UNIVERSITÀ DI PISA

Mobile & Cyber-Physical Systems

Academic Year 2020/2021

Project - Final Document

A Smart Camera for the Internet of Things

Jacopo Bandoni 561254

Giuseppe Bisicchia 559033

1 Introduction

In this document, we will describe our implementation of a SmartCam that aims to improve the sharing of video data from multiple cameras from a network point of view, using a hybrid approach to manage frames that involves both local memory and the cloud. This approach was recently introduced by the startup Verkada.

We will discuss the method and, we will show our implementation and a demo approaching the problem with a much lower hardware capability using a Raspberry Pi.

1.1 Context

Right now in the market, the two most common approaches in handling data from video cameras are:

- Cloud approach: where the video captured by the camera is sent straight to the cloud, enabling to keep the data secure and enabling to access it from anywhere just by connecting to the cloud.
- Local approach: where the video is stored locally, removing all the problem related to an internet connection.

Note that in both cases, generally, the client chooses a fixed amount of memory that can be stored and once that it's filled, the older video data is overwritten.

1.2 Problem

The ideal requirements for a smart system composed of multiple video cameras would be the ability to safely store and share video data from several cameras in real-time across the Internet. However, there are wide gaps in the previous two approach to meet up the ideal requirements.

The cloud approach can face difficulty in case of slow connection or band limits on the upload of multiple cameras in the same network, which can lead to network congestion and consequently to slowdowns. Furthermore, the numerous requests can lead to a risk of causing a server overload. Finally, centralisation could lead to availability problems.

The local approach fail both in sharing video data, having to access it in the proximity of the video camera, both in safely storing the data being it at risk of theft, sabotage or memory defect. Finally, computational and storage capabilities can be quite limited.

2 Proposed solution

Basing on the model of Verkada, we propose a hybrid solution where the video camera saves in its memory the video data, storing a frame every y milliseconds. In addition to that, every x milliseconds (with x much larger than y, generally in the order of seconds) the camera sends a frame to the cloud with some metadata:

- Date and time of when it was captured.
- Camera's ID that captures that photo.
- Possibly other, in future versions (e.g. position of the captured photo or presence of people).

Therefore in the camera, the frames are saved at a much higher rate compared to the cloud and we will refer to y as the *cap timer* and we will refer to x as the *server timer*.

It is also possible to ask the camera to compare the frame just taken with respect to the last one stored and, to save (and if necessary send) the new frame only if the differences between the two frames are above a certain threshold, that can be set by the user. This setting requires higher computational capacities but allows to save a lot of memory and to congest the network less. We will refer to that as the *optimised version*.

The client that wants to look at the camera's data asks the server for retrieving all the stored frame, and once it receives them, it can see an overview of what's happening in the supervised area. If the client sees something suspicious or just want to better understand what is happening in a given period it can ask the camera for better quality data which will be sent on demand from the local storage.

In addition, the user can also request to have at the maximum frequency live streaming to view the current situation.

We can observe how this approach is capable of guaranteeing access to the video data which is efficient from a network perspective limiting the sending of video data just when they are useful for the client. In this way, we can achieve a system that is more easily extendable and scalable to a greater number of devices and which is more suitable for low latency applications and slower or wobbly network scenarios.

2.1 Use cases

This system can be used for processing data for which the user needs to be informed as soon as possible (with little latency) or in applications with multiple cameras in the same network.

For example, in a company there is a need to keep track of the rooms where access is prohibited in certain time frames or by certain people, placing a camera in each room can be used to observe the situation at any moment and warn in time, when necessary, those who are responsible.

Another scenario in which the proposed system could be useful is that of smart cities. In such a scenario there can be numerous heterogeneous IoT cameras. In this case, it is important to have a global vision (albeit approximate) but also reasonable in terms of bandwidth used and memory occupied, but it may also be necessary to focus on a specific area to make detailed observations. The management and possible processing of continuous streaming by all the cameras could be very expensive. Additionally, cameras may not have easy and/or reliable continuous network access.

Note that other use cases can be set if we imagine implementing an AI model which acts on the cloud, or if it is possible on the cameras, which can be used to detect people identity or, more generally, the presence of people and perform notification alert without human intervention.

