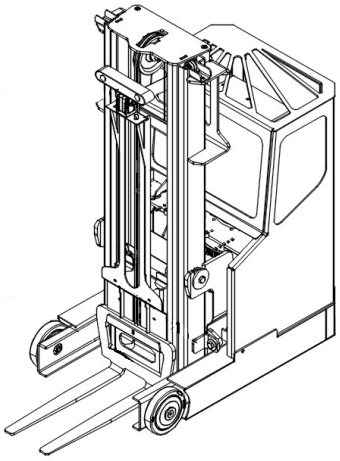
# Teknisk rapport
# Autonomstyrning av gaffeltruck

Version 1.0

L.A.M.A
22 december 2016



## Status

| | | |
|---|---|---|
| Granskad | Samtliga projektmedlemmar | 2016-12-15 |
| Godkänd | Andreas Bergström | 2016-12-21 |

## Projektidentitet

| | |
|---|---|
| **Gruppmail:** | jenst280@student.liu.se |
| **Hemsida:** | http://www.isy.liu.se/edu/projekt/reglerteknik/2016/forklift/ |
| **Beställare:** | Andreas Bergström, ISY, Linköpings universitet<br>**Telefon:** +46 (0)10-711 54 54, **Mail:** andreas.bergstrom@liu.se |
| **Kund:** | Magnus Persson, Toyota Material Handling<br>**Telefon:** +46 (0)705560620 , **Mail:** magnus.persson@toyota-industries.eu |
| **Kursansvarig:** | Daniel Axehill, ISY, Linköpings universitet<br>**Telefon:** +46 (0)13 284042, **Mail:** daniel@isy.liu.se |
| **Projektledare:** | Jenny Stenström |
| **Handledare:** | Erik Hedberg, ISY, Linköpings universitet<br>**Telefon:** +46 (0)13 281338 , **Mail:** erik.hedberg@liu.se<br>Samuel Lindgren, Toyota Material Handling<br>**Telefon:** +46 (0)767614024 , **Mail:** samuel.lindgren@toyota-industries.eu |

## Gruppmedlemmar

| Namn | Ansvarsområde | Telefon | Mail (@student.liu.se) |
|---|---|---|---|
| Johan Almgren | Testansvarig | 070-206 72 45 | johal611 |
| Henrik Andersson | Dokumentansvarig | 073-854 77 79 | henan562 |
| Gustav Elingsbo | Informationsansvarig | 073-685 19 75 | gusel411 |
| Mikael Hartman | Integrationsansvarig | 076-771 13 38 | mikha130 |
| Petter Lannerhed | Designansvarig | 070-627 82 52 | petla189 |
| Andreas Norén | Mjukvaruansvarig | 072-394 89 95 | andno111 |
| Jenny Stenström | Projektledare | 070-329 92 21 | jenst280 |

## Dokumenthistorik

| Version | Datum | Ändringar | Utförd av | Granskare |
|---|---|---|---|---|
| 0.1 | 2016-12-12 | Första utkastet | L.A.M.A. | Samtliga projektmedlemmar |
| 1.0 | 2016-12-15 | Första versionen | L.A.M.A. | Samtliga projektmedlemmar |

# Innehåll

# 1 Inledning

Detta dokument är en teknisk dokumentation för projektet *Planering och Sensorfusion för Autonom Truck* i kursen *TSRT10 - Reglertekniskt Projektkurs* som ges vid Linköpings universitet. Rapporten ska ge en teknisk beskrivning över projektets implementation och design.

## 1.1 Parter

I projektet ingår följande parter:

| | | |
|---|---|---|
| Kund | Magnus Persson | Toyota Material Handling |
| Beställare | Andreas Bergström | ISY |
| Handledare | Erik Hedberg | ISY |
| Teknisk support | Samuel Lindgren | Toyota Material Handling |
| Projektgrupp | L.A.M.A. | LiU |

## 1.2 Projektets bakgrund

Det finns en ökad efterfrågan för autonoma truckar och det är ett forskningsintensivt område. Fördelar som ökad säkerhet, lägre kostnader och möjlighet till mindre lageryta gör användningen av autonoma truckar ekonomiskt fördelaktigt. Toyota Material Handling har idag vissa självgående truckar men dessa är komplicerade att installera och inte särskilt flexibla. Att ta fram ett system som lätt kan integreras i en befintlig miljö och som snabbt kan anpassa sig till sin aktuella omgivning är av intresse, vilket ligger till grund för detta projekt.

## 1.3 Syfte och mål

Syftet med detta projekt har varit att projektgruppen ska lära sig att arbeta enligt projektmodellen LIPS samt att få tillämpa sina teoretiska kunskaper på ett verkligt problem. Tillsammans med support från handledare och Toyota har den befintliga trucken förbättrats avseende reglering, ruttplanering och autonomi. Dess mjukvarumodeller och simuleringsmiljö har uppdaterats. I början av projektet skrevs en kravspecifikation [1] för att definiera vad som skulle åstadkommas under projektet.

## 1.4 Användning

Den verkliga produkten är tänkt att utföra uppgifter i ett lager, så som att hämta laster i form av pallar och flytta dessa autonomt till givna pallplatser. Trucken i detta projekt är dock en nedskalad variant, i skala 1:3, som är tänkt att användas i demonstrationssyfte på mässor och olika företagsevent. Den ska visa upp Toyotas satsning på autonomi och även vad truckar kan tänkas göra i framtiden.

## 1.5 Definitioner

Nedan definieras vissa termer som är återkommande i dokumentet.

| | |
|---|---|
| ROS | Robot Operating System, en Open Source plattform som möjliggör och underlättar kommunikation mellan komponenter och moduler vid robotutveckling. |
| Nod | En process i ROS ramverket som självständigt utför beräkningar. Mjukvaran är uppbyggd av ett nät av noder som själva utför uppgifter och kommunicerar via topics, services och parametrar. |
| Topic | En namngiven bus där data kan delas mellan noder. En nod kan dela data på en topic och/eller lyssna efter ny data på en topic. |
| Service | Ett funktionsanrop mellan noder där en nod kör en funktion i en annan och inväntar svar. För att en service ska kunna användas måste en nod först erbjuda servicen. |
| Publisher | En funktionalitet som låter en nod dela data på en topic. |
| Subscriber | En funktionalitet som låter en nod lyssna efter och reagera på ny data på en topic. |
| Modul | Konceptuell uppdelning av systemets mjukvara baserad på funktionalitet. |
| IMU | Inertial measurement unit. |
| Gazebo | Simuleringsmiljö som används för att testa truckplattformens funktioner och programvara i en virtuell miljö. |
| MiniTruckAppen | Den Androidapplikationen som kan kommunicera med plattformen. |

# 2 Utvecklingsmiljö

## 2.1 Hårdvara

Hårdvaran består av en modell av en riktig truck i Toyotas sortiment i skala 1:3, även kallad Minireach. Den drivs i dagsläget av likström från ett litiumbatteri och kommunicerar med sin omgivning med hjälp av WiFi-uppkoppling.

