Building Communicating Web Applications Leveraging Endpoints and Cloud Resource Service

Kundan Singh
IP Communications Department
Avaya Labs
Santa Clara, USA
singh173@avaya.com

Venkatesh Krishnaswamy
IP Communications Department
Avaya Labs
Basking Ridge, USA
venky@avaya.com

***Abstract*—We describe a resource-based architecture to quickly and easily build communicating web applications. Resources are structured and hierarchical data stored in the server but accessed by the endpoint via the application logic running in the browser. The architecture enables deployments that are fully cloud based, fully on-premise or hybrid of the two. Unlike a single web application controlling the user's social data, this model allows any application to access the authenticated user's resources promoting application mash-ups. For example, user contacts are created by one application but used by another based on the permission from the user instead of the first application.**

We present aRtisy, a platform to further simplify web application development by using pre-built communication widgets for common use cases such as phone call, conferencing, call distribution and video publish or play. The architecture extends beyond web to native applications and reduces the barrier between web and non-web applications for communication. Our resource access protocol acts as a generic signaling mechanism for the emerging WebRTC (Web Real-Time Communications). We have implemented several applications completely in HTML5 running in the browser using this resource-based architecture.

Keywords-** **Web communication; end-to-end; HTML5; VoIP; WebRTC; WebSocket; RESTful

I. Introduction

A common trend in existing web applications such as photo sharing, social connections, blogs or document sharing is to manage and control the user's data. Although the data belongs to the user, it can be accessed only via one website in many cases. This trend causes several problems:

Redundancy: A user who creates her profile on many websites causes redundancy and fragmentation of data that is hard to manage and keep up-to-date. It would be nice if the user can change her data such as the current location or work place at one place to update her social presence everywhere.

Application lock-in: Websites control what applications may be used for accessing the user's stored data. For example, a user must use the web communicator provided by the website that stores her contacts data, even though she prefers a similar application by another website.

Rigid data boundary: Enterprises often block social websites because the websites do not restrict their data to enterprise-only contacts or interactions, though the software that powers the website can potentially work on the restricted private database.

Tied lifetime of data: The lifetime of the user data is tied to that of the website which tends to become obsolete. For example, the social connections data on MySpace or Friendster could not be reused when the users decided to migrate to Facebook.

Tying the user data to one application significantly limits the use of that data in today's web applications that often include cross site interactions [1]. A user could log in with her Facebook account on a third-party blog or chat with her Linked-In contacts on a web-based instant messenger. Such scenarios are possible if an application can access the user data controlled by another website. Some websites have APIs (Application Programming Interface) [2] to allow other sites to access some data. This is a workaround rather than directly solving the problem by separating the user's data from the application logic that accesses it.

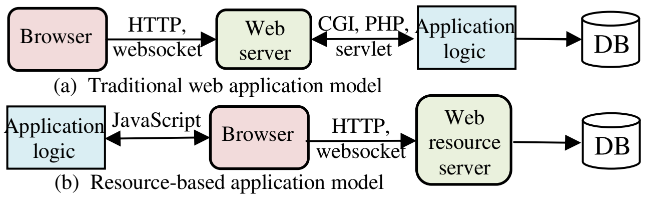


Figure 1. Traditional application model vs resource-based model

A traditional web model runs the application logic on the server side while using the browser as merely a front end device (Fig.1a). A Rich Internet Application (RIA) runs its application logic in the browser using a plugin or native HTML5. A resource-based application model has a data-centric RIA and a resource service (Fig.1b). Our web resource server (or *resource service*) exposes a generic protocol and API to access the data independent of any specific application. The resource service and the web application code can be separately managed, e.g., a user who visits the public social website from an enterprise network sees only the private connections and interactions stored on the enterprise resource service, or a user stores her profile and contacts on a cloud resource service but gives permission to other e-commerce or social sites to access them. We apply the resource model to build communicating web applications involving audio, video, text and presence.

We have built a developer platform called *aRtisy* to easily create such applications. Building communicating applications is hard. Although, the emerging WebRTC (Web Real-Time Communications) [3][4] brings real-time multimedia to the browser, developers still need to handle signaling, security, robustness, scalability and interoperability with legacy telecom equipments. Moreover, to help developers think in terms of simple application concepts rather than complex protocols, we identify common use cases as *communication widgets*. Each widget contains one application scenario, e.g., a phone call, multiparty conference, call distribution, or video publish or play. The platform allows interconnecting the widgets for quick application prototyping. The resource service also forms a generic signaling framework for WebRTC applications because a proprietary signaling is not suitable for cross site interactions.

We present related work and a background on the resource service in Sections III and II, respectively. Section IV describes our developer platform and communication widgets. We present RIAs such as video chat, instant messenger and video presence in Section V. Section VI describes cloud and on-premise deployment challenges, e.g., security, access control and interoperability with existing communication systems. Finally, our conclusions and future work are in Section VII.

II. RELATED WORK

The resource-based application model builds on a very old software engineering practice of separating the application data from the logic, e.g., MVC and MVP (model-view-controller or presenter) design patterns [5][6]. It moves this concept from a software application to the end user – the user stores her data on one system and uses the business logic and web interface on another. Recent proliferation of social web applications has amplified the need for such architectures [1][7]. Some websites let others access its data [2][8] but are limited due to its pair-wise application-centric sharing. We need a user-centric way to store and subscribe to the structured data as shown in Fig.2(c). We show the application in the browser in this paper, even though it can potentially run server side on another web server.

