Projekt kompetencyjny

Sem. VI Informatyka WEEIA

Półautonomiczny robot inspekcyjny

Analiza LLD

Spis treści

Wymagania 3

Dziennik zdarzeń (event log) 4

Budowa klasy 4

Konstruktor klasy ( \_\_init\_\_() ) 4

Dodawanie zdarzenia ( traceAdd() ) 4

Przykładowy log 4

Prototyp kodu: 5

Obsługa ramek ruchu 6

Interakcje 6

Dziennik zdarzeń 6

Wysyłanie ramek 6

Budowa ramki ruchu 6

Schemat budowy 6

Pola ramki 6

Budowa klasy 7

Konstruktor ( \_\_init\_\_() ) 7

Ustawienia pól ruchu ( setMovement() ) 7

Budowanie ramki ( getFrame() ) 7

Sprawdzanie poprawności ramki ( frameValidation() ) 7

Dekodowanie ramki ( decodeFrame() ) 7

Prototyp kodu 8

Menadżer połączeń 10

Funkcje 10

Serwer 10

Klient 10

Interakcje 10

Dziennik zdarzeń 10

Przekazywanie wiadomości 10

Połączenie klient <-> serwer 10

Budowa klasy 10

Konstruktor ( \_\_init\_\_() ) 10

Destruktor ( \_\_del\_\_() ) 10

Własne IP ( ownIP() ) 11

Czy port jest otwarty ( isOk() ) 11

Skanowanie ( scanRun() ) 11

# Wymagania

* Poruszanie się za pomocą dedykowanej aplikacji
* Poruszanie się za pomocą wgranej trasy
* Omijanie przeszkód
* Przesyłanie danych z urządzeń pomiarowych
* System zmiany urządzeń pomiarowych
* Dedykowana aplikacja

# Dziennik zdarzeń (event log)

Klasa obsługująca dziennik zdarzeń powinna być tworzona w aplikacji głównej robota, oraz aplikacji głównej terminala sterującego. Dziennik zdarzeń powinien być tworzony jako pierwszy, już przy inicjalizacji klasy. Obiekt dziennika powinien być przekazywany w konstruktorze wszystkich klas używanych w tym projekcie.

## Budowa klasy

### Konstruktor klasy ( \_\_init\_\_() )

Konstruktor powinien tworzyć plik o nazwie zadanej w parametrach konstruktora. Jeśli próba utworzenia pliku się nie powiedzie, powinien zostać wypisany wyjątek z informacji o błędzie z powodu którego nie udało się utworzyć pliku.

Konstruktor zapisuje też w polach klasy ścieżkę do pliku oraz nazwę pliku.

### Dodawanie zdarzenia ( traceAdd() )

Podstawową funkcjonalnością metody dodawania zdarzenia jest utworzenie łańcucha znaków w formacie:

XXXX: <wiadomość>, gdzie:

XXXX – miejsce z którego przyszła wiadomość, w formie łańcucha znaków, przekazywana przez klasę wysyłającą wiadomość. Obowiązkowo dużymi literami.

<wiadomość> - łańcuch znaków przekazywany przez klasę wysyłającą wiadomość do dziennika zdarzeń.

Dodanie zdarzenia jest realizowane przez otwarcie dokumentu w trybie nadpisywania (‘a’), oraz zamknięcie dokumentu po dodaniu wiadomości.

Otwieranie i zamykanie dokumentu w momencie dodawania wiadomości, ma umożliwić podgląd zdarzeń systemu w czasie rzeczywistym.

## Przykładowy log

CONN\_HNDL: Connection handler, welcome!

CONN\_HNDL: Own ip: 192.168.0.103 network: 192.168.0.

CONN\_HNDL: Adres: 192.168.0.105 port: 80 nazwa: 192.168.0.105

CONN\_HNDL: Adres: 192.168.0.105 port: 139 nazwa: 192.168.0.105

CONN\_HNDL: IP ADRESSES found: {('192.168.0.105', 80), ('192.168.0.105', 139)}

CONN\_HNDL: End of scanning!

FRAME\_HNDL: Frame handler welcome!

MAIN: Main client welcome

CONN\_HNDL: Conn not estabished for: ('192.168.0.105', 80)

CONN\_HNDL: Conn not estabished for: ('192.168.0.105', 139)

CLI: Command Line Client Welcome!

