Distributed Storage System

Abhijith Remesh
Email: abhijith.remesh@st.ovgu.de
Baizil MD
Email: manish.rama@st.ovgu.de

***Abstract*—Manufacturing companies use several software systems like ERP and MES so on, to support their internal production and business processes. The companies now tend to automate and digitalize their enterprise processes, thus transforming to Industry 4.0. This digital integration of production facilities enables the companies to get new information and useful knowledge about their processes in real time and thus take immediate actions. This project deals with the development and implementation of a software system for the use case 'Inventory level management'. The current inventory levels detected is stored in a distributed software architecture so that it provides high flexibility and scalability of the inventory management system. The inventory management system with a distributed storage architecture is realized using industrial weight scale, Raspberry Pi, MySQL and Node-RED. The number of weight scales and Raspberry Pi's can be scaled up with all the Raspberry Pi interfaced weight scales reporting their respective inventory levels to a single storage system which is developed in MySQL database. The Node-RED is coupled with the MySQL database displays the inventory levels to the user through the Node-RED dashboard and the user is able to interact with the Node-RED dashboard and also able to view the inventory levels of different weight scales. Several error correction techniques are employed to detect any discrepancies regarding the placement of the articles on the weight scale by analyzing the obtained weight of the article. This is done to ensure that the correct pallet is placed on the weight scale, the pallet does not contain mix of different articles which are undesirable and cause errors.**

I. INTRODUCTION

THE project deals with the design, development and implementation of a software system for the management of inventory levels which follows a distributed storage architecture. The software system also needs to scale up flexibly and efficiently on increasing the number of weight scales. This distributed storage system for the management of inventory levels of warehouse components is realized with an industrial weight scale, PCE-BSH-6000/PCE-BSH-10000 , a micro-controller, Raspberry Pi, MySQL database, Node-RED and a set of warehouse articles like screw, clothespin and so on. The Message Queuing Telemetry Transport (MQTT) protocol is used for the data communication which is a machine-to-machine and Internet Of things (IoT) connectivity protocol. It uses light weight publish-subscribe messaging transport. The application uses two MQTT clients, one at the Raspberry Pi end, and the other at the local machine end and a MQTT broker at the local machine end. The interface between the weight scale and the Raspberry Pi is serially through a Universal Serial Bus (USB) cable.

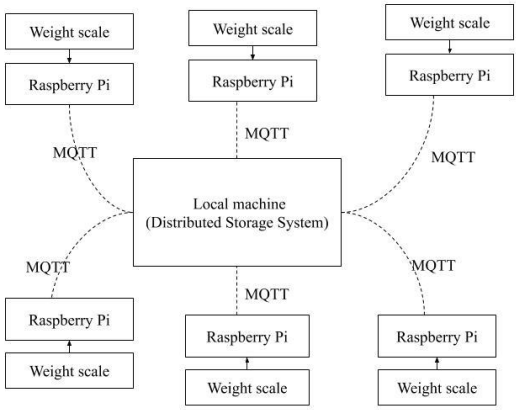


Fig 1. High level architecture

The diagram depicts a very high level architecture of the application in a abstract manner. The solid line between the weight scale and the Raspberry Pi indicates the serial connection. The dotted lines between the local machine and the Raspberry Pi indicates the wireless connection over MQTT protocol.The weight scale interfaced with the raspberry pi forms one unit and that unit can be scaled up in number, with each unit publishing its data over MQTT to a single storage system which is distrbuted across all the units.

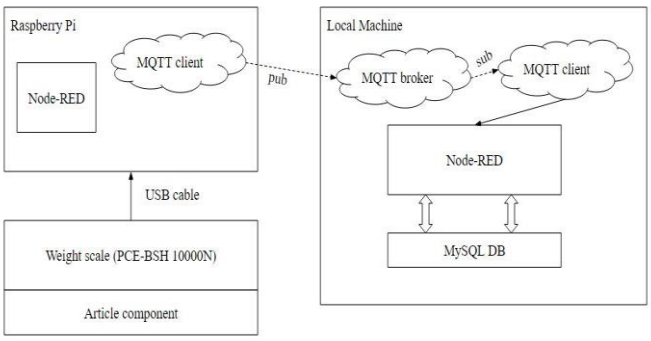


Fig 2. Detailed architecture

The figure depicts a more detailed architecture of the system, looking into the specifics and details of the system. As specified earlier, the industrial weight scale is connected to the Raspberry Pi serially through the Universal Serial Bus (USB) cable. The

Raspberry Pi acts as the MQQT client, publishing the data on a specific topic over the MQTT to a MQTT broker. The local machine where a MQTT broker and a client is installed, receives the data, stores the data in a MySQL database residing in the local machine and displays the current inventory levels shown by the weight scale on a web based user interfaced using Node-RED. The Raspberry Pi comprises of a MQTT client instance and a Node-RED instance for parsing the weight data and for monitoring the connection status of the weight scale, checking if the weight scale is connected to the Raspberry Pi or not. The local machine consists of a Node-RED instance, MQTT broker instance, MQTT client instance, MySQL database instance running on it. The number of weight scale interfaced Raspberry Pi can be scaled up so that all the respective MQTT clients of n weight scales can publish data to a common single storage residing at the local machine which seems distributed across all MQTT clients.

The application can be scaled up in two approaches: the first approach is to connect four weight scales to a single Raspberry Pi as a single unit, in a 4:1 ratio and then to increase the number of such units. The second approach is to connect one weight scale to one Raspberry Pi as a single unit, in a 1:1 ratio and then to increase the number of such units. The feasibility study of the two approaches has been analyzed and the second approach appeared to be more feasible relatively. The detailed feasibility study is discussed later in detail.

II. COMMUNICATION FLOW

THIS chapter discusses the communication flow of data in the application, from weight scale placed in a remote location to the users at another location in a chronological manner.

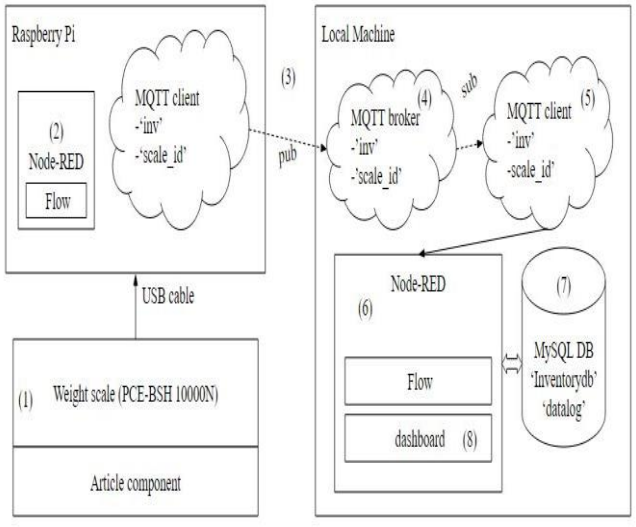


Fig 3. Communication flow

- The weight scale detects the weight data whenever an object is placed on the weight scale. That is, the weight scale captures any change in its weight readings.

