Politechnika Wrocławska

Wydział Informatyki i Zarządzania

Raport

System ewidencji czasu pracy wykorzystujący technologie Internetu Rzeczy

Przedmiot:	Podstawy Internetu Rzeczy Laboratorium
Imię i nazwisko autora:	Rafał Behrendt
Nr indeksu:	246643
Semestr studiów:	4
Data ukończenia pracy:	Maj 2020 r.
Prowadzący laboratorium:	mgr inż. Piotr Jóźwiak

HR EXCELLENCE IN RESEARCH

Politechnika Wrocławska
Wybrzeże Wyspiańskiego 27
50-370 Wrocław
wiz.pwr.edu.pl

Spis treści

1. Wymagania projektowe

System ewidencji czasu pracy wykorzystujący technologie Internetu Rzeczy jest projektem realizowanym w celu wdrożenia funkcjonalności umożliwiającej ewidencję czasu pracy pracowników, przykładowo w przedsiębiorstwie. System podzielony jest na część serwera, który odbiera dane od klienta i je przetwarza oraz część kliencką (terminale RFID), które pozwalają na skanowanie zdefiniowanych kart. System jest w stanie podjąć fizyczną interakcję z otoczeniem, komunikuje się z poszczególnymi częściami za pomocą protokołów internetu rzeczy, oraz stale pobiera, przetwarza i gromadzi dane. Aplikacja pozwala na uruchomienie wielu klientów dla jednego serwera.

1.1. Wymagania funkcjonalne

- Serwer pozwala przyłączyć do systemu klientów, czyli terminale RFID, którymi są laboratoryjne zestawy Raspberry Pi.
- Serwer pozwala usunąć z systemu terminale RFID, którymi są laboratoryjne zestawy Raspberry Pi.
- Serwer pozwala przypisać kartę RFID do pracownika.
- Serwer pozwala usunąć przypisanie karty RFID do pracownika.
- System, jako całość, rejestruje termin przybycia pracownika do miejsca pracy i terminal RFID (klienta), który użył pracownik.
- System rejestruje termin opuszczenia miejsca pracy przez pracownika i terminal RFID, który użył pracownik.
- W przypadku użycia nieznanej systemowi karty, system rejestruje ją w systemie, godzinę jej użycia i terminal.
- W przypadku użycia karty nieprzypisanej do żadnego użytkownika, system rejestruje godzinę oraz terminal, z którego skorzystano
- Serwer pozwala wygenerować raport czasu pracy dla wskazanego pracownika.
- Serwer pozwala na wyświetlenie wszystkich logów.
- Serwer pozwala na zdalne rozłączenie klientów.

1.2. Wymagania niefunkcjonalne

- Program napisany jest w języku programowania python v3.
- Komponenty systemu komunikują się ze sobą poprzez protokół MQTT.
- Do zarządzania protokołem komunikacyjnym wykorzystywany jest broker mosquitto.
- Komunikacja jest szyfrowana poprzez protokół SSH.
- Dane zapisywane są trwale przy pomocy systemu bazodanowego SQLite.

Aplikacja może być uruchomiona na systemie Windows bądź na dystrybucjach Linuxa. Wymagana jest instalacja powyższych komponentów.

2. Opis architektury systemu

System napisany został w języku programowania python v3, przeznaczony na zestawy raspberry pi – terminale RFID. Zestawy te skanować mogą karty pracownicze przy użyciu wbudowanych modułów RFID do skanowania kart.

System gromadzi dane w bazie danych opartej na systemie bazodanowym SQLite.

Aplikacja pozwala na połączenie wielu klientów do jednego serwera. Komunikacja odbywa się za przy pomocy protokołu MQTT, za pośrednictwem brokera mosquitto. Komunikacja szyfrowana jest za pomocą protokołu SSH. Połączenie możliwe jest tylko pod warunkiem posiadania odpowiedniego certyfikatu.

Do korzystania z aplikacji konieczne jest uwierzytelnienie. Stworzeni zostali dwaj użytkownicy – client logujący się na terminal oraz server logujący się na serwer. Mają oni zdefiniowane w pliku acl w pliku konfiguracyjnym brokera konkretne uprawnienia dotyczące połączenia.