CLI: EXIT: ['q']

## Prototyp kodu:

class traceHndl:  
 #variables  
 traceLog = 0  
  
 #constructor  
 def \_\_init\_\_(self):  
 try:  
 self.traceLog = open("log.txt","w") #creating file log.txt  
 self.traceLog.close()  
 except:  
 print("ERROR: unable to open log file")  
  
 def traceAdd(self,traceName,traceBuff): #adding trace to log.txt  
 self.traceLog = open("log.txt","a")  
 self.traceLog.write(str(traceName)+": "+str(traceBuff)+"\n")  
 self.traceLog.close()

# Obsługa ramek ruchu

Obsługa ramek ruchu robota odbywa się za pomocą klasy dedykowanej do tworzenia, wysyłania, weryfikacji i dekodowania ramek ruchu.

## Interakcje

### Dziennik zdarzeń

Obiekt dziennika zdarzeń powinien być przekazywany przez parametr konstruktora. W konstruktorze powinien nastąpić pierwszy zapis do dziennika zdarzeń, potwierdzający poprawną inicjalizację obiektu obsługującego ramki ruchu.

#### Ograniczenia

Z powodu dużej liczby informacji przetwarzanych przez ten obiekt, zapis do dziennika zdarzeń powinien odbywać się zawsze w przypadku sytuacji nie normalnych z punktu widzenia działania całego systemu, jednak, możliwe jest włączenie zapisu do dziennika dodatkowych informacji, w celu ewentualnego złapania i poprawienia błędu w działaniu programu.

### Wysyłanie ramek

Brak oddzielnego interfejsu do przesyłania ramek. Przesyłanie będzie odbywać się za pomocą gettera zwracającego liczbę typu unsigned int.

## Budowa ramki ruchu

### Schemat budowy

| 1110 | XXXX | 0111 |

| nagłówek | treść | zakończenie |

### Pola ramki

#### Nagłówek

Nagłówek składa się z 4 bitów gdzie wszystkie oprócz najmłodszego bitu, ustawione na wartość 1. Najmłodszy bit ustawiony na 0. Takie ustawienie odpowiada liczbie 3584. Z ramki o prawidłowo ustawionym nagłówku po operacji ‘i’ bitowego (bitwise and) z liczbą 3840 powinien wyjść wynik 3584. Każdy inny wynik wskazuje na błąd w nagłówku ramki.

#### Treść

Treść składa się z czterech bitów, gdzie:

Bit nr 1 (najstarszy) – odpowiada za ruch do przodu

Bit nr 2 – odpowiada za ruch w prawo

Bit nr 3 – odpowiada za ruch w lewo

Bit nr 4 (najmłodszy) – odpowiada za ruch do tyłu

#### Zakończenie

Zakończenie składa się z 4 bitów gdzie wszystkie oprócz najmłodszego bitu, ustawione są na wartość 1. Najmłodszy bit ustawiony jest na 0. Takie ustawienie odpowiada liczbie 7. Z ramki o prawidłowo ustawionym zakończeniu, po operacji ‘i’ bitowego z liczbą 16 powinien wyjść wynik 7. Każdy inny wynik wskazuje na błąd w nagłówku ramki.

## Budowa klasy

### Konstruktor ( \_\_init\_\_() )

Konstruktor ma za zadanie wpisać w pola klasy obiekt dziennika zdarzeń, oraz zresetować wszystkie pola, tak żeby żadne niepotrzebne dane nie dostały się w przestrzeń roboczą klasy.

### Ustawienia pól ruchu ( setMovement() )

Metoda klasy mająca na celu ustawienie wszystkich pól części ‘treść’ ramki. Metoda powinna przyjmować parametry odpowiadające ruchowi do przodu, do tyłu, w prawo i w lewo. Parametry powinny być reprezentowane przez wartości prawda/fałsz (true/false)

#### Przód/Tył/Stop

Na podstawie parametrów przód i tył, blok kodu zakoduje odpowiednio:

Jeśli przód i nie tył: najstarszy bit na 1, najmłodszy na 0,

Jeśli nie przód i tył: najstarszy bit na 0 i najmłodszy na 1,

Jeśli nie przód i nie tył: stop, najstarszy i najmłodszy bit na 0, metoda zostanie przerwana z kodem 1.