### 2.1.1 Sensorer

Utöver normal funktionalitet hos Toyotas truckar är Minireach utrustad med 2 LIDAR (lasersensorer) som är placerade fram och bak på trucken. Dessa används för att kartlägga och positionera trucken i rummet samt för att positionera in sig under pall. Trucken har dessutom en 3D-kamera för att kunna detektera AR-koder.

### 2.1.2 Motorik

Trucken har tre hjul, två fram och ett bak, och drivs på det bakre av dessa. Den har gafflar som kan höjas och sänkas, där 3D-kameran som sitter högst upp på detta parti kan vinklas upp och ner.

## 2.2 Ramverk mjukvara

Mjukvaran bygger på ramverket ROS om är en open source-plattform för att utveckla robotar. ROS bygger på att användare delar sina projekt med varandra och det finns en mängd olika funktionaliteter, så kallade paket, att ladda ner för att sedan anpassa till sin lösning. I paketen finns noder, program som kan köras parallellt och kommunicera med varandra genom topics, services och actions.

Ett exempel på ett färdigt paket är move_base som förutom att förflytta roboten även planerar en lämplig rutt baserat på den cost-map som finns över området.

### 2.2.1 Dynamic reconfigure

Med hjälp av dynamic regonfigure kan vissa parametrar ställas om tillfälligt både från simuleringsmiljön och i funktioner i koden.

## 2.3 Simuleringsmiljö

Större delen av projektet har genomförts i simuleringsmiljön Gazebo på grund av begränsad åtkomst till den fysiska trucken. Detta på grund av de risker som är sammankopplade med att testa kod på denna både för hårdvaran och för omgivningen.

Under projektet har truckens modell i Gazebo uppdaterats för att bli mer lik den verkliga trucken. Körriktningen har ändrats och brus har lagts till för att det mer ska likna verkliga förhållanden.

### 2.3.1 Skillnader från hårdvara

Eftersom projektet genomförts mestadels i simuleringsmiljön så bör det poängteras att vissa skillnader finns gentemot hårdvaran. I simulering kan i princip oändligt små förändringar i styrsignaler leda till förändringar av truckens läge. I praktiken har servon en minsta styrsignalsförändring som måste uppfyllas för att uppnå en praktisk förändring. En annan skillnad är att gafflarna ej kan lyftas till en bestämd höjd genom publicering på ett specifikt topic, vilket är möjligt i simuleringen.

# 3 Översikt av systemet

I detta kapitel ges en översikt av systemet.

## 3.1 Körmoder

Det finns tre olika moder som man kan köra i:

1. **Manuell körning:** Trucken körs manuellt givet kommandon från användaren. Dessa kan skickas från terminalfönster, androidapplikation eller spelkontroll som är ansluten mot trucken. Det manuella läget är alltid aktiverat och kommandon direkt från användaren kommer gå före övriga kommandon.
2. **Kartläggning:** Vid kartläggning får användaren köra omkring manuellt. En karta byggs upp och när användaren anser att den är klar kan kartläggning avslutas och kartan sparas undan. Kartan går sedan att använda för att köra autonoma uppdrag. Endast fasta hinder får finnas i kartan, eventuella pallar måste vara bortplockade.
3. **Autonomt uppdrag:** Trucken kan flytta en känd pall till en känd pallplats. Detta innebär att trucken måste ha sett de givna AR-koderna någon gång för att veta dess positioner. Läs mer om detta i kapitel 4.

## 3.2 Ingående delsystem

Mjukvaran består av ett antal delsystem. Delsystem består av kod som körs och har delats upp efter vilken typ av funktionalitet de innehåller. De delsystem som finns är:

- Beslutsfattning
- Förflyttning
- Kartläggning
- Positionering
- Användargränssnitt

De olika delsystemen beskrivs i kommande kapitel.

# 4 Beslutsfattning vid autonomt läge

Detta kapitel kommer beskriva hur vi har strukturerat utförandet av autonoma uppdrag med hjälp av så kallade SMACH-tillstånd.

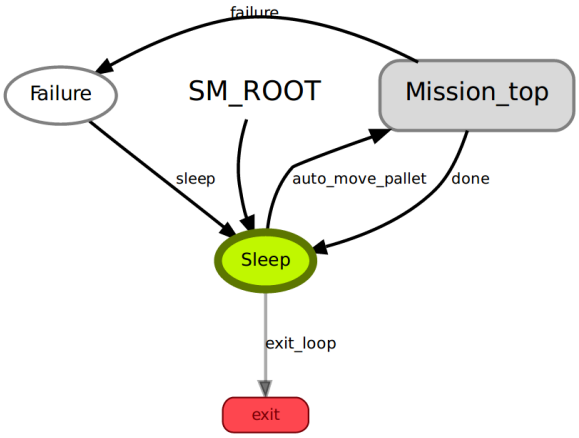
## 4.1 Introduktion till SMACH

Vi har använt oss av en tillståndstruktur som kallas SMACH, som finns implementerat i ROS tillsammans med programmeringsspråket Python [2]. Fördelen med detta är att tillståndsträd kan skapas och därmed dela upp funktionaliteten i mindre delar. Det skapar också möjlighet att gör olika saker beroende på resultatet från en tidigare uppgift. Till exempel om något går fel vid inkörning under pall, kan ett feltillstånd initieras och försöka utföra någon form av felhantering. SMACH-ger också möjlighet att bygga upp en uppgift i flera nivåer vilket innebär att ett tillstånd kan innehålla flera sub-tillstånd. Detta gör det lätt att bryta ner en uppgift i mindre och mindre bitar, vilket möjliggör parallell utveckling samtidigt som det ger en bättre överblick över systemet. De olika tillstånden kommunicerar sedan med övrig funktionalitet genom topics, services och actionhandlingar. För mer information kring hur SMACH fungerar se ROS-wiki [3].

## 4.2 Övergripande design

Ett tillståndsträd bestående av SMACH-tillstånd har strukturerats, för att lättare kunna följa koden och lägga till/ta bort tillstånd. Översta nivån av tillståndsträdet kan ses i figur 1. Tillståndsträdet körs på en separat nod, **executive**.

