Yima Server PE v1.1 Internals

Beomjoo Seo

Feb., 10, 2010

0 Compilation Guide

Let source files be downloaded and extracted to the following directory.

`YIMAHOME = /home/user_account/YimaPE_v1.1`

First, you need to set up your base port number, which is a unique per project group at $YIMAHOME/sysinclude/Yima_module_interfaces.h. Otherwise, your code would result in port conflicts with other server instances.

```
#define BASE_PORT_NUMBER 50000
```

In this example, your RTSP port number is assigned as 50000 and, therefore, your corresponding RTSP url is `rtsp://cervino.ddns.comp.nus.edu.sg:50000/xxx`. The rest of the compilation guide will be exactly same as that of the version 1.0.

1 Overview

This document describes the internal architecture of Yima Server Personal Edition version 1.1 for a class project. Since a lot of server codes, especially file interfaces, have been re-factored for readability, new codes are no longer compatible with the old Yima Client.

As illustrated in Figure 1, the yima server consists of three processes: RTSP front-end, RTSP back-end, and yima node. The RTSP front-end process receives RTSP commands from a remote RTSP client, parses them to construct corresponding internal messages (`SysmonDMsg_T`), and passes them to the RTSP back-end. The RTSP back-end, waiting the internal messages on a LINKPORT, relays the messages from the RTSP front-end to the yima node process, and vice versa. The yima node processes all the server functionalities such as session management, media file retrieval, packetization, and network delivery except RTSP commands handling. It consists of four threads (`scheduler, sysmond_w, sysmond_rsp`, and `MP4Flib`) and a set of file related interfaces, called `FLIB`, which are used by the scheduler and MP4Flib threads.

Scheduler is a thread whose main routine is defined at Scheduler::run_scheduler() in "Server/SCH_SYSMOND/scheduler_sysmon.cc". Its main functionality is to

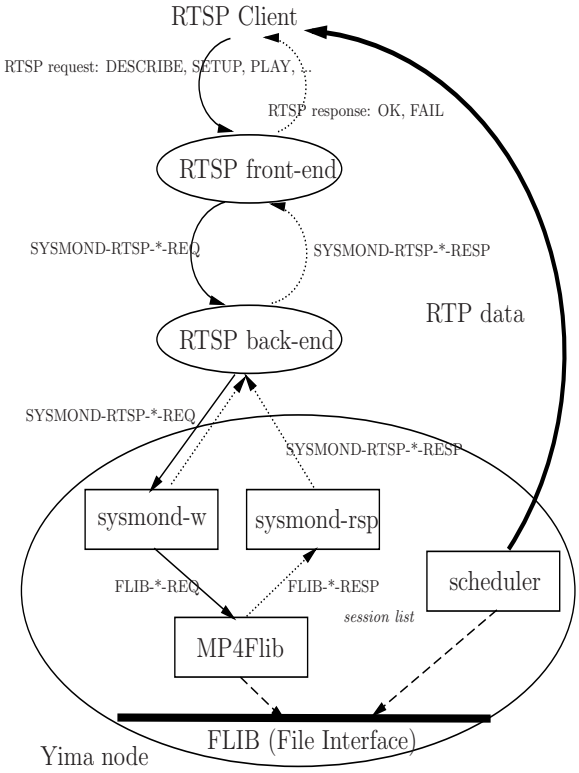

Figure 1: Internal software components of Yima server and their message flow.

send RTP packets to the network and to prepare a next RTP packet to be served. Following is a pseudo code of its main routine:

```
while forever
    wait until any playing session exists.
    compute minimum time to be served among active sessions.
    if minimum time > 0
        sleep minimum time
    for every session in active sessions
        play_session()
            if there is no next RTP packet to be served, return
            compute current time
            get the time of the last RTP packet sent
            if remaining time to be sent is within threshold
                transmit RTP packet
                if the last packet of a block is served
                    notify to mp4Flib thread
                get next RTP packet from Flib
```

```
recompute the next time to be delivered
```

SysMonD_w is a front-end thread of the yima node, waiting internal messages (defined as SysmonDMsg_T in "Server/sysinclude/Yima_module_interfaces.h") from an external RTSP back-end process. It interprets the internal messages and replying back to RTSP processes or forwarding the messages to MP4Flib for file-related operations.

SysMonD_rsp is a thread, waiting internal messages from MP4Flib thread on a message queue and replying back to the RTSP back-end process after performing node-wide operations.

MP4Flib_queue, waiting internal messages on an `MP4Flib_queue`, receives them coming from sysmond_w thread, executes related file interface functions, and sends response messages to spsmond_rsp thread via sysmond_queue.

