International Journal of Semantic Computing
© World Scientific Publishing Company

USER REACHABILITY IN ISLANDS OF WEBRTC COMMUNICATION APPS

KUNDAN SINGH

Avaya Labs Research, 4655 Great America Parkway
Santa Clara, CA 95054, USA
singh173@avaya.com
http://kundansingh.com

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Recent progress in Web Real-Time Communication (WebRTC) promotes multi-apps environment by creating islands of communication apps where users of one website or service cannot easily communicate with those of another. We describe the architecture and implementation of a multi-platform system to do user reachability in multiple communication services where users decide how they want to be reached on multiple apps, e.g., in an organization that has voice-over-IP, web conferencing and messaging from different vendors. We argue for user and endpoint driven reachability policies and cross-app interactions, instead of pair-wise service federation or global location service. Our architecture separates the user contacts from reachability apps, and has several independent and non-interoperable WebRTC-based apps for two-way and multi-party multimedia communication. Our software is implemented using HTML5 and JavaScript, and using a cross-platform development tool, runs as native apps on mobile as well as personal computers. Our flexible implementation can be used for enterprise or personal communications, or as a white-labeled app for consumers of a business.

Keywords: system design; mobile app; user reachability; WebRTC; caller policy; cross-platform.

1. Introduction

Today's communication systems often create silos or islands of communication where users of one service cannot easily talk to that of another. A typical user can be reached via email, instant messenger, office or personal phone number, one or more social network, and/or on audio and video collaboration apps such as Skype or Hangout. In such multi-apps environment, user reachability is often done manually via user presence and iteration, e.g., check if the user is online on Gtalk or Yahoo before sending a message there, or try Hangout video with fallback to phone.

WebRTC (Web Real-Time Communication)[4] is an emerging technology that allows plugin-free browser-to-browser audio and video flows. It basically enables a website to quickly become a standalone communication service provider. This creates too many such islands, each controlled by a WebRTC capable website, and further encourages this silos behavior. Many websites are already using WebRTC and the number is growing. These websites often require their users to only use the site specific apps, on web or mobile, and do not interoperate with other websites, even though such web services often have similar features of web conferencing or click-to-call. However, a user likes to reach and be reached from her people irrespective of the service or device, and be able to select the best available mode, device or app, e.g., use text message in noisy environment.

Past attempts to bridge the communication islands using pair-wise federations, global location service or multi-protocol clients are largely insufficient to tackle the growing number of WebRTC apps and services from a wide range of competing vendors. The emerging multi-apps communication environment is hard to interoperate or federate globally.

Our approach to bridge the gap, and to automate and simplify user reachability in such multi-apps environment is to *decouple the contacts from communication apps*. Contacts are managed by the user, or dynamically injected by her context, e.g., current browsing or calendar. We have developed such an app, Strata Top9, which is a front-end to launch and interact with other apps to reach a user on voice, video or text. The user independently installs the communication apps from various services. Strata determines the right app with automatic fallback, e.g., use the video app, and if fails, try a phone call.

We have also developed cross-platform communication apps using WebRTC to initiate a video call, join a conference or an upcoming meeting from calendar, discover and connect with other local users, or translate between speech and text modes. Our apps use existing services based on Avaya's IP office, Conferencing or Media Server, or for endpoint driven apps, a Resource Server. They focus on mobile usability, but can also be installed as native desktop apps or accessed in a browser.

We describe the architecture and implementation of this multi-apps user reachability using dynamic contacts and user driven policies. The paper contains motivational use cases (Section 2), differences from related work (Section 3), pieces of the system architecture (Section 4), and description of various communication apps (Section 5). Implementation details of various aspects of these apps including cloud hosting, cross-platform development and resource-based software architecture are presented in Section 6. Finally, we present our conclusions in Section 7.

2. Motivational Use Cases and Requirements

People use multiple communication apps due to device constraint, personal preference, enterprise policy, etc., e.g., Facetime on iOS vs. Hangout on Android, Facebook for friends vs. messaging or Voice-over-IP (VoIP) in office. In both personal and business communications, many users can be reached in multiple ways, on different apps, devices or communication modes, e.g., with multiple Unified Communication (UC) and messaging systems from different vendors seen in hospitals and banks today.

Consider the scenario in Fig.1 with three communication services and ten users connected to some of these. People are on multiple services, e.g., Gail on the hospital phone number and the public video call app. Richard (on left) uses a social app to reach his friends, some of whom work at First Hospital. The dotted arrows show the caller or receiver's preferred mode, e.g., Bobby likes to receive email; Richard prefers video to reach Kathy. Contacts in Richard's app show their preferred modes, or undefined "?" for pending contact requests.

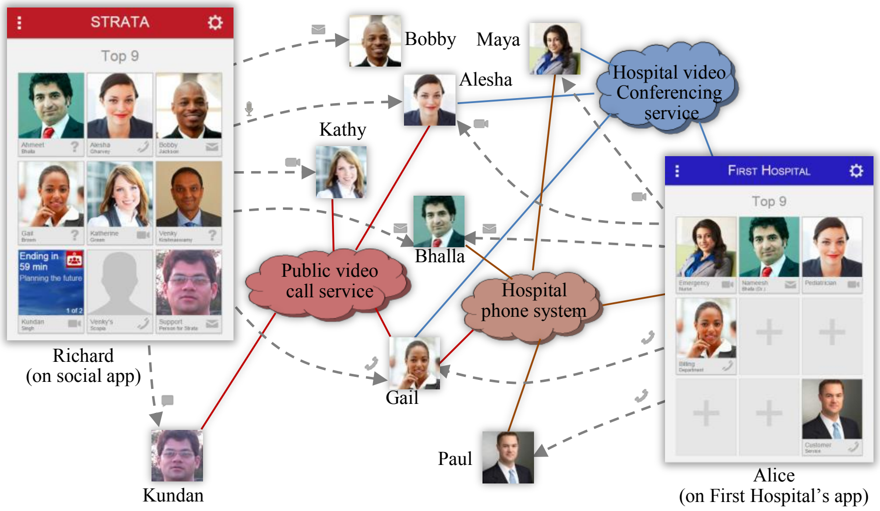


Fig. 1. Example containing multiple services and several user reachability scenarios. There are two screenshots of our app: the social app on left and a white-labelled app customized for a fictitious First Hospital on right.

Alice's app (right) downloaded from the First Hospital's website is pre-populated with important contacts of on-call nurse and billing department. They automatically update as the staff changes shift, indicating who will receive the call, and in what mode. In automatic fallback, if Alice's video call fails to the on-call nurse, Maya, the app tries to reach her phone. The patient can fill the empty slots with her pediatrician or primary care provider, or put other dynamic contacts, e.g., "Billing/Ace HMO" to directly reach the right person, unlike navigating voice prompts; or keywords "pregnancy, natural" to reach a nurse with matching skills. The contact picture can instead show dynamic content, e.g., next calendar meeting, live video of the doctor, or periodic snapshots from her webcam to show if she can receive a video call. The contact may be non-person, e.g., meeting bridge; and may not be call or text reachable, e.g., click to open/edit a shared document or personalized webpage. Important system requirements to support such use cases are:

1. Multiple communication apps and services, independent of each other and of the contact list.
2. User driven reachability decisions by caller and receiver, besides any service enforced reachability policies.
3. Diverse multi-platform reachability apps; selection of a specific app per call attempt.
4. Automatic fallback of apps, devices or modes; either caller or receive can set the preferred or required mode.

5. Minimum reachability via phone and email; e.g., when a doctor accepts the contact request from his patient, he gives a personal guarantee to respond in a timely manner.
6. Asymmetric contacts; e.g., a doctor does not have to add his patients in his contact list, to keep the list small.

3. Background and Related Work

Voice communication and email have historically provided universal reachability via phone numbers and email addresses. Today's communication tools of VoIP, Instant Messaging (IM), on web, or over-the-top apps are often based on open protocols, albeit in a service provider's "walled garden", which hinders reachability on another service, or locks the ecosystem [1][2][3]. WebRTC [4] for plugin-free browser-to-browser communication further makes it easy to create such silos [8]. Past user reachability efforts roughly fall under three overlapping categories: pair-wise federation, global location service or multi-protocol apps.

