UNIVERSITY OF COLORADO, BOULDER

Extended Lab – Time Lapse Project

ECEN 5623

Rahul Yamasani,Chinmay Shah

8/4/2016

NOTE:

Both of us are working as research assistants at UCB and each team has an ongoing experiment which is nearing to its completion and required our presence.

Even though we completed our implementation, we were lagging on the analysis part and required extra time to complete it. As a result, we are submitting post deadline.

We request you to consider our situation and permit us for the late submission.

Table of Contents

Introduction:

This project is to implement *"**Time Lapse**"* video by capturing frames at different rates and converting it into video using ffmpeg tool. The main aim of this project is to design and implement capturing of images using C200 Logitech webcam with optimum accuracy and minimum latency. Further the project deals with analysis and design of real time system with assignment of deadlines to different tasks involving capture of frames, compression, socket packet transmission and image processing features such as centroid detection, histogram and background detection.

This implementation is able to obtain a time lapse video with minimum latency (error of –ve one seconds/one frame) and provides detail analysis of average jitter, WCET and % missed deadlines for fps 1 Hz, 5Hz, 10 Hz and background subtraction.

Major Requirements:

1. Image Acquisition at a Frequency of 1 Hz using V4l2 libraries for (1800 frames)
2. Verifying image acquisition for a period of 30 minutes using time stamps and EXTERNAL WALL CLOCK.
3. Applying the JPEG Compression Algorithm to compress each and every image captured (1800 frames)
4. Providing infinite frame capture by sending the image over WLAN from Host system to target system using Socket Programming (1800 frames)
5. Performing Image Processing features for 1800 frames being captured at a frequency of 1 Hz.
    a. Image Histogram
    b. Detection of object and computation of centroid.
6. Detecting the changes that exceed more than 1% of the total pixel value by performing background subtraction at a frame rate of 10 Hz.
7. Creating MPEG encoded **Time Lapse** video stream using **ffmpeg** (at 30 fps), for the 1800 frames acquired.
8. Computation and analysis of jitter and latency of image acquisition at a frame rate of 1Hz
9. Computation and analysis of jitter and latency including the following services
    a. Compression
    b. Sending the images acquired through WLAN
    c. Spectral histogram Analysis
    d. Detection of object and computation of centroid
10. Computation and analysis of jitter and Latency of image Acquisition by running at higher frame rates namely 10 Hz, 5 Hz.

Functional Description:

Frame Capture:

The primary aim of the project is to capture images from webcam at a frame rate of 1 Hz. Logitech C200 [1] was used as the camera interface. As per the specifications, Logitech C200 operates at 30 FPS [2] supporting the maximum resolution of 640*480. The Logitech C200 requires installation of UVC drivers on the Host system. Webcam interface with the host system requires an API interface that interacts with UVC drivers and was facilitated by V4l2 libraries. This library contains ioctl function which verifies and sets various parameters that enable frame capture at 30 FPS with resolution of 640*480.

The image that is being captured has a frame is being stored in PPM format, with a header in including the following:

P6
#<name –a >
#Frame<99999>=HH:MM:SS P
VRES HRES
255

Fig1.1 (i)Format of header appended to each. ppm files

```
1 P6
2 #Linuxchinmay-Inspiron-75483.19.0-65-generic#73~14.04.1-Ubuntu SMP Wed Jun 29 21:05:22 UTC 2016
3 #Frame0000001=12:17:17 A
4 640 480
5 255
6 NULNULNULæyNæyNÓ[Ó[Ò[Ò[Ñ€YÑ€YÏ‰xÑ€YÑ€YÒ[Ò‰[Ò‰[õ€[õ€[õ€[õ€[õ€[õ€[õ€YÕ€YÕ€YÕ€YÕ€YÖZÖZÖZô,\ô,\ô,ZÓYÓ
7 H
8 FFFST
```

Fig1.1 (ii)Snapshot of header appended to each. ppm files

The implementation is carried as follows:

As explained in the major requirements, the implementation involves image capture in PPM format followed by compression. The compressed images are sent to target using TCP sockets [3]. Image processing features such as detection of object & computation of its centroid, spectral histogram analysis have been incorporated with above services. In addition to this, the implementation supports a higher rate mode that captures the images at frequencies higher than 1 Hz. This mode also implements background subtraction to capture the foreground changes.

Description of Threads:

The main function forks a Sequencer thread that has highest priority. This thread contains functions that initialize the webcam, enable VIDEO_STREAM_ON, read the frames and finally stops capturing that un initializes the device. In addition to this Sequencer thread forks the other threads with the following functions

THREAD 1: Sequencer thread
THREAD 2: process the YUV images captured
THREAD 3: compress the images captured
THREAD 4: send the compressed images from host to target via socket programming
THREAD 5: Histogram of the image being captured
THREAD 6: Detection of object and computation of centroid

Sequencer thread:

Sequencer thread is parent and is assigned with the highest priority. The sequencer thread creates six buffers which store the data from the incoming streamed images. The immediate step that has to be carried is to create a PPM image out of the buffer captured.

Process Image thread:

Logitech C200 captures the image in YUV format. To store the images in PPM format they have to be converted to RGB format first. This process of capturing input buffer to PPM format with appropriate headers is done by THREAD 2.

As the frames have been buffered (as six in present implementation) we may not obtain and display the same instance of time when this frame had been captured. Thus, the timestamp is acquired from these buffers. The UVC drivers allow this by setting the ***V4L2_BUF_FLAG_TIMESTAMP_MONOTONIC*** flag as shown in Fig1.2. This time is embedded in header as shown in Fig1.1.

```
715
716             buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
717             buf.memory = V4L2_MEMORY_MMAP;
718             buf.flags = V4L2_BUF_FLAG_TIMESTAMP_MONOTONIC;//to allow capturing of time from buffers timestamp
```

Fig 1.2 Settings to obtain timestamp from buffers for a frame acquired

Once this ppm image has been created, the next process is to compress it. Further thread2 posts the semaphore that THREAD 3 is waiting for in turn compressing image to transfer using Ethernet.

