
TZATZI Captura y gestión de datos en tiempo real para el transporte público



Desarrolladores e impulsores del proyecto

El presente proyecto es la piedra angular y requisito indispensable para acreditar el Diplomado en Internet of Things, que es una colaboración entre [Samsung Innovation Campus](#) con la [Universidad Autónoma Metropolitana](#) y [Código IoT](#).

Así mismo este proyecto Capstone, es un esfuerzo colaborativo entre los alumnos *Erick Huitziuit Morales García*, *Alfredo García Piñal* y *Alejandro Flores Jacobo* de la Universidad Autónoma Metropolitana, en donde los primeros dos pertenecen a la unidad académica Cuajimalpa y el tercero a la unidad Lerma.

Agradecemos a todas las instituciones y sus iniciativas por permitirnos cursar este diplomado con una beca totalmente cubierta, así como por todo el hardware que nos proporcionaron de forma gratuita para el desarrollo de las prácticas que se realizaron durante todo el curso.

También agradecemos todo el apoyo recibido por nuestra asesora e instructora Vilchis Leon Paloma Alejandra, quien durante meses nos proporcionó las herramientas y conocimientos propios de la industria del Internet of Things.

Introducción:

La CDMX es una de las ciudades más grandes y pobladas del mundo, por lo que es de esperar que el tránsito sea un sistema increíblemente saturado y concurrente, según datos del INEGI “En la Ciudad de México se transportaron, en junio de 2021, cerca de 100 millones de personas mediante el sistema público de transporte urbano de pasajeros. Con relación al mes anterior se trasladaron 2.8% más personas; al realizar la comparación con el mismo mes de 2020 se observó un incremento de 58.6 por ciento [1]”.

Para atender el sistema de transporte público, existe en la ciudad de México la *movilidad integrada*, que es un medio de pago electrónico con formato sin contacto diseñado para el transporte público, abarca desde servicios como el metro hasta el de bicicletas públicas.

Esto muestra los esfuerzos de la ciudad por unificar los servicios de transporte y las ventajas de aquel logro serían enormes, centralizando la movilidad se podrían gestionar de forma óptima los recursos y con ello acortar los tiempos de viaje, el gasto e inversión y mejorar la calidad de vida para una enorme parte de la población, donde según la plataforma de logística y transporte moovit insights “El Porcentaje de personas en Ciudad de México que viajan en transporte público por más de 2 horas diarias esto incluye viajes en Tranvía, Metro, Tren, Autobús & Tranvía, es 67% [2]”.

Sin embargo, la *movilidad integrada* no contempla todos los medios de transporte público de la CDMX, pues deja fuera a las rutas de autobuses que no son subsidiadas por el gobierno, es decir los autobuses convencionales y tiene sentido hacero, ya que son muy informales y difíciles de medir en cuanto a tiempos de llegada, salida, pasajeros etc.

Recordemos que en el [Internet of Things](#) “Mediante la informática de bajo costo, la nube, big data, analítica y tecnologías móviles, las cosas físicas pueden compartir y recopilar datos con una mínima intervención humana. En este mundo hiperconectado, los sistemas digitales pueden grabar, supervisar y ajustar cada interacción entre las cosas conectadas [3]” con lo que el mundo digital y físico no solo interactúan en forma de datos, también interactúan reaccionando a eventos y desplegado órdenes en cualquiera de las 2 direcciones.

El proyecto **TZATZI** que se describe en las siguientes páginas, propone todo una infraestructura basada en el ya mencionado [Internet of Things](#), incluyendo un prototipo que generará una big data en tiempo real del flujo del transporte.

También el fin es apoyar los [Objetivos de Desarrollo Sostenible \(ODS\)](#), también conocidos como Objetivos Mundiales, apoyados por el [Programa de las Naciones Unidas para el Desarrollo \(PNUD\)](#) que se adoptaron por todos los Estados Miembros en 2015 “como un llamado universal para poner fin a la pobreza, proteger el planeta y garantizar que todas las personas gocen de paz y prosperidad para 2030 [4]”.

Entender que medir es entender el problema:

El tránsito vehicular en las grandes ciudades es un fenómeno extremadamente complejo y caótico por la gran cantidad de variables que intervienen, como lo son las horas de mayor afluencia, el factor humano en relación al manejo de un vehículo, los tiempos de espera en los semáforos, la geometría de las calles y las rutas, entre otras cosas.

Enfocándonos en particular a los autobuses ¿Cuáles son las variables que se debe medir? La respuesta más obvia son los tiempos de llegada al principio y final de su ruta, como si se trataran de los tiempos de viaje de un tren, sin embargo, no es suficiente, pues un autobús realiza múltiples paradas durante todo su recorrido y esto es muy importante ya que aunque un camión salga y llegue puntual al final de su ruta, no lleva una velocidad constante y resulta que esto lo cambia todo.

Así llegamos a la conclusión de que la variable más importante a medir es la ventana de tiempo; definamos a una **ventana de tiempo** como el lapso de tiempo entre 2 autobuses contiguos que siguen la misma ruta, este lapso de tiempo se mide en una determinada posición geográfica de preferencia en una estación de la ruta de los autobuses, pero podría ser cualquier punto donde ambos autobuses pasen.

Por ejemplo, supongamos una estación de autobús llamada estación H que será nuestro punto de referencia, entonces ocurre que a las 13:00 hrs el autobús A pasa por la estación, después a las 13:15 pasa el autobús B por dicha estación. Esto quiere decir que la ventana de tiempo del autobús A en relación al B es de 15 minutos.

Definiendo un escenario

Para entender cómo afectan estas ventanas de tiempo las rutas de autobuses, formulamos un escenario para estudiarlo.

Supongamos una estación tal que:

1. Cada 2 min llega un usuario a esperar el autobús.
2. En promedio el autobús tiene 8 lugares disponibles al llegar a esa estación.
3. La ventana de tiempo ideal es de 15 min.

Veamos que si la ventana de tiempo de 15 min no se respeta y en su lugar se usa una ventana de 5 min (*Imagen 1. Ventana de 5 min*), entonces, cómo llega un nuevo pasajero cada 2 min, solo se acumularon aproximadamente 2 pasajeros y se desperdiciaron 6 lugares del autobús.

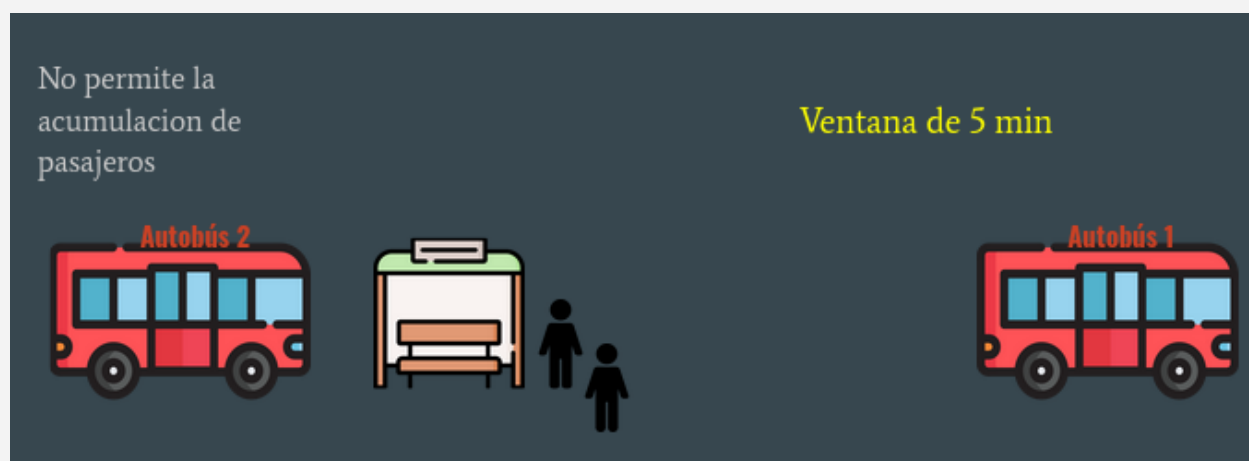


Imagen 1. Ventana de 5 min

Por otro lado si la ventana igualmente no se respeta, pero en su lugar se usa una ventana de 25 min (Imagen 2. Ventana de 25 min), entonces existe una saturación de pasajeros en espera de aproximadamente 22, no se desperdiciaron lugares, pero sí se quedaron 14 usuarios sin poder abordar.



