REST-Event: A REST 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 World Wide 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 describe a REST web service framework, REST-Event. It represents and organizes the concepts and elements of Event-Driven Architecture (EDA) as REST (Representational State Transfer) web services. Our approach leads to a layered event-driven web, in which event actors, subscriptions and event channels are separated. As an integration framework, REST-Event specifies a set of minimal REST services to support event systems, such that generic two-way event channels can be created and managed seamlessly through a process called subscription entanglement. A special form of event-driven web, called topic web, is proposed and built based on REST-Event. The advantages and applications of topic web are presented and discussed, including addressability, connectedness, dynamic topology, robustness and scalability. In addition, a prototype topic web for presence driven collaboration is developed. 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. 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 and follow them. However, in the era of social media, information updates become frequent and rapid. People need timely and almost instant availability of these dynamic contents to interactively use this information without being overwhelmed by the information overload. This demands the Web to evolve rapidly from a static and reactive informational space to a dynamic communication and collaboration oriented environment. As a consequence, this migration of Web can affect the application spaces of both consumers and enterprises for future services. The trend of a communication and collaboration Web pushes for an event-driven web, in which information sharing is driven by asynchronous events to support dynamic, real-time, or 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 for the following reasons.

First, most of the architectures, protocols, and programming languages for conventional distributed event notification systems were developed prior to the Web. As a result, these notification systems are not accessible to each other on the Web or fit the infrastructure of the Web.

Secondly, the current web technologies related to event notifications, including Atom [4][5], Server-Sent Events [9], Web Sockets [10], Bidirectional HTTP [33] and HTML5 [8], focus mainly on client-server interactions and are not sufficient to support integrations between web sites and web applications across organizational boundaries.

Therefore, there is an acute need for a unifying framework that can provide seamless integration of these notification systems with the infrastructure of web and web based services. Such a unifying framework can transform conventional notification systems into web services such that they become part of the Web. It can also be used to integrate and enable the existing web based applications, including those social network sites, which currently do not have a way to share events. If these two goals can be achieved effectively, then it could lead to a nested event notification system on the Web - an event-driven web extension to the current Web.

To develop such a unifying framework, we lay our foundation on Event-Driven Architecture (EDA) [12], in which information is encapsulated as asynchronous events propagated to the interested components when they occur. EDA defines the principles and architecture for event discovery, subscription, delivery and reaction, which are also key components in event-driven web for real-time communication and collaboration. Moreover, EDA is a natural fit for the event-driven web as both architectures assume a distributed system that are developed and maintained independently by different organizations without any centralized control.

To apply EDA to the web architecture, we represent and organize EDA concepts and elements as REST [1][2] web services in a framework called REST-Event. As an integration framework, REST-Event demands a set of minimal REST services supported by the systems to be integrated but at the same time supports different event channels between the systems. For this reason, we generalize the traditional one-way event channels, in which event notifications flow in one direction to two-way event channels, in which event notifications can flow in both directions. To hide complexity of managing two-way event channels, we introduce a process called subscription entanglement based on REST protocols. With this framework, the event-driven web can be built and operated as a distributed hypermedia system, for which REST is optimized.

By projecting EDA to REST, many important problems in conventional event notification systems can be resolved efficiently. The uniform interface, connectedness, and addressability of REST can facilitate the discovery of notification web services. The idempotent operations and statelessness of REST can enhance robustness and scalability to notification web services. Subscription entanglement hides system complexity from the clients.

To test REST-Event framework, a prototype event-driven web, topic web, which consists of distributed topic hubs, is implemented using REST-Event 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 our vision of event-driven web. Section IV introduces the REST-Event framework. Section V describes a special event-driven web called topic web that can be built from REST-Event. Section VI summarizes the advantages of the topic web. Section VII is dedicated to a prototype implementation and experimental study results. Findings of this paper are summarized in Section VIII.

II. Related Work

This paper extends our previous work [1] published in Service Computation 2010 in the following aspects: 1) the new framework is based on a new layered system with generalized event channels, whereas our previous work assumes all event channels are HTTP; 2) the new framework supports creation of two-way event channels in one transaction, whereas the previous framework only supports one-way event channels; 3) this paper clarifies the notions of entangled subscriptions; 4) the core resources and protocols of the framework are separated from the topic web, which is a system built from the described framework.