- The Raspberry Pi interfaced serially with the weight scale through USB cable collects this weight data and connection status, parses the weight data and computes the unique scale id of the device connected. The scale id is the combination of serial number of the Pi and the device number generated on device connection to the Raspberry Pi. This is done by the Node-RED instance running at the Raspberry Pi end.

- The MQTT client instance running at the Raspberry Pi end publishes the 'weight information' onto a MQTT topic 'inv' to the MQTT broker and publishes the 'connection status, 'scale id' onto a MQTT topic 'scale_id' to the MQTT broker.

- The MQTT broker instance running at the local machine receives this data and looks for any other MQTT clients subscribed onto these topics.

- The MQTT client instance running at the local machine which is subscribed onto these topics receives the 'weight information', 'connection status, 'scale id' respectively.

- These data from the MQTT client is captured by the Node-RED instance running at the local machine and is parsed and processed in various manners.

- The local machine contains two MySQL database instances, one for storing the master data information of the inventory components and another for storing the log information of the inventory components. Both these databases stays centralized and common for all the weight scales irrespective of the number of weight scales being used.

- The Node-RED instance running at local machine is integrated with the MySQL databases. This enables the user to select the article from the database, add new article to the database, and delete articles from the database and save the log information to the database.

- The Node-RED instance residing at the local machine also acts as the visualization platform and performs error correction techniques to detect any discrepancies in the article placed on the weight scale by running a series of python scripts.

III. DESIGN

THIS chapter discusses the different use cases available to the user to interact with the application using use case diagrams and the activity diagrams which tells the work flow of activities being operated at different levels namely user, application and database.

- Raspberry Pi end use case diagram
- Local machine end use case diagram
- Raspberry Pi end Activity Diagram
- Local machine end Activity Diagram

A. Use case diagrams

Use case diagram in general, describes the user interaction and the relationship with the system, how the user is involved in different use cases of the system. The use cases are represented by ellipses.

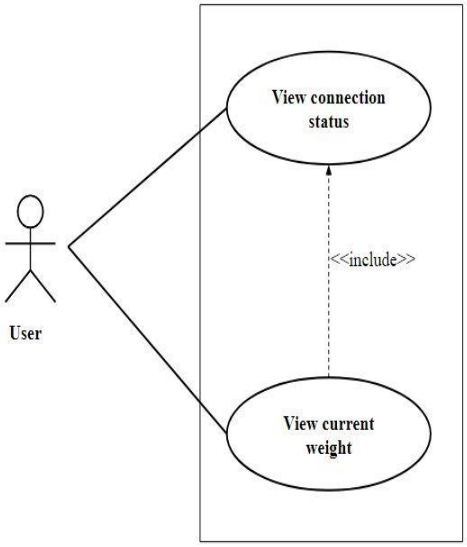


Fig 4. Raspberry Pi use case diagram

The above figure depicts the use case diagram at the Raspberry Pi end. As specified in the use case diagram, the user has only two use cases:

- 'View connection status' is to view the connection status which indicates if the scale is connected to the Raspberry Pi or not.
- 'View current weight' is to view just the current weight based on the article which is placed on the weight scale. The 'View connection status' exhibits a include relationship with the other use case 'View current weight' as the latter depends on the former. The current weight can only be viewed if there is an established connection between the weight scale and the Raspberry Pi.

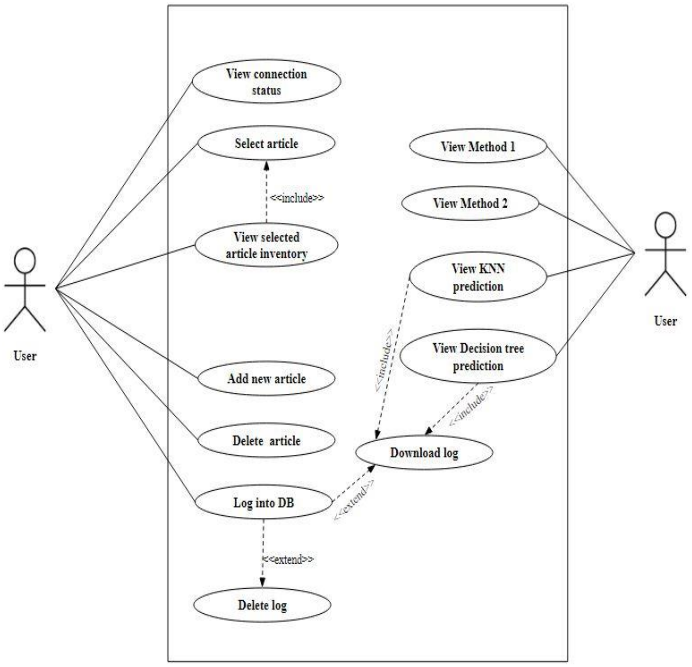


Fig 4. Local machine use case diagram

The above figure depicts the use case diagram at the local machine end. The different use cases available to the user are listed below:

- 'View connection status' is to view the connection status if the weight scale is connected to the Raspberry Pi or not and if connected, the user is also able to show the unique scale id of the weight scale.
- 'Select article' use case to enable the user to select an existing article name from the 'inventorydb' database (master data) so that the respective article is placed on the weight scale by the user.
- 'View Selected article inventory' use case to enable the user to view the article number, unit weight, obtained weight and number of pieces of the selected article. This exhibits a include relationship with the 'Select article' use case because the user can only view these article inventory details if and only if the article is selected in the first place.
- 'Add new article' use case which lets the user to add new article information into the 'inventorydb' database (master data) specifying the article name, article number and then scanning the unit weight of the article to be added.

- ‘Delete article’ use case which allows the user to delete any existing article in the ‘inventorydb’ database (master data).

- ‘Log into DB’ use case which lets the user to store the log information of articles placed on the weight scale and their respective inventory levels on ‘datalog’ database (transactional data).

- ‘Delete log’ use case which lets the user to delete all the inventory log information present in the ‘datalog’ database (transactional data). This use case exhibits a extend relationship to the ‘Log into DB’ as it acts as an extension to the ‘Log into DB’ use case.