Imagen 2. Ventana de 25 min

Además, los dos casos anteriores están conectados porque resulta que las variaciones en un ventana afectan a la ventana contigua al mismo autobús (Imagen 3. Variación en ventanas).

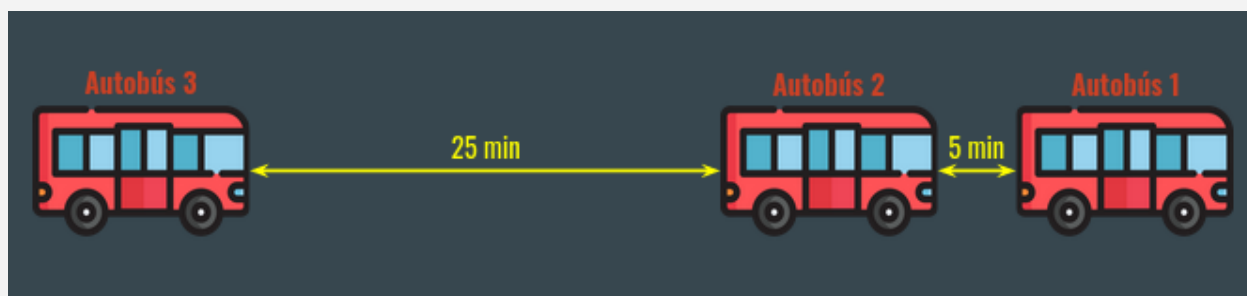


Imagen 3. Variación en ventanas

Conclusiones del escenario

1. Aun con autobuses con lugares disponibles para la cantidad exacta de pasajeros en un lapso determinado, si no se respetan las ventanas de tiempo, quedan usuarios sin poder abordar.
2. Uno de los factores clave en la optimización del tránsito público no es solo la rapidez, también la variación de la misma con respecto al resto de unidades de transporte público.

Propuesta TZATZI:

Solución obvia

Lo primero que se puede ingeniar para conocer las ventanas de tiempo en los autobuses es colocarles módulos GPS y luego conectar cada módulo a internet satelital.

No dudamos que esta solución funcione, pero el problema son los costes, si bien existen módulos GPS económicos, se tendría que colocar uno de estos a cada unidad de transporte, lo que en suma resultaría muy costoso, es verdad que muchos autobuses de primera clase o de lujo ya cuentan con estos módulos, sin embargo, esta no es una solución para autobuses que apenas y cuentan con los recursos suficientes para operar.

Solución TZATZI

Pero antes de exponer la propuesta recordemos el enfoque primario del proyecto y es el acorde con ofrecer soluciones integradas a los Objetivos de Desarrollo Sostenible (ODS) implementadas y apoyadas por el Programa de las Naciones Unidas para el Desarrollo (PNUD); TZATZI está enfocado en atender el noveno objetivo referente a la industria, innovación e infraestructura.

En particular el apartado 9.5 el cual consta de la meta de “Aumentar la investigación científica y mejorar la capacidad tecnológica de los sectores industriales de todos los países, en particular los países en desarrollo, entre otras cosas fomentando la innovación y aumentando considerablemente, de aquí a 2030, el número de personas que trabajan en investigación y desarrollo por millón de habitantes y los gastos de los sectores público y privado en investigación y desarrollo [5]”

TZATZI se une a este objetivo primordialmente porque al pertenecer a un país en vías de desarrollo como lo es México, es necesario crear soluciones a la medida de sus capacidades en lugar de desechar propuestas o ideas por no contar con los recursos, finalmente si se genera la big data esperada está a su vez podría funcionar para otros desarrolladores que ocupen estos datos para la construcción de más tecnologías, o bien el descubrimiento de conductas en el tránsito que no se habían sido avistadas antes, debido a la falta de medios para su medición y procesamiento.

La solución debe de ser entonces:

1. Innovadora.
2. Barata.
3. Estar dentro de las capacidades y recursos de la CDMX.

Resulta que las ventanas de tiempo, aunque no se conocen por ese nombre informalmente, son un problema bien conocido entre los dueños de las rutas de transporte y por supuesto los conductores.

La solución que ellos han implementado hasta ahora, es usar personas conocidas como checadores, gritones, viene viene de camiones, entre muchos otros nombres, estas personas realizan tareas fundamentales para agilizar el tránsito de los camiones y llevar un control de los mismos, de hecho el proyecto se basa e inspira en ellos, por lo cual se llama TZATZI que en náhuatl significa gritón.

Los gritones o checadores, apuntan manualmente en una libreta las ventanas de tiempo de los autobuses en relación a una parada o estación estratégica, así mismo informan a los conductores de sus ventanas de tiempo para que ellos determinen si ir más rápido o descansar.

Por ejemplo, si la ventana de tiempo ideal es de 15 min y un checador grita a un conductor que recién pasa por el lugar – 20 del 345 – significa “conductor del autobús tienes una ventana de 20 min con respecto del autobús 345 (Maneja más rápido)” otro ejemplo es si el checador grita al conductor – 8 del 345 – lo que significa “conductor del autobús tienes una ventana de 8 min con respecto del autobús 345 (esperate 7 min)”.

El problema de usar seres humanos es que la información no se centraliza, pues todo el registro se queda únicamente en sus libretas en lugar de notificarse globalmente a todo el sistema, además de que se abre un gran vector de riesgo al implementar el factor humano propenso a no poder procesar información simultánea y en masa y deficiencias propias de sistemas no computacionales

Así la solución consta de:

1. Identificará los camiones a través de reconocimiento de imágenes asociadas a códigos QR.
2. Igual que los checadores el prototipo estará colocado en estaciones o sitios estratégicos.
3. Notificará en tiempo real a un servidor que almacenará los datos en una base de datos y generará un dashboard en tiempo real.

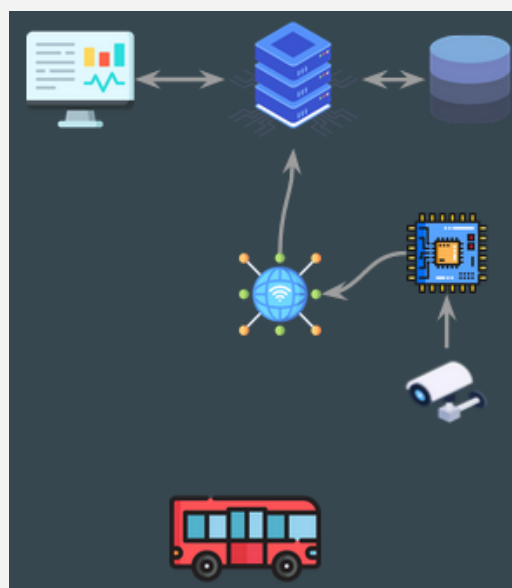



Imagen 4. TZATZI



Recordando la solución basada en GPS, además del coste existe un gasto de recursos innecesarios ya que un GPS conoce la posición en todo momento, pero nunca fue necesario ese requisito, pues no necesitamos saber a cada segundo donde están los autobuses, solo es necesario saber eso en determinadas posiciones estratégicas o estaciones, como ya lo hacen los checadores humanos pero ahora ofreciendo las bondades del internet of Things y centralizando los datos.

El nivel de escalabilidad es enorme, pues con un solo módulo checador colocado por ejemplo bajo un poste de luz para aprovechar la luz solar y la luz propia del poste durante la noche, se pueden detectar todos los autobuses no solo de la misma ruta, sino que también todos los autobuses de todas las rutas que pasen por ahí, todo con un solo módulo, además este sistema es totalmente compatible a cualquier otro medio de transporte que pueda llegar a implementar códigos QR por ejemplo las combis o peseras.

El código QR puede ser implementado con pintura sobre el vehículo o bien una simple lona bastaría para ser reconocido, aunado a todo esto resulta que los taxis otro medio de transporte que no se une a la movilidad integrada ya cuentan con un código en su techo, y es que por ley los taxis deben de llevar su placa grabada en el techo, si bien no es un código QR, si es un código alfanumérico que también puede ser detectado con visión por computadora.

Módulo checador:

Hardware

El coste del hardware es relativo a la fecha de Marzo de 2022 y a la ubicación de la ciudad de México, así que la cotización puede variar según el país, también se debe mencionar que el presupuesto se realizó en compras realizadas a minoreo, por lo que en producción los costes serían aún más baratos.

