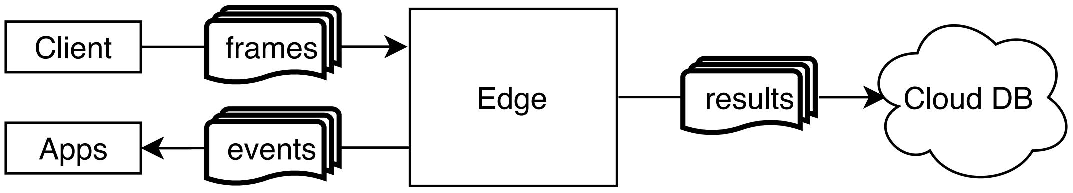

Figure 1: IoT object detection system overview

Abstract

This paper is a Master's project report that discusses the design, implementation, and performance evaluation of a prototype IoT Edge Object Detection system. The code is public on GitHub. [1]

1 Introduction

Many modern Internet of Things (IoT) applications rely on the efficient, real-time processing of video streams. Applications include Virtual and Augmented Reality (V/AR), video surveillance, self-driving cars, and more. A challenge that faces these applications is the lack of real-time image processing and storage in the cloud, as a result of high wide area latency and bandwidth costs. We introduce a prototype IoT Edge Object Detection (IoT detection) system that utilizes the edge and the cloud for different purposes. The edge is utilized for real-time object detection of a video stream, storage and access to those detection results, and filtering events on the detection of objects of interest. The cloud, on the other hand, is utilized for archival purposes. The IoT detection system was motivated by IoT edge data management and processing at the edge [6, 15, 18], and IoT application use cases such as home automation and surveillance.

In this report, we discuss the IoT detection system edge services and components (section 2), the system implementation (section 3), performance evaluation (section 4), possible use cases (section 5), future work and improvements (section 6), and conclusion (section 7).

2 System Overview

The IoT Edge object detection system consists of an edge node, client node, cloud node, and a set of applications, depicted in Figure 1. The client node is the source of image frames and is either on the same wireless network or the same node as the edge, whereas the cloud is a database that is hosted in a public or private cloud provider (for example, Amazon AWS). The application is independent of the system and can be located anywhere. The edge node contains a pre-trained model for objection detection, and a data store for the detection results, which are image frames along with metadata such as labels and bounding boxes.

The edge node periodically archives image frames to the cloud node while retaining the detection result metadata, serving as a cache and metadata log for the detection results. The edge node has provides two application services: the event-on-detect service and the event query service.

The event-on-detect service provides instantaneous event filtering, while the event query service provides event filtering in time. An event can be thought of as a structure that wraps up a name, image frame, and detection result metadata. For example, a MoreThanOnePerson event could be generated for when more than one person is detected, where the detection of the event would result in an the event that contains the image frame where more than one person is detected, the name of the event, and the detection result metadata. Then any detected MoreThanOnePerson events could be streamed instantaneously to the application or queried over time.

The system is geared towards IoT application use cases, such as smart home automation and surveillance [18] and face mask detection.

3 Edge Services and Components

The edge node exposes three remote procedure call (RPC) services: the uploader service, the event-on-detect service service, and the event query service. The uploader service is used solely by the data source client node, while the event-on-detect and event query services are used by the application. Data flows from the client to the edge via the uploader service, and is concurrently pipelined through the different components in the edge node. This section describes the services, their interfaces, and their component-wise implementations in the edge node.

[1] https://github.com/bosdhill/iot_detect_2020

3.1 Uploader Service

The uploader service provides a simple interface for the client node to upload image frames as a stream to the edge. The service provides a client-to-server streaming UploadImageFrames RPC that accepts a stream of *ImageFrame* messages, or image frames. The edge returns a success or failure message to the client node if the upload stream ends.