3 Architecture

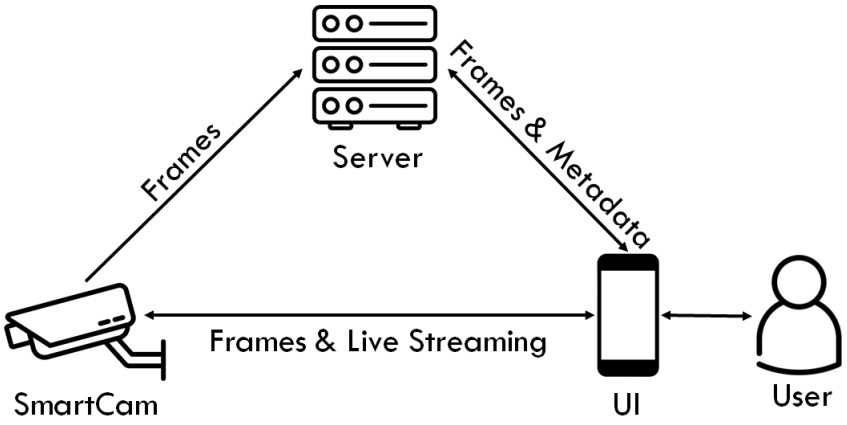

Figure 1: Blackbox view of the system.

The system's architecture can be decomposed into three main components:

1. Server
2. SmartCam
3. User Interface

The Figure 1 illustrates how these components interact with each other.

- The User Interface interacts with the Server retrieving the frames and/or metadata through a GET request.
- The SmartCam interacts with the Server to post the frames sent at a lower rate.
- The User Interface interacts with the SmartCam retrieving the frames recorded at a higher rate (relative to particular past time frame or just live).

In this way, the user can interact through an interface both with the server to have an overview of the areas that are being filmed and can directly request a more detailed video from the camera or request live footage at the maximum frequency. Furthermore, it is always possible to ask for only the metadata to process statics or in general elaborations that do not necessarily require the inbound of frames (in order to reduce traffic in the network). The frames can however be requested at a later time.

3.1 Requirements

From this kind of architecture we would like to expect that this approach can manage to be more efficient from a network perspective. In details we would like it to be capable of:

- Sharing data from everywhere with lower latency.
- Reliably storing data.
- Handling efficiently several devices in the same local network.

4 Implementation

In this section, we will see the implementation of the three different components, the Sever, the SmartCam and the User Interface, giving an overview of the implementation of each component with a description of the main features. Furthermore, the frameworks, libraries and, where necessary also the platforms used for execution, will be illustrated. The implementation is available on: `https://github.com/GBisi/MCPS`. The repository is divided into three sub-folders one for each component to launch the server and/or the SmartCam just enter the directory and run *./run.sh*.

4.1 Server

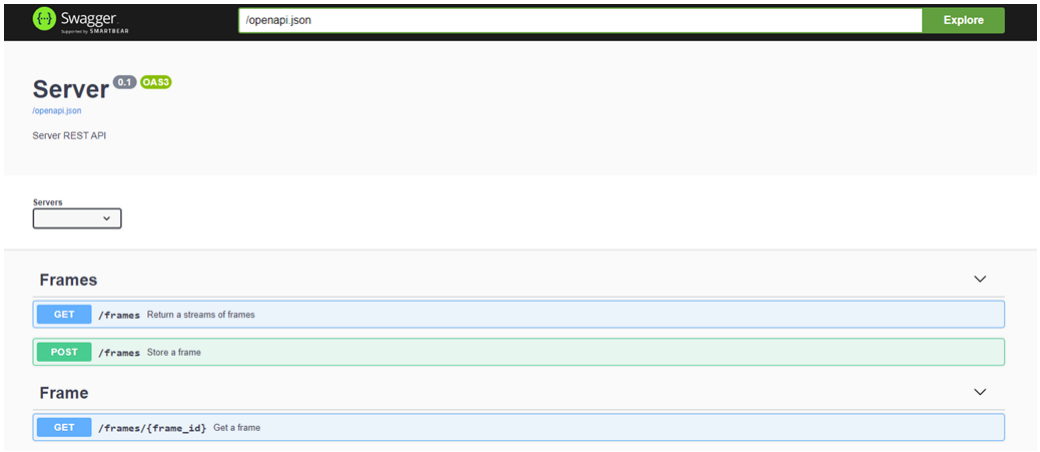

Figure 2: A screenshot of the Server interface.

