



Automatisering in de distributiecentra

Het onderzoeken, ontwerpen en bouwen van een automatisch distributiecentrum

Inhoudsopgave

| | |
|--|----|
| 1: Abstract..... | 2 |
| 2: Inleiding..... | 3 |
| 3: Voorafgaand onderzoek..... | 4 |
| 4: Ontwerp | 6 |
| 4.1 Mechanische componenten | 6 |
| 4.2 Electronische componenten | 9 |
| 4.3 Software..... | 10 |
| 5: Bouwen | 11 |
| 5.0 Bouwstappen | 11 |
| 5.1 Pre-discussie | 11 |
| 5.2 Materialen..... | 11 |
| 5.3 Schematics | 12 |
| 5.4 Bouwen (hardware) | 13 |
| 5.5 Bouwen (Software) | 18 |
| 5.5.1 Microcontroller | 18 |
| 5.5.2 Raspberry Pi | 20 |
| 5.5.3 Host..... | 26 |
| 5.6 Interactie met de daemon en implementatie van het product..... | 30 |
| 6: Resultaten | 32 |
| 7: Conclusie | 34 |
| 8: Discussie..... | 36 |
| 9: Bibliografie | 37 |
| 10: Evaluatie..... | 38 |
| 10.1 Evaluatie Mike..... | 38 |
| 10.2 Evaluatie Hugo | 39 |
| 11: Logboek..... | 40 |

1: Abstract

In dit verslag zullen we onderzoeken hoe automatisering in de distributiecentra werkt en zullen we een systeem ontwerpen. Na het ontwerpen van het systeem zullen we een prototype maken.

Bij het bouwen van het systeem zullen we op zowel de hardware als de software in gaan.

Daarna zullen we conclusies trekken, discussies stellen en evalueren.

Uit de resultaten valt afte leiden dat het systeem werkt, maar er nog ruimte is voor verbetering.

2: Inleiding

Online shoppen gaat vandaag de dag heel erg makkelijk. Je bestelt de producten, betaalt en klaar! Tegenwoordig kopen Nederlanders meer online dan in de normale winkels. Maar er komt heel wat kijken bij het proces tussen het binnenkrijgen van de order en het uiteindelijk leveren van het product. In dit verslag wordt er een systeem gebouwd dat een bijdrage levert aan het distributieproces.

Dus onze hoofdvraag is:

Hoe kan een webshop-order zo snel mogelijk verwerkt kan worden na binnenkomst met behulp van automatisering?

Met de deelvragen:

1. Hoe ziet het huidige distributieproces eruit in een distributiecentrum van een gemiddelde webshop en kan dat goedkoper?
2. Hoe kunnen we dat gebied automatiseren?
3. Hoe zou het systeem gebouwd kunnen worden?

Het uiteindelijke doel is om een werkend en betrouwbaar systeem te hebben dat die bijdrage kan leveren, bijvoorbeeld in de vorm van robotica.

Ons onderzoek bestaat uit het bekijken van filmpjes van bedrijven die laten zien hoe het proces in hun distributiecentra verloopt. Daaruit laten we een percentage zien in hoeverre het distributieproces geautomatiseerd is.

We gaan het zo aanpakken:

1. [Een klein onderzoek over het probleem en de oplossingen die te bieden zijn.](#) (Deelvraag 1)
2. [De ontwerpfase, waarbij we het systeem ontwerpen met de oplossing gevonden bij ons onderzoek.](#) (Deelvraag 2)
3. [Het bouwen van een prototype.](#) (Deelvraag 3)
4. [Conclusie.](#) (Hoofdvraag)

3: Voorafgaand onderzoek

Voordat gestart kan worden aan het bouwen van een systeem moet er gekeken worden waar de problemen in de huidige distributieprocessen liggen en hoe ze opgelost kunnen worden.

Omdat er geen mogelijk was tot het bezoeken van een distributiecentrum, is dit onderzoek gebaseerd op films die sommige bedrijven publiceren.

Deze films laten ons zien hoe een webshop-order wordt verwerkt in een distributiecentrum. Het distributieproces verschilt van webshop tot webshop, maar in grote lijnen is het allemaal hetzelfde.

De volgende stappen worden ondernomen:

1. Het verzamelen van producten: een medewerker zoekt alle producten op en verzamelt die.
2. De producten inpakken in de doos: Een medewerker zet alle producten in een doos.
3. De doos dichtmaken: Een medewerker maakt de doos dicht en maakt de doos klaar voor verzending. Soms geautomatiseerd
4. Transport naar vrachtwagen: Een systeem sorteert en brengt de doos naar de juiste plek

Wat opvalt is dat vooral, het verzamelen en inpakken van producten nog handmatig gaat. Bij alle processen daarna neemt automatisering het over.¹

We kunnen het percentage van automatisering (**LoA**, Level of Automatisation) gaan schatten (zie tabel OD.1). Dat schatten doen we puur door te kijken in de filmpjes en te kijken waar mensen nog werk verrichten.

Erg accuraat kunnen deze resultaten niet zijn, omdat het hier grotendeels over promotie video's gaat en niet alles goed wordt weergegeven. Maar we kunnen wel een schatting maken van het percentage LoA

Tabel OD.1: schematische weergave van automatisering in een typisch distributiecentrum

| Proces | Geautomatiseerd? | LoA |
|-----------------------------|------------------|-------|
| Producten verzamelen | Nee | 0% |
| Producten inpakken in doos | Nee | 0% |
| Doos dichtmaken | Soms | 10% |
| Transport naar vrachtwagens | Semi | 80% |
| Totaal gemiddelde | - | 22.5% |

22.5% LoA is niet veel

In de video 'Nieuw apparaat van 200 meter lang'² is te zien dat alles wordt verzameld om een soort plateau waarbij de machine zelf het pakket maakt. Dit betekent dus dat daar al een product voor is dat door in dit geval CoolBlue al wordt gebruikt.

Dus het waar dit verslag het zou beste kunnen richten is bij het verzamelen en inpakken van producten.

¹ (RTL Nieuws, 2016)

² (Coolblue, 2017)

Dus wij richten ons in dit verslag met name op het verzamelen en inpakken van producten, waarbij we specifiek inzoomen op het onderdeel: verzamelen.

Waar ook naar gekeken wordt is de vraag of automatiseren goedkoper is dan de inzet van werknemers. Na bij verschillende vacatures van verschillende webshops te hebben gekeken kwamen we uit op een gemiddeld uurtarief van €11,-. Dit is €1800,- per maand³, per werknemer. Sommigen geven tijdens een nachtdienst 50% toeslag³! Dus de kosten lopen vrij hoog op.

Een geautomatiseerd systeem is een eenmalige aankoop. Maar zou daarna moeten worden onderhouden door ander personeel. Dit hoeft niet zoveel te zijn als het aantal orderpicker-medewerkers

In Tabel OB.2 wordt de vergelijking gemaakt op jaarbasis van een distributiecentrum met 50 orderpickers die constant werken. We gaan ervan uit dat een onderhoudsmedewerker €20,- per uur kost en we maar 10 onderhoudsmedewerkers nodig hebben. (Een aanname zonder bron, maar logisch om te stellen omdat er niet evenveel monteurs nodig zijn). De aanschafprijs van het systeem is €100.000. Eventuele kosten qua belasting is te verwaarlozen omdat dit geen effect heeft op het resultaat. We gaan ervan uit dat één jaar 52 weken heeft.

| Scenario | Systeemkosten | Medewerkers | Uurtarief | Totaal (per jaar in €) |
|-------------------------------------|---|-------------|-----------|------------------------|
| $W_u = 40$ (aantal uur per week) | | | | $(U * N * W_u * 52)$ |
| Orderpickers | 0 | 50 | 11 | €1,1 miljoen |
| Automatisch systeem | 100.000 (eenmalige aankoop, daarna kan het jaren mee) | 10 | 20 | €0,4 miljoen |

We kunnen dus concluderen dat het gebruikmaken van een geautomatiseerd systeem 63% goedkoper is dan bij inzet van orderpickers in dit geval.

Conclusie: op de vraag: *“Hoe ziet het huidige distributieproces eruit in een distributiecentrum van een gemiddelde webshop en kan dit goedkoper?”* is het antwoord dat het distributieproces vooral handmatig wordt gedaan, en het duurder uitkomt vergeleken met een volledig geautomatiseerd proces.

³ (Werken bij Coolblue, nvt)

4: Ontwerp

Het is nu tijd om het systeem te gaan ontwerpen!

4.1 Mechanische componenten

We kunnen het probleem “Hoe kunnen we dit gebied automatiseren?” op verschillende vlakken oplossen.

Het systeem moet namelijk aan meerdere eisen voldoen:

- Het moet snel te werk kunnen gaan
- Het aantal ‘zwakke punten’⁴ moet minimaal zijn
- Het moet makkelijk te onderhouden zijn, zodat er bij fouten snel kan worden gehandeld.
- Het moet veilig zijn

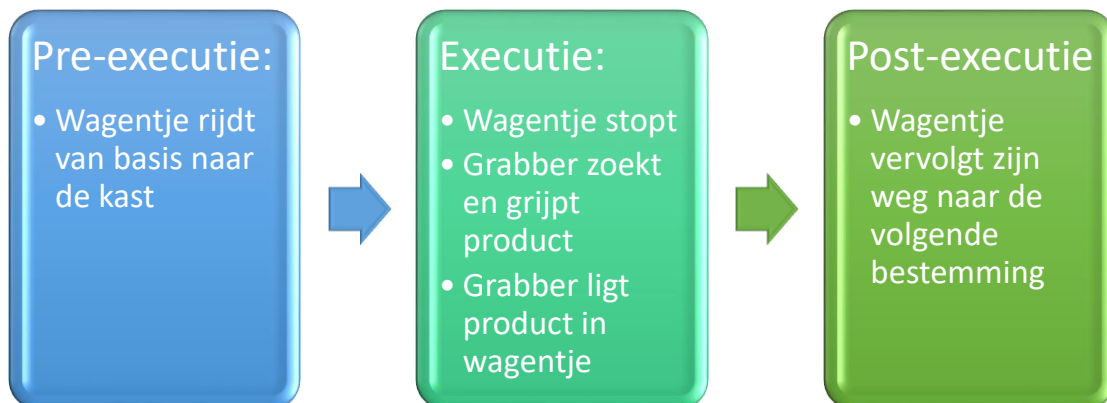
Om potentieel op alle vlakken iets te kunnen betekenen kwamen we snel op het idee om wagentjes te gebruiken al het transport. Hierbij is uitbreiding het makkelijkst mogelijk. Maar voor het grijpen van de producten is er meer aandacht nodig.

Met deze eisen in het achterhoofd zijn er twee verschillende concepten uit voorgekomen die we zelf hebben bedacht:

Concept 1: 2D Grabber

Dit concept is zeer simpel. Er is sprake van een kast met een gripper die zich in twee dimensies kan voortbewegen (x- en y-as) (zie visualisaties OW.3 t/m OW.5).

De verwachte flow⁵ wordt in figuur OW.1 weergegeven.



Figuur OW.1: De verwachte flow van concept 1

Het systeem wordt aangedreven door drie stappenmotoren die de benodigde kracht hebben om een bepaald gewicht aan te kunnen. Twee voor de lift (y-as) en één voor het bewegen van de x-as. Een visualisatie van het systeem zijn te zien in de figuren OW.3 t/m OW.5.

⁴ Met de zwakke punten wordt bedoeld bij alle punten die kunnen falen of het proces, in elke mogelijke manier, kunnen vastlopen

⁵ Met de ‘flow’ wordt bedoeld hoe een proces verloopt.

De belangrijkste voordelen van dit systeem zijn:

- Grote kasten met grote hoogtes zijn makkelijk mogelijk te maken met dit systeem
- Systeem is met stappenmotoren relatief snel en accuraat
- Vrij makkelijk voor te programmeren. Je hoeft alleen de posities voor te programmeren.

De belangrijkste aantal nadelen zijn:

- Bij zeer grote kasten kan het lang duren voordat de gewenste positie is bereikt
- Het is een serieel systeem. Dat wil zeggen het systeem maar één ding tegelijk kan doen. Dus bij veel vraag is de kans groot dat wagentjes moeten wachten totdat het systeem andere wagentjes heeft gevuld.
- Storingsgevoelig: Als de kast uitvalt, valt direct een hele producten-reeks weg.