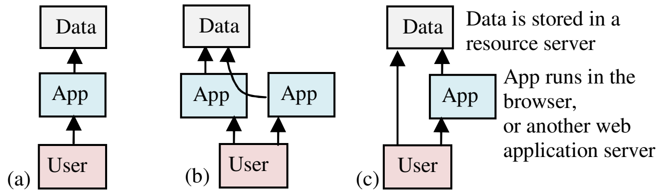

Data
App
User
(a)
Data
App
App
User
(b)
Data
App
User
(c)
Data is stored in a resource server
App runs in the browser, or another web application server

Figure 2. (a) A single application controls the user data, (b) An application allows another to access its data, (c) The user controls her data/approves apps

The concepts of distributed file systems can be applied to web applications [9][10][11]. Cloud storage such as drop-box, Amazon S3 and Google Drive can store files via web APIs. Web browsers can synchronize local application data with storage servers [12][13][14]. Some community projects address the social data's privatization problem [15][16][17]. Menagerie [18] allows sharing user's data across web applications using hierarchical naming. BStore [19] has a global web-accessible storage where the user controls which applications get access to her data. Any of these recent cloud storage systems can be modified to work in the resource architecture.

The push-based protocols used in publish/subscribe systems [20][21][22] can be modified to dispatch resource change events to the web application. The remote SharedObject in Flash Player [12] lets a programmer see synchronized data across browsers. Several large scale systems use data-centric middleware [23][24] for loose coupling among applications for scalability and robustness. Hookflash [25], a cloud framework, brings structured data to web and mobile applications using a peer-to-peer protocol. The choice of a particular protocol is not important as long as it supports data access and change events in the data-oriented programming model [26].

Our system needs simple primitives such as hierarchical access-controlled data, and the ability to store an object or array resource as persistent or transient data. The latter concept is particularly unique in our implementation and helps in creating robust communication applications without worrying about explicit cleanup of the transient resources when the browser terminates. We need a combination of user and application level access control so that resources can be owned by the user or the application, or both.

Communication widgets have been used in websites for a long time. Many instant messengers have widgets to enable multimedia communication from the browser such as Google Chat. Legacy telecommunication services are widgetized [27] to control the external phones or servers from web. Many conferencing and cloud telephony providers are embracing WebRTC [3] by creating APIs to embed their communication widgets in third-party customer websites. Unlike these that interface with external communication or application logic, we create resource-based widgets that include the application logic. Our video-io widget for WebRTC is inspired by an earlier work [28] that uses a Flash-based video-box application for many communication scenarios. Researchers [29] have created resource APIs for web conference signaling that can use such widgets. Interworking with the legacy communication systems is done in the resource service transparently to the application.

The novelty of our work is in (1) showing several non-trivial communicating applications entirely in HTML5 using the resource model and (2) identifying the widgets covering common use cases that can be interlinked to create a complete application. The resource service can be hosted in the cloud or on-premise, independent of the web application code.

III. BACKGROUND: RESOURCE SERVICE

A resource-based API was proposed in [29] for a web conference application. Here, we extend the work to generalize and apply it to many more applications. This section describes the resource service and the resource access protocol we use.

Resources are hierarchical data identified by relative paths on the server or absolute URLs with the server location. For example, /room/1234 is a chat room's resource. A structured resource is represented in JSON (JavaScript Object Notation), e.g., {"status": "busy", "message": "in a call"} is a user's presence status. A resource service provides generic resource access using HTTP verbs such as GET, PUT, POST and DELETE, while leaving the resource semantics to the client application logic.

The web application in the browser connects to the resource service over a persistent WebSocket connection [30]. It serves two roles: a channel to send asynchronous events from the server to the client, and a context to scope transient resource. The resource access protocol over WebSocket uses JSON message format with applicable attributes such as request method, resource path, data type of the resource, response code, event type, reason for failure and the actual entity (see Fig.3). The msg_id field is used to co-relate the response with the request. A persistent connection loses certain benefits such as caching of a pure stateless RESTful (Representation State Transfer) system, but may be compensated in the protocol, e.g., by using the if-modified-since attribute in the GET request.

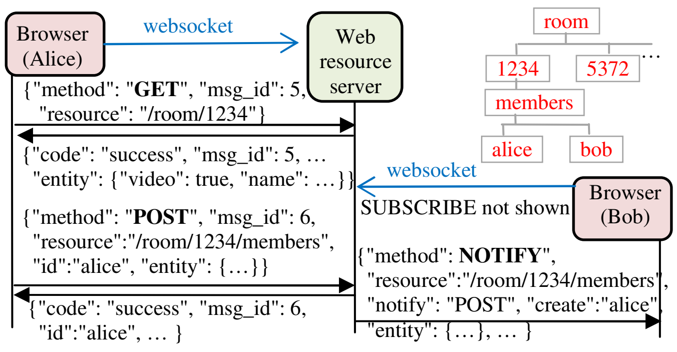

Browser (Alice)
websocket
Web resource server
{"method": "GET", "msg_id": 5, "resource": "/room/1234"}
{"code": "success", "msg_id": 5, … "entity": {"video": true, "name": …}}
{"method": "POST", "msg_id": 6, "resource":"/room/1234/members", "id":"alice", "entity": {…}}
{"code": "success", "msg_id": 6, "id":"alice", … }
room
1234
5372
…
members
alice
bob
websocket
SUBSCRIBE not shown
Browser (Bob)
{"method": NOTIFY", "resource":"/room/1234/members", "notify": "POST", "create":"alice", "entity": {…}, … }

