© 2001 IEEE. Reprinted, with permission, from Proceedings of the *IEEE International Conference on Multimedia and Expo 2001*, Tokyo, Japan, August 22-25, 2001, pp. 685-688.

DIGITAL TELEVISION APPLICATION MANAGER

C. Peng and P. Vuorimaa

Telecommunications Software and Multimedia Laboratory,
Helsinki University of Technology
P.O. Box 5400, FI-02015 HUT, Finland.
pcy@tml.hut.fi and petri.vuorimaa@hut.fi

ABSTRACT

The future digital television will provide viewers new interactive services, e.g., Electronic Programming Guide, TV shopping, video-on-demand, etc. The set-top box needs a middleware to launch these applications before interaction and collect garbage after quitting or switching to another service. It is the application manager's responsibility to manage the lifecycles of the services. Thus, the application manager plays a crucial role in controlling the applications and efficiently utilizing the limited resources of set-top box. This paper presents a design of an application manager, including a lifecycle model of an application and a communication protocol between an application and the application manager. Also, the Application Information Table (AIT), which carries the essential application signaling information in a transport stream is introduced. Finally, the implementation, which uses Java technology, is described in detail.

1. INTRODUCTION

Our work followed the Digital Video Broadcasting-Multimedia Home Platform (DVB-MHP) standard. It is a common platform for user to transparently access a range of multimedia services. It includes software architecture and hardware devices. Its hardware devices consist of the home terminal (e.g., set-top box, TV, and PC), its peripherals, and the in-home digital network [1]. Its system software includes a Real-time operating system, an interactive engine, a virtual machine, the libraries or Application Programming Interface, the databases, navigator, an application manager, etc.

The DVB-MHP defines an application as a functional implementation of an interactive service [2]. A DVB-MHP application can be categorized either as DVB-Java or DVB-HTML application. They all run in set-top box [3]. In this paper, the application manager manages only DVB-Java applications. Each application has a lifecycle (i.e., the sequence of steps by which an application is initialized, undergoes various state changes, and is eventually destroyed) [4]. Such DVB-Java applications are called *Xlet* applications. An *Xlet* is either resident in the set-top box or downloaded from object and data carousels or network and controlled by the application manager.

An application manager is a part of system software and residents in set-top box. The primary purpose of an application manager is to signal the state changes of an *Xlet* and bridge the gap for an *Xlet* to access set-top box resources. The application manager defines an application lifecycle model and a communication protocol between *Xlet* and the application manager.

The application manager is responsible for managing the application lifecycle, including checking the code and integrity, synchronizing the commands and information, obtaining and disposing the system resources, managing the error signaling and exceptions, initiating and terminating any new sessions, allowing the sharing of variables and contents, concluding in an orderly and clean fashion, and adapting the presentation graphic format to suit the platform display [1].

A DVB-Java application (i.e. *Xlet*) is actually a set of Java classes that operate together and need to be signaled as a single instance to the application manager so that it can control its state changes. Applications can be launched automatically via broadcast signaling or via Navigator controlled by viewers from a remote control. One critical design requirement is low start-up latency of each *Xlet* [3].

All the information of downloadable applications is stored in an AIT table, which is multiplexed and transmitted together with other elementary streams in MPEG-2 transport stream [3] [5]. The application manager needs this information to identify the locations and signaling information of the applications. All the signaling information of resident applications is stored in a system configuration file after they are downloaded or updated.

The AIT, Java classes and data associated with the applications are stored in either object and data carousels or delivered by IP via DVB multi-protocol encapsulation. Object carousels convey hierarchical file structures using MPEG-2 Digital Storage Media Command and Control (DSM-CC) User-to-User File and Directory objects over data carousel. Then, data carousel cyclically transmits data modules over broadcasting network.