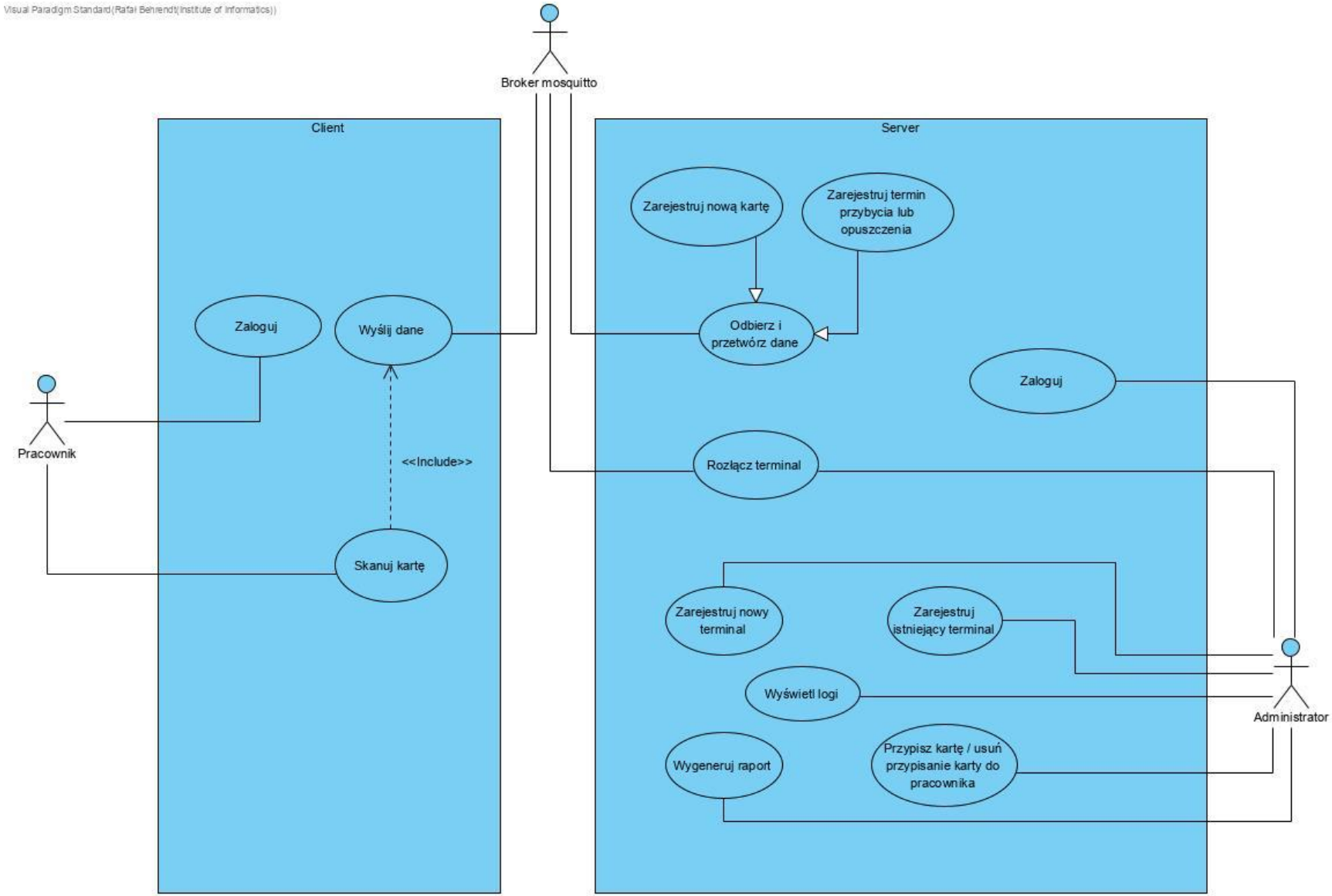
Struktura plików projektu jest następująca. W pliku głównym znajdują się skrypty uruchamiające aplikację oraz folder venv. W pliku tym znajdują się następujące foldery:

