Software Engineering en Gedistribueerde Applicaties

Eindverslag

Team F.A.C.H.T.

Chris Bovenschen, Harm Dermois,
Alexandra Moraga Pizarro, Tamara Ockhuijsen en Fredo Tan

6104096, 0527963, 6129544, 6060374, 6132421

Universiteit van Amsterdam

1 juli 2011

Inhoudsopgave

1 Inleiding **3**

2 Implementatie **4**
2.1 Low-level . . . 4
2.1.1 Listener . . . 4
2.2 Sensormodules . . . 4
2.2.1 Odometry . . . 4
2.2.2 Range scanner . . . 4
2.2.3 Sonar . . . 5
2.3 Mid-level . . . 5
2.3.1 Collision avoider . . . 5
2.3.2 Wall search . . . 5
2.3.3 Wall follow . . . 5
2.4 High-level . . . 6
2.4.1 Map maker . . . 6
2.4.2 Path finding . . . 6
2.5 Libraries . . . 6
2.5.1 Communicator . . . 6
2.5.2 Movements . . . 7

3 Tests **7**
3.1 Listener . . . 7
3.2 Odometry . . . 7
3.3 Range scanner . . . 7
3.4 Sonar . . . 7
3.5 Collision avoider . . . 7
3.6 Wall search . . . 7
3.7 Wall follow . . . 8
3.8 Wall combo . . . 8
3.9 Map maker . . . 8
3.10 Path finding . . . 8

4 Demonstratie **8**

5 Resultaten **8**

6 Discussie **9**

7 Conclusie **9**

8 Appendix **10**
K.1 Gebruik . . . 14
K.2 Werking . . . 14

1 Inleiding

Dit verslag beschrijft het ontwerpen, implementeren en testen van een gedistribueerde robotapplicatie. De robot die hiervoor wordt gebruikt, is de gesimuleerde P2DX. Deze heeft twee aangedreven wielen en een zwenkwiel. Het platform voor de robotsimulatie is USARSim. Het doel is om met behulp van de sensoren op de robot in een vrij volle ruimte de volgende acties uit te kunnen voeren:

- een pad langs een wand te rijden
- botsingen te vermijden
- zonodig een wand op te zoeken
- uit doodlopende stukken te ontsnappen
- bijhouden waar hij is en welk pad hij heeft gevolgd
- de omgeving in kaart te brengen

Het project is op te delen in verschillende fases. Aan het begin is er nagedacht over een ontwerp met een model, een planning en een specificatie van het eindproduct. Hieruit is een voorlopig werkplan ontstaan. In het definitieve werkplan is een uitgebreidere beschrijving van de realisatie van verschillende componenten met bijbehorende tests te vinden samen met de eerste inleverbare versie van de code. Het grootste deel van de tijd heeft gezeten in de implementatie en tests. Als laatst zijn de componenten geïntegreerd tot een geheel, getest en geoptimaliseerd.

Dit verslag behandelt eerst de implementatie van de componenten met de bijbehorende tests en de opzet van het experiment. Daarna worden de resultaten besproken en hieruit volgt een conclusie.

2 Implementatie

De modules zijn verdeeld in low-level, sensor, mid-level en high-level componenten. De listener is een low-level component, de sonar, laser en odometry zijn sensormodules, de wall search, wall follow en collision avoider zijn mid-level componenten en de map maker en path finding zijn high-level componenten. De movements en communicator zijn libraries die door andere modules kunnen worden gebruikt.

De listener is verbonden met de server via een TCP-verbinding en stuurt de data direct door naar de sensormodules. De sensormodules halen de juiste waarden uit de data die voor hun bestemd zijn en sturen deze door naar de mid-level modules, de map maker en path finding. Berekende kaarten van de map maker kunnen worden gebruikt voor de path finding. Path finding gebruikt de movements library om het pad te kunnen volgen. Als de robot aan het rijden is, staat de collision avoider te alle tijde aan.

2.1 Low-level

De low-level component verkrijgt rechtstreeks data van de robot en stuurt de ontvangen data van de robot direct door naar de sensormodules.

2.1.1 Listener

De listener zit direct aan de server vast. Hij haalt informatie binnen die de robot verstuurt en stuurt ook commando's naar de robot toe. De binnenkomende data worden meteen doorgestuurd naar de sensormodules. Eerst controleerde de listener de data ook op geldigheid en stuurde deze in het juiste formaat door naar de bijbehorende sensormodules. Echter werd de listener overwelmd door de data, waardoor deze nu slechts een doorgever is geworden.

