DVB-MHP/Java TV™ Data Transport Mechanisms

John Jones

Senior Java Architect
Sun Professional Services Pacific Region
Level 4, 33 Berry St
North Sydney NSW 2060

john.jones@aus.sun.com

Abstract

With the advent of digital television, more specifically interactive television, the emergence and adoption of the Java™ DVB-MHP standards here in Australia provides developers with a range of new technologies and issues.

An appreciation of broadcast technologies, in particular MPEG-2 and the Object Carousel are required to understand the facilities and constraints of this new Java™ technology environment.

This paper covers the embedded Java™ technology APIs related to the entire end-to-end broadcast chain: object carousel, the set-top-box environment, AWT GUI development, sending and handling data-updates return channel usage, connection, and back-end servers.

The emphasis here is on the Java™ DVB-MHP APIs that support communication mechanisms within the interactive television environment.[1]

Keywords: Java, MHP, DVB

1 Introduction

On the 31 October 2001, the Federation of Australian Commercial Television Stations (FACTS) Digital TV Strategy Group announced the Australian Free-To-Air television industry's support for DVB-MHP (Multimedia Home Platform) as the standard for delivery of interactive television services.

This paper examines design issues facing developers in this new area. In particular it focuses on the MHP Java™ technology APIs and facilities available to manage the flow of data through the system. Although the control and display of audiovisual content is a key aspect of most interactive applications, understanding the APIs and facilities available to move data through the system is essential in designing successful end-to-end interactive applications.

As a means of examining these APIs, a simple interactive application is used to illustrate the progress of data through the system at each stage in the application's life cycle.

2 Brief Introduction to Interactive TV

To understand the DVB-MHP APIs presented here, some explanation of the broadcast chain is required.

2.1 What is DVB-MHP

The Digital Video Broadcasting Project (DVB) is an industry-led consortium of over 300 broadcasters, manufacturers, network operators, software developers, regulatory bodies and others in over 35 countries committed to designing global standards for the delivery of digital television and data services.

The Multimedia Home Platform (MHP) defines a generic interface between interactive digital applications and the terminals on which those applications execute. This interface de-couples different provider's applications from the specific hardware and software details of different MHP terminal implementations. It enables digital content providers to address all types of terminals ranging from low-end to high-end set top boxes, integrated digital TV sets and multimedia PCs. The MHP extends the existing, successful DVB open standards for broadcast and interactive services in all transmission networks including satellite, cable, terrestrial and microwave systems. MHP is a DVB initiative.

DVB specifies Java™ technology as the application environment language. It also defines a suite of APIs that include most of the Java TV™ API, HAVi (user interface), DAVIC APIs and DVB APIs.

The applications downloaded to the Set-top-box (STB) are Java™ applications, built on a suite of APIs tailored specifically for the interactive TV environment.

MHP 1.0.1 defines two profiles:

1. Enhanced broadcast: The digital broadcast of audio and video services is combined with executable applications. These applications enable the viewer to interact locally. It does not require an interaction channel.
2. Interactive Broadcast: In addition to the features provided by enhanced broadcasting this also enables a range of interactive services associated or

independent from the broadcast services. This application area requires an interaction channel.

Although HTML support on the STB is an optional feature of the next version of MHP, MHP 1.1, this paper deals mainly with the second profile – interactive broadcast.

1.2 The Interactive Broadcast Chain

The design of interactive TV applications requires an understanding of technology that spans the complete broadcast chain including any available return channel. This encompasses not only the STB but also the broadcast equipment (head-end) and the application servers used to support the return channel communications. In simple terms, the complete assembly looks like:

