Designing a UPnP* AV MediaServer

Research & Development at Intel

(Version 1.00, 7-31-2003)

Designing a UPnP AV MediaServer

July 2003

List of Figures

List of Tables

Designing a UPnP AV MediaServer

July 2003

1 Introduction

This document offers a set of design guidelines, shares a number of Intel key learnings, and addresses many important issues related to UPnP AV MediaServer design and implementation. While it is hoped that it will reduce designer/implementer frustrations, and even time to market, its higher goal is to improve the interoperation of MediaServer devices and control points. Achieving this will not only make devices and control points work better together, but most importantly, it will improve the user's experience with UPnP AV devices.

After reading this document, implementers should:

- Understand the basic purpose of DIDL-Lite and its inherent flexibility
- Know the feature sets that MediaServers can exhibit
- Better understand some of the behaviors exhibited by MediaServers
- Be aware of common MediaServer implementation problems, and techniques to avoid and resolve such issues
- Know key questions to consider during the design phase of a MediaServer

1.1 MediaServer or MediaRenderer?

Whether a device is a MediaServer or a MediaRenderer (or both) is largely determined by the intended roles of the device. Initial research by Intel indicates that a MediaServer often only performs half of the functions of traditional consumer electronic (CE) Audio/Video (AV) devices. In practice, traditional AV devices are frequently combination devices with both a MediaServer and a MediaRenderer in the overall device hierarchy. For more information about building such a device, please refer to Intel's *Designing a UPnP AV MediaRenderer* document.

1.2 Basic Function

The basic function of a UPnP AV MediaServer is to implement the **ContentDirectory** and **ConnectionManager** services as defined by the UPnP AV Working Committee. The **ConnectionManager** service helps control points determine what type of content is on a MediaServer. The service may also provide the actions needed to setup a server-side controlled stream. The **ContentDirectory** service helps control points find content. The service may also allow control points to manage the advertised metadata hierarchy. In addition, a UPnP AV MediaServer may provide server-side transport controls for playback of the content stream.

1.3 Primary Service

The **ContentDirectory** service provides the core ability of a MediaServer—advertising, and possibly managing, content metadata. The **ContentDirectory** service can provide many optional actions. When selected, these actions require that decisions be made early in the design process. These decisions can have a dramatic impact on the implementation of a MediaServer, more so than on other UPnP devices.

1.4 Features

There are three broad categories of features for a MediaServer:

- Content discovery
- Content management
- Content distribution

The first two categories directly impact the advertised actions, while the third affects the complexity of the first two categories. From a UPnP AV perspective, a MediaServer is always a metadata server first, and optionally a content server. Discovery and management of content is always in the context of discovery and management of content metadata. Therefore, when discussing CDS hierarchies (a.k.a., CDS metadata hierarchy or content hierarchy), it is essential to remember that the control points must always view the exposed hierarchy as a logical representation of a metadata library and nothing more. Where and how the metadata, or the content binaries, are stored is not standardized in UPnP AV.

2 DIDL-Lite Basics

Before beginning the design of a MediaServer, the device architect must have a basic grasp of DIDL-Lite because the way DIDL-Lite is structured has a big impact on what it can and cannot do. Therefore, before choosing the MediaServer's features, the architect must understand and consider these factors.

The basic intent behind DIDL-Lite is to represent metadata for content. Metadata can be both brief and verbose, and good metadata can come packaged in either form. Good metadata exhibits the following characteristics.

1. Conforms to the DIDL-Lite syntax/schema
2. Avoids use of custom metadata fields
3. Describes the content
4. Describes where the content can be acquired

The first characteristic is very straightforward—XML fragments that look like DIDL-Lite may not be DIDL-Lite. DIDL-Lite is always qualified via its schema, even if practical implementations cannot perform run-time validation on their output[1].

The second characteristic limits the use of vendor specific metadata. Although some information falls outside the scope of DIDL-Lite, a lot of relevant metadata can be represented through DIDL-Lite. Implementers should use custom metadata (in *desc* element nodes) to augment the existing capabilities of DIDL-Lite. Implementers that heavily rely on custom metadata make it very difficult for others to interoperate with the MediaServer.

The third characteristic is subjective, but the intent is straightforward. When a user reads metadata values on a generic AV control point application, the user should be able to determine what the content is. This requires that implementations rely on brief and human friendly descriptions in the <dc:title> and <dc:creator> metadata fields. Implementers should avoid a convention that employs a Globally Unique Identifier (GUID) value for a title, while requiring a control point to parse through a custom block of metadata to find the real title.

The last characteristic is simple in its implications. Content should rely on resource (a.k.a., <res>) elements to describe where the content can be found. Proprietary techniques, such as those that would employ CDS **objects** with no <res> elements, while using a specially formatted title or a piece of custom metadata to instruct a proprietary control point, must be avoided.

The scope of the metadata associated with DIDL-Lite is very broad and defines at least 50 attributes and elements. This section does not go through most of these, but it does explain the basic attributes and elements

[1] Implementers have observed that the execution time for run-time validation of generated DIDL-Lite responses does not make schema validation a feature that is feasible for most MediaServer implementations. As such, developers need to ensure that the serialized DIDL-Lite properly matches the syntax allowed by the specification.

that implementers should employ in their MediaServer. The rest of this section describes and dissects the basic components of a CDS (**ContentDirectory** Service) metadata hierarchy (a.k.a., a CDS hierarchy).

2.1 Sample CDS Hierarchy

This section provides a sample CDS hierarchy that demonstrates some basic capabilities of DIDL-Lite. The CDS hierarchy has a root container (**@id**=0) with three child containers: *All Image Items* **(@id**=1), *Aerial Photography* **(@id**=18), and *Sample Playlist* (**@id**=28). The *Aerial Photography* container has a child item photograph with title, *A CityScape* (**@id**=19). The *All Image Items* container has a child reference item that points to the *A CityScape* photograph. The *Sample Playlist* has two audio items: *The Metro* (**@id**=100) and *In a Big Country* (**@id**=101).

```
<DIDL-Lite xmlns="urn:schemas-upnp-org:metadata-1-0/DIDL-Lite"
xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:upnp="urn:schemas-upnp-org:metadata-1-0/upnp">[2]
  <container id="0" searchable="1" parentID="-1" restricted="1" childCount="7">
    <dc:title>Root</dc:title>
    <upnp:class>object.container</upnp:class>
    <upnp:writeStatus>UNKNOWN</upnp:writeStatus>
            <container id="1" searchable="0" parentID="0" restricted="1" childCount="1">
              <dc:title>All Image Items</dc:title>
              <upnp:class>object.container</upnp:class>
              <upnp:writeStatus>UNKNOWN</upnp:writeStatus>
                    <item id="23" refID="19" parentID="1" restricted="1">
                      <dc:title>A Cityscape</dc:title>
                      <upnp:class>object.item.imageItem.photo</upnp:class>
                      <upnp:storageMedium>UNKNOWN</upnp:storageMedium>
                      <dc:date>2002-03-20</dc:date>
                      <dc:creator>Aerial Photography</dc:creator>
                      <upnp:writeStatus>UNKNOWN</upnp:writeStatus>
                      <res protocolInfo="http-get:*:image/jpeg:*" colorDepth="24"
                  resolution="1012x0768" size="217276"
                  importUri="http://172.16.0.41:62052/MediaServerContent_0/7/19/">http://172.16.0.41:6
                  2052/MediaServerContent_0/7/19/Aerial%20Photography%20-
                  %20A%20Cityscape.jpg</res>
       <res protocolInfo="http-get:*:image/jpeg:*" colorDepth="24" resolution="0079x0060"
     size="217276"
     importUri="http://172.16.0.41:62052/MediaServerContent_0/8/19/">http://172.16.0.41:62
     052/MediaServerContent_0/8/19/Aerial%20Photography%20-
     %20A%20Cityscape.jpg</res>
                    </item>
            </container>
            <container id="18" searchable="1" parentID="0" restricted="0" childCount="1">
              <dc:title>Aerial Photography</dc:title>
              <upnp:class>object.container.storageFolder</upnp:class>
              <upnp:storageUsed>2004484</upnp:storageUsed>
```

[2] Future DIDL-Lite fragments should assume the presence of the appropriate namespaces.

```
        <upnp:writeStatus>UNKNOWN</upnp:writeStatus>
            <item id="19" parentID="18" restricted="1">
              <dc:title>A Cityscape</dc:title>
              <upnp:class>object.item.imageItem.photo</upnp:class>
              <upnp:storageMedium>UNKNOWN</upnp:storageMedium>
              <dc:date>2002-03-20</dc:date>
              <dc:creator>Aerial Photography</dc:creator>
              <upnp:writeStatus>UNKNOWN</upnp:writeStatus>
              <res protocolInfo="http-get:*:image/jpeg:*" colorDepth="24"
            resolution="1012x0768" size="217276"
            importUri="http://172.16.0.41:62052/MediaServerContent_0/7/19/">http://172.16.0.41:6
            2052/MediaServerContent_0/7/19/Aerial%20Photography%20-
            %20A%20Cityscape.jpg</res>
              <res protocolInfo="http-get:*:image/jpeg:*" colorDepth="24"
            resolution="0079x0060" size="217276"
            importUri="http://172.16.0.41:62052/MediaServerContent_0/8/19/">http://172.16.0.41:6
            2052/MediaServerContent_0/8/19/Aerial%20Photography%20-
            %20A%20Cityscape.jpg</res>
             </item>
      </container>
      <container id="28" searchable="0" parentID="0" restricted="1" childCount="2">
        <dc:title>Sample Playlist</dc:title>
        <upnp:class>object.container.playlistContainer</upnp:class>
        <upnp:storageMedium>UNKNOWN</upnp:storageMedium>
        <upnp:writeStatus>UNKNOWN</upnp:writeStatus>
        <res protocolInfo="http-
    get:*:audio/mpegurl:*">http://172.16.0.41:62052/MediaServerContent_0/924/28/%20-
    %20SamplePlaylist.m3u</res>
             <item id="100" parentID="28" restricted="1">
              <dc:title>The Metro</dc:title>
              <upnp:class>object.item.audioItem</upnp:class>
              <dc:creator>Berlin</dc:creator>
              <upnp:writeStatus>NOT_WRITABLE</upnp:writeStatus>
              <res protocolInfo="http-get:*:audio/mpeg:*" size="4019086"
            importUri="http://172.16.0.41:62052/MediaServerContent_0/86/100/">http://172.16.0.41
            :62052/MediaServerContent_0/86/100/Berlin%20-%20The%20Metro.mp3</res>
             </item>
             <item id="101" parentID="28" restricted="1">
              <dc:title>In a big country</dc:title>
              <upnp:class>object.item.audioItem.musicTrack</upnp:class>
              <upnp:storageMedium>UNKNOWN</upnp:storageMedium>
              <dc:date>2002-03-20</dc:date>
              <dc:creator>Big Country</dc:creator>
              <upnp:writeStatus>NOT_WRITABLE</upnp:writeStatus>
              <res protocolInfo="http-get:*:audio/mpeg:*" size="3754112"
            importUri="http://172.16.0.41:62052/MediaServerContent_0/87/101/">http://172.16.0.41
```

```
                    :62052/MediaServerContent_0/87/101/Big%20Country%20-
                    %20In%20a%20big%20country.mp3</res>
                      </item>
                </container>
          </container>
        </DIDL-Lite>
```

This CDS hierarchy is only a logical representation of the entire metadata hierarchy; the back-end information system may not even use an XML database. Given the behavior of the **ContentDirectory**'s **CDS:Browse()** and **CDS:Search()** actions, the CDS implementation's responses will never return the entire hierarchy, as the methods only return flat listings of media objects.

2.2 Media Objects

A media **object** is an abstract concept used to refer to either a **container** or an **item** object. Intuitively, **container** and **item** elements represent media **objects** by the same name.

The purpose of a **container** is to be a logical parent to other media **objects**. Child **objects** can be other **containers** or media **items**. A **container** can also represent forms of content that involve multiple files, such as a playlist (@**id**=28).

The purpose of an item is to represent a single piece of consumable content. Individual songs, movie clips, and images are examples of media items (@**id**=19, @**id**=23, @**id**=100, @**id**=101).

Certain items have the unique ability to refer to other items and are known as **reference items**. Analogies for **reference items** include file shortcuts or symbolic links. A CDS uses a **reference item** to indicate that an **item** in a particular **container** is the same content as another **item** (often referred to as the **referenced item**). In the CDS hierarchy above, the **reference item** is the object with @**id**=23 and the **referenced item** is the object with @**id**=19. As a basic rule, in its responses to a **CDS:Browse()** or **CDS:Search()** action, a reference item must exhibit the same set as, or a superset of, the **referenced item's** metadata.

2.3 Media Classes

Although the <container> and <item> XML element names indicate the basic type of media **object**, the XML element name only provides a convenient way to specify rules for the DIDL-Lite schema. The specification authors wisely provided the concept of a *media class*. The *class* of a media object is identified through the <upnp:class> element; each media object can only have one media class.

Examining the CDS hierarchy reveals a few of the standardized media classes, including basic containers (**object.container**, @**id**=0, @**id**=1), photographs (**object.item.imageItem.photo**, @**id**=19, @**id**=23), a storage folder (**object.container.storageFolder**, @**id**=18), a playlist (**object.container.playlistContainer**, @**id**=28), an audio track (**object.item.audioItem**, @**id**=100), and a music track (**object.item.audioItem.musicTrack**, @**id**=101).

The complete list of standardized media classes can be found in the *ContentDirectory Specification 1.0*, but a brief analysis of the values shows an extremely extensible pattern. Every **item** or **container** object has a <upnp:class> value that begins with **object.item** or **object.container**, respectively. From there, additional modifiers can be appended to further describe the item. A modifier implies support for additional metadata fields, although in most cases the fields remain optional.

