http://dx.doi.org/10.35596/1729-7648-2024-22-4-105-113

Original paper

UDC 004.738; 615.47

MODEL AND STRUCTURE OF IoT NETWORK FOR ALZHEIMER'S DISEASE DIAGNOSTICS

ULADZIMIR A. VISHNIAKOU, CHUYUE YU

Belarusian State University of Informatics and Radioelectronics (Minsk, Republic of Belarus)

Submitted 22.01.2024

Abstract. The article presents the structure and model of an Internet of Things network that can be used for remote rapid detection of Alzheimer's disease. A local server model of the Internet of Things network has been created for personalized medical care on the client side. The model corresponds to the characteristics of the Internet of Things network: interconnection between devices, real-time communication, data processing and analysis, the use of various protocols for data transfer and exchange. When building the model, the Flask framework was used to create an application instance with a trigger condition for sending data from a smartphone to a local server via an HTTP request. The local server receives the HTTP request sent by the smartphone and processes the data. The result of the procedure is transmitted through the MQTT protocol to the MQTT client that has been subscribed to certain topics, i.e., the smartphone. Taking into account the selected structure and configuration of the Internet of Things network device, a complete model of this network was built, which can be applied to various applications. The functions and performance of the model are verified through experiments.

Keywords: Internet of things network, IT diagnostics, Flask, EMQX, MQTTX, HTTP, prediction.

Conflict of interests. The authors declare no conflict of interests.

For citation. Vishniakou U. A., Chuyue Yu. (2024) Model and Structure of IoT Network for Alzheimer's Disease Diagnostics. *Doklady BGUIR*. 22 (4), 105–113. http://dx.doi.org/10.35596/1729-7648-2024-22-4-105-113.

МОДЕЛЬ И СТРУКТУРА СЕТИ ИНТЕРНЕТА ВЕЩЕЙ ДЛЯ ДИАГНОСТИКИ БОЛЕЗНИ АЛЬЦГЕЙМЕРА

В. А. ВИШНЯКОВ, ЧУЮЭ ЮЙ

Белорусский государственный университет информатики и радиоэлектроники (г. Минск, Республика Беларусь)

Поступила в редакцию 22.01.2024

Аннотация. В статье представлены структура и модель сети интернета вещей, которые могут быть использованы для удаленного быстрого выявления болезни Альцгеймера. Создана локальная серверная модель сети интернета вещей для персонализированного медицинского обслуживания на стороне клиента. Модель соответствует характеристикам сети интернета вещей: взаимосвязь между устройствами, связь в режиме реального времени, обработка и анализ данных, использование различных протоколов для передачи и обмена данными. При построении модели использовался фреймворк Flask для создания экземпляра приложения с триггерным условием отправки данных со смартфона на локальный сервер через HTTP-запрос. Локальный сервер получает HTTP-запрос, отправленный смартфоном, обрабатывает данные. Результат процедуры передается через протокол MQTT для клиента MQTT, который был подписан на определенные темы, т. е. на смартфон. С учетом выбранной структуры и настройки устройства сети интернета вещей построена полная модель данной сети, которую можно применять к различным приложениям. Функции и производительность модели проверены с помощью экспериментов.

Ключевые слова: сеть интернета вещей, IT-диагностика, Flask, EMQX, MQTTX, HTTP, прогноз.

Конфликт интересов. Авторы заявляют об отсутствии конфликта интересов.

Для цитирования. Вишняков, В. А. Модель и структура сети интернета вещей для диагностики болезни Альцгеймера / В. А. Вишняков, Чуюэ Юй // Доклады БГУИР. 2024. Т. 22, № 4. С. 105–113. http://dx.doi.org/10.35596/1729-7648-2024-22-4-105-113.

Introduction

Internet of things (IoT) technology primarily relies on information collection devices to establish connections between objects and the network. It interacts on the basis of predefined protocols to perform functions such as tracking and managing information [1]. The monograph [2] provides information on the design of the Internet for the analysis of product quality and environmental sound processing. The construction of an intelligent medical system supported by the IoT technology contributes to the informatization of medical resources and personalized medical decision-making. This allows you to analyze, calculate and exchange huge amounts of data. Traditional hospital models rely heavily on complex processes of multiple analyses, which leads to a loss of time, the necessity of patient presence, and a decrease in efficiency. In addition, due to regional differences, the distribution of medical resources is uneven. The IoT model helps to solve these problems.