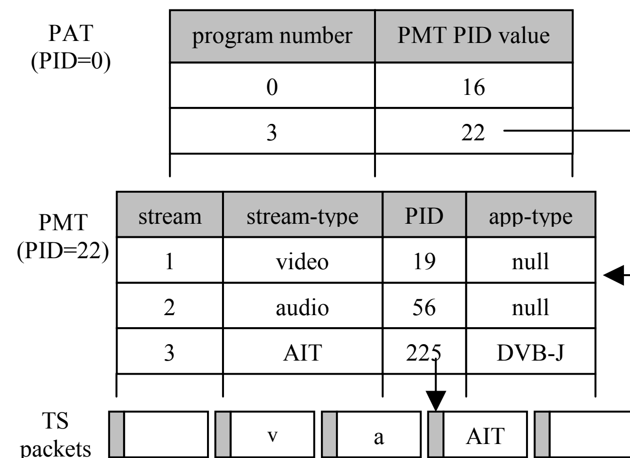

Figure 1. The procedure for application manager to identify AIT.

ISBN: 0-7695-1198-8 © 2001 IEEE

2. TRANSPORT STREAM AND AIT

2.1 Identifying AIT

The application manager is responsible for identifying AIT carried in transport stream packets. The signaling information contained in AIT is then used for managing an *Xlet* application. Figure 1 shows an example procedure for application manager to identify AIT in a MPEG-2 transport stream.

The Program Association Table (PAT) can be found from the transport stream packets with PID 0 [6]. Suppose that the AIT of an application is associated with program 3. Therefore, the Program Map Table (PMT) can be identified in the PID 22 packets. In PMT, if the field of stream-type indicates 0x05 (i.e., AIT), the PID 225 packets will carry the corresponding AIT. The AIT also carries *application_type* information (cf. Table 1) like in the PMT table. The two values will agree if the *AIT_version_number* is the same as the field carried in AIT. This information can be used to detect the change of the application version.

2.2 Description of AIT

The AIT has the fields of *application_type*, *version_number*, *application_control_code*, *common descriptor*, and *application information descriptor*. Table 1 lists the main fields and corresponding data of AIT used in the application manager.

Field	Value	Descriptions
Application_type	DVB-J	Java application
Application_control_code	0x02	Request by the viewer
protocol_id	0x0002	Transport via IP
Transport_protocol_label	http	Http access
URL_byte	tml.hut.fi	Location of classes
ISO_639_language_code	eng	English language
application_name_char	Screen Info	Application name
icon_locator_byte	/data/	Location of icon file
icon_flags	0000 0000 0000 0010	32x32 broadcast pixels on 4:3 display
base_directory_byte	/~pcy/	Directory name in the server
classpath_extension_byte	/hockey/	Path of classes
Initial_class_byte	ScreenInfo	Name of implementing Xlet

Table 1. The Sample AIT used in application manager.

In Table 1, *Application_control_code* value 0x02 means that the application will be activated by the viewer via remote control. *Icon_locator_byte* is the path relative to the base directory of the application classes in a server or object carousel. The *ISO_639_language_code* field indicates the broadcasting language of the TV program. The *protocol_id* field indicates that an application shall be downloaded via broadcasting network or IP. The *Initial_class_byte* conveys the starting Java class name used by the application manager to execute the *Xlet* application.

3. LIFECYCLE MODEL AND PROTOCOL

3.1 Application Lifecycle Model

A state machine diagram, which is used to model an application (or *Xlet*) lifecycle, is shown in Figure 2. In the lifecycle model, an *Xlet* application has four lifecycle states (i.e., *loaded*, *paused*, *running,* and *dead*).

When the initial Java class of an application is loaded and instantiated from data carousel or from set-top box, it enters the *loaded* state. After that, the application manager signals the *Xlet* to initialize itself. The *Xlet* enters the *paused* state. An *Xlet* in paused state is ready to run. The *running* state indicates that the application is executing. The running application can be frozen from *running* to *paused* state. An application in *paused* state can resume its execution. An *Xlet* can be terminated at any time and it changes into the *dead* state.