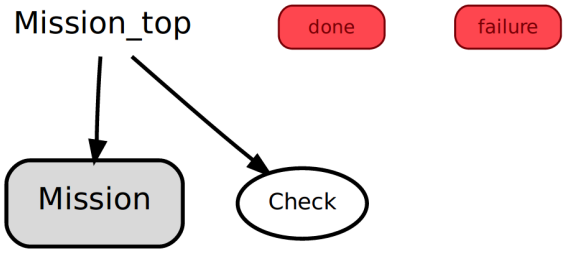
- **Sleep:** Tillstånd som ligger och väntar på instruktion. När ingen uppgift utförs är trucken i detta tillstånd. Startar ett nytt uppdrag när användaren publicerar på topicen */send_mission*. Meddelandetypen som skickas innehåller information om uppdrag ska starta och isåfall om vilken pall och pallplats som ska hämtas. Detta görs med nummer som AR-koderna på respektive pall och pallplats representerar. Dessa AR-koder kommer sedan skickas med i en speciell klass som heter ”Auto_move_info”. Denna klass innehåller även stöd för att skicka vidare positioner med mera när detta hämtas. Denna klass som skapas skickas med till alla tillstånd uner hela uppdraget.
- **Mission_top:** Tillstånd innehållandes många sub-states. Har funktionalitet för att utföra autonomt uppdrag. Kan returnera att uppdraget är klart, ”done”, eller att något har gått fel, ”failure”. Se även figur 2.
- **Failure:** Tillstånd ifall något har gått fel under uppdraget. Användaren meddelas och även felsökning kan läggas till, detta finns dock inte implementerat idag. Kan returnera ”sleep”.
- **Exit:** Är inget tillstånd utan bara vad den översta nivån kan retunera om det skulle vara en del av ett större tillståndsträd.




Figur 1: Översta nivån i beslutsträdet.

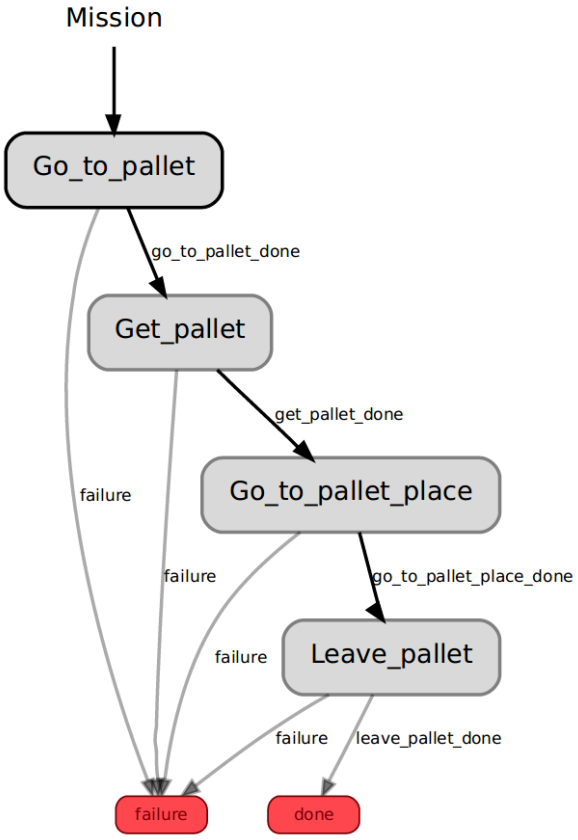
Tillstånden i figur 3 erhålls en nivå ner i **Mission_top**. Det innehåller:

- **Mission:** Innehåller funktionalitet för att utföra det autonoma uppdraget. Se även figur 3.
- **Check:** Tillstånd som körs parallellt med **Mission**. Detta tillstånd kollar om användaren skickar ett kommando för att avbryta uppdraget. **Check** har då möjlighet att avbryta **Mission** och det som händer där i.
- **done/failure:** Vad som kan returneras från **Mission_top**.




Figur 2: Tillstånd i **Mission_top**.

Ytterligare en nivå ner ligger funktionaliteten uppdelad för att utföra flytt av pall autonomt. Tillståndet **Mission** kan ses i figur 3 och innehåller tillstånd som beskrivs i avsnitt 4.2.1 till 4.2.4.




Figur 3: Tillstånd i **Mission**.

### 4.2.1 Kör till pall

I **Go_to_pallet** finns funktionaliteten för att hämta truckens position och positionera den framför pallen. Detta sker med följande tillstånd:

- **Get_pallet_position:** Hämtar ut vilken position pallen med den givna AR-koden har. Detta görs genom att kolla på det topic där alla pallar som finns i lagret publiceras.
- **Calculate_pallet_position:** Modifierar position till en position 0,7 m framför pallen med en vinkel mot pallen som trucken ska åka till.
- **Move_forklift_to_pallet:** Använder *Move_base* och skickar den modifierade positionen som kommer flytta trucken framför pallen. Detta sker genom en actionhandling.
- **Fail_go_to_pallet:** Om något går fel i de övriga tillstånden kommer de returnera ”failure” och hamna i detta tilstånd. Här kan eventuell felhantering ske men i dagsläget meddelas bara användaren att något gått fel sedan returneras ”failure” som kommer leda till det övergripandet feltillståndet och uppdraget avbryts.

Om inget fel inträffar kommer **Go_to_pallet** övergå till **Get_pallet**.

### 4.2.2 Hämta pall

I **Get_pallet** positionerar trucken in sig under pallen och lyfter upp den. Detta sker med följande tillstånd:

- **Search_AR_pallet:** Vinklar kameran tills en AR-kod upptäcks.
- **Go_under_pallet:** Anropar med en service *pallet_handler* som med en state machine utvärderar och positionerar in sig i hålen under pallen. Returnerar ”succeed” eller ”failure” beroende på hur uppdraget gick.
- **Lift_fork:** Lyfter upp gafflarna till önskad höjd genom att anropa en action som i sin tur publicerar målhöjden på en topic. Action-funktionen kan i framtiden utvecklas för att t ex kontrollera nuvarande höjd på gafflar och säkerställa att önskad höjd uppnåtts.
- **Fail_go_under_pallet:** Om tillståndet **Go_under_pallet** anses vara misslyckat returneras ”failure” och exekveringen går vidare till detta tillstånd. I nuläget utför detta tillstånd inget utan skickar det vidare till det övergripande feltillståndet, men möjligheten finns att lägga till ytterligare funktionalitet.
- **Fail_lift_pallet:** Om lyftet av gafflarna anses ha misslyckats fortgår exekveringen till detta tillstånd. Likt ovan gör detta tillstånd inget i nuläget utan går endast vidare till det övergripande feltillståndet.

Om inget fel inträffar kommer uppdraget fortsätta till **Go_to_pallet_place**.