3.2 Event-on-Detect Service

The event-on-detect service provides two actions for the application: (1) registration of real time event query filters with edge and (2) receiving a stream of events that satisfy the provided filters. The service accomplishes (1) through an interface for an application to fetch the edge's supported labels with the GetLabels RPC and register its real time event query filters on the edge with the RegisterEventQuery-Filters RPC. The service accomplishes (2) by streaming events to the application with the SendEvents RPC. The flow the application would follow would be to call the GetLabels RPC in order to fetch the object labels that the object detection model on the edge supports. With these object labels, the application generates a *EventQueryFilters* message, or event query filters, which are used in the event-on-detect component on the edge to filter detection results in real time. After the application generates the message, it registers its newly created event query filters by calling the RegisterEventQueryFilters RPC. Finally, the application can call the server-to-client streaming GetEvents RPC to receive a stream of real time *Events* messages, or events, filtered in real time by the event query filters the application provided.

3.3 Event Query Service

The event query service provides the application an interface to issue arbitrary queries in time on the detection results stored in the edge. The service accomplishes this with the same GetLabels RPC described above and a Find RPC that accepts event query filters from the application. The flow the application would follow would be to call the GetLabels RPC in order to fetch the object labels that the object detection model on the edge supports. With the knowledge of the object labels, the application would generate an event query filter with a time paramater and then call the Find RPC and receive the events filtered over time as a response.

```
// DetectionResult represents the result of the object detection model
message DetectionResult {
        // Whether or not the result has any objects
        bool empty = 1;

        // The time of detection
        int64 detection_time = 2;

        // A map of labels to number detected
        map<string, int32> label_number = 3;

        // A list of labels
        repeated string labels = 4;

        // The matrix representation of the image frame
        ImageFrame img = 5;

        // A map from label to its bounding box
        map<string, BoundingBoxes> label_boxes = 6;
}

message BoundingBox {
    int32 top_left_x = 1;
    int32 top_left_y = 2;
    int32 bottom_right_x = 3;
    int32 bottom_right_y = 4;
    float confidence = 5;
}
```

Figure 2: The DetectionResult message

3.4 Object Detection Component

The image frames streamed to the edge from the client are passed to the object detection queue. The object detection component pulls an image frame from the object detection queue and transforms the image frame into a blob used as input to the object detection model. The object detection model generates the object detection metadata which includes labels, bounding boxes, detection time, confidence per detected object, and the number of objects per label. The detection result metadata is then composed with the image frame to create a *DetectionResult* message, or detection result, as shown in Figure 2. The detection result is pipelined to both the data store component's worker queue and the event-on-detect component's worker queue.

3.5 Event-on-Detect Component

The event-on-detect component is the implementation of the event-on-detect service. When an application registers its event query filters with the RegisterEventQuery-Filters RPC exposed by the event-on-detect service, the component stores the set of event query filters for use in real time event query filtering.

When the event-on-detect component receives a detection result from the object detection component through its worker queue, it queries each event query filter against the detection result. For any event query filter that is satisfied, the event-on-detect component generates the corresponding event by composing the event name with the metadata and image frame. The event-on-detect component then returns the set of event messages to the application using the server-to-client streaming RPC GetEvents.

3.6 Data Store Component

The data store component's worker queue pulls detection result from the object detection component. The detection results are then inserted into the local database. The data store component also provides a Find, Delete, and Update interface for searching for, deleting, and updating the detection results stored in the local database. The interface is used by the event query component and the cloud upload component.

3.7 Event Query Component

The event query component is the implementation of the event query service, which exposes a server-client streaming Find rpc to applications. The Find rpc functions similar to Find in the data store interface, where instead of a Query Filter being passed, a EventQuery Filter message is sent by the application. The EventQuery Filter message contains a serialized database query named Query Filter, which is then passed to the data store component's Find method. Any detection results returned are then composed with *EventQuery Filters* to be converted into *Events* and returned to the application.

