

# *Android Stagefright Overview*



ANDROID

*chunghan.yi@gmail.com, slowboot*

본 문서는 *Android stagefright(gingerbread 기준)*의 기본 구조를 파악하기 위해 작성한 문서로, 인터넷에 떠도는 다양한 그림 이미지를 활용하였으며, 특히 아래 *awesome* 문서를 참조하였음을 밝힌다(원저자의 허락없이, 이미지를 복사하여 사용하였음. 문제가 된다면 말씀해 주세요 ^^).



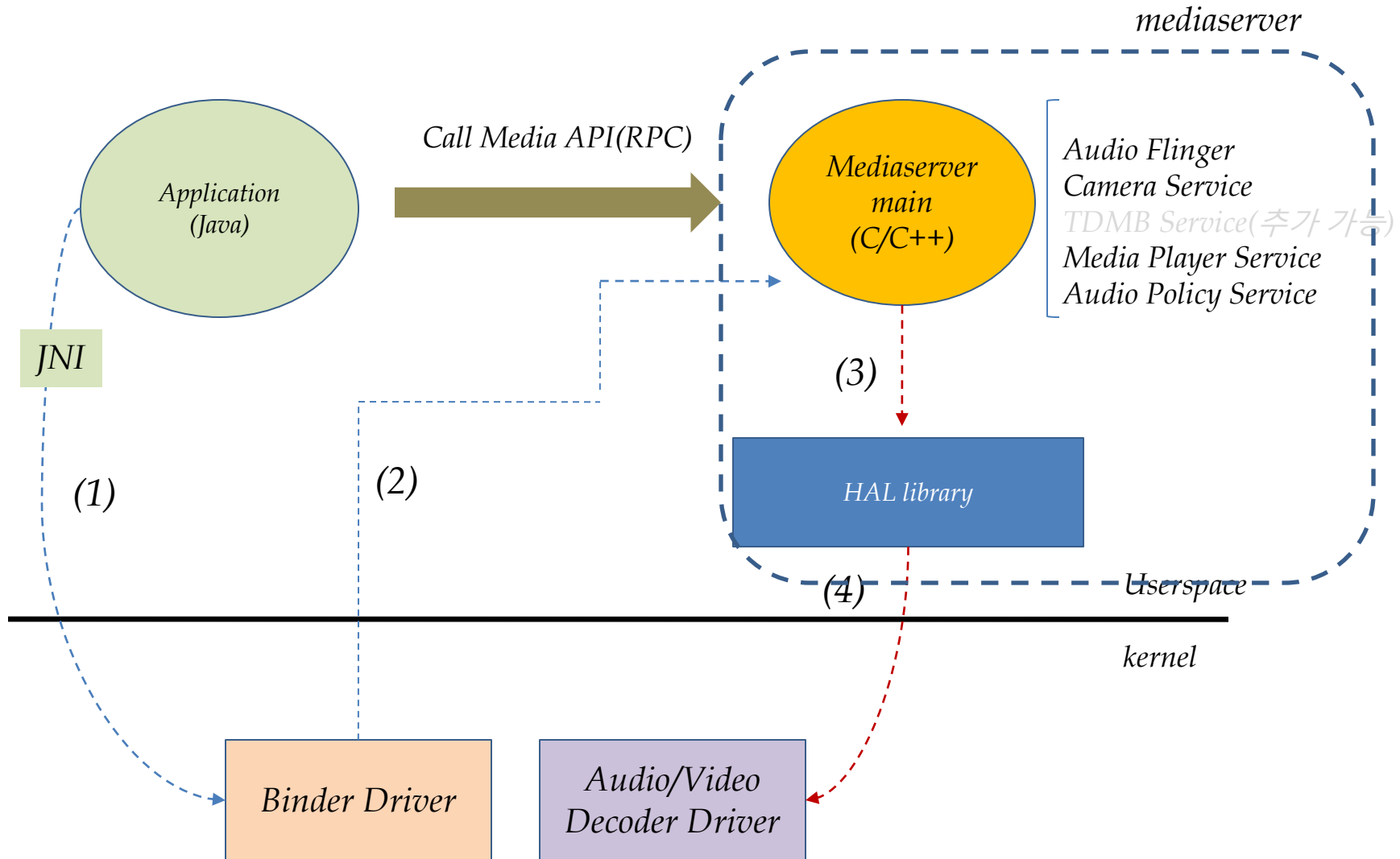
- 1) *01.Android-gingerbread-multimedia-framework-structure.pdf*[이슈 임베디드 포럼의 고현철님]
- 2) *AndroidMMF-Details\_v04.pdf*[windriver의 김태용님]
- 3) *Inside\_of\_Stagefright.pdf*[windriver의 김정호님]



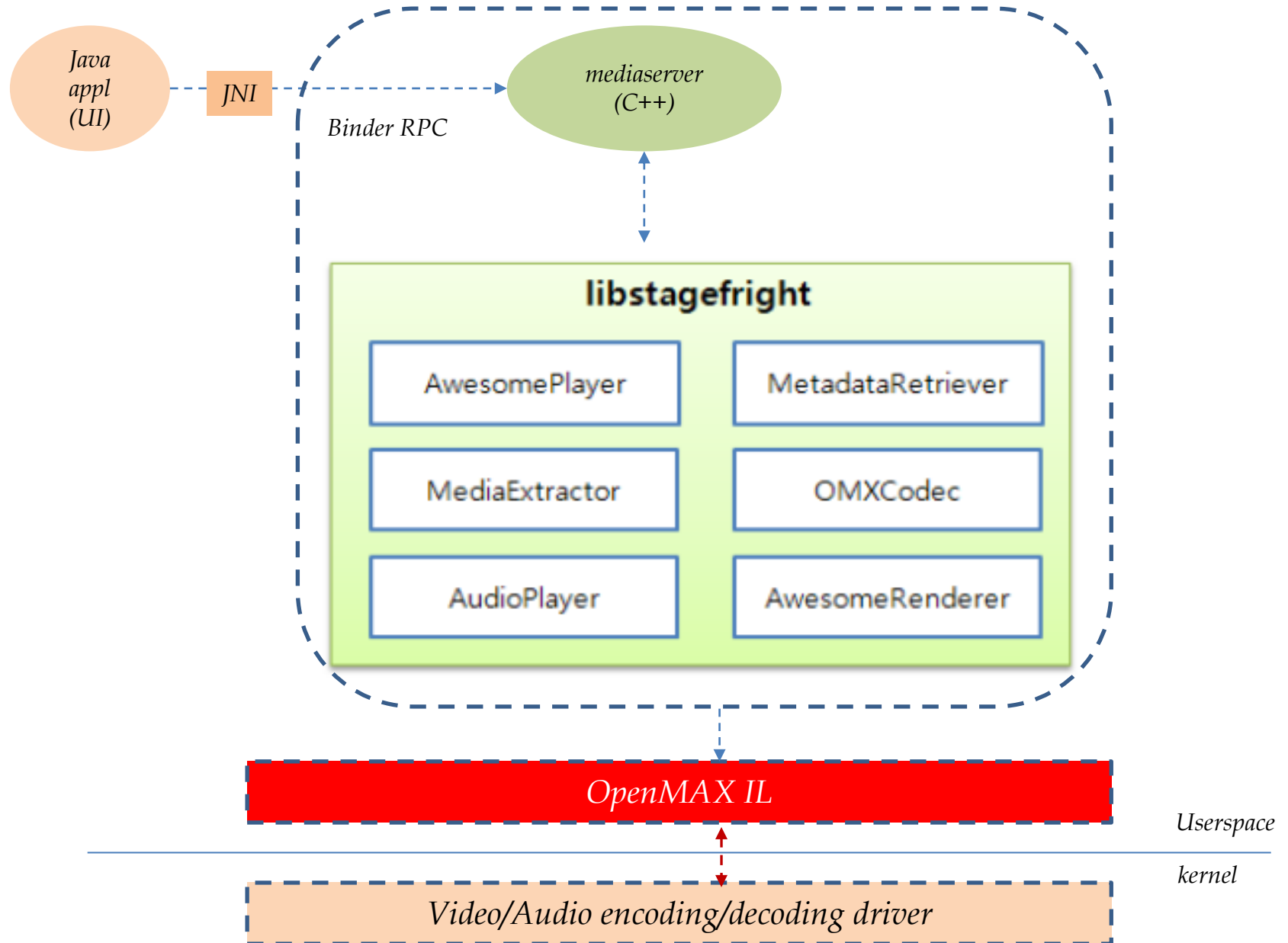
## Contents

- 1. mediaserver overview
- 2. Stagefright 전체 구조
- 3. Audio/Video flow
- 4. Audio Player
- 5. Video Player
- 6. Event Queue
- 7. A/V Sync
- 8. OMXCodec
- 9. Streaming Player
- TODO
- References