The authors presented the technology of recognition of Alzheimer's disease based on the analysis of voice data, machine learning and a neural network in [3]. In this article, the authors presented a model and proposed the structure of the IoT network for remote recognition of Alzheimer's disease. This helps in early diagnosis and surgical treatment, provides real-time monitoring, expands the voice database and can help improve the quality of medical services for patients with Alzheimer's disease.

Flask framework

Flask is a Python based development that relies on Jinja2 template rendering engine and Werkzeug WSGI routing service component as core of the micro-framework, with good scalability and compatibility, can help users to quickly realize a website or web service [4], its work process diagram is shown in Fig. 1. Routing, debugging and WSGI subsystem, are three core components of Flask framework, provided by Werkzeug. Among them, routing refers to mapping URI requests to the corresponding processing function, generally a view function, Werkzeug provides Flask framework internal routing system, which is responsible for route matching distribution and HTTP service response, support for URL routing request integration and can respond to multiple users access requests at a time. Debugging means that Werkzeug provides a complete debugging tool, including stack tracing and error alerts, able to pinpoint the error or exception code during the development process, and perform debugging. WSGI is known as the Web Server Gateway Interface, is a general interface specification between Python Web servers and Web applications, Werkzeug implements the WSGI protocol, so that Flask applications can run on WSGI-compliant Web servers.

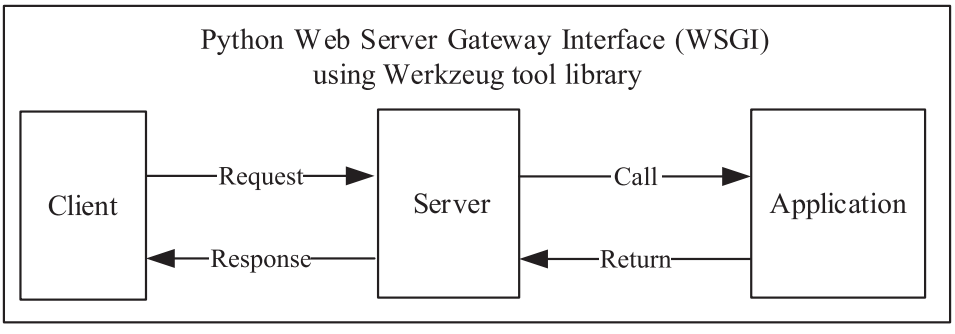

Fig. 1. The flask framework working process diagram

The template system is provided by Jinja2, a technology used for building dynamic web pages that allows for the separation of the page's structure and content. Jinja2 enables developers to embed dynamic Python code within HTML files, generating dynamic content.

Flask is currently a very popular web framework [5]. It not only allows developers to combine MVC (Model-View-Controller) pattern for development but also supports small teams in quickly building medium to small-sized websites or web services. Additionally, Flask offers strong customization capabilities and a powerful plugin library, enabling users to achieve personalized website customization while maintaining a concise, easily maintainable, highly secure, stable, and efficient core functionality. For instance, Flask can be extended with Flask-SQLAlchemy to implement database functionality, Flask-Login for user authentication, Flask-Mail for email functionality, and also supports adding ORM (Object-Relational Mapping), form validation tools, file uploads, and more. The basic pattern of Flask is to assign a view function to a URL in the program, whenever a user accesses the URL, the system will execute the view function assigned to the URL, get the return value of the function and display it to the browser.

The server in Fig. 1 is the WSGI server responsible for listening to network ports, receiving HTTP requests and invoking the WSGI application to process the requests. Werkzeug provides an implementation of the WSGI interface. When a client initiates an HTTP request, the request is sent to the listening interface of the WSGI server, based on the received HTTP request data, the server creates a dictionary of environment variables (environ), which will be used as parameters along with a callback function (start_response) to invoke the WSGI application object. An application in Flask is defined as an instance of a Flask object or its subclass, which is a WSGI application object. When a WSGI server calls the WSGI application, it is actually calling the Flask object, which will locate the object's route based on the URI path and method of the HTTP request. Upon finding a matching route (view function), Flask will call it to process the HTTP request and get a result, usually a response object will be returned, the response data includes status code, header and body of a HTTP response, then send the response back to the client.

