Prototype Investigation Update

Julian Horn

October 8, 2014

Agenda

- Reporting progress building on the Microsoft's earlier prototype CLI from that treats ripple as a platform name
- To Review, the idea in the prototype was:
  - Reuse CLI/cordova.js/plugin.xml machinery (<js-module> tag) to bring in JS for emulation during elaboration of cordova.js in ripple platform
  - Add JS to be invoked via exec under emulation below src/ripple in plugin
  - Files self-register with cordova/exec/proxy, so exec will call them
- Limitations
  - Can't extend emulator GUI this way
  - Tricky to manage "overlay ui", i.e. UI that appears on device

Exploration: Externalize Geolocation Support

- Move all the support for geolocation API, geolocation UI panel and cooperative code into a modified geolocation plugin
  - Extend Ripple as needed to pull this in during boot sequence
- Goals:
  - Minimize changes needed in original emulator sources
  - Minimize interface between emulator and Cordova CLI; (in fact no further CLI changes were needed)
- Just a refactoring; no user-visible changes EXCEPT:
  - Geolocation panel disappears if program doesn't use geolocation plugin

Review: Ripple Code Structure

- Ripple sources follow certain structural conventions
  - Core modules in lib/client
  - Ui modules in lib/client/ui/plugins
  - Cordova API emulation in lib/client/platforms/cordova/2.0.0/bridge
- API emulation modules are associated with exec service name via a mapping object named “emulator” in bridge.js

```
emulator = {
   ...
   "Geolocation": ripple('platform/cordova/2.0.0/bridge/geolocation'),
```

- Ripple intercepts Cordova exec; dispatches through mapping obj (first prototype removed this interception)

Review: How Ripple Emulates Geolocation

- Typical MVC pattern; intermediate "model" module manages state and thereby decouples UI from API
  - User manipulates UI => model changes
  - API exec/native emulation => Gets values/callbacks from model object
- Intermediate object implemented by `lib/client/geo.js`
- API exec/native layer implemented by `platform/cordova/2.0.0/bridge/geolocation.js`
- UI is typical emulator "panel" implemented by files below `lib/client/ui/plugins: geoView.js, geoView/panel.html`
- UI also uses global CSS, global images, and third party code

Review: How Ripple UI is Initialized/Assembled

- In addition to panels, Ripple has two other standard "UI plugin" types: dialogs and overlays
  - Overlay UI appears in the inner frame, as if from program under test
  - Dialog UI pops up, as if from Ripple
- UI plugin's HTML must reside in a whose ID is panel-views, overlay-views, or dialog-views, depending on what it is
- At ui initialization time accordions are assembled from panels
  - Panel is included if named in specification corresponding to current platform; "system" panels used in every platform
  - Panel order and left/right position is saved in emulator settings

How Emulator Code is Embedded in Plugin

- Directory src/ripple corresponds to `cordova/2.0.0/bridge`
  - Contains files like `geolocation.js` (as in previous prototype)
- New directory src/ripple/emulator corresponds to `lib/client`
  - Contains files like `geo.js`
- New directory src/ripple/emulator/ui corresponds to `lib/client/ui/plugins`
  - Contains files like `geoView.js` and `geoView/panel.html`
  - CSS used in *xxx*/*yyy*.html belongs in *xxx*/overlay.css
- Must name every file in plugin.xml (or CLI will drop it)
  - Files in src/ripple use <js-module>
  - All other files use <asset>

Why do you need that Emulator folder?

- Need to distinguish different kinds of files because they are treated differently as they are merged into Ripple
  - Emulator code needs to run in the context of the emulator UI window, not the inner frame where program under test runs
  - JS in emulator core must be loaded and initialized when the emulator boots, not when the program under test loads cordova.js
  - Core modules are invoked when UI initializes itself, so must exist first
  - UI initialization code looks for DOM elements, so must exist first
- Module naming conventions are different: Ripple expects to find UI modules as ripple('ui/plugins/xxx'), not ripple('xxx')

(Partial) Geolocation Plugin Tree

plugin.xml
www <- JavaScript interface to plugin goes here (unchanged)
src/ripple/ <- "Native" code part of plugin goes here (platform-specific)
 └ geolocation.js
    emulator/
     └ geo.js
        ui/
         └ geoView.js
            geoView/
             └ panel.html
                overlay.css <- extracted from ripple.css
                images/
                 └ arrow.png, compass.png <- moved from assets/client/images

