Use of 3rd Party UPnP implementation in HN

The purpose of the document is to explore the tasks required, and issues involved in incorporating 3rd party UPnP implementations. One of the first questions to ask is what parts of the implementation are going to be replaced. The OCAP RI implements several aspects of UPnP in order to be a DLNA compliant DMS and DMP. On top of that, it implements the HN-EXT API in terms of this underlying implementation. Here is an enumeration of several of the functional areas implemented, please refer to the HNP specification for details.

1. UPnP Control Point discovery, action invocation, subscription requests and notifications
2. UPnP Device advertisement as well as handling state variable and subscription requests
3. UPnP Device listening for and dispatching of action requests
4. UPnP Connection Management Service implementation (CMS)
5. UPnP Content Directory Service implementation (CDS)
6. UPnP Scheduled Recording Service implementation (SRS)
7. HTTP requests for icons
8. HTTP requests for recorded content
9. HTTP requests for live content

The 3rd party implementation may account for some or all of the above areas of functionality.

RI UPnP design

The RI implements most UPnP related activities in the OCAP Java stack. The exception to this is servicing HTTP requests for recorded and live content. In the case of content delivery the RI listens and takes the request in from the Java stack, formulates and sends the HTTP headers, then passes the socket through to the MPE porting layer to a native C implementation that is responsible for sending the content that was requested.

All advertisement and listening to incoming requests is done using an open source 3rd party library called Cybergarage. The version of Cybergarage we use is 1.6, but it has been heavily modified in order to satisfy requirements unique to OCAP as well as to be compliant and pass UPnP & DLNA certification tests. In OCAP 1.2, the UPnP Diagnostics API was introduced and includes unique requirements that call for the ability to view and modify all incoming and outgoing message traffic, including SSDP and SOAP requests and responses. This low level API is not traditionally found in 3rd party libraries and can represent a substantial amount of work by integrators in order to satisfy the specification.

Approaches

There are several approaches that can be taken. Each has a different level of involvement and risk.

1. Implement that UPnP Diagnostics API from scratch, in pure Java. The approach is rather involved, but will allow for a complete and proper implementation.
2. Replace the Cybergarage stack with another Java API. This approach avoids complications with JNI implementations, however the OCAP specifications for message manipulation may be prohibitive or require special access to code or functionality. The port will become more difficult the less similar the API is to Cybergarage. Also the 3rd party UPnP library would need to be able to run under the more restrictive JavaME 1.4 Personal Basis Profile environment.
3. Replace the Cybergarage stack with a C library. This approach will require JNI calls at appropriate times. The most logical area to target are UPnP Diagnostic API calls. Aside from the message handling issues as outlined above, there would be a need to handle dispatching of action requests back into the Java layer, and the responses written back to the C library. This could involve significant JNI traffic and marshaling of data back and forth.
4. Replace the Cybergarage stack and the implementation of the CMS, CDS and SRS. This approach abandons the RIs implementation of these services, and with it requires an across the board translation of the OCAP HN APIs to C library calls via JNI. Along with the challenges above, this approach would require significant work to translate HN-EXT calls into C native functions that manipulate the CMS, CDS and SRS as described in the specification. All of the work to port the UPnP Diagnostics API would still need to be done, and would need to provide access to your service implementations.

Classes of interest

MediaServer.java

This is the class that initializes the UPnP device and services provided by the RI. This could be left alone if using the UPnP Diagnostics API to initialize the device and services.

UPnPDeviceImpl.java

The implementation here utilized the Cybergarage Device class. This could be replaced with JNI calls or use of another Java API that provides similar semantics.

UPnPManagedDeviceImpl.java

This class holds an in memory object graph of sub-devices, services, icons and state variables as constructed using the UPnP Diagnostics API. Since Cybergarage like many 3rd party APIs does not allow for online manipulation of it's data structures, this class will remove and reload the Cybergarage Device class every time there is a change to the object graph representing this device and it's components. After reloading this class walks the re-created Device in Cybergarage and reassociates the Cybergarage constructs with the UPnP Diagnostics equivelents, Cybergarage Service to UPnPManagedService and so on.

UPnPIncomingMessageHandler.java / UPnPOutgoingMessageHandler.java

These interfaces are perhaps the most difficult to implement with a 3rd party library. We had to heavily modify the cybergarage code in order to implement the APIs that allow this feature.

MetadataNodeImpl.java

Data structure that contains all of the properties for a container or item. In the RI implementation this class is used in both client and server functionality. In order to facilitate both functions, MetadataNodeImpl can be constructed from and can produce an XML DIDL-Lite fragment. If a reimplementation includes CDS functionality, there needs to be a mapping created for MetadataNode. There are three main areas of difficulty to be aware of.

1. Support for nested levels of nodes
2. Multi-valued metadata support
3. Proper namespace management

General porting notes

- Some 3rd party libraries may assume the use of files for device description and or SCPDs. In the case of the RI and cybergarage, all of this is set dynamically without the use of files.
- Need to make sure that every time a reload is done the device is properly shutdown and sockets released. When a device is initially brought up with the stack, every modification could potentially cause the device to go off the network and back on as services are added.