Bacheloroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

Sindre Vatnamot Amble
Isaac Animpong
Martin Wester Hagen
Trym Helle

Implementering av et PLS-basert fjernstyringssystem for ubemannede overflatefartøy (USV)

Bacheloroppgave i BIELEKTRO
Automatisering og Robotikk & Elektronikk og Sensorsystemer
Veileder: Cevdet Islek - NTNU
Medveileder: Erik Morset - Maritime Robotics
Mai 2023

NTNU
Kunnskap for en bedre verden

MARITIME ROBOTICS

Sindre Vatnamot Amble
Isaac Animpong
Martin Wester Hagen
Trym Helle

Implementering av et PLS-basert fjernstyringssystem for ubemannede overflatefartøy (USV)

Bacheloroppgave i BIELEKTRO
Automatisering og Robotikk & Elektronikk og Sensorsystemer
Veileder: Cevdet Islek - NTNU
Medveileder: Erik Morset - Maritime Robotics
Mai 2023

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

NTNU
Kunnskap for en bedre verden

NTNU

Oppgavetittel: Implementering av et PLS-basert fjernstyringssystem for ubemannede overflatefartøy (USV) Implementation of a PLC based remote control system for unmanned surface vehicle (USV)			
Forfattere: Sindre Vatnamot Amble Isaac Animpong Martin Wester Hagen Trym Helle		Prosjektnummer: Innleveringsdato: Gradering:	E2312 22.05.2023 [x] åpen [ ] lukket
Studium:	Elektroingeniør – BIELEKTRO		
Studieretning:	Automatisering og robotikk Elektronikk og sensorsystemer		
Veileder internt: Institutt:	Cevdet Islek Institutt for teknisk kybernetikk		
Oppdragsgiver: Kontaktperson:	Maritime Robotics Erik Morset		

Sammendrag:
Denne bacheloroppgaven tar for seg utviklingen og implementering av et IP-basert fjernstyringssytem for en USV. Det er også sett på mulighetene for å kjøre Maritime Robotics OBS på en PLS ved hjelp av programmet Docker.

Abstract:
This bachelor's thesis deals with the development and implementation of an IP-based remote control system for an unmanned surface vehicle (USV). The possibilities to run Maritime Robotics' onboard system (OBS) on a PLC with the help of the Docker application has been reviewed.

Stikkord norsk:	Stikkord engelsk:
Fjernstyring, Docker, USV, PLS, IP, C++, RPi, Linux	Remote control, Docker, USV, PLC, IP, C++, RPi, Linux

Forord

Bacheloroppgaven er gjennomført som en avslutning på elektroingeniørstudiet ved Norges Teknisk-Naturvitenskapelige Universitet i samarbeid med bedriften Maritime Robotics AS. Arbeidet er utført av en tverrfaglig prosjektgruppe med kompetanse innen automatisering, robotikk, elektronikk og sensorsystemer. For bidrag til oppgaveløsningen ønsker vi å rette en spesiell takk til følgende:

Erik Morset, systemingeniør ved Maritime Robotics AS (MR), var ekstern veileder og har vært tilgjengelig og fasilitert for en god gjennomføring. Han har delt av sin kunnskap og erfaring, i tillegg til å ha motivert gruppen gjennom hele prosjektet.

Cevdet Islek, universitetslektor ved Institutt for Teknisk Kybernetikk NTNU, har stilt sin kompentanse til rådighet og gitt veiledning om utførelse av bacheloroppgaven og skriveprosessen.

Sindre Fossen og Jan Henrik Lenes, softwareingeniører ved Maritime Robotics, har gitt uvurderlig hjelp med programmering og systemforståelse.

Gruppen ønsker også å takke:

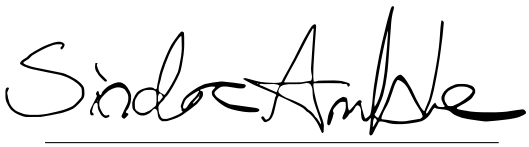
- Even Johan Christiansen for veiledning og anbefalinger til design av kretskort
- Elektronikk og prototypelaboratoriet for produksjon av kretskort
- WAGO Support for god støtte med PLS
- Alle ansatte ved Maritime Robotics AS for hjelp og støtte, i tillegg til godt arbeidsmiljø

Trondheim 22.05.23

Trym Helle

Isaac Animpong

Martin Wester Hagen



Sindre Vatnamot Amble

Sammendrag

Bacheloroppgaven tar for seg utvikling og implementering av et IP-basert fjernstyringssytem for en unmanned surface vehicle (USV). Det er også sett på mulighetene for å kjøre Maritime Robotics (MR) sitt ombordsystem (OBS) på en PLS ved hjelp av programmet Docker. Arbeidet har resultert i en fjernkontroll som kan styre en USV, samt en fullverdig versjon av OBS som kjører via en Docker Container-løsning på en WAGO programmable fieldbus controller (PFC) ved navn PFC200 750-8210.

Fjernkontrollen består av en Raspberry Pi (RPi), et egetutviklet kretskort og en joystick. Den gir operatøren det nødvendige grensesnittet og funksjonene som trengs for fartøystyring. Mulighetsrommet for styring av USV-en blir utvidet, ettersom fjernkontrollen kommuniserer direkte med OBS. Dette gjør det mulig å ta i bruk avanserte styringsfunksjoner som for eksempel *Station Keeping*. MR sitt nåværende fjernkontrollsystem består av en Hetronic-fjernkontroll som kommuniserer direkte med PFC-en ved hjelp av radiofrekvenser. Den har en begrenset rekkevidde, noe som medfører at det må være en Hetronic i hver havn hvor den skal legge til kai. Denne utfordringen blir løst med en IP-basert fjernkontroll som gjør at USV-en kan finmanøvreres fra et kontrollrom og legges til kai hvor som helst i verden. IP-fjernkontrollen er nyskapende og gir nye muligheter for fartøystyring.

Gruppens arbeid viste at det var mulig å kjøre OBS på PFC-en med Docker. Docker egnet seg for å løse problemet fordi det gjør det enklere å utvikle programmer på tvers av plattformer. Dette betyr da at man kan kjøre OBS, skrevet i C++, på en enhet som ikke er designet for å kompilere store C++ prosjekter. I arbeidet til gruppen ble det oppdaget en arkitekturforskjell mellom OBS-versjonene fra MR og arkitekturen til PFC-en. Denne hindringen førte til at gruppen prøvde ulike metoder for å få arkitekturene til å samsvare. Den endelige løsningen baserte seg på å bygge en versjon av OBS med lik arkitektur som PFC-en. Dette ble gjort ved å emulere ved hjelp av Docker. OBS-versjonen gruppen lagde er en fullverdig versjon av MR sitt OBS og kan kjøre på PFC-en. Dette ble vist via en simulering og visualisert i vehicle control station (VCS).

Rapporten viser til fremtidige muligheter for videreutvikling av systemet. Gruppen har flere tanker om forbedringer av systemet som MR kan ta med seg videre. Fjernstyringssystemet som er laget løser oppgaven på en formålstjenlig måte, og er en intuitiv ny styringsmåte.

Abstract

This bachelor's thesis deals with the development and implementation of an IP-based remote control system for an unmanned surface vehicle (USV). The possibilities to run Maritime Robotics' onboard system (OBS) on a PLC with the help of the Docker application has been reviewed. The work has resulted in a remote control with the ability to control a USV, as well as a full-flegded version of OBS which runs by way of a Docker Container solution. This runs on a WAGO programmable fieldbus controller (PFC) named PFC200 750-8210.

The remote control consists of a Raspberry Pi (RPi), a self-developed circuit board and a joystick. It provides the operator with the necessary interface and functions needed for vehicle control. The range of possibilities for controlling the USV is expanded as the remote controller communicates directly with OBS. This makes it possible to use advanced vehicle control functionality, for example *Station Keeping*. MR's current remote control system consists of a Hetronic remote control that communicates directly with the PFC using radio frequency. It has limited range, which insues that there must be a Hetronic remote controller in every port where it will dock. This challenge is solved by an IP-based remote control that allows the USV to be finely maneuvered from a control room and docked from anywhere in the world. The IP remote control is innovative and offers new possibilities for vehicle control.

The group's work showed that it was possible to run OBS on the PFC with Docker. Docker was suited to solve the problem because it makes it easier to develop applications across platforms. This means that one can run OBS, written in C++ on a device that is not designed to compile large C++ projects. Through the group's work an architectural difference was discovered between the OBS versions from MR and the architecture of the PFC. This obstacle led the group to try multiple methods in order to make the architectures match. The final solution was based on building a version of OBS with the same architecture as the PFC. This was done by emulating using Docker. The OBS version the group made is a full-fledged version of MR's OBS and kan run on the PFC. This was shown through a simulation and visualized in vehicle control station (VCS).

This thesis shows future possibilities for further development of the system. The group has several thoughts on different improvements to the system that MR could make use of in the future. The remote control system that has been created solves the task in an adequate way, and is an intuitive new way of control.

Innhold

Forord . . . . . **i**
Sammendrag . . . . . **ii**
Abstract . . . . . **iii**
Figurer . . . . . **vii**
Kodeutdrag . . . . . **viii**
Ordliste & Akronymer . . . . . **ix**
1 Introduksjon **1**
1.1 Bakgrunn 1
1.2 Dagens løsning 2
1.3 Ønsket løsning 2
2 Teori **3**
2.1 USV 3
2.2 Original systemarkitektur 3
2.3 WAGO PFC200 Kontroller 5
2.4 Kommunikasjon 7
2.4.1 IP 7
2.4.2 UDP-protokoll 7
2.4.3 TCP-protokoll 8
2.5 Programvare og maskinvare 9
2.5.1 Linux 9
2.5.2 RPi 9
2.5.3 Make 9
2.5.4 CMake 10
2.5.5 Qmake 10
2.5.6 Make, CMake eller QMake 10
2.5.7 C++ 11
2.5.8 Codesys 12
2.5.9 Protobuf 12
2.5.10 VCS 12
2.5.11 OBS 13
2.6 Docker 13
2.6.1 Dockerfilen 13
2.6.2 Docker Image 15
2.6.3 Docker Container 16
2.6.4 Dockerkommandoer 16
2.6.5 Docker Nettverk 17
2.6.6 Emulering av plattform 17
2.6.7 Effektivisering og optimalisering 18

2.6.8 Docker og virtuell maskin . . . 19
3 Metode . . . 21
3.1 Implementert funksjonalitet . . . 21
3.2 Kommunikasjonsoppsett . . . 21
3.2.1 Valg av kommunikasjonsprotokoll . . . 22
3.2.2 RPi og joystick . . . 22
3.2.3 RPi og PFC . . . 23
3.3 PFC . . . 23
3.3.1 Oppsett av PFC . . . 23
3.3.2 CODESYS V3.5 . . . 24
3.4 Fjernkontroll . . . 25
3.4.1 Joystick . . . 25
3.4.2 Utvikling av fjernkontroller . . . 26
3.4.3 Komponentoversikt . . . 28
3.4.4 Utregning av komponentstørrelser . . . 29
3.5 Fjernkontrollens programkode . . . 31
3.5.1 UDP funksjoner . . . 32
3.5.2 Joystickfunksjoner . . . 35
3.5.3 Brukergrensesnitt og feilmeldinger . . . 36
3.5.4 Ubenyttede funksjoner . . . 37
3.5.5 Kjøring av kode . . . 37
3.6 Integrasjon med MR sitt OBS og VCS . . . 38
3.6.1 Kommunikasjon . . . 38
3.6.2 Funksjonsutførelse . . . 42
3.6.3 VCS-integrasjon . . . 45
3.7 Docker . . . 48
3.7.1 Oppgave . . . 48
3.7.2 Mulige løsninger . . . 48
3.7.3 Planlagt løsning . . . 49
3.7.4 Endelig løsning og fremgangsmåte . . . 51
3.7.5 Bygging og kjøring av Dockerfiler . . . 52
3.7.6 Introduksjon til Dockerfilene . . . 53
3.7.7 Dockerfil for bygging av Debian-pakker . . . 54
3.7.8 Dockerfil for bygging av OBS binærfil og Debian-pakke . . . 55
3.7.9 Dockerfil for kjøring av OBS . . . 57
3.7.10 Effektivisering underveis . . . 58
3.8 Testing . . . 59
3.8.1 PFC . . . 59
3.8.2 USV . . . 59

4 Resultater 61
4.1 Ny systemarkitektur 61
5 Refleksjon og diskusjon 64
5.1 Fremtidige forbedringer og tillegg 64
5.1.1 Forbedringer for fjernkontrollen 64
5.1.2 OBS 66
5.1.3 Docker 66
5.2 Arbeidsprosessen og refleksjoner rundt denne 69
5.2.1 Arbeidspakker 70
6 Konklusjon 71
Kildeliste 72
A Gantt-diagram 76
B S-diagram 77
C Testskjema 78
D ipRemote.cpp 90
E ipRemoteLib.cpp 92
F ipRemoteLib.h 101
G MsgRemoteUdp.proto 103
H Dockerfil for bygging av Debian-pakker 104
I Dockerfil for bygging av OBS 106
J Dockerfil for kjøring av OBS 108
K Skjematikk 110
L Informasjonsplakat 113

