



KLE Technological
University
Creating Value
Leveraging Knowledge

BVB Campus, Vidyanagar, Hubballi – 580031, Karnataka, INDIA.

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Mini Project Report

On

Camera Virtualization for Android Automotive Operating System

submitted in partial fulfillment of the requirements for the award of the degree
of

Bachelor of Engineering

IN

COMPUTER SCIENCE AND ENGINEERING

Submitted By

Tulip Maurya 01FE22BCS073

Koustubh Kulkarni 01FE22BCS242

Harsh Goyal 01FE22BCS234

Nikhil B 01FE22BEC205

Under the guidance of

Mr. K M M Rajashekharaiiah

School of Computer Science and Engineering



KLE Technological
University
Creating Value
Leveraging Knowledge

BVB Campus, Vidyanagar, Hubballi – 580031, Karnataka, INDIA.

2024-2025

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that project entitled “Camera Virtualization for AAOS” is a bonafied work carried out by the student team Tulip Maurya-01fe22bcs073 , Koustubh Kulkarni-01fe22bcs242 , Harsh Goyal-01fe22bcs234 ,Nikhil B-01fe22bec205, in partial fulfillment of the completion of 5th semester B. E. course during the year 2024 – 2025. The project report has been approved as it satisfies the academic requirement with respect to the project work prescribed for the above said course.

Guide Name

Mr. K M M Rajashekharaiiah

Head , SoCSE

Dr. Vijayalakshmi M.

External Viva-Voce

Name of the examiners

Signature with date

1 _____

2 _____

ABSTRACT

This project focuses on the development of a camera virtualization solution for improving real-time video streaming in virtualized environments, specifically between a Linux virtual machine (VM) and an Android VM. The goal is to enable the Linux VM, acting as a host, to manage physical camera hardware and provide a virtual camera interface to the Android VM, which acts as the guest. The system utilizes VirtIO, a protocol designed for efficient device communication in virtualized environments, to facilitate shared memory access between the host and guest, ensuring low-latency and high-throughput data transfer. FFmpeg is employed for capturing and encoding video frames on the Linux side, which are then streamed to the Android VM. This setup eliminates the need for additional physical hardware in the Android VM while enabling Android applications to interface with a virtual camera as if it were a physical device. Although the Android Camera HAL remains unmodified due to hardware limitations and a lack of reference documentation, the project lays the groundwork for a flexible, scalable solution for camera virtualization in virtualized environments, offering significant improvements in workplace safety and monitoring systems.

Keywords : *Camera Virtualization ,Adroid VM, Video Streaming, Linux VM, Android Camera HAL.*

ACKNOWLEDGEMENT

We would like to thank our faculty and management for their professional guidance towards the completion of the mini project work. We take this opportunity to thank Dr. Ashok Shettar, Pro-Chancellor, Dr. P.G Tewari, Vice-Chancellor and Dr. B.S.Anami, Registrar for their vision and support.

We also take this opportunity to thank Dr. Meena S. M, Professor and Dean of Faculty, SoCSE and Dr. Vijayalakshmi M, Professor and Head, SoCSE for having provided us direction and facilitated for enhancement of skills and academic growth.

We thank our guide Mr. K M M Rajashekharaiiah, Associate professor and SoCSE for the constant guidance during interaction and reviews.

We extend our acknowledgment to the reviewers for critical suggestions and inputs. We also thank Project coordinator Dr. Uday Kulkarni, and reviewers for their suggestions during the course of completion. n. We extend our sincere gratitude to NVIDIA for providing us with the opportunity to work on this project and for their invaluable support. We are deeply thankful to our mentor Mr. Igor Mitsyanko , for his exceptional guidance and insightful feedback throughout the course of this work. his expertise and encouragement have been instrumental in the successful completion of this project We express gratitude to our beloved parents for constant encouragement and support.

Tulip Maurya- 01FE22BCS073

Koustubh Kulkarni- 01FE22BCS242

Harsh Goyal- 01FE22BCS234

Nikhil B - 01FE22BEC205

CONTENTS

ABSTRACT	i
ACKNOWLEDGEMENT	i
CONTENTS	iii
LIST OF FIGURES	iv
1 INTRODUCTION	1
1.1 Preamble	1
1.2 Motivation	1
1.3 Objectives of the project	2
1.4 Literature Review	2
1.5 Problem Definition	3
2 SOFTWARE REQUIREMENT SPECIFICATION	4
2.1 Overview of SRS	4
2.2 Requirement Specifications	4
2.2.1 Functional Requirements	4
2.2.2 Use case diagrams	5
2.2.3 Usecase Descriprion	5
2.2.4 Nonfunctional Requirements	6
2.3 Software and Hardware Requirement Specifications	7
2.3.1 Software Requirements	7
2.3.2 Hardware Requirements	7
3 PROPOSED SYSTEM	8
3.1 Description of Proposed System.	8
3.1.1 Modules of the Framework	8
3.2 Description of Target Users	10
3.2.1 Stakeholders	10
3.2.2 Design Principles Identified and Used	10
3.3 Advantages/Applications of Proposed System	11
3.3.1 Advantages	11
3.3.2 Applications	11
3.4 Scope of Proposed System	11

4	SYSTEM DESIGN	13
4.1	Architecture of the system (explanation)	14
4.2	Sequence diagram	15
5	IMPLEMENTATION	19
5.1	Proposed Methodology	19
5.1.1	Flowchart Description	19
5.1.2	Capture Video Frames in Linux	20
5.1.3	Share Frames to Android VM	21
5.1.4	Android VM Integration	22
5.1.5	Modification of Camera HAL	23
6	RESULTS DISCUSSIONS	24
6.1	Results	24
6.2	Discussion and Challenges	25
6.3	Summary of Results	26
7	CONCLUSIONS AND FUTURE SCOPE	27
	REFERENCES	28
8	Plagiarism Report	29

LIST OF FIGURES

2.1	use case diagram for camera initialization	5
3.1	Modules in our framework	9
4.1	Architecture for Virtualized Camera Access in an Automotive Infotainment System	13
4.2	sequence diagram for capturing frames in linux through v4l2 camera	16
4.3	sequence diagram for frame delivery process	17
4.4	Architecture for Virtualized Camera Access in an Automotive Infotainment System	18
5.1	flowchart	19
5.2	frames captured in linux	20
5.3	frames stored in shared folder	21
5.4	Shared folder being accessed in android VM	21
5.5	UDP stream on Android-X86 using VLC media player	22
6.1	frames captured in linux using ffmpeg	24
6.2	shared folder can be accessed	25
6.3	UDP stream on Android-X86 using VLC media player	25

Chapter 1

INTRODUCTION

This project focuses on developing a camera virtualization solution that enables real-time video streaming between a Linux virtual machine (VM) and an Android VM. The Linux VM acts as the host, managing physical camera hardware, while the Android VM, serving as the guest, accesses the camera feed through a virtual camera interface. By utilizing VirtIO, a high-performance communication protocol for virtualized environments, and FFmpeg for video encoding, this project aims to provide an efficient, low-latency solution for video streaming. This system allows Android applications to interact with the virtual camera as if it were a physical device, offering a scalable approach to camera virtualization in virtualized environments.

1.1 Preamble

This project aims to implement a camera virtualization system that facilitates video streaming between a Linux virtual machine (VM) and an Android VM. The Linux VM, acting as the host, will manage the physical camera hardware and provide a virtual camera interface to the Android VM. Using VirtIO, a protocol designed for efficient device communication in virtualized environments, the system will leverage shared memory to enable low-latency, high-throughput data transfer. FFmpeg will capture and encode video frames on the Linux side, which will be streamed to the Android VM. This solution eliminates the need for additional physical hardware in the Android VM while enabling seamless camera functionality.

1.2 Motivation

The rapid growth of virtualization technologies has created a demand for more efficient resource sharing between host and guest systems, particularly in cloud and development environments. In scenarios where real-time video capture and streaming are essential, such as surveillance, testing, and virtualized environments, traditional methods like USB passthrough or network streaming often introduce latency and bandwidth limitations. This project aims to overcome these challenges by providing a software-driven solution that allows for efficient, low-latency video streaming between VMs. The ability to virtualize camera devices enhances

system scalability and flexibility, especially in safety-critical applications like PPE compliance monitoring and automated surveillance.

1.3 Objectives of the project

The main objective of this project is to develop a camera virtualization system that enables seamless video streaming between a Linux virtual machine (VM) and an Android VM. This involves capturing video frames on the Linux VM and streaming them to the Android VM using VirtIO for efficient communication.