- certs – zawierający certyfikat konieczny dla działania aplikacji
- database – zawierający plik z bazą danych
- reports – folder w którym zapisywane są raporty pracy pracowników. Każdy raport ma nazwę odpowiednią dla ID danego pracownika.
- sources – folder z kodem źródłowym programu

2.1. Elementy składowe aplikacji

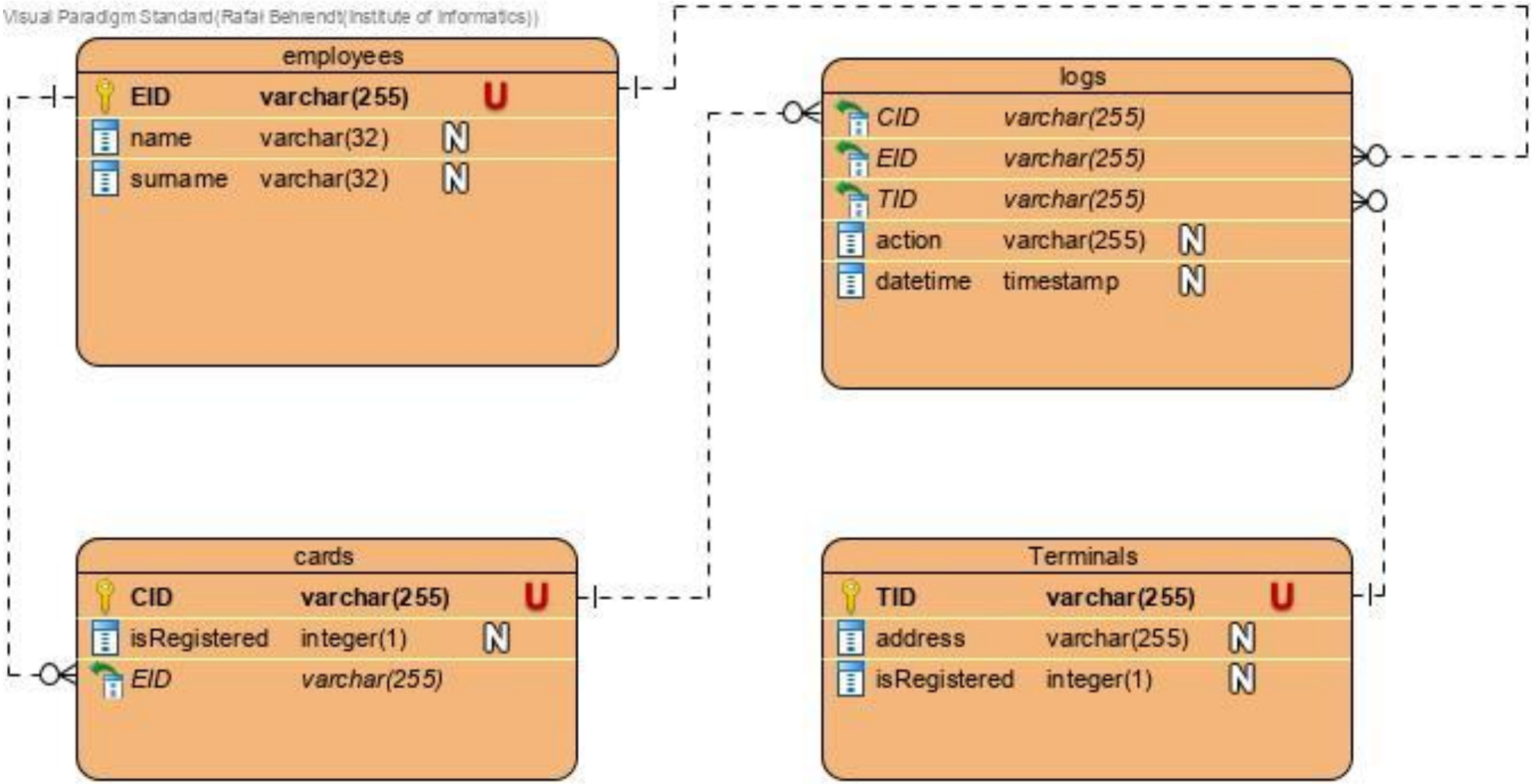
- Constants.py – plik ten zawiera stałe wykorzystywane w programie
- DBManage.py – plik zawierający elementy do tworzenia oraz resetowania bazy danych
- Employee.py – plik z klasą użytkownika
- Init.py – plik inicjujący program
- ManagementService.py – plik z klasą do zarządzania bazą danych, programem, tworzący przykładowe dane, wyświetlający dane z bazy danych
- resetApp.py – element aplikacji odpowiedzialny za przywracanie jej do stanu początkowego
- Server.py – plik zawierający klasę serwera
- serverGuiManager.py – plik odpowiadający za wyświetlany interfejs graficzny serwera
- Terminal.py – plik z klasą terminal
- terminalGuiManager.py - plik odpowiadający za wyświetlany interfejs graficzny terminala
- startClient.py – plik inicjujący klienta
- startServer.py – plik inicjujący serwer

