TECHNICAL REPORT

ISO/IEC TR 21000-12

First edition
2005-04-01

Information technology — Multimedia framework (MPEG-21) —

Part 12: Test Bed for MPEG-21 Resource Delivery

Technologies de l'information — Cadre multimédia (MPEG-21) —

Partie 12: Lit d'essai pour livraison de ressources MPEG-21

Reference number
ISO/IEC TR 21000-12:2005(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

In exceptional circumstances, the joint technical committee may propose the publication of a Technical Report of one of the following types:

— type 1, when the required support cannot be obtained for the publication of an International Standard, despite repeated efforts;

— type 2, when the subject is still under technical development or where for any other reason there is the future but not immediate possibility of an agreement on an International Standard;

— type 3, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example).

Technical Reports of types 1 and 2 are subject to review within three years of publication, to decide whether they can be transformed into International Standards. Technical Reports of type 3 do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 21000-12, which is a Technical Report of type [3], was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

ISO/IEC TR 21000 consists of the following parts, under the general title *Information technology — Multimedia framework (MPEG-21)*:

— *Part 1: Vision, Technologies and Strategy* [Technical Report]

— *Part 2: Digital Item Declaration*

— *Part 3: Digital Item Identification*

— *Part 5: Rights Expression Language*

— *Part 6: Rights Data Dictionary*

— *Part 7: Digital Item Adaptation*

— *Part 8: Reference Software*

— *Part 9: File Format*

— *Part 10: Digital Item Processing*

— *Part 11: Evaluation Tools for Persistent Association Technologies* [Technical Report]

— *Part 12: Test Bed for MPEG-21 Resource Delivery* [Technical Report]

— *Part 16: Binary Format*

The following parts are under preparation:

— *Part 4: Intellectual Property Management and Protection Components*

— *Part 15: Event Reporting*

Introduction

This document describes the component API of ISO/IEC TR 21000-12: Test Bed for MPEG-21 Resource Delivery. The test bed is mainly composed of a streaming player, a media server, and an IP network emulator. This document describes the API of each components of the test bed to facilitate a component oriented development process. This platform provides a flexible and fair test environment for evaluating scalable media streaming technologies for MPEG contents over IP networks.

Information technology — Multimedia framework (MPEG-21) —

Part 12:
Test Bed for MPEG-21 Resource Delivery

1 Scope

This Technical Report specifies a test bed that is designed to assist in performance assessment of MPEG-21, Scalable Video Coding (SVC) for streaming applications and for the evaluation of resource delivery technologies over unreliable packet-switched networks. A subset of MPEG-4 IPMP is also included in the test bed so that encrypted streaming and layered access functionality of a DRM system can be tested for different SVC designs.

2 Overview of Functionality

2.1 Functionality of Test Bed

This platform provides a flexible and fair test environment for evaluating scalable media streaming technologies for MPEG contents over IP networks. In particular, the test bed is designed for the evaluation of Scalable Video Coding (SVC). This test bed has capabilities of simulating different channel characteristics of various networks, therefore,

- Various scalable codec (audio, video, scene composition) technologies could be evaluated.
- Various packetization methods and file formats can be evaluated.
- Various multimedia streaming rate control and error control mechanisms can be plugged into the test bed and evaluated.

2.2 Target Use Cases

Currently, the test bed is targeted at scalable audio and video streaming applications with some DRM support.

2.3 Relation with Reference Software

It must be emphasized that the software provided with this TR is not part of the MPEG-21 reference software. In addition to providing some useful utility software for resource delivery system development, this TR tries to show a solid example of how MPEG technologies can be integrated together in a working system for scalable audio/video streaming applications.

2.4 MPEG Technologies within Test Bed

The following MPEG technologies are supported by the test bed:

- MPEG scalable audio and video codecs
- MPEG-4 on IP
- MPEG-4 IPMP (a small subset)
- MPEG-21 DIA Network Adaptation QoS

The architecture and the API are not tied to any particular media codecs. However, only the MPEG-4 FGS video and MPEG-4 BSAC audio are officially supported by the software.

2.5 API Overview and Language

The API is divided into three parts, namely, server component API, client component API, and common component API. The network emulator descried in clause 7 is a standalone application that is not part of the component-based framework. The programming language used to describe the API is C++. This is because the project is mainly implemented using the C++ language, except for the network emulator GUI, which is done in Java. This will make it easier for the implementers to follow the API and to design different modules to merge into the test platform.

3 Overall Architecture

The overall architecture of the Test Bed is illustrated in Figure 1.