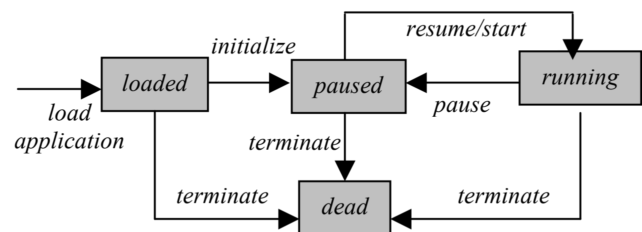

Figure 2. Application lifecycle model.

3.2 Communication Protocol

Each *Xlet* application cannot run by itself without the application manager. It is necessary to define a protocol between an *Xlet* and the application manager. Figure 3 shows a communication protocol between an *Xlet* and set-top box environment.

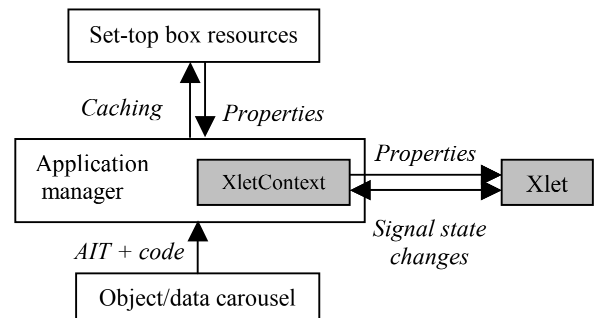

Figure 3. Protocol between an *Xlet* application and set-top box environment.

For downloadable *Xlets*, the code and the associated AIT can be loaded into set-top box memory from data carousel and initialized by the application manager. The application manager is able to control the state changes of *Xlet* and terminate an *Xlet* at any time via *XletContext* interface passed to the *Xlet*. The Xlet can also change its own state and get set-top box resources passed by the *XletContext*.

The *Xlet* signals its state change to the application manager using the callback mechanism. Whenever *Xlet* changes its state, the application manager must be notified about the state change so that it can track the state of the *Xlet* and maintain a data record (cf.

ISBN: 0-7695-1198-8 © 2001 IEEE

Table 2) for the purpose of resources sharing and limiting the number of simultaneous applications. An application manager cannot force an *Xlet* to provide its service. It means that, when an *Xlet* is in *running* state, all the functionality is provided by the *Xlet* itself. An *Xlet* can only be activated via broadcasting and viewer's request (cf. Figure 3 and Section 4.1).

4. JAVA IMPLEMENTATION

The application manager is implementation-dependent. In our system, the application manager was implemented as a runnable thread so that it can run concurrently with all the other threads in the system. It is always active when the television is on and preparing to run services. The application manager has a dynamic priority. It depends on the interaction, e.g., if no application is running, it has the lowest priority.

The *Xlet* and *XletContext* were designed and implemented as interfaces as specified in DVB-MHP [3] and Java TV API [4]. The interfaces defined the protocols of behavior that can be implemented by any applications anywhere to ensure the interoperability of applications.

It is reasonable to limit the number of applications that can be presented simultaneously. The applications can be presented and executed concurrently. Selecting one application may result in stopping running another (i.e., paused) one, but the classes resident in set-top box's memory may still exist until they are signaled for terminating permanently.

4.1 *XletContext* and *Xlet* Interfaces

The two interfaces define a set of abstract methods, but do not implement them. The application manager and applications implement the *XletContext* and *Xlet* interfaces, respectively. They agree to certain behavior as defined in the set of methods. The application manager and all the applications must include the class code together with their other part of class code. The *XletContext* acts as a communication bridge between the *Xlet* and the application manager (cf. Figure 3). An *XletContext* object is passed to the *Xlet* at the stage of *Xlet* initialization to permit *Xlet* accessing set-top box resources.