Zie appendix A

2.2 Sensormodules

De sensormodules ontvangen data van de listener. Ze kijken of de data voor hun bestemd zijn en anders negeren ze deze. Ze verkrijgen de sensorwaarden uit de data en kunnen worden aangeroepen voor de meest recente data. Zodra er ongeldige data binnenkomen, wordt de robot gestopt.

2.2.1 Odometry

Odometry haalt de drie waarden op die nodig zijn voor de bepaling van de positie van de robot. Dit zijn de x, y positie en de theta hoek in verhouding tot de startpositie en de richting van de robot. Deze waarden worden opgeslagen en kunnen worden aangevraagd.

Zie appendix B

2.2.2 Range scanner

De range scanner bevindt zich op een bepaalde hoogte boven de robot. Hij scant de omgeving horizontaal door middel van het roteren van de laser in een bepaald

interval. De range scanner leest alle 181 laserwaarden uit. Deze waarden worden opgeslagen en kunnen worden aangevraagd.

Zie appendix C

2.2.3 Sonar

De sonar stuurt een geluidsgolf, waardoor obstakels die op de grond liggen ook te detecteren zijn. Hij leest alle 8 sonarwaarden. Deze waarden worden opgeslagen en kunnen worden aangevraagd.

Zie appendix D

2.3 Mid-level

De mid-level componenten verzorgen het vermijden van botsingen en het zoeken en volgen van een muur.

2.3.1 Collision avoider

De collision avoider kijkt alleen naar de waarden van de sonar, omdat die in tegenstelling tot de laser wel objecten op de grond kan detecteren. Als de kleinste waarde van de sonar kleiner is dan 0,315 stopt de robot met rijden. Bij deze waarde zijn objecten op de grond nog zichtbaar. Zodra deze waarde te klein wordt, kan de sonar het object niet meer zien. Tevens wordt er met behulp van de snelheid uitgerekend na hoeveel seconden de botsing plaats zal vinden als de robot door blijft rijden. Deze berekening wordt gedaan op basis van de snelheid van de wielen, en kan dus in sommige gevallen afwijken.

Zie appendix E

2.3.2 Wall search

Bij het zoeken naar een muur draait de robot eerst 360 graden om te kijken waar de dichtstbijzijnde muur is. Zodra de kleinste sensorwaarde is gevonden, roteert de robot nog een keer totdat hij de juiste positie heeft gevonden. Hij rijdt recht op de muur af totdat hij op een bepaalde afstand van de muur af staat en dan gaat hij over op wall follow.

Zie appendix F

2.3.3 Wall follow

De muur wordt gevolgd die door wall search is gevonden. Als de muur zich meer aan de linkerkant van de robot bevindt, draait hij naar rechts en als de muur zich meer aan de rechterkant van de robot bevindt, draait hij naar links. De kleinste laserwaarde bevindt zich nu aan de rechter- of linkerkant. Hij blijft rechtdoor rijden als de kleinste laserwaarde niet kleiner en groter wordt dan een ingestelde waarde. Als hij te ver van de muur afwijkt, stuurt hij bij richting de muur. Als hij te dicht bij de muur komt, stuurt hij bij van de muur af. Als er geen muur meer wordt waargenomen, wordt de `wall_continued()` functie binnen de wall search aangeroepen. Deze functie kijkt of er in de buurt nog een muur is te vinden. Als dit zo is, gaat hij daarheen. Als er zich in de buurt nog een andere muur bevindt, zoekt wall search naar de nieuwe dichtstbijzijnde muur. Elke

keer wanneer de logica klaar is, wordt er nieuwe informatie opgevraagd van de sensormodules.

Zie appendix G

2.4 High-level

De high-level componenten houden bij waar de robot is geweest en kunnen hem naar een ander punt laten rijden. Dit zijn de modules die meer tijd nodig hebben dan de mid-level modules.

2.4.1 Map maker

De map maker maakt een kaart van de constante stroom van laser en odometry waarden die hij binnenkrijgt. Voor het maken van een kaart wordt de Simultaneous Localization And Mapping (SLAM) techniek gebruikt. De kaart heeft de vorm van een matrix. De bibliotheek die wordt gebruikt om deze map te maken is tinySLAM. Dit is een zeer simpel algoritme dat niet meer dan 200 regels code bevat. Het enige wat het elke keer doet, is de gevonden lijnen tekenen. Dit algoritme heeft nog heel wat uitbreidingen, maar alleen de basis van dit algoritme wordt gebruikt. De originele code is geschreven in C, maar met behulp van een gevonden versie in Python en een paar aanpassingen is deze bruikbaar geworden.