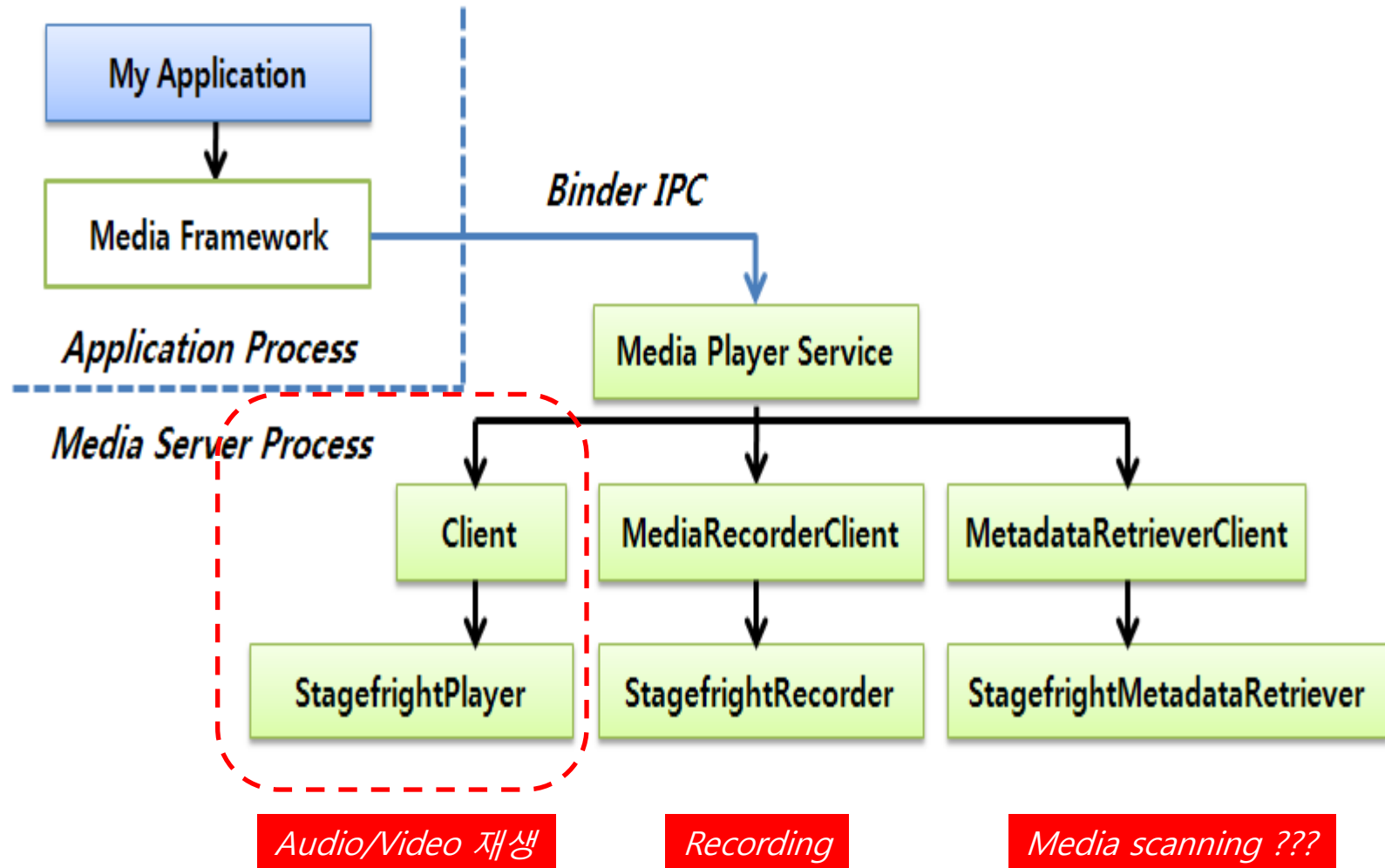
# 1. Mediaserver overview/1



# 1. Mediaserver overview/2

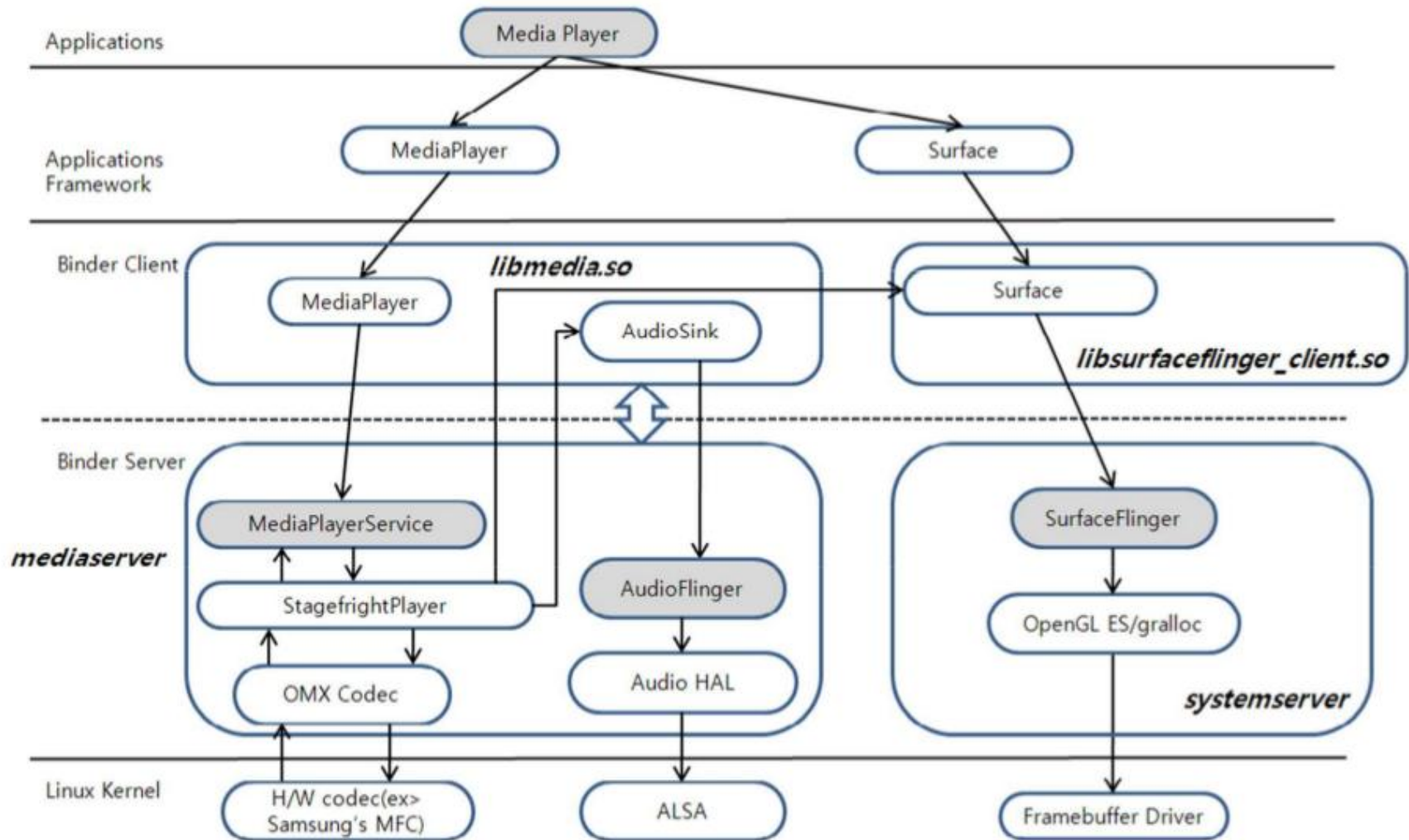


# 1. Mediaserver overview/3



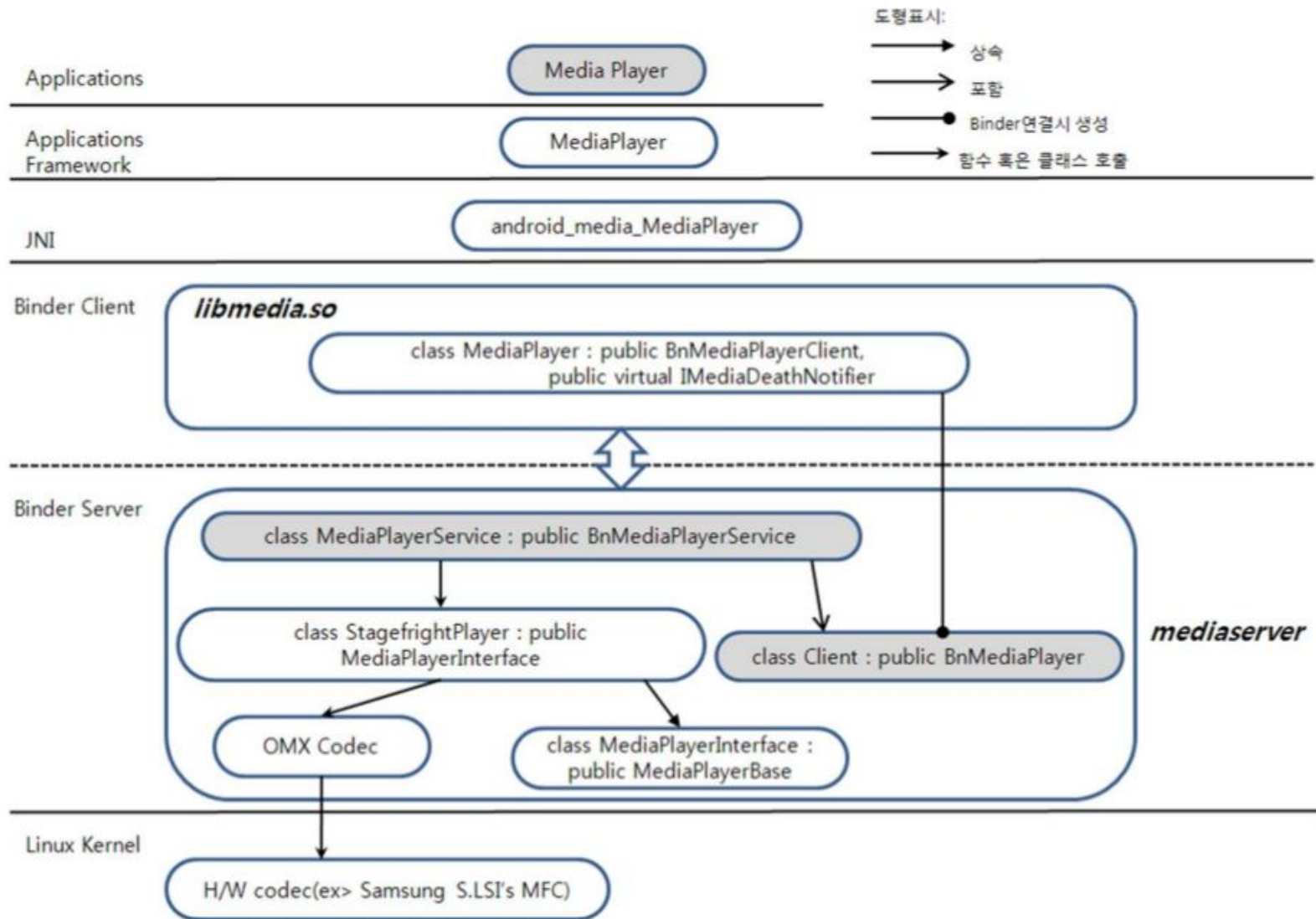
# 1. Mediaserver overview / 4

(\*) StagefrightPlayer/AudioFlinger/SurfaceFlinger간의 관계를 가장 잘 표현한 그림^^



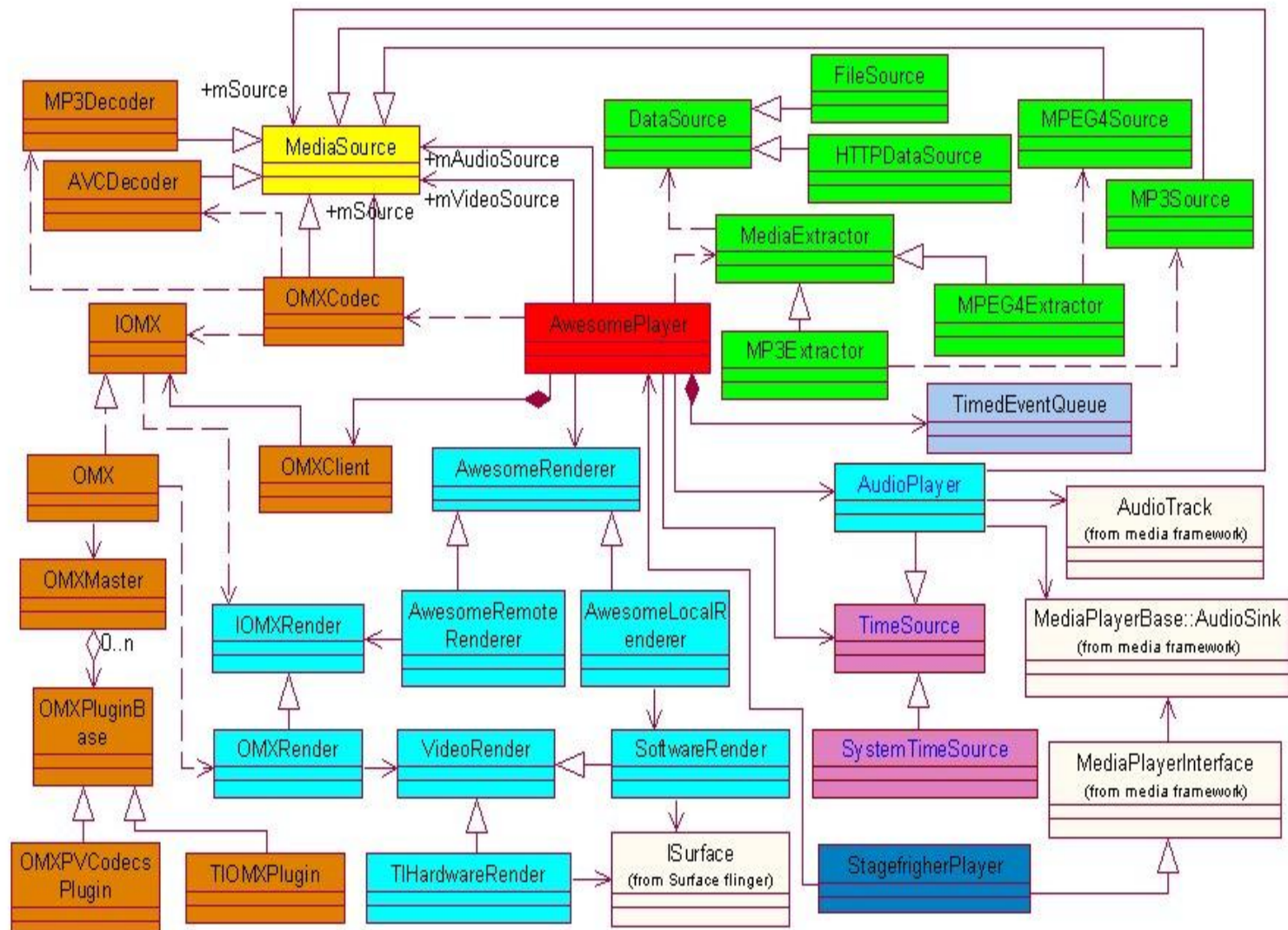


# 1. Mediaserver overview/5



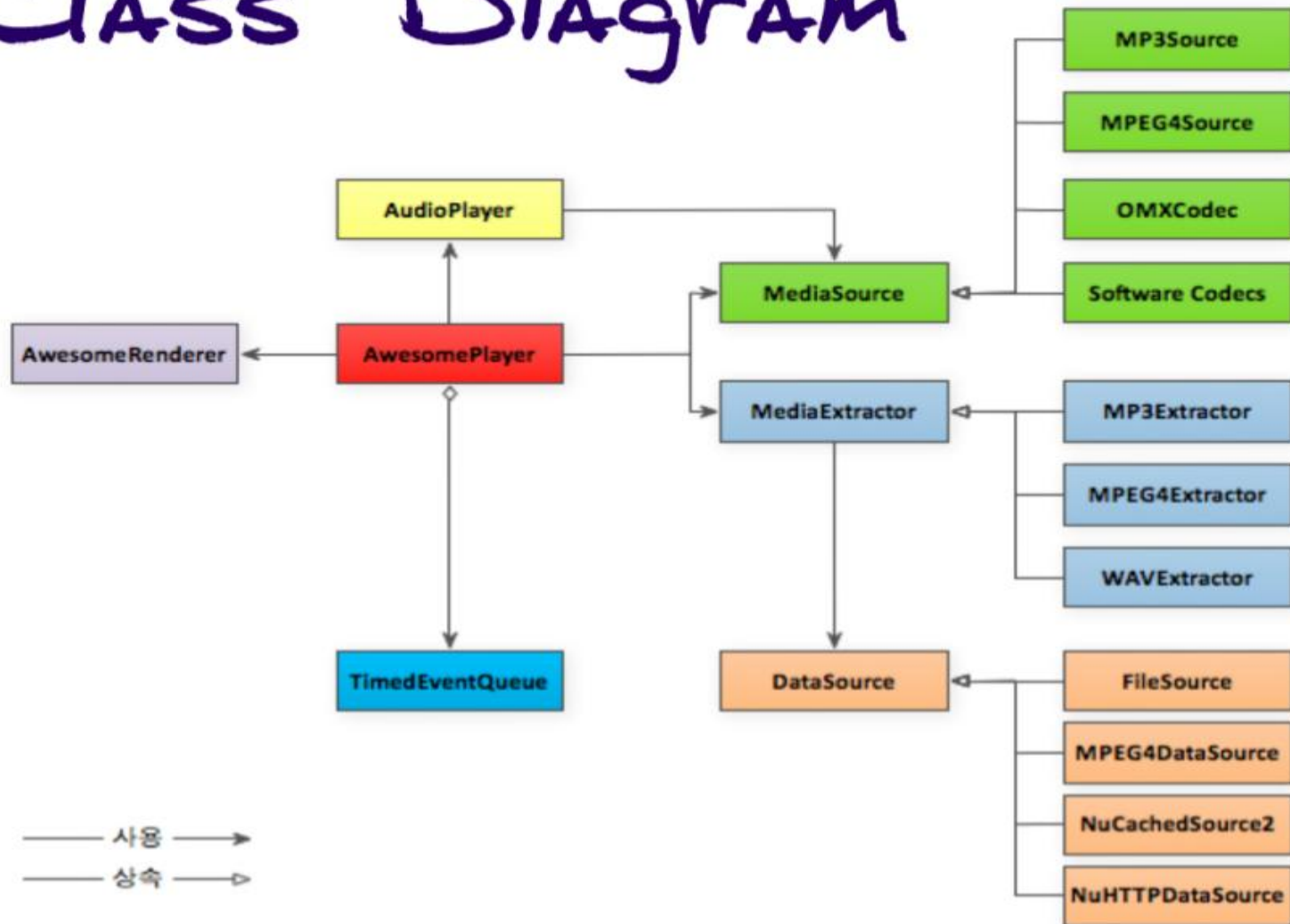
## 2. stagefright 전체 구조/1

(\*) StagefrightPlayer가 진입 포인트이며, AwesomePlayer가 main 역할 !  
(\*) Video는 AwesomeRenderer가 Audio는 AudioPlayer가 담당 ...



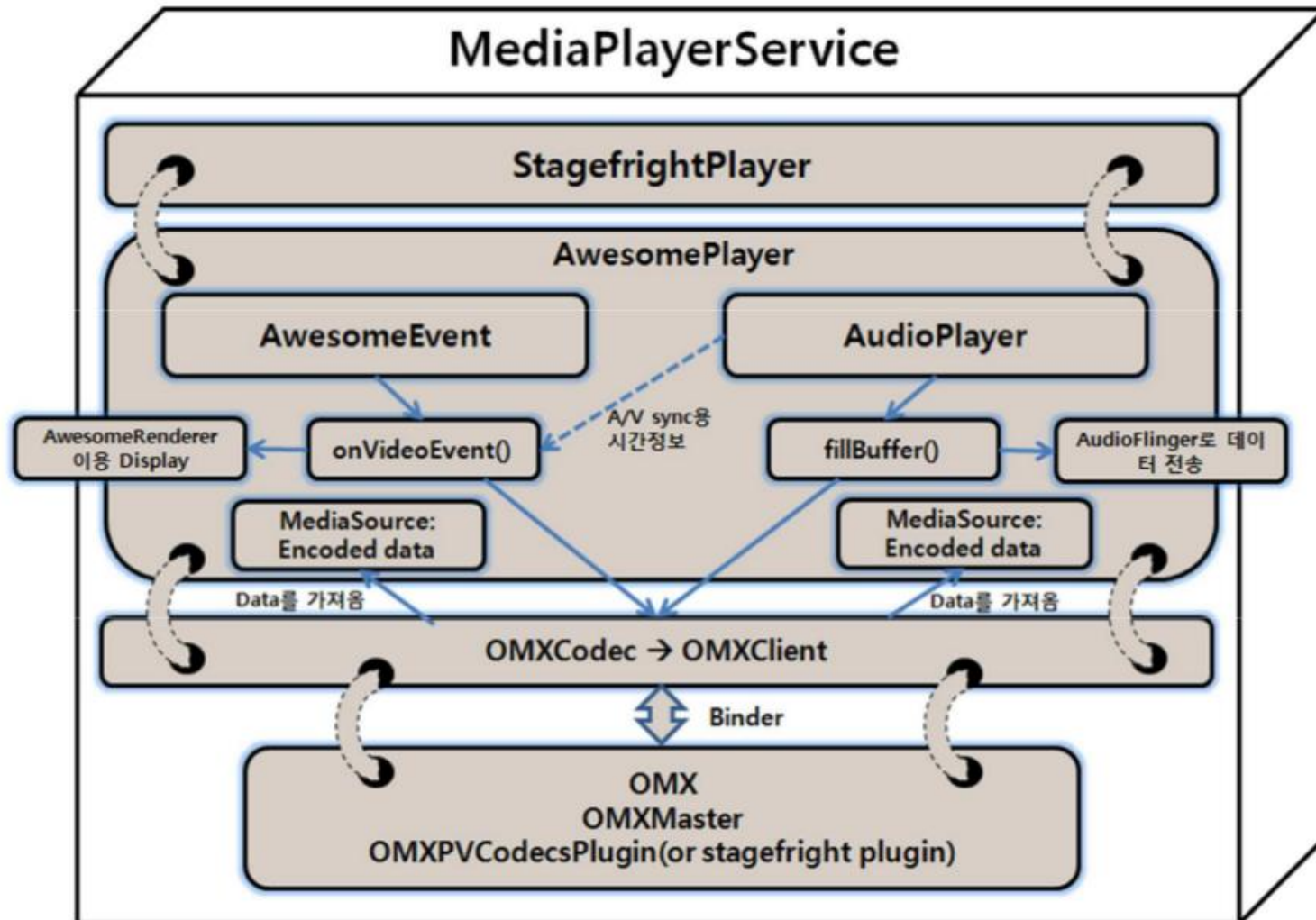
## 2. stagefright 전체 구조/2

# Class Diagram



## 2. stagefright 전체 구조/3

(\*) 문제의 중심은 OMXCodec, onVideoEvent(video) , fillBuffer(audio), 그리고, 이들을 연결하는 event queue !!!  
(\*) AwesomeEvent와 AudioPlayer는 decoding된 data를 처리하는 코드임.