REST stands for REpresentational State Transfer. It is an architecture style optimized for distributed hypermedia system as described in [2][3][4]. The fundamental constraint of REST is that the interactions between a client and servers should be driven by hypermedia. In other words, a client should be able to start from a single URI and transition to a desired state by following the links in the hypermedia provided by the servers. This constraint is realized by the following architectural constraints: 1) Addressability: each resource can be addressed by URI. 2) Connectedness: resources are linked to enable transitions. 3) Uniform Interface: all components in the system support the same interface, namely HEAD, GET, PUT, DELETE and POST. HEAD and GET are safe and idempotent. PUT and DELETE are not safe but idempotent. Idempotent operations can be executed many times by a server and have the same effect as being executed once. This property allows a client to resubmit a request in case of failures without worrying about undesired side-effect, such as paying something twice. 4) Statelessness: all requests to a resource contain all information necessary to process the requests, and the servers do not need to keep any context in order to process the requests. Stateless servers have much less failure conditions than stateful ones and are easy to scale and migrate. 5) Layering: intermediate proxies between clients and servers can be used to cache responses, enforce security polices, or distribute workloads.

RSS [7] and Atom [5] are two data formats that describe web feeds to be consumed by feed readers. A feed can be news, blogs, wikis, or any resource whose content may be updated frequently by the content providers. The content providers are responsible to publish their feeds. This is usually done by embedding the feed URI in a web page. The feed readers are responsible to find the feeds, for example by following feed URI. Once a feed is found, the feed reader fetches the updates by periodically polling the feed. 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 [23]. 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 a listener. Whenever there is an update, the feed publisher notifies its hub, which then fetches the feed and multicasts (push) it to the registered listeners. 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. This protocol only supports one-way event channels from a topic hub to its listeners. The event channels are also fixed to be Atom over HTTP. Also the system does not use subscription entanglement to manage event channels.

Many techniques have been developed over the years to address the asynchronous event delivery to the web browsers, such as Ajax, Pushlet [8], and most recently Server-Sent Events [10] and Web Sockets [11]. However, these techniques are not applicable to federated notification services where server to server relations and communication protocols are needed.

Bayeux [34] is a protocol that supports both HTTP long-polling and streaming mechanisms to allow a HTTP server to push notifications to a HTTP client. This protocol can be combined with normal HTTP request/response to support two-way event channels. But this protocol does not specify how to create and manage subscriptions.

BOSH (XEP-0124) [35] uses HTTP long-polling to emulate bidirectional TCP streams. XMPP also supports a publish/subscribe extension (XEP-0060) [36] to allow XMPP entities to subscribe and publish events to topics. But these protocols do not specify how to create and manage subscriptions.

Server-Sent Events [10] defines a protocol that uses HTTP streaming to allow a HTTP server to push notifications to a HTTP client. This protocol can be combined with normal HTTP request/response to create two-way event channels. However, the protocol does not specify how to create or manage subscriptions.

Google Wave is a platform and protocol to provide near real-time communication and collaboration between web browsers. Since Google Wave Federation Protocol [37] is based on XMPP, it does not support integration with the Web directly. Google Wave Client-Server Protocol [38] is based on WebSockets and JSON. The protocol does not specify how to create subscriptions.

Microsoft Azure cloud platform [39][40] has several built-in mechanism to support EDA (Event-Driven Architecture), including server-to-server event subscriptions, for example between an event queue and a router. But the platform does not support two-way event channels through subscription entanglement.

WIP [41] uses WS-Eventing to negotiate media transports for IP based multimedia communication. The basic idea in WIP is to treat media streams as two-way events and WS-Eventing is used to negotiate the media transport parameters. Although WIP supports a form of two-way event channels, there are some significant differences. First, WIP is based on WS-* instead of REST. Second, WIP is aimed to establish media transports (RTP) between two endpoints, whereas REST-Event is aimed to integrate different notification systems.

In software engineering, Publisher-Subscriber [16] or Observer [12] 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) [25], CORBA Event Service [26], CORBA Notification Service [26], which are not based on web services.

Recently, two event notification web services standards, WS-Eventing [19] and WS-Notification [20][21] are developed. However, these standards are not based on REST. Instead they are based on WSDL [28] and SOAP [29], which are messaging protocols alternative to REST [1]. WS-Topic [22] 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) [13][17] and its interaction with Service-Oriented Architecture (SOA) [18] 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 [14]. The system model of the notification services is based on an overlay network of event brokers, including those based on DHT [15]. 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 [14] is based on RESTful web services.