Pair-wise federation works for a few popular services, but does not scale with the growing number of WebRTC websites [5][6][7][8]. Lack of incentive to providers or less flexibility in server-side translation further hinders this approach. Projects like hookflash [24], &yet [25] and matrix.org [26] are emerging to provide global WebRTC signaling and location services. Convincing websites to use them or change apps to follow their APIs is hard; so they tend to form more isolated ecosystems. SigOfly [9] dynamically downloads the JavaScript code from the target's app provider for cross-service authentication and reachability, but requires the websites to use its APIs. Moreover, this approach does not work for installed mobile apps.

Pidgin and Trillian are multi-protocol apps. Due to lack of a signaling protocol specification in WebRTC – every site can implement its own call setup – such efforts are impractical with growth [7]. Both Session Initiation Protocol (SIP) and Extensible Messaging and Presence Protocol (XMPP) allow external protocol reachability [10][11], e.g., if lookup resolves to a mailto or http URL (Uniform Resource Locator), the caller is redirected to open an email or web client. These are not popular in today's proxy-focused services. User specified reachability with time-of-day, calendar or presence [10][12], or fine-grained user preferences to select mobility or mode [13][14][15] are known. These existing systems based on multi-protocols reachability do not work when, say, a SIP provider allows only its own app or device to connect to its service. In practice, existing multi-protocols reachability is not the same as the desired multi-apps.

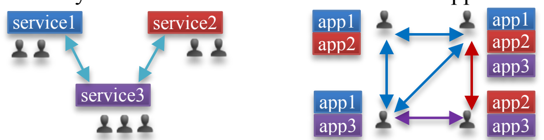


Fig.2. Reachability: pair-wise service federations vs. user driven apps in the endpoint.

We conclude that we are in a multi-services and multi-apps environment which is very hard to interoperate or federate globally. Thus, solving user reachability with user driven apps in the endpoint is a viable option. Unlike pair-wise service federation, we let the user select her reachability apps (Fig.2); this freedom promotes innovation. Toutain et al [8] realize that users are overwhelmed by the number of communication apps and need a simple way to reach their contacts. They conclude that the user's contacts must be independent of the services. Unlike ours, there is no implementation, and it proposes to interoperate identity management to tie the user presence to the contact list. Our app does not include presence, and with no global identity service, is easier to deploy or scale.

We use web-style code for call policy, unlike endpoint behavior in Extensible Markup Language (XML) [12]. Separating data from app logic is well studied [27][28]. Our use of resource-based software architecture continues from [16][17][18]. The ability to launch external apps is inspired by the now discontinued webintents [19]; albeit extended beyond a single device using shared data. In summary, ours is a pragmatic way to deal with emerging WebRTC-based systems and covers multiple modes, devices and non-interoperable apps even if on the same protocol. Our work is a continuation of [21] to include the implementation details of the resource-based software, cross-platform development, and cloud and mobile challenges and techniques.

4. System Architecture

We describe the individual pieces of our system such as separation of contacts and reachability apps, cross-apps interactions, on-demand reachability and caller policies.

4.1. *Important definitions*

Communication mode is one of video, phone or message. We use voice and phone interchangeably; phone does not mean a phone device, but may be a voice call on a softphone. Video or phone indicates real-time interaction. Message covers real-time as well as asynchronous apps, e.g., text chat vs. email, short message service (SMS), and voice/video messages. An app may have multiple modes, some limited by platform, e.g., no WebRTC video on Safari/iOS.

Communication app is typically a standalone application on desktop and/or mobile, or even in a browser. It may be limited by device or network, e.g., business IM only on Virtual Private Network (VPN). An app is often tied to a service: a VoIP provider or hosted conference system. We use *service* and *app* interchangeably.

User reachability is defined as the ability to reach a user on one or more communication apps or devices. We also refer to email clients and phones as apps, although not controlled by our architecture. A *reachability item* is a triplet of *mode*, *app* and target *value*, e.g., VoIP address, phone number, or click-to-call or conference URL. A reachability list can have items on the same app or mode as shown below. The value is interpreted by the app, e.g., (1) could become tel:+18002223333,,13001234# in that conference service, and (3) could be sms:+14151234567.

```
(1) {"mode":"phone", "app":"Scopia", "value":"13001234"}
(2) {"mode":"phone", "app":"Phone", "value":"+14151234567"}
(3) {"mode":"message", "app":"Phone", "value":"+14151234567"}
```

4.2. *Separating contact list from communication apps*

This separation is crucial to support multiple diverse apps. Fig.3 shows that the user's contacts can be populated in many ways: by provider, managed by user, imported from mobile phone, or by user's context, e.g., discover other app users in a hotel guest room or emergency situations using local multicast (serverless), or discover other viewers of a website using a browser extension. A contact item may be static or dynamic. The latter changes its reachability by time or other factors. It may be a shared group contact, where reachability is for any, all or some of the group members, e.g., for customer support or group meetings. It may be auto-populated by data mining, e.g., of email/IM to find my frequent/recent contacts, or by phrase "I will get back"; or from an email thread or invite group.

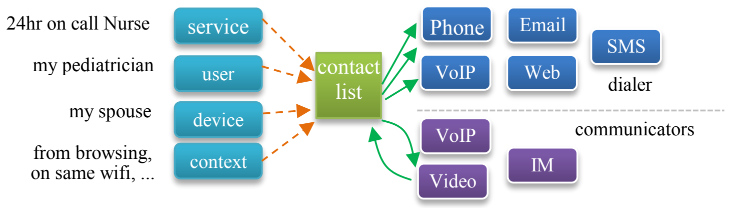


Fig.3. Separation of contacts and reachability apps including dialers and communicators.

We consider two types of apps: *dialers* and *communicators* (Fig.3). A dialer only does outbound interaction request. Once launched it does not return to the contact list, except on failure. A communicator can return to the contacts app for intermediate decisions or to apply policy on received requests.

4.3. *Cross-app interaction and handoff*

A dialer app usually registers as a protocol handler, e.g., a mailto or tel URL open the native email or phone client. A communicator is either separate or integrated with the contacts to simplify inter-app messages. A provider's dialer app is often used as is. A communicator requires changes to separate the call setup and conversation; e.g., a softphone that informs the contacts app on incoming call, and proceeds on approval. The user may change the mode to message (Fig.8a), or move it to another app or device, informing the caller about the change.

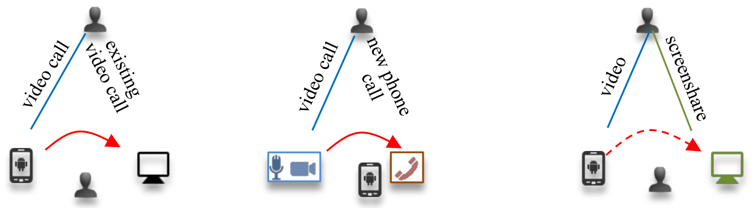


Fig.4. Call handoff: (a) from one device to another, (b) from one app to another, and (c) call component handoff, e.g., for screenshare.

Consequently, three types of handoffs in Fig.4 are (a) device, (b) app, or (c) call component, e.g., move a call to the desktop phone, a video call from desktop to phone, or share desktop screen in a mobile call or add mobile touch-input white-board to a PC call.

4.4. *Loosely coupled resource-based architecture*

We use resource-based software architecture that has loosely coupled independent apps with data-level mash-up [17][18]. As shown in Fig.5, the architecture has five crucial pieces. First, shared pieces of data or resources are stored in a file-like hierarchy. Second, the client accesses these resources over a WebSocket and HTTPS connection. Third, a resource entity of event is often represented in JSON (JavaScript Object Notation), e.g., {"name": "Alice Smith", "id": "6521"}. Fourth, the resource service is a simple and light weight web server backed by a database and supports secure and authenticated data access and event notifications. Fifth, the client-server API implemented in the JavaScript client library consists of data access and event notification primitives.

