Generalizing the “Like” button: empowering websites with monitoring capabilities

Jon Iturrioz  
Computer Science Faculty,  
University of the Basque Country  
Pº Manuel Lardizabal, 1  
20018 San Sebastián, Spain  
jon.iturrioz@ehu.es

Iker Azpeitia  
Computer Science Faculty,  
University of the Basque Country  
Pº Manuel Lardizabal, 1  
20018 San Sebastián, Spain  
iker.azpeitia@ehu.es

Oscar Díaz  
Computer Science Faculty,  
University of the Basque Country  
Pº Manuel Lardizabal, 1  
20018 San Sebastián, Spain  
oscar.diaz@ehu.es

ABSTRACT
Increasingly, a user’s action in a website might have an impact in other websites. The Like and ShareThis buttons are forerunners of this tendency whereby websites strive to influence and be influenced by the actions of their users in the websphere. The term Web Radar is coined to denote software that serves to impact a website (the host) from what is happening somewhere else in the websphere (i.e. the target). Current approaches provided limited expressivity in either the reactions (e.g. the Like button is limited to write entries on the user’s wall in Facebook), or the range of participating sites (pre-set in the Radar platform, e.g. Ifttt). We believe supporting Radars as configurable services might account for more domain-specific Radars, i.e. Radars where the monitoring sites, the tracking conditions and the reactions are not fixed by the Radar platform but rather determined by the Radar host. This vision is confronted with three main challenges: API heterogeneity, scalability and technical complexities. We address these matters in RadarThis, a service that permits webmasters to set Web Radars for their websites. We capitalize on YQL to hide API complexity, and use trigger-like syntax to specify custom radar strategies. A case study is presented using the website Instaper as the Radar host.

ACM Classification Keywords
H.3.5. Online Information Services: Data sharing

General Terms
Experimentation

Authors Keywords
Web2.0, YQL, Database triggers, Web Radar

1. INTRODUCTION

Though autonomous, websites are not totally isolated entities. A kind of “Web tightness” exists whereby websites influence and are influenced by other websites. Such tightness manifest itself in different forms: synchronization (for a user, changes on her Facebook profile picture might need to be propagated to her Twitter account), notification (when a new book is added to Kindle Top 100 Free eBooks, send me an email), resource sharing (archive your tweets in Evernote), automate social relationships (e.g. automatically greet new twitter follower of my account), etc. This tightness is commonly enforced by users in an ad hoc way. An increasing crop of mechanisms facilitate this endeavor. The Like1 and ShareThis2 buttons are two forerunners. By clicking on the Like button in a website, you are impacting your account in a different site, e.g. Facebook. When clicking ShareThis in Flickr, you might be bookmarking the URL of the current picture in your Delicious account. These attempts require the explicit intervention of users who click the button that enacts the impact. However, other scenarios (e.g. synchronization, notification, etc) might be conducted in a routinary way (e.g. every time I upload a presentation in Slideshare, communicate it to my followers in Twitter). These scenarios are better served through a trigger-like mechanisms as those provided by Ifttt [1] or AtomateIt [13]. The advantage here is that user intervention is not required but the mechanism takes charge for monitoring the happening (e.g. uploading a presentation) and conducting the reaction (e.g. tweet this happening to my followers).

In general, we term “Web Radar” (hereafter just Radar) a software that serves to impact a website (i.e. the host) from what is happening somewhere else in the websphere (i.e. the target).

So far, most of the trigger-like solutions rest on a separated website to support the Radar (e.g. Ifttt, AtomateIt). This facilitates the business model of these platforms where users have to register to the Radar platforms besides having their own accounts in the monitored and tracked websites. On the upside, this accounts for a centralized management where all radars sit in the same location. On the downside, the expressiveness of the Radars (i.e. what can you monitor, and how can you react) is pre-set by the platform which does not contemplate user or website specificities. They are one-size-fits-all approaches where the challenge rests on empowering end users to define their own triggers. The success

1http://developers.facebook.com/docs/reference/plugins/like/  
2http://sharethis.com/
of Ifttt evidences the need for synchronization and readiness in the websphere. Users have so demonstrated by defining over 87541 kind of triggers3 in Ifttt.