Figure 3. Example of data access and event notification on a resource tree

Table I summarizes the methods. PUT creates or overwrites a resource. POST creates a child resource under a parent's path. It fails if the request has child identifier and that child resource path exists. This is useful to avoid accidental overwrite, e.g., for new user signup with the same identifier. An application typically uses a resource path to identify a structured object or an array, but not both, e.g., /room/1234 is a structured object but /room is an array containing an ordered list of child identifiers including 1234. POST is also used to append a child to an array resource where the child identifier is irrelevant so that the resource service can create a unique identifier. GET returns the object resource with its attributes or the child identifiers of the array resource. To support pagination and search on arrays GET /users?like=alice% returns matching child identifiers starting with alice and /my/messages?offset=20&limit=10 returns third page of messages. We support attribute reference in GET and PUT, e.g., PUT /room/1234[video] modifies only the video attribute. DELETE is used to delete a resource or a resource sub-tree.

To support asynchronous events, e.g., another participant joins the room or an incoming instant message is received, we have SUBSCRIBE, UNSUBSCRIBE and NOTIFY, e.g., subscribe to /room/1234 and get notified when this resource or any of its immediate children changes. The client-initiated NOTIFY gets delivered end-to-end to all the subscribers of that resource.

TABLE I. METHODS IN THE RESOURCE ACCESS PROTOCOL

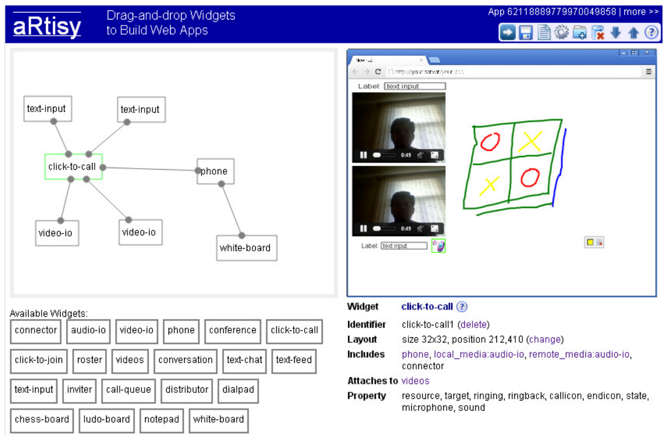
Method	Purpose
POST	Create a child resource if it does not exist. Append a child item under a parent array.
PUT	Create or update an object resource or its attribute. Modify a resource from persistent to transient or vice-versa.
GET	Read an object, an array or an attribute of an object resource.
DELETE	Delete an object resource or a resource sub-tree. Automatically for all transient resources on disconnection.
SUBSCRIBE	Subscribe to receive change notification of a resource or its immediate children.
UNSUBSCRIBE	Remove a previous subscription at the server. Automatically for all subscriptions on disconnection.
NOTIFY	(server to client) Indicates data change per subscription. (client to server to client) End-to-end message passing.

Each resource is created as either transient or persistent. A transient resource is automatically deleted by the server when the client connection that created the resource is disconnected. This enables automatic cleanup and robustness against browser crashes. When Alice closes her browser, her participant resource at /room/1234/members/alice is automatically removed, and all the other subscribers of members are notified.

A web developer sees a resource as a data object or array that are accessed (create-read-update-delete) or subscribed, to receive data change events, similar to the bindable objects and properties in ActionScript. For instance, a list display can bind to an array resource path, /room/1234/members, and update the list items based on the received data events on this resource.

IV. Communication Widgets

Identifying communication widgets helps reuse of common communication scenarios across applications and lets the developer think in terms of simple use cases rather than complex protocols and messages. Fig.4 shows an example application development in progress in our developer platform, *aRtisy*. It shows the interconnected widgets on the left and the editable visual layout of the application on the right.



Widget	click-to-call
Identifier	click-to-call1 (delete)
Layout	size 32x32, position 212,410 (change)
Includes	phone, local_media:audio-io, remote_media:audio-io, connector
Attaches to	videos
Property	resource, target, ringing, ringback, callicon, endicon, state, microphone, sound

Figure 4. Screenshot of aRtisy application builder showing interconnected widgets for a video phone application with shared white-board.

A. Widget Properties and Interconnections

A *web communication widget* is a collection of HTML, JavaScript and CSS (cascading style sheets) code to implement an application scenario. It is used as a standalone application or interconnected with other widgets or code to create a complete application. A widget has properties, methods and allowed interconnections to control or indicate its behavior, e.g., the conference widget is controlled by a resource path and a joining user name, and may interconnect with click-to-join. An instance in a running application is called a widget item. Two widgets interconnect via inclusion, attachment or property attachment.

Inclusion: Suppose A and B are widgets and A' and B' are their widget items respectively. When A *includes* B, then A' contains a reference to B'. If the application does not assign this reference then a new B' is automatically created and assigned when A' is constructed during application initialization. Thus, B is required for A to work, and will be created if needed. Most widgets include a connector to connect to the resource service.

Attachment: When A *attaches to* B, then A' can work without B', but if assigned, can delegate some behavior to B'. For instance, a text-chat can work as a standalone text-only chat application. It attaches to a conference or phone to facilitate text chat in a multiparty conference or two-party phone call.

Property attachment: In an application, an item's property may be set as constant, assigned from URL or set via a text input by the user. When a property of A is attached to B, then B' is a user input control to set or display that property of A'.