### 4.2.3 Gå till pallplats

I **Go_to_pallet_place** förflyttas trucken till pallplatsen vilket sker med hjälp av följande tillstånd.

- **Get_pallet_place_position:** Hämtar pallplatsen position givet dess AR-koden, för den pallplats där pallen ska lämnas. Detta görs på samma sätt som att hämta pallens position. Ett annat topic publicerar dock pallplatserna.
- **Calculate_pallet_place_position:** Beräknar en position 1 m framför pallplatsen.
- **Move_forklift_to_pallet_place:** Använder *Move_base* genom actionhandling för att flytta trucken till pallplatsen.
- **Fail_go_to_pallet_place:** Om något går fel i de övriga tillstånden kommer de returnera ”failure” och hamna i detta tilstånd. Här kan eventuell felhantering ske men i dagsläget meddelas bara användaren att något gått fel sedan returneras ”failure” som kommer leda ut till det övergripandet feltillståndet och uppdraget avbryts.

Inträffar inget fel kommer sista delen att exekveras i **Leave_pallet**.

### 4.2.4 Lämna pall

I **Leave_pallet** lämnas pallen på pallplatsen och trucken backar ut från pallen och avslutar det autonoma uppdraget.

- **Search_AR_pallet_place:** Likt i tillståndet Get_pallet letar kameran efter en AR-kod.
- **Go_over_pallet_place:** Detta tillstånd anropar en service, *deliver_pallet*, som med samma teknik som *pallet_handler* positionerar gafflarna framför den AR-kod som representerar pallplatsen.
- **Lower_fork:** Sänker ned gafflarna till 0.001 m.
- **Fail_go_over_pallet_place:** Om **Go_over_pallet_place** misslyckas och returnerar ”failure” inleds detta tillstånd. I nuläget utför detta tillstånd inget utan skickar det vidare till det övergripande feltillståndet, men möjligheten finns att lägga till ytterligare funktionalitet.
- **Fail_lower_forks:** Om **Lower_forks** misslyckas returneras ”failure” och aktiverar detta tillstånd. Likt de andra felhanteringstillstånden skickar detta tillstånd bara vidare till det övergripande feltillståndet men möjlighet att utöka felhanteringen finns.

Om allt går som det ska är uppdraget avslutat och trucken kommer hamna i tillståndet **Sleep**.

## 4.3 Instruktioner

Nedan listas alla kommandon som modulen använder för att kommunicera med övriga moduler och användaren via *topics*, *service* och *actionhandlingar*.

### 4.3.1 Från användare

Användaren publicerar på ett topic för att starta ett uppdrag, antingen via terminal eller androidapplikationen. Detta topic heter */send_mission* och ska bestå av meddelandetypen ”minireach_executive/SendMission”. Typen av meddelande skickas med enligt nedan:

- *start*: Om nytt uppdrag vill startas. Detta exekveras bara om förra uppdraget är avslutat.
- *stop*: Kan skickas under pågående uppdrag för att avsluta uppdraget.

och sedan AR-koden på pallen och pallplatsen.

### 4.3.2 Till övriga moduler

Nedan listas vilka topics, service och actionhandlingar som används i modulen:

- **Topics**
  - */send_mission*: För att lyssna på om användaren skickar nytt eller avbryter uppdraget.
  - */pallet_ar_markers*: För att lyssna vilka positioner pallarna har.
  - */station_ar_markers*: För att lyssna vilka positioner pallplatserna har.
  - */minireach/fork_position_controller/state*: Topic för att kontrollera gafflarnas höjd.

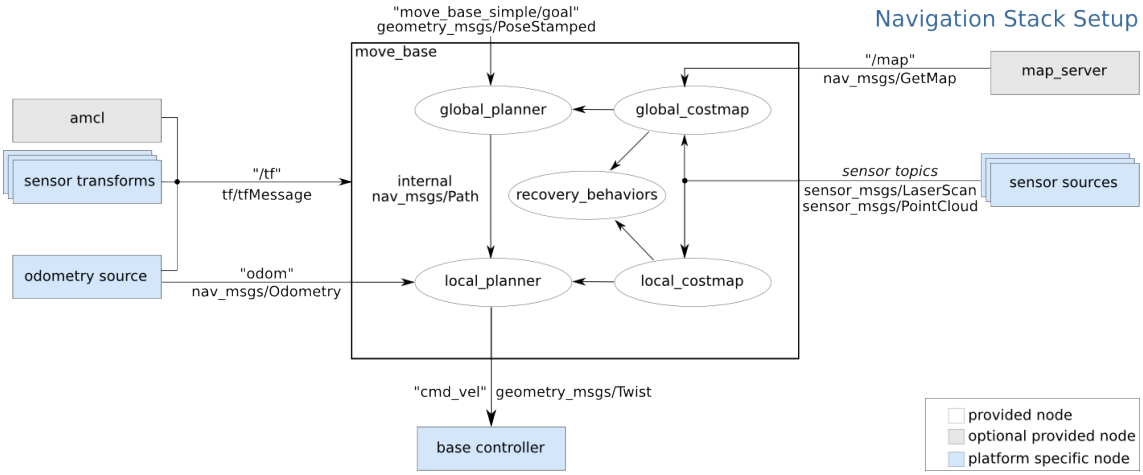
- **Service**
  - *camera_track_ar_frame*: Service som anropas för att scanna efter en given AR-kod. Returnerar sant/falskt beroende på om AR-koden hittas inom området för var 3D-kameran kan vinklas
  - *handle_pallet*: Service som anropas för att köra in under en pall.
  - *deliver_pallet*: Service som anropas för att köra in över pallplats, sänka gafflarna och sedan backa ut från pallen.
- **Actionhandlingar**
  - *move_base*: För att förflytta trucken till given position framför pall/pallplats.
  - *lift_fork*: För att höja/sänka gafflarna till önskad höjd.

## 4.4 Vidareutveckling

Det finns möjlighet till vidareutveckling för beslutsfattningen vid autonomt läge. Framförallt handlar det om felhantering om ett uppdrag inte kan utföras som tänkt. Det finns funktionalitet att gå tillbaka till tidigare tillstånd för att göra om det som har gått fel, alternativt att ny funktionalitet utvecklas för att kunna utföra uppdraget.

# 5 Förflyttning

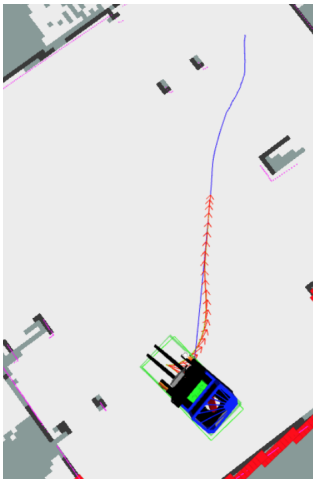
Förflyttning sker med hjälp av paketet **move_base**. Givet en målposition kommer paketet att implementera en action för att försöka förflytta trucken till målet. **move_base** använder sig av en global och en lokal ruttplanerare för att planera truckens rutt från startposition till målposition. I Figur 4 ses en överskådlig bild av **move_base**.




