Camera System design

Ref initial requirements / PDR here: <https://nymblelabs.notion.site/HLD-LLD-Design-Task-9578f45ef2ec4e51b70e7571da2647b8>

# Requirements

In short, we need a camera system that uses a camera service (a given usable library) to help users get images back.

## Functional requirements:

1. Asynchronous Image Capture request from clients -> basically use a client request and start the async process to send back the image
2. Same above request should be able to register callbacks to acknowledge success / failures and once acknowledge, let the clients know as well with either the image or the exception message why the image could not be sent
3. Prioritization of requests based on urgency (a parameter sent to us via clients / users)
4. Concurrent request handling -> basically simultaneous requests should be handled

## Non Functional requirements:

1. Performance: Highly performant in terms specially of latency, really needed for high priority requests. Either way, the latency should be almost real time.
2. Reliability: There shouldn’t be failures as much as possible and Availability should be high
3. Scalability: concurrent handling of requests should be there and the system should be able to adapt with growing scale of incoming requests

For now lets start with 10k requests per second

1. Extensibility and Maintainability: It should have flexible and reusable components with can adapt to future requirement changes as needed and have easy debugging and monitoring abilities.

## Assumptions:

1. Lets let it be cameraModule and its preexisting APIs that can be used be:

1. **cameraModule.startCaptureAsync()** -> to submit request and start with listeners
2. **cameraModule.registerSuccessCallback(callbackSuccessFunction)** -> to listen to events given back incase of success
3. **cameraModule.registerFailureCallback(callbackFailureFunction)** -> to listen to events given back incase of failure

2. CameraModule is a service which has efficient APIs mentioned above. APIs are good enough to handle scale and process them quick

3. Urgency will be part of user / client request only and cameraModule doesn’t have a mechanism of handling that and we need to handle this in our software

4. Although there is a mention of high priority and low priority requests, we can have a dynamic range of requests, three categories to start with:

1. 0 -> At this level its like a self driving car system which needs things very very fast
2. 1 -> a normal day to day usage where client expects fast response like use case of google image search but it is not life threatening
3. 2 -> The client has no urgency, suppose a tourist sends a request asking for an image for any template in some defined area
4. These above priorities should be extendable

5. We are not concerned much with the request data from the client yet. For example maybe user give longitude, latitude or some location name or some use case name and CameraModule can identify what image(s) to send based on these

7. We have not considered user signup process but this will contain user permissions based on type of client. Permissions to determine for example if client can make priority / urgency 0 requests or not.

