System Aspects of TV-Anytime Metadata Codec in a Uni-directional Broadcasting Environment

Minje Kim, Minsik Park, Seung-jun Yang, Ji Hoon Choi and Han-kyu Lee

Abstract — *Digital broadcasting environment in which digitalized AV contents can be sent has a room for data transmission as well. In this circumstance, effective encoding method of verbose XML type TV-Anytime metadata becomes important and also the broadcasting environment-specific refinement of TV-Anytime transportation specification is in great need. This paper suggests a well-tuned framework for encoding TV-Anytime metadata. Our TV-Anytime codec, the main subject of this paper, includes revision of the whole TVA codec system specification and external programming interfaces to communicate with the external modules in our uni-directional broadcasting head-end system.*

Index Terms — TV-Anytime, Digital Broadcasting, Personalized Broadcasting, MPEG-7

I. INTRODUCTION

TV-Anytime (TVA) [1] is becoming more popular in the digital broadcasting area because of its powerful description ability about multimedia contents. Its abundance causes the significant broadcasting data services, such as Electronic Contents Guide (ECG), content searching and acquiring, to be more powerful. Moreover, advanced PVR functions are possible using some of the content description metadata [2-4] and even interactive services are available using TVA phase 2 metadata [5] with benefit of bi-directional communication networks.

In the digital broadcasting environment, now it is possible to use broad bandwidth of IP network to send a bulk of TVA metadata. However, it is not guaranteed that every terminal is tapped into the network so that the service providers should be prepared to serve their metadata in uni-directional way to the consumers who are in those conditions.

TVA suggests a systematic way to convey metadata in a uni-directional environment [6]. It includes specific information about encoding, carrying, encapsulating, and managing fragmented TVA metadata. [1]We were eager to refine this architecture and adapt it to our transmission system, and finally, resulted in some issues to be decided when realizing this system.

This paper provides detailed realistic suggestions about several components which compound TVA codec. TVA codec works as a metadata encoding block of the metadata management system used in the transmission head end of our *Personalized Broadcasting* system for digital cable broadcasting. Section II introduces the refined TVA codec specification, including MPEG-7 systems, especially revised TeM followed by section III, which offers specification of programming interfaces to be used to communicate with metadata management system. Section IV concludes our works and discusses some future issues.

[1] Acknowledgement: This work was supported by the IT R&D program of MIC/IITA.[2006-S082-01, Development of digital multimedia broadcasting technology for personalized broadcasting]

II. TVA CODEC REFINEMENT

This section covers the specification of the proposed whole encoding system based on uni-directional TVA metadata encoding system [6]. TVA recommends profiled MPEG-7 BiM [7] with some restriction and we adopted it thoroughly, but its lack of regarding another possibility of compression method made us to revise MPEG-7 TeM technology and draw it into our system. We adopted not only TVA-profiled BiM as one of main metadata compression tool, but the *textual fragment encoding*, a revised version of MPEG-7 TeM, as another one. Additionally, this fact led us chain revisions of the entire system such as additional structure type, initialization message, encapsulation structure, etc.

A. Structure Type

TVA defines *container* as an atomic unit of grouping and sending fragmented metadata with related parameters. TVA metadata container is a top-level access unit in the uni-directional environment. For instance, in the MPEG-2 broadcasting environment, TVA metadata containers are able to be inserted into the data or object carousel.

TVA metadata containers can be classified as two groups, data and index containers, based on the type of content in the container. This and following several subsections, however, focus on the specification of *data containers* in detail.

Multiple structures can be involved in a certain container. They can be distinguished and pointed by *container_header* at the first place. Table I specifies *structure_type* which is an 8 bit field in the *container_header* identifying the type of structure in the same container. Most structure types in Table I have the same meanings with original TVA definition except following ones.

- *encapsulation*: This structure type acts as a header of subsequent *data_repository* structure. See section II.C for more detail.
- *data_repository*: This structure type wraps a number of

TABLE I
STRUCTURE TYPE ASSIGNMENT

Value	Description
0x00	Reserved
0x01	Encapsulation
0x02	Data Repository
0x03	Index List
0x04	Index
0x05	Multi field Sub Index
0x06	Fragment Locators
0x07	Moved Fragment
0x08 – 0xDF	TVA Reserved
0xE0	TVA Init Message
0xE1 – 0xFF	User Defined

TVA fragments, which are compressed with designated methods (e.g., BiM or textual encoding). See section II.D for more detail.

