Building a Media Pipeline

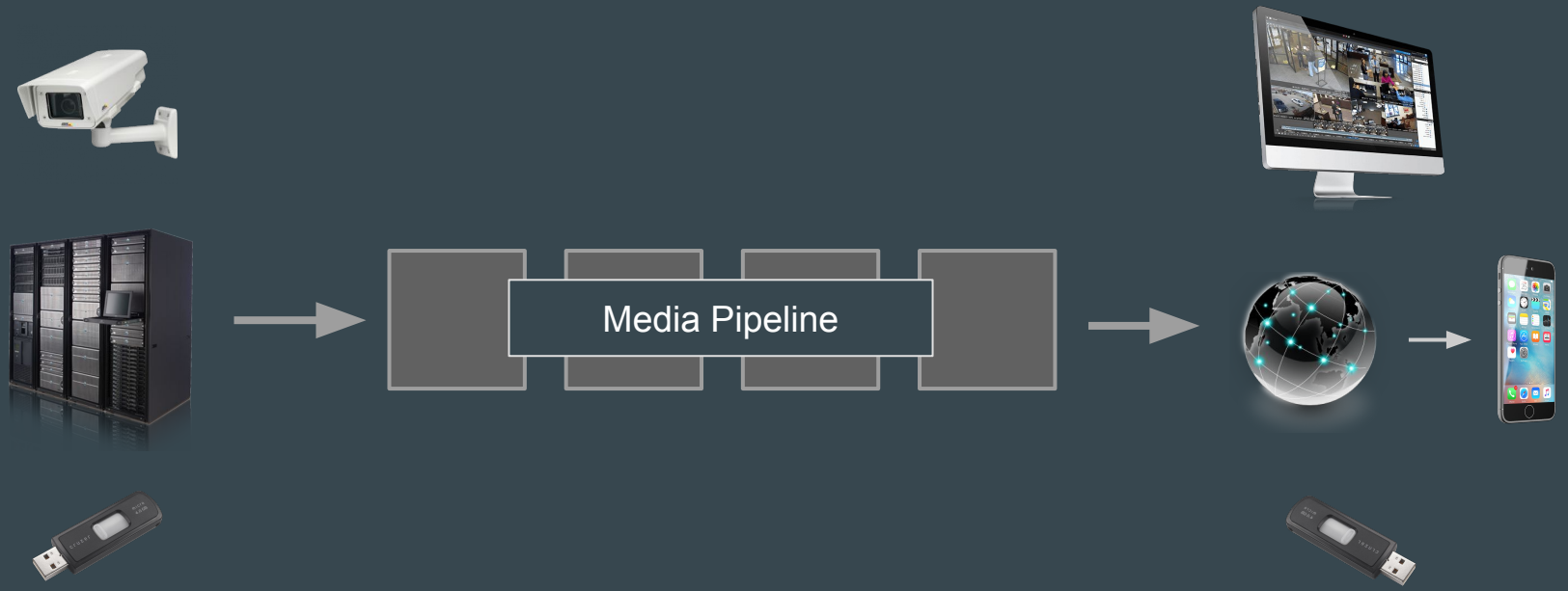
Tom Distler

2019

https://github.com/tjdistler/docs

Context

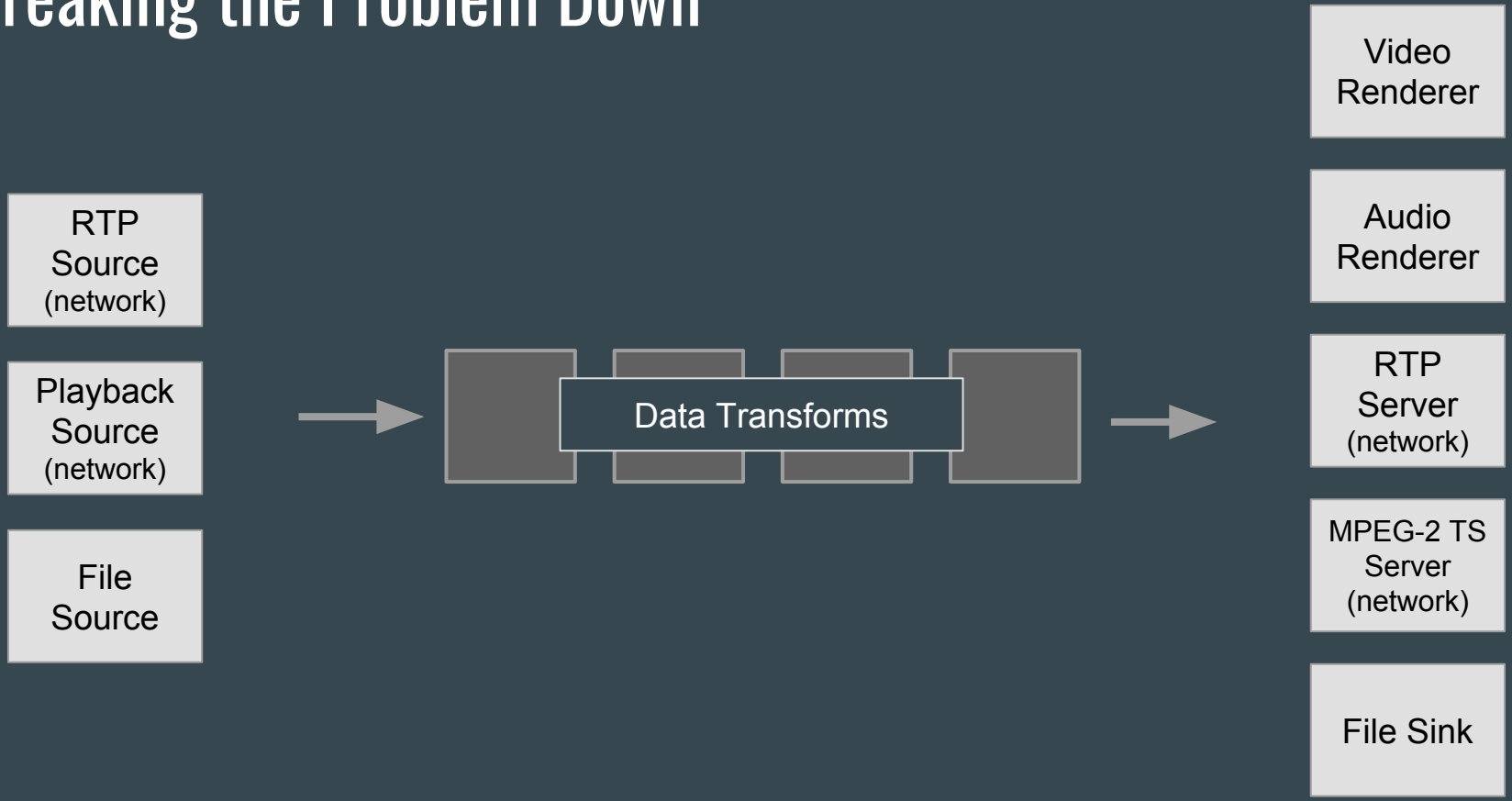
The Problem



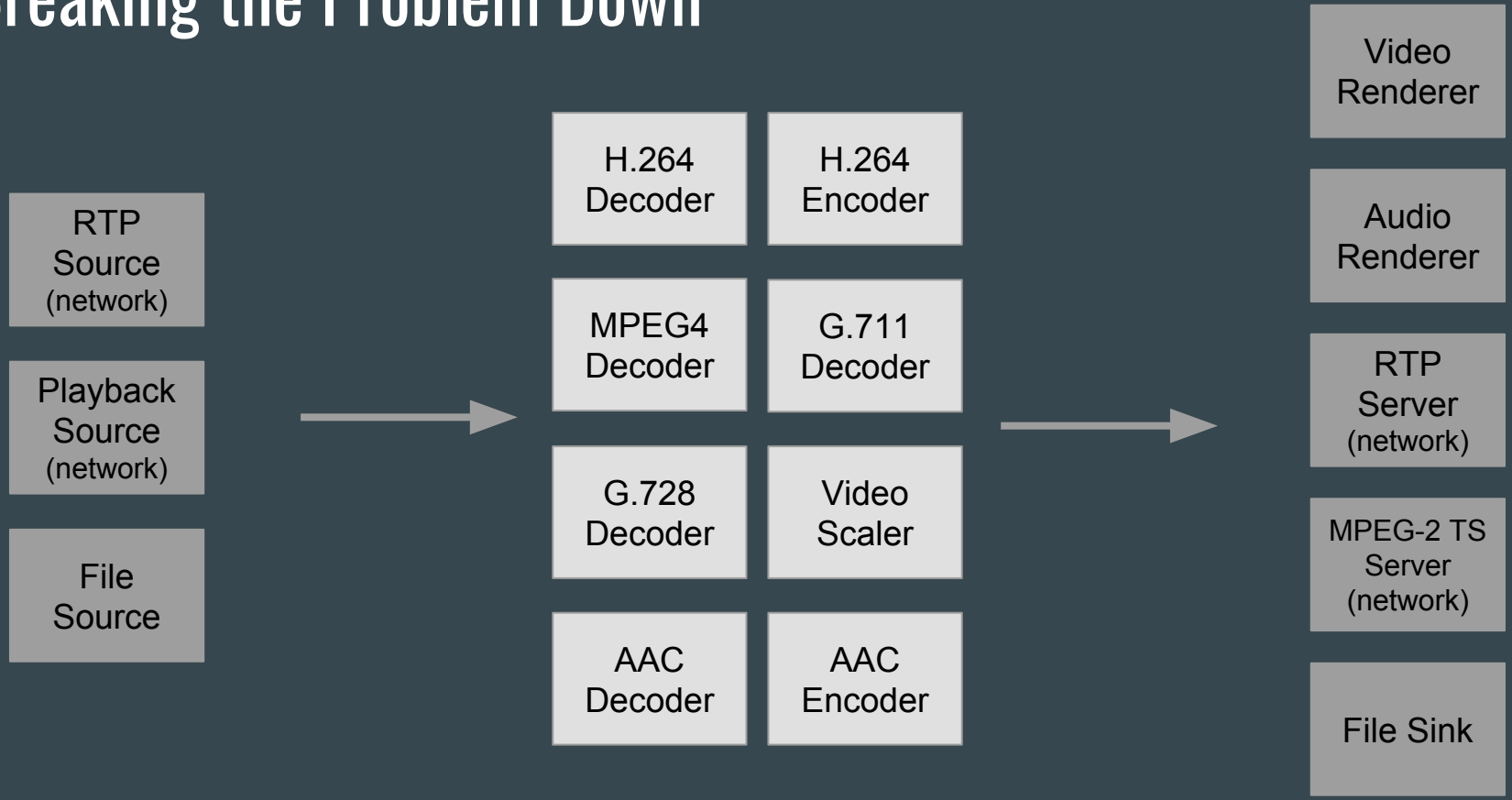
Multiple Sources

Multiple Targets

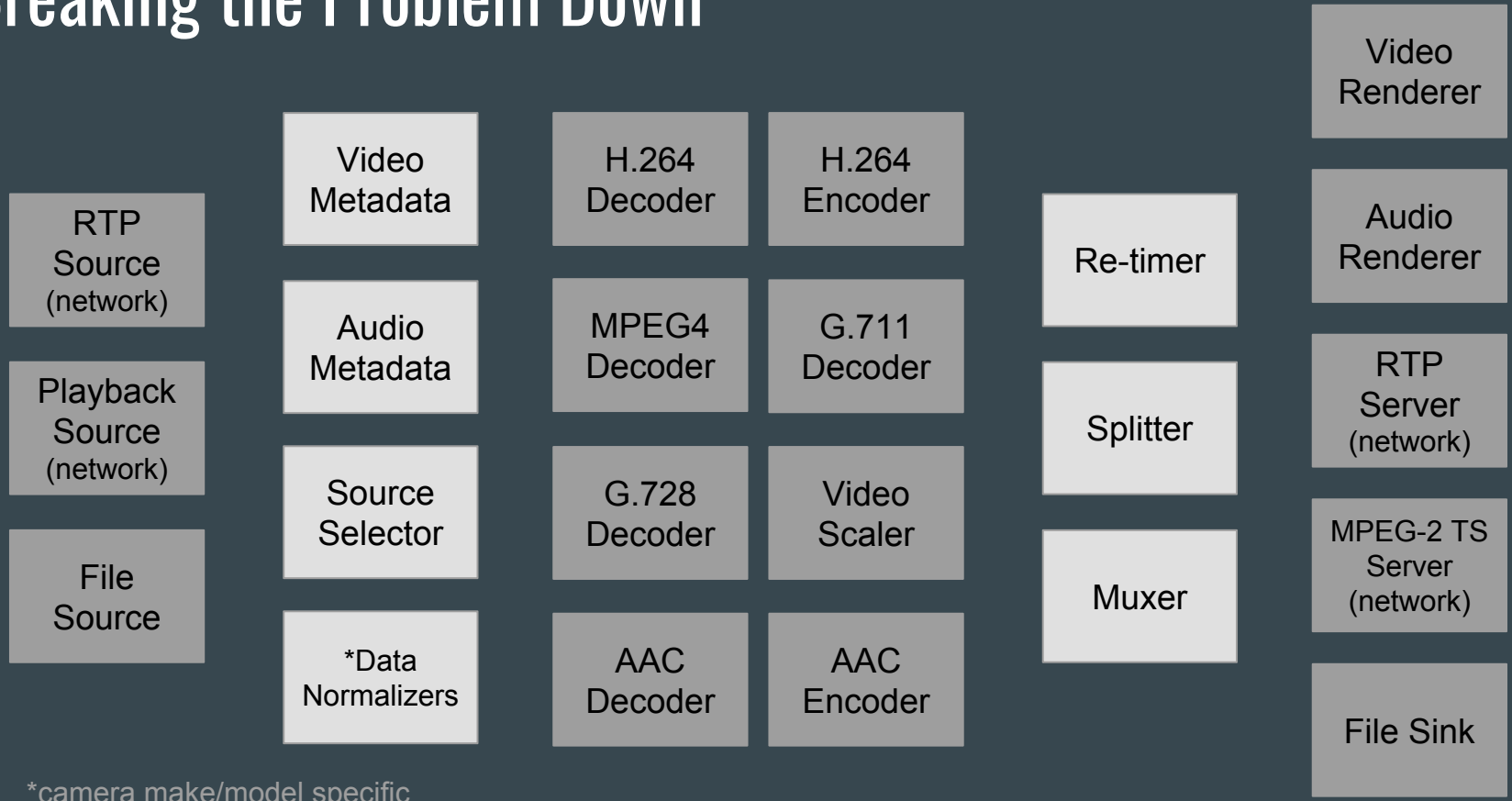
Breaking the Problem Down



Breaking the Problem Down

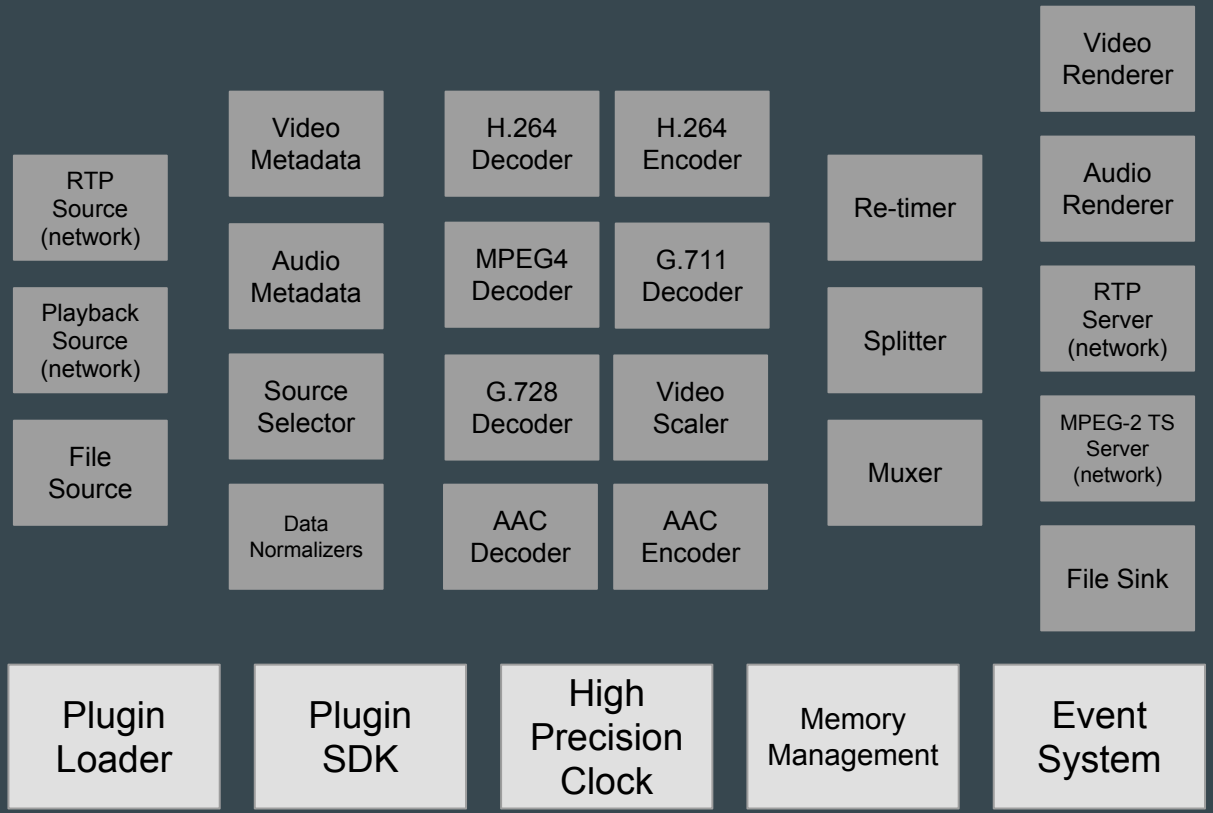


Breaking the Problem Down

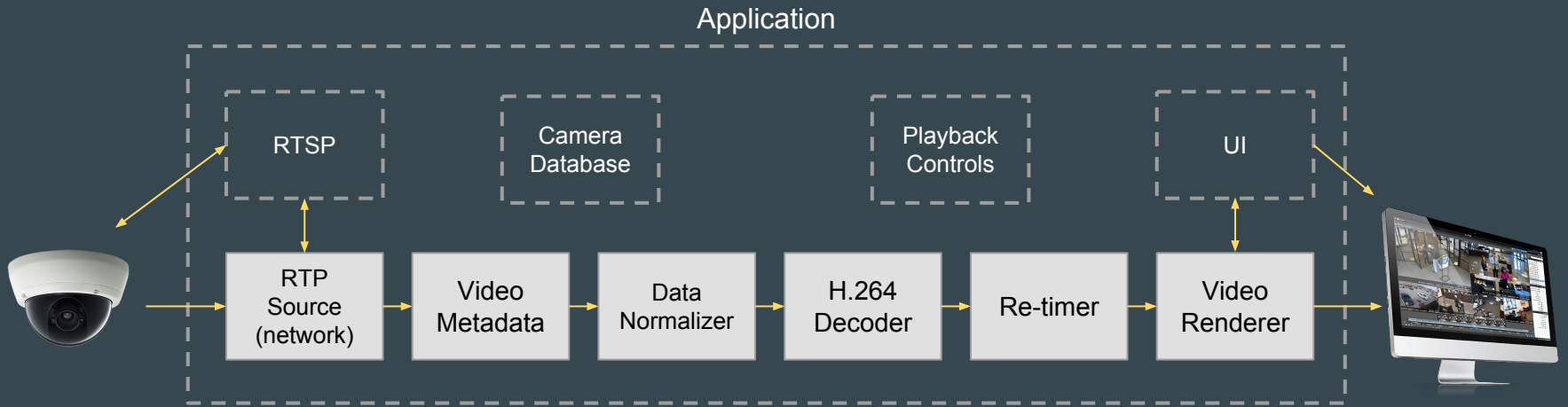


*camera make/model specific

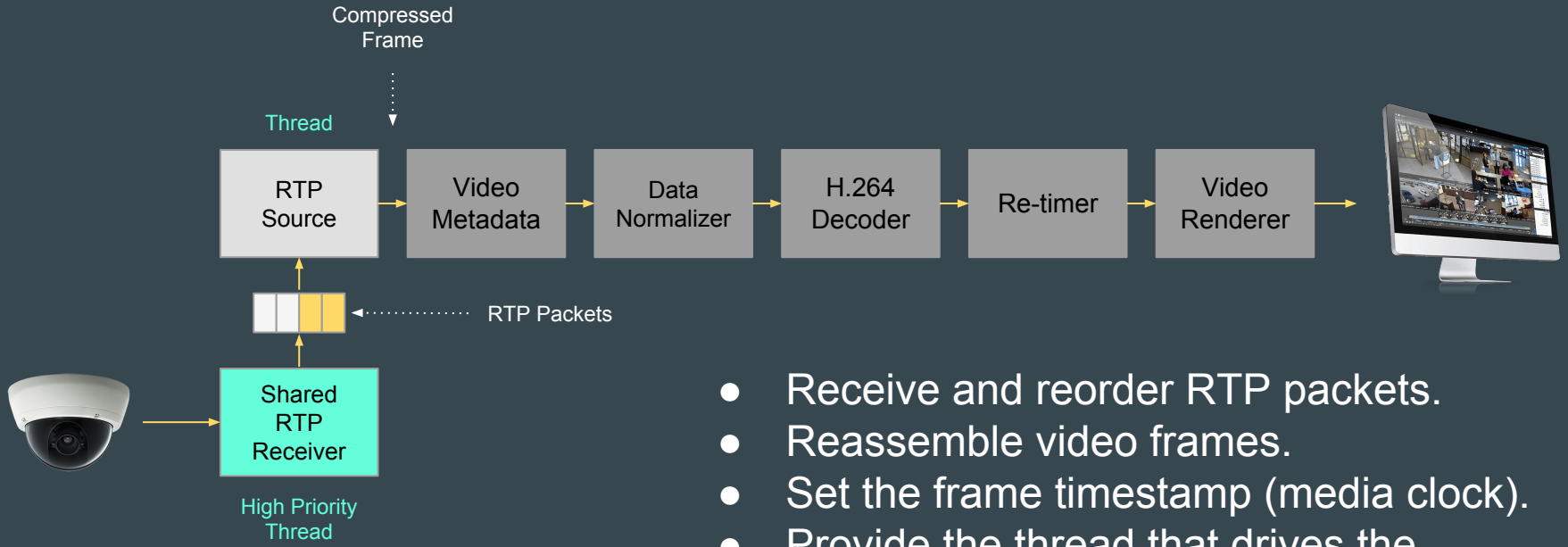
Breaking the Problem Down