- .’Download log’ use case which lets the user to download the inventory log information present in the ‘datalog’ database (transactional data) as a csv file to the local machine. This use case also exhibits a extend relationship to the ‘Log into DB’ as it acts as an extension to the ‘Log into DB’ use case.

- ‘View method 1’ use case which displays the user if the article kept on the weight scale is the intended correct article without any discrepancies. This is done by checking if the obtained weight captured lies within the permissible weight ranges.

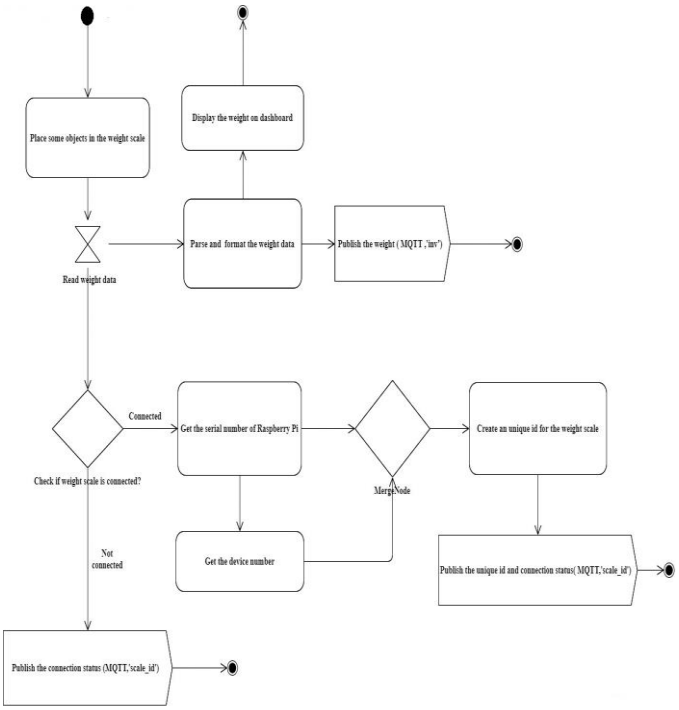
- ‘View method 2’ use case which displays the user if the article kept on the weight scale is the intended correct article. This is done by comparing the obtained weight with the product of unit weight and the obtained number of pieces.

- ‘View KNN prediction’ use case which displays the article predicted by the KNN model trained on the csv file (inventory log information) such that it indicates if the predicted article and the article actually kept on the weight scale are same or different.

- ‘View Decision tree’ use case which displays the article predicted by the decision tree model trained on the csv file (inventory log information) such that it indicates if the predicted article and the article actually kept on the weight scale are same or different.

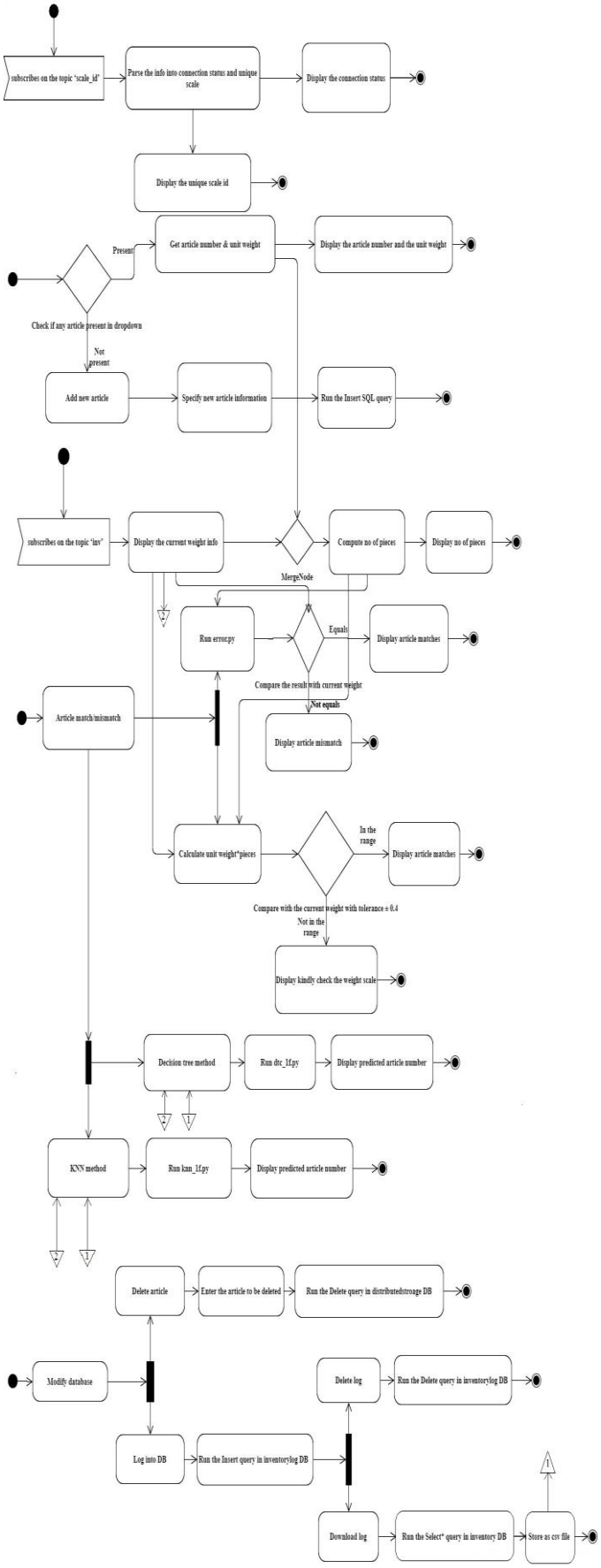
B. *Activity Diagrams*

Activity Diagram, in general represents the dynamic aspects and operations of the system there by specifying the work flow of stepwise activities and actions from one activity to another activity. It resembles the functionality of a flow chart.



- Download a copy of Raspbian OS and Ubuntu MATE for raspberry pi 3B+.
- Copy and then replace the following files from Raspbian [10] OS to Ubuntu MATE: *bootcode.bin, fixup.dat, start.elf, bcm270-rpi-3-b-plus.dtb, kernel17.img,* all contents from */lib/modules/4.9.80-v7+* and */lib/firmware/brcm/.*
- Flash the new OS to the SD card via Etcher and perform the similar steps for Ubuntu MATE as described on www.scionlab.org.