- *TVA Init Message*: This structure type conveys revised version of TVA-init message as a structure. See section II.B for more detail.

In the *container_header* there is a field, *structure_id*, which is dedicated to identify multiple occurrences or data type of a certain structure type. At first, we used a reserved value 0x02 for identifying the data repository encoded by the *textual fragment encoding* method. The other additional structure type, *TVA Init Message*, occurs only once in a fragment stream so that its *structure_id* has no meanings.

B. *TVA Init Message*

TVA does not specify the way to convey TVA-init message, but it should be ready for the terminal before starting to decode any fragment. In order to meet those needs, we define the *TVA Init Message* structure type and assign a certain exclusive *container_id* value, zero, to the container which has *TVA Init Message* structure in it. From now, we will refer this type of container as *TVA Main Container*.

Table II shows revised definition of TVA-init message. In our system TVA-init message occupies a structure of *TVA Init Message* type in the *TVA Main Container* with some revised fields. At first, we manipulated user defined values of *EncodingVersion* field. (See Table III.) The values 0x10 and 0x11 are indicators for *textual fragment encoding*, which mean the fragments in the stream are compressed by GZip or not, respectively.

Another additional field is *CharacterEncoding* field. In the TVA standard framework, if the *EncodingVersion* field has a meaningful value, such as 0x01, *CharacterEncoding* field conveys the character encoding scheme for all textual data used within the TVA metadata fragment stream. The TVA standard framework, however, does not consider another possibility of encoding version and we added one, so that a supplementary field for our *textual fragment encoding* is needed. This fact is represented as a conditional operation when the *EncodingVersion* equals that textual case.

TABLE II
DEFINITION OF TVA INIT MESSAGE

Syntax	No. of Bits	Mnemonic
TVA-init {		
EncodingVersion	8	uimsbf
IndexingFlag	1	bslbf
reserved	7	
DecoderInitptr	8	bslbf
if(EncodingVersion == ‘0x01’) {		
BufferSizeFlag	1	bslbf
PositionCodeFlag	1	bslbf
reserved	6	
CharacterEncoding	8	uimsbf
if (BufferSizeFlag == ’1’) {		
BufferSize	24	uimsbf
}		
}		
if(IndexingFlag) {		
IndexingVersion	8	uimsbf
}		
if(EncodingVersion == ‘0x10’ || EncodingVersion == ‘0x11’) {		
CharacterEncoding	8	uimsbf
}		
Reserved	0 or 8+	bslbf
DecoderInit( )		
}		

TABLE III
VALUES FOR THE ENCODINGVERSION PARAMETER

Value	Description
0x00	Reserved
0x01	TVA MPEG_7 profile (BiM) ISO/IEC 15938-1
0x02 – 0x0F	TVA reserved
0x10	GZip encoded
0x11	No Encoding i.e. raw XML Index
0x12 – 0xFF	User defined

DecoderInit() element is substituted by a proper format of *DecoderInit* message which is dependant on the encoding method specified in the *EncodingVersion* field. In the case of BiM (*EncodingVersion* = 0x01) we take the form as specified in [7] with TVA profile. If the *EncodingVersion* is indicating the case of *textual fragment encoding*, namely 0x10 or 0x11, the system replaces the *DecoderInit()* element with specially designed *DecorderInit* message. Table IV shows the detail of our *Textual_DecoderInit* message which is used to initialize the fragment stream whose data are encoded with *textual fragment encoding.*

We borrowed this *Textual_DecoderInit* message structure from [8]. The *fragment_type_identifier* notifies decoders of the fragment types used in the system along with their XPath information and binary identifiers. DVB’s ESG is defining its own fragment types, but we take the basic TVA fragment types instead of them. Service provider assigns zero to the *num_fragment_types* field in order to inform decoder when there is no use of user-defined fragment so that the fragment type notification is not needed.

The additional *fragment_type_identifiers* should take values starting from 0x0022 to 0xFFFE since the other ones

TABLE IV
DEFINITION OF TEXTUAL DECODER INIT MESSAGE

Syntax	No. of Bits	Mnemonic
Textual_DecoderInit () {		
version	8	uimsbf
length	8+	vluimsbf8
num_namespace_prefixes	8	uimsbf
for(i=0; i<num_namespace_prefixes; i++) {		
prefix_string_ptr	16	uimsbf
namespace_URI_ptr	16	uimsbf
}		
num_fragment_types	16	uimsbf
for(i=0; i<num_fragment_types; i++) {		
xpath_ptr	16	uimsbf
fragment_type_identifier	16	uimsbf
}		
}		

are already occupied for TVA basic fragment types, such as 0x0001 representing ProgramInformation fragment type. These pre-defined identifier values for basic fragment types are originally used for the index container. (See table 7 in [6].)