In addition to standard types, the framework for media classes allows implementers to define their own types in an intuitive manner. Additional information on vendor-extended classes can be found in section 7, *Appendix C: AV Working Committee Class Definitions* of the *ContentDirectory Specification 1.0*.

As a rule, CDS implementations should aspire to be as specific as possible when assigning the <upnp:class> value for a media **object**. In the CDS hierarchy above, two objects exist with media classes of **object.item.audioItem.musicTrack** and **object.item.audioItem** in the same parent playlist container. In this case the CDS implementation could not conclusively determine that the audio item was actually a music track. However, the CDS implementation was able to determine that the file was an audio item of some sort, so it assigned the **object.item.audioItem** class instead of using the **object.item** class. The latter would have been a legal assignment, but it would be less useful in the end.

2.4 Title and Creator

The <dc:title> and <dc:creator> properties are generally useful, and both can apply to any type of content. Of these, only <dc:title> is required, but <dc:creator>[3] information is a useful supplement. A media object can only have a maximum of one <dc:title> and one <dc:creator>. The values of these fields should be intuitive and useful. Implementers should not attempt to obfuscate metadata by encrypting the values so that only a proprietary control point can view the metadata.

CDS implementations should employ <dc:title> information that is concise and useful. If the MediaServer is lightweight and mirrors a file system, the <dc:title> value may simply be a filename. A more advanced MediaServer may actually describe the title of the content (such as a movie or song title). The <dc:creator> information should also be intuitive, although some MediaServer implementations may lack the capabilities for providing such metadata.

2.5 ObjectID and Parent ID Attributes

Every media object has an *objectID*, represented through the **@id** attribute in the <container> or <item> element declaration. The objectID is not intended as a user-friendly field, rather it is a way for control points and the MediaServer to refer to individual media objects in an unambiguous way. That being said, a media **object's** *objectID*, which is merely a string, has to be unique relative to the rest of the CDS hierarchy. Implementations that use a numerical counter or derivatives of a local file name already exist. One could even use something like a GUID string value.

A mistake observed by many at UPnP AV plugfests is a CDS implementation that does not provide an **@parentID** attribute for the *root* **container**. Even though a *root* **container** has no parent, the DIDL-Lite schema requires that it be given an **@parentID** attribute. The *objectID* value of "-1" has been reserved to represent a *root* **container's** parent non-existent object.

2.6 Restricted Attribute and WriteStatus Element

The **@restricted** attribute indicates that the media **object's** *state* is not intended to be managed by a control point, even if the **ContentDirectory** exposes actions that target metadata management. For **item** objects, the object's state is considered changed if the metadata of the object has changed. For **container** objects, object *state* is considered changed if the metadata or list of immediate child objects changes.

[3] Creator information should represent the *party most responsible for authoring the content*. Generally this is the musical artist or band that composed the music, the director or publishing company of a movie, or the author of a book.

While the value of the **@restricted** attribute conveys permissions for **CDS:CreateObject()**, **CDS:DestroyObject()**, and **CDS:UpdateObject()**, the <upnp:writeStatus> element reflects the permissions of changing the underlying binary content of individual <res> elements through **CDS:ImportResource()** and **CDS:ExportResource()**.

Ultimately, the **@restricted** attribute's role is simply that of an advisory metadata for a control point. Ultimately, it will be left to the MediaServer device to enforce any rules implied through the **@restricted** attribute's value.

A more in-depth discussion on using the element and attribute can be found in section 3.2.4, Restricted Attribute Versus writeStatus Element.

2.7 Object Resources

A media **object** represents metadata about content, but an object **resource** is object metadata that explains where the content can be found. Transport protocol, network, mime-type, encoding format, bit rate, image resolution, and other information specific to the content's binary representation is stored with a resource (a.k.a, <res>) element. Both **containers** and **items** can own zero or more resources, allowing DIDL-Lite a broad range of ways to represent different forms of content.

2.7.1 ProtocolInfo

A required attribute of a resource element is the *protocolInfo* string. The *protocolInfo* specifies the protocol, network, mime-type, and miscellaneous information about the content. In the sample CDS hierarchy, all of the content is available through the HTTP-GET protocol. HTTP-GET also requires that the network and miscellaneous fields be marked with a * value.

2.7.2 Resource URI

A **resource** element's value is often referred to as the *resource URI*, to indicate where the content can actually be acquired. If a resource's content is not available, the *resource URI* may be an empty string.

In the sample CDS hierarchy, the URI must be XML-escaped and HTTP-escaped, and the XML documents must not have the typical *<?xml…>* tag (which also specifies the text encoding) before the *DIDL-Lite* element. Please see section 5.15, Internationalization for a discussion on URI escaping and text encoding.

2.7.3 ImportURI

One important attribute of a resource element is the **res@importURI** attribute of the <res> element. It provides control points with information on how the underlying content can be modified with new binary information. The CDS specification mandates the value of **res@importURI** to be a HTTP URL to be used with the **CDS:ExportResource()** or **CDS:ImportResource()** actions[4].

2.7.4 Multiple Resources

A very interesting (and powerful) capability of DIDL-Lite is its ability to use the same metadata to represent multiple binary files that essentially have the same content. For example, the sample CDS hierarchy has two **objects** (**@id**=19 and **@id**=23) that specify two **resources**. The first **resource** in each object specifies the

[4] This URL can be used with the *DestinationURI* argument of the **CDS:ExportResource()** or the **CDS:ImportResource()**.

normal resolution of the image, but the second **resource** specifies a thumbnail version of the same image. This idea can easily be extended to a media **object** of a particular audio item having additional **resources** for different transcodings, protocols, or networks.

For information on how to use multiple **resources** to represent HTTP-GET content available on multiple IP addresses, please see section 5.4, IP Address Rules for HTTP-GET Content and section 6.3, Multi-NIC or Single-NIC Systems.

3 Determine the Desired Feature Sets

Like all development processes, determining the product's requirements is always the first step and this is especially important when designing a MediaServer. Experience has shown that designing for a minimal feature set, with the intent to add more features later by building upon the earlier code, can translate into implementation problems.

As a general rule, the design should reflect an expansive vision of the product's features, and implementation should include the back-end database or information system to support this vision. This allows features to be exposed incrementally as application logic matures throughout the development process. The key areas that can become problematic if not addressed early in the development cycle are discussed in section 5, Rules to Follow and section 6, Key Design Decisions.

3.1 Content Discovery and Distribution

The most important function a **ContentDirectory** service (CDS) provides is the means for a control point to find content. The authors of the CDS specification have been very insistent that a CDS is not a content store—it is a metadata store. As such, the primary job of a CDS is to provide users and control points with descriptions of content and instructions on where it can be found.

3.1.1 Content Discovery

A minimal CDS must implement little more than the **CDS:Browse()** action. This action allows a control point to enumerate the metadata hierarchy that is advertised by the MediaServer. Every response to a **CDS:Browse()** request is a list of media objects.

Unfortunately, the UPnP specification makes no accommodations for ease-of-use in finding the content of interest to a user. Using **CDS:Browse()** to find content amounts to a brute-force enumeration through a MediaServer's metadata hierarchy. To address this problem, the CDS specification also defines the optional **CDS:Search()** action, which can dramatically speed up the process of finding content through the means of a query.

Although the ContentDirectory specification does not make any requirements regarding the hierarchy represented by a CDS, vendors should always aspire to provide some form of hierarchical organization. Most CDS implementations ought to make it easy for users to find content by means of a container-based hierarchy. For example, a CDS that exposes audio content may have **container objects** for each artist and album. CDS implementations should avoid the practice of *root container dumping*, a practice typified by the root **container** being parent to all **objects** in the CDS hierarchy.

3.1.2 Content Distribution

As stated previously, a CDS is generally regarded as a metadata store. This distinction is important because a CDS makes no claim or guarantee that content discovered through the CDS is content that is locally stored on the MediaServer. The following are examples of how a CDS metadata hierarchy may relate to the actual content.

- The metadata hierarchy mirrors the content of a portion of the local file system or a hierarchical database.
- The metadata hierarchy is an aggregation of other content hierarchies found on the UPnP network.
- The metadata hierarchy reflects the available content from a premium subscription service, where the content physically resides somewhere across the Internet.

In addition to not making claims about the locality, CDS does not dictate the formats and protocols used to encode and transfer the data. In theory this means that a CDS implementation can advertise content with any permutation of format and transport protocol. Fortunately, a number of UPnP Forum member companies are relying on proliferated formats and protocols, instead of proprietary solutions.

A complete discussion of how content is actually acquired by a UPnP AV MediaRenderer falls outside the scope of this document. That being said, a brief analysis of current implementations indicates that HTTP–GET is the most common transport protocol seen at UPnP plugfests. Companies have also implemented IEEE 1394-based transport solutions. Typically, media formats have centered on various MPEG formats for audio and video, and the still-image formats employed by off-the-shelf digital cameras.

3.2 Content Management

This CDS feature set allows a control point to make changes to the advertised metadata hierarchy. The extent to which the metadata hierarchy can be changed is greatly influenced by the implementation.

For example, a CDS that exposes a metadata hierarchy that mirrors the content on a local file system may acquire its content descriptions from the actual files on the storage device. Such a CDS may not allow a control point to affect the descriptions of advertised content, but it might allow a control point to add and delete media objects that result in changes on the local file system.

The most important question to ask when considering content management is whether or not the product really needs it. Given the lack of security on a UPnP network, a lot of MediaServer implementers have already expressed concerns with allowing control points to manage a MediaServer's content and metadata. Allowing a control point that has not been authenticated to delete all of a person's personal content from a PC's hard disk is more of a liability than a feature. Yet, a mobile MediaServer (in the form of a PDA or mobile MP3 player) may want to allow control points to do exactly that. As such, the scope of MediaServer products that will want to allow control points to manage content will be few. Even so, the few that do want to enable the usage model should still follow some design guidelines.

There are innumerable ways to implement a CDS—unfortunately, the subtleties of the resulting behavior are also innumerable and at times this can be overwhelming. In the following subsections, various content management actions are discussed along with the types of tasks they can accomplish, and some observations are made on how they should behave.

3.2.1 Creating Media Objects

Creating a new media **object** (a.k.a., a metadata entry) begins with the **CDS:CreateObject()** action. A CDS can have varying levels of agent intelligence for accepting, rejecting, or even modifying the specified metadata. Generally, a polite implementation is one that will adhere to the rules in this section. Examples assume **CDS:CreateObject()** has sent the following DIDL-Lite metadata.

```
<DIDL-Lite xmlns:dc=”http://purl.org/dc/elements/1.1/”
xmlns:upnp=”urn:schemas-upnp-org:metadata-1-0/upnp/” xmlns=”urn:schemas-upnp-org:metadata-1-
0/DIDL-Lite”>
        <item id="" restricted="0" parentID=”parentContainerId”>
                <dc:title>New Track</dc:title>
                <dc:creator>Some Artist</dc:creator>
                <upnp:class>object.item.audioItem.musicTrack</upnp:class>
                <res protocolInfo=”http-get:*:audio/mpeg:*”/>
        </item>
</DIDL-Lite>
```

Figure 1: Sample Create Object Input

3.2.1.1 Support Specific Metadata; Reject Unsupported Metadata

Ideally, a MediaServer that employs content management abilities should support the full set of DIDL-Lite metadata elements and attributes defined in the CDS specification. Of course, this is not always possible and practical limitations may prevent this from becoming a widespread practice. Nevertheless, it is still important to balance the needs of devices and control points, therefore completely arbitrary behavior should not be employed.

As a rule, MediaServers should allow control points to manage the following non-mandatory metadata properties[5]: <dc:creator>, <res>, and all <res> attributes that are applicable to the content. As a rule, control points should provide accurate <dc:title>, <dc:creator>, and <res> information when creating and updating **objects** that depend on resource URIs that are not local to the MediaServer. If a control point is creating or updating an **object** that will have the content locally stored on the MediaServer, then <dc:title> and <dc:creator> info should be provided by the control point and the MediaServer should populate the <res> attributes and value after acquiring the content.

Undoubtedly, MediaServer implementations will encounter a scenario where a control point attempts to use metadata that is not supported. In such a scenario, the suggested course of action is to fail the request with error code 712 and use the error reason as a means to provide the control point with error-recovery information[6]. See section 3.2.1.1.1, CDS:CreateObject() and Error Code 712 for specifics on error recovery.

Returning a response indicating success while ignoring or removing metadata properties (e.g., not actually including the metadata fields in the DIDL-Lite response or subsequent **CDS:Browse()**/**CDS:Search()** requests) can be extremely troublesome if the control point that created the object is expecting the presence of those fields.

Two non-recommended responses to a **CDS:CreateObject()** request are shown below, both using the sample metadata in Figure 1: Sample Create Object Input. These improper responses would have been generated by a CDS implementation that did not support the <dc:creator> field.

In Figure 2: Bad DIDL-Lite Output—Ignoring Metadata, the <dc:creator> element has been completely removed. In Figure 3: Bad DIDL-Lite Output—Complete Truncation of Metadata Value, the <dc:creator> tag

[5] These metadata properties are standardized in the CDS specification but are not mandatory for the DIDL-Lite schema.
[6] This technique is not described or mentioned in the CDS specification, but it is certainly a viable and compatible measure to employ.

has been kept, but all of the creator data has been stripped out of it. This latter practice, although it follows the wording of the suggested rule, is not a recommended practice.