Instead of command prompt, Putty has been used to connect with the Raspberry Pi and then to run and host the SCION [1] network from the Raspberry Pi[2]. Thus, the SCION [1] network is established on the IoT device which acts as the SCION [1] server AS for the proposed IoT application.



IV. IMPLEMENTATION

This section explains in detail about the several hardware components and software utilities used in the application. The hardware components include the following, (i) Raspberry Pi 3B model [2] which acts as the primary micro-controller unit, (ii) PCE-BSH-10000N as the industrial weight scale. The software utilities used includes (i) MQTT as the IoT protocol, (ii) MySQL databases as storage platform and (iii) Node-RED as the flow based development and visualization tool. This sections explains how these hardware components and software utilities are interfaced and integrated with each other to form a standalone application which follows a distributed storage architecture.

A. Hardware components specifications

- Raspberry Pi 3B model

The application uses Raspberry Pi 3B model as the micro-controller unit. The Raspberry Pi 3B model contains the four USB ports so that a maximum of four weight scales can be connected to it. The feasibility study suggests to only connect one weight scale to one Raspberry Pi to avoid complexities and inefficiencies. The feasibility study of the two scalability approaches will be discussed in detail later.

- PCE-BSH-10000N weight scale

The application uses PCE-BSH-10000N as the industrial weight scale. It has a range of 1kg with a resolution of 0.2g and accuracy tolerance of 0.6g.The data interface is through USB and it can be powered through a battery or a main adapter. The interface of the weight scale is configured as “Stb” so that the data transmission only happens once the reading is stable. The USB output data format is as follows: Baud rate: 9600, data bit: 8, parity: None, stop bit: 1, Code: ASCII

- Warehouse articles

Three different sample articles have been used to place on the weight scale and to test the functionality of the system. Each of the article has the following attributes namely article name, article number and unit weight. This forms the content for the inventory master data table (‘inventorydb’) of the MySQL database.

Article name	Article number	Unit weight
731308	Waschenklammer	4g
708763	Tintenpatrone	1.4g
700555	Screw	3g

B. *Software utilities specifications*

- MQTT

The IoT, M2M protocol, MQTT is used for establishing data communication between the local machine and the Raspberry Pi. This uses a publish/subscribe architecture and uses less battery power. As specified earlier, the Raspberry Pi acts as the MQTT client (publisher) publishing the weight, connection status and unique scale id information on the topics 'inv' and 'scale_id' respectively to the MQTT broker which is running in the local machine. The MQTT client (subscriber) in the local machine which is subscribed onto these respective topics receives these information correspondingly from the MQTT broker. As a result, the local machine performs the role of MQTT broker and client whereas the Raspberry Pi performs the role of MQTT client.

- MySQL

MySQL database system has been used as the storage platform for storing the master data and the transactional data.

Two MySQL databases have been created namely 'distributedstorage' and 'inventorylog'.

The 'distributedstorage' database contains the table 'inventorydb' where the master data of warehouse articles (inventory master data) are stored.

id	article_number	article_name	unit_weight

In order to create this database and its respective table, the SQL script namely 'createinventorydb.sql' needs to be executed in the MySQL workbench.

The inventory master data information can be inserted in the 'inventorydb' table of 'distributedstorage' database in two ways, either by running a SQL script, 'insertmasterdata.sql' or by adding the article details through the Node-RED dashboard.

The 'inventorylog' database contains the table 'datalog' where the time stamped inventory log information is stored which forms the transactional data. The format of the table is as follows:

time	scaleid	article	unitweight	obtainedweight	count

This log information is saved on a csv file on user demand and is used for training a KNN model and a decision tree classifier which will predict the article based on just the 'obtainedweight' parameter. These prediction results are used in the error correction techniques which checks if the article predicted by these algorithms and the article actually kept on the weight scale are same or different.

- Node-RED

Node-RED, being a flow based development and visualization tool has been used which in turn interacts with the MQTT client/broker instances and the MySQL database instances. Both the Raspberry Pi and the local machines has separate Node-RED instances running. The deployment of flows follows a specific sequence such that the flow at the Raspberry end is to be deployed first and then the flow at the local machine end is deployed next.

The Node-Red flow at the Raspberry Pi end performs the following tasks, the task of reading the serial data (detected current weight data) from the Raspberry Pi, the task of retrieving the serial number of the Raspberry Pi (by running the serial.sh) and the randomly allocated device number (by running the devicenum.sh) whose combination forms the unique scale id, the task of publishing the weight data on the topic 'inv', the task of publishing the connection status/ scale id on the topic 'scale_id' and finally, the task of displaying the current weight data on the Node-RED dashboard at the Raspberry Pi end. The flow is available as pi.json file.

The Node-RED flow at the local machine end performs the following tasks: the task of subscribing on the 'inv' topic to receive the obtained current weight, the task of subscribing on the 'scale_id' topic to receive the connection status and the scale id information, the task of dropdown functionality to select articles which in turn passes a SELECT query to the MySQL database table inventorydb', the task of adding new article details which in turn passes a INSERT query to the MySQL database table 'inventorydb', the task of dropdown functionality to delete existing articles which in turn passes a DELETE query to the MySQL database table 'inventorydb', the task of displaying the selected article's number, unit weight and its obtained weight, its count (number of pieces) information, the task of logging (transactional data) article number, unit weight, obtained weight, timestamp, scale id and count into the MySQL database table 'datalog' through an INSERT query on user demand, the task of deleting the log information from the MySQL database table 'datalog' through a DELETE query on user demand, the task of writing the log information in the MySQL database table 'datalog' into a csv file on user demand and finally, displaying the article prediction results from the Method 1, Method 2, KNN method and decision tree method by running a series of python scripts namely knn_1f.py, dtc_1f.py.

V. ERROR CALIBRATION

This chapter discusses about the establishment of client server communication over the SCION [1] network. The sequence of operation is as follows:

- Initially, the SCION [1] AS node installed on the Raspberry Pi [2] initiated by specifying a particular port number and respective detected Internet Protoclol address (IP) of the Raspberry Pi [2]. This acts as the server of our application.
- Subsequently, the SCION [1] AS node installed within the virtual machine on personal computer is started by specifying a particular port number and respective IP address of the personal computer.
- As a result, the client server communication is enabled and the client should be able to receive the responses from the server.

The basic implementation of the above discussed process is explained below by setting up a basic SCION [1] server and client transmitting raw data packets between them. This is done for testing purposes.

- A basic SCION [1] server go script is developed which send sample data packets to SCION [1] client. By running the below command as below, the SCION [1] server gets started.