EMQX Broker

EMQX (also known as EMQX Broker) is an open-source, distributed MQTT message middleware server [4], used for implementing MQTT protocol-based message transmission and communication. It is widely applied in various real-time communication and message push scenarios, including the IoT, smart homes, and industrial control. The latest version, EMQX 5.0, adopts a brand-new Mria cluster architecture, allowing an EMQX cluster to support up to 100 million concurrent MQTT connections, significantly enhancing its scalability. Additionally, the Mria cluster architecture reduces the risk and impact of brain-splitting in large-scale deployments, providing customers with more stable access to IoT data services.

EMQX supports publish/subscribe messaging model and provides multiple functions including support for subscribing to wildcard topics, retaining message and Quality of Service (QoS). Wildcard support is that EMQX supports MQTT wildcard subscription. Clients can use specific wildcard symbols to subscribe vaguely, flexibly realizing topic-based message filtering, the wildcard symbol includes two kinds of symbols: "+" and "#". Retained message refers to the MQTT message retention mechanism supported by EMQX to maintain message persistence. When a client publishes a message with the retain flag to designated topic, EMQX server will retain the message and if a new client subscribes to the topic, the retained message will be immediately delivered to the new client. The retain flag is a boolean value, configurable by the client, which can be either "0" or "1". Quality of Service (QoS) support refers to EMQX's support for MQTT's QoS level settings. The quality of service for message distribution can be categorized into QoS 0, QoS 1, and QoS 2, again allowing the client specify its own quality of message delivery.

As a server-side software, EMQX Broker is responsible for maintaining subscription relationships, it receives MQTT messages from different clients and ensures that these messages are routed to the correct subscribers, acting as a relay station for the messages. EMQX Broker is a central server implementing the MQTT protocol and can be a local server or a cloud server. This project established a connection with the MQTT Broker server using python code, allowing the application to publish and subscribe to messages for real-time data transfer and communication. The used connection parameters and their explanations are shown in Tab. 1.

Table 1. Connection Parameters for the EMQX Broker

Parameters' names	Parameters' implications	Parameters' values
broker_address	The address of the MQTT Broker server is typically an IP address or domain name used to identify the location of the MQTT Broker on the network	'localhost'
broker_port	The port number of the MQTT Broker server is used to identify the port on which the MQTT Broker is listening on the server	1883
broker_username	Optional: Since the EMQX Broker requires authenticated users, the authors provide a username	Authors' username
broker_password	Optional: Since the EMQX Broker requires authenticated users, the authors provide a password	Authors' password
topic	MQTT message publishing and subscribing are done using topics. Clients can publish messages to specific topics or subscribe to messages from specific topics	'/flask/predict'

Model and structure of IoT network

This project adopted the architecture design of 'mobile frontend display + Flask framework backend + EMQX Broker.' The code has strong scalability and maintainability. The system structure and interaction process are shown in Fig. 2.