- Develop a virtual camera driver on the Linux VM to capture video from physical camera hardware.
- Implement a shared memory mechanism using VirtIO for low-latency video frame transfer to the Android VM.
- Integrate FFmpeg for efficient video capture, encoding, and streaming.
- Modify the Android Camera HAL to interface with the virtual camera as a native device.
- Ensure compatibility of the virtual camera with Android applications requiring camera functionality.

1.4 Literature Review

Camera virtualization in virtualized environments has garnered attention due to the increasing demand for flexible, scalable solutions in cloud and virtualized systems. Traditional approaches to camera sharing, such as USB passthrough or network-based streaming, face significant limitations in terms of bandwidth consumption, latency, and overheads. VirtIO, a protocol designed for low-latency communication in virtualized environments, has been recognized as an effective means to overcome these challenges. By providing direct access to shared memory, VirtIO enables efficient data transfer between the host and guest systems, making it ideal for applications such as video streaming and real-time media processing [1],[2] .

The Android Camera HAL (Hardware Abstraction Layer) provides a standardized interface for Android applications to access camera hardware, abstracting the complexities of different camera devices. The HAL ensures that Android applications can interact with camera hardware without needing to be aware of the underlying implementation details. The Android Compatibility Definition Document (CDD) for Android 14 defines the requirements for camera functionality, ensuring that the system supports various camera features and interfaces necessary for a wide range of devices [3]. The Camera HAL framework, as detailed in the

Android Compatibility Test Suite (CTS), further emphasizes the need for consistency and reliability across different camera implementations [4].

With the rise of virtualized environments, modifying the Android Camera HAL to support virtual cameras has become essential. The Camera HAL's modular architecture allows for the introduction of virtual camera devices, which can be managed by the Linux VM, providing access to the physical camera hardware. The integration of virtual cameras into the Android Camera HAL is a critical challenge, requiring seamless communication between the host and guest systems. The Camera Provider AIDL (Android Interface Definition Language) used in Android's hardware interfaces provides a means to interface with camera providers, making it a vital tool for ensuring compatibility with virtualized camera devices [5].

A significant challenge in camera virtualization is ensuring real-time performance, especially in terms of frame capture, encoding, and streaming. FFmpeg, an open-source multimedia framework, has been widely adopted for video capture and encoding due to its efficiency and versatility. FFmpeg allows for the real-time capture of video frames and encoding into various formats, ensuring that the video stream can be transmitted efficiently to the guest system [6], [7]. The combination of VirtIO for efficient data transfer and FFmpeg for video processing provides a robust solution for camera virtualization in virtualized environments.

Further research has explored similar virtualization techniques and their application in cloud-based video processing and surveillance systems. Existing work on cloud video streaming has demonstrated the feasibility of network-based camera sharing; however, such methods often struggle with high latency and quality degradation. The incorporation of VirtIO in these systems offers a promising approach to reduce these issues, while leveraging the flexibility of virtualization to create scalable solutions for camera access and video processing [8], [9].

1.5 Problem Definition

Design a system where the Android x86 virtual machine can access the physical camera hardware on the Linux host system, enabling seamless interaction for camera-dependent applications.

Chapter 2

SOFTWARE REQUIREMENT SPECIFICATION

2.1 Overview of SRS

This document outlines the requirements for the "Camera Virtualization for Real-Time Video Streaming in Virtualized Environments" system. The project aims to develop a camera virtualization solution that enables seamless video streaming between a Linux virtual machine (VM) and an Android VM, utilizing shared memory communication via VirtIO. By leveraging FFmpeg for video encoding and streaming, the system facilitates the efficient transfer of video frames from the host to the guest VM. This solution is intended to enhance the scalability, flexibility, and performance of camera access in virtualized environments, particularly for applications requiring real-time video processing and low-latency interactions.

2.2 Requirement Specifications

Functional requirements define the specific behaviors, features, and functions that the system must perform. These requirements focus on what the system is expected to do and outline the core tasks the system must accomplish to meet the objectives. They typically specify the interactions between the system and its users or other systems, as well as the system's response to certain inputs. In the context of this project, functional requirements may include capturing video frames from the physical camera, streaming them to a virtual machine, and ensuring that the guest system can interact with the virtual camera as if it were a physical device.

2.2.1 Functional Requirements

- FR1: The Android VM must perceive the virtual camera as a standard camera, ensuring compatibility with applications that require camera functionality.
- FR2: The Linux VM should share the camera feed with the Android VM, allowing simultaneous usage by both systems.

- FR3: The Linux VM should register image buffers that can be accessed by Android's camera service for real-time video streaming.
- FR4: A client-server protocol (such as VirtIO) should be used between the Linux host and Android VM to manage camera streams and metadata exchange efficiently.

2.2.2 Use case diagrams

The following diagram illustrates the interaction between the Android virtual machine, the camera application, and the Linux host system. It demonstrates the initialization, frame request, and transmission processes via the VirtIO protocol to enable seamless access to the physical camera.

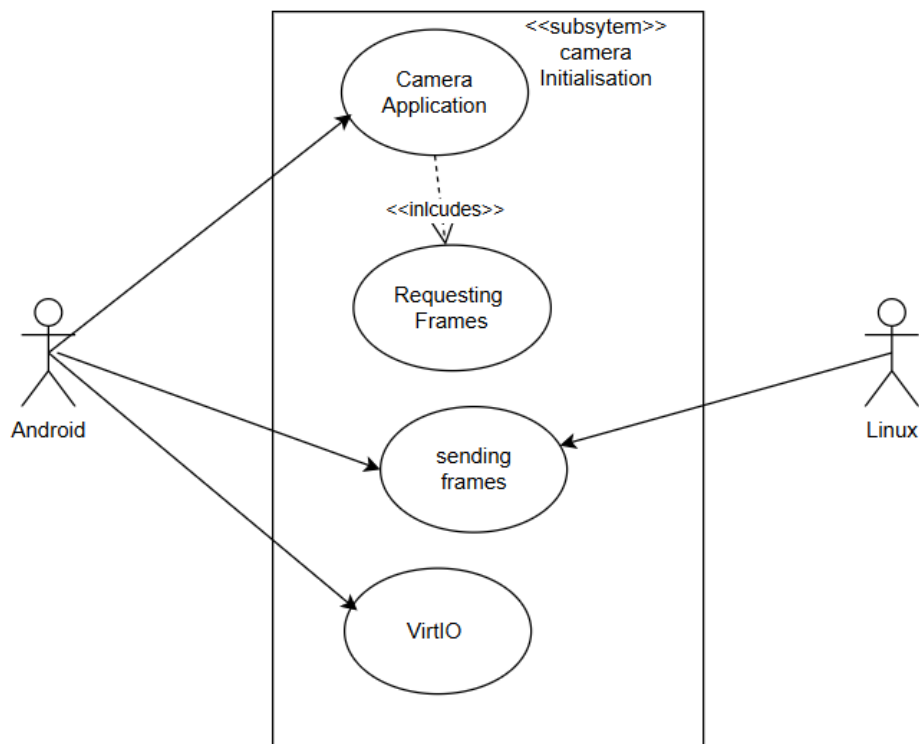


Figure 2.1: use case diagram for camera initialization

2.2.3 Usecase Descriprion

The system facilitates seamless camera access for an Android VM by enabling communication with the physical camera on the Linux host system. The diagram below illustrates the flow of interactions between the Android application, the VirtIO protocol, and the Linux host during camera initialization and frame transmission.

Use Case Name: Camera Access in Virtualized Environment

Actors:

- **Primary:** Android Application
- **Secondary:** Linux Host System

Goal: Enable the Android x86 virtual machine to access and utilize the physical camera on the Linux host system.

Preconditions:

- The Linux system must have a v4l2 physical camera.
- The VirtIO communication protocol must be set up between the Android VM and the Linux host.
- Camera initialization within the Android VM is completed.

Trigger: The Android application requests frames from the camera.

Flow of Events

Basic Flow:

1. The camera application in Android sends a request for frames.
2. The request is processed by the camera initialization subsystem.
3. subsystem interfaces with VirtIO to communicate with the Linux host system.
4. Frames are captured on the Linux host system and sent back to the Android VM via VirtIO.
5. The camera application receives the frames for processing.

Alternate Flow: If the camera initialization fails, the application logs an error, and the process terminates.

Postconditions:

- Frames are successfully transmitted to the Android application.
- The Android VM establishes a real-time connection with the camera through VirtIO.

Exceptions:

Communication errors in VirtIO.

Camera hardware not detected on the Linux host.