Live Video Pipeline Walk-thru

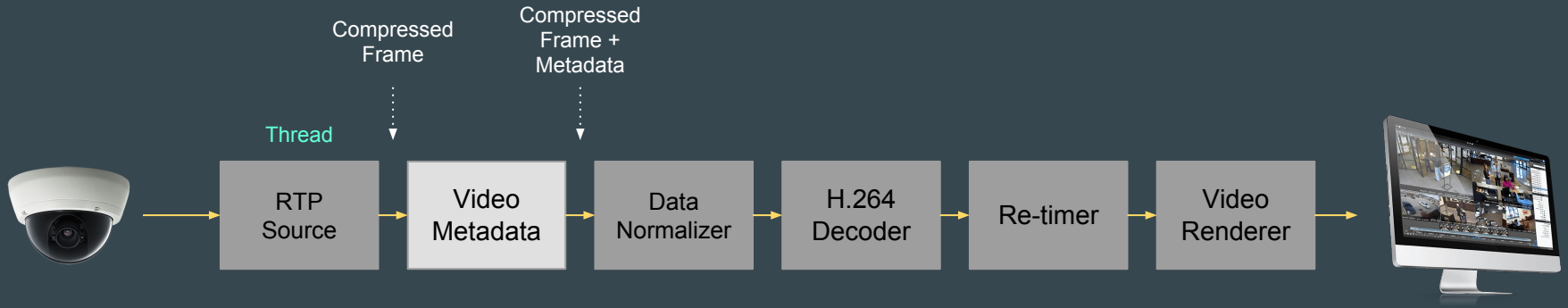


Live Video Pipeline Walk-thru



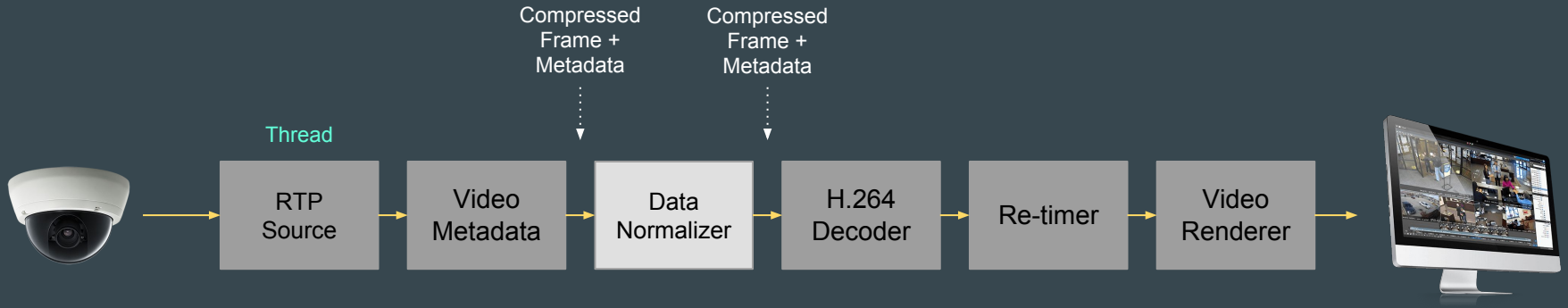
- Receive and reorder RTP packets.
- Reassemble video frames.
- Set the frame timestamp (media clock).
- Provide the thread that drives the front-end of the pipeline.

Live Video Pipeline Walk-thru



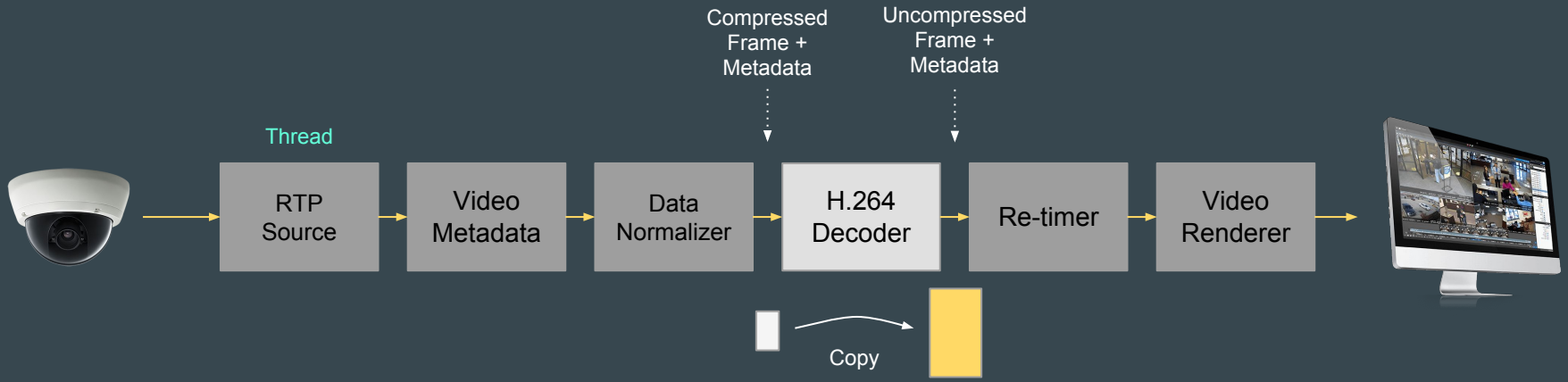
- Determine video codec.
- Read media-info headers for width/height, etc.
- Attach metadata and UTC timestamp to buffer.

Live Video Pipeline Walk-thru



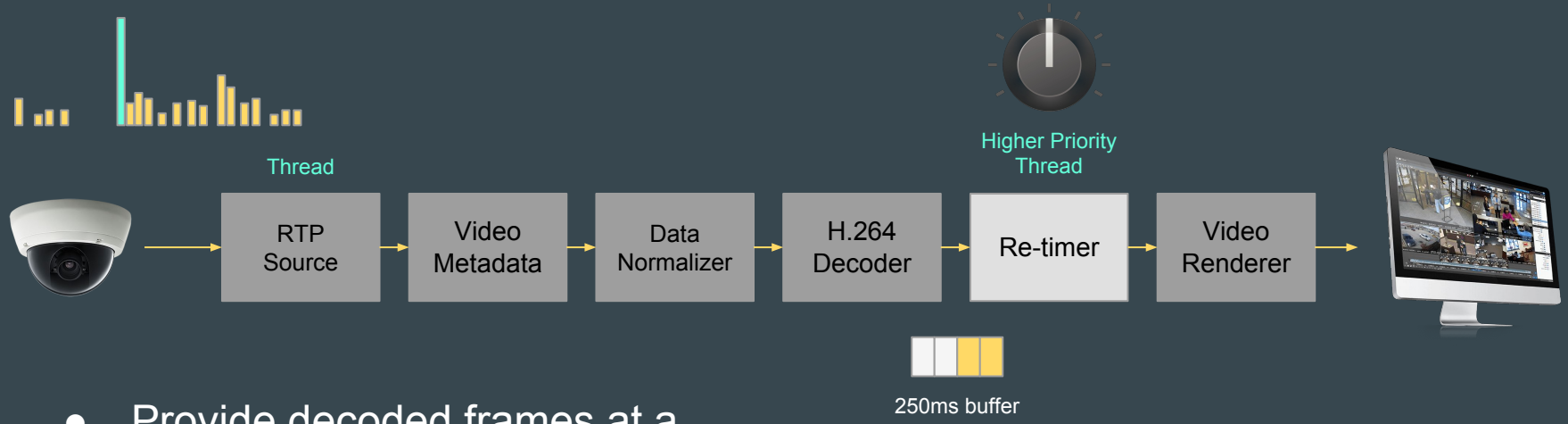