## 2. stagefright 전체 구조/3 - 계속

- 0) 주체 및 연결 고리

- OMXCodec(Decoder 포함): encoding된 data를 decoding해줌. Decoding된 data를 Video, Audio Player에게 전달함(실제 decoding은 하부 단에서 진행^^).
- AwesomeEvent(onVideoEvent): decoding된 video data를 읽어, renderer에 전달함
- AudioPlayer(fillBuffer): decoding된 audio data를 읽어 audioflinger에 전달
- Event Queue: OMXCodec과 Audio/Video player간의 event 교환 용도
- 마지막으로 재생할 media(file 혹은 stream)가 있어야 겠지요^^

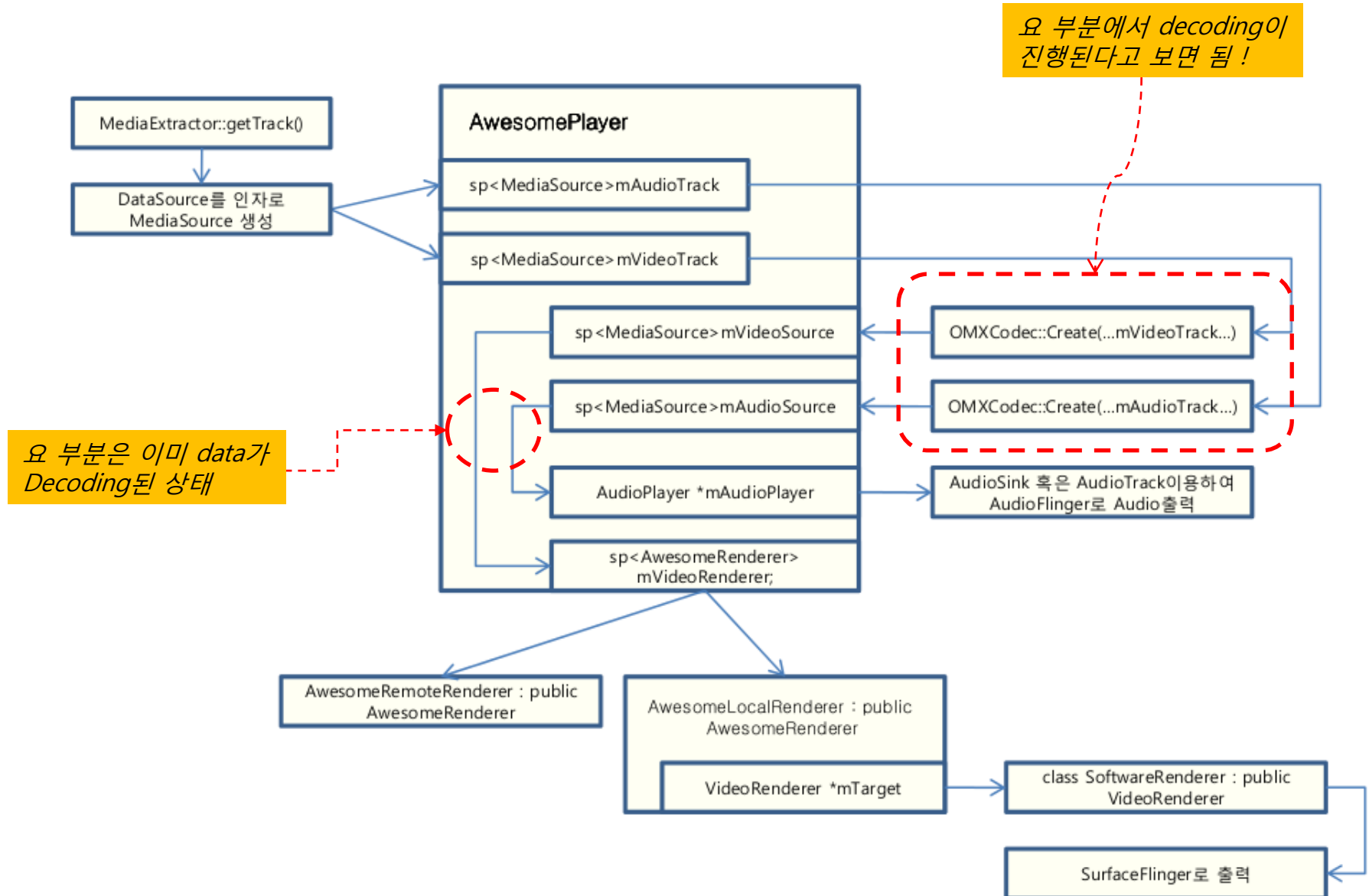
- 1) Encoding된 data(MediaSource)가 OMXCodec(decoder와 연결)으로 들어간다.

- Media Extractor를 통과한 Video, Audio data가 OMXCodec으로 전달된다.
- 이 과정에서 OMXCodec과 EMPTY\_BUFFER를 주고 받는다. 즉, encoding된 data를 decoding할 수 있는 상태가 되었으니, buffer를 채워달라는 의미로 ..

- 2) AwesomePlayer 중 onVideoEvent 함수와 AudioPlayer 코드는 decoding을 거친 data를 Video Renderer와 AudioFlinger로 넘기는 역할을 수행한다.

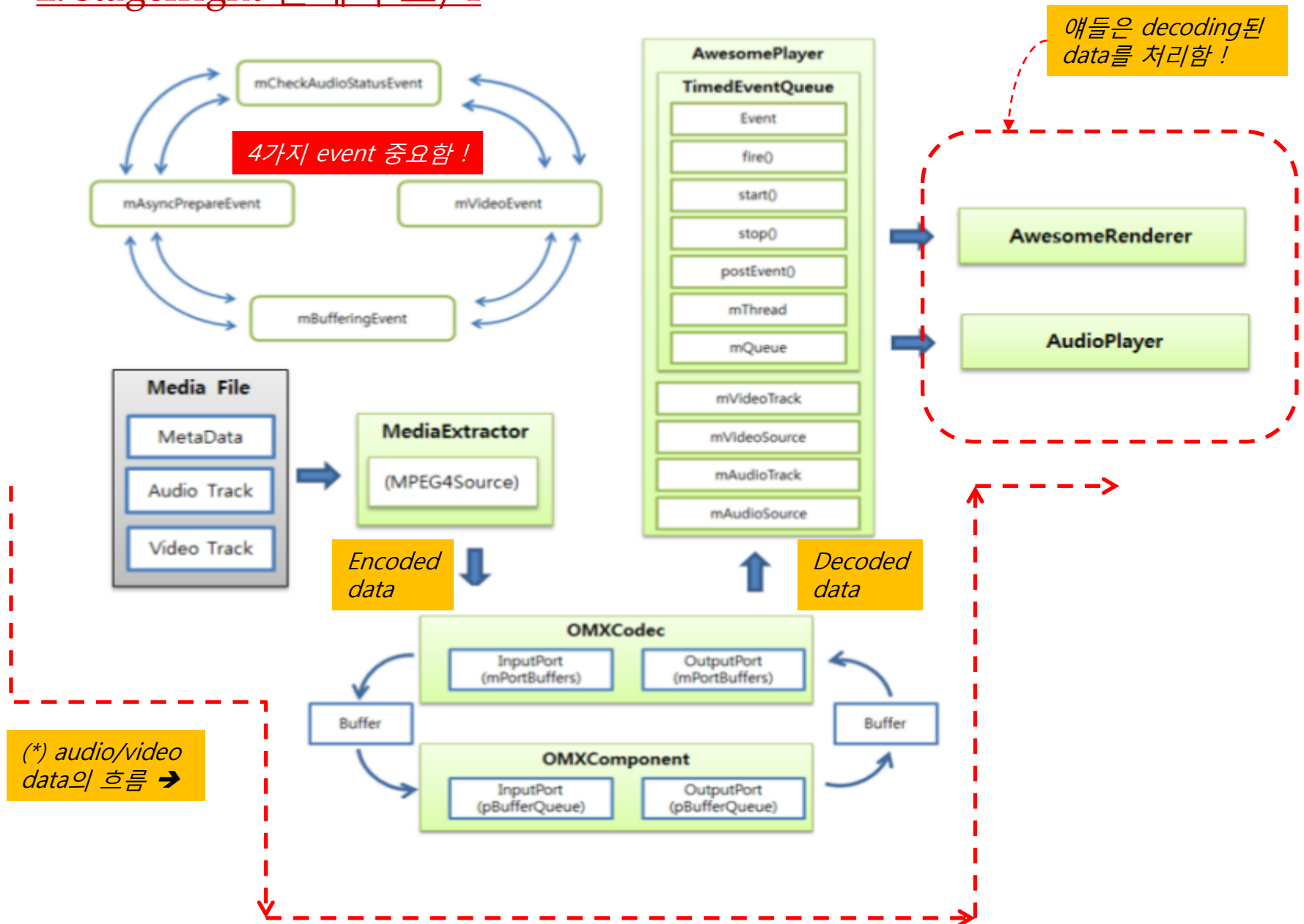
- *onVideoEvent( ) method에서는 decoding video data를 read하여, video renderer로 전달하며,*
- *fillBuffer( ) method에서는 decoding된 audio data를 read(fillBuffer 함수에서)하여, 이를 audio flinger로 전달한다.*
- 이 과정에서 OMXCodec과 FILL\_BUFFER를 주고 받는다. 즉, decoding된 data이 있으니, 읽어 가라는 의미로 ...

## 2. stagefright 전체 구조/3 - 계속





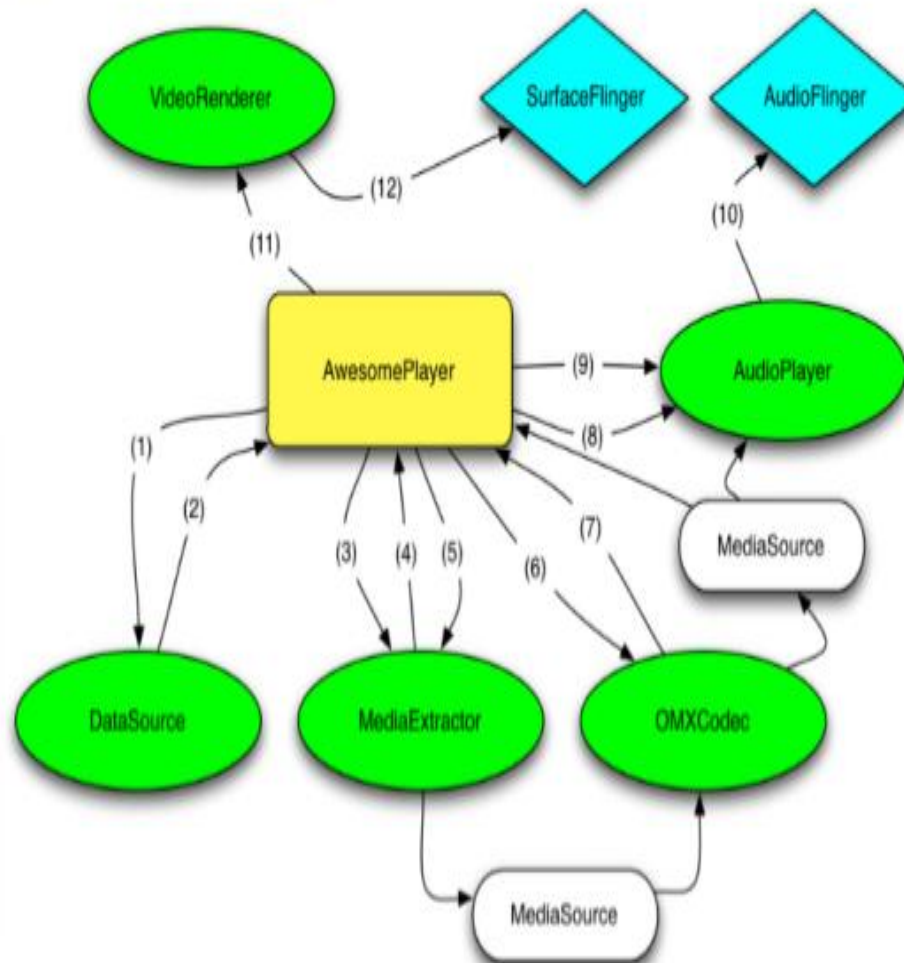
## 2. stagefright 전체 구조/4



## 2. stagefright 전체 구조/5

# Data Flow

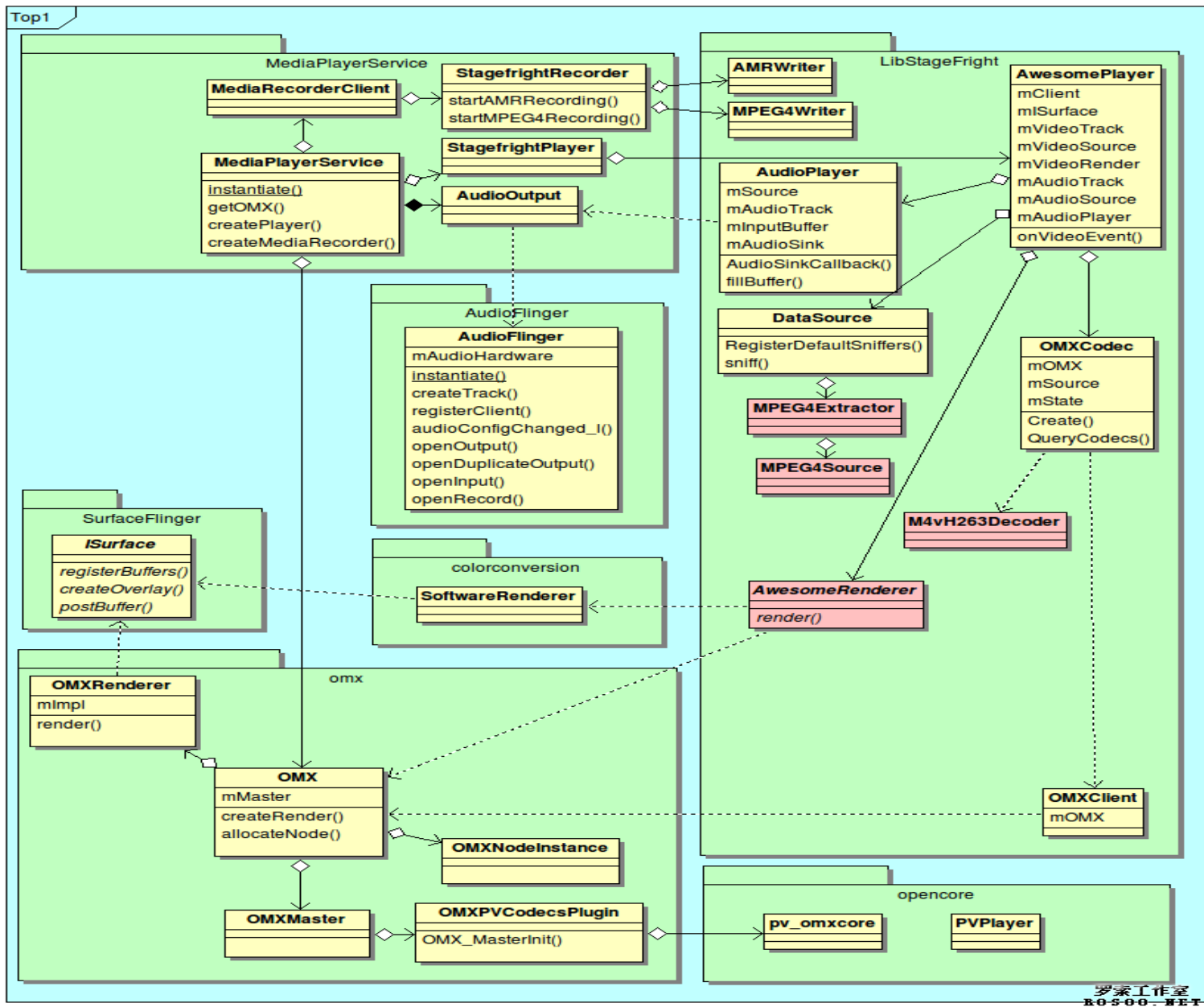
(\*) HTTP로 실시간 stream을 download 받아 Play할 경우의 흐름도 임^^  
(file play의 경우도 크게 다르지 않음)