But the interest is not only from the user side. Our premise is that Radar behaviour is also of key importance for Web2.0 sites. After all, a Web2.0 motto is “data is the next Intel Inside” [11] which denotes the importance of data (versus functionality) on this kind of systems. So, any mechanism that fuels data provisioning becomes paramount for this kind of websites. Radars facilitate data provisioning by providing a way to feed the website, other than directly accessing the website. Facebook soon realized this fact, and facilitates the Like button. By embedding the Like button, target websites make feeding Facebook a click away without the need to directly accessing Facebook. This rocketed Facebook contributions. Unlike platform-based Radars like Ifttt, here the reaction is Facebook specific: it is up to Facebook to interpret what clicking the Like button means. That is, happenings and reactions are Facebook specific.

We believe this kind of exogenous feeding can benefit a large range of Web2.0 websites. Unfortunately, two main stumbling blocks arise. First, most of websites might lack Facebook’s “persuasion” to convince other websites (the targets) to embed their buttons. This corners the push solution where users directly notify the happening of interest in the targets through clicking. Alternatively, pull approaches resort to periodic polling the targets to detect happenings of interests. This is what Ifttt-like platforms do. The issue is the upfront investment required for websites to replicate this pull architecture as part of their offerings. The good news is that Radar functionality is amenable to be offered as a service where website-specific semantics can be externalized through call-backs. This paper investigates the feasibility of this vision by describing the RadarThis service. The challenges faced by RadarThis, and main contributions of this work include:

- a definition of web radars, as an intuitive metaphor that shelter hosts from the target heterogeneity (Section 4)
- appropriate credential management (Section 5)
- scalable architecture (Section 6)

Next sections describe the RadarThis service along these challenges. But first, we phrase our efforts within related work.

2. RELATED WORK

We characterize the design space of Radars along the following dimensions (see Table 1):

- reaction expressiveness. Which kind of reactions are permitted? The impact can be limited to a “fixed” action; sticking to some “pre-set” actions; or be “free” to more sophisticated reaction description,
- target scope. Which kind of targets are supported? The range of target websites can be “onlyOne”, “pre-defined” from a fixed list, intentionally “limited” by exhibiting some technical characteristic that permits pulling (e.g. RSS-enabled) or be “open” to any collaborative site (this is for push approaches),
- transparency. Is the Radar functionality transparent to the participant websites? Are websites aware of the Radar functionality? Values include “hostAware”, “targetAware” and finally, “transparent” if the Radar is defined without the collaboration of the participating websites. The latter is supported as an autonomous Radar platform.

Like and ShareThis illustrate the collaborative approach. Here, the target collaborates to notify happenings of interest (i.e. “targetAware”). By clicking the Like button (i.e. “push” approach), the user impacts a fixed host (the user’s Facebook account) in a set way (i.e. introducing the like message in the user’s wall). Alternatively, the ShareThis button permits to select from a preset list of hosts where the impact is also fixed. The Radar is opened to any target and it is supported as a browser extension or it can be embedded in the target website by the administrator. In both cases the data delivery follows a push approach and the reaction expressiveness is fixed. Next we explore the “pull” data delivery radars.

Mendes et al. [9] illustrates single-site Radars where the target being tracked is limited to a single website, Twitter. The Radar is supported as a separated website (i.e. “transparent”). Mendes et al. focus on tracking tweets. Rather than using hashtags, the selection criteria to determine tweets of interest is described through an RDF query. This implies that tweets need first to be automatically annotated along an ontology. Next, the RDF query is addressed along the so-annotated tweets. When an interesting tweet is spot, the reaction is fixed and limited to a notification to the user. The architecture relies on Push hubs, where the published data is in a RSS format. On the other hand, ReactiveTags [8] tracks tagging sites’ annotations with specific tags (the so-called reactive tags). Unlike Mendes et al., ReactiveTags tracks multiple target sites. Targeting end users, ReactiveTags resorts to SIOC [4] to lower the specification bar. Sites, and the selection and impact criteria are defined in terms of SIOC Items queries and SIOC Items occurrences, respectively. The data is tracked on the target through the API mechanism.