2.2.4 Nonfunctional Requirements

- NFR1: The system should minimize latency and reduce the need for copying image data between the Linux host and Android VM to optimize real-time performance.
- NFR2: The physical camera must remain fully controlled by the Linux VM to avoid interference or conflicts with the Android VM's access.
- NFR3: The system should be scalable, allowing support for multiple camera feeds and the integration of various camera applications in Android without requiring major changes to the underlying infrastructure.

2.3 Software and Hardware Requirement Specifications

The successful implementation of the camera virtualization system requires specific software and hardware configurations. These specifications are detailed below:

2.3.1 Software Requirements

- A Linux-based operating system, such as Ubuntu, to serve as the host environment.
- Android-x86 to run the Android VM.
- QEMU as the hypervisor to manage the virtual machines.
- Source code of Android AAOS (Android Automotive OS) for system modifications and integration.
- Protocols such as VirtIO for efficient shared memory communication between the Linux host and Android VM.
- FFmpeg for video frame encoding and streaming.

2.3.2 Hardware Requirements

- 64 GB of RAM to handle the computational requirements of building and running the system.
- 400 GB of disk space for building the Android source code and managing virtual machine files.
- A modern multi-core CPU to support virtualization and real-time video processing.
- A v4L2 camera compatible with the Linux VM for video capture.

Chapter 3

PROPOSED SYSTEM

3.1 Description of Proposed System.

The proposed system focuses on implementing a camera virtualization framework to enable real-time video streaming between a Linux-based host and an Android-x86 virtual machine (VM). The system utilizes VirtIO for efficient shared memory communication, allowing low-latency transfer of video frames and metadata. FFmpeg is employed to capture, encode, and stream video data, ensuring compatibility with various formats and applications.

The Linux VM acts as the host, directly managing the physical camera hardware, capturing video frames, and registering image buffers. The Android VM perceives the virtual camera as a standard device, allowing seamless integration with Android applications and services. A client-server protocol is implemented to facilitate communication between the host and guest VMs, ensuring reliable metadata exchange and frame synchronization.

The system supports scalability for handling multiple camera streams and aims to provide robust fault tolerance, maintaining stability even in the event of communication disruptions. This virtualization approach is designed for high-risk environments, enhancing the usability of Android VMs in applications such as AI-driven video analytics, training simulators, and testing environments.

By leveraging modern virtualization and video streaming technologies, this system addresses latency and scalability challenges, offering an efficient and reliable solution for camera virtualization in virtualized environments.

3.1.1 Modules of the Framework

The camera virtualization system consists of several interdependent modules that work together to provide seamless virtualized access to camera hardware. The key modules are described below:

- **Camera Virt Host:** This core module is responsible for managing and controlling the virtualized camera functions. It interacts with the physical camera hardware and provides virtualized access to other modules and applications. It ensures efficient operation and facilitates communication between the host and guest systems.

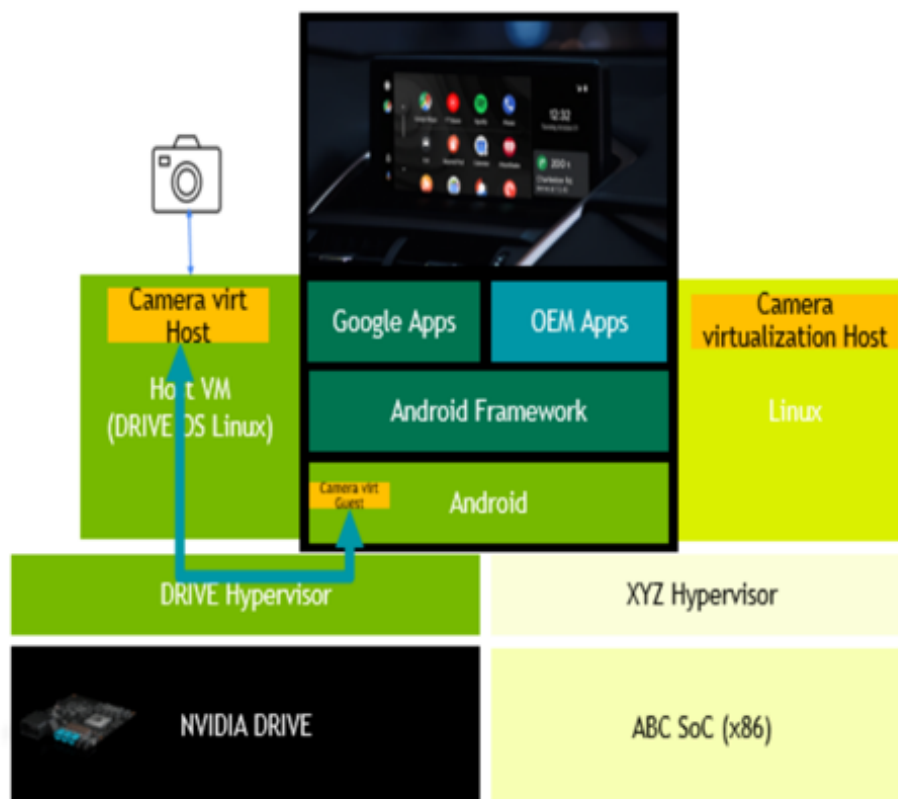


Figure 3.1: Modules in our framework

- **Android Framework:** This is the central framework of the Android operating system. It provides the foundational services and APIs necessary to run Android applications, including those that require camera functionality. It integrates with the virtual camera interface to enable application-level access.
- **DRIVE Hypervisor:** A specialized hypervisor that enables multiple operating systems, such as Linux and Android, to run concurrently on the same hardware. It ensures resource isolation and facilitates communication between the Android VM and the Linux VM, leveraging the NVIDIA DRIVE platform for optimized performance.
- **Camera Virtualization Host:** This module manages the physical camera hardware and provides virtualized access to it. It captures video frames, processes them if necessary, and shares them with the guest system (Android VM) through shared memory or other communication protocols like VirtIO.
- **Shared Memory Interface:** While not explicitly listed, this interface is crucial for low-latency communication between the host and guest systems. It ensures efficient transfer of video frames and metadata, minimizing performance bottlenecks.: The Figure 6.3

3.2 Description of Target Users

3.2.1 Stakeholders

The key stakeholders of the proposed camera virtualization system include:

- **System Developers:** Developers working on virtualized environments, such as QEMU and Android-x86, who require virtual camera functionality for testing and debugging.
- **Researchers:** Researchers in AI, computer vision, and video analytics who use virtualized cameras for training machine learning models and testing real-time applications.
- **Organizations:** Companies and institutions involved in high-risk industries, AR/VR development, and hardware/software integration that require robust camera virtualization solutions for safety monitoring and testing.
- **End Users:** Individuals or teams using Android VMs for specific applications, including simulation environments, AI pipelines, and mobile app testing.

3.2.2 Design Principles Identified and Used

The following design principles were identified and used to guide the development of the system:

- **Scalability:** The system is designed to handle multiple camera feeds and accommodate various Android applications without requiring significant reconfiguration. This ensures adaptability to evolving user needs.
- **Low Latency:** By leveraging shared memory protocols such as VirtIO, the system minimizes latency during video frame transfers, ensuring real-time performance for critical applications.
- **Modularity:** The system architecture follows a modular approach, enabling independent development and maintenance of components such as the camera virtualization host, guest driver, and Android HAL.
- **Interoperability:** The system integrates seamlessly with standard Android frameworks, ensuring compatibility with a wide range of camera-dependent applications.
- **Efficiency:** Resource utilization is optimized, with minimized copying of image buffers between the Linux host and Android VM, reducing overhead and enhancing performance.
- **Fault Tolerance:** The system ensures stability and reliability, maintaining functionality even in the event of communication disruptions or hardware failures.

These principles collectively ensure that the system meets the requirements of its stakeholders while delivering high performance and ease of use in virtualized environments.

3.3 Advantages/Applications of Proposed System

The proposed camera virtualization system provides essential benefits and practical applications. By leveraging advanced virtualization and communication protocols, it addresses key challenges in virtualized environments while offering significant utility across diverse domains.

3.3.1 Advantages

- **Low Latency:** Shared memory protocols like VirtIO minimize latency, enabling real-time video streaming between the Linux host and Android VM.
- **Scalability:** The system supports multiple camera feeds and accommodates various applications without major architectural changes.
- **Seamless Integration:** The solution integrates effectively with Android frameworks, ensuring compatibility with existing camera-dependent applications.

3.3.2 Applications

- **AI and Machine Learning:** Enables real-time video streaming for training and testing object detection and video analytics models.
- **Software Testing:** Provides developers with a virtualized environment to test Android applications that rely on camera functionality.
- **Safety Monitoring:** Facilitates PPE compliance monitoring and enhances workplace safety in high-risk industries through virtualized camera feeds.