3.8 Cloud Upload Component

The cloud upload component is used for archiving data from the edge data store by using the data store component's interface to access results and a connection to a cloud database to upload data. It operates in two phases: (1) periodically uploading data and then leaving only the metadata in the local database and (2) removing stale metadata. The edge data store can be thought of as a log, where before the first time the cloud upload component performs any actions, all of the log's detection result records contain both image frames and metadata.

In (1), the cloud upload component uses the data store component's interface to retrieve a *batchSize* number of detection results after $uploadTTL$ seconds, and then uploads them to the cloud database. It then removes the image frame fields from the detection result. This incrementally transforms the log of detection results into a metadata log at rate of *batchSize* detection results over $uploadTTL +$ upload latency seconds. In (2), after $deleteTTL$ seconds the cloud component deletes any metadata in the data store, which completes the archival for that period.

4 System Implementation

The IoT detection system was first written in Python 3, but by observing the need for concurrent processing, the system was converted entirely to Golang. Golang provides easy to write concurrency through the use of goroutines, which are lightweight threads managed by the Go runtime that can communicate through channels, which can be thought of as FIFO queues [1]. This made concurrent processing both easy to write and reason about.

The gRPC framework was chosen for the external communication interface of the system. The gRPC framework uses the Protobuf (protocol buffers) messaging format, which is a highly-packed, highly-efficient messaging format for serializing structured data [5]. The use of gRPC suited the Golang implementation well since it is a strongly typed language that uses structs.

In this section, we will examine the implementation processes and implementation details of selected parts of the system, such as the data store component, event querying and real time event filtering, and the cloud upload component.

4.1 Data Store Component

The databases that were experimented with in edge data store component were memdb, sqlite 3, and MongoDB.

4.1.1 Memdb

Memdb was tested using the go-memdb package, which implements a simple in-memory database built on immutable radix trees [7]. Single board computers lack memory and the detection results grow in size quickly, so go-memdb ended up running out of memory within a few minutes of testing. Implementing custom indexes for non primitive fields to retrieve detection results in go-memdb also proved to be nontrivial. Go-memdb's reliance on memory would mean uploading to the cloud too frequently, and so system performance would decrease dramatically. The issues of restricted memory and complex index implementations proved go-memdb to be a bad fit for implementing the data store component.

4.1.2 SQLite

The next database that was tested was SQLite 3 which is a small, fast, self-contained, high-reliability, full-featured, SQL database [17]. The Golang interface that was tested was the go-sqlite3 package [13]. The SQLite 3 database was mainly considered for its serverless architecture and documentation and support for Golang. The schema was designed as 3 tables: the image table, the bounding box table, and the labels table. Implementing the data store interface using sql queries on the separate tables proved to be complicated, and providing support for arbitrary queries on detection results would have proved tedious.

4.1.3 MongoDB

The last database implementation that was tested is the one currently in use, which is MongoDB. MongoDB is popular a document database designed for ease of development and scaling, with the document data structure composed of field and value pairs [10]. MongoDB documents are similar to JSON objects, with an example detection result document in Figure 4. The package used for the MongoDB driver was mongo-go-driver [14]. MongoDB allowed for flexible storage and querying, and used BSON for the queries in the Golang implementation. This allowed

```
{
  "_id": {
    "$oid": "5fc71a6fbbcb66a4b1761acc"
  },
  "empty": false,
  "detectiontime": {
    "$numberLong": "1606883951485141000"
  },
  "labelnumber": {
    "bus": 1,
  },
  "labels": [
    "bus",
  ],
  "img": {
    "image": {
      "$binary": "BAwLEBgXGSEgFx8eFx8eFx8eFx8eFx8eFh4dF...",
      "$type": "0"
    },
    "rows": 480,
    "cols": 854,
    "type": 16
  },
  "labelboxes": {
    "bus": {
      "labelboxes": [
        {
          "topleftx": 143,
          "toplefty": 114,
          "bottomrightx": 481,
          "bottomrighty": 367,
          "confidence": 0.4919454753398895
        }
      ]
    },
  }
}
```