It is important to note that the **res@importURI** values use IP addresses instead of a host name. For information on IP addresses and HTTP related content, see section 5.4, IP Address Rules for HTTP-GET Content.

```
<DIDL-Lite xmlns:dc=http://purl.org/dc/elements/1.1/ xmlns:upnp=”urn:schemas-upnp-org:metadata-1-
0/upnp/” xmlns=”urn:schemas-upnp-org:metadata-1-0/DIDL-Lite/”>
        <item id="12" parentID="10" restricted="0">
                <dc:title>New Track</dc:title>
                <res importUri=“ http:/172.16.0.43:4000/item?id=12”
                protocolInfo=”*:*:audio/mpeg:*”/>
                <upnp:class>object.item.audioItem.musicTrack</upnp:class>
        </item>
</DIDL-Lite>
```

Figure 2: Bad DIDL-Lite Output—Ignoring Metadata

```
<DIDL-Lite xmlns:dc=http://purl.org/dc/elements/1.1/ xmlns:upnp=”urn:schemas-upnp-org:metadata-1-
0/upnp/” xmlns=”urn:schemas-upnp-org:metadata-1-0/DIDL-Lite/”>
        <item id="12" parentID="10" restricted="0">
                <dc:title>New Track</dc:title>
                <dc:creator></dc:creator>
                <res importUri=“http:/172.16.0.43:4000/item?id=12”
                protocolInfo=”*:*:audio/mpeg:*”/>
                <upnp:class>object.item.audioItem.musicTrack</upnp:class>
        </item>
</DIDL-Lite>
```

Figure 3: Bad DIDL-Lite Output—Complete Truncation of Metadata Value

3.2.1.1.1 CDS:CreateObject() and Error Code 712

In order to allow a control point to differentiate between a DIDL-Lite syntax error and an unsupported metadata error, CDS implementations need to have a way to inform a control point of the type of error. Unfortunately, the CDS specification only specifies error code 712 without much instruction as to how it should be used to accommodate both error scenarios. Intel has two suggestions for enabling robust feedback to a control point.

The first technique is to provide two different error reasons when using error code 712.

- DIDL-Lite Syntax Error: An error message that indicates the error is syntax related, such as malformed XML, or the XML does not conform to the DIDL-Lite schema.
- [CSV of allowed tags]: A comma-separated value list of supported metadata tags that provides a control point an opportunity to retry the request with an appropriate set of metadata.

For example, if a CDS implementation does not support the <dc:creator> element and it receives the sample metadata in Figure 1, it could return error code 712 with an error reason of:

dc:title,upnp:class,res,res@protocolInfo,res@bitrate

If the CDS XML syntax error was encountered, a generic XML error message could take its place.

Another alternative is to split the usage for error-code 712 and use a proprietary code (code=899) to report the metadata properties of the [CSV of allowed tags]. This method provides control points with the ability to easily distinguish between the different errors. The drawback is it uses a proprietary error code value which may be used by other vendors. Using error-code 712 at least provides a normative error code to examine between vendors, although, in either case, control points that want to recover from the error need to have logic for retrying with the appropriate metadata set.

3.2.1.2 Modify Metadata Values Within Reason

Modifying the request's metadata field values is sometimes acceptable. For example, truncating a long <dc:title> value to a length supported by the CDS is generally considered acceptable. In another example, if a control point creates a **container** object resulting in the creation of a local file system folder, then the associated <upnp:class> value of the **object** could be changed from the control point's original **object.container** value to 'object.container.storageFolder' because the latter provides more accurate information.
An example of bad metadata value truncation is found in Figure 3: Bad DIDL-Lite Output—Complete Truncation of Metadata Value. As a general rule, MediaServer implementations that allow content-management should not be truncating short string values that are already less than 30 bytes[7]. For more information on string lengths, see section 5.9.1, Max String Lengths: 255 Bytes/1KB/1MB+ and section 5.9.2, Min String Lengths: 30 Bytes.

3.2.1.3 More Accurate Metadata is Good

Adding metadata fields is also acceptable[8]. For example, a CDS adds a date metadata field to indicate the time when the entry was created. In another example, a CDS acquires the actual binary content, analyzes the file for its characteristics, and updates its metadata to reflect the new information. Fields such as image resolution, movie or audio bit rate, and file size are all metadata fields that could be added to metadata entry without the control point's knowledge.

As an example, an **object** created with the sample metadata in Figure 1 could have the following form after the actual binary was acquired by the CDS.

[7] UTF-8 encoded characters can be 3-bytes for one character. Assuming short string value is about 10 characters, this amounts to 30 bytes. Admittedly, the minimum number of 10 is somewhat an arbitrary value (in case implementers want to enforce a size less than 255 bytes for a short string).
[8] Some argue that fields should never be added. Instead, a control point should always create a reference to the original metadata and then modify the reference item. A reasonable conclusion is that original metadata should never be touched. Unfortunately, this can result in *littering* of a container with duplicate media objects.

```
<DIDL-Lite xmlns:dc=http://purl.org/dc/elements/1.1/ xmlns:upnp=”urn:schemas-upnp-org:metadata-1-
0/upnp/” xmlns=”urn:schemas-upnp-org:metadata-1-0/DIDL-Lite/”>
        <item id="12" parentID="10" restricted="0">
                <dc:title>Run Out of Tunes</dc:title>
                <dc:creator>Some Garage Band</dc:creator>
                <res protocolInfo=”http-get:*:audio/mpeg:*”
                bitrate=”16384”>http://172.16.0.43:4000/12/Some%20Garage%20Band%20-
                %20Run%20Out%20of%20Tunes.mp3</res>
                <upnp:class>object.item.audioItem.musicTrack</upnp:class>
        </item>
</DIDL-Lite>
```

Figure 4: Adding/Correcting Metadata

Note the following changes from the original input metadata.

- <dc:title> and <dc:creator> information is more specific.
- **res@importURI** has been removed to indicate the CDS will not allow the content to be overwritten (after acquisition).
- *Resource URI* has a fully qualified, non-relative URI, with the URI already escaped, and ending with a friendly file name. See section 5.7, Device-Friendly Resource URI Paths, for more information.
- The *protocolInfo* attribute has changed its first field to include http-get.
- The **res@bitrate** attribute has been set to indicate 128 kbps. Note that the **res@bitrate** attribute value is in kilobytes per second. (128 * 1024 / 8 = 16384)

3.2.2 Reference Items

Another way to create an object is to use the **CDS:AddReference()** action. This action is a convenient way to have one **container** list an **item** in another container. (A **container** cannot refer to another **container**.) A good analogy for a **reference item** is that of a Windows* file system shortcut, or symbolic links in Linux.

A good use of **reference items** is demonstrated by the *.NET AV MediaServer* included with Intel® Tools for UPnP Technologies. The CDS has 4 containers that aggregate content advertised by the CDS: *All Music*, *All Movies*, *All Images*, and *All Playlists*. The descendent items in those folders are all **reference items** that point to other **items** found throughout the entire CDS hierarchy. This makes it easier for control points that rely on **CDS:Browse()** to find content of a particular type.
The presence of **reference items** in the CDS hierarchy does not necessarily mean that the CDS grants a control point the ability to add or destroy **reference items**.

3.2.3 Specifying the Actual Resource/Content

There are two primary ways to specify the actual resource/content. The first approach is to declare a complete **resource** metadata element (a.k.a., a <res> element) which includes a valid URI. This is most often applicable when MediaRenderers and other sinks will acquire the content directly from another (Internet) web server.

The second approach is to declare an empty <res> element and allow the CDS to populate the resource element’s values automatically. This method requires the control point to get the content to the MediaServer. Depending on the supported transports, the control point will either call **CDS:ImportResource()** (on a CDS that needs to acquire the content) or **CDS:ExportResource()** (on a CDS that has the content already).

Although UPnP AV does not generally specify how devices use out-of-band streaming protocols, UPnP AV does explicitly state the mechanism for asynchronous content transfers between MediaServers to be HTTP-GET and HTTP-POST[9]. To transfer the content asynchronously from one MediaServer to another using HTTP-POST, control points can invoke **CDS:ExportResource()** on the MediaServer that has the content. Similarly, control points can use **CDS:ImportResource()** to instruct a MediaServer to acquire content from a known HTTP URL using HTTP-GET. A sample **res@importURI** for this is shown below.

```
<res importUri="http:/172.16.0.43:4000/item?id=12" protocolInfo="*:*:audio:*"/>
```

Figure 5: Sample ImportURI

After the binary is acquired, the *protocolInfo* string is updated to match the information in the binary. For an example, see Figure 4: Adding/Correcting Metadata.

3.2.4 Restricted Attribute Versus writeStatus Element

The **ContentDirectory** specification provides two properties for conveying read-only behavior on a CDS object: **@restricted** and <upnp:writeStatus>. The former conveys permissions for calling **CDS:CreateObject()**, **CDS:DestroyObject()**, **CDS:UpdateObject(),** and **CDS:AddReference()**. The latter conveys permissions for **CDS:ImportResource()** and **CDS:ExportResource()**.

Just as each **object** only has a single **@restricted**, an **object** can only have one <upnp:writeStatus> element. This is somewhat problematic because **objects** can have multiple <res> elements, which means that the value of <upnp:writeStatus> is effective for all <res> elements. When the <upnp:writeStatus> element is not present, a control point should assume that the associated <res> elements cannot be modified[10].

```
<item id="12" parentID="10" restricted="1">
        <dc:title>Run Out of Tunes</dc:title>
        <dc:creator>Some Garage Band</dc:creator>
        <res importUri="http:/172.16.0.43:4000/item?id=12" protocolInfo="http-get:*:audio/mpeg:*"
        bitrate="16384">http://172.16.0.43:4000/12/Some%20Garage%20Band%20-
        %20Run%20Out%20of%20Tunes.mp3</res>
        <upnp:class>object.item.audioItem.musicTrack</upnp:class>
</item>
```

Figure 6: Item With Read-Only Metadata

In the sample above, the **item**'s metadata and resources are read-only, which prevents a control point from affecting the metadata or resource binaries through **CDS:DestroyObject()**, **CDS:UpdateObject()**, **CDS:DeleteResource()**, or **CDS:ImportResource()**, or by sending an HTTP-POST to the specified **res@importUri** value.

[9] There is some irony in this decision, given the mantra of UPnP AV not specifying the out-of-band protocols, but the motivation to standardize the asynchronous transfer model between MediaServers does have some merits. Arguably, some could say that the transfer protocol should have been left unspecified too.

[10] Admittedly, <upnp:writeStatus> is metadata with little or no impact on the permissions. A strict interpretation of the CDS specification (with the v1.01 clarifications) indicates that the **@restricted** conveys permissions for control points to affect state changes on an **object**.

If a CDS implementation exposes a CDS **object** that is not restricted but cannot fulfill a **CDS:DestroyObject()**, **CDS:CreateObject()**, or **CDS:UpdateObject()** request, then the MediaServer should return error code 720 to indicate the request could not be processed.

3.2.4.1 Restricted Containers and Creating Objects

MediaServer implementations that want to allow control points to create new CDS **objects** in an existing **container** must ensure that the existing **container** is not restricted. This is unfortunately a limitation in the CDS specification[11].

If a **container** object has an **@restricted** value of false, then MediaServer implementations can also use zero or more <upnp:createClass> elements to specify what types of **objects** can be created in the **container**. If a **container** does not specify any <upnp:createClass> elements, then control points must be able to create any type of **item** or **container** in the existing **container** object. Otherwise, control points must limit their object creation to those specified in the <upnp:createClass> elements.

MediaServer implementations should always specify the **@includeDerived** value for a <upnp:createClass> element. Control points will likely assume an **@includeDerived** value of true if a <upnp:createClass> does not specify a value.

The presence of <upnp:createClass> elements does not guarantee success for a **CDS:CreateObject()** or **CDS:AddReference()** request. A CDS implementation may have a number of other reasons for rejecting a request to create a new object beyond a determination of control point permissions.

```
<container id="12" parentID="10" restricted="0">
        <dc:title>Stuff</dc:title>
        <upnp:class>object.container</upnp:class>
        <upnp:createClass includeDerived="1">object.item.audioItem</upnp:createClass>
        <upnp:createClass includeDerived="0">object.container.playlistContainer</upnp:createClass>
</container>
```

Figure 7: Restricted Container Allows New Child Objects

Figure 7 demonstrates the DIDL-Lite metadata for a **container** that allows control points to create any **object.item.audioItem**-derived **object** or any **object.container.playlistContainer object**.

3.2.4.2 Read-only Resources

If a CDS **object** has multiple <res> elements, then each <res> element should also employ the **res@importURI** attribute to indicate if a resource binary can be updated through **CDS:ExportResource()** or **CDS:ImportResource()**, or deleted through **CDS:DestroyObject()**.

The **res@importURI** attribute is useful because each CDS **object** is entitled to only one <upnp:writeStatus> element. However the **res@importURI** attribute can have a 1:1 relation to individual <res> elements.

[11] There are many other ways the specifications can be interpreted, but the UPnP AV 1.01 clarifications to the specifications restrict implementations to a particular interpretation of the **@restricted** attribute. Implementers should keep in mind that they are always entitled to expose a non-restricted **object** and fail the request with an error 720. Ultimately, AV Charter 2 needs to define a finer granularity for **object** and <res> permissions.

Although the presence of a **res@importURI** attribute can convey write permissions for a **resource** (both its XML attributes and underlying content), its presence does not convey any permissions for modifying metadata of the actual object.

```
<res importUri="http:/172.16.0.43:4000/item?id=12" protocolInfo="*:*:audio/mpeg:*"/>
```

Figure 8: Indicates Permission to Delete and Overwrite Binary

```
<res importUri="" protocolInfo="*:*:audio/mpeg:*"/>
```

Figure 9: Indicates Permission to Delete Binary

```
<res importUri="http:/172.16.0.43:4000/item?id=12" protocolInfo="*:*:audio/mpeg:*"/>
```

Figure 10: Indicates No Permission to Delete or Overwrite

It should be noted that the CDS specification requires that object state cannot be changed by a control point. Therefore, a control point should not attempt to modify (in any way) the **resources** of a restricted CDS **object**[12].

