Raport

System ewidencyjny czasu pracy wykorzystujący technologie Internetu Rzeczy

Przedmiot:	**Podstawy Internetu Rzeczy Laboratorium**
Imię i nazwisko autora:	Kaja Limisiewicz
Numer indeksu:	202103
Semestr studiów:	4
Data ukończenia pracy:	Maj 2020r.
Prowadzący laboratorium:	mgr inż. Piotr Jóźwiak

Spis treści

1. Wstęp

Projekt ma na celu stworzenie systemu, który rejestruje czas pracy pracowników w pewnym przedsiębiorstwie. Składa się z aplikacji centralnej - aplikacji serwera, która przetwarza dane nadsyłane od klientów i je gromadzi, oraz z aplikacji klienckiej, która domyślnie ma być uruchamiana na zestawach Raspberry Pi wyposażonych w czytniki kart RFID. Ewidencja czasu pracy pracownika bazuje na danych o czasie, w którym pracownik zeskanował swoją kartę RFID - raz przy wejściu i raz przy wyjściu z miejsca pracy, przy czym dostępnych czytników kart (klientów) może być wiele.

2. Wymagania projektowe

2.1. Wymagania funkcjonalne

- wprowadzanie i usuwanie klientów - terminali RFID
- wprowadzanie i usuwanie pracowników
- przypisywanie i usuwanie przypisania pracownikowi karty RDIF
- rejestrowanie czasu przybycia i wyjścia pracownika
- rejestrowanie terminalu, którego użył pracownik
- rejestrowanie nieznanych systemowi kart RDIF i terminu ich użycia
- generowanie raportu czasu pracy dla wskazanego pracownika

2.2. Wymagania niefunkcjonalne

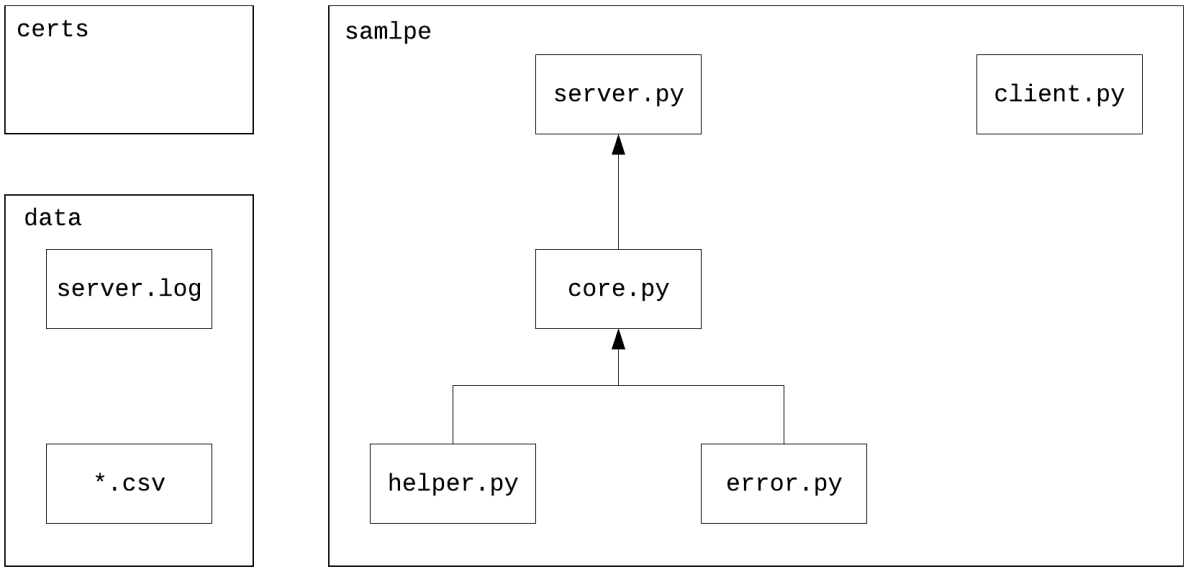
- język programowania Python 3
- protokół transportowy MQTT
- baza danych CSV

3. Opis architektury systemu

3.1. Aplikacja

Aplikacja dzieli się na aplikację serwera i aplikację klienta. Aplikacja klienta jest niezależna i służy do sczytywania kodów kart RFID i przesyłania danych do serwera za pomocą protokołu MQTT.

