R-Event: A RESTful Web Service Framework for Building Event-Driven Web

Li Li
Avaya Labs Research
Avaya Inc.
Basking Ridge, New Jersey, USA
lli5@avaya.com

Wu Chou
Avaya Labs Research
Avaya Inc.
Basking Ridge, New Jersey, USA
wuchou@avaya.com

***Abstract*— As the Web is becoming a communication and collaboration platform, there is an acute need for an infrastructure to disseminate real-time events over the Web. However, such infrastructure is still seriously lacking as conventional distributed event-based systems are not designed for the Web. To address this issue, we develop a RESTful web service framework, R-Event. It represents and encapsulates the structural elements of Event-Driven Architecture (EDA) into the infrastructure of REST (Representational State Transfer), the architectural style that underlies the Web. Our approach leads to an event-driven web consisting of 4 layers of RESTful web services. The R-Event framework implements the layers that are pivotal to the event-driven web. The core component of this framework is federated topic hubs that provide services for notification publication, subscription, delivery, tracking, and linking. The advantages and applications of this approach are presented and discussed, including the important features of addressability, connectedness, dynamic topology, robustness, scalability, and efficient notifications. A prototype system for presence driven collaboration is developed and the preliminary performance tests show that the proposed approach is feasible and advantageous.**

Keywords - Web service; REST; Topic Hubs, Event-driven; EDA.

I. INTRODUCTION

The Web has undergone a rapid evolution from an informational space of static documents to a space of dynamic communication and collaboration. However, to some large extent, the Web is still a reactive informational space and information sharing is still mostly pull based. Consequently, there could be significant latency between the availability of a piece of information and the use of that information. This model of information sharing has worked well for the Web, but is becoming increasingly insufficient for new emerging applications.

In the early days of Web, changes to web content were infrequent and a user could rely on web portals, private bookmarks, or search engines to find information. However, in the era of Web 2.0, dynamic and user generated contents become increasingly popular, such as blogs, wikis, mashups, folksonomies, social networks, etc. People are demanding timely and almost instant availability of these dynamic contents, and interactive use of this information, without being overwhelmed by the information overload. This drives the Web from an informational space towards a communication and collaboration oriented environment that affects both consumer and enterprise application spaces. These new trends demand an event-driven web in which information sharing is driven by events to support the dynamic and near real-time information exchange.

Despite many existing event notification systems developed over the years, infrastructures and technologies for such an event-driven web are still seriously lacking. As the architectures, protocols, and programming languages of the existing event notification systems are developed outside of the web, there is an acute need for a unifying framework that can provide a seamless integration of these notification systems with the infrastructure of web and web based services.

For such a unifying framework, we lay our foundation on Event-Driven Architecture (EDA) [12], in which information is modeled as asynchronous events that are pushed to the interested parties as they occur. By synchronizing the states of the communicating parties through events, EDA makes real-time communication and collaboration possible. Moreover, EDA is a natural fit for the Web as both do not assume any centralized control logic. However, the current web protocols are based on client-server architecture which does not readily support EDA. Even though some recent standards and industrial efforts, such as Atom [4][5], Server-Sent Events [9], Web Sockets [10] and HTML 5 [8], introduce the notion of feed and event, they are aimed at the web browsers and human users. As far as we know, there is no research work to combine EDA and REST to enable and support federated event-driven web services.

Because EDA is an abstract architecture whereas REST has concrete protocol (HTTP), we need to first resolve how to project the elements of EDA to those entities of REST [1][2] in a consistent framework. In our approach, we found that many important features and problems in conventional event notification systems can be established and resolved efficiently in our REST based framework. For instance, the uniform interface, connectedness, and addressability of REST can apply and facilitate the discovery of notification web services. The idempotent operations and statelessness of REST can add robustness and scalability to notification web services. Furthermore, projecting EDA to REST can facilitate transformation of conventional notification systems into RESTful web services, because EDA can be viewed as a generalization of the architectural elements in those notification systems.

In our approach, the key concepts of EDA are projected into 4 layers of an event-driven web. Each layer consists of interconnected resources that collectively provide RESTful web services for applications. This projection leads to our RESTful web service architecture, R-Event that defines the notification web services for such event-driven web. To maximize the reuse and interoperability, these layers are weaved and combined through RESTful web services composition and linking. A prototype event-driven web consisting of topic hubs and topic webs is implemented to demonstrate the feasibility and advantages of this approach.