2.2. Diagram przypadków użycia



Rys. 1 - Diagram przypadków użycia

2.2. Struktura bazy danych



Rys. 2 – Struktura bazy danych

3. Opis implementacji i zastosowanych rozwiązań

3.1. Uprawnienia i autoryzacja

3.1.1. Uprawnienia poszczególnych użytkowników

Client:

- Odczyt z tematu server/name
- Odczyt i zapis na temacie CID/TID

Server:

- Odczyt i zapis na temacie server/name
- Odczyt i zapis na temacie CID/TID

3.1.2. Dane dotyczące logowania

Dane dotyczące logowania na terminalu:

Login: client
Hasło: password

Dane dotyczące logowania na serwerze:

Login: server
Hasło: p@ssw0rd

Dane dotyczące autoryzacji są zdefiniowane w pliku konfiguracyjnym brokera mosquitto i zostały wprowadzone na potrzeby laboratorium. Powyższe dane mogą być zmienione ale są niezależne od aplikacji, tylko od ustawień brokera.

Serwer subskrybuje temat *CID/TID* – nasłuchuje nadchodzących wiadomości z brokera na tym temacie. Klient z kolei subskrybuje temat *server/name.* Podczas uruchamiania aplikacji należy podać login i hasło.

3.1.3. Fragmenty kodu

startServer.py

```
1. def runServer():
2.     login = input("Login: ")
3.     password = getpass()
```

Server.py

```
1. def connectToBroker(self, login, password):
2.     self.client.tls_set(Constants.CERT_PATH)
3.     self.client.username_pw_set(username=login, password=password)
4.     self.client.connect(self.broker, self.port)
5.     self.client.on_message = self.processMessage
6.     self.client.loop_start()
7.     self.client.subscribe("CID/TID")
```

startClient.py

```
1. def runClient(terminal):
2.     login = input("Login: ")
3.     password = getpass()
```

Terminal.py

```
1. def connectToBroker(self, login, password):
2.     self.client.tls_set(Constants.CERT_PATH)
3.     self.client.username_pw_set(username=login, password=password)
4.     self.client.connect(self.broker, self.port)
5.     self.client.on_message = self.processMessage
6.     self.client.loop_start()
7.     self.client.subscribe("server/name")
8.     self.scanCardMQTT(Constants.CLIENT_CONN)
```

Z modułu Constants pobierane są stałe wykorzystywane w aplikacji. Oba fragmenty aby móc ustanowić połączenie z brokerem muszą mieć certyfikat, który im na to pozwoli. Następnie ustawiane są dane logowania, podane podczas uruchomienia. Ustawiane jest również jaką metodę wykonać ma klient bądź serwer w przypadku odebrania wiadomości, oraz na jakie kanały subskrybują. Ponadto gdy serwer połączy się z brokerem, to poprzez metodę *scanCardMQTT* wysyła wiadomość, dzięki której powiadamia serwer o swoim przyłączeniu się do sieci.

3.2. Obsługa bazy danych

Obsługa bazy danych podzielona jest na część do zarządzania (tworzenie table, relacji, oraz ich usuwanie), oraz część obsługi, którą zajmuje się serwer. W poniższych fragmentach podane jest kilka wybranych przykładów.