Table II lists the aRtisy widgets and the relative software complexity, not including the third-party libraries for JSON, MD5 and Base64. Our Python-based resource server has less than thousand lines of code and uses [31] and PostgreSQL. Typically, a widget that can have user interface is included in an iframe and referenced by its iframe Window. Headless widgets are included as scripts and referenced as instantiated JavaScript objects. We define setProperty and getProperty methods in each widget to control its properties, and dispatch a propertyChange event to indicate any change to the parent.

B. Connector to Resource Service

The connector widget is used to connect to the resource service for various data access and event notifications (Section II), and is included by almost all other widgets. An application can set its url property to a separate resource service.

TABLE II. CURRENT LIST OF COMMUNICATION WIDGETS. THE APPROX SOURCE LINES OF CODE COMPARES RELATIVE SOFTWARE COMPLEXITY.

Widget	Purpose	LoC
connector	Connect to the resource service for data access and events. Most others include a connector.	400
video-io, audio-io	A video box to publish or play a media stream. The two widgets share most of the code.	1250
phone	Headless call signaling/call state for two-party call.	400
conference	Headless control and membership management for multiparty conference.	350
click-to-call	A button to initiate, answer or terminate a call, and includes a phone and two audio-io widgets.	450
click-to-join	A button to join or leave a conference, and includes a conference widget.	200
roster	A roster display of participants or contact list, and attaches to conference.	650
videos	A video conference layout which inclues zero or more video-io, one per participant.	700
conversation	A multimedia chat box includes conference, roster, videos, text-chat; also has file-sharing, emoticons.	350
text-chat	A text chat application that includes text-feed and can attach to conference or phone.	300
text-feed	A list display of a dynamic array resource of text messages or comments.	350
text-input	A user input control to set or indicate a widget's property via property attachment.	150
inviter	Extends phone and includes conference – enables inviting a target user to a conference.	400
call-queue	Extends phone but puts incoming call to a pending calls list to answer by the end user.	400
distributor	Extends phone to do automatic call distribution to one or more targets in parallel or sequence.	350
dialpad	A touch-tone keypad interface that attaches to click-to-call or click-to-join.	150
notepad	A shared notepad with real-time updates of text, and attaches to a phone or conference.	500
white-board	A shared white-board with real-time updates of drawings, attaches to a phone or conference.	300

A widget hides the implementation details or application requirements while exposing easy to understand properties and interconnections. For example, only audio-io and video-io deal with WebRTC. If the user's browser does not have WebRTC, the application should fall-back to alternative technologies such Flash Player [12]. Thus, fall-back changes are localized to only these two widgets while preserving all the applications that used them. Similarly, only the connector widget deals with WebSocket [30], and if the browser lacks its support, can be changed to use another channel such as [32][33].

C. Widgetizing Audio and Video for WebRTC

The emerging WebRTC standards define protocol and APIs to bring real-time communication to the browser [3][4]. It allows an application to create a Peer Connection between two browsers using ICE (Interactive Connectivity Establishment) [34] to exchange real-time media streams such as live audio and video. The signaling path involves negotiation of session parameters (offer-answer) and ICE candidates, but is left to the application. In these early days of WebRTC, a website tends to create its own signaling path limiting cross-site mash-ups.

In the Flash-platform, a developer can easily build many communication scenarios such as two-party calls, multiparty conference, live broadcasts or panel discussions by setting a couple of properties on a simple video box [28]. Similarly, our video-io widget for WebRTC has a video box to either publish the locally captured camera and microphone or play a remote stream or a media file from the web, while hiding the signaling negotiations. Its publish or play property is set to the resource path identifying a real-time stream to publish or play, respectively. The src property allows playing stored media file similar to an HTML5 video element [35]. It defines other properties to enable or disable the audio or video input or output. The servers property is an array of ICE server locations.

This widget creates a peer connection from a publisher to the players on the same resource path. When its publish property is set to /path/to/stream1, it prompts the user for device access in the browser. Once approved, the widget becomes a publisher, creates /path/to/stream1/publisher and listens for any change on /path/to/stream1/players. When another widget's play property is set to the same path, it adds an array item under /path/to/stream1/players and becomes a player. The resource service informs the publisher about the new player. The publisher initiates a new Peer Connection [3] to the player. The signaling message containing WebRTC negotiation is sent using NOTIFY on a resource. The player sends its message to the publisher's resource path, and vice-versa. The message flow of Fig.5 shows one publisher and two players.

We modify the publisher to subscribe to the player's resource path to receive NOTIFY so that all the messages related to one peer connection are exchanged on a single resource path preventing accidental interference. A sender attribute separates publisher-to-player messages from player-to-publisher. When the player detects a change in the publisher resource, it resets its previous peer connection and waits for new negotiations. When the application clears the publish or play property or the user closes the browser terminating the connection, the transient resources for the publisher or player are removed, informing the other party, which resets its side of the widget.