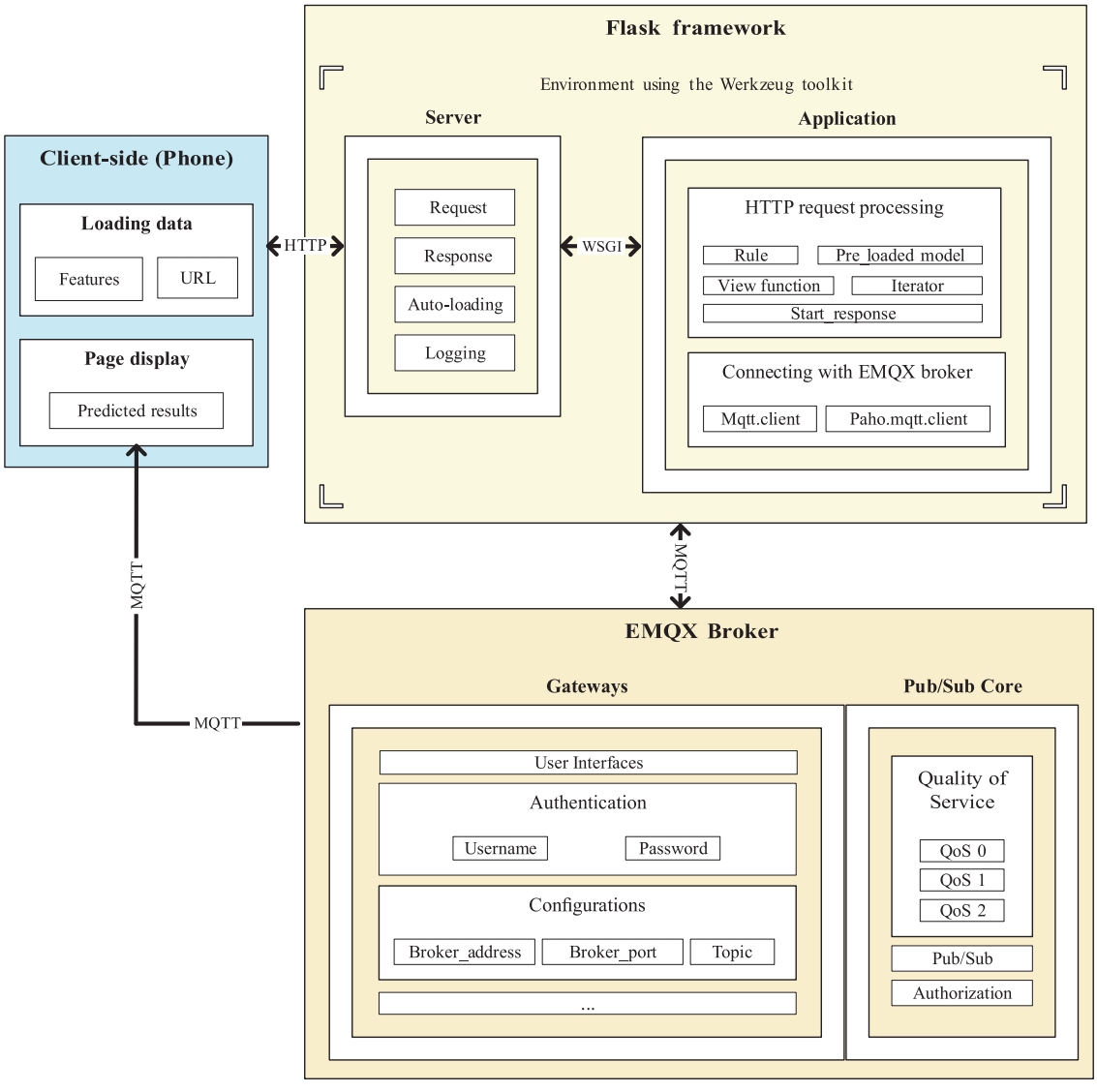

Fig. 2. Structure of IoT network for AD diagnostics

The Frontend layout of the screen was written in Java, it realized the process of uploading data to the local server through HTTP protocol and received the prediction result after the data was processed through MQTT protocol. The Result was presented on the screen. The function of local server was implemented by Flask framework and EMQX Broker, Flask framework received HTTP request sent from phone, parsed and found a matching view function, executed it, returned a HTTP response. EMQX Broker precisely sent the prediction result to the client subscribed to the topic cross-platform to realize the real-time communication. The following is a specific materialization of all the processes in the code, which was shown in three blocks from client-side to server-side in detail.

1. Uploading data from client to the local server:

a) reading the pre-collected data features from mobile phone. After collected participant's related speech, it will be processed and feature-extracted by the data preprocessing and feature extraction algorithms installed on the client. The authors directly load the data feature file in this project, which has a file extension of 'csv';

b) setting URL for the request. The authors set URL as "*http://192.168.100.14:5000/predict*", containing local IP address and local port in data format, 192.168.100.14 was the local IP address of authors' computers, 5000 is authors' local port number, which can be used to identify the web service, indicating that the web service listens on port "5000" on computer. '*/predict*' was the path portion of URL, which can be used to identify a specific resource or function in the Web service. In this project, function 'predict' can predict whether a participant has Alzheimer's disease based on the data uploaded by the client and generate a diagnostic prediction of Alzheimer's disease;

c) sending a POST request and receiving HTTP response

Project code as: "*response = requests.post(url, json=data[1])*"

'requests' is a Python third-party library for sending HTTP requests. POST requests are usually used when the client submits data to the server to perform various operations or create new resources. The *requests.post()* method in this line of code accepted two parameters, one was the URL address to be requested, the other was the requested data. Response variable can store the HTTP response returned by the *requests.post()* method;

d) checking the status code of the HTTP response object

Project code as: "*if response.status_code == 200:*
result = response.json()
prediction = result['prediction']
print('Prediction result:', prediction)
else:
print(' Request failed:', response.status_code)"

The HTTP status code is a three-digit numerical code returned by the server in response to a request, it was used to indicate the result of the request's processing. In particular, status codes starting with '2**' indicate successful responses, and specifically, a status code of 200 means the request has been successfully processed. Therefore, authors checked whether the status code of the HTTP response object was 200, if it was, printed the prediction result from the data returned by the server, if not, then printed the current status code received, which will help debugging the project.