III. EVENT-DRIVEN WEB

To project EDA to REST, we model the EDA concepts, such as subscription, publisher, listener and broker, as interconnected resources that support the uniform interface of REST. As the result, a distributed event notification system becomes the event-driven web: a web of resources represented as distributed hypermedia that propagates and responds to events as envisioned by EDA. There is no longer any boundary between different event notification systems as they can expose their interfaces through these resources and become part of the event-driven web.

The event-driven web is a layered system with the following layers as shown in Figure 1.

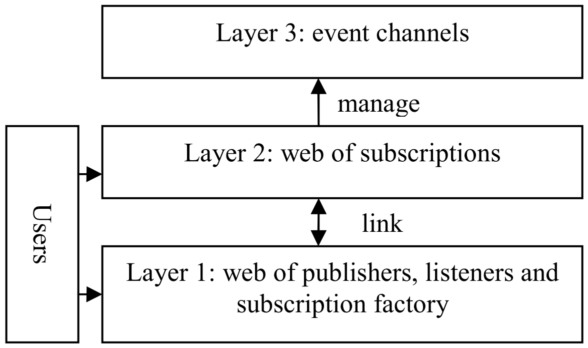

Layer 3: event channels
manage
Layer 2: web of subscriptions
link
Users
Layer 1: web of publishers, listeners and subscription factory

Figure 1. Three layers of the event-driven web

Layer 1 is a web of event publishers, listeners and subscription factories. A publisher is a resource that advertises events. A listener is a resource that accepts event notifications. A subscription factory is a resource that accepts subscriptions on behalf of a publisher. These resources expose their functions through REST services. They serve as the access points to a complex event notification system treated as a black box to the Web.

Layer 2 is a web of subscription resources that are created from the resources in Layer 1. Subscription resources exist as entangled pairs and each pair defines an association between an event publisher and an event listener, in which event notifications are sent. These associations are referred as event channels. These entangled subscriptions resources provide REST services to manage the event channels. Layer 1 and 2 resources are connected so navigations between layers are supported.

Layer 3 consists of event channels created from subscription resources in Layer 2. Unlike resources in layers 1 and 2 that provide REST services, the event channels may use any protocol to transmit event notifications, such as HTTP, JMS, and TCP/IP. In fact, the event notifications do not have to be discrete messages and can be media streams that are transmitted over RTP as proposed in WIP [41]. If two event systems do not use the same protocol for notifications, adaptors may be used to convert notifications.

To build such a layered system through integration of existing distributed event systems, REST-Event framework defines a set of minimal REST services required for the resources in Layer 1, upon which subscriptions can be created. The REST-Event itself defines the protocols for the subscription management. To be flexible, REST-Event does not define the protocols for Layer 3 since these protocols are defined by the existing event systems. REST-Event does not define any event filter languages either as they are also defined by the existing event systems. However, the notification protocols and filters can be specified in subscriptions.

IV. REST-EVENT FRAMEWORK

REST-Event framework defines a minimal set of resources and protocols to support the creation and management of event channels through subscriptions. The following subsections describe the core resources and protocols, namely: Discovery, Creation and Deletion in this framework.

A. *Discovery Protocol*

REST requires that a REST API should be entered with a single URI without any prior knowledge except media types and link relations [3]. This means, when being provided with a URI, an event subscriber must be able to determine if the URI points to an event publisher, and if so, which resource can be used to create subscriptions. To satisfy this requirement, REST-Event requires an event publisher to support the Discovery Protocol. The protocol contains three elements: a HTTP HEAD request from a subscriber to a publisher, a HTTP response from the publisher to the subscriber, and a special link relation `subscribe` in the response message. Suppose the given URI is http://www.host1.com/topic1, then the HEAD request and response could be:

```
HEAD /topic1 HTTP 1.1
Host: www.host1.com

200 OK HTTP 1.1
Link: </topic1/factory1>; rel=subscribe
```

The special `subscribe` link in the response tells the subscriber two things: 1) the requested resource is an event publisher; and 2) the location of its subscription factory that accepts subscription requests to this event publisher.

The link is specified in the header following RFC5988 [42]. This approach allows an event publisher to delegate its subscription management to another resource. Specifying the link in the header instead of the body has the advantage that the link is independent of the outgoing representations.