DBManage.py

```
1. def initializeDB():
2.     conn = sqlite3.connect(DATABASE_PATH)
3.     c = conn.cursor()
4.
5. c.execute("""CREATE TABLE IF NOT EXISTS cards (
6.     CID text PRIMARY KEY NOT NULL,
7.     isRegistered integer,
8.     EID text,
9.     FOREIGN KEY(EID) REFERENCES employees(EID)
10.  )""")
11.
12.  conn.commit()
```

```
1.  def resetDB():
2.     conn = sqlite3.connect(DATABASE_PATH)
3.     c = conn.cursor()
4.
5.     c.execute("DROP TABLE IF EXISTS employees")
6.     c.execute("DROP TABLE IF EXISTS logs")
7.     c.execute("DROP TABLE IF EXISTS cards")
8.     c.execute("DROP TABLE IF EXISTS terminals")
9.     conn.close()
10.     initializeDB()
11.
12.
```

Powyższe fragmenty przedstawiają metodę inicjalizacji bazy danych (wraz ze stworzeniem jednej z tabel) oraz metodę do resetowania bazy danych.

Server.py

```
1. def loadTerminals(self):
2.    conn = sqlite3.connect(Constants.DATABASE_PATH,
   detect_types=sqlite3.PARSE_DECLTYPES)
3.     c = conn.cursor()
4.     self.listOfTerminals.clear()
5.     c.execute("SELECT TID FROM terminals WHERE isRegistered = '1'")
6.     lot = c.fetchall()
7.     for terminal in lot:
8.        self.listOfTerminals.append(terminal[0])
9.     print("Loaded terminals")
10.    conn.close()
```

```
1. def registerTerminal(self, TID):
2.    conn = sqlite3.connect(Constants.DATABASE_PATH,
   detect_types=sqlite3.PARSE_DECLTYPES)
3.    c = conn.cursor()
4.    c.execute("UPDATE terminals SET isRegistered = '1' WHERE TID =
   '{}'".format(TID))
5.    conn.commit()
6.    self.loadTerminals()
7.    conn.close()
```

Metoda *loadTerminals()* ładuje zarejestrowane terminale z bazy danych na serwer. Metoda *registerTerminal()* rejestruje terminal na serwerze poprzez zmianę flagi *isRegistered* z 0 na 1. Jest wiele metod obsługujących bazę danych, opierają się one również na zapytaniach SQL.

3.3. Komunikacja klient - serwer

Komunikacja na linii klient serwer odbywa się poprzez temat CID/TID, na który informacje publikuje klient. Temat ten subskrybuje serwer, który w przypadku odebrania informacji wykonuje metodę *processMessage().* Wyświetla ona informację co się stało, oraz jeżeli to konieczne, to dokonuje odpowiednich transakcji w bazie danych.

Server.py

```
1. def processMessage(self, client, userdata, message):
2.     decodedMessage = (str(message.payload.decode("utf-8"))).split(".")
3.     if decodedMessage[0] == Constants.CLIENT_CONN:
4.        print(decodedMessage[0] + " : " + decodedMessage[1])
5.        self.gui.setTerminalOnline(decodedMessage[1])
6.     elif decodedMessage[0] == Constants.CLIENT_DISCONN:
7.        print(decodedMessage[0] + " : " + decodedMessage[1])
8.        self.gui.setTerminalOffline(decodedMessage[1])
9.     else:
10.         self.gui.setTerminalOnline(decodedMessage[1])
11.        returnedVal = self.receiveData(decodedMessage[1], decodedMessage[0])
12.        self.gui.log(returnedVal, "yellow")
```