Compression:

This function is implemented by THREAD 3.

One of the goals is to support infinite frame capture. To facilitate this, we have written code that transfers images from host to target using TCP sockets. But here the images being captured are in PPM format, which contains the raw data and have file sizes in the order of 900 kb. Transferring them within the specified deadlines would be a tedious task. As a solution, the images being captured are compressed first and then transferred.

As stated in the project proposal we have checked for various compression algorithms to compress the PPM images. The compression if images can be lossy or lossless. Lossless image compression is used for applications such as medical imaging, technical drawings where detailing to the highest degrees is required [4]. The prominent among lossless compression algorithm is .png compression. On using this algorithm, the compressed images are in the order of 400-430 kb. Still this consumes a lot time in file transfer. So we shifted our focus to lossy compression as data loss is permissible as long we protect image clarity.

Keeping this view in mind, we have used a JPEG compression algorithm. Using this compression algorithm, images after compression are turning out to be in the order of 50-60 kb and the results are quite promising for file transfer. The novelty in such a high compression ratio lies in the JPEG algorithm. For the present experiments ppm image (**901Kb**) was converted to .jpg (**79Kb**) as in *Fig 2*.

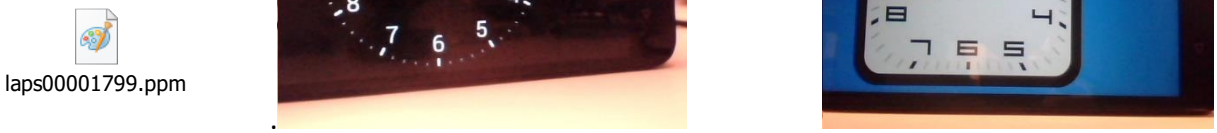


Fig 2) Images of external clock in. ppm format converted to .jpg as shown above (901Kb to 79Kb) - (Frame1799- laps00001799.ppm - → laps00001799.jpg -right most)

Why JPEG compression is effective?

The goal of any image compression algorithm is to reduce the data redundancy. To be brief, the JPEG compression algorithm converts RGB (Red, Green, Blue) components of the image into YUV (luminance, blue chrominance and red) where luminance has the information regarding brightness of the pixel while the chrominance containing information on hue. The components in YUV are typically less correlated in comparison with RGB components. In addition to this

Furthermore, human eye is more sensitive to luminance than chrominance. It means that we can neglect the larger variations in chrominance without effecting the perception of the image.
Since this transformation is invertible, we will be able to recover the *(R,G,B)* vector from the *(Y, C_b, C_r)* vector. This is important when we wish to reconstruct the image [5].

Basically, In JPEG compression algorithm, the image is divided into pixels of 8*8 blocks each and likewise unnecessary chrominance components are removed. Thus the compression not only reduces data redundancy, but also preserves image clarity.

Implementation of Image compression is as follows:

1. Read the PPM file created from the process_image thread using the function imread
2. Convert the raw data from PPM to Mat format
3. Write the generated jpg file to host system using imwrite

Transmission of Data:

This function is implemented by THREAD 4(clientSender) which is posted from THREAD 3

Transmission of data is trivial in supporting infinite frame capture. It has been mentioned that compression has significantly decreased the file size thereby increasing the probability for smooth transfers. Transferring of files is done using network programming. We have created two sockets one on the server and the other one on the client using TCP [6].
In our implementation the server is receiver of data while client being the sender. We have

1. HOST- Client (Sends the data)
2. Target- Server (Receives the data)

We have used the port number **50000** and implemented the server client system on both localhost systems and remote systems.

The following steps occur while establishing the TCP connection between two computer sockets.

1. The server initiates a socket and communication is done using the port 50000
2. The receiver side (server), has accept () invoked which waits until the client is connected.
3. Once the server starts waiting, the client initiates a socket object specifying the server name of the Target (receiver) / “localhost” using the port number 50000
4. The socket created on the client attempts to connect the server using the specified host name and port number.
5. Once the connection has been established, the client sends data
6. The server side has accept(), that receives the data being sent from the client

Centroid Detection

This service is implemented by THREAD 5

Drawing the outline of object and Computation of its centroid

1. Read the jpeg image created from the Thread process_image() using imread()
2. Convert the image into grayscale using cvtColor on opencv. This will facilitate in removing the redundant RGB data by converting into a grayscale image.
3. Blurring the grey scale image to reduce high frequency using blur() function on opencv. This facilitates the contour detection more accurate.
4. Using an edge detection algorithm would help in finding the edges on the image. We used Canny() to perform this
5. Finding the contour using findContours()
6. Getting the moments using mu(contours.size())
7. Computing the centroid of each and every image
8. Drawing the contours on the image and corresponding centroids.
9. Saving the image using imwrite() function.