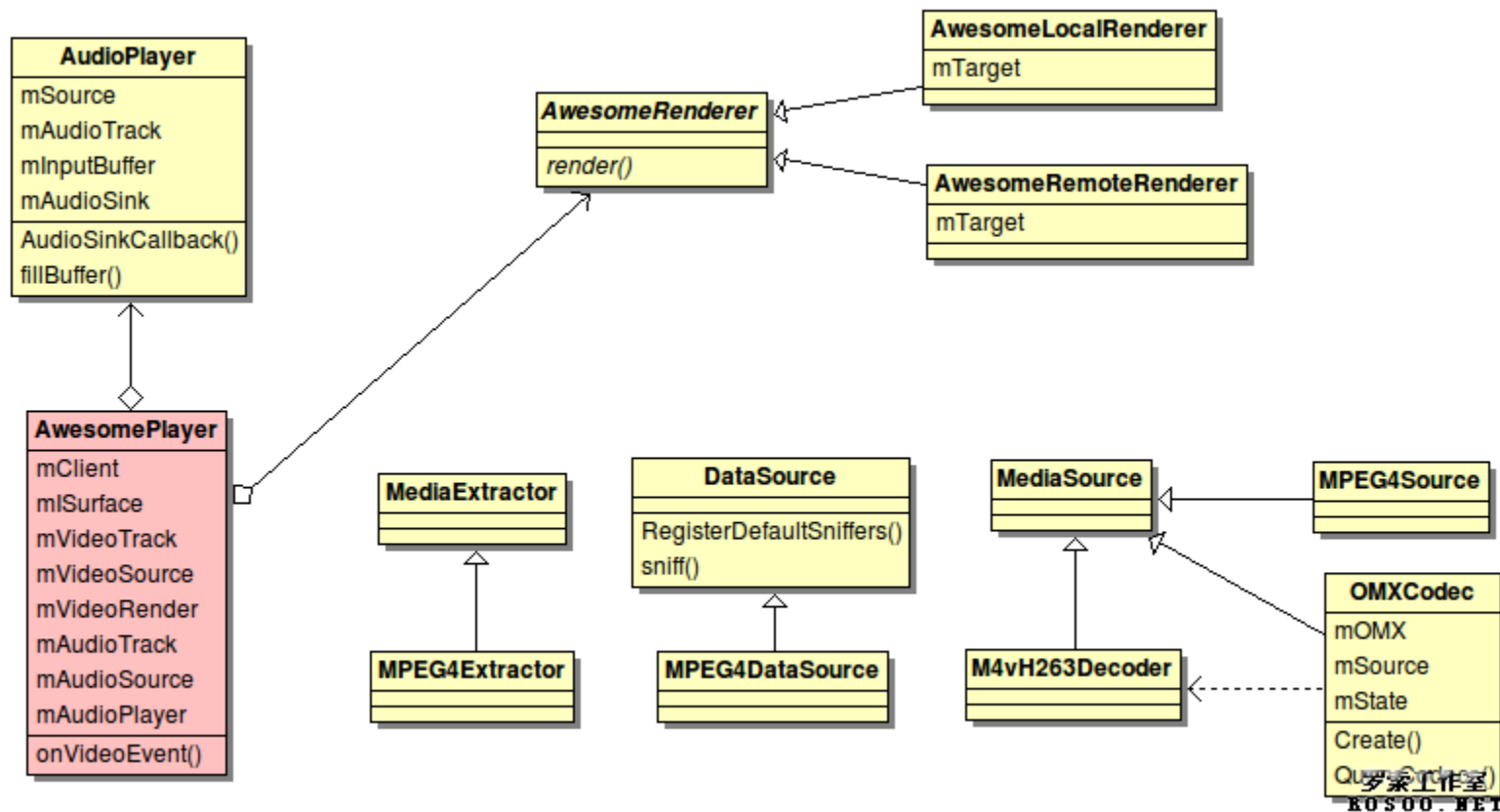
- 1) Create DataSource with URI
- 2) Return NuHTTPDataSource Object
- 3) Create MediaExtractor with NuHTTPDataSource Object
- 4) Return MPEG4Extractor Object
- 5) Call MediaExtractor::getTrack () and Get Audio, Video MediaSource
- 6) Create OMXCodec with Audio, Video MediaSource
- 7) Return MediaSource for Audio and Video codec
- 8) Create AudioPlayer with MediaSource for Audio codec
- 9) Play AudioPlayer and then read RAW audio data from audio MediaSource
- 10) Sink RAW audio data to AudioFlinger
- 11) Get YUV video data from video MediaSource and then render YUV video
- 12) Sink YUV video data to SurfaceFlinger



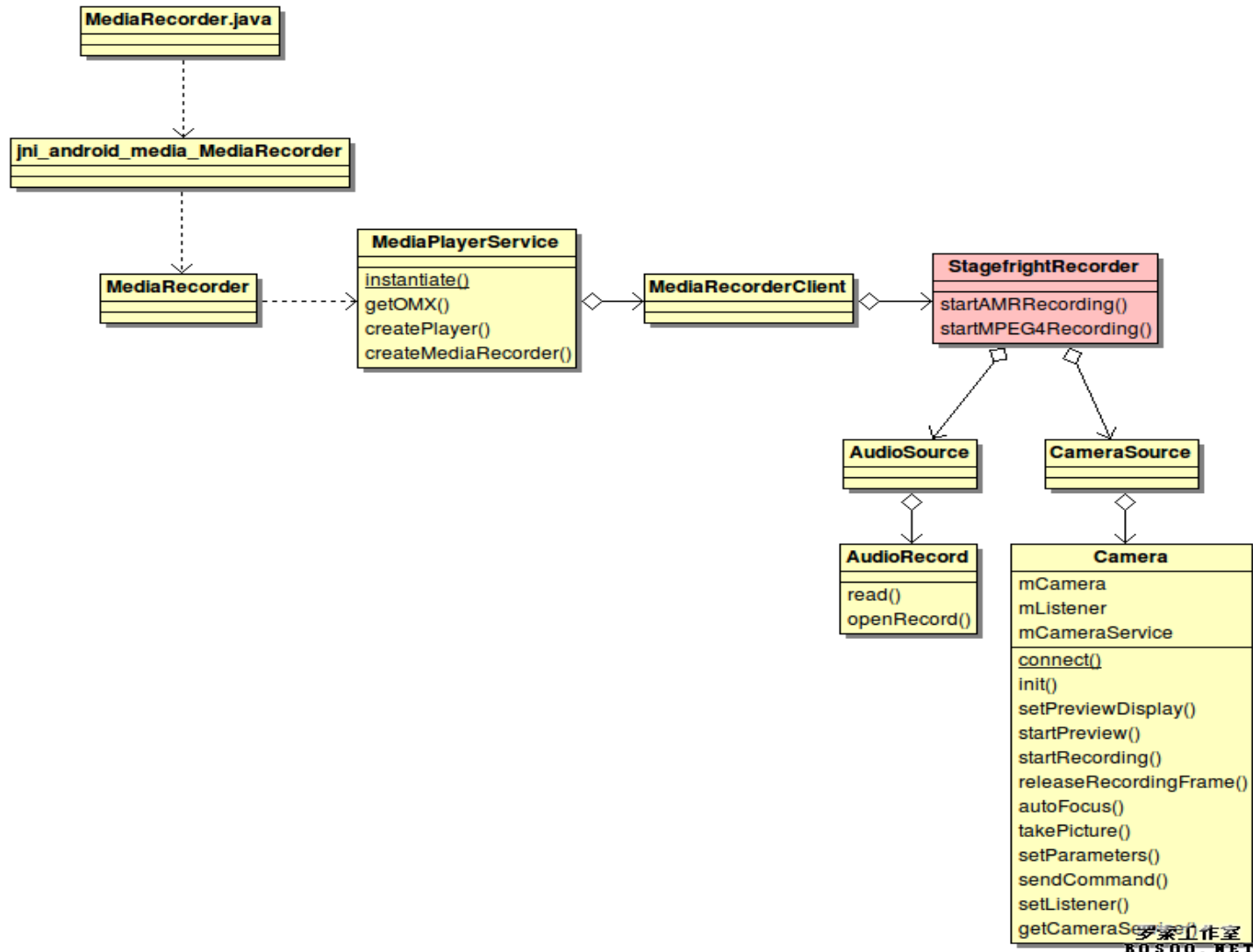
## 2. stagefright 전체 구조/6



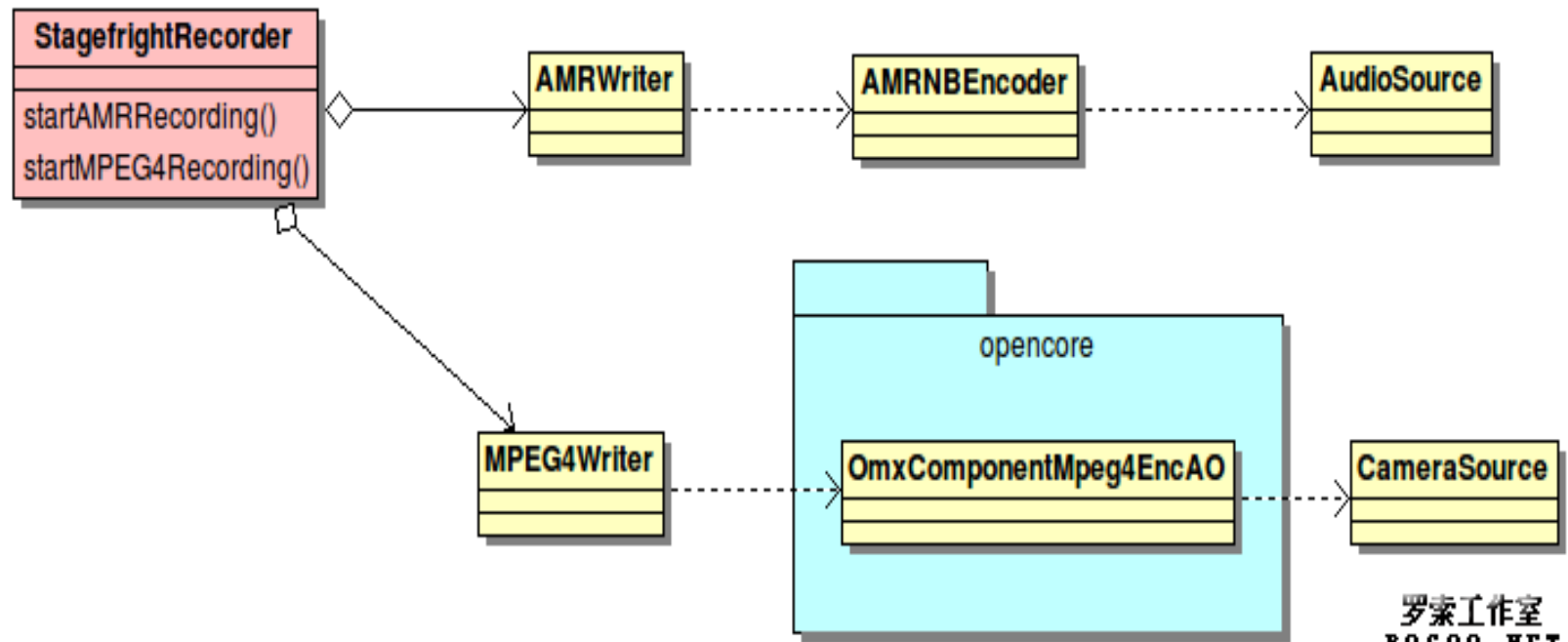
## 2. stagefright 전체 구조/7 - Media Playing/Data Source/Codec



## 2. stagefright 전체 구조/8 - Media Recording/1

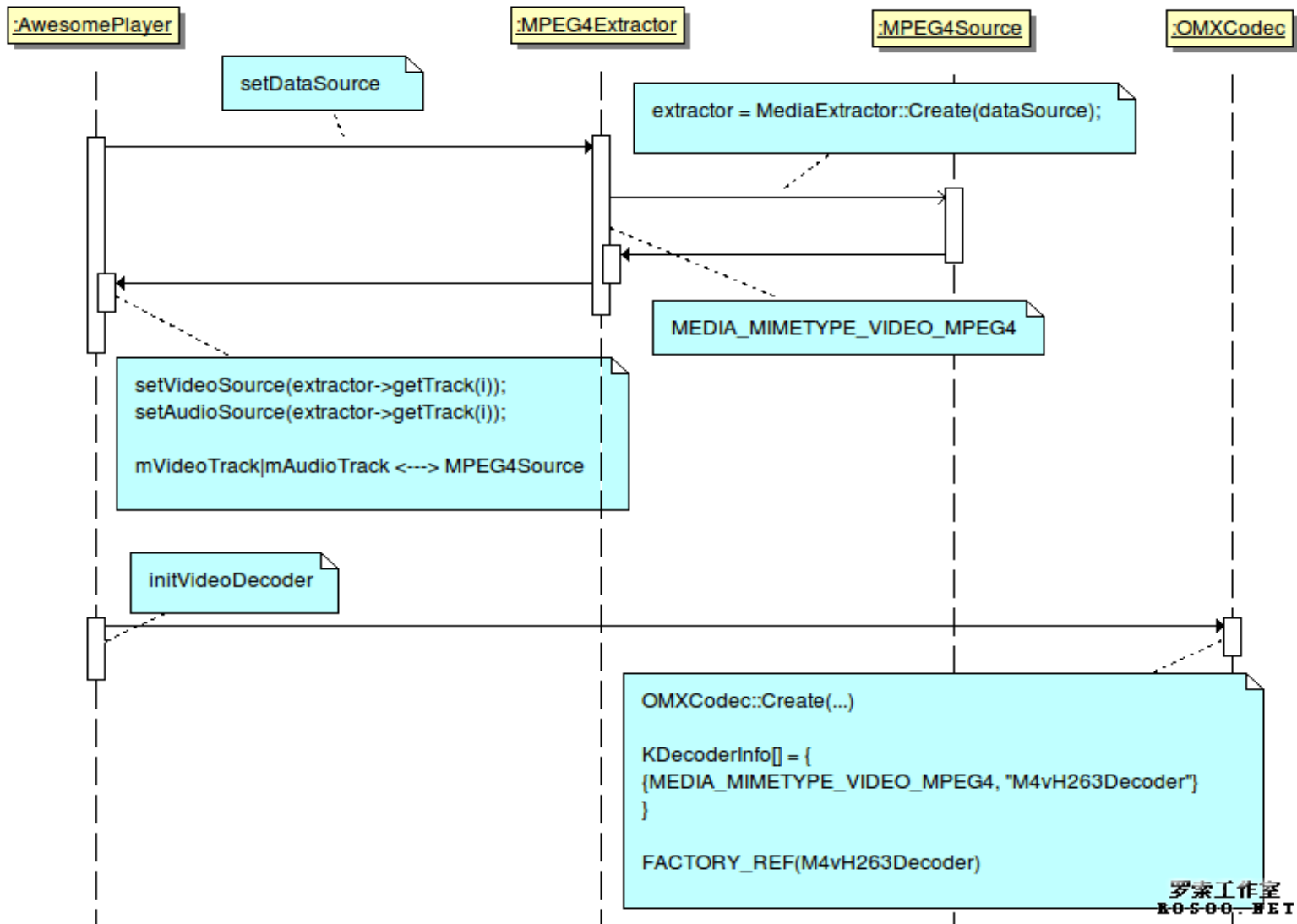


## 2. stagefright 전체 구조/8 - Media Recording/2



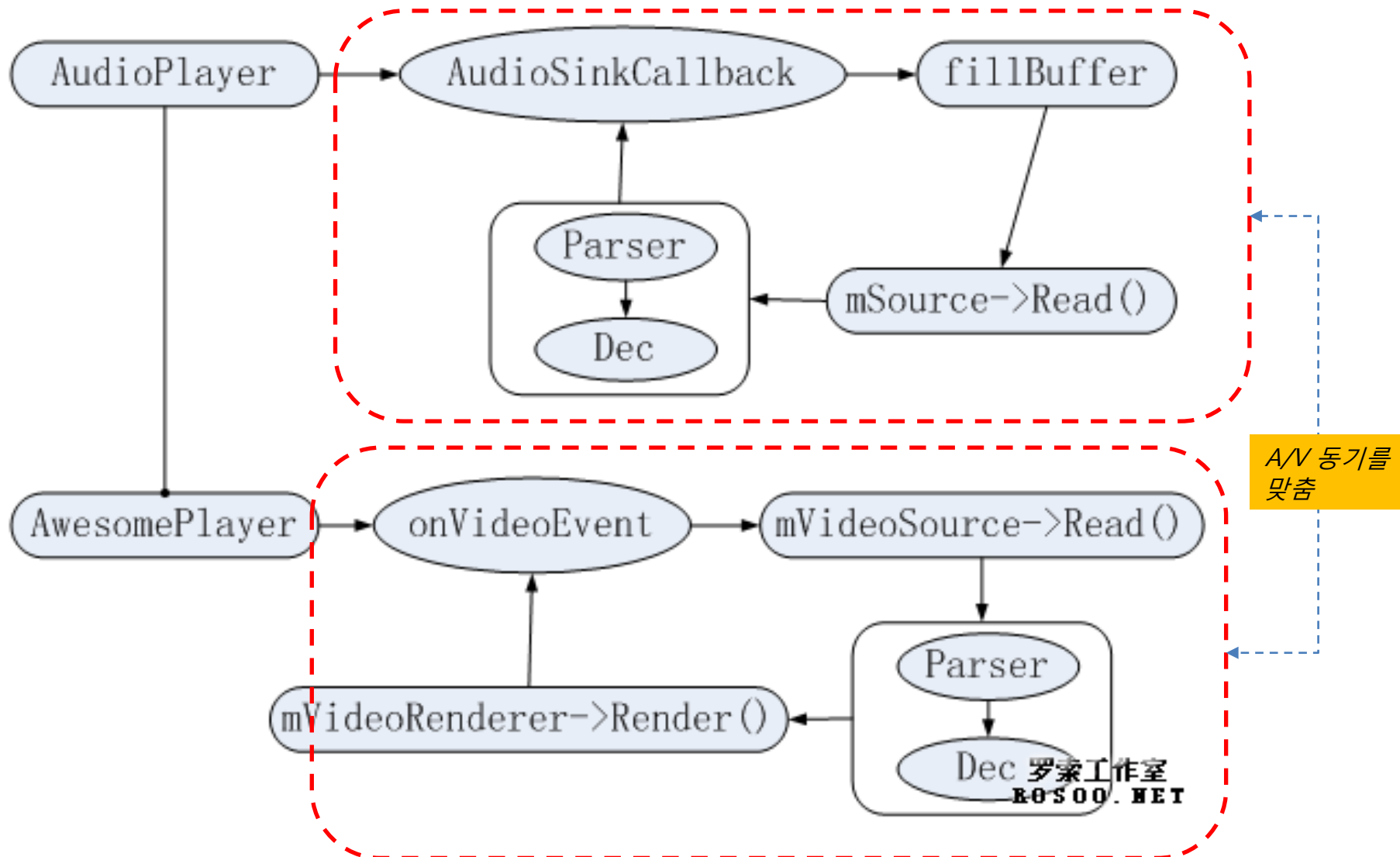
### 3. Audio/Video flow(1) : Video의 경우

(\*) audio flow의 경우도 비슷함 !

