CSE 546 - Project-1 Report

Monil Nisar
Samip Thakkar
Siddharth Sheladiya

1. Problem Statement

With advancement in IoT(Internet of Things) devices, the growth of data has been explosive. Hence, the computation and processing of this large scale data takes a toll on the cloud resources and it struggles to meet all the computational demands due to the high response time. Here we are trying to address this problem with the help of edge computing by building an elastic and responsive application which utilizes cloud resources and the resources of IoT devices , as a result reducing the end-to-end latency.

In this project, we are building a surveillance application which is built upon Raspberry Pi and Amazon Web Services(AWS). Raspberry Pi detects the motion which triggers its camera to start recording the video wherein it detects the objects in the video. Here the Raspberry Pi acts as an edge and AWS provides the cloud infrastructure. As the Raspberry pi's processor cannot handle multiple video processing requests at the same time, we take help of the cloud resources which help the pi to carry out the process. Hence, the Raspberry pi provides responsiveness and AWS provides scalability.

2. Design and Implementation

2.1 Architecture

We have designed a master-worker architecture where one of the instances acts as a master(controller) and other instances acts as workers where the darknet processing is done. First of all the pi records the videos after detecting motion with the help of a PIR sensor. The recorded videos are diverted according to the processing_flag which saves the state of pi's processor i.e. processing_flag if true then processor is busy and vice versa. If the raspberry pi's processor is free(which is the initial condition) then the video is diverted to its processor for processing otherwise the video is uploaded to s3 bucket and added to the input queue where it waits for its turn to be processed.

On the other side of the architecture, the master is working independently from the pi and is continuously polling the input queue for the videos. So when master detects the presence of video in the input queue, it launches a worker instance from its pool of worker instances and increases the threshold(initially 0) by 1. The threshold decides whether to launch another worker or not on the basis of length of input queue. Similarly, the threshold is decreased by 1 when the instance is closed so that it is ready to be launched again. The Master also continuously polls the output queue for a closing message from the pi. This message helps the master to decide when to terminate the program.