Explain briefly the design principles identified and used. Why it is suitable.

3.4 Scope of Proposed System

The proposed camera virtualization system is designed to provide an efficient and scalable solution for integrating virtualized camera functionality in virtual environments. The scope of the system is defined by its ability to virtualize camera feeds, enabling seamless communication between the Linux host and Android VM using shared memory protocols such as VirtIO.

The system is specifically focused on low-latency real-time video streaming, making it ideal for applications such as AI training, software testing, and safety monitoring. However, the scope is bounded by the following limitations: it does not support advanced camera features like HDR or multi-sensor fusion, nor does it address hardware-specific configurations beyond the standard Linux and Android frameworks. Additionally, the system relies on a Linux-based host for managing the physical camera and assumes the availability of sufficient computational resources, such as 64GB RAM and 400GB disk space, to support virtualization and source code builds.

Overall, the system provides a robust foundation for camera virtualization while remaining extensible for future enhancements to address advanced use cases and broader functionalities.

Chapter 4

SYSTEM DESIGN

This chapter provides details about the system's architecture and the implementation of its major components. The following sections describe the complete operation of the proposed camera virtualization system, including the host and guest system interactions, as well as the communication mechanisms that enable seamless integration of camera functionalities within the virtualized environment. The architecture emphasizes low-latency frame transfer, real-time processing, and resource efficiency through the use of shared memory protocols like VirtIO.

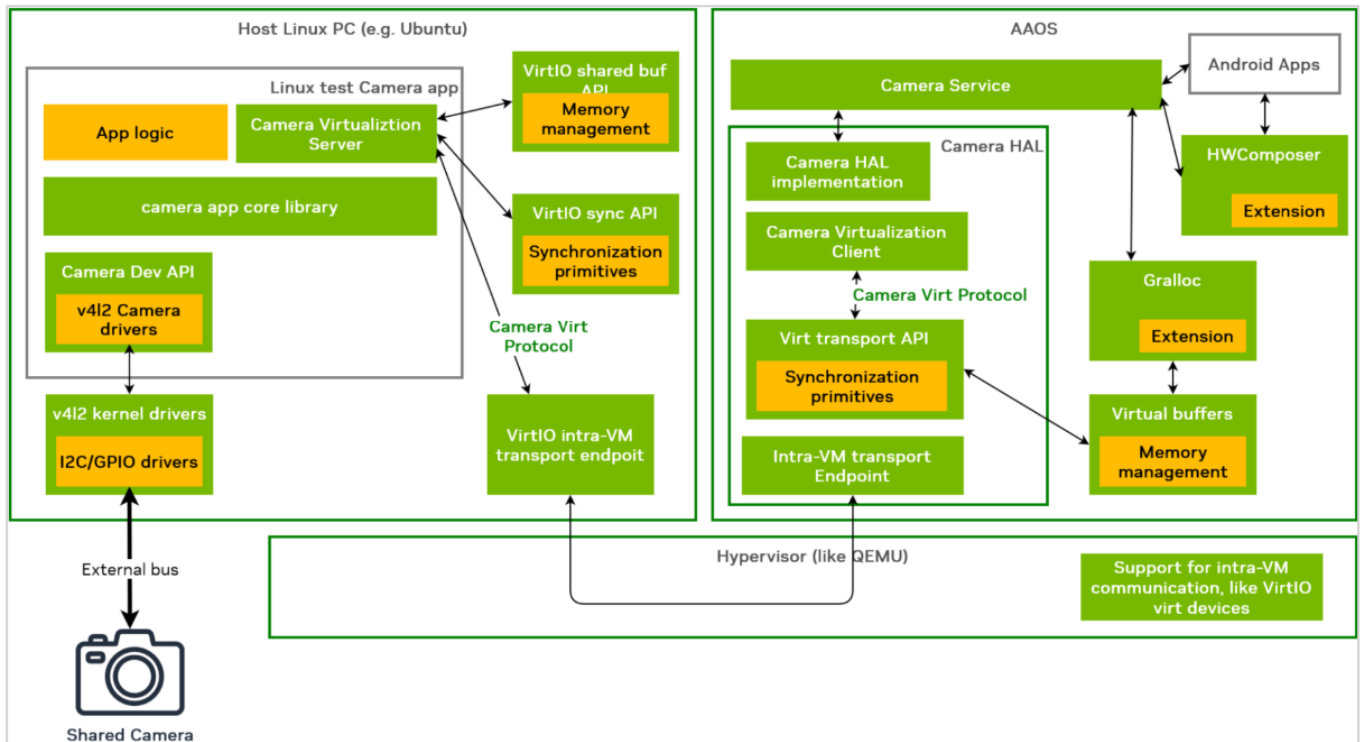


Figure 4.1: Architecture for Virtualized Camera Access in an Automotive Infotainment System

4.1 Architecture of the system (explanation)

Algorithm 1 outlines the initialization, frame capture, and request-handling processes for the host Linux system managing the physical camera and sharing frames with the guest VM.

Algorithm 1 Host Linux (Camera Virtualization Server)

```

1: Initialization:
2: Initialize camera hardware and drivers (v4l2)
3: Configure camera settings (resolution, frame rate, etc.)
4: Allocate memory buffers for captured frames
5: while Camera service is active do
6:   Frame Capture Loop:
7:   Capture a frame from the camera using v4l2 driver
8:   Process the frame (e.g., stabilization, noise reduction)
9:   Encode the frame into a suitable format (e.g., NV12, YUV420)
10:  Wait for a request from the guest VM
11:  if Request received from the guest VM then
12:    Copy the frame data to the shared memory buffer
13:    Notify the guest VM that the frame is ready
14:  end if
15: end while
16: Request Handling:
17: while Request from guest VM exists do
18:   if New frame is available then
19:    Send the frame data to the guest VM using VirtIO transport
20:   end if
21: end while

```

Algorithm 2 details the VirtIO-based communication mechanism for low-latency frame transfer between the host and guest virtual machines.

Algorithm 2 Frame Transfer (VirtIO Transport)

```

1: Initialization:
2: Establish connection between host and guest VM using VirtIO transport
3: Allocate memory buffers for frame transfer
4: Frame Transmission:
5: while Frame data is available from host VM do
6:     Copy frame data from host VM to guest VM memory
7:     Notify guest VM that the frame has been received
8: end while
9: Request Handling:
10: while Requests exist from guest VM do
11:     Notify host VM that a frame is requested
12:     Wait for host VM to provide the requested frame
13: end while

```

Algorithm 3 describes the initialization, frame request, and delivery processes of the Android Camera Service in the guest VM.

Algorithm 3 Guest Android (Camera Service)

```

1: Initialization:
2: Initialize the Camera Service
3: Establish connection with the host VM using VirtIO transport
4: Frame Request Loop:
5: while Camera service is active do
6:     Request a frame from the host VM using VirtIO transport
7:     Wait for frame data to be received
8:     Copy received frame data into appropriate memory buffers
9:     Frame Processing and Delivery:
10:    Process the frame (e.g., color space conversion, scaling)
11:    Deliver the processed frame to the requesting Android application
12: end while

```

4.2 Sequence diagram

.

