# Configuring Push-Based Web Services

Lars Brenna Dag Johansen

Dept of Computer Science
University of Tromsø, Norway
{*larsb, dag*}*@cs.uit.no*

## Abstract

*Much of the contents 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. 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. A 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 polling scheme can be applied. RSS feeds demonstrate how this poses a scaling problem due to more traffic. RSS suffers from the inability to express user interests, often referred to as subscriptions, close to the data sources. Hence, a huge amount of RSS data is transmitted over the wire [4].

Added polling 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 polling. It is only when data changes, that it is potentially transmitted.

In the WAIF[2] project[1], we build extensible mediator structures [7] 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 architectural issues for push-based web service wrappers. Next, in section 3, we present our wrapper implementation, the WAIF Proxy. Section 4 describes experiments, and, before we conclude, section 5 discusses the overall concept.

## 2 An Architecture for Push-based Web Service Wrappers

The component model presented by web service technologies is normally designed to expose a pull-based 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 complementing the traditional pull-based web service model. This would reduce the amount of unnecessary pulling. Our conjecture is that we can build systems that mini-

[1]Wide Area Information Filtering, is a joint project with Cornell University and UC San Diego. `http://waif.cs.uit.no`. This work is sponsored by the Norwegian Research Council IKT-2010 Programme and Programme No. 162349.

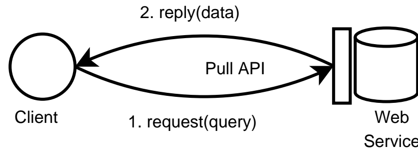


**Figure 1.** Traditional pull-based client-server architecture.

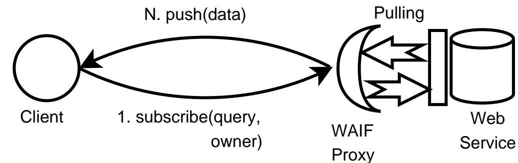


**Figure 2.** A push-based architecture where the wrapper does all the pulling.

mize 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 like, for instance, Slashdot[2] punish users performing excessive polling by blocking the originating IP for some time. Data filters shared by many users and placed close to remote sources can reduce the amount of unnecessary pulling drastically. Although filtering can be computationally intensive, it can be justified if a significant amount of the data sent over the wire turns out to be uninteresting to the user.

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 [5] 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. 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. Next, the wrapper is activated, which pushes updates whenever they happen. 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 identify several fundamental design principles for push-based web service wrappers. These principles are not limited to any current technology, like .Net[3], J2EE[4] or BEA[5], neither do they conflict with the more expressive interface improvements developed for Semantic Web Services[6].

**Persistent Requests Close to Data**
This is also called *upstream evaluation*, and it can reduce the amount of unnecessary data sent over the network. It can also be used to move computational load from thin clients to more powerful servers.

**Maintain 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.

**Enable Event Correlation**
To further increase client's 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 gathered over time or by different sources.

## 3 The WAIF Proxy

Using the derived design principles for push-based web service wrappers, we have designed and implemented a prototype Internet service wrapper, the *WAIF Proxy*. Designed to run close to popular Internet services, it is easily customized to extend any pull-based resource, be it web pages, databases, file systems, or web services.

The basic functionality of a WAIF Proxy is to monitor some source by regular pulling, and to let users of a pull-based source subscribe to events generated by data changes. Our implementation is a Python[7] package containing a set of basic modules optimized for

[2] `http://slashdot.org`

[3] `http://www.microsoft.com/net/`
[4] `http://java.sun.com/j2ee`
[5] `http://www.bea.com/content/products/weblogic/`
[6] `http://swws.semanticweb.org`
[7] `http://www.python.org`

a range of situations. This enables easy monitoring of items in RSS streams, objects on web services and even low-level file system events. Not 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, for example personalized commuting information. 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.

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 [1, 6, 3] 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 its object-oriented implementation makes extending and adding custom event handlers easy. It is divided into two components, one component to handle incoming events and subscriptions, and one component for event processing and output. Both components are customizable and extendible to fit any pull-based data resource. New customized proxies should build upon our Python module in order to cooperate correctly with other WAIF Proxies.

The internal structure of the WAIF Proxy is shown in Figure 3. There are two input channels, both accessed via the same SOAP interface. 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. New or updated subscriptions can also produce events, for instance if a new

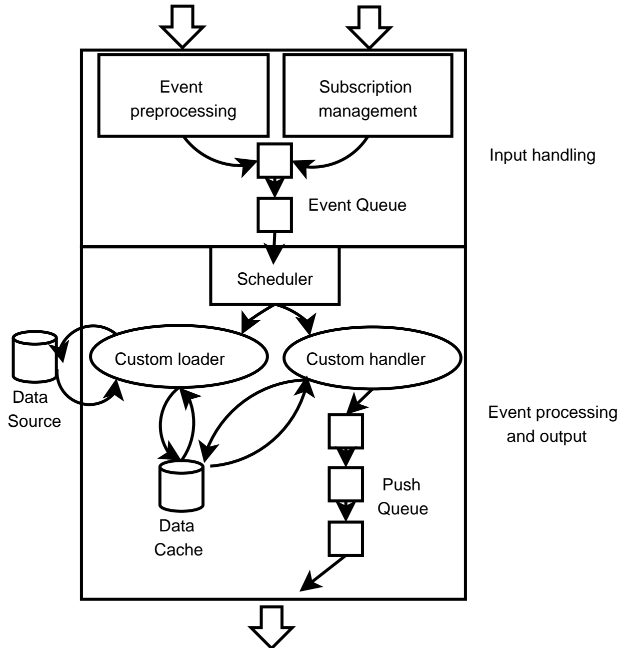


**Figure 3.** Internal structure of a WAIF proxy.

user can instantly be delivered events from the proxy data cache. Such events are also put on the internal event queue. Asynchronous event handling is enabled by having a separate thread for each of the two components. Thread-safe communication is achieved by using shared Python Queue objects.