Gdy serwer odbierze dane sprawdza jakiego typu jest to wiadomość – terminal połączył się, lub rozłączył się, albo zeskanowano kartę. W przypadku informacji związanych z połączeniem serwer zmienia wyświetlany status terminala w menu i podaje odpowiednią informację.

```
1. def receiveData(self, TID, CID):
2.     conn = sqlite3.connect(Constants.DATABASE_PATH,
   detect_types=sqlite3.PARSE_DECLTYPES)
3.     c = conn.cursor()
4.     print("Received card {} from terminal {}".format(CID, TID))
5.     if self.listOfTerminals.__contains__(TID):
6.         c.execute("SELECT * FROM cards WHERE CID='{}' AND isRegistered =
   1".format(CID))
7.         cardQuery = c.fetchone()
8.         if cardQuery is not None:
9.             c.execute("SELECT EID FROM cards WHERE CID='{}'".format(CID))
10.            employee = c.fetchone()
11.            if employee[0] != "None":
12.                if self.checkedInEmployees.__contains__(employee[0]):
13.                    retVal = "Checking employee {} out".format(employee[0])
14.                    print(retVal)
15.                    self.checkOut(TID, CID, employee[0])
16.                    return retVal
17.                else:
18.                    retVal = "Checking employee {} in".format(employee[0])
19.                    print(retVal)
20.                    self.checkIn(CID, TID, employee[0])
21.                    return retVal
22.            else:
23.                retVal = "Scanned unbound card {}".format(CID)
24.                print(retVal)
25.                self.logUnboundCardScan(CID, TID)
26.                return retVal
27.        else:
28.            retVal = "Registering new card {}".format(CID)
29.            print(retVal)
30.            self.registerUnknownCard(CID, TID)
31.            return retVal
32.    else:
33.        print(Constants.TERMINAL_NOT_REGISTERED)
34.        return Constants.TERMINAL_NOT_REGISTERED
```

W przypadku zeskanowania karty serwer realizuje metodę *receiveData()* która obsługuje otrzymane w wiadomości ID karty (CID) oraz ID terminala z którego skorzystano (TID). Serwer łączy się z bazą danych i wyświetla informację o odebraniu karty od terminala. Serwer zakończy metodę i poda stosowną informację, jeżeli terminal z którego skorzystano nie jest zarejestrowany na serwerze. W innym przypadku sprawdza czy karta jest znana:

- Jeżeli tak, to system sprawdza czy karta jest przypisana do danego użytkownika i w logach rejestruje jego rozpoczęcie bądź zakończenie pracy. W innym przypadku w logach rejestrowane jest użycie karty, która nie jest przypisana do żadnego użytkownika.
- W przeciwnym przypadku system rejestruje w bazie danych nową, niepowiązaną z żadnym użytkownikiem kartę i rejestruje to zdarzenie w logach.

Terminal.py

```
1. def scanCardMQTT(self, msg):
2.     print("msg: {} | TID: {}".format(msg, self.TID))
3.     self.client.publish("CID/TID", msg + "." + self.TID)
```

Do wysyłania wiadomości terminal wykorzystuje metodę *scanCardMQTT()*. Publikuje ona na temacie wiadomość, którą może być ID karty, informacja o połączeniu się z brokerem, lub rozłączeniu.

3.4. Komunikacja serwer - klient

Komunikacja na linii serwer klient odbywa się poprzez temat server/name, na który informacje publikuje serwer. Temat ten subskrybuje serwer, który w przypadku odebrania informacji wykonuje metodę *processMessage().* Komunikacja na tej linii realizowana jest celem zdalnego rozłączenia terminala z sieci. Serwer publikuje wiadomość informującą o powodzie rozłączenia, a terminal wyświetla ją, po czym kończy pracę.

Terminal.py

```
1. def processMessage(self, client, userdata, message):
2.     decodedMessage = (str(message.payload.decode("utf-8"))).split(".")
3.     if decodedMessage[0] == self.TID:
4.         messagebox.showinfo("Message from server", "You have been disconnected
   by server. Reason : {}".format(decodedMessage[1]))
5.         self.terminalGui.quit()
6.         self.client.disconnect()
7.         exit(0)
```

Server.py

```
1. def disconnectTerminal(self, TID, message):
2.     self.gui.setTerminalOffline(TID)
3.     self.client.publish("server/name", TID + "." + message)
```

Serwer publikuje wiadomość wraz z ID terminala na temacie server/name. Odbierając wiadomość terminal sprawdza, czy opublikowane ID zgadza się z jego ID. Jeżeli tak, to wyświetlane jest powiadomienie o rozłączeniu przez serwer wraz z wiadomością, po czym terminal kończy pracę.