3.2.5 Modifying Metadata Entries

After a CDS creates a media **object** (and completes any additional work related to acquiring locally stored content), a CDS may allow a control point to modify the metadata throughout the course of the **object's** lifetime. There may be any number of reasons for this, including:

- The CDS allows media **objects** to have multiple **resources**, thus allowing a control point to add new resources.
- The CDS allows control points to provide extended user-provided descriptions of content or allows the user to rate content after it has been consumed through custom metadata.

Whatever the reason, the CDS specification provides the **CDS:UpdateObject()** action for achieving this task. The semantics stated in the specification are intuitive, but there are a couple of things to keep in mind.

3.2.5.1 Properly Interpret CSV Arguments

The comma-separated value lists in the **CDS:UpdateObject()** arguments need to match each other. The first element in the *CurrentTagValue* argument/list corresponds to the first element in the *NewTagValue* argument/list. MediaServer implementations need to interpret the orderings of the list to properly interpret the control point's request.

Take note that empty strings are used as placeholders in *CurrentTagValue* and *NewTagValue* when declaring new XML elements and deleting existing XML elements. Empty strings are easily represented by a single comma; since the arguments are comma-separated lists, portions of the string may have multiple consecutive commas.

The examples below describe a metadata entry that can be modified, followed by a **CDS:UpdateObject()** request to change the title and artist info, add a date and description, and remove the creator. The example does

[12] This is largely a limitation with how the specification is interpreted. Implementers should keep in mind that it is acceptable to expose a non-restricted **object** and fail UPnP actions that attempt to modify metadata, children, or resources.

not indicate if the MediaServer returns a success because a MediaServer may have individual fields set to exhibit read-only behavior[13], or specified new fields may not be supported.

```
<item id="12" parentID="10" restricted="0">
        <dc:title>Run Out of Tunes</dc:title>
        <dc:creator>Some Garage Band</dc:creator>
        <upnp:artist role=”Lead Singer”>Some Guy</upnp:artist>
        <res importUri=“http:/172.16.0.43:4000/item?id=12” protocolInfo=”http-get:*:audio/mpeg:*”
        bitrate=”16384”>http://172.16.0.43:4000/12/Some%20Garage%20Band%20-
        %20Run%20Out%20of%20Tunes.mp3</res>
        <upnp:class>object.item.audioItem.musicTrack</upnp:class>
</item>
```

Figure 11: Media Item Indicating that One or More Fields may be Modifiable

```
UpdateObject
(
  “12”,

  “<dc:title>Run Out Of Tunes</dc:title>,<upnp:artist role=”Lead Singer”>Some
  Guy</upnp:artist>,,<dc:creator>Some Garage Band</dc:creator>,”,

  “<dc:title>New Title</dc:title>,<upnp:artist role=”Drummer”>Some
  Guy</upnp:artist>,<dc:description>New Description</dc:description>,,
  <dc:date>2002-01-01</dc:date>”
  )
```

Figure 12: Valid CDS:UpdateObject() Request to Modify Metadata

3.2.5.2 Enforce Completeness of XML Elements

Any XML element in the comma-separated value list needs to be a complete representation of the entire element in its original form—including attributes. The examples below describe **CDS:UpdateObject()** requests on a media item. They will fail because the CDS specification makes no mention of partial modifications to an XML element.

```
<item id="12" parentID="10" restricted="0">
        <dc:title>Run Out of Tunes</dc:title>
        <upnp:artist role=”Lead Singer”>Some Guy</upnp:artist>
        <res importUri=“http:/172.16.0.43:4000/item?id=12” protocolInfo=”http-get:*:audio/mpeg:*”
        bitrate=”16384”>http://172.16.0.43:4000/12/Some%20Garage%20Band%20-
        %20Run%20Out%20of%20Tunes.mp3</res>
        <upnp:class>object.item.audioItem.musicTrack</upnp:class>
</item>
```

Figure 13: Metadata and Resource Field Has Multiple Attributes

[13] For example, a MediaServer may not allow a CP to change the title and creator, but it might allow the CP to change the description field.

```
UpdateObject
(
  “12”,
  “<res protocolInfo=”http-get:*:audio/mpeg:*” bitrate=”16384”
  importUri=“http:/172.16.0.43:4000/item?id=12”>http://172.16.0.43:4000/12/Some%20Garage%20Ban
  d%20-%20Run%20Out%20of%20Tunes.mp3</res>,<upnp:artist>Some Guy</upnp:artist>”,
  “<res protocolInfo=”http-get:*:audio/mpeg:*” bitrate=”12345”
  importUri=“http:/172.16.0.43:4000/item?id=12”>http://172.16.0.43:4000/12/Some%20Garage%20Ban
  d%20-%20Run%20Out%20of%20Tunes.mp3</res>,<upnp:artist>Some Other Guy</upnp:artist>”
  )
```

Figure 14: Invalid CDS:UpdateObject() Request and Attribute Order, Incomplete Original XML

The **CDS:UpdateObject()** request in Figure 14 has the following errors with the *CurrentTagValue* argument.

- The **res@protocolInfo** and **res@bitrate** attributes are not in the same order as the original XML. If a MediaServer wants to be robust, it must choose to ignore the ordering difference.
- The <upnp:artist> element is missing the **upnp:artist@role** attribute. The MediaServer must fail the entire request (even if it does not care about the ordering error) because the XML element is not complete compared to the original metadata.

3.2.5.3 Changing Object-Level Attributes

If a MediaServer wants to allow a control point to change a media **object** attribute (such as **@restricted**), the MediaServer must accept a request that specifies the entire media **object** XML as the current tag value. The MediaServer should still expect the media **object** in its entirety. Figure 15 shows how such a request might appear; the highlighted section shows that the control point is changing the object’s **@restricted** value from *false* to *true*.

```
UpdateObject
(
  “12”,
  “<item id="12" parentID="10" restricted="0"><dc:title>Run Out of Tunes</dc:title><upnp:artist
  role=”Lead Singer”>Some Guy</upnp:artist><res importUri=“http:/172.16.0.43:4000/item?id=12”
  protocolInfo=”http-get:*:audio/mpeg:*”
  bitrate=”16384”>http://172.16.0.43:4000/12/Some%20Garage%20Band%20-
  %20Run%20Out%20of%20Tunes.mp3</res><upnp:class>object.item.audioItem.musicTrack</upnp:cla
  ss>
  </item>”,
  “<item id="12" parentID="10" restricted="1"><dc:title>Run Out of Tunes</dc:title><upnp:artist
  role=”Lead Singer”>Some Guy</upnp:artist><res importUri=“http:/172.16.0.43:4000/item?id=12”
  protocolInfo=”http-get:*:audio/mpeg:*”
  bitrate=”16384”>http://172.16.0.43:4000/12/Some%20Garage%20Band%20-
  %20Run%20Out%20of%20Tunes.mp3</res><upnp:class>object.item.audioItem.musicTrack</upnp:cla
  ss>
  </item>”
  )
```

Figure 15: CDS:UpdateObject() Changing MediaObject Attribute

3.2.5.4 CDS:UpdateObject() Follows CDS:CreateObject() Rules

A CDS may reject or modify a request to add, remove, or modify an existing metadata field of a media **object**. A polite CDS implementation will apply the same (or very similar) metadata rules used for the **CDS:CreateObject()** action to the **CDS:UpdateObject()** implementation. However, product requirements may dictate that only some metadata fields can be changed after object construction.

3.2.5.5 Restricted Tag Conveys Modify Permissions

A CDS implementation should convey permission to modify a particular media **object** through its **@restricted** attribute. For more information on the @restricted attribute, see section 3.2.4, Restricted Attribute Versus writeStatus Element.

3.2.5.6 CDS:UpdateObject() Not For Deleting Resources

A CDS should reject a request to delete a **resource** element through **CDS:UpdateObject()**. Likewise, an implementation may reject a request to modify a **resource**'s metadata through **CDS:UpdateObject()**. Convention states that a control point should use **CDS:DeleteResource()** to delete a resource. Figure 16 describes a **CDS:UpdateObject()** call that improperly attempts to remove a **resource** from the media **object** described in Figure 4: Adding/Correcting Metadata.

```
UpdateObject
        (
        “12”,
        “<res protocolInfo=”http-get:*:audio/mpeg:*”
        bitrate=”16384”>http://172.16.0.43:4000/12/Some%20Garage%20Band%20-
        %20Run%20Out%20of%20Tunes.mp3</res>”,
        “”
        )
```

Figure 16: Bad CDS:UpdateObject() Request—Removes Resource

3.2.5.7 CDS:UpdateObject() and Read-Only Tags

There are a number of metadata fields that should not be changed when a control point requests a change. These include requests to change attribute values like **@id**, **@parentID**, or **@refItem**.

A CDS can choose to allow or reject requests that change additional fields like resource URI values, **@importURI** values, **@protocolInfo**, and other fields the implementer decides are read-only fields.

CDS:UpdateObject() changes should only return a success if the entire set of proposed changes is accepted by the CDS. Implementations should return error code 705 to indicate that a **CDS:UpdateObject()** request could not be completed because the proposed changes affect read-only behavior. Additionally, the error reason associated with the error code should be a comma-separated value list of metadata fields that can be changed by a control point. For example, a CDS implementation that only allowed a control point to change the title and creator metadata could send “*dc:title,dc:creator*” as its error reason[14].

[14] The CDS specification does not specify or mandate this behavior, but such a convention should be adopted so as to provide control points with the opportunity to recover if a MediaServer does not support metadata provided in the **CDS:UpdateObject()** invocation. This technique allows control points with the additional logic to retry with a subset of the

3.2.6 Destroying Media Objects

As **CDS:CreateObject()** is to object creation, **CDS:DestroyObject()** is to object deletion, including the deletion of **reference items**. What follows in this section are both rules and suggestions for interpreting the CDS specification.

3.2.6.1 Restricted Attribute Conveys Destroy Permissions

A CDS implementation should convey permissions to destroy a particular media **object** through its **@restricted** attribute. The **@restricted** attribute describes the permissions of a single media object with regards to its metadata and child **objects**. The **@restricted** attribute does not infer permissions about the underlying **referenced** object.

Perhaps the most important behavior to consider when implementing **CDS:DestroyObject()** is destroying child **objects** of a restricted **container**. The CDS specification (with the v1.01 clarifications) states that the **@restricted** attribute of a **container** limits a control point's behavior—control points cannot change the state of a **container**, including adding child **objects** to or removing them from the **container**. Of course, enforcing this to the letter may be undesirable as it may require **container** hierarchies that a vendor does not wish to employ or it may encourage implementations to have either completely restricted or completely unrestricted metadata hierarchies.

In the end, it seems reasonable that a control point be able to invoke **CDS:DestroyObject()** on an unrestricted child **object** even if the parent **container** is restricted. Control points simply need to have the logic to gracefully handle the error case where the MediaServer denies the request to destroy the object, which is something control points need to do regardless of the value of the **@restricted** attribute.

For more information on recursive destroy/delete behavior, see section 3.2.6.2, Recursive Behavior for CDS:DestroyObject(). For more information on the @restricted attribute, see section 3.2.4, Restricted Attribute Versus writeStatus Element.

3.2.6.2 Recursive Behavior for CDS:DestroyObject()

The v1.01 clarifications to the CDS specification leave the capability for recursive destroy to the decisions of the implementer. This is extremely important, as it impacts both the MediaServer control points and devices.

The impact on control points is significant because a control point that seeks to be interoperable with every MediaServer (when using **CDS:DestroyObject()**) must destroy a branch of a CDS hierarchy by deleting **objects** in a depth-first-search manner. Essentially, before a **container** can be destroyed, all of its descendents must be destroyed. Although this framework is horribly inefficient (as it requires numerous SOAP invocations) the process does allow for interoperable and predictable behaviors and allows control points to adapt to errors that occur.

The impact on MediaServers is beneficial to the vendor because it allows a wide variety of implementations. MediaServers that rely on a local file system can mimic the behavior of their local file system. Likewise, MediaServers that employ ACID[15] rules can properly implement those rules for their metadata database. Regardless of the back-end information system, all MediaServer implementations that support control point

metadata without affecting the behavior of existing control points. For more information, see section 3.2.1.1.1, CDS:CreateObject() and Error Code 712.

[15] ACID rules are the basic rules employed by most database systems. The acronym stands for Atomic Consistent Isolated and Durable.

management of a CDS hierarchy should have no problem with destroying individual leaf **objects** in a call to **CDS:DestroyObject()**.

Even with these limitations though, vendors providing MediaServer control points and devices can still implement recursive destroy. The key thing to remember is that a vendor's control point must never assume that other vendors employ the same rules for recursive destroy. Therefore, it becomes imperative for control points to check the MediaServer's model and manufacturer information before attempting to destroy **containers** with child objects.

3.2.6.3 Deleting Resources

One area that is intentionally left ambiguous by the CDS specification is the manner in which a CDS handles locally stored content. The specification does not indicate whether deleting a media object results in the removal of a locally stored binary[16]. Despite the ambiguities, MediaServer implementers should consider the information imparted in this section.

3.2.6.3.1 CDS Spec Does Not Require Removal of Resources

Whether a CDS actually deletes underlying content files is dependent on the needs of the implementation. There are instances when a CDS should actually delete files from a local file system, and there are other times when a CDS should create the appearance that a binary file was deleted. Either behavior is permissible, although implementations should use the presence or absence of **CDS:DeleteResource()** in the SCPD file of the CDS to determine the policy.