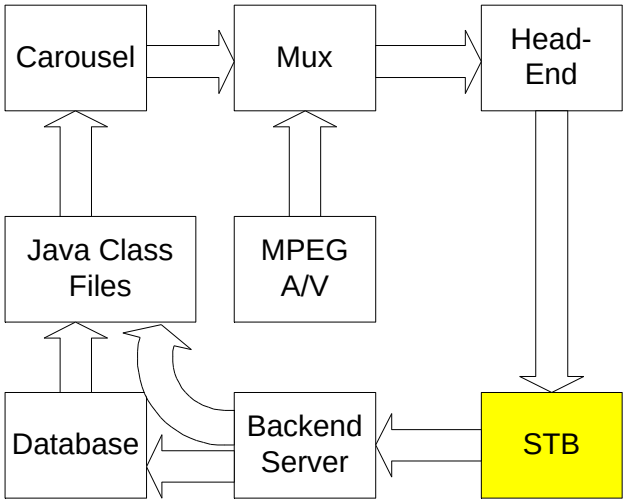

Figure 1

The diagram illustrates the use of a carousel to continuously play-out a Java™ application. The application and corresponding audio-visual material are then multiplexed to form a single transport stream. The transport stream is broadcast through a head-end. The resulting broadcast is received and decoded by the set-top-box (STB), the audio-visual content played and the Java™ application run. Subsequent user interactions with the application lead to information being sent via the return channel to a back-end server. Depending on the application, this information may result in modifications to the current application content, e.g. voting information or stored for later processing in a database, e.g. for an on-line shopping application.

1.3 MPEG-2

MPEG-2 plays a central role in digital TV. It was designed specifically for encoding audio-visual content for TV. Not only is it a standard for encoding audio and video, but it is also used as the means by which raw data and applications are transported in the broadcast stream. It does this by defining a transport format. This format has at its core a packet definition. Individual data-streams may be multiplexed together by labelling each packet with a corresponding stream identifier: PID. For example, a video stream may be packetized and each instance labelled as PID 1234, the audio stream as 1235 and a raw data stream as 1236. These packets may then be multiplexed in the same physical stream. The packet ratios reflect the required bandwidth of each stream – gaps in the stream are padded with special 'blank' packets.

Where the stream is to be broadcast and not just recorded, it must be modulated and up-converted to the appropriate frequency for transmission. The modulation mechanism varies depending on whether the transmission is terrestrial, satellite, cable or microwave. These issues do not play a significant role in the application architecture and are not discussed here.

MPEG-2 also defines a means of identifying which packet identifiers mean what, so that when the streams are re-assembled we know what to do with them. The scheme is quite simple; there are a number of reserved PIDs that hold information about the other PIDs. PID 0 is the Program Map Table (PMT), which is a map of all the Program Application Tables (PAT), which contains a list of PIDs for audio-visual and data streams used for a given program. There are a host of other pieces of information embedded in these packets that relate to clock references, audio encoding method etc. DVB have extended this scheme for MHP by specifying how to embed a Java™ application within the stream, this includes information on how to specify the main class, class search path and the application argument list etc.

1.4 Carousel

Although MPEG-2 provides a means of transporting our Java™ application along with the audio-visual content, there is a problem in that the viewer may change channel and select our (Java™) program at any point in the transmission. Unless we elect to view that program from the beginning, our application would already have been broadcast. As a result, the STB and the viewer would have missed it. We need a means of sending it over and over again so that the STB can pick up and assemble a complete application at any time during the transmission. This is exactly what a broadcast carousel does – it keeps playing the same application around and around. The packetized application is then continuously multiplexed with the audio-visual content for transmission. We can now join the transmission at any time and still have access to the interactive TV application.

1.5 Xlet

MHP applications are not complete Java™ applications in the normal sense. These applications are much more like Applets in that they are loaded and run by a life cycle manager, in this case residing on the STB. These applications are called Xlets. To implement an Xlet interface you have to implement a small number of methods that control its life cycle.

2 Simple Model Application

To illustrate the key data transport APIs involved in developing an interactive TV application, we are going to use a simple example: a TV polling application. The data transport issues raised will be typical of those for most interactive applications. Using the logical diagram (Figure 1) we will examine the key mechanisms and constraints provided by MHP at each stage in the application's life cycle.

The stages discussed are:

1. Down loading a Java™ application to set-top-box via the broadcast stream
2. Starting the application
3. Viewer selects candidate list and votes by selecting a corresponding number on the IR control
4. System sends voting information via the return channel to a server
5. Screen display periodically updates to show candidate votes.

3 API Mechanisms and Constraints

In this examination of our simple application, only the important transport mechanisms are illustrated and discussed – there are many other issues related to interactive TV development that are not covered here, e.g. graphics, keyboard handling, multiple audio-video streams etc.

3.1 Stage 1 - Transmitting the Application

The sequence of events is as follows:

- The Carousel picks up the Java™ application and resource files (GIFs, text files etc) and repeatedly plays them out to a multiplexer.
- The multiplexer combines the audio-visual material and the raw application data files and plays them out to a head-end for broadcast.

3.1.1 Modules and Granularity

Although this first stage of the process is independent of the MHP APIs, its understanding is important for what follows later.