Zie appendix H

2.4.2 Path finding

Path finding kan alleen gebruikt worden binnen de al ontdekte gebieden en kan dus alleen een route plannen binnen een bekend domein. Hierin probeert hij een zo kort mogelijk pad te vinden naar de bestemming. Er wordt gebruik gemaakt van het A ster zoekalgoritme om het efficiëntste pad te vinden tussen twee punten.

Zie appendix I

2.5 Libraries

Alle modules kunnen gebruik maken van de beschikbare libraries. Ze kunnen deze libraries importeren en hun functies gebruiken.

2.5.1 Communicator

De comunicator library leest een bestand in dat de lokaties van de modules bevat. Via deze data maakt hij verbindingen aan met andere modules. De communicator importeert drie thread klasses en een derde klasse. De eerste thread luistert naar de gemaakte verbindingen. De tweede luistert naar een poort om te bepalen of er nieuwe verbindingen gemaakt moeten worden. De derde thread klasse maakt een verbinding aan met een andere module en onderhoud deze. De thread die luistert, handelt de berichten af volgens het communicatieprotocol. Hierin staan alle mogelijke berichten en tags die gebruikt worden bij de communicatie en er staat ook in wat hun doel is. Hierna handelt elke module zijn ontvangen data anders af.

Zie appendix K

2.5.2 Movements

Dit is een library waarin alle bewegingen zijn gedefinieerd. Deze library wordt gebruikt voor de modules die de robot willen besturen. In de movements staan functies gedefinieerd voor het vooruit, naar links, naar rechts en achteruit rijden. Tevens is het mogelijk om naar links en rechts te roteren, een spoor achter te laten, de camera te zien vanuit de robot en de robot te laten stoppen met rijden.

Zie appendix L

3 Tests

Voor elke module is er een aparte test geschreven om er zeker van te zijn dat elke module op zichzelf goed werkt. Elke module wordt direct aan de simulator vastgezet en getest met de directe data die binnenkomen.

3.1 Listener

De listener wordt getest door data te verkrijgen en deze uit te printen.

3.2 Odometry

De odometry wordt getest door de robot te laten rijden, 100 berichten te verkrijgen van de simulator en hiervan de odometriewaarden te printen. De odometriewaarden worden getest op geldigheid.

3.3 Range scanner

De range scanner wordt getest door de robot te laten rijden, 100 berichten te verkrijgen van de simulator en hiervan de 181 laserwaarden te printen. De laserwaarden worden getest op geldigheid.

3.4 Sonar

De sonar wordt getest door de robot te laten rijden, 100 berichten te verkrijgen en hiervan de 8 sonarwaarden te printen. De sonarwaarden worden getest op geldigheid.

3.5 Collision avoider

De collision avoider wordt getest door de robot vanuit verschillende startposities rond te laten rijden zonder dat er botsingen ontstaan.

3.6 Wall search

De wall search wordt getest door de robot vanuit verschillende startposities de dichstbijzijnde muur te laten zoeken.

3.7 Wall follow

De wall follow wordt getest door de robot vanuit verschillende startposities een muur te laten volgen. Het is de bedoeling dat deze module wordt aangeroepen wanneer er al een muur is gevonden. Dan kan de robot de muur naar links en naar rechts toe volgen. Binnen deze gebruikte methode zijn stompe bochten ook mogelijk.

3.8 Wall combo

De wall combo is geschreven om te testen of de wall search goed werkt in samenwerking met de wall follow. Deze test simuleert de werking in een gedistribueerd systeem.

3.9 Map maker

De map maker wordt getest door hem een output bestand met laser- en odometriewaarden van de wall combo te laten lezen en de kaart ervan te tekenen.

3.10 Path finding

De path finding wordt getest door een punt op te geven waar de robot naartoe moet rijden. Hij moet hierbij botsingen ontwijken en naar het punt toe rijden op een zo efficiënt mogelijke manier.

4 Demonstratie

De robot wordt geïnitialiseerd op een bepaalde plaats. Hij zoekt een muur en blijft deze muur volgen. Zodra hij de opdracht krijgt om naar een punt toe te rijden, wordt path finding ingeschakeld. De map maker werkt continu de kaart bij en de collision avoider zorgt ervoor dat er geen botsingen plaatsvinden. Als hij geen nieuwe opdracht krijgt om naar een ander punt toe te rijden, gaat hij weer een muur zoeken en deze volgen.