The rest of the paper is organized as follows. Section II introduces the background and related work. Section III describes the model of event-driven web. Section IV introduces the R-Event framework and its components, e.g. topic hub and topic web. Section V summarizes the advantages of this approach. Section VI is dedicated to a prototype implementation and experimental study results. Findings of this paper are summarized in Section VII.

II. RELATED WORK

REST stands for REpresentational State Transfer, the architecture style underlying the Web as described in [1] [2] [3]. The fundamental concept of REST is a resource. REST promotes the following architectural choices: 1) Addressability: each resource can be addressed by URI. 2) Connectedness: resources are linked to provide navigations. 3) Uniform Interface: all resources support a subset of the uniform interface, namely GET, PUT, DELETE and POST. GET is safe and idempotent, while PUT and DELETE are idempotent. Idempotent operations can be resubmitted if failed without corrupting resource states. 4) Statelessness: all requests to a resource contain all of information necessary to process the requests, and the servers do not need to keep any context about the requests. Stateless servers are robust and easy to scale. 5) Layering: intermediate proxies between clients and servers can be used to cache data for efficiency.

RSS [6] and Atom [4] are two data formats that describe the published resources (feeds), including news, blogs, wikis, whose contents are updated by the content providers. The content providers syndicate the feeds on their web pages for the feed readers which fetch the updates by periodically polling the feeds. However, such polling is very inefficient in general, because the timing of the updates is unpredictable. Polling too frequently may waste a lot of network bandwidth, when there is no update. On the other hand, polling too infrequently may miss some important updates and incur delay on information processing.

To address the inefficiency of poll style feed delivery, Google developed a topic based subscription protocol called PubSubHubbub [22]. In this protocol, a hub web server acts as a broker between feed publishers and subscribers. A feed publisher indicates in the feed document (Atom or RSS) its hub URL, to which a subscriber (a web server) can registers the callback URL. Whenever there is an update, a feed publisher notifies its hub which then fetches the feed and multicasts (push) it to the registered subscribers. While this protocol enables more efficient push style feed updates, it does not describe how hubs can be federated to provide a global feed update service across different web sites. The protocol defines the unsubscribe operation by overloading POST which should have been DELETE. Also the subscriptions are not modeled as addressable resources.

Many techniques have been developed over the years to address the asynchronous event delivery to the web browsers, such as Ajax, Pushlet [7], and most recently Server-Sent Events [9] and Web Sockets [10]. However, these techniques are not applicable to federated notification services where server to server relations and communication protocols are needed.

In software engineering, Publisher-Subscriber [15] or Observer [11] is a well-known software design pattern for keeping the states of cooperative components or systems synchronized by event propagation. It is widely used in event-driven programming for GUI applications. This pattern has also been standardized in several industrial efforts for distributed computing, including Java Message Service (JMS) [24], CORBA Event Service [25], CORBA Notification Service [26], which are not based on web services.

Recently, two event notification web services standards, WS-Eventing [18] and WS-Notification [19][20] are developed. However, these standards are not based on REST. Instead they are based on WSDL [27] and SOAP [28], which are messaging protocols alternative to REST [1]. WS-Topic [21] is an industrial standard to define a topic-based formalism for organizing events. However, these topics are not REST resources but are XML elements in some documents.

Recently, much attention has been given to Event-Driven Architecture (EDA) [12][16] and its interaction with Service-Oriented Architecture (SOA) [17] to enable agile and responsive business processes within enterprises. The fundamental ingredients of EDA are the following actors: event publishers that generate events, event listeners that receive events, event processors that analyze events and event reactors that respond to events. The responses may cause more events to occur, such that these actors form a closed loop.

A comprehensive review on the issues, formal properties and algorithms for the state-of-the-art event notification systems is provided in [13]. The system model of the notification services is based on an overlay network of event brokers, including those based on DHT [14]. There are two types of brokers: the inner brokers that route messages and the border brokers that interact with the event producers and listeners. A border broker provides an interface for clients to subscribe, unsubscribe, advertise and publish events. An event listener is responsible to implement a notify interface in order to receive notifications. However, none of the existing notification systems mentioned in [13] is based on RESTful web services.