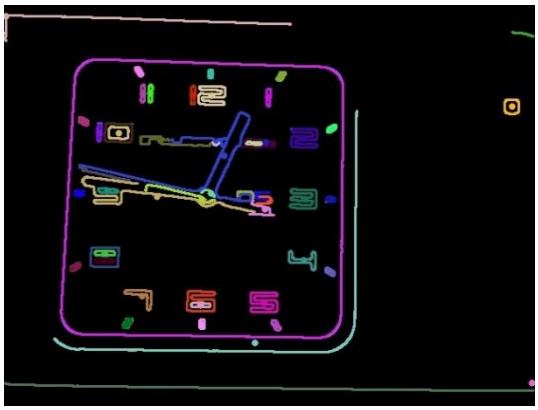


Fig 3) Centroid of closed contours for original image as shown in Fig 2. –(Frame1799- cent00001799.jpg)

Image Histogram:

This service is implemented by THREAD 6

The image histogram represents the tonal representation of an image in graphical form. In digital Image processing histogram is used image enhancement. It is also used by Google search in its google image search. In our implementation, thread compression (Thread 3) posts thread Histogram (Thread 6) every time an image is capture is complete and send through Ethernet.

Implementation of Image Histogram: [7]

1. Load the jpeg image using imread
2. Splitting the image in R, G and B components using the opencv function split
3. Calculate the Histogram of each component channel using the opencv function calcHist
4. Store the image on to disk using imwrite

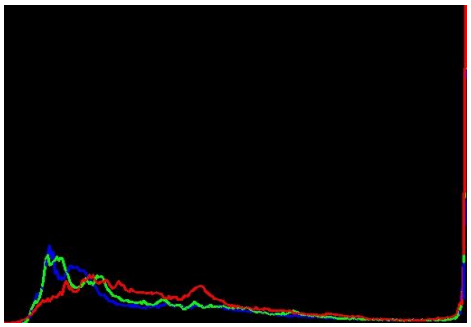


Fig4.1

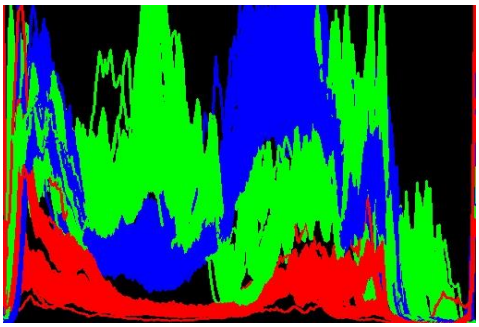


Fig4.2

Fig 4.1) Histogram(right) of original image as shown in Fig 2. – (Frame1799- hist00001799.jpg)
Fig 4.2) Intermediate step for histogram creation (of another image)

Foreground Masking (Background subtraction)

As a part of running at high rate we have implemented running a high rate foreground mask has been implemented. This part of code runs at separately by #defines **highRate** allowing acquisition at high rate.

We are using basic implementation of subtracting previous frame from present image as seen from Fig5.1). The foreground image is dilated and eroded after Thresholding for better and sharper foreground results.

We check for a change in 1% of total pixels after obtaining foreground. We store the original when this condition is true and analyze it for 1800 frames. We weren't able to complete analysis with different algorithms and test its efficiency

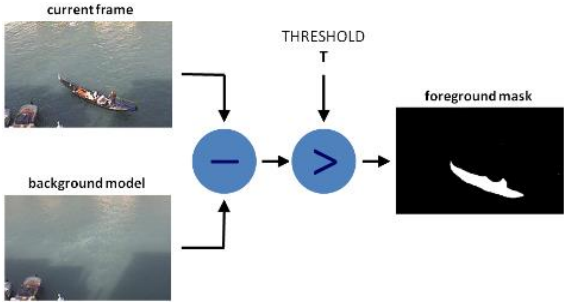
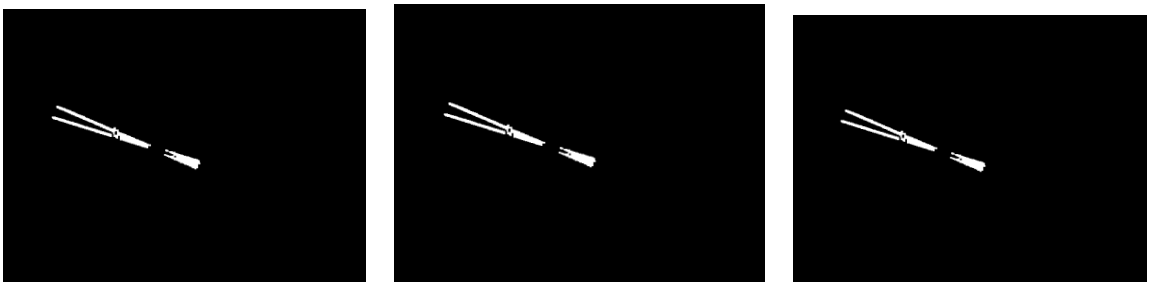


Fig 5.1) Implementation of background subtraction using Basic Difference

As seen from Fig5.2) intermediate steps to trigger acquiring of background images stored in bckgndxxxx.jpg format

Fig 5.2) Intermediate steps of transformation in between of background detection for foreground masking



(i) (ii) (iii)

Fig 5.3) Intermediate outputs of background subtraction using basic difference
(left – frame ground, middle – erode o/p and right – dilated) after foreground masking and thresholding

Flowchart of the Code:

The following flowchart describes the path of the code.

1. On start, the program enters the main function. The main function verifies the input arguments and opens appropriate webcam. It also spawns a sequencer thread.
2. The sequencer thread initializes the camera device and starts capturing the frames. In addition to this it spawns threads such as process_image, compression, sendData, Image histogram and Drawing the contour and detection of centroid.

   Initially the flow of the code was as follows

   1) Sequencer (initialize device)
   2) Read frame (captures images and nanosleep ())
   3) Process image
   4) Compression
   5)Send data // Image Histogram // Centroid and contour

   Threads Send data, Image Histogram and Centroid & contour uses the images Compression. To make this possible we have serialized these threads with the compression thread.
   The calculated latency in the process was around **12 sec**. On investigating the code written for v4l2 libraries we were able to deduce that process_image converts the YU YV format to RGBRGB and required a lot of mathematical computations. From these observations, it was concluded that execution time involved in process_image () is the root cause for the above latency.

