Proximus Distributed Ad Server System [DASS]

Release Date: October 2, 2001

Author: Ryan Walker [ryan.walker@kazestudios.com]

Background:

DASS was originally conceived approximately 18 months ago. The concept of the product was to build a mobile ad distribution network with the ability to distribute mobile advertising using Global Scales Sheeva devices. At the time of this document, version 1.5 is distributed into the field and version 1.6 is about 10 days from being released that corrects some implementation and architectural challenges.

Challenge:

The current architecture is not designed to scale. After a very close look at the underlining code, it was discovered that the core framework was not designed using best practices. After 6 months of actively trying to support the product in the field, it was realized the distribution model and the design was overly complicated and required a considerable amount of refinement. There are also business rules that have wavered from release to release that have further reduced the stability of the code and the interfaces that are to be supported. During the 1.6 development sprint, moving to a more simplified delivery model became painfully obvious. In addition, there were several methodologies in practice that caused a significant use of memory and resources for the host machine.

This document attempts to outline an approach to solve the challenges discovered within the shipping DASS product.

Approach Server:

To simplify the delivery model of campaigns and the aggregation of device logs, a RESTful web service model is being proposed. The DASS system does not require an active real-time or near-real-time integration model in order to be successful.

To accomplish this approach, the server will be implemented in J2E 1.6, utilizing a simplified set of RESTful URI’s over HTTP Web Bindings. This model was chosen because it scales easily without the implementation of new code and can have an initial SSL security model. One of the advantages here is that the cost for producing a DASS server is reduced to a mid-level virtual server with no more than 250 MB + 2GB of ram.

1. Server Technologies:
   1. Java SDK 1.6
   2. RESTful Servlet Engine – Jersey 1.8 + Grizzly 1.9
      1. Note: Grizzly was utilized here because it provided seamless integration with Jersey and can easily be upgraded to Glassfish if more power is required.
      2. Again, this was utilized for its simplicity and streamline implementation. Configuration takes minutes to setup and run.
   3. JAXB 2.2 (JSON/XML payload support)
   4. MySql 5.2
2. Request Model;
   1. The request model consists of four distinct resource sets. These sets identify the data management functionality for the system. It is designed to be a stateless transaction model that allows integration with ANY UIX (i.e. Ajax+HTML, Flash, Mobile Native Code, etc.)
      1. Resources
         1. Users: (System Users, Simple Role Management, Authentication, Authorization);
            1. The sole purpose for this system is to link resource management with an accountable contact.
            2. This uses a simplified security model where the users’ credentials consist of a unique email address, an MD5 encoded password + a phone number. No other details about the user will be maintained by this system.
            3. URI:

/users

**POST**; ability to add a new user. Use requires the USER\_ACCOUNT\_CREATION role.

**GET**; ability to list users. (Note: this should be constrained by http request attributes that limit access to users with the USER\_ACCOUNT\_CREATION role.

/users/{userid}

**GET**; access to details about a specific user.

(Note: passwords will not be returned within this request.)

**PUT**; ability to update user details.

(Note: passwords cannot be retrieved with the chosen model. ALL passwords will be stored in MD5 format and compared directly during a session request.)

**DELETE**; ability to “virtually” delete an account from the system.

Access granted to users with the USER\_ACCOUNT\_CREATION role.

Note: users are not actually deleted from the system.

/users/{userid}/sessions

**GET**; provides a history of login sessions for the specified user.

**Query String Params**:

**startdate –** date to limit the history that is being returned.

**stopdate –** date to limit the end of the history to be returned.

**page –** offset integer of the reports.

**limit** – number of items to include per page.

**POST**; provides the ability to validate a users email + password combination to simulate session management.

The data returned consist of a time sensitive nounce that validates requests to the api. This nounce should be included as an http host header **DASS-SESSION-KEY**.

/users/{userid}/sessions/{sessionid}

**DELETE**; immediately expires the nounce associated with the provided user id.

* + - 1. **Entity** 
         1. Formerly company in the previous system. Entities were introduced to handle the various relationships that Proximus maintains.
         2. Entity Types: sales & development partners, clients and resellers.

/entities

**GET**; list of all entities available within the system. Only allowed access to users with ENTITY\_ACCOUNT\_CREATION access.

**POST**; create a new entity. ENTITY\_ACCOUNT\_CREATION access required.

/entities/{entityid}

**GET**; get details about this entity.

**PUT**; manage/update details about the specified entity id.

**DELETE**; virtually delete the specified entity id.

/entities/{entityid}/users

**GET**; list of users associated with this entity. This is provided as a logical shortcut to retrieve users associated with an entity. Use the **Users** resource to manage the resource.