Figurer

1 Mariner USV 3
2 Den originale kommunikasjonsarkitekturen for USV og Hetronic fjernkontroll. 4
3 Bilde av WAGO PFC200 750-8210 (Wago, udatert) 5
4 Bilde av WAGO PFC 750-890 på Mariner 6
5 Illustrasjon av UDP kommunikasjon mellom en avsender og mottaker. . . . 8
6 Illustrasjon av hvordan TCP 3-veis håndtrykkprosessen fungerer (GeeksforGeeks, 2021). 8
7 Illustrasjon av Raspberry Pi 4B (Singh, 2021).. 9
8 C++ logo (Wikimedia Commons, 2017). 11
9 Illustrasjon av hvordan de forskjellige delene av Docker henger sammen (Bikram, 2021). 13
10 Visuell beskrivelse av oppbygningen av et Docker Image med flere lag og eksempler på inneholdet i hvert lag (Flade, 2020) 15
11 QEMU logo (Wikipedia, 2012) 17
12 Visuell beskrivelse av forskjellen mellom virtuelle maskiners og Dockers arkitektur (Arora, 2022) 20
13 Illustrasjon av en simplifisert forskjell mellom TCP og UDP (Taylor, 2020) 22
14 Oversikt over komponentene på WAGO PFC200 750-8210 og deres plassering (Wago, udatert). 23
15 Bilde av PFC-riggen med WAGO PFC200 750-8210 24
16 Joysticker som ble brukt og aksevisualisering. 26
17 Versjon 1 og 2 av fjernkontroll 27
18 Endelig versjon av fjernkontroll med HAT 28
19 Overordnet koblingsskjema av fjernkontrollen 28
20 Oversikt over funksjonene til komponentene på fjernkontrollen 29
21 Flytskjema av koden på fjernkontrollen 32
22 F9-menyen i VCS for UdpRemoteInterface-klassen 46
23 VCS med meldinger 48
24 Resulterende binærfil og Debian-pakke av OBS 57
25 Den endelige kommunikasjonsarkitekturen mellom USV og den nye fjernkontrollen. 62
26 Flytskjema for kommunikasjon mellom forskjellige styringsenheter for Mariner USV med den nye fjernkontrollen inkludert. 62

Kodeutdrag

1 Grunnlegende Dockerfil 14
2 Dockerkommando 16
3 Dockerfil uten lagoptimalisering 18
4 Dockerfil med lagoptimalisering 19
5 Oppkobling av server og klient 33
6 Oppkobling av server og klient 33
7 Joystickverdier 33
8 Sending av verdier 34
9 Mottaking av verdier 34
10 Sjekk av nye meldinger 34
11 Oppstart av kommunikasjon med joystick 35
12 Oppstartsrutine 35
13 Tilbakemelding 36
14 Modusskifte 36
15 Kompilering av fjernkontrollerkode 37
16 Kjøring av fjernkontrollerkode 38
17 Kode for å lese UDP-meldinger 39
18 Struct som mottas fra RPi 39
19 Kode for å overføre mottatt melding til definert struct 40
20 Protobuf meldingsstrukturen som sendes 40
21 Funksjon for å sende tilbakemeldinger til fjernkontrollen 41
22 Kode som regner ut pådrag og sender ut på aktuatorer 42
23 Kode for å aktivere Station Keeping 43
24 Kode for å legge til verdier i VCS-menyen 45
25 Planlagt løsning for Dockerfil, pseudokode 50
26 Installering av QEMU 52
27 Start Docker daemon 52
28 Bygging av Dockerfil 53
29 Kjøring av Dockerfil 53
30 Docker FROM og USER 54
31 Kopierer internavhengige bibliotek 54
32 Installering av nøvendige avhengigheter 55
33 Kopiering og installering av Debianpakker 56
34 Bygging av Debianpakke 57
35 Kopierer filer for simulering og kjøring av OBS 58
36 Kjører OBS automatisk ved åpning av Containeren 58

Ordliste & Akronymer

ARM hard float (*ARMhf*) ARM Hard Float er en Debian port for ARM Prosessorer. 51, 52, 57

C++ C++ er et høynivå objektorientert programmeringsspråk. ii, 1, 11, 12, 14, 23, 31, 34, 49, 57, 69

Central processing unit (*CPU*) En CPU er datamaskinens hjerne, og utfører beregninger, heter hovedprosessor på norsk. 17

Docker Docker er en programvare for å implementere kode uavhengig av hvilken plattform eller arkitektur det er laget for originalt. ii, vii, 2, 5, 6, 13–20, 23, 24, 48–59, 61, 64, 66–69, 71

Docker Container En Docker-container er en kjørende versjon av et Docker image. ii, 14, 16–18, 20, 23, 25, 52, 54, 59, 61, 67, 68, 71

General purpose input and output (*GPIO*) GPIO er generelle inn- og utdata ledere koblet til et kretskort. 29, 30, 36, 38

Hardware attached on top (*HAT*) En HAT er et toppfestet kretskort, ofte til bruk på mikrodatamaskiner som Raspberry Pi (RPi). vii, 27, 28, 65

Hetronic Hetronic er MRs radiobølgebaserte fjernkontroll til Mariner. ii, vii, 1–5, 61, 63, 65, 66

Input/output (*I/O*) I/O er en forkortelse for inn- og utdata fra et system. 3

IP internettprotokoll. ii, 2, 7, 25, 26, 32, 38, 45–47, 50, 61, 63, 66–68, 71

LED lysemitterende diode. 21, 26, 30, 64–66

Mariner Mariner er en USV utviklet av Maritime Robotics. vii, ix, 1, 3, 5, 6, 22, 62, 65

MR Maritime Robotics AS. i, ii, ix, 1–3, 5, 11–13, 21, 23–25, 38, 43, 44, 48–60, 65, 68, 69, 71

NTNU Norges Teknisk-Naturvitenskapelige Universitet. i, 27

OBS ombordsystem. ii, vii, 1–5, 11, 13, 21–23, 25, 26, 31, 38, 40–43, 45, 46, 48–53, 55–59, 61, 64–69, 71

OS operativsystem. 6, 7, 9, 10, 14, 16, 18–20, 59

Otter Otter er en USV utviklet av Maritime Robotics. 65

PFC programmable fieldbus controller. ii, vii, x, 1–7, 21–25, 48–53, 57–59, 61, 66–68, 71

PLS programmerbar logisk styring. i, ii, x, 5, 6, 12, 20, 22, 68, 69

PS4 PlayStation 4 spillkonsoll. 25, 26

QEMU QEMU er en programvare som emulerer andre datamaskinarkitekturer på en vertsmaskin. vii, 17, 18, 52

QT Creator QT Creator er et PC-program for programmering av C++-kode. QT Creator har støtte for egne QT-bibliotek, i tillegg til funksjonalitet for make, CMake og qmake, som alle er varianter av et automatiseringsprogram for kodekompilering. 10–12

Revolutions per minute (*RPM*) På norsk: Omdreiningstall eller turtall. 4, 21, 36, 63, 66, 71

RPi Raspberry Pi. ii, vii, ix, 7, 9, 21–23, 25, 27–31, 37, 39–41, 45–47, 50, 59, 61, 65, 66

TCP transmission control protocol. vii, 7, 8, 22, 23, 66, 67

UDP user datagram protocol. vii, 7, 8, 12, 21–23, 31, 32, 38, 40

USB universell seriebuss. 21, 22, 25

USV unmanned surface vehicle. ii, vii, 1–5, 12, 13, 21–23, 25, 27–29, 32–38, 40–49, 59–68, 71

VCS vehicle control station. ii, vii, 2, 3, 5, 12, 28, 38, 40, 42, 45–48, 61–64, 66

Visual Studio Code (*VSCode*) VSCode er et koderedigeringsprogram som kan brukes til de fleste programmeringsspråk. 14

VPN virtuelt privat nettverk. 25

WAGO WAGO GmbH er en produsent av elektroniske komponenter, blant annet PLS-er som de kaller PFC. i, ii, vii, 3, 5, 6, 23, 24, 48–51, 59, 61

1 Introduksjon

Denne bacheloroppgaven omhandler utvikling av en fjernstyringsmodul til USV, og designvalg rundt denne løsningen. En stor del av rapporten diskuterer programkode i C++ og implementeringen av dette. Arbeidsgiver, Maritime Robotics AS (MR), har engelsk som arbeidsspråk og det er derfor naturlig at programmering og kommentering av kode forekommer på engelsk.

Deler av denne bacheloroppgaven er underlagt en taushetserklæring, og noe kode vil derfor ikke diskuteres i detalj eller være vedlagt. Denne koden vil forklares kort i rapporten, og det skal være tydelig hva den gjør, selv om den tekniske måten dette gjøres på ikke forklares.

1.1 Bakgrunn

Maritime Robotics AS (MR) er en Trondheimsbasert bedrift som produserer unmanned surface vehicle (USV), også kalt ubemannede fartøy. MR produserer USV-er som er mellom to og ni meter lange. Slike ubemannede fartøy krever avanserte stryringssystemer og sikkerhetsfunksjoner, nettopp fordi de er ubemannede. MR ønsket å utrede muligheten for å legge avansert funksjonalitet fra styringssystemet de har på sine USV-er på PFC-en som sitter i disse. Dette er ønsket implementert både for å kunne benytte PFC for styring, men også for å ha en redundans i tilfelle forbindelsen mellom PC-en, hvor ombordsystemet (OBS) er lokalisert, og PFC-en brytes. Det var også et ønske om å implementere en fjernkontroll med mulighet for internett-basert finmanøvrering. Ved prosjektets start hadde den eksisterende Hetronic-fjernkontrollen kun mulighet for grunnleggende manøvrering, hvor man må styre pådraget helt manuelt i X- og Y-retning separat, i tillegg til diverse brytere og lignende.

MRs ønske var da å undersøke om det er mulig å flytte avansert funksjonalitet som ligger på hovedsystemet over på PFC-en, helst i sin helhet, men om nødvendig en nedskalert versjon. Hovedfokuset var på funksjonalitet som omhandler båtens evne til å holde en posisjon, til tross for eksterne faktorer som bølger og lignende. Funksjonen blir kalt *Station keeping*, og er en form for dynamisk posisjonering av overflatefartøyene. MRs mindre fartøy har allerede denne funksjonaliteten sikret, fordi disse kun styres av en PC, og man har da all funksjonalitet på ett og samme sted. De større USV-ene av Mariner-klassen har et skillet mellom PC og PFC, og som følge av de nåværende PFC-enes begrensninger kan man ikke ha avansert matematikk som thrust-allokering og *Station Keeping* rett på PFC-en.

1.2 Dagens løsning

Som tidligere nevnt består MR sin løsning i dag av to separate funksjoner; meldinger fra VCS til OBS til PFC til motorer, og meldinger fra en radiobasert fjernkontroll, kalt Hetronic, til PFC til motorer. Disse to funksjonene snakker ikke med hverandre, så alt av de avanserte funksjonene som ligger på OBS vil ikke kunne implementeres når Hetronicen er i bruk.

Hetronicen MR benytter i dag er basert på radiofrekvenser og har noen begrensninger, som for eksempel rekkevidde. Hetronicen brukes i hovedsak til finmanøvrering når en USV skal legge til kai. I en slik situasjon kreves det altså at en operatør med Hetronic skal være til stede. Dersom USV-en skal kjøres fra A til B manuelt, som er lengre enn det man kan se, må det være en Hetronic på begge stedene. Dersom USV-en skal styres manuelt på åpent hav må Hetronicen være i nærheten.

1.3 Ønsket løsning

Den ønskede løsningen fra MR sin side var todelt. Det var et ønske om å lage en IP-basert fjernkontroll, som kan brukes over 4G, WiFi, satelitt og lignende, uansett hvor operatøren er i verden. Det andre ønsket var å implementere OBS-styringssystemet direkte på en PFC. MR ville at gruppen skulle se nærmere på bruk av Docker på PFC-en, men ga også friheten til å utføre egne undersøkelser og velge selv. OBS på PFC vil gi en redundans hvis man skulle miste kommunikasjon mellom PC-en som vanligvis kjører OBS og PFC-en som utfører pådragene OBS ber om. Hvis man da har både OBS og utførelsen av pådrag på samme enhet, eliminerer man muligheten for kommunikasjonsbrudd mellom OBS og PFC. I tillegg vil det være nødvendig å ha et slikt type system hvis man skal gjøre større båter autonome, for å ha nødvendige sikkerhetstiltak til stede.

Hovedforskjellen på Hetronicen som brukes i dag og den som var ønsket utviklet gjennom bachelorarbeidet er hvordan disse fjernkontrollene kommuniserer med USV-en. En IP-basert fjernkontroll kan, i motsetning til en som benytter radiofrekvenssystemer, brukes hvor som helst så lenge den har internettilgang. Med en slik løsning blir rekkevidde ikke et problem og USV-en kan manøvreres manuelt fra hvor enn operatøren måtte ønske. Andre fordeler med den nye fjernkontrollen vil være at den kan benytte seg av OBS-funksjonalitet som for eksempel *Station Keeping*, noe som ikke er integrert på den eksisterende Hetronicen.