   The issue can be solved, if the execution of process_image () is made to execute in concurrent with Read_frame (time elapsed in between frame capture) With this is idea the code flow has been modified to as follows:

   1)Sequencer (initialize device)
   2) Read_frame (captures and nanosleep ()) // Process image
   3)Compression
   4) Send data // Image Histogram // Centroid and contour

*// - parallel, numbered – concurrent

   On restructuring our implementation, the process_image () was getting executed in parallel with read_frame (). This change increased the capture efficiency thereby reducing the latency to +/- 1 sec described in the following sections.

3. ** Important workaround – While analyzing at later stage of implementation we observed a lag in time from first frame. The observation was that initial six frames always showed same time on external clock. Further we could conclude this from time on header of each ". ppm" file while acquiring timestamp from buffer.

   On further digging it was noticed the number of frames were dependent on number of buffers used with V4L2 drivers.
   Thus to resolve this lag/latency of **six** seconds and provide accurate results we are discarding initial frames equal to number of buffers used for capturing these images.

The detailed flow of our implementation is explained in detail in the following flowchart.

5. A shell script is implemented to create time-lapse videos which takes fps as input and converts. ppm or .jpg files into a .mp4 video. This is called by a system call from C implementation and allows choice of fps (image2video.h).

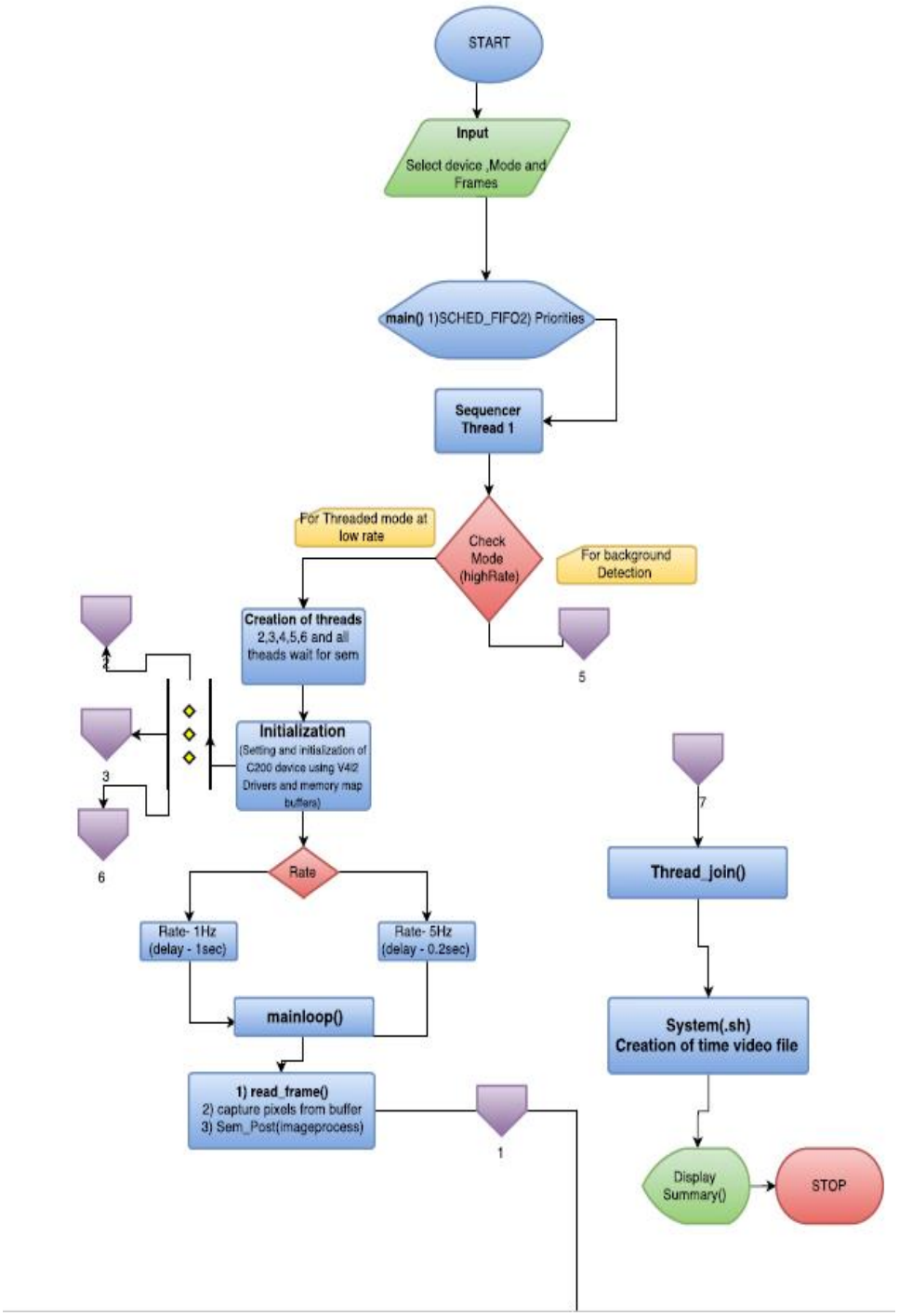


Fig 6.1 Flow chart for code design (part 1/3)