B. *Creation Protocol*

In conventional event-based systems, a subscription represents a one-way event channel, in which event notifications flow from the publisher to the listener. However, in many cases, two-way event channels, in which event notifications can flow in both directions are necessary in communication and collaboration systems. It is possible for a subscriber to create a two-way event channel in these systems with two separate subscriptions, each representing a one-way event channel. But this approach has the following drawbacks. First, the system complexity is exposed to the subscriber, which is typically a human user. Second, the system relies on an external entity (subscriber) to control its state. If the external entity leaves the system in an inconsistent state, for example failing to delete one subscription, then it is difficult for the system to detect and recover from the inconsistence.

To address this issue, REST-Event supports creation and management of two-way event channels in one transaction initiated by a subscriber. The conventional one-way event channels are a special case of two-way event channels. This approach hides the complexity and keeps the system in the loop so that any failure can be detected and recovered. Two-way event channels are created by a process called subscription entanglement, in which a pair of subscriptions are created and linked to have the same lifecycles. Subscription entanglement is realized by the subscription protocol that involves interactions between three entities: an event subscriber, a subscription factory and an event listener. REST-Event therefore requires that event systems to be integrated exposing a subscription factory resource and a listener resource that support the Creation Protocol.

Without losing generality, we assume the subscription protocol is defined in terms of HTTP and XML according to REST. Figure **2** illustrates the protocol messages exchanged between the involved resources: `topic1` is an event publisher resource and `facotory1` is the subscription factory of `topic1` of the first event system; `topic2` is an event listener resource and `factory2` is the subscription factory of `topic2` of the second event system.

The Creation Protocol creates a two-way event channel between the two systems through subscription entanglement as follows.

Step 1: the subscriber sends a HTTP POST request to `http://www.host1.com/topic1/factory1` to create a subscription that specifies a two-way event channel consisting of two one-way channels. Both outbound and inbound channels specify a source URI and a sink URI. In this case, the outbound channel tranmits notifications from the source `topic1` to the sink `topic2` and the inbound channel goes the opposite direction. In general, the two event channels can be separated with 2 different pairs of sources and sinks, but always have the same lifetime as specified by the expiry element. Each event channel can have its own filter:

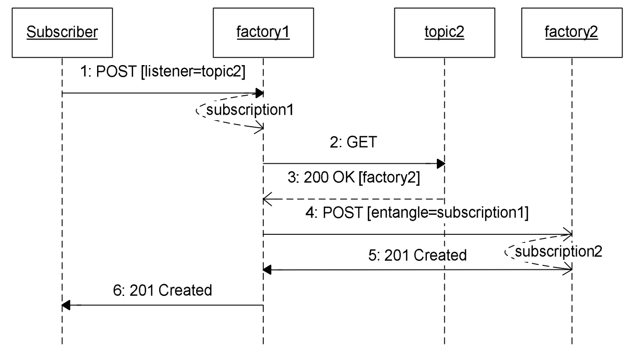


Figure 2. Creation Protocol

```
POST /factory1 HTTP 1.1
Host: www.host1.com
Accept: application/xml

<subscription>
    <expiry>…</expiry>
    <outbound>
       <source
href="http://www.host2.com/topic1" />
       <sink
href="http://www.host2.com/topic2" />
       <filter>…</filter>
    </outbound>
    <inbound>
       <source
href="http://www.host2.com/topic2" />
       <sink
href="http://www.host2.com/topic1" />
       <filter>…</filter>
    </inbound>
</subscription>
```

The `factory1` resource will process this request and create a subscription resource `subscription1` that represents the outbound event channel from `topic2` to `topic1` in the request.

Steps 2-3: The `factory1` resource sends a HTTP GET to the `topic2` resource to discover its factory using the Discovery Protocol discussed before.

Step 4: The `factory1` resource sends a POST request to `factory2` found above:

```
POST /factory2 HTTP 1.1
Host: www.host2.com
Accept: application/xml

<subscription>
    <expiry>…</expiry>
    <link                        rel="entangle"
href="http://www.host1.com/subscription1" />
    <outbound>
       <source
href="http://www.host2.com/topic2" />
       <sink
href="http://www.host1.com/topic1" />
       <filter>…</filter>
    </outbound>
</subscription>
```

Step 5: The `factory2` resource creates `subscription2` and links it with `subscription1`. On success, it responds with a URI to the created `subscription2`:

```
201 Created HTTP 1.1
Content-Type: application/xml
Location: http://www.host2.com/subscription2
```

Upon receiption of the response, the `factory1` links `subscription1` to `subscription2`.