The above TVA-init message definition is sent as a structure of data container, *TVA Main Container*, with the structure type value 0xE0. *TVA Main Container* also contains several other *data_repository* type structures. The first *data_repository* contains strings which are pointed by certain pointer fields in the *TVA Init Message.* For example, *xpath_ptr* field in *Textual_DecoderInit* message is a byte offset from the start of the string type first *data_repository* to the XPath string related with the fragment type of current loop. In this case, this *data_repository* has *structure_id* equal to 0x00, which means *string_repository*. String type data repository is originally used in the index container, but we borrowed it for our use along with basic *fragment_identifier* values. Another *data_repository* occurs in the *TVAMain Container* when there is a TVAMain fragment to be sent. In this optional case, *structure_id* indicates the encoding type of this fragment.

C. Encapsulation

TVA provides *encapsulation* structure which explains subsequent *data_repository. Encapsulation* structure takes a hybrid form of a header followed by iterative entries. *Encapsulation_header* designates specific reference format of following fragments. TVA allocated 0x00 as reference format of BiM encoded fragment and we additionally assigned a user-defined value 0xE1 for the *textual_fragment_encoding.*

Each *encapsulation_entry* contains reference, version and id of related fragment. *Fragment_reference()* element is an offset in bytes from the start of the *data_repository* to the first byte of referred fragment. Of course there is a possibility that the *data_repository* is *textual fragment encoding* format, but this offset-based reference format remains consistent in that case.

D. Data Repository

Data_repository is a structure type for sending actual fragment data or key values, but we do not cover the *string_repository* case for indexing key values in this paper (another usage of *string_repository* has been already described in the former clause).

Structure_id field in the *container_header* decides which type of data the *data_repository* carries. Table V defines the syntax of *data_repository* including the additional possibility of *textual_fragment_repository* encoded with the *textual fragment encoding* scheme, which will be described below.

All fields in the *data_repository* follow TVA specification, but *textual fragment repository* needs a particular definition like in Table VI and VII. Table VI shows the iterative form of *data_repository* and each of its fragment entry takes the fields in Table VII, which have following meanings:

- *fragment_type*: The fragment type identifier value of each fragment entry. The identifier values could have been already defined by TVA as basic ones or additionally defined by service provider in *Textual_DecoderInit.*
- *data_length*: This field inform decoders of the length

TABLE V
DEFINITION OF DATA REPOSITORY

Value
data_repository () {
if (structure_id == 0x00) {
string_repository()
}
else if (structure_id == 0x01) {
binary_repository()
}
else if (structure_id == 0x02) {
textual_fragment_repository()
}
else {
user_defined_data_structure()
}
}

TABLE VI
DEFINITION OF TEXTUAL FRAGMENT REPOSITORY

Syntax
textual_fragment_repository () {
for(i=0; i<fragment_count; i++) {
textual_fragment()
}
}

TABLE VII
DEFINITION OF TEXTUAL FRAGMENT

Syntax	No. of Bits	Mnemonic
textual_fragment () {		
fragment_type	16	uimsbf
Data_length	8+	Vluimsbf8
for(i=0; i<Data_length; i++) {		
data_byte[i]		
}		
}		

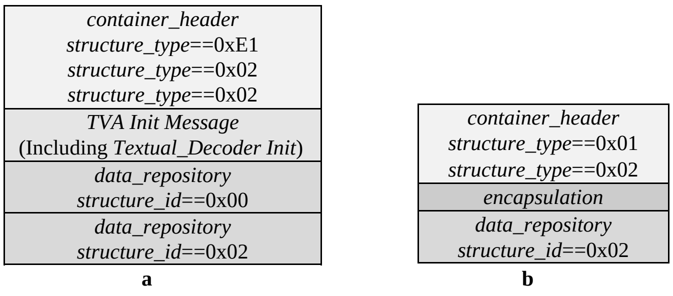

Fig. 1. Example of structure organization of data containers: a – a TVAMain Container, b – a general data container

of following fragment data in byte.