AtomateIt [13] focuses on RSS providers. Queries can be run against a pre-loaded base of RSS feeds. If the query is satisfied, users can select from a “pre-set” collection of reactions (e.g. publish a tweet). Extending either the RSS providers (i.e. the targets) or the action list involves con-

<table>
<thead>
<tr>
<th>Web Radar</th>
<th>Reaction expressiveness</th>
<th>Target scope</th>
<th>Data delivery</th>
<th>Transparency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Like button</td>
<td>fixed</td>
<td>open</td>
<td>push</td>
<td>targetAware</td>
</tr>
<tr>
<td>ShareThis</td>
<td>fixed</td>
<td>open</td>
<td>push</td>
<td>targetAware</td>
</tr>
<tr>
<td>Mendes et al.</td>
<td>fixed</td>
<td>onlyOne</td>
<td>pull</td>
<td>transparent</td>
</tr>
<tr>
<td>ReactionTags</td>
<td>pre-set</td>
<td>limited (API)</td>
<td>pull</td>
<td>transparent</td>
</tr>
<tr>
<td>AtomateIt</td>
<td>pre-set</td>
<td>limited (RSS)</td>
<td>pull</td>
<td>transparent</td>
</tr>
<tr>
<td>Ifttt</td>
<td>pre-set</td>
<td>predefined</td>
<td>pull</td>
<td>transparent</td>
</tr>
<tr>
<td>RadarThis</td>
<td>free</td>
<td>limited (ODT)</td>
<td>pull</td>
<td>hostAware</td>
</tr>
</tbody>
</table>

Table 1: Design space of Web Radars.
siderable programming. The Radar is supported as a separated website. Finally, Ifttt illustrates the “predefined” option where the target websites are fixed. Users create triggers connecting two services. Ifttt presets the way the information is transformed between both services. The Radar is supported as a separated website.

3. A CONCEPTUAL MODEL FOR RADARS

For the purpose of this work, a Radar is a software that serves to impact a website (the host) from what is happening somewhere else in the websphere (i.e. the target). Figure 1 depicts the main conceptual entities: a radar monitors one or more targets through alert points that conduct some reactions when the target meets some circumstances for a given website. Websites can play the role of the host or the target. A host monitors targets by providing the means for its users to dynamically define alert points upon the tracked targets. In this case, the host can set radar quadrants as a way to constrain the targets to be monitored. Finally, a host contains resources which are to be impacted by the alert point’s reaction. Next, we illustrate this model through an example.

Let’s consider Instapaper\(^4\) play the role of the host. Instapaper is a neat tool to save web pages for reading later. Broadly, it is a bookmarking tool where the Web resource is a bookmark. Instapaper’s webmaster conducts a survey to see which other websites are amenable to become feeders of Instapaper’s resources, i.e. bookmarks. This defines the quadrant to constraint the potential target websites. Let’s Arxiv\(^5\) be one such target website. Arxiv is a web archive for electronic preprints of scientific papers. All it rests is to define a radar that monitors Arxiv to provision Instapaper. This implies introducing an alertPoint that defines the circumstances that makes Arxiv’s preprints worth provisioning Instapaper. These circumstances are user specific. Therefore, Instapaper offers a GUI for its users to indicate those criteria. Figure 2 shows an example where the user indicates as monitoring criteria the following: ACM as the publisher and “internet” in the title.

This scenario also illustrates the different stages in Radar definition and enactment (see Figure 3):

- at design time, the host analyses the best radar strategy for its purpose. This includes: (1) setting the quadrant (i.e. the range of website amenable to monitoring), (2) defining how its resources can be eventu-

\(^4\)http://www.instapaper.com/
\(^5\)http://arxiv.org/

Figure 1: A conceptual model for Radars.

![Figure 1: A conceptual model for Radars.](image)

Figure 2: The host: RadarThis offers an editor for AlerPoint definition. Circumstances are domain specific. In this case, publication characteristics are used to describe the resources worth following.

- at definition time, the users specify the alert point. Such definition is complemented by the host with credentials (see section 4.4), which account for a secure environment where only permitted web resources can be consulted/modified.

- at triggering time, the radar detects whether circumstances are being met, and if so, conducts the impact on the host.

The challenges ahead, i.e. an intuitive metaphor for Radar definition, credential management, and a scalable architecture, are next addressed in Section 4, 5 and 6, respectively.