3.5. Interfejsy graficzne

W celu ułatwienia komunikacji użytkownika z aplikacją stworzone zostały interfejsy graficzne, które pozwalają na realizację wszystkich wymagań funkcjonalnych. Okno serwera zapewnia możliwość transakcji na bazie danych – m.in. rejestrowanie i usuwanie terminali, przypisywanie oraz usuwanie przypisania kart z użytkownikami. Pozwala także na wyświetlenie logów oraz generowanie raportów w pliku CSV. Serwer wyświetla także aktualny zarejestrowanych terminali oraz umożliwia zdalne rozłączanie ich.

Interfejs graficzny terminala zapewnia możliwość skanowania dostępnych kart. Wyświetla również informację w przypadku zdalnego rozłączenia. Uruchamiany jest wraz ze startem aplikacji.

Interfejsy graficzne zbudowane zostały za pomocą narzędzia tkinter.

4. Opis działania i prezentacja interfejsu

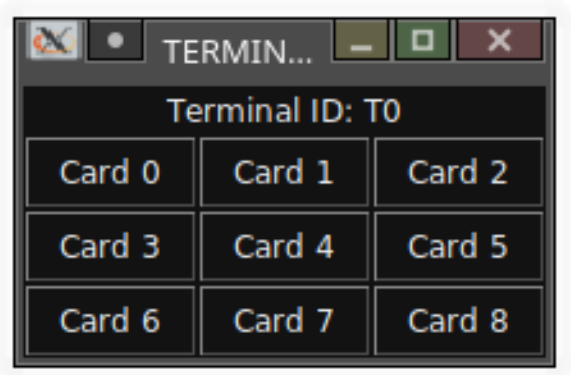
Do aplikacji dołączone są trzy skrypty uruchamiające – startServer, startClient oraz restartApp, których rozszerzenia są zależne od systemu na którym aplikacja ma działać. Skrypty należy uruchomić z poziomu wiersza poleceń.

- startServer – uruchamia część serwera po autoryzacji odpowiednim loginem i hasłem.
- startClient – uruchamia część klienta po autoryzacji odpowiednim loginem i hasłem. Wymaga parametru, który będzie ID uruchamianego terminala. Podanie liczby naturalnej skutkować będzie wybraniem istniejącego terminala z bazy danych i uruchomieniem go (pod warunkiem, że liczba nie jest większa niż liczba zapisanych terminali). W przypadku podania błędnej liczby, lub braku parametru, aplikacja wyśle stosowny komunikat oraz dostępne terminale, które można uruchomić.
- resetApp – skrypt resetuje bazę danych aplikacji.

Aplikacja nie będzie działać poprawnie jeżeli broker nie zostanie odpowiednio skonfigurowany. W tym celu należy skonfigurować broker zgodnie z instrukcjami z laboratorium 7 oraz 8, modyfikując je o odpowiednie hasła oraz uprawnienia ACL opisane w punkcie 3.1. Ponadto adres brokera w kodzie aplikacji będzie musiał zostać zmieniony na nazwę komputera, na której aplikacja jest uruchamiana.

4.1. Terminal

Funkcjonalność terminala sprowadza się do skanowania kart. Poniższy zrzut ekranu prezentuje wygląd uruchomionego terminala. Dostępne są przyciski dla każdej karty w bazie danych. Przyciśnięcie któregoś z nich wywołuje metodę wysyłającą powiadomienie o zeskanowaniu danej karty na danym terminalu.