Carousels may be configured to play-out different groups of files at different rates. There are benefits in partitioning the application to take advantage of this fact. E.g. the initial icon and associated menu could be sent more frequently than other supplementary information. Arranging the content in this manner will result in the viewer seeing the application appear more rapidly than if the files had equal priority. This rapid appearance is important to give viewer the notion of a responsive system.

The bandwidth available for application transport varies depending on how the transport stream is constructed. As a simple example, an application of 1 Mega Byte transported over a 1 Mega bit per second link will take approximately 10 seconds to download (cycle) – including transport headers etc. On average, this means that it will take 10 seconds for the application to be fully available for the user.

Under the scenes, these modules are sent as 64K chunks. As spare space is not used for other purposes, there is little benefit in attempting to reduce the space by packing a group of files that are smaller than 64K. There may be some benefit in placing related files in the same module as some set-top-boxes may also choose to cache their files in this manner – although there are no guarantees.

3.2 Stage 2 - Loading the Application

The sequence of events is as follows:

- The STB de-constructs the individual streams and assembles the data and class files.
- The STB reads the information tables and determines which class file to run as an Xlet.
- The Xlet is started.

3.2.1 Restricted Environment

STBs have a restricted run-time environment; they are short on memory, disk space and CPU speed. It is important that applications have a small footprint. This not only affects the download speed to the STB but also its chances of being run at all! Special attention must be given to discarding resources when they are not needed. Although Java™ provides garbage collection, it is recommended that some operating system resources be explicitly discarded, e.g. image buffers and file descriptors.

3.3 Stage 3 - Running and displaying the Application

This is the first stage in the process that MHP plays a part in. The sequence is as follows:

- The application loads resources (when required) as if they exist locally.
- Typically when initialisation is complete, a visual indicator of some kind is displayed at the bottom corner of the screen – known as a 'bug' in the industry.

3.3.1 Xlet Life Cycle

The key Java™ classes are as follows:

```
javax.tx.xlet.Xlet
javax.tx.xlet.XletContext
```

The class `javax.tx.xlet.Xlet` is the entry point for all applications, an Xlet's state can change either by having one of the methods on its Xlet Interface called, or by making an internal state transition and notifying the application manager via the `XletContext` Object.

The following diagram illustrates these state transitions:

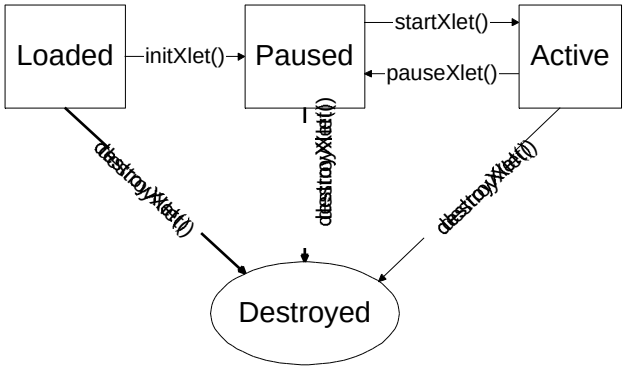

Figure 2

3.3.2 Accessing Files – Carousel

The key Java™ classes are as follows:

```
org.dvb.dsmcc.DSMCCObject

org.dvb.dsmcc.AsynchronousLoadingEvent
```

The file system of the carousel (all the application and resource files) appear as a 'mounted' drive on the STB. All the files are visible and accessible - but are read-only. MHP specifies the DVB carousel file interface. This file interface differs from the Java TV™ carousel interface in that it supports events that may be used to notify the application of a file change, or to detect when a file has been loaded asynchronously. Both these APIs extend the standard Java™ File class. Once opened, a file can be used in much the same way – remembering that they are read-only.

The following code illustrates the asynchronous file loading mechanism. Lines of particular interest are highlighted:

```
import java.io.File;
import org.dvb.dsmcc.*;

class DSMCCObjectExample
 implements AsynchronousLoadingEventListener,
 ObjectChangeEventListener {
 // Voting file object
  DSMCCObject votes;
  /**
   * Initialise the file object, set the
   * callback interfaces to 'this', and
   * initiate async load.
   */
  void init(){
    try{
      votes= new DSMCCObject("votes.txt");
      votes.asynchronousLoad(this);
    }catch(InvalidPathNameException e){
      e.printStackTrace();
    }
  }

  /**
   * Implement AsynchronousLoadingEventListener interface.
   * @param event notification.
   */
  public void receiveEvent(AsynchronousLoadingEvent event){
    if (event instanceof SuccessEvent){
      DSMCCObject aFile = (DSMCCObject)event.getSource();
      System.out.println("Votes loaded:" + event);
    } else {
      System.out.println("Async load failed: " + event);
    }
  }
```