Aplikacja znajduje się w repozytorium `sample`. Aplikacja serwera odczytuje i zapisuje komunikaty od klientów oraz sama może je nadawać. Za pozostałą funkcjonalność serwera odpowiada moduł `core.py`, który pozwala serwerowi zarządzać czytnikami, pracownikami oraz otrzymywanymi wiadomościami. Dodatkowo, korzystając z wbudowanej biblioteki `logger`, zapisuje wszystkie wydarzenia serwera w pliku `server.log`. Moduł `core.py` jest wspomagany przez moduły `helper.py` i `error.py`. Moduł `helper.py` jest



odpowiedzialny za odczytywanie i zapisywanie plików `.csv`. `error.py` to bardzo zaczyn do modułu obsługującego błędy.

Inne pliki zawarte w repozytorium to folder `certs` z certyfikatami wygenerowanymi do uwierzytelniania klientów MQTT, folder `setup` z plikami konfiguracyjnymi brokera Mosquitto i folder `data` z bazą danych.

3.2.Baza danych

Do przechowywania danych użyto biblioteki CSV. CSV (*Comma Separated Values*) to popularny, bardzo prosty sposób przechowywania danych. Pierwsza linia pliku to nagłówek, który określa znaczenie wartości rekordów, pozostałe linijki zawierają dane. Wartości są oddzielone od siebie przecinkami.

Dane projektu są przechowywane w folderze `data`. Dane są reprezentowane przez cztery pliki:

- `readers.csv`,
- `employees.csv`,
- `cards.csv`
- `logs.csv`.

Plik `readers.csv` przechowuje dane o historii zarejestrowanych czytników kart RFID. Poszczególny rekord składa się z informacji o identyfikatorze czytnika, dacie zarejestrowania, ewentualnej dacie wyrejestrowania i opcjonalnie może zawierać krótki opis czytnika (np. wejście główne). Nagłówek pliku `readers.csv`:

```
reader_id,date_registered,date_unregistered,description
```

Plik `employees.csv` przechowuje dane o historii zarejestrowanych w firmie pracowników. Poszczególny rekord składa się z informacji o identyfikatorze pracownika, imieniu, nazwisku, dacie rejestracji i ewentualnej dacie wyrejestrowania. Nagłówek pliku `employees.csv`:

```
id,name,surname,date_registered,date_unregistered
```

Plik `cards.csv` przechowuje dane o historii zarejestrowanych w firmie kart RFID i przypisanych im pracownikach. Poszczególny rekord składa się z informacji o identyfikatorze karty, identyfikatorze pracownika i dacie przypisania karty do pracownika. Nagłówek pliku `cards.csv`:

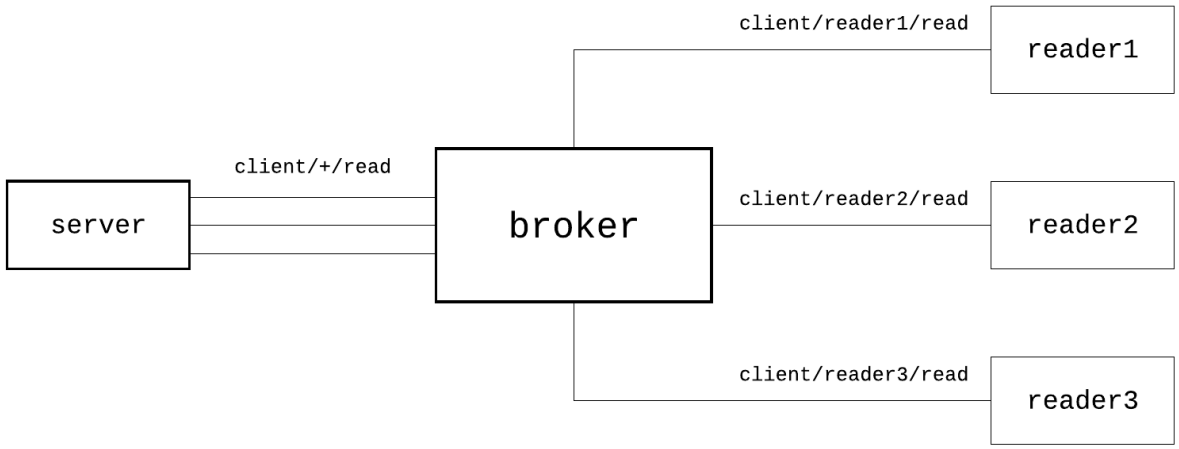
```
card_id,employee_id,date_actualized
```

Plik `logs.csv` jest istotą aplikacji i przechowuje dane o historii sczytanych kart. Poszczególny rekord składa się z danych o identyfikatorze karty, identyfikatorze pracownika, dacie sczytania i identyfikatora czytnika, na którym dokonano sczytania. Nagłówek pliku logs.csv:

```
date,card_id,user_id,reader_id
```

Co więcej, w folderze `data` zawiera się plik `server.log` z wszystkimi wydarzeniami z serwera.