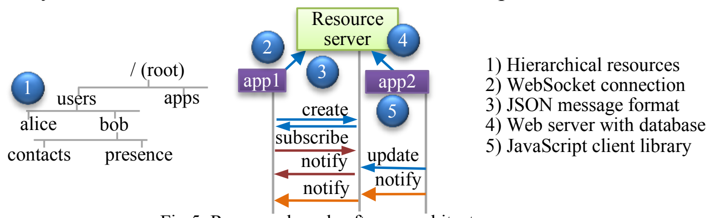


Fig.5. Resource based software architecture.

The server stores the resources without knowing its semantics. The app logic in the client defines the semantics. For example, apps can agree on using /users/{user}/presence as the user’s presence resource. Then one app subscribes to this resource, and another app updates it, which triggers an event to the first. The first app can then update the presence status icon representing that contact user. The server can also transparently forward end-to-end event notifications, e.g., to send call event or WebRTC signaling data. A resource path could represent a single record or an ordered list, in which case the list items consist of the immediate children of that resource path. The resources are owned by the end user and not by the app, i.e., the user can give access permission of her data to any other app, without requiring approval from an existing app. More details on application mash-up, use of WebRTC, and user driven access control are in [17][18].

Our client apps implement many scenarios: voice and video call, conferencing, text chat, contact list and user reachability. The basic concepts of shared data access and event notifications available in the resource based software architecture are used to implement all the application logic in the endpoint, including for user profile, contact list, contact request, user reachability, call attempt fallback, caller policy, video conferencing, call membership, text chat, file sharing, and multi-party calls. The contacts and communicator apps mash-up at the data level, e.g., for received call event, or caller policy access. Data namespaces enable multi-tenancy and app customization. Implementation details on how Strata uses the resources for various application logic are in Section 6.

4.5. *Proactive presence vs. on-demand reachability*

We prefer on-demand reachability to active user presence, i.e., the caller side tries to reach the receiver's reachability items with fallbacks (Fig.6). There are many reasons for this decision as follows.

(1) Relying on presence fails in a multi-apps environment because email, phone or conference bridge codes are always present. The question is not whether the user is available, but where.

(2) Softphones supporting presence often use different protocols that may not be completely interchangeable via signaling translation.

(3) Presence systems scale poorly due to rich presence traffic, or periodic refresh of presence soft state on battery constrained or mobile devices.

(4) An online status does not guarantee a call answer, and may require fallback in any case.

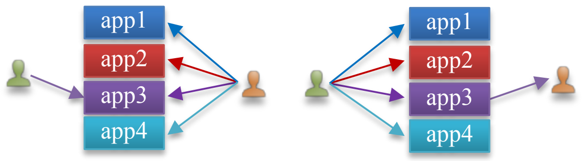


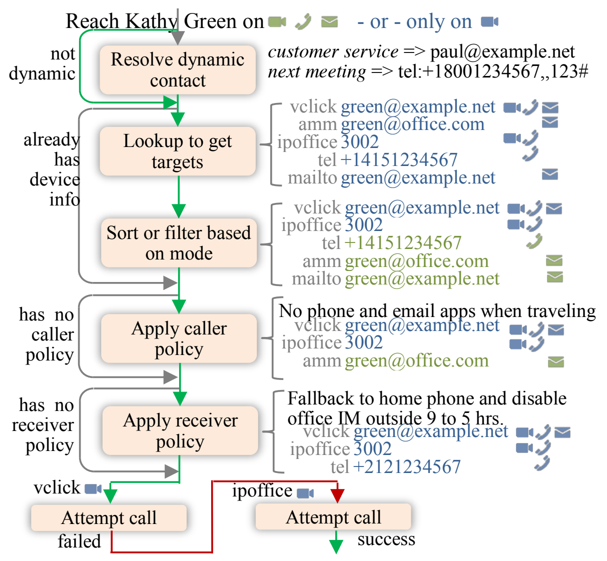
Fig.6. Reachability: (left) in proactive presence, receiver publishes presence in apps, and the caller knows which to use before a call attempt, vs. (right) caller tries the apps on demand during call-setup until picked by the receiver.

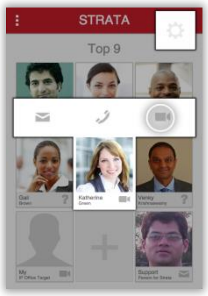
Our desktop app uses persistent WebSocket, on which we may enable presence if needed. Our mobile app uses WebSocket only when the device is awake, but uses platform specific low power event channel when asleep, e.g., Google Cloud Messaging (GCM) on Android. Our contacts app does not use presence, but a launched app such as a third-party instant messenger can still use it internally to determine if the call will succeed. The on-demand and active presence are combined in practice.

4.6. *User reachability and fallback*

Fig.7 shows an example user reachability process when Richard tries to reach Kathy. The algorithm runs on the caller's Strata app, but can instead be at the server. The first step is to resolve any dynamic contact, e.g., to get the next meeting for a calendar contact, or to map "customer service" to the currently reachable agent. The contact type defines the tool to resolve, e.g., to extract a bridge number data from calendar. This step is skipped as it is not a dynamic contact in this example.

Next, all target reachability items are fetched. This does not apply if a specific item is found in the previous step. Based on Kathy's email and phone number entered during signup, three default items are pre-populated: (1) the default communicator app for video, voice and text, (2) the phone app for voice call, and optionally mobile SMS, and (3) the email app for message. She may not be available now on Strata, or may be on many devices, or on her employer's apps based on Avaya IP office (ipoffice) or Messaging (amm). She has already configured all her items before. The items are ordered in decreasing preference: the default (Vclick) first; phone and email last; and other items (amm, ipoffice) in between. Receiver can modify the default values or their order if needed (Fig.8f).





Screenshot of mode selection in a call attempt, instead of using default mode.

Fig.7. Example user reachability process in Strata Top9

Depending on the mode of the call attempt, the reachability items are sorted and filtered. Richard may initiate conversation in default mode, say video, by clicking on Kathy’s contact, or select a specific mode, say video, by click-and-hold on her contact as shown in the screenshot in Fig.7. The two cases behave differently in our app. The former falls back to other modes if Kathy is not reachable on video, but the latter fails if the selected mode is not in the list. In the former, the list is sorted for video, phone and message, in that order because the preferred mode for this contact is video. In the latter, items without video are removed.

Next, the optional caller and receiver policies are applied. In this example, Richard’s caller policy disallows using any phone or email apps when he is traveling, which filters out those from the list; and Kathy’s receiver policy disallows employer’s messenger and adds a last resort as her home number to reach her outside office hours.

Finally, the items are attempted with sequential fallback, e.g., if Kathy is offline on Strata (unreachable via Vclick), try video on ipoffice, and then a phone call to her home number (a mode fallback). Mode fallback is not done within the same app, e.g., if the Vclick video call fails, then do not reattempt voice or message on Vclick. However, an individual app may support mode handoff or transfer, e.g., an incoming video call in Vclick can be answered as a voice or message session (Fig.8a).

Typically, fallback can happen only if a call attempt on an app can return an error. This works for communicators and dialers that can return the result to the contacts app. It does not work for some dialers, e.g., native phone or email clients opened using a tel, sms or mailto URL. We have developed a modified phone dialer (Engagement Dialer) using our VoIP system that can return a result. Multiple line presence is the responsibility of the app, e.g., Vclick supports it and enables the user to run Strata on multiple devices, where the first one to answer is connected; whereas the IP Office app logs out the previous device when the user logs in from a new one.

4.7. *User driven policies*

User customized reachability order (Fig.8f) works for most people. We also support programmable policy for finer control. The caller policy is applied to outbound request, and the receiver one to inbound. They are written in JavaScript-like code with only a few supported constructs as shown below.