3.3.3 Accessing Files – Return Channel

With return channel support, resources and zip files may be referenced by the use of an URL; they may then be loaded using HTTP. It is possible to use the broadcast channel as a means of bootstrapping other Java™ applications off the network – for this to be a desirable option, the applications would likely need to be relatively large and infrequently accessed, e.g. games etc.

Note: DVB-J (The Java platform defned as part of the MHP specifcation) prohibits run-time code extension from data sources less secure than the original program code. This is accomplished by preventing applications from directly subclassing `java.lang.ClassLoader`, and instead providing a secure subclass named DVBClassLoader.

3.3.4 AWT Restrictions

The key Java™ classes are as follows:

```
org.havi.ui
```

Although a detailed examination of the GUI APIs is beyond the scope of this paper, it is worth noting, that there are significant design constraints imposed by MHP. Firstly, the heavyweight widgets are not required. For developers familiar with richer OS environments, this may come as something of a surprise. There are no AWT buttons, no text entry fields, and no menus. The AWT graphics primitives such as `java.awt.Container` and `java.awt.Graphics` are present however.

For those wishing to use AWT widgets, there are two choices: either hand-code your own library, or use the (Home Audio Video Interoperability) HAVi controls. Although DVB have selected the HAVi user interface components for the MHP specification, product differentiation and presentation are key design issues – as a result it is likely that many developers and broadcasters will develop their own distinctive GUI libraries.

3.3.5 DVB-HTML

An alternative to custom AWT or HAVi controls is the use of DVB-HTML – optional in MHP 1.1, if present it may be used for all display and user interaction. Access to the underlying Java™ application may be accomplished via ECMAScript (aka JavaScript) via the ECMAScript-Java bridge. DVB-HTML provides a sub-set of XHTML with some DVB extensions. The intention is that it will provide the means to deliver 'super teletext' and some Internet support. DVB-HTML is intended to provide more than just a passive-browsing environment. The mapping of DSM-CC stream events to DOM events provides a means to synchronize or animate the browser with real time events.

3.4 Stage 4 – Voting - Using the Back Channel

The sequence is as follows:

- The user votes for a given candidate by selecting their name from a list
- The application establishes a socket connection and sends the relevant information.

3.4.1 Return Channel Network Usage

Two Java™ classes of particular interest are:

java.net.URLConnection
java.net.Socket

The MHP specification supports a sub-set of the java.net package. The intention here is that the STB is able to create a socket connection to a back-end server to send and receive information. Possibly one of the simplest means of utilising this facility is to provide an HTTP post through the use of a URLConnection class. The back-end server can then be implemented as a Web based J2EE™ application – or other. A plain Socket connection may used instead if required e.g. to exchange serialised objects.

3.4.2 Connection Support

The Java™ class of interest here is:

org.dvb.net.rc.RCInterface

Although the core Java™ libraries provide a TCP/IP return channel API, there are no standard packages within the Java™ API to support the ability to establish the physical connection over which this facility may be used. Also, as MHP does not specify how this return channel is to be physically connected, support must be given for a number of different mechanisms e.g. ISDN, dial-up modem, broadband cable etc. These connection types may be split into two distinct groups: permanently connected and those that require explicate connection. The MHP specification provides a means of identifying the connection type and the effective baud-rate – for asymmetric channels (e.g. V90 modems) the data rate coming into the MHP application is given. This information is useful for a number of reasons e.g. some options, including streaming media may be disabled for low-speed connections and enabled for high-speed etc.

DVB have defined a return channel connection manager to handle (possibly many) connection types. From the application's perspective the manager is implemented as a singleton. A return channel interface may be retrieved by providing the connection object e.g. URLConnection. The retrieved RCInterface object will provide the connection type and baud-rate.

3.4.3 Establishing a Return Channel Link

org.dvb.net.rc.ConnectionRCInterface
org.dvb.net.rc.ConnectionRCEvent

The ConnectionRCInterface models a connection based return channel network interface – e.g. a dial-up modem connection. A connection 'target' is specified in terms of the telephone number to dial, the user name and the password to use – there are variations, but essentially that's it. To track the progress of the connection the application may register for connection related events e.g. 'connection failed'.