- Raspberry Pi 4 modelo B - 8Gb, costo aproximado 2 mil pesos (100 UDS).
- Cámara Raspberry Pi V1.3 1080p 5Mp, costo aproximado 170 pesos (8.5 USD).
- Carcasa de acrílico costo aproximado 80 pesos (4 USD).

La carcasa fue diseñada especialmente para el prototipo (*Imagen 5. Carcasa*) y cuenta con una tapa más grande para usar el espacio sobrante para remachar a la superficie en donde se desee colocar el prototipo. El material es acrílico resistente, ya que cualquier tipo de metal produciría estática, además de que entre todos los materiales es el más

resistente-barato y es suficiente para garantizar impermeabilidad y resistencia al medio ambiente pero se está abierto a nuevas propuestas de polímeros.

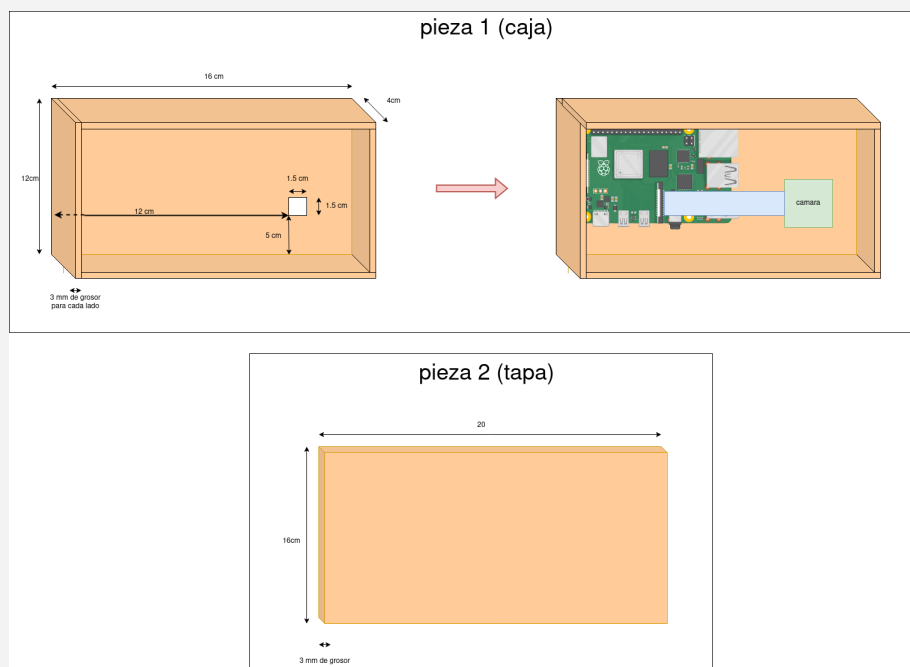


Imagen 5. Carcasa

La carcasa se diseñó en 3D usando el software de AutoCAD y se materializó en una impresora 3D, sin embargo, si el prototipo se lleva a producción los costes serían relativamente bajos pues se diseñaría un molde en lugar de imprimirse uno por uno.

Para armar el prototipo basta con conectar la banda de la cámara raspberry pi v1.3 al módulo de conexión a la cámara del raspberry pi 4 como se muestra en el diagrama (Imagen 6. Diagrama).

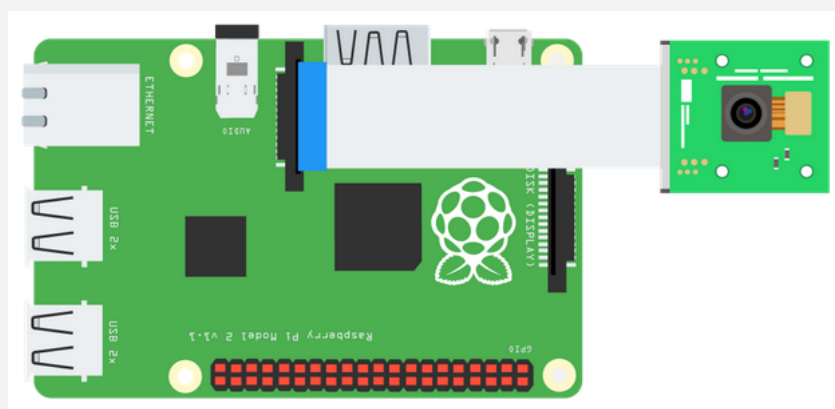



Imagen 6. Diagrama



Y colocando la lente de la cámara en el orificio destinado a eso en la carcasa (Imagen 5. Carcasa), para fijar la raspberry a la carcasa se recomienda atornillar a la carcasa usando los orificios que ya tiene la raspberry o bien pegar una carcasa comercial compacta de raspberry a la carcasa diseñada.

Software

Toda la documentacion del codigo asi como del diseño en 3D esta disponible en el repositorio <https://github.com/Huitziuit/Caspton-Project-TZATZI> aqui puede encontrar instrucciones detalladas sobre el codigo asi como las dependencias y configuraciones necesarias las cuales fueron en la raspberry pi:

- Instalar el sistema operativo Raspberry Pi OS 'Buster' Debian 10, es importante mencionar que no es la versión más nueva del OS de Raspberry, pero es hasta ahora la única que soporta OpenCV sin complicaciones.
- Configurar comunicación ssh y conexión automática a red WIFI
- OpenCV, que es una biblioteca libre de visión artificial originalmente desarrollada por Intel.
- Python 3
- Mosquitto MQTT

En la raspberry solo se ejecuta un único programa llamado estacion.py que en resumidas palabras, se conecta al cluster MQTT de HiveMQ que más adelante se explicará detalladamente exactamente que es, por ahora entendamoslo como un broker público intermediario entre nuestro prototipo checador y nuestro servidor.

Luego, usando Opencv se obtienen imágenes de lo que capta la cámara y si se detecta un QR entonces, se pinta un cuadro azul a su alrededor en un ventana que arroja el programa y muestra la información asociada a ese QR, si la información corresponde a un QR ya registrado entonces se envía un mensaje con la información del vehículo detectado al cluster MQTT.

Este mensaje contiene la placa del autobús, el checador que lo detecto (es decir la unidad Raspberry), el número del camión, ruta, fecha y además para demostrar la escalabilidad del proyecto se decidió agregar una par de datos extra generados aleatoriamente que representaran la cantidad de pasajeros que entran y salen de la unidad, para finalmente ser enviado en forma de un JSON.

Pruebas

Para probar de forma óptima el prototipo, [Hernandez Neri Guillermo Augusto](#), quien diseña y crea autobuses a escala de la CDMX, nos proporcionó tres de sus unidades a escala (*Imagen 7. Autobuses a escala*).



Imagen 7. Autobuses a escala

A cada una de ellas se les colocó un código QR en la parte superior (*Imagen 8. QR en autobuses*), lo que sería equivalente al techo de una autobus real, ese código QR contiene un id único que servirá para identificarlos con el prototipo checador.



Imagen 8. QR en autobuses

Ahora probaremos cada uno de los autobuses a escala, para ello conectaremos por HDMI el raspberry a una pantalla con lo que se observa la interfaz gráfica del sistema operativo del raspberry, esto no será necesario cuando el prototipo ya esté en producción, pero por razones de prueba y documentación procederemos de esta manera en esta ocasión.

Al pasar el primer autobús por el campo de visión de la cámara del raspberry, observamos que el código QR es detectado (*Imagen 9. Autobús 1 en checador*), muestra el mensaje que ese QR contiene y lo proyecta en el video en vivo que se ejecuta sobre el escritorio del sistema operativo del raspberry, además en la consola del programa observamos que el mensaje JSON que será enviado al broker, podemos observar que el mensaje contiene información relacionada a la placa del autobús, nombre del conductor, checador (raspberry) que lo detectó, la ruta, número del camión, fecha en que se detectó el autobús y datos aleatorios del número de pasajeros que entran y salen, como ya se mencionó, sí bien los checadores no están diseñados para detectar cuántas personas suben o bajan de las unidades de transporte, este apartado del mensaje se creó para demostrar la escalabilidad del proyecto, es decir, observar que más datos e incluso más sensores podrían ser agregados al sistema TZATZI.