- Correct camera make/model specific problems:
  - Incorrect RTP clock frequency.
  - Missing/incorrect frame dimension information.

Live Video Pipeline Walk-thru



- Decodes into fixed size buffers.
- Decode times:
  - 18-28 ms for I-frames
  - 7-14 ms for P-frames

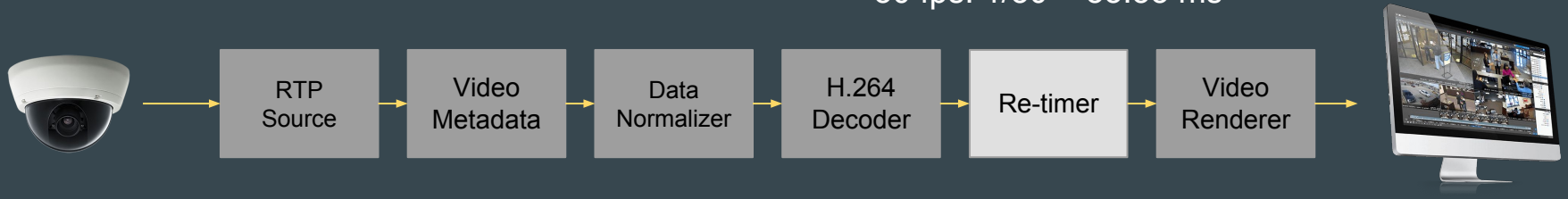
Live Video Pipeline Walk-thru

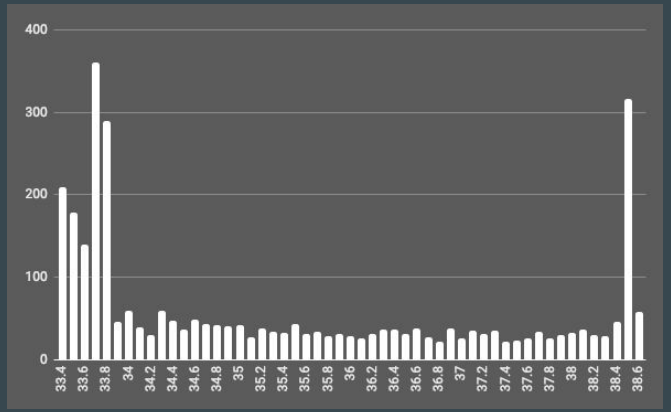


- Provide decoded frames at a consistent rate to the renderer.
- Correct for input stream jitter.
- Handle low-latency vs buffered mode.
- Buffer frames during playback.