8. THere is no need to keep an audit / history as of now

9. CameraSystem has different region support in terms of CDN (continous delivery network)

# HLD

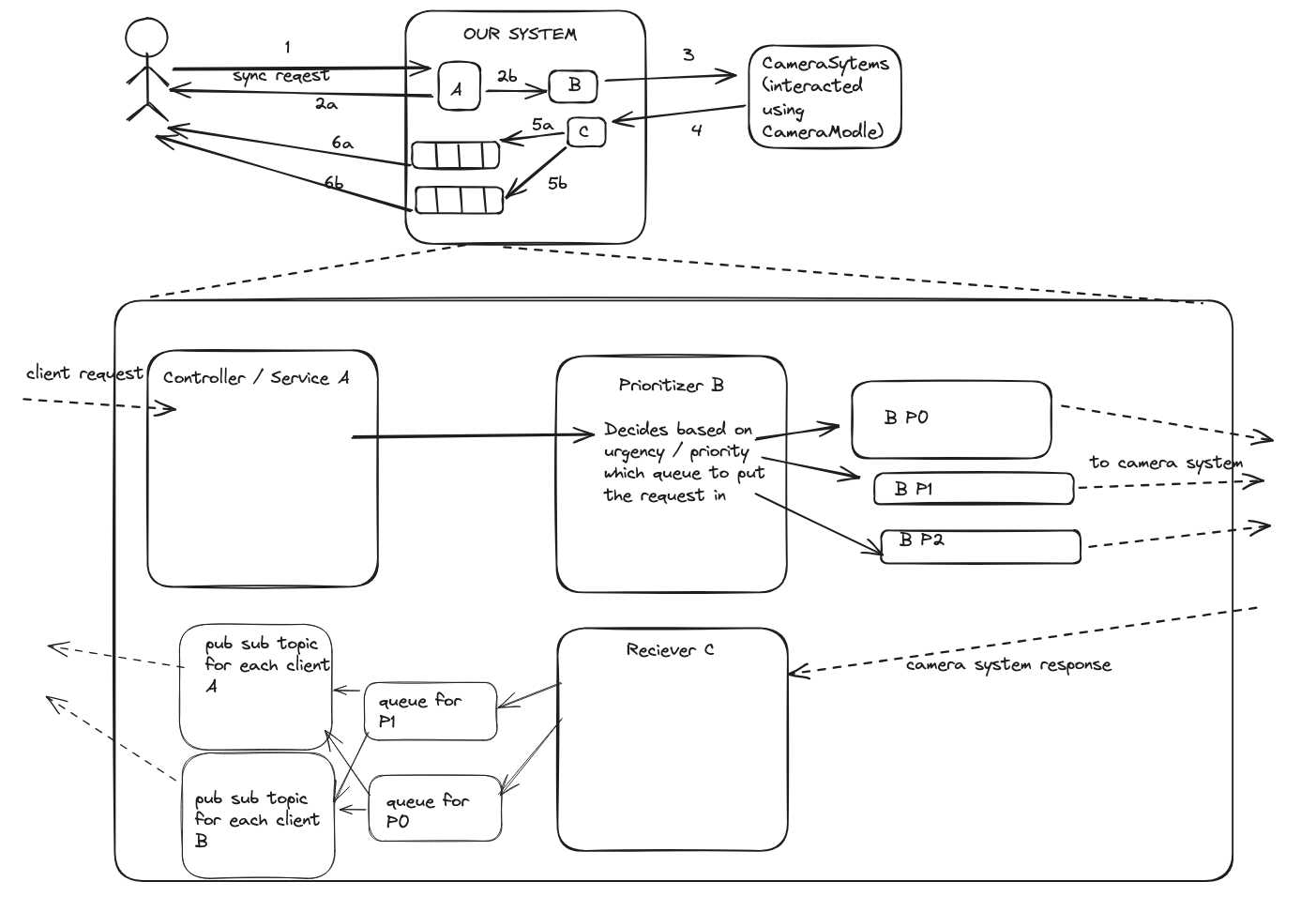
The basic flow as per arrow markings:

1. Client sends in request
2. a. Our system component A (the service layer) responds back with acknowledgment of request processing

b. Our system further component B send the request to another component B to setup the priority and send the request to external Camera System using the camera Module

1. Component B sends request to Camera System
2. Asynchronous waiting and send back of data via the camera System
3. Success / failure responses with images / exception messages are put into topic via queue to send back the image data / exception data
4. Client reads from these queues

Note here, A is the controller and service and B is the priority system and C is the service layer again to handle responses.

Note 5a and 5b are examples of two different delivery routes based on the priority of the requests.

## Technology choices:

1. Our service needs to be deployed in a auto scalable infra. Maybe Ec2 clusters with load balancers or to start with we can use fully managed systems like AWS Elastic Beanstalk or GCP GAE.
2. Our queues need not be process requests in order. Best would be Kafka because they manage things on their own. If there are budget constrains maybe something like SQS works best. But as per our system’s requirements Kakfa would be the best choice with even order guarantee (if needed) as it provides very high throughput in terms of latency
3. As we extend and scale, pub sub topics should be client owned so we should have a module to send data to any topic. But to start with maybe we can have pub subs at our end. Something like GCP pub sub or AWS SNS would work. Although SNS may require additional overhead of maintaining image in some object storage like AWS S3 or GCP GCS.

# LLD

1. Sample request from client:

CaptureRequest loggingRequest = new CaptureRequest(

2, // P2 - Lowest priority

Map.of(

"longitude", "37.7749",

"latitude", "-122.4194",

"purpose", "routine-logging",

"cameraId", "rear-camera"

)

);

cameraSystem.submitCaptureRequest(loggingRequest);

2. Controller API handler A

CaptureResponse submitCaptureRequest(int priority, SuccessCallback successCallback, FailureCallback failureCallback, Map<String, String> metadata);

CaptureResponse will have fields to acknowledge back to client saying we have successfully registered the request or not, it should also have a request id