2 Teori

2.1 USV

Mariner USV er en autonom båt utviklet av Maritime Robotics AS. Den er 595 cm lang, veier 1900 kg og er designet for blant annet å kunne utføre geologiske og hydrologiske kartleggingsoppdrag, miljøovervåkning og områdepatruljering. Den er dieseldrevet, utstyrt med vannjet og baugthrustere, og kan modifiseres med tilleggsutstyr etter ønske. Mariner er designet for å være så vedlikeholdsfri som mulig, og er beregnet for operasjon både offshore, langs kysten og i innsjø. Mariner har både "Maritime Broadband Radio" (MBR) med rekkevidde på 30 km, og satelittkommunikasjon med global rekkevidde for styring (Maritime Robotics, udatert). Mariner kan styres både lokalt med Hetronicen og gjennom dataprogrammet Vehicle Control *Station Keeping* (VCS) på PC. Styringssystemet på Mariner består av en kombinasjon av PC med MRs eget ombordsystem (OBS) og ulike varianter av WAGO PFC. Det er PC-en som står for utregninger, og PFC-en som har kontroll over alt av maskinvare.

Figur 1: Mariner USV

2.2 Original systemarkitektur

Først vil systemarkitekturen slik MR hadde den før prosjektets oppstart beskrives. På USV-en har man et OBS på en PC som står for styring ved normal drift. Der ligger den avanserte funksjonaliteten, og denne PC-en er koblet videre til PFC-en, som står for det meste av I/O for systemet. PFC-en er i hovedsak en mellomstasjon

for kommandoer fra OBS. Funksjonalitet for smart-posisjonering, dvs. at USV-en ligger i ro når den ikke får beskjed om å bevege seg til tross for vær og vind, ligger på PC-en, ikke på PFC-en. I tillegg er det en ekstern fjernkontroll som er koblet til styresystemene på PFC-en, en Hetronic-fjernkontroll, som skal kunne brukes til å manuelt manøvrere USV-en så lenge den er i sikte. Denne fjernkontrollen er i hovedsak brukt som en løsning for finmanøvrering av USV-en, og en eventuell nødløsning hvis tilkoblingen mellom PC og PFC brytes mens USV-en er i sikte. Hvis man av andre grunner vil styre USV uten assistanse fra OBS-et, kan man bruke kontrolleren. Dette betyr at man bare har grunnleggende styringsmuligheter dersom man mister kontakten mellom PC og PFC. Figur 2 viser den originale systemarkitekturen.

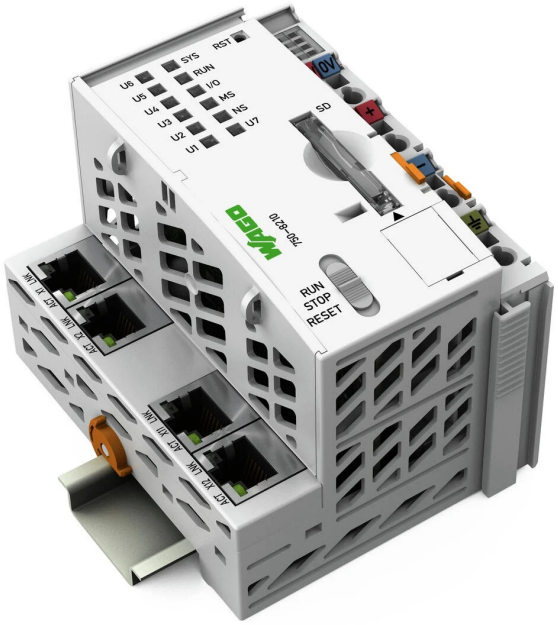
Figur 2: Den originale kommunikasjonsarkitekturen for USV og Hetronic fjernkontroll.

Denne arkitekturen har noen fordeler og noen ulemper. En klar fordel er at man, ved bruk av denne fjernkontrollen, kan ta full kontroll over USV-en, og overstyre eventuelle andre operatører som sitter på PC. Dette gjør at noen med synslinje til USV-en kan sikre at det ikke skjer ulykker og lignende. Det er også enklere å detaljstyre USV-en når man ser den, noe gruppen også selv erfarte. En svakhet med dette systemet er at man må se USV-en for å styre, siden fjernkontrollen har en rekkevidde på rundt 400 meter. Noen ganger kan det være ønskelig å kunne styre med fjernkontroll og den presisjonen det gir, selv om operatøren ikke er fysisk i nærheten av USV-en. Dette kan for eksempel være hvis man ønsker å legge USV-en til kai, siden bruk av ruteplanlegging ikke er til hjelp i det tilfelle. En annen ulempe kan være at operatøren har to joysticker på Hetronic-fjernkontrollen. Det kan være utfordrende for en uerfaren operatør. Samtidig gir dette gode muligheter for å styre nøyaktig, og man kan styre både baugthruster, vannjeten og motorens RPM med disse joystickene. I tillegg er det mange brytere på fjernkontrollen, hvor man blant annet kan skru av og på motoren, sette USV-en i forover- eller revers-kjøring og mer. Dette gir mange muligheter for en operatør, og er ønskelig funksjonalitet å ta med

videre.

Styringsmulighetene på det originale systemet er dermed to forskjellige. Man kan styre med VCS, hvor man har god oversikt over kartdata, generell posisjon og informasjon om USV-en. Den andre muligheten er å styre ved hjelp av Hetronic-fjernkontrollen. Den gir mulighet for mer detaljstyring, men man må være i nærheten av USV-en. Man har heller ikke planleggingsmulighetene VCS gir. Dessuten vil det være kostbart for kunder å eie flere -fjernkontroller, ettersom prisen er omtrent NOK 100 000 per fjernkontroll.

2.3 WAGO PFC200 Kontroller



Figur 3: Bilde av WAGO PFC200 750-8210 (Wago, udatert)

WAGO PFC200 er en programmable fieldbus controller og det var denne gruppen fikk tilgang til under bachelorarbeidet. PFC er WAGOs eget navn på PLS. MR ønsket å implementere denne versjonen av PFC på deres USV-er, for å kunne ha OBS-styringssystemet direkte på enheten som også driver det meste av styringen av USV-er. Kontrolleren er en kompakt PLS med IT funksjoner, samlet på en enhet. MRs nåværende løsning for Mariner tar i bruk en WAGO PFC 750-890. Figur 4 viser et bilde av denne PFC-en fra innsiden av en Mariner USV. Det brukes også andre PFC-er fra WAGO som ikke har støtte for Docker eller den nyeste versjonen av CODESYS.

Figur 4: Bilde av WAGO PFC 750-890 på Mariner

PFC200 kontrolleren er en form for PLS som har støtte for Docker og CODESYS V3.5. PLS-er brukes for å kontrollere diverse elektromekaniske prosesser og de kommer med flere fordeler, blant annet er de lette å programmere og operere. De er raske i utførelsen av operasjoner og inkluderer oftest flere programmeringsspråk. WAGO sin PFC200 kontroller kan programmeres i "Ladder diagram, strukturert tekst, funksjonsblokker, instruksjonsliste eller sekvensiellefunksjoner" med programmet CODESYS V3.5 (Inductive Automation, 2020). Strukturert tekst minner om programmeringsspråkene C og Pascal. Bruken av strukturert tekst gir tilgang til programmering med funksjoner, variabler og ulike datatyper. Kontrollstrukturer i strukturert tekst gjør det mulig å skrive presis og tydelig kode.

En av fordelene med WAGO PFC200 750-8210 er at den kan kombineres med høy-nivå programmeringsspråk. I tillegg har den CODESYS-basert "runtime environment" og sanntidsdyktig Linux operativsystem (Wago, 2018).

PFC100 og PFC200 seriene fra WAGO har et integrert Linux operativsystem (OS). Linux OS kommer med flere fordeler, blant annet gir den brukere muligheten til å tilpasse kildekoden for ønskede krav til program som utvikles. Linux OS er åpen kildekode, på denne måten blir det aktivt optimalisert av utviklere, i tillegg har det høy stabilitet (Wago, 2018). Bedrifter kan redusere programvare kostnader ved å

bruke Linux OS, ettersom det er gratis, og eliminerer kostnadene for lisensiering av OS. Høy stabilitet er alltid en fordel for utvikling av programmer og andre systemer. En annen fordel med Linux OS er at den er kjent for å ha høy sikkerhet, og sjeldent utsettes for skadevare eller virus. PFC-er drar nytte av å ha Linux OS på flere måter, spesielt ved at Linux systemet kan fullstendig tilpasses ønskelig operativt bruk. Økning i teknologisk utvikling gir økning i cybertrusler, med Linux OS på PFC-en økes sikkerheten, noe som er kritisk for fremtiden og programvarens sikkerhet (WAGO, udatert).

2.4 Kommunikasjon

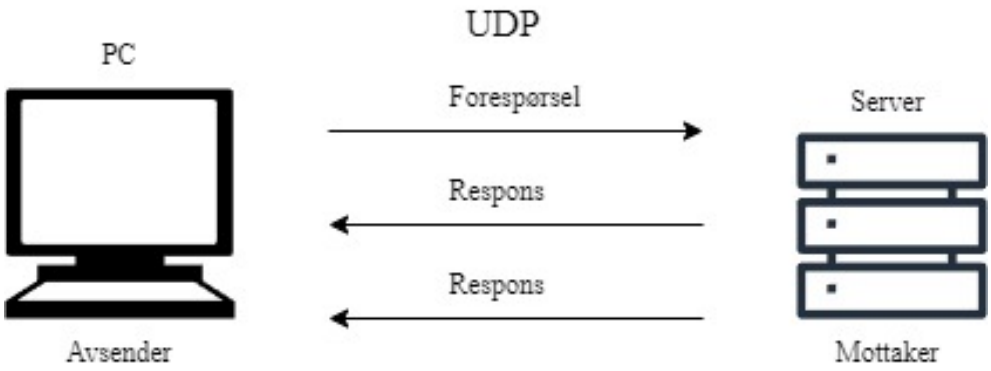
2.4.1 IP

Internettprotokoll (IP) er en verdensomfattende protokoll som gjør det mulig å sende data fra en enhet til en annen over internett. En enhet kan være hva som helst som er koblet opp mot internettet, slik som en PC, mobil eller RPi. Alle disse enhetene har hver sin IP-adresse som ikke er lik noen andres (Kerner, 2021). Et eksempel på en slik adresse kan være 192.168.1.192. Siden alle IP-adresser er unike, fungerer adressen som et fingeravtrykk, og kan brukes til å identifisere en enhet. Mye likt som at en gateadresse og husnummer kan identifisere en lokasjon for et hus, kan en IP-adresse identifisere en enhet på internett. IP er en del av "internett laget" i et kommunikasjonssystem. Dette laget er forbindelsesløs, som betyr at det ikke er noen kontinuerlig forbindelse mellom enhetene som kommuniserer. For å kunne opprettholde en forbindelse mellom to enheter, må man benytte en protokoll fra transport laget. TCP og UDP er eksempler på protokoller som ligger på dette laget (GeeksforGeeks, 2023a).

2.4.2 UDP-protokoll

UDP er en nettverskprotokoll som brukes til rask og enkel kommunikasjon over et IP-basert nettverk. Nettverksprotokoll er en felles betegnelse på sett med regler som styrer datakommunikasjon mellom ulike enheter i et nettverk. Den valgte protokollen bestemmer hva som blir kommunisert, hvordan det blir kommunisert, og når det blir kommunisert. Den tillater forskjellige tilkoblede enheter å kommunisere med hverandre, uavhengig av interne og strukturelle forskjeller (GeeksforGeeks, 2023b). UDP-protokollen gir ingen garanti for at datapakken blir levert og krever heller ingen bekreftelse fra mottaker. Dette gjør protokollen mindre pålitelig, men til gjengjeld veldig effektiv og rask. UDP blir ofte brukt i tidskritiske applikasjoner som kan akseptere noe tap av data. Det er ikke ofte det går tap av data, men siden det

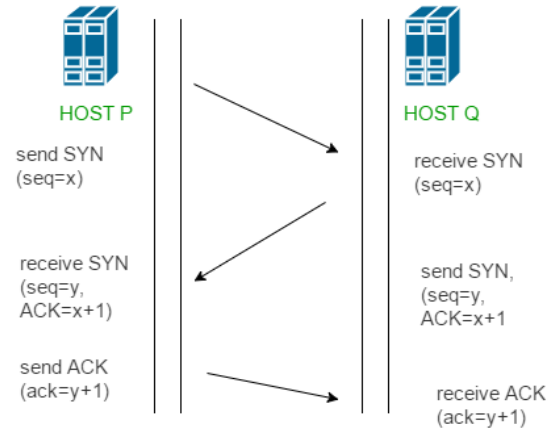
ikke kreves noen bekreftelse fra mottaker, er en aldri helt sikker på om alle pakkene mottas (Wikipedia, 2023).



Figur 5: Illustrasjon av UDP kommunikasjon mellom en avsender og mottaker.

2.4.3 TCP-protokoll

TCP er en nettverksprotokoll som baserer seg på å være tilkoblingsorientert. Dette betyr at de enhetene som skal kommunisere sammen må etablere en forbindelse for overføring av data, og bør lukke denne forbindelsen etter overføring av data er gjennomført (Palak Jain 5, 2022). Dette gjør at en TCP tilkobling er mer pålitelig og sikrere på at dataen kun kommer til mottakeren. Dette gjøres ved å benytte en "treveis håndtrykkprosess". Dette "håndtrykket" må gjennomføres hver gang det ønskes ny kommunikasjon mellom to enheter. Det er en protokoll som oftest brukes i det daglige, siden den er tilstrekkelig pålitelig. Sending av e-post og surfing på internett er eksempler på bruksområder (Wikipedia, udatert[g]).