4. DEFINING WEB RADARS

Radars can potentially be defined in any website keeping Web resources. Web resources are generally handled through open APIs. The heterogeneity in the schemas and signature of open APIs is a known fact \(^3\). It is not realistic to assume the host’s webmaster would face this potential heterogeneity at the risk of a high upfront. This is not new. Mashup platforms are already striving to such heterogeneity in their attempt to remix or reshape website resources. The Yahoo Query Language (YQL) \(^10\) is being provided by Yahoo to face this specific matter. The “metaphor” is to look at websites as table providers: each API method is encapsulated with a table view where input parameters and (some) output parameters are regarded as table attributes, and hence, being able to be used for selections. We extend this metaphor by describing Radars as database-like triggers over these tables. To avoid confusion with database triggers, we will term “trigger” those artifacts that monitor and react in a distributed web-based setting.

As an example, consider Arxiv and Instapaper as hosting tables of “e-prints” and “bookmarks”, respectively. You can be interested in saving into your Instapaper account, new stored e-prints that contain the word “internet” in the title. The trigger expression will look something like:

```
ON INSERT a new e-print LIKE “internet” DO
    INSERT a bookmark(e-print permalink) INTO my Instapaper account.
```

This section addresses the definition of tryggers. First, we outline the YQL platform.
4.1 A brief on YQL

A tenant of current Web development is the release of data through accessible APIs. APIs permit third parties to get access to the data so that it can then be rendered, remixed or reshaped along the consumer wishes. The problem is that APIs are as heterogeneous as the applications they support. YQL is all about simplifying API access. YQL builds on previous experience on Yahoo Pipes\(^6\), but rather than providing a graphical, client-based approach, move to the server all the burdens of scalability and API heterogeneity while providing an SQL-like interface. The “metaphor” is to look at websites as table providers: each API method is encapsulated with a table view where input parameters and (some) output parameters are regarded as table attributes, and hence, being able to be used for selections. The following select statement illustrates a YQL’s SELECT to access the Slideshare API but now conceived as the table slideshare.slideshows:

```
SELECT * FROM slideshare.slideshows WHERE user IN ("cheilmann", "ydn", "jonathantrevor")
```

To realize this metaphor, YQL offers both a service and a language. So far, YQL offers SELECT, INSERT, UPDATE and DELETE statements that, behind the curtains, invoke the corresponding API methods. YQL handles for you parameter passing, credential handling or implicit iterations (for joins). The mapping from the YQL syntax to API requests is achieved through the so-called “Open Data Tables” (ODT). ODTs hold all the intricacies of the underlying APIs. Figure 4 shows the slideshare.slideshows ODT.\(^7\) The document’s main tags are <meta> and <bindings>. The former contains descriptive information about the ODT (3-12), such as author, description, documentation link or YQL example query (10). The bindings (13-23) indicate how SQL operations are realized as API operations. An entry exists for each operation (e.g. <select>, <insert>, <delete>, <update>). For the sample case, the ODT only supports SELECT by providing the <url> pattern to invoke (16) and the description of the possible YQL statement input field (18-21). Bracelets parts (e.g. \{user\}) account for variables to be instantiated when SELECT is enacted (e.g. user will refer to “cheilmann”, “ydn” or “jonathantrevor”). Once the API method is invoked, the “itemPath” attribute (14) specifies the path where results are located in the method’s response while the “produces” attribute refers to the format of the query result (XML, JSON).

4.2 Trygger definition

YQL provides atomic database operations on ODT tables. We extend this vision by introducing “tryggers”, i.e. event-condition-action rules \([12]\) that act upon tables pertaining to different sites. Rules permit to enact small pieces of code (the DO part) when some update operation is conducted.

\(^6\)http://pipes.yahoo.com

\(^7\)Available at http://www.datatables.org/slideshare/slideshare.slideshows.xml
(the ON part) provided some condition is met (the WHEN part). Akin to the YQL metaphor, we use SQL-like syntax for trigger specification (see Figure 5).

Triggers are based on YQL’s ODT tables. The whole idea is to tap into the existing ODT repository. Indeed, most of the ODT tables used throughout the paper were provided by the YQL community. Even so, YQL permits ODTs external to the system to be operated upon through the USE clause. This clause just provides the URL that holds the ODT definition. From then on, no difference exists in handling ODTs kept outside YQL boundaries.

The event. In SQL, events are associated with the insertion, deletion or update of database tuples. Since YQL’s ODT handle both insertions and deletions, it could be potentially possible to define as events the happening of these operations. Notice however, that YQL is just another client built on top of somewhere else’s APIs. These APIs can be accessed by other clients, and hence, “the tables” can be changed by programs other than YQL. Monitoring only YQL events would make these other changes go unnoticed. There-fore, the only way to be aware of insertions (YQL mediated or not) is by change monitoring. Hence, the trigger event is “after select changes” which is risen when two consecutive pollings incrementally differ. This implies that triggering ODT tables should account for selections (i.e. the ODT file should contain a <select> tag). No “statement” triggers are supported. Only “for each row”, i.e. each new tuple causes the enactment of the trigger. The event payload refers to this new tuple, kept in the system variable NEW.