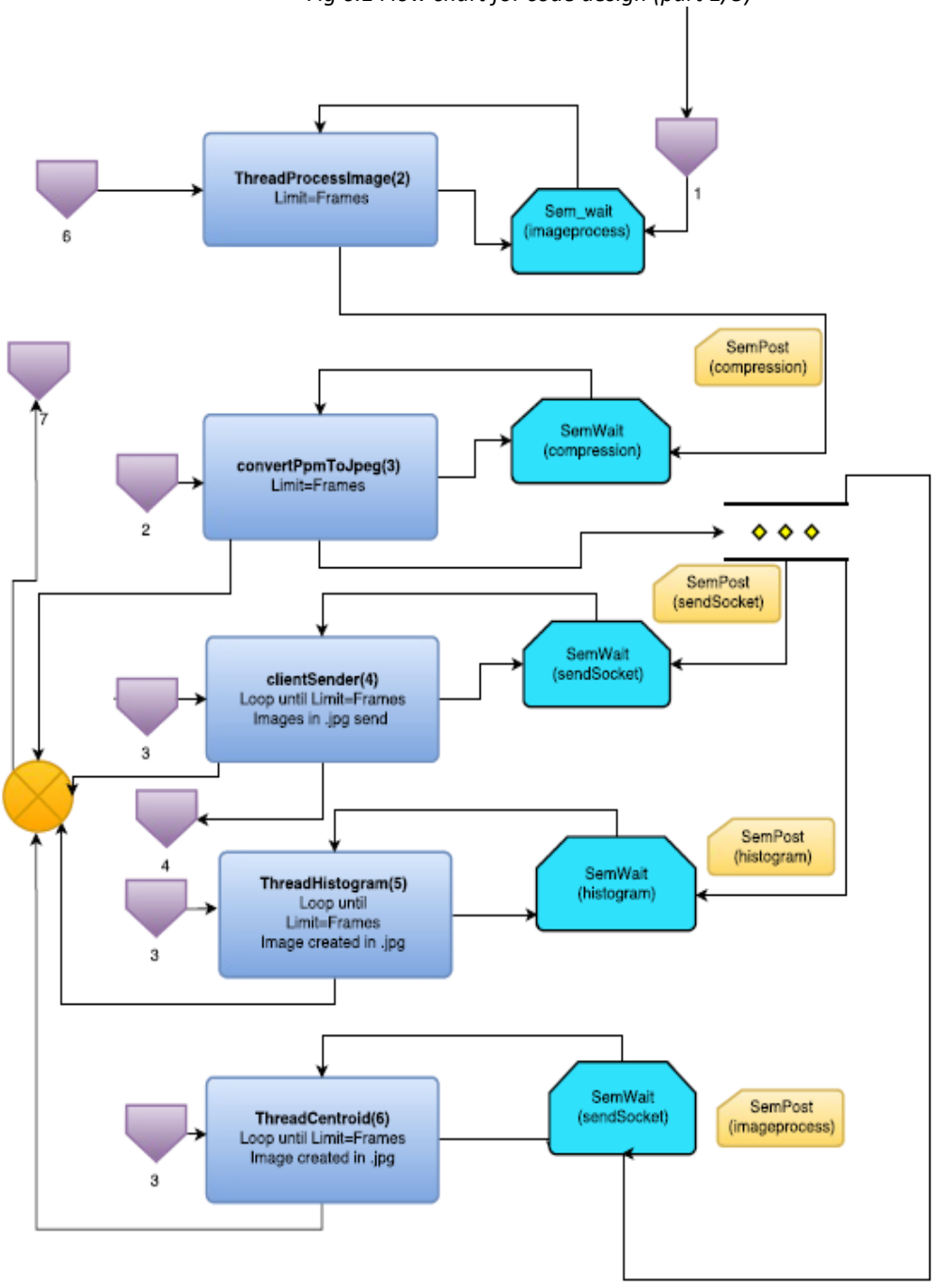


Fig 6.2 Flow chart for code design (part 2/3)