Figur 6: Illustrasjon av hvordan TCP 3-veis håndtrykkprosessen fungerer (GeeksforGeeks, 2021)

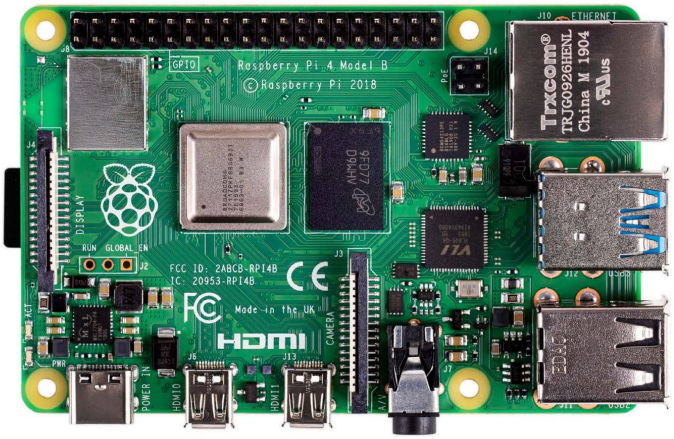
2.5 Programvare og maskinvare

2.5.1 Linux

Linux er et fritt og åpent operativsystem som ble opprettet av Linus Torvalds i 1991. Det er en åpen og gratis kildekode som kan endres og distribueres til hvem som helst både kommersielt og ikke-kommersielt under "GNU General Public License". I dag brukes Linux ofte i innebygde systemer som rutere, automatiseringskontroller, videospillkonsoller, smartklokker osv. Linuxkjernen i seg selv er ikke et operativsystem, men utgjør grunnsteinen som forskjellige operativsystemer kan bygges på, som for eksempel Ubuntu eller Raspberry Pi OS (ankita_saini, 2021).

2.5.2 RPi

Raspberry Pi er en rimelig datamaskin bygget på et enkelt kretskort på størrelse med et kredittkort som enkelt kan kobles til en dataskjerm eller TV. Den bruker Raspberry Pi OS som er Linux basert, og er derfor et fritt og åpent operativsystem. Raspberry Pi 4 modell B som blir brukt av gruppen under prosjektet ble utgitt i juni 2019, som i skrivende stund er den kraftigste og raskeste RPi laget, og kan egne seg som en vanlig PC (Raspberry Pi Foundation, udatert).



Figur 7: Illustrasjon av Raspberry Pi 4B (Singh, 2021).

2.5.3 Make

Make er et byggautomatiseringsverktøy som brukes i programvareutvikling for å bygge kjørbare programmer fra større prosjekter, med alle avhengigheter. Make

klarer å bygge dette fra kildekode ved hjelp av å lese makefilen, som gir verktøyet instruksjoner. Make-filer lager et sett med regler for hvilke deler av et program som må kompileres, og gir make instruksjon om å kompilere delene av programmet (Wikipedia, udatert[c]). Å skrive en egen makefil er veldig komplisert og krever stor kodeforståelse og en del erfaring. For å gjøre denne prosessen enklere finnes det programvare som generer Make-filer, CMake og QMake er blant disse.

2.5.4 CMake

CMake er en plattform-uavhengig programvare for byggautomatisering som brukes i programvareutvikling. Den er ikke i seg selv et byggesystem, men generer byggefiler for make-systemet. CMake kan brukes for å unngå at utviklere må skrive egne Make-filer til programmene de lager. Programvaren kan også brukes til testing, pakking og installasjon programvare (Wikipedia, udatert[a]).

2.5.5 Qmake

Qmake er på lik måte med CMake en programvare som automatiserer genereringen av Make-filer. Den ble utviklet av "Qt Project", som også utviklet programmet "QT Creator". Makefilene som genereres av QMake er plattformspesifikke, som betyr at de kun kan benyttes på den aktuelle plattformen de er laget på. Disse makefilene er basert på prosjektets QMake prosjektfiler (Wikipedia, udatert[d]). Qt er et programvarebibliotek bestående av flere programmer som brukes for utvikling av kode, som kan kompileres til å kjøre på ulike plattformer og OS, for eksempel macOS og Linux. "QT Creator" er det integrerte utviklingsmiljøet som brukes av utviklere for å lage programvare som kjører på ulike plattformer (Wikipedia, udatert[f]). Utviklingsmiljøet har god struktur og gir utvikleren god oversikt over egne prosjekt, "QT Creator" tilhører programvarebiblioteket Qt (Wikipedia, udatert[e]).

2.5.6 Make, CMake eller QMake

Programvaren CMake er mye raskere på å bygge filer enn Make, i tillegg til at den har støtte for flere verktøy, noe som gjør CMake veldig brukervennlig. Et Make bygg vil "glemme" hvordan det ble bygd, den husker ikke på hvilken kilde den stammer fra eller hvilke kompileringsmetoder som ble brukt. En av fordelene med å bruke CMake som utvikler er at CMake sporer denne informasjonen, slik at det er mulig å ha oversikt over hva som har blitt gjort. Ved bruk av Make må en lage denne oversikten selv. CMake bruker avhengighetene mellom forskjellige "mål" for bygging,

hvor et "mål" er en enkel output-fil. CMake vil administrere mappestrukturen for utvikleren, mens med Make må alt gjøres av utvikleren selv (Wikipedia, udatert[a]).

Både CMake og QMake er nyttige automatiseringsverktøy, og hvis man benytter Qt programvaren vil QMake være bedre egnet. Hvis man ønsker å bruke CMake eller make vil denne prosessen være mer innviklet. Dette er fordi QMake er tilpasset bruk på Qt prosjekter. Utenfor et Qt prosjekt, er mulighetene med QMake begrenset i forhold til CMake. CMake er industristandarden for C++, men har dårligere syntaks enn QMake. En av CMakes fordeler er at den har støtte for flere biblioteker og tredjeparts bibliotek, noe som er manglende i QMake. Dette kan være veldig nyttig som utvikler, ettersom tredjeparts bibliotek kan være nødvendig for et prosjekt. Den største forskjellen mellom CMake og QMake er at CMake kan brukes til mye mer enn QMake, hvis man lærer det godt, selvom QMake er enklere å bruke.

2.5.7 C++

C++ er et objektorientert programmeringsspråk, laget for mange bruksområder. Disse inkluderer operativsystemprogrammering, visuelle brukergrensesnitt og skytjenester. I tillegg er det en stor mengde biblioteker tilgjengelig, som man kan benytte seg av i utviklingsprosessen (Lenka, 2023). MR sitt OBS er skrevet i C++, så dette språket måtte benyttes for å kunne fullføre integrasjon av den egenutviklede fjernkontrollmodulen med MRs egne systemer. Dette inkluderer også bruk av programmet QT Creator. Dette er en IDE, "Integrated Development Environment", som også inkluderer funksjonalitet for make-filer. Dette er filer som benyttes for å enkelt kompilere store prosjekter, da filstrukturen kan bli stor og kompleks etterhvert som filer med funksjoner og klasser legges til. Make-filen vil da bygge prosjektet i riktig rekkefølge, litt som man bygger grunnmur før man bygger taket på et hus. I tillegg vil bare filer som er endret siden forrige kompilering, og filer som er avhenging av endrede filer kompileres (gurukiranx, 2021). Dette sparer en del tid hvis man hyppig tester ut nye funksjoner og tillegg til programmet. Denne make-funksjonaliteten var allerede i bruk hos MR, men var uansett i bunn og grunn en nødvendighet, da OBS-et inneholder for mange filer til å enkelt kompilere ved hjelp av kommandolinjen.

Figur 8: C++ logo (Wikimedia Commons, 2017)

Et godt åpen kildekode byggesystem for C++ er CMake, den kan brukes på tvers av ulike plattformer og brukes i prosjektet for å bygge, teste og pakke sammen programvare. CMake støtter applikasjoner som er avhengige av mange biblioteker. Den brukes i forbindelse med byggemiljøer som Make og tidligere nevnte QT Creator. For C++ er Make en av de beste byggesystemene, CMake er etterfølgeren til dette byggesystemet (Exterman, 2021). Qmake, standard byggeinstruksjons generatoren til QT Creator, er også et godt alternativ.

2.5.8 Codesys

Codesys er et integrert utviklingsmiljø og en programvareplatform for utvikling av industriell automasjons-teknologi. Et av bruksområdene til Codesys er programmering av PLS-er og I/O-kontrollere. Det er mulig å skrive programmer i flere ulike programmeringspråk, deriblant IEC 61131-3 standarden, som definerer tre grafiske og to tekstbaserte programmeringspråk, for eksempel ladder og strukturert tekst (CODESYS Group, 2023).

2.5.9 Protobuf

Protocol Buffer (Protobuf) er Google sitt eget verktøy for bruk av structs på forskjellige programmeringsspråk og plattformer. Det lages en egen .proto fil hvor structen deklareres, og kan enkelt implementeres inn i egen kode. Dette medfører en enklere måte å overføre data mellom to enheter og forskjellige programmer, for eksempel over UDP. Protobuf er gratis og en åpen kildekode som alle har rettigheter til å bruke (Google LLC, udatert). Når en .proto fil kompileres, genereres det to nye filer som kan brukes opp mot egen kode. En fil blir av typen .pb.cc, som må legges ved når hovedkoden kompileres, og en .pb.h fil som må inkluderes som et bibliotek inne i hovedkoden.

2.5.10 VCS

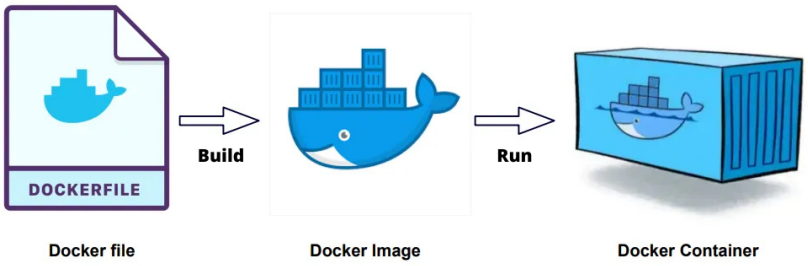
VCS er MRs brukergrensesnitt for kommunikasjon med deres USV-er. Der kan man se kart og posisjonsdata, motor og styringsdata, hastighet og retning. I tillegg kan man styre diverse brytere på USV-en, for eksempel for lys, vinsj og målesensorer. I tillegg kan man bytte mellom forskjellige moduser som manuell styring, *Station Keeping*, ruteplanlegging og kurs. I tillegg kan dette programmet kobles opp mot en USV-simulator som MR bruker internt.

2.5.11 OBS

OBS er styringssystemet som kontrollerer MR sine USV-er. Dette er selve programmet som tar imot kommandoer og gjør beregninger for hvordan USV-en skal bevege seg. Deretter sender OBS-et disse kommandoene til motor og lignende, slik at USV-en beveger seg. Dette programmet styrer også tilleggsutstyr og kommunikasjon til og fra USV-en. Kort sagt er OBS i kontroll av alt som skjer på USV-en.

2.6 Docker

Docker er et program som skal bidra til plattformuavhenging kodeutvikling og kompilering, som skal muliggjøre bruk av samme program på forskjellige typer maskiner. Dette gjøres ved at man først konstruerer en Dockerfil, som er en slags mal for hva man ønsker å gjøre med et sett med filer. Deretter bygges denne Dockerfilen, og gir da et Docker Image. Dette er en samling av alle ønskede filer, i tillegg til eventuelle biblioteker og andre avhengigheter, som kan lastes ned på enhetene Imaget skal benyttes på. Når man så kjører dette Docker Imaget vil man få en Docker Container, som da er programmet som kjører. Docker har et godt oppslagsverk som kalles Docker Reference, hvor alt av Dockers funksjonalitet er forklart. Dette gjør Docker til et godt verktøy for både nybegynnere og erfarne. Man har tilgang til gode råd for optimalisering av Dockerfil, Image og Container, både for hastighet og plassbruk. Det er altså tre steg for å benytte seg av et program gjennom Docker, og disse vil utdypes i de følgende avsnittene.



Figur 9: Illustrasjon av hvordan de forskjellige delene av Docker henger sammen (Bikram, 2021).

2.6.1 Dockerfilen

Som tidligere nevnt benytter Docker en Dockerfil for å klargjøre en mal for kjøring av et program på ulike datamaskiner. Denne filen brukes til å spesifisere hva man vil inkludere i Docker Imaget. Dette kan for eksempel være filer med kildekode,

ferdige programmer eller andre typer filer, her er det få begrensninger. Man kan også spesifisere forskjellige betingelser som må utføres for at programmet skal kjøre som ønsket, deriblant OS, versjon av dette og installerte biblioteker og programmer. Dette kan tenkes på som et kart den som koder lager, som Docker trenger for å komme seg frem til målet, at en Docker Container kjører, og inneholder alt brukeren trenger. Et grunnleggende eksempel på en Dockerfil er vist i kodeutdrag 1.

```
FROM gcc:4.9
COPY . <path-to-file>
WORKDIR <path-to-file>
RUN g++ -o testprogram testprogram.cpp
CMD ["./testprogram"]
```

Kodeutdrag 1: Grunnlegende Dockerfil

Linje for linje gjør Dockerfilen følgende:

- **FROM** spesifiserer et "base" Image man skal hente fra, i dette tilfellet GCC for C++ med versjon 4.9.
- **COPY** forteller Docker hvilken filplassering man skal kopiere fra. Her blir det fra mappen Dockerfilen ligger i, definert som ".", til en mappe i Containeren som brukeren selv kan bestemme, f.eks. /root/home/docker.
- **WORKDIR** forteller Docker hva som er arbeidsmappen inne i Docker Containerens filstruktur.
- **RUN** kjører en kompileringskommando, her for å kompilere .cpp (C++) filen til en "executable", en eksekveringsfil. Ekseveringsfil er også kjent som binærfil.
- **CMD** kjører eksekveringsfilen, som betyr at programmet kjøres.

