RTCWeb
Internet-Draft
Intended status: Standards Track
Expires: January 10, 2013

M. Isomaki
Nokia

July 9, 2012

RTCweb Considerations for Mobile Devices
draft-isomaki-rtcweb-mobile-00

Abstract

Web Real-time Communications (WebRTC) aims to provide web-based applications real-time and peer-to-peer communication capabilities. In many cases those applications are run in mobile devices connected to different types of mobile networks. This document gives an overview of the issues and challenges in implementing and deploying WebRTC in mobile environments. It also gives guidance on how to overcome those challenges.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 10, 2013.

Table of Contents

1. Introduction

Web Real-time Communications (WebRTC) provides web-based applications real-time and peer-to-peer communication capabilities. The applications can setup communication sessions that can carry audio, video, or any application specific data. To be reachable for incoming sessions setups or other messages, the applications must keep persistent connectivity with their "calling site".

In the last few years, mobile devices, such as smartphones or tablets, have become relatively powerful in terms of processing and memory. Their browsers are becoming close to their desktop counterparts. So, from that perspective, it is feasible to run WebRTC applications in them. However, power consumption and highly diverse nature of the connectivity still remain as specific challenges. A lot of work is done to address these challenges in e.g. radio technologies and hardware components, but still by far the most important factor is how the applications and protocols and application programming interfaces are designed.

Section 2 of this document gives an overview of the characteristics of different mobile networks as background for further discussion. Section 3 introduces the specific issues that WebRTC protocols and applications should take into consideration to be mobile-friendly.

The current version of the document misses all references and lot of details. It may have some errors. Its purpose is to get attention to the topics it raises and start discussion about them.

2. Common mobile networks and their properties

The most relevant mobile networks for WebRTC at the moment are Wi-Fi and the different variants of cellular technologies.

Many characteristics of the cellular networks are covered in Section 3 in the context of the particular issue under discussion. The following is a very brief description of the power consumption related properties of WCDMA/HSPA networks. The details vary, but similar principles apply to other cellular networks, at least GPRS/EDGE and LTE.

In simplified terms, the WCDMA/HSPA radio can be in three different types of states: The power-save state (IDLE, Cell_PCH, URA_PCH), a shared channel state (Cell_FACH) or a dedicated channel state (Cell_DCH). The power-save states consumes about two decades less power than the dedicated channel state, while the shared-channel state is somewhere in the middle. The state machine works so that if

a device has only small packets (upto ~200-500 bytes) to send or receive, it will allocate a shared channel, that operates on low data rate. If there is more traffic (even a single full size IP packet), a dedicated channel is allocated. Starting from the power-save state, the channel allocation typically takes somewhere between 0.5 and 2 seconds, depending on the network and the exact power-save state. Only after that, the first packet is really sent. If two cellular devices were to exchange packets with each other starting from the power-save state, the initial IP-level RTT could be easily 3-4 seconds.

The channel is kept for some time after the last packet has been sent or received. The dedicated channel drops to power-save via the shared channel. The timers from dedicated to shared and shared to power-save are network dependent, but typically somewhere between 5 and 30 seconds. So, in some networks sending a single ping every 30 secods is enough to keep the power consumption constantly at the maximum level, while in others the power-save state is entered much faster. The total radio power consumption does not actually depend so much on overall volume of traffic, but on how long a dedicated or shared channel is active. So, for instance a 1 kB keep-alive sent every 30 seconds for an hour (total ~100 kB of traffic) consumes much more (even an order or magnitude more!) than a single 10 MB download, assuming that will finish in a minute or two.

The applications have no control over the radio states, but the Operating System and the Radio Modem software can do something about them. In the newer specifications (and devices and networks) it is possible for the device to explicily ask the radio channel to be abandoned even immediately after the last packet. For instance, if the device were somehow to know that no new packets are to be sent for some time, it could do such signaling and save power.

The bottom line is that applications and protocols should keep as long intervals between traffic as possible, giving the radio as much low-power time as possible. The intervals that are more than a few seconds may help, but at least intervals that are longer than 30 seconds will definitely help. On the other hand, the initial RTT after an interval will be long. This issue is covered in Sections 3.1 and 3.2.

The other key characteristic of cellular networks is that they have long buffers and run link-layer in "acknowledged" mode, meaning all lost packets are retransmitted. This means TCP will easily create long delays and ruins real-time traffic. This is covered in Section 3.4.

The third characteristic is that mobile devices often change networks

on the fly, typically between cellular and Wi-Fi. Most devices only run a single interface at a time. From networking perspective this means that the device's IP address changes, and e.g. all its TCP connections are lost. This is covered in Section 3.3.

3. Specific issues and how to deal with them

3.1. Persistent connectivity to the Calling Site

Many WebRTC apps want to be reachable for incoming sessions (JSEP Offers) or other types of asynchronous messages. For this purpose they need some kind of a persistent communication channel with their "Calling Site". Two standard approaches for this are WebSockets and HTTP long-polling. In both of these cases a TCP connection is used as the underlying transport.

