Master Thesis

HTTP Live Streaming for Zoomable Video

By

Yan Luo

Department of Computer Science

School of Computing

National University of Singapore

2010/2011

Master Thesis

HTTP Live Streaming for Zoomable Video

By

Yan Luo

Department of Computer Science

School of Computing

National University of Singapore

2010/2011

Advisor: Assoc Prof.Ooi Wei Tsang
Deliverables:
Report:1 Volume

Abstract

HTTP Live Streaming is a HTTP-based media streaming communication protocol implemented by Apple Inc. It provides the solution for encoding, storing and transferring media data, making the streaming session fully over standard HTTP and it can automatically adapt to the available data rate.

VLC media player is an open source media player developed by VideoLAN project. It comprises a multimedia player, encoders and streamers which support various media formats, including HTTP Live Streaming Protocol.

However, VLC media player and HTTP Live Streaming Protocol are not designed to support region-of-interest (RoI) based streaming video playback. Dynamic RoI zooming and panning operations are not supported in a server/client distribution model.

In this paper, we present a RoI-based streaming system over HTTP Live Streaming Protocol, using VLC media player to playback the media streaming at client side. Firstly, we propose a tiled streaming solution to encode, segment, transmit, decode and reconstruct the media stream, which supports zooming and panning operations with dynamic RoI. Secondly, new extended information tags and rules are defined over HTTP Live Streaming Protocol to support a zoomable feature for video streaming. Thirdly, we design a new architecture for the VLC media player which supports the downloading and simultaneous decoding of multiple streams and the reconstruction of the streams for the playback of synchronized RoI frames. Such an integrated approach provides a zoomable video streaming system over HTTP Live Streaming Protocol. Moreover, the resolution of RoI is adaptive to RoI size. Our experimental study shows that the system can offer a smooth switch between different zoomable levels and different tiled streams when a user makes zooming or panning operations.

List of Figures

Table of Contents

Chapter 1

Introduction

In the past 5 years, consumer mobile devices have been a major growth engine in the market. The penetration rate of smartphones is expected to surpass 50% in the US market in 2011. The market is expected to enjoy at least double digit-growth in the following few years. Thanks to this skyrocketing growth, mobile devices have become widely accepted by consumers. A natural question arises. What do users do with their mobile devices? Browsing the internet, playing games and checking email. Among all of these popular activities, video watching always ranks highly in user behavior surveys. YouTube announced that its daily video delivery to mobile devices has passed 200 million last year. Moreover, CISCO has reported a survey result that mobile video has achieved the highest growth rate among all application categories for mobile networks. Based on all these facts, we find that designing a platform to offer a more robust user experience of video watching in mobile devices is an important research topic, in view of current trends in the industry.

Since the inception of mobile devices, mobile device makers have never stopped competing to develop newer models with larger screens and higher resolutions. Meanwhile, High-definition video has become more prevalent with the increasing bandwidth of mobile networks. However, this trend cannot persist indefinitely, due to physical limitations such as screen size. Users will not accept a mobile device with a 20 inch screen no matter how wonderful the resolution is as portability is a fundamental and necessary property of mobile devices. Due to this natural limitation, High-definition video in reality is hardly displayed with its original resolution in mobile devices. To offer a more robust user experience without increasing the screen size, a zoomable feature can be a practical tool for the user. It allows users to see regions which they are interested in on a larger scale. Moreover, the available bandwidth in a mobile network is still very limited compared to a cable network, so with proper redesign of the media stream, a zoomable feature can help to reduce unnecessary data transmission. Unfortunately, there currently isn't a well-developed media streaming communication protocol in the industry which natively supports a zoomable feature for video streaming. Thus, in this project, we aim to implement a media framework that supports zoomable video streaming.

A number of protocols have been designed for streaming media. RTP/RTSP is a solution that is heavily deployed in the industry. Thanks to the great success of iPhone, Apple has become a leader in the mobile market. We observe a much more aggressive pickup in HTTP based streaming compared to RTP/RTSP. HTTP Live Streaming, proposed by Apple Inc as part of its QuickTime and iPhone software system, is a HTTP-based media

streaming communication protocol. Unlike RTP/RTSP streaming, HTTP Live Streaming Protocol has a simpler structure of file fetching and playback control, as it does not take care of package loss and re-fetching issues. It represents a much simpler and cheaper way to stream a media. Moreover, this approach incorporates the advantages of using standard web protocol. The media player can easily be implanted into web browsers. Besides Apple, all other big names, such as Microsoft and Adobe, have announced their media solutions (Microsoft IIS Smooth Streaming, Adobe Flash Dynamic Streaming and Apple HTTP Adaptive Bitrate Streaming) to support HTTP-based streaming. This is a clear indication that HTTP based streaming will dominate the mobile media streaming market, and hence the reason why we have chosen to develop the zoomable video streaming system over HTTP Live Streaming Protocol rather than RTP/RTSP.

Besides video streaming, another essential part of the video watching service is the media player at client side. Among the various high quality open source media players, we decided to implement this zoomable video streaming feature over VLC. Originally starting off as an academic project, VLC has gained its reputation as a well-developed and reliable cross-platform media player. All major audio and video codecs and file formats as well as many streaming protocols are now supported by VLC. Substantially, its modular design makes it easier for developers to plug-in new features. Based on the above reasons, we have done our implementation over VLC.

Our project has proposed a full solution of zoomable video streaming solution over the latest video streaming protocol, HTTP Live Streaming Protocol. This paper will cover the details of this complete solution by the

following topics:

- The structure of the RoI-based streaming designed to support dynamic RoI retrieval;
- The new information tags and rules incorporated in HTTP Live streaming Protocol to support zoomable streaming;
- The new architecture of VLC media player which works with this zoomable video streaming system;
- The merits of the system and its limitations.

We have implemented the proposed solution on the open source media player VLC and our analysis helps to show the advantage and drawback of the system.

In the next chapter, we briefly review some related work of RoI-base streaming. Chapter 2 provides some background information on HTTP Live Streaming Protocol and VLC media player. In Chapter 4, we present our system design for zoomable video streaming over HTTP Live streaming and VLC. Chapter 5 analyses the system performance and discusses the limitations of the system. Finally, we conclude this paper in Chapter 6.

Chapter 2

Related Work

Zooming operations allow a user to select a region-of-interest (RoI) and magnify and display it in whole framesize with higher resolution. Panning allows a user to change the coordinates of the RoI inside the frame without changing the size of the RoI. Users can see any arbitrary RoI using zooming and panning tools in high resolution video.

Unfortunately, current video standards do not support arbitrary and interactive cropping of RoI. Several papers have proposed solutions for coding video which allow for cropping. The latest H.264/AVC scalable extension standard supports spatially scalable coding with arbitrary cropping in its Scalable High Profile (14) (4). However, the video only allows predetermined spatial resolutions and cropping and little research has been conducted on the interactive RoI-based streaming of encoded video.

Mutiple studies have done on RoI based video coding. Aditya propose to use P slices for random access to RoI for every spatial resolution instead of generating multi-resolution presentations. (7) Their study analyses the

optimal slice size to balance the compression efficiency and pixel overhead. Sivanantharasa has proposed to use fexible macroblock ordering (FMO) and a rate control technique to improve the picture quality in RoI minimum bandwidth occupancy. (12) Ming-Chieh proposed to code the video using a fuzzy logic control method to give adaptively weighted factors to each macroblock. (10) It results to give foreground more clarity and to leaves background with less clarity. The base assumption in his method is that audience has more chance to pick up their RoI on foreground instead of background.

Our previous work has demonstrated the usefulness of zooming and panning (11). The study has shown that zooming and panning operations can be easily performed during local playback as the RoI can be cropped from the original video and scaled up for display. However, our research will focus on the more challenging question of how to support zooming and panning operations during remote playback. The media data is stored on a distant server and the media player at client side needs to fetch necessary data about the RoI from the server. In this situation, a substantial problem is bandwidth. To transfer the complete high resolution video would occupy substantial bandwidth. However, not having the original high resolution video at client side makes it difficult to display high-resolution RoI when the user performs zooming or panning actions.

In order to support zoomable video streaming with dynamic RoI, our project group has proposed two methods for RoI-based streaming to support RoI-based media data storing and retrieval, which we refer to as *tiled streaming* and *monolithic streaming*. Both methods can encode, store, and

stream video in a manner which supports dynamic selection of RoI for video cropping (11). Tiled streaming partitions the video frame into a grid of tiles and encodes each tile as an independently decodable stream. Monolithic streaming applies to video encoded using off-the-shelf encoders and relies on pre-computed dependency information for sending the necessary bits for the RoI.

2.1 Tiled Streaming

The tiled streaming method is inspired by Web-based map service, which divide a large map into grids of small square images. When a user browses a map, the small images that overlaps with RoI are transferred and used to reconstruct the RoI. Similarly, in tiled streaming, video frames are partitioned into a grid of tiles. The whole video can be considered as a three dimensional matrix of tiles. Tiled frames in different x-y positions are encoded independently as separate media streams. For a given RoI, the set of tiled streams covering the RoI region is transmitted to reconstruct the RoI. For panning operations, the client media player is allowed to dynamically include new tiled streams and remove unnecessary tiled streams, since each tiled stream is encoded independently.

The reconstruction of tiled streams requires synchronization between different tiled streams. The details of this will be covered later in section 4.6. In our project, we have explored two ways of dividing the frame, shown in Figure 2.1.

The most straightforward and practical approach divides the frame into aligned tiles with the same size.

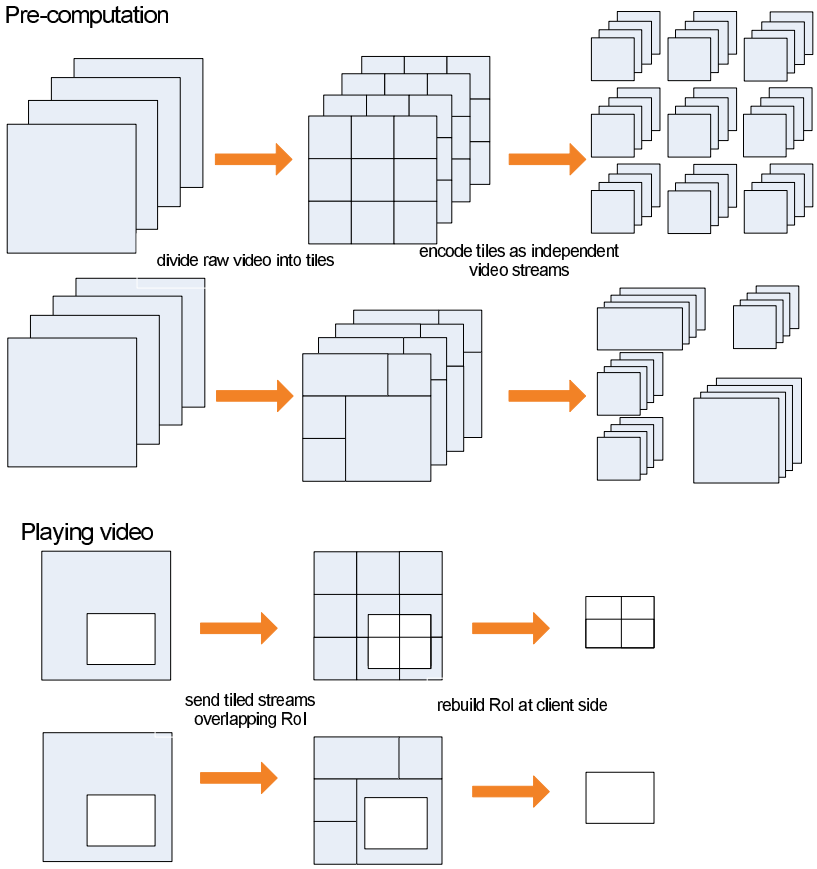

Figure 2.1: Tiled streaming and optimal tiled streaming

To further reduce the volume of the data transmission, the second method divides the frame to a grid of tiles, but with arbitrary tile sizes based on each tile's access probability. This method is known as *optimal tiling streaming*. In video compression technology, the compression efficiency decreases as the tile size decreases. Partitioning the video into a grid of arbitrary tiles based on the popularity of RoI helps to optimize the bandwidth efficiency and ease storage requirements. Our experiment has shown that this optimization can provide around 20% bandwidth consumption reduction. Although the optimal tiling method has the advantage of reduced bandwidth consumption, it requires additional data on RoI popularity which is generally not available for video files. Moreover, the computational cost of this optimization process is not low, and is currently hard to incorporate in real time.