Dit systeem is prima, maar kent dus zijn beperkingen.

Concept 2: Vervoersmiddel met ingebouwde grijparm

Bij dit systeem worden de wagentjes zelf gebruikt om producten te verzamelen door middel van een robotarm die aan het wagentje is vastgemaakt.

De verwachte flow wordt in figuur OW.2 weergegeven

De grijparm bevat slechts een lineaire actuator. Een visualisatie van het systeem is te zien in figuur OW.8



Figuur OW.2: De verwachte flow van concept 2

De belangrijkste voordelen zijn:

- Systeem is parallel. Er kunnen meerdere producten uit een kast worden gehaald op hetzelfde tijdstip, op voorwaarde dat ze niet elkaar in de weg zitten.
- Minder storingsgevoelig: Als een wagentje uitvalt, dan kan die makkelijk worden vervangen
- Minste kans op storingen in de flow. Producten worden direct zelf gepakt.

De belangrijkste nadelen zijn:

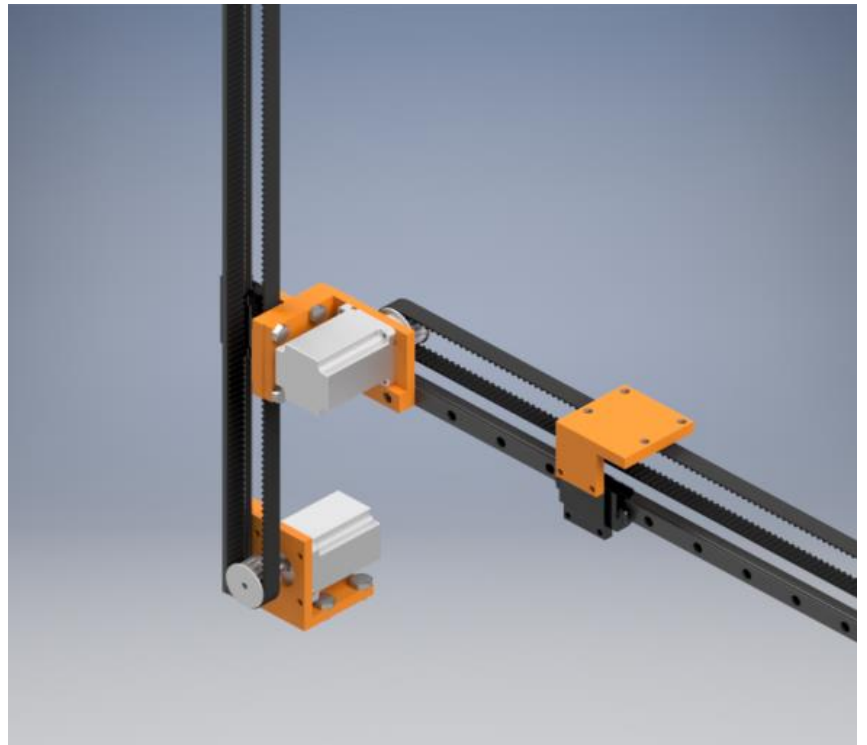
- Kan geen grote hoogteverschillen tolereren. Dit komt vanwege het zwaartepunt die dan te hoog komt te liggen
- Vrij complex systeem, moeilijk voor te programmeren. Het is zo moeilijk om voor te programmeren want je moet rekening houden met waar de wagentjes zijn op de nul positie en hoeveel wagentje er zijn en waar de producten zijn. Je werkt namelijk met meer variabelen. Het is zo

Ook dit systeem is een goed systeem. Het is alleen in contrast met de 2D Grabber een minder goed systeem als de kasten hoog zijn. Dan is dit systeem minder .

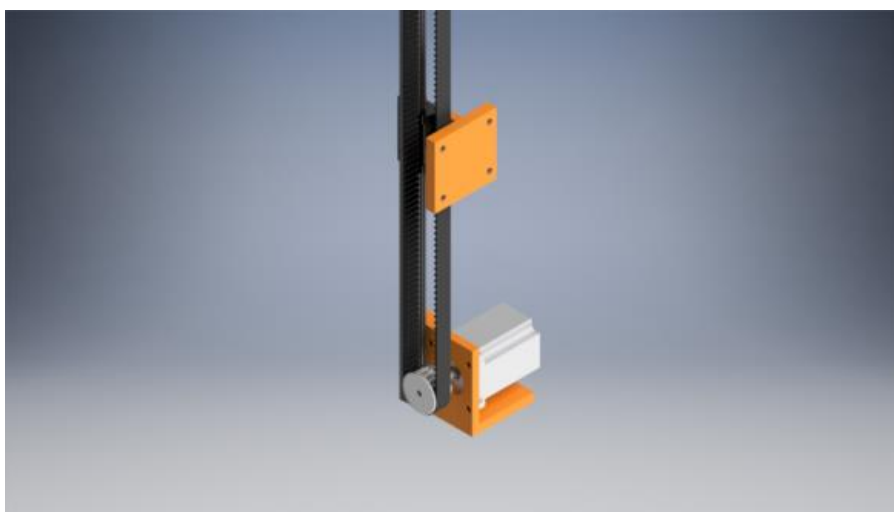
Met alle voor- en nadelen opgeteld kan geconcludeerd worden dat het tweede concept (Vervoersmiddel met ingebouwde grijparm) een beter systeem is dan de 2D Grabber. De grootste limitatie bij de 2D grabber is zijn seriële paradigma. Hierdoor is bij uitval direct een groot gedeelte van het distributieproces niet toegankelijk totdat het probleem is opgelost.



Figuur OW.3: Een visualisatie van concept 1.



Figuur OW.5: Connectie van de x-as aan de y-as.



Figuur OW.4: De stepper motor in detail

4.2 Electronische componenten

Qua elektronica hebben we een compact computersysteem nodig voor de besturing en communicatie tussen de host⁶ en de wagentjes. Hierbij is een Raspberry Pi⁷ zeer geschikt. Bij deze situatie is een Raspberry Pi Zero de beste optie. Deze computer is net wat groter dan een USB stick en heeft de volledige functionaliteit die we willen hebben. Daarbij is het computertje voor slechts €5,- te kopen en voor €10,- met WiFi en Bluetooth functionaliteit. We zullen deze computer gebruiken om de instructies, die via WiFi worden verzonden, te verwerken. Maar omdat de Raspberry Pi Zero (Figuur OW.6) maar één processing thread⁸ heeft is het raadzaam om een microcontroller⁹ (bijv. een Arduino) alle mechanische onderdelen te laten besturen en een andere de coördinaten van het navigatiesysteem te verwerken. Er zijn drie protocollen waarmee de Raspberry Pi en de microcontroller mee kunnen communiceren: SPI, UART en I²C¹⁰. Alle protocollen kunnen worden geïmplementeerd. Maar omdat misschien er sprake is van uitbreiding wordt gebruik van I²C en SPI aangeraden. Welke van de twee gebruikt moeten worden hangt af van de applicatie. Een groot verschil tussen I²C en SPI is dat SPI bi-directioneel kan communiceren (full-duplex), terwijl I²C dit niet kan (half-duplex). Zo zou de microcontroller die de motoren aanstuurt beter af zijn op I²C terwijl het navigatiesysteem SPI nodig heeft, omdat die telkens coördinaten geeft aan de Raspberry Pi.

Voor de microcontroller maakt het eigenlijk niet uit welke architectuur¹¹ wordt gebruikt, maar het gebruik van development boards is afgeraden zoals een Arduino (Figuur OW.7). Dit komt omdat development boards niet gespecialiseerd doel hebben en dus impact kunnen hebben op de prestaties van de chip. Dit geldt niet voor het doorontwikkelen van prototypes, daar zijn development boards juist wel goed. De microcontrollers zelf worden vaak gebruikt in industriële processen, en zijn dus geschikt voor industriële doeleinden. Ze kunnen uiteraard voor prototypes en particuliere doeleinden worden gebruikt.



Figuur OW.7: Een Arduino Uno development board



Figuur OW.6: Een Raspberry Pi Zero

⁶ Een host in de computerwereld kun je zien als een CEO in een bedrijf: De host geeft opdracht aan de andere devices en die verwerken die opdracht.

⁷ (Raspberry Pi, sd)

⁸ In [computer science](#), a **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a [scheduler](#), which is typically a part of the [operating system](#). (Wikipedia, sd)

⁹ Een **microcontroller** is een [geïntegreerde schakeling](#) met een [microprocessor](#) die wordt gebruikt om elektronische apparatuur te besturen. (Wikipedia, sd)

¹⁰ SPI = Serial Peripheral Interface, UART = Universal Asynchronous Receiver/Transmitter.

¹¹ Met de **computerarchitectuur** wordt de opbouw van de fundamentele operationele structuur van een computersysteem bedoeld. (Wikipedia, sd)

4.3 Software

Nu er een basis is voor de hardware, kunnen we gaan kijken naar de software.

Qua software moeten aan de volgende eisen doen:

- Stabiel, zo min mogelijk last van eventuele crashes
- Makkelijk voor de eindgebruiker
- Sterk besturingsmechanisme.
- Zo direct mogelijk met de hardware kunnen communiceren

Wat het systeem moet ook de volgende functionaliteit hebben:

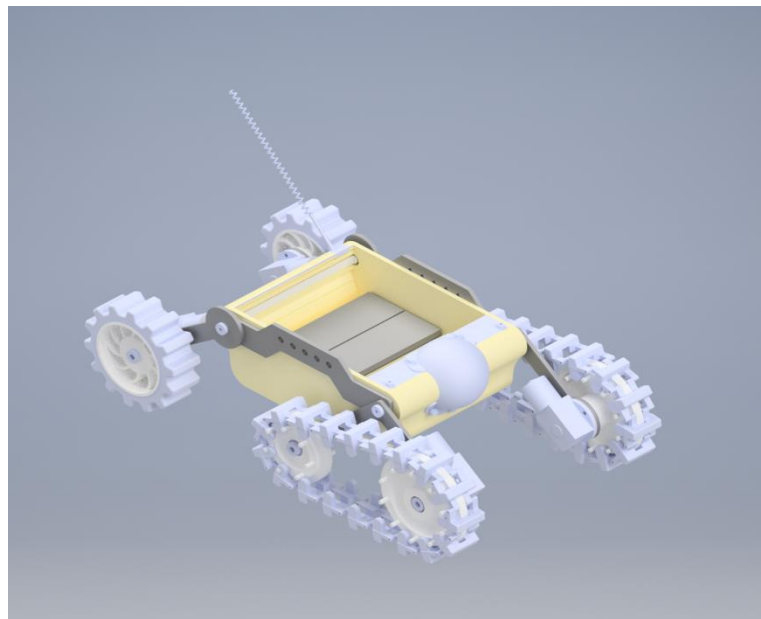
- Zelf kunnen navigeren in de ruimte
- Zelf het product kunnen verzamelen m.b.v. de hardware

Om zelf te kunnen navigeren kan er gebruik gemaakt worden van driehoek-navigatie. Hierbij worden minimaal drie radiozenders gebruikt om de locatie te bepalen van een wagentje. Deze navigatiemethode legt ook de basis van GPS. Een voordeel van dit systeem is dat het vrij goedkoop te bouwen is en er bij uitval van één radiobaken het systeem nog steeds functioneel kan zijn. Een nadeel is dat het niet altijd accuraat is met obstakels.

Een ander navigatiemiddel is 'dead-reckoning'. Hierbij voorspel je de locatie op basis van een beginpunt, de tijd en de snelheid. Een groot voordeel hiervan is dat je onafhankelijk bent van enige externe bron. Maar elke verandering in de koers of locatie, bijvoorbeeld door een stootje van een onderhoudsmonteur, zorgt er direct voor dat de navigatie niet meer werkt.

De driehoek-navigatie heeft hier dus de voorkeur, mits er sprake is van minimaal vier radiobakens waarbij de vierde als backup werkt.