III. EVENT-DRIVEN WEB

To project EDA to REST, we model the EDA concepts notification, subscription, publisher, and reactor as interconnected resources that support the uniform interface of REST. As the result, an event notification system becomes an event-driven web: a web of resources that responds to events as envisioned by EDA. There is no longer any boundary between different event notification systems as the event-driven webs are interconnected into the Web and interoperable under REST. Because an event-driven web is built on layered resources, we divide it into 4 layers as in Figure 1.

Layer 1 is a web of event publishers. They could be any resource that generates, advertises and publishes its events.

Layer 2 is a web of subscription resources that depends on Layer 1. Subscription resources define how notifications flow from the publishers to the reactors. They provide services for subscribers to manage the subscription links, such as change the filter, as well as to deliver and track the notifications.

Layer 3 is a web of notifications that depends on Layer 2. Notifications are treated both as resources and messages. This approach allows us to link notifications with relevant subscriptions and topics to facilitate information sharing and discovery. It also allows us to link notifications according to message exchange patterns and participants to capture the social interactions in communications and collaborations.

Layer 4 is a web of reactors that depends on Layer 3. The resources in this layer receive, process and react to the notifications. A reactor can be both a listener and publisher.

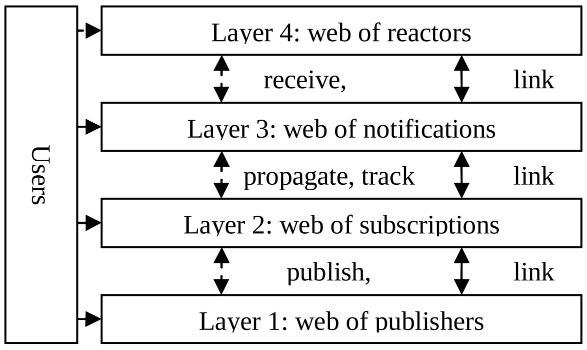
Users
Layer 4: web of reactors
receive,
link
Layer 3: web of notifications
propagate, track
link
Layer 2: web of subscriptions
publish,
link
Layer 1: web of publishers

Figure 1: Mapping EDA to layers of web

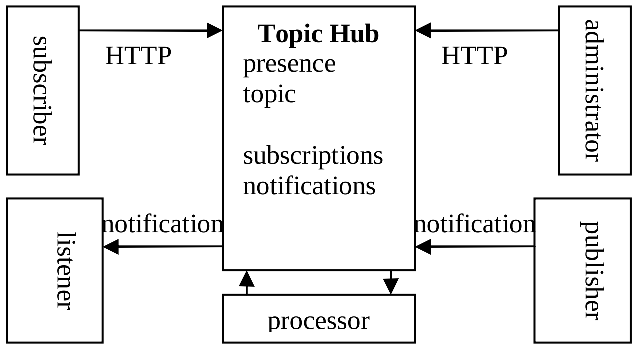

Figure 2: Topic hub resources and interactions

It should be pointed out that the resources in these layers are interconnected, such that a user can enter an event-driven web from any layer and navigate to other layers. Because layers 2 and 3 constitute services shared by publishers and reactors, they are pivotal to the event-driven web. We propose R-Event, a RESTful web service architecture to implement these two layers.

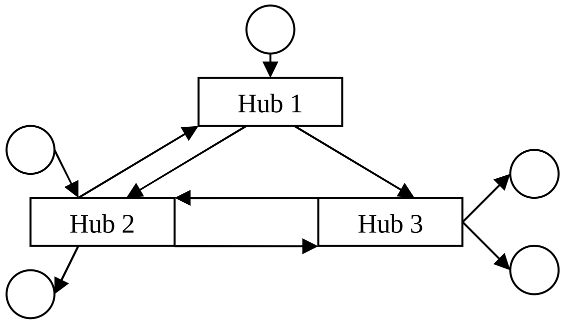

Figure 3: A topic web

IV. R-EVENT FRAMEWORK

The basic component of the R-Event framework is a topic hub that provides RESTful web services for notification publication, subscription, delivery, tracking and linking. A topic hub hosts three types of resources: topic, subscription and notification. Each hub also hosts a presence resource through which an administrator can start or shut down the services. A hub can be owned and operated by a single user or shared by a group of users. A topic hub can also invoke distributed event processors to process notifications. The high level interactions between a topic hub and its clients and servers are illustrated in Figure 2.