5 Resultaten

Het is gelukt om alle modules op zichzelf te laten werken. Deze zijn allemaal getest en voldoen aan onze verwachtingen. De integratie ervan is echter nog niet helemaal gelukt. De integratie van zeven van de negen modules is getest. De collision avoider en path finding zijn de enige modules die niet zijn geïntegreerd. Alle sensormodules blijven altijd waarden ontvangen, maar de methodes van de mid-level en high-level modules blijven soms vastzitten. In deze modules wordt er gewacht totdat de data zijn verkregen die zijn verzocht, maar soms blijven ze op dit punt hangen. Dit kan komen doordat een verzoek verloren gaat en de uitvoering kan daarom niet verder gaan. Ook is het gelukt om coreSLAM te gebruiken door de gevonden code aan te passen.

6 Discussie

De map maker zou verbeterd kunnen worden door het tekenen van de kaart in een andere thread en het ontvangen en bijwerken ervan in dezelfde thread plaats te laten vinden. Het tekenen duurt ongeveer twee seconden, want het schrijven ervan gebeurt in een apart bestand die elke keer wordt overschreven. Om dit te verbeteren zou er ook minder getekend kunnen worden. Een verbetering voor de communicator is het uitzoeken waarom er af en toe deadlock plaatsvindt. De berichten zouden ook in een andere vorm opgestuurd kunnen worden waardoor ze sneller gemaakt, verstuurd en geïnterpreteerd kunnen worden.

De wall follow en de wall search hadden beter één module kunnen zijn. Deze twee modules hebben elkaar vaak nodig. Als er één module wordt gemaakt van de twee, zou dit geen problemen opleveren bij het autonoom laten rijden van de robot. Ook kunnen de path finding en map maker in dezelfde module worden geplaatst. De kaart is een heel groot object, het is beter om dat niet te hoeven versturen. Als alles lokaal kan gebeuren, zou dit veel winst opleveren, bijvoorbeeld bij het comprimeren en versturen van de bestanden.

De collision avoider moet nog worden geïmplementeerd en er moet nog een tweede queue worden gemaakt waar belangrijkere commando's in komen te staan. In de wall search kan de logica nog iets worden verbeterd, waardoor een binnenbocht beter kan worden genomen. Als laatste kan er nog een GUI gemaakt worden om het de besturing van de robot gebruiksvriendelijker te maken. Python is makkelijk in gebruik, maar verliest wel performance. Het werkt langzaam en er is minder controle over de datatypes die worden gebruikt. Ook de fouthandeling is moeilijk en niet altijd vanzelfsprekend.

De projectplanning is te strikt aangehouden. Dit zorgt ervoor dat er te weinig tijd is genomen voor de integratie en de communicatie. Dit deel van het project was toch tijdsintensiever dan vooral werd gedacht. Dit is ook te zien in het eindproduct dat nog niet helemaal werkt.

7 Conclusie

Het is gelukt om een modulair systeeem te maken om een robot autonoom te laten rijden. Alle modules zijn op zichzelf klaar voor gebruik. Het gedistribueerde systeem werkt, maar bevat nog een aantal fouten waardoor het nog niet stabiel en snel genoeg is om de robot gedistribueerd te laten werken. Het is een leerzame maand geweest waarin er veel is geleerd over samenwerking en het maken van een groot project in Python.

8 Appendix

A. Listener

De listener stuurt berichten van de robot door naar de modules odometry (ODO), range scanner (RSC) en sonar (SNR). Als een datapakket incompleet is, dat wil zeggen dat er geen `\n` aan het einde is, wordt er een nieuw datapakket achter geplakt.

B. Odometry

De odometry module krijgt data binnen van de listener. De data worden gesplitst in arrays op elke `\r\n` die er in voorkomt. Vervolgens wordt de functie `re.findall('\{[^\}]*\}|\S+', string[i])` gebruikt om elke array te splitsen in elementen die beginnen en eindigen met een accolade en overige elementen. Elke array bevat sensorinformatie als het eerste element van de array gelijk is aan `SEN`. Bij berichten van het type odometry staat er op de tweede plaats in de array het element `{Type Odometry}`. Er is echter tussen het eerste en tweede element het element tijd toegevoegd om te kijken of de odometriewaarden nieuw zijn of niet, waardoor het `{Type Odometry}` element nu op de derde plaats in de array staat. De odometry module gebruikt de array als dit laatste element op die plaats is gevonden. De tijd van het bericht wordt samen met de odometriewaarden op de vijfde plaats in de array als een string meegegeven aan de functie `odometry_module()`. De odometriewaarden komen als volgt binnen: `{Pose 0.0000,0.0000,0.0000}`, oftewel x,y,theta. De `odometry_module()` functie verkrijgt de drie getallen uit deze string, controleert of deze waarden geldig zijn door ze om te zetten in floats en test of de theta tussen -3.15 en 3.15 ligt. Alleen als de waarden geldig en nieuw zijn, worden ze in een string van het formaat `tijd+x,y,theta` doorgestuurd naar de mid-level modules.

