulling, and to check the amount of pull requests returning false positives. Since our experiments are implemented using the WAIF Proxy, this is easily measured by issuing subscriptions and recording the amount of output the subscribers receive.

Note that the cost related to matching data updates to subscriptions and pushing events to subscribers, although interesting, is outside the scope of this article.

The computer running the experiments from Tromsø is a Dell Dimension 360 (Win XP Sp2) with a 3.2GHz P4 CPU and a 100Mbps network connection. The computer we used at Cornell University is a Dell Dimension 8100 (Red Hat Linux) with a 1.4GHz P4 CPU and a 100Mbps network connection.

5.1 RSS News Feeds

Our first experiment is based on measuring the filtering performance of the WAIF Feed Proxy described in Section 4.1. The application is a WAIF Proxy able to pull any RSS feed in a given interval and generate alerts to subscribers when new items appear in a feed. In our experiment, we consider realistic subscriptions to popular newsfeeds, but do not actually push updates to real subscribers. When a feed contains new entries, we declare them true positives. A pull returning old entries, or only slightly changed entries, is declared a false update. The latter result is declared when a feed changes the order of its feed entries or changes a summary text without actually changing the target link.

By regularly pulling, we investigate how often a set of selected feeds are updated with new content. Our conjecture is that with a moderate pull interval, rarely updated feeds will mostly generate false positives. Others with higher update frequency will give a higher share of true positives. The ratio between true and false feed updates depends on the content and the publisher of the feed, since different information topics will have different publication rates.

Table 9 contains the ten news feeds used for our experiment. They were selected to capture feeds with different characteristics of their content and their update frequencies.

Extensive measurements [10] has been performed at Cornell University on the quantitative characteristics of RSS as a publish/subscribe system for the Web. This study shows that we can expect a representative selection of feeds to show very different update frequencies. According to their study, approximately 55% are updated hourly, while 25% have updates within days or weeks. While this extensive study of approximately 100,000 feeds discussed collective properties, we can reproduce their results in concrete examples. Hence, we can use their analysis to suggest how some feeds could benefit from pushing their updates to subscribers, while other feeds should rather be frequently pulled.

The results from pulling the ten feeds in Table 9 every 30 minutes over 10 days is shown in Table 10. Each feed has been pulled 500 times, and the proxy tries to use conditional GET where ETags or modified headers are available. The first pulls returned between 6 and 24 entries per feed, and the data in the table are based on the 499 following polls. The first column, *Updates*, show us that nine out of ten feeds are either not responding with HTTP/304 Not Modified or are actually updated within every 30 minutes. ACM Queue turns out to have a very well-behaving feed, since it only

Feed Title	Base URL	Content Type
AP	`http://hosted.ap.org/lineups/TOPHEADS.rss`	AP Top Headlines
AP Sports	`http://hosted.ap.org/lineups/SPORTSHEADS.rss`	AP Sports Headlines
Reuters	`http://today.reuters.com/rss/topNews`	Reuters Top News
Aftenposten	`http://www.aftenposten.no/eksport/rss-1_0/`	Norwegian News
BBC News	`http://newsrss.bbc.co.uk/rss/newsonline_uk_edition/world/rss.xml`	World News
Slashdot	`http://rss.slashdot.org/Slashdot/slashdot`	News for nerds
CNN.com	`http://rss.cnn.com/rss/cnn_topstories.rss`	CNN Top Stories
Google News	`http://news.google.com/news?q=web+services&output=rss`	Search "Web Services"
Google News	`http://news.google.com/news?q=whale+shark&output=rss`	Search "Whale Shark"
ACM Queue	`http://acmqueue.com/rss.rdf`	IT Trends

Table 9. Selected News Feeds

Feed Title	Updates	Entry updates	Feed length
AP Headlines	490	4816	10
AP Sports	490	4814	10
Reuters	492	892	10
Aftenposten	499	513	10
BBC News	426	448	24
Slashdot	496	364	10
CNN.com	499	184	6
Google WebSrv	499	117	10
Google WhaleSh	499	40	10
ACM Queue	11	2	10

Table 10. Feed Test Results For 500 Pulls

claimed to be updated 11 times after the initial pull. However, the next column, *Entry updates*, shows how many feed entries were actually updated, according to our duplicate detection algorithm. We see that the two feeds from Associated Press (AP) seem to update all 10 entries for every pull, e.g. within every thirty minutes. However, this result may not be correct, as the AP server seems to use randomized `HTTP/302` redirection between different servers. If the URIs in the feed changes for every pull, our scheme will fail to detect duplicate entries with different links. Further, we see Reuters are frequently updated, with 1.8 updated feeds per 30 minutes. Our two personlized feeds from Google News on "Web services" and "whale sharks" did not return many entry updates. The latter only 40 had new entry updates, but we had to pull and evaluate every time because the Google server does not respond to conditional `GET` operations.

