Cloud Computing Report

Khalil Ouald Chaib

August 2024

1 How To Run The Project

1. If the metrics-server is not installed, you can install it by running the following command:

   ```
   kubectl apply -f
   https://github.com/kubernetes-sigs/metrics-server/
   releases/latest/download/components.yaml
   ```

2. Run the build.sh script
3. Go to the cameras folder
4. Put all the images in the images_test folder
5. Run npm install
6. run npm main.js
7. Once your finished run the deleteall.sh script

2 Architecture and Design

Figure 1 shows an overview of the architecture of the system.

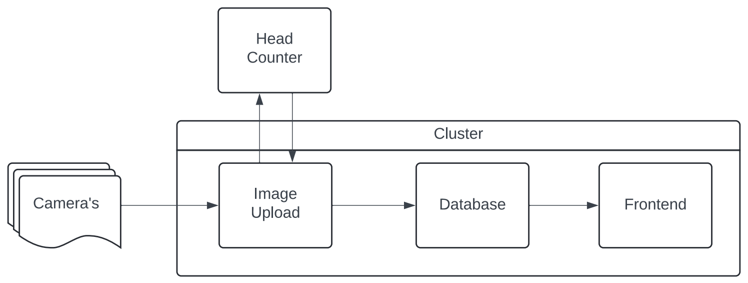

Figure 1: Architecture

2.1 Camera's

The flow of data begins with simulated camera inputs. The assumption was made that the camera's are outside the cluster. To emulate the behavior of physical cameras, a script named **main.js** is used. This script is designed to send periodic HTTP POST requests to the image upload service, simulating the upload of images captured by cameras at different intervals.

The script uses the `fs` (File System) module to read image files, `axios` for making HTTP requests, and `rxjs` for handling asynchronous event streams.

Key functionalities implemented in the script include sending the images to the image upload server, simulating four different cameras, each with a unique ID. Each camera sends images at different intervals:

- **Camera 1**: Every 10 seconds.
- **Camera 2**: Every 30 seconds.
- **Camera 3**: Every 5 seconds.
- **Camera 4**: Every 60 seconds.

Images are stored in a directory called `images_test`, and the script randomly selects an image from this directory each time an upload is triggered. The selected image, along with the corresponding camera ID, is sent as part of a `FormData` object, which includes the image file and a form field for the camera ID.

The script leverages `rxjs` to manage the timing and flow of these operations. Each camera's image upload process is managed by an `observable`, which emits values at the defined intervals. The `switchMap` operator is used to handle the asynchronous HTTP requests, ensuring that each image upload is properly managed.

2.2 Image Upload

The Image Upload Service plays a important role in the system, acting as the entry point for receiving images from various cameras. This service is implemented using the Express framework, which handles HTTP requests, and Multer, a middleware for handling file uploads. When a camera captures an image, it sends a POST request to the service at the `/crowdy/image/upload` endpoint. This endpoint receives the image file, which is temporarily stored on the server.

To manage the flow of incoming image data efficiently, the service utilizes the `rxjs` library, which is a tool for handling asynchronous data streams in a reactive manner. By wrapping the image upload process in an `Observable`, the service treats each image upload as a stream of events, allowing it to process multiple uploads concurrently without blocking the server. This approach is particularly advantageous in environments where real-time processing is critical, as it ensures that the system remains responsive under heavy loads.

Once an image is received, the service initiates a series of asynchronous operations. The image is first sent to the Head-counting Service, where it is analyzed to determine the number of people present. The count returned from this service, along with the original image and camera ID, is then forwarded to the Database Service.

The use of `rxjs` in this context provides several benefits. It allows the service to manage the asynchronous nature of image uploads smoothly, ensuring that each image is processed as soon as it is received, without waiting for other uploads to complete. This non-blocking, reactive approach is essential for handling the high concurrency typical in a system receiving data from multiple cameras.

2.3 Database

The Database Service is a critical component that not only stores headcount data but also manages real-time analytics and aggregation. Upon receiving headcount data from the Image Upload Service, this service stores the information in a database that is structured to support both real-time and historical data analysis.