(a) To reach my colleagues, prefer video and avoid my personal instant messenger.	`if (receiver.email =~ "*@office.com") { prefer("video"); exclude("message", "AIM"); }`
(b) Always call my cell after office hours irrespective of caller's preferred mode, and stop further policy lines.	`if (now.hh >= 17) { choose("phone","Phone","+1212123456"); break; }`
(c) When I am traveling, only receive message mode; and fallback to my personal messenger service.	`if (location.address.country != "India") { include("message"); deprecate("message", "AIM", "alice"); }`

The script supports simple as well as nested if-else controls, and a break to stop further script processing. JSON objects representing the caller, receiver, current time and location, and comparators and regular expression are used in the policy decision. The caller and receiver objects contain the user attributes, e.g., user identity, name, preferred mode and contact's approval state. The now object has the current time in various formats, including time zone and UTC (Coordinated Universal Time) data. The location object has the current device location fetched using HTML5 and Google's geocoding APIs (Application Program Interfaces). The app does not retrieve the device's location unless the script uses location. Some examples are shown below.

caller	{"email": "bob@example.net", "name": "Bob Wilson", "type": "video", "state": "approved"}
receiver	{"email": "alice@office.com", "name": "Alice Smith", "type": "phone", "state": "pending"}
now	{"YYYY": 2015, …, "hh": 19, "mm": 38, …, "tz": "+07:00", "string": "2015-08-03 19:38:42", …, "utc": { "time": … }}
location	{"address": {"street_number": …, "locality": …, "state": "California", "country": "United States",…, "short": {"country": "US", …}}}

The script uses some functions to alter the behavior: include, exclude, prefer, deprecate and choose. Each function takes three parameters: mode, app and target value. Only choose requires all three, but others treat app and value as optional. These functions manipulate the reachability list shown in Fig.7. The choose function deletes the list, and adds only a single reachability item supplied in the function. The include and exclude functions filter the list to include only desired items or exclude undesired ones. If an optional parameter is missing, it acts as a wildcard, matching any item. The prefer and deprecate functions re-order the list to move certain items to the beginning (most preferred) or the end (least preferred). If all three parameters are supplied, then these two functions also act as a way to inject one reachability item at the beginning or end of the list.

Note that these policies apply only during initiation, not in an active call. The policy engine is currently in the Strata app, and hence, only used if the call is initiated or received by this app. We have web-based policy script editing, and in future will have graphical interface with drag-and-drop editing.

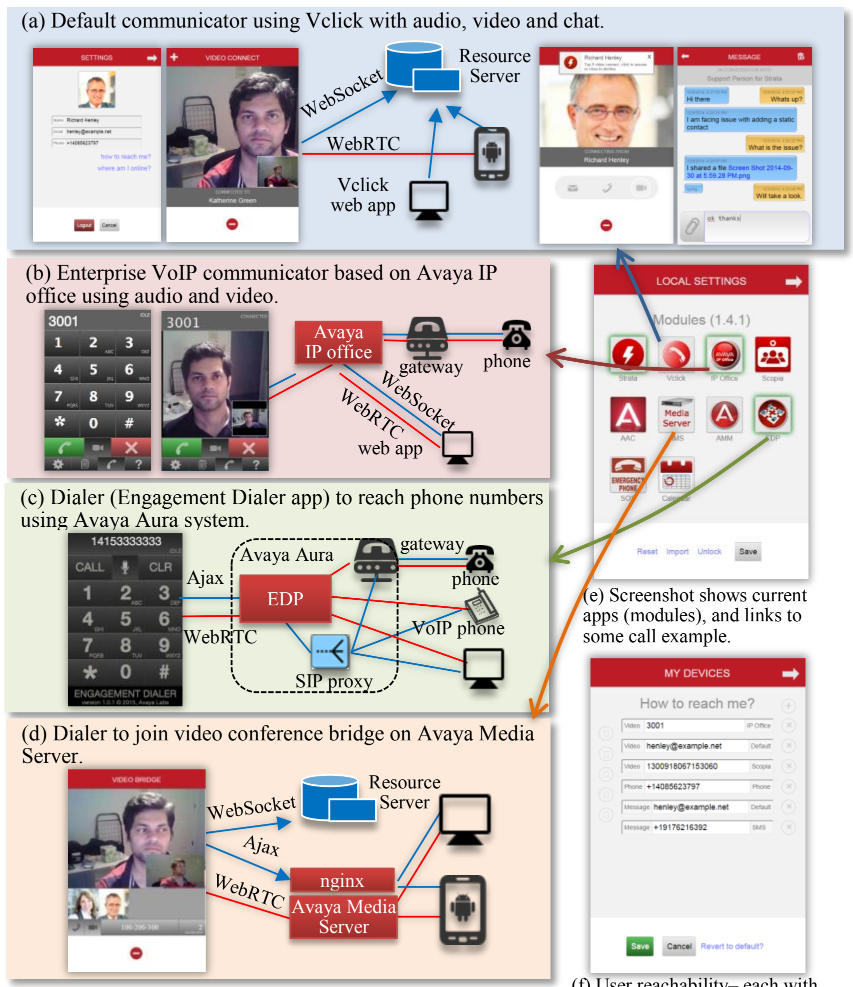
5. User Reachability Apps

While the Strata app deals with contacts and reachability policies, it can launch an individual reachability app to initiate or answer an actual conversation. Our implemented reachability apps are listed below, and some of them that are based on WebRTC are shown in Fig.8. These apps cover a wide range of communication scenarios, e.g., client-server media path vs. peer-to-peer media flows, dialers vs. communicators, experimental vs. commercial systems, and thin-client vs. thin-server.

Default communicator using Vclick: Vclick [18] is a collection of loosely-coupled apps that mash up using the resource server and are independent of legacy VoIP systems. The realization in Strata includes only a subset of apps – for text chat with optional attachments and speech/text translation, and full mesh WebRTC-based voice/video calls and conferences (Fig.8a). Section 6 contains more implementation details.

IP office phone: Avaya IP office is a VoIP system for small and midsize businesses. We built IP office phone, a communicator app, to connect to this VoIP system to make or receive voice or video calls. Besides the separate app (Fig.8b), an integrated-to-Strata version is implemented. Unlike the peer-to-peer media path of Vclick, it anchors the media path at the server. This is not uncommon in cloud telephony, e.g., for media services of recording, interactive voice response or telephony gateway.

Engagement dialer: It allows dialing out a phone number using the enterprise or cloud VoIP service of Avaya's Engagement Development Platform (EDP) and Aura software suite (Fig.8c). It handles the tel URLs including optional pauses and touch-tone digits, e.g., tel:+18001234567,,,123#. Thus, Strata can use this to reach phone numbers or conference bridges, e.g., from tablets or desktops. If the target value of the contact is empty, it opens a generic phone dialer, allowing the user to enter the target number. This avoids having to add a one-time phone number in contacts.



IN CONVERSATION WITH
Support Person for Strata

Hi there

Whats up?

I am facing issue with adding a static contact

What is the issue?

I shared a file Screen Shot 2014-09-30 at 5.59.28 PM.png

Will take a look.

ok thanks

How to reach me?

Video	3001	IP Office
Video	henley@example.net	Default
Video	1300918067153060	Scopia
Phone	+14085623797	Phone
Message	henley@example.net	Default
Message	+19176216392	SMS

Fig.8. Screenshots and system architecture of the WebRTC apps in Strata Top9 implementation

Avaya Media Server (AMS) app: AMS allows multi-party audio and video conference using RESTful (Representational State Transfer) APIs for control and WebRTC for media. It does audio mixing and video switching based on active speaker. We built an AMS dialer app (Fig.8d) that uses the resource server to manage conference membership and moderator information, and to join the video bridge, without any legacy VoIP signaling. Implementation details are in Section 6.

Avaya Multimedia Messaging (AMM) app: AMM also has RESTful APIs to enable multi-party messaging. We built an AMM communicator app to send and receive text messages from Strata.

Emergency call (SOS) dialer: This dialer extends the Engagement Dialer app to support emergency calls and has two additional features: supply caller's location data using HTML5 geolocation APIs, and receive out-of-band media from the emergency call taker, e.g., picture or video instructions to help the caller in an emergency situation. The target number is pre-configured and does not need to be explicitly dialed. The app is particularly useful for campus use case, where the university configures its local emergency number and other attributes in its customized Strata app, and the students can quickly reach the campus security to receive help. Furthermore, such a call automatically falls back to country specific emergency number, e.g., 911 in USA, if the security staff could not be reached.