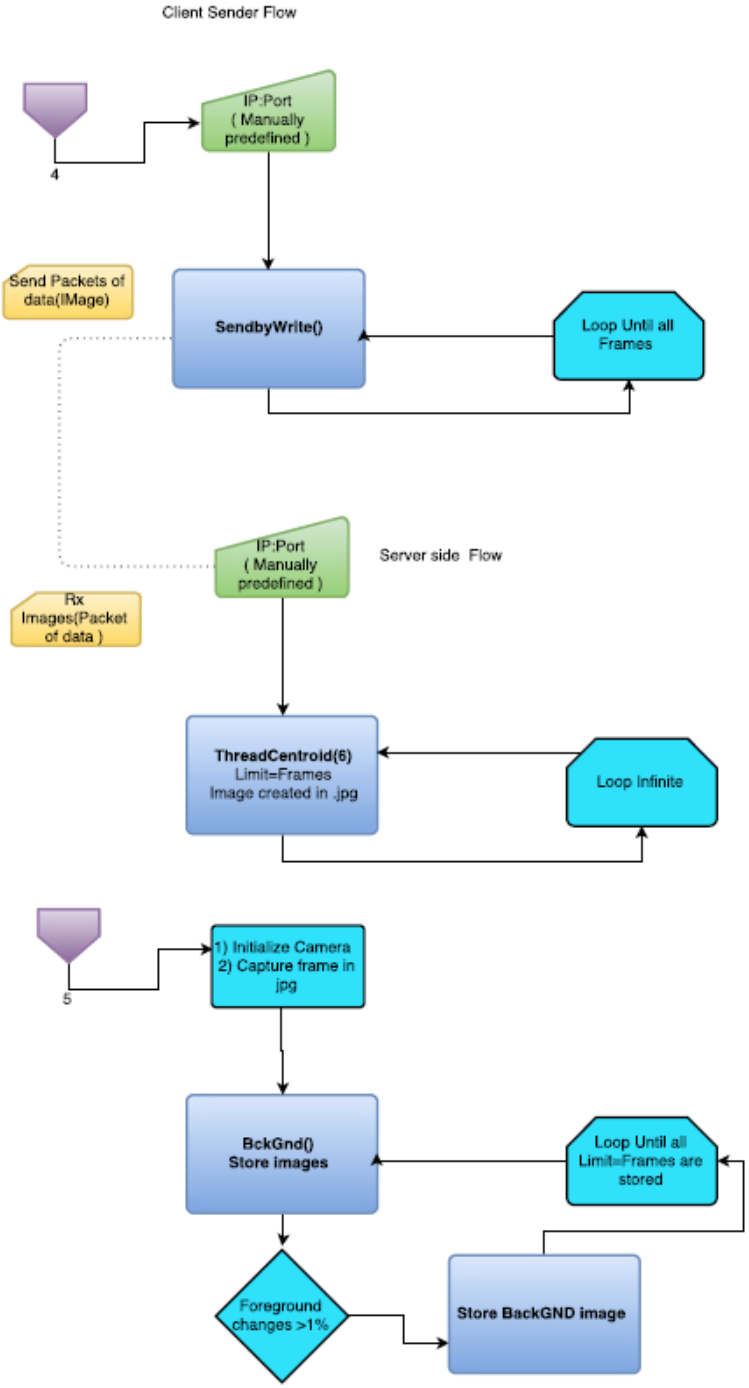


Fig 6.3 Flow chart for code design (part 3/3)

Design and Cheddar Analysis:

Design

Time lapse has been designed and scheduled as follows:
The threads – *Sequencer* (Thread 1), *Process Image* (Thread 2), *Compression* (Thread 3) are scheduled using the round robin analogy.

This is because processing of image to generate PPM file (YUV to RGB conversion) needs frame to be captured and subsequently compression a ppm file. Furthermore, sending via TCP socket and the other two image processing features require compressed image.

While the image processing specific threads SendData (THREAD 4), Image Histogram (THREAD 5) and Centroid detection (THREAD 6) run in parallel and are executed using SCHED_FIFO policy.

To perform cheddar analysis, we require specific deadlines for these each and every thread implemented. Accordingly, we have run analysis for calculating average execution time running at 1Hz and 10Hz for 200, 400 and 600 frames. The results were as follows, thus assisting us to set deadlines and perform cheddar analysis

	AVG Execution time(MS)	**with non-local host external server**	
Thread Name/No of Frames	**200 frames**	**400**	**600**
Process	6.235393	6.306202	7.812719
Compression	4.984245	5.734601	6.478983
Socket Send	654.218125	626.131733	6.208076
Centroid	15.474274	16.722788	18.739058
Histogram	7.929482	8.794421	12.189763

Table 1.1: Average execution time of different processes/Threads for 200,400 and 600 frames (with external system as receiving server)

We could observer major lags when server for receiving images was an external server than a local host as seen from Table 1.2 as the Socket send thread execution time is less while using a local host to receive images

	AVG Execution time(MS)	**with local host as server**
Thread Name/No of Frames	**200**	**400.000000**
Process	7.959344	6.265666
Compression	5.788208	6.161226
Socket Send	6.034707	6.328642
Centroid	14.681404	19.509767
Histogram	9.662909	11.317921

Table 1.2: Average execution time of different processes/Threads for 200,400 and 600 frames (with local host as receiving server)

Thus, from the above results we were able to decide (for local host for *"Socket Send'*) following deadlines for these threads running fps @ 1Hz.

Deadline set (time for execution time of each thread)	**Time(MS)**
Process	15
Compression	6
Socket Send	11
Centroid	25
Histogram	20

Table 1.3: Deadlines set for Thread (with local host as receiving server)

Cheddar Analysis

Summing the above discussion, it is understood that the first three threads are scheduled using SCHED_RR and the remaining three threads using SCHED_FIFO. In addition to this each of the above services has frequency of 1 Hz. So for our system to be feasible

Execution time of (T1 + T2 + T3) + max(Exec time of (T4 + T5 + T6)) < 1 sec

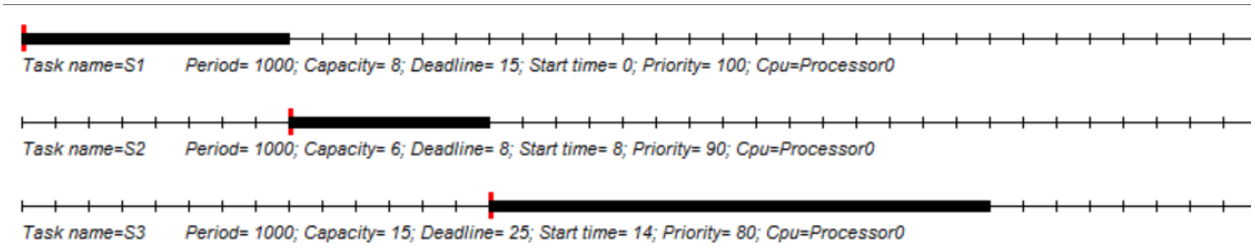
In addition to this each and every service has its own individual deadline.