The *Xlet* interface defines four methods that must be implemented in each application. The methods include *initXlet()*, *startXlet()*, *pauseXlet()*, and *destroyXlet()*. It allows the application manager to create, initialize, start, pause, and destroy an *Xlet*. Each *Xlet* application implements these methods to update its internal activities and resource usage as directed by the application manager. The *XletContext* interface defines four methods that must be implemented by the application manager. It includes *getXletProperties()*, *resumeRequest()*, *notifyPaused()*, and *notifyDestroyed()*. This interface provides each *Xlet* application with a mechanism to retrieve properties and a way to signal internal state changes to the application manager.

4.2 Functionality of the Application Manager

The application manager consists of some functions as well as the functions contained in the *XletContext* methods. One of these functions includes caching the applications information. That is, the application manager maintains a hash table that is used to monitor the state changes and resume execution of *Xlet*s running in set-top box. Table 2 shows the table cached by the application manager during system running. The *Xlet*s, i.e., Ice Hockey [7], Navigator [8], and Teletext [9], are three resident applications which were developed in the Future TV project.

Xlet name	Class loader	Xlet object	Initial class	State
Ice Hockey	loader1	xlet1	ScreenInfo	running
Navigator	loader2	xlet2	NaviApp	paused
Teletext	loader3	xlet3	TextTVApp	paused

Table 2. Caching table used by the application manager.

Another function is to identify the application information from the AIT. When the application manager receives the viewer's request to start the application during watching the program, it creates the data in the above table (the first entry). This procedure includes decoding transport stream to get application information (e.g., location of classes, etc.) from the AIT and save them in the system configuration file.

Another function is managing remote control key events. The application manager and applications have their own key listeners. When a new application starts, the current key listener must be removed and a new one is added. Other functions include calculating and signaling the available memory, handling errors occurred during system execution, etc.

4.3 ClassLoader and Class

The key java technology to develop the application manager is to use *java.lang.Class* and extend *java.lang.ClassLoader*.

If several applications are executed simultaneously, the application manager must cope with the problem of name collisions, which are caused by package naming scheme. Java Virtual Machine handles these problems through its class loader architecture known as namespace mechanism. Every class in an application is loaded by an associated *ClassLoader* object. The Java Virtual Machine treats classes loaded by different class loaders as entirely different types [10].

When the application manager is running, the *java.lang.Class* can dynamically load additional classes as extension module. Class's *newInstance()* is used to create new instance of the class. The classes can be loaded by initial class name into the application manager. Classes are introduced into the java environment when they are referenced by name in a class that is already running (i.e., application manager).

Two class loaders were defined in our system (i.e., *FileClassLoader* and *URLClassLoader*), which were extended from *java.lang.ClassLoader*. *FileClassLoader* can load classes from local class files (e.g., Digital Teletext service and Navigator application use this class loader). *URLClassLoader* can load classes over the internet (cf. Section 4.4 Ice Hockey application). Both class loaders override the abstract method *loadClassBytes()* The difference between *FileClassLoader* and *URLClassLoader* is the source of classes located. The *loadClassBytes()* method in *URLClassLoader* uses *java.net.URLConnection* to get input stream (class bytecodes). While *FileClassLoader* uses *java.io.FileInputStream* to read bytecodes.

The most important aspect of designing the two class loaders is to implement the abstract method *loadClass()* inherited from *ClassLoader* in correct order. The first step is to check if the primordial class loader can resolve this class name. All Java

ISBN: 0-7695-1198-8 © 2001 IEEE

Virtual Machines include a primordial class loader that is embedded in the virtual machine [10]. Then, *loadClassBytes()* method try to load the class from the source (loal file or URL). The next step is to call the *defineClass()* method to verify that the class bytes are legal Java class file. The final step is to call *resolveClass()* method, which causes any classes that are referenced by this class explicitly to be loaded and a prototype object for this class to be created.

4.4 Running System

The application manager is started up automatically after set-top box is powered up and keeps running until the set-top box is powered down.

Figure 4. Ice hockey application.

Figure 4 shows a screenshot of the running system. The upper left image is an Ice Hockey application (an *Xlet*), which was launched by the application manager as a viewer's request. It used the AIT table shown in Table 1. The application manager maintained its data record as shown in Table 2. The DOS prompt on the right displayed the results of classes loaded, when the application was in *loaded* state.