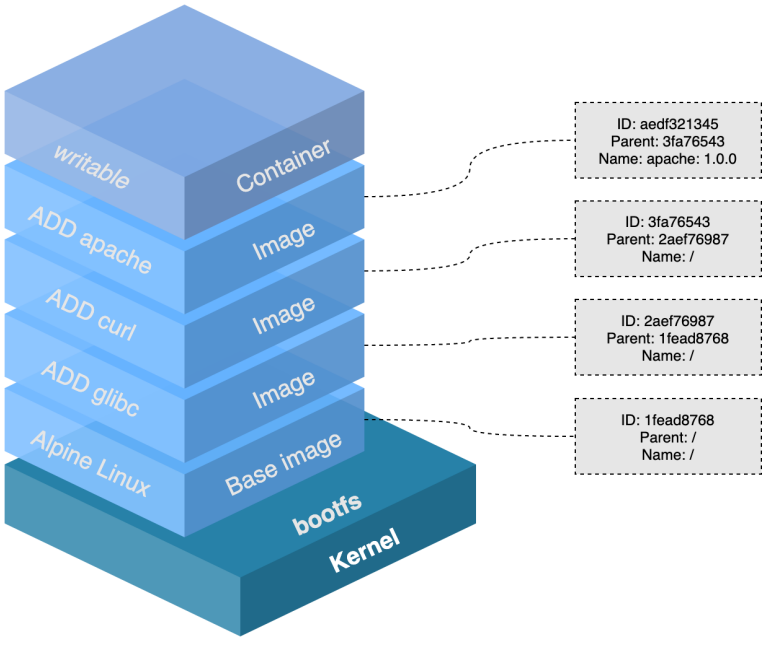
Dette eksempelet er en mulig måte å skrive en Dockerfil, men måten dette gjøres på varierer fra programmeringsspråk til programmeringsspråk, i tillegg til det spesifikke programmets krav. Kort fortalt gis Docker beskjed om akkurat hva den skal gjøre, og så stiller den opp med alle eventuelle tilpasninger for den aktuelle plattformen programmet skal kjøres på.

Docker leser instruksjonene i en Dockerfil for å bygge Docker Imaget automatisk (Docker Inc, udatert[c]). Ved hjelp av programmer som Visual Studio Code kan man programmere instruksjonene i en Dockerfil, for å bygge et Docker Image. Når man lager en Dockerfil er det viktig å spesifisere navnet på filen som "Dockerfile", andre navn vil ikke være gyldige. I et prosjekt er det ikke mulig å ha mer enn en

Dockerfil, det vil si at alt i prosjektet må gjøres ved hjelp av én Dockerfil. På denne måten unngår man at instruksjonene til programmet blir delt opp, istedet får man god oversikt over alt av instruksjonene til Imaget man ønsker å lage. Dockerfilen vil som følge ikke kunne samarbeide med andre Dockerfiler.

2.6.2 Docker Image

Et Docker Image er altså en kompilert Dockerfil, og inneholder flere forskjellige lag med informasjon. Hvert lag har en egen ID, som identifiserer innholdet i det. Imaget inneholder alle de nødvendige instruksjonene som kreves for å kjøre en Container på støttede plattformer. Hvert nye lag blir lagt over det foregående laget, og legger supplerende informasjon oppå. Laget som blir lagt over, er avhengig av laget som ligger direkte under det for å fungere. Effektiviteten til Docker Imaget ligger i hierarkiet av lagene, og størrelsen på hvert av lagene. Ved å organisere lagene etter endringshyppighet, det vil si at lagene som det endres mest på bør ligge høyere opp i hierarkiet, vil endringer på Imaget kreve mindre tid og prosesseringskraft for gjenoppbygge Imaget, enn om man endrer et av de lavere lagene (Kisller, 2023). Figur 10 viser hvordan oppbyggningen av et Docker Image kan se ut.



Figur 10: Visuell beskrivelse av oppbygningen av et Docker Image med flere lag og eksempler på inneholdet i hvert lag (Flade, 2020)

Her ser man altså en illustrasjon av lagmodellen Docker benytter seg av, med en vertikal oppbyggning. Docker Images kan lagres både lokalt på en enhet, og i skytjenesten Docker Hub, hvor man så kan hente ned Images på alle kompatible enheter over internett.

2.6.3 Docker Container

Et Docker Image er tilpasset til den aktuelle plattformen det lastes ned på, men programmet kjører ikke automatisk. For å kjøre programmet, bruker man Dockerkommandoen "docker run", som kjører Imaget i form av en Container. Docker Containere kan tolkes som veldig raske mikrodatamaskiner, men det spesielle med Docker Containere i forhold til normale datamaskinprogrammer, er at de er helt isolerte fra vertsmaskinen, sikre og lite ressurskrevende (Docker Inc, udatert[e]). Fra Containerens perspektiv er den en egen datamaskin. En Docker Containere vil ha like egenskaper som OS-et det er bygd fra, for eksempel hvis en container bruker et Linux Base Image. Det vil si at man navigerer og skriver på lik måte som man ville gjort på terminalvinduet til OS-et. Man kan altså skrive kommandoer i Containeren, dersom de nødvendige avhengighetene er installert.

Et Image skaffer det tilpassede filsystemet som inneholder alt som kreves for å kjøre applikasjoner. En av de største fordelene med Docker Containere i industrien er at når en utvikler skriver kode, kan denne implementeres i Docker og deretter vet man at koden fungerer optimalt overalt, så lenge plattformen er tatt hensyn til i utviklingsprosessen. Dette forenkler utviklingsprosessen hos bedriften og gjør den mer effektiv. Det er viktig å sørge for at alt i Containeren støtter kjøring på arkitekturen til den ønskede maskinen. Dette kan for eksempel være kodekompilatorer som GCC, eller biblioteker som *net-tools*.

2.6.4 Dockerkommandoer

Docker Containere startes ved hjelp av ulike Dockerkommandoer, som instruerer Containeren til å starte med forskjellige egenskaper. For å bruke Dockerkommandoer må Docker først installeres på maskinen. Når Docker er tilgjengelig på maskinen kan man skrive "docker" etterfulgt av ulike kommandoer, parametere og til slutt navnet på Imaget man vil kjøre. For å få en oversikt over eksisterende kommandoer kan man skrive "docker help" (Docker Inc, udatert[f]). Blant de mest brukte parametrene er "docker build" og "docker run". Den førstnevnte brukes for å bygge Dockerfilen, og lage Docker Imaget. Den sistnevnte brukes for å lage og kjøre en ny Container fra Imaget som tidligere ble laget. "Docker reference" spesifiserer hvordan de ulike kommandoene skal brukes (Docker Inc, udatert[b]).

```
$ docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

Kodeutdrag 2: Dockerkommando

Docker parametrene har ulike flagg med alternativer som tilpasser hva kommandoen

gjør. For eksempel kan man ved bruk av "docker run" legge til alternativene "–rm, –platform og –name". For å automatisk fjerne Containeren hvis den eksisterer kan man bruke "–rm". Plattformen man ønsker å kjøre Imaget på angis med "–platform". Hvis PC-en kjører på en AMD-basert CPU og Linux ARM plattformen spesifiseres, vil ikke PC-en med AMD kunne kjøre Containeren til Imaget. Dette er nyttig dersom man ønsker å kjøre Docker Containeren på en maskin med annen arkitekturen enn den tilhørende maskinen hvor Imaget ble bygd. Kommandoen "–name", spesifiserer navnet som angis til Docker Containeren når den kjøres (Docker Inc, udatert[b]). Det er mange ulike Docker parametere som kan ta i bruk flere forskjellige flagg, derfor er det viktig å sette seg godt inn i Docker dokumentasjonen tilgjengelig. Det finnes i tillegg kommandoer for å laste opp til og hente fra Docker Hub.

2.6.5 Docker Nettverk

Når det kommer til kommunikasjon inn og ut av en Docker Container benyttes Docker-nettverk. Dette gjøres ved at enten Docker eller brukeren selv definerer et nettverk, som betyr at hver Container har en mulig "veg" den kan sende informasjon på. Det kan opprettes flere slike nettverk, og man kan derfor isolere Containere på forskjellige nettverk. For eksempel hvis man vil at noen nettverk skal kunne snakke sammen, men ikke andre, eller at en Container skal være uten nettverkstilgang (Choudhary, 2021). Dette er nyttig i storskala bruk av Containere, da man kan få god oversikt i informasjonsstrømmene på de forskjellige nettverkene. I tillegg er det nyttig for å sikre at forskjellige Containere faktisk har tilgang til hverandres informasjon. Det er nyttig hvis man har mange forskjellige programmer som jobber sammen for å utføre en større oppgave.

2.6.6 Emulering av plattform

Docker gjør det mulig å bygge multiplattform Images. Programmet gjør det enklere å utvikle Containere på og for ulike enheter, med diverse arkitekturer. Docker sine offisielle Images har støtte for en mengde forskjellige arkitekturer. Det er også mulig å kjøre en Container tilrettelagt for flere plattformer samtidig. Docker har tre ulike metoder for å bygge multiplattform Images, med støtte av Dockerfilen (Docker Inc, udatert[d]), og en metode vil utdypes.

Figur 11: QEMU logo (Wikipedia, 2012)

Denne metoden er å bruke QEMU emulering. QEMU er en rask maskinemulator med åpen kildekode, og brukes for å emulere en mengde maskinvarearkitekturer. Med QEMU kan man kjøre et annet OS på toppen av maskinens system (SUSE, udatert). Docker har integrert støtte for QEMU. QEMU-metoden gjør det enklelt å kjøre ønsket platform på maskinen. "docker buildx build –platform [...]" er kommandoen som brukes dersom man ønsker å kjøre en Docker Container med flere plattformer spesifisert. For å bruke denne kommandoen for multiplattform Docker Images må QEMU installeres. En stor fordel med å bruke QEMU metoden er at utvikling av program på PC, antageligvis er raskere enn på maskiner med tregere prosessorer, selv om man må emulere. Man vil på denne måten spare tid på bygging, testing og kjøring av Docker Containere for flere plattformer.

2.6.7 Effektivisering og optimalisering

Det finnes forskjellige grep man kan ta for å optimalisere Docker Containere. Et Docker Image kan bygges på grunnlag av et "Base Image". "Base Imaget" brukes for å danne alle bildene i en Container. Det finnes ulike måter å sette opp et "Base Image". Docker har offisielle images som kan settes til bruk som "Base Image", for eks. Ubuntu, Debian eller Alpine. En annen løsning er å lage et "Base Image" fra bunnen av, eller gjøre endringer på ett av de offisielle bildene (IBM, 2021). For å optimalisere et Docker Image, er valg av "Base Image", veldig avgjørende for Image størrelsen (Wilson, B. og Khandelwal, S., 2021). En annen optimaliseringsmetode innbærer å bygge Images med en flertrinnsmodell. Denne eliminerer uønskede lag i Docker imaget, som igjen fører til at Imaget blir mindre i størrelse. Å minimere antall lag i et Docker Image, fører også til minimering av størrelsen på Image som bygges, og er en enkel og effektiv måte å optimalisere Images på (Wilson, B. og Khandelwal, S., 2021). Selv om kodesnuttene fra Dockerfilene i kodeutdrag 3 og 4 gjør det samme, viser kodeutdrag 4 et litt mindre Image.

Kodeutdrag 3 viser et eksempel på en Dockerfil uten lagoptimalisering, hvor forskjellige pakker installeres.