Capture Frames using ffmpeg in Linux through v4l2 Camera

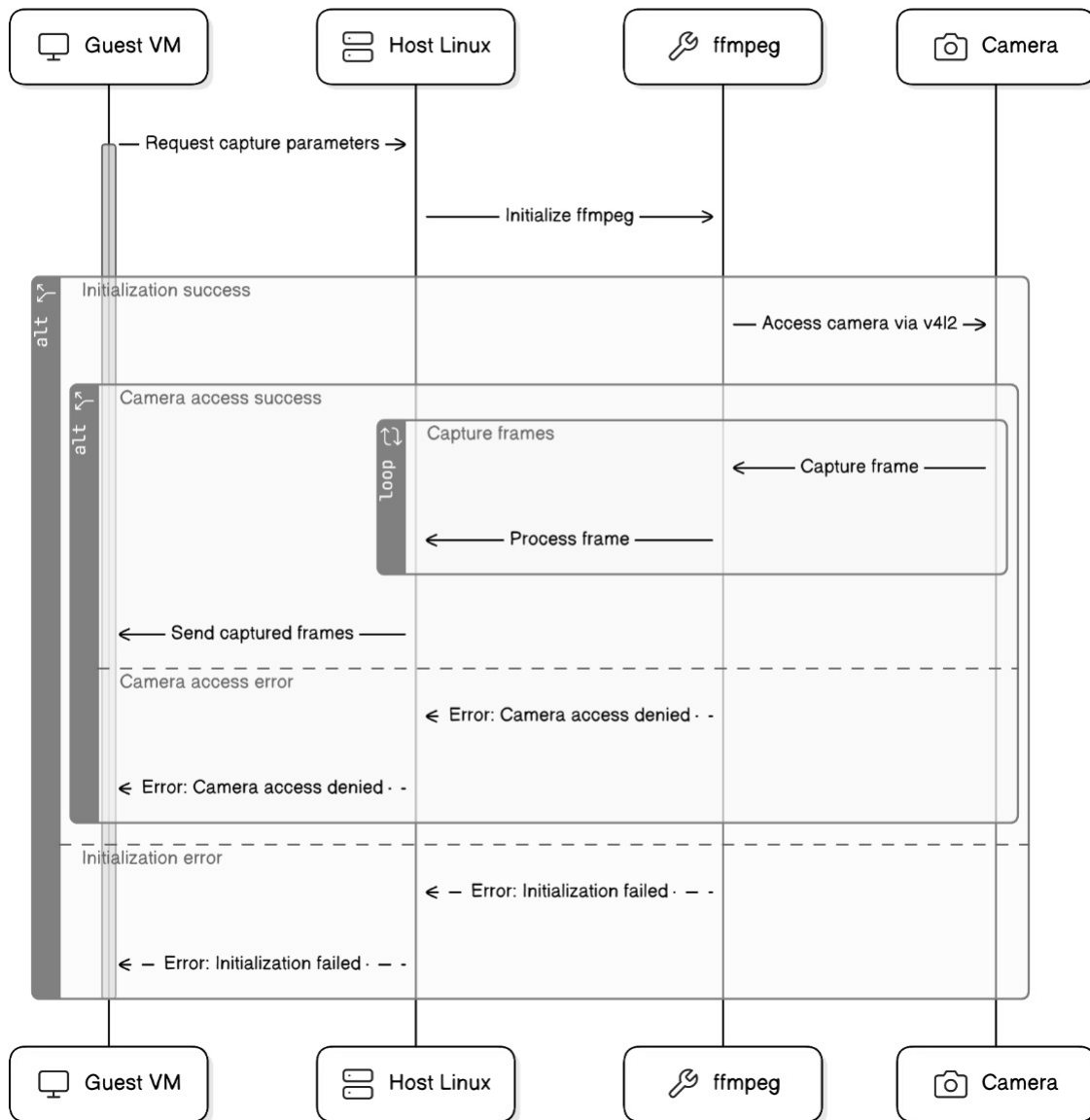


Figure 4.2: sequence diagram for capturing frames in linux through v4l2 camera

Frame Delivery Process

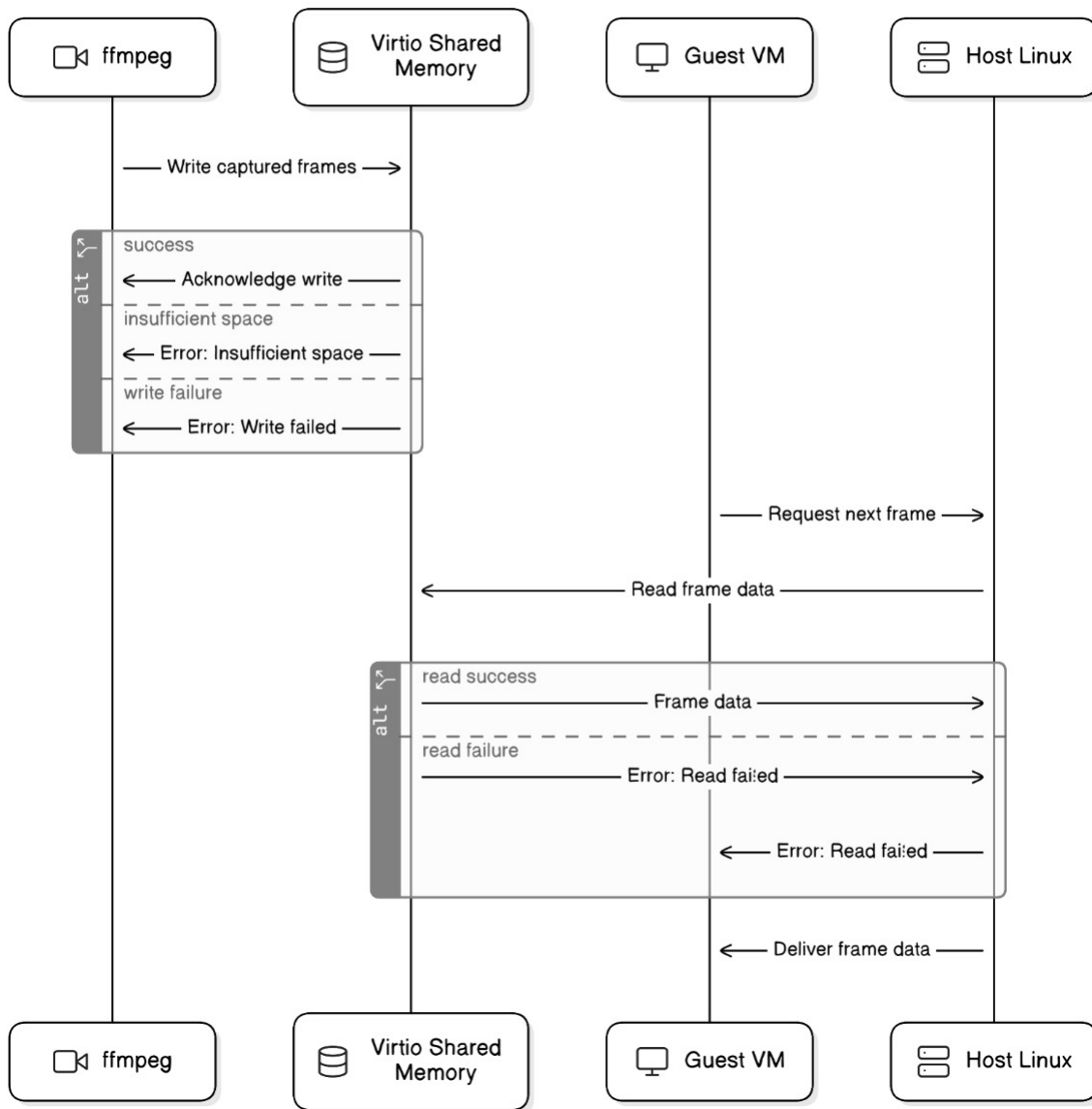


Figure 4.3: sequence diagram for frame delivery process

Frame Processing in Android VM

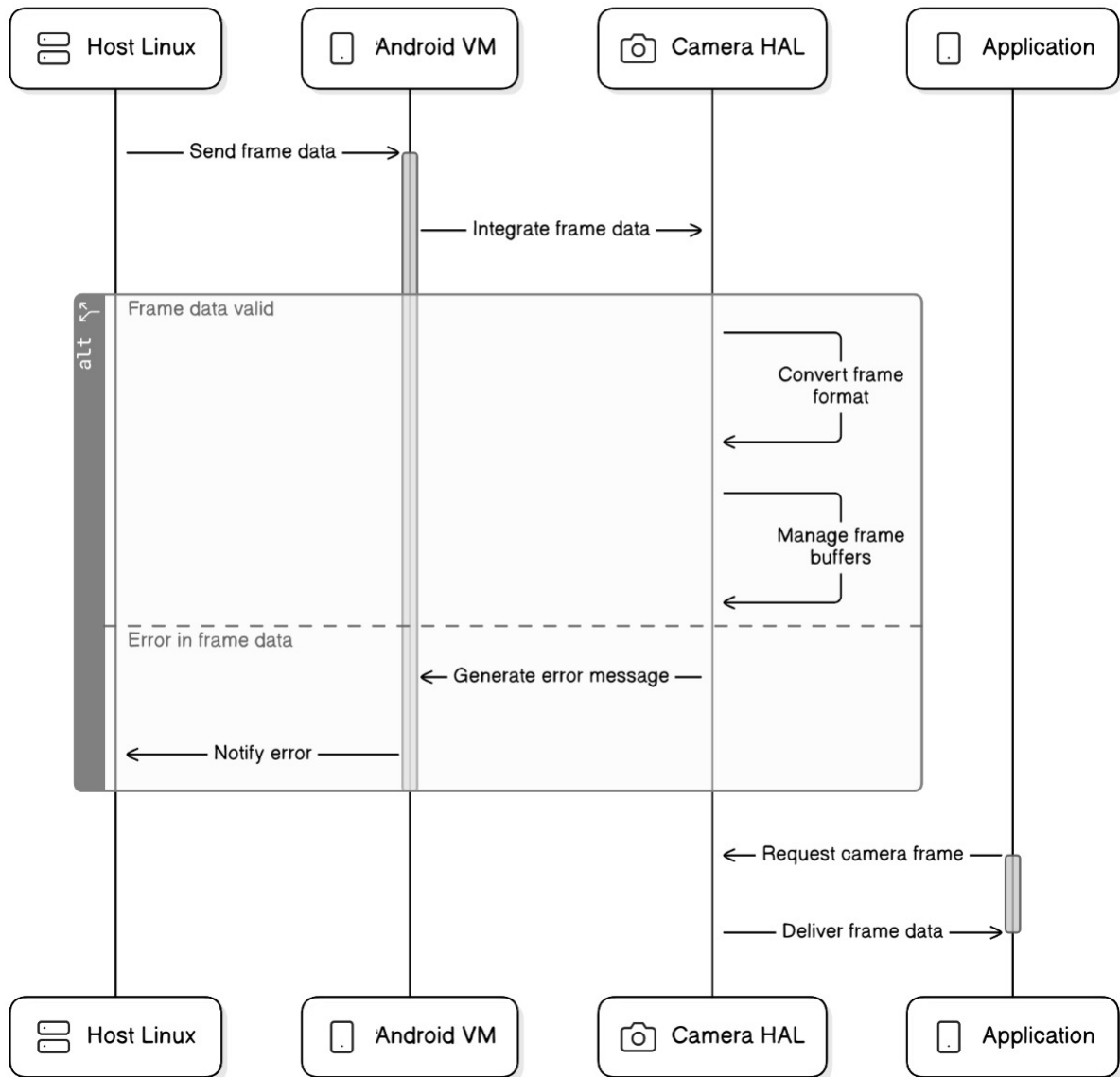


Figure 4.4: Architecture for Virtualized Camera Access in an Automotive Infotainment System

Chapter 5

IMPLEMENTATION

This chapter provides a detailed description of the implementation of the proposed system, including the methodology, modular design, and key algorithms used.

5.1 Proposed Methodology

This section details the step-by-step methodology for implementing the proposed system. The process involves capturing video frames in the Linux environment, transferring these frames to the Android VM using various methods, and ensuring seamless communication between the host and guest systems.

5.1.1 Flowchart Description

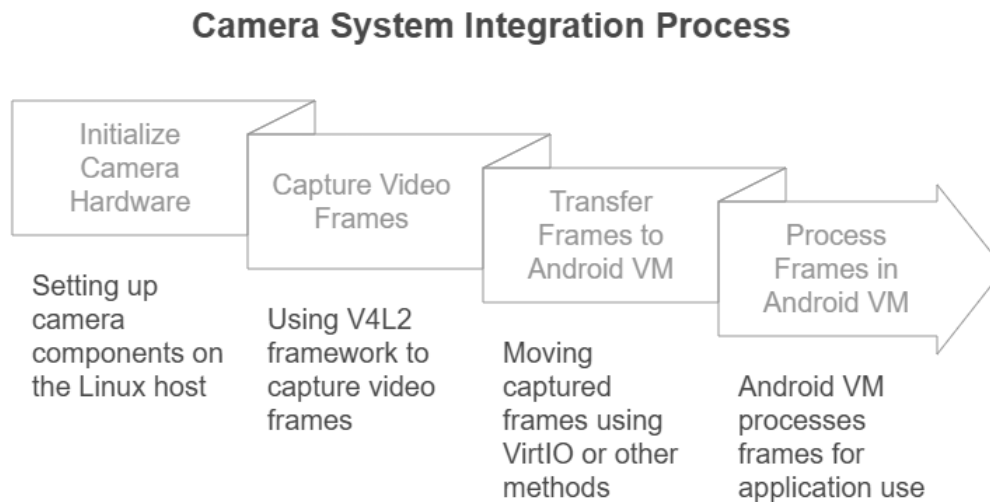


Figure 5.1: flowchart

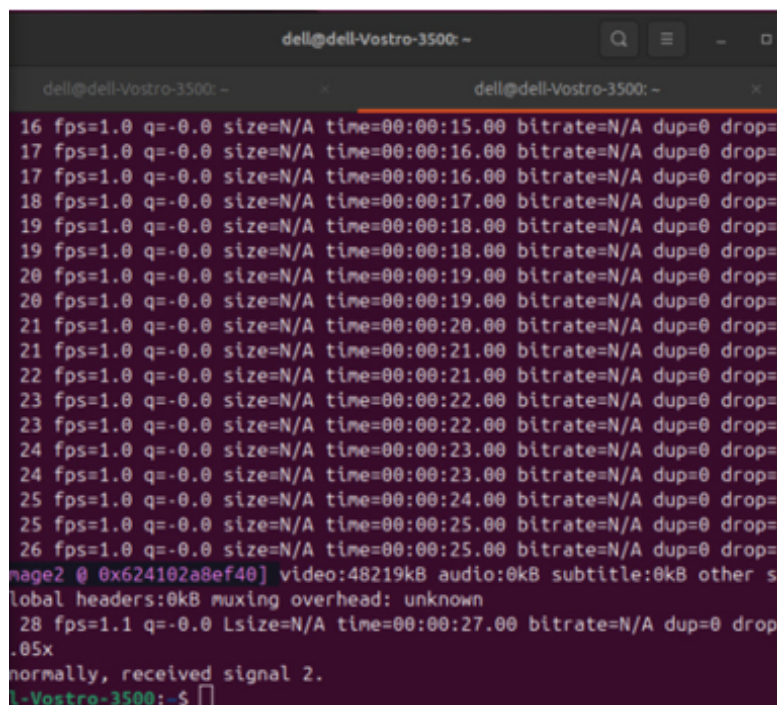
The overall methodology can be represented in a flowchart. Below is the explanation for each stage:

- **Step 1:** Initialize and configure the camera hardware in the Linux host.
- **Step 2:** Capture video frames using the V4L2 framework.

- **Step 3:** Transfer captured frames to the Android VM using VirtIO, shared folders, or a network bridge.
- **Step 4:** On the Android VM, process the frames and deliver them to camera-based applications.

This methodology ensures efficient and low-latency communication between the host and guest systems, enabling seamless camera virtualization and integration.

5.1.2 Capture Video Frames in Linux



```

dell@dell-Vostro-3500: ~
dell@dell-Vostro-3500: ~
16 fps=1.0 q=-0.0 size=N/A time=00:00:15.00 bitrate=N/A dup=0 drop=