Live Video Pipeline Walk-thru

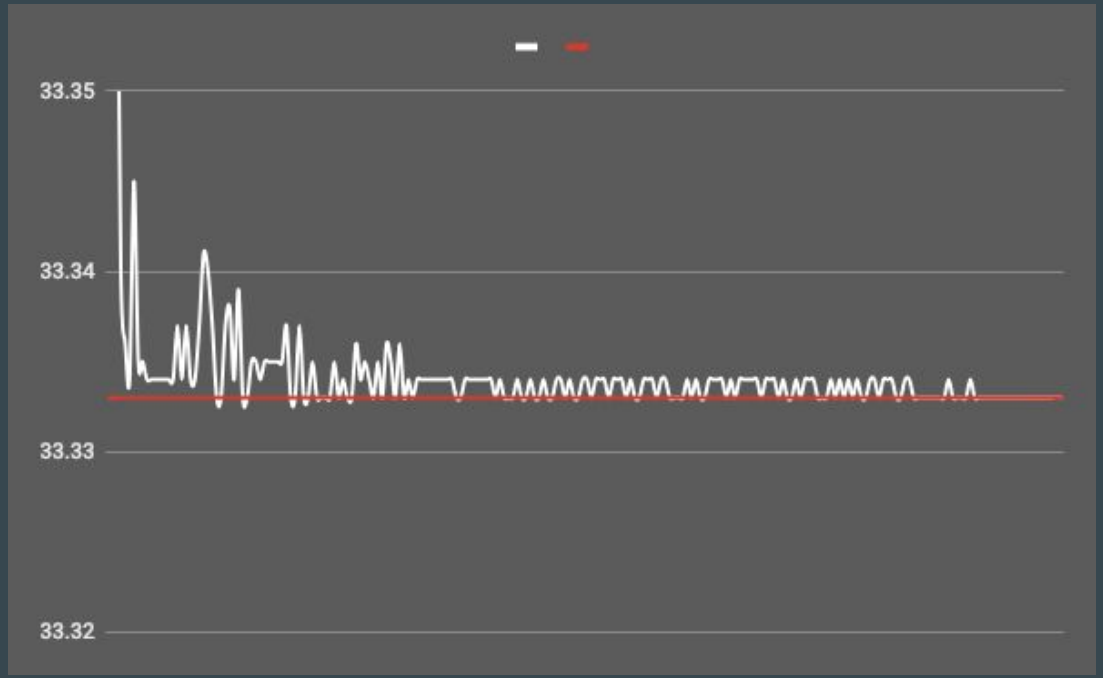
30 fps: 1/30 = 33.3$\overline{3}$ ms





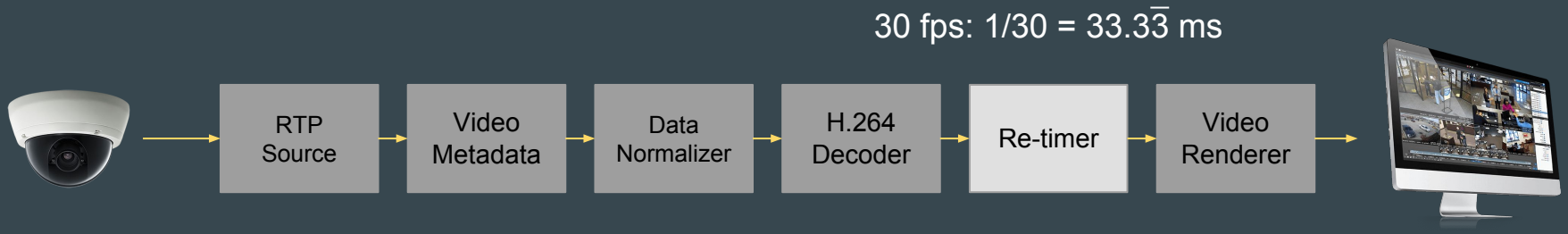

```
now = read_clock()
slept = now - prev
prev = now
sleep_time -= slept // 33.33 - 34.5 = -1.17
sleep_time += target // -1.17 + 33.33 = 32.16
sleep((int)sleep_time) // 32 ms
```

Live Video Pipeline Walk-thru



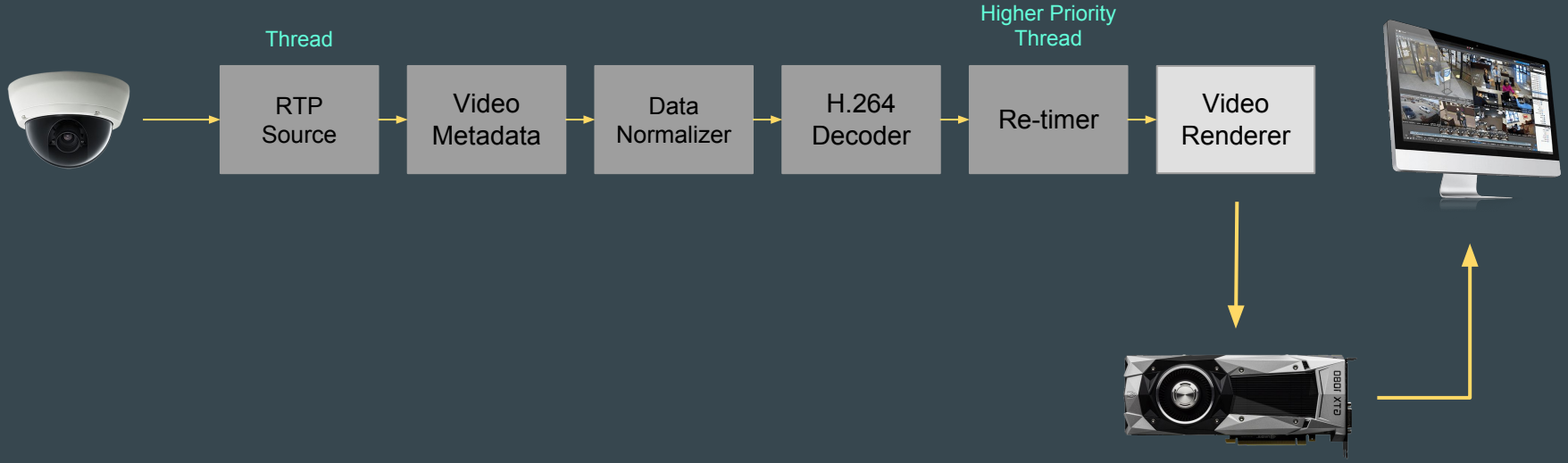
Average frame timing over 5 minutes

Live Video Pipeline Walk-thru

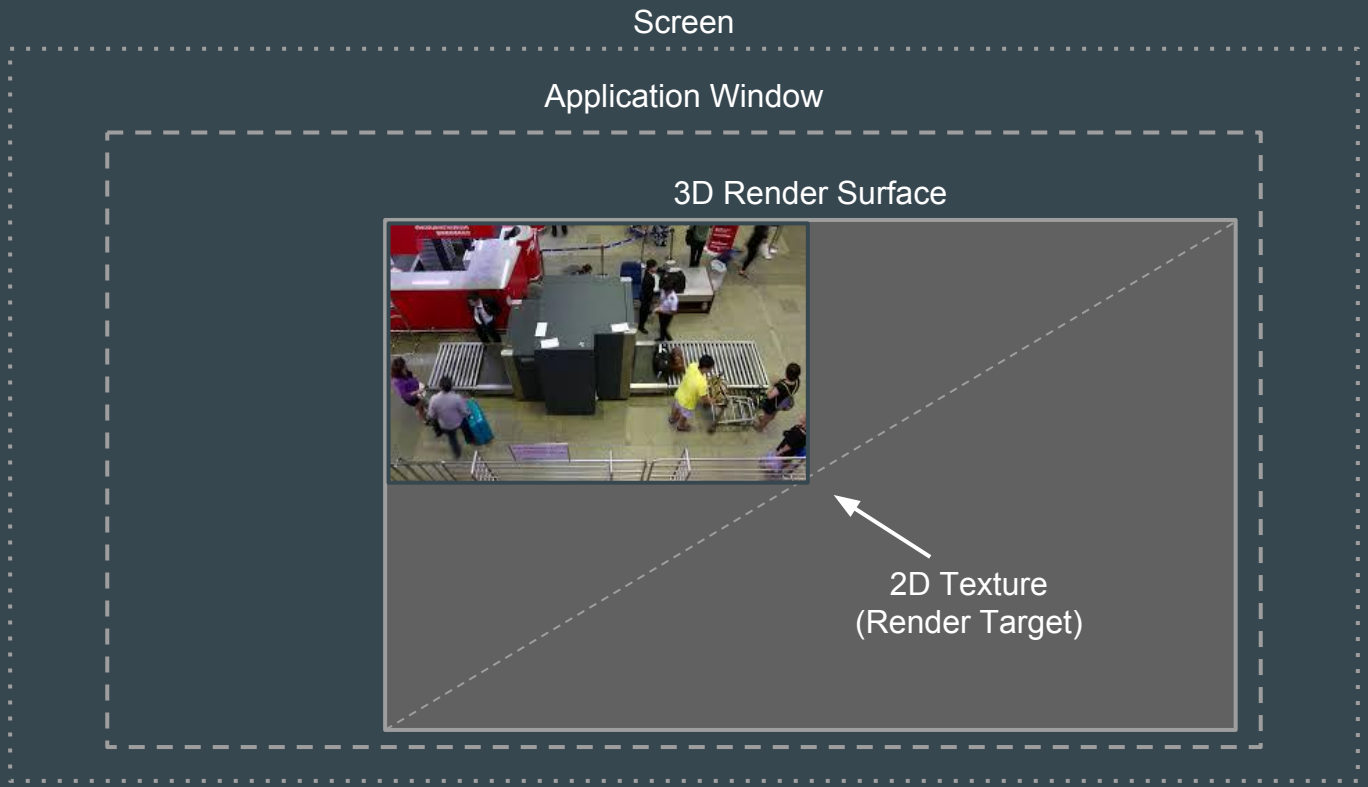



```
now = read_clock()
slept = now - prev
prev = now
sleep_time -= slept
sleep_time += target
sleep((int)sleep_time)
```