```
FROM raspbian:buster
RUN apt-get update
RUN apt-get upgrade
RUN apt-get install -y wget gnupg2 ccache devscripts equivs
RUN apt-get install -y project
RUN apt-get install -y net-tools
```

Kodeutdrag 3: Dockerfil uten lagoptimalisering

I kodeutdrag 3 kan man se at det er mange linjer som kjører samme RUN-kommando. Disse kan samles til en linje, som da blir ett enkelt lag, kontra de 5 lagene det er originalt. Løsningen vises i kodeutdrag 4.

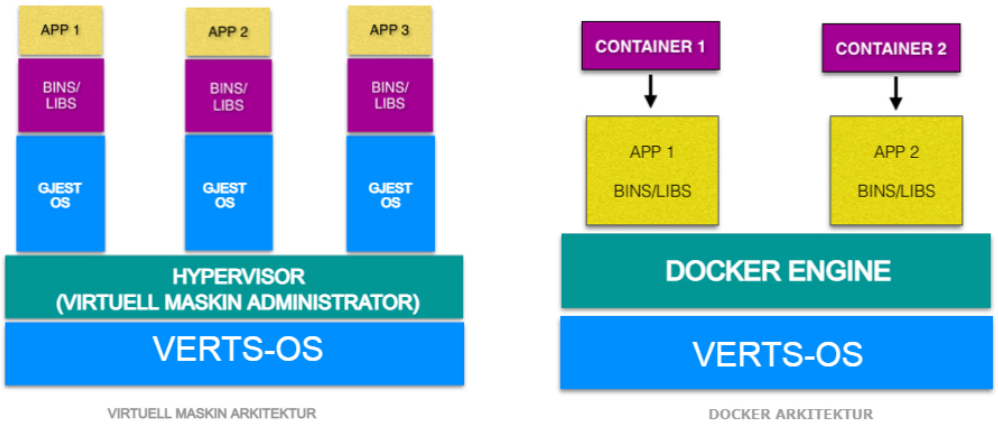
```
FROM raspbian:buster