3.3. Architektura sieciowa



W pliku `aclfile.conf` zostały zdefiniowane dostępne tematy dla każdego z użytkowników. Tematy zostały zdefiniowane tak, aby każdy podłączany czytnik kart RFID miał automatycznie przydzielony własny temat, którego subskrybentem jest też serwer, ale żaden z pozostałych czytników już nie. Zdefiniowano pulę 10 takich tematów 'prywatnych'.

Poza tym, wszystkie czytniki i serwer są subskrybentami tematów `client/logs` i `server/logs`. Temat `server/log` został stworzony, aby była możliwość wysyłania komunikatów do wszystkich czytników. Temat `client/logs` służy do informowania innych użytkowników brokera o nowo przyłączonych czytnikach.

4. Opis implementacji i zastosowanych rozwiązań

4.1. Wczytanie danych

Przy każdym włączeniu aplikacji serwera, dane z plików .csv są pobierane i przetwarzane tak, że rekordy istotne są instancjami odpowiednich klas. Zastosowałam takie rozwiązanie, żeby ułatwić wyszukiwanie i modyfikację danych. Instancje są przechowywane w listach.

W pliku `helper.py` znajdują cię zdefiniowane klasy oraz odpowiadające im loadery.

```
class Card:
    def __init__(self, rfid_tag, employee_id, date):
        self.__rfid_tag = rfid_tag
        self.__employee_id = employee_id
        self.__date_actualized = date

def __load_cards():
    file = open(cards_file, newline='')
    reader = csv.reader(file)

    header = next(reader)
    # header = [rfid_tag, employee_id, date_actualized]

    for row in reader:
        # ten fragment sprawdza czy dana karta pojawiła się już
        # wcześniej w bazie i jeśli tak, usuwa obiekt nieaktualny
        index = get_index(row[0], 'cards')
        if index is not -1:
            cards.pop(index)

        rfid_tag = row[0]
        if row[1] is '':
            employee_id = -1
        else:
            employee_id = int(row[1])
        date_actualized = row[2]

        cards.append(Card(rfid_tag, employee_id, date_actualized))

    file.close()
```

4.2. Wprowadzanie nowych danych

Przyciski GUI aplikacji korzystają z metod zawartych w pliku core.py. I, na przykład, metoda rejestrująca nowy czytnik w bazie wygląda tak:

```
def register_reader(name=''):
# nowy czytnik jest inicjalizowany i dodawany do listy czytników
    readers.append(Reader(reader_id=Reader.reader_id_counter,
      date_registered=datetime.today().strftime('%d/%m/%Y %H:%M:%S'),
      date_unregistered='', description=name))

# zmienna globalna, odpowiedzialna za nadawanie unikatowych
# identyfikatorów jest aktualizowana
    Reader.reader_id_counter += 1

# wydarzenie jest raportowane do server.log, a zmiana jest dopisywana do
# bazy
    __log_reader_register(readers[-1].get_id())
    save_changes(-1, 'readers')
```

Za sam zapis danych do bazy odpowiadają metody:

```
def save_changes(index, mode):
    db, file_path = __list_and_path(mode)

    file = open(file_path, "a", newline='')
    writer = csv.writer(file)

    row = db[index].get_data()
    writer.writerow(row)

    file.close()

def __list_and_path(mode):
    return {
        'cards': (cards, cards_file),
        'employees': (employees, employees_file),
        'readers': (readers, readers_file)}[mode]
```

4.3. Interfejs graficzny

Do stworzenia interfejsu graficznego skorzystałam z biblioteki `tkinter`. Aplikacja serwera korzysta z jednego okna i kilku `messageboxów` do wprowadzania danych. Aplikacja klienta składa się tylko z jednego okna.