3.2.6.3.2 No CDS:DeleteResource() Means CDS Handles Resource Removal

If a CDS has not implemented **CDS:DeleteResource()**, but has implemented **CDS:DestroyObject()**, it indicates to a control point that local binaries will be handled by the conventions of the CDS implementation. For most MediaServer implementations, it will be sufficient to provide **CDS:DestroyObject()** without **CDS:DeleteResource()**[17].

3.2.6.3.3 CDS:DeleteResource() Means Delete Binary

The suggested guideline for **CDS:DeleteResource()** is that MediaServers should implement this method if and only if the MediaServer specifically wants to bestow control points with explicit control over the storage lifetime of content owned by the MediaServer. An example where this model is appropriate would involve an **object** with multiple resources. If a control point discovers that one of the **object's** resources is corrupt, it may allow the user to delete the corrupt resource and leave the other resources intact.

Therefore, if a control point calls **CDS:DeleteResource()** before calling **CDS:DestroyObject()**, then the MediaServer should delete binaries from the MediaServer's local storage[18]. Likewise, if a control point calls **CDS:DestroyObject()**, without first calling **CDS:DeleteResource()**, then the MediaServer should employ its own policies for removing content binaries to the MediaServer.

[16] Keep in mind that CDS exposes metadata. It makes no claims about how and where the underlying binary content is stored.
[17] MediaServers can be implemented with a resource-deleting policy in mind, which reduces the need for **CDS:DeleteResource()**. Control point logic is certainly simpler if it defers decisions for delete content binaries to the MediaServer.
[18] Implementations may want to move the binaries to a staging area before actually deleting them, so as to allow the user to confirm the actual removal by using a local user interface at a convenient time.

4 Advanced MediaServer Features

The third set of features applicable to a MediaServer device is not so much a feature set affecting UPnP actions as much as it is a feature set that affects how a MediaServer can behave given its set of actions. Although the features discussed in this section are not discussed in the UPnP AV specifications, they can add value to a MediaServer without adding additional UPnP actions. Even though advanced MediaServer features fall outside the scope of UPnP AV, they are extremely relevant to many implementations.

4.1 Out-of-Band Content Management

Vendors are mistaken when they conclude that a CDS must provide content management actions if it will have metadata hierarchy that changes. A CDS implementation is allowed to change its CDS hierarchy whenever it wants. The only restriction is that metadata must always be presented using correct syntax.

Consider a premium content service that uses a MediaServer to advertise its content. The MediaServer's CDS would probably not allow the user to manage metadata in any way. Most likely, the CDS would periodically contact the premium-content master server over the Internet and acquire a manifest of content that the user can consume for a specified period of time. If appropriate, the CDS hierarchy would change according to the terms of a subscription. For example, content might disappear if consumed too many times. If the user purchases a higher-tier of service, additional content might appear.

4.2 Content Aggregation and Metadata Mirroring

MediaServers advertise content and control points discover content—the tasks seem to be mutually exclusive, but there is a usage scenario that involves both activities by a single UPnP entity: *content aggregation*. A MediaServer that has the content aggregation ability uses a control point to enumerate the content of other MediaServers and then uses its own MediaServer capabilities to advertise that same content again. At first, this feature might not seem too useful, but it is useful in providing other control points with a single point of access for finding content on the network. It is unlikely that the such a MediaServer will download the content onto other MediaServers, thereby restricting itself to the role of being a metadata mirror instead of a content mirror. (Metadata mirroring is a subset of aggregation. Content mirroring is an example of full aggregation, which involves downloading the content from a content-origination MediaServer and becoming a redistribution point.) Typically, aggregating/mirroring MediaServers will have the **CDS:Search()** action enabled to allow a control point to search through CDS hierarchies that are advertised through CDS implementations that do not implement **CDS:Search()**.

4.2.1 Avoid Mirroring of Server-Side Controlled Content

Although content aggregation works for client-side controlled streams (e.g., streams where the MediaRenderer owns the transport controls), metadata mirroring cannot be done reliably for content that is server-side controlled (the MediaServer owns the transport controls).

Consider a MediaServer with content intended to be transported across an IEEE 1394 interconnect with the transport protocol IEC-61883. If a different MediaServer aggregated the metadata and advertised it in its exact form, a control point would attempt to invoke **CM:PrepareForConnection()** on the MediaServer with the mirrored metadata. Such a move would fail the control point because the original MediaServer has the actual content and it is the one that expects to get the **CM:PrepareForConnection()** request.

Some have suggested that the aggregating MediaServer proxy the call to **CM:PrepareForConnection()** and use its own control point to invoke **CM:PrepareForConnection()** on the content-originating MediaServer.

Thereafter, the aggregating MediaServer needs to monitor the content-originating MediaServer's connections in order to properly report the state of the proxied connection. The idea is certainly possible, but may run into problems with certain out-of-band protocols that depend on information sent in **CM:PrepareForConnection()**. Vendors that pursue this usage model are encouraged to ensure that the out-of-band protocol mechanism can accept **CM:PrepareForConnection()** arguments in a proxied manner.

4.3 Content Bridging and Transcoding

Fundamentally, the content bridging feature allows network entities to consume content through a MediaServer when the original content binary may not be consumable. Such a definition has broad implications when looking at a digital ecosystem of multiple UPnP networks, transport layers and protocols, and media formats. Content bridging is an extremely useful behavior for a MediaServer that intends to advertise content on multiple network interfaces because network entities on one interface may not be able to access content sources on the other network[19]. Such a MediaServer provides the bridge that would allow seamless interaction of clients and servers on different networks.

Before continuing, it should be noted that metadata mirroring behavior is fundamentally different than bridging and transcoding behavior. Metadata mirroring is achieved using a simple algorithm to exactly mirror metadata while preserving the original resource-URI address. In contrast, bridging and transcoding do not provide an exact mirror of the metadata because the resource URI only gives the appearance that the bridging/mirroring MediaServer actually hosts the content.

A bridging MediaServer with multiple network interfaces and content aggregation abilities can modify its CDS hierarchy so that the resource paths point to another MediaServer. By advertising resources that point to the bridging MediaServer, a device gives the impression that a local MediaServer hosts the content, while, in fact, it is actually stored on a non-local MediaServer. When network entities attempt to access the data on the bridging MediaServer, the non-local MediaServer can stream the content from the other network, and retransmit the stream to a network entity that would normally not be able to access the non-routable content.

Another form of content bridging is transcoding—a feature that allows network entities to request content in a media format different from the original source. For example, consider the presence of two MediaServers with one of them advertising support for MPEG2 streams. The second MediaServer has transcoding capability and could choose to advertise that same content on the first MediaServer, except that the resource would indicate MPEG4 content. When the transcoding MediaServer gets a request for the MPEG4 content, it would request the original MPEG2 stream from the original source, and transcode the content in real-time to the MPEG4 requester. A transcoding MediaServer could also implement some caching algorithms to optimize future requests for transcoded content.

While bridging requests and transcoding are useful, the idea of bridging across different transport layers is a very interesting feature. It would allow for scenarios like IEEE 1394 endpoints requesting content through a MediaServer that obtains the actual content through an HTTP-GET request.

4.4 Content Snippets

Providing snippets of content has similarities to providing transcoding of content. Basically, a MediaServer provides a short snippet of content at the request of the user to allow the user to quickly determine if the

[19] A simple example is a MediaServer with two network cards, each with IP addresses that belong to two different IP networks. If IP endpoints on one network cannot reach the other, the MediaServer is in the perfect position to bridge content.

content is wanted. This is extremely useful given the size of good quality digital images, audio, and movies. The following are examples of snippets a MediaServer could be asked to provide.

- Thumbnail images of photographs or other bitmap images.
- The first thirty seconds of an audio track or movie clip.
- Highlights extracted from an Internet sports broadcast.

The recommended way to provide content snippet capability is by adding additional resources to a media object. The first resource in the media object should continue to be the full-content resource. If the media object already has multiple resources (possibly for additional transcoded formats or protocols), the snippet resource should follow the resource that is most similar to the encoding and network protocol[20].

4.5 Autonomous User Agents

The concept of autonomous user agents is not new; in fact, the *content aggregation* feature could even qualify as a user agent. Other examples involve intelligent management of other CDS entities on the network. Agents could be configured to:

- Back up metadata (and possibly content) found on other MediaServers
- Actively monitor usage
- Push content to other CDS entities

A possible scenario that employs both usage monitoring and pushing of content involves a lightweight MediaServer embedded in a UPnP enabled car stereo. In this implementation an agent is initially given a general knowledge of the user's preferences and daily monitors content usage. Anytime this feature is enabled and the car is present on the WLAN, the agent refines and augments the music directory to more closely match the user's tastes. To enhance an otherwise boring daily commute, the agent compiles and supplies a fresh, mini-directory of songs sure to please the listener.

5 Rules to Follow

Innovation is always of value, but there are a number of key design principles to adhere to when building a MediaServer. Many of these suggestions appear obvious, but observations at UPnP AV plugfest events have shown that it is easy for competent developers to make serious mistakes in their CDS implementation.

The rules in this section are intended to supplement the rules found throughout section 3.2, Content Management.

5.1 DIDL-Lite Writing Rules

The most common problem for UPnP implementers has been XML errors—the disease of malformed XML has infected many UPnP devices, and MediaServers are no exception. In addition to XML errors in device descriptions, service descriptions, GENA events, and SOAP messages, a MediaServer has the DIDL-Lite XML to contend with.

Malformed or invalid DIDL-Lite is unforgiving. Observed symptoms include crashing CDS control points and content that never appears in CDS control point applications. To make matters worse, a MediaServer is not adversely affected

[20] In theory, XML is always unordered. In practice, many implementations assume some ordering. Realistically, a control point that is prepared to handle multiple resources is going to need to examine all resources in order to find the appropriate full-length or snippet resource.

by the plight of control points. The solution for this common MediaServer problem resides in the MediaServer's logic for serializing XML. The applicable design principle is simple—limit the use of hard-coded XML.

In memory-constrained implementations, this can be done by combining the C language's *sprintf* method with *#define* preprocessor (that defines the format for DIDL-Lite XML strings). A good implementation will define all of these strings in one location, which is invaluable at a plugfest if the XML needs debugging.

Implementations that have more memory, and can afford an object-oriented design, should leverage classes that are designed to serialize DIDL-Lite, and (possibly) classes that are designed to serialize well-formed XML. Implementations that use an object-oriented database built from the ground up can make those classes responsible for serializing their own XML. While these suggestions seem obvious to most designers, observations at plugfests indicate that people are straying from basic design rules.

Remembering and applying the rules below will reduce malformed XML and make the difference between schema-compliant DIDL-Lite and invalid metadata.

5.1.1 Properly Escape XML

It should not be forgotten that DIDL-Lite is packaged as the body of a SOAP response and that the DIDL-Lite portion of the SOAP payload needs to be properly escaped. Also, data values in the DIDL-Lite need to be doubly-escaped; once for DIDL-Lite itself, twice for the SOAP response.