2 RTSP

2.1 RTSP front-end

It receives RTSP commands from a RTSP client, translates them to server-specific request messages (SysmonDMsg_T), sends the converted messages to RTSP back-end process, receives response messages from the RTSP back-end, again converts the responses to printable characters, and sends the characters back to the RTSP client. Following is the pseudo description of the RTSP front-end routine.

```
infinite loop
    wait on sockets
    who sent the message ?
        if RTSP client
            create a RTSP session and session ID
            fill client information
        else if RTSP back-end process
            if read message successfully
                find the RTSP session
                convert system message to a RTSP response
                send the RTSP response to the RTSP client
                update session state
        else // continue to read data from RTSP client
            read message
            if reading failed
                clean up corresponding session structure
            else
                find the RTSP session
                convert RTSP request to a system message
                send the system message to back-end process
```

It maintains an internal structure called `conntable`, which stores intermediate parsing results of RTSP commands and can be accessible via socket descriptor or via session ID. You may add other information for your own purposes. When the RTSP front-end server receives a new TCP connection request from a RTSP client, it first finds an empty slot from the `conntable` table and associates it with a unique RTSP session id and a newly reallocated TCP socket descriptor.

Conversion from RTSP request to system message

ConvertRequest converts following RTSP commands to their corresponding system messages (SYSMOND_RTSP_*_REQ) that are later processed by the RTSP back-end process.

Conversion from system message to RTSP response

ConvertResponse converts system messages generated from the back-end process (SYSMOND_RTSP_*_RESP) to their RTSP response messages such as `RTSP/1.0 200 OK` or `RTSP/1.0 FAIL`.

2.2 RTSP back-end

Its primary functionality is to simply forward an incoming system message from RTSP front-end process to a number of yima node instances (sysmond_w, in particular), collect response messages from the instances, and relay them to the RTSP front-end process. Since our project uses a single yima node instance and the code does not execute any conversion on the message, this process may well be considered as a dummy process.

3 message flow

In this section, we describe how messages flow throughout yima server components.

DESCRIBE

When a DESCRIBE command is reached at the RTSP front-end, it is first converted to a SYSMOND_RTSP_DESCRIBE_REQ message, in which the session id, movie name, and client address fields are filled accordingly. Such fields are later used by sysmond_w, which newly allocates a session structure that is internally managed by the yima node process and attaches it to the session list, which is shared by all threads in the yima node process.

Sysmond_w then sends a FLIB_GET_NPT_RANGE_REQ message to MP4Flib thread, which, in turn, calls Flib_get_npt_range() function to configure content-specific information. After the function return, the MP4Flib sends its response

message (FLIB_GET_NPT_RANGE_RESP) to sysmond_rsp, which again sends a SYSMOND_RTSP_DESCRIBE_RESP message to the RTSP back-end.

The RTSP front-end finally receives the response message from the RTSP back-end and replies to the RTSP client with a RTSP response message that includes SDP information that contains streaming related information for the client to figure out how to set up the connections for all the media in the movie file during the SETUP stage.

SETUP

The RTSP front-end converts a SETUP request to a SYSMOND_RTSP_SETUP_REQ message, identifying client RTP and RTCP port numbers pre-allocated by the RTSP client. Such port numbers are then piggybacked on a SYSMOND_RTSP_SETUP_REQ message.

Sysmond_w, upon the reception of the message, configures the port numbers in a corresponding session structure and sends a SYSMOND_RTSP_SETUP_RESP message back to the RTSP front-end via the RTSP back-end, specifying server-side RTP and RTCP[1] port numbers per connection.

The RTSP front-end builds a RTSP response message that includes the server-side RTP port and RTCP port numbers.

PLAY

A PLAY command is either converted to SYSMOND_RTSP_PLAY_REQ if there is a specified NPT range or to SYSMOND_RTSP_RESUME_REQ, otherwise.

Sysmond_w for the SYSMOND_RTSP_PLAY_REQ identifies the NPT ranges, configures the start time (in the unit of microseconds) for scheduler to begin to send out RTP packets, and sends a FLIB_PLAY_REQ to MP4Flib. For the SYSMOND_RTSP_RESUME_REQ, it checks current status of the session, configures the session for scheduler to resume the delivery of RTP packets, and sends a SYSMOND_RTSP_RESUME_RESP back to the RTSP front-end via the RTSP back-end.