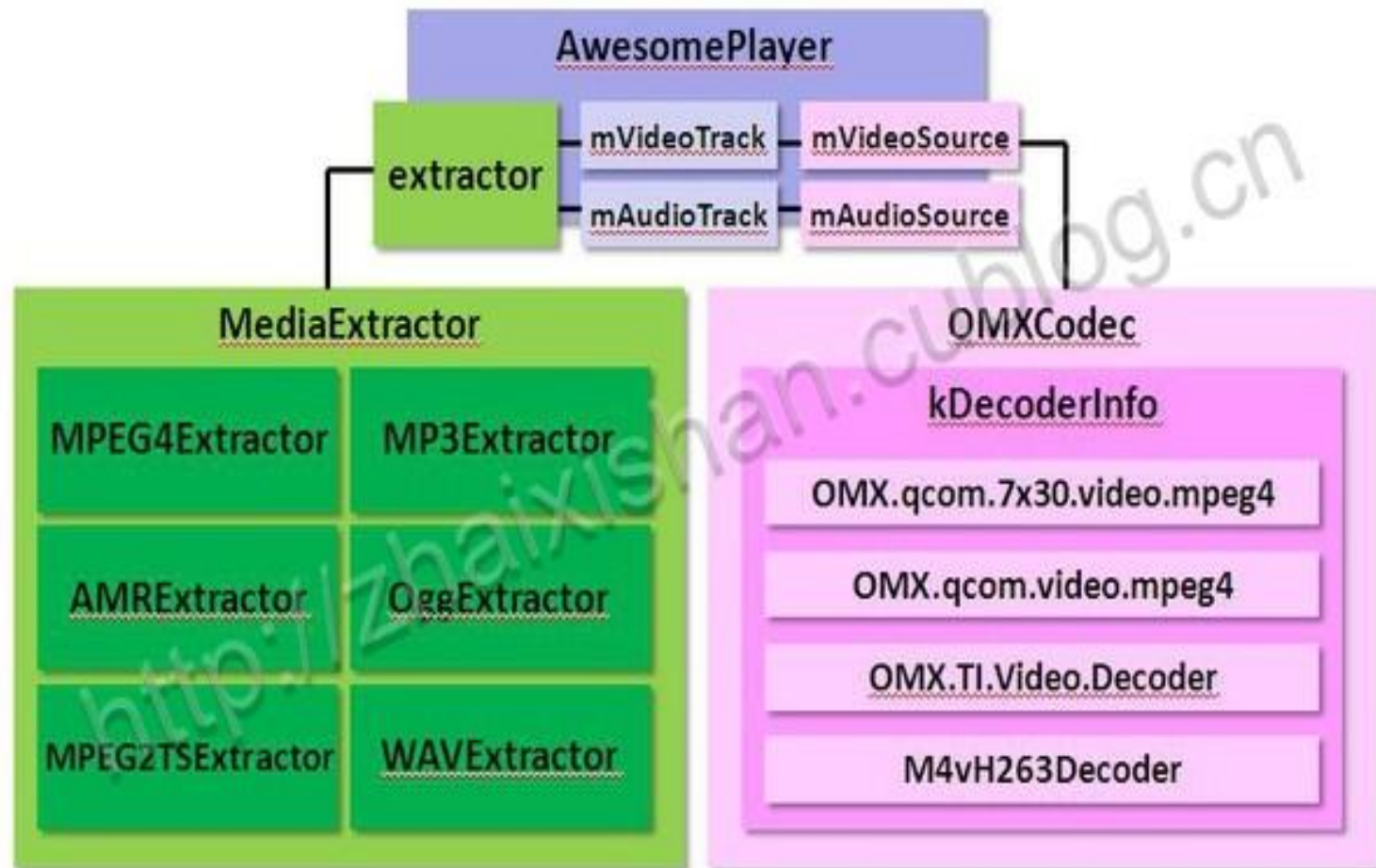


### 3. Audio/Video flow(2)

(\*) 그림이 좀 이상함(특히 화살표^^). 아래 부분에서 read()는 Decoding이 끝난 data를 읽어 가는 것을 의미하며, 각각의 data는 Video renderer와 audio flinger로 전달될 것임.

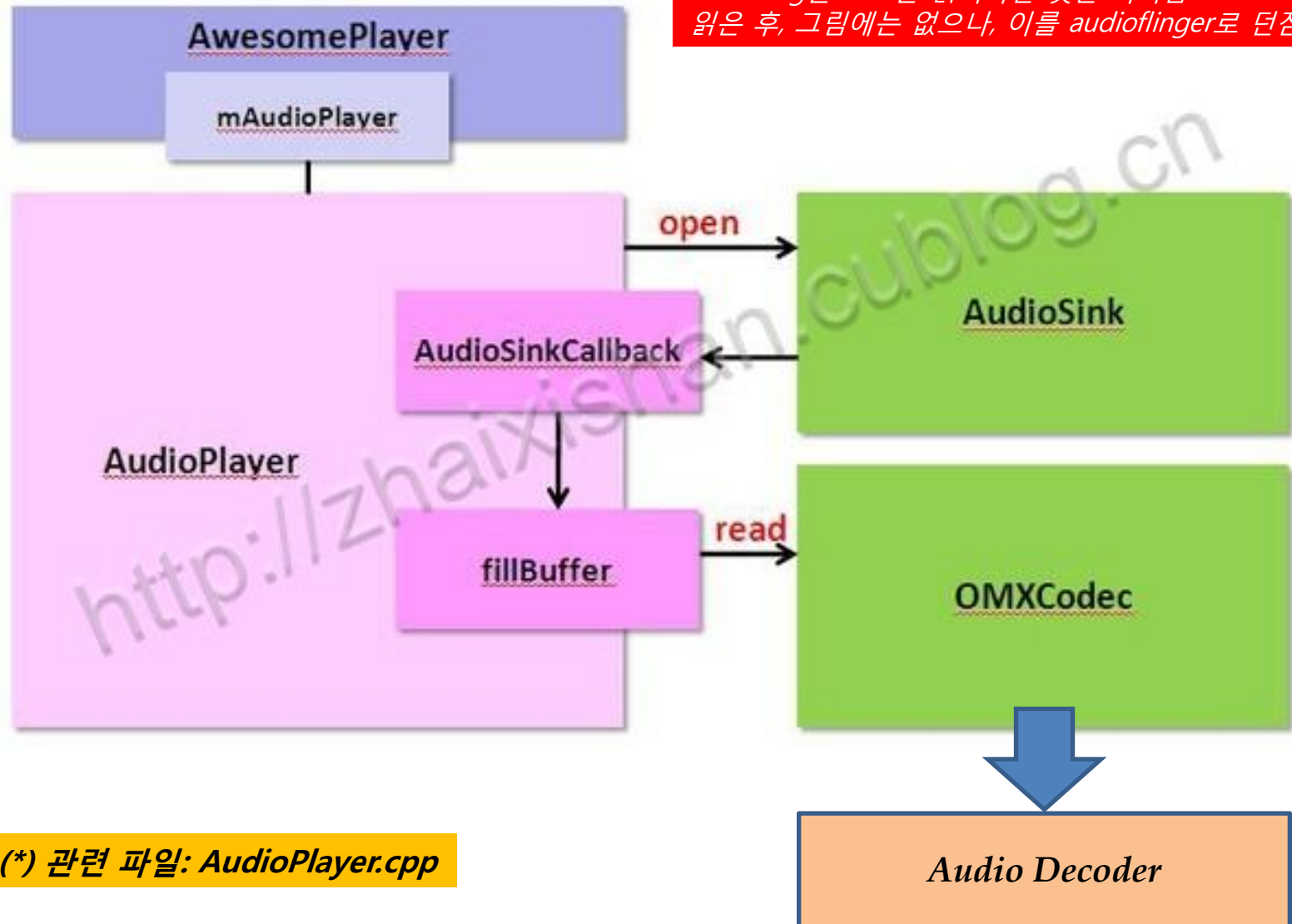


### 3. Audio/Video flow(3)



## 4. Audio Player(fillBuffer)/1

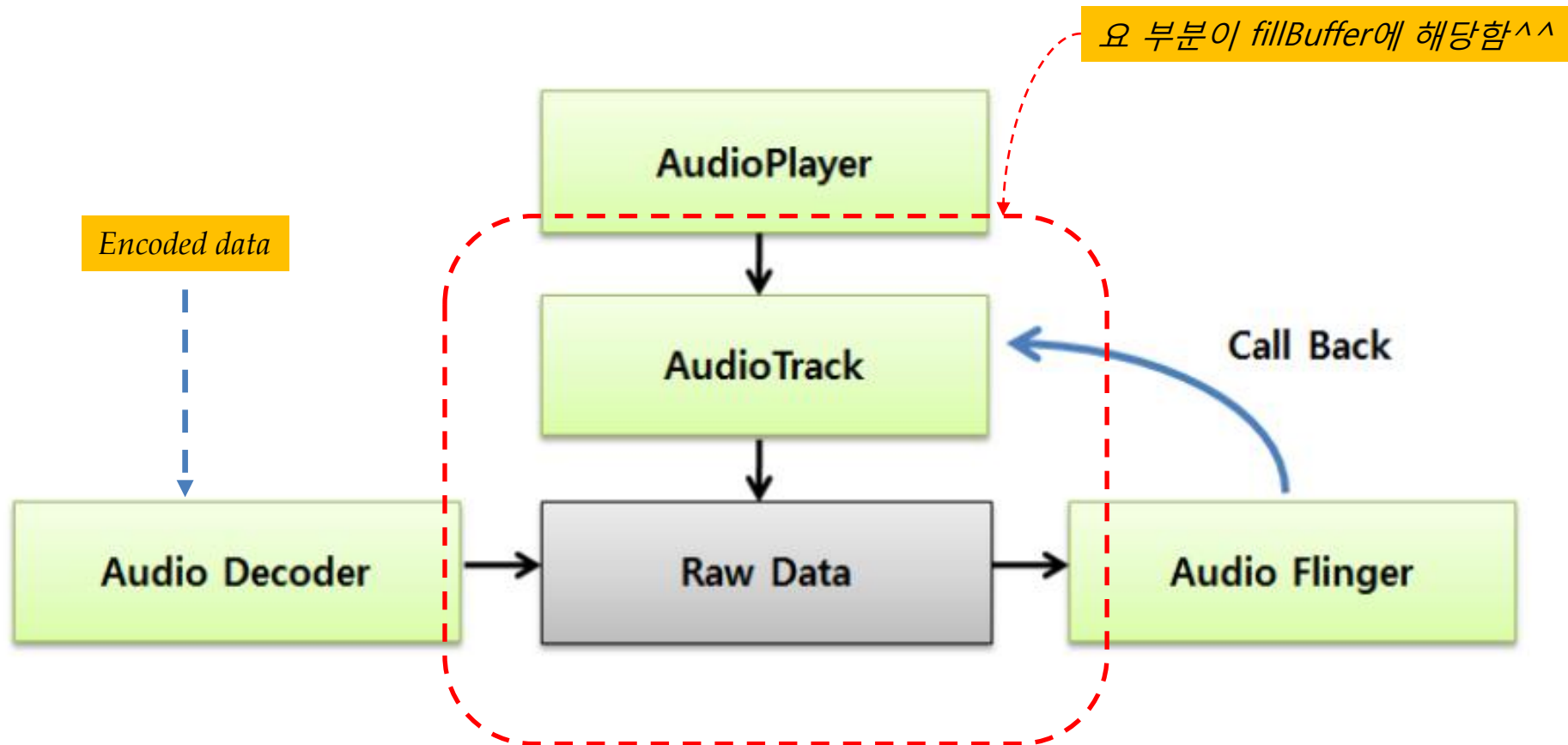
(\*) 그림(화살표)이 좀 이상함 !  
아래 그림에서 fillBuffer는 OMXCodec으로 부터  
Decoding된 data를 읽어가는 것을 의미함 !  
읽은 후, 그림에는 없으나, 이를 audioflinger로 던짐 !



(\*) 관련 파일: *AudioPlayer.cpp*



#### 4. Audio Player(fillBuffer)/2



## 5. Video Player(1)

(\*) video 관련 event가 발생할 때, 호출되는 method

onVideoEvent

(\*) video event를 발생시킴

postVideoEvent\_l()

(\*) decoding된 video data를 읽음

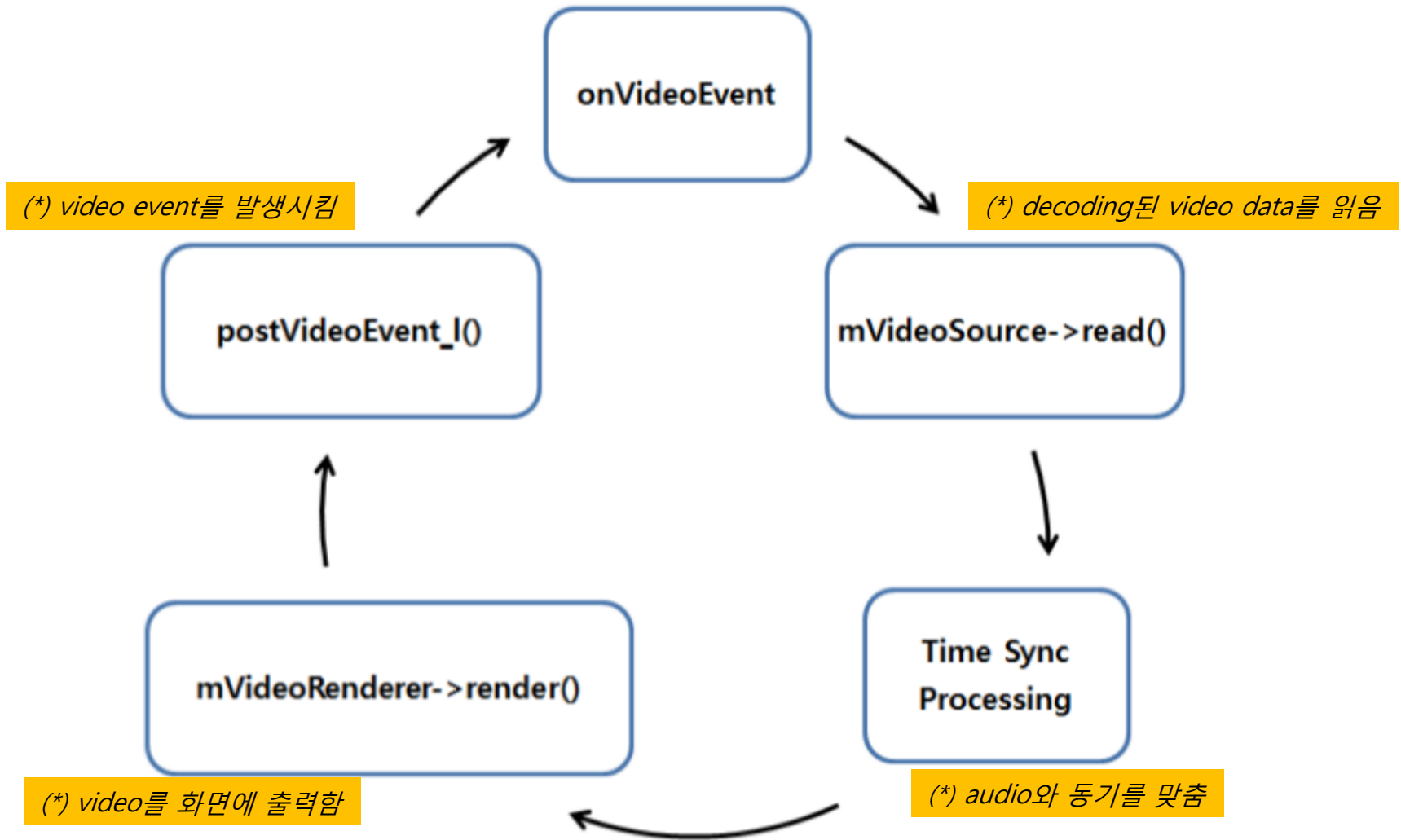
mVideoSource->read()

mVideoRenderer->render()