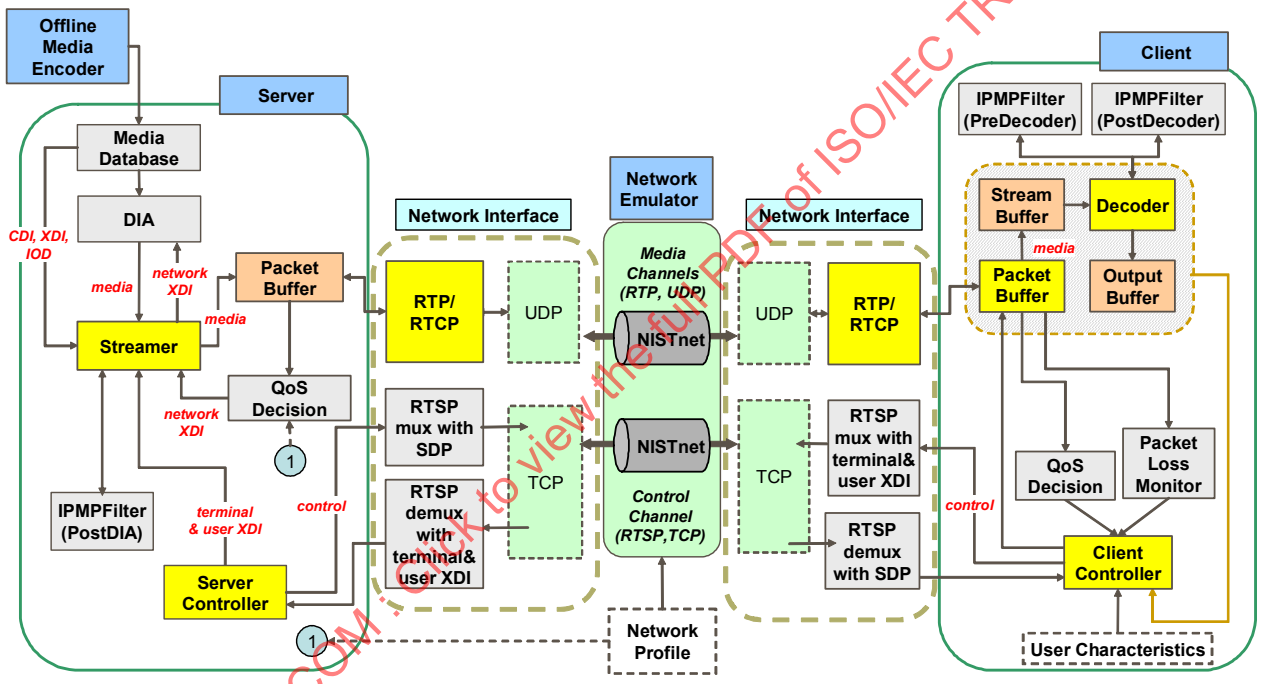

Figure 1 — Architecture of the resource delivery test bed

Subsequently, the various server and client components that can fit in this test-bed architecture are described in clause 4.

4 Client Components

4.1 Introduction

This section lists the detail component APIs so that third party component providers can design their own client modules that can be integrated into this test bed.

4.2 Decoder Object

This is the base object for all decoders, including real-time and offline decoders. The task of fetching coded units from the input streams and storing decoded units into the output stream is carried out by this base object. This is done as follows:

1. **Decoder** gets some coded bits from the input bitstream by calling **StreamBuffer::GetBitstreamData()**. A bitstream can contain multiple layers of sub-bitstreams. If no data in any layers of the sub-bitstream is available, the **Decoder**'s thread is suspended till data is available.
2. **Decoder** calls a private virtual function, *Decode()*. This function is overloaded by specific decoders and performs the actual decoding of a coding unit.
3. The output of *Decode()* is saved in a buffer via the method **OutputBuffer::WriteDecodedData()**. The GUI can later calls **OutputBuffer::GetDecodedData()** to display (video) or play (audio) the decoded content. A presentation timestamp will be provided in this later API.

Methods
The following methods are implemented by the base class **Decoder**.

Decoder()

The object constructor (serious initialization is done in *Setup*()). No work is done till *Start()* is called.

int Setup(int track_id, uint8 *config, int32 config_length, int32 codec_type, void* streambuffer_link, void* outputbuffer_link)
This function initializes the decoder. The video header (or video configuration descriptor is passed in for parameters like video width & height or audio frame size). *config_length* is the length in bits of *config*, and *streambuffer_link* and *outputbuffer_link* are the pointers to StreamBuffer object and OutputBuffer object respectively. A non-zero error code returns if it fails.

void Start()

Start the **Decoder** thread, *Run()*, to perform the decoder task.

void Close()

Must be called before the **Decoder** object is deleted in order to terminate the **Decoder** thread.

static void Run(void *)

This method, which runs the **Decoder** thread, fetches compressed units from the input **StreamBuffer** object, calls *Decode()* to decode them, and saves the decoded presentation units to the output **OutputBuffer** object through the **OutputBuffer::WriteDecodedData()** method. The function terminates when the input stream returns END_OF_STREAM on a call to *GetBitstreamData()*.

bool Decode()

Decode one coding unit. The function returns TRUE if successful.

uint32 GetAttributes(ATTRIBUTE_CODE code, void *value)

Depending on the value of *code*, the requested attribute is returned through the pointer parameter *value*. In some codecs, this function returns meaningful values after the *Setup()* function (i.e. after the decoding of the audio-visual configuration header). A nonzero error code returns if it fails. For some codecs (e.g. ITU-T H.263), this function only returns meaningful values after decoding of a frame (after calling the function *Decode()*).

When the attribute "SUPPORTED_CODEC_NAMES" is requested, the function returns a pointer of an array of UTF-8 strings of codec names. Each string in the array begins with an one byte length field, followed by the chacters. Strings are concatenated one after another, the whole string arrays ends with a zero byte. For example, a **Decoder** object may returns a pointer points to the following string array:

{ 0x03, 'A', 'S', 'P', 0x03, 'F', 'G', 'S', 0x00 }.

The default implementation of this function always returns zero.