We have divided our services as follows (*Table 1.4*):

Service No	**Function**	**Execution Time**	**Deadline (MS)**	**Frequency (sec)**
S1	PPM generation	7.959344	15	1
S2	Compression	5.788208	21	1
S3	Send compressed image and Image Processing features	Max (6.034707, 14.68, 9.66) = 14.68	Max (11, 25, 20) = 46	1

Table 1.4: Services(Si), Execution time(Ci) and Deadline (Di) of this system (Round Robin)

Performing the cheddar analysis for the above three services:



Feasibility Test:

Result: Feasible

```
Scheduling simulation, Processor Processor0 :
- Number of context switches :  7
- Number of preemptions :  0

- Task response time computed from simulation :
    S1 => 8/worst
    S2 => 6/worst
    S3 => 15/worst
- No deadline missed in the computed scheduling : the task set seems to be
schedulable.
```

For Worst Case Execution analysis:

Execution time of T1+T2+T3 < 1 sec, which is 7.959344 + 5.788208 + 14.68 = 28.427552 MS < 1000ms

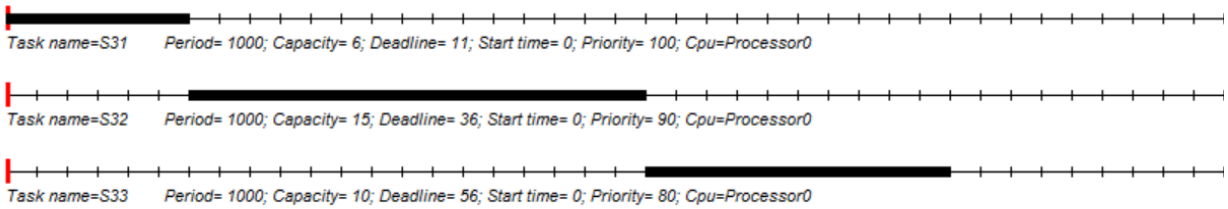
Hence Feasible by worst case analysis:

For service S3, as explained above it in turn comprises of three services namely send compressed image, Creating Image Histogram and Centroid detection, scheduled using SCHED_FIFO (*Table 1.4*):

Service No	Function	Execution Time	Deadline (MS) wrt to start	Frequency (sec)
S31	SendData	6.034707	11	1
S32	Histogram	14.68	36	1
S33	Centroid	9.66	56	1

Table 1.5: Services(Si), Execution time(Ci) and Deadline (Di) of this system(SCHED_FIFO)

The result of cheddar analysis for is as follows:



Feasibility Test:

Result: Feasible

```
Scheduling simulation, Processor Processor0 :
- Number of context switches :  7
- Number of preemptions :  0

- Task response time computed from simulation :
    S31 => 6/worst
    S32 => 21/worst
    S33 => 10/worst
- No deadline missed in the computed scheduling : the task set seems to be
schedulable.
```

Feasibility based on Worst Case Analysis:

Result: Feasible

```
2) Feasibility  test based on worst case task response time :

- Bound on task response time :  (see [2], page 3, equation 4).
    S33 => 31
    S32 => 21
    S31 => 6
- All task deadlines will be met : the task set is schedulable.
```

Observation and Analysis

The following analysis was performed on native Linux (Ubuntu) with no affinity (multiple core)

1. For 1 Hz clock

a. Results on time stamps

As we can see from the following snapshots of headers of laps00000001 and laps00001800.ppm

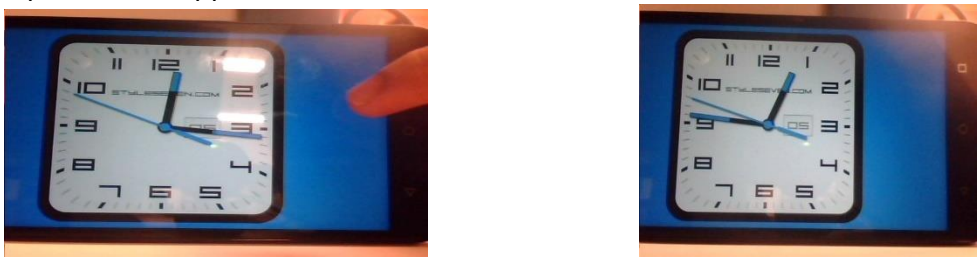


Fig 7.1) Accurate time results with a lag of 1 second (images of laps00000001 -left and laps00001800 - right) showing almost 30 minutes

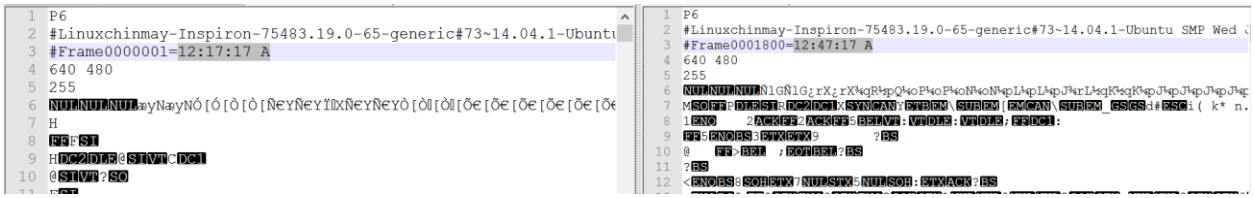


Fig 7.2) Accurate time stamps without any lag and exactly 30 minutes (images of laps00000001 –left and laps00001800- right)

b. On Video – two different time-lapse video observing external clocks are attached as part of submission. From that it can be observed that is has 0 latency in seconds scale. **On a whole the image capture has latency of 230 milliseconds which are not observable in seconds scale**.

c. Jitter and WCET analysis

We can observer that by setting deadlines in above table 1.3 most of the deadlines are met and jitter is quite less. However due to WCET for each thread being far away from averages the “average missed jitter” is high.