  "*go run Scion_server.go -s 19-ffaa:1:161,[192.168.137.185]:30102*".

- A basic SCION [1] client go script is developed which obtains the data packets from SCION[1] server. By Running the below command as below, the SCION [1] client starts.

  "*go run Scion_client.go -s 19-ffaa:1:161,[192.168.137.185]:30102*".

Validation and verification of the communication is done by the successful reception of the raw data values at the client SCION [1] AS node.

A. Method 1

This chapter discusses the slight modification in the SCION[1] client go script to receive the JSON object encapsulating product data, weight values and UID from the hosted SCION[1] server AS node. Apart from that, the script also performs the hosting of a Hyper Text Transfer Protocol (HTTP) socket which is utilized for exposing the JSON object over a HTTP connection so that the visualization tool Node-Red [6] can use a http get request to access the JSON object and then visualize the same. The script is executed by running the below command as shown.

- "*go run Scion_client.go -c 19-ffaa:1:bfa,[192.168.1.130]:30102 -s 19-ffaa:1:161,[192.168.137.185]:30102* "

Where "*30102"* indicates the port number,"*19-ffaa:1:bfa,"* indicates the SCION [1] client AS node, "*[192.168.1.130]"* indicates the client IP address(Personal Computer), "*19-ffaa:1:161*" indicates the SCION [1] server AS node and "*[192.168.137.185]"* indicates the server IP address(Raspberry Pi [2]). The complete script can be found in the appendix section.

B. Method 2

This section discusses the integration of the weight acquisition go script with the basic SCION [1] server go script developed earlier. As a result, the final SCION [1] server go script "*Weight_server_full_rv2*.go" performs the below functionalities.

- Act as a TCP Socket client and fetch UID values from Python TCP socket server whenever the card is swiped with RFID sensor.
- Fetch appropriate weight values from load cell [4]-HX711 [5] sensor.
- The product data like name, expiry date, capacity and so on are also hard coded for each UID in the SCION [1] server go script.
- Amalgamate product data associated with respective UID and their corresponding weight as one compact JSON object and send it to requesting client SCION [1] AS node.

This script upon execution produces a JSON object with product UID, product data and the respective real time weight readings at the SCION[1] client AS node. The script is executed by running the below command as shown.

- "*go run weight_server_full.go -s 19-ffaa:1:161,[192.168.137.185]:30102*"

Where "*30102"* indicates the port number, "*[192.168.137.185]*" is the detected dynamic IP of the Raspberry Pi [2] and "*19-ffaa:1:161*" is the SCION[1] AS node installed on the Pi. The complete script can be found in the appendix section.

C. KNN method

THIS chapter discuses the chronological sequence of data flow when a request is made from the client SCION [1] AS end that is, from the Node Red dashboard end.

The below block diagram depicts the complete components involved in the scope of application. The part (A) is within the scope of this report and the part (B) corresponds to the other report. When a request is initiated from the client SCION [1] AS node, the chronological order of data flow is as follows:

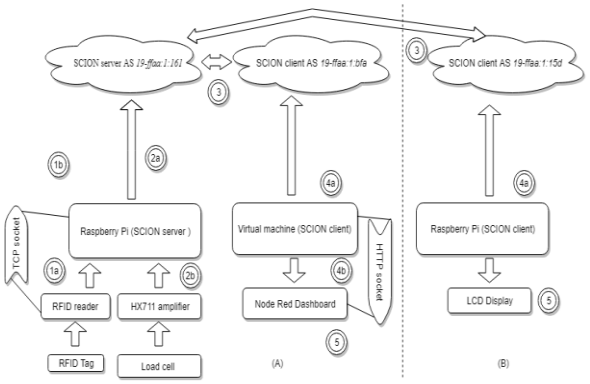


Fig.4:Client-server data flow

- The Raspberry Pi [2] initiates the RFID reader to read the UID of the scanned RFID tag if detected.
- The Raspberry Pi [2] hosts a TCP socket server bound to a specific port. The data packet containing the UID is converted to JSON format and then send as bytes to this TCP connection and starts listening for any incoming request. Whenever this TCP server receives a request, it responds with this data packet.
- Then, the Raspberry Pi [2] runs and connects to the SCION [1] network as a SCION [1] AS server node by identifying the SCION [1] server address, scion path and dispatcher path. Once identified, the SCION [1] server starts listening over a UDP connection for any incoming requests. Then a TCP connection is established with that specifc port by the SCION [1] server and the HX711 [5] sensor is also initialized. Whenever any incoming request is encountered from the SCION[1] AS client node, it sends a sample data packet as request to the TCP socket over a TCP connection. It gets back the UID as the TCP response from the TCP server.
- If the UID obtained is a valid existing one, then the real time weight reading from the HX711[5]-load cell [4] unit is fetched and then appended to the data packet. Moreover the UID, current number, current time are also appended to the data packet which is then converted into JSON format.
- This data packet is then sent to the respective client SCION[1] AS node over the SCION[1] network.
- The client SCION[1] AS node now receives the data packet as a JSON object encapsulating the UID, product name, product expiry date, current time, unit weight, capacity, current number and weight.
- The client SCION [1] AS hosts a HTTP socket on a specific port and establish a HTTP connection and starts listening. Then, it exposes this data packet as JSON object over this defined port. Whenever, this HTTP server encounters a request from any browser or user, it sends with this JSON object as response.
- The visualization tool, Node-Red [6] dashboard makes a HTTP GET request to this defined port, and gets this JSON object as response which is then processed, parsed and displayed using several node functions available in the Node-Red[6].

The other section is not discussed in this report as it is not within the scope of our application. This is how the data flow happens between the SCION [1] server AS node and the SCION [1] client AS node. The SCION[1] server AS node acts as a client with respect to the TCP server and it acts as a server for the [1] client AS node at the same node. Similarly, the SCION [1] client AS node acts as the client with respect to the SCION [1] server AS node and acts as a server for the Node-Red [6] dashboard simultaneously. It can be said that both the SCION [1] server and SCION [1] client AS node perform dual roles, that of a client and server.

D. Decision tree method

THIS chapter discuses the chronological sequence of data flow when a request is made from the client SCION [1] AS end that is, from the Node Red dashboard end.