Quantitative measures on the updates of Web data, and RSS feeds especially [10], have suggested that each feed should be pulled on an individual schedule, which our findings clearly support. However, since most of the pull operations does not really return new, valid updates (true positives), a more push-based scheme where only the updates are pushed can be suggested.

In this case, subscribers to our feed proxy could have received up to 92% fewer messages (in the case of the "whale shark" feed). We also see that feeds from the federated news agencies AP and Reuters mostly get forwarded right away because they have many updates. As such, the extra processing to confirm these updates are mostly wasted.

Our results tell us that the update characteristics of feed data is important for the success of pushing feed updates. It seems that rarely updated feeds are better suited for push, because of their low update ratio.

5.2 Stock Quotes

Our stock quote application is a WAIF Proxy built to pull stock quotes from a publicly available Web service. It generates alerts upon certain price movements. Section 4.2 gives a case study of this application.

For our experiment, we chose to monitor ten of the most active stocks on the Nasdaq and NY Stock Exchange during June 2005 (measured both in number of transactions and dollar volume). The stocks are listed with symbols and company names in Table 11.

Our alarm service wraps one of XMethods[14] demo

---

[14]`http://www.xmethods.net`

Symbol	Company Name
INTC	Intel Corporation
MSFT	Microsoft Corporation
SUNW	Sun Microsystems, Inc.
GOOG	Google Inc.
CSCO	Cisco Systems, Inc.
YHOO	Yahoo!, Inc.
LU	Lucent Technologies, Inc.
XOM	Exxon Mobil Corporation
TWX	Time Warner Inc.
PFE	Pfizer, Inc.

Table 11. Monitored Stock Quotes.

Web services; *Delayed Stock Quote*. It's interface is very simple; the function `getQuote(symbol)` returns the price for that stock as a float, with a 20 minute delay compared to real-time trading.

The machine and bandwidth resources of the XMethods service are limited, and we do not know anything about the efficiency of its implementation. However, this is a typical scenario for developers of applications that rely on publicly available Web services. We only know its IP address, indicating a location in San Jose, CA[15].

For an update to be declared a true positive, the stock price had to move beyond a certain threshold. Hence, when a stock price had increased or decreased a certain percentage since the last true positive was recorded, it would be considered a new true positive. In this way, we avoided storing more than one value per stock, and only triggered alarms when the threshold had been reached.

A stock traded in very high volumes is expected to have many small and insignificant changes, but we wanted to record only significant changes. Although subscribers to our stock alert service would probably want to set this threshold themselves, we tested our service with three threshold levels; 1.0%, 0.5% and 0.25%. On average, we do not expect that the most traded stocks move more than that during one trading day, and for large trading volumes, a change of only 1% can be significant in a trading context. If the subscriber is not actively trading at the moment, pulling for updates once per minute should be more than enough. Since the Web service we based our wrapper on seems to be located in San Jose, Ca., we decided to test both from the University of Tromsø, Norway, and from Cornell University, NY, to investigate the impact of long distance latency. Detailed test results are shown in Ta-

---

[15]`http://www.geobytes.com/IpLocator.htm?`
`GetLocation&ipaddress=64.124.140.30`

ble 12. The test data was collected between September 26. and October 11., 2005.

Site	Threshold	Pulls	TP	Ratio
Tromsø	0.25	13310	1475	0.1108
Tromsø	0.50	7640	177	0.0231
Tromsø	1.00	6980	55	0.0079
Cornell	0.25	7750	852	0.1099
Cornell	0.50	7670	196	0.0255
Cornell	1.00	9130	45	0.0049

Table 12. Test Results.

Each stock quote was pulled in approximately 3000 times per trading day. The tests run with the highest movement threshold clearly yielded fewer true positives, both with an update ratio well below 1.0%. The middle threshold value gave ratios at approximately 2.5%, and the lowest threshold gave many more true positives, with both ratios at around 11%.