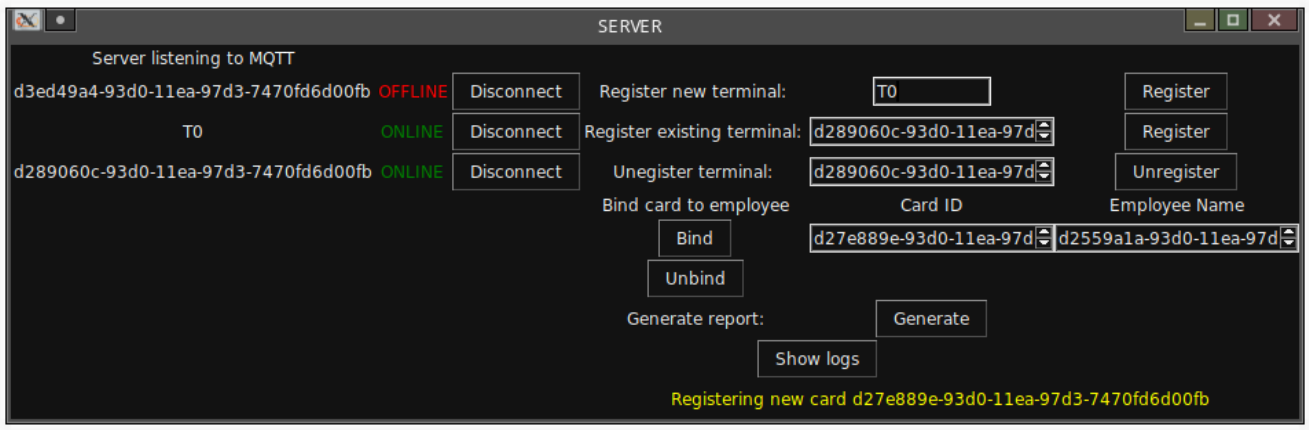
Rys. 3 – interfejs terminala

Ponadto jeżeli serwer zdalnie rozłączył terminal, wyświetli on wiadomość informującą o tym.

Rys. 4 – komunikat o rozłączeniu

4.2. Serwer

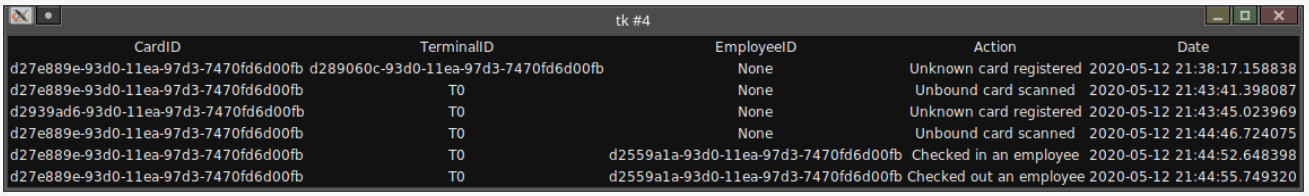
Poniższy zrzut ekranu prezentuje wygląd uruchomionego serwera. Po lewej widać zarejestrowane na serwerze terminale oraz ich aktualny status. Każdy z nich można rozłączyć przyciskiem Disconnect. Po prawej z kolei jest część umożliwiająca zarządzanie elementami aplikacji. Każda opcja jest opisana poniżej.



Rys. 5 – interfejs serwera

- Register new terminal – rejestruje nowy, nieistniejący dotąd terminal na serwerze, oraz dodaje go do bazy danych. W rubryce należy wpisać ID nowego terminala – musi być unikalne wśród już istniejących.
- Register existing terminal – terminale można wyrejestrować. Nie oznacza to jednak że serwer o nich zapomina, nadal istnieją w bazie danych i można je ponownie zarejestrować.
- Unregister terminal – umożliwia wyrejestrowanie terminala.
- Bind card to employee – przycisk Bind wiąże ID wybranej karty z ID wybranego pracownika. Przycisk Unbind z kolei usuwa powiązanie danej karty niezależnie od wybranego pracownika.
- Generate report – generuje raport czasu pracy wybranego w rubryce wyżej pracownika. Raporty zapisywane są w pliku reports z rozszerzeniem *.csv.
- Show logs – pokazuje dotychczasowe logi zebrane przez serwer.

Poniższy zrzut ekranu pokazuje przykładową odpowiedź serwera po wybraniu opcji Show logs.



Rys. 6 – wyświetlone logi

5. Podsumowanie

Wykonany projekt pozwala na realizację wszystkich wymagań funkcjonalnych. Zastosowane narzędzia są zgodne z wymaganiami kursu.

6. Literatura

Instrukcje do laboratorium 7 oraz 8
http://www.steves-internet-guide.com/mosquitto-tls/
https://docs.python.org/3/
https://www.sqlite.org/docs.html

7. Aneks

Kod w formie elektronicznej udostępniony został na e-portalu w laboratorium 9.