C. Range scanner

De range scanner module krijgt data binnen van de listener. De data worden gesplitst in arrays op elke `\r\n` die er in voorkomt. Vervolgens wordt de functie `re.findall('\{[^\}]*\}|\S+', string[i])` gebruikt om elke array te splitsen in elementen die beginnen en eindigen met een accolade en overige elementen. Elke array bevat sensorinformatie als het eerste element van de array gelijk is aan `SEN`. Bij berichten van het type range scanner staat er op de tweede plaats in de array het element tijd en op de derde plaats in de array het element `{Type RangeScanner}`. De range scanner module gebruikt de array als dit laatste element op die plaats is gevonden. De tijd van het bericht wordt samen met de laserwaarden op de zevende plaats in de array als een string meegegeven aan de functie `range_module()`. De tijd en de laserwaarden worden gesplitst en gecontroleerd op geldigheid door ze om te zetten in floats. Alleen als de waarden geldig en nieuw zijn, worden ze in een string van het formaat `tijd+x,x,...,x` doorgestuurd naar de mid-level modules.

D. Sonar

De sonar module krijgt data binnen van de listener. De data worden gesplitst in arrays op elke `\r\n` die er in voorkomt. Vervolgens wordt de functie `re.findall('\{[^\}]*\}|\S+', string[i])` gebruikt om elke array te splitsen in elementen die beginnen en eindigen met een accolade en overige elementen. Elke array bevat sensorinformatie als het eerste element van de array gelijk is aan `SEN`. Bij berichten van het type sonar staat er op de tweede plaats in de array het element tijd en op derde plaats in de array het element `{Type Sonar}`. De tijd van het bericht wordt samen met de sonarwaarden als een string meegegeven aan de functie `sonar_module()`. Deze functie splitst de tijd en de sonarwaarden en controleert ze op geldigheid door ze om te zetten in floats. Alleen als de waarden geldig en nieuw zijn, worden ze in een string van het formaat `tijd+1,2,3,4,5,6,7,8` doorgestuurd naar de mid-level modules.

E. Collision avoider

De collision avoider module maakt gebruik van een semafoor 'flag'. Zodra de flag gelijk is aan nul, zal de robot vooruit rijden en wordt `getdata()` aangeroepen. BLABLABLA. Zodra het type sensor gelijk is aan sonar wordt er gekeken of de lengte van de data groter is dan 9. Hierin wordt de `min_sonar_val()` functie aangeroepen met als parameter de meegegeven sonarwaarden. Hierbij wordt de `string_to_float()` functie gebruikt. Elke sonarwaarde wordt omgezet in een float en deze floats worden vervolgens in een array gestopt en geretourneerd. Deze worden gesorteerd en er wordt 1 bij de eerste indexwaarde opgeteld, zodat er van 1 tot 8 wordt geteld in plaats van 0 tot 7. De `min_sonar_val()` retourneert twee waardes, waarvan de eerste de minimale waarde is en de tweede de indexwaarde waar zich de minimale waarde bevindt. Voor elke sonarwaarde wordt er gekeken of de float van elke waarde kleiner is dan de zogenaamde level, die als .25 is gegeven. Dit zal de robot genoeg ruimte geven om te kunnen roteren zonder dat deze in aanraking komt met een object. Als één van de twee meest rechter sonarwaarden kleiner zijn dan de meest linker sonarwaarde, wordt de robot naar links gedraaid. Zijn de twee meest linker sonarwaarden kleiner dan de meest rechter sonarwaarde, dan wordt de robot naar rechts gedraaid. Zodra de minimale waarde bij de voorste twee indexwaarden ligt en de flag gelijk is aan nul, wordt de `calc_collision()` functie aangeroepen voor een mogelijke botsing. Door middel van de `calc_collision()` en de `calc_speed()` kan de tijd in secondes worden berekend voordat de robot in aanraking komt met een object. Als de meest rechter sonarwaarde eveneens kleiner is dan de meest linkerwaarde zal de robot naar rechts draaien, en andersom. Tenslotte wordt er een 1 geretourneerd.