In message count, this means that a subscriber to our service could receive 99% less messages, if he only cares about movements larger than 1.0%. Still, the stocks are monitored every minute so the user receives the required information quickly. When the threshold level decreases, more updates will qualify as true positives and less bandwidth is saved. Lowering the threshold for update pushes, means moving closer from push to pull. Still, with the lowest threshold we tested, 90% of the updates were false positives.

The latency measurements are omitted since they did not pose a significant cost, and did not suggest moving the evaluation closer to the source.

6 Discussion

Urgency matters. The ability to capture one or a series of remote events as early as possible is a key differentiator in a modern society. An obvious example is a stock broker getting public information about a traded company slightly prior to another. We have studied an Internet structuring technique we conjecture as important in this context, and we will discuss some of our experience in this section.

6.1 An Expressive Push Structure

The widely adopted client-server model is not always the best structuring solution for modern Internet applications. First, to capture remote events as early as possible, a client needs to pull data over the wire very frequently. This stresses the remote server, as experienced by popular RSS publishers [10]. Also, far too much data is sent over the network as a result.

Second, the session based client-server scheme requires that a user (or client program) is constantly in the computational loop. This creates unnecessary interrupts at the client side, in particular if the frequency of false positives is high. The best would then be to offload this validation process to the opposite edge-point of the Internet and only send high precision data over the wire.

Push technology alone does not solve these problems properly. The reason is that it is hard to specify exactly what to push over the wire due to lack of expressiveness in the server API. High expressiveness comes with an additional burden, since more fine-grained computations must take place. Hence, typical push inspired server APIs like, for instance, RSS feeds, has a coarse-grained, topic-based push interface. Similarly, publish-subscribe systems like, for instance, Gryphon [15], Siena [4] and SCRIBE [5] provide topic-based, or limited content-based, subscriptions to published information.

In the WAIF [9] approach, we strive for extreme expressiveness of subscriptions. Our approach is through deployment of very personalized filtering code on the path between an Internet publisher and the client. We have shown the potential in doing such expressive upstream evaluation close to the data source; one anecdotic example indicates that over 99% of communication can be avoided.

We deploy these filters at mediator structures [16] that turn existing Internet services into publishers. Similarly, we change traditional browsing clients, into asynchronous subscribers. Our goal is to complement the pull-based Internet, with a push-based one delivering high-precision information in a timely manner. We do not envision that we can change the API of, for instance, Google or Amazon, but we can build external proxy structures turning such services into expressive, push-based ones.

Our results indicate that we must balance when to deploy expressive push and when to deploy client-server structures. The obvious solution for a client-server solution, for instance, would be a server containing only historic data. The other extreme is a server with rapidly changing content. At first, one might conjecture that frequent changes at a server would be the ideal candidate for push and upstream evaluation. We are not so sure, though, that there is a simple answer to this, since frequent updates typically imply heavy computations at the server. Hence, from the server perspective, it is probably better that data is handed off remotely. At the same time, many true positives also imply network transfers and client interruptions. A coarse grained RSS push might then be a good candidate here.

The approach we have taken, by an extra intermediate structure pulling the server for data, and doing the additional parsing of the data before potentially pushing it to clients, is a good scaling approach. The server does not do the parsing, and the client evaluation and most of the network traversal is avoided. Hence, extra functionality comes at the cost of this incremental proxy structure.

When data are updated less frequently, push is a stronger candidate than client-server techniques. This way unnecessary polling for new updates is avoided.

6.2 Push vs. Pull

Both news feeds and stock alert services are typical applications where we know nothing about when and how large data changes will appear. Although the news feed experiment showed us that push is not necessarily the best option, our results suggested that stock quote alerts are very suitable for push.

In a very realistic usage scenario, we showed that up to 99% of pulled stock quote data were false positives, i.e. wasted traffic. These results suggest that a more push-based architecture is very suitable for this type of application.

However, one may argue that our clear results in the latter experiment are caused by selectively picking stocks that fit our requirements. It is reasonable to presume that choosing a representative selection of all publicly traded stocks may give us results similar to the RSS feed experiment.

In that case, our results support the discussion put forth in [3], where algorithms for optimal combinations of push and pull strategies are discussed. Still, we conjecture that complementing traditional pull-based Web services with a push-based interface can significantly reduce the amount of unnecessary traffic on the Internet.