2.2 Monolithic Streaming

Tiled streaming reduces transmission redundancy.

However, it still transmits unnecessary bits. To overcome this drawback, our project group proposes another method named *monolithic streaming* which only transmits the data that are requested for decoding the RoI. This method uses video streams encoded with a standard encoder. But the server has to analyze the dependencies among macroblocks and maintain dependent trees of macroblock. When it receives an RoI query, the server only transmits the macroblock that is needed to decode the RoI. For any given RoI and macroblock m, the server needs to check if m is needed for the given RoI:

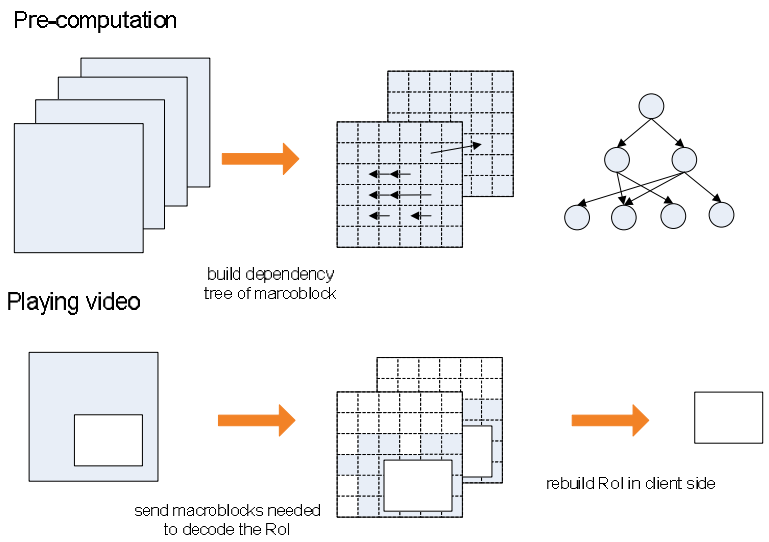

Figure 2.2: Monolithic streaming

- If m falls within RoI, m is clearly needed;
- If there exists a macroblock m' that falls outside RoI but m' depends either directly or indirectly on m, then m' is needed as well.

During run-time, the server needs to lookup every client's request in these dependency trees and put only the needed macroblocks in the transmission packets. The client can play it with a standard video decoder. Although the region outside the RoI is not fully decoded, this deficiency can be ignored since the outside region is not shown to viewers. Figure 2.2 gives an illustration.

Chapter 3

Background

In this chapter, we undertake a basic background study of HTTP Live Streaming Protocol in section 3.1 and VLC Media Player in section 3.2.

3.1 HTTP Live Streaming Protocol

To design a zoomable video streaming protocol over HTTP Live Streaming, it is fundamental that we first understand the current design of HTTP Live Streaming Protocol.

HTTP Live Streaming (HLS), based on HTTP, is designed to send both live or VoD media to iPhone using an ordinary Web Server (6). HLS works by breaking the overall stream file into a sequence of media files. The client fetches one short chunk of media from the server by sending a HTTP request, then decodes and displays the media stream. Media data is ready to transmit once it is created. Therefore, it allows the media to be played nearly in real-time. Comparing to RTP/RTSP, HLS has mainly two advantages.

- Content providers can send live or pre-recorded audio/video using an

ordinary Web Server;

- Unlike RTSP/RTP, which is running default on port 554 which is often blocked by firewall, HLS has the capability to send the data traversing the firewall and proxy server as all data transmissions are over HTTP protocol.

In addition, HLS allows index files to refer to alternative streams of the same content and the index files are sent to the client. Therefore, the client can pick up the optimal streams among all alternative streams based on the network's bandwidth. So it can in fact achieve an auto-adaptive bitrate streaming system.

In this section, we will discuss the fundamental components of the HLS system, such as format of media file and playlist file, and distribution architecture of client/server. In our project, we will follow the latest version 06.

3.1.1 Architecture of HTTP Live Streaming

Conceptually, a media streaming system based on HTTP Live Streaming consists of three major components: the server, the distributor and the client (8). Figure 3.1 is a diagram showing the overall architecture of a HLS system. The details of the server, the distributor and client behavior will be covered in this section.

Server

The server's main functionality is to prepare the media file. It consists of two parts: media encoder and stream segmenter (9). The media encoder

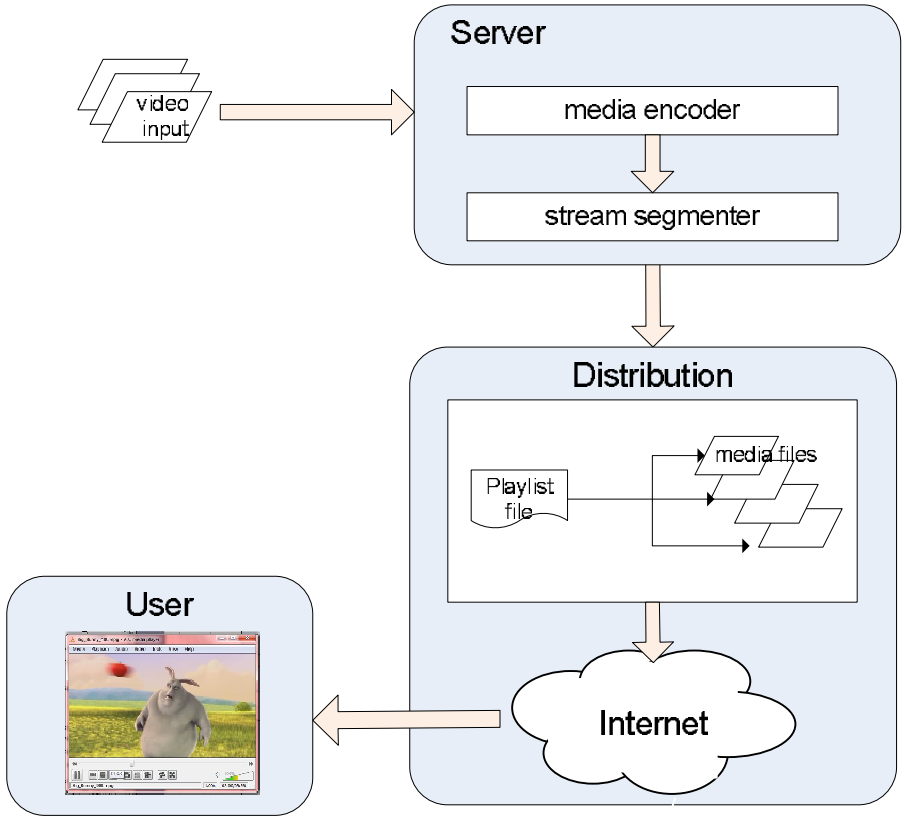

Figure 3.1: Architecture of HTTP Live Streaming

is responsible for encoding media input and preparing to encapsulate it for delivery. Currently, the media stream must be formatted as an MPEG-2 Transport Stream or MPEG-2 audio elementary stream. After the media data is formatted, the stream segmenter takes over the media stream and divides it into a series of small media files. The media files should be able to reconstruct a seamless video stream later at client side. An index file, called the playlist file, is created by the stream segmenter as well. The playlist file contains URIs and additional information about all the media files. Details about the format of the playlist file and media files will be covered in section 3.1.2.

Distributor

The distributor's responsibility is to accept client requests and delivers media data. It is usually a standard web server that sends the playlist file and media files over HTTP. No customized module is needed and little configuration is required.

Client

The client determines which media files to fetch and downloads and decodes them. It reconstructs a seamless video stream and plays it to the user. To stream the video, it always begins by downloading the playlist file, based on a URL identifying the stream. The client fetches the media files in sequence and reassembles them into a video.

3.1.2 Playlist file and Media file

In the HTTP Live Streaming system, to stream and playback a media, two types of files are required: a playlist file and media files. The playlist file contains all the necessary information of a series of media files, which can be used to download the media files in order to rebuild a continuous stream at client side. It is basically an index file and does not have the encoded media data. The media file is the one that contains actual encoded media data.

Playlist file

The playlist file consists of a list of ordered media URIs and information tags. Each URI refers to a media file, which is a consecutive segment of a continuous stream. The playlist file is a pure text file, which follows the Extended M3U Playlist File standard. It has extension name .m3u8, associated with HTTP Content-Type "application/vnd.apple.mpegurl".

The example below is a typical playlist file (13):

```
#EXTM3U
#EXT-X-TARGETDURATION:10
#EXTINF:10,
http://media.example.com/segment_01.ts
#EXTINF:10,
http://media.example.com/segment_02.ts
#EXTINF:10,
http://media.example.com/segment_03.ts
#EXTINF:10,
http://media.example.com/segment_04.ts
#EXT-X-ENDLIST
```

The EXTM3U tag indicates the beginning of a playlist file. The EXTINF tag describes the media file identified by the URI in the next line. Its format is:

```
#EXTINF:<duration>,<title>
```

The "duration" attribute specifies the duration of the media file in seconds. The title is an optional field, mainly for human readability.

The media file is indicated by URI in the following line. Each media file has to be a segment of the overall stream. It should at least contain one key frame and enough information to initialize the decoder. The format of the media file can either be MPEG-2 Transport Stream or MPEG-2 audio Stream. The parameters for decoding in one stream should maintain consistency. The following tags are defined to support HTTP Live Streaming: EXT-X-TARGETDURATION, EXT-X-MEDIA-SEQUENCE, EXT-

X-KEY, EXT-X-PROGRAM-DATE-TIME, EXT-X-ALLOW-CACHE, EXT-X-PLAYLIST-TYPE, EXT-X-STREAM-INF, EXT-X-ENDLIST, EXT-X-DISCONTINUITY, and EXT-X-VERSION. Not all tags will be discussed in this section. Only some commonly used ones are covered here.

The EXT-X-TARGETDURATION tag specifies the maximum media file duration. Therefore, the duration of EXTINF tag of each media file should not exceed the maximum duration limitation. Otherwise, error may occur in client side. Its format is:

```
#EXT-X-TARGETDURATION:<s>
```

Each media file URI in the playlist should have a unique sequence number. The sequence number of each URI is equal to the sequence number of the previous URI that it preceded plus one. The EXT-X-MEDIA-SEQUENCE tag indicates the sequence number of first URI in playlist file. If the playlist file does not have an EXT-X-MEDIA-SEQUENCE tag, the first URI has the sequence number 0. Its format is:

```
#EXT-X-MEDIA-SEQUENCE:<number>
```

The EXT-X-ALLOW-CACHE tag indicates whether the client can cache download the media file. It should appear not more than once in the playlist file. Its format is:

```
#EXT-X-ALLOW-CACHE:<YES|NO>
```

The EXT-X-ENDLIST tag indicates the end of a playlist file. It only occurs once in the file. For VoD, the EXT-X-ENDLIST exists when the playlist file is loaded. For live streaming, the client periodically reloads the playlist file until it hits the EXT-X-ENDLIST tag. Its format is:

```
#EXT-X-ENDLIST
```

The EXT-X-STREAM-INF tag indicates the following URI in the playlist is not a media file but *a playlist file*. Its format is:

```
#EXT-X-STREAM-INF:<attribute-list>
```

The playlist file is called *a variant playlist file*.