F. Wall search

G. Wall follow

De wallfollow haalt elke iteratie laser waardes op en gaat daarmee werken. In de wallfollow gebruiken we de `min_laser_val` om de kleinste waarde te vinden

en op welke index die waarde zit. We gebruiken een flag om de de treshold te bepalen die je van de muur af mag gaan. Dit doen we omdat als je een muur hebt gevonden dan mag je wat ruimer rijden, omdat je toch weet dat je een muur aan het volgen bent, maar je moet toch een treshold hebben want je moet nog wel kunnen bepalen wanneer de muur ophoudt. De logica begint door te kijken te kijken of je voorste lasers iets te dichtbij vinden. Dit zou betekenen dat er een muur voor je zit. Je bent een muur aan het volgen dus je kijkt of de kleinste waarde links of rechts van het midden zit. Hierna ga je van de muur wegdraaien. Dit werkt beide kanten op. Hierna wordt er gekeken of je te dicht bij de muur zit. Als dit zo is wordt er eerst gekeken of er aan de andere kant ook iets dicht bij zit. In dit geval blijf je rechtdoor gaan anders wordt er bij gestuurd. Daarna bekijken we of de robot te ver van de muur is en sturen we bij naar de muur. Als dit allemaal niet waar is dan roepen we de `wallfollow` aan. Deze checkt eerst nog even of we echt een muur aan het volgen zijn en aan welke kant die zit en dan geeft hij het commando om rechtdoor te rijden. Wanneer we niet binnen de treshold blijven gaan we over naar de `wallsearch` die dan weer een nieuwe muur gaat zoeken.

H. Map maker

De mapmaker maakt gebruik van tinySLAM een bibliotheek voor SLAM die te vinden is op Hier hebben we een python versie van gevonden. Hoe deze python bibliotheek werkt is. Elke keer dat je de map wilt updaten moet je de methode `makeMap` aan te roepen. De laserwaardes kunnen direct meegegeven worden aan deze methode, maar de positie die je van de odometry mee krijgt moeten eerst worden getransformeerd naar het midden van de het plaatje. Ook wordt de theta van de odometry omgezet van radialen naar graden. In de `makeMap` methode wordt voor elke laserstralen een beetje aan gepast en daarna de `ts_map-update` aangeroepen die vervolgens alle stralen in de map tekent. Hierna kun je op elk moment een map tekenen met de methode `drawMap`. Deze methode krijgt een map mee om te tekenen. Binnen de methode gaat hij alle punten op je map af en tekent een grijze/witte/zwarte pixel afhankelijk van wat er in de map staat. Deze pixels worden geschreven naar een .ppm bestand, want de teken bibliotheken die er standaard werden gebruikt waren niet aanwezig op de computers.

I. Path finding

J. Communicatieprotocol

```
TAG:
REQ: data sturen
RCV: data ontvangen
CMD: movement commando's
NEX: go to the next module

data TAG:
SNR: sonar
```

```
ODO: odometry
RSC: rangescanner
MAP: map

module TAG:
ODO: Odometry
RSC: Rangescanner
SNR: Sonar
WFW: Wall Follow
CAV: Collision Avoider
WSC: Wall Search
MAP: Map Maker
PLA: Route Planner
LIS: Listener

dividers:
#: eindetransmissie
!: transmissie divider

Vorm:
REQ:
REQ!"own module TAG"#
example: REQ!WFW# this send to a sensor module will return the data of that module.
RCV!"data TAG"!"data"#
example: RCV!SNR!x,x,x,x,x,x,x,x# this will send the data to a module.
CMD!"commando"# CMD!drive# will send a drive command.
NEX!"side"# example: NEX!1# 1 if wall is on the right side.


data:
SNR:
naar sensor module:
{Name F1 Range "x"} {Name F2 Range "x"} {Name F3 Range "x"} {Name F4 Range "x"}
{Name F5 Range "x"} {Name F6 Range "x"} {Name F7 Range "x"} {Name F8 Range "x"}
uit sensormodule:
{Time}+x,x,x,x,x,x,x,x

RSC:
naar sensor module:
{Range "x","x",...,"x"}
uit sensormodule:
{Time}+x,x,x,x,x,x,x,x,x,x,....,x,x,x

ODO:
naar sensor module:
{Pose x,y,theta}
uit sensormodule:
{Time}+x,y,theta

MAP:
```

```
"# of characters""type"_"# of characters""type"_...._"# of characters""type"
```

K. Communicator

K.1 Gebruik

Elke module maakt via deze library connecties aan. Dit doen ze door eerst een configreader aan te maken. Daarna het opstarten van de acceptor thread. Deze thread moet op Daemon gezet worden zodat hij sluit wanneer het hoofdprogramma sluit.

```
{}
configreader = config_reader()
print("config reader gestart")
accept_thread = acceptor(running, list, "LIS", configreader.addresses)
accept_thread.setDaemon(True)
accept_thread.start()
```

Hierna kan je de connecties op gaan zetten. Dit doe je op de volgende manier.

```
{}
sonar = connection(running, "SNR", configreader, list)
sonar.setDaemon(True)
sonar.start()
```

Deze code zet een connectie op met de sonar module. Gebruik `sonar.send_data(message)` om iets te versturen over de gemaakte connectie deze connectie zal zelf kijken of de connectie nog wel leeft en indien hij niet meer werkt dan zal hij opnieuw proberen te connecten met de module.