2. Designing a Flask program:

a) importing modules

Project code as: "*from flask import Flask, jsonify, request*
from joblib import load
import pandas as pd"

The authors introduced three classes from Flask framework to build web applications, where '*Flask*' is the main class of it, designed to create instances of Flask applications, '*jsonify*' is a helper function provided by the Flask framework that converts a Python object into a JSON-formatted HTTP response, and '*request*' can represent an HTTP request initiated by a client. The 'joblib' library is a library for serializing and deserializing python objects in python. The 'load' function can load python object from disk which has been saved to disk by 'dump' function. The 'pandas' third-party library has also been imported into the current code environment, providing extensive functionality for handling manipulation of tabular data (such as csv);

b) declaring object

Project code as: *app = Flask(__name__)*

After imported all the relevant modules, one can start creating a Flask instance, the line of code created an instance of a web application using the Flask framework. Instance '*app*' can be used to define different routes, handle HTTP requests and errors, set configuration options, etc. The "*= Flask(__name__)*" part was for initializing 'app' variable, which was created by calling the constructor from 'Flask' class;

c) setting up route

Project code as: *@app.route('/predict', methods=['POST'])*
def predict():

Routing is the process of defining a mapping between a URL path and its corresponding view function in Flask application. '*@app.route*' is a decorator that defines a route in Flask. Routing in Flask is usually done by using a decorator, which specifies a URL path and associates it with a view function. The URL path of this project's decorator was '*/predict*', implying a client needs to send a request to the '*/predict*' path for triggering its bound view function *predict()*. '*methods=['POST']*' means that only POST method can trigger this route when a client sends its HTTP request.

In Flask framework, the routing module of Werkzeug library is responsible for implementing URL parsing. Different URLs correspond to different view functions. The routing module parsed URL from the request, matched it to its corresponding view function and executed this function to generate a response message. A view function is a python function in the web framework that handles an HTTP request and returns an HTTP response;

d) data processing. Data processing refers to the process of parsing and handling the received request data in a Flask application. The purpose of data processing is to extract necessary information from the request for use in business logic.

Project code as: *data = request.json*
received_data = pd.DataFrame(data)
received_data_series = received_data.iloc[0]

In the 'predict' view function, authors first stored the JSON data from the HTTP request in the variable named 'data', the structure and content of 'data' corresponds to the JSON data sent in the HTTP request. The variable '*received_data*' has the same content with variable 'data', but its format was converted into DataFrame, the first row of variable '*received_data*' was extracted from its DataFrame table and stored in a Series object called '*received_data_series*'. '*iloc*' is the method provided by the pandas library for locating the data;

e) responding business. Business response refers to the procedure in the view function where the received request data is used for business logic and generate an HTTP response to be returned to the client. The view function is the primary place in the Flask program where business logic is handled, and it ultimately returns the generated response

Part of project code as: *def predict():*
prediction = predict_with_model(received_data_series)
return jsonify({'prediction': prediction})

After received the processed data, a function named '*predict_with_model*' was called and data was input to this function for prediction operation. '*predict_with_model*' was a function defined by authors in the code, which contained the encapsulated machine learning model. After the data has been input, the prediction method of the model will be called to make a prediction, its generated result was stored in variable 'prediction'. The obtained prediction result was dictionary data, which was sent back to the client as a response object after being converted into JSON format by '*jsonify()*';

f) starting up service

Project code as: *app.run(host='0.0.0.0', port=5000)*

'*run()*' is the method used to start the local development server in the Flask application. '*host*' is the listening address of the specified server, when its value equals to '0.0.0.0', the server will listen to all useful network interfaces, making the Flask application accessible to both local and public networks. The value 5000 for '*port*' is the default port number of the Flask development server. Combining all the above code, the experimental result of running this line was:

The client initiated the HTTP request by making a request to the URL, the server listened on the *host='0.0.0.0'* address and *port 5000*, the client's request reached the server and matched the route with the path '*/predict*', which triggered its bound view function to process the request, the view function encapsulated the result of processing, that is, the generated response prediction, into an HTTP response, which was sent back to the client, who received the HTTP response from the server, and the prediction result was displayed.