The example below is a typical variant playlist file:

```
#EXTM3U
#EXT-X-STREAM-INF:PROGRAM-
ID=1,BANDWIDTH=12800
http://example.com/low.m3u8
#EXT-X-STREAM-INF:PROGRAM-
ID=1,BANDWIDTH=25600
http://example.com/mid.m3u8
#EXT-X-STREAM-INF:PROGRAM-
ID=1,BANDWIDTH=76800
http://example.com/hi.m3u8
```

For the EXT-X-STREAM-IN tag, the following attributes are defined: BANDWIDTH, PROGRAM-ID, CODECS and RESOLUTION. The value of BANDWIDTH is the upper bound of the overall bitrate of each media file. It is a compulsory attribute for EXT-X-STREAM-INF. The PROGRAM-ID attribute indicates the unique identifier of a particular presentation. Multiple EXT-X-STREAM-INF tags with the identical PROGRAM-ID value indicate different streams of same content. The CODEC and RESOLUTION attributes are rarely used.

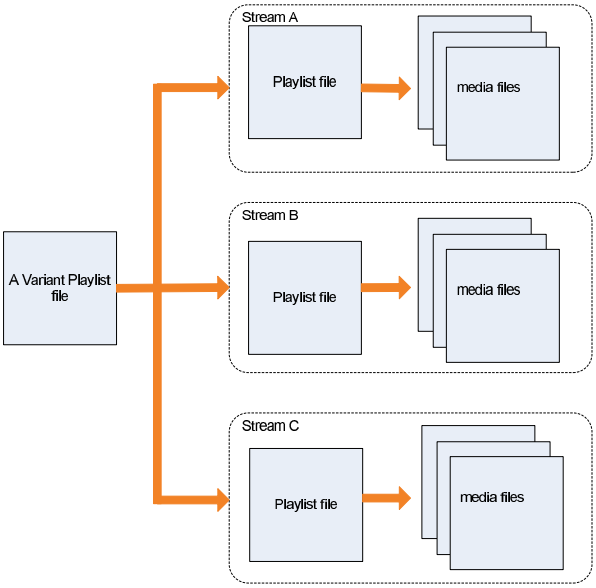

Figure 3.2: Variant playlist file referring to multiple alternative streams

Figure 3.2 illustrates a variant playlist file which refers to multiple alternative streams. Usually, a variant playlist file points to alternate streams of the same content so that it can be used to deliver multiple streams with varying quality levels for different bandwidth. It allows the client to choose the stream that fits best the current networking condition. Based on this mechanism, HLS can achieve an adaptive bitrate streaming system over HTTP.

Media file

Each media file URI identifies a media file, which is a segment of the overall video stream. The media file is formatted as an MPEG-2 Transport Stream or an MPEG-2 audio elementary stream (5).

3.1.3 Server and Client Actions

In section 3.1.1, we illustrated three components of the HTTP Live Streaming system: the server, the distributor and the client. A standard web server can work as a distributor in HLS, without any customized modules. Thus, this section we only focus on new actions of the client and server which support current zoomable video streaming protocol standards.

Server Action

The server is responsible for creating the playlist file and slicing the stream into a series of individual media files. Each media file contains a segment of the stream, in another word, a small time period of the media stream. The details of encoding are outside the scope of our project and will not be discussed in this paper. A valid URI is needed for the playlist file and each media file requires a URI as well so that the user can fetch each media file through the corresponding URI. The format standard has been covered in section 3.1.2. Inside a playlist file, an EXT-X-TARGETDURATION tag is created. Its value should be larger than any duration value in EXTINF Server, should there be no change in this value in stream's lifetime.

For a live stream, the server needs to update the playlist file once a new media file is created by the segmenter. However, the server only needs to follow certain rules while updating the playlist file. Only the following listed modifications are allowed:

- Append lines;
- Remove media file URI. Note the removal should follow the sequence

they appear;

- Change the value of EXT-X-MEDIA-SEQUENCE tag;
- Add or Remove EXT-X-STREAM-INF tags;
- Add an EXT-X-ENDLIST tag to indicate the termination of current stream.

Any changes outside of the above list are prohibited.

A playlist file containing the final media file must contain an EXT-X-ENDLIST tag. If the server wants to remove the entire stream, it should make the playlist file unavailable to the client before it removes the media files. Moreover, all media files should remain accessible to users at least for the duration the playlist file is available.

In section 3.1.2, we introduce the variant playlist file. It allows the server to offer alternative streams of the same content with different bitrates. In a variant playlist file, each alternative stream contains at least one EXT-X-STREAM-INF. Different streams with same content must have identical PROGRAM-ID values. For streams with the same PROGRAM-ID value, the following rules are applicable:

- All streams must present the same content;
- All playlist files for alternative streams must have the same target duration;
- Content which does not appear in all variant playlist files can only appear either at the beginning/ the end of the playlist file;
- Matching timestamps are a must in variant streams;

- All streams should have the same encoded audio bitstream.

In summary, we have covered the server's duty of encoding and segmenting the video stream. The server's actions for both VoD and live streams have been discussed in detail as well.

Client Action

To playback a stream, the client has to begin with downloading a playlist file based on a given URL to identify the stream. The fetching of the playlist file and media files are all over HTTP. For a variant playlist, the playlist files of all alternative streams should also be downloaded.

The client checks the loaded playlist file to ensure it follows the format in 3.1.2. All undefined "#EXT" tags and comments should be ignored. The client is requested to download and decode the media files in advance before the video stream is played. If the EXT-X-ENDLIST tag does not exist in the playlist file, the stream is a live stream. In this case, the client keeps reloading the playlist file periodically until it finds an EXT-X-ENDLIST tag.

If the playlist file contains the EXT-X-ALLOW-CACHE tag with value NO, the client should not cache the obtained media files after they have been played. In the case of variant streams, the client is allowed to switch to alternative streams based on the current networking condition at any point of run-time. However, it should not reload playlist files which are not being played. It should always stop reloading the old playlist file before it starts to load the new playlist file.

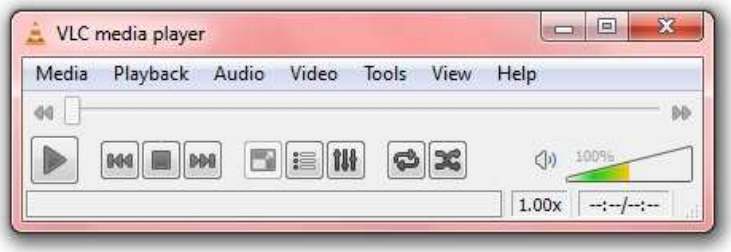

Figure 3.3: The interface of VLC media player

3.2 VLC Media Player

In our project, we implement the zoomable RoI-based streaming playback function using the VLC media player. In this section, a briefing description of the VLC project is given. VLC is an open-source media player and its framework is developed by the VideoLAN Group. The media player works cross-platform. Most common platforms are supported, such as Windows, Linux and Mac.

Figure 3.3 shows the interface of VLC media player. As our implementation of supporting zoomable streaming feature uses VLC, it is imperative to understand VLC's modular structure and architecture, especially the way the media player works with HLS Protocol. A detailed review of VLC supporting HLS is given in this section.

3.2.1 Overview of VLC

The VLC project includes multimedia player, encoder and streamer, supporting various audio and video codecs and file formats as well as many streaming protocols. The numbers of VLC codecs are provided by the *libav-*

codec library and the *ffmpeg* project (2). However, it defines its own muxer and demuxers to take care of the encoding/decoding process.

Modules in VLC

VLC has a very modular design which dynamically loads and unloads modules. This design provides an easier approach to plug-in new file formats, codecs and streaming methods. The core of VLC media player manages the threads and modules, the clock and the low-level controls in VLC. The VLC core links and interacts with modules. Modules can be dynamically loaded and unloaded (3). VLC modules have 2 major properties:

- VLC_MODULE_CAPACITY: category of the module;
- VLC_MODULE_SCORE: priority of the module.

In our project, we will focus our implementation on the following major capacities of modules:

- demux: a demuxer handles different file formats, like ts
- stream filter: a streamer streams media data, like http
- video filter: a video filter adds visual effects to output video, like magnify

Thread and Synchronization

The dynamic loading and unloading modules are achieved by dynamic creation and destruction of multiple threads. VLC is a heavily multiple thread application (1). The threading structure is modeled on *pthreads*. VLC offers wrapper thread functions, listed as: vlc_thread_create, vlc_thread_join,

vlc_mutex_init, vlc_mutex_lock, vlc_mutex_unlock, vlc_mutex_destroy, vlc_cond_init, vlc_cond_signal, vlc_cond_broadcast, vlc_cond_wait, vlc_cond_destroy.

In VLC, the decoding and playing are done by different threads *asynchronously*. It uses Producer/Consumer module to send the media data from one thread to the other. VLC supports multiple input threads for loading multiple files at the same time. However, the current interface does not allow multiple video output threads. The decoder thread is responsible for decoding the media data and computing Presentation Time Stamps (pts) of each frame. The video output threads put frames in sequence and make sure that they are displayed at the right time.

As the VLC media player keeps a heavily multiple threading structure, it is important to understand the methods by which threads communicate with each other. In VLC, there are mainly two methods of communication between threads and modules:

- Message passing: Threads can send control queries to each other. A thread module has a control function to deal with the received queries. If the query succeeds, it will return VLC_SUCCESS to caller. Otherwise, it returns VLC_EGENERIC;

- Object variables: VLC has a powerful "object variable" infrastructure. This structure is created to pass information between modules; There is a callback function which can be associated with variables. The callback function is triggered when the variable changes.

3.2.2 HTTP Live Streaming in VLC

Support for HTTP Live streaming is available in the latest VLC project. Figure 3.4 below illustrates the details of the structure of the HTTP stream module, input module and video output thread.

We discuss the details of each module's function in Figure 3.4:

- Stream filter module (stream.c): The stream filter module loads media files and send them in blocks to other modules. Httplive.c is the stream filter file for HTTP Live Streaming.

- Demux module (demux.c): The demux module is responsible for handling different "file" formats. The demuxer modules pull data from the stream modules. For HLS, ts.c is the demux module file for handling TS files.

- Decoder module (decoder.c): The decoder module is designed to play the mathematical part of the process of playing a stream. Note that the decoder module does not take care of the reconstruction of packets into a continuous stream. The decoded media data will be received through the shared structure decoder_fifo_t.

- Video output module(video_output.c): The video output module gives VLC the ability to display output with almost any hardware, on any platform.

In Figure 3.4, oval indicates the module is a thread. The solid arrow indicates the module keeps a reference to the object that it points to.