K.2 Werking

De config_reader leest eerst de config.cfg file in een array. Deze wordt dan gesplitst en in de klasse als global gestopt met de naam addresses.

```
f = open('config.cfg')
addresses = []
config = f.readlines()
for i in range(len(config)):
    config[i] = config[i].strip()
    addresses.append(config[i].split(' '))
self.addresses = addresses
```

De andere methode in de config_reader geeft een socket terug waarover de gevraagde connectie is gemaakt. Hij kijkt eerst in de de meegekregen list. Of de connectie die gevraagd is al gemaakt is hiervoor heeft hij het ip adres nodig deze haalt hij uit de global addresses. Indien de connectie al is gemaakt dan geeft hij de socket van die connectie terug. deze wordt gelezen uit de list.

```
for i in range(len(self.addresses)):
    if self.addresses[i][0] == module:
        address = self.addresses[i]
```

```
        ip = socket.gethostbyname_ex(address[1])
for i in range(len(list)):
    if list[i][1][0] == ip[2][0]:
        return list[i][0]
```

De acceptor maakt eerst wat globale variabelen aan waaronder de thread waarvan hij luistert naar connecties. Een deel van deze variabelen wordt meegegeven.

```
def __init__ (self, running, list, module, addresses):
    threading.Thread.__init__(self)
    self.running = running
    self.list = list
    self.module = module
    self.addresses = addresses
    # array with the received data:
    # [0]: sonar
    # [1]: odometry
    # [2]: rangescanner
    # [3]: map
    # [4]: command
    # [5]: nex
    self.memory = ["","","","","",""]
    self.waiting_for_data = 0
    self.request_data = []
    communicationthread = communication(self.list, self.running, self.memory, self.waiting
    communicationthread.setDaemon(True)
    communicationthread.start()
    self.communicationthread = communicationthread
```

De memory is een array waarin de ontvangen data wordt opgeslagen tot hij wordt verwerkt. De twee methodes erna zorgen ervoor dat de modules kunnen wachten op data. Deze nethoden refereren naar methodes in de luister thread.

```
def set_wait(self, number):
    self.communicationthread.set_wait(number)
def get_wait(self):
    return self.communicationthread.get_wait()
```

De main van de thread zorgt ervoor dat er connecties kunnen worden gemaakt naar de module waar deze thread in is opgestart. Hij kijkt in de addresses variabel op welke IP en port hij moet luisteren. daarna begint hij met luisteren op deze socket. Daarna gaat hij in een loop waarin hij blijft wachten tot er iemand wilt connecten met deze module. Als er een connection binnenkomt dan zet hij deze connection in de list.

```
for i in range(len(self.addresses)):
    if self.addresses[i][0] == self.module:
        TCP_IP = self.addresses[i][1]
        TCP_PORT = int(self.addresses[i][2])
BUFFER_SIZE = 1024
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
    s.bind((TCP_IP, TCP_PORT))
    s.listen(0)
    s.setblocking(0)
except socket.error:
    print 'kip'
    self.running = 0
    s.close()
    self.communicationthread.join()
    sys.exit()
while self.running:
    try:
        connection = s.accept()
        connection[0].setblocking(0)
        print 'voor append'
        self.list.append(connection)
        print 'na append'
    except(socket.error):
      if self.running == 0:
          break
```

De communication thread is de thread die luistert naar de connecties om te kijken of ze data ontvangen. Als eerste maakt hij wat globale variabelen aan die hij mee heeft gekregen dit zorgt ervoor dat ze linked zijn met de variabelen in de acceptor thread.

```
def __init__(self, list, running, memory, waiting_for_data, request_data):
    threading.Thread.__init__(self)
    self.list = list
    self.running = running
    self.memory = memory
    self.waiting_for_data = waiting_for_data
    self.request_data = request_data
```

Daarna 2 methodes waarnaar gerefereerd werd in de acceptor thread.

```
    def set_wait(self, number):
        self.waiting_for_data += number
    def get_wait(self):
        return self.waiting_for_data
```