/entities/{entityid}/campaigns

**GET**; list of campaigns associated to the specified entity.

/entities/{entityid}/devices

**GET**; list of devices associated to the specified entity.

* + - 1. **Campaigns**
         1. Unified resource that handles every type of campaign present and future.

Note: campaigns are filtered to a specific company based on the callers DASS-SESSION-KEY to ensure access to the specified user account.

A GLOBAL\_RESOURCE\_ACCESS flag defines global access to all resources. Access to resources that are not linked by the DASS-SESSION-KEY are answered with a 403 – Forbidden Access code unless the GLOBAL\_RESOURCE\_ACCESS flag is available on the callers DASS-SESSION-KEY.

Campaign resources are “Pushed” to the server via Ajax binary data and stored to the servers file storage system. Information storage should be segmented by entity id for faster access and easier resource management.

* + - * 1. Campaign Types: Bluetooth, Wifi/SSID, Kiosk

/campaigns

**GET**; list of campaigns within the system.

**POST**; create a new campaign.

/campaigns/{campaignid}

**GET**; available information about the provided campaign id.

**PUT**; update campaign details.

/campaigns/{campaignid}/devices

**GET**; list devices that this campaign has been distributed to.

/campaigns/{campaignid}/journal

**GET**; this returns the available batch ids order by date and/or device associated with the provided campaign id. Journals contain a timestamp, device serial number plus a set predetermined size of access / error log entries for the associated device. Query parameters should be added to provide granular report access in addition to specific types of campaigns and/or campaign types.

/campaigns/{campaignid}/journal/{batch-id}

**GET**; access the batch information associated with the batch id in addition to each log entry.

* + - 1. **Devices**
         1. Resource that handles the registration, management and central repository for device logs.

Note: Device types are currently identified for vanity reasons.

When the Kiosk campaign becomes active, device type will play a role in determining the type of campaigns that may be distributed to the device.

* + - * 1. Resources:

/devices

**GET**; list of registered devices within the system. The term registered is loosely used here to reference devices that have been provisioned.

**POST**; create/register a new device to be provisioned. This can be completed manually by a UIX interface or auto-magically when a device self registers itself.

**Self-Registration** is defined as the process of synching with the server to provide location, serial number and fundamental device information.

/devices/{deviceid}

**GET**; a single device described by the specified device id.

The details include the following:

Device type (sheeva, guru, dream plug v1, dream plug v2, dream plug v3 and display plug aka kiosk)

What version of the Proximus software is installed?

How long has it been online since last reboot?

What customer is the device assigned to?

Serial number & device nickname.

**PUT**; used to update device details related to company registration, serial #’s & device nickname.

/devices/{deviceid}/actions

**GET**; returns an ordered list of pending actions for a device. Each action has an action id, instruction, resource uri to support the instruction, created timestamp and an expiration timestamp.

**POST**; insert an instruction in the action queue for a device.

At this time, available instructions are restricted to a few operations.

Install a new campaign.

Force a journal entry post.

Install an update to the application software.

Force a reboot

/devices/{deviceid}/actions/{actionid}

**GET**; retrieve the full instruction set with details/comments if provided.

**PUT**; re-activate, de-activate, adjust settings for the instruction defined by the action id.

/devices/{deviceid}/pulse

**POST**; used to notify the system that the device is still online. The pulse should be sent on a 2:1 or 3:1 sample rate to ensure that the device is given enough opportunities to notify the system that it is still online.

The return payload from a pulse action may be utilized to provide update instructions for the device. (i.e. reboot, download a new version of a campaign, install a new campaign, etc.)

/devices/{deviceid}/journal

**POST**; used to submit device logs in a batch form to the server.

* + - 1. **Infrastructure**
         1. This section is experimental and may later be divided out into smaller resources.
         2. The purpose of this resource is to centralize the distribution of device utilities and software.

/sysops

**GET**; List of available nodes that are available to the Proximus network. May require restriction to this resource as an open get.

**POST**; add a new node to the resource pull. This may be restricted and not allowed from the API interface. TBD.

/sysops/{nodeid}

**GET**; retrieve details about the system node.

Details include

Product or development node.

Server software version.

Device client software version.

Device client software updates.

**PUT**; push updates for distribution from the node.

The sysops/{nodeid} will be the repository for all device software updates.

**DELETE**; decommission a node.

Changes the status of the node to offline.

Any requests to retrieve updates will be answered with a 403 permission denied or 404 resources not found. TBD.