Building the Prototype Geolocation Plugin

- Move files from Ripple code base to plugin tree
  - Extracted CSS for Geolocation panel from ripple.css and put in overlay.css (not strictly necessary, but it's better structure)
    - Had to move image files referenced from overlay.css via relative path
  - Seven files in all: geo.js, geolocation.js, geoView.js, panel.html, overlay.css, compass.png, and arrow.png
- Plugged in JS doesn't run inside Ripple module, so it can't see global names; had to invent a way to export them
  - Solution: added 'thirdparty' ripple module

```
$ = ripple('thirdparty').$; // now I can see $
```

Summary of Source Changes

- In geolocation.js:
  1. Replaced “ripple” with “parent.ripple” everywhere (as before)
  2. Added self-registration line (as before):
     ```
     require("cordova/exec/proxy").add("Geolocation",module.exports);
     ```
- In geoView.js:
  1. Import third-party global packages
     ```
     var
          OpenLayers = ripple('thirdparty').OpenLayers,
          $ = ripple('thirdparty').$;
     ```
- That’s it!

Ripple Client-side Changes (the beef)

- Dynamically add plugin contributions during boot sequence
  - Enumerate plugins folder to identify plugins; for each plugin:
  - Look for "emulator" subdirectory
  - If present, enumerate JS files and look for ui subdirectory
  - If present, enumerate JS files and look for subdirectories
  - For each subdirectory, enumerate CSS and HTML files
- Must wait for everything to load, otherwise you can get transient errors in initialization e.g. looking for element that doesn't exist
- Modify specification, initialization to exclude no-longer-built-in UI
- Modify ui initialization include contribution of plugged-in UI

Two New Modules

- pluginExtensions.js:
  - Enumerates plugins folder, discovers ripple contributions to core and UI
  - Loads core files via <script> tags
- pluginUi.js
  - Enumerates content of UI extensions discovered earlier
  - Loads CSS, HTML, and JS for UI extensions
  - Calculates specification for UI plugins

Treatment of Cordova "exec"

- First prototype did not intercept Cordova exec; always calls "fail" if no emulation implementation exists in plugin
- This prototype continues to intercept exec function; uses implementation registered with cordova/exec/proxy if present otherwise uses emulator's built-in implementation
- Two reasons for this:
  - Ripple has code that repairs call to missing API implementation; want to keep and later extend this to handle third party plugins
    - This code belongs in emulator, not in cordova.js for ripple platform
  - Emulator may want to provide code if plugin doesn't (or if it's broken)

Ripple Server-side Changes

- Added three new routes to help with plugin discovery and input
- /ripple/extensions – used in <script> for core JS files
  - Adds "ripple.define(*'xxx'*, … around file contents
- /ripple/uiextensions – used in <script> for UI JS files
  - Adds "ripple.define('ui/plugins/*xxx*', … around file contents
- /ripple/directory - for directory enumeration
  - Ripple client side doesn't know where project sources are, so it's not so easy to use file system access to explore plugin tree
  - More elegant solutions are clearly possible; could have CLI enumerate files and create an emulator extensions JSON file emulator could read

Some Thoughts

- Tight binding between emulator and plugins may be problematic for derived emulator vendors like Intel
  - Can fork the emulator, but can't fork the entire universe of plugins
  - Bugs in plugin look like emulator bugs that emulator vendor can't fix
- Plugin-emulator interface is problematic
  - Every emulator file is a module; everything is public; same is true for DOM node IDs, CSS class names and so on
  - Plugin JS can call anything, so there is really no "interface"
  - Any change to the emulator can potentially break a plugin

More Thoughts

- Plugin author == tools developer model may be problematic for non-core plugins
  - Really different skill set
    - I don't have high hopes that many plugin authors will supply ripple code
    - Others disagree; we'll have to wait and see how this plays out
- Testing is problematic
  - It's easy to cause fairly subtle bugs in the emulator
  - Ripple unit test framework won't work on plugin-resident code
  - No good "integration test suite" for "built" form of Ripple
  - Hard to test for plugin that works in Ripple and fails in XDK

Follow Up

- Prototype Ripple sources is available in fork of Ripple repository https://github.com/IntelXDK/incubator-ripple
  - Need to add a little readme that describes what this is
  - Assumes the MSOpenTech prototype CLI that recognizes “ripple” as valid platform name is present on your path
- Prototype gelocation plugin is available in fork of plugin repo https://github.com/IntelXDK/cordova-plugin-geolocation
- Look at branch ripple_platform_prototype
- Questions/feedback to Julian.c.horn@intel.com