The software test platform was Windows 98 and JDK 1.3. The application classes were downloaded dynamically from http://www.tml.hut.fi/~pcy/hockey/. Table 3 lists the start-up and switching latencies of three *Xlet* applications. The results show that the starting time of each application via the application manager is fast. The latencies are caused by initializing the *Xlet*s.

5. CONCLUSIONS

In this paper, we have presented a novel design of a basic digital television application manager. In particular, a well-defined state machine (i.e., application lifecycle model) and a protocol between *Xlet* and the application manager were described, and the AIT for identifying the source of the code and data and application signaling information was introduced. To achieve these design goals, the Java class loader and class technology of Java Virtual Machine were used to load application classes from different sources and solve name collisions.

In addition, Java thread, key event, and garbage collection mechanisms were used. Our work demonstrated the possibility of the proposed application lifecycle model and the protocol, and flexibility of implementation. It is a valuable basis for future set-top box manufacturers to develop their own system software.

The *Xlet* state machine should be designed to ensure that the behavior of an *Xlet* is as close as possible to the behavior of television viewers expect, especially the perceived startup latency of an *Xlet* should be very short. We didn't give performance evaluation about the start-up latency of the application manager and *Xlet* since it depends so much on the set-top box environment like operating system and memory, etc. This would be very important. Our following work will focus on overall system testing and performance evaluation.

Xlet	Start-up/Switching time (second)	
IceHockey	4	(start-up)
Navigator	6	(start-up)
Digital Teletext	8	(start-up)
IceHoceky - Navigator	4	(switching)
IceHockey – Teletext	8	(switching)
Navigator - IceHockey	4	(switching)
Navigator - Teletext	7	(switching)
Teletext - IceHockey	4	(switching)
Teletext - Navigator	4	(switching)

Table 3. Start-up and switching latencies.

6. ACKNOWLEDGEMENTS

The Future TV project (http://futuretv.uta.fi) is funded by the National Technology Agency of Finland together with major Finnish television, telecommunications, and digital media companies. The author C. Peng would like to thank Nokia Oyj foundation for the support during the research work.

7. REFERENCES

[1] Evain J. - P. "The Multimedia Home Platform – an overview". *EBU Technical Review*, spring 1998.

[2] Gerard O. *The Essential Guide to Digital Set-top Boxes and Interactive TV*. Prentice Hall PTR, 2000.

[3] DVB-TAM232. "Multimedia Home Platform". *European Broadcasting Union*, February 2000, pp.101-120.

[4] Bart C., Jon C., Bill F., Linda K., David R., James V., and TaoY. "Java TV API Technical Overview". *The Java TV API Whitepaper Version 1.0, Release Candidate D*, July 11, 2000.

[5] Sarginson P.A. MPEG-2: Overview of the systems layer, *Research and Development Report*, No. 1996/2.

[6] Jeffery O. "A Rational approach to testing MPEG-2". *IEEE SPECTRUM*, May 1997.

[7] Peng C. and Vuorimaa P. "Development of Java User Interface for Digital Television". *Computer Graphics, Visualization, and Interactive Digital Media, WSCG'2000*, Czech Republic, Feb. 7-10, 2000.

[8] Peng C. and Vuorimaa P. "A Digital Television Navigator". *Proceedings of the Internet and Multimedia Systems and Applications, IMSA'2000*, Las Vegas, USA, Nov. 19-23, 2000, pp. 69-74.

[9] Peng C. and Vuorimaa P. "Digital Teletext Service". *Proceedings of Computer Graphics, Visualization and Computer Vision, WSCG'2001*, Czech Republic, February 5 - 9, 2001, pp. 120-125.

[10] Lindholm T. and Yellin F. The Java Virtual Machine Specification. *Sun Microsystems, Inc.*, 1999.

ISBN: 0-7695-1198-8 © 2001 IEEE