6.3 Personal Overlay Network Systems

We are past the point when multiple users shared a single computer. Now, a single user typically uses a network of computers, some of them servers shared with others. Technology as peer-to-peer file sharing networks like, for instance BitTorrent [6], blurs the picture even more, but where the picture is that a single user can have very many computers serving his needs.

We envision this trend to be further developed towards what we call personal overlay network systems (PONS). This is a dedicated network serving the needs of a single user. In a PONS, a user can distribute

and install highly personalized modules which should be doing as much as possible of often repetitive and tedious personalized tasks. Input also comes through feedback from other users by collaborative recommendation techniques. Only when certain events or combinations of events occur, should a user be alerted. This overlay structure is transparent for the user, giving the impression of a personal overlay network system pushing data towards him.

Expressiveness in a PONS is far better than in existing systems since we can deploy any type of programmed Web service in this system. However, the novice user does not know that code is configured on his or her behalf, where this code is actually running, and the like.

7 Related Work

We have basically patched the initial client-server architecture to accommodate the exponential growth of dynamic content on the Web. This includes, for instance, web proxies, DNS name resolution techniques, scalable server farms, search engines, directory services, multi-threaded browsers, and the like. However, we argue that this is probably not sufficient, in particular if we want to provide support for a new class of applications automating more of the tedious tasks of all the millions of users on the web. Hence, we conjecture that the structure of the Internet is ripe for change.

We will now discuss different technologies that have all affected our design choises while developing WAIF Proxies as push-based wrappers for Web services.

7.1 Extensibility

In WAIF, we investigate expressive push structuring techniques. That is, we study how to extend existing mediator structures using Web service technology. Extensibility as a concept has been applied to computer systems for many years. Extensible operating system services and active networks, for instance, contain a body of related work. For instance, SPIN [2] and Exokernel [7] demonstrate how to dynamically extend operating systems at run-time. A system like STP [11] uses untrusted mobile code to upgrade communicating end-hosts at the transport level.

A key difference with WAIF, though, is that we extend user space web servers at run time, not in-kernel or protocol functionality. This is similar to how Web services in, for instance, Microsoft .NET can be used, where Internet services can be composed from other Web services.

7.2 Publish/subscribe systems

Existing publish/subscribe systems like, for instance, Siena [4], Gryphon [15], and Scribe [5] demonstrate that push-based approaches solve some of the scaling issues of the current Web. To accommodate scale, upstream evaluation techniques are combined with downstream distribution in these systems. Similar push-based approaches are also emerging with alert and subscription services added to popular web sites. For instance, both Google News and NY Times, provide a subscription-based service alerting users when interesting data appear at the servers. Nevertheless, two problems with this type of publish/subscribe systems are that they are proprietary and have coarse granularity. The latter might result in far too much data being sent over the wire, adding to the scaling problem.

Recently proposed publish/subscribe systems that specificly targets efficient distribution of RSS feed updates using collaborative polling are described in [13, 12].

7.3 WS-Eventing

WS-Eventing is a protocol[16] that allows Web services to subscribe to or accept subscription for event notification messages. As such, it is very similar to the way WAIF Proxies extend the existing API of pull-based resources like, for instance, Web services. The suggested standard will enable better interoperability between event producers and different routing substrates, a problem associated with current proprietary publish/subscribe systems. Only a early proposal for a protocol specification existed when we started our project in late 2003, so we did our own version. WAIF Proxies are generalized wrappers, and may contain functionality equal to current implementations of this protocol. Filters in WS-Eventing are optional, like in WAIF Proxies, and implementations are suggested, to use XPath[17] predicate expressions as filters.

7.4 Mobile Agents

Running specialized code on remote data sources is an application area well known within mobile agent research. The advantages of this approach were outweighed by the disadvantages, so mobile agent technology has never found a broad use [8]. Still, we believe that upstream evaluation of data close to the source

---

[16]http://msdn.microsoft.com/library/default.asp?url=
/library/en-us/dnglobspec/html/ws-eventing.asp

[17]http://www.w3.org/TR/xpath

is a viable technique for many systems like, including push-based Web services. Like mobile agents, we are able to evaluate data differently for each subscriber. The key to achieving the same expressiveness as with mobile executable code, is to export a highly configurable interface for subscribers. We claim that WAIF Proxies export this kind of interface, and thus can do many of the same tasks as mobile agent systems, without the security or performance issues often associated with mobile agents.