ATTRIBUTE_CODE is defined as follows:

```
typedef enum
{
        // common attributes
        SUPPORTED_CODEC_NAMES

        // for visual
        FRAME_WIDTH,     // the width in pixels of a video frame
        FRAME_HEIGHT,    // the height in pixels of a video frame
        FRAME_BITS_PER_PIXEL,  // bits per pixel in a video frame
        FRAME_PIXEL_FORMAT, // pixel format in a video frame

        // for audio
        PITCH,        // the pitch of an audio sample
        AUDIO_FREQUENCY,    // the frequency of an audio sample
        NUM_CHANNELS,    // number of audio channels
        BITS_PER_SAMPLE,    // number of bits per audio sample
        AUDIO_CU_DURATION,  // the duration in milliseconds of a sample
} ATTRIBUTE_CODE;
```

As you can see from this definition, codes for different types of decoders are mixed. A decoder is expected to return value only for attribute codes that are relevant to it. Otherwise it returns zero. The returned attribute is always type-casted to uint32. The method returns zero if the requested attribute is not recognized or available by the decoder.

4.3 OutputBuffer Object

This object class is used as an abstraction of the output device (for either audio and video). The decoder, after decoding, should dump the decoded data into this object. The client GUI would then send the data to the actual device based on the timestamp.

Methods

OutputBuffer()

This is the object constructor.

int Setup(int32 buffer_size)

buffer_size is the size of one decoded data unit (e.g. an YCbCr video frame or a PCM audio frame) in bytes.

int WriteDecodedData(uint32 timestamp, uint8 *data, int32 data_length)

This method sends a decoded unit to the output buffer. The pointer **data* points to the decoded frame and the size of the decoded data is specified by *data_length*. The timestamp, *timestamp*, of the decoded frame is in milliseconds. The function returns zero upon success, otherwise, it returns a non-zero code when it fails.

int GetDecodedData(uint32 *timestamp, uint8 **data, int32 *data_length)

The client GUI will call this method to render the decoded content (audio or video). The indirect pointer ***data* returns a pointer to the decoded data and the size of the decoded data is returned via *data_length*. The presentation time is returned via the parameter **timestamp*. The client GUI should render the data at time *timestamp* (in millisecond). The function returns zero upon success, otherwise, it returns a non-zero code when it fails.

int ReleaseBuffer(void)

After client player gets decoded data, it will call this method to release the control to the buffer which is obtained by using **GetDecodedData()**.

4.4 StreamBuffer Object

StreamBuffer is a FIFO buffer that holds raw bitstream data. Even though the data structure is intended for raw data bits, the actual implementation should record raw bitstream boundary "markers" set by the transport layers, e.g. IP packet boundaries (see method **GetBitstreamData()** for more explanations), through some auxiliary data structure.

Methods

StreamBuffer(int number_of_tracks)
This is the object constructor. *number_of_tracks* is the total number of media tracks (e.g. audio-visual tracks or multiple levels/layers for scalable codecs).

int Setup(int track_id, int32 buffer_size, void *packetbuffer_link)
buffer_size is the size of the raw bitstream buffer in bytes. *Packetbuffer_link* is the link to **PacketBuffer** object.

int SetBookmark(int track_id, int32 bookmark_number)
This method records the current position in stream buffer to a bookmark array for track *track_id*, and the corresponding index in this array is *bookmark_number*. This method is always called by **Decoder** object to do backward searching preparation. It retruns a non-zero code when it fails.

int GotoBookmark(int track_id, int32 bookmark_number)
This method will move the current position to that recorded in the bookmark array by index *bookmark_number* for track *track_id*. It returns a non-zero code when it fails.

int32 GetBitstreamData(int track_id, int32 nbytes, uint8 *data)
This method retrieves *nbytes* bytes from the track *track_id* of a **StreamBuffer** object, and returns the value to the caller. The returned data is stored in the pointer *data*. The function returns zero upon success, otherwise, returns a non-zero code when it fails.

int32 GetOffsetToNextDataBoundary(int track_id)
In a streaming-over-IP system, bitstream data has a natural boundary set by the transport layer (e.g. for each frame, slice or video packet, etc.). This function returns the number of bits to the next marker (or packet boundary) position in track *track_id*. Some decoders can use this information to avoid resync marker searching. It returns zero if there is no further boundary point.

int PutBitstreamData(int track_id, int32 nbytes, uint8 *data)
This method adds *nbytes* bytes into track *track_id* of the **StreamBuffer** object. The output data is stored in the pointer *data*. The function returns zero upon success, otherwise, returns a non-zero code when it fails.

void SetDataBoundaryPoint(int track_id)
This method set the current bit position as a data boundary point of track *track_id*. This information will be used by succeeding calls to *GetOffsetToNextDataBoundary()*. This method is usually called by a **PacketBuffer** object.

4.5 PacketLossMonitor Object

This module handles lost packet monitoring and retransmission. The method *CheckPacketLoss()* will be triggered occasionally from client-side timer to issue retransmission requests.