The following text block is an example of a properly escaped response to a **CDS:Browse()** request. Observe the double-escaping of the container's title (highlighted below), which would actually read: *Bill & Bob's Songs*.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Envelope s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <u:BrowseResponse xmlns:u="urn:schemas-upnp-org:service:ContentDirectory:1">
      <Result>&lt;DIDL-Lite xmlns="urn:schemas-upnp-org:metadata-1-0/DIDL-Lite"
xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:upnp="urn:schemas-upnp-org:metadata-1-0/upnp"&gt;
 &lt;container id="6" searchable="0" parentID="5" restricted="1" childCount="0"&gt;
   &lt;dc:title&gt;Bill &amp;amp; Bob's Songs&lt;/dc:title&gt;
   &lt;upnp:class&gt;object.container.playlistContainer&lt;/upnp:class&gt;
   &lt;upnp:storageMedium&gt;UNKNOWN&lt;/upnp:storageMedium&gt;
   &lt;upnp:writeStatus&gt;UNKNOWN&lt;/upnp:writeStatus&gt;
   &lt;res protocolInfo="http-
get:*:audio/mpegurl:*"&gt;http://134.134.19.123:57298/MediaServerContent_0/1/6/%20-
%20Bill%20%26%20Bob%27s%20Songs.m3u&lt;/res&gt;
 &lt;/container&gt;
&lt;/DIDL-Lite&gt;</Result>
      <NumberReturned>1</NumberReturned>
      <TotalMatches>1</TotalMatches>
      <UpdateID>3</UpdateID>
    </u:BrowseResponse>
  </s:Body>
</s:Envelope>
```

Figure 17: XML Escaping Example With DIDL-Lite

5.1.2 Numerical Data Types are not String Data Types

Implementers unfamiliar with XML often conclude incorrectly that XML elements and attribute values can be an empty string to indicate an undefined value. The UPnP framework defining standard numerical data types does not allow an empty string to represent a valid numerical value.

5.1.3 Do not Forget ParentID and other Required Metadata

Since a root has no parent container, it is easy to think that the root container of a CDS hierarchy does not need to print the *parentID* attribute. This is wrong—root containers have a parent of –1. This mistake is surprisingly common, but is easily found and fixed. The CDS specification also defines other metadata that is required, including: a *dc:title* element and a *upnp:class* element. These elements are in addition to other attributes that belong to a *container* or *item* element.

5.1.4 Do not include the XML declaration and comments

The merits of allowing **CDS:Search()** and **CDS:Browse()** responses to declare their DIDL-Lite payloads with the typical "*<?xml ...>* element has been debated at UPnP plugfests. The majority agree that it is best not to include the declaration and to assume UTF-8 encoding (as per the XML specification). A similar question has been raised about allowing XML comments in the DIDL-Lite. To simplify DIDL-Lite parsing, CDS implementations should avoid responses with XML comments.

Developers should be aware that their SOAP-level XML needs to have the proper XML declaration. See section 5.1.1, Properly Escape XML, for an example with the XML declaration at the SOAP level, but missing from the DIDL-Lite level.

5.2 Implement Proper UpdateID Support

MediaServers, both lightweight and advanced, need to properly populate the *UpdateID* output argument in a **CDS:Browse()** response for a container. This rule applies to both forms of **CDS:Browse()** requests: *BrowseMetadata* and *BrowseDirectChildren*. This rule does not apply to Browse requests on an item[21].

A MediaServer that does not properly implement functionality for tracking container UpdateID values can cause problems for control points. Depending on the mistake, a control point logic may incorrectly conclude that nothing has changed or that something has changed.

5.3 Implement CDS.SystemUpdateID and CDS.ContainerUpdateIDs

CDS implementations need to implement the *CDS.SystemUpdateID* state variable. A CDS that never events a change in its CDS hierarchy will never inform control points about changes in its CDS hierarchy. A CDS implementation that requires a control point to periodically examine the CDS hierarchy is an invalid implementation.

Similarly a CDS should implement the *CDS.ContainerUpdateIDs* state variable. Although the specification does not require this state variable, practice has determined that it is extremely useful and helps minimize the number of Browse requests that are placed on the CDS.

[21] Unless the specification explicitly prohibits such behavior, a CDS implementation may even opt to have UpdateID values for items.

5.4 IP Address Rules for HTTP-GET Content

MediaServers that distribute content through HTTP-GET should avoid using the hostname of the MediaServer when providing the content's URL in CDS responses. Hostnames pose problems for UPnP networks, as DNS servers may not be available to resolve a hostname.

CDS implementations on host machines with multiple network interfaces unfortunately cannot reply with any of the IP addresses of the host machine. Also, they need to take active steps to ensure that contents served from the MediaServer are routable by consuming endpoints. Unfortunately, no technique exists to ensure that content found through a CDS request is routable from the source to the consuming endpoint, but there are two methods that minimize non-routable content symptoms for content that should be routable.

- If the UPnP stack permits acquiring the IP address/port where a CDS request was received and the device's HTTP server can accommodate virtual directories, a CDS implementation can use the IP address/port where the request was received as the base IP address for the content. This technique is relatively simple to implement, but it has shortcomings. These become evident when a control point and a MediaServer are on the same host machine and the CDS responses point to content accessible on one interface, but a target MediaRenderer can only access content (on the MediaServer) through another interface. Other than this, the technique is very reliable.
- For CDS implementations that cannot guarantee that a CDS control point will not reside on the same host machine (such as a desktop PC), the technique is slightly different. In this case, it is recommended that a CDS expose multiple resource elements for each piece of accessible content. For example, a MediaServer, with 3 IP addresses, hosting a media object with an MP3 resource would serialize the media object with 3 resource elements. As a courtesy to control points that are not on multiple network interfaces, the first resource element should use the IP address where the request was received. The other two resource elements are provided for CDS control points that are capable of determining if content is routable based on a comparison of the IP addresses for the content and the MediaRenderer. For additional information, please see section 6.3, Multi-NIC or Single-NIC Systems.

5.5 Build CDS DIDL-Lite Responses Dynamically

Do not serialize CDS responses straight from a DIDL-Lite document when that document represents the entire CDS hierarchy—it can be disastrous. In the long term it leaves the developer very little implementation flexibility. Also, it becomes cumbersome, if not impossible, to print CDS responses so that the network locations of resources dynamically adjust to match the network location of the MediaServer.

The ideal technique is to sufficiently decouple the back-end database/information-system from the DIDL-Lite XML used in CDS responses. Should the CDS implementation grow in complexity, sufficient abstraction of the information system from the XML writing modules will allow the CDS implementation to evolve gracefully compared to an implementation whose information system was directly tied to the application logic. For suggestions on decoupling the information system from the CDS logic, see section 6.2, Information System's Infrastructure.

5.6 ObjectID Lifetime

CDS implementations should avoid recycling objectID strings. For example, when an object is removed from a CDS hierarchy and a short time later a new object is added, the new object should not have the same objectID as the previously removed object.

Similarly, CDS implementations should avoid changing the objectID of a media object simply because its metadata changed. This type of behavior tends to hinder control points from *remembering* media objects, an ability necessary for functionality like a *favorite items* list.

5.7 Device-Friendly Resource URI Paths

When advertising URIs, a MediaServer should include some kind of user-friendly information towards the end of each URI[22]. This information can be the title, the title and creator, or just the name used by the local file system. Doing this will improve the usability of lightweight devices that employ URIs to deliver information to the user about content in playback. As an additional courtesy to devices, a URI path should end with the proper file extension. See section 5.14.3, Mime-Types and File Extension Mappings, for more information.

The total length of a URI should not exceed 1024 bytes and implementers are strongly encouraged to further restrict the length to no more than 255 bytes. A URI of 255 bytes should be long enough to do basic URI obfuscation in addition to providing a means to encode internationalized URIs. See section 5.15 for more information on Internationalization.

MediaServers should use an IP address instead of a hostname when advertising a URL (a.k.a., HTTP URI) because not all networks (especially home networks) will have a server to provide hostname resolution. Requiring name resolution functionality for MediaRenderer devices will also increase the memory footprint and code complexity for those devices.

For recommendations on serializing DIDL-Lite responses on a host machine with multiple network interfaces, read section 6.3, Multi-NIC or Single-NIC Systems.

5.8 Device Friendly Object IDs

As a general rule, a CDS should impose reasonable lengths on its object ID strings. The total length of an objectID should not exceed 1024 characters and implementers are further encouraged to restrict the length to 255 characters.

5.9 Additional Metadata Value Rules

Additional rules for metadata values also exist. Sometimes these rules help devices, but they more often help control points determine reasonable lengths for string data.

5.9.1 Max String Lengths: 255 Bytes/1KB/1MB+

Each field of common, string-based metadata intended for short information should be no more than 255 bytes long. Short fields include: *dc:title, dc:creator, upnp:class, upnp:producer, artist, author, director, genre,* and *album.* Remember that 1 byte does not necessarily equal 1 character because UTF8 encodings allow up to 3 bytes per character.

Some common strings, like URI values and *dc:description*, will need a little more space.

Long-information (*upnp:longDescription*) CDS implementations should be free to have reasonably bounded string lengths on the order of a megabyte or more. Likewise, control points that request such information

[22] The text after the last "/" (or appropriate delimiter) character should have a friendly title (and maybe the creator), or use the name of the local file. Ideally, the URI will end with "*[creator] – [title].[file extension]*".

should be prepared to accept this quantity of metadata. To avoid usability issues resulting from response times of 30 seconds or more, the CDS hierarchy should not employ metadata with excessive string lengths.

5.9.2 Min String Lengths: 30 Bytes

If an implementer imposes a maximum string length that is shorter than the values in 5.9.1, he should not make it less than a total of 10 characters, or 30 bytes. This is appropriate for short information, like creator and title, but for longer information strings the implementer will have to exercise good judgment and provide an appropriate length.

5.9.3 Use 1 and 0 Instead of True and False

To accommodate simplified control points, MediaServers should always report Boolean values of true and false in terms of 1 and 0.

5.9.4 Trim White Spaces From Metadata

As a general rule, CDS implementations should expose metadata with white-space characters trimmed from the beginning and end of string values. This may not apply to some fields, but it does apply to short string values and URI values. Trimming white-space characters can be particularly helpful to control points on embedded platforms.

5.10 More Metadata is Good

A CDS implementation should provide as much metadata as it can because users value more information—even if they will not actually use it. Of course platforms hosting the MediaServer may impose natural limitations.

Limitations on the variety of metadata fields is largely dependent on the purpose of the CDS and its platform. The amount of metadata provided by a MediaServer running on a hand-held PDA or a digital camera may be extremely limited compared to the amount of metadata provided by a MediaServer running on a PC. Although the PDA and PC may both mirror the local file systems, the PDA may not have the resources to thoroughly interrogate MP3 files and respond with metadata acquired from the file's ID3 tags. Similarly, the digital camera and PDA may both run on resource-limited platforms, but because the digital camera limits its metadata to the photographic information, it might provide more useful information about its pictures than the PDA.

5.11 More Accuracy and More Clarity Please

Although more information is good, inaccurate information is useless, and ambiguous information can be extremely difficult to use. For example, a CDS implementation that always reports media items with a class of *object.item* is providing extremely vague information. Likewise, a title like "*Radio Station 1"* is a legal title, but *"KXYZ, 99.9 FM"* is a better, more useful title. As mentioned before, the platform and focus of a CDS will most likely impose natural limitations, but users always benefit from greater clarity and accuracy in metadata.

5.12 Implement Metadata Filtering

In addition to providing as much metadata as it can, a CDS implementation must behave properly when a control point issues a **CDS:Search()** or **CDS:Browse()** action with the *Filter* argument. The only time a MediaServer should respond with all of its metadata is when a control point specifies a star-value (*) for the *Filter* argument. If a control point specifies an empty string, only the fields needed to remain DIDL-Lite schema compliant should be returned[23]. With the exception of the star-value case, MediaServers are behaving incorrectly when they respond with metadata fields that are not present in the *Filter* argument.