Step 6: The `factory1` resources returns a response to the subscriber that contains the link to `subscription1` for the subscriber to access the entangled subscriptions:

```
201 Created HTTP 1.1
Location: http://www.host1.com/subscription1
```

This completes the creation of entangled subscriptions that are mutually linked. If there is a failure in steps 2-5, the partial subscriptions will be deleted and the subscriber will receive an error status code in response.

To create a one-way event channel from `topic1` to `topic2`, we just remove the `<inbound>` element from the request message in Step 1 and the `<outbound>` element from the request in Step 4. The rest messages will be the same.

The `factory2` discovered in steps 2-3 can be cached so the total number of messages can be reduced to 4 when the Subscription Protocol is repeated on the same resources.

C. *Deletion Protocol*

Since the entangled subscriptions have the same lifecycle, deleting any one of them will delete the other. The Deletion Protocol is illustrated by the following sequence diagram (Figure **3**).

When a client deletes a subscription, the resource will delete the local subscription state to reclaim the space. It then deletes the entangled subscription to maintain the same lifecycle. The deletions can also be initiated by a HTTP server that terminates a local subscription for various reasons, such as the server is shutting down or reclaiming spaces.

V. TOPIC WEB

This section demonstrates the use of REST-Event framework in creating a form of event-driven web called topic web. A topic web consists of federated topic hubs that implement the REST-Event protocols. A topic hub hosts many topic resources that are linked into a topic tree. A topic resource is a kind of event broker. A topic web can be regarded as the event delivery backbone in the conventional distributed event systems. But a topic web offer more flexibility and extensibility than conventional event delivery backbones.

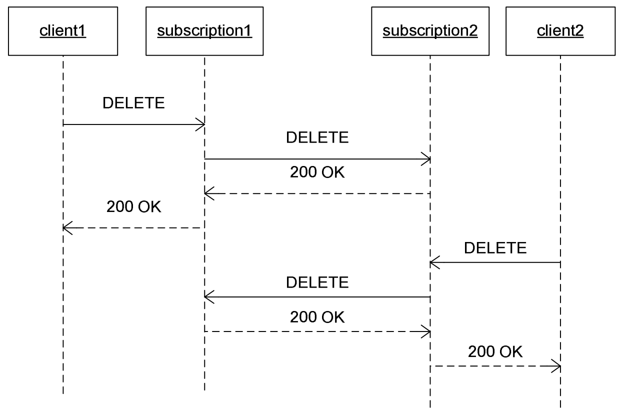


Figure 3. Deletion Protocol

A topic hub hosts resources required by REST-Event: topic, which is a publisher and listener, subscription factories and subscriptions. 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 **4**.

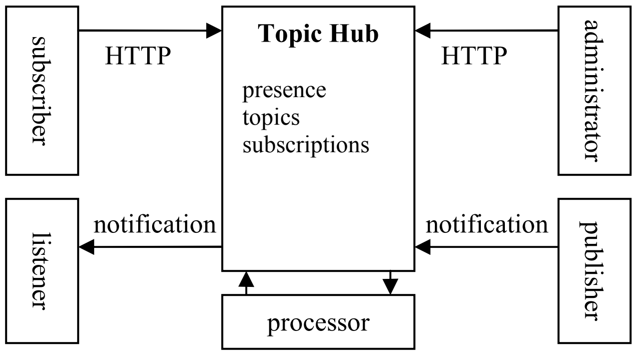


Figure 4. Topic hub resources and interactions

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 Java Servlet on a HTTP server, a standalone HTTP server, or embedded in another application.

A topic hub can be a gateway between conventional event systems and the REST web services. In this sense, a topic hub represents a complex event system hidden to the Web. This approach can significantly reduce the cost of web service development while reusing the existing event infrastructures to ensure quality of services.

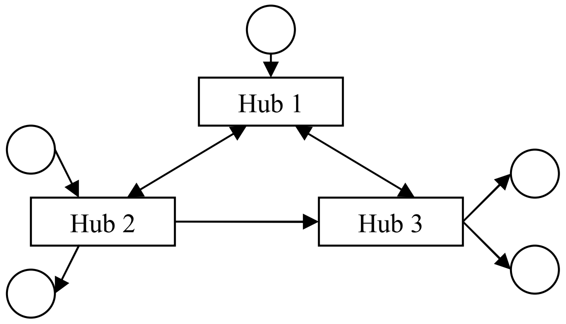


Figure 5. A topic web

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.