Methods

PacketLossMonitor(int32 number_of_track)

This is the object constructor. *number_of_tracks* is the total number of media tracks (e.g. audio-visual tracks or multiple levels/layers for scalable codecs).

int Setup(void *packetbuffer_link, void *clientcontroller_link)
The parameter *packetbuffer_link* is a pointer to the **PacketBuffer** object, and *clientcontroller_link* is a pointer to the **ClientController** object.

void EnableRetransmission(int track_id, bool on_off_flag)

This method turns on/off the retransmission mechanism for track *track_id*. The state is controlled by *on_off_flag*. If *on_off_flag* = 0, the mechanism is turned off, otherwise, it's turned on.

void CheckPacketLoss()

This function checks all the track buffers whose retransmission mechanism has been turned on to see if there are any packet losses. If there are packet losses and the condition is valid for retransmission, this method calls **ClientController::RetransmissionRequest()** to issue retransmission requests.

4.6 ClientController Object

This object class integrates and controls the components of client (include the component of IPMP). The design principle of the **ClientController** APIs tries to follow the "maximum freedom" rule. Even though the naming of the methods is adopted from the RTSP model (such as, DESCRIBE, PLAY, etc.), there is no coercing that RTSP has to be used as the control protocol.

Methods

ClientController()

This is the object constructor.

int Setup(const void *client_info)

This method set up the **ClientController** object. The input parameter *client_info* is intentionally left as a free-form parameter. In the current implemtation, this parameter is a pointer to a structure that contains information about session ID, media info (for SDP [5]), terminal capabilities, etc. The structure is highly

dependent on the control protocols used. The function returns zero upon success, otherwise, it returns a non-zero code when it fails.

void Start()

Start the **ClientController** thread, *Run()*, to perform the control event processing loop.

void Close()

Must be called before the **ClientController** object is deleted in order to terminate the **ClientController** thread.

static void Run(void *)

This method, which runs the **ClientController** thread, sits in a loop waiting for the GUI user inputs and processing these inputs. If failure or exception occurred, this thread should set an event to acknowledge the UI. The event handler can be fetched via *GetAttributes()* method.

int GetAttributes(const char *attribute_name, void *value)

This method returns a server or client attribute (specified by *attribute_name*). The requested attribute is returned through the pointer parameter *value*. If the returned value is NULL, then *attribute_name* is an invalid (or unsupported) attribute. If it is a server attribute, the client could use the GET_PARAMETER method of RTSP to get it. The function returns zero upon success, otherwise, it returns a non-zero code when it fails.

int SetAttributes(const char *attribute_name, void *value)

This method sets the value of client or server attributes. If it is a server attribute, the client could use the SET_PARAMETER method of RTSP to set it. The attribute is specified by *attribute_name*, and its value is pointed to by *value*. The function returns zero upon success, otherwise, it returns a non-zero code when it fails.

int RetransmissionRequest(uint32 *packet_numbers, int nPackets, int track_id)

A **PacketLossMonitor** object calls *RetransmissionRequest()* to issue a retransmission with specified packet numbers in packet number list (*packet_numbers*). The number of packets in the list is given by the parameter *nPackets*. The requested missing packets belong to the media track *track_id*. The function returns zero upon success, otherwise, it returns a non-zero code when it fails.

void OnStartPresentation(void *parameters)

This call-back method will be called upon the receiving of a “start media presentation” request from the GUI. The input parameter *input* is intentionally left as a free-form parameter. When the user issues a PLAY request through the client GUI, this methods will be called. Three types of (RTSP) control requests will be processed here, namely DESCRIBE, SETUP, and PLAY.

void OnStopPresentation(void *parameters)

This call-back method will be called upon the receiving of a “end media presentation” request from the GUI. The input parameter *input* is intentionally left as a free-form parameter. The control requests TEARDOWN will be processed in this method.

5 Server Components

5.1 Introduction

This section lists the detail component APIs so that third party component providers can design their own server modules that can be integrated into this test bed.

5.2 MediaDatabase Object

This object class currently uses a simple directory tree structure for content storage. The server configuration file must sit in the root directory of the content database.

Methods

MediaDatabase()

This is the object constructor.

int OpenMediaFile(int track_id, char *filename)

This method opens file *filename* for track *track_id*. This method is usually called by the **Streamer** object.

int CloseMediaFile(int track_id)

This method closes media file for track *track_id*. This method is usually called by the **Streamer** object.

int GetNextResourceUnit(int track_id, uint8 *buffer, int32 *buffer_length, uint32 *timestamp)

This method retrieves one resource unit (can be a video frame, video packet, or an audio frame) from track *track_id* along with its timestamp. This function is usually called by the **DIA** object and the data will be stored in the buffer pointed to by *buffer*. The parameter *buffer_length* is an input/output parameter that stores maximum buffer length (in bytes) as an input parameter and returns the actual data size. Timestamp of the data will be returned via *timestamp*. The function returns a user-defined status code that controls the transmission behavior of the streamer.

int GetMediaConfigHeader(int track_id, uint8 *buffer, int32 *buffer_length)

This method retrieves the media configuration header (for example, the MPEG-4 VOL header) from track *track_id* for SDP composition. This function is usually called by the **ServerController** object and the data will be stored in the buffer pointed to by *buffer*. The parameter *buffer_length* is an input/output parameter that stores maximum buffer length (in bytes) as an input parameter and returns the actual data size. The function returns zero upon success, otherwise, it returns a non-zero code when it fails (for example, if the buffer is too small to store the next decoding unit).

5.3 ServerController Object

The design principle of the **ServerController** APIs, just like the **ClientController** APIs, tries to follow the "maximum freedom" rule. Even though the naming of the methods is adopted from the RTSP model (such as, DESCRIBE, PLAY, etc.), there is no coercing that RTSP has to be used for the implememtation. To create a multicast **ServerController**, for example, most of these APIs will be NULL APIs that does not do anything.