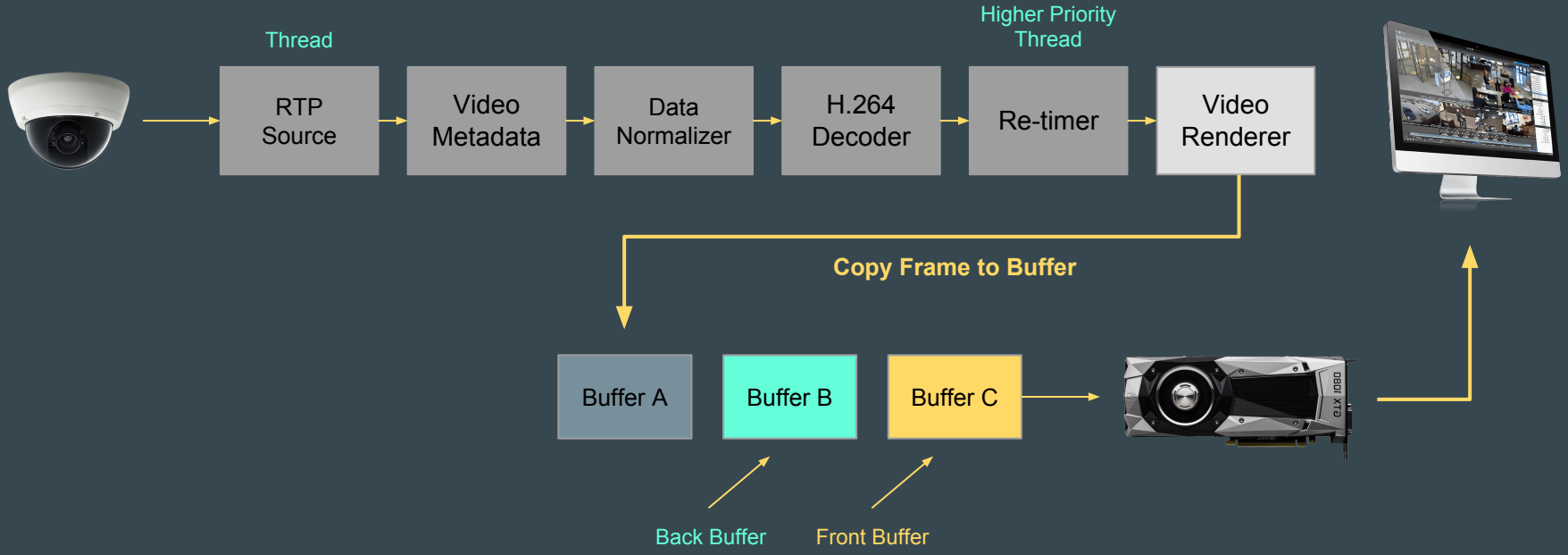
Live Video Pipeline Walk-thru



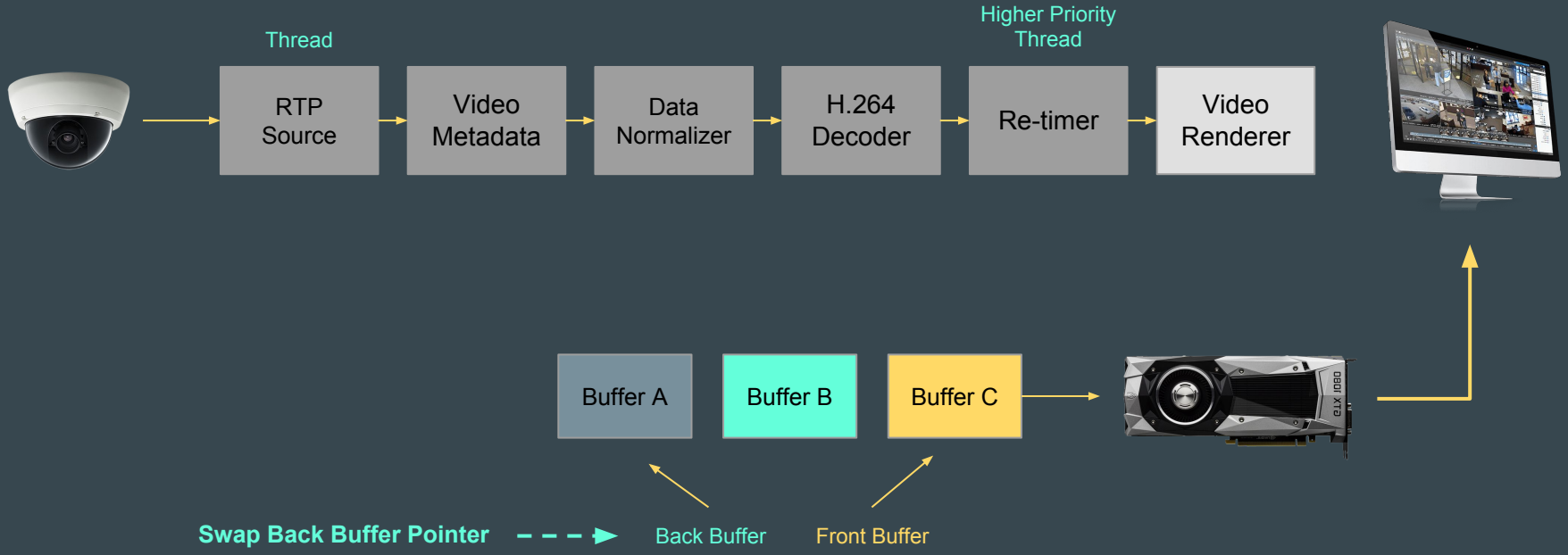
Live Video Pipeline Walk-thru



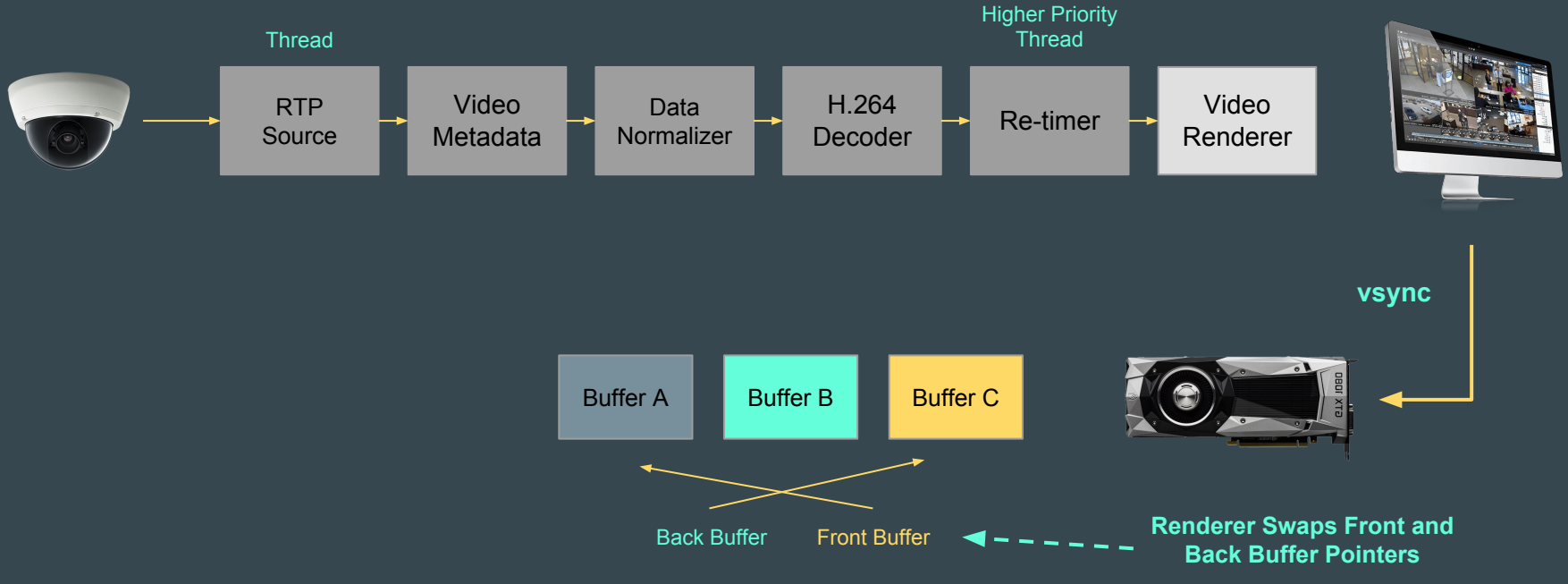
Live Video Pipeline Walk-thru



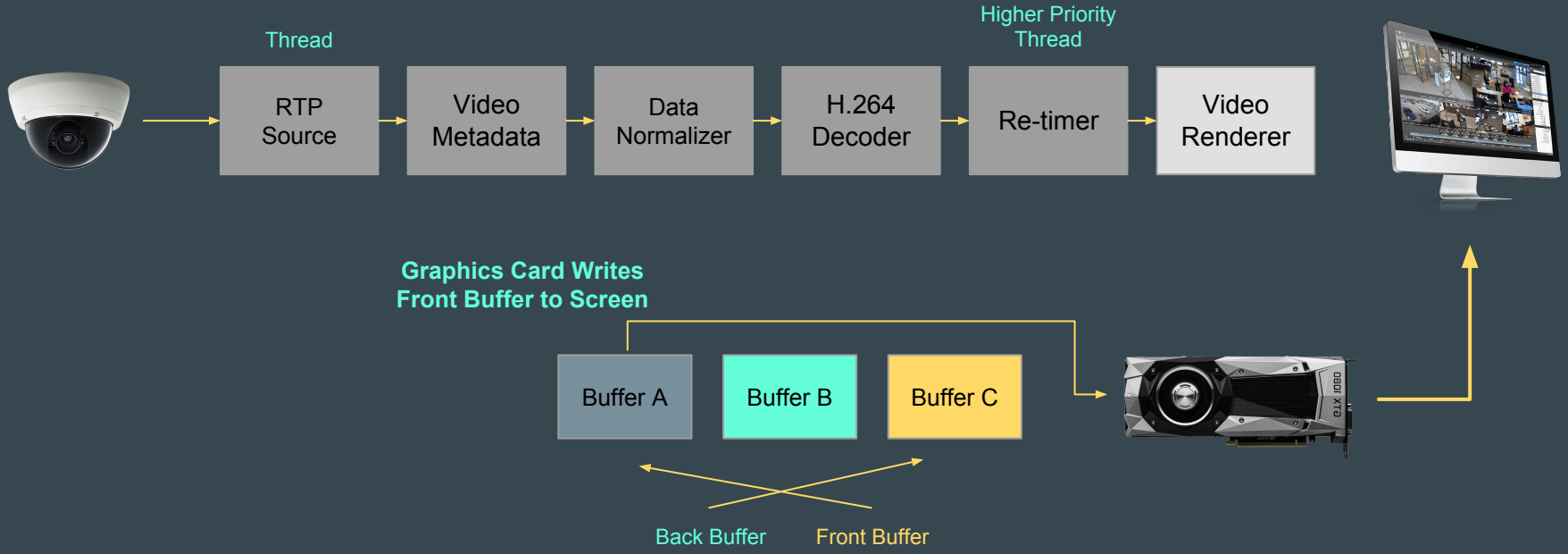
Live Video Pipeline Walk-thru



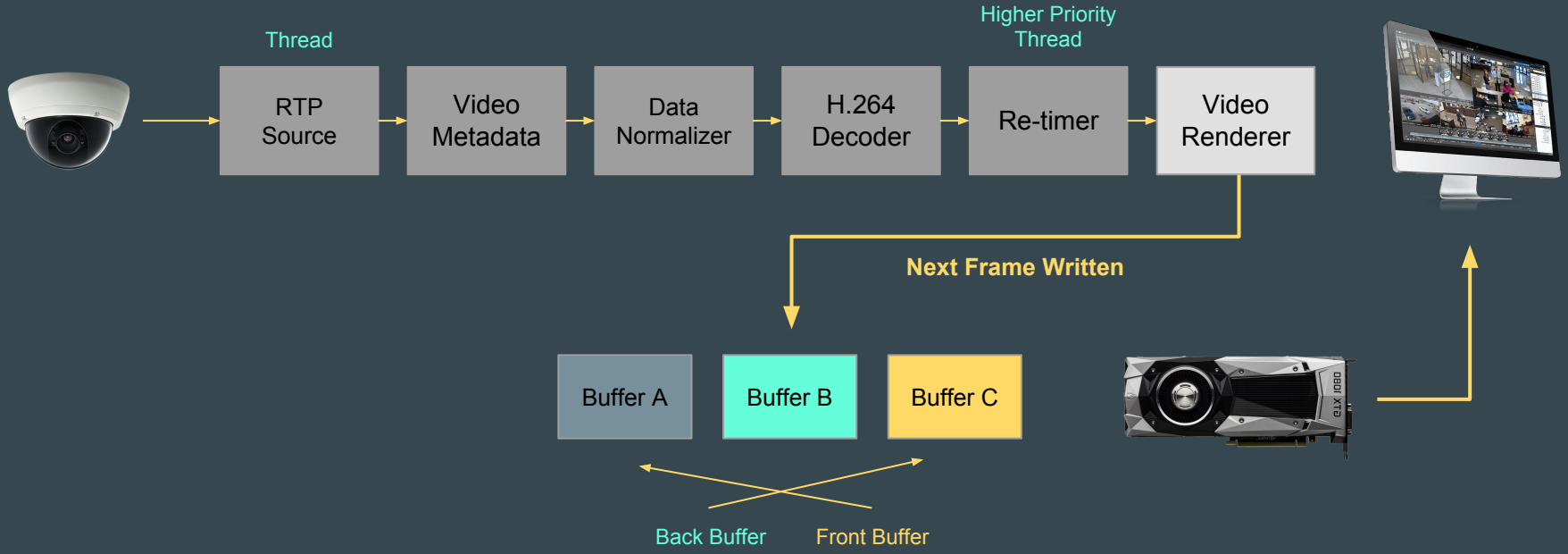
Live Video Pipeline Walk-thru



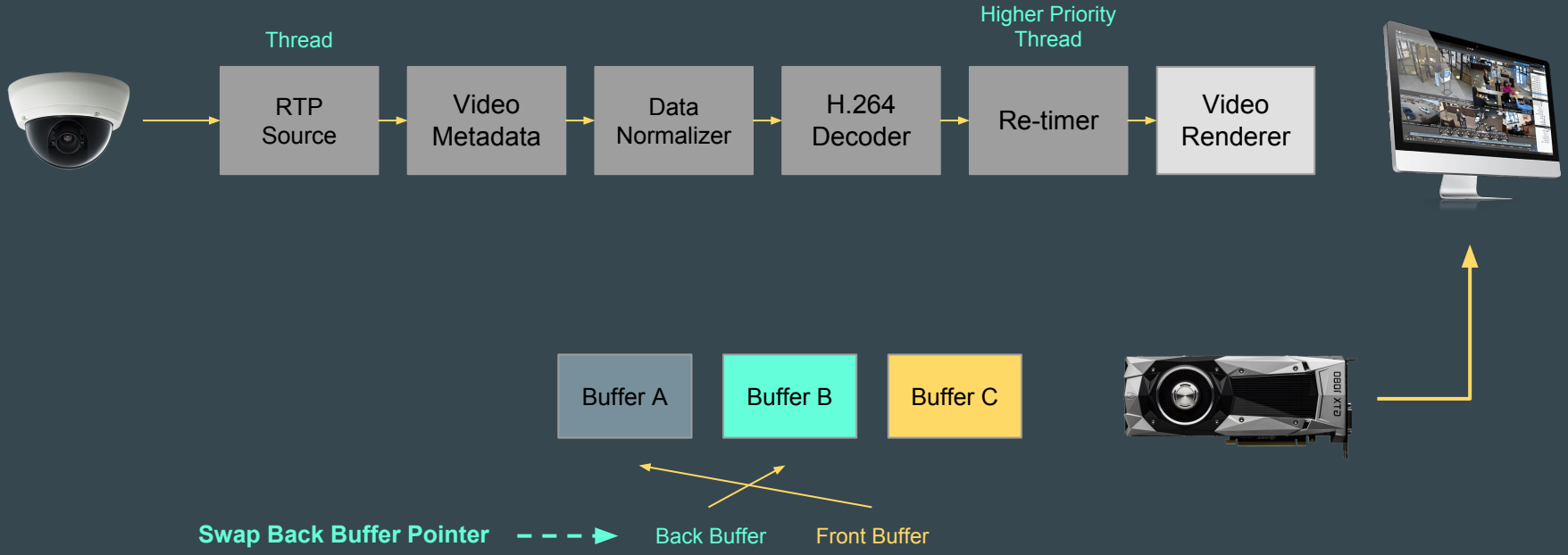
Live Video Pipeline Walk-thru