- *data_byte*: XML type TVA metadata fragment which might be compressed by GZip or not.

E. Metadata Container Example

The specific scheme to send metadata container is beyond the coverage of this paper, but the there are some points associated with our topic. In our system metadata containers are transported by profiled object carousel and the naming of container file is tightly coupled with the container id. We also defined several extensions to indicate the type of container, e.g., ".d" for data container file, ".i" for index container file and ".b" for the mixed one.

Fig. 1 shows simplified organizations of a *TVAMain Container* and a general data container to give you an idea about the constitution of related structures. One can catch up our detailed specification of *textual fragment encoding* and related revision of TVA framework by looking at Fig. 2.

Note that there is no command field in the container that denotes what the decoder does with a certain fragment. In our policy, decoder decides to add the fragment if its id is a brand new one. If the decoder already has a fragment which has the same id with the one of in the input stream, decoder should compare their version to decide whether ignore or replace it. What if a fragment in the decoder is no longer transmitted? Then the decoder abandons it.

III. User Interfaces

There are several interfaces which are used to enter our TVA codec. Next several clauses categorize those interface methods. We designed a C++ class, *TVAManager*, which is responsible for interfacing with external modules. All the interface methods are members of an object instance of *TVAManager* class. This object instance is an access points for several TVA metadata services. Metadata service concept is caused by the needs of service provider who will want to separate their metadata into several groups based on their goal of metadata service. Our system introduced this concept with its identification scheme.

A. Initializing Interfaces

- *TVAManager(int NumberOfServices, int *ServiceIDs, int *InitialContainerNums, int MaxContainerSize)*: This is a constructor method of the *TVAManager* class which initializes the number of metadata services and their IDs, initial blank container numbers, maximum size of those containers.
- *void SetTVAInit(int seviceID,TVAInitDataType TVAInit, TextualDecoderInitDataType TextualDecoderInit)*: This method delivers several parameters to generate *TVA Init Message* structure. *TVAInit* has essential fields to teach TVA codec which encoding version and character encoding to use, whether to use index or not, for example. The next parameter contains fields used to construct *Textual_DecoderInit()*. *SetTVAInit* method is overridden by two other versions. User can use *BiMDecoderInit* instead of *TextualDecoderInit* when the *encoding_version* field of *TVAInit* represents that this metadata service is compressed by MPEG-7 BiM. Even the last parameter is allowed to be omitted if the user just wants to use default setting. Note that user have to designates which specific metadata service will be initialized with this method by assigning proper value to the *serviceID* parameter.

B. Generating Interfaces

- *void CreateTVAMainContainer(int serviceID)*: This method actually generates a *TVA Main Container* file for a particular metadata service.
- *void CreateContainer(int numberOfFragment, string payloadFileName, bool *command, int *contextValue, int *fragmentID, int *fragmentVersion, int *serviceID, int *containerID, int *containerVersion)*: This method creates or edit general data container. Essential input from external modules is usually a bulk of fragments, which are segmented (not valid) XML document. The second parameter, *payloadFileName*, provides path information of the input fragments file. Each fragment in the input file has its corresponding information and the other parameters give this information by array type. *command* carries an array whose *i*-th element describes operation type of *i*-th fragment in the container file (e.g., add, replace, and delete). *contextValue* array tells fragment type identifier values of every fragment. *fragmentID*, *fragmentVersion* and corresponding metadata service id information is also passed on. *containerID* and *containerVersion* parameters are needed when the external module has current distribution of all the fragments over containers and wants to designate the container level location of fragments. The external caller can always maintain this mapping information in all the cases since the acquiring interface (see clause III.C), which has to be called successively right after the calling of *createContainer*, returns allocation table of fragments. Note that the first call of this method omits the last two parameters.