For a unicast system, an individual server controller will be created for each connecting client.

Methods

ServerController()

This is the object constructor.

int Setup(const void *server_info)

This method set up the **ServerController** object for a connecting client. The input parameter *server_info* is intentionally left as a free-form parameter. In existing implementation, this parameter is a pointer to a structure that contains information about session ID, media info (for SDP [5]), etc. The structure is highly

dependent on the control protocols used. The function returns zero upon success, otherwise, it returns a non-zero code when it fails.

void Start()

This method starts the **ServerController** thread, *Run()*, to perform the control event processing loop.

void Close()

This method must be called before the **ServerController** object is deleted in order to terminate the **ServerController** thread properly.

static void Run(void *)

This method, which runs the **ServerController** thread, sits in a loop waiting for the control message inputs from the control channel and processing these inputs by calling the callback function. To adapt the framework to different propocols, only the callback function needs to be changed.

void Callback(int32 message_id, void *parameters)

This callback function is called by control protocol module when a message is received. The pointer *parameters* specifies the input parameters from the message. The callback method shall configure the server state, process the requests, and send ACK back to the client.

int GetAttributes(const char *attribute_name, void *value)

This method returns a server or client attribute (specified by *attribute_name*). The requested attribute is returned through the pointer parameter *value*. If the returned value is NULL, then *attribute_name* is an invalid (or unsupported) attribute. If it is a client attribute, the server could use the GET_PARAMETER method of RTSP to get it. The function returns zero upon success, otherwise, it returns a non-zero code when it fails.

int SetAttributes(const char *attribute_name, void *value)

This method sets the value of client or server attributes. If it is a client attribute, the server could use the SET_PARAMETER method of RTSP to set it. The attribute is specified by *attribute_name*, and its value is pointed to by *value*. The function returns zero upon success, otherwise, it returns a non-zero code when it fails.

5.4 Streamer Object

The **Streamer** object performs several tasks: packing bitstreams into RTP payload and stores them into packet buffer. This object also performs rate adaptation and packet scheduling.

Methods

Streamer(char *ip_addr, void *DIA, void *QoS, int number_of_tracks)

This is the object constructor. The parameter *ip_addr* is the IP of the remote host, *DIA* is a pointer to the associated **DIA** object, *QoS* is a pointer to the **QoSDecision** object, and *number_of_tracks* is the total number of media tracks (e.g. audio-visual tracks or multiple levels/layers for scalable codecs).

void Setup(int track_id, int32 client_port, void *rtp_info)

This method commands the the streamer to create a session for a client and retrieve the related RTP information such as SSRC, time base, sequence number offset, and local port. These are necessary information for the sending controller to communicate with receiveing controller via SDP. The data type of *rtp_info* can be a user-defined data structure that contains variables for storing necessary information.

void Start()

Start the streamer thread, *Run()*, to perform the streaming task.

void Close()

Must be called before the **Streamer** object is deleted in order to terminate the streamer thread.

static void Run(void *)

This method runs the streamer thread. It call **DIA::GetNextResourceUnit()** to read some data for packetization. It performs rate adaptation by discarding some data from enhancement layers. The *Run()* thread is created as soon as a client connects. However, it will not start sending data until an RTSP "PLAY" request from the client is received. Upon receving a PLAY request, the **ServerController** object calls **Streamer::StartSending()** to start sending the RTP packets. The *Run()* thread also calls the method **QoSDecision::CalculateCurrentBandwidth()** to retrieve available network bandwidth in order to perform rate adaptation.

void StartSending(int track_id)

This method sets the sending flag for the corresponding client so that the *Run()* thread starts pumping out RTP packets. The parameter *track_id* informs the streamer which media track to send.

void RetransmissionRequest(int track_id, uint32 seq_no)

This method sets retransmission flag for packet *seq_no* of track *track_id*. **ServerController** calls this method to notify the **Streamer** object to resend the missing packet.

void SetAdaptationParameters(void *parameters)

The ServerController object uses this function to pass the client terminal capability information (such as maximal width, height, and video frame rate or audio sampling rate it can support) into the Streamer. The pointer parameter is left as a free form pointer so that different implementation of ServerController and Streamer can design their own data type.

5.5 File Format

Currently, the file format for a content maintained by **MediaDatabase** object is just a set of raw bitstreams (ASP, FGS, and BSAC) and XML files (CDI, XDI) located in a directory under the name of the content. The design simplify the system and the authoring tools. However, for a large scale system, this type of content management is not very efficient. Eventually, ISO file format should be used in this reference platform.

5.6 DIA Object

The DIA object performs media resource adaptation by DIA engine processing. At first, the DIA engine will get the CDI and static XDI information from **ServerController** object for initial setup. Besides, the dynamic XDI will be set in the adapting process. The adapted resource will be generated with CDI and these two XDIs information. After adaptation, **Streamer** object willl get adapted resource by **GetNextResourceUnit** in DIA object.