The topic hub is a light weight component and it can be run on any devices, including mobile phones that support HTTP protocol. It can be a servlet on a HTTP server, a standalone HTTP server, or embedded in another application. The interactions between the topic hub and its clients and servers are all based on RESTful web services.

The topic hub can also be used as a gateway that translates conventional event infrastructures into REST web services. This approach can significantly reduce the cost of web service development while ensuring the quality of services.

Because a topic hub is based on REST design, it is stateless. Consequently, a topic hub can shut down and restart safely without losing any of its services, provided that the resource states are persisted. This is especially useful when the hubs are hosted on mobile devices, which can be turned on and off. Because a topic hub is stateless, it is also scalable. We can add more topic hubs to support more clients without worrying about client session replica or affinity.

Topic hubs can be interconnected by subscriptions to provide routing services to notifications. An example topic web is illustrated in Figure 3, where topic hubs are represented as rectangles and publishers/listeners are represented by circles. The arrows indicate the subscription links on which notifications flow.

The following section describes the elements in R-Event framework in a more formal setting. In these descriptions, the left-side symbol of an equation represents a resource and the right-side tuple represents the key properties of the resource defined by this framework. Implementations can add more properties to these resources as needed.

A. Topic Tree

A topic tree is a set of topics organized as a tree. A topic is a resource to which events can be published and subscribed. More formally, a topic t has a set of events E, a set of children topics C:

$t = (E, C), C=\{ t_j \mid t_j \textit{ is a child topic of } t\}$.

B. Subscription

Conceptually, a subscription is a directed link from a publisher (p) to a listener (l). We extend subscription to have a set of alternative listeners (L), filter (f), expiration (d), and status (u), such as active or paused. More formally, we have:

$s = (p, L, a, f, d, u), L = \{l|l \textit{ is a listener}\}$

A notification n can propagate to one of the listeners in L if and only if the filter is evaluated to true, i.e. $f(n)=true$. Which listener is selected is determined by an algorithm a, defined by the subscriber. A simple algorithm is to try listeners according to the order they are created until one succeeds.

Subscriptions can be used to link two topics by treating them as either publisher or listener. A subscription link from a publishing topic to a listening topic is represented by two subscription resources, each as a subordinate resource of the involved topics. On the publishing topic, it is called outbound subscription (*os*), as notifications flow out of it. On the listening topic, it is called inbound subscription (*is*), as notifications flow into it. The two matching subscriptions are double linked to keep their correspondence. More formally, we have:

$os = (L, a, f, d, u), L=\{l|l=(t_j, is, g(is))\}, a(L)=l$

$is = (l, g(l)), l \in L$

Here each listener resource l consists of: 1) listening topic t_j; 2) inbound subscription *is*, and 3) the presence of *is*: $g(is)$. An inbound subscription consists of: 1) the listener l; and 2) the presence of l: $g(l)$.

C. Topic Web

Given a set of topic hubs $H=\{h_i\}$ where each hub hosts a set of topic trees $T(h_i)=\{t|t \textit{ is a topic on } h_i\}$, these topic trees form a web of topics linked by subscriptions. More formally, a topic web $W(H)$ on top of a set of hubs H is defined as:

$$W(H) = \bigcup_{h_i \in H} T(h_i)$$

D. Notificatiion

A notification is also modeled as an addressable resource that can be updated. More formally, we have:

$n_i=(o, r, b, R)$,

$r=\{(t,m)|t \textit{ is a topic, } m \textit{ is timestamp}\}$,

$R=\{n_j|n_j \textit{ is a response to } n_i\}$

where:

- *origin (o)*: the URI of the original notification as it was posted. Any propagated copy of the original notification inherits this property to track its origin.
- *route (r)*: the ordered set of topics (t) and timestamp (m) visited by this notification during delivery. This is used to detect loop and to expose topics to listeners.
- *about (b)*: the URI of the notification that this message responds to.
- *Responses (R)*: the set of notifications responded to this notification

The topic web may contain cycles of subscriptions. To facilitate loop detection, each notification message has a special property route, which contains a list of topics visited by the notification during propagation. Each hub checks if the current topic is in the list. If so, a loop is found and the notification will not be propagated. Otherwise, the hub appends the topic to the list and propagates the notification.