Next meeting/Calendar: The calendar app uses a light-weight proxy to periodically fetch the user's calendar from her enterprise mail exchange server, and displays one or more ongoing or upcoming meetings. The picture cycles through multiple overlapping meetings if needed, and allows click to join via video or phone, instead of a manual dial-in of bridge number and code. This dynamic contact maps to a reachability item on AAC (Avaya Aura Conferencing), Scopia or phone depending on the meeting data. AAC and Scopia represent a series of Avaya UC products for audio/video conferencing and online collaboration.

Furthermore, Strata can launch existing apps, e.g., email, phone, third-party Jabber apps, or conference client apps of AAC and Scopia, or can join their voice bridges. Fig.8e shows the available modules or apps that are configured. New apps can easily be added and configured to be launched from Strata. We also have a WebRTC-based Scopia dialer app for audio video conferencing with limited features, e.g., without application sharing or text chat.

6. Implementation

Strata Top9 is an app for desktop and mobile to quickly connect with any of the user's top 9 contacts. It is a front-end to launch other apps including the ones listed in the previous section. This section describes some crucial design and implementation aspects of Strata and the various reachability apps.

6.1. *Resources for endpoint-driven app-logic*

First, we describe the shared resources used by Strata and Vclick to do endpoint driven app-logic using the resource-based software architecture. Fig.9 shows the hierarchical resources, and the description below contains its resource references, e.g., (#6) refers to /apps/strata/users/email2/inbox/{id}. The resource server can generate random unique id for a list item resource if not supplied by the app, e.g., a new conversation resource (#12) at /apps/vclick/chats/{id}. A resource can be created as persistent or transient. A transient one is automatically deleted when the creator of the resource disconnects from the server.

For example, user profile (#1) persists, but user's presence (#3) is implicitly removed by the server when the app crashes or is closed, indicating offline. A semi-transient resource is removed when there are no more subscribers to its parent path, e.g., messages (#13) in an ad hoc chat session are deleted when there are no more participants in that session.

6.1.1. *Strata: user profile, contacts, app instances*

All the user data in Strata are stored under the tree rooted at the user resource (#0). The server enforces user permissions and access control as needed. The user profile (#1) includes the user name, picture bitmap encoded as Base64, and user's reachability items. The default profile (#2) is used to create a new user for customers of a tenant group, e.g., to import pre-populated hospital contacts. An app instance creates and listens to the user presence (#3). This is used for sending paging (or session-less) messages, e.g., one-time text, picture or video. Session messages use Vclick resources described later.

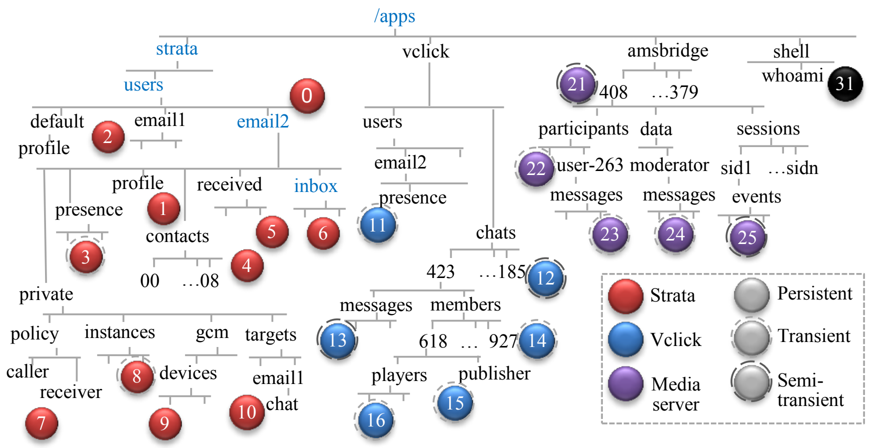


Fig.9. Resources used by Strata and Vclick.

The Top 9 contact items (#4) are numbered from 00 to 08, and record the application name, mode and contact email or app-specific value. If there are more than nine outbound contacts, subsequent ones are not numbered. The received contact items (#5) are not numbered, and record the approval state, mode and sender's email. The user's inbox (#6) is used by other users to send a message to this user, e.g., a contact request or approval response. The receiver cleans up inbox to avoid duplicate processing. The policies (#7) allow programmable call initiation in outbound and inbound directions for this user. The caller app fetches the policy, and applies it at the endpoint with no server-side logic.

When the user logs in, the app creates an instance resource (#8) so that the user's app instances can detect and interact with each other about where the user is available. The native Strata app on Android creates a cloud message authorization resource (#9) for potential callers. Cloud messaging is described later in this section.

6.1.2. *Vclick: conversation, video conference, text chat*

Strata uses Vclick resources for actual conversations. Vclick creates semi-transient chat messages without chat history. However, to preserve chat history, Strata creates persistent messages for text chat or file sharing (#13), and stores a reference (#10) to the last chat session with the target users. The app takes the responsibility of explicitly deleting older messages. Vclick uses the presence resource (#11) to send call signaling and chat events. For compatibility, Strata also creates and subscribes to its own user presence (#11), but does not subscribe to or show the presence of user's contacts. However, this design allows other developers to create a Strata-compatible but presence-enabled app using these resources, if needed.

All the data related to a single conversation are under chat root (#12). To join a call, the app creates a member (or participant) resource (#14). A user can join multiple times with random suffix appended to this resource path. It has additional attributes such as for camera and microphone state, so that a participant can know when another one is muted. The publisher and player resources to enable full mesh conference using named streams are described later in this section.

6.1.3. *Centralized conference*

Our media server (AMS) app joins centralized conferences. The media server runs in slave configuration and is controlled by a master application via web APIs. We promote one of the participant apps to master (or moderator), which controls the server in each conference. The moderator exchanges signaling events between the media server and other participants. If the moderator leaves, another one is picked up automatically. This keeps the app-logic in the endpoint instead of using another application server as master.

The clients use the resource server to co-operatively store call membership data and to exchange signaling events. All of conference resources are within its root (#21). A participant creates and subscribes to its call membership (#22). Participant attributes include mode, mute status and a profile picture. We allow mixed audio and video participants in a single conference. The participant and moderator resources (#23, #24) are used to exchange signaling events with each other. The media server identifies each media flow (or session) by its identifier. A participant listens for its session events. The events generates by the server are captured by the moderator, and published to the appropriate session's event list (#25), and thus, picked up by the right participant.

The resource server has some built-in resources that are used to access certain authentication data. For example, "who am I?" (#31) is used to determine the namespace and email address from the connection token of the endpoint.

6.2. *Work flow and app-logic*

6.2.1. *User signup and login*

First, the user signs up on our cloud-based web portal using her email address. If the Strata app is approved for the user, the portal creates an authentication token on the resource server for this signup. The user then downloads, installs and launches the app on desktop or mobile. On the login screen, the user enters her email and password of the portal, or the auth-token of the resource server. The server supports multiple tokens per email to help in multi-tenancy. If email is used to login, the app retrieves the auth-token from the portal for the default tenant group.

Using the token also keeps the resource server independent of any single portal, e.g., external partners can obtain their users' tokens from us, instead of requiring the users to sign up on our portal. The auth-token is saved on the device, and enables auto-login on subsequent app launches. The email and password are not saved to reduce the risk of clear-text password leak from the device. On app startup, if a valid saved token exists, the app successfully connects to the resource server over WebSocket.

6.2.2. *User profile*

Once connected, the app fetches the email and namespace (#31) associated with the auth-token. If the namespace is not default, it applies any tenant specific User Interface (UI) customizations. It fetches the user profile (#1). If it exists, the login completes. If it does not, then the user is prompted to create the profile. Fig. 10 shows the screen transitions.

Creating a user profile involves updating the name and phone number, and creating a profile picture, either uploaded from the file system or captured from the device camera. The picture is resized and cropped for efficiency, e.g., an uploaded picture of 1800×2400 is center cropped to 1800×1800 and then resized to 140×140. An initial reachability list is derived based on the email and phone number, and can later be altered on the reachability page. If the profile is missing, then this login creates it, and the login completes.

After login, Strata creates and subscribes to other resources (e.g., #8, #9), and initializes other apps, e.g., to login on IP office, or to begin periodic task to fetch calendar appointments. Finally, it shows the Top 9 page.