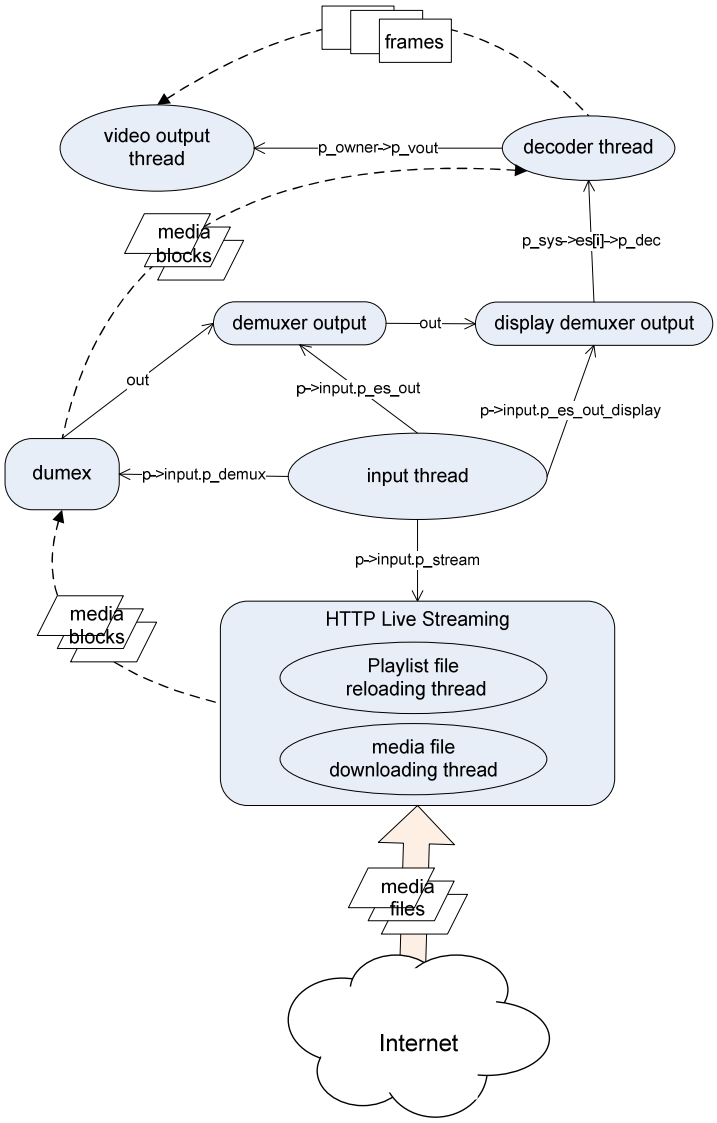

Figure 3.4: VLC module structure of streamer/demuxer/decoder

The steps below show how HTTP Live Streaming Protocol works in VLC:

1. The httplive module is loaded as a stream filter module in VLC if a HTTP video streaming is called; the input thread initializes the corresponding demuxer and decoder modules;

2. TS media files are downloaded in sequence by httplive module;

3. The demux module fetches the media blocks from httplive module and sends it to the decoder thread;

4. The decoder thread is responsible for fetching the media data and preparing decoded frames for the video output thread; It checks the timestamp of each frame to make sure that they can be played at the right time;

5. The decoder thread sends frames to a FIFO buffer between decoder thread and video output thread;

6. The video output thread fetches the frames from the FIFO buffer and displays the frames in sequence to users.

3.2.3 Zoomable Interface in VLC Media Player

The latest VLC media player offers a built-in zooming and panning interface. This function is implemented as a video output filter module, in file magnify.c.

A screenshot of the zooming feature is shown below.

Figure 3.5: Zooming/Panning Interface in VLC

The small window in the top-right corner is a preview window. The entire frame is scaled down and displayed in the preview window. Inside the preview window, a small rectangle represents the RoI region. The user can drag the small RoI rectangle in the preview window to move it. The full screen displays the scaled up RoI in the preview window. Below the preview window, there is a triangular region named the zoom gauge. It allows the user to specify the RoI size. The preview window can be hidden or shown by clicking "VLC ZOOM HIDE".

Chapter 4

System Design for Zoomable Video Streaming

4.1 Introduction

The zoomable video streaming system aims to offer a robust user experience. We want to display the RoI with clarity when the user zooms in. Without always sending the high resolution video stream which occupies too much bandwidth resource, we can encode the original media video stream into different streams with different framesizes, known as different zoomable levels. The zoomable level is marked in order; a low level represents a small framesize. Generally, a low zoomable level stream is displayed when RoI is large. On the other hand, a high zoomable level is used when RoI is small. Thus, the media player selects the most suitable media stream among various zoomable levels based on RoI coordinates and size. The client loads the needed tiled streams from server and displays it to the user. With different

zoomable level streams, we are able to offer zooming and panning operations in a RoI-based video streaming and a clearer RoI is displayed when a user zooms in. Useless tiled streams will not be transferred, and this helps to reduce bandwidth consumption.

4.2 Streaming the Media Data

In section 2.1, we discussed two methods to encode, store and stream video that support dynamic RoI selection: *tiled streaming* and *monolithic streaming*.

Tiled streaming is a straightforward approach of zoomable video streaming. In section 2.1, we presented two tiled streaming solutions, normal tiled streaming and optimal tiled streaming. Normal tiled streaming divides the frame into a grid of aligned tiles with the same tile size, while optimal tiled streaming partitions the frame into a set of tiles with arbitrary tile size.

For monolithic streaming, the server has to analyze the dependencies among macroblocks and maintain the dependent trees of macroblocks. In section 2.2, we have illustrated the macroblock lookup in the dependency trees. For each RoI query, the server has to lookup and pack up the needed macroblocks to send it back to the client. Therefore, we found difficult to offer good scalability at server side. Moreover, constructing the dependency trees can hardly be done in real time, so it is hard to implement it to support live streaming.

Based on the above arguments, we found tiled streaming to be a better solution for our zoomable video streaming system.

Once we decided on using tiled streaming, another question arose. Be-

tween normal tiled streaming and optimal streaming, a choice needed to be made. After the optimization procedure, optimal tiling streaming results in a reduction of transmission data compared to normal tiled streaming. However, the optimization procedure requires information on the RoI's popularity. This data is usually not available for raw video data. Moreover, this procedure is difficult to compute in real time.

Based on the above reasons, and in order to offer a more practical tool, with the expandability to live stream, we found normal tiled streaming to be a more reasonable solution for our zoomable video streaming system.

With tiled streaming, during run-time, for each RoI query, lookup of needed tiles is simple and straightforward. Only the tiled streams that overlap with RoI will be sent to the client.

There comes another common question. Who is responsible for the lookup of the tiled streams? There are two optional designs:

- The server does the lookup. The client sends RoI coordinates to the server and the server sends back the tiled streams that overlap with RoI;

- The client does the lookup. The client loads the information (size, coordinates) of all tiles and finds which tiled streams to load. It sends HTTP requests to the server to fetch the needed tiled streams directly from server.

Among these two options, the second choice does not increase the computation load at the server side when the number of clients increases. Thus, we prefer the second option as it has better scalability.

In summary, we choose tiled streaming to stream the media data in our zoomable video system. The client loads the information on the tiled streams and takes the responsibility of choosing which tiled streams to fetch.

4.3 Architecture of HTTP Live Streaming

In section 3.1.1, we illustrated the architecture of HTTP Live streaming. In the design of a system which has the zoomable feature, we use the same architecture, consisting of: the server, the distributor and the client. Some new behaviors are defined in order to support zoomable video streaming.

4.3.1 Server

The server still has two parts: a media encoder and a stream segmenter. The following steps demonstrate how the server streams a video media:

1. The encoder encodes the raw video stream into different zoomable level streams. Among all zoomable levels, the one with lowest resolution is referred to as *the preview stream*, which does not need to be tiled.

2. The segmenter divides the preview stream into a series of media files and prepares the playlist file of the preview stream;

3. For other zoomable levels, except *the preview stream*, the encoder partitions each video stream into aligned tiled streams and encodes each tile as an independent tiled stream;

4. The segmenter divides each tiled stream into a sequence of media files;

5. The server prepares a playlist file for each zoomable level. The file format will be covered in detail in Section 4.4.

6. Finally, a master playlist file is created for this zoomable video stream.

4.3.2 Distributor

In section 4.2, we have shown a design in which the client is responsible for selecting which tiled streams to download. The server and the distributor are not aware of the RoI's information. Thus, the responsibility of the distributor does not change at all. It only takes care of responding to client requests and delivering media files. The behavior of the distributor is exactly the same as was described in section 3.1.1.

4.3.3 Client

The client is still responsible for determining the media files to fetch, download and decode. Additionally, some new duties are assigned to the client. Firstly, the client needs not only to decide which segment to download but also to decide which tiled stream to download based on RoI information. Secondly, the client needs to handle the synchronization of tiled streams in order to reconstruct a seamless continuous video stream. It must handle the occasional delay or loss of some tiled streams.

4.4 Playlist file for Zoomable Video Streaming

To extend the current HTTP Live Streaming Protocol standard to support zoomable video streaming, additional information tags and new rules are

defined. Details will be discussed in this section.

As mentioned in section 3.1.2, in the HTTP Live Streaming system, in the streaming and playback of media, two types of files are used: the playlist file and the media file.

In a zoomable tiled streaming system, we define a few information tags and introduce new rules of the playlist file. Thus, each zoomable level is encoded as an individual media stream. The encoding and decoding do not depend on data from other zoomable levels. One playlist file contains all the information of one zoomable level stream. A variant playlist file, which contains URIs of the playlist files of each zoomable level is prepared by the server, and is known as *the master playlist file*.

Each zoomable level is encoded in a different framesize. In each zoomable level, the media is divided into multiple tiles and each tiled stream is encoded as an individual media stream. The encoding and decoding of each tiled stream does not depend on other tiles. *Note that we always keep the tile size unchanged for different zoomable level streams.*

Figure 4.1 illustrates a master playlist file for tiled video streaming.

In section 3.1.2, the information tags supporting HTTP Live Streaming was covered in detail. To support tiled streaming in HLS, additional tags now need to be defined: EXT-X-TILE-SIZE and EXT-X-TILED-STREAM.

The example below is a typical master playlist file for tiled streaming:

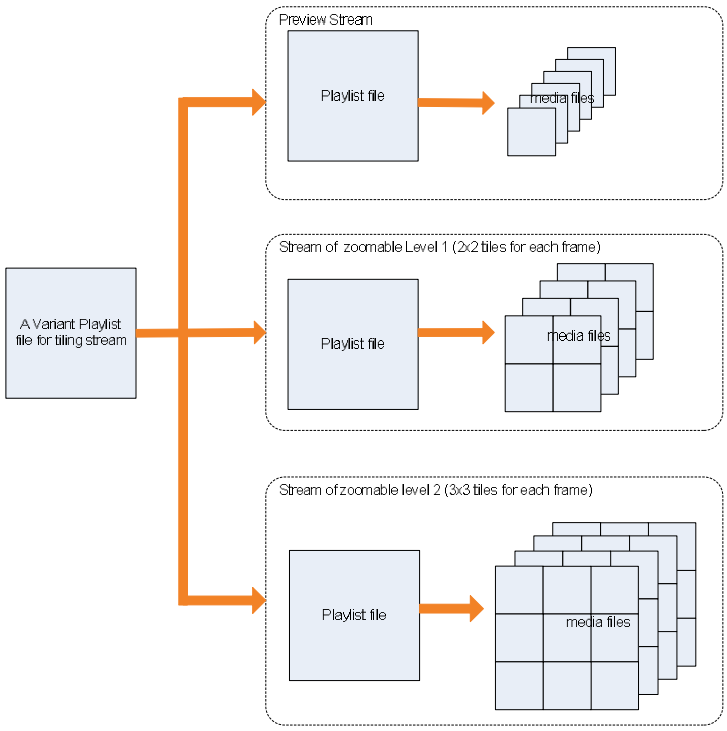

Figure 4.1: Variant playlist file referring to multiple zoomable tiled streams

```
#EXTM3U
#EXT-X-TILED-STREAM:
#EXT-X-STREAM-INF:PROGRAM-
ID=1,BANDWIDTH=1280000
http://example.com/preview.m3u8
#EXT-X-STREAM-INF:PROGRAM-
ID=1,BANDWIDTH=1280000
http://example.com/level1.m3u8
#EXT-X-STREAM-INF:PROGRAM-
ID=1,BANDWIDTH=1280000
http://example.com/level2.m3u8
#EXT-X-STREAM-INF:PROGRAM-
ID=1,BANDWIDTH=1280000
http://example.com/level3.m3u8
```

The EXT-X-TILED-STREAM tag specifies that this variant playlist file is a master playlist file for tiled streaming. Its format is:

`#EXT-X-TILE-STREAM:`

In order to reduce the number of playlist files, we put all the necessary information for one zoomable stream into one playlist file. For tiled streaming, each segment consists of multiple tiles. Therefore, a new format needs to be defined. Some rules have to be followed:

- Tiled streams for the same zoomable level must be encoded with the same time duration.