17 fps=1.0 q=-0.0 size=N/A time=00:00:16.00 bitrate=N/A dup=0 drop=
17 fps=1.0 q=-0.0 size=N/A time=00:00:16.00 bitrate=N/A dup=0 drop=
18 fps=1.0 q=-0.0 size=N/A time=00:00:17.00 bitrate=N/A dup=0 drop=
19 fps=1.0 q=-0.0 size=N/A time=00:00:18.00 bitrate=N/A dup=0 drop=
19 fps=1.0 q=-0.0 size=N/A time=00:00:18.00 bitrate=N/A dup=0 drop=
20 fps=1.0 q=-0.0 size=N/A time=00:00:19.00 bitrate=N/A dup=0 drop=
20 fps=1.0 q=-0.0 size=N/A time=00:00:19.00 bitrate=N/A dup=0 drop=
21 fps=1.0 q=-0.0 size=N/A time=00:00:20.00 bitrate=N/A dup=0 drop=
21 fps=1.0 q=-0.0 size=N/A time=00:00:21.00 bitrate=N/A dup=0 drop=
22 fps=1.0 q=-0.0 size=N/A time=00:00:21.00 bitrate=N/A dup=0 drop=
23 fps=1.0 q=-0.0 size=N/A time=00:00:22.00 bitrate=N/A dup=0 drop=
23 fps=1.0 q=-0.0 size=N/A time=00:00:22.00 bitrate=N/A dup=0 drop=
24 fps=1.0 q=-0.0 size=N/A time=00:00:23.00 bitrate=N/A dup=0 drop=
24 fps=1.0 q=-0.0 size=N/A time=00:00:23.00 bitrate=N/A dup=0 drop=
25 fps=1.0 q=-0.0 size=N/A time=00:00:24.00 bitrate=N/A dup=0 drop=
25 fps=1.0 q=-0.0 size=N/A time=00:00:25.00 bitrate=N/A dup=0 drop=
26 fps=1.0 q=-0.0 size=N/A time=00:00:25.00 bitrate=N/A dup=0 drop=
image2 @ 0x624102a8ef40] video:48219kB audio:0kB subtitle:0kB other s
Global headers:0kB muxing overhead: unknown
28 fps=1.1 q=-0.0 size=N/A time=00:00:27.00 bitrate=N/A dup=0 drop=
.05x
normally, received signal 2.
dell@dell-Vostro-3500: ~$

```

Figure 5.2: frames captured in linux

To capture video frames, the Video4Linux2 (V4L2) framework is employed, providing a standardized API for interacting with video devices. For example, the following command is used:

```
ffmpeg -f v4l2 -i /dev/video0 -r 1 -vcodec png -b:v 2M -f image2
/home/dell/video_frames/frame_%03d.png
```

- `/dev/video0` specifies the video device.
- `-r 1` captures one frame per second.
- `-vcodec png` encodes the frame in PNG format.