6.2.3. *Contact list: outbound and received*

The Top 9 page has nine contact slots. Both in personal and business communications, people often initiate conversation with only a handful of contacts on a regular basis or attend online meetings based on a few workflows, e.g., from their calendar events. Since it is easy to change a contact slot, the limit of 9 is not an issue for many people. It allows an aesthetically sound 3×3 layout on a phone, but can be changed to 4×4 or, in landscape mode, 4×2. Moreover, additional contacts can appear in subsequent pages beyond the first Top 9 page. User adds a new contact in an empty slot, or changes an existing one. If the target is not a Strata user, it allows inviting via email.

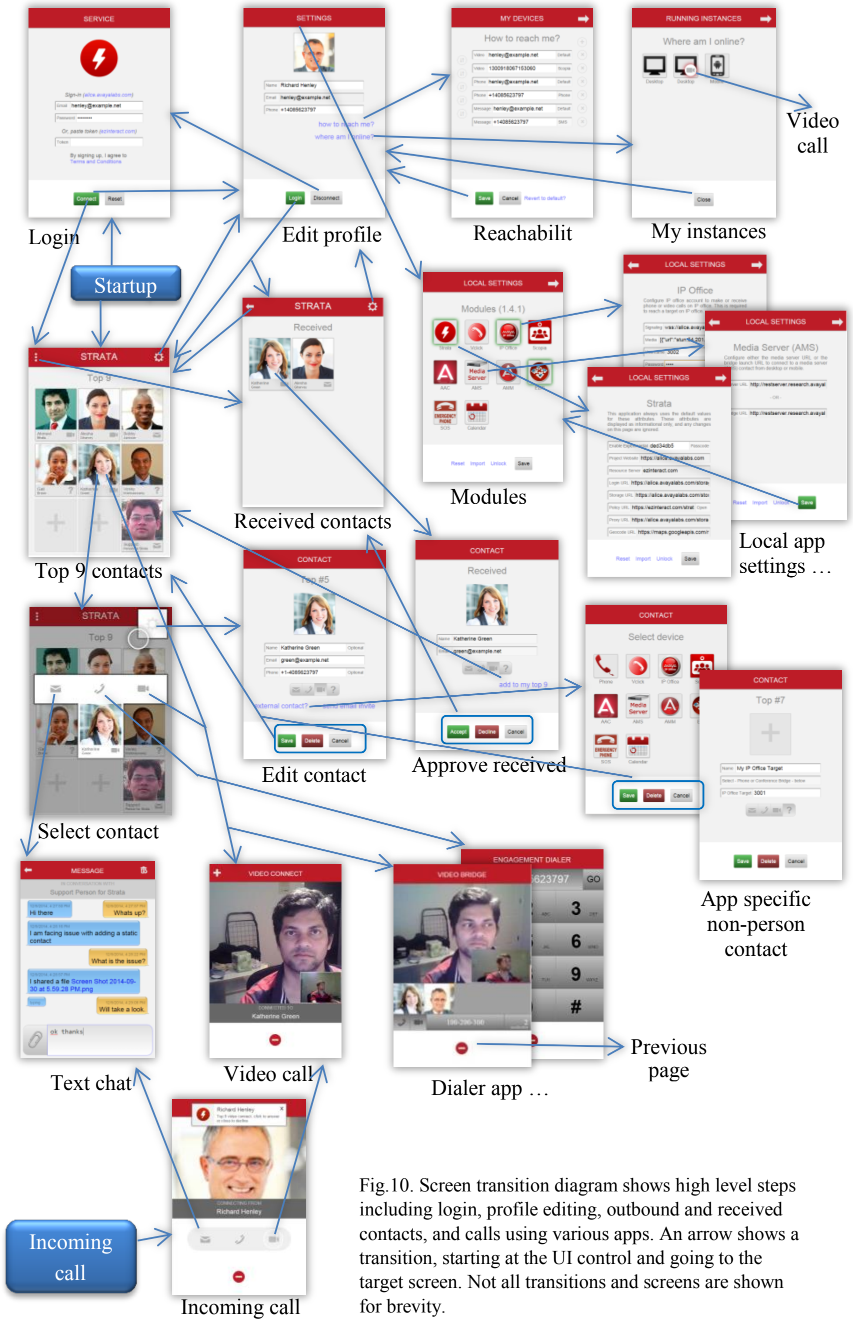


Fig.10. Screen transition diagram shows high level steps including login, profile editing, outbound and received contacts, and calls using various apps. An arrow shows a transition, starting at the UI control and going to the target screen. Not all transitions and screens are shown for brevity.

The Top 9 page includes person and non-person contacts. The received page only shows those who sent me a contact request. While the Top 9 page has fixed numbered slots on its first page, the received page items are not numbered. Furthermore, the Top 9 page allows swapping the contact slots, e.g., to move contact of slot #7 to #2. The receiver accepts or declines the contact request, or changes the approval state at any time on the received page. A non-person contact can use any app and does not need approval, e.g., next meeting in calendar or specific dial-in bridge (Fig.10). The user may optionally select a preferred mode in Top 9 (caller) and/or received (receiver) contact. Caller and receiver can set preferred mode independently. In a call attempt, the caller and receiver preferences are used in that order, if available, otherwise a fallback to message mode is assumed. Moreover, a caller or receiver policy, if available, can override the preference.

Same contact can be in multiple slots, e.g., for different preferred modes, or for calendar contacts, to show multiple upcoming meetings. User can change the name, phone number and preferred mode in each slot of the Top 9 page. When Richard adds Kathy in her Top 9 contact, Kathy's record is added to Richard's contacts resource (#4), and a contact request message is added to Kathy's inbox (#6) (See Fig.9 for resources). The inbox message is displayed by Kathy's app, immediately if online, or the next time she goes online. When Kathy accepts or declines the contact request, the response message is added to Richard's inbox, and Richard's record is added to Kathy's received list (#5). Kathy can change the response later on the received page. When Richard's app receives the answer, it shows the notification and updates the record (#4). Both parties subscribe to each other's profiles (#1), so that any change is detected and updated in the contact display. The app also subscribes to all the created contact and received resources, so that it can maintain display consistency across multiple app instances of this user.

6.2.4. *User reachability*

The user manages her reachability (Fig.8f) to enable others to reach her when offline on Strata. The reachability page allows adding, removing, editing and reordering the user's reachability items. The reachability data is stored in the resource server. The app specific data, e.g., IP office login credentials, are in device's local storage, so that separate app instances on different devices can be customized, e.g., to use IP office only on the work computer but not the personal tablet.

6.3. *WebRTC for multimedia communication*

We use WebRTC for audio/video flows. While Strata does not deal with WebRTC, many of the reachability apps do. WebRTC enables a web page to establish a peer connection between two browsers, and transport captured media from one to another.

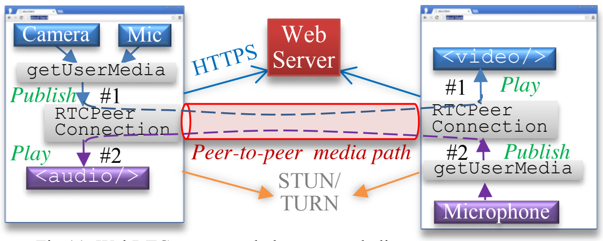


Fig.11. WebRTC conceptual elements and client server system

6.3.1. *Background*

Fig.11 shows various elements in an asymmetric call with audio and video in one direction and audio-only in the reverse. A page can capture from local mic and/or camera using getUserMedia as a local media stream abstraction, create RTCPeerConnection, a peer-to-peer abstraction between browser instances, and send a media stream from one browser to another. RTCPeerConnection emits signaling data such as session description and transport addresses, which must be sent and applied to the RTCPeerConnection at the other browser to establish a media path. For this, web apps typically use WebSocket or Ajax over HTTPS between the browser and the web server. This is called the notification system of WebRTC, and is outside the specification. We use our resource server for notification. Additionally, media relays and reflective servers are used to discover server reflexive and/or relayed addresses for media paths crossing network boundaries.