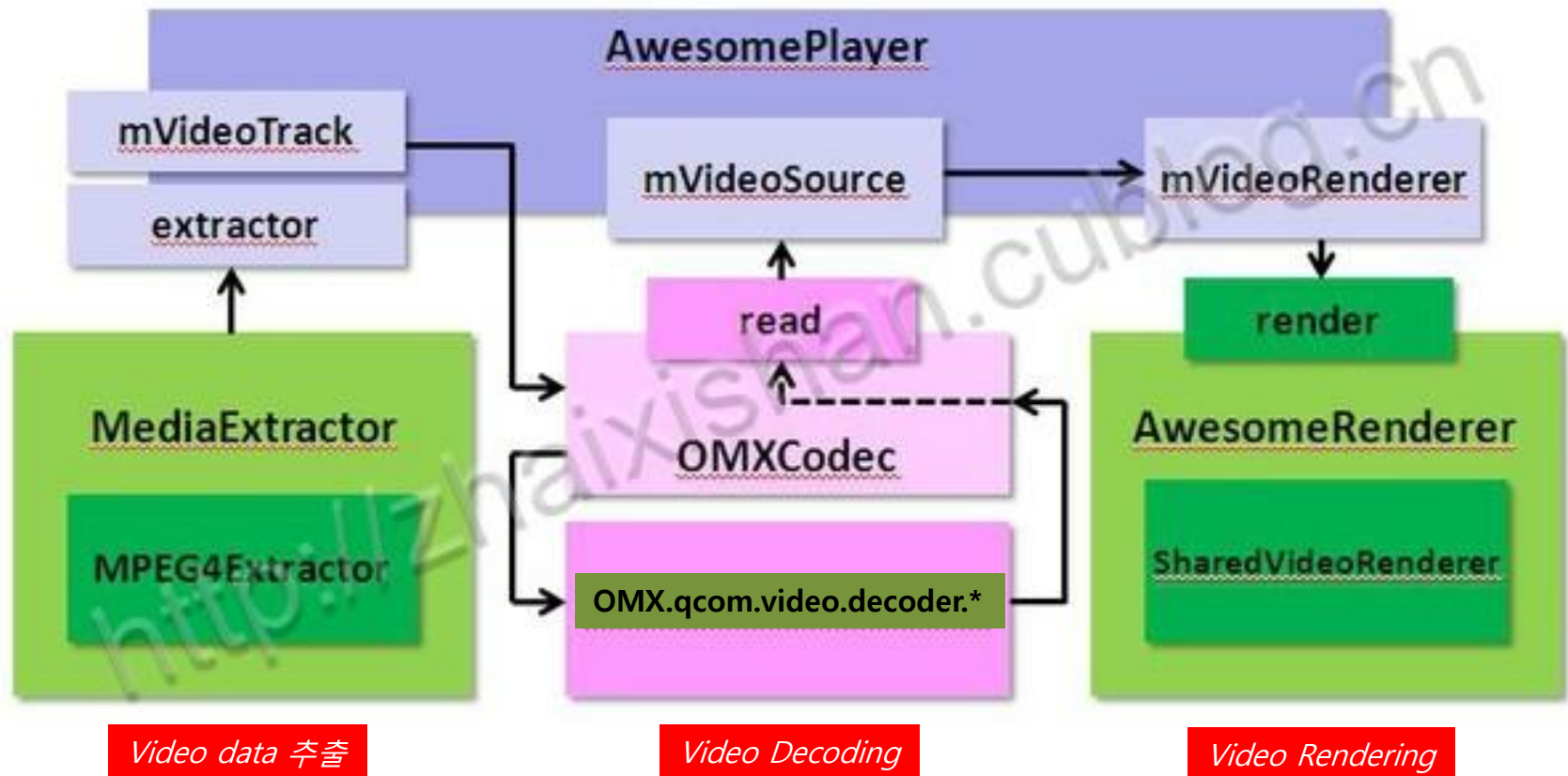
(\*) video를 화면에 출력함

Time Sync  
Processing

(\*) audio와 동기를 맞춤



## 5. Video Player(2)



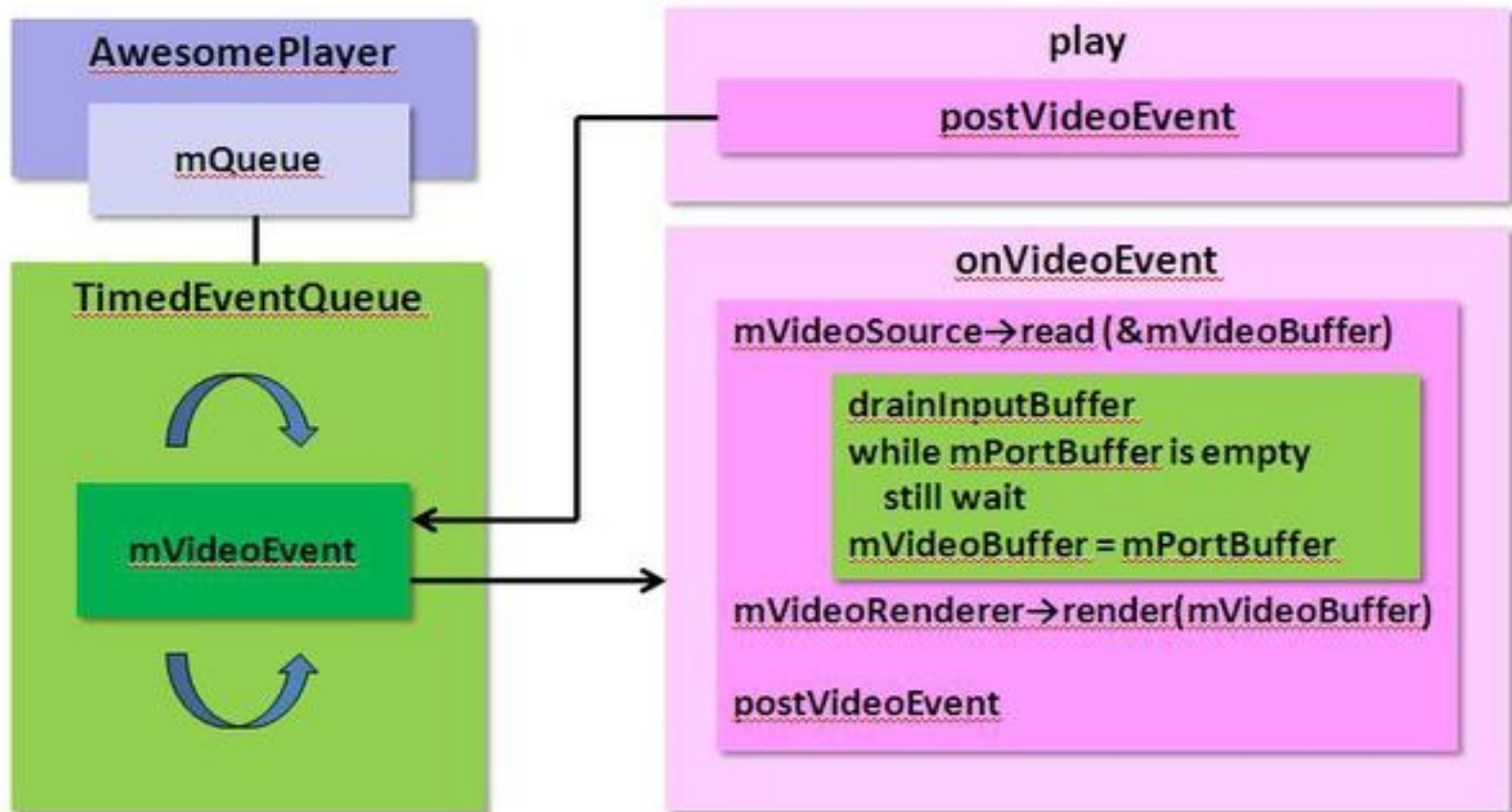
(\*) media file → MediaExtractor → mVideoTrack → Video Decoder(H/W) → OMXCodec → read(decoded video data) → mVideoSource → mVideoRenderer → render 의 순으로 Video data가 흘러가게 됨.

## 5. Video Player(3) : onVideoEvent

(\*) play 시작 시, event 발생

(\*) event가 발생하면, decoding된 결과를 read()하여, render() 한 후, Video event 발생

(\*) event는 mQueue 형태로 관리 !!!



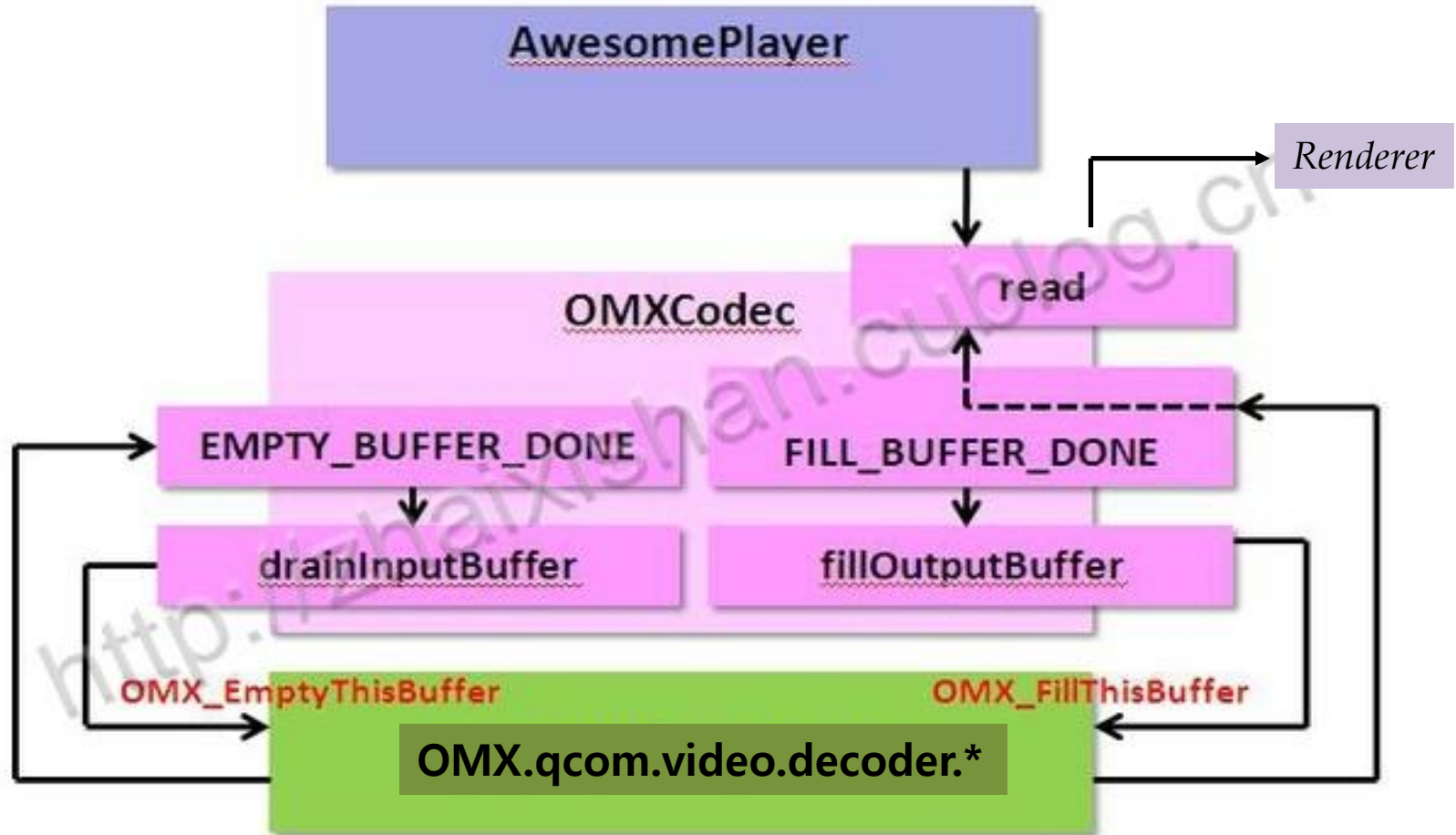
(\*) **mVideoSource->read()** : decoding된 video data를 rendering하기 위해 읽어감을 의미

→ **onVideoEvent** 그림의 녹색 사각형은 **VideoDecoder**와 연관이 있으며, 다음 페이지를 참조하기 바람.

(\*) **mVideoRenderer->render()**: 위의 read()함수를 통해 읽어드린, buffer 내용을 화면에 출력함을 의미

(\*) 관련 파일: **AwesomePlayer.cpp**

## 5. Video Player(4) – OMXCodec FILL\_BUFFER/EMPTY\_BUFFER



(\*) **FILL BUFFER DONE**: Video decoder에서 decoding된 data를 buffer에 채운 후, 발생시키는 event. 이렇게 채워진 data는 read() 과정을 거쳐, 화면에 출력(rendering)하게 됨.

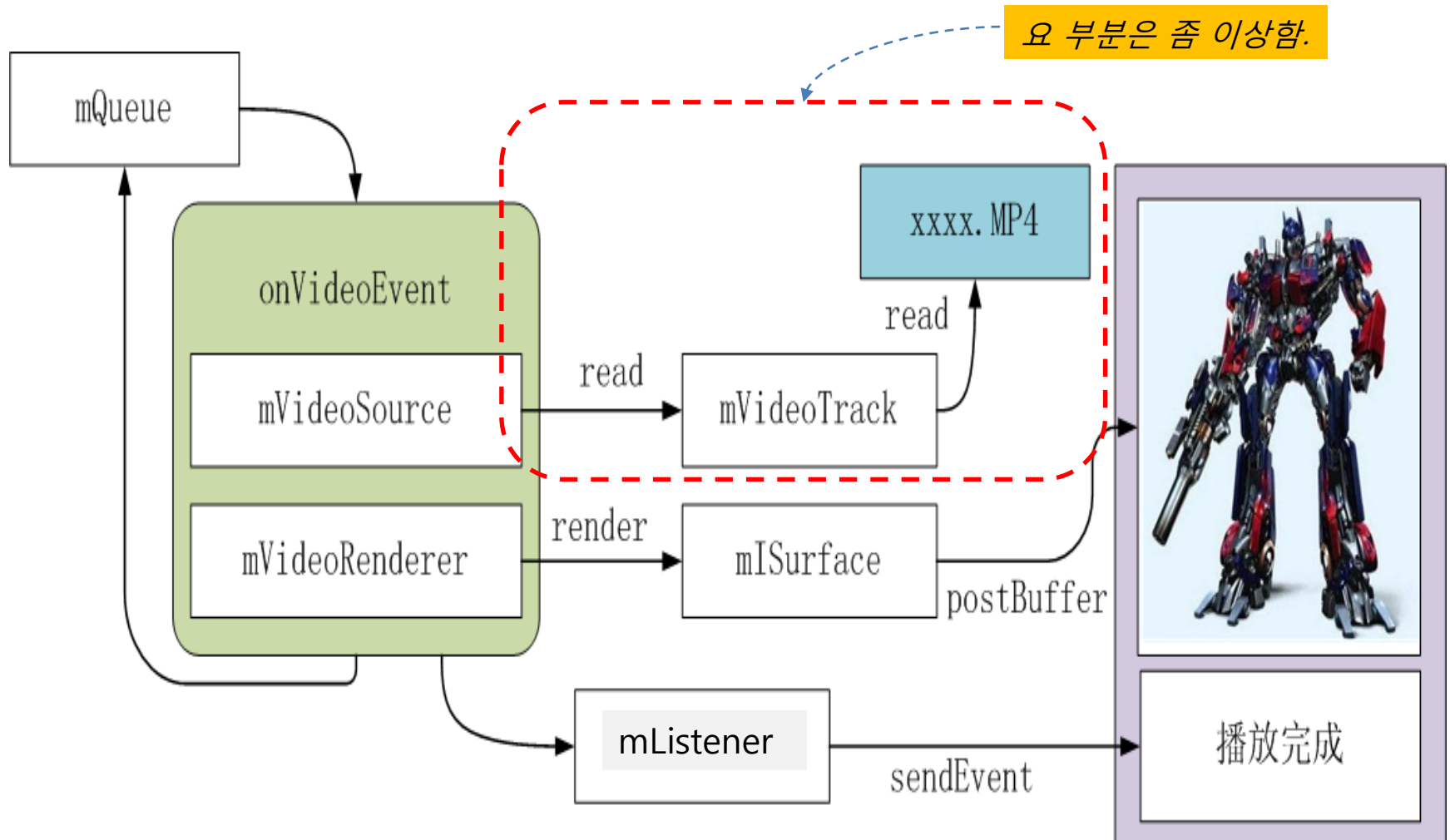
→ **mOMX->fillBuffer()**

(\*) **EMPTY BUFFER DONE**: decoding할 data가 필요할 때, 즉, 이미 이전 data를 모두 decoding 완료하여 새로운 data가 필요할 때, video decoder가 상위 모듈에 요청하는 event로 보임 !