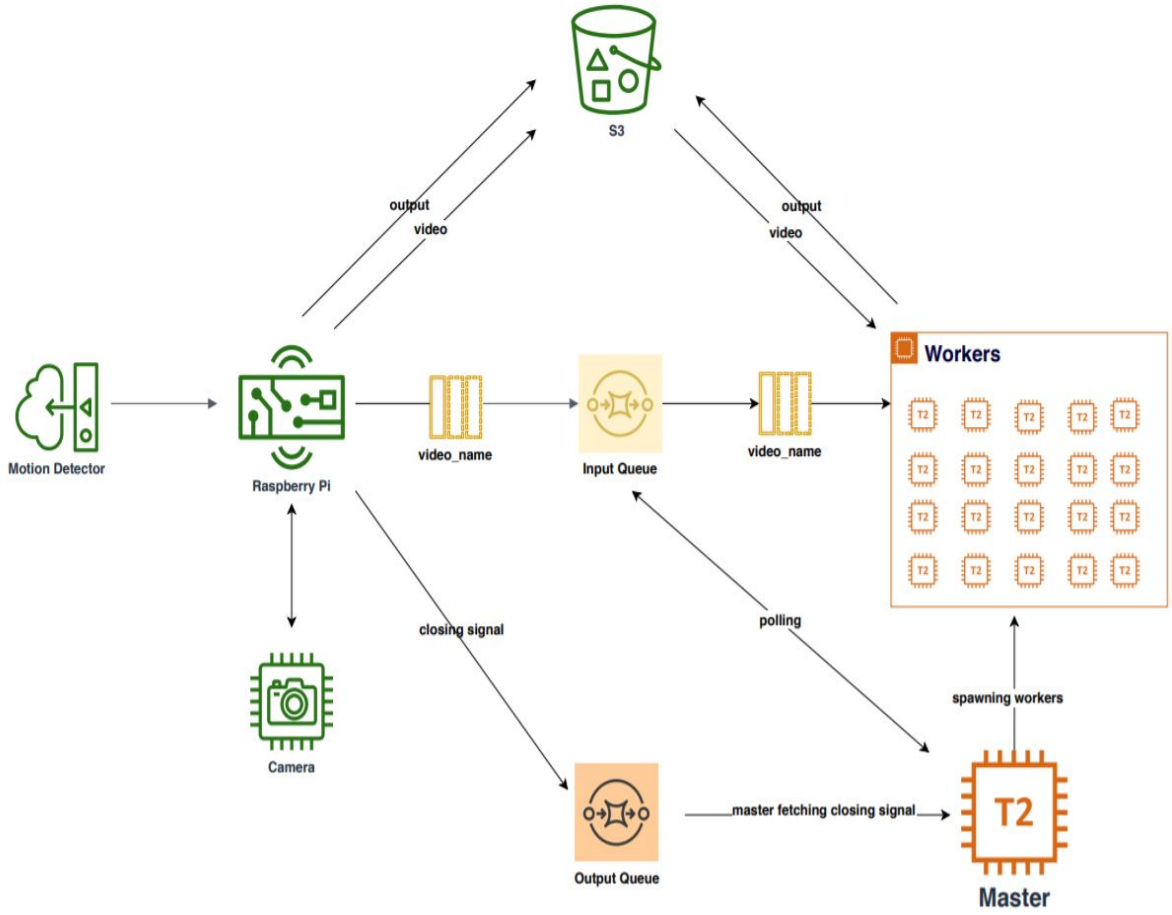

Fig-1: Architecture

The worker instance simply fetches a video from the input queue and downloads it from the s3 bucket.The video is processed on the basis of the darknet model and the output is sent to the s3 bucket. This worker process is repeated until the input queue is empty.

2.2 Autoscaling

Autoscaling is one of the five characteristics of cloud computing. Auto scaling helps the users and business to make sure no resources stay idle and go waste. Autoscaling helps businesses to maximize the resource utilization and make the most profit out of it. Our architecture of edge computing uses AWS resources to scale out. We have a master-ec2 running as our master instances. The master will determine whether there is a need to scale-out or not. If there is a requirement, then it will start worker instances. The number of worker instances will depend on the number of videos that are needed to be processed.

In our version, one process on pi acts as a producer that records video, while another acts as a consumer that processes it locally. Videos are consumed on the pi first, but if pi cannot handle the workload, additional help is provided by AWS resources. For that, the master will initiate worker instances as needed. To keep everything in sync, a shared SQS between the instances and pi is used, which would serve as a buffer for consumer instances. The master keeps check on the number of videos in the queue, and when it exceeds the threshold, the master launches some instances and scales-out the performance. And when there are less videos in the queue, it stops the instances and scales-in.

3. Testing and Evaluation

The evaluation of the architecture is done with end-to-end latency as a parameter. The performance of system is also considered as a factor ie. the accuracy of the object detection code. An end-to-end baseline latency is provided and all the videos should be processed within the given time. The program runs darknet object detection, which is a pre-trained model for object detection. We actually implemented multiple architectures. One of the architectures used provided an average latency of 40 seconds on pi and after

some modification and delaying some threads, the average per video latency was decreased to 36 seconds. This was a huge gain as pi processed almost 60% of videos. For worker instances, the average processing time ranged from 3 to 4 minutes as the starting and stopping instances used a lot of time. So we had made a custom waiter function to detect the state of instance and this gave little performance boost.

After testing and improving the architecture and code, the average end-to-end latency came to be around 4 minutes and 50 seconds for recording and processing 10 videos, which looked great.

4. Code

- ## Raspberry Pi

  The Raspberry Pi acts as a recorder which records the video for 5 seconds as soon as it detects motion from the PIR sensor. The code initially starts recording as the motion is detected, and as soon as a video is recorded, it creates a thread to upload it to the S3 bucket. Meanwhile it keeps on looking for motion. Also, it creates a child process that would use darknet to detect objects in the video if the pi is available, that is if pi is not processing any other video, otherwise it sends the video to SQS so that it can be processed by an ec2 instance because pi cannot process two videos in parallel. After all the videos are recorded and Pi has completed processing, it then uploads the results to S3 bucket. This multiprocessing model leads to complete utilization of the resources from pi first and then EC2.

- ## Master

  The EC2 Master is one of the instances which acts as a Controller and launches multiple worker instances according to increasing load of videos. The code in the master instances has mainly 3 functions assigned to it.

- ***Scaling_ec2***: This functions decides whether to launch an instance on the basis length of input queue. It continuously polls the input queue and calls another function spawn_ec2 to launch the instances. Considering the scenario where there may be need of launching multiple instances together, we have a multithreading implementation for launching multiple instances. So now each thread handles a single instance and performs function spawn_ec2.

- ***Spawn_ec2***: This function is called when it is decided which instance has to be launched. It takes (shared, instance_id, instance_threshold) as a parameter which is used launching the instances. The instance is launched with the help of boto3 client methods. We have implemented our own waiter function which checks the instance status every 7.5 seconds to reduce the latency. After the instance status is confirmed running, we pass the command to run the python program on the worker instance using **ssh** command along with the instance key pair and public dns id which is fetched before passing the command. The command is wrapped in Xvfb run which transports the displays so we do not have to manually do it every time before running the command. After getting the control back from the worker to master, the length of input queue is checked again and if it is greater than 1, again the program is made to run which saves us the time of launching the instance again. The worker is then closed with the help of ec2 client and again sent back to the worker pool where it is ready and waiting to be used again.

- ***Signal_fetcher***: The signal fetcher function acts as a separate process which continuously polls the output queue for the message from the Raspberry pi that it has now stopped recording all the videos and this makes the master decide

whether to terminate the program after the queue is empty or to wait for more videos.