6.3.2. *Video widget*

In Vclick, all the WebRTC app-logic is hidden in the video widget abstraction [17]. A video widget can publish or play a named stream. A stream can have one publisher and zero or more players. A full mesh N-party video conference has N active streams, one per participant. Each participant publishes its stream, and plays from all others. Thus, each endpoint uses N video widgets, each representing one participant, where one widget publishes the local media stream, and the other N-1 widgets play the remote streams. We use unidirectional media flow in a peer connection; with another publisher-player pair in reverse. This allows simple application abstraction to implement two-party, multiparty, one-way and broadcast scenarios.

The publisher and players can come and go in any order. The app correctly creates connections and media flows without custom call signaling, by merely using the call and membership resources. The member resource path (#14, in Fig.9) is used as her stream name. To join a call, the participant creates a publisher resource (#15) for its own stream. Each of the other participants creates a player resource (#15) for this publisher stream. The publisher and player resources are used for exchanging WebRTC signaling data, and they create a media flow from the publisher to every player of that stream.

6.4. ***Cross platform development***

We developed Strata and other WebRTC apps in HTML, JavaScript and CSS (Cascading Style Sheets), and using ChromeApp and Apache Cordova tools and frameworks [20], ported to native apps on desktop as well as mobile [22]. Our apps can run in four cross platform scenarios – as web app on desktop and mobile, and as installed app on desktop and mobile. Creating a cross-platform app using Chrome Cordova Apps (cca) tools is a three step process: (1) create the web app using HTML5 technologies that runs in a browser, (2) convert it to a ChromeApp that runs and behaves as a native app on PCs and laptops, and (3) convert it to a native mobile app, e.g., Android .apk, using Apache Cordova.

6.4.1. *Componentization in HTML5*

First, we create the endpoint software as components with separates and loosely coupled source files. Iframes are widely-deployed form of componentization for web apps [23]. Such components avoid leaking elements of one component into another, and implicitly clean up any residual state on unload. For example, Engagement Dialer uses a headless JavaScript library in an iframe, so that when the call terminates, any per-call dangling references are cleaned up by the browser. Furthermore, iframes based components do not need single-page-application frameworks which are unsuitable for long lived mobile apps due to bulky script injection or memory leak buildup over time. Strata can launch an external reachability app. It can also load it within an iframe if that app is also built using cca tools. The apps of Fig.8 are actually loaded in iframes within Strata.

6.4.2. *ChromeApp restrictions*

Next, the web app is converted to a ChromeApp [20], which is a package of web files that runs as native app the Chrome browser's rendering engine and its Native Client plugin. It can be distributed on the Chrome Web Store. It runs as a standalone app, and cannot interact with or modify the visited web pages. ChromeApp differs from web pages: it can use new APIs to access local storage, file system or power settings, but cannot use some features such as cookie, blocking prompts or inline scripts. The images and resources from external sites must be packaged in or loaded via Ajax.

To do the conversion, we move all inline scripts to a separate script file, e.g., add event listener instead of inline onclick handler, change external image load to Ajax instead of setting image source URL, and substitute restricted APIs if possible. A manifest file specifies the app permissions to use notifications, cloud messaging, location, storage, or to capture audio, video or desktop, or to reach external web services.

6.4.3. *Packaged native mobile app*

Finally, we use the cca tools to create a native mobile app, to be uploaded to Google Play Store (Android) or Apple App Store (iOS). The conversation from a packaged app is usually seamless for app permissions to appropriate features and plugins in Cordova.

Some features require manually adding the necessary plugins plugins. We use Cordova plugins to enable audio/video capture, storage, notifications, GCM, idle detection, media recording, location, device contacts, file handling and transfer, and web intents.

6.4.4. *WebRTC in native mobile apps*

Although WebSocket is supported in Cordova on Android as well as iOS without a plugin, WebRTC is included by Cordova only on Android, but is not natively available on iOS. We use cordova-plugin-iosrtc to enable WebRTC on our iOS app. However, such plugins have limited capability. For example, Safari's video element does not support WebRTC, hence an overlay is created on top to display the real-time video. Such overlay must be moved and resized whenever the attached video element is altered. The overlay remains on top, and hence, interferes with CSS z-order. This plugin does not work well from within an iframe. We send plugin API calls from iframes to the top-level window on iOS in our JavaScript wrapper library as shown in Fig.12.

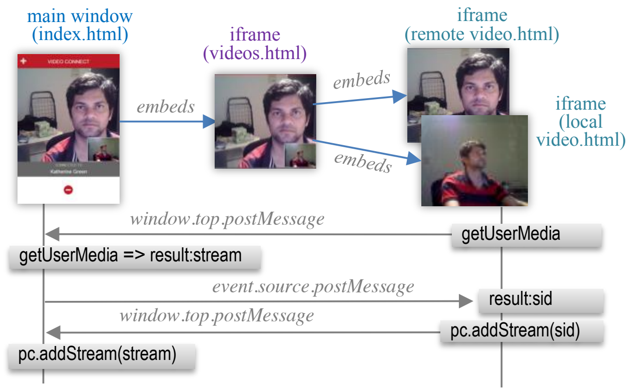


Fig.12. WebRTC APIs from embedded iframes to the top-level window in an iOS app using postMessage.

6.4.5. *Protocol handler to launch apps*

We use HTML5 registerProtocolHandler, Android web-intent or iOS app extension to launch a dialer app when the browser opens, say, a "tel:" URL. For example, Engagement Dialer, if installed, is launched to call out when the user clicks on (or a mobile app opens), say "tel:+12123334444". Similarly, A web+vclick:*user@domain* link dials-out and web+vclick: call:*call-id* joins the call via Vclick.

Such URL-based loosely coupled interactions enable app innovations, e.g., a new app can replace Engagement Dialer without changing Strata, or a "facetime:" reachability can be implemented without affecting the other apps.

6.5. ***Cloud hosting and multi-tenancy***

Several server pieces to support Strata are hosted on Amazon cloud, e.g., resource server, media relay and web portal. We support hybrid deployment, e.g., Strata connected to the cloud resource server can launch a reachability app to connect to a private media server or IP office system within an enterprise. Furthermore, if Strata is used within an enterprise network, it can utilize the corporate directory name search when adding a contact, or dialing out a phone number via name. Separation of user data from the application logic further enables us to dynamically change the resource server, e.g., to a private one when Strata is used within a closed network or VPN.

Strata supports multi-tenancy using data namespaces. The resource server uses namespaces to partition data; resources of separate namespaces do not interfere with each other. The same application installer can be distributed to customers of different businesses such as a hospital, bank or city-hall, but behaves differently like a customized reachability app for that business, e.g., to reach hospital, bank or city-hall staff. The auth-token used when connecting to the resource server determines the namespace, which in turn defines the tenant group, i.e., its tenant data and user interface branding (Fig.1).

6.6. ***Scalability and robustness***

6.6.1. *Websocket: keep-alive or not?*

Keeping the application logic in the endpoint further simplifies the server, and makes it scalable and robust. We use client-server WebSocket between the Strata app and the resource server. This is useful on web pages and desktop apps to receive asynchronous events such as incoming call. Moreover, periodic keep-alives can keep the connection active. When it fails, reconnection is attempted. Strata automatic fails over to a secondary server when the primary fails. However, such long running connections with keep-alives are not scalable, and do not work on mobile.

To conserve power, long running native mobile apps should not keep persistent WebSocket connection. Keep-alives or unbounded reconnection attempts are resource intensive and should be reduced. Our app stops keep-alives when the mobile device goes to sleep or the app is not in foreground. It uses exponential back-off timer with a cap to attempt reconnections, and stops them if the failure persists for some time. When the device becomes active again, it does a one-time check to detect connectivity and to reconnect if needed.

6.6.2. *Cloud messaging on mobile*

Without persistent WebSocket connection, the app's ability to receive asynchronous events is not trivial. By design, we avoid presence, and furthermore, we use low powered shared Google Cloud Messaging (GCM) that can receive events even when the device is asleep. When received, it then wakes up if needed, and connects to the server to fetch any call data, e.g., incoming invite or received text message. The server informs all the devices after recovery from a failure, so that the devices that had stopped their reconnection attempts can now connect. The caller sends the outgoing call invite via both the resource server and GCM, (using #11 and data of #9 in Fig.9), in case the receiver's device is asleep or not connected to the server.