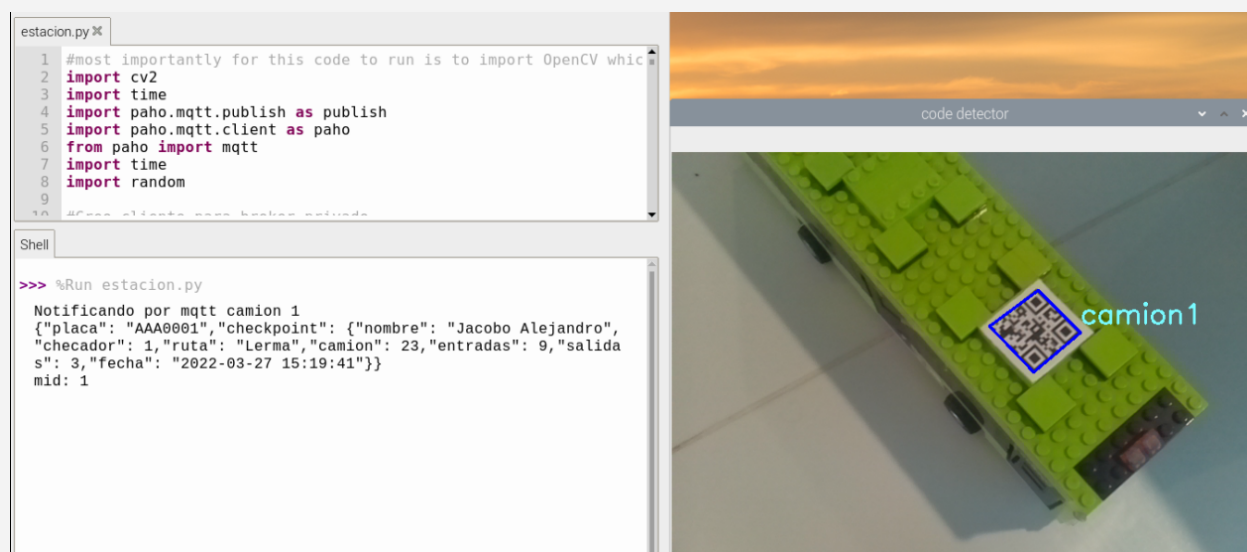


Imagen 9. Autobús 1 en checador

Observemos ahora, cómo pasan por el checador los siguientes dos camiones pertenecientes a la misma ruta y el mismo módulo checador es capaz de enviar los datos correspondientes a cada uno (*Imagen 10. Autobús 2 en checador* e *Imagen 11. Autobús 3 en checador*).



Imagen 10. Autobús 2 en checadore

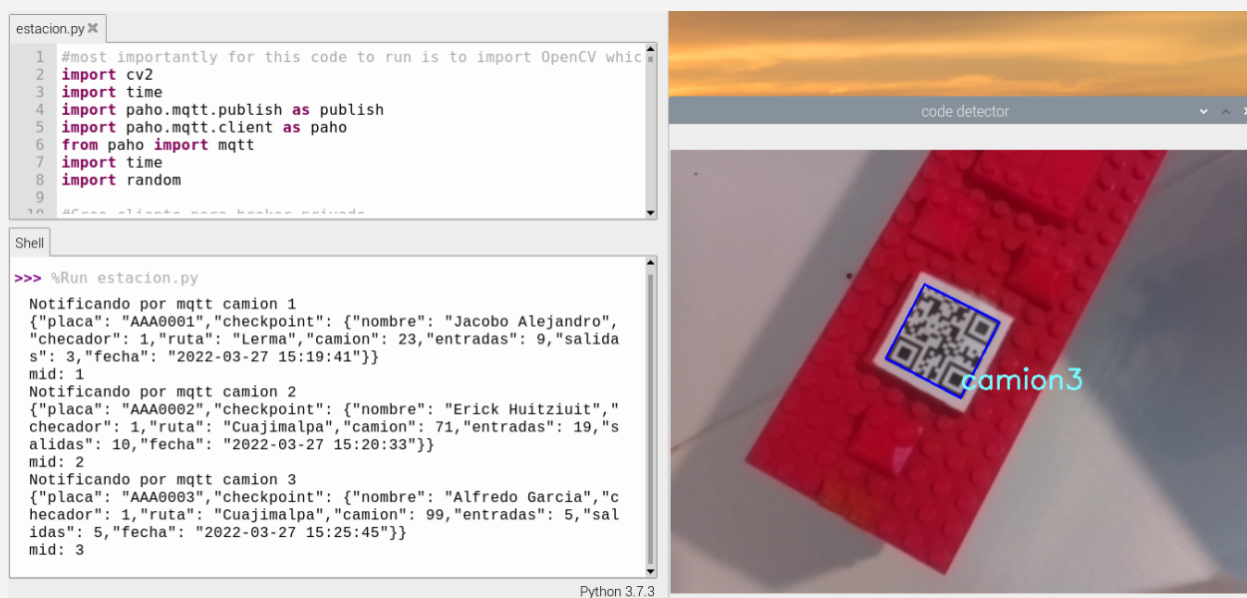


Imagen 11. Autobús 3 en checadore

Cluster:

Usando la plataforma HiveMQ se consigue un cluster, el cual es un sistema distribuido que actúa como un único agente MQTT lógico para los clientes MQTT (Protocolo Machine to Machine popular para sistemas IoT) conectados.

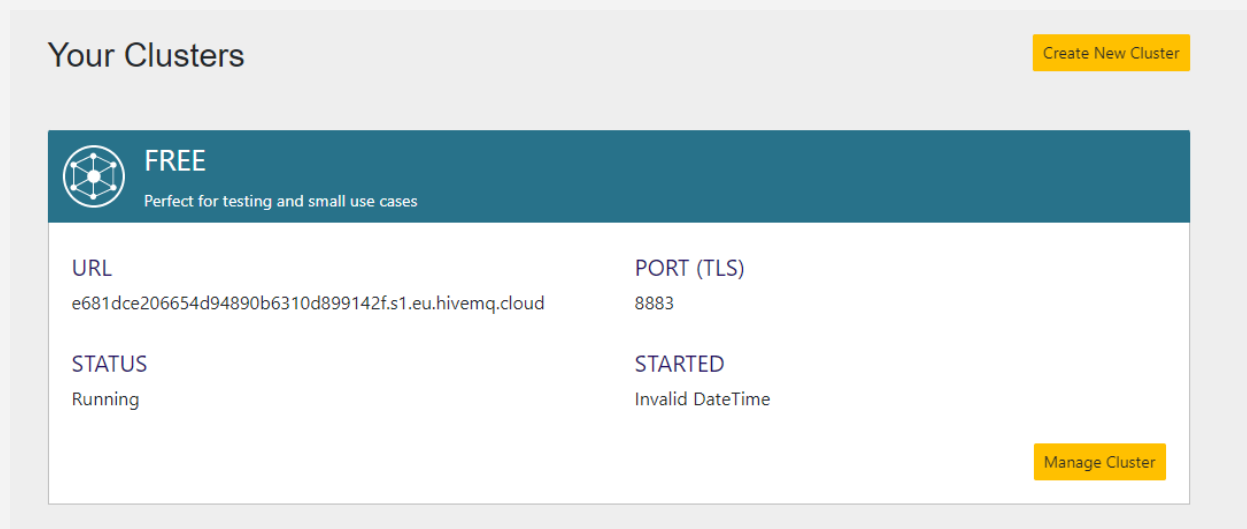


Imagen 12. Cluster gratuito de HiveMQ

El cluster gratuito que proporciona HiveMQ (Imagen 12. Cluster gratuito de HiveMQ) será suficiente para el tamaño actual del proyecto, pero es necesario crear una credencial (imagen 13. Información de la credencial), la cual permite a los clientes MQTT publicar y suscribirse al clúster de HiveMQ Cloud.

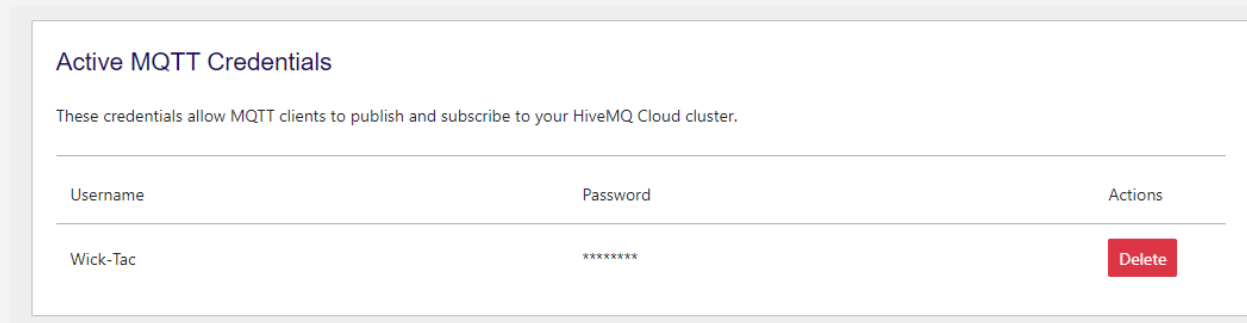


Imagen 13. Información de la credencial.