The condition. It checks the event payload, i.e. the NEW variable.

The action. SQL actions stand for procedures that are stored in the database. Procedures can be atomic (i.e. a single statement) or full-fledged programs (described using e.g. PL/SQL or Java). Back to the triggers, actions can be either atomic or composed. Atomic actions comprise a single statement, specifically, any “insert/update/delete” statement defined in a YQL’s ODT table. Action parameters can refer to either constants or NEW. However, single-statement actions fall short in numerous scenarios: when payload parameters need some processing before being consumed; when data kept in other ODT tables is to be retrieved; when more than one ODT table needs to be changed. This leads to composed actions, i.e. programs. Akin to YQL, the programming language is JavaScript. Trigger’s JavaScript permits both YQL actions and access to the system variable NEW. Next section illustrates this syntax throughout different examples.

4.3 Trygger examples

From Arxiv to Instapaper. Arxiv is an archive for electronic preprints of scientific papers. Instapaper is a neat tool to save web pages for reading later. The trygger will monitor new preprints in Arxiv published in any “ACM” journal that contains “Internet” in the title. On detecting such preprints, the trigger creates a new entry in the Instapaper host. This example involves two ODT tables, namely arxiv.search and instapaper.unread. Being the triggering ODT, the former necessarily needs to hold a <select> tag, together with the columns that stand for the query parameters (i.e. “search_query”). The outcome is an ATOM file kept in “entry”. Figure 6-A provides the code.

On adding a new preprint in arxiv.search, the trygger checks whether the query expression is “ti:Internet” (an Arxiv syntax to recover documents with “internet” in their titles), and the “entry” document holds a <journal_ref> tag whose content includes ACM. The interesting part comes with the trygger’s action. The action constructs a tuple out of NEW. Since “entry” holds an XML document, its content can be obtained using E4X dot expressions. To avoid clumsy expressions, an XMLNAMESPACES declaration is introduced (also available in SQL and YQL).

From Twitter to Facebook. Andy Young developed SelectiveTwitter11 as a Facebook App whereby tweets ending in the hashtag #fb are directly propagated to your Facebook status. This application can be conceptually described as a sharing rule: “on introducing a tweet that contains #fb, do accordingly update my Facebook wall”. Facebook developers were forced to provide a generic hashtag (i.e. #fb) to accommodate no matter the user. However, users can certainly be interested in monitoring domain-specific hashtags. For instance, when in a conference (identified through a hashtag #SAC2014), you could be interested in tracking #SAC2014-containing tweets from the Twitter accounts of some attendees. Unlike the “selectivetwitter” case, this approach is more natural (i.e. you are not forcing your col-

---

8Action statements are enacted as independent entities. So far, tryggers do not support transaction-like mechanisms.

9ODTs used throughout can be consulted at https://github.com/yql/yql-tables.

10ECMAscript for XML (E4X) is a programming language extension that adds native XML support to ECMAScript. http://en.wikipedia.org/wiki/ECMAScript_for_XML

11http://www.facebook.com/selectivetwitter. To date, this enhancement obtains 3.6 out of 5 review summary based on 2,666 reviews.
leagues to introduce the #fb hashtag but a tag that might already exist) and domain-specific (i.e. you can attach distinct sharing behaviour to different hashtags). Figure 6-B provides a trygger that supports domain-specific hashtag tracking. We resort to two ODT tables, “twitter.search” and “facebook.setStatus” that support selections and insertions on Twitter and Facebook, respectively.

This rule is triggered if the delta of two consecutive selects on twitter.search is not empty. If so, the condition checks two filters. First, the “q” column should hold “from:alice” or “from:bob” (“q” stands for the username of the Twitter account as named by the ODT provider). Second, the “text” column that holds the tweet message which should contain the string #SAC2014. If satisfied, the trygger’s action results in a new tuple being inserted into the facebook.setStatus table. Columns of the newly created tuple (i.e. status) can be provided as constants or be obtained from NEW.