Figur 4: Skiss över ruttplanering. Bild hämtad från [4].

## 5.1 Ruttplanering

För att trucken ska nå en målposition krävs en planering av trajektorian från utgångsposition till målet. Denna ruttplanering sker med en global och en lokal ruttplanerare. I Figur 5 syns trajektorian både från den globala och lokala ruttplaneringen.



Figur 5: Bild över trajektoria från den globala ruttplaneraren (blå) och den lokala ruttplaneraren (grön med röda pilar).

### 5.1.1 Global planering

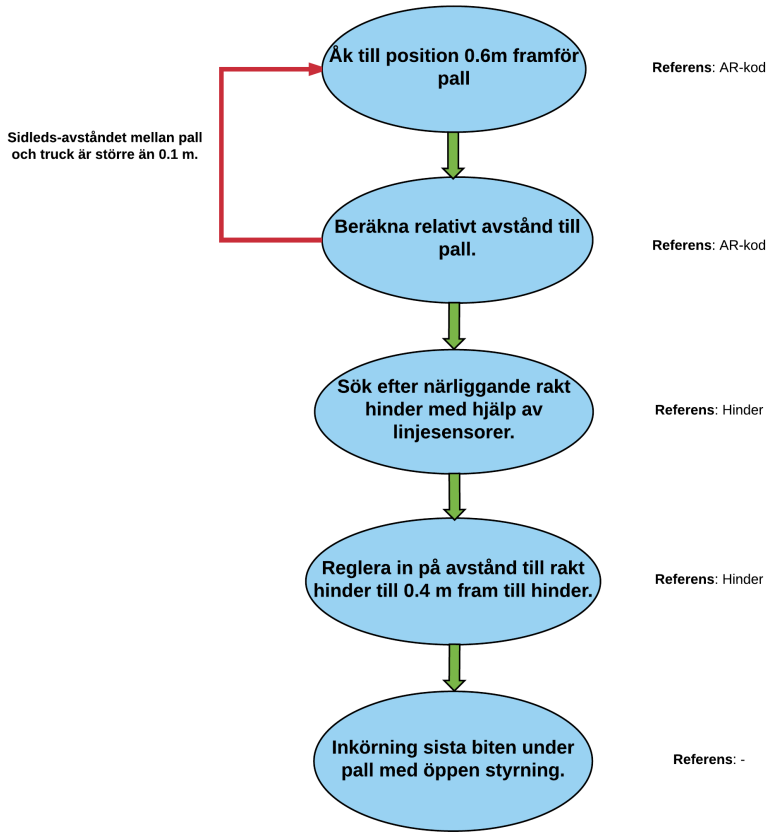
Den globala ruttplaneraren utnyttjar information från en existerande karta av omgivningen för att skapa en global kostnadskarta. Kostnadskartan byggs upp genom att kostnader tilldelas till olika områden på kartan. Nära omgivningen till hinder och väggar får en hög kostnad eftersom det är icke önskvärt att trucken kör nära dem. Utifrån kostnadskartan skapas en trajektoria från nuvarande position till målposition. Om en trajektoria-kandidat passerar genom ett hinder så betraktas den som ogiltig.

### 5.1.2 Lokal planering

Den lokala ruttplaneringen sker med en metod kallad *Timed elastic band* (TEB). Metoden är en online optimeringsalgoritm som lokalt planerar trajektorian. Algoritmen tar hänsyn till avstånd till hinder och exekveringstid samtidigt som den följer kinematiska begränsningar. Användaren kan justera viktparametrar för att få önskat körbeteende. Till exempel kan avvikelse från global planeringsrutt eller avstånd till hinder modifieras. Likt den globala planeringen byggs en lokal kostnadskarta upp vilken skapas med information från sensordata. Det är den lokala ruttplaneraren som sedan skickar styrkommandon vidare till aktuatorer.

## 5.2 Finreglering av position vid pallyft

Vid lyft och avlämning av pall krävs en noggrann positionering av trucken. Detta säkerställer att lyftet genomförs utan att rubba lådan och att avlämning sker inom det specifierade området. Både upphämtning och avlämning av pall sker som tidigare beskrivits i avsnitt 4, med hjälp av tillståndsstrukturen SMACH. Tillståndet **Go_under_pallet**, som beskrivs i avsnitt 4.2.2, använder sig av servicen *handle_pallet* i *pallet_handler*. Denna service utför hela sekvensen från utgångspositionen 0.6 m framför pall till det att gafflarna är i pallens hålrum. Eftersom **move_base** tar hänsyn till avstånd till hinder och att pallen ses som hinder går det inte att ge ett **move_base**-mål att köra in under pallen med en given riktning på gafflarna. Dessutom är det inte säkert att gafflarna skulle positioneras rakt in i pallens hålrum då. Därför har *handle_pallet* en annan förflyttningsalgoritm som är uppdelad i flera states, vilka kan ses i Figur 6.




Figur 6: En översiktlig bild av de states som ingår i **handle_pallet**. De koordinatsystem som används som referens i varje state finns även listade.

Som ses i figuren används två olika koordinatsystem som referens under inkörningen, AR-kod och hinder. Hårdvaruproblem medförde att uppskattningen av AR-kodens position visade sig vara för dålig för att användas vid inkörning under pallen. Därför valdes att under sista biten av inkörningen se lådan som ett hinder och att använda detta hinder som referens. Med hjälp av **detect_obstacles**, vilken finns beskriven i avsnitt 7.3, avsöks därför ett område framför trucken, för att hitta ett hinder framför truckens gafflar som utbreder sig ortogonalt mot gafflarnas riktning. I och med att trucken i detta steg står framför pall så är det hinder som detekteras framsidan på pallen. Positionen och orienteringen på detta hinder används för att skapa styrkommandon, som gör att trucken närmar sig mitten av lådans framsida på ett så rakt sätt som möjligt. Eftersom den LIDAR-sensorn i gafflarnas riktning ej är placerad centralt på trucken kommer **detect_obstacles** att detektera den ena sidan på pallen när trucken kommer nära. Därför körs sista biten in under pall med öppen styrning. Vid lämning av pall används motsvarande algoritm med skillnaden att linjesensorn är aktiv och riktad bakåt vid utbackningen från pallen, vilket utgör motsvarande öppna styrning. Under denna öppna styrning så används dock **detect_obstacles** för att undersöka ifall hinder dykt upp i truckens planerade väg och sekvensen avbryts i så fall.