- /home/dell/video_frames/frame

This process ensures frames are extracted and stored in a predefined directory for further sharing.

5.1.3 Share Frames to Android VM

Sharing captured frames with the Android VM can be achieved using VirtIO or other methods:

- **VirtIO Shared Memory:** VirtIO provides low-latency and high-throughput communication by sharing memory between the Linux host and Android VM. However, issues like missing IVSHMEM or VirtIO-shm kernel module support can complicate DMA-based transfers.
- **Shared Folder:** A shared folder can be mounted using the following QEMU configuration:

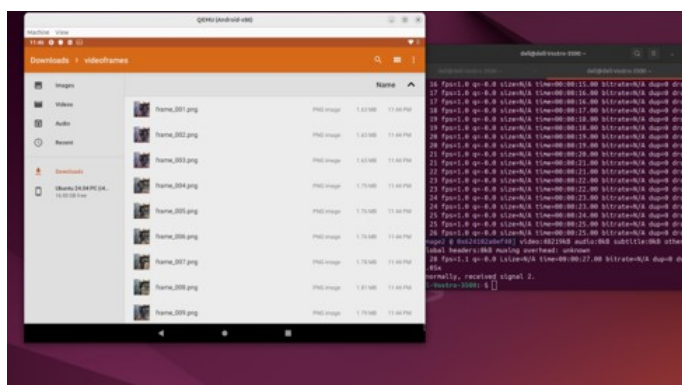


Figure 5.3: frames stored in shared folder

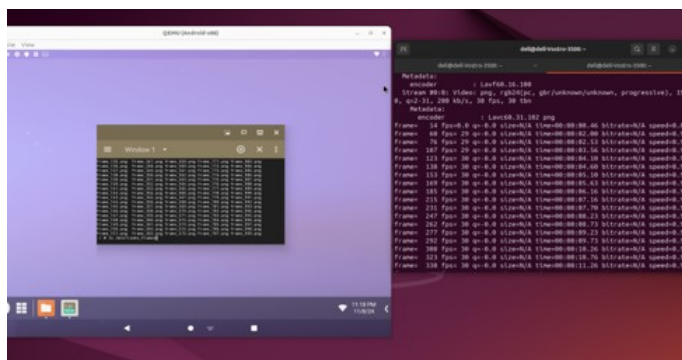


Figure 5.4: Shared folder being accessed in android VM

```

qemu-system-x86_64 \
-enable-kvm \
-m 2048 \
-drive file=qemu.img,format=qcow2,if=virtio \
-virtfs local,path=/tmp/video_frames,mount_tag=video_frames,security_m
-device virtio-9p-pci,fsdev=video_frames,mount_tag=video_frames \
-name "Android-x86"

```

This method uses VirtIO 9P to allow the guest OS to access the folder and fetch frames.

- **Network Bridge (UDP Stream):** To transfer frames over the network, a bridge interface is used to link host and guest network interfaces. Frames are streamed using FFmpeg:

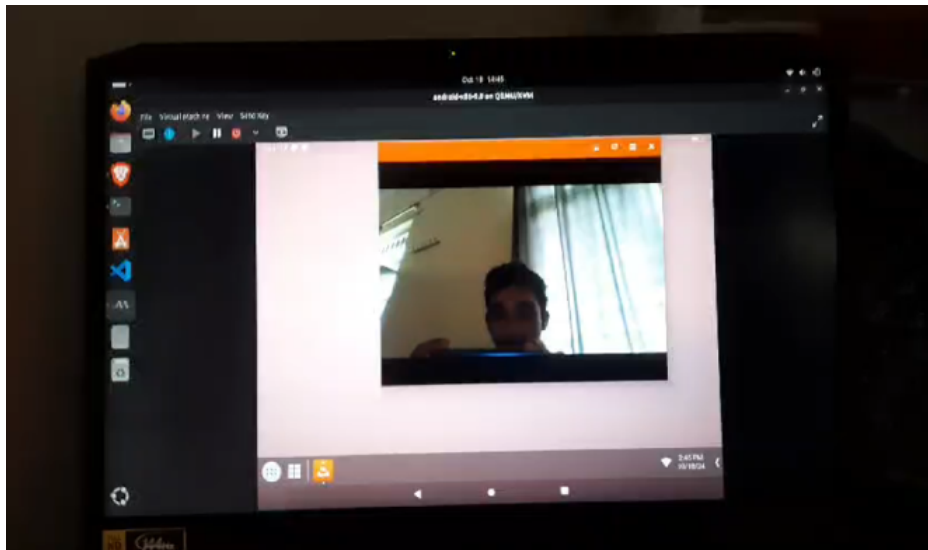


Figure 5.5: UDP stream on Android-X86 using VLC media player

```

ffmpeg -f v4l2 -i /dev/video0 -c:v libx264 -f mpegts udp://<Guest_IP>:

```

This method captures and encodes video frames for real-time streaming to the Android VM.

5.1.4 Android VM Integration

On the Android VM, the camera service initializes and requests frames from the Linux host. The received frames are processed and delivered to Android applications for use.

5.1.5 Modification of Camera HAL

The final step involves modifying the Camera Hardware Abstraction Layer (HAL) in the Android source code to support virtualized camera functionality. The Camera HAL is a critical component that bridges the Android Framework and the underlying camera hardware or virtual camera interface.

1. Understand the Existing HAL Architecture

- Analyze the default Camera HAL implementation provided in the Android source code.
- Locate the source files under the directory:

```
<android_source_root>/hardware/interfaces/camera/
```

- Study the interface methods defined in `ICameraProvider` and `ICameraDevice`.

2. Modify the HAL to Support Virtual Camera

- Add a new virtual camera implementation in the HAL to fetch frames from shared memory or network streams.
- Update the `getCameraInfo()` and `openCamera()` methods to handle the virtual camera.
- Implement frame handling logic in the HAL to fetch frames from shared memory, such as:

```
memcpy(frame_buffer, shared_memory_address, frame_size);
```

3. Compile and Deploy the Modified HAL

- Rebuild the Android source code to include the modified Camera HAL.
- Flash the updated system image onto the Android VM.

4. Test the Virtual Camera Integration

- Use Android's **Camera2** API to validate that camera-based applications can access the virtual camera seamlessly.
- Verify frame latency and consistency using sample applications.

By modifying the Camera HAL, the Android VM can seamlessly integrate the virtualized camera, enabling real-time access to frames shared by the Linux host. This ensures compatibility with Android applications and supports advanced use cases.

Chapter 6

RESULTS DISCUSSIONS

This section presents the results achieved from implementing the proposed system up to the frame transfer stage, along with a discussion of the challenges encountered and alternative methods utilized. Snapshots of outputs are included to provide clarity.