Servidor:

En el servidor, el software necesario es:

- Ubuntu Its 20.04.
- Python 3.6 (o superior).
- Distribución ANACONDA.
- Cuenta HiveMQ Cloud.
- Node-Red 2.1.6
- MySQL 8.0

Librerías necesarias para python, ANACONDA:

- paho-mqtt 1.6
- mysql-connector-python 8.0
- módulo JSON.

Se implementaron 3 programas del lado del servidor .

1. Suscripción y broker (sub_y_broker.py).
2. Mensaje (mensaje.py)
3. Publicación y simulación (pub_y_simulacion.py)

Cada uno de estos programas hace posible la comunicación con el prototipo checador a través del broker, la captura de mensajes JSON y la gestión de los datos capturados.

A continuación se presenta la descripción de cada uno y si se desea indagar más a fondo en el código, se puede consultar el repositorio ya mencionado desde la sección anterior.

Suscripción y broker (sub_y_broker.py)

La función de este programa, consiste en realizar la comunicación entre el broker (HiveMQ Cloud) que recibirá todas las notificaciones de lo que ocurre en la raspberry, con el cliente que es nuestro servidor.

El código se divide en cuatro bloques:

1. Conexión al broker
2. Conexión con la base de datos
3. Sentencias para tratar la suscripción.
4. Sentencias para tratar la publicación.

Conexión al broker

En esta sección, se hace uso de la credencial del cluster creada previamente y la URL de acceso gratuito, que se consigue al crear una cuenta en HiveMQ (la librería paho-mqtt hace posible esta conexión).

```
'''Sentencias para conectarse al Broker HivEMQ'''  
  
# usando MQTT versión 5 aquí, para 3.1.1: MQTTv311, 3.1: MQTTv31
```



```

# userdata son datos definidos por el usuario de cualquier tipo,
actualizados por user_data_set()
# client_id es el nombre dado del cliente
client = paho.Client(client_id="", userdata=None, protocol=paho.MQTTv5)
client.on_connect = on_connect

# habilite TLS para una conexión segura
client.tls_set(tls_version=mqtt.client.ssl.PROTOCOL_TLS)

try:
    # establecer nombre de usuario y contraseña de la credencial
    client.username_pw_set("username", "password")
    # conectarse a HiveMQ Cloud en el puerto 8883 (predeterminado para
MQTT MQTT v3.1)
    client.connect("URL", 8883)
except:
    print("No se pudo conectar al Broker MQTT.")
    print("Cerrando programa.")
    sys.exit()

# los callbacks de la siguientes, dan inicio a las funciones ademas de
resguarda de nuevo la información para cada loop
client.on_subscribe = on_subscribe
client.on_message = on_message

```

Conexión con la base de datos

Más adelante en el código nos encargamos de que la información recibida por el broker, se guardará en una base de datos SQL de forma local.

```

'''Sentencias para conectarse a la base de datos MYSQL'''

try:
    ## para conectar con la base de datos requerimos del metodo connect()
    ## requiere 4 parametros 'host', 'user', 'passwd', 'database'
    db = mysql.connect(
        host = "localhost",
        user = "Wick-Tac",
        passwd = "password",
        database = "Checadores_de_transporte"
    )

```

```
except:
    print("No se pudo conectar a la base de datos.")
    print("Cerrando programa.")
    sys.exit()
```

Sentencias para tratar la suscripción

En esta parte del programa, indicamos al broker a que tema debe de suscribirse, el tema utilizado en este proyecto es “checadores”, además, también se imprime en pantalla el tema al que se está suscrito y la calidad que se usará para el protocolo MQTT.

```
'''Sentencias para tratar la suscripción.'''

# Configura el callback para diferentes eventos para ver si funciona,
imprimir el mensaje, etc.
def on_connect(client, userdata, flags, rc, properties=None):
    print("CONNACK received with code %s." % rc)
    # suscríbese a todos los temas de la enciclopedia usando el comodín
    '#'
    client.subscribe("checadores", qos=1)

# imprimir a qué tema se suscribió
def on_subscribe(client, userdata, mid, granted_qos, properties=None):
    print("Subscribed: " + str(mid) + " " + str(granted_qos))
```

Sentencias para tratar la publicación

Una vez que se ha definido un tema, en esta sección se procesan los mensajes recibidos por el broker, aquí se hace uso de la clase “mensaje” que se explicará más adelante y que contiene funciones para poder integrar la información en la base de datos.

```
'''Sentencias para tratar la publicación.'''

# imprimir mensaje, útil para verificar si fue exitoso
def on_message(client, userdata, msg):
    print(msg.topic + " " + str(msg.qos) + " " + str(msg.payload))

    # crea un archivo donde sobrescribira el mensaje mqtt
```

```

f = open("mqtt.txt", "w")
# se convierte en un string el msg.payload
mensaje = str(msg.payload)
# se elimina los dos primeros caracteres (b') y el ultimo caracter
(') de la cadena
mensaje = mensaje[2:-1]
# se escribe el mensaje en el archivo
f.write(mensaje)
# se cierra el archivo
f.close()

try:
    # La declaración with simplifica el manejo de excepciones al
    encapsular tareas comunes de preparación y limpieza
    # ademas de cerrar en automatico el archivo
    with open('mqtt.txt') as read_file:
        # el str captado del archivo se guarda en una variable
        data = read_file.readline()

    # se inicializa el objeto mensaje para extraer los datos del json y
    poder usar la funciones para mysql
    # los parametros son el str del archivo y se inicializa el objeto
    mysql en la clase Mensaje
    mensaje = Mensaje(data, db.cursor())

    # se comprueba si la tabla existe
    if(mensaje.comprobar_tabla()):
        # si existe se agregan los datos a la tabla
        mensaje.insertar_datos()
    else:
        # si no existe primero la crea y despues agrega los datos
        mensaje.agregar_tabla()
        mensaje.insertar_datos()

    # se realiza la confirmacion de la sentencias ejecutas del objeto
    mensaje a la base de datos con el objeto mysql.connect()
    db.commit()

except FileNotFoundError:
    # sentencias en si surge el error de FileNotFoundError se captado
    print("El archivo de mensajes mqtt no se ha creado.")
    print("La base de datos en mysql no se ha modificado")

```

Mensaje (mensaje.py)

El broker, sólo acepta mensajes en forma de strings, por lo tanto se recurrió a la estrategia de enviar desde la raspberry el mensaje en un string pero con la estructura de un JSON, cuando el mensaje llega al broker y este a su vez lo envía a nuestro servidor es necesario formatear el string a un genuino JSON, para ello se usa esta clase, que además declara el esquema de la tabla de la base de datos y crea la función para insertar datos

El objeto cuenta con tres métodos principales además del constructor:

1. Agregar una tabla.
2. Insertar datos.
3. Comprobar tabla.

Constructor.

La clase, requiere de los parámetros, mensaje recibido y la inicialización de la clase mysql para trabajar con la base de datos.