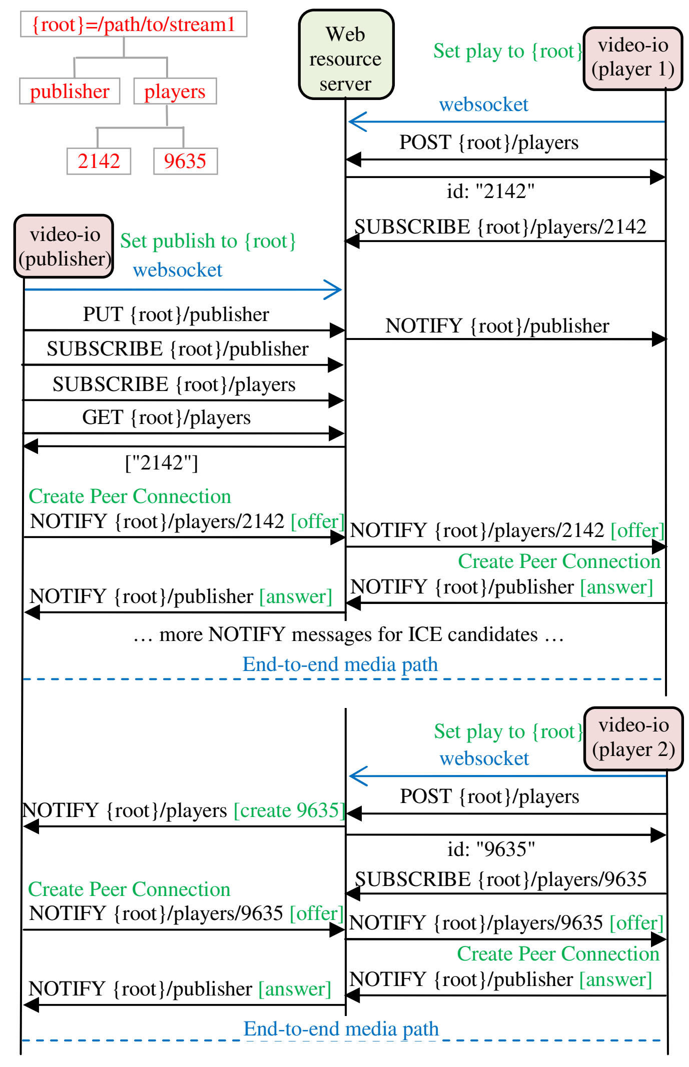


Figure 5. An example message flow with video-io widgets for publish and play using WebRTC. For brevity, the irrelavant messages and JSON format are not shown and the resource path of /path/to/stream1 is replaced by {root}.

Our resource design allows the publisher and players to arrive in any order and enables the publisher to stream to any number of players. The widget hides the application messages while exposing the simple concepts of video publish and play. Many widgets can be launched as standalone web applications because their properties are settable via URL parameters, e.g., visit /sdk/video-io.html?publish=/path/to/stream1 to start publisher.

The audio-io widget is similar to video-io except that it deals with only audio, and may not need a display. The three widgets namely, connector, video-io and audio-io, are sufficient to build many application scenarios. For example, a four-party video conference application has one publish and three play video-io widgets all using the same resource path in one conference.

D. Telephony-centric Widgets

The telephony-centric widgets emulate existing telephony use cases, e.g., phone is a headless widget for two-party calls. It has two controlling properties, local and target subscriber's resource paths, e.g., /path/to/alice. It sends signaling messages such as invite, accept, reject, cancel and end using NOTIFY on the target resource path. Additionally, the data message is used to send application-specific external data such as text chat or session negotiation between the two parties. It uses a unique call identifier for all the messages in one call attempt. The caller's resource path is included in the invite message, and a reason text in the reject message. The phone's state machine is shown in Fig.6, and an example message is {"type": "invite", "invite-id": "5324", "from": "/path/to/caller"}

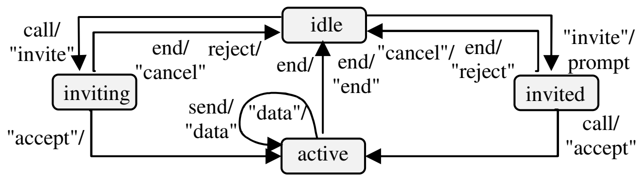


Figure 6. State machine implemented by the phone widget. The NOTIFY messages are in quotes such as "invite", and the user actions/prompts without.

The click-to-call widget is a web audio phone with a button to initiate or answer a call. It delegates the call control logic to the included phone widget, and the WebRTC-based media processing to the two included audio-io widgets, one for publish and another for play, i.e, it binds the phone's resource property to the publishing audio-io's publish property, and its target_ resource property to the playing audio-io's play property.

A web developer may modify the widgets to change the web phone's behavior. For example, interconnecting the audio-io references to video-io widgets makes it a video phone while using the same call control. The phone reference be set to another telephony widget such as a call-queue or automatic call distributor. The call-queue widget extends phone to put a received call invitation in a list displayed to the user, and lets the user select a call to answer or decline. The distributor widget extends phone to allow multiple target resources. When a call is placed, an invite is sent to all the target resources either in parallel or sequence based on its distribution property. These telephony widgets are compatible with phone using the same messages and can be used wherever the phone widget is applicable.

The conference widget represents a multiparty conference bound to a resource path under which all its conference and membership resources are created. When multiple widgets set their resource property to the same path, they subscribe to the same conference and listen for change in its members. If resource is set to /path/to/conf1, then /path/to/conf1/members is an array resource of its members. When the me property is set to local user's data, the widget joins the conference on the user's behalf and adds the data in the members array. These two properties are sufficient to control the widget. Additionally, it defines join_id and myid to control and indicate the local member's child identifier, if needed.

The phone and conference widgets have the core telephony concepts useful in many applications. The click-to-join widget provides a button to join or leave a conference. The inviter extends phone for use in a conference, i.e., a user invites another one in the conference using the same protocol as phone.