C. Acquiring Interface

- *ServiceInfoType GetContainers (int serviceID)*: This method returns mapping information between generated

Internal features of container		Internal features of structure		Values of structure fields
container header		structure_type		0x01(encapsulation)
		structure_id		XXX
		structure_ptr		XXX
		structure_length		XXX
		structure_type		0x02(data_repository)
		structure_id		XXX
		structure_ptr		XXX
		structure_length		XXX
encapsulation structure	encapsulation header	reserved_other_use		XXX
		Reserved		XXX
		fragment_reference_format		0xE1 (Reference to a Textual Fragment)
	encapsulation entry	fragment reference()	textual_fragment_ptr	offset of the first fragment
		fragment_version		XXX
		fragment_id		XXX
	encapsulation entry	fragment reference()	textual_fragment_ptr	offset of the second fragment
		fragment_version		XXX
		fragment_id		XXX
Data Repository	Textual Fragment	fragment_type		0x0001 (ProgramInformation)
		data_length		XXX
		data_bytes		XXX
	Textual Fragment	fragment_type		0x0002 (GroupInformation)
		data_length		XXX
		data_bytes		XXX

Fig. 2. Example of data container having two fragments

containers and fragments for a specific metadata service. The return value also provides path information of generated containers so that the external module can take and send them to the end users eventually.

D. Data Structure

There are several data structures which act as sources of initialization messages: *TVAInitDataType* for *TVA Init Message, TextualDecoderInitDataType* for *Textual_DecoderInit* and *BiMDecoderInitDataType* for binary *DecoderInit*. Of course these are specially designed for our system, but we do not cover their details since their fields directly reflect the initialization messages above. Instead of that, this clause shows some data structures which finally organize hierarchical mapping table of fragments and containers in a metadata service.

- *typedef struct {int FragmentID; int FragmentVersion; int FragmentContextValue; string FragmentPayload;} FragmentInfoType*: *FragmentInfoType* C++ structure contains the whole information about a certain fragment, e.g. its 24bit ID, version, type identifier value, and real fragment body in segmented XML document form. *FragmentPayload* is used internally to maintain real body of fragments, but it does not actually have to have a value when it is involved in returning value of *GetContainers* method since the fragment body data is needless information in order to know the distribution of fragments.
- *typedef struct {bool IsTVAMainContainer; int ContainerID; int ContainerVersion; int NumberOfFragments; FragmentInfoType *FragmentInfo; string ContainerFilePath} ContainerInfoType*: *ContainerInfoType* tells us about a certain container with its expressive fields. This type also contains an array element of *FragmentInfoType* which is a current list of fragments contained in this container. *ContainerFilePath* is an access point of generated container file for external caller and *IsTVAMainContainer* flag teaches us whether this container is *TVAMainContainer* or not.
- *typedef struct {int NumberOfContainers; ContainerInfoType *ContainerInfo; int ServiceID;} ServiceInfoType*: Each generated *ServiceInfoType* variable corresponds to one metadata service. *ContainerInfoType* array explains ID and version information of each container, and where they are. We have already seen this *ContainerInfoType* carries involved fragment information, *FragmentInfoType*, as its array element. External module uses this structure in order to synchronize its own mapping information with TVA codec.

E. Execution Example

This clause provides a sequence of calling examples from the initialization to acquisition

- *TVAManager TVAManager(*numberOfServices, *serviceID, *maxContainerNums, maxContainerSize)*: Firstly, this method constructs an instance of *TVAManager* class in order to create initial blank containers.
- *ServiceInfo = TVAManager.GetContainers(serviceID)*: The caller gets initialized status of TVA codec right after creating a number of *TVAManager* instances by calling this method. It returns initialized number of containers, container IDs and container versions starting from zero for each case of metadata service. Note that the constructor of *TVAManager* is designed to create one *TVAMain Container* template. The requirement is fulfilled by setting

IsTVAMainContainer flag bit on in the *ContainerInfoType* variable whose container ID is zero.

- *TVAInitDataType TVAInit*: Next, the caller needs to declare a variable of *TVAInit* data type and assign proper values to the internal fields. As the occasion demands, *BiMDecodeerInitDataType* or *TextualDecoderInitDataType* variables are made with this.
- *TVAManager.SetTVAInit(serviceID, TVAInit)*: In this step, the caller creates *TVA Init Message* of a certain metadata service using the *TVAInit* variable created in the previous step. Default decoder init message is generated if above parameter set is used, but user can deliver a specified decoder init message by using other overridden versions of this method if necessary.
- *TVAManager.CreateTVAMainContainer(serviceID)*: An actual *TVA Main Container* file is created.
- *TVAManager.CreateContainer*(parameter set 1): This is the first allocation of fragments into a container file so that it does not designate specific containers and that is why the parameter set 1 does not include container IDs. Fragments are evenly allocated to initial blank containers.
- *ServiceInfo=TVAManager.GetContainers(serviceID)*: From now, the external caller should catch up with the distribution of fragments. This acquiring process must be called right after every *CreateContainer*, regularly. The returned *ServiceInfo* data is used by the subsequent call of *CreateContainer* to decide where to place a new fragment or to detect where the fragment to be updated is located.
- *TVAManager.CreateContainer*(parameter set 2): Now the external module can designate specific location of fragment to be updated, added, or deleted. The second overridden parameter set surely has correspondent container information for every fragment.
- *ServiceInfo = TVAManager.GetContainers(serviceID)*: The purpose of this calling is the same with the prior one.