## 5.3 Styrning av övriga leder

Trucken har även aktuatorer som inte är involverade i förflyttning av truckens grundläggande referensram, *basefootprint_reversed*. Dessa akuatorer används för att styra truc-

kens gafflar respektive kamerastativ. Gafflarna styrs genom att önskad gaffelhöjd publiceras på topic *minireach/fork_position_controller/position/command*. Kamerastativet styrs på motsvarande sätt genom att en vinkel publiceras på topic *minireach/camera_tilt_controller/command*. Dessa meddelande resulterar i att intensitet läggs på motsvarande led baserat på avvikelse mellan aktuell mätsignal samt önskat värde, som då sätts som referens. PID-regulatorerna fork_position_controller respektive camera_tilt_controller bestämmer mängden intensitet som ska läggas på respektive led baserat på denna avvikelse samt parametrarna för respektive regulator som finns definierade i *minireach/minireach_control/config/minireach_control.yaml*.

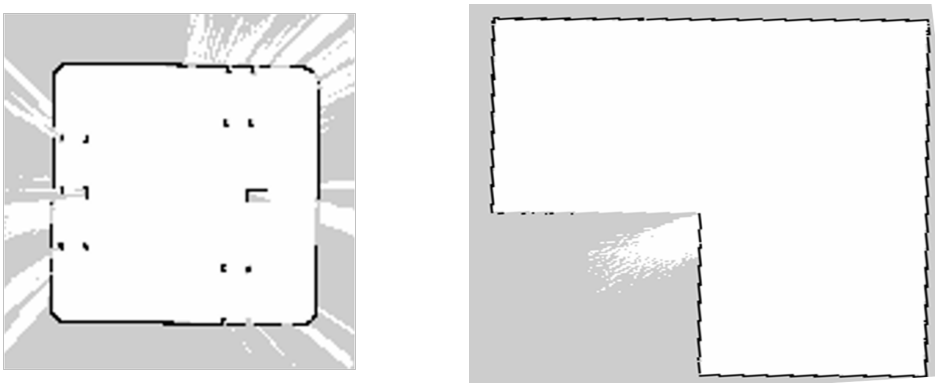
## 5.4 Vidareutveckling

När **move_base** får en körinstruktion att förflytta sig i riktningen ortogonalt mot gafflarna har trucken en tendens att snurra ett helt varv för att uppnå detta istället för att köra den direkta vägen. En potentiell lösning till detta är att korta av prediktionshorisonten i den lokala planeraren så att den är kortare än $\pi$ multiplicerat med den angivna svängradien. Detta säkerställer att lösningen som inte snurrar undersöks innan den som snurrar vilket tillsammans med korrekt inmatade kinematiska tvång gällande till exempel vinkelhastighet borde resultera i ett önskvärt körbeteende.

# 6 Kartläggning

## 6.1 Cartographer

Kartläggning sker med hjälp av paketet **Google Cartographer** som är en SLAM-algoritm, Simultaneous Localization And Mapping. Den tar in sensordata från LIDAR-sensorerna och bildar, tillsammans med vetskapen om hur roboten rör sig, en uppskattning av rummets utformning. Den publicerar en karta i form av en sannolikhetsmatris över rummets beläggning där sannolikheten för att en pixel är upptagen graderas mellan 0 och 1. Värdet 0 innebär en helt ledig pixel och representeras som vit i kartan, en helt upptagen pixel ges värdet 1 och representeras av en svart pixel. Okända pixlar tilldelas värdet -1 och ses som grå i kartan. Kartor över de två testmiljöerna som använts vid simulering ses i figur 7.



(a) Karta över den första testmiljön. (b) Karta över den andra testmiljön.

Figur 7: De två olika demokartorna som används vid simulering.

Kartläggningen tar endast hänsyn till fasta hinder eftersom positionen hos hinder i rörelse tillslut kommer att ses som ledig då det förflyttat sig ur sin ursprungsposition. Därför bör lagret antingen vara tomt från tillfälliga hinder så som pallar under kartläggningen eller så kan den färdiga kartan redigeras så att tillfälliga hinder tas bort i efterhand.

Kartan sparas ner i två filer, en fil i YAML-format och en bildfil i pgm-format. YAML-filen refererar till bildfilen samt innehåller information om kartans upplösning, origo och vilka tröskelvärden som är satta på sannolikhetsmatrisen för att kartan ska uppfatta en pixel som ledig eller upptagen.

Under kartläggningen måste trucken framföras manuellt, och användaren själv måste avgöra när kartan anses vara god nog för att sparas ner och användas. Detta på grund av att det i dagsläget inte finns någon fungerande autonom kartläggningsfunktion implementerad.

## 6.2 Vidareutveckling

En framtida förbättringsmöjlighet är att använda ROS-noden **frontier_exloration** för att göra kartläggningen autonom. Den fungerar i stora drag så att den ser till att inte ha okända pixlar innanför väggar i ett fördefinierat avsökningsområde.

## 6.3 Kartläggningsmodul

### 6.3.1 Övergripande design

Kartläggning är väldigt beräkningskrävande och ett krav har varit att kunna ladda in en befintlig karta. Målsättningen har blivit att ha två separata driftslägen, ett för kartläggning och ett för positionering.

För att klara av detta har noden **switch_nodes** utvecklats med funktionen att kunna öppna och stänga ner de aktuella noderna. Denna nod har två services, */to_mapping* och */to_positioning*, som sköter denna övergång.

Kartan skapas enligt beskrivning i sektion 6.1 och publiceras på topicen **map**.

### 6.3.2 Modulbeskrivning

Modulens services beskris nedan:

- */to_mapping*: stänger ner positioneringsnoden, **AMCL**, samt öppnar upp kartläggningspaketet, **Google Cartographer**.
- */to_positioning*: sparar den nuvarande kartan och robotens position. Stänger sedan ner kartläggningen och öppnar upp positioneringen med den nya kartan och robotens position som estimerad startpunkt.

# 7 Positionering

## 7.1 Global positionering

Global positionering sker med hjälp LIDAR-sensorerna. ROS-noden **AMCL**, Adaptive Monte Carlo Localization, tar in sensorinformationen och positionerar sig till den kända kartan genom användning av ett partikelfilter och Monte-Carlo-analys.

## 7.2 AR-koder

När 3D-kameran detekterar en AR-kod sparas dess position ner till en fil så att trucken senare ska kunna positionera sig mot upptäckta pallar och pallplatser utan att titta på dessa. När trucken kör in mot en pall kommer 3D-kameran att följa koden och ha den inom synhåll under processen.