E. Web-centric Widgets

The roster widget displays a list of users, their presence statuses and custom messages based on an array resource path.

It can show contact list in an instant messenger or display the membership list when attached to a conference.

The videos widget supports video conferencing bound to a resource path and includes video-io items, one per participant. If attached, it enables video in an otherwise audio-only conference or phone. It automatically shows tile or picture-in-picture layout based on the participants' count. The application may override the layout by supplying a static set of video-io widgets, e.g., in a fixed four-party video conference with 2x2 layout.

The text-chat widget implements a multi-party text chat. It includes a text-feed to show the messages in an array resource, and a text input control to post a message to that resource. The message object contains the sender information, the date/time of creation, the message type and the actual message either in plain text or HTML. The persistent property determines whether the message resources are persistent or transient. The chat history storage and access is implicit in the resource model. The text-feed widget can display any list of messages such as forum posts or comments on a page.

The text-chat widget also includes popular instant messenger features such as emoticons, is-typing indication and file sharing using the resource model. A user can drag a file from her desktop to this widget to share it with other connected users. The file content is encapsulated as a data URL in the chat message resource, without an explicit file sharing protocol. The is-typing message in NOTIFY is sent on the conference resource to all the participants when a user starts or ends typing.

The conversation widget includes conference, text-chat, videos and roster for a complete multimedia conversation. Similar to conference, it defines the root resource path and myid to subscribe and join a conference path. The user interface allows initiating audio and video, or sending text messages in a conversation. The call_type attribute in the resource object indicates whether this conversation has audio, video or text-only, to benefit a new participant. The resources may reuse the centralized conference data model [36].

We have implemented other web centric widgets such as shared notepad and white-board, e.g., using HTML5 SVG (Scalable Vector Graphics), to enable rich interactions. These widgets (videos, text-chat, notepad or white-board) may be linked to a conference or phone, or a compatible extension. The attachment determines whether the application logic uses the conference member's or the phone subscriber's resource path.

The ability to interconnect smaller use cases to build an application is a very powerful and flexible way to create modular and extensible applications such as video blog, video presence, online games with interactions, "go to a page to meet", or "do something on the web together with your friend".

V. Example Applications

We describe some rich Internet applications we have built using the resource model.

A. *Public Chat Service*

When a user visits the website, the client application joins the public room's resource, i.e., /apps/public-chat/public. Each room has two child array resources, userlist and chathistory, for the current participants' list and chat messages. The application subscribes to these resources to display the user roster and chat history. It allows video conferencing using fixed 3x2 layout of video-io widgets. Unlike existing symmetric video conferencing, it allows asymmetric media paths – each user decides who she wants to see and hear from. A user can send private message to another user via a direct NOTIFY on the target user's resource instead of a POST to chathistory. We disallow listing /apps/public-chat, hence one can use a private room identity instead of public.

B. *Instant Messenger and Communicator*

The communicator application is in HTML5 and does not use legacy SIP [37] or XMPP [20]. A user signup creates a user resource, e.g., /users/venky@avaya.com. All the contacts and profile data such as presence status are rooted under this. A roster widget binds to the user's contacts. The conversation widget does multimedia chat starting as two-party and extending to multiparty when the user drags more contact items to the conversation window. Each user has an inbox resource for offline messages and missed call invites. Each conversation binds to a message thread resource. Our design allows merging conversations, e.g., when an incoming thread is received from a person for which the user already has a conversation.

The user profile and conversation data are not controlled by a single application. For example, another application shows a presence icon bound to the target user's presence resource, and can be embedded in other places such as corporate directory or user's home page. Clicking the icon opens a conversation widget which seamlessly interacts with the target user's communicator because both applications share the same resource design.

C. *Video Presence*

The video presence web application shows a list of contacts based on an array resource path managed by the owner. Each item in the list is the live video feed from that contact. The owner can click on a contact's video to initiate a voice (and text) call. Our web application uses video-io in the list item, and conference, inviter, text-chat and one or more audio-io in a call.

We want to keep the capture frame rate very small for video presence, but bump it up during the call. Due to lack of such controls in the existing WebRTC APIs, we created a desktop application using Adobe Integrated Runtime (AIR) and the same resource paths, but with Flash-based media [28] instead of WebRTC. It provides certain benefits for this use case – lower bandwidth when not in a call due to frame-rate control, and lower upstream bandwidth in a call due to client-server media path instead of full-mesh of WebRTC. The resource model extends beyond web to native applications and gives seamless interoperability between the two.

D. *Personal Wall*

This is an enterprise social network where a person visits someone's wall to post messages, or share calendar events, business cards or files. All the shared data are resources with appropriate permissions. The wall owner can enable her camera or upload a media file using video-io for her video presence to a visitor. We will incorporate contextual sharing of enterprise data [38], e.g., when Alice visits Bob's wall, she could see the last few conversations and shared files they have had outside the application, e.g., via email or phone calls.

These applications are fairly small with less than a thousand to a few thousand lines of code, but demonstrate a wide variety of application use cases built in the resource model. We use Google Chrome on desktop that contains HTML5 support of WebSocket and WebRTC. Our widgets also work on Ericsson's Bowser (iOS) that does not support WebRTC APIs in iframes, and requires non-trivial hacks to move them to the top-Window.

WebRTC is still evolving, particularly in media recording and peer-to-peer data channel. We deliberately left out examples of widgets and applications based on these features. We expect them to play crucial role in our future work.