3. Configuring and using the EMQX Broker

Although the HTTP request is already capable of returning the prediction results to the original client and displaying the results, but it is not possible to save the prediction results or distribute them to other clients, in order to achieve this function, the authors used the EMQX broker, through the MQTT protocol to achieve the prediction results published based on the topic filtering:

a) importing module

Project code as: *import paho.mqtt.client as mqtt*

The authors imported '*paho.mqtt.client*' library into the current code environment, which is a Python implementation of an MQTT client used to communicate with an MQTT Broker;

b) setting the connection parameters for the MQTT Broker server as shown in Tab. 1;

c) initializing the MQTT client

Project code as: *mqtt_client = mqtt.Client()*

Typically, when using the MQTT protocol, it is necessary to create an MQTT client object. The authors created an MQTT client by using the MQTT client's class '*Client*', which was used for communicating with EMQX Broker. '*mqtt_client*' was the name of a variable that was used to store the created MQTT client object, afterward, it can be used to invoke methods of the MQTT client;

d) connecting to EMQX Broker

Project code as: *mqtt_client.connect(broker_address, broker_port)*
mqtt_client.username_pw_set(broker_username, broker_password)

The authors established connection to the EMQX Broker by calling the '*connect()*' method based on the previously set parameters, broker address and broker port. Since EMQX Broker requires authenticated the user, the '*username_pw_set()*' method of '*mqtt_client*' class was called, which was used to set the username and password required to connect to the EMQX Broker, with the parameters that have been previously set. After a successful connection, the MQTT client implemented by Python code can be started publishing and subscribing to messages;

e) starting the client loop

Project code as: *mqtt_client.loop_start()*

In the MQTT protocol, the client needs to continuously maintain communication with the MQTT agent (usually an MQTT server) to handle message subscriptions and publications in real-time. To achieve this purpose, the MQTT client needs to run a loop in a separate thread, typically starting the loop immediately after completing the connection operation. This ensures that the client can receive and process MQTT Broker-sent messages in real-time. '*mqtt_client.loop_start()*' method started a MQTT client's loop in a new thread. This loop ran in the background and did not block the execution of the main thread, allowing the MQTT client to process other operations simultaneously;

f) publishing prediction result to designated topic

Project code as: *mqtt_client.publish(topic, prediction)*

4. Displaying results:

a) client page. The client page realized by Java in this project was shown in Fig. 3, it can be seen that participants can choose from three functions: "record your voice", "speech to text" and "prediction", of which, speech to text is an example of data preprocessing methods, developers can supplement or modify this function according to the actual design requirements. After the participant has recorded his/her voice message to the cell phone, the cell phone will save the voice message and carry out the data preprocessing and feature extraction operation, when the participant triggered the "prediction" function, the cell phone as a client sent the corresponding feature file to the server. After received the prediction result from the server or EMQX Broker, the cell phone parsed it and displayed the result on the page, as shown in the figure;

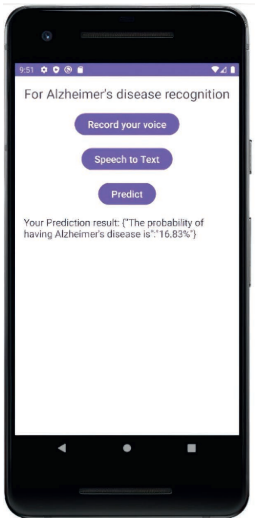

Fig. 3. Client-side page display

b) EMQX Broker page. The page of the EMQX Broker after being successfully connected by devices was shown in Fig. 4. It can be seen that there were 2 devices connected to the broker, in this project implementation, the cell phone and the MQTTX client, with the number of subscribed topics being 1;