Most cellular networks have a firewall preventing incoming TCP connections, even when they allocate public IPv4 or IPv6 addresses. Also NATs are becoming more popular with the exhaustion of IPv4 address space. The firewall and NAT timers for TCP can range between 1 and 60 minutes, depending on the network. To keep the TCP connection alive, the application needs to send some kind of a keep-alive packets with high enough frequency to avoid the timeout.

If the WebRTC app intends to run for a long periods of time (even when the user is not actively interacting with it), it is of utmost importance to keep this keep-alive traffic as infrequent as possible. Every wake-up of the radio consumes a significant amount of power, even if it is needed just for sending and receiving a couple of IP packets. It makes a huge difference, if there are for instance 6 vs. 60 of these wake-ups every hour. A naiive application may want to make it sure it sends frequently enough for all possible networks. That leads to unacceptable power consumption. A smarter application will try to figure out a suitable timeout for a given network it is using, and can save a lot of power in networks with longer timers.

There are further strategies to manage the keep-alives so that they consume least amount of power. It is best to send as small keep-alive messages as possible. HSPA/WCDMA networks have a special shared radio channel (FACH) that can carry small amounts of traffic. Its power consumption is typically less than half of the dedicated channel. Depending on the network, a packet of a couple of hundred bytes will usually only require FACH, while a thousand byte packet will require the dedicated channel to be activated. So, a WebSocket PING-PONG is better than an HTTP POST or GET with all the Cookies and other headers attached. If there are multiple applications or connections to be kept alive, the Browser or the underlying platform

should offer some kind of a synchronization for them, so that the radio is woken only once per cycle.

The most efficient approach would be to multiplex the initial incoming messages for all applications over the same TCP connection. This would require the use of some kind of a gateway service in the network. Such "notification" services are available on many platforms, but at the moment they are not typically available for browsers or web applications. It would be useful to standardize or develop Javascript APIs for this purpose. There is W3C work on Server-sent events. Also, the Open Mobile Alliance (OMA) has started work on standardized "notification" services. Be the services standards based or proprietary, the most relevant part to get done would be to give WebRTC and other Web applications access to them. Such services are always subject to privacy concerns, so at minimum the messages passed over them should be end-to-end encrypted. (Traffic analysis threats would still remain.)

3.2. Media and Data channels

Real-time media (audio, video) is typically sent and/or received constantly, while the media channel is established. This means radio needs to be on constantly, and there is little for the application to do to preserve power. (Choosing a hardware accelerated video codec over a non-HW-supported one is one thing the application may be able to influence.) At least in LTE there are techniques called Discontinuous Transmission/Reception (DTX, DRX), that operate even in the timeframe of tens of milliseconds and can affect power consumption e.g. for VoIP. It is an open issue if WebRTC stacks can be somehow optimized for them.

The Data Channel may however be often low-volume or even idle for long periods of time. For instance an IM connection may be idle for minutes or even hours. There can be many apps that want to keep such a connection available just in case there is some traffic to be sent or received infrequently. The WebRTC Data Channel is based on SCTP over DTLS over UDP. This means it needs keepalives in the order of 30 seconds in cellular networks, meaning the radio will be active most of the time even if no user traffic is sent. It is not possible to keep such a channel on for a long time due to power consumption.

Applications can choose different strategies to deal with this problem. One approach is to avoid Data Channels completely for low-volume or infrequent traffic and send it via the Web servers over HTTP or WebSockets. This is probably the best approach. The other approach is to tear down the Data Channel after some timeout and re-establish it only when new traffic needs to be sent. This may create some lag in sending the first message after the interval. The third

option is to transport the Data Channel over TCP, e.g. using a yet undefined "HTTP tunneling fallback" mechanism. This would be almost identical to the first approach, except that logically the application would still be using a WebRTC Data Channel. It is not yet clear if this will be feasible due to ICE concent refreshes that may need to occur frequetly as well (every 30 seconds?). They are sent end-to-end so one side of the Data Channel can not by itself even affect their rate.

3.3. Recovery from interface switching

Most mobile platforms only support Internet connectivity over only one interface at a time. In practice this is either a cellular or a Wi-Fi interface. From radio hardware perspective there would be no need for such a limitation, but it is driven by simplicity and power preservation. The devices typically have a hard-coded or configurable priority order for different networks. The most common policy is that any known Wi-Fi network is always preferred over any cellular network, but even more complex policies are possible.

When the device detects a higher priority network than the one currently in use, it will by default attach to that network automatically. After a successful attachment to the new network, the device turns the old network (and interface) off. In most platforms applications have no control over this. In a typical situation the switch-over leads to a change of IP address, and for instance all TCP connections becoming disconnected, and any state tied to them needs to be recreated.