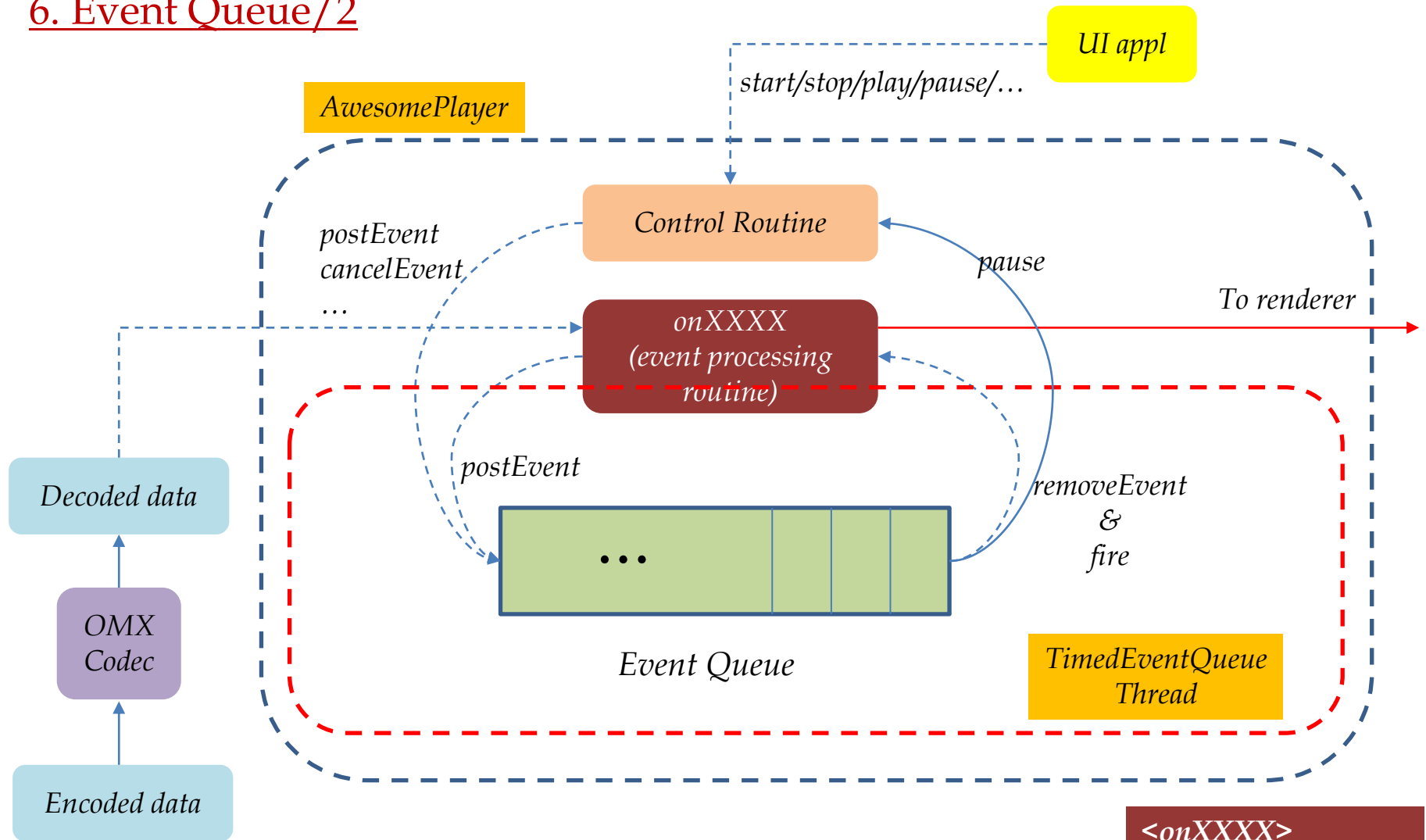
→ **mOMX->emptyBuffer()**

(\*) **관련 파일**: **OMXCodec.cpp**

## 6. Event Queue/1



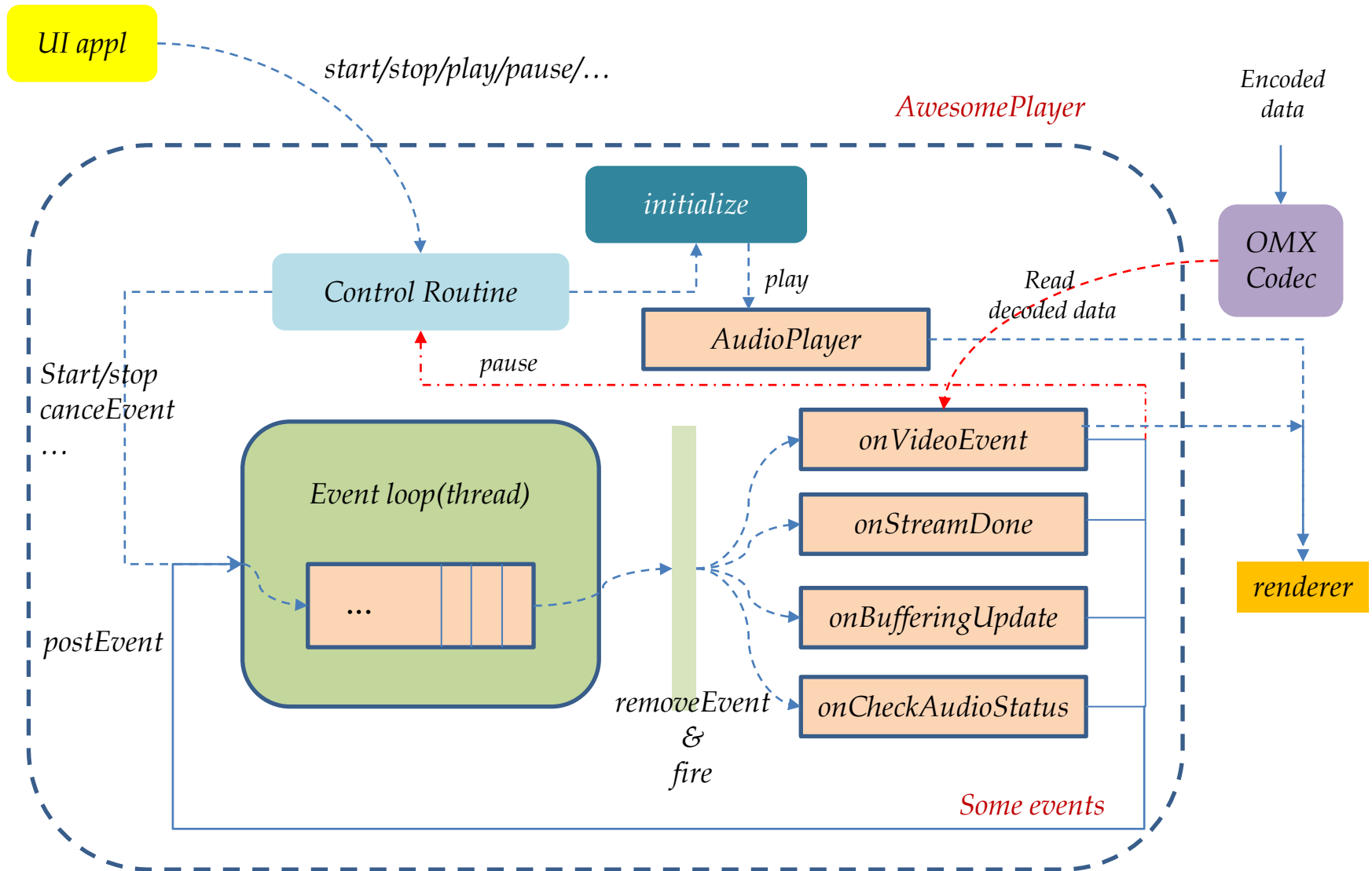
## 6. Event Queue/2



(\*) event driven 방식이므로, 초기화(event queue 생성 및 event handler 등록 및 각종 루틴 초기화) 후, event가 발생할 때마다 이를 처리하고, 처리 과정에서 다시 새로운 event를 발생시키는 형태로 운용된다고 보면 될 듯 ^^

**<onXXXX>**  
onVideoEvent  
onStreamDone  
onBufferingUpdate  
onCheckAudioStatus  
onPrepareAsyncEvent

## 6. Event Queue/3





## 6. Event Queue/4

- 1) TimedEventQueue를 상속받은 AwesomePlayer는 event loop(for loop)을 돌며, event queue로 들어온, event를 하나씩 꺼내어 처리한다.
- 2) Event queue에서 꺼내온 event의 type에 따라, 각각 onVideoEvent, onStreamDone, onBufferingUpdate, onCheckAudioStatus method가 호출되는 데,
- 3) 이 중, onVideoEvent는 OMXcodec으로 부터 decoding된 data를 읽어서, renderer에게 넘겨주는 역할을 하며, 자체적으로 VideoEvent를 다시 발생시킨다.
- 4) onStreamDone은 재생 중이던 stream이 play를 종료하고자 할 때에 호출되며, 따라서 최종적으로 pause\_l() 함수를 호출하게 된다.
- 5) 또한, onBufferingUpdate는 streaming play시, buffer 내용이 갱신될 때마다 호출되며, buffer pause 상태(재생할 data가 없는 상태) 등도 감지하는 역할을 한다.
- 6) 마지막으로 onCheckAudioStatus는 글썽 그대로 audio의 상태를 체크하기 위해 호출된다.
- 7) AwesomePlayer는 event loop을 돌며, event를 처리하는 것 이외에도, binder를 통해 application으로 부터 전달된, 각종 playback command(start, stop, play, pause, prepare ...)도 받아 처리한다.
  - ➔ 앞서 기술한 event를 처리하고 있는 도중에, playback command가 도달할 경우, 현재 처리중이던 event를 처리한 후, playback 명령이 처리되는 것으로 보임.

## 7. A/V Sync(1)

- Audio Video Sync

*Audio time is used as reference. In Stagefright, awesome player uses Audio Player for playing out audio. This player implements an interface for providing time data. Awesome player uses this for rendering Video. Basically Video frames are rendered when their presentation time matches with that of audio sample being played.*

## 7. A/V Sync(2)

```
• size_t AudioPlayer::fillBuffer(data, size)
{
    ...

    mSource->read(&mInputBuffer, ...);

    mInputBuffer->meta_data()->findInt64(kKeyTime,
    &mPositionTimeMediaUs);
    mPositionTimeRealUs = ((mNumFramesPlayed + size_done
    / mFrameSize) * 1000000) / mSampleRate;
    ← 여기서 계산한 시간을 video 쪽에서 이용하여 동기를 맞춤 !!
    ...
}
```

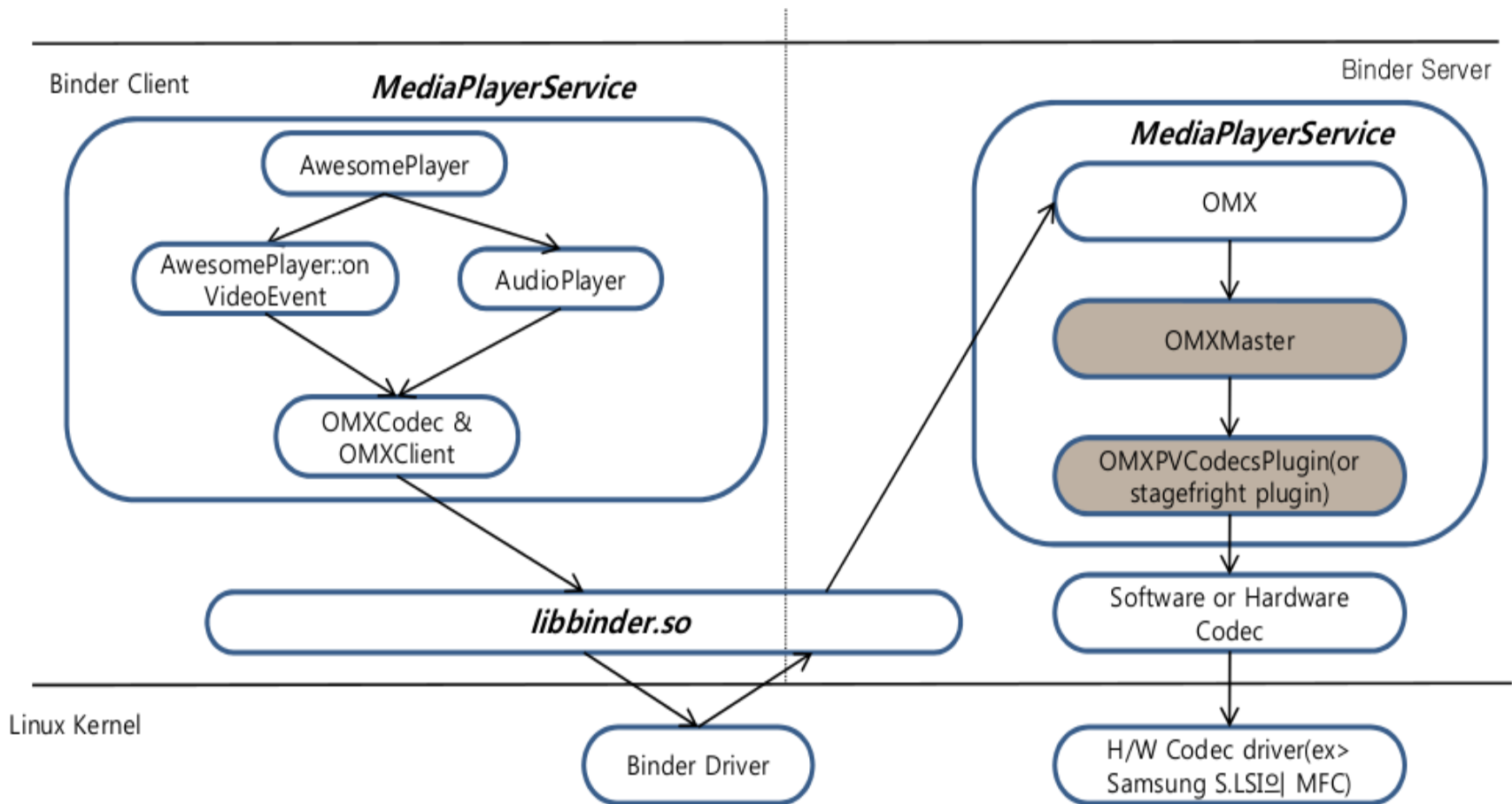
## 7. A/V Sync(3)

- ```
void AwesomePlayer::onVideoEvent()
{
    mVideoSource->read(&mVideoBuffer, &options);
    [Check Timestamp]
    mAudioPlayer-
    >getMediaTimeMapping(&realTimeUs, &mediaTimeUs)
    ← 이 함수 안에서, AudioPlayer에서 계산한 mPositionTimeRealUs 값을 참조하
    여, realTimeUs를 계산하게 됨.
    이후, 아래 코드에서 realTimeUs를 사용하여, frame을 drop하는 등의 처
    리 진행!

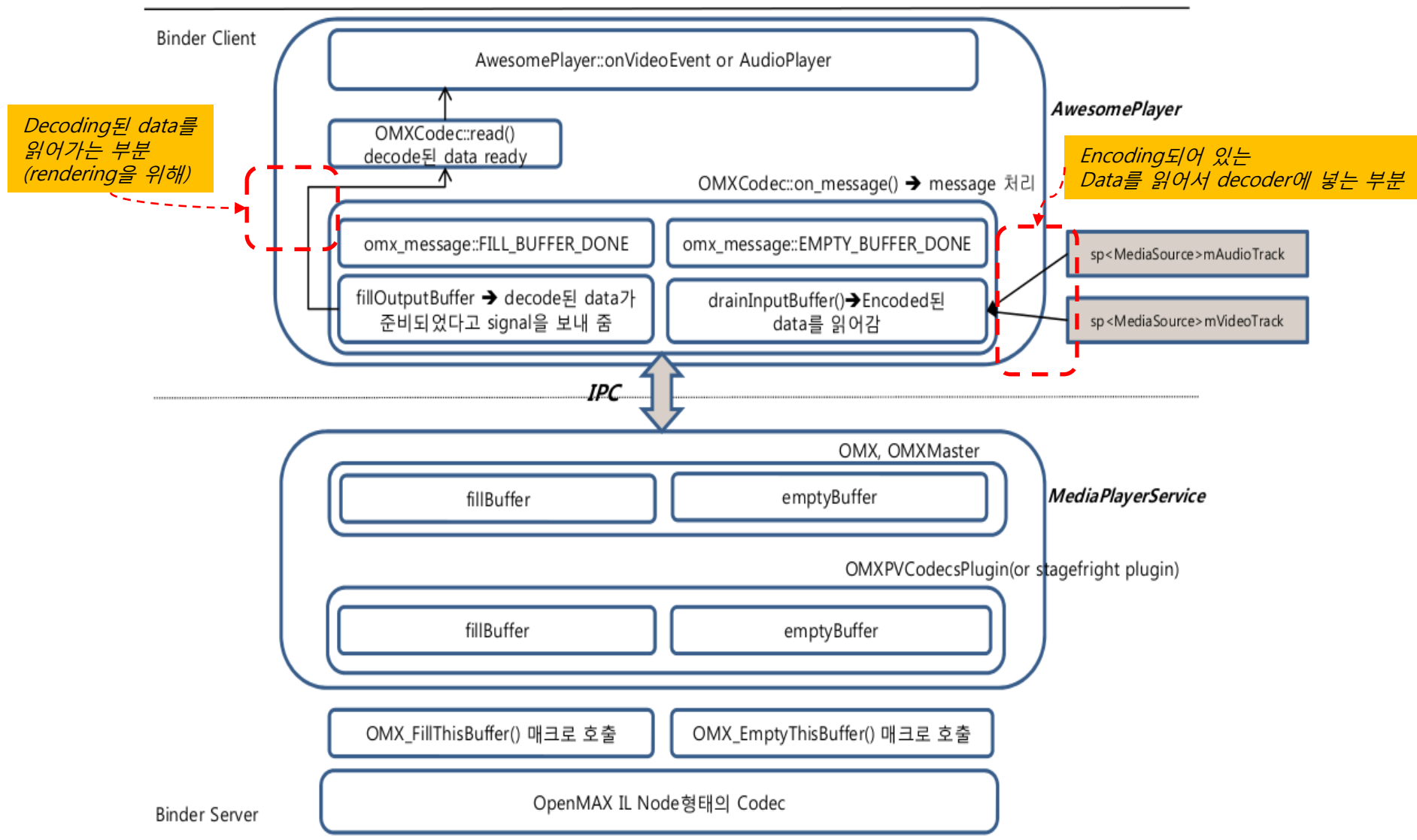
    ...
    mVideoRenderer->render(mVideoBuffer);

    postVideoEvent_1();
}
```