Figure 3: An example MongoDB Detection Result document

for the serialization of the MongoDB query filters as BSON. The mongo-go-driver BSON package includes a simple marshalling and unmarshalling interface, and so the query filter field in the *EventQuery Filter* protobuf message comprises of a serialized MongoDB BSON query filter. The simple BSON serialization and query filter construction allowed BSON query filters to be used in the local data store interface, the event-on-detect component, and the event query component. MongoDB also provides an easy to set up cloud database service that can be accessed with the mongo-go-driver, named MongoDB Atlas [11]. MongoDB Atlas simplified the cloud upload implementation since it did not require setting up a separate cloud server. The only shortcoming of the mongo-go-driver package was that the actual construction of query filters was minimally documented as the driver is relatively new, and so required extra experimentation and combing through documentation and source code.

```
message EventQueryFilter {
    // Used to query detection results stored within the last n seconds.
    // This is field is ignored in real time event filtration.
    int64 query_seconds = 1;

    // The name of the event, which is used when returning the event to the application.
    string name = 2;

    // MongoDB bson query
    bytes filter = 3;

    // flags determine the granularity of data returned in an Action
    uint32 flags = 4;

    // Flags enum provides for the bitwise field flags
    enum Flags {
        // Dummy flag
        FLAGS_UNSPECIFIED = 0;

        // Return only metadata
        METADATA = 1;

        // Return bounding boxes
        BOXES = 2;

        // Return jpg image
        IMAGE = 4;

        // Return annotated jpg image image
        ANNOTATED = 8;

        // Return confidence
        CONFIDENCE = 16;

        // Return empty results
        EMPTY = 32;
    }
}
```

Figure 4: The EventQuery Filter protobuf message

4.2 Event Querying and Real Time Event Filtering

The event query and event-on-detect components and service interfaces use the *EventQueryFilter* protobuf message in Figure 4 above. The reasoning was to create a single message that could be used for both interfaces. The message includes a *query_seconds* field used by the event query component to query over time, the *name* field for the event name used in the *Event* message, the *filter* field which is

```
{ $or: [ { <expression1> }, { <expression2> }, ... , { <expressionN> } ] }

{ $and: [ { <expression1> }, { <expression2> } , ... , { <expressionN> } ] }

{ <field>: { $all: [ <value1> , <value2> ... ] } }
```

Figure 5: The supported event-on-detect component MongoDB query filters

the serialized MongoDB BSON query filter, and *flags* used to configure the fields to be returned in the *Event*, such as to configure the event to contain only metadata.

The event-on-detect component's event query filter's *filter* field supports the MongoDB $or, $and, and $all query filters as shown in Figure 5. The $or and $and query filters are used to compose comparison query filters, whereas the $all query filter is used to check if the labels provided in the *filter* is a subset of the *labels* array of any detection results. Some example query filters are shown in Figure 6.

The event-on-detect component relies on a real time event query filter component to deserialize and parse the *filter* provided by the event query filter into a Golang struct, which is stored in memory. The struct is used to compare filters against the detection result the object detection component pipelines to the event-on-detect component. Each particular query that needed to to be mapped to a native Golang struct, which is why there's only a limited number of MongoDB query filters that are supported.

On the other hand, the event query component supports arbitrary MongoDB query filters, which are only modified query by time before passing the *filter* to the mongo-go-driver package's Find interface. The event query component creates a new query filter that aggregates detection results stored in the last *query_seconds* before filtering by the provided *filter*.

4.3 Cloud Upload

The cloud upload component consists of two cron jobs: the cloud upload job and the local delete job. The cloud upload job is scheduled to run every *uploadTTL* seconds and the local delete job is scheduled to run every *deleteTTL* seconds with gocron [4]. The *uploadTTL* parameter defines how long to maintain image frames on the edge, or the metadata log compression frequency, since the image frames take up the majority of space. The deleteTTL parameter defines the length of time to store metadata on the edge, or metadata log expiration frequency, which should be

```
{
  "$and": [
    {
      "labelnumber.person": {
        "$gte": 1
      }
    },
    {
      "labelnumber.bus": {
        "$gte": 1
      }
    }
  ]
}
```

```
{
    "labels": {
      "$all": [
        "person",
         "bus"
      ]
    }
}
```

Figure 6: Event-on-detect MongoDB query filter examples

a much longer period than *deleteTTL*. The cloud upload and local delete jobs share a lock to remain mutually exclusive.