The below block diagram depicts the complete components involved in the scope of application. The part (A) is within the scope of this report and the part (B) corresponds to the other

VI. DEPLOYMENT

THIS chapter explains how the graphical user interface has been implemented at the client SCION [1] AS node. The flow-based, visual programming development tool, Node-Red [6] has been employed for this purpose. Node-Red [6] requires data in the JSON Object format so that the desired flow can be developed using nodes which is why, the data packets are transferred over the SCION [1] network as JSON objects. The tool makes use of a “http” input node which performs a HTTP GET function to access the JSON object which is why, the SCION[1] client go script has been modified to incorporate a HTTP socket server being hosted at a specific port. The data packet, in particular the JSON object being received at the client SCION [1] AS node is exposed over a HTTP socket server which is then accessed by the Node-Red [6] HTTP GET input function node. This JSON Object is then parsed into different and separate data using the other node functions available on the Node-Red [6] platform. The different main node functions used for the visualization are as follows.

- A “*http*” input type node is used to access the JSON object (data packet) hosted on a HTTP server at port *4000*. The node is configured accordingly to use a *GET* method at the Uniform Resource Locator (URL) (port 4000). The JSON object is received as a message payload at the Node-Red [6] end.
- Several change type nodes are used to parse several data information from the entire message payload(JSON object). Change type nodes are used to process and parse the temperature, humidity, product name, expiry date, capacity,

current number, weight information from the message payload.

- Several dashboard nodes like "*text*", "*gauge*", "*chart*" are used to visualize all these parsed separate information as a gauge, wave, level default text types of visualization.

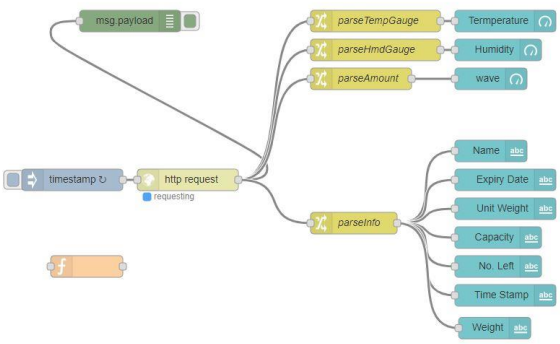


Fig.5:Node-red flow based diagram

This acts as a Graphical User Interface for the users so that they can monitor the inventory status of products in real time which eliminates the need to do frequent physical inventory inspection and take business decisions accordingly. This also facilitates the users to take business decisions either to replenish the decreasing stock or to grant discounts in real time.

VII. RESULTS

THIS chapter describes the results observed after the completion of relevant development tasks during the entire course of the application development which involves the following:

- Rigid assembly with the appropriate placement of the load cell [4], RFID reader module and the Raspberry Pi [2] to form a stand-alone "*smart pallet"* unit. The assembly comprises of a mounting plate which acts as the platform for keeping the RFID tag attached product pallet and a placeholder unit which contains the RFID, load cell [4] connected Raspberry Pi[2].

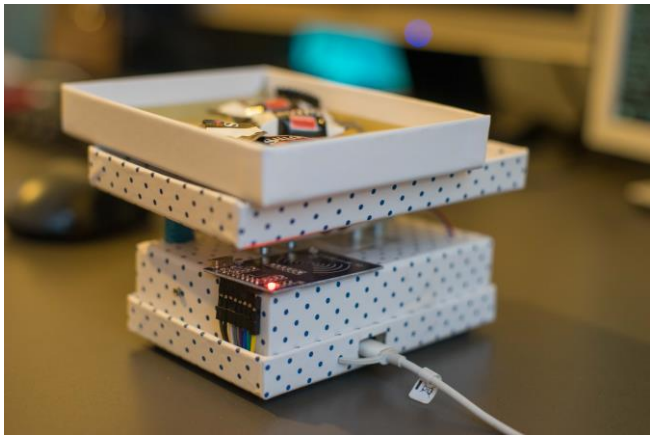


Fig.6:Smart pallet assembly

- Reception of raw weight readings from the load cell [4]-Hx711 [5] sensor unit which changes proportionally with the applied load.

Fig.7:Raw weight readings

- Reception of the JSON object at the client SCION[1] AS node wherein it hosts over a HTTP socket on port 4000 which contains the UID of the scanned tag, the product name, weight reading, product expiry date, pallet capacity and so on. The weight readings are dynamics and it keeps on varying as per the applied load.

Fig.8:JSON data at port 4000

- Visualization of the product details, especially the real time weight readings on the Node-Red [6] browser platform. The Dashboard visualization in the form of gauges, level represents the changes in weight graphically in real time.

Fig.9:Node-Red dashboard visualization

VIII. CONCLUSION AND FURTHER WORK

This scientific work focuses on the implementation of an IoT application over the SCION[1] network. The scope of the scientific work in general involves the establishment of a client server communication on the SCION[1] network by employing one SCION [1] AS node as the server and the other SCION [1]AS node as the client on an IoT application context. We came up the concept of real time inventory management of products as an IoT use case which is then deployed over the next generation network architecture, SCION[1]. The application which we have realized is just an initial development and posses wide scope of developmental progress in several application contexts. This application can find it's usage in wholesale, retail, manufacturing, warehouses and logistics domains wherever an inventory status and management of products are required on it's further development like using load cell [4] of wide range. The application can extend it's functionalities to automatic discount generation in real time, automatic order generation for stock replenishment in real time, monitor the sales data of the product which can further used for reconciliation with the point of sales report. All these scopes could been explored if there were no time constraints.

Another possible extension is to incorporate other sensors like temperature, humidity and pressure sensors so which would make the use case more realistic and even applicable to monitoring and transportation of cold storage and fresh food products where the real time temperature, pressure and humidity of the environment can be known and hence the temperature of the cold storage area can be remotely controlled and changed accordingly. The same use case also applies to the monitoring of chemicals at chemical industries.

As a conclusion, the existing application has basically three entities (I) one being the SCION[1] server entity at the pallet integrated with Raspberry Pi [2], RFID reader, load cell [4], HX711 [5] amplifier and then, the SCION [1] client entity at the pallet end integrated with another Raspberry Pi [2] and LCD so that product details are displayed on the LCD and at last, another SCION [1] client entity at the user side, where the product details are monitored in real time. The application can be imagined to be extended such that products stored in all pallets can be interfaced on a single SCION[1] AS server configured Raspberry Pi [2] using some switching mechanism and then, these product data are then transferred over the SCION [1] network on to the multiple SCION[1] client AS at remote locations. The provision of updating and rewriting the RFID tag details attached to each of the pallet can also be considered as a further possible extension.