Methods

DIA()

This is the object constructor.

int Setup(int track_id, int engine_id, string *CDI, string *XDI, MediaDatabase *MDB)

The **ServerController** object uses this function to initialize the **DIA**. The *engine_id* indicates what kind of adaptation engine to be used. *CDI* provides the properties (ex: location, type, frame rate) of media resource. *XDI* is specified for the adaptation guide settings of media resource. *MDB* is a pointer of **MediaDatabase.** So that DIA can access original media resource.

int SetXDIInfo(int track_id, string *XDI)

This method is used to set the dynamic XDI information. For example, the **Streamer** object can use this method to pass the dynamic network information into the **DIA** object for setting adaptation parameters.

int GetCDIInfo(int track_id, string *CDI)

This method retrieves the adapted CDI information from the **DIA** object.

int GetNextResourceUnit(int track_id, uint8 *buffer, int32 *buffer_length, uint32 *timestamp)

This method retrieves one adapted resource unit (e.g., a video frame, video packet, or an audio frame) from track *track_id* along with its timestamp. This function is usually called by the **Streamer** object and the data will be stored in the buffer pointed to by *buffer*. The parameter *buffer_length* is an input/output parameter that stores maximum buffer length (in bytes) as an input parameter and returns the actual data size. Timestamp of the data will be returned via *timestamp*. The function returns a user-defined status code that controls the transmission behavior of the streamer.

6 Common Components

6.1 Introduction

This section lists the APIs of the components that are common to both the client module and the server module. Even though these components function differently when they are used in the client and in the server, they are listed under common components since both the client side and the server side requires matching pairs of these components.

6.2 PacketBuffer Object

Real-time Transport Protocol/Real-time Control Protocol (RTP/RTCP, RFC-1889) is used as the media transport mechanism. This object class implements an RTP packet buffer data structure. This class is RTP payload independent and calls jRTPLib ([1]) for standard RTP/RTCP composition and transporting. The main method, *Run()*, of this object class should run in its own thread.

A data structure, rtp_header, used in this object is defined as follows,

```
typedef struct _rtp_header
{
        uint32 seq_no;       // extended sequence number
        uint32 rtptime;      // rtp timestamp
        uint8  PT;           // payload number
        uint8  marker;       // marker bit(s)
} rtp_header;
```

Both client and server uses the same **PacketBuffer** object with different initializations. The reason is to force this module to be updated simultaneously and consistently for both the client and the server.

Methods

PacketBuffer(bool for_server, int32 number_of_tracks)

This is the object constructor (serious initialization is done in *Setup()*). The parameter *for_server* is set to true if this object is intended for server, false if it is intended for a client. The parameter *number_of_tracks* specifies the total number of media tracks in the streaming session.

This class has two different overloading Setup() methods, one for the client and the other for the server. These methods are defined as follows.

int Setup(int track_id, int32 number_of_packets, int32 max_packet_size)

On the client side, this function initializes the packet buffer object for media track *track_id*. The parameter *number_of_packets* is the total (maximum) number of packet allowed in the packet buffer. *max_packet_size* is the permissible maximal size of a single packet. The function returns zero upon success, otherwise, returns a non-zero code when it fails.

int Setup(int track_id, int32 number_of_packets, int32 max_packet_size, void* extra_parameter)

On the server side, this method initializes the packet buffer object for media track *track_id*. The parameter *number_of_packets* is the total (maximum) number of packet allowed in the packet buffer for retransmission. *max_packet_size* is the permissible maximal size of a single packet. The function returns zero upon success, otherwise, returns a non-zero code when it fails.

The parameter *extran_parameter* is the pointer of RTP related information (Ex: SSRC) for sever setup.

void Start()

Start the packet buffer thread, *Run()*, to perform the packet buffer task.

void Close()

Must be called before the **PacketBuffer** object is deleted in order to terminate the packet buffer thread.

static void Run(void *)

This method runs the packet buffer thread. On the client side, it fetches RTP packets from the input sockets, stores them in the packet ring buffer, and waits for **StreamBuffer** object to retrieve bitstream data. On the server side, this method is not used.

int ReadCodedUnit(int track_id, uint8 *buffer, int32 *buffer_length, uint32 *timestamp)

This method retrieves enough packets from the packet buffer to compose one decoding unit (can be a video frame, video packet, or an audio frame) from track *track_id* along with its timestamp. This function is usually called by the **StreamBuffer** object and the data will be stored in the buffer pointed to by *buffer*. The parameter *buffer_length* is an input/output parameter that stores maximum buffer length (in bytes) as an input parameter and returns the actual data size. Timestamp of the data will be returned via *timestamp*. The function returns zero upon success, otherwise, it returns a non-zero code when it fails (for example, if the buffer is too small to store the next decoding unit).

int32 WritePacketData(int track_id, rtp_header *header, uint8 *data, uint32 length, uint32 time_schedule)

This method composes some RTP payloads and the corresponding RTP information for track *track_id*. The parameter *data* points to the input bitstream data with size *length* bytes. *time_schedule* is the scheduled time

(in milliseconds) for this packet to go out. The function returns zero upon success, otherwise, it returns a non-zero code when it fails.

int GetLostPacketSequenceNumber(int track_id, int32 **seq_no_list)

This method returns channel condition to the caller(s), namely the **QoSDecision** object and/or the **PacketLossMonitor** object. The parameter *track_id* specifies the media track in interest. The function returns the total number of lost packet since last call. A pointer to an array of lost sequence number is returned via the output parameter *seq_no_list*.

int SendRetransmissionPacket(int track_id, uint32 seq_no, int32* payload_length)