Event channels between topic hubs are created and managed by REST-Event protocols. An example topic web is illustrated in Figure **5**, where topic hubs are represented as rectangles and publishers/listeners are represented by circles. The arrows indicate the event channels.

The following paragraphs describe the elements in topic web in a more formal setting with set-theoretic notations.

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:

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

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

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

A. *Resource Design*

The key properties, interfaces and relations of the resources are depicted in the UML class diagram in Figure **6**.

Each resource on a hub is addressed by a URI. The following templates are used to reflect the subordinate relations defined above:

- Topic t: */topics/{t}*;
- Child topic t_j of topic t: */topics/{t}/topics/{t$_j$}*;
- Subscription factory of topic t: */topics/{t}/subscriptions;*
- Subscription s of topic t: */topics/{t}/subscriptions/{s}*;

Entangled subscriptions between topic *ta* on hub A to topic *tb* on hub B is established by a user using a web browser following the REST-Event Creation Protocol.

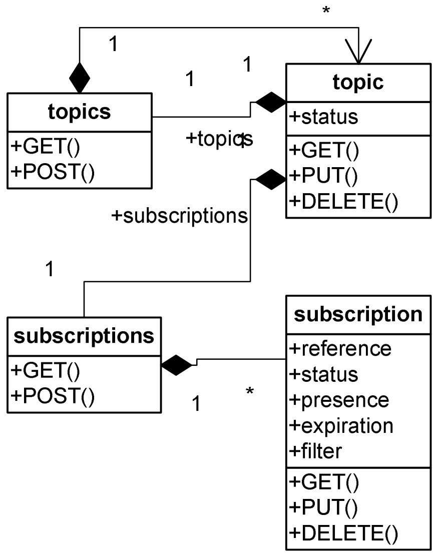


Figure 6. Main resources on topic hub

A notification is propagated between hubs as follows:

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

The topic web 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 [10].

B. 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, 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 [32] 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.

VI. Features of Topic Web

On surface, the topic web built by REST-Event framework, as described in the previous section, appears similar to the broker overlay network in the conventional notification architecture [14]. However, it has the following advantages due to a REST based design.

A. Addressability and Connectedness

Unlike conventional broker overlay networks that are closed systems whose accessibility are prescribed by the APIs, a topic web is open, addressable and connected. Unlike in a 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. 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 areas" 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 ondemand such that alerts from field technicians can propagate to proper sales and supporting engineers who are 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 (subscriptions). 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 can be made 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 [30] 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}

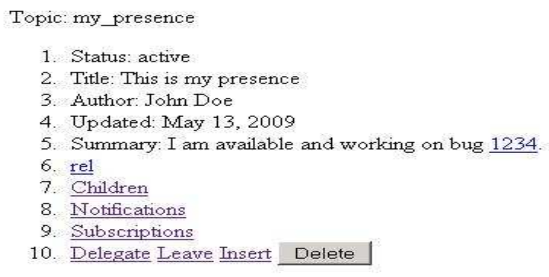
The robustness and scalability also come from the statelessness of REST design. The statelessness means that a topic hub can process any request in isolation without any previous context. By removing the need for such context, we eliminate a lot of failure conditions. In case we need to handle more client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

VII. Implementation and Experiments

A prototype topic web has been developed based on the described REST-Event framework. The notification system allows users within a group to publish and subscribe presence information and text messages. Users can respond to received messages 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 over the event channels so that interested members can set up a new workflow using the proposed topic web.

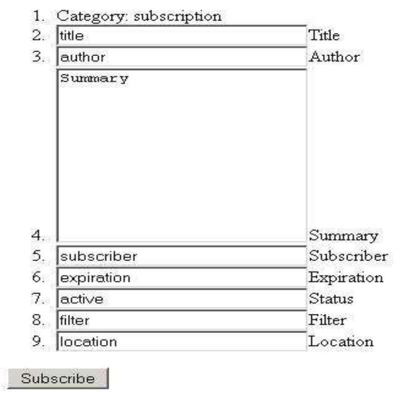
Users interact with the topic web with Web browsers without any download. The following is a screenshot of a web page of a particular topic (Figure **7**).

From a topic page, a user can follow the link to the subscription factory page to create subscriptions (Figure **9** and Figure **9**).