If the caller sends an event via the resource server, the server responds whether the event was delivered to one or more subscribers. However, when the caller sends a GCM message, it does not get any feedback about delivery. Furthermore, the WebSocket may get disconnected not immediately but sometime after the device goes to sleep. This poses problem of reliably delivering the call invite event to the target within a reasonable time frame, especially when more reachability items are available for fallback.

To solve this, we delay the fallback to the next reachability item, and we let the caller re-send the invite via the resource server after a timeout. If the receiver is connected to the server, the GCM event is ignored as duplicate. If the receiver's device is asleep and not connected to the server, the GCM event wakes it up, and the app can then receive and respond to the second invite on the resource server. If the receiver is not available anywhere for this reachability item, the delayed fallback after a timeout tries the next item. Other platforms have alternatives to GCM, and are particularly useful on iOS where the browser immediately terminates WebSocket when the device goes to sleep.

6.6.3. *Signaling and media paths*

We use client-server signaling and full mesh media paths in the default communicator. The resource service scales like a pub/sub system using data partitioning, in-memory cache and subscriber aggregators. However, due to full mesh media flows, the client endpoint's CPU often gets congested with more than 3-party calls. Moreover, the upstream links of some endpoints may get congested with more than 4 or 5 participants. In that case, we can switch to a centralized conference server using our AMS app, if it turns into 4-or-more party call.

Ability to pick up an ongoing call from another device further improves robustness, e.g., if user is experiencing poor network connectivity on her mobile phone, she can pick up that call from her desktop.

7. Conclusions and Future Work

We have described the user reachability problem and how it is aggravated in WebRTC-based multi-apps environment. To solve the problem, we have presented the architecture and implementation of our multi-platform system consisting of separate contacts and communication apps. Our Strata Top9 app covers many WebRTC-based cross-platform apps for VoIP and multimedia conferences. The white-labeled Strata app can be customized for specific businesses. Many of our apps are focused on enterprise use cases, but the flexible architecture can include other social apps, e.g., we have created separate mobile apps for SIP-in-JavaScript and LAN video phone to connect to public VoIP service, and to discover and connect to others in the same local area network (LAN), respectively. We are also modifying the Vclick webapp to inject dynamic contacts from the browsing context to the Strata app, e.g., to show who else is viewing the department webpage.

Our work shows that many useful features such as user reachability and handoff across devices or apps are possible with user driven apps. We focus on user driven reachability and policy decisions, unlike a global location service or pair-wise federations to make our system useful in practice for emerging WebRTC apps. Endpoint driven apps are also useful when local context is needed, e.g., user's location in dialing out an emergency call. Separate resource servers can be used for different groups or organizations. Our resource oriented software architecture allows an app to dynamically pick the data server independent of where the app is loaded from.

Our apps are implemented with *write-once-run-anywhere* model, written in HTML5, JavaScript and CSS, and using Cordova Chrome Apps tools, are ported to wide range of cross-platform scenarios. We have described the design and implementation of Strata and various reachability apps with details on the resource based software, cloud hosting for multi-tenancy and cross-platform development.

In the future, instead of exposing the reachability items to the caller app, we will create a server side policy engine that will hide any sensitive data. The policy engine could use other contextual data, e.g., input from Global Positioning System (GPS) could indicate driving, and thus, disallow message or allow only hands-free call; underlying network with or without VPN, could affect the call security requirement and disable certain apps; received call could be transferred to a recordable bridge for accounting or if calendar shows a shared meeting; mobile data usage could be used to downgrade a video call to a low bandwidth voice; or background noise level could disallow a voice call, or trigger speech/text translation. Privacy of such detailed contextual input is paramount. Thus, a server side policy engine to aggregate and/or filter sensitive data is preferred.

Acknowledgments

The Strata Top9 project is a joint work with Steve Brock, Joyce Fong, Venkatesh Krishnaswmany and Laurent Philonenko. The following people have helped in integration or evaluation of some of our implemented apps: Biswajyoti Pal, Thiru Arjunan, Jaydeep Bhalerao, Gaurav Badge, Ramanuja Kashi, Stephen Whynot, P. Krishnan and Navjot Singh.

References

[1] J. Grégoire, "On embedded real-time media communications", Proceedings of the 1st workshop on all-web real-time systems, Bordeaux, France, Apr 2015.
[2] N. Paterson, "Walled gardens: the new shape of the public Internet", Proceedings of the 2012 iConference, ACM, 2012.
[3] R. Tworeck, "The walled garden in reverse – open web", Online, Mar 2013, http://webrtcstrategies.com/, retrieved Jan 2016.
[4] A.B. Johnston and D.C. Burnett, WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web, third edition, Digital Codex, 2014, ISBN 978-0985978860.
[5] L. Strand and W. Leister, "A survey of SIP peering", In proceedings of NATO architects of secure networks (ASIGE), May 2010, Genova, Italy.
[6] P. Saint-Andre, "XMPP protocol flows for inter-domain federation", XEP-0238, XMPP standards foundation, 2008.
[7] Sites that use or demo WebRTC, Online, http://www.webrtcworld.com/ webrtc-list.aspx, retrieved Jan 2016.

[8] F. Toutain, E. Huérou and E. Beaufils, "On webco interoperability", Proceedings of the 1st workshop on all-web real-time systems, Bordeaux, France, Apr 2015.

[9] K. Hänsge, S. Drüsedow, P. Chainho, M. Maruschke, "Signalling-On-the-fly", in Innovations in Services, Networks and Clouds (ICIN 2015), Paris, France, Feb 2015.

[10] J. Rosenberg et al., "SIP: session initiation protocol", RFC 3261, IETF, Jun 2002.

[11] P. Saint-Andre, J. Hildebrand, "reachability addresses", XEP-0152, XMPP standards foundation, 2014.

[12] X. Wu, H.Schulzrinne, "Programming end system services using SIP", IEEE international conference on communications (ICC), Anchorage, Alaska, May 2003.

[13] J. Rosenberg, H. Schulzrinne, P. Kyzivat, "Caller preferences for SIP", RFC 3841, IETF, Aug 2014.

[14] M. Boussard et al., "Communication hyperlinks: call me my way", 13th international conference on intelligence in next generation networks, (ICIN), 2009, Bordeaux, France.

[15] S. Shanmugalingam, N. Crespi, P. Labrogere, "My own communication service provider", International congress on ultra modern telecommunications and control systems and workshop, 2010, Moscow.

[16] 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.

[17] K. Singh and V. Krishnaswamy, "Building communicating web applications leveraging endpoints and cloud resource service", IEEE International Conference on Cloud Computing, Santa Clara, CA, Jun-Jul 2013.

[18] K. Singh and J. Yoakum, "Vclick: endpoint driven enterprise WebRTC", IEEE International Symposium on Multimedia (IEEE ISM), Miami, FL, Dec 2015.

[19] G. Billock, J. Hawkins, P. Kinlan, "Web Intents", W3C draft, 2013, http://www.w3.org/TR/web-intents/.

[20] Run Chrome Apps on mobile using Apache Cordova, https://developer.chrome.com/apps/chrome_apps_on_mobile, accessed Jan 2016.

[21] K. Singh, "User reachability in multi-apps environment", IEEE International Symposium on Multimedia (IEEE ISM), Miami, FL, Dec 2015.

[22] K. Singh and J. Buford, "Developing WebRTC-based team apps with a cross-platform mobile framework", IEEE Consumer Communication and Networking Conference (CCNC), Las Vegas, NV, USA, Jan 2016.

[23] T. Leithead and A. Eicholz, "Bringing componentization to the web", http://windowsforum.com/threads/211241, retrieved Jan 2016.

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

[25] Otalk: an open-source platform for building realtime applications, online, http://otalk.org, retrieved Jan 2016.

[26] A. Prokop, "Solving the WebRTC interoperability problem", online, http://www.nojitter.com/post/240169575/solving-the-webrtc-interoperability-problem, retrieved Jan 2016.

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

[28] T. Berners-Lee, "Socially aware cloud storage", notes on web design, Aug 2009, http://www.w3.org/DesignIssues/CloudStorage.html.