This function is usually called by **Streamer** object for sending retransmission packet (extended sequence number *seq_no*) for track *track_id.* The parameter *payload_length* is the pointer of payload length of the retransmission packet. This method returns a non-zero code when it fails.

void GetChannelCondition(int track_id, int32 *FPL, int32 *CPL, int32 *xSeqNum, int32 *jitter, int32 *RTT, void *app_specific_parameters)

This method returns the current channel condition of track *track_id* to the caller, namely the **QoSDecision** object class. On ther server side, these information mainly comes from the RTCP feedbacks. On the client side, these information comes directly from the receiving statistics of RTP packets. All the parameters are output parameters; and the meanings of parameters are listed as follows [3]:

1. *FPL*: Fraction of packet loss.
2. *CPL*: Culmulative number of packet loss.
3. *xSeqNum*: the highest extended sequence number.
4. *jitter*: packet inter-arrival jitter.
5. *RTT*: estimated round-trip time.
6. *app_specific_parameters*: user-defined channel parameters. All the above parameters are standard channel condition parameters defined in RTP specification [3]. The pointer *app_specific_parameters* is a free-form parameter for user to define extended rate adaptation mechanisms. The information may pass back from the clients to the server via the RTCP APP packets.

int GetCurrentBufferDepth(int track_id)

This method returns the current depth (in milliseconds) of media data in the track *track_id* buffer. The method returns zero upon success, and a non-zero error code upon failure.

int GetNetworkAttribute(int track_id, uint8 *attributes, void* value)

This method returns the network information (such as UDP ports) via the parameter *attribute*, and the requested attribute is returned through the pointer parameter *value*. This method is usually called by the **ClientController** object to retrieve network related information maintained by the **PacketBuffer** object. This parameter is intentionally left as a free form parameter. The method returns zero upon success, and a non-zero error code upon failure.

int SetNetworkAttribute(int track_id, uint8 *attributes, void* value)

This method sets the network information specified by the parameter *attribute*, and the requested attribute is set through the pointer parameter *value*. This parameter is intentionally left as a free form parameter. A typical use of this function is for **ClientController** to set network related information (such as IP address). The method returns zero upon success, and a non-zero error code upon failure.

6.3 QoSDecision Object

This object class estimates the channel condition and provides some QoS information for rate adaptation.

Methods

QoSDecision()

This is the object constructor.

int Setup(bool for_server, void *extra_parameter, void *PB)

This function initializes the **QoSDecision** object. The parameter *for_server* is set to true if this object is intended for server, false if it is intended for a client. *extra_parameter* is a free form pointer that can be used to pass some extra info to setup the object. Finally, *PB* is a pointer to the associated **PacketBuffer** object. In the existing implementation, this parameter passes in the file name of the simulating network profile. The function returns zero upon success, otherwise, returns a non-zero code when it fails.

int CalculateCurrentBandwidth(void *QoS_parameter)

This method returns the current bandwidth to the caller. For a system with true dynamic streaming rate control, the method calls the method **PacketBuffer::GetChannelCondition()** to obtain statistics about the channel and calculates the bandwidth accordingly. The parameter *QoS_parameter* is a free form pointer that can be used to provide implementation specific parameters. The returned value is the available channel rate (in bps) associated with the input *QoS_parameter* of a specific media.

6.4 IPMP Objects

The MPEG-4 IPMP system is mainly composed of three parts, namely Terminal, Message Router, and Tool Manager. The Terminal is the consumer of IPMP protected contents. The functions of a Terminal in IM1 are mapped to the functions of the client of the test bed, with some modifications.

All the messages come from an IPMP Tool to another IPMP Tool or the Terminal to an IPMP Tool are routed by Message Router. IPMP Message Router could just be a conceptual entity that implements the Terminal-side interface between the Terminal and an IPMP Tool.

The function of the Tool Manager is to create instances of IPMP Tools, and to connect the IPMP Tool to the corresponding position to perform IPMP functions.

