Push Design Draft

General rules:

1. do as much business logic as possible in Javascript, only UPnP specific interfacing will go to C++
2. use existing SB Library capabilities to store and manage device playlists
3. use existing MediaServer classes to obtain media metadata (DLNA profiles, properties, album art)
4. the nature of push does not ask for immediate reactions (in terms of hunting millisecond timeouts) so synchronous operations will be avoided as much as possible. XPCOM interface will be designed as asynchronous (no immediate return values). Platinum operation is asynchronous by nature (HTTP timeouts).
5. make as an integral part of existing MediaServer service to simplify UPnP engine startup and shutdown. The UPnP engine is understood as application wide singleton, so nontrivial syncing issues may arise on addon startup and shutdown, if the MediaServer and Push were separate services.

Dependencies and separation of concerns:

- **GUI (JS)** is heavily Songbird specific. Responsible for managing the playlists, hence communicating with SB library and issuing commands to the device abstractions. It should not care about UPnP specific issues like protocol formatting or what method is needed to set they playback URI.
- **Service** is Mozilla specific but not necessarily Songbird specific. Depends on Platinum. Responsible for keeping the state information about the devices and translation of events and media items between frontend and network layer. It doesn't care about playlist management and have no reason to access the Songbird library directly. However will be calling into existing MediaServer code.
- **Platinum** is not dependent on Mozilla neither Songbird and is an event source and push action sink for the Service.

Identified abstract objects:

Network is a representation of UPnP cloud of devices. Is a source of events about devices appeared and lost in the network. Gives a possibility to manually enforce a discovery, i.e. refresh the UPnP cloud content. Will result in emitting series of events for each device appeared/lost. The actual timing of the enforced discovery (how long to wait for the devices to finally appear or not appear) is a subject of investigation.
Interfaces: sbISharingPushNetwork.idl, sbISharingPushNetworkListener.idl

Device is a representation of UPnP MediaRenderer. Is a source of events about device state transitions and property changes, including failed action requests. Gives functions to invoke various playback related actions on the device.

Interfaces: sbISharingPushDevice.idl, sbISharingPushDeviceListener.idl

Various findings:

Platinum has its own pool of worker threads. Javascript runs in frontend GUI thread. No need for introducing another thread in Service (C++) layer has been identified. Service appears to be enough as a threadsafe stateful interface between UPnP operation and UI. Repeated tasks (like manual replacement for insufficient LastChange scope) might be implemented as Platinum async tasks.

Displaying of devices which are known to be existing on the network in the past but are not currently, is considered inappropriate for UX. Hence there is no need to keep persistent listing of devices (in database or elsewhere). Only the devices currently on the network are being tracked in Service. However when device reappears on the network, user would perhaps expect that the last known playlist reappears (if it is still valid). Naming convention for playlists will be chosen so that a renewed device can find its playlist anytime later. Some UI way of removing orphaned playlists will need to be figured out (for device which the user is convinced that will never again appear) - perhaps in Settings pane for Sharing.

Playlist management will be done entirely in Javascript. The reason is that SimplyShare asks for nonlinear playback selections (shuffling, repeating) and moreover user can shuffle the tracks manually at any time. Transporting whole playlist order to the Service backend at every such indirection is unfeasible. Javascript will be responsible for sending the wanted playback item to the Service as soon as possible, and do it again whenever the already set item might change (for instance, when user manually shuffles a linear playback)

While UPnP AVTransport defines Prev() and Next() functions, these are not related to our mode of operation. It is suitable for either a pre-created, fixed playlists (like m3u files) or a remote control of prerecorded device (like CD player). The only exception is usage of Next() instead of Play(), when the sink device supports SetNextAVURI (UPnP optional). This will be handled at the level of Service as UPnP specific implementation. Javascript layer will only know how to set next item and start playback (of the first item).

A single given media item assigned to playlist is considered constant, except borderline cases like changing MP3 tags or replacing an album art. Furthermore, many actions may occur on the same media item, like scrubbing, pausing, resuming and replaying over in the playlist. Furthermore an arbitrary amount of LastChange evented variable callbacks may occur during the playback on device. It might be reasonable to develop a translation cache in Service layer, which will remember any media item being transported to device for the first time, and then reuse the already known UPnP object id (and subordinated metadata). The caching capability of existing server code needs to be investigated.

Nontrivial stateful logic will have to be developed in Service to provide setting “next item” even if the physical device does not support it.