Una vez que el constructor convierte el string al tipo de dato JSON, este puede descomponer el mensaje para extraer de manera individual cada dato que fue enviado por el checador (la raspberry).

```
'''constructor recibe un string y una base de datos.'''
def __init__(self, jsonData, db):

    # instanciar el objeto mysql de forma "privada"
    self.__db = db

    # se le asignan cada atributo del json
    # se le pasa un string y la funcion json.loads lo trasnforma en un
    json para poder extraer los datos
    self.jsonPython = json.loads(jsonData)
    self.s_placa = self.jsonPython["placa"]
    self.s_nombre = self.jsonPython["checkpoint"]["nombre"]
    self.i_checador = self.jsonPython["checkpoint"]["checador"]
    self.s_ruta = self.jsonPython["checkpoint"]["ruta"]
    self.i_camion = self.jsonPython["checkpoint"]["camion"]
    self.i_pasajeros_E = self.jsonPython["checkpoint"]["entradas"]
    self.i_pasajeros_S = self.jsonPython["checkpoint"]["salidas"]
```

```

        self.dt_fecha = self.jsonPython["checkpoint"]["fecha"]
        # por si es necesario extraer la fecha en el tipo de dato datetime
        # self.dt_fecha =
        datetime.strptime(self.jsonPython["checkpoint"]["fecha"], "%d-%m-%Y
%H:%M:%S") # se convierte el string a datetime

```

Agregar tabla

El método agregar tabla, sirve para definir los parámetros que requiere el query de la base datos para realizar la creación de una tabla, toma como base el nombre de la placa del vehículo que envía el módulo checador y es recibida por el broker es decir que por cada vehículo existe una tabla generada automáticamente.

```

'''función para agregar una tabla, depende del mensaje checador del
mqtt.'''
def agregar_tabla(self):

    # tipo de datos para la creación de la tabla
    placa = "PLACA VARCHAR(10)"
    nombre = "NOMBRE VARCHAR(30)"
    ruta = "RUTA VARCHAR(15)"
    camion = "CAMION INT"
    checador = "CHECADOR INT"
    pasajeros_E = "ENTRADAS INT"
    pasajeros_S = "SALIDAS INT"

    checkpoint = nombre+", "+ruta+", "+camion+", "+checador+", "
    sensores = pasajeros_E+", "+pasajeros_S
    fecha = ",FECHA DATETIME"

    # nombre de la tabla
    placa = self.s_placa
    query = "CREATE TABLE "+placa+" (" +checkpoint+sensores+fecha+)"

    # se indica la instrucción del query a la base de datos.
    self.__db.execute(query)

```

Insertar datos

Este método se encarga de realizar un query a la base de datos para insertar los datos de acuerdo a la información extraída del JSON recibido por el broker

```
'''función para insertar datos a una tabla.'''
def insertar_datos(self):

    # nombre de la tabla
    placa = self.s_placa
    # definiendo el query
    query = "INSERT INTO "+placa+" (NOMBRE, RUTA, CAMION, CHECADOR,
ENTRADAS, SALIDAS, FECHA) VALUES (%s,%s,%s,%s,%s,%s,%s)"

    ## almacenar valores en una variable
    values = (
                self.s_nombre,
                self.s_ruta,
                str(self.i_camion),
                str(self.i_checador),
                str(self.i_pasajeros_E),
                str(self.i_pasajeros_S),
                str(self.dt_fecha)
            )

    ## indica la instrucción del query a la base de datos junto con los
    valores indicador por los atributos.
    self.__db.execute(query, values)
```

Comprobar tabla.

Este método hace una consulta a la base de datos, si existe una tabla que coincida con el nombre de la placa (modulo TZATZI) a la recibida del broker en formato JSON entonces retorna verdadero en caso contrario un falso.

```
def comprobar_tabla(self):

    # Realiza la consulta de todas la tablas.
    self.__db.execute("SHOW TABLES")
```

```

        # la funcion fetchall obtiene todas las filas de un resultado de
        # consulta. Devuelve todas las filas como una lista de tuplas.
        # Se devuelve una lista vacía si no hay ningún registro que
        # recuperar.
        tuplas = self.__db.fetchall()

        # retorna un False si no hay ninguna tabla creada
        if( str(tuplas) != ""):
            for tablas in tuplas:
                #comprueba cada una de las tablas con el numero de tabla
                #del json del objeto
                if(str(tablas) == "("+self.s_placa+",)"):
                    #retorna un True si existe la tabla del numero del
                    #cheCADOR del mensaje json
                    return True

            # retorna un False si no existe la tabla (placa del mensaje
            #JSON)
            return False
        # retorna un False si no hay ninguna tabla creada
        else:
            return False

```

Publicación y simulación (pub_y_simulación.py).

Sirve para publicar a un broker y utiliza las mismas sentencias de conexión al broker vistos en el apartado del programa sub_y_broker.py.

```

# configurar devoluciones de llamada para diferentes eventos para ver si
# funciona, imprimir el mensaje, etc.
def on_connect(client, userdata, flags, rc, properties=None):
    print("CONNACK received with code %s." % rc)

# con esta devolución de llamada puede ver si su publicación fue exitosa
def on_publish(client, userdata, mid, properties=None):
    print("mid: " + str(mid))

# imprimir a qué tema se suscribió
def on_subscribe(client, userdata, mid, granted_qos, properties=None):

```

```

    print("Subscribed: " + str(mid) + " " + str(granted_qos))

# usando MQTT versión 5 aquí, para 3.1.1: MQTTv311, 3.1: MQTTv31
# userdata son datos definidos por el usuario de cualquier tipo,
# actualizados por user_data_set()
# client_id es el nombre dado del cliente
client = paho.Client(client_id="", userdata=None, protocol=paho.MQTTv5)
client.on_connect = on_connect

# habilite TLS para una conexión segura
client.tls_set(tls_version=mqtt.client.ssl.PROTOCOL_TLS)

# establecer nombre de usuario y contraseña
client.username_pw_set("<Tu usuario HiveMQ>", "Tu contraseña")

# conectarse a HiveMQ Cloud en el puerto 8883 (predeterminado para MQTT
MQTT v3.1)
client.connect("e681dce206654d94890b6310d899142f.s1.eu.hivemq.cloud", 8883)

client.on_publish = on_publish

```

La diferencia de este programa con respecto al de `sub_y_broker.py` es que en esta se implementan líneas de código para simular el envío de mensajes al broker lo cual viene muy bien para realizar pruebas antes de integrar el módulo checador al sistema.

Estas sentencias se caracterizan por simular de manera aleatorio los datos del mensaje que se enviará al broker en un formato JSON.

```

# ***** INICIO DEL CODIGO PARA SIMULAR LOS DATOS Y MENSAJE DEL JSON QUE SE
PUBLICARA,
#     SI YA CUENTA CON UN MENSAJE JSON PARA MANDAR PUEDE COMENTAR LAS LINEAS
SIGUIENTES Y INICIAR LA VARIABLE (mensaje) PARA SUS FINES *****

try:

# una sola publicación, esto también se puede hacer en bucles, etc.

    #bandera = 0
    #while(bandera <10):

```



```

while(True):

    # Variables simuladas
    #placa = '"placa": "AAA0003"'

    placa = '"placa": "AAA000'+str(random.randint(1,3))+'"'
    checador = '"checador": '+str(random.randint(0,15))
    camion = '"camion": '+str(random.randint(0,30))

    entradas_n = random.randint(0, 20)
    entradas = '"entradas": '+str(entradas_n)
    salidas = '"salidas": '+str(random.randint(0, int(entradas_n)))

    fecha = '"fecha": '+'"' + (time.strftime('%Y-%m-%d %H:%M:%S',
time.localtime())) + '"'

    # mensaje json
    mensaje = ('{' + placa + ', "checkpoint": {"nombre": "Pedro
Gutierrez", '+ checador + ', "ruta":
"Olivos", '+ camion + ', '+ entradas + ', '+ salidas + ', '+ fecha + '}}')
    print(mensaje)

    client.publish('checadores', payload=mensaje, qos=0)
    #bandera +=1
    time.sleep(1)

except KeyboardInterrupt: #precionar Ctrl + C para salir
    print("\nCerrando.")

```

Si durante la fase de pruebas del servidor se desean enviar datos concretos y no aleatorios, se dejaron comentadas las últimas líneas del programa, para que los desarrolladores puedan usarlas para enviar valores concretos simplemente descomentándolas y omitiendo el código responsable de valores aleatorios..

```

# ***** FIN DE LAS LINEAS DE CODIGO PARA SIMULAR DATOS, INICIE UN OBJETO
(mensaje) PARA SUS FINES Y DESCOMENTE LA SIGUIENTE Y ULTIMA LINEA DE CODIGO
*****

```

```
# mensaje = "TU MENSAJE json"

# una sola publicación, esto también se puede hacer en bucles, etc.

#try:
#    client.publish("topic", payload="mensaje", qos=1)
#except KeyboardInterrupt: #precionar Ctrl + C para salir
#    print("\nCerrando.")
```

Base de datos MySQL

Esta parte es fundamental, pues la base de datos finalmente centralizará toda la información en un único lugar, permitiendo una fuente de información confiable de lo que ocurre en tiempo real en las calles con respecto al transporte, con la cual se podrá hacer todo tipo de estudios, desarrollar sistemas que encuentren patrones en los datos o servir de fuente primaria para el desarrollo de proyectos destinados al transporte.