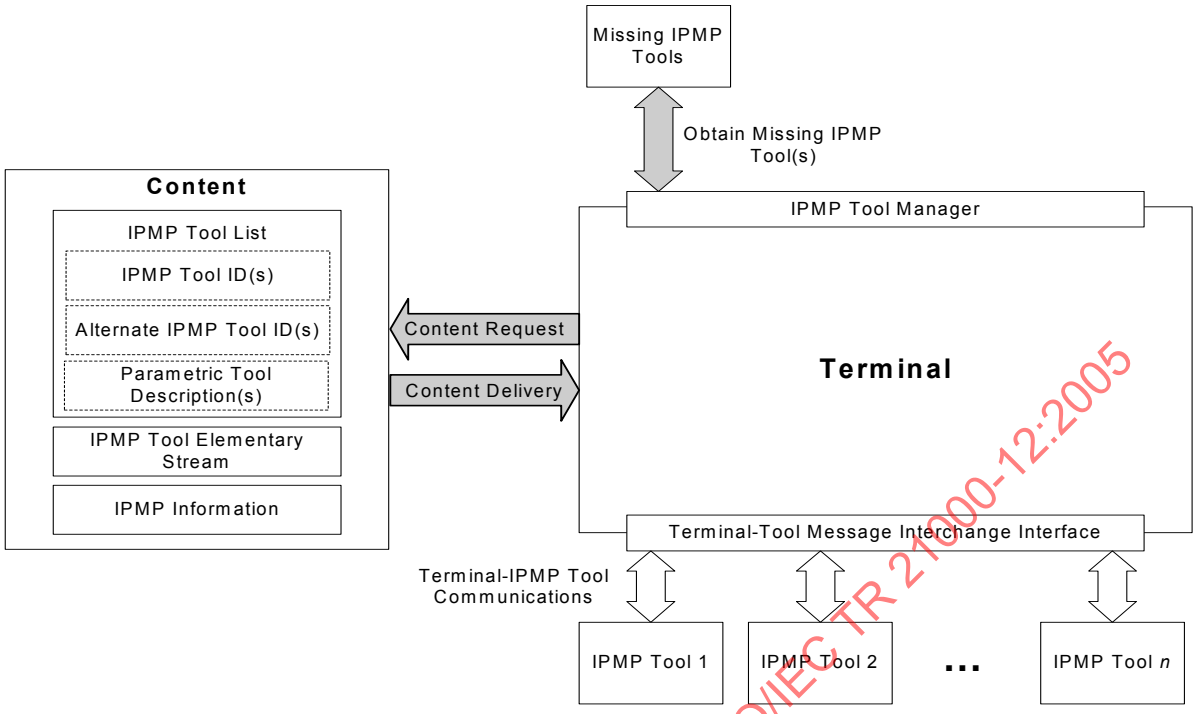

Figure 2 — IPMP architecture

The detailed component behaviors and data structure definitions could be found in [6]. Based on the original MPEG-4 IPMPX architecture, we define the IPMP APIs of the Testbed in the following sections.

6.4.1 MessageRouter Object

All IPMP messages are routed by **MessageRouter.** It also handles **IPMPToolDescriptor**s and **ESD**s. On receiving an **IPMPDescriptorPointer**, it asks **ToolManager** to instantiate tool at given control point.

Methods

int ProcessIPMPToolDescriptor(IPMPToolDescriptorD* IPMPtooldescriptor)

This API is called by the **Terminal** to process the IPMP Tool Descriptor. When the **Terminal** receives the IOD from the server, it calls this function to pass *IPMPtooldescriptor* contained in IOD to **MessageRouter**. The function returns zero when succeed.

int ProcessESD(ESD* esd)

This API is called by the **Terminal** and processes the ES Descriptor. When the **Terminal** receives the IOD from the server, it calls this function to pass all ESDs contained in IOD to **MessageRouter**. The function returns zero when succeed.

int ProcessIPMPDescritorPoniterD (uint32 contextID, IPMPDescriptorPointerD* ptr)

This API is called when processing an ES Descriptor. **MessageRouter** instructs **ToolManager** to instantiate an **IPMPTool** at the given control point of the given ES. The *contextID* specifies the context where the pointer resides. The *ptr* contains the information of which IPMPToolDescriptor is to be used. The function returns zero when succeed.

int ReceiveMessage(IPMPToolMessageBase* msg)

This API is called by an **IPMPTool** or the **Terminal**. On receiving a message (the parameter *msg*),

MessageRouter determines and sends it to the corresponding recipient. The function returns zero when succeed.

IPMPContext* GetContext()

This API provides means to access the corresponding context. All IPMP objects can refer to each other by navigating through the context tree.

6.4.2 ToolManager Object

ToolManager is responsible to parse the **IPMPToolListDescriptor** and to resolve the **IPMPTool**s on the list. It is also responsible to instantiate and destroy the **IPMPTool**s.

Methods

int ReceiveToolListDescriptor(ToolListDescriptorD* tool_list)

This API is called by the **Terminal** and processes the IPMP tool list. When **Terminal** receives the Tool List Descriptor in the IOD, the function is called to pass *tool_list* to **ToolManager**. Then, the **ToolManager** resolves the tool IDs for later instantiation. Note that the server-side and client-side resolving process may be quite different. The function returns zero when succeed.

IPMPTool* ConnectTool(uint32 context_id, IPMPToolDescriptorD* descr)

Instantiate and connect a tool. When this method is invoked, the tool is instantiated by the parameters specified in *descr*. Then, the **ToolManager** connects it to the specified location (*context_id* specifies the ES to apply, and the control point is specified in *descr*).

int DisconnectTool(IPMPTool* tool_ptr)

Disconnect the given tool pointed by *tool_ptr* from the **IPMPFilter**. The function returns zero when succeed.

IPMPContext* GetContext()

This API provides means to access the corresponding context. All IPMP objects can refer to each other by navigating through the context tree.

6.4.3 IPMPTool Object

IPMPTool perform the function of IPMP, such as en/decryption, or watermarking.

Methods

int ReceiveMessage(IPMPToolMessageBase* msg)

This API is called by the **MessageRouter** to pass a message to the **IPMPTool** object. The parameter *msg* is the IPMP message to be handled by this tool. The function returns zero when succeed.

int ProcessData(uint32 timestamp, uint8* in_data, uint32 in_size, uint8 out_data, uint32* out_size)**

This API is called by the **IPMPFilter** to process the given content data. The parameter *in_data*, is the input buffer of length *in_size*. The indirect pointer ***out_data* returns a pointer to the processed data, and the size returned as *out_size*. The output buffer should be created during the invocation of the method, and should be released somewhere else. The **IPMPFilter** should take care of the release operations, except the last output buffer of the IPMP tool chain. The function returns zero when succeed.