W każdym innym przypadku funkcja powinna dokonać wpisu w dzienniku zdarzeń z komentarzem o błędzie, oraz przerwać wykonywanie metody z kodem -1.

#### Prawo/Lewo

Na podstawie parametrów prawo i lewo, blok kodu zakoduje odpowiednio:

Jeśli lewo i nie prawo: bit nr 3 na 1 i bit nr 2 na 0,

Jeśli prawo i nie lewo: bit nr 2 na 1 i bit nr 3 na 0,

Jeśli nie prawo i nie lewo: bity nr 2 i nr 3 na 0,

W każdym innym przypadku funkcja powinna dokonać wpisu w dzienniku zdarzeń z komentarzem o błędzie, oraz przerwać wykonywanie metody z kodem -1.

### Budowanie ramki ( getFrame() )

Budowanie ramki odbywa się poprzez zebranie odpowiednich pól i ustawienie ich w odpowiednich miejscach ramki:

Nagłówek: liczba 14 przesunięta bitowo o 8 miejsc w prawo

Zakodowana część ruchu przesunięta bitowo o 4 miejsca w prawo

Zakończenie ramki: dodanie liczby 7 na końcu ramki.

Po zbudowaniu ramki, ramka powinna być zwrócona w formie unsigned int oraz zapisana w dzienniku zdarzeń.

### Sprawdzanie poprawności ramki ( frameValidation() )

Sprawdzenie poprawności danych odbywa się poprzez porównanie ramki przychodzącej do ramki pustej której wartość to 3591. Jeśli nagłówek i ramka są poprawnie zapisane, metoda powinna zwrócić prawdę, w przeciwnym wypadku fałsz,

### Dekodowanie ramki ( decodeFrame() )

Dekodowanie powinno odbywać się poprzez sprawdzenie poprawności ramki przez oddzielną metodę, oraz rozbicie część ‘treść’ na poszczególne bity i zapisanie ich do odpowiadających pól klasy.

## Prototyp kodu

class frameHndl:  
 frameHeader = 7<<1  
 movePart = 0  
 endFrame =15>>1  
  
 # FRAME STRUCTURE  
 # 1110 | xxxx | 0111  
 # header | move | end  
 # X X X X  
 #front right left rear  
 # empty frame = 3591  
  
 #trace part  
 traceName = "FRAME\_HNDL"  
 eventLog = 0  
  
 #constructor  
 def \_\_init\_\_(self,tracePoint):  
 self.eventLog = tracePoint  
 self.eventLog.traceAdd(self.traceName,"Frame handler welcome!")  
  
 def setMovement(self,front,right,left,rear):  
 self.movePart=0<<3;  
  
 #front/rear/stop part  
 if(front and not rear): #front movement -> first bit  
 self.movePart = self.movePart | 1<<3  
 elif((not front) and rear): #rear movement ->last bit  
 self.movePart = self.movePart | 1  
 elif((not front)and (not rear)): #stop -> left/right part not needed  
 self.movePart = 0  
 return 1  
 else: #ups! something goes wrong,  
 self.eventLog.traceAdd(self.traceName,"ERROR: front == rear: f: "+str(front)+" r: "+str(rear)) #error in logs  
 self.endFrame = self.endFrame >> 1 #demage frame , for make sure frame will be not used  
 return -1  
  
 #left/right part  
 if(left and not right): #left  
 self.movePart = self.movePart | 1<<1  
 elif(right and not left): #right  
 self.movePart = self.movePart | 1<<2  
 else: #something goes wrng  
 self.eventLog.traceAdd(self.traceName, "ERROR: left == right: l: " + str(left) + " r: " + str(right)) #error in logs  
 self.endFrame = self.endFrame >>1 #damage of frame  
 return -1  
  
 return 0

#building frame  
def getFrame(self):  
 self.frameHeader = self.frameHeader << 8 #header  
 self.movePart = self.movePart << 4 #movement part  
 frame = self.frameHeader|self.movePart|self.endFrame #all frame  
 self.eventLog.traceAdd(self.traceName, "frame: " + "{0:b}".format(frame)) #write frame in logs  
 return frame  
  