Przykładowe okno dialogowe:

```
def sign_card():
    rfid_id = simpledialog.askstring('Przypisanie karty RFID
            pracownikowi', 'Podaj numer karty RFID', parent=window)
    employee_id = simpledialog.askstring('Przypisanie karty RFID
            pracownikowi', 'Podaj numer pracownika', parent=window)
    register_card(rfid_id, int(employee_id))
    logs_box()
```

4.4. Protokół MQTT

Protokół został zaimplementowany za pomocą biblioteki `paho-mqtt 1.5.0`. Implementacja w aplikacji serwera:

```
import paho.mqtt.client as mqtt

broker = "Kajas-MBP"
port = 8883
client = mqtt.Client()

def run_receiver():
    connect_to_broker()
    client.subscribe("client/logs")
    client.subscribe("client/+/read")
    . . .
    disconnect_from_broker()

def connect_to_broker():
    client.tls_set("./certs/ca.crt")
    client.username_pw_set(username="server", password="1234")
    client.connect(broker, port)
    client.on_message = process_message
    client.loop_start()
```

Serwer korzysta z certyfikatu uwierzytelniania wygenerowanego dla danego brokera. Aby nawiązać połączenie, serwer podaje wcześniej zdefiniowane hasło. Więcej o implementacji i konfiguracji w pkt. 5.1.1

5. Opis działania i prezentacja interfejsu

5.1. Instalacja

5.1.1. Generowanie certyfikatów

Bezpieczeństwo komunikacji w systemie wymaga szyfrowania. W projekcie jest wykorzystany protokół kryptograficzny TLS. Certyfikaty zostały wygenerowane za pomocą OpenSSL:

```
openssl genrsa -des3 -out ca.key 2048
```

```
openssl req -new -x509 -days 1826 -key ca.key -out ca.crt
```

```
openssl genrsa -out server.key 2048
```

```
openssl req -new -out server.csr -key server.key
```

```
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key
-CAcreateserial -out server.crt -days 360
```

5.1.2. Konfiguracja Mosquitto

Aby skorzystać z protokołu komunikacyjnego MQTT potrzebujemy brokera. Zainstalowałam i skonfigurowałam brokera Mosquitto.

```
brew install mosquitto
```

```
brew services start mosquitto
```

W pliku konfiguracyjnym podałam ścieżki certyfikatów. Następnie stworzyłam użytkowników, nadałam im hasła:

```
mosquitto_passwd -c passwd.conf server
```

```
mosquitto_passwd -b passwd.conf client 1234
```

i w pliku konfiguracyjnym `mosquitto.conf` wyłączyłam możliwość komunikacji użytkownikom anonimowym oraz podałam ścieżkę pliku z hasłami:

```
allow_anonymous false
```

```
password_file /usr/local/Cellar/mosquitto/1.6.9/etc/mosquitto/
passwd.conf
```

```
brew services restart mosquitto
```

W celu kontroli przepływu komunikatów pomiędzy poszczególnymi użytkownikami systemu, stworzyłam plik `aclfile.conf` z listą ACL oraz w pliku konfiguracyjnym Mosquitto podałam jego ścieżkę. Więcej na temat kontroli w punkcie 3.3 Architektura sieciowa.

Treść pliku `aclfile.conf`:

```
# This only affects clients with username "server".
user server
topic server/name
topic worker/name
topic client/+/read
topic client/logs
topic server/logs

# This only affects clients with username "client".
user client
topic read server/name
topic worker/name
topic client/reader1/read
topic client/reader2/read
topic client/reader3/read
topic client/reader4/read
topic client/reader5/read
topic client/reader6/read
topic client/reader7/read
topic client/reader8/read
topic client/reader9/read
topic client/reader10/read
topic client/logs
topic read server/logs
```

5.1.3. Biblioteki

Działanie programu wymaga zainstalowania bibliotek:

`pip install paho-mqtt` - biblioteka do obsługi komunikacji MQTT

`pip install tkinter` - biblioteka GUI

`pip install logger` - biblioteka logera

`pip install csv` - biblioteka obsługi plików .csv

5.2. Uruchomienie aplikacji

Aplikację na serwerze uruchamia się przez komendę

```
python3 sample/server.py
```

Aplikację na czytniku kart RFID uruchamia się przez komendę

```
python3 sample/client.py
```

5.3. Screeny

Screeny z działania systemu znajdują się w repozytorium na GitHubie

6. Podsumowanie

Projekt spełnia początkowe założenia projektowe, a nawet je rozszerza. Jego struktura pozwala na łatwe rozbudowanie o kolejne funkcjonalności.

Projekt jest jedynie bazą systemu, którą można rozbudować. W szczególności można rozbudować system generowania raportów. Bazując na zebranych danych, można rozszerzyć funkcjonalność o generowanie raportów o dostępnych czytnikach, zarejestrowanych pracownikach, o kartach i ich użytkownikach, o ostrzeżeniach generowanych przez system oraz o użytych kartach niezarejestrowanych w systemie.

System mógłby również mieć dokładniejszą i bardziej rozbudowaną obsługę wyjątków i błędów.

7. Aneks

kod projektu znajduje się w repozytorium GitHuba

`https://github.com/limisie/iot_rfid_reader.git`