After Execution	**For 1800 frames with server as local host**			**Avgas jitter**	**Avgas Jitter**
Thread Name/No of threads	**Average Execution time(MS)**	**Jitter (Missed %)**	**WCET Execution time(MS)**	**Missed Deadline(MS)**	**Deadline Not Missed(MS)**
Process	8.000012	0.022702	170.150742	23.439024	8.266855
Compression	6.083409	0.013843	125.51004	24.959999	5.862998
Socket Send	5.515253	0.034464	10.686711	8.611111	2.612458
Centroid	18.373581	0.027793	172.724396	21.08	7.843911
Histogram	10.753313	0.020011	77.772369	18.611111	-10.277369

Table 1.4: Average execution, jitter, average Jitter (missed) and WCET for 1800 frames @1Hz

** *Images submitted in "ppm files for 1 Hz Verfication" folder*

2. For high rate - 10 Hz

a. Results on time stamps -As we can see from the following snapshots of headers of laps00000001 and laps00006000.ppm. We can observe a latency accumulation as duration of execution of all tasks is greater than .1 secs.

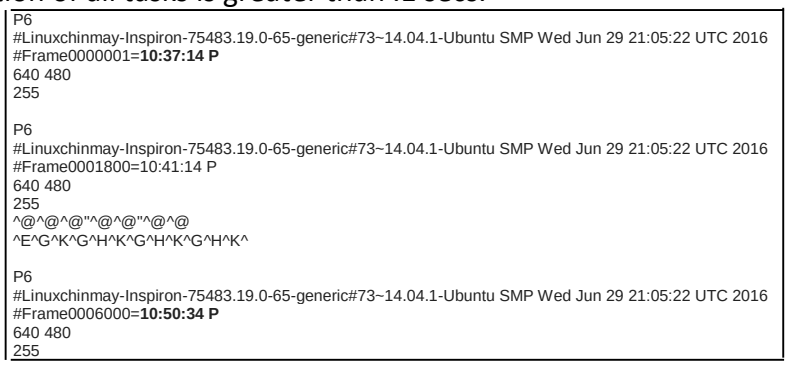


Fig 8.1) Snapshot header of images showing accurate time stamps with lag of 3 minutes 20 seconds (images of laps00000001, laps00001800 and laps00006000)

** *Images submitted in "ppm files for 10 Hz" folder*

b. After running multiple tests and verification at different rates, it was leading miss in deadlines and crashing for 10 Hz included with socket send. This analysis provided is @10 Hz excluding data been sent to receiver over Ethernet.

	For 6000 frames with server as local host			**Avgas jitter**	**Avgas Jitter**
Thread Name/No of threads	**Average Execution time(MS)**	**Jitter (Missed %)**	**WCET Execution time(MS)**	**Missed Deadline(MS)**	**Deadline Not Missed(MS)**
Process	10.308614	0.007826	988.805725	39.044182	6.211571
Compression	5.300388	0.001166	903.485657	64.203125	5.328172
Centroid	18.145617	0.011002	992.008057	82.156624	5.984056
Histogram	13.517635	0.029505	992.056152	52.846153	10.958821

Table 1.5: Average execution, jitter, average Jitter (missed) and WCET for 6000 frames @10Hz (with same deadlines as for 1 Hz) excluding sending packets over Ethernet

3. For high rate - 5 Hz

After Execution	***For 3000 frames with server as local host***		
Thread Name/No of threads	***Average Execution time(MS)***	***Jitter (Missed %)***	***WCET Execution time(MS)***
Process	*6.807995*	*0.005323*	*64.136665*
Compression	*6.15986*	*0.029940*	*163.836349*
Socket	*5.644835*	*0.120373*	*14.754226*
Centroid	*17.462093*	*0.003334*	*109.366959*
Histogram	*12.116971*	*0.005669*	*88.49337*

Table 1.6: Average execution, jitter, average Jitter (missed) and WCET for 3000 frames @5Hz (with same deadlines as for 1 Hz) and socket thread functional

4. Foreground Masking running @ 10 Hz

Analysis separately for

After Execution	**For 1800 frames with server as local host**	Jitter		**Avgas jitter**	**Avgas Jitter**
Thread Name/No of threads	**Average Execution time(MS)**	**(Deadline Missed %)**	**WCET Execution time(MS)**	**Missed Deadline(MS)**	**Deadline Not Missed(MS)**
Background Subtraction	8.000012	0.001658	101.960487	14.66666	23.174419

Table 1.7: Average execution, jitter, average Jitter (missed) and WCET for 1800 frames @10Hz which are set by finding average execution time for 100 frames

Conclusions:

The images being captured are in PPM format with embedded header of timestamps. As our system is memory bound, it is designed to transfer files using socket programming thereby supporting infinite frame capture. To facilitate the infinite frame capture, the PPM images are compressed considerably by using JPEG compression algorithm. In addition to this it the project implements image processing features such as Image Histogram and computation of centroid. The programming is done using V4l2 libraries for frame capture, opencv for image processing and POSIX threads to support parallel processing.

Furthermore, we have provided analysis of jitter and latency which are less to obtain a feasible result We have analyzed the Time Lapse by capturing 1800 frames at a frequency of 1 Hz The real-time accuracy of image capture after capturing 1800 frames is 0 sec in seconds scale. However, the total latency was coming around 200 milliseconds. Such high accuracy was possible by minimizing the frame grab time by using v4l2 buffers which is explained in detail in the above sections and skipping initial frames.

Acknowledgement:

We would like to thank Prof. Sam Siewert for his continuous support in every aspect of this course. We would also be thankful to Akshay singh for his constant feedbacks on the Lab work.

References:

[1] http://support.logitech.com/en_us/product/webcam-c200

[2] http://www.tigerdirect.com/applications/SearchTools/item-details.asp?EdpNo=5008719

[3] http://cs.ecs.baylor.edu/~donahoo/practical/CSockets/PracticalSocketC.pdf

[4] https://en.wikipedia.org/wiki/Image_compression

[5] http://www.ams.org/samplings/feature-column/fcarc-image-compression

[6] https://en.wikipedia.org/wiki/Transmission_Control_Protocol

[7]http://docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/histogram_calculation/histogram_calculation.html