It is important that WebRTC applications are made robust enough to survive this behavior. Many native applications deal with it by listening to "disconnect" and "reconnect" events through the APIs they are using. For WebRTC apps the first priority is to re-establish its "signaling" connectivity to the "Calling Site". If that connectivity is based on a WebSocket, the application needs to react to the "onerror" event through the WebSocket API and establish a new connection and setup all state related to it. (Say, if the application was using SIP over WebSockets, it might have to re-REGISTER on the SIP level.) If the disconnect was caused by interface switching and the switch-over succeeded cleanly, it would be possible to setup the new connection immediately. In some cases the disconnect could last longer, and the application would have to retry the connection until connectivity is regained.

It would be advisable to make the reconnect step as lightweight as possible in terms of RTTs required. For the browser and the web application platform, it is important that the "disconnect" event gets propagated to the applications as fast as possible.

For HTTP long-polling, it would similarly be important to notice that the underlying TCP connection has become stale, and a new poll needs to be sent as quickly as possible.

The application may also attempt to update any peer-to-peer sessions it is having at the time of the switch-over. At this point of RTCWeb standardization it is not yet clear how much control over this the protocols and APIs will exhibit. There are many layers on which the recovery can be done. It is possible to try to deal with it using ICE. This would require knowing when the currently used ICE candidate becomes unusable, as it is bound to a removed interface. The failure of ICE connectivity checks provide that information, but possibly after some delay. (Frequent connectivity checks are not an issue as long as media is actively sent or received, but would be costly over an idle or low-volume media channel, such as a Data Channel. If media traffic is infrequent, the speed of detection may not be that critical for user experience anyway.) If an interface really became unusable, it would be better to have an explicit event to signal that all ICE candidates bound to it are likely unusable as well, so the application could act immediately. If a new interface became available, the application could restart ICE and start using the new candidates gathered.

The PeerConnection API offers a few events for these purposes, at least "icechange" and "renegotiationneeded". With these the application can learn about problems with the currently used candidates. There is also a method "updateIce" by which the application can restart the ICE candidate gathering process. It is however not yet entirely clear how these event handlers and methods should be best used to deal with an interface change, and whether they even are a feasible tool for dealing with it. It is also important to note that no new offers or answers could be sent or received until the "signaling channel" (e.g. the Websocket connection) was first re-established.

If the lower-level instruments fail, the application could create a new PeerConnection, and recreate the media channels. This would be a heavier operation, but in some cases it might still be better than leaving the recovery entirely to the user, i.e. explicitly making a new call from the UI.

There are certain things that the underlyind platform (Operating System, Connection Manager etc.) can also implement to make interface switching smoother for the applications. One possibility would be to keep the old interface available for a short duration even after a new higher priority interface becomes available. This would allow applications to deal with the change in a more proactive fashion. There are also protocols such as Multipath TCP that could be used to

switch e.g. WebSocket connections to a new interface without always resorting to the application support.

3.4. Congestion avoidance

Cellular mobile networks have notoriously large buffers. Their link layers also typically operate in an "acknowledged" mode, meaning that the lost frames (or packets) are retransmitted. Retransmission creates head of line blocking on the queue. This means packets are seldom lost, but delays grow large. The individual users or endpoints are often isolated from each other so that the network capacity is divided among them more or less evenly. However, all traffic to and from the same endpoint ends up in the same queue. In WebRTC context this means that plain TCP traffic will easily ruin real-time traffic due to the buffering.

WebRTC protocols should be desinged to avoid this. If Data Channels transfer a lot of data in parallel to the real-time streams, they should not use the loss-driven (TCP) congestion control algorithms but something that reacts to queue growth much faster. IETF LEDBAT WG may have something to offer for this case. If the browser wants to protect its real-time strams in general against all TCP (HTTP, WebSocket) traffic, it might be best for it to also restrict the number of simultanous TCP connections in use, for instace to retrive a website. The HTTP 2.0 work done in IETF HTTPBIS WG should prove helpful in this case.

Cellular networks also do have their in-built Quality of Service mechanisms that can be used to differentiate service for different packet flows. These are not widely used in HSPA/WCDMA, but LTE may change the situation to some extent. The QoS policy is enforced by the network, and requires a contract with the operator. It is thus likely only available for services with some relation to the access operator. How the WebRTC application or the browser deal with that is TBD. Technically DiffServ marking is probably the only dynamic approach to indicate the priority of a particular flow.

4. Security Considerations

Not explicitly covered in this version.

5. Acknowledgements

Bernard Aboba and Goeran Eriksson provided useful comments to the document. Dan Druta has worked on Web notifications in the context of WebRTC.

6. References

Author's Address

Markus Isomaki
Nokia
Keilalahdentie 2-4
FI-02150 Espoo
Finland

Email: markus.isomaki@nokia.com