IV. Conclusion

This paper suggests practical approaches about some implementation issues used in our digital cable broadcasting environment. There are still ongoing discussions about managing metadata, such as integrated maintenance of fragment metadata when they are used both in uni-directional and bi-directional broadcasting environments. In other words, service providers who want to share TVA metadata between their various broadcasting service types should establish proper metadata management policies.

References

[1] The TV-Anytime, “TV-Anytime Forum,” http://www.tv-anytime.org/, 2007.

[2] S. Lim, J. Choi, J. Seok and H. Lee, “Advanced PVR architecture with segment-based time-shift,” International Conference on Consumer Electronics, 2007. Digest of Technical Papers.

[3] ETSI TS 102 822-2 V1.3.1, “Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime");Part 2: System description,” 2006.1

[4] ETSI TS 102 822-3-1 V1.3.1, "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime"); Part 2: Metadata; Sub-part 1: Phase 1 – Metadata schemas," 2006. 1

[5] ETSI TS 102 822-3-3 V1.3.1, "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime"); Part 3: Metadata; Sub-part 3: Phase 2 – Extended metadata schema, " 2006. 1

[6] ETSI TS 102 822-3-2 V1.3.1, "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime"); Part 3: Metadata; Sub-part 2: System aspects in a uni-directional environment," 2006. 1

[7] ISO/IEC 15938-1, “Information Technology — Multimedia Content Description Interface — Part 1: Systems,” 2002

[8] ETSI TS 102 471 V1.2.1, “Digital Video Broadcasting (DVB); IP Datacast over DVB-H: Electronic Service Guide (ESG),” 2006.11

Minje Kim is a member of research staff working on multimedia technologies for interactive and intelligent digital broadcasting in ETRI, Korea. He got his B.S. degree from Ajou University and M.S. degree from Postech, in computer science, in 2004 and 2006, respectively. His research is based on statistical machine learning and he is widening his application area from signal processing, multimedia broadcasting and user-friendly multimedia consuming systems.

Minsik Park received the BS degree in electrical engineering from Kwangwoon University of Seoul, Korea in 1997 and the MS degree in mechatronics from Kwangju Institute of Science and Technology, Gwangju, Korea in 1999. Since 1999, he has been a senior member of research staff in ETRI(www.etri.re.kr), where he has developed an advanced digital television technology such as data broadcasting and personalized broadcasting.

Seung-Jun Yang received the BS degree in computer science from Suncheon National University, Korea in 1999 and the MS degree in computer science from Chonnam National University, Korea in 2001. Since 2001, he has been a senior member of research staff in Broadcasting Media Group of ETRI, where he has developed an advanced digital television technology such as data broadcasting and personalized broadcasting. He has been involved in making domestic personalized broadcasting standard, transmission and reception standard for terrestrial personalized broadcasting, as a member in Telecommunications Technology Association. His research interests include TV-Anytime metadata, personalized broadcast systems, and metadata generation.

Ji Hoon Choi received the B.S and the M.S degrees in electronic engineering from Kyunghee University, Suwon, Korea, in 1999 and in 2001, respectively. In 2001, he joined the Data Broadcasting Research Group in ETRI(Electronics and Telecommunications Research Institute, Deajeon, Korea, where he have been working on the interactive data broadcasting system. His research interests are the data broadcasting and network QoS.

Han-kyu Lee received the BS and MS degrees in electronics engineering from Kyungpook National University, Daegu, Korea, in 1994 and 1996, repectively. In 1996, he joined Electronics and Telecommunications Research Institute(ETRI), Daejeon, Korea, where he has worked for research and development of broadcasting and multimedia technologies. Since 2005, he has served as team leader for Personalized Broadcasting Research Team of ETRI. His main research intrestes are in the areas of signal processing, intelligent and interactive systems for multimedia.