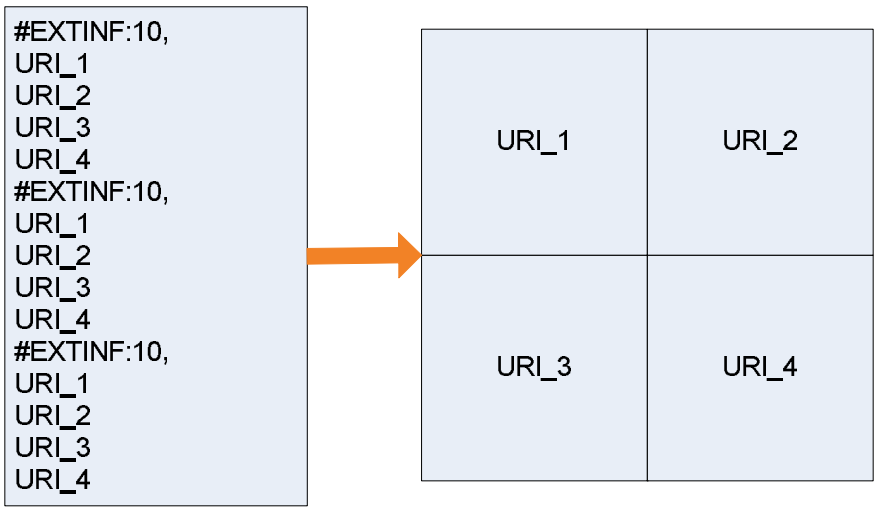

Figure 4.2: One-to-one mapping between URIs and tiles

- A new information tag EXT-X-TILE-SIZE tag is defined. The EXT-X-TILE-SIZE tag specifies the number of tiles in one frame. Its format is:

  ```
  #EXT-X-TILE-SIZE:<width>,<height>
  ```

  This value cannot be changed in the stream's lifetime.

- The EXTINF tag can be followed by multiple URI. The number of lines should be equal to the total number of tiles in the frame. It describes the media files identified by the URIs in the next few lines.

  ```
  #EXTINF:<duration>
  ```

  The "duration" attribute specifies the duration of all the media files.

In the original specification, each EXTINF tag is followed by one valid URI which specifies the media file of the segment. In tiled streaming, each

segment consists of multiple tiles. Therefore, multiple URIs are requested for each segment. Each EXTINF tag is followed by width*height number of URIs and each URI matches one tile. URIs are listed in sequence. The following example illustrates how the one-to-one relationship between URIs and tiles is set up. Take 2x2 tiled streaming as an example. Exactly 4 valid URIs should be listed below each EXTINF tag. The URIs of the tiles are listed from the first row to the last row. In the same row, tiles are always listed from left to right. So the first URI always indicates the top-left tile and the last URI gives the bottom-right tile. Figure 4.2 shows an example of how each URI is mapped to its corresponding tile.

```
#EXTM3U
#EXT-X-TARGETDURATION:10
#EXT-X-MEDIA-SEQUENCE:1
#EXT-X-TILE-SIZE:2,2
#EXTINF:10,
http://media.example.com/segment_01_01.ts
http://media.example.com/segment_01_02.ts
http://media.example.com/segment_01_03.ts
http://media.example.com/segment_01_04.ts
#EXTINF:10,
http://media.example.com/segment_02_01.ts
http://media.example.com/segment_02_02.ts
http://media.example.com/segment_02_03.ts
http://media.example.com/segment_02_04.ts
#EXT-X-ENDLIST
```

The example above is a typical playlist file for one zoomable level in tiled video streaming. In the above example, each frame in a zoomable level stream is divided into 2x2 tiles. For each segment, four valid URIs are given, which matches exactly four tiles.

4.5 Zoomable Interface for Tiled Streaming

Figure 3.5 shows the built-in zoomable feature in VLC. To have the zoomable feature in tiled streaming, we decided to keep the zoomable feature offered by VLC unchanged, in order to offer a consistent user experience. Although there is a huge difference in the underlying downloading and decoding procedures of the original design versus the design for zoomable tiled streaming, we maintain the same interface so that these underlying changes are hidden to the user. The structure of HLS gives rise to some limitations of this zoomable video streaming system. The limitation will be covered in section 4.7.

4.6 Multiple-Threading Streamers/Deumxers/Decoders

The Httplive module supports HTTP Live streaming in VLC. It is responsible for downloading media files in sequence and passing data in blocks to decoder threads. To realize multiple tiled streaming, multiple threads are needed to handle the downloading and decoding of different tiles. Certain mechanisms are required to synchronize the different threads at some stage. Another essential function is to synchronize and assemble the decoded tiled frame to rebuild an entire frame.

Figure 4.3 illustrates the system design for multiple tiled streaming in VLC.

To support multiple tiled streaming, multiple threads structure applies to the following modules:

- Stream module: A stream module is responsible for downloading the master playlist, the playlist files of each zoomable level and all the media files. For multiple tiled streams, there are two optional designs. The first design dynamically creates new stream threads for each tile. The second design initializes a fixed number of stream modules at the beginning and dynamically takes the responsibility of handling the tiled stream. In reality, the creation and termination of threads consume substantial resources and affects the efficiency of the program, causing an obvious delay in playback. Based on the above reasons, we felt the first design would deliver a better result.

  In the current design, among all the streamers, there is one master httplive stream module. This master stream module has two responsibilities. Firstly, the master thread handles all control queries from other modules, especially control queries from its parent, the input thread, and passes them to its child threads. Secondly, it takes care of downloading the preview stream. Note that *the preview stream* is not a tiled stream, but a normal HTTP stream. Therefore, the position of RoI is irrelevant to the preview stream. The preview stream is always downloaded, decoded and passed to the video output module. We decided to take advantage of this design, and will discuss its details in section 4.7.2.

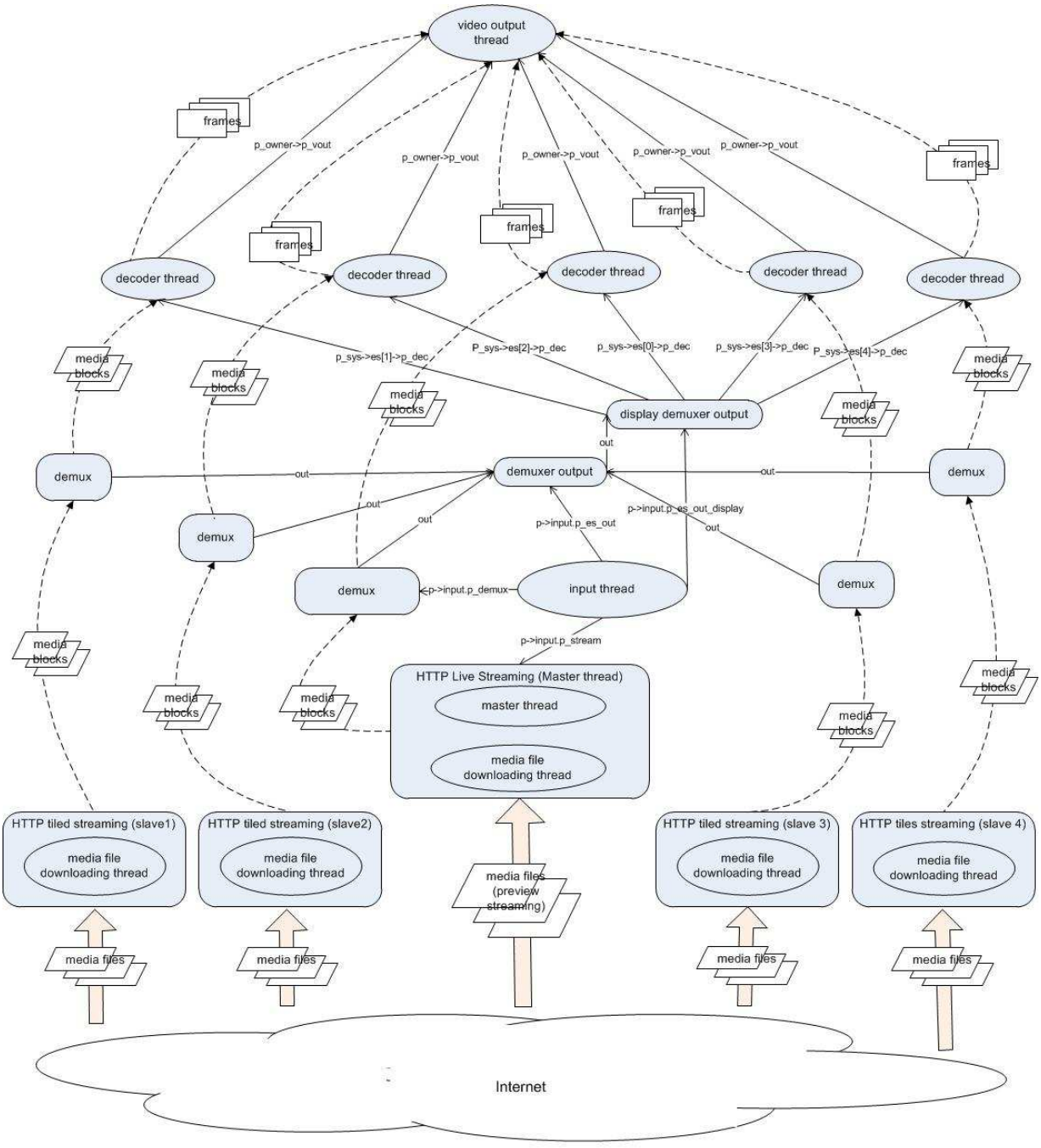

Figure 4.3: VLC module structure of streamer/demuxer/decoder for multiple tiled streaming

- Demux module: A demux module fetches media blocks from the stream module and passes them to the decoder module. It is responsible for processing some control queries from the input thread too. In order to follow the architecture of the original design, we decided to maintain a one-to-one correspondence between the streamer, demuxer and decoder. Unlike a stream module, a demuxer works individually and is not assigned any master control responsibility. They all work as peers.

- Decoder module: Decoder modules do the mathematical calculations. The conversion of media block data to video frames is done within decoder threads. As there is one-to-one mapping between the decoders, streamers and demuxers, we will initialize a fixed number of decoders at the beginning. Similar to the demuxer, all decoder threads work as peers; no master is required. However, as we can observe from Figure 4.3, there is only one video output module. Therefore, with multiple decoders and a single video output module, it is a challenge of critical section (CS) design. Various rules have been set up to deal with this issue and details will be covered in section 4.6.2. Except for the passing of video frames to the video output module, decoders work independently to decode the tiled frames. We do not impose any synchronization control mechanisms between them.

Besides the multiple-threading of streamers, demuxers and decoders, new designs are introduced for the other modules:

- Video output thread: In the original design, it receives decoded video frames from the buffer and displays it in the right sequence on screen.

Due to the structure of the multiple-thread decoders and single video output module, a critical section design sits between the video output thread and multiple decoders. Details of this will be covered in section 4.6.2.

- Magnify module (video output filter): Previously, the magnify module works as a video output filter. It receives decoded frame from the video output and does the resizing and cropping work in order to display the scaled up RoI. In a tiled streaming system, not the entire frame data is downloaded. Only a partial frame, overlapping with RoI, is downloaded, decoded and sent to the video output module. Thus, the magnify module should do the resizing and cropping not only based on the coordinates of the RoI but also include the position of each tiled stream. Details are given in section 4.6.3.

- Magnification control module: VLC simply offers the zoomable feature as a video output filter. The magnify module holds information about the RoI as primitive information as no other module needs to know the RoI. However, in our new tiled streaming system, a few number of modules require RoI information. For example, stream modules need to know the size and coordinates of the RoI in order to decide which tiled stream to download. Working as a video output filter, the magnify module lies at the end of a chain of operations: downloading, decoding, assembling and playing. We need to design a mechanism to pass RoI information from the interactive video filter layer back to streamers without breaking the current structure of VLC project.

Therefore, we create a new *control module*: magnification. The details of how the RoI size and coordinates are passed back to the streamers are elaborated in detail in 4.6.3.

4.6.1 HTTP Stream Module

The stream module takes care of downloading tiled media data. The behavior of the streamer is covered in this section.