Topic: my_presence

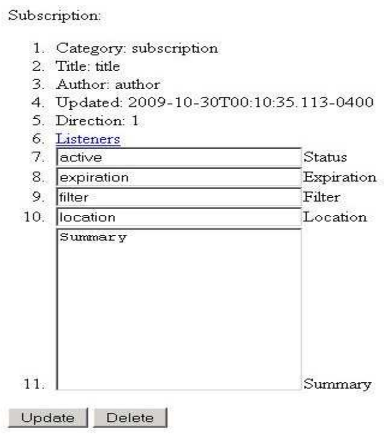
1. Status: active
2. Title: This is my presence
3. Author: John Doe
4. Updated: May 13, 2009
5. Summary: I am available and working on bug 1234.
6. rel
7. Children
8. Notifications
9. Subscriptions
10. Delegate Leave Insert Delete

Figure 7. A topic web page


1. Category: subscription
2. title Title
3. author Author
4. Summary Summary
5. subscriber Subscriber
6. expiration Expiration
7. active Status
8. filter Filter
9. location Location

Subscribe

Figure 8. Page for creating subscription

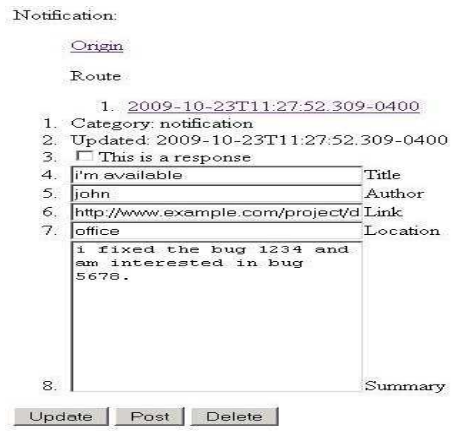

Subscription:

1. Category: subscription
2. Title: title
3. Author: author
4. Updated: 2009-10-30T00:10:35.113-0400
5. Direction: 1
6. Listeners
7. active Status
8. expiration Expiration
9. filter Filter
10. location Location
11. Summary Summary

Update Delete

Figure 9. Page for a created subscription

In this prototype system, a user can post a message to a topic using a web browser (Figure **10**). The topic hub will propagate the message over the event channels. All users who subscribe to the topic directly or indirectly through other topics will receive the message in a notification. In this topic web, notifications for text messages are also modeled as resources that can be linked to track the interaction history. When a user posts a message to a topic, it is saved by the topic hub and all notifications for this message are linked to the original copy. If another user responds to this message, the response is again saved in a topic hub and linked to the original message. A user can follow this response chain through the hyperlinks embedded in the notifications. In some sense, the messages are like tweets. However, the topic web is not a single web site as www.twitter.com. Instead, the topic web is a distributed system consists of many such web sites.



Notification:

Origin

Route

1. 2009-10-23T11:27:52.309-0400

1. Category: notification
2. Updated: 2009-10-23T11:27:52.309-0400
3. ☐ This is a response
4. i'm available — Title
5. john — Author
6. http://www.example.com/project/d — Link
7. office — Location

```
i fixed the bug 1234 and
am interested in bug
5678.
```

8. Summary

Update Post Delete

Figure 10. Page for posting a message

The prototype was written in Java using Restlet 1.1.4 [24]. 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 in Figure **11**.

For this prototype, we used OpenSSL package [31] 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.

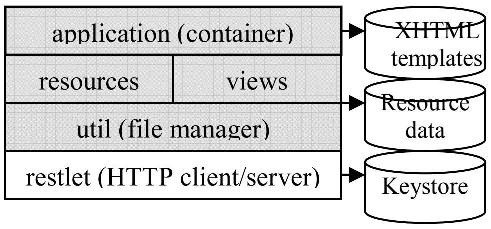


Figure 11. Topic hub stack

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 REST-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.

VIII. CONCLUSIONS

In this paper, we described an approach - REST-Event framework for event-driven web, in which elements of EDA (event-driven-architecture) can be projected and represented by REST resources, protocols and services. The basic REST resources, protocols, services and securities in this framework were specified and constructed. Moreover, a special event-driven web, topic web, was proposed and built based on REST-Event. We studied features in REST-Event approach, including addressability, dynamic topology, robustness, and scalability, etc., and compared them with the conventional notification systems.