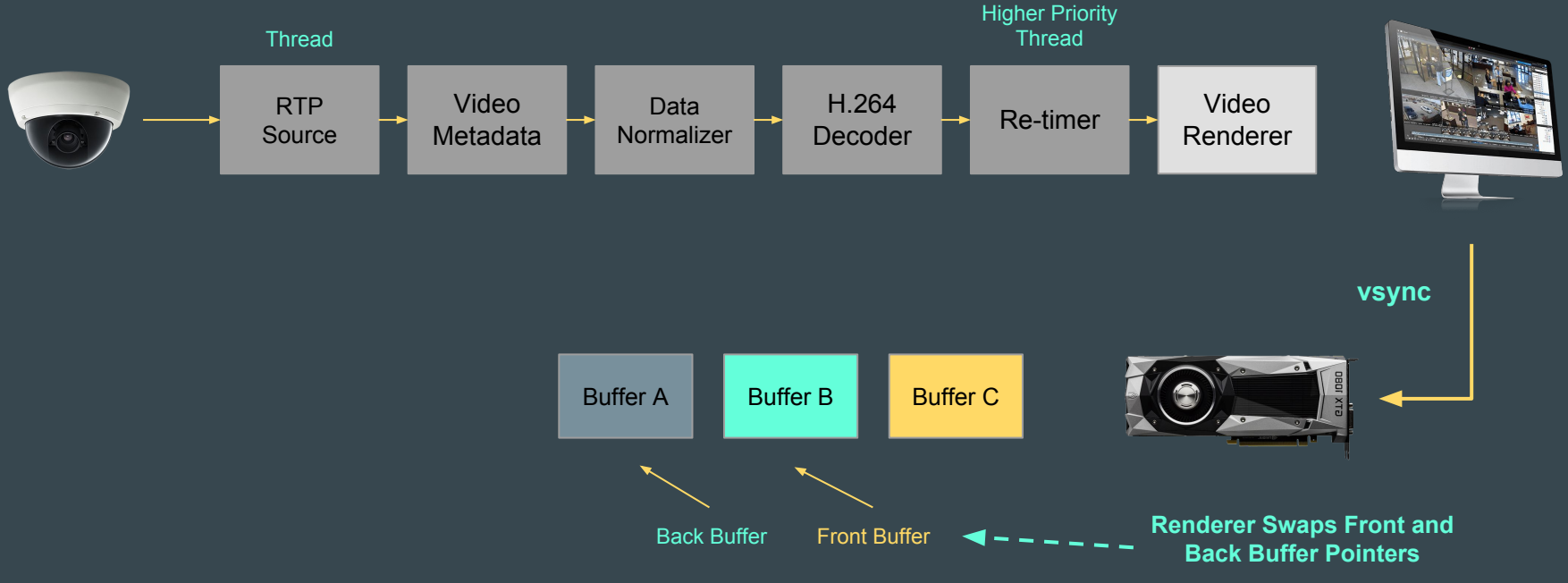
Live Video Pipeline Walk-thru



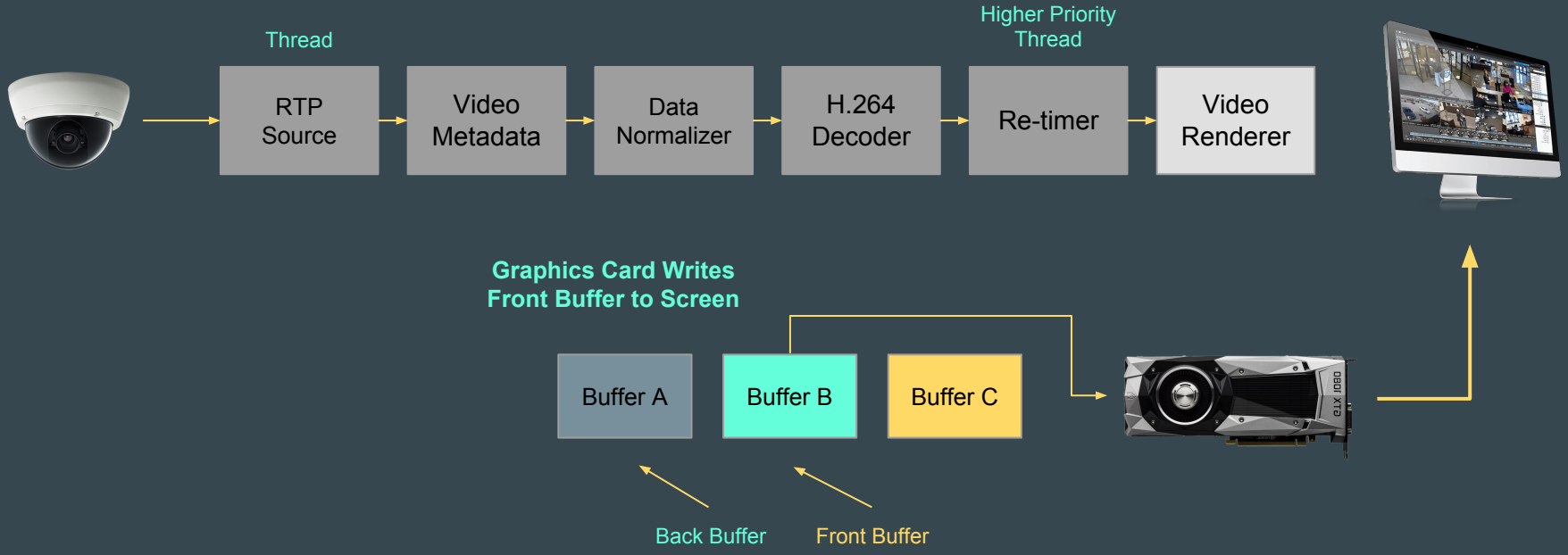
Live Video Pipeline Walk-thru



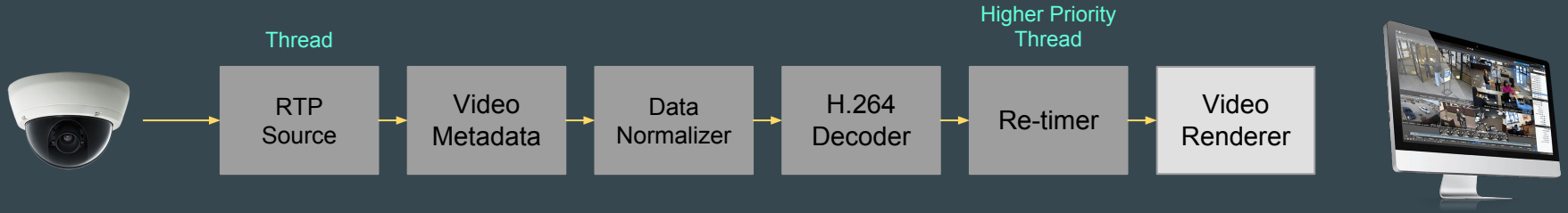
Live Video Pipeline Walk-thru



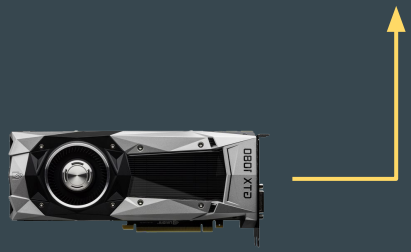
Live Video Pipeline Walk-thru



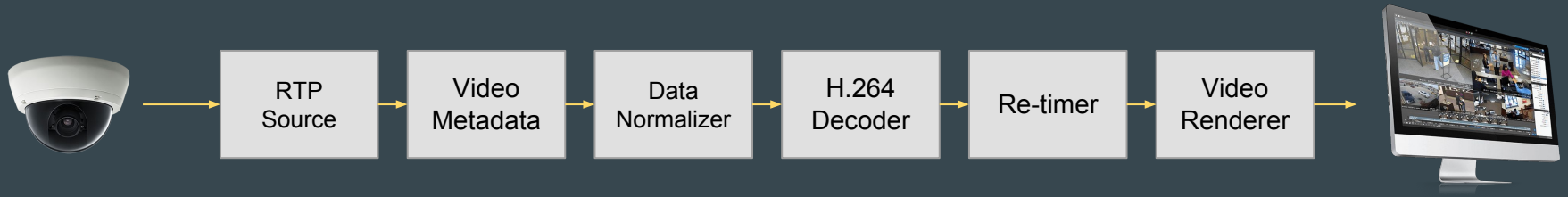
Live Video Pipeline Walk-thru



- GLSL pixel shaders running on GPUs:
  - Color conversion.
  - Brightness/contrast/level.