```
<DIDL-Lite xmlns:dc=http://purl.org/dc/elements/1.1/ xmlns:upnp="urn:schemas-upnp-org:metadata-1-0/upnp/" xmlns="urn:schemas-upnp-org:metadata-1-0/DIDL-Lite/">
        <item id="12" parentID="10" restricted="0">
                <dc:title>Run Out of Tunes</dc:title>
                <dc:creator>Some Garage Band</dc:creator>
                <dc:date>2002-01-02</dc:date>
                <res protocolInfo="http-get:*:audio/mpeg:*"
                bitrate="16384">http://172.16.0.43:4000/12/Some%20Garage%20Band%20-
                %20Run%20Out%20of%20Tunes.mp3</res>
                <upnp:class>object.item.audioItem.musicTrack</upnp:class>
        </item>
</DIDL-Lite>
```

Figure 18: Metadata Filtering Example, Filter = "*"

<DIDL-Lite xmlns:dc=http://purl.org/dc/elements/1.1/ xmlns:upnp="urn:schemas-upnp-org:metadata-1-0/upnp/" xmlns="urn:schemas-upnp-org:metadata-1-0/DIDL-Lite/">
<item id="12" parentID="10" restricted="0">
<dc:title>Run Out of Tunes</dc:title>
<dc:creator>Some Garage Band</dc:creator>
~~<dc:date>2002-01-02</dc:date>~~
<res protocolInfo="http-get:*:audio/mpeg:*"
~~bitrate="16384"~~>http://172.16.0.43:4000/12/Some%20Garage%20Band%20-%20Run%20Out%20of%20Tunes.mp3</res>
<upnp:class>object.item.audioItem.musicTrack</upnp:class>
</item>
</DIDL-Lite>

Figure 19: Metadata Filtering Example, Filter = "dc:title, dc:creator, res"

<DIDL-Lite xmlns:dc=http://purl.org/dc/elements/1.1/ xmlns:upnp="urn:schemas-upnp-org:metadata-1-0/upnp/" xmlns="urn:schemas-upnp-org:metadata-1-0/DIDL-Lite/">
<item id="12" parentID="10" restricted="0">
<dc:title>Run Out of Tunes</dc:title>
~~<dc:creator>Some Garage Band</dc:creator>~~
~~<dc:date>2002-01-02</dc:date>~~
~~<res protocolInfo="http-get:*:audio/mpeg:*"~~
~~bitrate="16384">http://172.16.0.43:4000/12/Some%20Garage%20Band%20-~~
~~%20Run%20Out%20of%20Tunes.mp3</res>~~
<upnp:class>object.item.audioItem.musicTrack</upnp:class>
</item>
</DIDL-Lite>

Figure 20: Metadata Filtering Example, Filter = ""

[23] It is a common mistake for people to assume that creator and resource fields are required, but they are not.

The examples above describe three DIDL-Lite outputs. Strikeout text indicates material that would be missing from the DIDL-Lite. Figure 18 shows a completely unfiltered response. Figure 19 shows a response that includes title, creator, and resource elements. (Readers should observe that the bit rate attribute is missing from the response because it is not specified in the filter, nor is the attribute required for the DIDL-Lite schema.) Figure 20 is a completely filtered response, with only the necessary elements to provide a schema-compliant DIDL-Lite response.

5.13 HTTP Rules

This section outlines the HTTP rules that should always apply when building UPnP devices.

5.13.1.1 Never Use HTTP 0.9

Devices should never use HTTP 0.9 for requests or responses. This version of HTTP is obsolete and causes problems for many recipients.

5.13.1.2 Closing Sockets After HTTP 1.0 Responses

In HTTP 1.0 server implementations, the server is responsible for closing the socket. Substantiation for this behavior is found in this text from the HTTP 1.0 specification:

> "Except for experimental applications, current practice requires that the connection be established by the client prior to each request and closed by the server after sending the response." (Section 1.3, RFC 1945)

5.13.1.3 Always Respond Before Closing the Socket

Before closing the socket, devices must always respond to an HTTP request with an HTTP response. The response may be a valid HTTP response with the intended body or an HTTP error message. Devices that close the socket without sending a response can cause problems for some HTTP clients. When an HTTP client that pipelines requests attempts to recover from an unanswered request by retrying the original request, the retry-recovery algorithm may actually result in the device being flooded with requests.

5.13.1.4 Always Specify the Content-Length

Devices should respond to HTTP messages with the content-length field[24]. This behavior resolves the ambiguities resulting from platforms that set the socket's LINGER flag to true. Devices that set the LINGER flag to true without specifying the content length have sometimes caused control points to process a subsequent event before the initial event. This occurs because the socket used for the initial event is closed by the device after the socket for a subsequent event closes.

When devices specify the content length in all their HTTP messages, it allows the control point to close the socket if the software logic needs to prevent timing errors between separate HTTP sessions.

[24] The context of this section is UPnP-related HTTP messages. The content-length header may not be present when distributing live content streams. Also, the content-length field has a different purpose when used in a response to a HEAD request.

5.13.1.5 HTTP Header Rules

The following rules apply to the headers of HTTP messages.

- HTTP header names are case insensitive. Devices should not falter when control points issue HTTP requests with valid header names.
- HTTP headers may have leading and/or trailing white space characters before the colon (:) delimiter. Devices and control points need to properly parse HTTP messages with such headers.
- HTTP-GET messages must have the "HOST" header entry. This is required by the HTTP specification. HTTP-GET requests must properly escape the requested URI target.

Plugfests have shown that a number of interoperability problems are caused by the false assumption that everybody formats their HTTP headers in the same manner. The syntax rules for HTTP headers are well defined, but the formatting of the syntax is not.

5.14 MediaServer Support for HTTP-GET Content

To maximize interoperability and usability with MediaRenderer devices, MediaServers that distribute content through HTTP-GET should implement the rules defined in this section.

5.14.1 No PrepareForConnection for HTTP-GET MediaServers

Given that HTTP-GET is probably the most prevalent transport protocol used for streaming content at UPnP AV plugfests, it is important to understand how the ambiguities of connection lifetime can cause problems for a MediaServer.

Consider a MediaServer that implements PrepareForConnection and supports many types of protocolInfo, including those that rely on HTTP-GET. When a control point invokes PrepareForConnection on the MediaServer, the device responds with a ConnectionID. Subsequently, the control point instructs the MediaRenderer to stream and play the content.

When the renderer begins streaming, how does the MediaServer know that a particular HTTP-GET request from the renderer is actually a request intended for a particular connection? All the MediaServer knows is that it is supposed to prepare for an http request for some kind of content. It lacks the information that would provide the logical binding between a specific ConnectionID and the underlying HTTP-GET request. Attempting to *guess* the binding[25] based on time, protocol, and content type is an extremely flawed response in an environment with multiple streams and/or multiple renderers[26].

Assuming a control point has rights to close a connection (with the intent to create a new one), the device faces yet an additional obstacle—actually closing the correct connection. Is the MediaServer supposed to close a

[25] The binding in question is the logical relationship between an HTTP request and a PrepareForConnection invocation.
[26] Some have suggested that providing the intended URI in the PrepareForConnection would be sufficient information for the MediaServer. Unfortunately, many renderers are likely to implement SEEK (or Next with playlist files) by using HTTP-GET requests with the RANGE header as a means of seeking-to-positions within a file. Every time the renderer seeks, the renderer issues a new request. All this leads to the question: Does connection lifetime end with an HTTP request or through ConnectionClose? If it ends with the HTTP request, the framework is faulty—especially given that a Stop transport action will likely close the HTTP session that is streaming the content. If it ends with ConnectionClose, the device becomes dependent on a control point calling the method. While a solution based on ConnectionClose is architecturally good, it leaves a MediaServer open to the possibility of connections that are never closed. The only way for a MediaServer to close connections is to monitor HTTP activities. Unfortunately, since content advertised through a MediaServer has no requirement to actually be stored/served by the MediaServer, there is no feasible solution.

socket? Given that MediaServers are not even guaranteed to be hosting the content that they are advertising, how is this to be done? Even if the behavior is as simple as removing a connection ID from a list (and not closing a socket), this opens the device to a potential memory leak at unclosed connections.

In another proposed solution, MediaServer's always return the same connection ID for a protocol like HTTP-GET, and reply with new connection IDs for protocols that have hard bindings. Unfortunately, this type of inconsistency requires control points to understand how protocols and transports behave, and this runs counter to UPnP AV being agnostic about those things.

The truth is that PrepareForConnection is extremely unfriendly to HTTP-GET MediaServers. The underlying problem is that HTTP-GET connections are ethereal in nature. In fact, the actual content may not even be hosted by the MediaServer. An HTTP-GET connection exists insofar as it is necessary to logically represent an AV connection. Unfortunately, an HTTP-GET connection does not have a set of reasonable and consistent assumptions for MediaServers and control points. The connection provides no useful purpose and only adds to the ambiguity and possible abuse of a MediaServer.

If a MediaServer wants to mix server-side controlled streams (such as IEEE1394) with client-side controlled streams (HTTP-GET), the vendor should implement two MediaServers. The MediaServer with server-side control abilities will have the PrepareForConnection action, while the client-side MediaServer will not. Both MediaServers could be embedded, or one could be the parent of the other. The device hierarchy does not matter compared to the logical separation of the devices.

5.14.2 HTTP-GET ProtocolInfo For All Content Types

MediaServers that can serve any type of content (such as a MediaServer exposing a local file system) can use the "*http-get:*:*:**" protocolInfo string to inform control points that all forms of content can be acquired from the MediaServer. This protocolInfo string can be used in the **ConnectionManager** service's *GetProtocolInfo* action and in the associated state variables for representing the allowed source types for the MediaServer. The only time this string should be seen as CDS metadata is in a media object entry with a resource that has been declared (and the MediaServer does not know the content type because it has yet to acquire the content).

5.14.3 Mime-Types and File Extension Mappings

The typical format for an HTTP-GET protocolInfo string is: *http-get:*:[mime-type]:**. Table 1 contains a listing of common mime-types that should be used to populate the third field of a protocolInfo string. In addition to using protocolInfo strings that are advertised by **ConnectionManager**, **ContentDirectory** services need to use the proper protocolInfo strings in their advertised resource (a.k.a., res) elements. To enable friendly behavior for many HTTP streaming engines, **ContentDirectory** services are encouraged to use the associated file extensions for advertised resources.

Table 1: Mime-Types and File Extension Mappings

Mime-Type	File Extension	Description
	Audio Formats	
audio/aiff	.AIF, .AIFF	Apple* Audio Interchange File Format
audio/basic	.AU, SND	Sun Microsystems* Unix* audio and Sound audio formats
audio/lpcm	.LPCM	LPCM audio
audio/midi	.MID, .RMI	Musical Instrument Digital Interface audio format
audio/mp1	.MP1	MPEG audio layer 1

Mime-Type	File Extension	Description
audio/mp2	.MP2	MPEG audio layer 2
audio/mpeg	.MP3	MPEG audio layer 3
audio/x-ac3	.AC3	AC-3
audio/x-aac	.AAC	AAC
audio/x-atrac3	.AT3P	Sony* ATrac-3Plus*
audio/x-dts	.DTS	DTS
audio/x-ogg	.OGG	Ogg Vorbis* audio format
audio/x-quicktime	.MOV, .QT	Apple QuickTime* audio format
audio/x-pn-realaudio	.RA	Real Networks* audio format
audio/wav	.WAV	Microsoft Waveform* audio format
audio/x-ms-wma	.WMA	Microsoft Windows Media Audio format
Video Formats		
video/x-dv	.DV	Digital Video format
video/x-motion-jpeg	.MJPG	Motion JPEG video format
video/quicktime	.MOV, .QT	Apple QuickTime video format
video/MP1S	.mpeg, .mpg, .mpe, .m1v	MPEG-1 System Stream
video/mpeg2	.MPEG2, .MPG2	Moving Picture Experts Group 2 video format (program and transport streams)
video/MP2T	.mp2t	MPEG-2 Transport Stream
video/MP2P	.mp2p, .vob	MPEG-2 Program Stream
video/MP4V-ES	.m4p, .mp4	MPEG-4 Stream
video/x-pn-realmedia	.RM	Real Networks* video format
video/x-ms-wmv	.WMV	Microsoft Windows Media Video format
Image/Icon Formats		
image/bmp	.BMP	Microsoft Bitmap raster format
image/gif	.GIF	Graphics Interchange Format*
image/x-icon	.ICO	Microsoft Icon* format
image/jpeg	.JPG, .JPEG	Joint Photographic Experts Group image format
image/png	.PNG	Portable Network Graphics format
image/x-quicktime	.QTI, .QTF, .QTIF	Apple QuickTime image format
image/tiff	.TIF, .TIFF	Tagged-Image File Format
Control/Metafile/Streaming Formats		
video/avi	.AVI	Microsoft Audio Visual Interleave file format
video/x-ms-asf	.ASF	Microsoft Advanced Streaming Format
video/x-ms-asx	.ASX	Microsoft Advanced Stream Redirector file format
audio/x-mpequrl	.M3U	MP3 play list metafile format
audio/x-pn-realaudio	.RAM	RealAudio media metafile format
application/smil	.SMI, .SMIL	Synchronized Multimedia Integration Language
Miscellaneous		

Mime-Type	File Extension	Description
application/octet-stream	(any)	Any unknown data types
text/html	.HTM, .HTML	HTML files
text/plain	.TXT	Plain text files

5.14.4 HTTP-HEAD

MediaServers should be able to respond to HTTP-HEAD requests from a MediaRenderer. This allows renderer devices to acquire the HTTP headers of a file. The headers returned should include the content type and the content length of the actual file (if known). Clients requesting an HTTP-HEAD are expected to understand that the content length header may not represent the actual content length of the body. If the content length is not known (perhaps it is live content distributed over HTTP), the content-length field should not be included in the server's response to the HEAD request.

5.14.5 HTTP-RANGE

In order to allow renderers to efficiently implement *Seek* behavior with HTTP-GET based content, the devices need the ability to specify ranges of content. Furthermore, the ability to download portions of a playlist file (such as an M3U or ASX) is invaluable to a renderer that cannot download the entire playlist. For this reason a MediaServer should implement support for HTTP-RANGE. The only time a device should not worry about supporting RANGE requests on content is when the content is some form of live stream, where RANGE really does not apply.

Even though HTTP 1.0 does not specifically describe RANGE as an option[27], MediaServers serving their local content should be able to accept an HTTP 1.0 GET request with the RANGE option. Requiring a lightweight MediaServer (or the client MediaRenderer) to support HTTP 1.1 is probably more work than it is worth. However, given that HTTP 1.0 permits the presence of additional headers, adding functionality for a MediaServer to support the RANGE option in HTTP 1.0 is both easy and beneficial to client devices. In the worst case, a client can always get a 200-OK response from the server and receive the entire file.

MediaServers that respond to an HTTP 1.1 request with an HTTP 1.1 response should properly support multi-part RANGE requests. Although many argue that supporting multi-part RANGE requests is not worth the effort, implementers need to respect the specifications that are foundational to the UPnP framework. That being said, MediaServers that cannot support multi-part ranges can always respond with an HTTP 200 OK message and send the entire file. A MediaServer (either HTTP 1.0 or 1.1) that supports a single RANGE, but lacks support for multiple ranges, should never respond with an incomplete set of ranges or ranges that do not correspond to those specified in the request.

5.14.6 Content-Length

When HTTP web servers respond to an HTTP request, the responses should include the *Content-Length* whenever possible. Providing clients with information about the total length of a file (such as an audio, video, or playlist file) greatly improves the opportunities for a client to optimize the processing of the downloaded content. A valid exception to this rule is when the device is responding with a live stream.

[27] Some contend that the specification alludes to the presence of RANGE.

5.14.7 Chunked Encoding

MediaServers can make use of chunked HTTP 1.1 responses. Ideally, chunked responses are limited to live-content streams, but different implementations may have reason to use chunked responses for all HTTP 1.1 traffic. The most common reason being that underlying media distribution middleware may be implemented outside of the UPnP implementer's control. Whatever the reason, implementers are forewarned that not all HTTP clients have robust implementations and that support of chunked responses is often ignored. As a corollary, implementers of HTTP clients are strongly encouraged to build robust implementations that properly handle all aspects of HTTP 1.1, including chunked encoding.

5.14.8 Pipelining Support and Persistent Connections

An HTTP 1.1 client that issues pipelined requests is issuing its multiple HTTP requests on the same socket. This type of an HTTP socket is known as a *persistent connection*[28]. Contrary to popular belief, this is possible for both HTTP 1.0 and 1.1, although HTTP 1.0 implementations of persistent connections have some flaws when proxy servers become part of the usage. Intel sees no harm in vendors choosing to implement HTTP 1.0 persistent connections because activating a persistent connection is always initiated from the HTTP client. HTTP 1.0 server implementations that do not support persistent connections need to respond appropriately by sending a complete response to the first request, followed by closing the socket.

In any case, using the same socket to issue requests saves time and platform resources because the TCP phase of socket creation and tear-down is removed between requests. The pipelining of requests on a persistent connection works the same for HTTP 1.0 and HTTP 1.1, with the exception that HTTP 1.1 supports chunked encoding and is friendly to proxy servers.

Most HTTP requests work fine in a pipelined series of requests. However, the HTTP HEAD request is an exception to this rule. As it stands the response to a HEAD request can specify the content length as the "size of the entity-body that would have been sent had the request been a GET" (section 14.14, HTTP 1.1 specification, RFC 2068). Proper interpretation of this header field is the burden of the HTTP client. Some could argue that the HTTP server should always close the socket after a HEAD response, but Intel believes the benefits of persistent connections outweigh the problems caused by bad HTTP clients. As such, Intel believes it is ultimately the responsibility of the client to properly interpret the HEAD response.