RUN apt-get update && apt-get upgrade && apt-get install wget
    gnupg2 ccache devscripts equivs && apt-get install project &&
    apt-get install net-tools -y
```

Kodeutdrag 4: Dockerfil med lagoptimalisering

Det er fordeler og ulemper med å samle flere kommandoer til ett lag. Det kan argumenteres at det er mindre lesbart, ettersom man skriver mye forskjellig rett etter hverandre. Samtidig vil en slik samling redusere størrelsen av Docker Imaget noe og gjøre Dockerfilen mer kompakt. I tillegg skal ikke en Dockerfil leses av andre enn den som skriver den, senere skal man kunne laste den Docker Imaget laget av Dockerfilen, og bruke den, uten å vite hvordan Dockerfilen er bygget opp.

2.6.8 Docker og virtuell maskin

Ved installasjon opprettes det en Docker Engine som kobles på OS på datamaskinen. OS styrer datamaskinens prosesser, minne, all programvare og maskinvare, i tillegg til at den håndterer koordinering av ressurser. Alle behovene til applikasjonene på datamaskinen dekkes ved hjelp av OS (Cegal, udatert). En virtuell maskin er et selvstendig driftsmiljø, uten tilgang til sin vertsmaskin, og oppfører seg som en egen datamaskin (Dictionary.com, udatert). Docker er forskjellig fra en virtuell maskin, ved at Docker ikke legger et helt eget OS på toppen av maskinens OS, men heller bruker Docker Engine til å kjøre programmene (Arora, 2022). Dette er vist i figur 12.



Figur 12: Visuell beskrivelse av forskjellen mellom virtuelle maskiners og Dockers arkitektur (Arora, 2022)

I figur 12 kan man se at Docker krever færre lag oppå vertsmaskinens OS enn det virtuelle maskiner krever, som fører til mindre ressursbruk og krav til vertsmaskinen. En viktig forskjell er at Docker Containere bruker vertsmaskinens operativsystemkjerne, kalt "kernel" på engelsk, mens virtuelle maskiner har sine egne operativsystemkjerner (Kasireddy, 2016). Denne kjernen er det som styrer ressursallokering for en maskin, det vil si hvilke prosesser som får bruke hvilke ressurser, både når, hvor lenge og hvor mye (pp_pankaj, 2022). Det at en virtuell maskin har egen kjerne, gjør at det blir enda et lag med ressursallokering på toppen av vertsmaskinens egen allokering. Det vil kreve mer ressurser og derfor vil det være igjen færre ressurser til de oppgavene som skal gjøres av andre prosesser. På grunn av denne forskjellen er Docker bedre egnet til enheter med relativt lav prosessorkraft, for eksempel PLS-er. Dette gjør det også fordelaktig å bruke i en slik setting, da det går raskt å sette opp. I tillegg kreves det lite arbeid for å lage en ny versjon av programvaren og deretter sende den inn på en enhet. Dette er forskjellig fra en virtuell maskin der all kode er OS-avhengig, og må tilpasses hvis OS-et er annerledes fra en enhet til en annen. Docker gjør mye av dette arbeidet for brukeren.

3 Metode

I denne seksjonen vil arbeidet som har blitt utført under bacheloroppgaven utdypes. Det innebærer alt fra planlegging av den nye systemarkitekturen, til fungerende program implementert på PFC-en og testen av denne funksjonen på en USV.

3.1 Implementert funksjonalitet

Denne seksjonen vil kort forklare hva slags funksjonalitet som er implementert for joystick, OBS og kommunikasjonen mellom disse.

- Hente inn x, y og z data fra joystick til RPi.
- Skalere dataene, deretter legge disse inn i en struktur og sende den over UDP til en port på enheten som kjører OBS.
- OBS leser denne UDP meldingen, og oversetter og legger dataene tilbake i en tilsvarende struktur som er på sendersiden, dvs. på RPi-en.
- OBS henter ut verdiene den ønsker, og sender disse inn i MRs funksjoner for å styre USV.
- OBS sender tilbake informasjon om fart, RPM på motor og lignende, til RPi-en, slik at man eventuelt kan se denne informasjonen på fjernkontrollsiden.
- OBS registrerer kommunikasjonstap og setter USV i *Station Keeping*, dvs. at den holder posisjonen, inntil kommunikasjon er gjenopprettet.
- OBS har mulighet til at bruker selv setter USV i *Station Keeping*.
- RPi har LED-dioder som viser forskjellige alarmer og om fjernkontrollen er i kontroll av USV-en.

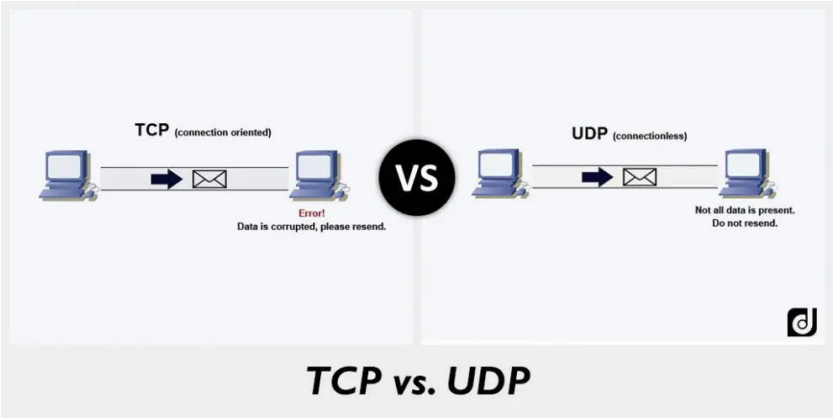
3.2 Kommunikasjonsoppsett

På begynnelsen av prosjektet ble kommunikasjon over Ethernet og USB valgt. Det er flere grunner til dette, viktigst av alt var at MR allerede benyttet seg av Ethernet til mye av kommunikasjonen sin. Dette er en kablet forbindelse, som sikrer mer pålitelig kommunikasjon enn trådløs kommunikasjon, nettopp fordi man har en maskinvare-kobling mellom enheter. Dette gjør utvikling lettere, fordi man reduserer muligheten for tap og feil som følge av at enhetene kommuniserer trådløst. Senere, både for

simulering og test på USV, ble trådløs kommunikasjon benyttet mellom RPi og OBS, enten OBS ble kjørt på Mariner, PC eller PFC. Dette er fordi det enten var nødvendig, som i Mariners tilfelle, eller nyttig for å se hvordan systemet taklet dette kommunikasjonsmediet.

3.2.1 Valg av kommunikasjonsprotokoll

Når det kommer til valg av kommunikasjonsprotokoll stod det mellom de to standardene UDP og TCP. Det var et større fokus på å få sendt over datapakkene i sanntid, raskest mulig og ofte. Siden UDP er raskere, enklere og mer effektivt enn TCP, falt valget på å bruke UDP. I tillegg var meldingene som skulle sendes korte, det vil si at man klarte å sende all informasjonen i en UDP-pakke, og ikke måtte dele opp. Hvis man måtte dele opp meldingen i flere UDP-pakker, ville det vært mer hensiktsmessig å bruke TCP. Dette er fordi TCP gir muligheten til sikre at pakker sendes i riktig rekkefølge (Palak Jain 5, 2022), og man derfor vil sikre at meldingen settes sammen i riktig format. Dette er ganske essensielt når man styrer et fysisk fartøy, da en ødelagt melding har potensialet til å forårsake alvorlig skade. Dette kan eksempelvis hende hvis meldingen, som i dette tilfellet sier noe om hastighet, blir tolket som en mye høyere verdi, og fartøyet setter avgårde i høy hastighet. Fartøyet kan da potensielt treffe noe eller noen før man klarer å få tilbake kontrollen, og ødelegge seg selv, andre gjenstander eller skade mennesker og annet liv.



Figur 13: Illustrasjon av en simplifisert forskjell mellom TCP og UDP (Taylor, 2020)

3.2.2 RPi og joystick

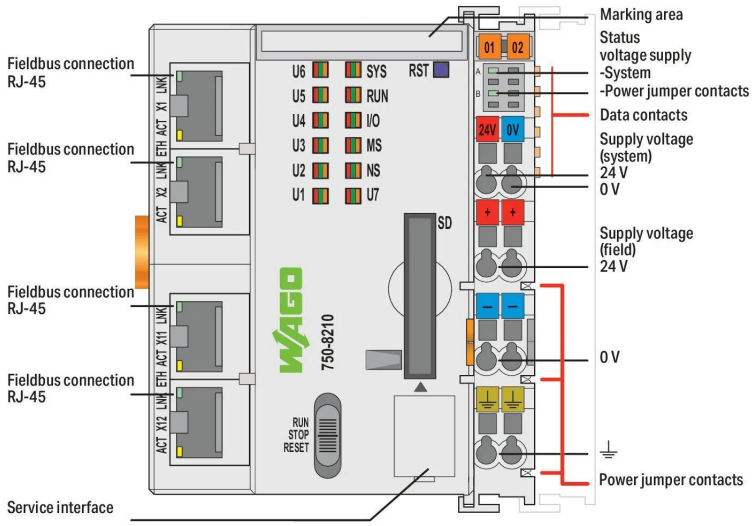
Kommunikasjonen mellom RPi og joystick skjer over USB. Joysticken hadde kun mulighet for kommunikasjon over USB, så dette satt en automatisk begrensning på mulighetene for forskjellige typer kommunikasjon. Det ble laget en kode i C++ på RPi-en som tok imot signalene fra joysticken og sendte dem videre til PLS-en.

3.2.3 RPi og PFC

For å kunne sende inputsignalene fra joysticken til PFC-en, ble det lagd en C++ kode som kan kommunisere med PFC-en over en Ethernet-kabel, ved å bruke UDP protokollen. Styringssystemet er et tidskritisk system, hvor forsinkelser i dataoverføring kan medføre problemer, derfor ble UDP-protokollen valgt fremfor TCP. Utover i prosjektgjennomførelsen kommuniserte RPi og PFC-en over 4G.

3.3 PFC

Gruppen fikk tilgang til en PFC-rigg med en WAGO PFC200 750-8210 for å løse oppgaven. Denne serien av WAGO sine PFC-er kommer med Docker ferdig installert. Det å kunne bruke Docker på PFC-en, gjør det mulig å implementere og kjøre MR sitt OBS på en USV. Før Docker Containeren kunne implementeres måtte PFC-en settes opp ved å installere den nyeste programvaren. Dette måtte gjøres for at kommunikasjon til og fra PFC-en ble feilfri, via Ethernet og DHCP (Dynamic Host Configuration Protocol). I tillegg måtte CODESYS V3.5 installeres og gruppen måtte sette seg inn i hvordan man manøvrerer rundt inne på PFC-en.



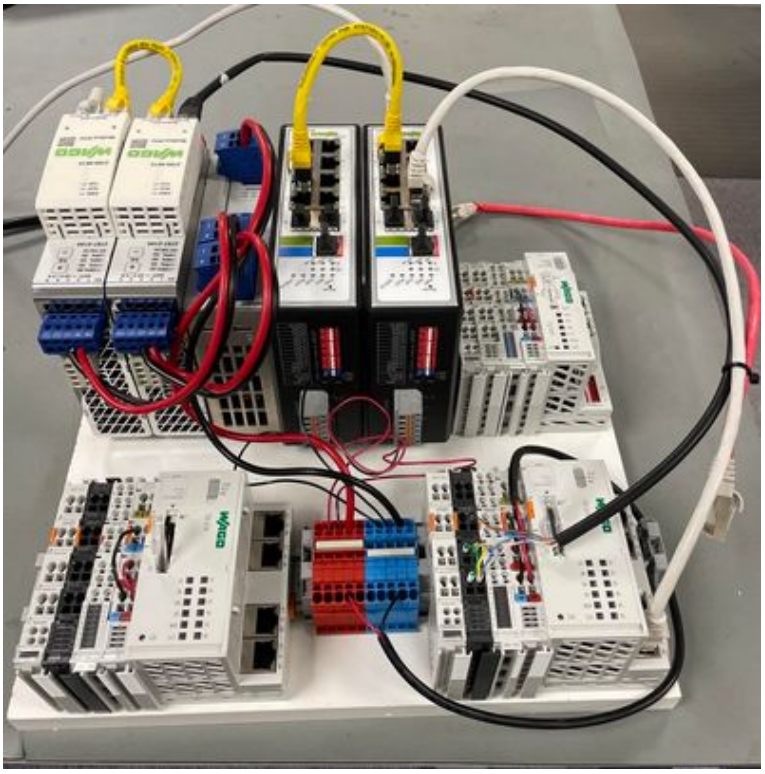
Figur 14: Oversikt over komponentene på WAGO PFC200 750-8210 og deres plassering (Wago, udatert).

3.3.1 Oppsett av PFC

Fastvare oppdateringen tok lenger tid enn forventet, og derfor ble Docker-delen av prosjektet noe forsinket. For oppsett av PFC-en ble det fulgt et dokument fra WAGO Norge Support. Dette spesifiserte hvordan man skulle gå frem med å bruke minnekort

for å oppdatere fastvare på PFC-en, se (Jansrud, 2021). Etter flere forsøk med å oppdatere fastvaren med minnekortet, fortsatte PFC-en å miste kommunikasjon med PC-en via Ethernet. Det ble mye feilsøking rundt problemene som oppstod, hvor WAGO Norge Support til slutt ble involvert. Til å begynne med klarte ikke PFC-en å starte opp fra minnekortet, i stedet startet den fra internminnet. Dette resulterte i at det ikke ble mulig å kopiere fra minnekortet over til det interne minnet. Det vil si at feilen oppstod under steg 4 av 5 i dokumentet som ble tilsendt fra WAGO (Jansrud, 2021).

Årsaken til problemene som oppstod var at fastvareutgaven MR fikk på minnekort fra WAGO var en utviklingsversjon, som vil si at den ikke var til kommersiell bruk. Utgaven var ikke ferdig testet hos WAGO, og det kunne derfor være noen feil og mangler med fastvaren. I samarbeid med WAGO fikk gruppen den nyeste og oppdaterte versjonen av fastvaren. Med oppdatert versjon av fastvaren fikk gruppen satt opp PFC-en riktig, og alt var klart for implementering av Docker.



Figur 15: Bilde av PFC-riggen med WAGO PFC200 750-8210

3.3.2 CODESYS V3.5

WAGO PFC-en som ble brukt har støtte for CODESYS V3.5 programvaren. For kommunikasjon mellom de ulike systemkomponentene ble ethernet/IP benyttet. Siden CODESYS V3.5 har støtte for dette passet det godt til prosjektutførelsen, og

dette ble benyttet i startfasen av prosjektet. Koden som ble utviklet på CODESYS-plattformen hadde som oppgave å "lese" structen som blir sendt fra fjernkontrollen i tillegg til å sende informasjon tilbake. IP-fjernkontrollen og PFC-en ble satt opp som henholdsvis server og klient ved hjelp av CODESYS V3.5. Dette ble gjort før det ble oppdaget at IP-fjernkontrollen kunne kommuniserer direkte med Docker Container hvor OBS kjører. CODESYS V3.5 ble derfor ikke en del av den endelige løsningen på prosjektoppgaven.

3.4 Fjernkontroll

Det ble benyttet en RPi som et mellomledd mellom joystick og OBS. En RPi er relativ liten i forhold til en normal PC, men likevel kraftig nok og passer derfor godt til de oppgavene den ble brukt til. Joysticken som brukes kobles til RPi-en med en USB-kabel. PFC eller PC med OBS er koblet til ved bruk av ethernet-kabel under testing og simulering. Den endelige løsningen baserte seg på Wifi og 4G kommunikasjon mellom USV med OBS og RPi. For å oppnå en sikker og pålitelig kobling, ble det benyttet en VPN løsning fra Zerotier. Dette er en løsning MR selv benytter opp mot sine USV-er. Denne VPN-løsningen gjør at man har kontakt med OBS uansett hvilket nettverk RPi-en er koblet til. Det er dermed mulig å både sende til og motta forskjellige parametre fra USV-en gjennom OBS.

Grunnen til at RPi ble valgt til dette prosjektet, var at MR bruker RPi til sine egne prosjekter. Gruppen ønsket dermed å gi MR muligheten til å benytte og videreutvikle gruppen sitt produkt. Linux, som RPi-en er basert på, er også gratis å bruke, og MR benytter hovedsakelig kun Linux til sine USV-er. Gruppen så derfor på dette som en god mulighet til å lære seg å bruke Linux.

3.4.1 Joystick

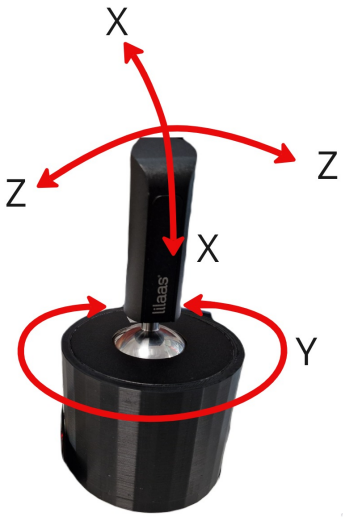
For å simulere og teste det endelige systemet var det behov for en joystick. Joysticken måtte ha to akser som kan gi USV-en informasjon om retning og pådrag. Det første og naturlige valget ble en type joystick som var intuitiv og enkel i bruk. I startfasen av prosjektet var det usikkerhet rundt hvilken type joystick som skulle benyttes og hva som var tilgjengelig fra MR sin side. For å komme i gang med utviklingen av koden til kontrolleren, falt valget på en PS4-kontroller, se figur 16 (a). Dette var noe gruppen selv hadde tilgjengelig, og den har de nødvendige funksjonene som var ønskelig for å test kode og kommunikasjon med RPi og OBS. Etterhvert fikk gruppen tilgang til en joystick fra Logitech av typen Logitech G Extreme 3D Pro Joystick som lignet mer på sluttproduktet, vist i figur 16 (b). Ettersom Logitech

joysticken hadde 6 akser og flere knapper, måtte disse kartlegges. Dette ble gjort ved å lese hvilken akse eller knapp inndataen kom fra. Figur 16 (c) er en L04 fra Lilaas og ble den endelige joysticken som gruppen benyttet til IP-fjernkontrollen. Joysticken er mer industriell og har kun 3 akser å ta hensyn til. Kartleggingskoden som ble utviklet under bruk av Logitech joysticken ble fortsatt brukt, selv om dette ikke var like nødvendig. Ved å gjøre dette er det deremed mulig å benytte samme kode, uansett hvilken joystick som kobles til. X-aksen blir brukt til pådrag i forover og bakoverretning, Z-aksen til dreiemomentpådrag til høyre og venstre og Y-aksen brukes til å styre baugthruster, se figur 16 (d).

(a) PS4-kontroller

(b) Logitech G Extreme 3D Pro Joystick

(c) Lilaas L04 joystick



(d) Illustrasjon over hvordan aksene fungerer på joysticken

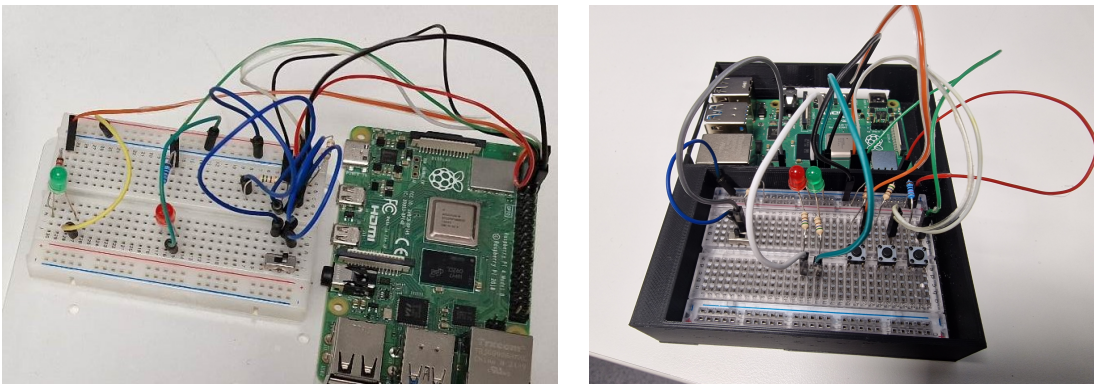
Figur 16: Joysticker som ble brukt og aksevisualisering.

3.4.2 Utvikling av fjernkontroller

Under testing og simulering opp mot OBS, ble det et ønske om å kunne skru av og på fjernkontrollen på en enkel måte, og noen LED-lys som kunne signalisere om fjernkontrollen var på eller av. Ved å bruke et koblingsbrett, en bryter, to LED-lys og kabler ble første versjon av fjernkontrollen laget. Denne kunne bare skru av og på fjernkontrollen, samt signalisere med rødt lys at den var av og grønt for på.

Utover prosjektet så gruppen på andre funksjoner som kunne være gunstige å

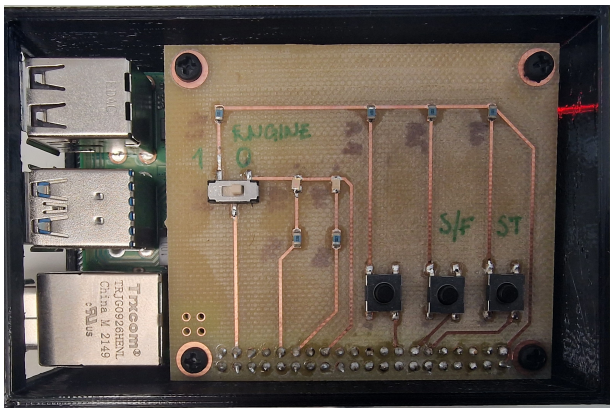
styre fra fjernkontrollen. Det ble lagt til 3 nye knapper, som gjorde det mulig å implementere nye funksjoner. Å koble opp denne løsningen ble fort veldig rotete og vanskelig å ta med seg rundt under testing. Dermed laget gruppen en 3D printet boks hvor både RPi og koblingsbrettet fikk plass. Dette gjorde det lettere å ta med seg fjernkontrollen rundt, og gruppen slapp å koble fra kabler når den skulle pakkes ned.



(a) Versjon 1 (b) Versjon 2

Figur 17: Versjon 1 og 2 av fjernkontroll

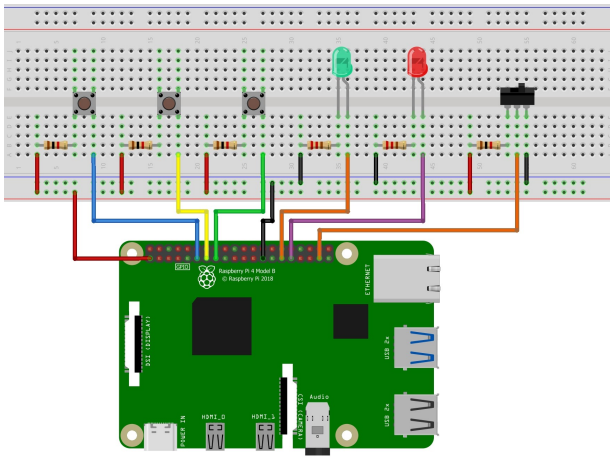
Selv om versjon 2 av fjernkontrollen fungerte til sitt formål, var løse kabler fremdeles en utfordring. Det hendte ofte at disse ble koblet ut under testing, noe som utgjorde en fare for å ødelegge komponenter og det var tidkrevende å koble dem tilbake. Dette kan også skape farlige situasjoner når USV-en opereres. Dersom USV-en er satt i drift og en ledning trekkes ut slik at knappene ikke fungerer, vil operatøren miste kontroll. Dermed vil uforutsette hendeleser kunne oppstå, som for eksempel hvis USV-en er på kollisjonskurs eller drifter mot land kan ikke operatøren forhindre det. Derfor ble det besluttet at gruppen ønsket å lage et eget kretskort til RPi-en, ofte kalt HAT. Denne HAT-en fungerte helt likt som versjon 2, men uten noen løse ledinger og komponenter. Kretstegneprogrammet Altium Designer ble brukt for å designe et kretskort og Elektronikk og prototypelaboratoriet (Elprolaben) ved NTNU bidro med fresing og lodding av komponenter. HAT-en gjorde det dermed mulig å ta med seg fjernkontrollen enkelt rundt uten å være redd for å koble ut noen ledninger. Denne løsningen kan ses i figur 18.



Figur 18: Endelig versjon av fjernkontroll med HAT

3.4.3 Komponentoversikt

Figur 19 viser en oversikt over hvordan alle disse komponentene er koblet opp mot RPi-en.



Figur 19: Overordnet koblingsskjema av fjernkontrollen

De tre knappene er designet for å skru av og på forskjellige funksjoner som:

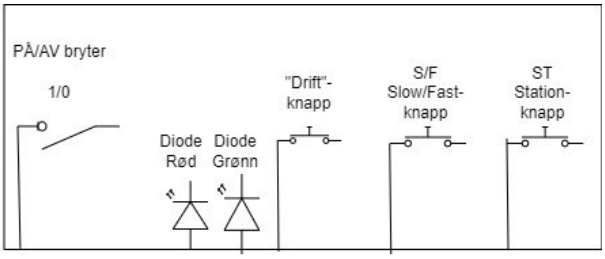
- **Station Keeping:** Det blir regnet ut et punkt for USV-en i forhold til nåværende fart, posisjon og retning, som USV-en skal ligge i ro rundt.
- **Slow/fast mode:** Setter pådraget på USV til enten 0.5 ganger vanlig operasjonspådrag (*slow*), eller gir ut vanlig operasjonspådrag (*fast*). *Slow* modus er tiltenkt bruk i havneområde og andre områder hvor operatør må vise forsiktighet. *Fast* modus er tiltenkt bruk i åpne farvann uten hastighetsbegrensinger. Man kan i tillegg begrense pådrag ytterligere gjennom VCS.

- **Drift:** Sender ut 0 i pådrag fra fjernkontrollene, og gjør at USV vil "drifte" med vind, bølger og strøm. Denne funksjonaliteten er tiltenkt hvis operatør ønsker å la USV drifte, uten å være redd for å gi uønskede pådrag med joysticken.

Det er lagt til to dioder, en rød og en grønn, for å vise status for de forskjellige funksjonene, samt om fjernkontrollen er på eller av. Her er en oversikt over hva de forskjellige lyskombinasjonene betyr:

- **Rødt lys:** Hvis bryterne står i posisjon "0" er fjernkontrollen av, hvis bryteren står i posisjon "1" er kontrolleren på, men står i "Drift".
- **Grønt blinkende lys:** Fjernkontrollen og joysticken er på, og er i "slow-modus".
- **Grønt konstant lys:** Fjernkontrollen og joysticken er på, og er i "fast-modus".
- **Rødt og grønt lys:** Fjernkontrollen er på, og står i "*Station Keeping*"
- **Rødt blinkende lys:** Noe er galt/feil

Bryteren på fjernkontrollen har to mulige posisjoner; "0" og "1". Posisjon "0" signaliserer at fjernkontrollen er av, og operatøren kan dermed ikke påvirke USV-en på noen måte. Står bryteren i posisjon "1", vil fjernkontrollen være på, og operatøren kan styre USV-en.



Figur 20: Oversikt over funksjonene til komponentene på fjernkontrollen

3.4.4 Utregning av komponentstørrelser

For å unngå skader og ødeleggelser på GPIO-pinnene til RPi-en, har hver krets en egen motstand for å regulere spenning og strøm som går til pinnene. Ved å benytte Ohms lov er det mulig å regne ut hvor store motstander man må benytte for å unngå for store strømmer.

$$U = R \cdot I \tag{1}$$

Spenningen ut fra GPIO-pinnene ligger på 3.3V og tåler å kun gi ut 16 mA i strøm. Den kan gi ut mer enn 16 mA, men dette vil kunne ødelegge RPi-en (Mosaic Documentation Web, udatert). Derfor er det viktig å ha en motstand som gjør at strømmen ikke blir høyere enn 16 mA. Ved å se på databladet til LED-lysene, vil spenningen over disse være rundt 2.2V, og maks strøm gjennom dioden er 20 mA (Digi-Key, udatert). Dette betyr at GPIO-pinnene som er komponenten som må tas hensyn til.

$$\begin{aligned} R &= \frac{U_{Tot} - U_{LED}}{I_{Tot}} \\ R &= \frac{(3.3 - 2.0)\,V}{16\,mA} \\ R &= 81.25\Omega \end{aligned} \tag{2}$$

Dette vil si at man trenger minst en motstand på 81.25 Ω for å unngå å skade noen av komponentene. Siden det ikke er ønskelig å ligge så tett opp til grensen på RPi-en og ikke ha veldig stor lysstyrke på diodene, ble en motstand med høyre resistans benyttet. Valget falt på 220 Ω, som gir en strøm på:

$$\begin{aligned} I &= \frac{U_{Tot} - U_{LED}}{R_{Tot}} \\ I &= \frac{(3.3 - 2.0)V}{200\,\Omega} \\ I &\approx 0.006\,A = 6\,mA \end{aligned} \tag{3}$$

6 mA er godt innenfor RPi-en sine grenser, men gir fortsatt nok strøm til dioden for å lyse godt.

Motstandene som er koblet til knappene er en sikkerhet for å unngå skade på RPi-en hvis man programmerer GPIO-pinnen feil. Det er derfor viktig å ha en så stor motstand så det ikke skal kunne være mulig å gå mer enn 16 mA gjennom den kretsen. Omgjøring av formel (1) gir:

$$R = \frac{U_{Tot}}{I_{Tot}}$$
$$R = \frac{3.3\,V}{16\,mA}$$
$$R = 206.25\,\Omega$$

Dette betyr at man trenger minst en motstand på 206.25 Ω, men for å være sikker på å ikke ødelegge noe valgte gruppen å benytte en motstand på 1 kΩ. Omgjøring av formel (1) gir:

$$I = \frac{3.3\,V}{1\,k\Omega}$$

$$I = 3.3\,mA$$

Dette vil gi en strøm på 3.3 mA, som er trygt innenfor grensene.

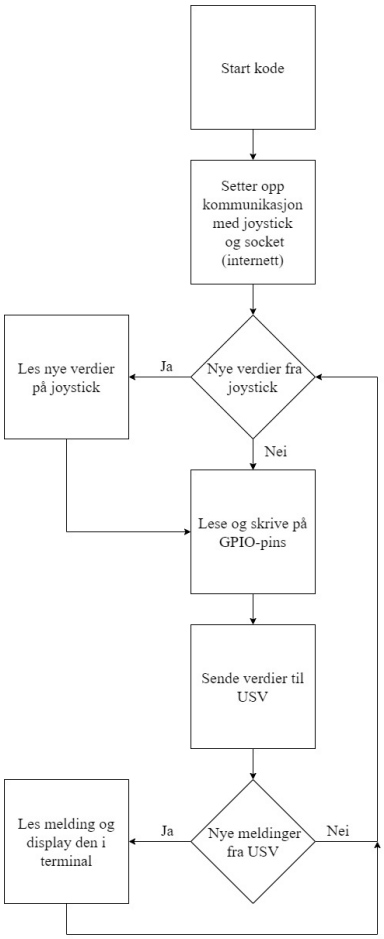
3.5 Fjernkontrollens programkode

Koden som ligger på RPi-en er C++ kode som tar inn signaler fra joysticken. Dette gjør den ved hjelp av et bibliotek som allerede ligger inne i Linux, "joystick.h". Dette biblioteket tar inn data fra joysticken og returnerer hvilken akse som beveges og dens verdi. Disse verdiene lagres så i egendefinerte variabler, og sendes til OBS ved bruk av UDP-kommunikasjon.

For å gjøre lagring og sendingen av verdiene så enkel som mulig, ble gruppen enig om at det skulle brukes struct. Struct er en funksjon i C++ som grupperer flere variabler som man ønsker skal høre sammen. Denne gruppen av variabler kan inneholde forskjellige datatyper og variabler. Dette gjør det mulig å sende hele denne gruppen av variabler samtidig i samme UDP-datapakke. Dermed slipper en å passe på at hver datapakke kommer i riktig rekkefølge eller å telle antall "bytes" for å lagre variablene på riktig plass hos mottaker. Ved å legge på en "header" i structen, muliggjør det for mottaker å kunne sjekke hvilken struct som ble motatt. Dette er spesielt gunstig å ha hvis det sendes flere forskjellige structs. Det er også blitt lagt til en "feet" til structen som mottaker kan bruke for å sjekke om hele meldingen er motatt riktig.

Koden som er blitt utviklet på fjernkontrollen består av tre C++ filer og tre protobuf-filer. Blant de tre C++ filene er det en hovedfil, en hvor alle funksjonene

er lagret og siste er en "headerfil". I hovedfilen, kalt "ipRemote.cpp" er hvor "main" funksjonen ligger, og er hvor fjernkontrollerkoden styres fra. Alle funksjonene som brukes i hovedfilen ligger i "IpRemoteLib.cpp" filen. Ved hjelp av headerfilen "ipRemoteLib.h" kan begge cpp-filene snakke med hverandre. Dette betyr at en funksjon som ligger i "ipRemoteLib.cpp" kan kalles på og brukes i "ipRemote.cpp".



Figur 21: Flytskjema av koden på fjernkontrollen

3.5.1 UDP funksjoner

Kommunikasjon mellom fjernkontrollen og USV-en følger nettverksprotokollen UDP. Systemet er tidskritisk, så derfor er etablering av kommunikasjon med høy hastighet prioritert. USV-en og fjernkontrollen kommuniserer på to forskjellige porter og de opptrer derfor både som server og klient. For å opprette UDP kommunikasjon mellom nodene må en socket initieres med informasjon om hvilken port og hvilken IP-adresse dataen skal sendes og mottas på. I funksjonskallet hvor socket initieres blir den boolske variabelen *server* satt til *true* for socketen som skal opptre som server. Videre bindes socketen til IP-adressen og porten som ble angitt.Dette er vist i kodeutdrag 5.