Master and Child Threads

As mentioned before, a fixed number of stream modules are initialized and streamers will dynamically decide which tiled stream to work on based on RoI information in our system. We choose this value as four. In total, we have five streamers. One is assigned as the master module and the other four modules function as child streamers. The master stream is responsible for initializing the child streamers. When the master stream loads the playlist, it checks the whether the file contains the EXT-X-TILE-STREAM tag. If no, it works as a normal HTTP variant playlist file. If yes, thereby indicating tiled streaming, the master streamer will call child creation methods in the input thread to initialize new stream/demux/decoder modules. Each child streamer will get an ID when it initialized. Each streamer is responsible for downloading one tiled stream at a time. The master stream is always attached to the preview stream and the child streamers select the tiled stream based on RoI information and its own ID.

During run-time, the master listens to the control queries and broadcasts it to its children while downloading the preview streaming, Figure 4.4

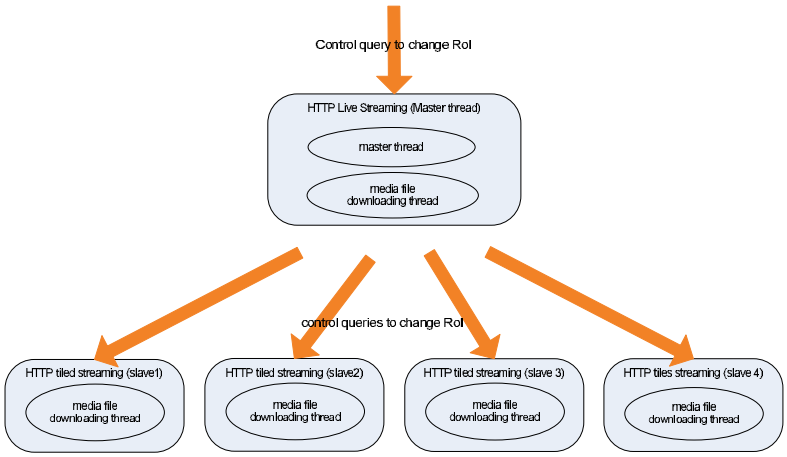

Figure 4.4: Message passing between master and child stream modules

illustrates the broadcasting procedure.

Tile Selection

In section 2.1, we show that the raw video is decoded in different zoomable levels. Each zoomable level consists of multiple tiled streams. Moreover, the distributor works as a standard web server and does not perform any computation related to RoI changes. The client is aware of all tiled streams' information from the master playlist file and it will decide which tiled streams to download based on the current RoI. In our system, we have four child streamers to download tiled streams. There is a simple question: which zoomable level to pick up?

Since there are fixed numbers of child stream modules in the system, we should make sure that the RoI region will always be covered by four tiles. (the tiles have same size at different zoomable levels). We propose a simple approach to maximize the quality of RoI: pick up the highest (with best

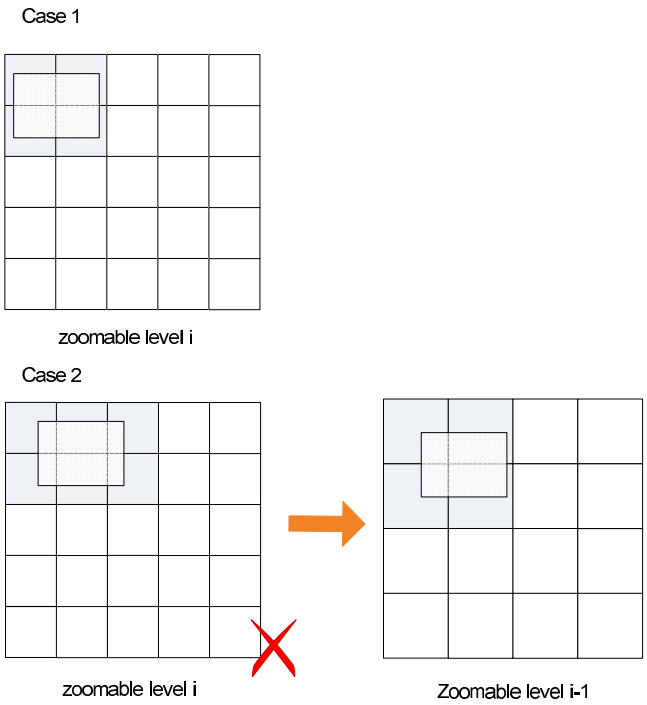

Figure 4.5: Stream modules select zoomable level

resolution) zoomable level that can cover the RoI region with less than four tiles.

Figure 4.5 demonstrates two cases using the same RoI in different positions:

- Case 1: RoI region can be covered by four tiles in zoomable level i;
- Case 2: RoI moves to the right and can no longer be covered by only four tiles at zoomable level i. Thus, we switch to one level lower to i-1. In this level, the RoI can be covered by four tiles as the figure shows. However, we can see that there is an obvious drawback of the design. If the user drag the RoI from the left size all the way the the right edge of the frame, we can easily find the zoomable level switch between i and

i-1 back and forth for many times. A better solution is to maintain 9 streams. Thus, in any case, without change the RoI size, any RoI can be covered by 9(3*3) tiles, For RoI with same size, no zoomable level switch will be triggered, which seems a much ideal solution. However, due to the memory limitation, the maximum number of streamers that we can initilize in VLC is 5. We keep 4 slave streamers in our implementation. Moverover, due to the length of each ts file(typically a few seconds), although the streamers will keep receiving RoI update, the zoomable level switch only happens every few seconds. In reality, user's move interactive action, such as dragging usually does not last more than a few second. Thus, the switch between zoomable level does not happen back and forth.

4.6.2 Decoder Module and Video Output Module

In our system, we have multiple-thread decoder modules but only one video output module. All decoders do the decoding work individually and send the decoded frame in sequence to the video output module.

Every time a decoder fetches one block from the demuxer, it decodes the block data into multiple frames. Afterwards, the decoder sends the decoded frames to the video output module. As mentioned in section 4.6, there are multiple decoders competing to send the decoded frames to the video output thread. A critical section (CS) design is used here when multiple decoders pass frames to the video output module. (since the decoder possesses a buffer to store decoded frames, we do not add a duplicated buffer in this CS design).

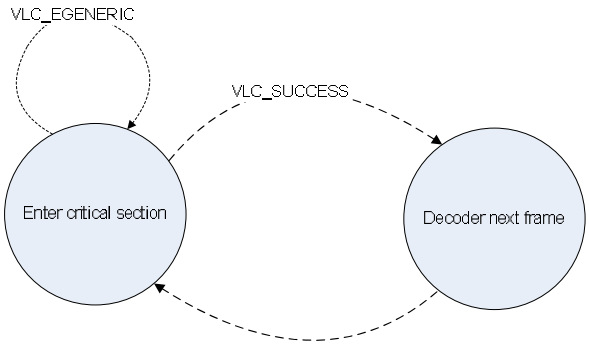

Figure 4.6: Automation of the decoder

The video output threads have a "put_tile_picture" function. This function can only be called exclusively by one decoder at a time. Figure 4.6 demonstrates the automation when the decoder attempts to enter this critical section. When the decoder enters the critical section, it copies pixels from the tiled frame to the assembled frame.

- The decoder tries to obtain the key to enter the function "put_tile_picture" in the video output module;
- If the function returns VLC_SUCCESS (either the tiled frame has been copied to the assembled frame or the tiled frame is too late), the decoder continues to process the next frame;
- If the function returns VLC_EGENERIC, the decoder waits for a while, then tries to enter the CS again.

Figure 4.7 illustrates the decision tree inside the critical section. The video output module will check both the PTS of each tiled frame and the flag of each decoder to make sure all tiled frames are synchronized. Once the assembled frame is ready, it will be pushed to the FIFO buffer queue in the

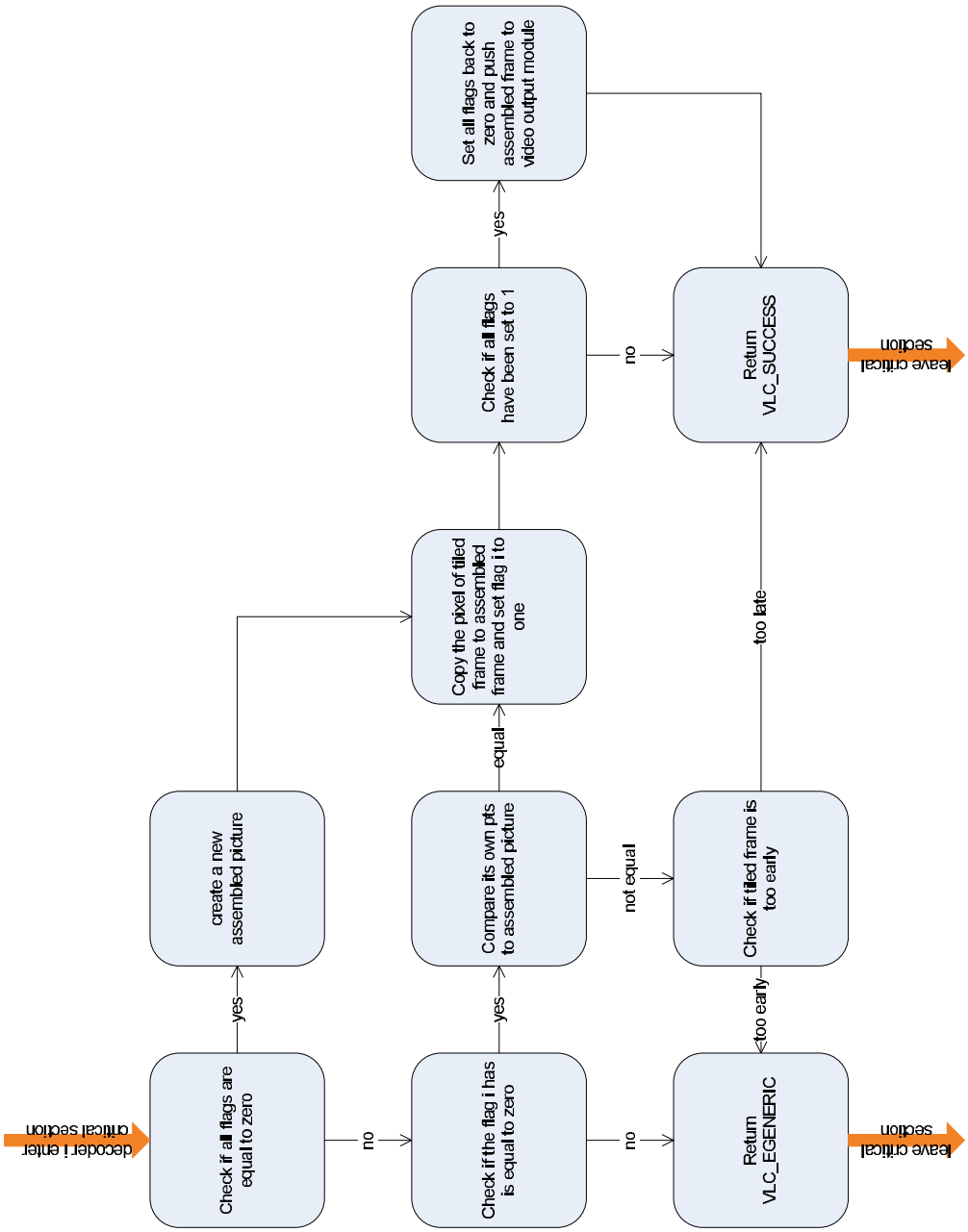

Figure 4.7: Decoder enters the video output module

video output thread. The video output thread will pop up the assembled frame and display them in sequence later.

4.6.3 Magnification Control Module

The procedure below outlines how the magnify module passes RoI information to the stream module.

- Magnify (video filter module): Besides recording RoI information as primitive data, the magnify module uses VLC "object variable" infrastructure to pass RoI information. It attaches RoI size and coordinates as VLC "object variables" to the video output thread.

- Magnification (control module): Working as a control module, VLC core module sends event signals to the magnification module whenever there are user mouse/keyboard interactivities. The magnification module checks the RoI "object variable" attached to video output when it receives the user interactive event signal. If the RoI value has been modified, the magnification module triggers the callback function associated with RoI "object variable".

- "Object variable" callback: The callback function is associated with RoI "object variables". When the magnification module calls the RoI's callback function, the function generates control queries which are sent to the input thread.