Er zijn vele programmeertalen die we kunnen gebruiken, maar de taal die het beste kan communiceren met de hardware en niet te abstract is, is C. Maar C heeft een nadeel: hij kan geen classes gebruiken. Daarom is C++ de beste optie. Het is een extensie op C en kan wel classes opslaan.



Figuur OW.8: Een visualisatie van het wagentje

Conclusie: het antwoord op de vraag “Hoe kunnen we dit gebied automatiseren?” is: Door wagentjes te gebruiken die zelf producten kunnen verzamelen, en die via een lokaal navigatie systeem kunnen navigeren door de ruimte.

5: Bouwen

Nu het ontwerp klaar is kunnen we een prototype van het systeem gaan bouwen. In dit hoofdstuk zal het proces van het bouwen van het prototype worden laten zien en worden beschreven. Tevens zal de vraag hoe we het systeem gaan bouwen (deelvraag 3) worden beantwoord.

5.0 Bouwstappen

Het is uiteraard handig om te weten in welke stappen het systeem wordt gebouwd. Het ideale stappenplan is als volgt:



5.1 Pre-discussie

Wegens het ontbreken van onderdelen is er gekozen om de volgende onderdelen te schrappen of te vervangen:

- Lokale navigatiesysteem (vervangen door dead-reckoning)
- Grijparm (geschrapt)

Verder zal het prototype zich houden aan het ontwerp. Voor meer uitleg: Raadpleeg hoofdstuk 8.

5.2 Materialen

Het prototype gaat werken met verschillende onderdelen. **Deze materialen kunnen op deze manier worden opgedeeld:**

- ❖ 2x Wagentje
 - 2x DC Motor (12V)
 - 1x 12V stroombron (batterij)
 - 1x Raspberry Pi Zero W
 - 1x Atmel ATMEGA328P (Arduino Uno)
 - 1x Texas Instruments L293D Double H-Bridge IC
 - ❖ 1x Host
 - 1x een pc met internettoegang waarop Linux of Windows draait met .NET Core geïnstalleerd
- Qua software wordt de Arduino IDE gebruikt om de microcontroller te programmeren; Geany voor het programmeren van de software die de Raspberry Pi zal draaien en Visual Studio voor het programmeren van de software die de host draait. Alle programmering van de microcontroller en de Raspberry Pi zal in C++ worden gedaan. De software op de host wordt hoofdzakelijk in C# en Python geschreven.

De schematic van het prototype is simpel. Op de bladzijde daarna wordt de schematic uitgelegd.



| | |
|---------------------|--------------|
| TITLE: prototype | |
| Document Number: | REV: |
| Date: not saved! | Sheet: 1/1 |

We zien helemaal links van de schematic dat de elektriciteit van de voeding (met een spanning van $\geq 12V$) binnenkomt en door de eerste voltage regulator¹² gaat en daarna de naam **PP_12V0** krijgt. **PP** geeft aan dat deze lijn stroom levert; **12V** voor de spanning en **0** voor het lijnnummer. Dus stel: er zijn twee 12V groepen, dan zou de tweede lijn **PP_12V1** genoemd worden. Een deel van die stroom gaat naar een Voltage Regulator die de 12V omzet naar 5V. Die lijn heet **PP_5V0**. Vervolgens zien we als we naar rechts gaan de pinnen die op de Raspberry Pi zitten. De *SDA* en *SCL* pinnen worden gebruikt, die I²C mogelijk maken. Daar rechtsonder zien we de L293D IC¹³. Deze is nodig omdat de motoren op 12V werken, en de microcontroller slechts 5V kan uitzenden. Om toch de 12V te kunnen bereiken wordt deze IC gebruikt. Daarbij kan deze IC makkelijker van polariteit wisselen. Zo kan er makkelijk achteruit/vooruit worden gereden. Deze IC wordt ‘omringd’ door een dikke blauwe lijn. Dat is een bus¹⁴. Die bus zorgt ervoor dat we makkelijk alles kunnen aansluiten in de schematic. Het is niks meer dan een verzameling groene lijnen (aansluitingen). Deze groene lijnen hebben allemaal namen. **Deze namen zijn als volgt te beschrijven:**

- *M1_EN* zet motor 1 aan of uit
- *M2_EN* zet motor 2 aan of uit
- *M1_C1* en *M1_C2* worden gebruikt om motor 1 aan te sturen.
- *M2_C1* en *M2_C2* worden gebruikt om motor 2 aan te sturen.
- *M1+* en *M2+* zijn de + polen van de motoren
- *M1-* en *M2-* zijn de – polen van de motoren

Bovenin zien we een circuit met een knop (S1). Zodra de knop wordt ingedrukt, wordt de microcontroller gereset die tevens getriggered kan worden door GPIO4 van de Raspberry Pi. Helemaal rechts zien we dat de uitgangen van de microcontroller worden gelinkt aan de bus om vervolgens gebruikt te worden bij de L293D IC.

Wee

5.4 Bouwen (hardware)

Er waren twee wagentjes beschikbaar op school (zie figuur BW.1 en BW.2), dus die zijn gebruikt in dit prototype. Het eerste wat er is gedaan is het installeren van de Arduino en de L293D IC op een breadboard en die aansluiten. Zie figuur BW.3, BW.4, BW.5 en BW.6. Zoals gezegd: Het bouwen van het apparaat is niet zo bijzonder. Vandaar ook deze korte paragraaf.

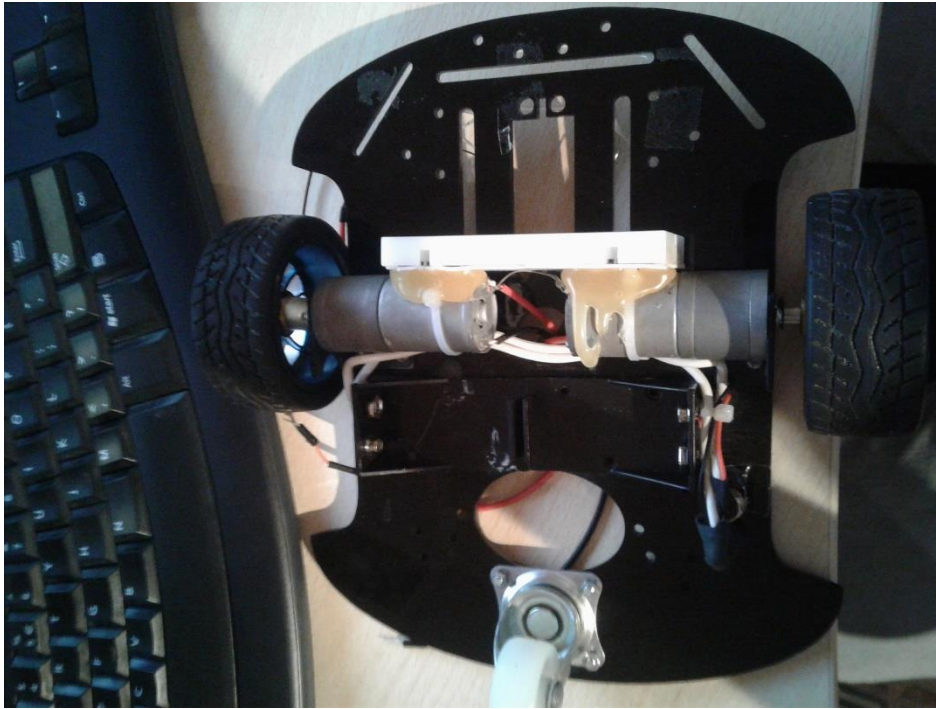
¹² Een voltage regulator is een systeem dat een bepaalde spanning stabiel uitstuurt, ongeacht de spanning die binnenkomt. Zo wordt voorkomen dat een apparaat te veel of te weinig spanning krijgt.

¹³ Deze IC bevat twee H-Bridge transistor circuits.

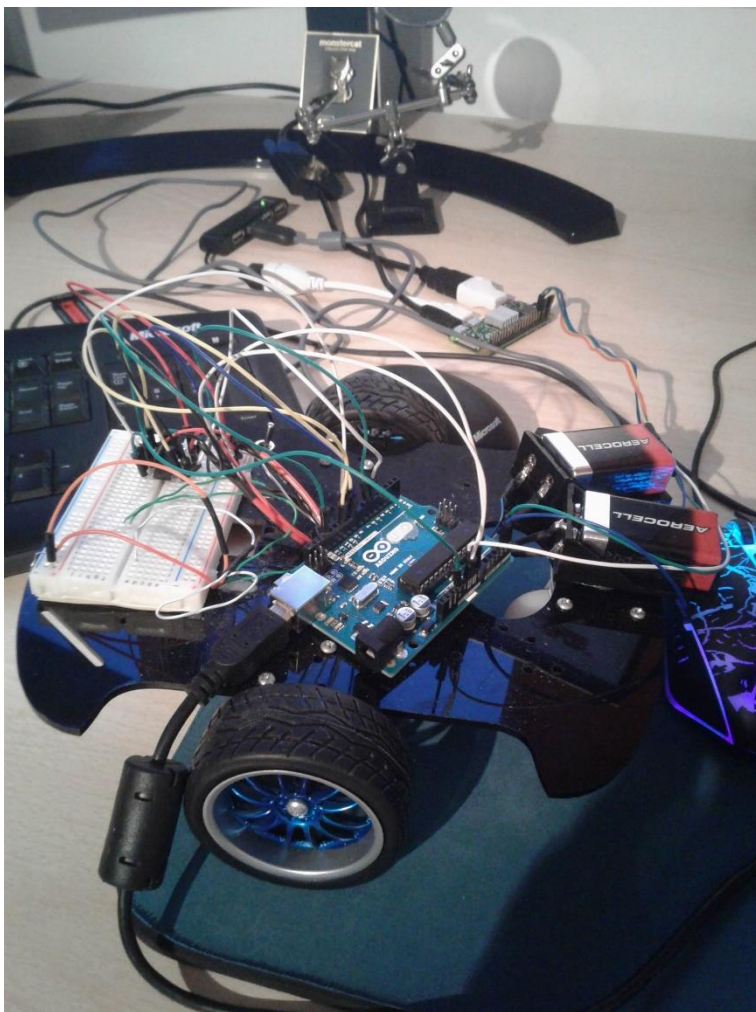
¹⁴ Een bus is een verzameling aansluitingen die tot één aansluiting worden gevormd, om vervolgens weer later tot meerdere aansluitingen te vormen.