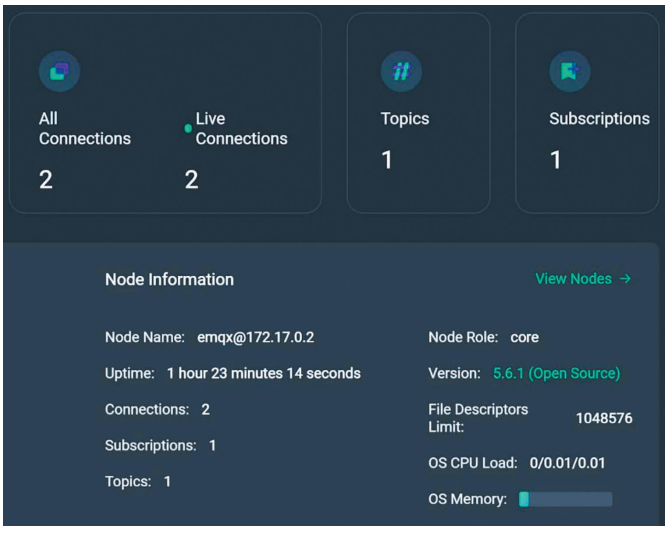

Fig. 4. EMQX

c) MQTTX page. Since in this project, the predicted result that displayed on the mobile phone page was parsed from the HTTP response. To examine the feasibility of distributing the prediction results using the EMQX Broker, the authors used MQTTX to simulate a new client, whose page display was shown in Fig. 5. MQTTX is an open source, cross-platform MQTT 5.0 client tool built on Electron, which is primarily used for debugging and testing MQTT communication. As can be seen from the figure, after subscribed to the topic "*/flask/predict*", MQTTX, as another client, received the prediction results transmitted via the MQTT protocol.

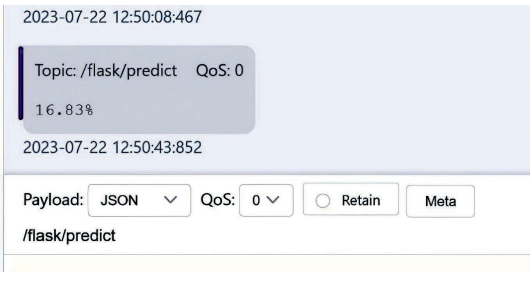

Fig. 5. MQTTX

Conclusion

1. The article presents the model and structure of the Internet of things network, which are designed to determine the presence of Alzheimer's disease in patients, using the author's developments in the technology of IT diagnostics of this disease. An instance of the network project was created using the Flask framework, described how to create a WSGI application from app.py to process client requests. The process of using the *app.route* decorator method of an instance of the Flask program to create viewing functions that are performed when accessing routes was explained, thus, business logic was implemented for fart information and result display functions.

2. It also describes how to use the JSONIFY function to return data in JSON format to the client based on a call to the view function. It explains how to start the Flask web application service using the *app.run()* method. To expand the scope of providing prediction results, the EMQX Broker was used to facilitate data transmission over the MQTT protocol. The layout of the client page of this project was developed using Java, which included displaying the prediction results of Alzheimer's disease on smartphones of a patient and a doctor.

References

1. Dong L. (2023) Application of Internet of Things Technology in Hospital Information Construction. *Shihezi Science and Technology*. (3), 77–78.
2. Vishniakou U. A. (2023) *Specialised IoT Systems: Models, Structures, Algorithms, Hardware, Software Tools.* Minsk: Belarusian State University of Informatics and Radioelectronics.
3. Vishniakou U. A., Chuyue Yu. (2023) Using Machine Learning for Recognition of Alzheimer's Disease Based on Transcription Information. *Doklady BGUIR*. 21 (6), 106–112. http://dx.doi.org/10.35596/1729-7648-2023-21-6-106-112.
4. Jiafa C., Yujing H. (2022) Application of Flask Framework in Data Visualization. *Fujian Computer*. 38 (12), 44–48.
5. Grinberg M. (2018) Flask Web Development: Developing Web Applications with Python. *O'Reilly Media, Inc.*

Authors' contribution

All authors have equally contributed to writing the article.

Information about the authors

Vishniakou U. A., Dr. of Sci. (Tech.), Professor at the Department of Infocommunication Technologies, Belarusian State University of Informatics and Radioelectronics (BSUIR)

Chuyue Yu, Postgraduate at the Department of Infocommunication Technologies, BSUIR

Address for correspondence

220013, Republic of Belarus,
Minsk, P. Brovki St., 6
Belarusian State University
of Informatics and Radioelectronics
Tel.: +375 44 486-71-82
E-mail: vish@bsuir.by
Vishniakou Uladzimir Anatol'evich