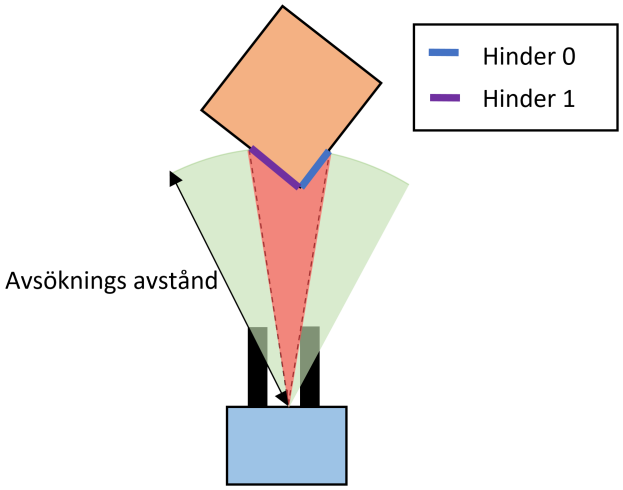
### 7.2.1 Spara positioner

Trucken kan, som tidigare nämnt, spara detekterade AR-koder till en fil. Varje AR-kod är av objekttypen *AlvarMarker* och innehåller ID och position i rummet enligt den nuvarande kartans koordinatsystem. Det görs skillnad på pall och pallplats så de tillhörande AR-koderna sparas till olika filer. Det är från början definierat vilka AR-koder som kommer identifiera pallar och vilka som kommer identifiera pallplatser. Om sedan samma AR-kod ses igen kommer dess nya position i rummet att sparas till filen och skriva över den gamla positionen.

För närvarande finns det två services som kan användas för att endera ta bort en specifik AR-kod från fil eller för att ta bort alla kända AR-koder. Den första servicen */remove_marker*, som kallas med ID-numret av den AR-kod som önskas tas bort från fil. Den andra servicen */clear_all_markers* tar bort alla AR-koder från filerna.

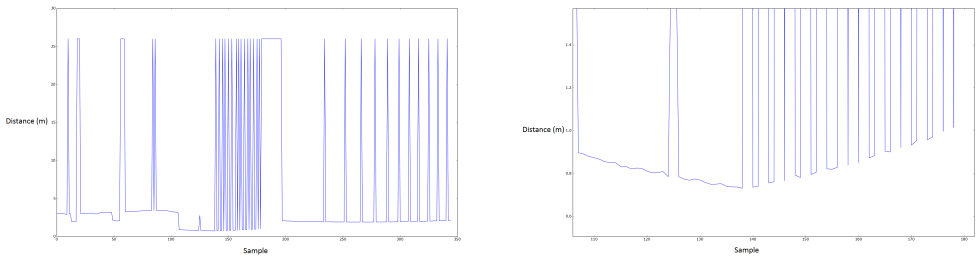
## 7.3 Hinderpositionering

Det ingår funktionalitet för att undvika att köra in i hinder i den lokala ruttplaneraren som baseras på mätvärden från LIDAR-sensorerna. Det finns dock ett visst behov av liknande funktionalitet även då den planeraren inte används. Detta både för att få redundans till det existerande system av säkerhetsskäl samt för att få en noggrannare uppskattning av truckens relativa position och vridning mot hinder jämfört med vad den globala positioneringen kan ge.




Figur 8: Skiss över hinderdetektion.

Funktionaliteten ges i noden **detect_obstacles** och dess parametrar kan styras genom dynamic reconfigure. Funktionen tar ett önskat avsökningsområde i 2D, parallellt mot golvet. Detta är definierat genom ett avsökningsavstånd och ett vinkelintervall inom $[-2\pi,\ 2\pi]$, där vinkeln 0 går mellan gafflarna och framåt. Vinkeln ökar moturs sett i Figur 8. Först tas sensorvärden från LIDAR-sensorerna fram för det önskade vinkel intervallet.




Figur 9: Signal från LIDAR sensorerna över ett utvalt intervall i en simulerad situation likt Figur 8. Hela signalen ses i vänstra figuren och det kortare intervall där avstånd till hinder är kortare än avsökningsavståndet till i figuren till höger.

En avsökning av vinkelintervallet görs efter punkter med avstånd kortare än avsökningsavståndet för att markera ett område där ett eller flera hinder bör finnas. Detta område kan ses i Figur 9. Mätvärden utanför intervallet $[0.05,\ 25]m$ anses defekta och det framgår att signalen innehåller flera defekta mätningar. Dessa filtreras bort på det markerade området genom att anta samma värden som tidigare icke-defekt mätning.

Målet är att upptäcka brytpunkter mellan hinder likt den mellan hinder 0 och hinder 1 i Figur 8. Detta görs genom att köra signalen från det markerade området efter bortfiltrerade defekta mätningar genom ett Gaussiskt filter följt av en kubisk interpolering. Nollställen till interpoleringen markerar då brytpunkter mellan hinder.

Resulterande hinder publiceras som transformer där hindrets mittpunkt är punkten som markeras. Hindrens position publiceras även på topicet */obstacles* där även hindrets bredd

ingår. Hindren indexeras med ökande nummer från 0 utgående från den minsta avsökta vinkeln till den högsta.

Denna information används vid pallyft för att få en mer exakt relativ position och rotation mellan truck och den pall som ska lyftas. Även vid kortare sträckor där trucken kör utan att använda sin lokala planerare används informationen för att undvika att köra in i hinder.

# 8 Användargränssnitt

Det finns behov för en användare att kommunicera med trucken både för att kunna övervaka dess uppfattning av omgivningen samt för att kunna starta uppdrag. Denna kommunikation sker genom publiceringar och avläsningar av olika topics. Det finns även möjlighet att skicka instruktioner genom services.

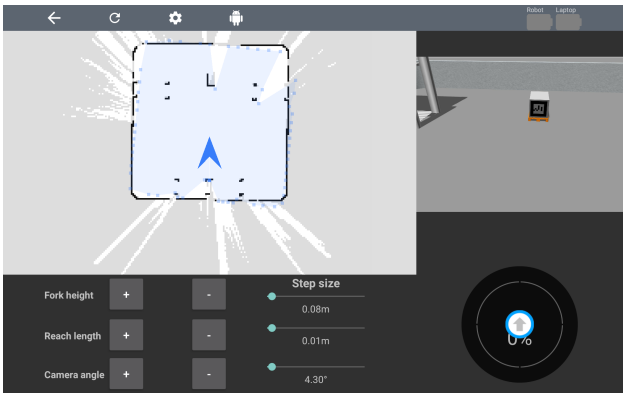
## 8.1 Terminal

Eftersom MiniReach är en utvecklingsplatform sker den mesta kommunikation med trucken via en terminal, på en dator med ROS installerad. Med en extern dator uppkopplad mot datorn finns större möjlighet till en detaljerad feedback över eventuella problem och brister. För information om hur uppkoppling mot trucken praktiskt går till se användarhandledning [5]. I och med att topics och services matas in direkt med denna metod finns ingen begränsning på vilken av truckens funktionalitet som kan kontrolleras och läsas av.