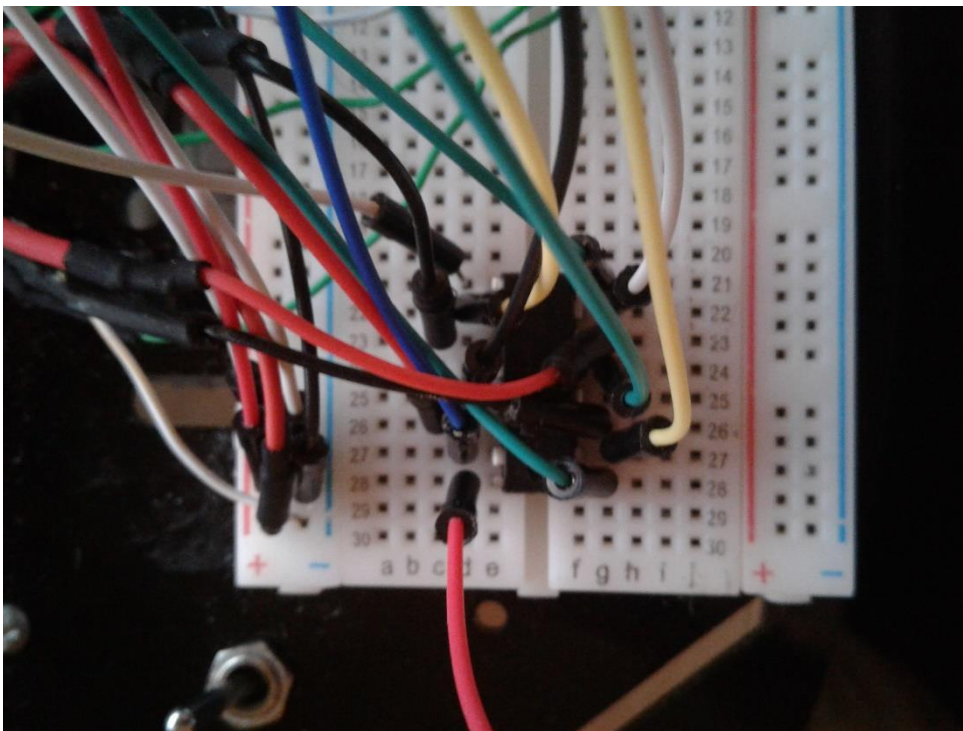
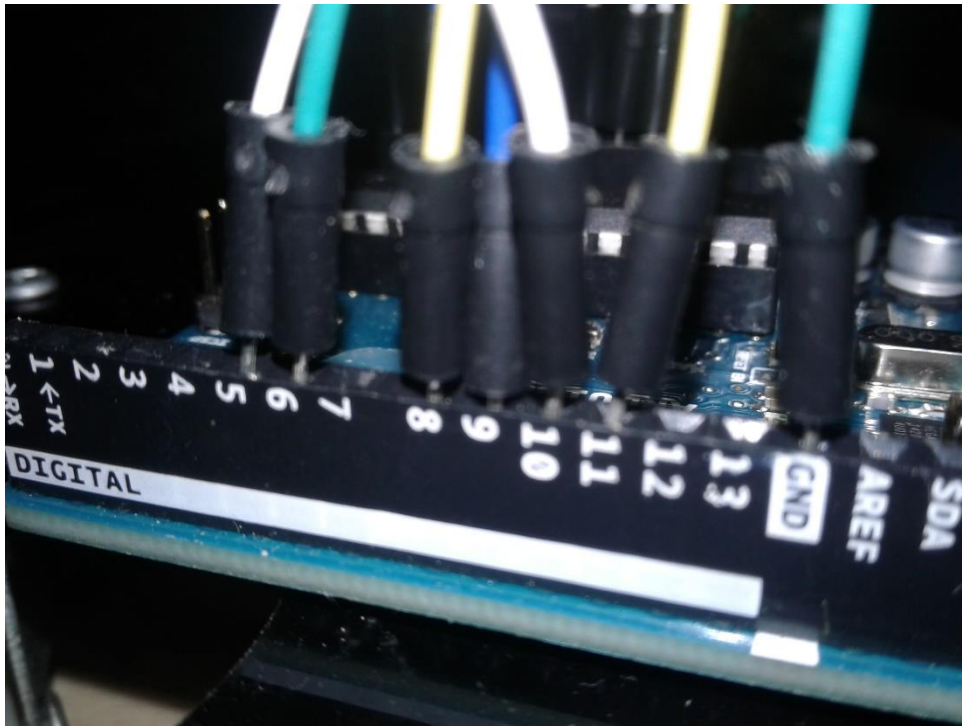


Figuur BW.1: "Barebone" wagentje. Bevat op deze foto alleen wielen, motoren en bedrading.



Figuur BW.2: Onderzijde van de wagen





Figuur BW.4: L293D IC aangesloten aan de motoren en de microcontroller



Figuur BW.6: De aansluitingen op de Raspberry Pi Zero. Let wel op dat de HDMI en de linker USB poort normaal niet aangesloten zijn. De rechter USB poort is voor de voeding, die uiteraard wel aangesloten is.

5.5 Bouwen (Software)

(Woesthuis, sd): Zie de repository in <https://github.com/hugopilot/pws>

5.5.1 Microcontroller

Zie *~/Wagentje (client)/Microcontroller* voor de volledige code in de repository.

De software voor de microcontroller heeft maar twee doelen:

1. I²C pakketten van de Raspberry kunnen decoderen
2. De twee motoren aan te kunnen sturen.

We beginnen bij deze code (BW.C0):

```
#include "Motor.h"

Motor m1;
Motor m2;

void setup() {
    // put your setup code here, to run once:
    m1.Init(M1_ENABLE, M1_CTRL_1, M1_CTRL_2);
    m2.Init(M2_ENABLE, M2_CTRL_1, M2_CTRL_2);
    Wire.begin(I2C_ADDRESS);
    Serial.begin(DEBUG_ADDRESS);
    Serial.println("SETUP OK");
    Wire.onReceive(command_receive);
}
```

*BW.C0; Naar: *~/Wagentje (Client)/Microcontroller/DistroLink_Motor.ino**

In de eerste drie regels zien we de motoren definiëren. Dit wordt gedaan door de `Motor` class¹⁵ te gebruiken. Hierdoor kunnen we makkelijk twee motoren besturen. Maar voordat we dat kunnen doen moeten we de class toevoegen aan onze code. Dit doen we d.m.v `#include "Motor.h"`. In `Motor.h` zitten alle variabelen voor de `Motor` class. Aangezien we twee motoren hebben noemen we de twee classes `m1` en `m2`.

Daarna zien we `void setup()`. Alle code hierin zal tijdens het opstarten worden uitgevoerd en alles instellen, vandaar de naam `setup`.

Eerst zien we `m1.Init(M1_ENABLE, M1_CTRL_1, M1_CTRL_2);`. Deze code geeft alle pinnummers door aan de class, zodat die alle signalen door kan sturen. Zoals te zien zijn er 3 pinnen per motor nodig.

Tussen de haakjes zien we `M1_ENABLE`, `M1_CTRL_1` en `M1_CTRL_2`. Deze variabelen zullen worden doorgegeven aan de class.

Een lijstje van wat elke variabele voorstelt:

¹⁵ Een class is een verzameling variabelen en uitvoerbare opdrachten in C++. (cppreference, sd)

- `M1_ENABLE` zet de motoren aan of uit. Als dit snel genoeg gebeurt (40kHz, zo'n 400.000 keer per seconde) kunnen we deze gebruiken om de snelheid te manipuleren.
- `M1_CTRL_1` is een besturingspin. Deze beïnvloedt de richting van de motor (vooruit/achteruit) en hangt af van `M1_CTRL_2`. Als bijvoorbeeld `M1_CTRL_1` aan staat en `M1_CTRL_2` uit staat, draait de motor vooruit.
- `M1_CTRL_2` is hetzelfde als `M1_CTRL_1`.

Daarna zien we `Wire.begin(I2C_ADDRESS);`. Deze initialiseert de I²C communicatie.

Aangezien de microcontroller een slave is, geven we direct het adres door. Dit adres zit in variabele `I2C_ADDRESS`. De twee volgende regels slaan we over: die zijn niet van belang. De laatste regel, `Wire.onReceive(command_receive);`, geeft aan dat als er een opdracht van de Raspberry Pi binnenkomt, dat de microcontroller dan `command_receive` moet uitvoeren. Daarna komen we in de originele code `void loop()` tegen. Maar aangezien dat die niks doet gaan we daar niet op in.

De `command_receive` opdracht, die te vinden is op de volgende bladzijde, ziet er complex uit terwijl die eigenlijk heel simpel is. Het enige wat die doet is 'in het geval dat ik dit nummer binnenkrijg, dan doe ik deze actie'. Dus zodra een nummer binnenkomt gaat de microcontroller kijken wat die moet doen. Bijvoorbeeld: Als nummer 4 binnenkomt gaat de microcontroller `case '4':` uitvoeren. Daarin staat `m1.change_dir(FORWARD);` en `m2.change_dir(FORWARD);` die aan beide classes zeggen dat de motoren naar voren moeten draaien. De volledige source is te vinden bij BW.C0.

Tabel BW.7 geeft een lijst van alle commando's die de microcontroller geeft.

| Binaire code (4-bit binary) | Nummer (decimal) | Actie |
|-----------------------------|------------------|---------------------------------|
| 0000 | 0 | Stop alle motoren |
| 0001 | 1 | Stel de snelheid maximaal in |
| 0010 | 2 | Ga naar links |
| 0011 | 3 | Ga naar rechts |
| 0100 | 4 | Ga vooruit |
| 0101 | 5 | Ga naar achteren |
| 0110 | 6 | Stel snelheid in |
| 0111 | 7 | Scherpe draai naar links |
| 1000 | 8 | Scherpe draai naar rechts |
| 1001 | 9 | Zet besturingmogelijkheden uit |
| 1010 | 10 | Zet besturingmogelijkheden aan. |

Tabel BW.7: Schematische weergave van het protocol en de acties die de microcontroller uitvoert.

```

void command_receive(int byteCount){
    while(Wire.available() > 0){
        char cmd = Wire.read();
        Serial.println(cmd);
        switch(cmd){
            case '0':
                m1.change_dir(STOP);
                m2.change_dir(STOP);
                break;
            case '1':
                m1.set_speed(FULL_SPEED);
                m2.set_speed(FULL_SPEED);
                break;
            case '2':
                m1.change_dir(STOP);
                m2.change_dir(FORWARD);
                break;
            case '3':
                m1.change_dir(FORWARD);
                m2.change_dir(STOP);
                break;
            case '4':
                m1.change_dir(FORWARD);
                m2.change_dir(FORWARD);
                Serial.println(m1.get_speed());
                break;
            // Gaat zo door
            default:
                continue;
        }
    }
}

```

5.5.2 Raspberry Pi

Zie ~/Wagentje (client)/Raspberry Pi voor de volledige code in de repository

De meeste code zit in de Raspberry Pi. De client (wagentje) is bij dit prototype voor het uitvoeren van de meeste taken verantwoordelijk. Er is bijvoorbeeld een lokaal bestand dat alle stappen bevat om naar de positie te rijden.

Om te beginnen moeten eerst de kernfuncties worden gebouwd. Denk hieraan aan een implementatie van het TCP protocol, I²C protocol en command-translators¹⁶.

Het I²C protocol hebben we in Python geschreven, aangezien we het niet op C++ aan de praat kregen. De gebruikte libraries werkte blijkbaar niet goed genoeg in C++. Python was een goede tweede keuze, omdat het snel te programmeren was en makkelijk te integreren in het C++ programma.

Om het in C++ te laten werken is `system()` gebruikt. (Zie *motorcontrol.cpp*, line 7 tot 9). Hetzelfde hebben we gedaan voor de reset functie. Met deze functie kan de Raspberry de microcontroller resetten als die bijvoorbeeld geen respons meer krijgt. Dat gaat via een shell script (zie *resetmicrocontroller.sh*) en in C++ wordt het uitgevoerd via de `system()` functie. (Zie *motorcontrol.cpp*, line 10 tot 12). Het TCP protocol is wel volledig in C++ geschreven. In dit verslag gaan we niet diep de code in van de kernfuncties aangezien die meestal abstracte code bevatten die niet 'in een paar zinnen' uit te leggen is. Daarbij werkt het niet direct mee aan de werking van het systeem. De code daarvan is te vinden in de map ~/Wagentje (client)/Raspberry Pi met de namen *motorcontrol.cpp* en *tcp_handler.cpp*

Zoals gezegd in het begin slaat de Raspberry zelf een bestand op met daarin alle stappen die een Raspberry moet nemen om bij een positie te komen. Om dit efficiënt te kunnen doen wordt gebruik van XML¹⁷ aangeraden. Met XML kan namelijk snel door programma's de benodigde informatie eruit worden gehaald en XML heeft een hele fijne structuur die goed werkt voor de beoogde doeleinden. Het XML bestand moet onderscheid kunnen maken tussen verschillende posities en modi.

Een voorbeeld van het XML bestand:

```
<Positions>
  <Position name="A1">
    <tdfb>
      <step>FWD,2000</step>
      <step>LFT,3000</step>
      <step>FWD,2000</step>
      <step>SLFT,1265</step>
      <step>FWD,5000</step>
      <step>STOP</step>
    </tdfb>
    <tdtb>
      <bstep>BCK,3000</bstep>
      <bstep>SLFT,1265</bstep>
      <bstep>FWD,2000</bstep>
      <bstep>RGHT,3000</bstep>
      <bstep>STOP</bstep>
    </tdtb>
  </Position>
</Positions>
```

¹⁶ Een command-translator zet een inkomend commando over naar een actie.

¹⁷ eXtensible Markup Language is een manier om gestructureerde data op te slaan. HTML is o.a. gebaseerd op XML. (Wikipedia, sd)

De logica hiervan is makkelijk te begrijpen.