The Server, available on `https://mcpserver.eu.pythonanywhere.com/ui/`, was developed in python3 using the Flask microframework and Swagger (OpenAPI) to offer a standard REST interface and at the same time through the server page `https://mcpserver.eu.pythonanywhere.com/ui/` also a visual interface to interact with the server, mainly for testing purposes. The database was implemented with SQLAlchemy. Frames are saved directly to the file system for efficiency reasons, while related metadata and a reference to retrieve the file in the file system are stored in the database. Finally, the server is hosted at `https://eu.pythonanywhere.com/`. The server offers three main features:

- Store frames and related metadata.

- Send a frame and its metadata upon request.
- Send a stream of the stored frames.

The server, therefore, acts as an aggregation point for the data coming from the SmartCams. In this way, it is possible to have only one control point for managing all the frames.

The server, as illustrated in Figure 2, offers three APIs to implement the functionality described above.

- *GET /frames*: with this call, it is necessary to specify in the query field of the request the ID of the camera (source) of which the user want to obtain the stream of frames. It is also possible to specify a start and end period, to obtain the frames sent only during that period to be able to make targeted searches. Finally, it is possible to specify if the metadata are needed, in this case only a list of JSON objects is returned.
- *POST /frames*: send a frame to the server, it is necessary to specify in the header of the HTTP packet the ID of the camera and the timestamp of the moment in which the frame was obtained. Returns the frame metadata as a JSON object.
- *GET /frames/frame_id*: requests a specific frame through its ID. The frame is returned along with its metadata in the packet headers.

The metadata are easily extensible to support specific applications, the default ones provided in this implementation are the ID of the frame, the ID of the source, the timestamp of the moment in which the frame was obtained, the timestamp of the moment in which the server received frame and frame size.

Finally, the stream is implemented as *Motion JPEG* (M-JPEG) because it tolerates rapidly changing motion in the video stream and is not computationally intense to manage.

4.2 SmartCam

The SmartCam is the central component of the system. It implements frames management internally and decided when to store one and when to send one to the server.

The SmartCam is also implemented with the triplet python3, Flask and Swagger. Furthermore, the OpenCV and scikit-image libraries have been used for the management and processing of the frames.

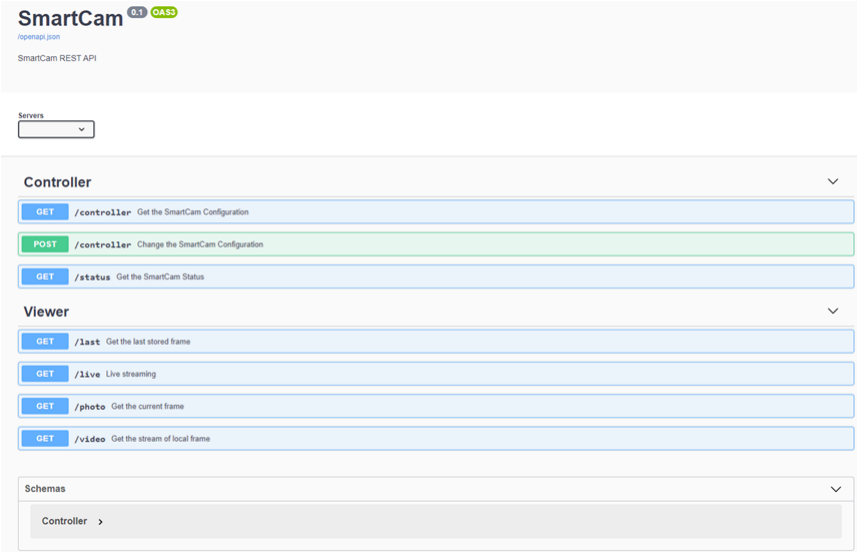

Figure 3: A screenshot of the SmartCam interface.