The event processing thread 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 component 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 sourcecode, 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 like Python dictionaries suit this purpose well.

A subscription is initiated with a SOAP call to the `subscribe` function with the following mandatory parameters:

| | |
|---|---|
| `waifID` | User ID of subscription owner. |
| `taddr` | Target URI for event notifications. |
| `params` | Optional parameters. See params table. |

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 certain optional parameters that will be used by the proxy:

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

The `interval` and `countdown` parameters will initiate timers that after a given 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 subscriptions.

Communication between proxies or between a proxy and a user is established by subscriptions, and events are pushed accordingly. External events from other proxies is delivered via the `notify(userID, 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 means we implement no topic constraints. However, collaborating proxies will need a common data format.

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

### 3.3 Event Format

We separate between internal and external events, but they have a similar format. When an external event is delivered to the WAIF proxy via the SOAP interface, it must have the following format:

| | |
|---|---|
| `waifID` | User ID of receiving subscriber. |
| `subID` | Subscription ID. |
| `message` | An associative array for event data. |

Both external events delivered via the `notify` function and internal events are added to the same event queue. When an internal event is given to a handler, it has the following format:

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

# 4 Experiments

To demonstrate the potential advantages of using push-based WAIF proxies as wrappers for pull-based Web services, we have set up an experiment. The basic idea is to investigate an application typically implemented using frequent pulling, and to check the amount of pull requests returning false positives. Our application is a WAIF Proxy set up to pull stock quotes from a public Web service, and generate alerts upon certain price movements.

We selected the ten most active stocks on the Nasdaq and NY Stock Exchange during the beginning of June 2005, and set up an alarm service wrapping a demo stock quote Web service available from XMethods.net[8]. A custom `loader` function was implemented to pull the stock quote service's `getQuote(ticker)` function, once for each stock ticker. For each iteration of the pull sequence, the quotes were compared to quotes previously stored in the local data cache. Whenever a stock had moved beyond a certain threshold, the update was considered a true positive, thus replacing the cached quote. Users of this wrapping service are able to subscribe to alerts on some stock moving beyond some threshold. Alerts are generated when stocks monitored by active subscriptions have true updates, and forwarded as outgoing events.

Note that the cost related to matching data updates to subscriptions and pushing events to subscribers, although interesting, is not a topic in this article.

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[9]. Also, a stock quote alert service is a typical application where we know nothing about when and how large data changes will appear.

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, 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 clients pull frequency. 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.

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 broken. 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 two threshold levels; 1.0 % and 0.5 %. 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% is worth a lot of money. If the subscriber is not actively trading at the moment, one update pull per minute should be more than enough. Since the web service we based our wrapper on seem to be located in San Jose, Ca., we decided to test both from Troms and from Cornell University, NY, to investigate the impact of long distance latency. Detailed test results are shown in the following table:

| **Site** | **Threshold** | **Pulls** | **TP** | **Ratio** |
|---|---|---|---|---|
| Tromsø | 1.0 % | 3300 | 15 | 0.0045 |
| Tromsø | 0.5 % | 2880 | 64 | 0.0230 |
| Cornell | 0.5 % | 2910 | 65 | 0.0220 |

The first test was run with the highest movement threshold, and yielded fewer true positives. Each stock quote was pulled in approximately 3000 times each trading day. Of these, the amount of true positives ranged from 15 (0.5%) to 65 (2.3%).As expected, the lowest movement threshold yielded significantly more true positives. Still, they were only about 2% of the total number of updates. The latency measurements are omitted since they did not pose a significant cost.

In a very realistic usage scenario, we have showed that up to 98% of the updates were false positives, i.e. wasted traffic. Our results suggest that a more push-based architecture is very suitable for this type of applications. Thus, complementing traditional pull-based Web services with a push-based interface can significantly reduce the amount of unnecessary traffic on the Internet.

# 5 Discussion

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

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

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

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.

Existing pub/sub systems like, for instance, Siena[1], Gryphon[6], and Pastry[3] demonstrate that push-based approaches solve some of these scaling issues. 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 pub/sub systems are that they are proprietary and have coarse granularity. This might result in far too much data being sent over the wire, adding to the scaling problem.

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 push-based ones. Next, we configure and build distribution and fusion networks between these web publishers and remote users. This overlay structure is transparent for the user, giving the impression of a personal overlay network system (PONS) 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. This PONS can also be combined with existing pub/sub systems to scale data distribution.

The recent developments in Semantic Web Services display a trend where remote services and data are given very expressive interfaces. Attaching structured metadata to a service interface 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.

# 6 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.

## 6.1 Acknowledgements

We would like to thank the other members of the WAIF research group, especially Dmitrii Zagorodnov, and the anonymous referees for their insightful comments on earlier versions of the paper.

# References

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

[2] 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.

[3] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.

[4] 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.

[5] K. Sivashanmugam, K. Verma, A. P. Sheth, and J. A. Miller. Adding semantics to web services standards. In L.-J. Zhang, editor, *ICWS*, pages 395–401. CSREA Press, 2003.

[6] R. E. Strom, G. Banavar, T. D. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. C. Sturman, and M. Ward. Gryphon: An information flow based approach to message brokering. *CoRR*, cs.DC/9810019, 1998.

[7] G. Wiederhold. Mediation in information systems. *ACM Comput. Surv.*, 27(2):265–267, 1995.