We openen het bestand met `<Positions>` en is de 'root-node' in het bestand. Deze geeft aan dat het bestand vanaf hier opent.

Daarna openen we een `<Position name="A1">` node. Deze node geeft aan dat we een positie gaan definiëren en dat zijn naam "A1" is.

Daarin openen we eerst de `<tdfb>` node (tdfb staat voor: 'Time to Drive From Base'¹⁸). Met daarin veel `<step>` nodes. In deze nodes zit het volgende format `<step>COMMANDO, VERTRAGING</step>`.

Met COMMANDO worden dingen als "vooruit" of "achteruit" bedoeld. En met VERTRAGING wordt de vertraging (in milliseconden) tot de uitvoering van volgende opdracht bedoeld. Die vertragingen zijn bedoeld bijvoorbeeld te zeggen: 10 seconden vooruit en dan 3 seconden naar rechts draaien.

Vervolgens sluiten we TDFB node met `</tdfb>`. Daarna openen we een `<tdtb>` node (tdtb staat voor: 'Time to Drive To Base'¹⁹) met daarin `<bstep>` nodes. Deze nodes hebben hetzelfde format als `<step>`. Daarna sluiten de definitie van de positie met `</Position>` en daarna sluiten we het document met `</Positions>`.

Alle commando's worden in Tabel BW.8 uitgelegd.

In het programma wordt het Xml-bestand gelezen en opgeslagen in verschillende structuren:

```
struct step{
    command dir;
    long time;
};
struct position{
    // name of the position
    std::string name;
    // Time to Drive From Base
    std::vector<step> tdfb;
    // Time to Drive To Base
    std::vector<step> tdtb;
};
```

Tabel BW.8: Schematische weergave van alle commando's die mogelijk zijn.

| Commando | Actie |
|----------|-------------------|
| FWD | Vooruit |
| LFT | Linksaf |
| RGHT | Rechtsaf |
| SLFT | Draai naar links |
| SRGHT | Draai naar rechts |
| BCK | Achteruit |
| STOP | Stop met rijden |

8: Schematische weergave van alle commando's die in het XML bestand mogelijk zijn.

We zien twee `struct`'s met de namen 'step' en 'position'. In step worden het commando en de vertraging opgeslagen, net zoals in het XML document. In position worden de naam van de positie en de stappen opgeslagen. De stappen worden opgeslagen door een vector²⁰ van 'step' te maken (`std::vector<step>`).

Om het op deze manier op te slaan kunnen we makkelijk in de code de posities manipuleren of gebruiken.

De besturing zelf gaat volgens de volgende flow:

¹⁸ TDFB = De stappen die nodig zijn om vanaf het basispunt naar de beoogde positie te komen.

¹⁹ TDTB = De stappen die nodig zijn om vanaf een positie naar het basispunt te rijden.

²⁰ Een vector in C++ is een verzameling van een meerdere variabelen waarvan zijn lengte dynamisch kan worden bepaald. Dit is een dynamische array in C++. (cppreference, sd)



Figuur BW.9: De stappen (flow) die het pakken van producten vereist.

Die commando's die de server kan sturen zijn in Tabel BW.9 weergegeven:

| Commando | Acties |
|------------------|--|
| NOTG | Geeft aan dat er een nieuwe opdracht aankomt. Het volgende TCP pakket bevat de instructies |
| TESTALLPS | Geeft opdracht aan de client om alle posities na te gaan. Dit is om te testen of alle posities goed zijn geprogrammeerd. |
| FLREBOOT | Start het wagentje en de microcontroller opnieuw op |
| INITHDSK | Zet een verbinding met de server en het wagentje op. |
| POSBROAD | Geeft aan dat er een XML bestand aankomt met daarin (geüpdatete) posities. |

Tabel BW.9: De commando's die mogelijk zijn tussen de host en het wagentje.

In de code wordt dat op deze manier vertaald:

```
#include <iostream>

#include "host_handler.h"

// SEND ASCII
int main(int argc, char** argv){
    // Fill in the folder where the XML is located
    host_handler host("/home/pi/Desktop/pws");
    // Start the client
    host.start(argv[1], std::stoi(argv[2]));
}
```

Figuur BW.C4 De functie waar het programma mee start. Zie ~/Wagentje (client)/Raspberry Pi/main.cpp

De main() functie zorgt alleen ervoor dat host.start() wordt gestart.

In host.start() wordt vervolgens deze code uitgevoerd:

```
void host_handler::start(std::string IP, int port){
    // check if files are existent
    if(!check_file_exists(srv_path)){
        //serverSetup()
    }
    if(!check_file_exists(pos_path)){
        //positionSetup()
        return;
    }

    // Parse the XML file and convert into std::vector<position>
    std::cout << "Parsing POSITION file...";
    std::cout << pos_path << "...";
    tSleep(1000);
    std::vector<position> positions = ConvertXMLtoPositions(pos_path);
    std::cout << "done\n";
    MotorController::SendCommand(ENABLE, 4);

    // Connect to server
    tcp_client cl(IP, port);
    if(!cl.init()){
        return;
    }

    // Handshake flow
    cl.send_data("INITHDSK");
    std::string response = cl.receive_data(512);
    if(response == "HSKERROR"){std::cout<<"ERROR: Server returned handshake-error\n"; return;}
    std::cout << "Received " << response << " from server\n";

    // Start to listen

    while(true) { CommandParser(cl, positions); }
}
```

Figuur BW.C3: Een stukje code uit ~/Wagentje (client)/Raspberry Pi/host_handler.cpp

De code leest eerst het XML bestand en verbindt vervolgens met de host. Na de authenticatie wordt CommandParser(cl, positions); uitgevoerd.

```

void CommandParser(tcp_client tcpc, std::vector<position> posiss){
    bool busy = false;
    std::string d = tcpc.receive_data(512);
    if(d == "PAUSE")
        MotorController::SendCommand(STOP, I2C_ADDRESS);
    else if(d == "NOTG"){
        if(busy){
            tcpc.send_data("ERRBSY");
            return;
        }
        busy = true;
        // Give the OK command;
        tcpc.send_data("OK");
        // recieve the next packet
        std::string pd = tcpc.receive_data(1024);

        // Split the recieved string
        std::vector<std::string> posis = SplitByComma(pd);

        // Search and find all positions
        std::vector<position> exec = FindAllPositionsByName(posis, posiss);

        // Loop though all positions and execute them
        for(uint16_t d = 0; d < exec.size(); d++){
            position ps = exec.at(d);
            executemovements(ps, true, I2C_ADDRESS);
            tSleep(5000);
            executemovements(ps, false, I2C_ADDRESS);
        }

        // Client is not performing a task anymore.
        busy = false;
        // Send the OK response
        tcpc.send_data("OK");
    }
    else if (d == "TESTALLPS"){
        if(busy){
            tcpc.send_data("ERRBSY");
            return;
        }
        busy = true;
        TestAll(posiss);
        busy = false;
        tcpc.send_data("OK");
    }
}

```

Figuur BW.C4: Een stukje code uit ~/Wagentje (client)/Raspberry Pi/host_handler.cpp

CommandParser kijkt welk commando binnen is gekomen en voert de acties uit.

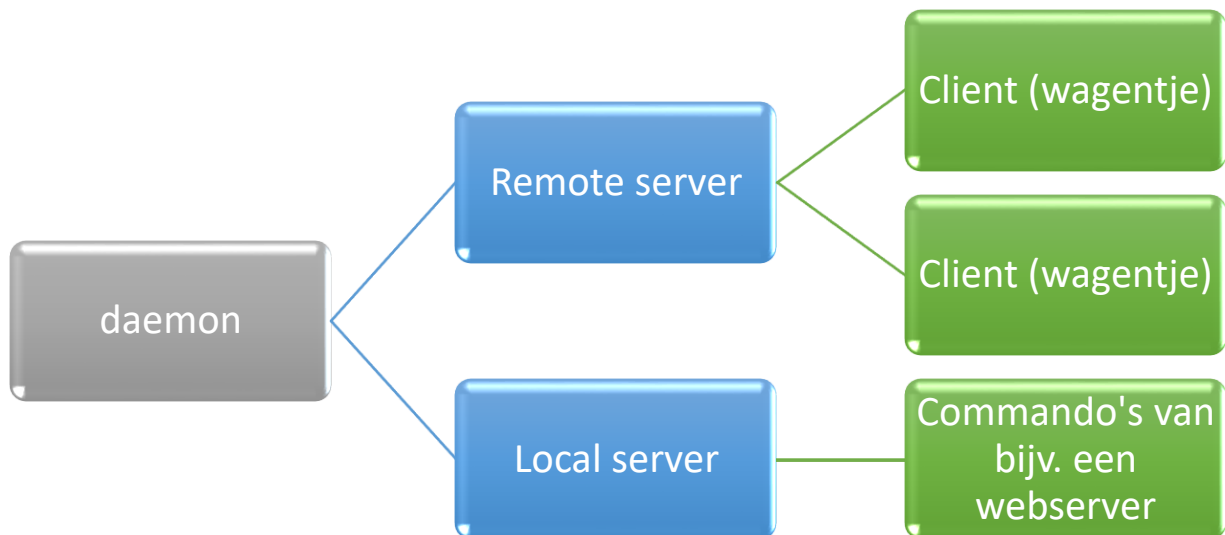
5.5.3 Host

De host bestaat uit een daemon²¹ die runt op een server.

De daemon draait altijd en is het hart van de host. De daemon krijgt commando's van bijv. de webserver en de daemon geeft dan commando's aan de Raspberry Pi.

De daemon werkt als twee TCP servers. De één om de clients aan te sturen, de ander om commando's binnen te krijgen.

Dat werkt als volgt (Figuur BW.10):



Figuur BW.10: De verschillende onderdelen van het TCP protocol die de daemon faciliteert.

Een belangrijke noot is dat de local-server maar één connectie accepteert. Dit is uit veiligheidsoverwegingen. Als de webserver in het begin de connectie heeft gemaakt, kan niemand anders er meer tussenkomen.

²¹ **Daemon** is de term die binnen [Unix](#) en Unix-achtige [besturingssystemen](#) (zoals [Linux](#) en [FreeBSD](#)) gebruikt wordt om een [proces](#) aan te duiden dat op de achtergrond actief is om bepaalde (onderhouds)taken uit te voeren of om diensten te verlenen aan andere computerprogramma's. In [Windows](#) worden zulke achtergrondprogramma's meestal *services* genoemd. (Wikipedia, sd)

```

public void StartService()
{
    // Start the service and clear the clients list.
    dHCPclients = new List<DHCPClient>();
    remsrv.Start();
    locsrv.Start();
    var acceptingThread = new Thread(() => AcceptConnections());
    var connectionManager = new Thread(() => ConnectionManager());
    var cmdService = new Thread(() => CommandService());
    var garbageService = new Thread(() => GarbageCollection());
    acceptingThread.Start();
    connectionManager.Start();
    cmdService.Start();
    garbageService.Start();
    DEBUG("TCP services started!");
}

```

Figuur BW.C5: De StartService functie. Zie ~/Server (host)/daemon/daemon/Handlers/TcpClientHandler.cs

In de code (Figuur BW.C5) worden die blokken letterlijk als aparte processen gezien. Threads⁸ heten die in de computerwereld.

Er is te zien dat `remsrv.Start()`; en `locsrv.Start()`; worden aangeroepen. Deze starten de TCP servers.

Ook is te zien dat er vier threads worden aangemaakt: `acceptingThread`, `connectionManager`, `cmdService` en `garbageService`.

De functies van deze threads zijn als volgt:

- `acceptingThread` : Accepteert connecties van clients.
- `connectionManager` : Kijkt of er clients zijn die geen connectie meer hebben.
- `cmdService` : Zet de commando's van de lokale server om naar commando's voor de clients, zoekt een beschikbare client uit en verzend vervolgens dat commando aan de client.
- `garbageService` : Zorgt ervoor dat we niet na een uur 1GB aan RAM-geheugen nodig hebben. Alle variabelen die niet worden gebruikt worden van het geheugen verwijderd. Hiermee hebben we ongeveer 1 MB per client aan RAM-geheugen nodig.