4.4 Handling credentials

A trygger action aims at inserting some web resource in the account of a host’s user. This requires appropriate credentials. When the user consents the creation of a trygger, the host should authorize RadarThis to access this account on behalf of the user. However, directly handling out credentials to RadarThis can put some users off. YQL has a feature that enables to store environment files keeping parameters to be consulted when YQL operations are enacted. When the host creates a trygger, it also creates a YQL environment file with the user credentials. YQL returns a token. This YQL token is used by RadarThis when the trygger action is enacted. In this way, host users do not disclose their credentials to RadarThis. In fact, the username and password parameters shown in the figure 6 do not really appear in the trygger and the credentials are passed to RadarThis through a YQL Token that reference to a YQL environment file where the credentials reside.

5. CREDENTIAL MANAGEMENT

RadarThis provides a set of REST end-points for trygger definition and management. Similar to other open APIs, consumer applications should first get both a publisher key and a secret access key from RadarThis. API requests require both the publisher key and the associated secret access key. That said, we will not included such keys in the next examples for keeping the API requests as clean as possible. The API services account for two main uses: authorization and trygger management. Next subsections delve into the details.

5.1 Authorization Service

RadarThis follows OAuth mechanisms whereby consumers can be authenticated using their Facebook or Twitter credentials. The goal is to obtain a user authorization token for a RadarThis user without disclosing users’ credentials (i.e. login/password). For instance, Instapaper defines tryggers on behalf of its users. When a user wants to define a trygger, Instapaper starts by authenticating the user through e.g. his Twitter account:


The redirect_url indicates the Instapaper listener that will receive the user authorization token. The response of this method contains an authentication_url parameter for Instapaper to redirect its users to Twitter:

{"status":"SUCCESS","authentication_url":"https://twitter.com/oauth/authorize?oauth_token=GoNtXbSlOzGvkJ7FLTvWSV0EMY0G"}

Once the Instapaper user is being authenticated by Twitter, the user is redirected back to the Instapaper redirect_url (i.e. http://instapaper.com/OAuth.php) which is now extended with the user authorization token:

http://instapaper.com/OAuth.com?token=a963821d260a9eeec16553314a

Instapaper uses the token to mediate with RadarThis on behalf of the user.

5.2 RadarThis managing API

Table 2 defines the RadarThis end-point URLs for trygger management. All requests must have at least three parameters (omitted in the table): two application parameters, publisher key and secret access key, and the user authorization token. The creation of a trygger (i.e. add.php) needs as input parameters the trygger definition and a YQL-token that permits RadarThis to transparently get login and passwords using YQL environment tables (see subsection 4.4). This method returns the status of the operation (e.g. SUCCESS, FAIL) and a tryggerID if success.
6. SCALABLE ARCHITECTURE

RadarThis is an OAuth [7] server that provides an API for programatically requesting the definition of “tryggers”. It follows a loosely coupled architecture for distributed trigger management. The implementation relies on existing standards and protocols, and it is entirely HTTP-based. RadarThis services take the form of REST-based requests. The architecture of RadarThis revolves around three main components (see Figure 7):

1. **The RadarThis Service.** As any other open API, consumers of RadarThis services require the previous obtention of some API keys. Such keys can be obtained from the RadarThis site (www.radarthis.org) and should be provided by the consumer at the time the service is requested (e.g. creating a trigger). The obtention of the API keys requires consumers to have an account at RadarThis. To this end, RadarThis follows OAuth mechanisms whereby consumers can be authenticated using their Facebook or Twitter credentials.

2. **The MARS Engine.** Tryggers are realized as ECA rules. The MARS framework is used as the rule engine [2]. Its functions include: ECA rule registration (ECA engine), event occurrence signaling (Event Engine) and rule firing (i.e. rule’s action enactment) (Action Engine). MARS provides the general framework for supporting ECA rules in a distributed environment. However, both events and actions are domain specific. Their realization and definition need to be externalized from MARS into so-called brokers. RadarThis introduces two brokers.

3. **The RadarThis Brokers.** The Event Broker monitors and generates events. So far, only one type of event is handled: changes in ODT tables. Hence, the way to detect a change is by periodically polling the underlying website. For this purpose, we use the PubSubHubbub protocol [5]. On receiving the signal from the PubSubHubbub hub, the Event Broker generates an event occurrence for each NEW change. On event signaling, the MARS engine triggers the associated rules. Rule triggering is conducted through the Action Broker.