# frame decoder  
def decodeFrame(self, frame):  
 frame\_tmp = frame; # to do not destroy frame  
 if frameValidation(frame\_tmp): #validate if frame is not destroyed  
 self.eventLog.traceAdd(self.traceName, "Frame validation OK: " + "{0:b}".format(frame))  
 else:  
 self.eventLog.traceAdd(self.traceName, "Frame validation NOK: " + "{0:b}".format(frame))  
 return -1 #brake due to wrong frame  
  
# frame valiadation  
def frameValidation(self, frame):  
 frametmp = 7 << 9 | 0 << 8 | 15 >> 1; #clean frame (without any data)  
 frame = frametmp & frame; #if header and end of frame are k, should return empty frame  
 return frame == 3591; #check if empty frame

# Menadżer połączeń

Menadżer połączeń w zależności od miejsca odpalenia, jest serwerem, jeśli jest odpalony na robocie, oraz jest klientem, jeśli jest odpalony na terminalu sterującym robotem.

## Funkcje

### Serwer

Funkcja menadżera połączeń w trybie serwera polega na odebraniu połączenia z terminala sterującego, oraz przekazanie otrzymanych danych do klasy zarządzającej robotem. Powinien także wysyłać dane do terminala sterującego.

### Klient

Klient w pierwszej kolejności powinien zdeterminować własne IP oraz sieć, następnie połączyć się z serwerem na podstawie adresu IP i portu serwera wprowadzonego przez użytkownika. Jeżeli zostanie wybrany tryb skanowania sieci, wtedy klient powinien przeskanować sieć wi-fi pod kątem adresów IP urządzeń dostępnych w sieci, oraz otwartych portów, następnie spróbować połączyć się z IP serwera.

## Interakcje

### Dziennik zdarzeń

Najważniejsze zdarzenia powinny być zapisywane do obiektu dziennika zdarzeń. Własny adres IP oraz w przypadku skanowania adresy IP znalezione razem z portami powinny być przekazane do dziennika zdarzeń. Połączenia nawiązane i przerwane powinny być zapisane w dzienniku zdarzeń, w przypadku zerwania połączenia powinien pojawić się kod błędu. W przypadku wyjątków w działaniu programu powinien zostać zapisany stos procesu, lub inne dane dostępne z poziomu działającego programu.

### Przekazywanie wiadomości

Przekazywanie wiadomości powinno odbywać się poprzez metodę klasy obsługującej połączenie. Wiadomość zwrotna powinna wyzwalać odpowiednią metodę klasy obsługującej wiadomość. Wiadomości nie powinny być w żaden sposób zmieniane ani nadpisywanie przez obiekt klasy zarządzającej połączeniem.

### Połączenie klient <-> serwer

Połączenie klient <-> serwer pomiędzy terminalem i robotem będzie realizowane przez protokół TCP/IP, realizowany przez lokalną sieć Wi-Fi do której będzie podłączony zarówno serwer jak i klient

## Budowa klasy

### Konstruktor ( \_\_init\_\_() )

Konstruktor powinien wpisać obiekt dziennika zdarzeń w odpowiednie pole klasy, oraz odpalić serwer lub klienta w zależności od podanego parametru konstruktora. Powinien też wypisać w linii komend własny adres IP.

### Destruktor ( \_\_del\_\_() )

Destruktor powinien wyczyścić połączenia poprzez zamknięcie używanych socketów internetowych.

### Własne IP ( ownIP() )

Metoda zwracająca adres IP maszyny na której jest uruchomiona. Metoda ta powinna odpowiednio dla systemów Windows i Linux wskazać adres IP interfejsu eth0 i dla Mac OS X adres IP interfejsu en0

### Czy port jest otwarty ( isOk() )

Metoda próbująca utworzyć połączenie do adresu IP i portu podanego w parametrach wywołania funkcji. Ograniczenie czasowe dla próby połączenia powinno wynosić mniej niż 1s, w celu szybszego działania skanera. Jeśli uda nawiązać się połączenie, adres IP i port powinny być zapisane w liście która jest polem klasy.

### Skanowanie ( scanRun() )

Metoda skanująca wszystkie adresy IP w sieci oraz porty od 1 do 150 z pominięciem własnego adresu IP. Metoda ta ma budować adres IP, przekazywać razem z portem do metody ‘isOk()’, oraz w przypadku zwrócenia przez metodę wartości ‘prawda’, zapisać adres IP port w dzienniku zdarzeń.