- Input control queries: When the input thread receives the control queries from the callback function, it will generate stream control query and send it to the master stream module.

In section 4.6.1, Figure 4.4 illustrated how the master streamer broadcasts the control queries to its child streamers. Therefore, all httplive stream modules possess RoI information. They can determine which tiled stream to work on based on its ID, RoI size and RoI coordinates.

4.7 Selection of Zoomable Levels and Tiled Streams

In the zoomable tiled streaming system, the media file is encoded at different zoomable levels. Each zoomable level is encoded in different framesize and is divided into different numbers of tiles, with the same tile size. When the user zooms in to a particular RoI, the tiled stream at the higher zoomable level is fetched from the server and displayed at client side. When the user zooms out, the system switches back to the lower zoomable level and displays video with lower resolution to the client.

4.7.1 Change of Zoomable Level

Section 4.6.1 describes the system design for the selection of proper zoomable level. But how does the system smooth the transition between different zoomable levels? In section 4.3, we observe that the whole frame is divided into aligned tiles and each tiled stream is stored in a series of media files for each zoomable level.

The child stream module executes the following steps when it receives the control query to change RoI:

- The child stream module checks whether it needs to switch to another zoomable level;

- If yes, the child stream module finds the URIs from the playlist file and fetches the media data for the new zoomable level.

- If no, the child stream then checks whether it needs to switch to another tile. If yes, it switches to a new tiled stream. The details will be discussed in section 4.7.2. If no, it keeps downloading media data from the same tiled stream.

In the HLS system, each tile is encoded in individual streams and each stream consists of a series of TS media files. Each TS file contains a few-second context of a tiled stream. Unlike UDP/RTP streaming, HTTP Live Streaming Protocol does not allow users to randomly access any specified play time. The client can only download segments in sequence or estimate the needed segment based on duration and play time.

To demonstrate zoomable level switching, suppose the video starts at 0 second play time and the duration of each segment is 3 seconds. The stream module receives a zoomable change at 10 seconds of play time. It asks the stream module to switch from zoomable level i to level i+1. There are two choices to handle in this situation:

- Re-fetch the segment media of 9-12s periods of zoomable level i+1. Decode it and the video output will play it once the decoded frame is ready.

- Fetch the segment media of 12-15s periods of zoomable level i+1. Send block data to decoder. The video output will play it from 12 second play time.

The first approach is able to display higher resolution RoI immediately after

the user changes the RoI. However, due to retransmission delay and decoding delay, an obvious delay in playback is caused when the switch of zoomable level is triggered. Unlike the first method, the second method achieves a much smoother transition, but takes longer to display the high resolution RoI. In our opinion, a limited-delay during the switch from low to high-resolution is a less annoying experience compared to a pause in playback. Thus, the second design is chosen for our zoomable video streaming system.

4.7.2 Change of Tiled Stream

In section 4.7.1, we discussed how the zoomable video streaming system handles the switch of zoomable levels. In this section, we will cover the switch between different tiles.

Figure 3.5 shows the zoomable interface in VLC. When user drags the RoI in the preview window without modifying the zoomable gauge, the stream modules do not need to change the zoomable level in most cases. However, the system only has four child stream modules so only four tiles can be downloaded and decoded simultaneously. Thus, if the tiles that overlap with RoI change, the stream modules need to switch the tiled stream which they are working on based using the given RoI information and their ID. As with changing the zoomable level, we have two possible designs in this case. The first approach is to force the stream module to re-download the tiled segment for the current time period. The second approach is to switch the tiles for subsequent segments of play time. Based on similar concerns as before, we prefer the second solution.

Figure 4.8 illustrates how child stream modules change tiles.

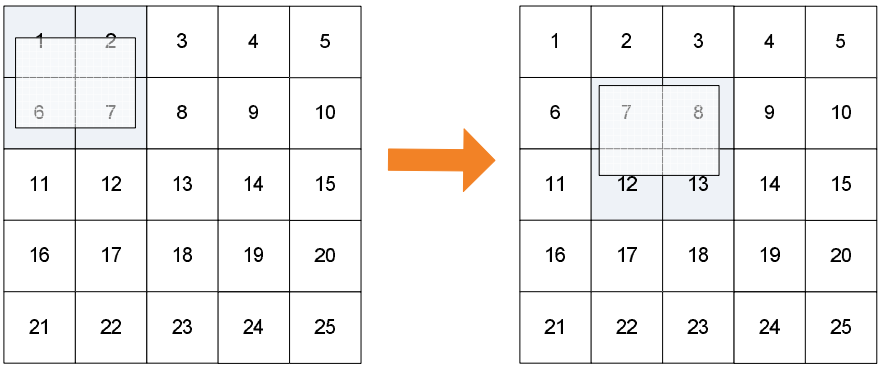

Figure 4.8: Stream modules select tiled streaming

Before the RoI moved, it was overlapping with tile 1, 2, 6, 7. After it moved, it overlaps with tile 7, 8, 12, 13. So the four child stream modules have to switch as follows:

- Slave 1: tile 1 → tile 7;
- Slave 2: tile 2 → tile 8;
- Slave 3: tile 3 → tile 12;
- Slave 4: tile 7 → tile 13;

Unlike switching of zoomable level, there is another issue in the switching of tiled stream. When the user switches RoI, the downloading and decoding of new tiles needs time. There is a delay between the time the switch tile query is sent out and the time the new tiled frame is ready. In the switching of zoomable level, we can replace the RoI with lower resolution frames before the higher one is ready. Take the example shown Figure 4.8; When the user moves the RoI to the right, the visual data of tile 12 and 13 are not available immediately. There is a delay between the receipt of the RoI change query

and the loading of the new tiled stream.

We need to figure out another solution to handle this time gap. In section 4.6.1, we have demonstrated that the master stream module is not only responsible for handling control queries but also for downloading the preview stream. Unlike other zoomable levels, the preview stream contains the entire video image and will not be affected by any RoI change query. In other words, the preview stream is always available no matter how the RoI changes. In order to smoothen the RoI jump, when the RoI needs to be displayed but the new tiles are not yet ready, we propose to use the preview stream to temporarily fill the RoI region. This mechanism can offer a smoother switch when the child stream modules change tiled streams.

Chapter 5

System Performance Study

The main purpose of this project is to deliver a full solution for RoI-based zoomable video streaming over HTTP Live Streaming. This chapter will focus on the analysis of various useful tools that we have developed to realize this system. The advantages and limitations of the system are covered in this chapter. No quantitative studies have been done in our project as no existing system could be used as a reference for comparison.

Figure 5.1 illustrates a comparison between video playback with and without the zoomable feature of tiled streaming. The first snapshot shows the low resolution video playback without the zoomable feature and the second snapshot shows the original High-Definition video playing locally. The last snapshot shows the same RoI region in the zoomable video streaming system. We observe that with the zoomable feature, the media play displays a clearer RoI when the user zooms in. In Figure 5.1, the displayed zoomable level consists of 36 (6*6) tiles. We estimate that loading 4 tiles instead of the whole frame reduces roughly 89% data redundancy, compared to down-

loading the entire frame. An online video is also available in youtube to demonstrate this system. (http://youtu.be/pGxgEIwFqx0)

As we re-use the zoomable interface of VLC, we do not offer any user study anaylsis about whether the interface is easy to use here. The main purpose of the paper is to propose a feasible solution to support RoI based video on HTTP Live Streaming Protocol. For real mobile device, the popular multi-touch zoom-in/out feature is no doubt a better interative method for zoomable video compared to the zoomable interface offered by VLC.

5.1 Support of Live Stream

This project aims to provide a solution for zoomable video streaming. However, in chapter 4, we mainly discussed the system design for VoD video streaming. The original design of HTTP Live Streaming Protocol does not only support pre-computed video streaming, but also live video as well. In this section, we give a brief discussion on the potential to extend our system to support live streaming.

5.1.1 Server and Distributor

We have developed a useful tool to encode a raw video stream in multiple zoomable levels, including one preview stream and several zoomable levels with different framesize. In each zoomable level except the preview stream, the frames are encoded in a grid of aligned tiled streams. This encoding process consists of three steps:

1. The raw video stream is encoded in different zoomable level (with different framesize);

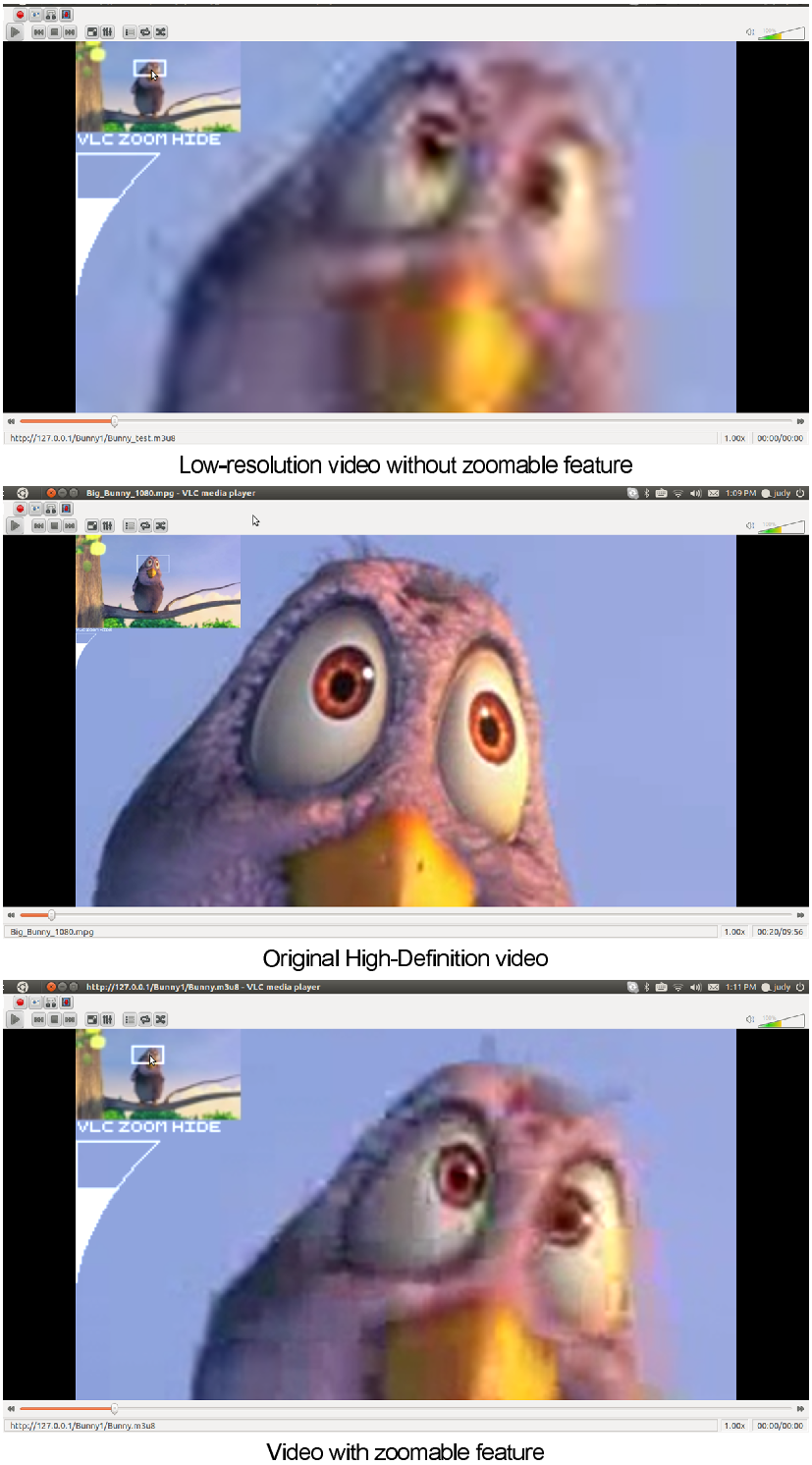

Figure 5.1: Comparison between video playback with zoomable feature and without zoomable feature

2. At each zoomable level, the encoder divides the video stream into tiled video streams;

3. The segmenter divides each tiled media into a series of TS media files;

4. The segmenter prepares a playlist file for each zoomable level;

5. A master playlist file is prepared.

For VoD video streaming, all the above steps are pre-computed and stored at server side. The distributor replies the clients' requests and delivers the demanded media file. Our system needs no more than a standard web server to work as a distributor. We have a client, which loads the master playlist file and the playlist files of each zoomable level. It is responsible for deciding which tile/segment to download. Thus, as long as the web server can handle the clients' download requests, the zooming and panning operations do not demand additional computations from the server.