Se utilizó el sistema de gestión de base de datos MySQL por ser Open Source y a su gran versatilidad y apoyo por la comunidad. Para poder trabajar con el proyecto y MySQL fue necesario utilizar el plugin de autenticación `mysql_native_password` (*Imagen 14. Plugin de autenticación del usuario*) y crear una base de datos con el nombre de `Checadores_de_transporte`.

```
mysql> SELECT user, authentication_string, plugin, host FROM mysql.user;
```

user	authentication_string	plugin	host
Wick-Tac	*2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19	mysql_native_password	localhost
.Vmk4dZsKbGd.CF0eBShdRs/I00zDfbzkmctK8EFM2yD		caching_sha2_password	localhost
mysql.infoschema	\$A\$005\$THISISACOMBINATIONOFINVALIDSALTANDPASSWORDTHATMUSTNEVERBRBEUSED	caching_sha2_password	localhost
mysql.session	\$A\$005\$THISISACOMBINATIONOFINVALIDSALTANDPASSWORDTHATMUSTNEVERBRBEUSED	caching_sha2_password	localhost
mysql.sys	\$A\$005\$THISISACOMBINATIONOFINVALIDSALTANDPASSWORDTHATMUSTNEVERBRBEUSED	caching_sha2_password	localhost
root		auth_socket	localhost

```
6 rows in set (0.00 sec)

mysql>
```

Imagen 14. Plugin de autenticación del usuario

Recordemos que en la base de datos, para cada placa de algún vehículo se tiene una tabla asociada (*Imagen 15. Tablas en la DB*).

```
Database changed
mysql> show tables;
+-----+
| Tables_in_Checadores_de_transporte |
+-----+
| AAA0001 |
| AAA0002 |
| AAA0003 |
+-----+
3 rows in set (0.00 sec)
```

Imagen 15. Tablas en la DB

```
mysql> select all * from AAA0001 order by FECHA DESC;
```

NOMBRE	RUTA	CAMION	CHECADOR	ENTRADAS	SALIDAS	FECHA
Pedro Gutierres	Olivos	11	12	8	3	2022-03-29 16:49:06
Pedro Gutierres	Olivos	16	12	16	7	2022-03-29 16:49:05
Pedro Gutierres	Olivos	27	8	18	2	2022-03-29 16:49:04
Pedro Gutierres	Olivos	9	9	2	2	2022-03-29 16:49:03
Pedro Gutierres	Olivos	23	2	13	4	2022-03-29 16:48:54
Pedro Gutierres	Olivos	22	14	20	12	2022-03-29 16:48:51
Pedro Gutierres	Olivos	8	3	9	9	2022-03-29 16:48:50
Pedro Gutierres	Olivos	14	3	11	9	2022-03-29 16:48:49
Pedro Gutierres	Olivos	5	13	16	5	2022-03-29 16:48:42
Pedro Gutierres	Olivos	26	6	5	4	2022-03-29 16:48:37
Pedro Gutierres	Olivos	5	13	7	3	2022-03-29 16:48:31
Pedro Gutierres	Olivos	13	2	13	9	2022-03-29 16:48:30
Pedro Gutierres	Olivos	29	15	6	1	2022-03-29 16:48:28
Pedro Gutierres	Olivos	15	5	9	5	2022-03-29 16:48:27
Pedro Gutierres	Olivos	5	7	12	12	2022-03-29 16:48:26
Pedro Gutierres	Olivos	17	10	0	0	2022-03-29 16:48:24
Pedro Gutierres	Olivos	10	5	9	7	2022-03-29 16:48:23
Pedro Gutierres	Olivos	3	3	11	4	2022-03-29 16:48:15
Pedro Gutierres	Olivos	1	9	10	7	2022-03-29 16:48:13
Pedro Gutierres	Olivos	5	0	17	8	2022-03-29 16:48:11
Pedro Gutierres	Olivos	27	7	17	8	2022-03-29 16:48:08
Pedro Gutierres	Olivos	9	9	7	4	2022-03-29 16:48:05
Pedro Gutierres	Olivos	9	5	20	0	2022-03-29 16:48:04
Pedro Gutierres	Olivos	22	7	13	9	2022-03-29 16:48:02
Pedro Gutierres	Olivos	10	0	9	1	2022-03-29 16:48:00
Pedro Gutierres	Olivos	25	15	16	12	2022-03-29 16:47:58

Imagen 16. Ejemplo de datos de una tabla en tiempo real

Dashboard:

Si bien, la persistencia de la base de datos ya le otorga un valor enorme a todo el proyecto, hace falta observar los beneficios de esta en alguna implementación, para lo cual se diseñó un dashboard que grafica en tiempo real los eventos registrados por los módulos checadores.

Durante las pruebas usamos como material de estudio 3 autobuses a escala, donde 2 de ellos pertenecen a la misma ruta.

Para poder implementar el dashboard es necesario instalar node red 2.2.2 además de los siguientes nodos:

- node-red-dashboard 3.1.4
- node-red-node-mysql 1.0.0
- node-red-contrib-loop 1.0.1

El dashboard se divide en las siguientes secciones:

Checadores en tiempo real

Realiza una gráfica en tiempo real de cada una de las unidades de transporte registradas en la base de datos, las gráficas se presentan como checadores con respecto al tiempo, de esta manera podemos monitorizar cada una de las unidades y así saber cual es la ruta que siguen y su ventana de tiempo (*Imagen 17. Bloque de nodos que genera las gráficas de cada unidad*).

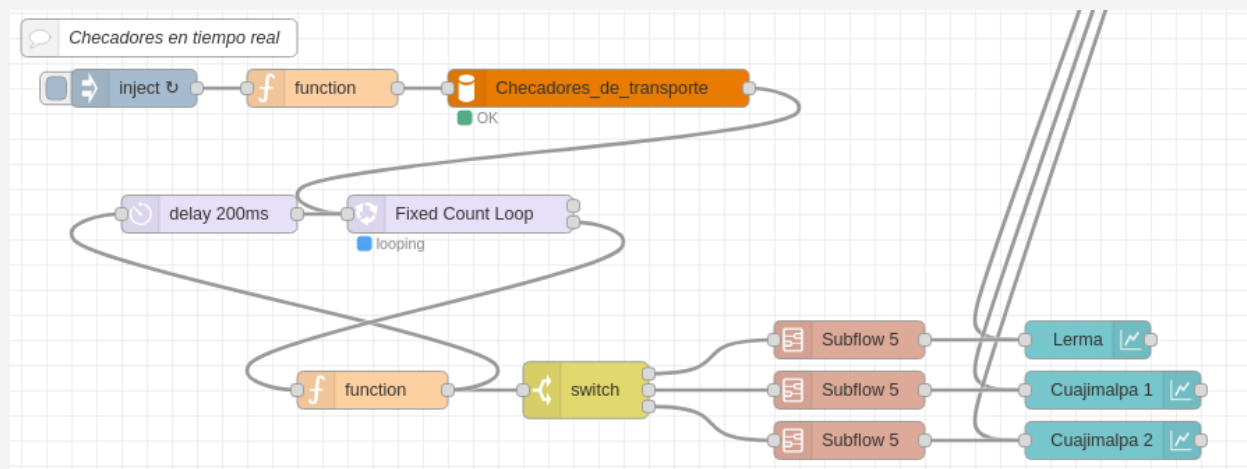


Imagen 17. Bloque de nodos que genera las gráficas de cada unidad

Registros

La función de estos nodos es realizar una captura de la gráfica de alguna unidad de transporte, ya que las gráficas de *Checadores en tiempo real*, *Pasaje*, *Entradas y salidas* cambian con respecto al tiempo debido a que se actualizan cada 200 milisegundos (*Imagen 18. Nodos que realizan la captura*).

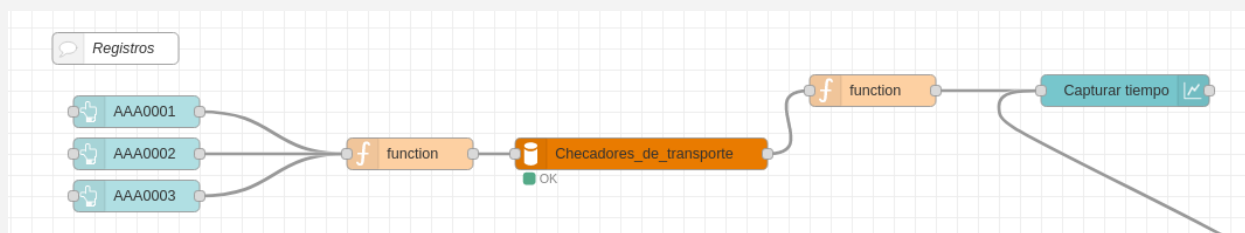


Imagen 18. Nodos que realizan la captura

Entradas y salidas.