7.5 Semantic Web

The recent developments in Semantic Web[18] services display a trend where remote services and data are given very expressive interfaces. Attaching structured metadata to a service interface using the Resource Description Format(RDF)[19] allows client programs a better semantic understanding of that service, and better expressiveness to help extract more specific and relevant data. These interfaces are ideal for push-based applications, and may produce very interesting events. Better structuring of information also improves event matching, often the bottleneck of push-based subscription systems.

7.6 Adaptive Push-pull

Distribution strategies for dynamic Web data has been the subject of extensive research. Bhide et.al. [3] argues that since the popularity of Web objects vary over time, it is hard to a priori determine whether to use push or pull for a specific data item or stream. To aid this problem, they present and discuss adaptive algorithms able to optimally combine push and pull at a particular instant.

8 Concluding Remarks

In WAIF, we are investigating how to structure next-generation large-scale pervasive systems. The key design principle we advocate is proactive computing where information providers initiate dissemination of information. We conjecture that Internet applications filtering data close to remote sources scale better than pure client-server solutions. The potential net effect is that redundant or obsolete data are not pulled down over the wire. Remote filtering, however, suffers from the lack of expressiveness. That is, it is hard to take into account individual and diverse user needs through

standard, fixed server APIs. A typical Internet server is not extensible, especially not for this type of transformation to an Internet publisher.

We are interested in transforming traditional Internet services normally accessed through a client-server API into publishers. Fortunately, it is possible to add an intermediate proxy structure to a communication path between a client and a server. This proxy resides close to the Internet server wrapping it with a new API. In this case, it transforms a standard client-server API into a push-based one. Highly personalized filters running as Internet service front-ends now determine when and what type of data to push. We have experienced that the inherent structure of Web services lends itself naturally to this type of problem, and experiments indicate that a push-based Web Service wrapper can filter out redundant data without reducing service value.

Acknowledgements

Bibliography

[1] I. M. Arntzen and D. Johansen. A programmable structure for pervasive computing. In *The IEEE/ACS International Conference on Pervasive Services (ICPS'04)*, pages 29–38. IEEE Computer Society, 2004.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283, New York, NY, USA, 1995. ACM Press.

[3] M. Bhide, P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: Disseminating dynamic web data. *IEEE Transactions on Computers*, 51(6):652–668, 2002.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383, Aug. 2001.

[5] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), Oct. 2002.

---

[18]http://www.w3.org/2001/sw/

[19]http://www.w3.org/RDF/

[6] B. Cohen. Incentives build robustness in bittorrent. In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[7] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, New York, NY, USA, 1995. ACM Press.

[8] D. Johansen. Mobile agents: Right concept, wrong approach. In *5th IEEE International Conference on Mobile Data Management (MDM 2004)*, pages 300–301. IEEE Computer Society, 2004.

[9] D. Johansen, R. van Renesse, and F. B. Schneider. Waif: Web of asynchronous information filters. In A. Schiper, A. A. Shvartsman, H. Weatherspoon, and B. Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 81–86. Springer, 2003.

[10] H. Liu, V. Ramasubramanian, and E. G. Sirer. Client and feed characteristics of rss, a publish-subscribe system for web micronews. In *USENIX Internet Measurement Conference (IMC)*, New Orleans, Louisiana, Oct. 2005.

[11] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, and T. Stack. Upgrading transport protocols using untrusted mobile code. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 1–14, 2003.

[12] V. Ramasubramanian, R. N. Murty, and E. G. Sirer. Corona: A high-performance publish-subscribe system for web micronews. In *http://www.cs.cornell.edu/people/egs/beehive/corona/*.

[13] D. Sandler, A. Mislove, A. Post, and P. Druschel. Feedtree: Sharing web micronews with peer-to-peer event notification. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS'05)*, Ithaca, New York, Feb. 2005.

[14] K. Sivashanmugam, K. Verma, A. P. Sheth, and J. A. Miller. Adding semantics to web services standards. In L.-J. Zhang, editor, *Proceedings of the International Conference on Web Services, ICWS '03*, pages 395–401. CSREA Press, 2003.

[15] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An information flow based approach to message brokering. In *International Symposium on Software Reliability Engineering (ISSRE '98)*, 1998.

[16] G. Wiederhold. Mediation in information systems. *ACM Computing Surveys (CSUR)*, 27(2):265–267, 1995.