`dHCPclients = new List<DHCPClient>();` is een lijst van de class `DHCPClient`. Omdat deze door meerdere threads wordt bewerkt en gelezen, is de lijst `volatile`. Dit betekent dat meerdere threads de lijst mogen lezen en schrijven. Hierdoor worden crashes voorkomen.

De daemon gebruikt de volgende flow voor zijn werkwijze (BW.11):



Figuur BW.11: De typische flow voor de daemon.

Zie de code op de volgende bladzijde (Figuur BW.C6).

Via `localclient.Receive(cmdBuffer);` krijgt de daemon commando's binnen vanuit de local server (stap 1). Zodra die binnen komt wordt er gekeken welk commando bedoeld is met `switch (Scmd)`. Daarna zoekt de daemon een beschikbare client (stap 2).

Een client is beschikbaar onder de volgende voorwaarden:

1. De client is binnen een bepaalde tijd (3 seconden) te bereiken
2. De client is niet volgens de server al bezig (`bool DHCPClient._isBusy`).
3. De client geeft bij het geven van de opdracht geen foutmelding terug (`if (Encoding.ASCII.GetString(tB) == "ERRBSY")`)

Daarna verzendt de daemon het commando naar de client (stap 3) met `c._socket.Send(Encoding.ASCII.GetBytes(Rcmd[1] + "\0"))`


```

private void CommandService()
{
    while (true)
    {
        // One socket is accepted per command
        Socket localclient = locsrv.AcceptSocket();
        localclient.ReceiveTimeout = 10000;
        DEBUG("A local client has connected!");
        // Wait for a command to come in
        StringBuilder SB = new StringBuilder();
        byte[] cmdBuffer = new byte[1024];
        try { localclient.Receive(cmdBuffer); } catch (SocketException) { DEBUG("Exception
thrown: Local client probably timed out"); }
        SB.Append(Encoding.ASCII.GetChars(cmdBuffer));

        // Split the command first into sections
        string[] Rcmd = SB.ToString().Split('(', ')');
        string Scmd = Rcmd[0];
        switch (Scmd)
        {
            case "NOTG":
                bool used = false; // This states if the current command has been sent to a
client.

                do
                {
                    Thread.Sleep(10); // Wait 10 ms because of refreshing.

                    foreach (DHCPClient c in DHCPClients) // Loop though all clients
                    {
                        if (c._isBusy) { continue; } // If it's busy, skip to the next one
                        c._socket.Send(Encoding.ASCII.GetBytes("NOTG\0")); // Send the NOTG
command.

                        DEBUG("NOTG sended");
                        byte[] tB = new byte[8]; // Tiny buffer
                        c._socket.ReceiveTimeout = 3000; // Give the client 3s to react
                        try { c._socket.Receive(tB); } catch (SocketException) { /*Nothing*/
}

                        if (Encoding.ASCII.GetString(tB) == "ERRBSY") { DEBUG("Client is
busy"); c._isBusy = true; continue; } // If the clients states it's busy, update and skip
                        c._socket.Send(Encoding.ASCII.GetBytes(Rcmd[1] + "\0")); // Send
positions to client

                        c._isBusy = true; // Update busy status
                        DEBUG(string.Format("Command to {0} was sent",
c.ip.Address.ToString()));
                        var ho = new Thread(() => BusyClient(c)); // Assign a new thread
that keeps track of the status of the client
                        ho.Start();
                        used = true; // Command has been used
                        break; // Break the loop
                    }
                } while (!used);
                break;
            // Gaat zo door...
        }
    }
}

```

Figuur BW.6: Een stukje uit code van de CommandService() functie. Zie ~/Server (host)/daemon/Handlers/TcpClientHandler.cs

Deze functie blijft zich zolang de thread bestaat zich oneindig herhalen d.m.v. `while (true) { } .`

5.6 Interactie met de daemon en implementatie van het product

De software is nu uitgebreid uitgelegd, maar de bedoeling is dat er (correct) gebruik van gemaakt wordt.

Dit is het protocol waar de implementatie voor nodig is door anderen.

Bijvoorbeeld: Als een persoon een webshop runt met een webserver en met het protocol wil integreren is deze sectie extra interessant.

Het programma werkt met .NET Core²². Dit geeft de mogelijkheid om de daemon op alle soorten besturingssystemen (Linux, macOS, FreeBSD, Windows) uit te voeren.

Dit betekent wel dat .NET Core Runtime (versie > 2.1) geïnstalleerd moet zijn op de server.

Om het programma te starten dient men naar de map met het programma te gaan en daar een terminal te starten. Het programma start (met alle standaardinstellingen) met `dotnet daemon.dll`.

De interactie met de daemon is simpel.

De daemon ontvangt zijn commando's via localhost (127.0.0.1) en via poort **8383**. De poort kan gewijzigd worden met de opstart-argumenten²³. Dit gaat met het `--localport` argument.

Dus bijvoorbeeld: `dotnet daemon.dll --localport 2001`. De daemon zal dan de commando's via poort 2001 ontvangen.

De daemon zal standaard zijn clients aansturen op poort **2121**. Ook dit kan veranderd worden met `--remoteport`.

Dus bijvoorbeeld: `dotnet daemon.dll --remoteport 2000 --localport 2001`

De daemon zal dan zijn clients aansturen op port 2000 en zijn commando's op port 2001 ontvangen.

Bij het verzenden van commando's naar de lokale server wordt het volgende format gebruikt: `COMMANDO(ARGS)`. Het kan zijn dat een commando geen argumenten (ARGS) bevat. Dan hoeft er niks tussen de haakjes. **Belangrijk: De haakjes moeten altijd mee verzonden worden, zelfs als er geen argumenten worden meegezonden! Dus: `COMMANDO()`.**

In Tabel BW.12 staan de commando's die verzonden kunnen worden.

| Commando | Argumenten | Actie |
|-------------------|--|---|
| NOTG | Posities (scheiden met ','). Dus NOTG(A1,A5,B2). Geen spaties! | Pakt de producten bij de aangegeven posities. |
| TESTALLPS | ID van client | Test alle verschillende posities van de client |
| FLREBOOT | ID van client | Start de client opnieuw op. Zowel Raspberry Pi als de microcontroller worden gereset. |
| LISTDEVICE | Geen | Geeft een lijst terug met alle clients met hun ID, IP-Adres en poort. Dit is in JSON formaat. |

²² Te downloaden via <https://dotnet.microsoft.com/download>

²³ Een opstartargument is extra informatie die het desbetreffende programma gebruikt bij het opstarten. (Wikipedia)

In volgende versies van het systeem zouden meer commando's gebouwd worden. Dit kunnen bijvoorbeeld commando's zijn die diagnostische informatie terug geven.

Het aansluiten van het systeem op het internet (WAN) wordt STERK AFGERADEN. Dit is vanwege het feit dat het systeem op dit moment geen enkele vorm van authenticatie bevat en dus makkelijk te hacken is. Hou dit systeem op het LAN en zorg er voor dat er niet makkelijk toegang is op dat netwerk.

Enkele tips om cyberaanvallen te voorkomen:

1. Zet een volledig afgesloten netwerk op.
2. Zet op de wifi een sterke key met WPA2 of WPA3. Het liefst random gegenereerd.
3. Zet SSID-broadcasting uit en geef deze een niet-voor-de-hand-liggende naam. Het liefst een random naam. Hiermee voorkom je dat hackers het netwerk überhaupt kunnen zien.

Ook de server zelf moet enige vereisten hebben.

De systeemvereisten zijn als volgt:

- Een processor met minimaal 2 cores met een kloksnelheid van minimaal 2,5 GHz
- Een processor met 4 cores en een kloksnelheid van 3,6 GHz wordt aangeraden.
- Minimaal 2 GB RAM
- 2 Ethernet devices die elk een ander netwerk kunnen bedienen. Hierdoor kan zowel een webserver als de daemon worden gedraaid en worden de netwerken geïsoleerd van elkaar.
- Minimaal 3 GB schijfruimte (voor .NET Core Runtime, programma zelf en cache)

De client zou normaal zelf met het programma moeten opstarten. Maar bij de eerste keer moeten er eerst wat dingen ingesteld worden. Dit is het één keer handmatig opstarten van de client. Om dat te doen moet ook hier eerst een terminal worden geopend. Er van uitgaande dat de terminal in de home folder start dient eerst dit commando ingevoerd te worden: `cd "Desktop/pws/Wagentje (client)/Raspberry Pi"`. Om het programma te starten dient `./PiDistro <IP van host> <poort>` ingevuld te worden. In het geval dat het bestand niet bestaat moet u eerst het programma compileren. Dit doet u door `g++ -Wall -o "PiDistro" "main.cpp" "tcp_handler.cpp" "motorcontrol.cpp" "host_handler.cpp"` uit te voeren.

Is het nog te volgen?

Om hierbij deelvraag 3 te beantwoorden: Het systeem werkt naar behoren zoals beoogde resultaat. Wat niet bereikt is, is het implementeren van het driehoeks-navigatie systeem en het grijparmpje. Het systeem werkt met het host-client concept. Dat concept wordt in de code gehanteerd. De code van de Raspberry kan het wagentje besturen en tegelijkertijd met de host communiceren. De code van de host kan orders van bijvoorbeeld een webshop binnenkrijgen en vervolgens omzetten naar een opdracht voor een client.

6: Resultaten

Onze bevindingen zijn dat het prototype naar behoren werkt.

Er zijn voor het prototype 7021 lijnen aan code geschreven in 4 verschillende programmeertalen.

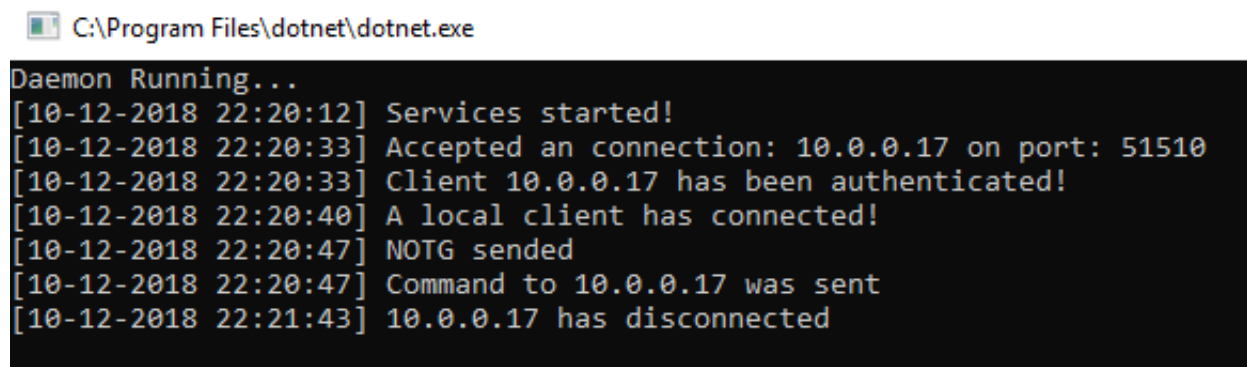
Onze tests gebruikten alle elementen die zijn behandeld in hoofdstuk 5:

- 2x Wagentje
- Een server
- Een gekalibreerd XML bestand

Na veel verschillende scenario's te hebben geoefend en het systeem een beetje te hebben 'gepest' is geen enkel programma gecrasht. Dit duidt dus aan dat het systeem stabiel is.

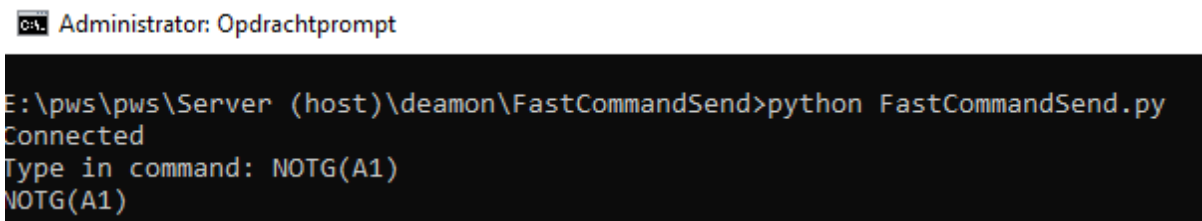
Ondanks het feit dat enorm veel gelukt is, hebben we maar eigenlijk 3 van de 4 stappen kunnen voltooien. De enige stuk waar wel op geanticipeerd is, maar niet is volbracht is het kunnen pakken van producten. Hier zal in hoofdstuk 8 over gepraat.