5.15 Internationalization

UPnP devices and control points should work together even if they are not from the same region of the world. The UPnP specification makes heavy use of the UTF-8 encoding; implementers that are not familiar with this encoding are encouraged to research it and use it. Many developers are often surprised to learn that 16 bits (which allows for 65535 characters) is not sufficient to encode every letter of every alphabet in the world. In fact, it does not even come near what would be required. The UTF-8 encoding allows for more than 16 bits per character, therefore there is no reason to truncate UTF-8 encoded characters.

Control Points and renderer devices that do not have a user interface do not need to worry about the specifics of UTF-8. Most western string operations in languages such as C and C++ will still work with UTF-8 encoded strings since UTF-8 strings are also null-terminated.

While the XML specification allows for both UTF-8 and UTF-16 encodings, only UTF-8 should be used when implementing UPnP devices. Many developers incorrectly assume that UTF-16 can encode more characters, or

[28] Full discussion on persistent connections and the pipelining of requests can be found in the HTTP 1.1 specification, RFC 2068.

is required to encode Asian characters. Both UTF-8 and UTF-16 can encode characters that are over 24 bits wide. Besides the additional flexibility gained from more bits, UTF-8 is also backward compatible with the ASCII character set.

When encoding a URI, always encode it using UTF-8. If URI escaping is needed, encode the URI into UTF-8 before performing the escaping operation. The opposite is also true, always un-escape a URI according to the URI's scheme before decoding it into the UTF-8 format.

5.16 Represent Media Collections with Container Objects

The **ContentDirectory** specification uses **<container>** elements to represent photo albums, music albums, playlists, and other forms of media collections. However, the specification also defines the **object.item.playlistItem** media class for playlists.

Developers should avoid using the **object.item.playlistItem** media class, and other item representations, for media collections. Having multiple methods for representing media collections can be problematic for control points.

CDS implementations should always use media classes that derive from **object.container.playlistContainer**. A container and item object both allow advertisement of resources, but a container allows the implementation to optionally provide additional metadata about the content in the media collection.

As a corollary, a container representing a collection of media should at least have one resource for the container-derived media class. This allows control points the ability to use the container's resource to obtain the entire collection in the form of a playlist-like[29] file.

A CDS implementer is encouraged to provide CDS metadata for individual content pieces that make up the media collection. The metadata for each content piece would appear as a child object[30] of the parent container. If resource URI values are known for the individual content pieces, a CDS implementation should also provide that information as part of the child object. As a general rule, a playlist-like file that refers to its individual content pieces through URI values[31] should have the child objects expose those URI values. A file format that actually embeds all individual content pieces into its binary format will not likely expose URI values for individual content pieces; however, those implementations may still be able to expose basic metadata.

5.17 Advertise Fully Qualified, Non-Local URIs

A MediaServer should not expose local path information for individual content pieces in CDS metadata, in a playlist, or in other similar media collections. Local file paths, or even UNC-formatted paths, are of little use to most renderers. Similarly, relative URI paths are of little use to many clients.

5.18 Playlist Files and Metadata

Practice has determined that control points should avoid sending DIDL-Lite metadata to renderer devices when calling the **AVTransport**'s SetAVTransportURI action. Instead of the control point sending metadata to the renderer device, the MediaServer implementation should embed metadata into its hosted playlist resources.

[29] There may be other forms of playlist files, like a scripted slideshow or presentation file format.
[30] It is theoretically possible that a child object can be another container object. This is not recommended for most forms of playlists, although some content forms may require this type of a model.
[31] The M3U file format is an example of such a file format. In its basic essence, an M3U file has a set of URI references to other files.

At a minimum this information should be the title of the individual content, but creator information can also be of great help.

As a practical example, consider the Nullsoft* M3U and Microsoft* ASX file formats. Both formats allow metadata to be embedded into the playlist files. If a renderer can parse the files for URI values, then the renderer can additionally extract metadata about the current track. Supplying even the most basic metadata provides a better user experience than no metadata.

5.18.1 Recommendations for M3U Metadata

Given the common use of M3U and the numerous possibilities for how metadata can be stored in the file, some suggestions on how to format the metadata are in order. When exposing and hosting M3U files, MediaServers should use the Extended-M3U format. The Extended-M3U file should conform to the following conventions:

- An M3U playlist file is a carriage-return, line-feed (CR/LF) delimited list of explicit, non-relative, fully-qualified URI strings. An M3U file may optionally contain lines that begin with the token #EXTINF followed by information such as duration and other short textual fields. A #EXTINF line must precede the line with the associated URI. The #EXTINF entries end with CR/LF.
- Playlist files that contain #EXTINF tokens must also begin the file with the #EXTM3U token on a single line. The #EXTM3U line ends with CR/LF.
- The format of a #EXTINF lines is as follows (all characters are literal, except brackets and text enclosed in brackets):

  #EXTINF:[integer duration in seconds; -1 if duration is unknown],[text]
- A UPnP media server content directory service that advertises an M3U playlist file with #EXTINF tokens should set the [text] field to contain the artist information and title, delimited by the space-dash-space string (-). If only one piece of information is known, then the space-dash-space delimiter should still be present to simplify renderer parsing of the text.
- If a URI's scheme has specialized escaping conventions, such as for HTTP, then all URI values that specify that scheme must be properly escaped according to the scheme's specific conventions. See section 5.15, Internationalization, for additional information on escaping with international characters.

6 Key Design Decisions

Experience has shown that the process of building a MediaServer can have many pitfalls. This section provides encouragement and advice for the implementer to consider during the design phase.

6.1 Information System's Metadata Fields

The CDS specification provides a wealth of knowledge on CDS-normative metadata. The specification authors, when tasked with determining the (minimum) metadata fields that are normative, decided that very few are actually required. Even the tables that describe metadata appropriate for certain media classes indicate that most metadata fields are not required. The freedom to specify very minimal metadata sets affords MediaServer implementers a great deal of flexibility as they carefully consider the metadata requirements of their CDS.

6.1.1 Thinking Ahead

Implementers are encouraged to have an expansive vision of their product's features and implementation that is balanced with thoughtful limitations. Implementers should not design MediaServers that can accommodate all types and sizes of metadata, nor should they design MediaServers that can accommodate only a single product revision.

The scope of the metadata fields should be liberal. If a product might eventually need metadata for a certain task, the initial design should account for that potential. It is easier during initial design to declare that all media objects have an empty string for a *description* metadata field than to be forced later to build a new database to accommodate a description element.

Keep in mind that metadata fields are tied into the information system. In a well designed MediaServer, the exposed features are sufficiently decoupled from the information system to in no way hinder potential capabilities. In summary, the information system should be feature rich and robust—even when the product does not seem to warrant an information system with these capabilities.

6.1.2 Settings Limits on Search

The **CDS:Search()** action is an extremely convenient method, but CDS implementers are not obligated to allow a control point to search on every CDS metadata field that is employed by the MediaServer. Even though a MediaServer could be created with the ability to perform a search query that involved any type of metadata, an analysis would likely indicate that the trade-offs and/or requirements placed on the implementation are unacceptable. As an example, consider an implementation where the information system will store lyrics. If the implementer chooses to allow a control point to issue a query that requires examination of song lyrics, the implementation is likely to fail the requirement to respond within 30 seconds because the additional requirements placed on the relational database would be too great.

Implementations that limit the scope of searches should at least provide the ability to search on basic fields like title, creator, media class, and for the protocolInfo attribute on a resource element. Other fields are always a bonus, but these metadata fields are very much at the core of most search queries.

6.2 Information System's Infrastructure

The design decisions regarding the technique for storing and using metadata are very much tied to the decisions that determine the supported metadata. The two general techniques that seem to be at the forefront are the database approach and the file system approach.

6.2.1 The Relational Database Approach

Many view the relational database approach as ideal because it usually provides sufficient layers of abstraction to allow product evolution over a long period of time. This approach is also the technique of choice for those who already have a database infrastructure that catalogs content. This technique is also great for implementations that are required to handle **CDS:Search()** requests since many relational databases excel in such tasks.

A well thought-out design properly isolates the information system from the CDS-specific logic, building the CDS logic as a front-end module for the database. Such a technique allows the CDS logic to focus on accepting the CDS requests, performing operations (both query and modifications) on the database, and transcribing the metadata in the database into DIDL-Lite in CDS responses. As for the database, its design and purpose is focused on defining the supported metadata fields, storing the metadata (and possibly the content too), and responding to operations that can eventually be translated into CDS-compliant response messages.

As a side note, implementers interested in using an existing relational database may find it a practical necessity to augment the existing database to accommodate certain requirements of a CDS response. Assuming that the relational database has columns for title and creator metadata, a CDS is still required to support the **CDS:Browse()** action. One of the more subtle requirements of this action is that every **container object** is required to report the number of child objects through the **container@childCount** attribute. Thus the typical relational database, essentially consisting of a flat list of content metadata, may require modifications so that the database provides information needed about the **container** hierarchy exposed by the CDS.

6.2.2 The File System (or Hierarchical Database) Approach

Not every implementation demands a relational database approach, and some implementers do not have the platform resources for a database. This being the case, the most common alternative has been to use a local file system for both content storage and the information system. This technique works when the file system and the content on the file system have sufficient information for populating the metadata fields. (It should be noted that a file system is essentially an instantiation of a hierarchical database.)

As a practical example, consider a lightweight MediaServer running on a hand-held PDA. The MediaServer exposes the contents of the local file system as a hierarchy of media containers and media items. Each item's title is the name of the file on the storage, and a resource is a mapping that allows a network entity to download the file.

When a MediaServer runs on a platform with greater resources, like a desktop PC, a more complex implementation of this technique is appropriate. Such a MediaServer can implement a modest, object-oriented database that ties into the file system and provides a real-time representation of the file system in object form. Instead of providing the filename as the title, the CDS logic could analyze the files and obtain more useful information to use as metadata. For example, Microsoft Word documents contain metadata about the document. Audio files, like MP3 and WMA, have information about the artist, title, and album of the song. Images have information about resolution and possibly the time and place of image capture.

The biggest drawback of this technique is that it seldom scales as well as a database approach when it comes to handling search. This technique usually involves linear-time search algorithms, which may only work with small CDS hierarchies. Indexing techniques can improve performance, but it is unlikely such an implementation will match a database. This technique requires the implementer to value the efficiency of mirroring a content store over the efficiency of handling search requests.

6.2.3 Building an Efficient CDS

Those interested in building reusable CDS implementations have often wondered if it is possible to build a CDS solution that is flexible enough to accommodate all types of back-end information systems. While it is possible to build a MediaServer implementation that can support a variety of information systems, the larger risk is creating an inefficient MediaServer implementation.

Generalized CDS implementations often exhibit bad performance. Although some may think that using all of the permitted 30 seconds for responding to a CDS request is acceptable, an intuitive observer will notice that such an assumption exaggerates the average consumer's patience threshold. Given technological societies' trend for instant gratification, the consumer of UPnP devices are more likely to be impatient.

Because a MediaServer is often a hub for content distribution, the likelihood of multiple control points making queries in a rapid or simultaneous manner is very much a reality. For this reason, CDS implementations (especially those that represent large content repositories) have a practical necessity for being extremely optimized. (Although there exists no performance requirement, some have argued that CDS responses should never take longer than 10 seconds. Some even say 5 seconds.) Although the appearance of high performance can often be gated by the performance of the control point browsing the content, MediaServer devices are no less responsible for ensuring that queries by answered quickly.

Because of the practical need for quick response times, a CDS implementation often needs to have an optimized infrastructure for handling **CDS:Search()** and **CDS:Browse()** actions. Software layers designed to abstract the detailed methodology of enumerating the information-system often lead to bad performance. The methodology for enumerating a relational database is fundamentally different than enumerating a hierarchical database. In fact, the way in which one enumerates a relational database (with intent to expose a hierarchy, as expected by CDS control points) can easily differ as a result of the desired container hierarchy.

So the message to implementers of CDS middleware is simply this: Properly scope your choice of supported information-systems. Properly analyze the risk posed by attempting to add abstraction layers between the logic that obtains metadata information and the logic that serializes the DIDL-Lite. Always remember control points that enumerate a CDS will almost never see CDS metadata as a relational list because the baseline means of enumerating a MediaServer's content is always that of a hierarchy.

6.3 Multi-NIC or Single-NIC Systems

A CDS designed to run on a set-top box, or on a closed system with a single network interface, can define itself as a single-NIC CDS implementation. Every other implementation should run properly on a host machine with multiple network interfaces. It is clear that host machines with multiple network interfaces are a permanent part of our digital ecosystem and implementers need to accommodate them. Any CDS implementation designed to run on a high-end PDA or modern PC is headed for disaster if it does not account for multiple network interfaces.

As such, the following rules should be applied to MediaServer implementations that serve content and support multiple network interfaces.

- Represent content available on multiple network interfaces with multiple resource elements of the media object, where each resource has a URI that is routable from one of the machine's interfaces.

- When serializing multiple resources of an item, first group the resources by their interface, and then, within these groups, order the resources according to what the print order would be if there was only a single network interface.

- When serializing multiple resources of a media object, make sure the first group of resources have URI values that are accessible from the network interface that received the CDS query/request.

There is an exception to the definition given above of what a single-NIC CDS implementation is and it occurs under these conditions:

- An IEEE1394/CAT5 MediaServer is part of a stereo system
- Other AV components are designed to handle all UPnP activities on the CAT5 network
- The IEEE1394 network is reserved for the out-of-band transport of content

- The CDS (logically) advertises its content on the CAT5 network

In such a scenario, the implementation is treated as a single-NIC implementation, as its UPnP activities only happen on a single network interface.

7 Summary

In the interests of interoperability, Intel has provided this document as a means of sharing key learnings and suggestions to the industry. While this document addresses many important issues related to MediaServer design, it is acknowledged that implementers are still left with many unanswered questions. It is hoped that future efforts by individuals, the industry, and the UPnP AV Forum will resolve these questions, even as new ones are raised.