The event detection process is described in continuous line on the Figure 7. The PubSubHubbub hub periodically polls the triggering website by issuing the YQL SELECT query described in a trygger’s event part (1.1-1.3). When a user on the target site inserts a new item (2.1) the dotted line data flow starts. The hub detects an incremental change at the next polling time and it sends to the RadarThis Event Broker the new event occurrence (2.2). The broker acts as a mediator and it signals this happening to the Event Engine (2.3). The Event Engine forwards the occurrence to the ECA Engine (2.4) which retrieves the tryggers that match the type of the new event occurrence and triggers the action through the Action Engine (2.5). The Action Engine forwards the petition to the RadarThis Action Broker (2.6) that knows how to handle the statements of the trygger’s action. Finally the YQL service processes the trygger’s action (2.7-2.9).

![Figure 7: Architecture. Throughput is expressed as calls per minute (c/m).](image)

### 6.1 An evaluation of scalability

Being a pull approach, RadarThis throughput is mainly limited by the polling frequency. The polling service rests on two external services: YQL Service and the API of the target website whose content is being polled. Next, we elaborate on the limitations imposed by these APIs and their impact on RadarThis. Figure 7 depicts this situation as funnels that limit the throughput of RadarThis.

In order to control the use of Open APIs by 3rd party applications, providers impose the so-called rate limit to regulate the frequency of invocation. This limit admits different thresholds based on the authorization level of the caller: authorized user (OAuth), the application making the call (identified through an API key) or the machine through which the call is issued (identified by its IP). In addition, the limit can be set at different granularity levels: a single method, a set of methods grouped in an endpoint or the whole API. The table 3 summarizes the rate limits for some popular Open APIs as publicized in their websites.

Based on these figures, the host creates and manages their API keys at trygger creation time (i.e. its YQL queries). RadarThis invokes the services on behalf of the hosts using its API keys. RadarThis controls the invocation frequency of each event taking into account the API key of the trygger and the API rate limit of each website. As an example, if a host creates 300 tryggers whose events are based on Twitter using the same API key, RadarThis will notice that a maximum of a 300*2 seconds (i.e. 10 minutes) will be needed to signal a new event occurrences. That is, the delay time increases linearly based on the number of tryggers that
Table 3: Web site API rates.

<table>
<thead>
<tr>
<th>Website</th>
<th>Authorized Caller</th>
<th>Regulated Resource</th>
<th>Rate Limit</th>
<th>1 call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arxiv</td>
<td>application</td>
<td>all methods</td>
<td>1/3</td>
<td>3</td>
</tr>
<tr>
<td>Ebay</td>
<td>user</td>
<td>all methods</td>
<td>20000/day</td>
<td>1.15</td>
</tr>
<tr>
<td>Facebook</td>
<td>user</td>
<td>all methods</td>
<td>1/12</td>
<td>1</td>
</tr>
<tr>
<td>Flickr</td>
<td>user</td>
<td>application</td>
<td>1000/60</td>
<td>3.6</td>
</tr>
<tr>
<td>FourSquare</td>
<td>user</td>
<td>group of endpoints</td>
<td>500/60</td>
<td>7.2</td>
</tr>
<tr>
<td>Twitter</td>
<td>user</td>
<td>each method</td>
<td>15/15</td>
<td>1/2</td>
</tr>
<tr>
<td>Vimeo</td>
<td>user</td>
<td>all users</td>
<td>1500/5</td>
<td>0.7</td>
</tr>
<tr>
<td>YQL service</td>
<td>IP</td>
<td>all users</td>
<td>2000/60</td>
<td>1.8</td>
</tr>
</tbody>
</table>

7. CONCLUSION

The departing premise of this work is that websites 2.0 would benefit from radars that act as purveyors of content for the website. The idiosyncrasies of each website advice for radars to be site-specific. This paper proposes an ubiquitous, platform-agnostic Web-based radar framework for helping website masters.

8. REFERENCES


\[ \text{Entrants} = \left\lfloor \frac{\text{50000}}{\text{20000} \times \frac{1}{15}} \right\rfloor = \left\lfloor 3.5 \right\rfloor = 4 \]

We assume that the API rate limit is proportionally distributed along time, i.e., if in 60 minutes the rate is 20000, in 1 minute the rate is 333.