The SmartCam has been implemented and tested through the RaspberryPi 3B platform.

The SmartCam has also been designed to be independent of the hardware used to obtain the frames. The logic is abstracted through any python class that offers the *get_frame()* and *get_id()* methods. The SmartCam has been tested using an IPCamera and a so-called VirtualCamera, that is, using a pre-recorded video which is used as a basis for the collection of frames.

Thanks to Swagger, the Smartcam also offers a graphical interface through the URL */ui*, as shown in Figure 3.

The tasks of the cameras are to periodically obtain the frames and store them locally and send them, even with a different period, to the server. Furthermore, the user can directly ask the camera to take a frame or to obtain the last frame stored. Furthermore, the user can request the stored frames or live streaming in the form of video. The SmartCam can also be configured to compare, each time it is taken, a new frame with the last one stored and calculate the difference between the two frames using the *Structural Similarity Index Measure* (SSIM) and take it if the index is below a certain user-configurable threshold.

Finally, it is possible to change the configuration of the SmartCam at run-time. As illustrated in Figure 3, the REST APIs offered by the SmartCam are:

- *GET /controller*: get the current configuration as a JSON object. The configuration parameters are easily extensible. The basic ones are:
  - *cap_timer*: indicates how often a new frame is taken.
  - *server_url*: the server URL to send the frames to. If set to null the SmartCam does not send anything.
  - *server_timer*: indicates how often a new frame is sent to the server.
  - *server_ratio*: indicates after how many stored frames one should be sent to the server. If both this and the *server_timer* are active then the lesser of the two is chosen. In this way, it is possible to describe different management policies taking into account also the possible latencies.
  - *threshold*: indicates the minimum index level difference between the frame that has just been triggered and the last one below which the new frame is kept (and in case sent to the server). If the threshold is set to 1, the comparison is deactivated because the condition will always be true anyway.
- *POST /controller*: updates one or more parameters of the SmartCam configuration.
- *GET /status*: get the current status of the SmartCam. Information such as the status of the CPU (percentage, frequency and temperature), of the memory (free and total) and the disk (free and total).
- *GET /last*: returns the last stored frame.
- *GET /live*: sends the current stream at the maximum frequency.
- *GET /photo*: takes a photo on the spot.
- *GET /video*: requires a stream of the saved frames being able to filter them based on a specific time window (as in the server).

As it is possible to observe, therefore, the Smartcam can be dynamically configured in order to implement different frames management policies.

To conclude, it is possible to think of the SmartCam as divided into three interacting components:

- *Viewer:* takes care of interacting with the user, receiving requests and sending replies, perhaps interacting with the other components.

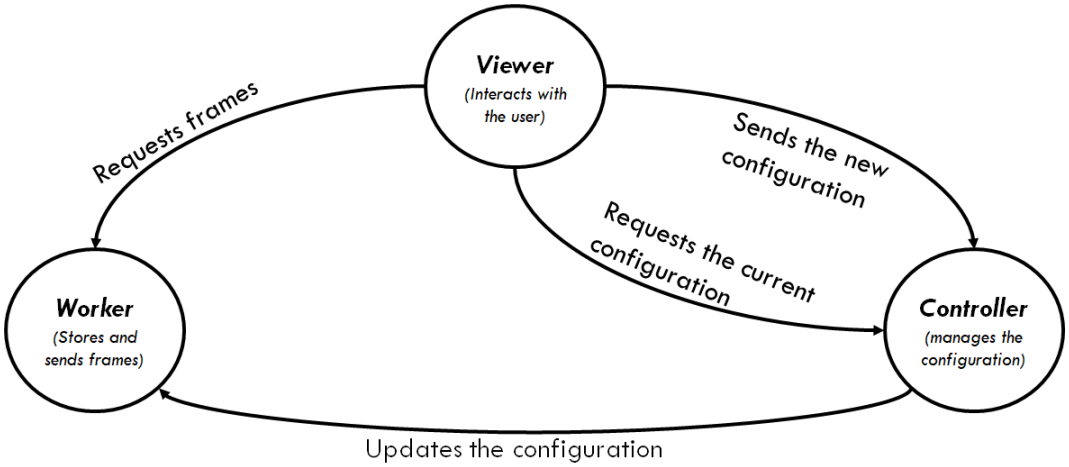

Figure 4: The SmartCam components.