- **Worker**

  The main implementation of object detection is done on the worker instances. As the instance used has to be ***t2.micro***, we decided to keep the code serial for the worker instances to reduce the latency. It fetches messages from the input queue and downloads the video from the s3 bucket which is then processed with the help of darknet command which is passed with the help of ***os.system*** command. The output is initially saved in a text file which is then parsed using the ***parse_output*** function which then reduces the output file to a list of objects detected and output is written in file which is then sent to the s3 bucket as a key-value pair. This process is repeated till the queue is not empty.

- **Steps to run the application:**
  - Creating AWS services:
    - Create a Bucket with an unique name
    - Create a Standard Queue with an unique name
    - Create an EC2 t2.micro instance with name Master using community AMI **ami-0903fd482d7208724**
    - Create an EC2 t2.micro instance with name Worker using community AMI **ami-0903fd482d7208724**
  - Setting up the environment:
    - Raspberry Pi
      - Install everything from **requirements.txt**
      - Copy aws credentials in the credentials file
    - Master Instance
      - Install everything from **requirements.txt**
    - Worker Instance
      - Install everything from **requirements.txt**

        - Transfering all darknet files to home directory.
            - $~:mv darknet darknet2
            - $~:mv -rf darknet2/* ./
            - $~:rm -rf darknet
        - Create a snapshot of the current instance and create multiple worker instances using this snapshot-image**.**

- Copy the following files:
    - Unzip the folder, keep all the files in the same folder.
    - Copy ***recorder_pi.py*** in Raspberry pi's darknet folder.
    - Copy ***copy_aws_credentials.py*** in pi's home folder.
    - Copy ***master.py*** in master instance's darknet folder.
    - Copy ***worker.py*** in all the worker instance's home folder.

- Setting up correct credentials:
    - From pi, run *copy_aws_credentials.py*

      python3 copy_aws_credentials.py

    - From Master instance, run *master.py*

      python3 master.py

    - From pi, run *recorder_pi.py*

      python3 recorder_pi.py

5. Individual Contribution (Monil)

- **Design**
  - Structure of the directory and environment on the instance was set up by me. This included moving directories, installing dependencies, and preparing files.
- **Implementation**
  - The program, **recorder_pi.py**, to record the video whenever motion is detected was written by me. I had written a multi-processing program that would allow pi to record video as well as process video both at the same moment. Proper resource utilization was the goal. Because of it, the program on pi would consume videos from the SQS once the pi has no more videos to record, and hence scale-out.
  - We found out that the default waiter provided by the EC2 client takes up a lot of time. So I prepared my version of waiter, exclusively for our application to reduce latency.
  - The code to extract objects detected by darknet was written by me. Also implemented the multi-threaded code to upload video and results to S3 and input messages to SQS. As uploading files does not require much CPU processing and it is basically an I/O operation that could be done independently, creating a thread for it looked ideal.
  - I prepared a supporting code that would copy the credentials to all instances, so one does not have to copy them manually.
- **Testing**
  - Tested the darknet code on pi and worker instance, if the proper results are being generated. Added the logic of collecting the error stream into a file that was used to debug and analyse later.
  - Tested time utilization for video processing with different architecture.

6. Individual Contribution(Siddharth)

- **Design**
  - Created multiple architecture for using different combinations of instances.
  - First of all created an architecture where Raspberry pi was assigned the task of the controller, but later we realized that making pi as a controller makes the application fully hardware dependent and failure in hardware can cause application crash midway during the processing.
  - Then I came up with a master-worker architecture where one of the instances acts as a master and controls all the worker instances. Using this architecture made the application more loosely coupled and assured that the processing would not stop on the instances even if the Raspberry pi is damaged.
- **Implementation**
  - Implemented **autoscaling** in master.py, where the autoscaling is handled using the length of the queue and assigning a threshold to each instance which increases and decreases according to the number of instances started and closed.
  - Implemented **worker.py**, where the code to process the videos is implemented. As the t2.micro instance cannot handle multiple heavy processes, worker.py is made serial to reduce the latency.
  - Implemented the **motion detection** part of the application which includes dealing with the PIR sensor and testing it using a simple python program.
- **Testing**
  - Tested the autoscaling part using various numbers of instances and calculated the optimum threshold for the autoscaling part.

7. Individual Contribution(Samip)

- **Design**
  - Framework validation/testing: Once the setup was done, performed the validation/testing and reported the errors found and contributed in solving the errors.
  - Helped in setting up the multithreading environment for the faster execution and uploading the videos and results to S3 and the messages to SQS.

- **Implementation**
  - Instance creation (on ec2): Set up the environment on EC2 and monitored the execution of the project and reported the errors and helped in solving them.
  - Researching about boto3: As the project is implemented in Python, researched about boto3 and other libraries for working with AWS services in python and implementing the code. Research about the API boto provides for low-level access to AWS services.
  - Helped other team-mates during the implementation of independent modules and error reporting and solving.

- **Test**
  - Once all the parts of the projects were implemented and tested independently, combined all the parts and performed the final testing to check for the answers for different inputs and report the latency and work towards optimizing it.