## 8. OMXCodec(1)



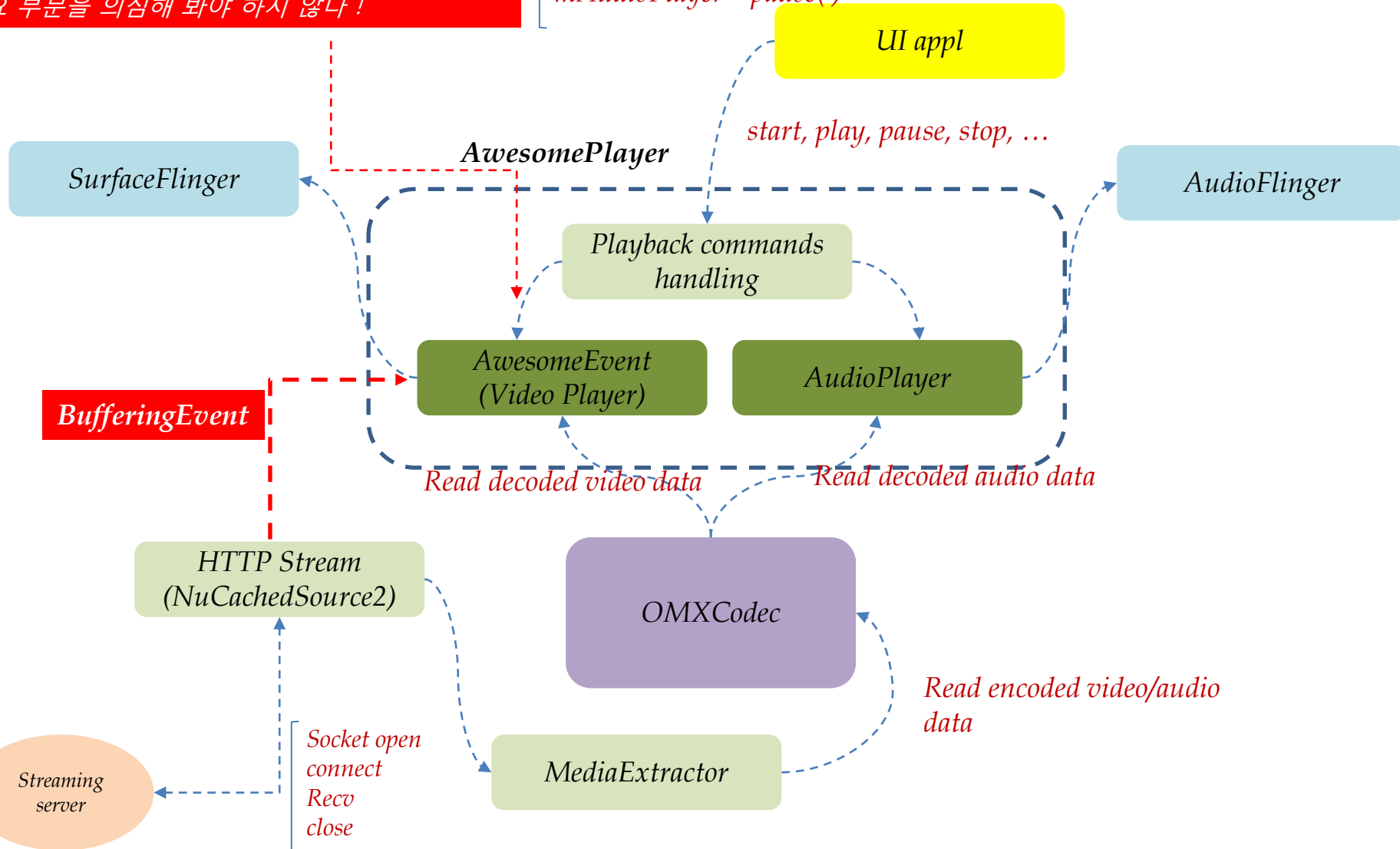
## 8. OMXCodec(2) : FILL\_BUFFER/EMPTY\_BUFFER



## 9. Streaming Player(1) - HTTP/1

(\*) 사용자가 progress bar를 앞으로 움직인 후, pause 버튼을 눌렀는데, ANROI 발생한다면, 요 부분을 의심해 봐야 하지 않나!

`cancelEvents()`  
`mAudioPlayer->pause()`



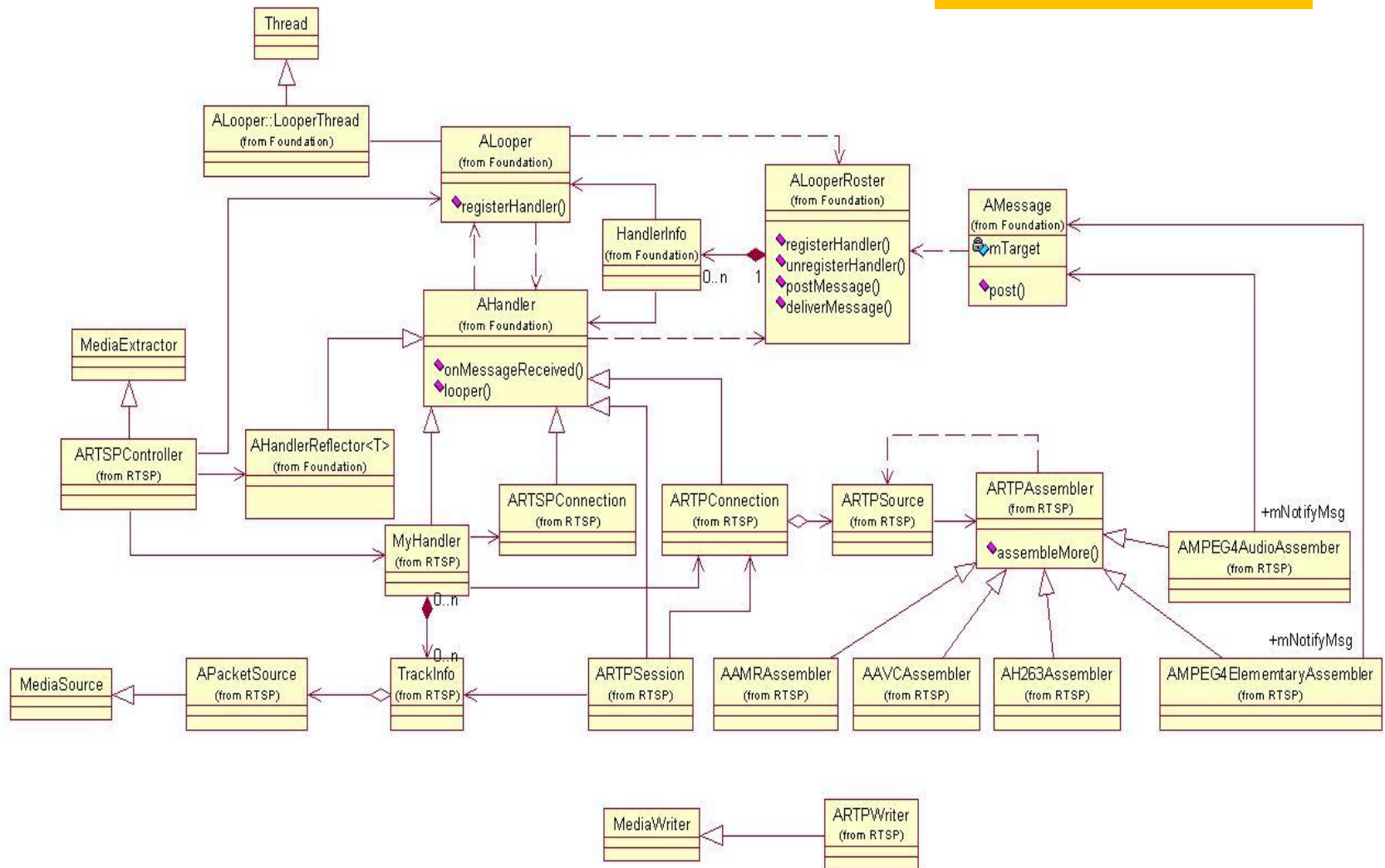
## 9. Streaming Player(1) - HTTP/2

- **<네트워크 상태가 안 좋은 상태에서, 프로그램스 바를 앞으로 전진한 후, Pause 버튼을 선택할 경우, AwesomePlayer에서 하고 있는 일>**
- 1) HTTP connection을 새로 맺고, data를 recv한다.
  - ➔ Recv timeout이 30초로 되어 있음.
- 2) Video Player는 decoding된 data을 읽어 renderer에게 전달하고 있음.
  - ➔ 네트워크 상태가 안 좋을 경우, underrun에 가까운 상황이 발생하게 되므로, renderer에게 전달하는 것도 매우 느려질 것임. OMXCodec으로 부터 event 발생이 느려질 것임.
- 3) Audio Player도 decoding된 data를 읽어 audio flinger에게 전달하고 있음.
  - ➔ 네트워크 상태가 안 좋을 경우, underrun에 가까운 상황이 발생하게 되므로, audioflinger에게 전달하는 것도 매우 느려질 것임. OMXCodec으로 부터 event 발생이 느려질 것임.
- 4) OMXCodec은 Media Extractor에서 추출한 video/audio data를 계속 읽어 들여, decoder에 전달하고 있음.
  - ➔ 네트워크 상태가 안 좋다면, 역시 이 과정 역시 지지부진할 터 ..
- **<동시에 진행되어야 할 부분>**
- *HTTP stream reading*
- *OMXCodec(decoding) - 애플은 binder로 구분되어 있음(독립적으로 진행)*
- *Event Queue processing - 애플은 pthread로 되어 있음.*
- *VideoPlayer*
- *AudioPlayer - 애플은 video쪽에서 시작 시킴*
- *Playback 명령 처리 - 애플은 binder로 받으니, 독립적으로 진행될 듯 함.*
- **<BufferingEvent>**
- *테스트해 보니, Pause 명령을 받고, pause되는 것이 아니라, buffer pause되면서, pause가 되고 있음.*
- *(\*) 테스트해 보니, ANR이 발생하는 이유가 socket connect 후, recv 등에서 blocking이 되면서 발생하는 것 같다.*



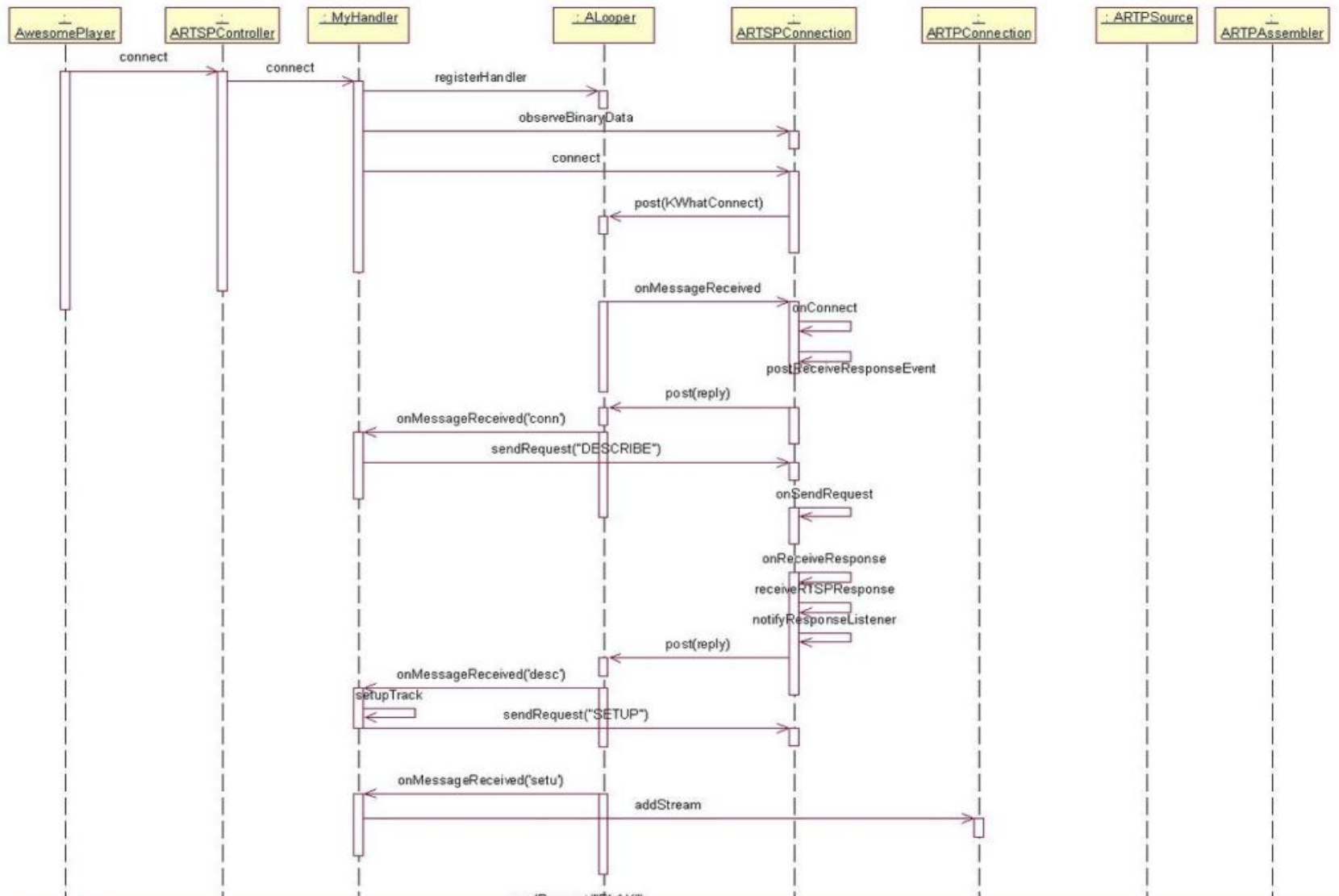
## 9. Streaming Player(2) - RTSP/1

(\*) 좀 더 분석해야 함.



## 9. Streaming Player(2) - RTSP/2

(\*) 아래 부분이 생략된 그림임.




## TODO

- Extractor
- Codecs
- Mpeg2ts

*Thanks a lot !*



## References

- 1) *01.Android-gingerbread-multimedia-framework-structure.pdf*[by 고현철]
- 2) *AndroidMMF-Details\_v04.pdf*[by 김태용 - windriver]
- 3) *Inside\_of\_Stagefright.pdf*[by 김정호 - windriver]
- 4) *Internet[china sites for some images]*
- 5) \_Dev\_Guide\_MMF2.ppt[by 이충한]
- 6) *The\_OpenMAX\_Integration\_Layer\_standard.pdf*[by Giulio Urlini - Advanced System Technology)