6.1 Results

The following key results were achieved during the implementation:

1. **Frame Capture on Host Linux:** Video frames were successfully captured using the V4L2 framework. The frames were stored in a predefined directory on the host system with high fidelity and minimal processing delays. Figure 6.1 shows an example of a captured frame.

```
dell@dell-Vostro-3500: ~
dell@dell-Vostro-3500: ~
16 fps=1.0 q=-0.0 size=N/A time=00:00:15.00 bitrate=N/A dup=0 drop=
17 fps=1.0 q=-0.0 size=N/A time=00:00:16.00 bitrate=N/A dup=0 drop=
17 fps=1.0 q=-0.0 size=N/A time=00:00:16.00 bitrate=N/A dup=0 drop=
18 fps=1.0 q=-0.0 size=N/A time=00:00:17.00 bitrate=N/A dup=0 drop=
19 fps=1.0 q=-0.0 size=N/A time=00:00:18.00 bitrate=N/A dup=0 drop=
19 fps=1.0 q=-0.0 size=N/A time=00:00:18.00 bitrate=N/A dup=0 drop=
20 fps=1.0 q=-0.0 size=N/A time=00:00:19.00 bitrate=N/A dup=0 drop=
20 fps=1.0 q=-0.0 size=N/A time=00:00:19.00 bitrate=N/A dup=0 drop=
21 fps=1.0 q=-0.0 size=N/A time=00:00:20.00 bitrate=N/A dup=0 drop=
21 fps=1.0 q=-0.0 size=N/A time=00:00:21.00 bitrate=N/A dup=0 drop=
22 fps=1.0 q=-0.0 size=N/A time=00:00:21.00 bitrate=N/A dup=0 drop=
22 fps=1.0 q=-0.0 size=N/A time=00:00:21.00 bitrate=N/A dup=0 drop=
23 fps=1.0 q=-0.0 size=N/A time=00:00:22.00 bitrate=N/A dup=0 drop=
23 fps=1.0 q=-0.0 size=N/A time=00:00:22.00 bitrate=N/A dup=0 drop=
24 fps=1.0 q=-0.0 size=N/A time=00:00:23.00 bitrate=N/A dup=0 drop=
24 fps=1.0 q=-0.0 size=N/A time=00:00:23.00 bitrate=N/A dup=0 drop=
25 fps=1.0 q=-0.0 size=N/A time=00:00:24.00 bitrate=N/A dup=0 drop=
25 fps=1.0 q=-0.0 size=N/A time=00:00:25.00 bitrate=N/A dup=0 drop=
26 fps=1.0 q=-0.0 size=N/A time=00:00:25.00 bitrate=N/A dup=0 drop=
frame2 @ 0x624102a8ef40] video:48219kB audio:0kB subtitle:0kB other s
lobal headers:0kB muxing overhead: unknown
28 fps=1.1 q=-0.0 Lsize=N/A time=00:00:27.00 bitrate=N/A dup=0 drop
.05x
normally, received signal 2.
l-Vostro-3500:~$
```

Figure 6.1: frames captured in linux using ffmpeg

Frame Sharin

2. to Android VM: The captured frames were transferred to the Android VM using alternative methods, as shared memory-based transfers could not be implemented due to missing kernel module support (details in discussion). The two methods successfully implemented are:

- **Shared Folder:** A shared folder `/video_frames` was mounted on the guest Android VM using VirtIO-9P, enabling the Android VM to access the captured frames. Figure 6.2 shows the shared folder setup.

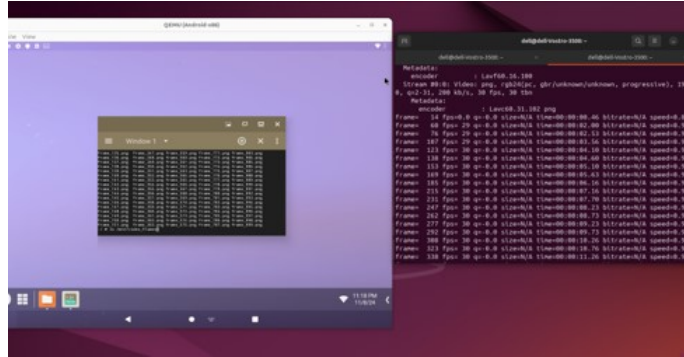


Figure 6.2: shared folder can be accessed

- **UDP Stream:** A network bridge was used to establish communication between the host and the guest. Using FFmpeg, frames were streamed in real-time over a UDP connection, demonstrating low latency and efficient transmission.

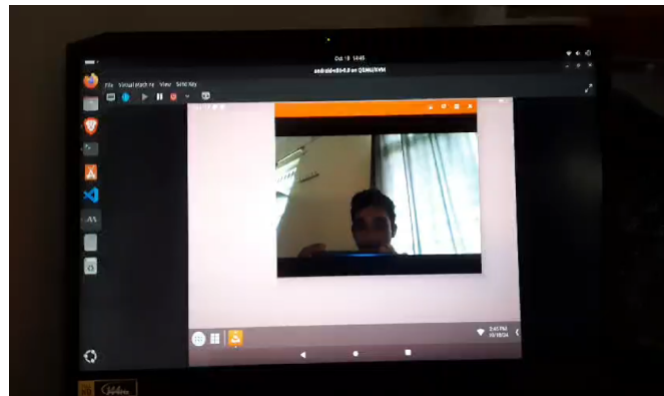


Figure 6.3: UDP stream on Android-X86 using VLC media player

6.2 Discussion and Challenges

During the implementation, certain challenges were encountered, particularly regarding shared memory support:

- **Issues with Shared Memory:** The QEMU setup lacked the necessary IVSHMEM or VirtIO-shm support. Kernel module checks (`lsmod | grep virtio` and `lsmod | grep ivshmem`) confirmed that the relevant drivers were not loaded on the system, making it impossible to utilize Direct Memory Access (DMA) for frame sharing. Figure ?? provides a snapshot of the module check results.
- **Shared Folder Approach:** As an alternative, a shared folder was mounted between the host and the guest using VirtIO-9P. This approach enabled efficient file transfers, but it requires manual synchronization of frames and introduces minor latency due to file system overhead.
- **UDP Stream Approach:** A network bridge was configured to facilitate real-time frame transfer using FFmpeg. This method provided lower latency compared to the shared folder approach, making it suitable for video streaming applications. However, it requires proper network configuration and encoding overhead.

6.3 Summary of Results

The implemented system successfully captures and transfers video frames between the Linux host and the Android VM. While shared memory methods could not be implemented due to missing kernel support, alternate methods such as shared folders and UDP streams provided effective solutions. The results demonstrate:

- Reliable frame capture on the Linux host using V4L2.
- Successful frame sharing to the Android VM via shared folders and UDP streams.
- Low latency and high efficiency using the UDP streaming approach, suitable for real-time applications.

The next step involves modifying the Camera HAL in the Android VM to integrate the transferred frames with Android applications.

Chapter 7

CONCLUSIONS AND FUTURE SCOPE

The project successfully implemented a system to virtualize camera frames and transfer them from a Linux host to an Android Virtual Machine (VM). Key features achieved include the reliable capture of video frames using the V4L2 framework and their seamless transfer to the Android VM using alternative methods such as VirtIO-9P shared folders and UDP streaming. Despite the limitations in shared memory support, these methods demonstrated efficient communication between the host and guest systems. The implemented solution lays the groundwork for future camera virtualization projects and serves as a proof-of-concept for integrating physical hardware with virtualized environments.

For future work, the project can be enhanced by incorporating Direct Memory Access (DMA)-based frame sharing through IVSHMEM or VirtIO-shm, provided kernel support is added. This would significantly reduce latency and improve performance. Furthermore, the modification of the Camera HAL in the Android VM remains an essential next step to enable seamless integration of transferred frames with Android applications. Future enhancements could also include real-time encoding optimizations, multi-camera support, and extending the system for use with additional guest operating systems. The ultimate goal is to create a fully integrated, high-performance virtual camera system for diverse virtualization environments.

REFERENCES

- [1] “VirtIO: A Low-Latency Solution for Virtualized Environments”. *International Journal of Virtualization and Cloud Computing*, 15(5):301–310, June 2023.
- [2] “Leveraging VirtIO for Efficient Resource Sharing in Virtualized Systems”. *Computer Science Journal*, 10(4):120–130, April 2022.
- [3] “Android 14 CDD - Camera Requirements”. *Android Compatibility Definition Document (CDD)*, August 2023.
- [4] “Android Camera HAL CTS Testing”. *Android Compatibility Test Suite (CTS)*, August 2023.
- [5] “Camera Provider AIDL Target Test”. *Android Camera Provider AIDL Interface*, September 2023.
- [6] “FFmpeg: Efficient Video Encoding and Streaming in Virtualized Environments”. *Journal of Multimedia Systems*, 5(8):1023–1034, July 2021.
- [7] “Optimizing FFmpeg for Real-Time Video Streaming in Virtualized Systems”. *Video Processing Journal*, pages 233–240, March 2022.
- [8] “Virtualization Techniques in Cloud-Based Video Streaming”. *Cloud Computing Review*, pages 56–65, January 2020.
- [9] “Optimizing Video Streaming in Cloud-Based Surveillance Using Virtualization”. *Cloud Surveillance Systems*, pages 98–110, April 2021.

Chapter 8

Plagiarism Report

Attach your plagiarism report of this mini report here. Make sure that plagiarism is below 20 %.