The following code fragment illustrates the use of the org.dvb.net.rc package to establish a physical connection to a service hosting a given URL. Lines of particular interest are highlighted:

```
import java.io.*;
import java.net.*;
import org.dvb.net.rc.*;

class RCInterfaceManagerExample
  implements ConnectionListener{
  static String TEL_NUM = "123456789";
  static String SERVER_URL = "tvserver.com";
  static String USER_NAME = "user";
  static String PASSWD = "passwd";

public void establishConnection()
  throws UnknownHostException,
         PermissionDeniedException,
         IncompleteTargetException,
         IOException{

  // get return-channel manager instance.
  RCInterfaceManager rcManager=
    RCInterfaceManager.getInstance();
  // create back-end server address object
  InetAddress inet = InetAddress.getByName(SERVER_URL);
  // get return-channel interface to access server
  RCInterface rcInterface = rcManager.getInterface(inet);
  // what type of interface do we have for this connection?
  int rcType = rcInterface.getType();
  // check if we have a connectable interface
  if(rcInterface instanceof ConnectionRCInterface){
    // cast to a connectable interface
    ConnectionRCInterface connection =
      (ConnectionRCInterface)rcInterface;
    // add myself as a connection listener
    connection.addConnectionListener(this);
    // are we connected?
    if(connection.isConnected()){
      // yes, ok.
    } else {
      // no, assemble telephone number, login and password
      ConnectionParameters cp =
        new ConnectionParameters(TEL_NUM, USER_NAME, PASSWD);
      // inform the connection interface of the target
      connection.setTarget(cp);
      // now tell it to connect - this is an async call, the
      // callback handler will be informed.
      connection.connect();
    }
  } else {
    // Not a connectable interface, assune connected       }
}

  /**
   * Implement ConnectionListner
   * @param event for connectionChanged
   */
```

```
  int rate = -1;
  public void connectionChanged(ConnectionRCEvent event){
    if(event instanceof ConnectionTerminatedEvent){
      // Handle Connection Terminated.
    } else if(event instanceof ConnectionFailedEvent){
      // Handle Connection Failed.
    } else if (event instanceof ConnectionEstablishedEvent){
      // Handle Connection Etsablished.
      ConnectionRCInterface rcInterface =
        (ConnectionRCInterface)event.getSource();
      rate = rcInterface.getDataRate();
    } else {
      // unknow event!
    }
  }
}
```

3.5 Stage 5 - Updates and Events over the Broadcast Channel

The sequence is as follows:

- Votes accumulated at the back-end server are prepared for broadcast
- New polling status broadcast to STB
- STB receives update and informs application
- Application updates the TV display.

3.5.1 File Update Events

The following Java™ objects are of interest:

org.dvb.dsmcc.DSMCCObject
org.dvb.dsmcc.ObjectChangeEvent

There are a number of ways to achieve updates, possibly one of the simplest is to assemble the voting totals in a text file (possibly a property file), and submit the new file to the carousel for broadcast. Object Carousels support the notion of a version number for a file. Strictly speaking this version number is for the module used to transport the file. The `DSMCCObject` provides a means of not only registering for asynchronous loading events, but also for change events. The `ObjectChangeEvent` carries with it the current version and also the `DSMCCObject` that has changed. For applications sensitive to more than one file change, it may be advisable to place the files in separate modules for transport – otherwise unrelated code may be triggered unnecessarily.

The following code illustrates the use of `DSMCCObject` for file access and change notification.

Lines of particular interest are highlighted:

```
import java.io.File;
import org.dvb.dsmcc.*;

class DSMCCObjectExample
  implements
  AsynchronousLoadingEventListener,
  ObjectChangeEventListener {

  // Voting file object
  DSMCCObject votes;

  /**
   * Initialise the file object, set the call-back
   */
  void init(){
    try{
      votes = new DSMCCObject("votes.txt");
      votes.addObjectChangeEventListener(this);
    } catch(InvalidPathNameException e){
      e.printStackTrace();
    }
  }
  /**
   * Implement ObjectChangeEventListener interface.
   * @param changeEvent for file.
   */
  public void receiveObjectChangeEvent(ObjectChangeEvent
    changeEvent){
    int version = changeEvent.getNewVersionNumber();
    DSMCCObject changedFile =
     (DSMCCObject)changeEvent.getSource();
    if(votes.getURL().equals(changedFile.getURL())){
      updateScreenDisplay(changedFile);
    } else {
      System.out.println("File in same module:"+
        changeEvent);
    }
  }
}
```