In addition, we developed a prototype REST-Event based system using secure HTTP. Preliminary performance tests showed that the proposed approach is feasible and advantageous.

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

REFERENCES

[1] Li Li and Wu Chou: R-Event: A RESTful Web Service Framework for Building Event-Driven Web, Service Computation 2010, pages 7-13, Lisbon, Portugal, November 21-26, 2010.

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

[3] 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: January 5, 2012.

[4] 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: January 5, 2012.

[5] The Atom Syndication Format, 2005, http://www.ietf.org/rfc/rfc4287.txt, Last Accessed: January 5, 2012.

[6] The Atom Publishing Protocol, 2007, http://www.ietf.org/rfc/rfc5023.txt, January 5, 2012.

[7] RSS 2.0 Specification, 2006, http://www.rssboard.org/rss-specification, Last Accessed: January 5, 2012.

[8] Pushlets, http://www.pushlets.com/, Last Accessed: January 5, 2012.

[9] HTML Working Group, 2009, http://www.w3.org/html/wg/, Last Accessed: January 5, 2012.

[10] Hickson, I. (ed), Server-Sent Events, W3C Working Draft 29 October 2009, http://www.w3.org/TR/eventsource/, Last Accessed: January 5, 2012.

[11] Hickson, I. (ed), The Web Sockets API, W3C Working Draft 29 October 2009, http://www.w3.org/TR/websockets/, Last Accessed: January 5, 2012.

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

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

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

[15] 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.

[16] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. West Sussex, England: John Wiley & Sons Ltd., 1996.

[17] 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: January 5, 2012.

[18] Michelson, B. M. (2006). Event-Driven Architecture Overview, http://soa.omg.org/Uploaded%20Docs/EDA/bda2-2-06cc.pdf, Last Accessed: January 5, 2012.

[19] 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: January 5, 2012.

[20] Graham, S., Hull, D., and 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: January 5, 2012.

[21] 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: January 5, 2012.

[22] 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: January 5, 2012.

[23] Fitzpatrick, B., Slatkin, B., and Atkins, M., PubSubHubbub Core 0.2, Working Draft, 1 September 2009, http://code.google.com/p/pubsubhubbub/, Last Accessed: January 5, 2012.

[24] Restlet, RESTful Web framework for Java, http://www.restlet.org/, Last Accessed: January 5, 2012.

[25] JMS (2002). Java Message Service, version 1.1, 2002, http://www.oracle.com/technetwork/java/index-jsp-142945.html, Last Accessed: January 5, 2012.

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

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

[28] 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: January 5, 2012.

[29] 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: January 5, 2012.

[30] Fielding, R., et al. Hypertext Transfer Protocol – HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html, Last Accessed: January 5, 2012.

[31] OpenSSL: http://www.openssl.org/, Last Accessed: January 5, 2012.

[32] The OAuth 1.0 Protocol: http://tools.ietf.org/html/rfc5849, Last Accessed: January 5, 2012.

[33] RFC6202: Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP, http://tools.ietf.org/html/rfc6202, Last Accessed: January 5, 2012.

[34] The Bayeux Specification: http://svn.cometd.com/trunk/bayeux/bayeux.html, Last Accessed: January 5, 2012.

[35] XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH): http://xmpp.org/extensions/xep-0124.html, Last Accessed: January 5, 2012.

[36] XEP-0060: Publish-Subscribe: http://xmpp.org/extensions/xep-0060.html, Last Accessed: January 5, 2012.

[37] Google Wave Federation Protocol: http://wave-protocol.googlecode.com/hg/spec/federation/wavespec.html, Last Accessed: January 5, 2012.

[38] Google Wave Client-Server Protocol: http://wave-protocol.googlecode.com/hg/whitepapers/client-server-protocol/client-server-protocol.html, Last Accessed: January 5, 2012.

[39] Event driven architecture onto the Azure Services Platform: http://www.microsoft.com/belux/architect/issue_3/azure_services_platform.aspx, Last Accessed: January 5, 2012.

[40] Event-Driven Architecture: SOA Through the Looking Class: http://msdn.microsoft.com/en-us/architecture/aa699424, Last Accessed: January 5, 2012.

[41] Wu Chou, Li Li, Feng Liu, Web Services for Communication over IP, IEEE Communication Magazine, vol. 46 no. 3, page 136-143, March 2008.

[42] RFC9588: Web Linking, http://tools.ietf.org/html/rfc5988, Last Accessed: January 5, 2012.