MP4Lib for the reception of the FLIB_PLAY_REQ message calls Flib_play(), which makes the FLIB to load first two block files onto a memory, and Flib_getNextPacket() to fetch the first RTP packet from the loaded block data, and sends a FLIB_PLAY_RESP to sysmond_rsp.

Sysmond_rsp reads the first RTP packet by setting up the pointer to the RTP packet (`nextRTPPkt_p`), its sequence number (`nextRTPPktSeqNo`), the RTP packet length (`nextRTPPktSize`), and the pointer to the first loaded block data (`blockPtr`). After changing the session state to `PLAYING`, it then sends a SYSMOND_RTSP_PLAY_RESP back to the RTSP back-end.

[1] In the project, you don't have to implement this protocol.

PAUSE

A PAUSE command is transformed to a SYSMOND_RTSP_PAUSE_REQ message. Sysmond_w changes the current status of the session to `READY`, which forces scheduler to stop the sending of RTP packets, and sends SYSMOND_RTSP_PAUSE_RESP back to the RTSP front-end via the RTSP back-end.

TEARDOWN

A TEARDOWN request is converted to a SYSMOND_RTSP_TEARDOWN_REQ by the RTSP front-end. Additionally, it check the incoming request whether QUIT is specified. If so, it sets up the `crashFlg` in the message.

Sysmond_w changes the status of the session to disable the delivery of its RTP packets and sends a FLIB_TEARDOWN_REQ to MP4Flib, which sends a SYSMOND_RTSP_TEARDOWN_RESP message back to sysmond_rsp, which sets the status to `TEARDOWN`, and forwards it to the RTSP front-end. Consequently, the scheduler thread, upon the detection of the TEARDOWN session during the inspection of every session, removes the session from the session list. The RTSP front-end responds with the SYSMOND_RTSP_TEARDOWN_RESP and returns the result to the RTSP client.

4 FLib: File Interface

The file interface maintains two data structures: a locally managed session structure (`struct sessionInfo`) and cached block data (`struct usedBlock`). Both structures are implemented as a singly linked list. To access the session structure, the interface function requires session id as an input. To access the block data, it requires the block name of a movie file. Here describes file interface functions implemented.

- getBlockNum() obtains a block number from a given block name. If no block number is associated with, it will return zero. For example, for a block file name `test.mp4_2`, it returns 2.
- InitDisks() fetches the path name of the directory that stores movie files from a configuration file.
- InitMovies() loads movie information (name, size, playback time) from a configuration file. You may modify this routine to fetch more information related with the movie playback (*i.e.*, time-stamp information, RTP packet lengths, SDP, or RTP packet offset in a block).
- getMovieDetails() returns the pointer to the loaded information of a specified movie.
- getStartBlockName() returns the file name of a starting block from a given movie name and starting time. For simplicity, it ignores the starting time,

returning the fixed starting block name. You need to implement to find the appropriate starting block number by examining timestamp information that matches the starting time.

- getNextBlockName() returns the file name of next block to be fetched from given movie name and current block name. If there is no block to read next, the returned next block name will be a null-terminated character. You also need to implement missing parts.
- getNumPackets_PackSize() returns the number of RTP packets of a given block data (via block name) and their packet lengths. If packet length field is given as a NULL pointer, you don't have to return the length information. You need to complete the writing of this function.
- readBlock() reads a given block file and loads the block data onto the memory (specified in the input parameter) and returns the number of packets and the lengths of individual packets in the block data.
- localReadBlock() scans the cached block list whether a requested block is already loaded. If so, it returns the memory location of the loaded block data, the number of packets in the block, and the lengths of individual packets in the block. Otherwise, it allocates a new memory location that holds whole block data, loads the block data onto the memory, and attaches to the cached block list.
- releaseUsedBlock() releases the use of a given block data area. If no stream uses the block data, it frees the memory location of the block data and the cached entry as well.
- Flib_play() allocates, if necessary, a new session structure and attaches to the head of the session list and **reads two block files in advance**.
- Flib_getNextPacket() accesses a given session, find the RTP packet location and its packet length, and returns its sequential number, timestamp (which will be used for scheduling). If the packet reaches to the end of the block data, it sets up the flag that is notified by the scheduler thread, which in turn triggers MP4Flib to prepare the next block to read. Scheduler thread will continue to call this function when necessary.
- Flib_tear_down() frees the session structure and used block data.
- Flib_lastpktofBlock() computes the next block to read, loads its block file to a memory, and releases previously used blocks.
- Flib_get_npt_range() reads duration information from the movie structure and configures starting and ending time.

Some functions are left incomplete for you to finalize the code for the project. You may add your own proprietary codes if necessary.