REFERENCES

[1]SCIONLab Coordination Service, SCION is the first clean-slate Internet architecture designed to provide route control, failure isolation, and explicit trust information for end-to-end communication. Available: https://www.scionlab.org

[2]Raspberry Pi - A small and affordable computer that you can use to learn programming. Available: https://www.raspberrypi.org/

[3]balenaEtcher, An open source project by balena. Available: https://www.balena.io/etcher/

[4] Micro load cell data sheet
Available:
https://www.robotshop.com/media/files/pdf/datasheet3133.pdf

[5] HX711, 24-Bit Analog-to-Digital Converter (ADC) for Weigh Scales. Available: https://cdn.sparkfun.com/datasheets/Sensors/ForceFlex/hx711_english.pdf

[6] Node-Red, Flow-based programming for Internet of things. Available: https://nodered.org/

[7]Go, The Go Programming Language. Available: https://golang.org/

[8] Python, Python is a programming language that lets you work quickly and integrate systems more effectively. Available: https://www.python.org/

[9]MFRC522 Standard performance MIFARE and NTAG front end. Available: https://www.nxp.com/docs/en/data-sheet/MFRC522.pdf

[10] Raspbian, Raspbian is the Foundation's official supported operating system. Available: https://www.raspberrypi.org/downloads/raspbian/

[11] docker, Build, Ship, and Run Any App, Anywhere. Learn about the only enterprise-ready container platform to cost-effectively build and manage your application Available :https://www.docker.com/

[12] Wireshark, Wireshark is the world's foremost and widely-used network protocol analyzer. Available: https://www.wireshark.org/

[13] SCION tutorials, Welcome to SCION Tutorials. Available: https://netsec-ethz.github.io/scion-tutorials/

[14] Ubuntu Mate for Raspberry Pi 2 and 3, Available: https://ubuntu-mate.org/raspberry-pi/

[15] Github, GitHub brings together the world's largest community of developers to discover, share, and build better software. Available: https://github.com/

[16] HX711-Go library, MichaelS11
Available : https://github.com/MichaelS11/go-hx711

[17] SCION Browser AS Visualizations. Available: https://netsec-ethz.github.io/scion-tutorials/as_visualization/browser_asviz/

[18] SCION bwtester application. Available :https://netsec-ethz.github.io/scion-tutorials/sample_projects/bwtester/

[19] Network Security Group at ETH Zürich. Scion-apps. Available: https://github.com/netsec-ethz/scion-apps

APPENDIX

(a)RFID_TCPserver.Py

```
import RPi.GPIO as GPIO
import MFRC522
import signal
from multiprocessing import Process, Value, Array
import Adafruit_DHT
import socket
import time
import json


continue_reading = True

sensor = Adafruit_DHT.DHT11
pin = 17
MIFAREReader = MFRC522.MFRC522()
sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
sock.bind(('0.0.0.0',10000))
sock.listen(1)

def readTempAndHumidity(arr):
while continue_reading:
arr[1], arr[0] = Adafruit_DHT.read_retry(sensor, pin)
# if arr[0] is not None and arr[1] is not None:
#     print('Temp={0:0.1f}*
Humidity={1:0.1f}%'.format(arr[0],arr[1]))
# else:
#     print('Failed to get reading. Try again!')
# if continue_reading == False:
#     return
time.sleep(2)

# Capture SIGINT for cleanup when the script is aborted
def end_read(signal,frame):
global continue_reading
# print "Ctrl+C captured, ending read."
continue_reading = False
sock.close()
#TempReadProcess.join()
GPIO.cleanup()

# Hook the SIGINT
signal.signal(signal.SIGINT, end_read)

arr = Array('d',(0.0,0.0))

TempReadProcess = Process(target=readTempAndHumidity,
args=(arr,))
TempReadProcess.start()


uid_str = "[]"
while True:
c,a = sock.accept()

while True:
data = c.recv(128)
temperature = int(arr[0])
humidity = int(arr[1])

if continue_reading:

# Scan for cards
(status,TagType) =
MIFAREReader.MFRC522_Request(MIFAREReader.PICC_
REQIDL)

# If a card is found

(status,uid) = MIFAREReader.MFRC522_Anticoll()
if status == MIFAREReader.MI_OK:
uid_str = str(uid)
```

```
(status,TagType) =
MIFAREReader.MFRC522_Request(MIFAREReader.PICC_
REQIDL)


data = {"UID" : str(uid),
"Temperature" : temperature,
"Humidity" : humidity}
print(str(data))
data = json.dumps(data)
c.send(bytes(data))
if not data:
c.close()
break
```