The cloud upload job consists of two phases which are (1) retrieve the oldest *batchSize* number of detection results (2) upload the retrieved results to the remote MongoDB instance (3) remove the image frame field for the uploaded detection results in the local MongoDB instance. The last phase of the cloud upload job ensures only the metadata remains on the edge, and reduces the size of the local Mongo database, creating an incrementally growing metadata log. The local delete job consists of a phase, which is to delete all the detection results that do not have an image frame. The empty image frame field of a detection result acts as a marker for the delete job to locate the expired metadata.

4.4 Object Detection

The object detection component was implemented using gocv, which provides Golang bindings for the OpenCV 4 computer vision library [9]. OpenCV 4 was chosen as it had many examples and accessible object detection model implementations for Tiny YOLO v2 [16] in Golang. The Tiny YOLO v2 model that used was the Caffe framework implementation. This was chosen mainly for examples that were found using it, such as the one adapted for use in object detection component using the darknet region layer in go [12].

The object detection component receives image frames from the client connection component, which are converted into gocv.Mat matrices and then reshaped into

416 by 416 pixel blobs. The blobs are input into the Tiny YOLO v2 model to generate the detection result metadata. The gocv.Mat is not written to disk at any time during the object detection process. The resulting blob is then reshaped to its original size and then is simultaneously pipelined to the event-on-detect and data store components.

5 Performance Evaluation

The IoT detection system was tested with the client node, edge node, and application running locally on a 2017 Macbook Pro with a 2.3 GHz Dual-Core Intel Core i5 processor and 8 GB RAM, which eliminates any network latency other than for the cloud upload component. The cloud upload component was connected to a remote MongoDB instance on MongoDB Atlas, which was hosted in AWS in Oregon us-west-2. The system was evaluated by using a looped mp4 video stream, which contains a large number of person, truck, and bus objects per image frame. A test application was written to solely measure the throughput of the event-on-detect and edge query services response messages, and the object detection component's image frame processing throughput.

5.1 Testing Configuration

The testing script was designed as a simple stress test in order to measure the performance of the real time and query over time aspects of the system. Each test in the testing script was configured to run 10 times with each run lasting 60 seconds with the database empty before the first test run. The application in both tests uses the AtLeastOnePerson event query filter, which is satisfied if there is at least one person in the image frame. This filter was chosen because it would be satisfied often in the mp4 stream of image frames, and so would generate many events.

In the first test, the application was configured to send an event query request every 10 seconds with the AtLeastOnePerson filter and to query results stored in the last 60 seconds. The same application was also registered to receive a stream of events that satisfy the AtLeastOnePerson filter through the event-on-detect service.

In the second test, the application only tests the real time aspect of the system by registering to receive a stream of events that satisfy the AtLeastOnePerson filter.

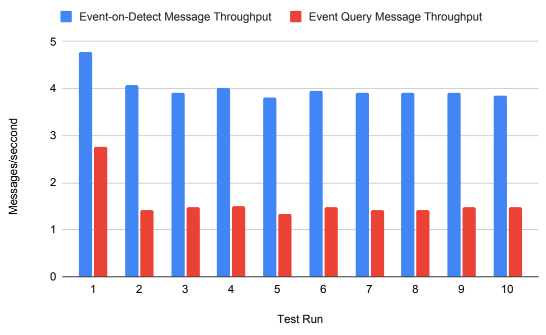

Figure 7: Application message throughput

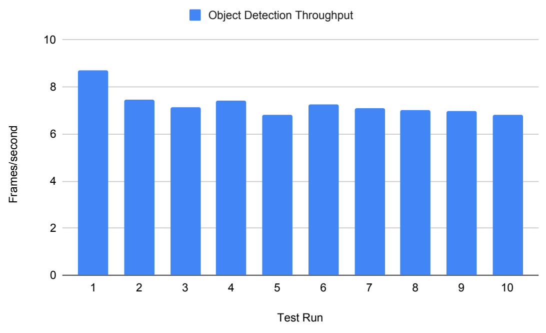

Figure 8: Object detection throughput

The cloud upload component was enabled in both tests but was not specifically configured or monitored during the tests.