In figuur RE.1 en RE.3 is de output van de terminals te zien. In Figuur RE.2 sturen we handmatig het NOTG(A1) commando. Dit zou uiteraard automatisch kunnen gaan via een webserver. Hiermee laten we zien dat de communicatie en protocollen werken.



```
C:\Program Files\dotnet\dotnet.exe
Daemon Running...
[10-12-2018 22:20:12] Services started!
[10-12-2018 22:20:33] Accepted an connection: 10.0.0.17 on port: 51510
[10-12-2018 22:20:33] Client 10.0.0.17 has been authenticated!
[10-12-2018 22:20:40] A local client has connected!
[10-12-2018 22:20:47] NOTG sended
[10-12-2018 22:20:47] Command to 10.0.0.17 was sent
[10-12-2018 22:21:43] 10.0.0.17 has disconnected
```

Figuur RE.1: De output van de daemon bij de test. Hierbij zien we dat er een client verbindt en dat even daarna het commando, weliswaar handmatig, NOTG(A1) wordt verstuurd. Na de executie hebben we de client de verbinding laten verbreken met de host. Hierbij zien we dat de host dat merkt en hier rekening mee houdt.



```
C:\> Administrator: Opdrachtprompt
E:\pws\pws\Server (host)\daemon\FastCommandSend>python FastCommandSend.py
Connected
Type in command: NOTG(A1)
NOTG(A1)
```

Figuur RE.2: Een simpel python scriptje waarbij we handmatig commando's kunnen sturen naar de daemon. In dit geval hebben we NOTG(A1) ingevuld en verstuurd. Zie beschrijving RE.1 voor meer info.

```
pi@raspberrypi: ~/Desktop/pws/Wagentje (client)/Raspberry Pi
Bestand  Bewerken  Tabbladen  Hulp
^C
pi@raspberrypi:~/Desktop/pws/Wagentje (client)/Raspberry Pi $ ./PiDistro 10.0.0.
117 2121
Parsing POSITION file.../home/pi/Desktop/pws/positions.xml...done
Connecting...success
Sending data...sended!
Received INITHSOK from server
Sending data...sended!
Going from base to A1
Sending 4
Sending 2
Sending 4
Sending 6
Sending 4
Sending 0
Going to base from A1
Sending 5
Sending 6
Sending 4
Sending 3
Sending 0
Sending data...sended!
^C
pi@raspberrypi:~/Desktop/pws/Wagentje (client)/Raspberry Pi $
```

Figuur RE.3: De output van de Raspberry Pi. Hier zien we dat het commando binnen komt en dat de Raspberry deze uitvoert. Na deze uitvoering hebben we handmatig het programma gestopt (zie de ^C helemaal onderin. Deze duid de toetsencombinatie CTRL+C aan, die het programma aangeeft te stoppen). Hierdoor werd de verbinding tussen de host en de client verbroken.

7: Conclusie

We vroegen ons af hoe wij automatisering konden toepassen in de distributiecentra. Vandaar ook de vraag: *Hoe kan een webshop-order zo snel mogelijk verwerkt kan worden na binnenkomst met behulp van automatisering?*

Om als eerst nog samen te vatten wat de antwoorden waren op de deelvragen:

1. Het distributieproces wordt vooral handmatig gedaan, en het dunder uitkomt vergeleken met een volledig geautomatiseerd proces. Dit kunnen wij concluderen uit het voorafgaand onderzoek dat we hebben gedaan. Zie hoofdstuk 3
2. Door wagentjes te gebruiken die zelf producten kunnen verzamelen en zelf kunnen navigeren door een ruimte heb je een parallel systeem dat zo min mogelijk zwakke punten heeft. Hierdoor kan het systeem vrijwel altijd volledig opereren, zelfs als één wagentje uitvalt. Zie hoofdstuk 4
3. Het systeem werkt naar behoren zoals beoogde resultaat. Wat niet bereikt is, is het implementeren van het driehoeks-navigatie systeem en het grijparmpje. Het systeem werkt met het host-client concept. Dat concept wordt in de code gehanteerd. De code van de Raspberry kan het wagentje besturen en tegelijkertijd met de host communiceren. De code van de host kan orders van bijvoorbeeld een webshop binnenkrijgen en vervolgens omzetten naar een opdracht voor een client. Zie hoofdstuk 5

Om dan om uiteindelijk als laatste de hoofdvraag te beantwoorden:

Het snel verwerken van een order zit hem allemaal in welke stappen het meeste tijd kosten. Hierbij is in hoofdstuk 3 geconstateerd dat vooral het order picken het meest langzaam gaat. Dit gaat vrijwel altijd handmatig. Hier zou automatisering perfect werken. In vergelijking met mensen is een automatisch systeem qua verwerking veel sneller.

In hoofdstuk 4 werd er geconcludeerd dat wagentjes het beste werken voor een dergelijk automatisch systeem.

Dus hoe verwerk je zo snel mogelijk een webshop order door middel van automatisering?

TL;DR: door een automatisch systeem te bouwen kun je orders veel sneller en goedkoper laten verwerken.

Aangezien de vertragende factoren mensen in dit geval zijn zou automatisering veel te bieden hebben. Daarnaast is het goedkoper dan menselijke werkers. Door een goed systeem te ontwerpen en te bouwen is het erg stabiel en is er ruimte voor bepaalde componenten om te falen, zonder dat dit het hele systeem beïnvloed.

Volgens de resultaten werken de protocollen zoals geanticipeerd. Het systeem werkt. Maar in dit prototype is niet de grijparm geïnccludeerd. Dit is dus een onderdeel voor eventueel vervolgonderzoek.

We kunnen per stap, zoals beschreven in hoofdstuk 5.0 concluderen:

- **Wagentje laten rijden:** Hier is het alleen nog essentieel dat alleen de Raspberry Pi goed kan communiceren met de microcontroller, om er voor te zorgen dat ie kan rijden.
 - Systeem reageert als verwacht. Bij het verzenden van commando's met I²C reageert de microcontroller zoals verwacht. Werkt volledig.
- **Wagentje te kunnen laten rijden naar een positie d.m.v. driehoeks-navigatie:** Bij deze stap wordt het protocol tussen de host en de client gerealiseerd.
 - TCP communicatie werkt naar behoren. Systeem reageert goed op commando's
 - Xml-bestand is leesbaar voor de client en kan gebruikt worden.
 - Driehoeks navigatie niet geïmplementeerd, vervangen door dead-reckoning.

Doordat dit prototype al heel goed werkt en er makkelijk ruimte is voor uitbreiding kunnen we stellen dat automatisering dus een zeer goede optie is.

Door dit systeem te koppelen aan de webserver, of een andere server die verantwoordelijk is voor het doorgeven van orders, kunnen orders vrijwel direct worden verwerkt door de wagentjes die rondrijden in het distributiecentrum zonder enige tussenkomst van mensen.

8: Discussie

Om direct terug te komen op hoofdstuk 5. Wij konden niet snel beginnen met het bouwen van het systeem. Dit kwam omdat wij de onderdelen niet snel genoeg konden binnenhalen. Hierdoor ontstond er zulke vertraging dat wij niet meer in staat waren om zowel het navigatiesysteem als het grijp-arm-systeem op tijd af te kunnen maken. Vandaar hebben wij gekozen deze niet in dit prototype te includeren.

Omdat wij zelf het systeem volledig zelf hebben gebouwd en er geen bronnen te vinden zijn op het internet zijn sommige eisen die gesteld zijn in hoofdstukken 4 en 5 zelf bedacht. De totale lijst aan eisen is daarom een punt van discussie. Ook is het zo dat als er eisen zijn die niet zijn uitgewerkt een klein effect kunnen hebben op de resultaten.

Ook is het zo dat we al van tevoren van het onderzoek was gestart al een concept hebben bedacht, concept 1, maar dat moesten we eigenlijk niet doen want dan hadden we misschien andere concepten gehad.

iets anders wat de resultaten kon veranderen is dat we een groter onderzoek hadden moeten doen dat kwam omdat we nu alleen maar filmpjes van YouTube hebben gekeken en dat was ons onderzoek we hadden eigenlijk naar de distributie centrums zelf heen gemoeten want toen konden we meer variabelen mee nemen naar de resultaten.

Als laatst hadden we eigenlijk naar meer concepten moeten kijken want nu hebben we 2 concepten bedacht en als we meer concepten hadden bedacht dan hadden we misschien een betere concept bedacht.

Onderwerpen voor vervolgonderzoek

1. Implementatie van de grijp-arm. Hierbij moet je denken aan vragen als: Hoe kan de grijp-arm de producten pakken, zelfs als de groottes verschillen.
2. Implementatie van het navigatiesysteem d.m.v driehoeks-navigatie
3. Praktisch onderzoek: Hoe goed werkt dit systeem in de praktijk?

9: Bibliografie

Coolblue. (2017, december 6). Nieuw apparaat van 200 meter lang. nvt, nvt, nvt: YouTube.
Opgehaald van https://www.youtube.com/watch?v=QKSQg4_15PI

cppreference. (sd). *Class declaration*. Opgehaald van cppreference.com:
<https://en.cppreference.com/w/cpp/language/class>

cppreference. (sd). *std::vector*. Opgehaald van cppreference.com:
<https://en.cppreference.com/w/cpp/container/vector>

Raspberry Pi. (sd). *Raspberry Pi | Home*. Opgehaald van Raspberry Pi Foundation:
<https://raspberrypi.org>

RTL Nieuws. (2016, december 19). Van magazijn tot voordeur: zo komt een pakketje naar je toe. nvt, nvt, nvt: YouTube. Opgehaald van <https://www.youtube.com/watch?v=dSdwBtgGB7s>

Werken bij Coolblue. (nvt, nvt nvt). *Orderpicker Tilbug (40u)*. Opgehaald van Werken bij Coolblue:
<https://www.werkenbijcoolblue.nl/vacatures/vacature-orderpicker-tilburg-40u>

Wikipedia. (sd). *Command-line interface -> Arguments*. Opgehaald van Wikipedia:
https://en.wikipedia.org/wiki/Command-line_interface#Arguments

Wikipedia. (sd). *Computer Architecture*. Opgehaald van Wikipedia:
https://en.wikipedia.org/wiki/Computer_architecture

Wikipedia. (sd). *Daemon (Unix)*. Opgehaald van Wikipedia:
[https://nl.wikipedia.org/wiki/Daemon_\(Unix\)](https://nl.wikipedia.org/wiki/Daemon_(Unix))

Wikipedia. (sd). *Extensible Markup Language*. Opgehaald van Wikipedia:
https://nl.wikipedia.org/wiki/Extensible_Markup_Language

Wikipedia. (sd). *Microcontroller*. Opgehaald van Wikipedia:
<https://nl.wikipedia.org/wiki/Microcontroller>

Wikipedia. (sd). *Thread (computing)*. Opgehaald van Wikipedia: Naar:
[en.wikipedia.org/wiki/Thread_\(computing\)](https://en.wikipedia.org/wiki/Thread_(computing))

Woesthuis, H. (sd). *pws*. Opgehaald van GitHub: <https://github.com/hugopilot/pws>

10: Evaluatie

10.1 Evaluatie Mike

We zijn te vroeg begonnen met te ontwerpen van 3D-modellen. Dat heeft ons veel onnodige tijd gekost omdat we hebben het weg gegoooid. Dat hebben we gedaan omdat we van concept zijn veranderd.

In de weken voor dat het moest worden ingeleverd worden had ik minder tijd dan normaal. Dat kwam dankzij privé redenen. Maar vanaf de laatste vrijdag voor dat het moesten inleveren had ik weer tijd waar door ik mijn laatste uren kon gaan maken. Blijkbaar hadden we meer tijd voor het verslag nodig dan gepland. Dat kun je zien aan de ongeveer 10 uur wat we boven de 80 uur zitten.