E. Resource Design and Hub Protocols

The key properties, interfaces and relations of the resources are depicted in the UML class diagram in Figure 4.

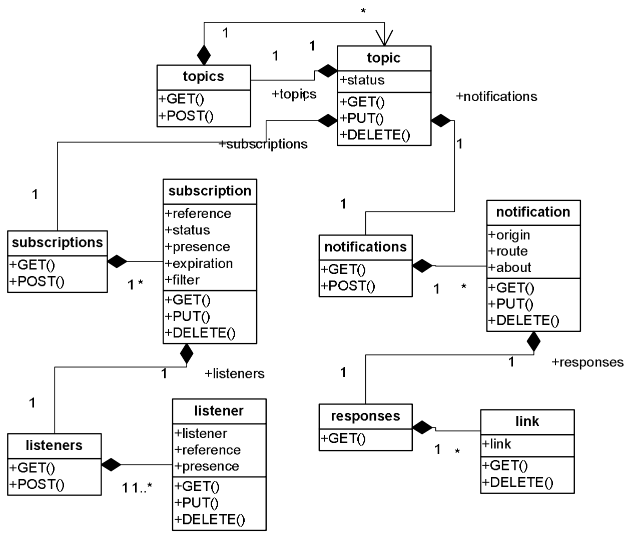

Figure 4: Main resources on topic hub

To facilitate client access, each resource on a hub is addressed by a predefined URI template that reflects the subordinate relations defined above:

- Topic t: */topics/{t}*;
- Child topic t_j of topic t: */topics/{t}/topics/{t_j}*;
- Subscription s of topic t: */topics/{t}/subscriptions/{s}*;
- Listener l: */topics/{t}/subscriptions/{s}/listeners/{l}*;
- Notification with UUID *{n}* on topic t: */topics/{t}/notifications/{n}*.

A subscription link from topic *ta* on hub A to topic *tb* on hub B is established by a user using a web browser as follows:

1. The user requests a subscription resource under *ta* with POST;
2. Before returning to the user, hub A creates the outbound subscription under *ta* and requests the corresponding inbound subscription under *tb* with PUT (nested inside the POST);
3. Both requests succeed and the response is returned to the user;

A notification is propagated between hubs by a user as follows:

1. The user posts a notification to a topic on hub A using POST that returns when the resource is created;
2. The notification is delivered by a scheduler to all listening topics with PUT that maintains the original UUID assigned to the notification by hub A; as the result, all the propagated notifications on different hubs can be identified by the same UUID;

This framework does not define the representations of its resources, which is left to the implementations. Different representations (media types) of the same resource are supported through HTTP content negotiation. The communications between web browsers and the topic hubs are also outside the scope of this framework, as we expected they can be addressed by the upcoming W3C standards [9].

F. *Security*

The communication between the topic hubs are secured using HTTPS with PKI certificates based mutual authentication. For this to work, each topic hub maintains a X.509 certificate issued by a CA (Certificate Authority) that is trusted by other hubs. It is possible or even preferable, to obtain two certificates for each topic hub: one for its client role and one for its server role, such that these two roles can be managed separately.

The communications between the topic hubs and web browsers (users) are also secured by HTTPS. In this case, the browser authenticates the topic hub certificate against its trusted CA. In return, the users authenticate themselves to the hub using registered credentials (login/password or certificate). Once a user is authenticated to a topic hub A, it employs role-based authorization model to authorize a user for his actions.

If the user wants to create a subscription link from hub A to hub B, B has to authorize A for the inbound subscription. To satisfy this condition, the user first obtains an authenticated authorization token from hub B. The user then sends this token with the subscription message to hub A. Hub A uses this token to authorize itself to hub B for the inbound subscription creation. Once hub B creates the resource, it returns an access token to hub A to authorize it for future notifications to that topic.

An alternative to the above scheme is to use the OAuth 1.0 Protocol [31] that allows a user to authorize a third-party access to his resources on a server. In this case, hub A becomes the third-party that needs to access the topic resources on hub B owned by the user. Here is how it works at a very high level: 1) the user visits hub A to create a subscription to hub B; 2) hub A obtains a request token from hub B and redirects the user to hub B to authorize it; 3) the user provides his credentials to hub B to authorize the request token and hub B redirects the user back to hub A; 4) hub A uses the authorized request token to obtain an access token from hub B and creates the inbound subscription on B.