5.2 Event-on-Detect and Event Query

This test combines the measurement of system throughput when querying events over time as well as instantaneous event filtering. The application was configured to issue a Find request every 100ms, or 10 requests a second, and queried events detected in the last 60 seconds. The average throughput for test application and the object detection component are shown Figures 7 and 8, respectively. From Figure 7, the first test run with the empty database shows the highest average throughput for both services with 4.77 messages per second and 2.77 messages per second for the event-on-detect and event query service, respectively.

The peak average message throughput can be attributed to the initially empty database, which limits the number of detection results read in the Find event query service request. As the test runs continue, the average message throughput decreases and remains above 3.9 messages per second for the event-on-detect service,

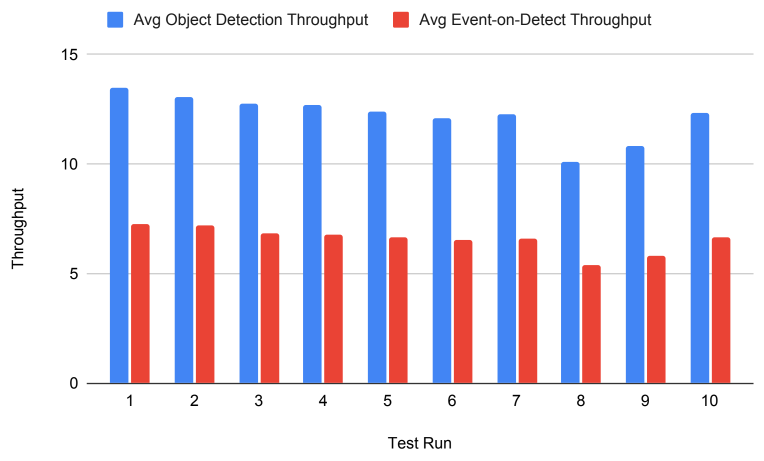

Figure 9: Object Detection and Event-on-Detect throughput

and around 1.33 messages per second for event query service. The reduced throughput is due to the heavy concurrent reads and writes to the database, as concurrent reads and writes on a MongoDB collection must share a reader-writer lock, so heavy concurrent writes and reads will slow down system performance [3]. The throughput remained around 3.9 frames per second as the tests continued. Since the event query service is for querying in time, the message sizes are much greater than the number of events returned in the event on detect GetEvents stream. For example, the total number of events returned to the event query application was 21364 vs 244 for the event on detect application in test 2.

From Figure 8, the object detection throughput is also at its peak average of 8.72 frames per second when the database is empty during the first test run, and remains over 6.3 fps thereafter. This corresponds to the high message throughput recorded on the application in the first run. Since image frames are pipelined from the object detection component to the data store and event-on-detect components, the concurrent reads and writes on the database can reduce the object detection component throughput.

5.3 Event-on-Detect

The event-on-detect service was tested alone to determine the throughput of the system when only streaming events in real time to the application. From Figure 9, the peak average throughput of 13.47 frames per second and 7.28 response messages per second occur during the first test for the object detection component throughput and event-on-detect service, respectively. The peak average throughput of both the object detection component and event-on-detect service are over 1.5 times their values when compared to the first test. The average throughput of the object detection component remains over 10 frames per second on average, with the average throughput of the event-on-detect service remaining over 5 messages per second, so the change in throughput in subsequent runs is not as dramatic as in the first test.