3. Prioritizer B: It ensures that higher-priority requests (e.g., P0) are processed before lower-priority ones (e.g., P2). The prioritizer dynamically manages multiple priority queues, processes requests based on their urgency, and integrates seamlessly with the Camera system using camera module and uses ExecutorService (java example, there would be others for other languages) to achieve this. Note how PriorityBlockingQueue (java example) helps us handle concurrent requests.

public class Prioritizer {

private final Map<Integer, PriorityBlockingQueue<CaptureRequest>> priorityQueues;

private final ExecutorService executorService;

private final CameraModule cameraModule;

public Prioritizer(CameraModule cameraModule, Map<Integer, Integer> priorityLevels) {

this.cameraModule = cameraModule;

this.executorService = Executors.newCachedThreadPool();

// Initialize priority queues based on provided priority levels

this.priorityQueues = new ConcurrentHashMap<>();

for (Map.Entry<Integer, Integer> entry : priorityLevels.entrySet()) {

int priority = entry.getKey();

int initialCapacity = entry.getValue();

this.priorityQueues.put(priority, new PriorityBlockingQueue<>(initialCapacity, Comparator.comparingInt(CaptureRequest::getPriority).reversed()));

}

}

public CaptureResponse prioritizeRequest(CaptureRequest request) {

PriorityBlockingQueue<CaptureRequest> queue = priorityQueues.get(request.getPriority());

return new CaptureResponse(false, "Invalid priority level: " + request.getPriority(), null);

// Add the request to the appropriate queue, note request has requestid

queue.offer(request);

// Submit the processing task to the executor service

executorService.submit(this::processRequests);

return new CaptureResponse(true, "Request registered successfully.", requestId);

}

private void processRequests() {

for (int priority : priorityQueues.keySet()) {

PriorityBlockingQueue<CaptureRequest> queue = priorityQueues.get(priority);

CaptureRequest request = queue.poll();

if (request != null) {

// Register the callbacks with the CameraModule

cameraModule.registerSuccessCallback(request.getSuccessCallback());

cameraModule.registerFailureCallback(request.getFailureCallback());

// Start the asynchronous capture

cameraModule.startCaptureAsync();

break; // Process one request at a time

}

}

}

public void addPriorityLevel(int priority, int initialCapacity) {

priorityQueues.put(priority, new PriorityBlockingQueue<>(initialCapacity, Comparator.comparingInt(CaptureRequest::getPriority).reversed()));

}

public void removePriorityLevel(int priority) {

priorityQueues.remove(priority);

}

}

Note how the reqievers / callback listeners are also added here

4. Receiver C

Now the receiver knows at events for ‘cameraModule.registerSuccessCallback(request.getSuccessCallback());’

And similarly for failure and then based on their original priority sends back to 3 different queues

Which based on the client id can route to client’s pub sub topic

(Extendibility: as clients grow it’ll be hard to maintain pub sub topics so maybe they should be moved to client’s side later as the software’s scale grows)

And we will need a module D to route from these queues to the topic used.

# Other areas of consideration

1. **Monitoring:** must be taken into consideration and to start with we should ensure proper logging, exception handling and use the cloud service provide’s inherent monitoring system and add custom metrics to monitor. Areas to monitor are:
   1. Request counts and latencies and success rate
   2. Prioritizer component’s thread health
   3. Queue depth, adds, completions and failures
   4. Success / failure rate from the output received and send to camer system
   5. Other infra like CPU, memory, autoscaling, etc
2. **PubSub topics:** Although mentioned that these will behandled by us, it actually makes more sense to have the client set these up at their end or with growing number of customers, we wont be able to handle too many topics
3. **If new priority / use case comes in**: We can easily accommodate new queues to handle such cases
4. **Timeouts and retries**: Note that receiver would need some kind of timeout to tell the client to resend request. Note that this should be different based on the priority / urgency. These is no mention of this in the above design. In appropriate cases, retires should also be there.
5. **Database:** We need a fast database. Maybe something like a noSQL GCP datasore or AWS dynamo db works well. Or maybe something else. This will be needed to maintain request id and other info. This will be also needed to validate if a client id falls in a client type that allows it to make P0 requests or not. This will help control throttling as well
6. **CDN** continuous delivery network**:** Each of the services should be at different CDNs based on the clients location for distributed processing to help with latencies. Note that backup management should also be handled incase on service is down
7. **Security:**
   1. We need rate limits to throttle client requests per client. These can be part of plan they use as well. Like client A needs higher rates than client B
   2. Authentication module using the Database mentioned in point 5