In both approaches, the user does not have to share his credentials on hub B with hub A.

V. ADVANTAGES OF EVENT-DRIVEN WEB

On surface, the event-driven web built on top of the R-Event framework, as described in the previous section, appears similar to the broker overlay network in the conventional notification architecture [13]. However, it has the following advantages due to a REST based design.

A. *Addressability and Connectedness*

Unlike conventional broker overlay networks that are a closed system whose usability is prescribed by the API, all resources in a topic web are addressable and connected. Unlike in conventional broker overlay network that distinguishes between inner, border, or special rendezvous brokers, a topic web consists of homogeneous topic hubs with the same type of web services. The users can navigate and search the topic web to find the interested information using regular web browsers or crawlers. The addressability and connectedness increase the “surface area” of the web services such that the information and services in a topic web can be integrated in many useful ways beyond what is anticipated by the original design.

B. *Dynamic and Flexible Topology*

Unlike in conventional broker network where brokers have fixed routing tables, a topic web can be dynamically assembled and disassembled by users for different needs. Its topology can be changed on the fly as subscriptions are created and deleted and hubs join and leave the topic web. For example, a workflow system can be created where work items are propagated as notifications between users. In an emergence situation, a group of people can create an ad-hoc notification network to share alerts and keep informed. In an enterprise, a topic web about a product can be created on-demand such that alerts from field technicians can propagate to proper sales and supporting engineers in charge of the product to better serve the customers. In any case, once the task is finished, the topic web can be disassembled or removed completely. In this sense, a topic web is similar to an ad-hoc peer-to-peer network. However, a topic web is based on REST web services whereas each type of P2P network depends on its own protocols.

In conventional notification services, a broker routes all messages using one routing table. Therefore, it cannot participate in more than one routing topology. In our framework, a hub can host many topics, each having its own routing table (subscription links). As a result, a hub can simultaneously participate in many different routing networks. This gives the users the ability to simultaneously engage in different collaboration tasks using the same topic web.

C. *Robustness and Scalability*

Topic hubs are robust because its resource states can be persisted and restored to support temporary server shutdown or failover.

The safe and idempotent operations, as defined by HTTP 1.1 [29] also contribute to the robustness. Our framework uses nested HTTP operations where one operation calls other operations. We ensure that such a chain of operations is safe and idempotent by limiting how operations can be nested inside each other as follows:

nested(GET)={GET}
nested(POST)={GET,POST,PUT,DELETE}
nested(PUT)={GET,PUT,DELETE}
nested(DELETE)={GET,PUT,DELETE}

The robustness and scalability also come from the statelessness of REST design. The statelessness means that a topic hub can process any request in isolation without any previous context. By removing the need for such context, we eliminate a lot of failure conditions. In case we need to handle more client requests, we can simply add more servers and have the load balancer distribute the requests at random to the servers who share the resources. If the resource access becomes a bottleneck, we can consider duplication or partition of resources. This robustness and scalability is crucial when a topic hub serves as the gateway to large-scale notification systems.

D. Efficient Notifications

In conventional notification systems, notification is a message that can only be transmitted and stored. In our framework, notifications are also modeled as REST resources that provide services. Such model addresses the following issues in notification services:

Inline update: Because notifications are treated as addressable resources, a publisher can update a posted notification (using PUT) without having to create a new one. The updates will propagate over the subscription links in the topic web. This kind of inline update is more difficult to achieve in conventional notification services that treat notifications as messages.

Duplicate notification: In the topic web, a topic may receive different copies of a notification from multiple routes or multiple inline updates of the same notification, leading to potential duplicated notifications. Because our framework uses PUT to deliver notifications, the duplicate notifications to a hub become multiple updates to a resource. Therefore, we can use HTTP `ETag` and `If-None-Match` headers to efficiently detect duplicate notifications and avoid spurious alerts to the users. Compared to the solution proposed in [13], this approach solves the difficult problem without constraining the topology of the topic web.