3.5.2 Multicast and Datagram Sockets over Broadcast Channel

The following Java™ objects are of interest:

java.net.MulticastSocket
java.net.DatagramSocket

Where support for IP over the broadcast channel is included, the `java.net` classes: `MulticastSocket` and `DatagramSocket` are provided. Effectively this means that the MPEG-2 transport stream acts as a transport medium for IP packets. These classes could be used in much the same way as would be normally; e.g. each packet could contain an array integers representing the current vote tally.

Note that this support is optional in MHP 1.0.1. Interoperable applications could use his as an optimized mechanism for fast updates, but it is recommended that it be backed up by something else, e.g. carousel updates.

3.5.3 Stream Events

The following Java™ object is of interest:

org.dvb.dsmcc.DSMCCStreamEvent

Another possibility for obtaining updates and events is the use of `DSMCCStreamEvent`. These events only provide a small amount of data (as a byte array). As the event information includes a normal-play-time (NPT) indicator they are better suited to application signalling related to the audio-visual content.

3.6 Conclusion

MHP has at its core the Java TV™ specification. This API attempts to maintain existing concepts and re-

implement them for this new environment. E.g. files and Sockets. As the MHP specification grew this simple model has been supplemented, and in some cases replaced by other APIs. As an example, the original Java TV™ Carousel file package implements the Java™ File class and could be used in much the same way. DVB have replaced this package with the `DSMCCObject` which provides a much richer set of functionality relating more closely to the DSM-CC object carousel facilities. Although this does provide the developer with more flexibility in the design of the application, the resulting code risks being more closely coupled to the network signalling.

3.6.1 OO Model Degradation

The Java TV™ API model sought to extend the existing Java™ APIs by specialising classes and interfaces. The success of this approach is dependent on two key factors:

1. That the semantics of the original classes and interfaces are sufficient to represent their new extensions, and
2. The programming model and idioms are still valid in the new environment.

3.6.2 Liskov Substitution

The ability to extend a class to fit a new problem is a major attraction of OO languages. A difficulty arises when only some methods of the class or interface may be implemented and the remaining methods must throw a‘not implemented’ exception. This violation of Liskov Substitution is not uncommon in hierarchies that have been extended over time to accommodate new problem domains. Although regarded as bad practice, it must be weighed against the benefits in each case. Often these benefits are not the result of saving the developer a day or so in re-implementing code, but rather the ability to use the class in other already available (closed) libraries i.e. where the base class or interface is passed into a method. This not only allows the developer to leverage existing libraries for manipulation, but also allows sub-systems to be easily tested in a convenient environment that does not support the extended concrete classes. Where the convenience of extension is outweighed by the degradation of the class hierarchy, the class should be dropped and a new solution found.

The MHP API has already changed to the extent that it does not support the Java TV™ Carousel API.

3.6.3 Programming Models and Idioms

Java™ technology provides language level support for threads. This fosters a programming style that relies heavily on threads as a means of managing access to resources that are not immediately available e.g. reading data from a Socket. Embedded systems often use an event driven model as a means of notifying the application of incoming data or state transitions. Although more difficult to manage, event models use substantially less memory than threads. The publish/subscribe pattern figures prominently in MHP as a means of managing this event model. As shown above: the DSM-CC object carousel API favours event notifications over ‘thread blocking’ to indicate the success or failure of a file loading attempt. The return connection package supports event notification to indicate the success or failure of a connection attempt.

3.6.4 Re-purposing and Re-use

To the extent that MHP has moved away from established Java™ APIs and application development, the ability to re-use and re-purpose existing objects and frameworks becomes can be more difficult. The size and complexity of MHP requires developers that have specialist domain knowledge along with a solid grounding in traditional Java™ development.

4 References

Digital Video Broadcasting (DVB) (2001): Multimedia Home Platform (MHP) Specification 1.1. European Telecommunications Standards Institute 2001. TS102 812. (v1.1.1)

Java TV™ Web Site.
http://java.sun.com/products/javatv/index.html

MHP Organization Web Site.
http://www.mhp.org

DVB Organization Web Site.
http://www.dvb.org

DAVIC Web Site.
http://www.davic.org

HAVi.
http://www.havi.org

•