In this case, supporting live video streaming does not always impose any additional requirements on the distributor. However, for the encoder and segmenter, some tasks must be performed in real time: re-encoding the raw video into different zoomable levels, partitioning each zoomable level into multiple tiled streams and dividing the tiled streams to a sequence of small media files.

Figure 5.2 illustrates this process

1. Server sends A/V input signal to multiple encoders simultaneously through a-multiplexer.

2. Encoders encode the video into small tiles directly instead of encoding into one frame and dividing it into smaller tiles.

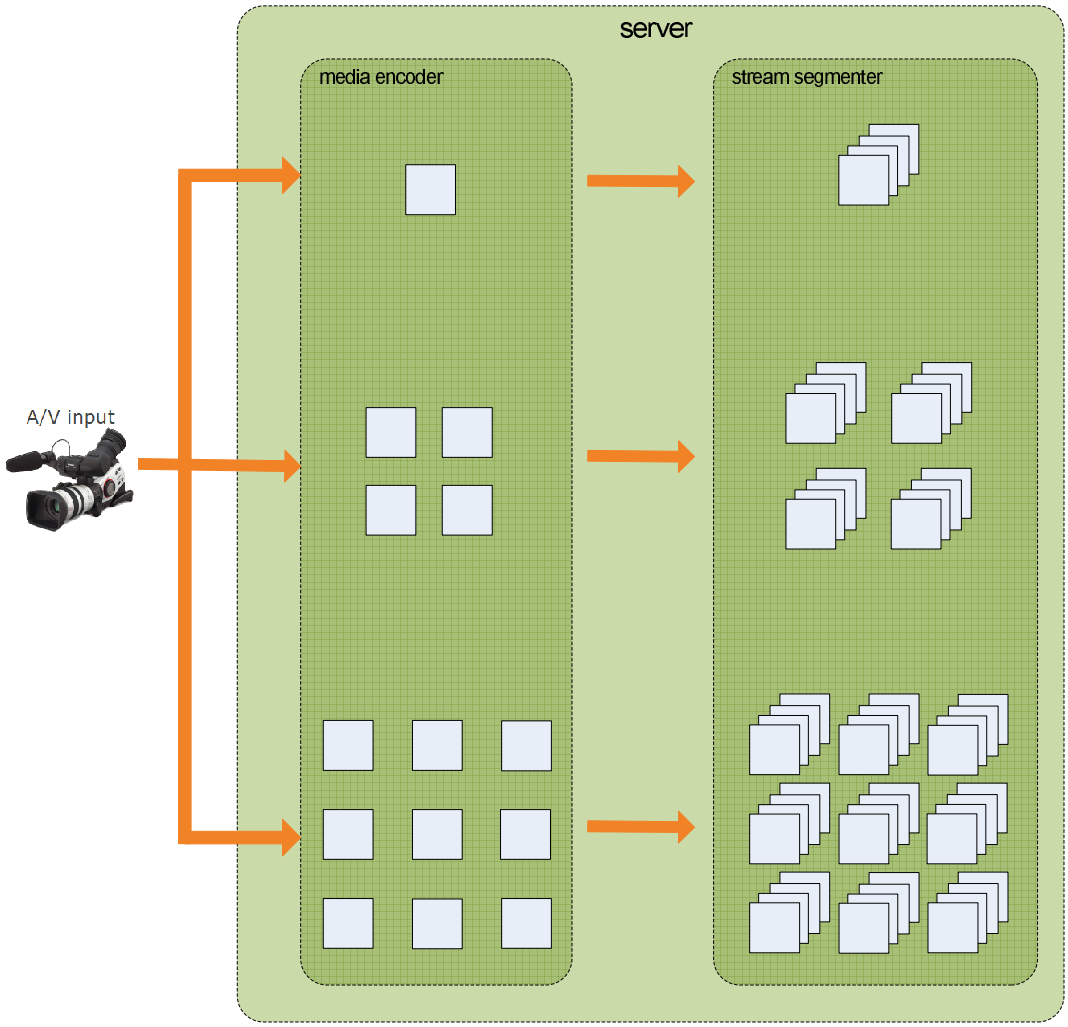

Figure 5.2: Server for zoomable live streaming

3. Multiple segmenters can process the tiled stream simultaneously

For a live stream, the server needs to update the playlist file when new media files are created by the segmenter. The server needs to follow the rules below while updating the playlist file. Only the listed modifications are allowed on the playlist file of each zoomable level:

- Append lines; Note the URIs of multiple tiles after same EXTINF tag should be added at same time;
- Remove tiled file URI. Note their removal should follow the same sequence as their appearance and the tiles in same EXTINF tag should be removed at the same time;
- Add an EXT-X-ENDLIST tag to indicate the termination of current stream.

Any changes outside of the above list are prohibited.

Note that we use *ffmpeg* and an open source *segmenter* in current system, which only works for VoD system. No real tool that supports live stream has currently been developed. Although we do not offer any demonstration for a live stream, this section has illustrated a convincing analysis that our system can be extended to support live stream.

5.1.2 Client

The difference between VoD and live streaming is much simpler in client side. In client side, VLC media player loads the master playlist file and the playlist files of each zoomable level. Generally, the live streaming does not modify the master playlist file in its life time. Thus, the client does not need

to reload the master playlist file. However, the client needs to reload the playlist files of each zoomable level periodically. In tiled streaming, the new information tag EXT-X-TILE-SIZE is defined. We do not allow change of EXT-X-TILE-SIZE values in the stream's lifetime. The server updates the URI of each tile in the zoomable level playlist file and the details of this have been covered in section 5.1.1. The client keeps reloading the playlist file till it reaches the EXT-X-ENDLIST tag.

5.2 Zooming and Panning Operations in VLC

5.2.1 Zoomable Interface in VLC

As mentioned in section 3.2.3, the latest VLC has a zooming and panning interface. Figure 3.5 shows a screenshot of the interface. In our zoomable video streaming system, selecting the zoomable level and tiled stream is automatically done by the stream module based on RoI size and coordinates. We realized a zooming and panning experience similar to the zoomable feature of playing a high resolution video locally. However, due to bandwidth restrictions and some drawbacks of the design, there are a few problems in the system, which we will discuss.

5.2.2 Response Delay of Zooming and Panning Operations

We managed to offer a zoomable interface, but due to the limitations of our system design, occasional delays may occur when a user changes the size of RoI (change of zoomable level) or drags the RoI. A response delay may happen under any of these two circumstances:

- Stream module changes zoomable level;
- Stream module changes tiled stream.

Response Delay of Change in Zoomable Level

In section 4.7.1, we have demonstrated how the stream module chooses the zoomable stream when the RoI size changes. When the stream module switches between zoomable levels, it downloads the consequent segment from the new zoomable level. Therefore, it is a limitation of the system that the zoomable feature will have a few-second delay when a user zooms into the RoI.

The RoI will only be displayed at higher resolution after a few seconds when the current segments ends. This design compensates the response time to a smooth playback when the switch of zoomable level occurs. Fortunately, we can limit this response delay by setting an appropriate segment duration at server side. Generally, the maximum delay is up to the time duration of the current segment. The recommended setting of segment duration in HTTP Live Streaming standard is 10 seconds. However, such a duration is too long for our system. We recommend a segment duration of less than 5 seconds. With a reduced delay, the system can deliver a more an acceptable experience.

Response Delay from Change of Tiled stream

As with switching zoomable level, there is a delay between the receipt of control queries for a RoI change and when the new decoded tileds frame become prepared and ready. Temporarily replacing tiled stream with the

preview stream can smoothen playback when the stream modules switch tiled streams. However, a limitation is that the RoI image cropping from the preview stream has very low resolution. The delay depends on how quickly the user drags the RoI. If the stream module keeps changing the tiled stream, the delay may last longer. General speaking, if the user drags the RoI too quickly at higher zoomable levels, a long delay usually results.

Due to the resource limitation of the project, we are not able to conduct a qualitative user study to collect feedback about the response delay. Based on our observation, the response delay varies with different user behavior and the system setting, such as how often the user will drag/change the RoI, how long the segment length has been set. From the demo video we have seen, as we use the preview stream to cover the gap between the switch request is received and the real data has been fetched, the swtich is still smooth. But the user will experience a few seconds of blur frames.

Chapter 6

Conclusion

In this paper, we have presented a complete solution for zoomable video streaming system over HTTP Live Streaming. We have proposed tiled streaming as the RoI-based streaming method for streaming the media data. In terms of the format standard for the playlist file in HTTP Live Streaming Protocol, new Extended M3U tags were introduced and new rules were defined to support zoomable video streaming. We have used VLC as the media player at client side. Thus, a new architecture of multiple-threading stream/demux/decode modules had to be designed and implemented over the VLC project. We have also conducted a performance study of this zoomable video streaming system and discussed its limitations and the possibility of supporting live stream.

References

Introduction to libVLCcore. http://wiki.videolan.org/LibVLCcore.

VLC Media Player. http://en.wikipedia.org/wiki/VLC_player. VideoLAN.

VLC Modules Loading. http://wiki.videolan.org/Documentation:VLC_Modules_Loading.

Scalable Video Coding Applications and Requirements, 2005. ISO/IEC. ISO/IEC JTC 1/SC 29/WG 1.

Information Technology - Generic Coding of Moving Pictures and Associated Audio Information. In *International Organization for Standardization, ISO/IEC International Standard 13818*, page JTC1/SC29, October 2007.

HTTP Live Streaming Overview. https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/StreamingMediaGuide/HTTPStreamingArchitecture/HTTPStreamingArchitecture.html, 2011. Apple Inc.

D. V. A. Mavlankar, P. Baccichet and B. Girod. Optimal Slice Size for Streaming Regions of High Resolution Video with Virtual Pan/Tilt/Zoom Functionality. In *Proceeding of 15th European Signal Processing Conference (EUSIPCO)*, September 2007.

A. Fecheyr-Lippens. A Review of HTTP Live Streaming. http://andrewsblog.org/a_review_of_http_live_streaming.pdf, 2010.

C. McDonald. HTTP Live Video Stream Segmenter and Distributor. http://www.ioncannon.net/projects/http-live-video-stream-segmenter-and-distributor/, 2009.

C.-T. H. Ming-Chieh Chi, Mei-Juan Chen. Region-of-Interest Video Coding by Fuzzy Control for H.263+ Standard. In *International Symposium on Circuits and Systems, 2004. ISCAS '04*, volume 2, pages 93–6, May 2004.

A. C. N. Quang Minh Khiem, G. Ravindra and W. T.Ooi. Supporting Zoomable Video Streams with Dynamic Region-of-Interest Cropping. In *Proceedings of the ACM SIGMM conference on Multimedia systems MMSYS' 10*, pages 259–270, Phoenix, Arizona, USA, 2010.

W. F. P. Sivanantharasa and H. K. Arachchi. Region of Interest Video Coding with Flexible Macroblock Ordering. In *Proceedings of the IEEE International Conference on Image Processing*, pages 53–56, 2006.

E. R. Pantos. HTTP Live Streaming draft 06. http://tools.ietf.org/html/draft-pantos-http-live-streaming-06, March 2011. Appple Inc.

C. Segall and G. Sullivan. Spatial Scalability within the H.264/AVC Scalable Video Coding Extension. In *IEEE Transactions on Circuits and Systems for Video Technology*, volume 17/9, pages 1121–1135, 2007.