- *Controller:* manages the configuration and notifies the Worker of changes. The Viewer will inform it of user requests for changes.
- *Worker:* takes care of taking the frames according to the configuration, compare them if required, and in case save them and send them to the server.

It is important to note that the SmartCam is implemented as a single entity but this division can be useful to illustrate how the various functions interact with each other.

4.3 Interface

The User Interface was developed using Flutter, a framework for developing applications in both IOS and Android and it is structured in the following way:

- Initially it's shown a *connection screen* with a form where the user inserts the Camera's URL.
- When the form is submitted, an HTTP request is sent to the camera to access some configuration's data like its ID and the URL of its server.
- If the connection to the camera is successful is shown the *frame preview screen* where, once the user updates the page, has established the

connection with the server and loaded all the frames with the relative metadata.

- Next to the *frame preview screen* there is the *streaming screen* where the user can ask the camera for a video in a specific time frame.

For the testing and execution of the application was used the official Apple simulator from the Simulator Kit.

5 Experiments

The experiments were performed on a Rasberry Pi to measure both the load on the device's hardware and the load on the network.

We used different configuration to compare the different metrics used in the evaluation. In particular we collect the results of 6 different configurations:

- *Cap timer*$= 0$ and *Server timer* $= 0$
- *Cap timer*$= 0.5$ and *Server timer* $= 1$
- *Cap timer*$= 1$ and *Server timer* $= 5$

And each one of this configuration is tested both with and without the *optimised version*[1]. For what it concerns the device hardware we measure:

- CPU percentage used
- CPU frequency
- CPU temperature
- memory used
- disk used

For what it concerns the load on the network we measure the bandwidth (measured in MB/s) of each configuration. Those measurements were performed in an interval of 5 minutes, with a break between one and the other of 15 min. In particular:

[1]By optimised version we mean a configuration in which the process of comparing between frames is active. For the tests the threshold has been set at 0.9, i.e. a frame is discarded if it differs from the last saved by less than 10%.

- For the measurements concerning the hardware we relied on the psutil library to measure CPU, memory and disk statistics every 250 ms. We also performed a control test that lasted 5 minutes to have a basic reference. The data were then processed to obtain the average and/or historical values.

- For the measurement concerning the load on the network in the SmartCam component we use the *framesize*'s metadata. Once 5 minutes are passed and the tests are finished we retrieve the metadata of all the frame obtained in the test's interval from the server and, we sum all the values of the different *framesize*s.

As illustrated in Figure 5 and as we could expect as the *cap timer* and *server timer* decrease the load on the CPU decrease, even if not substantially and, the bandwidth decrease by $\frac{1}{3}$ comparing the first with the second configuration and much more comparing the first with the third one.

It is also interesting to note that the average frequencies of the CPU are quite similar in all the experiments although with a slight decrease with increasing timers and slightly higher values in the optimised versions.

Furthermore, it can be seen that the temperatures increase much more in the optimised versions but that there are no significant differences between them.

For what it concerns the optimised versions it considerably raises the load on the CPU and even on the memory but it improves the performance regarding the disk percent and the bandwidth.

As far as memory is concerned, it is also interesting to note that between the three configurations of cap and server timers the results are quite similar. Comparing the optimised versions with the non-optimised ones it can be seen that there is an average and a greater variation in the optimised versions i.e. the CPU-intensive.

6 Demo

For the demo, we will illustrate the SmartCam startup and, we will show the functionalities through the Swagger interface, among which we will show how it is possible to dynamically configure the SmartCam.

Later, again through the interface offered by Swagger, we will illustrate how to interact with the Server to obtain frames and metadata.

Finally, we will present the UI of the final User through the Apple simulator.

Figure 5: Some graphs illustrating the results of the experiments. For the bar graphs, the 5 minutes average (1200 samples) was calculated. For the historian, for reasons of greater ease of reading, it was decided to plot one point every 12, for a total of 100 samples equally spaced in time.