Het kostte zoveel tijd omdat er geen, en als het er was heel slecht, communicatie. Dat kwam omdat we niet op de zelfde momenten aan het verslag konden werken. Dat kwam door het privé leven van mij.

Ook is het zo dat er op de laatste dag ook problemen waren met het syncen van de documenten. Ik denk zelf dat het probleem was omdat mijn eigen server die ook alles synct met OneDrive. En mijn eigen server had wat mankementen de laatste tijd. Maar we waren wel een oplossing gevonden en dat was dat we het documenten op Hugo's share gedropt en vanuit daar gewerkt. Maar dat kostte wel zo'n 3 a 4 uur om het te oplossen.

De laatste negatieve verhaal dat ik ga vertellen is dat omdat onze profielwerkstuk begeleider alleen maar dingen zei tegen een van ons. En met onze slechten communicatie die we toen hadden. Kwam het bericht of opmerking slecht of zelfs niet over. Dat kostte voor veel tijd en ook problemen omdat het misverstanden opleverde.

Maar buiten dat hebben we wel een leuke tijd gehad. We hadden veel negatieve momenten gehad. Maar meer positieve momenten. Wij hadden er veel van geleerd en ook veel ervaring op gedaan buiten dit ontwerp. En dat is belangrijk.

Dus als conclusie is het was heel leuk om dit Profiel Werkstuk te doen. En ik neem de dingen die we hiervan hebben geleerd mee de rest van mijn leven in.

10.2 Evaluatie Hugo

Planning

Mike en ik hadden in het begin geen goede afspraken gemaakt. We zijn in het begin al gaan ontwerpen, maar dat konden we allemaal weggooien omdat het niet meer goed uitpakte voor ons onderzoek. In de laatste maanden hadden ik veel stress omdat Mike niet veel tijd had en ik heel veel zelf heb moeten doen. Ook dit is een resultaat van slechte afspraken. Een leermoment voor mij. Wij gingen uit dat de deadline op 16 december was, omdat dit in de jaarplanning stond. Een paar weken voor de deadline kwam ik erachter dat de deadline op 10 december was. Hierdoor zat ik zo'n beetje elke dag constant te piekeren. Zelfs op de 10^e zelf hebben we tot de laatste seconde gewerkt aan het verslag.

Samenwerking

Als ik terugkijk vindt ik de samenwerking matig. Mike en ik kunnen het goed met elkaar vinden, maar er zijn momenten geweest waarbij ik niet meer wist wat ik met Mike moest. Grootste gedeelte daarvan kwam van zijn planning en de andere prioriteiten van hem (o.a theorie-examen rijlessen).

In the end hebben we er samen ervoor gezorgd dat alles op zijn plaats staat

Verbeterpunten

Betere afspraken en planningen vanaf het begin maken met elkaar en rekening houden met andere activiteiten (examens, etc , etc).

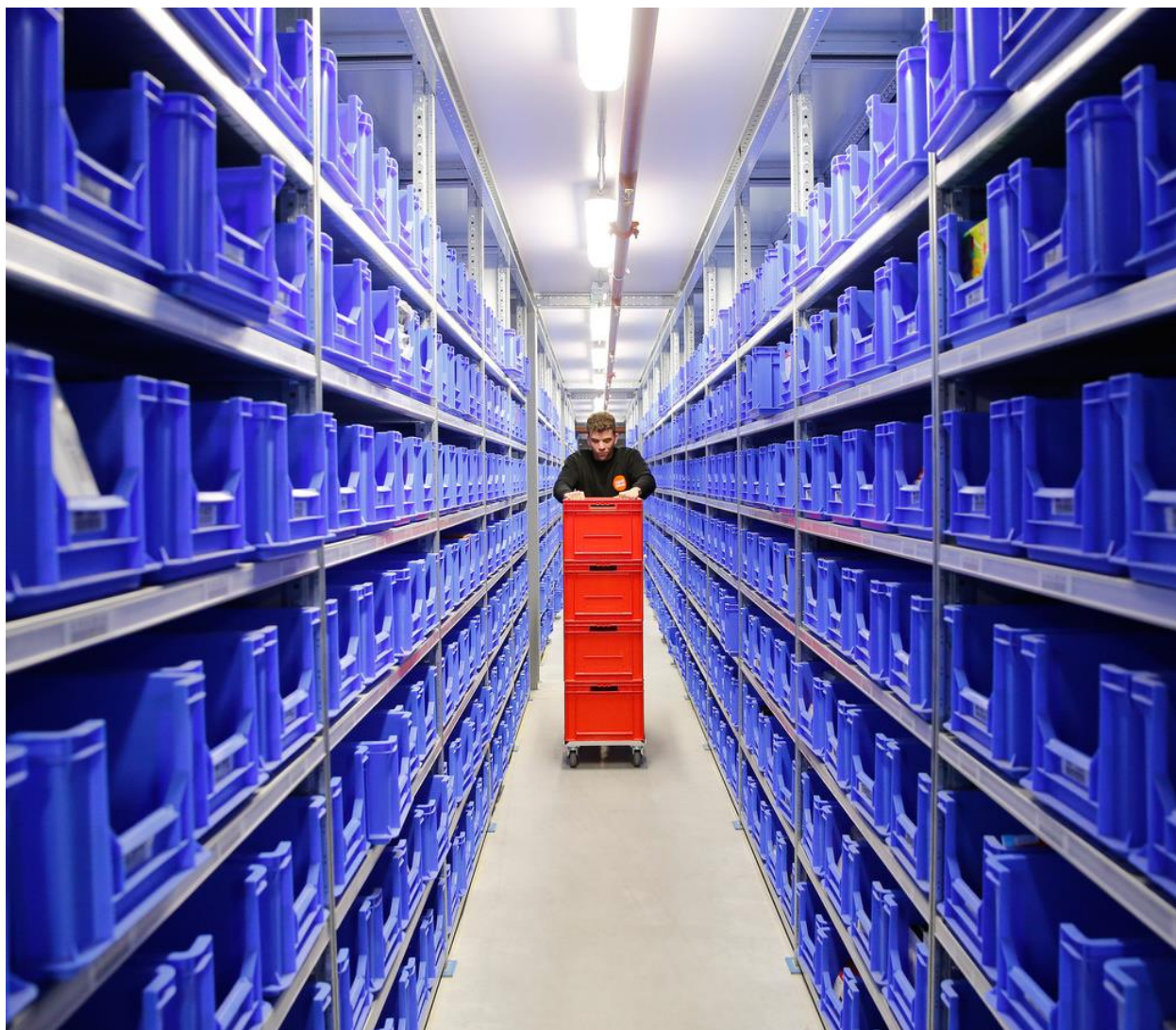
Goede punten

We hebben enorm veel ambitie en lol gehad, ondanks andere omstandigheden. Ook het feit dat we beide thuis waren in het onderwerp heeft geholpen.

11: Logboek

| Hugo Woesthuis | | |
|-------------------|--|------------|
| Aantal uur | Taak/Beschrijving | Datum |
| 1,5 | Filmpjes opzoeken van distributiecentra en analyse | 16-6-2018 |
| 0,5 | PWS Gesprek | 18-6-2018 |
| 3,0 | Schrijven onderzoek | 5-7-2018 |
| 4,0 | Schrijven Inleiding | 5-7-2018 |
| 1,5 | Basis concept systeem | 12-7-2018 |
| 6,0 | Draft uitwerken tot schematic en ontwerpbladen | 14-7-2018 |
| 2,0 | 3D/CAD ontwerp | 16-8-2018 |
| 1,0 | PCB Design | 27-9-2018 |
| 6,0 | Schematic Design | 28-9-2018 |
| 0,5 | PWS gesprek | 4-10-2018 |
| 2,0 | Onderzoek schijven | 17-10-2018 |
| 0,5 | PWS gesprek | 7-11-2018 |
| 3,0 | Onderzoek en software schijven | 13-11-2018 |
| 0,5 | PWS gesprek | 14-11-2018 |
| 2,0 | Onderzoek schrijven | 14-11-2018 |
| 4,0 | Onderzoek afronden, verslag combineren | 15-11-2018 |
| 6,0 | Sectie bouwen schrijven | 20-11-2018 |
| 1,0 | Software schijven voor Raspberry Pi | 20-11-2018 |
| 0,5 | PWS Gesprek | 21-11-2018 |
| 7,0 | Bouwen schrijven verslag | 22-11-2018 |
| 5,0 | Software schijven voor Raspberry Pi | 23-11-2018 |
| 1,0 | Bouwen schrijven verslag | 23-11-2018 |
| 3,0 | Schrijven verslag | 24-11-2018 |
| 3,0 | Bouwen van de software (Raspberry Pi) | 25-11-2018 |
| 8,0 | Schrijven verslag (bouwen) | 26-11-2018 |
| 2,0 | Programmeren | 27-11-2018 |
| 0,5 | PWS gesprek | 28-11-2018 |
| 3,0 | Programmeren en schrijven van verslag (Hoofdstuk 5) | 30-11-2018 |
| 2,0 | Programmeren (daemon) | 5-12-2018 |
| 0,5 | PWS Gesprek | 5-12-2018 |
| 4,0 | Verslag schrijven (Hoofdstukken 6 en 7) | 7-12-2018 |
| 4,0 | Finaliseren van het verslag | 9-12-2018 |
| 6,0 | Definitief verslag schijven, checken en digitaal signeren. | 10-12-2018 |
| TOTAAL | | 94 |

| Mike Krop | | |
|------------|---|------------|
| Aantal uur | Taak/Beschrijving | Datum: |
| 2,0 | Filmpjes bekijken over de processen in een distributiecentra | 16-6-2018 |
| 0,5 | PWS gesprek | 18-6-2018 |
| 4,0 | Server share aanmaken, syncen met het programma OneDrive | 24-6-2018 |
| 1,5 | Hoofd- en Deelvragen bedenken, verwerken en uitwerken | 25-6-2018 |
| 0,5 | Logboek maken en invullen | 25-6-2018 |
| 2,0 | Schrijven Inleiding | 5-7-2018 |
| 3,5 | Schrijven Onderzoek | 5-7-2018 |
| 3,5 | Schrijven Werkwijzen | 6-7-2018 |
| 10,0 | 3D Ontwerp ontwerpen | 10-8-2018 |
| 1,0 | Protocol schrijven | 27-9-2018 |
| 3,5 | Basis leggen voor de Arduino code | 28-9-2018 |
| 0,5 | PWS gesprek | 4-10-2018 |
| 4,5 | 3D Ontwerp opnieuw designen | 20-10-2018 |
| 1,0 | Share opnieuw organiseren | 4-11-2018 |
| 0,5 | PWS gesprek | 7-11-2018 |
| 0,5 | PWS gesprek | 14-11-2018 |
| 3,5 | Overal aan werken binnen het verslag | 20-11-2018 |
| 0,5 | PWS gesprek | 21-11-2018 |
| 4,0 | De Commentaar punten verwerken en andere dingen verbeteren | 24-11-2018 |
| 3,5 | Uiterlijk van Verslag doen | 26-11-2018 |
| 6,0 | Overal aan werken binnen het verslag | 27-11-2018 |
| 4,0 | Programmeren | 27-11-2018 |
| 0,5 | PWS gesprek | 28-11-2018 |
| 2,0 | Uiterlijk van Verslag doen | 30-11-2018 |
| 5,0 | Overal aan werken binnen het verslag | 2-12-2018 |
| 0,5 | PWS gesprek | 5-12-2018 |
| 4,0 | Bouwen | 5-12-2018 |
| 2,0 | Uiterlijk van Verslag doen | 6-12-2018 |
| 4,0 | Schrijven Hoofdstukken 6 en 7 | 7-12-2018 |
| 2,0 | Evaluatie schrijven | 8-12-2018 |
| 2,0 | Uiterlijk van Verslag doen | 8-12-2018 |
| 4,0 | Finaliseren van het verslag | 9-12-2018 |
| 7,0 | Finaliseren van het verslag | 10-12-2018 |
| 3,0 | Uiterlijk van Verslag doen en alles klaar maken voor inleveren. | 10-12-2018 |
| | | |
| TOTAAL | | 97 |



Dit is het einde van het verslag

Met dank aan meneer Van Dijk voor de begeleiding!