The higher average throughput can be attributed to the database handling mainly writes, with the only other writes and reads coming from the cloud upload component albeit intermittently. Although the low database reads remove some performance bottlenecks, ultimately the object detection model Tiny YOLO v2 will become the bottleneck.

5.4 Summary

The first test shows that the database read-write concurrency can affect the system performance when there are heavy reads from the application issuing Find requests through the event query service. The second test shows that the system can reach consistent performance when its used purely for streaming events in real time, but is eventually limited by the performance of the object detection model. The tests also do not account for network latency and bandwidth costs, as well as the compute and storage limitations of commodity single board computers, such as the Raspberry Pi 4 and Nvidia Jetson Nano.

6 Use Cases

This section briefly explores some use cases for the IoT detection system, such as an infrastructure for an IoT security and home monitoring system, and a face mask detection application.

6.1 Security Monitoring System

For an IoT security and home monitoring system, the client node and edge node can be on the same edge device, or there can be many client nodes that act as peripheral devices and upload image frames to the edge device. The client and edge nodes would be on the same wireless local network. The application can utilize the event-on-detect and event query service in a variety of ways, such as a real time notification system.

For use as a real time notification system, the user can define event query filters for custom events, such as a MoreThanOneCar event, where a camera or client node is placed in front of the driveway and streams image frames to the edge device. When more than one car in the driveway is detected, the edge node can stream the metadata of the event using the server-side SendEvent streaming rpc to the user's application directly which can either be a mobile or web browser application. For example, Android supports server-side streaming rpcs [2]. Since the application is assumed to be near the edge, the latency between the application and edge should be much smaller than if the application were receiving updates from a similar cloud implementation.

If the user desired to query past MoreThanOneCar events, the application can send a request to the edge query service to query since the time of the notification, and the user can receive the image frames as a response. The cloud database would be used for long term storage of the surveillance and monitoring detection results, or the user could opt for purely local storage to avoid cloud storage costs.

6.2 Face Mask Detection

A fask mask detection application could be implemented with a face mask detection model on an NVIDIA Jetson Nano, where the model was able to analyze a frame in less than 1/10th of a second, giving it an overall frame rate of around 12fps [8]. Using this model on the edge node, the face mask application can be geared towards detecting masks on the faces of employees, such as warehouse workers. The business would desire real time event alerts since a business owner would want to know if employees are entering a building without a face mask.

The client nodes can be positioned in areas where there is high foot traffic, high ceilings, and light, such as the entrance to the warehouse. With the client and edge nodes following a similar topology in the security monitoring system use case, the cameras facing the entrance will stream their image frames to the edge node. The application could be a web browser used as a real time face mask monitoring dashboard for the warehouse, where the operator would receive NoMask events and notifications in real time. The cloud database could also be used for long term storage, and be kept as a persistent log for proof of face mask violations.

7 Future Work and Improvements

The IoT detection system does not currently support queries for image frames that have been evicted to the cloud. The system could benefit from a cloud server implementation that could handle cloud uploads as well as application queries that are redirected from the edge node. This would enable the system to provide the feature of queries unlimited in time.

As for testing, the system would need to be configured and tested on an edge device such as an NVIDIA Jetson Nano with a CUDA-enabled graphics processing unit, and compared against a similarly priced single board computer without a dedicated GPU, such as the Raspberry Pi 4. This can also show that the system could gain performance when running a object detection model that supports a CUDA-enabled GPU. This type of test would also gain more insight into how the system performs when the nodes are connected over a wireless network.

8 Conclusion

This report discussed the design, implementation, and performance evaluation of a prototype IoT Edge Object Detection system. The data store, event-on-detect, event query, cloud upload, and object detection components and their interactions were described, as well as the interfaces of overlaying services that are exposed by the edge. The local performance stress testing identified the system performance bottlenecks under specific loads. Finally, two potential use cases of the system were proposed, as well as possible changes that could be made to the design and testing of the system.