VI. Challenges: Cloud and On-Premise

We show challenges and potential solutions in deployments that are on cloud or on-premise, or a hybrid where an enterprise has the private resource service accessed by public websites.

A. *Security and Access Control*

The transport security is provided by secure WebSocket over TLS for client-server, and encrypted peer-to-peer media path in WebRTC. A traditional server-side web application that hides the user data typically authenticates the user before allowing access. Once authenticated, the application ensures access of only the authorized user data. Separating the data from the application in the resource model requires separate end user and application authentication so that a resource may be owned by a user or an application, or both. Existing systems [39][19][18][15][16][9][11] already define several access control mechanisms for web resources inspired by Unix file permissions. Our summary of requirements in the context of resource model follows.

Each resource has an owner and an optional group. A user belongs to zero or more groups. The ownership/permissions are set when a resource is created, and not changed afterwards. A resource has three permission-sets for owner, group and others, each with flags to read, write, append, traverse and send-event. A read permission is needed to GET a resource, a write permission to PUT an object resource or POST or DELETE a child resource, and a traverse permission to access the child resources in the hierarchy. The write and append permissions are separate because certain array resources such as inbox should allow append from others but not write. The send-event flag allows sending end-to-end messages on that resource. By default a resource's owner and group depend on the creator. If the setgid flag is set on the parent, a new child resource inherits the owner and group from its parent, useful for creating public repository.

The user owns /users/{user-id} and the application owns /apps/{app-id}. The resources under /users/{user-id}/apps/{app-id} are application specific resources for this user, and must be authenticated by both. A soft-link /apps/{app-id}/users/{user-id} points to this, so that an application can manage all its users under its resource path. When a webpage tries to access a protected resource, the service asks for authentication, and remembers the authorized identity (user or application) for a subsequent access on that connection. The user authentication is directly between the browser and the resource service. The application authentication involves the web server that hosts the application code to generate the secure token based on a private application key [2].

B. *Security and Cross Domain*

While the user trusts the resource service with her data, she may not trust the web application. A naïve resource service may allow the connector from any cross-domain website to connect [40][41]. This poses a security threat from the web application which may secretly deliver sensitive user data such as password to the hosting website. One solution is for the connector to load the real-connector code from the resource-service website in a separate Window and use postMessage to communicate between the two [35][19]. The real-connector prompts for password, hides all the sensitive data and co-relates the website origin with the data access request and its response. Other solutions are possible, e.g., a browser plugin or extension for the connector to deliver the webpage signature to the resource service that allows only authorized web pages to connect, or a proxy at the resource server to load the web application but prevent any data back to the website. A detailed analysis of cross-origin attacks is for further study.

C. *Robustness against Failures*

Failure in client, server or network is possible. A transient data, e.g., presence or call member, is automatically deleted and others notified when the owner disconnects, giving robustness against browser crashes. We can apply existing techniques for robustness as follows. A client can attempt to reconnect with random exponential back-off to reduce churn after a server failure. Connecting to multiple servers reduces failover latency but requires sophisticated database replication and notification. Periodic keep-alive can detect temporary network failures. Finally, with the controlled and restricted API of the resource service, the server is shielded from the bugs in the application logic such as race-condition or dead-lock.

D. *Interoperability with SIP/RTP*

Existing enterprise communication systems are based on Voice-over-IP protocols such as SIP and RTP [37]. Unlike our loosely coupled widgets such as phone, conference and video-io, the session oriented VoIP systems tightly couple the call signaling session with media streams. For instance, changing from voice-only to audio-video call not only requires adding a new media stream over RTP, but also re-negotiating the session via SIP. This makes it difficult to transparently interoperate between our widgets and SIP/RTP, e.g., translation must know whether the phone widget is attached to audio-io or video-io.

We plan to solve this by using server-side widgets to mimic web-widgets and do interoperability. For example, a gateway adaptor could connect to the resource service, and provide a resource path, /dev/sip. The resource service then delivers any access request under this path to the adaptor. When the adaptor detects an invite message on its resource, it initiates a corresponding SIP INVITE request. The concept of mounting a path to an adaptor has been practiced before [19]. The interoperability details are for further study.

VII. Conclusion and Future Work

The resource-based model enables data-centric design and data-oriented programming in web applications. It provides three main benefits over today's social websites: user-centric (unlike app-centric) storage and data management, easy data sharing beyond a single application to facilitate mash-ups, and separation of the application logic from the data during run-time, e.g., an enterprise could host its own a resource service for the enterprise-only connections and interactions. Moving the application logic to the endpoint enables scalable and robust servers because the servers do not have many stateful processes. The advantages are more applicable to the web applications requiring cross-site communication mash-ups.

We have shown that many communication scenarios can be built completely in HTML5 running in the browser. We described an application independent signaling for WebRTC using video-io. With more social business websites involving real-time interactions [42], we expect more such signaling protocols and libraries to appear in near future. A drawback of our approach is that any visitor can potentially steal the application logic by viewing the webpage source code.

We continue to expand our inventory of widgets in aRtisy. We have built widgets beyond traditional communication such as shared boards for Chess and Ludo games that let the users play without imposing game rules. We are building gateway adaptors for SIP and XMPP-based systems and other enterprise applications, e.g., mail exchange and corporate directory to enable web developers to view those as resources. We plan to explore plug-n-play application models, e.g., a user could use her home webcam or current location as resources to attach in an application such as home security.