The database is designed to maintain several key data structures:

- **Real-Time Total Head Count**: This is an aggregated count of all individuals detected across all cameras in the system at any given moment. It provides a snapshot of the total number of people present as detected by all cameras.
- **30-Second Interval Head Count List**: The service aggregates headcount data every 30 seconds, storing a list that reflects the total head count from all images processed within each interval. This list is crucial for understanding fluctuations in crowd density over time and can be used for generating time-based reports and analytics.
- **Camera-Specific Data Map:**: The service maintains a map where each camera's ID serves as the key, and the corresponding value is the last image sent by that camera along with its associated total headcount. This allows the system to track the most recent data for each camera, facilitating quick access and real-time monitoring of specific camera feeds.

To manage the continuous flow of data, the service uses `rxjs` and its `bufferTime` operator. This setup enables the service to buffer incoming data for a specified duration (e.g., 30 seconds) before processing it in batches. This approach allows for efficient aggregation and reduces the computational load on the system.

The service also integrates with `Socket.IO` to provide real-time updates to connected clients. As soon as new data is processed—whether it's an update to the real-time total head count or a new entry in the 30-second interval list—the service emits this data to clients, ensuring that monitoring dashboards and other user interfaces reflect the latest information.

2.4 Services

Several services were configured as load balancers to facilitate external and internal accessibility. This decision was driven by specific requirements for each component within the Kubernetes cluster.

The image-upload service was designated as a load balancer because its IP address needed to be exposed to external entities, including the cameras that send images and the head-counting service responsible for processing them. This configuration ensures that these external components can reliably connect to the image-upload service, regardless of the internal setup of the cluster.

Similarly, the head-counting service was also configured as a load balancer to simulate its role as an external server. Although the head-counting service operates within the cluster for this setup, configuring it as a load balancer provides a realistic simulation of external server interactions, allowing the system to mimic a scenario where this service might be hosted outside the cluster.

The front-end service, which interfaces with end users, required exposure outside the cluster to allow user interactions with the application. By using a load balancer, the front-end service is accessible through a stable endpoint, ensuring a reliable connection for users interacting with the application.

The database service, traditionally not exposed outside the cluster for security reasons, was configured as a load balancer in this setup. This configuration was necessary to enable the front-end service to connect directly to the database. There were connectivity issues because the client could not access the environment variables to obtain the database IP address. Although this approach ensured successful connectivity and data retrieval, exposing a database via a load balancer is generally not recommended for production environments due to potential security risks. Typically, databases should be kept private and accessed securely within the cluster.

3 Autoscaler

The Kubernetes Horizontal Pod Autoscalers (HPAs) configured for the database, head-count, image-upload, and frontend deployments manage the dynamic scaling of pods based on CPU utilization. Each HPA monitors CPU usage and adjusts the number of replicas to maintain optimal performance.

The database-hpa scales between 2 and 10 replicas based on a target CPU utilization of 50%. It scales up by 2 pods or 75% of the current pods every 60 seconds and scales down by 50% every 60 seconds, with a stabilization window of 30 seconds. This ensures that the database deployment can handle varying loads efficiently while minimizing sudden scaling changes.

Similarly, the head-count-hpa scales between 3 and 30 replicas with a target CPU utilization of 50%. It scales up by 5 pods or 50% of the current pods every 30 seconds and scales down by 50% every 60 seconds. The stabilization window for this HPA is set at 30 seconds, promoting consistent scaling behavior.

The image-upload-hpa follows comparable scaling policies, scaling between 2 and 10 replicas with a target CPU utilization of 50%. It scales up by 3 pods or 75% of the current pods every 60 seconds and scales down by 50% every 60 seconds. The stabilization window is set to 30 seconds.

The frontend-hpa is scaling between 2 and 10 replicas. It scales up by 3 pods or 75% of the current pods every 30 seconds and scales down by 50% every 60 seconds, with a stabilization window of 60 seconds. This setup reflects the frontend's higher scalability requirements and variable load handling.