Live Video Pipeline Walk-thru



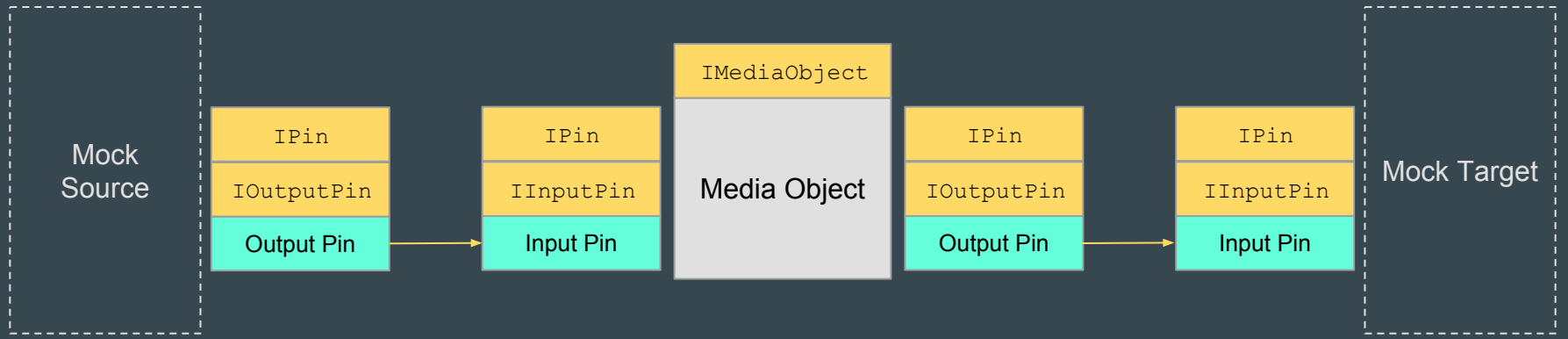
Questions so far?

Testing

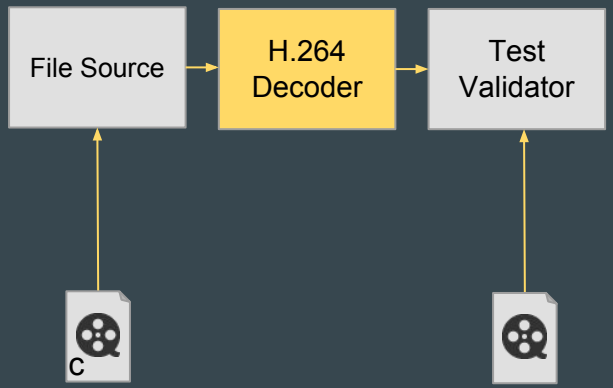
Test Methodology

- Everything was implemented in terms of interfaces, so mocking was easy.
- Everything had well defined inputs and outputs, so test harnessing was easy.
- `tcpreplay` and file exporting allowed us to capture real network streams and replay them:
  - Sample streams covering every supported codec, resolution, and framerate sped up development.
  - We captured every problem stream we encountered and wrote regression tests around them.
  - `tcpreplay` recreated network jitter, bursting, and dropped packet errors.
- We had different levels of testing:
  - Unit tests for the framework and each plugin (`cppunit`).
  - Pipeline integration tests running full pipelines with mocked sources and render targets.
  - Application integration tests using UI test tools and keyboard/joystick emulators.

Interfaces Allow Easy Mocking

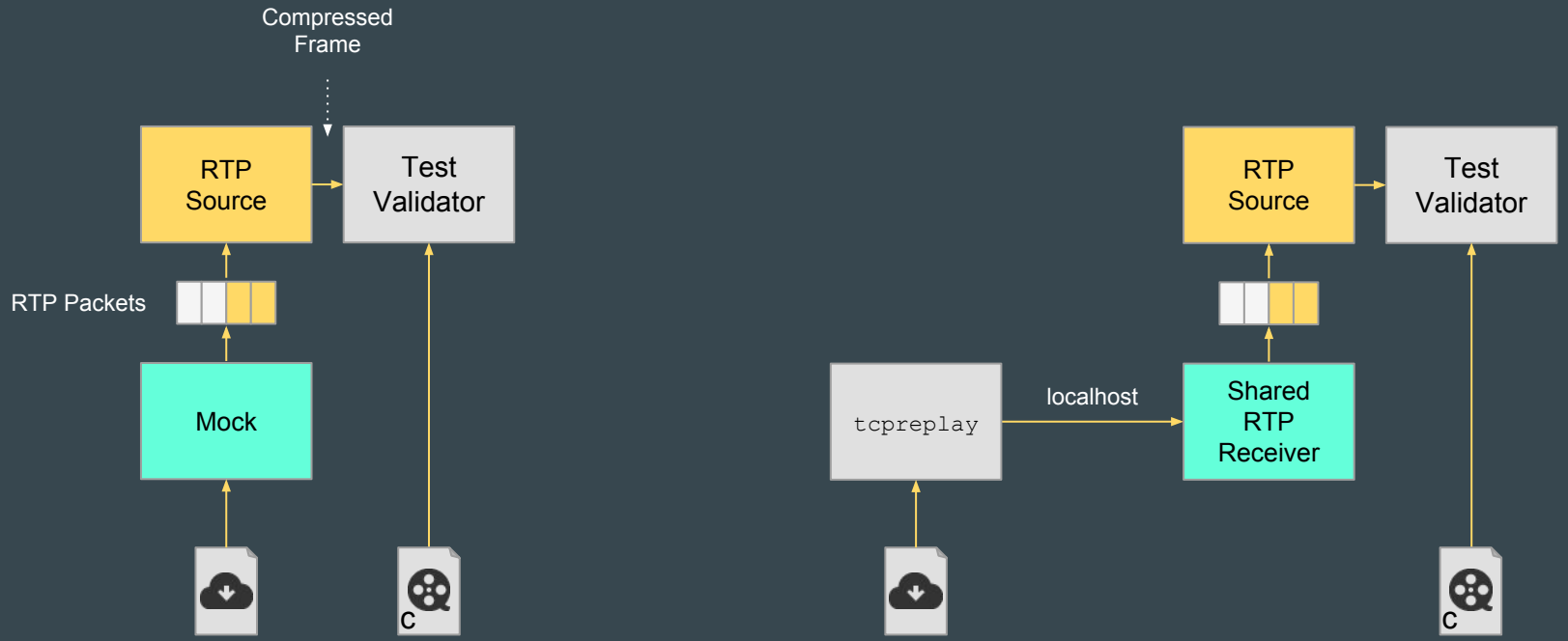


Media Object Testing

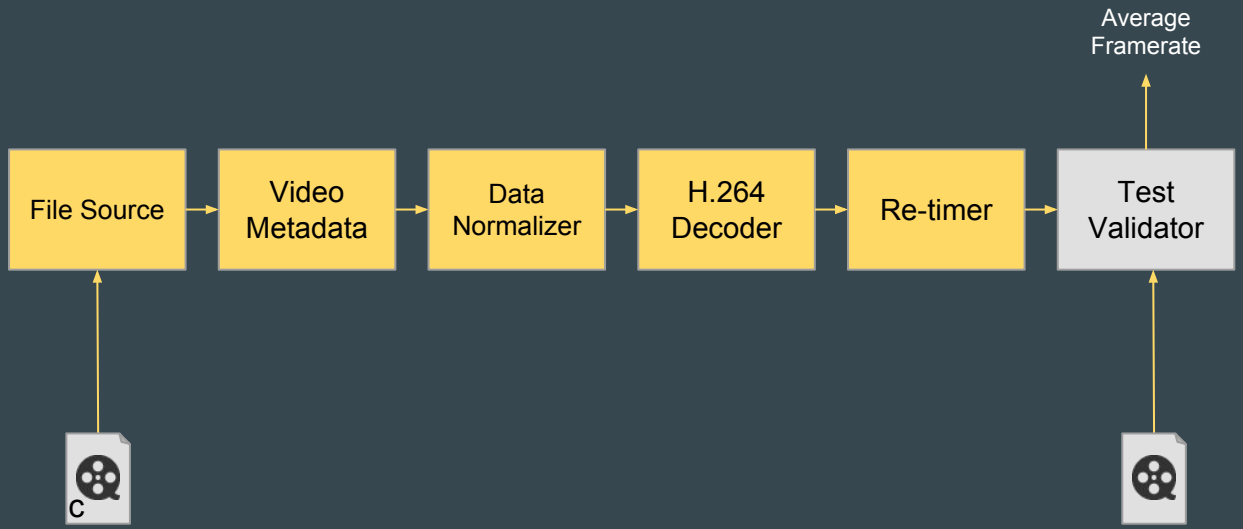


- Some test cases:
  - Valid stream
  - Wrong codec
  - Codec change
  - Resolution change
  - Missing frame
  - Corrupted header
  - Truncated frame
  - Decode Performance
  - ...

Media Object Testing



Pipeline Integration Testing



Pipeline Integration Testing

- We built a custom REPL shell + Lua programming language.
- Pipelines and media objects were first-class entities.
- Allowed us to script complex use cases:
  - Efficient for code-compile-test development cycle.
  - Great for recreating bugs (QA started using it to recreate bugs for us!)
  - Allowed us to interactively create test scenarios and export them easily.