(b)weight_server_full_rv2.go

```
package main

import (
//"bufio"
"encoding/json"
"flag"
"fmt"
"log"
"net"
"time"

//"os"
//"strings"

"sync"

"github.com/MichaelS11/go-hx711"

"github.com/scionproto/scion/go/lib/sciond"
"github.com/scionproto/scion/go/lib/snet"
)

type Rfidth struct {
Humidity int
UID string
Temperature int
}

type Message struct {
Name string
Exp string
Time string
Unitweight float64
Capacity int
CurrentNo int
TempL int
TempH int
HmdL int
HmdH int
Temp int
Hmd int
Wght int
UID string
}

func check(e error) {
if e != nil {
log.Fatal(e)
}
}

var weightData map[string]string
var weightDataLock sync.Mutex

func init() {
weightData = make(map[string]string)
}

// Obtains input from weight observation application

func printUsage() {
fmt.Println("weightserver -s ServerSCIONAddress")
fmt.Println("The SCION address is specified as ISD-
AS,[IP Address]:Port")
fmt.Println("Example SCION address 17-
ffaa:0:1102,[192.33.93.173]:42002")
}

func main() {

var (
serverAddress string
sciondPath string
sciondFromIA bool
dispatcherPath string

err error
server *snet.Addr

udpConnection snet.Conn
)

var msg = []Message{Message{"Sallos",
"10/10/2020", "", 5.4, 21, 0, 0, 40, 10, 90, 20, 30, 0, "[249, 20,
56, 86, 131]"},
Message{"Toffee", "20/10/2020", "", 4, 50,
0, 0, 50, 10, 90, 20, 30, 0, "[41, 106, 118, 72, 125]"}}
var noPallette = Message{"NoPallete", "", "", 0, 1, 0,
0, 50, 0, 100, 0, 0, 0, "[]"}
var sendData Message
sendData = noPallette
var rfth = Rfidth{UID: "[]"}
// Fetch arguments from command line
flag.StringVar(&serverAddress, "s", "", "Server
SCION Address")
flag.StringVar(&sciondPath, "sciond", "", "Path to
sciond socket")
flag.BoolVar(&sciondFromIA, "sciondFromIA",
false, "SCIOND socket path from IA address:ISD-AS")
flag.StringVar(&dispatcherPath, "dispatcher",
"/run/shm/dispatcher/default.sock",
```

```
                "Path to dispatcher socket")
                flag.Parse()

        // Create the SCION UDP socket
        if len(serverAddress) > 0 {
                server, err =
    snet.AddrFromString(serverAddress)
                check(err)
        } else {
                printUsage()
            check(fmt.Errorf("Error, server address
      needs to be specified with -s"))
        }

        if sciondFromIA {
            if sciondPath != "" {
             log.Fatal("Only one of -sciond or -
      sciondFromIA can be specified")
            }
            sciondPath =
  sciond.GetDefaultSCIONDPath(&server.IA)
        } else if sciondPath == "" {
            sciondPath =
      sciond.GetDefaultSCIONDPath(nil)
        }
    snet.Init(server.IA, sciondPath, dispatcherPath)
    udpConnection, err = snet.ListenSCION("udp4",
            server)
        check(err)

        defer udpConnection.Close()

        err = hx711.HostInit()

    hx711, err := hx711.NewHx711("GPIO6", "GPIO5")
            check(err)
        // SetGain default is 128
    // Gain of 128 or 64 is input channel A, gain of 32 is
            input channel B
        // hx711.SetGain(128)
    // make sure to use your values from calibration
            above
        hx711.AdjustZero = -72782
        hx711.AdjustScale = -1960

        tcpreq := "hello from client"

      tcpAddr, err := net.ResolveTCPAddr("tcp",
          "0.0.0.0:10000")
            if err != nil {
                panic(err)
            }
    conn, err := net.DialTCP("tcp", nil, tcpAddr)
            defer conn.Close()
            if err != nil {
                panic(err)
            }

      receivePacketBuffer := make([]byte, 2500)
      sendPacketBuffer := make([]byte, 3000)
        tcpReply := make([]byte, 1024)

            for {
                _, clientAddress, err :=
  udpConnection.ReadFrom(receivePacketBuffer)
                check(err)

            // Packet received, send back response to
                same client

            //time.Sleep(200 * time.Microsecond)
                conn.Write([]byte(tcpreq))

                n, err := conn.Read(tcpReply)
                check(err)
            err = json.Unmarshal(tcpReply[:n], &rfth)
                check(err)
                if rfth.UID != "[]" {
                for i := 0; i < len(msg); i++ {
                    if string(rfth.UID) ==
          (msg[i].UID) {
                        sendData =
              msg[i]
                        break
                    }
                }
            }
        data, err := hx711.ReadDataMedian(11)
                check(err)

            if string(sendData.UID) != "[]" {
                sendData.CurrentNo = int(data /
      sendData.Unitweight)
                sendData.Time =
    string(time.Now().Format("Jan 2 15:04:05"))
                } else {
                sendData = noPallette
                }
            sendData.Temp = rfth.Temperature
            sendData.Hmd = rfth.Humidity
            sendData.Wght = int(data)
            b, _ := json.Marshal(sendData)
                fmt.Println(string(b))

            copy(sendPacketBuffer, b)

                for i := 0; i < 3; i++ {
                    _, err =
  udpConnection.WriteTo(sendPacketBuffer[:len(b)],
            clientAddress)
                    if err == nil {
                        break
                    } else {
                     log.Println(err)
                    }
                }
                check(err)
            sendData = noPallette
            }
```

```
}
```

(c) weight_client_retry.go

```
package main

import (
    "flag"
    "fmt"
    "log"
    "net/http"

    "github.com/scionproto/scion/go/lib/sciond"
    "github.com/scionproto/scion/go/lib/snet"
)

func check(e error) {
    if e != nil {
        log.Fatal(e)
    }
}

func printUsage() {
    fmt.Println("scion-sensor-server -s ServerSCIONAddress -c ClientSCIONAddress")
    fmt.Println("The SCION address is specified as ISD-AS,[IP Address]:Port")
    fmt.Println("Example SCION address 1-1,[127.0.0.1]:42002")
}

var (
    clientAddress  string
    serverAddress  string
    sciondPath     string
    sciondFromIA   bool
    dispatcherPath string
    udpConnection  snet.Conn
    err            error
    local          *snet.Addr
    remote         *snet.Addr
)

func main() {

    // Fetch arguments from command line
    flag.StringVar(&clientAddress, "c", "", "Client SCION Address")
    flag.StringVar(&serverAddress, "s", "", "Server SCION Address")
    flag.StringVar(&sciondPath, "sciond", "", "Path to sciond socket")
    flag.BoolVar(&sciondFromIA, "sciondFromIA", false, "SCIOND socket path from IA address:ISD-AS")
    flag.StringVar(&dispatcherPath, "dispatcher", "/run/shm/dispatcher/default.sock", "Path to dispatcher socket")
    flag.Parse()

    // Create the SCION UDP socket
    if len(clientAddress) > 0 {
        local, err = snet.AddrFromString(clientAddress)
        check(err)
    } else {
        printUsage()
        check(fmt.Errorf("Error, client address needs to be specified with -c"))
    }
    if len(serverAddress) > 0 {
        remote, err = snet.AddrFromString(serverAddress)
        check(err)
    } else {
        printUsage()
        check(fmt.Errorf("Error, server address needs to be specified with -s"))
    }

    if sciondFromIA {
        if sciondPath != "" {
            log.Fatal("Only one of -sciond or -sciondFromIA can be specified")
        }
        sciondPath = sciond.GetDefaultSCIONDPath(&local.IA)
    } else if sciondPath == "" {
        sciondPath = sciond.GetDefaultSCIONDPath(nil)
    }
    snet.Init(local.IA, sciondPath, dispatcherPath)
    udpConnection, err = snet.DialSCION("udp4", local, remote)
    check(err)

    http.HandleFunc("/", dataHandler)

    http.ListenAndServe(":4000", nil)

}

func dataHandler(w http.ResponseWriter, r *http.Request) {
    receivePacketBuffer := make([]byte, 2500)
    sendPacketBuffer := make([]byte, 0)
    n := 0
    for i := 0; i < 3; i++ {
        n, err = udpConnection.Write(sendPacketBuffer)
        check(err)

        n, _, err = udpConnection.ReadFrom(receivePacketBuffer)
        if err == nil {
            break
        } else {
            log.Println(err)
        }
    }
```

```
check(err)

fmt.Println(string(receivePacketBuffer[:n]))

w.Write(receivePacketBuffer[:n])
}
```

(d)SCION Raspberry installation method