## 8.2 Android-applikation

Ett alternativt gränssnitt finns under utveckling i form av en Android-applikation. Denna utvecklas i Googles utvecklingsmiljö AndroidStudio. För att sätta upp miljö och få en översikt av basfunktionalitet i applikationen se användarhandledning [5].

Applikationen är skriven i Java utgående från öppen källkod för ROS-adaption till Android. I denna finns färdig funktionalitet för bland annat uppkoppling till ett existerade ROS-nodnätverk. Android-applikationen kommer då att kunna hanteras som övriga noder. Applikationen kan utnyttja samma kommunikationssätt som övriga noder, både läsa av och publicera topics men också utnyttja och erbjuda services. Det grafiska gränssnittet kan ses i Figur 10.




Figur 10: Applikation under körning i manuellt läge.

Gränssnittet består primärt av sektionerna kamera och karta samt en joystick för manuell styrning och stegvis ändring av övriga styrbara leder. Övriga vyer är autonom styrning samt inställningar som båda kan nås via det övre menyfältet.

### 8.2.1 Karta

Kartan är uppbyggd av flera lager som var för sig subscribar på ett topic och visar upp informationen grafiskt. Pose-lagret är ett undantag till ovan eftersom den istället publicerar på ett topic. Det går att kombinera lagren fritt för att få önskade vyer. Samma gester som är tillgängliga i kart-applikationen Google Maps fungerar i denna karta. De tillgängliga lagren med tillhörande topics och funktion listas i tabell nedan.

| Lager | Topic | Funktion |
|---|---|---|
| OccupancyGrid | */map* | Bakomliggande kända kartan som trucken positionerar sig inom. |
| LaserScan | */scan* | Mätningar från LIDAR-sensorerna. |
| Path | */move_base/.../global_plan* | Planerad global körrutt för truck. |
| Robot | */base_link* | Markerar robotens position. |
| Pose | */move_base_simple/goal* | Låter användaren markera punkt på karta dit trucken ska köra. |

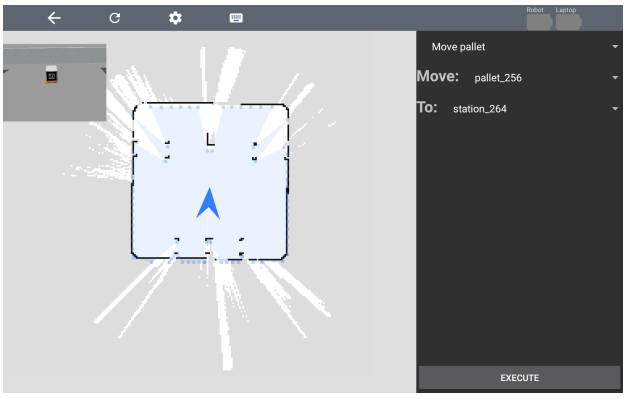
### 8.2.2 Kamera

RGB bilden från truckens 3D kamera visas separat från kartan. Bilden läses av från topicen */camera/rgb/compressed*. Denna publiceras för tillfället endast i den simulerade miljön då bilden kräver hög bandbredd. Det finns möjlighet att låta trucken istället publicera en mer komprimerad bild anpassad för Android applikationer.

### 8.2.3 Manuell styrning

Ett färdigt paket från Google ger joysticken som möjliggör kombinerade rörelser i x- och y-led sett från base_link. Den tillåter även rotation på plats då endast rörelse i y-led anges. Joysticken publicerar på */nav_vel* topicen.

Truckens övriga leder (gaffelhöjd, gaffelinskjutning, kameravinkel) kontrolleras stegvis, där steglängden ställs in med respektive slider. De värden som senast skickats från applikationen till trucken visas men ingen feedback från truckens faktiska sträckor och vinklar tas in från trucken.




Figur 11: Applikation under körning i autonomt läge.

### 8.2.4 Kommandon

Då applikationen körs i autonomt läge används det grafiska gränssnittet från Figur 11. Karta och kamera har samma funktionalitet som vid manuell körning. Till höger listas de tillgängliga kommandona som kan skickas till trucken. Då varje kommando kräver separata parametrar skapas resterande del beroende på valt kommando.

Det enda kommando som finns implementerat på trucken är **Move pallet** som förflyttar en pall med känd position till en pallplats med känd position. Detta är på grund av att det är det enda implementerade kommandot i applikationen, men det finns ett ramverk för att lägga till nya kommandon då dessa blir tillgängliga. **Move pallet** hämtar tillgängliga pallar och pallplatser från topicsen */pallets* och */stations*. Dessa listor uppdateras endast då kommandot markeras.

### 8.2.5 Vidareutveckling

Applikationen har problem med minnesläckor, vilket kan göra att Android-plattformen får slut på minne vid byte mellan manuellt läge och autonomt läge upprepade gånger. Detta behöver åtgärdas delvis genom att gå in i källkoden till ROS Android-anpassningen och se till att alla trådar som öppnas även hanteras vid avslut.

Det öppnas fler trådar än även moderna Android-plattformer kan hantera, där flera av dem som kopplas till topics har callbacks med frekvenser på upp till 15Hz. Detta gör att för stor del av tillgängliga beräkningskraften används till detta. Det lämnar inte tillräckligt med beräkningskraft över till grafiktråden. För att undvika detta kan lager i kartan samt kamerabildströmmen inaktiveras.

I applikationens övre meny finns indikatorer över batteristatus i Android-plattformen samt trucken. Denna funktionalitet har tagits bort från källkoden till ROS Android vilket betyder att den även kan tas bort i denna applikation.

# Referenser

[1] (2016, Sep.) Kravspecifikation - autonom styrning av gaffeltruck. L.A.M.A. TSRT10 LiU. Version 1.0.

[2] http://www.ros.org/. Open Source Robotics Foundation. Besökt: 2016-12-12.

[3] http://wiki.ros.org/. Open Source Robotics Foundation. Besökt: 2016-12-12.

[4] http://wiki.ros.org/move_base. Open Source Robotics Foundation. Besökt: 2016-12-14.

[5] (2016, Dec.) Användarhandledning - autonom styrning av gaffeltruck. L.A.M.A. TSRT10 LiU. Version 1.0.