VI. IMPLEMENTATION AND EXPERIMENTS

A prototype event notification system has been developed based on the described R-Event framework. The notification system allows users within a group to publish and subscribe presence information. Users can respond to received presence information to enable real-time collaboration. For example, when an expert becomes available through his presence notification, a manager may respond to the notification and propose a new task force be formed with the expert as the team leader. This response is propagated to the group so that interested members can set up a new workflow using the proposed topic web.

The prototype was written in Java using Restlet 1.1.4 [23]. The implementation followed the Model-View-Controller (MVC) design pattern. The Model contains the persistent data stored on disk. The Controller contains the resources and the View contains the view objects that generate XHTML pages from the XHTML templates. The topic hub stack was implemented by four Java packages, as illustrated below.

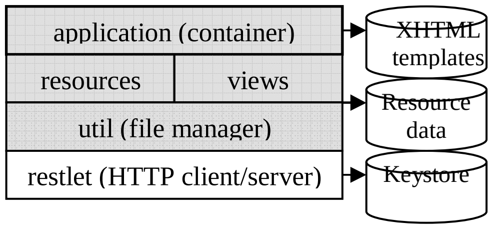

Figure 5: Topic hub stack

For this prototype, we used OpenSSL package [30] as the CA to generate certificates for the topic hubs, and Java keytool to manage the keystores for the hubs. Resources states are managed by a file manager that synchronizes the access to them. A hub used a separate thread to dispatch notifications from a queue shared by all resources. Because HTML form only supports POST and GET, we used JavaScript (XMLHttpRequest) to implement the PUT and DELETE operations for pages that update or delete resources.

Users interact with the services using web browsers (Firefox in our case). For demo purpose, the notifications were delivered to the browsers using automatic page refreshing. This is a temporary solution as our focus is on communications between hubs, instead of between browser and server. However, the R-Event framework should work with any client side technologies, such as Ajax or Server-Sent Event technologies.

We measured the performance of the prototype system in a LAN environment. The hubs were running on a Windows 2003 Server with 3GHz dual core and 2GB RAM. The performances of several key services were measured, where S means subscription, L means listener, and N means notification. The time durations for each method are recorded in the following table. The time duration includes processing the request, saving data to the disk, and assembling the resource representation.

TABLE 1: PERFORMANCE MEASURED IN MILLISECONDS

task/time	POST S	POST L	PUT S	POST N	PUT N
avg	14.1	38.9	6.2	9.5	0
std	13.7	16.8	8.0	8.1	0

The table shows that adding a listener (POST L) takes the longest time and this is expected because it is a nested operation, where

t(POST L)=processing time + network latency + t(PUT S).

The time to update a notification (PUT N) is ignorable (0 ms) and this is good news, since we use PUT to propagate notifications.

VII. CONCLUSIONS

The contributions of this paper are summarized as follows:

1. We presented an approach and a framework in which the elements in EDA can be projected and represented by REST resources, protocols and services;
2. We developed a RESTful web service framework, R-Event, based on this projection. The REST resources, protocols, services and securities are defined formally as well as described informally;
3. We illustrated that an event-driven web can be built using this framework, and discussed the advantages, including addressability, dynamic topology, robustness and scalability, etc. of this approach over conventional notification systems.
4. We developed a prototype using secure HTTP. The preliminary performance tests showed that the proposed approach is feasible and advantageous.

Our plan is to test the framework in a large scale network environment and analyze its behaviors and performance in those deployments.

REFERENCES

[1] Richardson, L. and Ruby, S., *RESTful Web Services*, O'Reilly Media, Inc. 2007.

[2] Fielding, R., *Architectural Styles and the Design of Network-based Software Architectures*, Ph.D. Dissertation, 2000, http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. Last Accessed: August 27, 2010.

[3] Jacobs, I. and Walsh, N., (eds), *Architecture of the World Wide Web, Volume One*, W3C Recommendation 15 December 2004. http://www.w3.org/TR/webarch/, Last Accessed: August 27, 2010.

[4] The Atom Syndication Format, 2005, http://www.ietf.org/rfc/rfc4287.txt, Last Accessed: August 27, 2010.

[5] The Atom Publishing Protocol, 2007, http://www.ietf.org/rfc/rfc5023.txt, August 27, 2010.

[6] RSS 2.0 Specification, 2006, http://www.rssboard.org/rss-specification, Last Accessed: August 27, 2010.