De main van deze thread handelt de berichten af volgens het protocol. Elk ontvangen bericht wordt anders afgehandeld door middel van if else statements. Ook zorgt deze thread ervoor dat als een connectie dood is dan wordt hij hier uit de lijst verwijderd.

```
def run(self):
    BUFFER_SIZE = 1024
    data = ""
    data_incomplete = 0
    while self.running:
        if(len(self.list) == 0):
            time.sleep(0.1)
```

```
for i in range(len(self.list)):
while 1:
    try:
        self.list[i][0].setblocking(0)
        data = self.list[i][0].recv(BUFFER_SIZE)
            if not data:
            self.list.pop(i)
            print i
            print "pop van list"
            break
        if data_incomplete:
            datatemp += data
            data_incomplete = 0
            data = datatemp
        if(len(data) > 1):
            if data[len(data)-1] != '#':
                datatemp = data
                data_incomplete = 1
                continue
        else:
            continue
        messagesplit = data.split("#")
        for j in range(len(messagesplit)):
            datasplit = messagesplit[j].split("!")
            if len(datasplit) < 1:
                continue
            if(datasplit[0] == "REQ"):
                self.request_data.append(datasplit[1])
                data = " "
                datasplit = []
            elif(datasplit[0] == "CMD"):
                self.memory[4] = datasplit[1]
                data = " "
                datasplit = []
            elif(datasplit[0] == "RCV"):
                if(datasplit[1] == "SNR"):
                    self.memory[0] = datasplit[2]
                    if(self.waiting_for_data > 0):
                        self.waiting_for_data -= 1
                    data = " "
                    datasplit = []
                elif(datasplit[1] == "ODO"):
                    self.memory[1] = datasplit[2]
                    if(self.waiting_for_data > 0):
                        self.waiting_for_data -= 1
                    data = " "
                    datasplit = []
                elif(datasplit[1] == "RSC"):
                    self.memory[2] = datasplit[2]
                    if(self.waiting_for_data > 0):
```

```
                                self.waiting_for_data -= 1
                            data = " "
                            datasplit = []
                        elif(datasplit[1] == "MAP"):
                            self.memory[3] = datasplit[2]
                            if(self.waiting_for_data > 0):
                                self.waiting_for_data -= 1
                            data = " "
                            datasplit = []
                    elif(datasplit[0] == "NEX"):
                        self.memory[5] = datasplit[1]
                        data = " "
                        datasplit = []
                except(socket.error):
                    if self.running == 0:
                        break
                except(IndexError):
                    break
                break
```

De laatste thread klasse in deze library is de connection deze begint ook met wat globale variabelen te initialiseren deze variabelen worden meegegeven wanneer hij wordt aangemaakt.

```
def __init__ (self, running, module, configreader, list):
    threading.Thread.__init__(self)
    self.running = running
    self.module = module
    self.list = list
    self.configreader = configreader
    self.socket = self.configreader.connection(self.list, self.module)
    self.connected = 1
```

De methode van deze klasse stuurt meegekregen data over de socket van de klasse.

```
def send_data(self, data):
    if self.connected:
        self.socket.send(data)
```

De main onderhoud de connectie op een soortgelijke manier als de config_reader een al gemaakte connectie teruggeeft. Als hij al gemaakt is dan probeert hij opnieuw te connecten. en zet hij zijn connected flag op 0 voor not connected.

```
def run(self):
    while self.running:
        for i in range(len(self.configreader.addresses)):
            if self.configreader.addresses[i][0] == self.module:
                address = self.configreader.addresses[i]
                ip = socket.gethostbyname_ex(address[1])
        for i in range(len(self.list)):
            if self.list[i][1][0] == ip[2][0]:
```

```
            self.connected = 1
            break
        else:
            self.connected = 0
    if not self.connected:
        self.socket = self.configreader.connection(self.list, self.module)
```

L. Movements

Elke module kan een commando type aanroepen in de movements library via de `handle_movements()` functie. Hierin staan alle mogelijke bewegingen die behandeld kunnen worden. `go_drive()` zorgt ervoor dat de robot vooruit en achteruit kan rijden, bochten en rotaties kan maken. Hierbij wordt voor zowel het linker- als het rechterwiel dezelfde parameter meegegeven. Bij `go_rotate_right()` en `go_rotate_left()` wordt een rotatie uitgevoerd om de as van de robot. De meegegeven parameter bepaalt de snelheid van de rotatie. Om naar rechts te draaien moet de parameter van het rechterwiel groter zijn dan die van het linkerwiel en omgekeerd. De functies `go_right()` en `go_left()` worden hierbij gebruikt. De robot remt bij het gebruik van `go_brake()`, hierbij staan beide wielen op een snelheid van 0. Door de wielen een gelijke negatieve waarde mee te geven in de functie `go_reverse()`, zal de robot achteruit rijden. De functie `go_camera()` zorgt ervoor dat door middel van het indrukken van de linkermuisknop het beeld zichtbaar is vanuit de camera van de robot. Om hier vervolgens uit te komen, dient er op de rechtermuisknop gedrukt te worden.