REFERENCES

[1] T.Berners-Lee, "Socially Aware Cloud Storage", Notes on web design, Aug 2009, http://www.w3.org/DesignIssues/CloudStorage.html

[2] E.Hammer, "The oAuth 1.0 protocol", IETF RFC 5849, Apr 2010.

[3] WebRTC 1.0: Real-time communication between browsers, W3C Working Draft, Aug 2012, http://www.w3.org/TR/webrtc/

[4] Real-Time Communication in WEB-browsers (RTCWEB) IETF working group, http://tools.ietf.org/wg/rtcweb

[5] T.Reenskaug, "Thing-model-view-editor", Xerox PARC technical note, May 1979.

[6] S.Goderis, "On the separation of user interface concerns: a programmer's perspective on the modularisation of user interface code", PhD Thesis, Vrije Universiteit Brussels, pp.15-21, 2008.

[7] E.Naone, "Who owns your friends?", MIT Technology Review Magazine, Jul/Aug 2008.

[8] D.Hardt, "The oAuth 2.0 authorization framework", IETF RFC 6749, Oct 2012.

[9] A.M.Vahdat, P.C.Eastham and T.E.Anderson, "WebFS: a global cache coherent file system", Technical report, UC Berkeley, 1996.

[10] B.Callaghan, "WebNFS client specification", RFC 2054, 1996

[11] F.Hsu and H.Chen, "Secure file system services for web 2.0 applications", ACM Cloud Computing Security Workshop, Chicago, IL, Nov 2009.

[12] Remote shared objects in Flash Player, http://livedocs.adobe.com

[13] M.B.de Jong and F.Kooman, "remotestorage", IETF internet draft (work in progress), Dec 2013

[14] SyncFileSystem API, The chromium projects, 2013.

[15] The DataPortability project to connect, control, share and remix, 2007-2009, http://dataportability.org

[16] Unhosted web apps: freedom from web 2.0's monopoly platforms, http://unhosted.org

[17] The diaspora project: a privacy aware, personally controlled, distributed open source social network, 2010, http://diasporaproject.org

[18] R.Geambasu et al., "The organization and sharing of web service objects with menagerie", World Wide Web Conference (WWW), 2008.

[19] R.Chandra, P.Gupta and N.Zeldovich, "Separating web applications from user data storage with BStore", USENIX Conference on Web Application Development (WebApps), Boston, MA, Jun 2010.

[20] P.Saint-Andre, "Extensible messaging and presence protocol (XMPP): instant messaging and presence", IETF RFC 3921, Oct 2004.

[21] Pubsubhubbub: a simple web-hook based pubsub protocol and open source project, 2008, https://code.google.com/p/pubsubhubbub/

[22] Distributed publish/subscribe event system, open source project, 2007, http://pubsub.codeplex.com/

[23] OMG Data Distribution Service (DDS) for real-time systems, Dec 2004, http://www.omg.org/spec/DDS/

[24] M.Hapner et al., "Java messaging service (JMS) API 1.1", JSR-000914, Sun Microsystems, Apr 2012.

[25] R.Raymond, "Open peer – a proposed peer-to-peer signaling for WebRTC", Hookflash whitepaper, 2012.

[26] R.Joshi, "Data-oriented architecture: a loosely coupled real-time SOA", Whitepaper, Aug 2004, http://www.rti.com

[27] X.Wu and V.Krishnaswamy, "Widgetizing communication services", IEEE International Conference on Communications (ICC), May 2010.

[28] K.Singh and C.Davids, "Flash-based audio and video communications in the cloud", Technical Report, Jun 2011, arXiv:1107.0011 [cs.NI]

[29] C.Davids et al., "SIP APIs for voice and video communications on the web", International conference on principles, systems and applications of IP telecommunications (IPTcomm), Wheaton, IL, Aug 2011

[30] The WebSocket API, W3C candidate recommendation, Sep 2012, http://www.w3.org/TR/websockets/

[31] Pywebsocket project: webSocket server and extension for Apache HTTP Server, http://code.google.com/p/pywebsocket/

[32] I.Paterson et al., "Bidirectional-streams over synchronous HTTP (BOSH)", XEP 0124, version 1.10, Jul 2010, http://xmpp.org

[33] The Channel API for Google App Engine, Mar 2012, https://developers.google.com/appengine/docs/java/channel/

[34] J.Rosenberg, "Interactive connectivity establishment (ICE): a protocol for network address translator (NAT) traversal for offer/answer protocols", IETF RFC 5245, Apr 2010.

[35] HTML5: a vocabulary and associated APIs for HTML, W3C candidate recommendation, Dec 2012, http://www.w3.org/TR/html5

[36] O.Novo et al., "Conference information data model for centralized conferencing (XCON)", IETF RFC 6501, Mar 2012.

[37] J. Rosenberg, et al., “SIP: Session Initiation Protocol”, IETF, RFC 3261, June 2002.

[38] K.K.Dhara et al., "Reconsidering social networks for enterprise communication services", IEEE Globecom, Florida, Dec 2010.

[39] Web access control, http://www.w3.org/wiki/WebAccessControl

[40] Cross-origin resource sharing (CORS), W3C candidate recommendation, Jan 2013, http://www.w3.org/TR/cors/

[41] D.Fernandez, "Cross-domain REST calls using CORS", Blog article, Nov 2012, http://blogs.mulesoft.org

[42] A.Lepofsky, "Social business 2013: less talking, more doing", Constellation Research, Dec 2012, http://www.constellationrg.com/blog