References

[1] *A Tour of Go*. URL: https://tour.golang.org/concurrency/1 (visited on 11/23/2020).

[2] *Basics tutorial*. gRPC. URL: https://grpc.io/docs/platforms/android/java/basics/ (visited on 12/07/2020).

[3] *FAQ: Concurrency — MongoDB Manual*. URL: https://docs.mongodb.com/manual/faq/concurrency (visited on 12/07/2020).

[4] *go-co-op/gocron*. original-date: 2020-03-20T15:33:05Z. Dec. 6, 2020. URL: https://github.com/go-co-op/gocron (visited on 12/06/2020).

[5] *gRPC vs. REST: How Does gRPC Compare with Traditional REST APIs?* API Blog: Everything You Need to Know. Section: REST API. Sept. 4, 2020. URL: https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis/ (visited on 12/08/2020).

[6] Yundi Guo et al. "Distributed and Efficient Object Detection via Interactions Among Devices, Edge, and Cloud". In: *IEEE Transactions on Multimedia* 21.11 (Nov. 2019). Conference Name: IEEE Transactions on Multimedia, pp. 2903–2915. ISSN: 1941-0077. DOI: 10.1109/TMM.2019.2912703.

[7] *hashicorp/go-memdb*. original-date: 2015-06-16T22:54:29Z. Dec. 1, 2020. URL: https://github.com/hashicorp/go-memdb (visited on 12/02/2020).

[8] *How to Build a Face Mask Detector With a Jetson Nano 2GB and AlwaysAI - ExtremeTech*. URL: https://www.extremetech.com/computing/317028-how-to-build-a-face-mask-detector-with-a-jetson-nano-2gb-and-alwaysai (visited on 12/08/2020).

[9] *hybridgroup/gocv*. original-date: 2017-09-18T21:54:17Z. Dec. 1, 2020. URL: https://github.com/hybridgroup/gocv (visited on 12/02/2020).

[10] *Introduction to MongoDB — MongoDB Manual*. URL: https://docs.mongodb.com/manual/introduction (visited on 12/02/2020).

[11] *Managed MongoDB Hosting — Database-as-a-Service*. MongoDB. URL: https://www.mongodb.com/cloud/atlas (visited on 12/02/2020).

[12] Denny Mattern. *dymat/GOLOv2*. original-date: 2018-04-12T09:12:58Z. Nov. 2, 2020. URL: https://github.com/dymat/GOLOv2 (visited on 12/06/2020).

[13] mattn. *mattn/go-sqlite3*. original-date: 2011-11-11T12:36:50Z. Dec. 2, 2020. URL: https://github.com/mattn/go-sqlite3 (visited on 12/02/2020).

[14] *mongodb/mongo-go-driver*. original-date: 2017-02-08T17:18:02Z. Dec. 2, 2020. URL: https://github.com/mongodb/mongo-go-driver (visited on 12/02/2020).

[15] Dan O'Keeffe, Theodoros Salonidis, and Peter Pietzuch. *Frontier: resilient edge processing for the internet of things*. June 2018. URL: https://doi.org/10.14778/3231751.3231767 (visited on 04/01/2020).

[16] Joseph Redmon and Ali Farhadi. "YOLO9000: Better, Faster, Stronger". In: *arXiv:1612.08242 [cs]* (Dec. 2016). arXiv: 1612.08242. URL: http://arxiv.org/abs/1612.08242 (visited on 04/01/2020).

[17] *SQLite Home Page*. URL: https://www.sqlite.org/index.html (visited on 12/02/2020).

[18] Rich Wolski et al. "CSPOT: portable, multi-scale functions-as-a-service for IoT". en. In: *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing - SEC '19*. Arlington, Virginia: ACM Press, 2019, pp. 236–249. ISBN: 978-1-4503-6733-2. DOI: 10.1145/3318216.3363314. URL: http://dl.acm.org/citation.cfm?doid=3318216.3363314 (visited on 04/01/2020).