Su función es presentar una comparación entre el número de pasajes respecto al checador anterior con el checador actual, de esta manera se pueden realizar un análisis de los puntos y horarios donde hay mayor flujo de pasajeros en determinada ruta.

Sí bien el módulo checador no está diseñado para contar a los pasajeros, esta sección plantea demostrar la escalabilidad y fácil cohesión de futuras funciones al proyecto, TZATZI no se limita a realizar el registro con los checadores, sino que también aceptaría cualquier nuevo sensor a su repertorio (Imagen 19. Entradas y salidas).

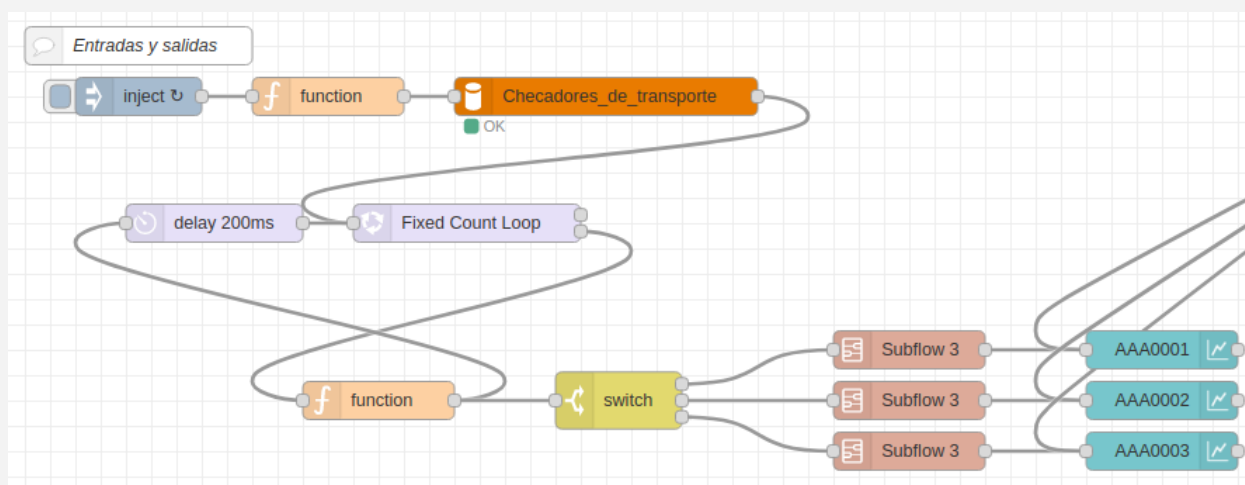


Imagen 19. Entradas y salidas

Pasaje

Esta sección de nodos genera una gráfica de la sumatoria de pasajeros, esta se realiza de manera continua cada que pasa una unidad vehicular por el checador, este es el segundo ejemplo de escalabilidad del proyecto.

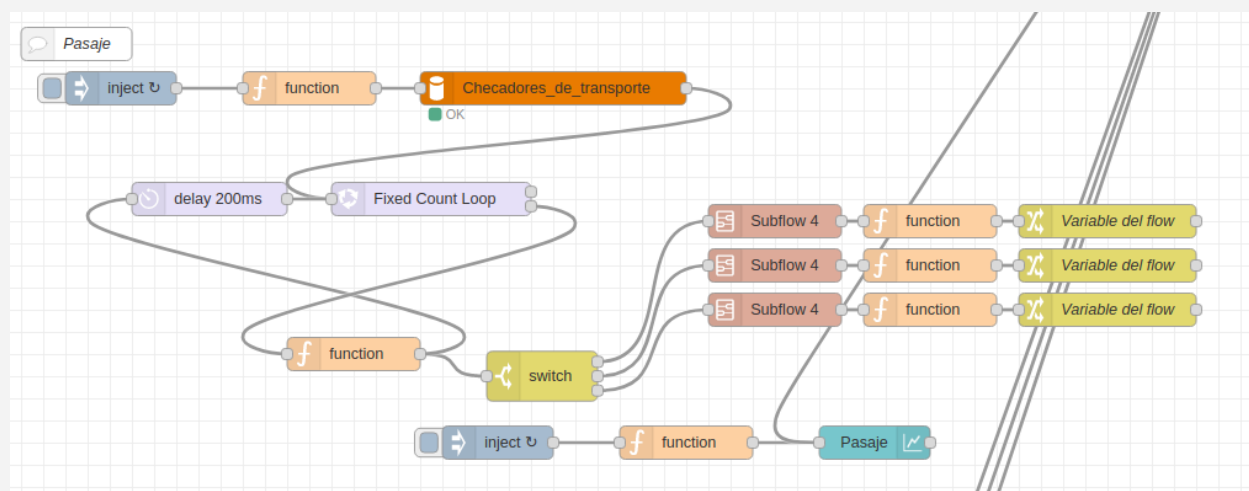


Imagen 20. Pasaje

Finalmente usando nuestros autobuses a escala, así lucen los datos recibidos por el servidor en el dashboard (Imagen 21. Prueba del Dashboard en Node-Red).



Imagen 21. Prueba del Dashboard en Node-Red.

Referencias:

1. INEGI. (2021, 13 de agosto). ESTADÍSTICA DE TRANSPORTE URBANO DE PASAJEROS CIFRAS DURANTE JUNIO DE 2021. Instituto Nacional de Estadística y Geografía (INEGI).
https://www.inegi.org.mx/contenidos/saladeprensa/notasinformativas/2021/ETUP/ETUP2021_08.pdf
2. Estadísticas y datos de transporte público en Ciudad de México. (s. f.). The World's Most Popular Urban Mobility App.
https://moovitapp.com/insights/es-419/Moovit_Insights_Índice_de_Transporte_Público_México_Ciudad_de_Mexico-822
3. ¿Qué es el Internet de las cosas (IoT)? (s. f.). Oracle | Cloud Applications and Cloud Platform. <https://www.oracle.com/mx/internet-of-things/what-is-iot/>
4. *Objetivos de Desarrollo Sostenible | El PNUD en México.* (s. f.). UNDP.
<https://www.mx.undp.org/content/mexico/es/home/sustainable-development-goals.html>
5. *Objetivo 9: Industria, innovación e infraestructura | El PNUD en México.* (s. f.). UNDP.
<https://www.mx.undp.org/content/mexico/es/home/sustainable-development-goals/goal-9-industry-innovation-and-infrastructure.html#targets>

Bibliografía:

1. "HiveMQ Introduction :: HiveMQ Documentation". HiveMQ - Enterprise ready MQTT to move your IoT data.
<https://www.hivemq.com/docs/hivemq/4.7/user-guide/introduction.html> (accedido el 6 de abril de 2022).
2. "paho-mqtt". PyPI. <https://pypi.org/project/paho-mqtt/> (accedido el 6 de abril de 2022).
3. "mysql-connector-python". PyPI. <https://pypi.org/project/mysql-connector-python/> (accedido el 6 de abril de 2022).
4. "json â ” JSON encoder and decoder — Python 3.10.4 documentation". 3.10.4 Documentation. <https://docs.python.org/3/library/json.html> (accedido el 6 de abril de 2022).
5. "MySQL :: MySQL 8.0 Reference Manual :: 3.1 Connecting to and Disconnecting from the Server". MySQL :: Developer Zone.
<https://dev.mysql.com/doc/refman/8.0/en/connecting-disconnecting.html> (accedido el 6 de abril de 2022).
6. "node-red-dashboard". Library - Node-RED.
<https://flows.nodered.org/node/node-red-dashboard> (accedido el 6 de abril de 2022).
7. "node-red-node-mysql". Library - Node-RED.
<https://flows.nodered.org/node/node-red-node-mysql> (accedido el 6 de abril de 2022).
8. "node-red-contrib-loop". Library - Node-RED.
<https://flows.nodered.org/node/node-red-contrib-loop> (accedido el 6 de abril de 2022).
- 9.

Contacto:

Desarrolladores del proyecto

- e.huitzilopochtli@gmail.com
- ghostrekoon@gmail.com
- alfredosys@outlook.com

Autobuses a escala

- avatar.neri96@gmail.com