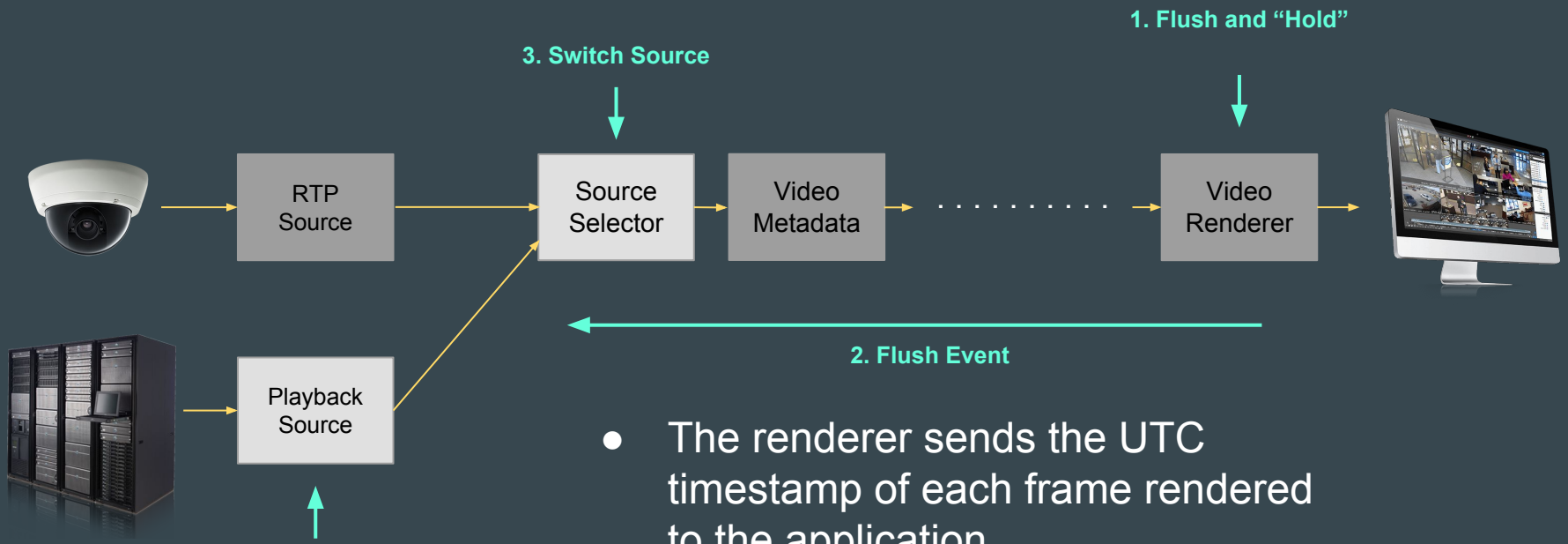
```
-- Build and run pipeline
local p = pipeline.new(template_name)
p:setrendertarget(rt)
local fsrc = p:getobject(file_source_name)
fsrc:setparam("filename", "./camera.mp4")
p:run(1.0)
...
```

The End

Questions?

Playback

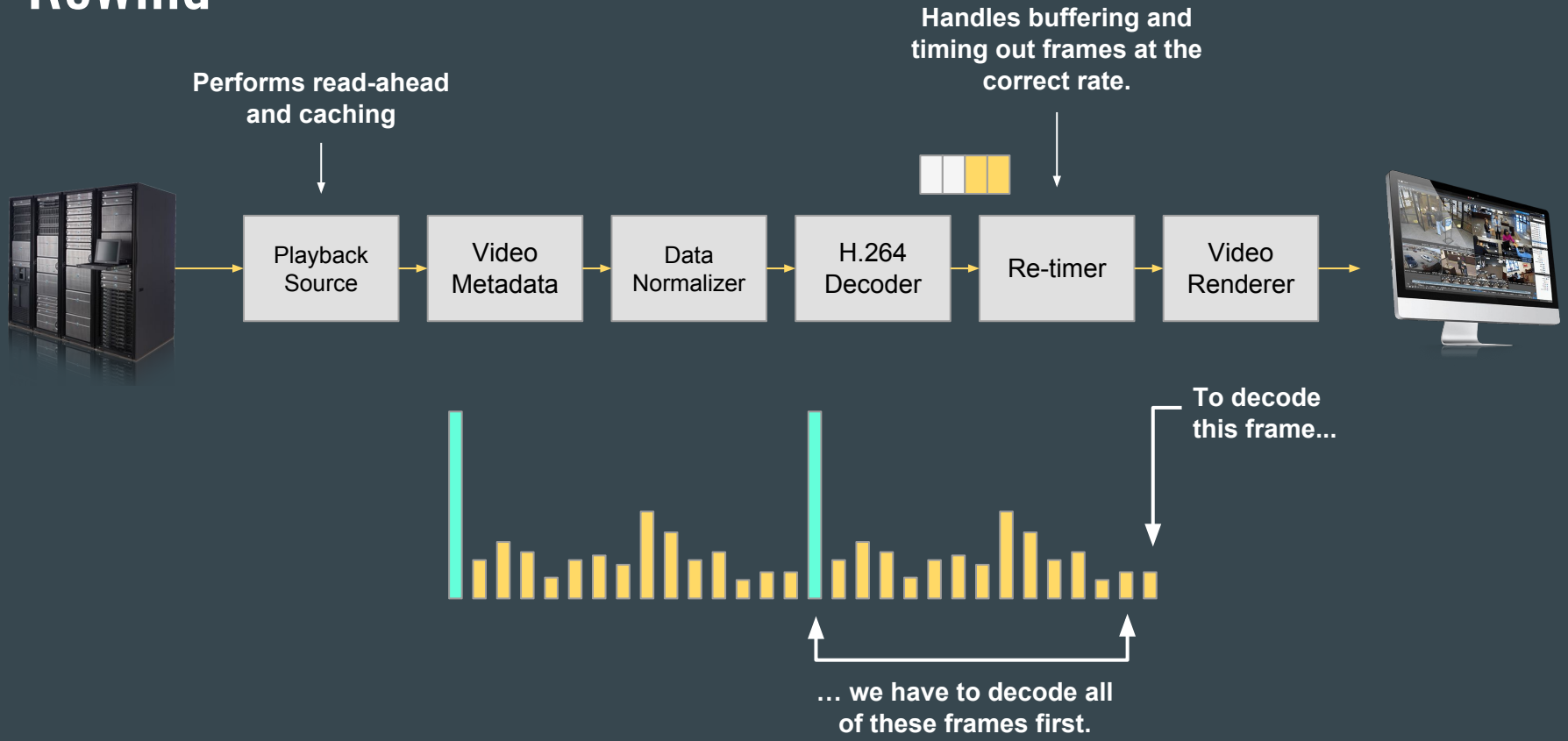
Live-to-Playback Transition



- The renderer sends the UTC timestamp of each frame rendered to the application.
- Necessitates client and storage system use NTP.

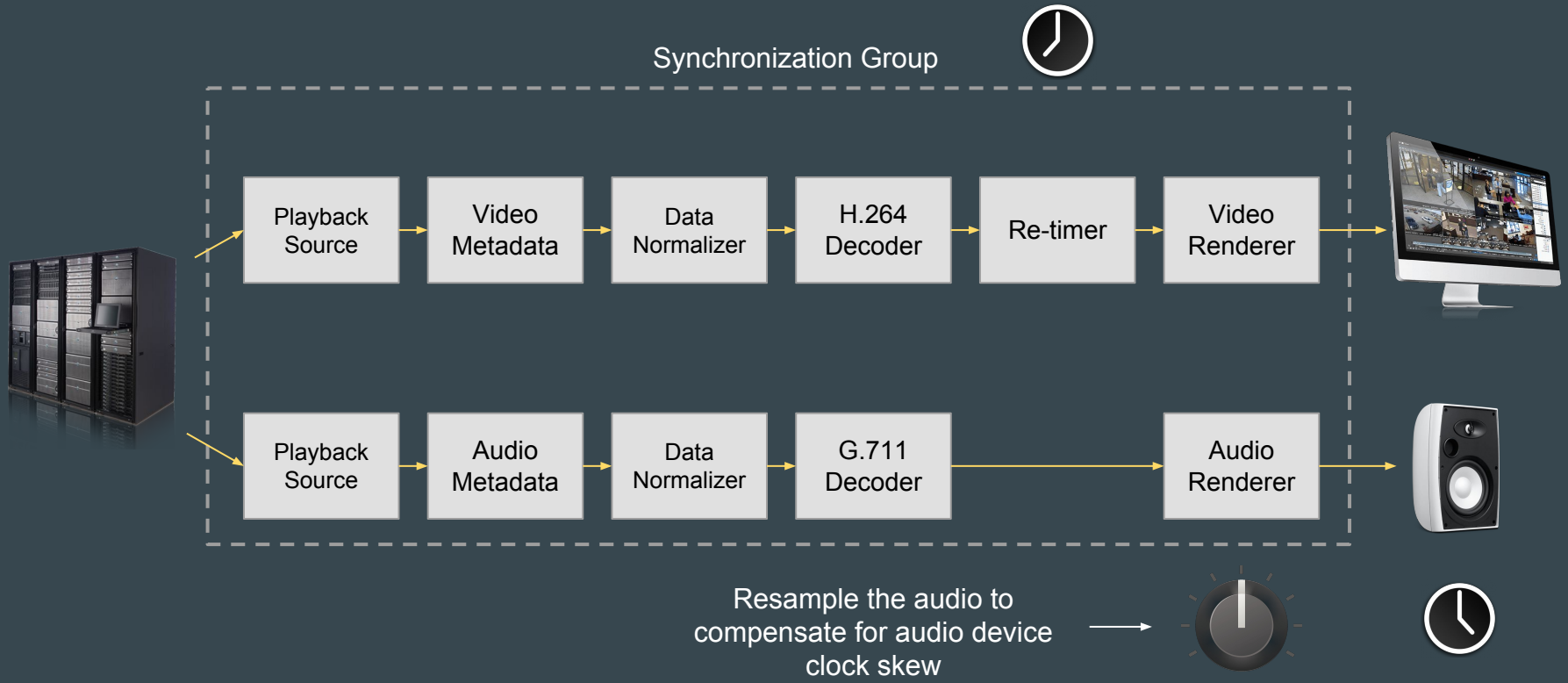
Rewind

Rewind


Performs read-ahead and caching
Handles buffering and timing out frames at the correct rate.
Playback Source
Video Metadata
Data Normalizer
H.264 Decoder
Re-timer
Video Renderer
To decode this frame...
… we have to decode all of these frames first.

Audio

Audio Synchronization



Low-latency mode

Low-latency Mode

3. Camera changes RTP packet type from 98 to 97

4. Source emits “low-latency” event. Re-timer subscribes to this event. Source sets “low-latency” bit on each buffer.

RTP Source → Video Metadata → Data Normalizer → H.264 Decoder → Re-timer → Video Renderer

5. Re-timer flushes buffers to renderer. Disables all buffering and timing

2. Keyboard sends PTZ command

1. User begins PTZ

Low-latency Mode

3. Camera changes RTP packet type back to 98

4. Source stops setting “low-latency” bit on buffers.

5. Re-timer exits low-latency mode and reenables buffering and timing

RTP Source → Video Metadata → Data Normalizer → H.264 Decoder → Re-timer → Video Renderer

6. Re-timer gradually builds up buffer by degrading framerate.

2. Keyboard stops sending PTZ commands

1. User ends PTZ