[7] Pushlets, http://www.pushlets.com/, Last Accessed: August 27, 2010.

[8] HTML Working Group, 2009, http://www.w3.org/html/wg/, Last Accessed: August 27, 2010.

[9] Hickson, I. (ed), Server-Sent Events, W3C Working Draft 29 October 2009, http://www.w3.org/TR/eventsource/, Last Accessed: August 27, 2010.

[10] Hickson, I. (ed), The Web Sockets API, W3C Working Draft 29 October 2009, http://www.w3.org/TR/websockets/, Last Accessed: August 27, 2010.

[11] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns*, Addison-Wesley, 1995

[12] Taylor, H., Yochem, A., Phillips, L. and Martinez, F., *Event-Driven Architecture, How SOA Enables the Real-Time Enterprise*, Addison-Wesley, 2009.

[13] Mühl, G., Fiege, L. and Pietzuch, P.R., *Distributed Event-Based Systems*, Springer, 2006.

[14] Rowstron, A., Kermarrec, A.M., Castro, M. and Druschel, P., SCRIBE: The design of a large-scale event notification infrastructure, Proc. of 3rd International Workshop on Networked Group Communication, November 2001, pp 30-43.

[15] Buschmann, F. et al. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. West Sussex, England: John Wiley & Sons Ltd., 1996.

[16] Chandy, K. M. (2006). Event-Driven Applications: Costs, Benefits and Design Approaches, Gartner Application Integration and Web Services Summit 2006, http://www.infospheres.caltech.edu/node/38, Last Accessed August 27, 2010.

[17] Michelson, B. M. (2006). Event-Driven Architecture Overview, http://soa.omg.org/Uploaded%20Docs/EDA/bda2-2-06cc.pdf, Last Accessed August 27, 2010.

[18] Davis, D., Malhotra, A., Warr, K. and Chou, W., (eds), Web Services Eventing (WS-Eventing), W3C Working Draft, 5 August 2010. http://www.w3.org/TR/ws-eventing/, Last Accessed August 27, 2010.

[19] Graham, S., Hull, D., Murray, B., (eds), Web Services Base Notification 1.3 (WS-BaseNotification), OASIS Standard, 1 October 2006. http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf, Last Accessed August 27, 2010.

[20] Chappell, D. and Liu, L., (eds), Web Services Brokered Notification 1.3 (WS-BrokeredNotification), OASIS Standard, 1 October 2006. http://docs.oasis-open.org/wsn/wsn-ws_brokered_notification-1.3-spec-os.pdf, Last Accessed August 27, 2010.

[21] Vambenepe, W., Graham, S. and Biblett, P., (eds), Web Services Topics 1.3 (WS-Topics), OASIS Standard, 1 October 2006. http://docs.oasis-open.org/wsn/wsn-ws_topics-1.3-spec-os.pdf, Last Accessed August 27, 2010.

[22] Fitzpatrick, B., Slatkin, B. and Atkins, M., PubSubHubbub Core 0.2, Working Draft, 1 September 2009, http://code.google.com/p/pubsubhubbub/, Last Accessed August 27, 2010.

[23] Restlet, RESTful Web framework for Java, http://www.restlet.org/, Last Accessed August 27, 2010.

[24] JMS (2002). Java Message Service, version 1.1, 2002, http://www.oracle.com/technetwork/java/index-jsp-142945.html, Last Accessed August 27, 2010.

[25] Event Service Specification, Version 1.2, October 2004, 2004.

[26] Notification Service Specification, Version 1.1, October 2004.

[27] Christensen, E., Curbera, F., Meredith, G. and Weerawarana, S., Web Services Description Language (WSDL 1.1), W3C Note, 15 March 2001. http://www.w3.org/TR/wsdl, Last Accessed August 27, 2010.

[28] Gudgin, M., et al, SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), W3C Recommendation, 27 April 2007. http://www.w3.org/TR/soap12-part1/, Last Accessed August 27, 2010.

[29] Fielding, R., et al. Hypertext Transfer Protocol – HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html, Last Accessed August 27, 2010.

[30] OpenSSL: http://www.openssl.org/, Last Accessed August 27, 2010.

[31] The OAuth 1.0 Protocol: http://tools.ietf.org/html/rfc5849, Last Accessed August 27, 2010.