```
// Makes the socket address
memset(&addr, 0, sizeof(addr));  // Resets addr memory
addr.sin_family = AF_INET;  // Sets the IP to IP-4
addr.sin_addr.s_addr = IP;  // Sets the IP-address
addr.sin_port = htons(PORT) // Sets the port
```

Kodeutdrag 5: Oppkobling av server og klient

Kodeutdrag 6 viser hvordan socketen på fjernkontrollen som opptrer som server vil binde seg til klientaddressen fra USV, og klienten på fjernkontrollen forsøker å opprette en forbindelse med serveren hos USV-en.

```
if(server){
    // If socket is a server, it will bind the socket to the IP and
       port
    bind_result = bind(sock, (const struct sockaddr *)&addr, sizeof(
       addr));
}
else{
    // If socket is a client, it will connect the socket to the IP and
       port
    connect_result = connect(sock, (sockaddr *)&addr, sizeof(addr));
}
```

Kodeutdrag 6: Oppkobling av server og klient

Etter at en forbindelse mellom USV-en og fjernkontrollen er opprettet kan data overføres mellom enhetene. For enkel overføring av dataen er den organisert i en struct som er lik på sender og mottakersiden.

```
struct joystickValues{
    uint16_t msg_header = 0x45;
    bool active = false;
    bool station = false;
    bool slow = true;
    int16_t x = 0;
    int16_t y = 0;
    int16_t z = 0;
    uint16_t msg_tail = 0x1A4;
};
```

Kodeutdrag 7: Joystickverdier

Denne structen, se kodeutdrag 7, består av et hode som sier hvilken struct som blir sendt, en boolsk variabel som sier den er aktiv eller ikke, om USV skal være i *Station Keeping* og om *slow* er aktivert. Som en sikkerhetsmekanisme er *slow* aktivert og

brukeren må skru den av for å kunne gå over i *fast*. I tillegg sendes verdiene i x-, y- og z-retning på joysticken. Til slutt sendes en fot som benyttes av mottaker for å sjekke at hele meldingen er motatt. Det er laget en egen funksjon, kalt *sendStruct*, som sender data fra fjernkontrollen til USV-en. I tilfeller hvor dataen ikke sendes riktig vil en feilmelding vises i terminalvinduet, og funksjonen returnerer "1".

```
int send_result;

send_result = sendto(sock, (char *)&jv, sizeof(jv), 0, (const
    struct sockaddr *) &addr, sizeof(addr));
if((send_result < 0) && !errorFlag){
    cout << "Error sending data" << endl;
    return 1;
}
```

Kodeutdrag 8: Sending av verdier

Fjernkontrollen mottar data fra USV-en i form av en protokoll buffer, som blir gjort gjennom funksjonen kalt *receiveProtobuf*. Denne funksjonen tar imot den sendte dataen fra USV-ens klient og lagrer den i en protobuf struct. Kodeutdrag 9 viser mottak av meldingen, uthenting av informasjonen og lagring i protobuf structen.

```
receive_result = recvfrom(sock, (char *)buffer, MAXLINE, 0, (struct
    sockaddr *) &addr, &len);
if(receive_result < 0){
    cout << "Error reciving data" << endl;
    return 1;
}

// Parse the received data to the protobuf message
fb.ParseFromArray(buffer, receive_result);
return 0;
```

Kodeutdrag 9: Mottaking av verdier

En ulempe med å bruke *recvfrom* for å motta meldinger er at den vil holde koden fryst helt til en melding har kommet. Dette betyr at koden ikke vil gå videre før USV-en har sendt en ny melding som fjernkontrollen kan lese. For å unngå at hele koden stopper, ble en funksjon kalt *checkForMsg* opprettet. Denne har i oppgave å sjekke om det er kommet noen nye meldinger, og benytter seg av *select*. Denne funksonen er inkludert fra C++ sine egne biblioteker og returnerer 0 hvis det ikke er motatt noen nye meldinger eller antallet "bytes" som ble motatt. Dermed er det mulig å sjekke etter nye meldinger fra USV-en og unngå at hele koden stopper i påvente av nye meldinger.

```
timeout.tv_usec = 10000; // Sets the wiating time to 0.01 sec
int read_result = 0;

read_result = select(sock + 1, &rfds, NULL, NULL, &timeout);  // Checks
    for new messages

if (read_result > 0){  // New message from USV
    return 1;
}
else{      // No new message from USV
    return 0;
}
```

Kodeutdrag 10: Sjekk av nye meldinger

3.5.2 Joystickfunksjoner

Funksjoner som omhandler joysticken er gruppert sammen. De omfatter åpning av joystickens filplassering for å muliggjøre kommunikasjon, av og på funksjonalitet, lesing av joystickens posisjon, skalering av verdiene som blir motatt og oppdatering av structen som skal sendes. Status for tilkobling av joysticken vil vises i terminalen.

```
js = open(device, O_RDONLY);
if(js < 0 && !errorFlag){
    cout << "Error connecting to joystick" << endl;
    return 1;
}

cout << "Joystick connected" << endl;
return 0;
```

Kodeutdrag 11: Oppstart av kommunikasjon med joystick

Når fjernkontrollen skrus på, starter motoren på USV-en og den boolske variabelen *active* i structen joystickValues (jv) settes til *true*. Brukeren får en melding i terminalen om fjernkontrollens tilstand. Det er lagt inn sikkerhetsmekanismer i programkoden som hindrer brukeren i å skru på kontrollen dersom joysticken ikke er i senterposisjon. Dette skal forhindre at USV-en får utilsiktet instruks om å kjøre.

```
if(digitalRead(switchOn) && (jv.x == 0) && (jv.z == 0) && (jv.y == 0)){
    if(!jv.active){
        cout << "Controller is active" << endl;
        jv.active = true;
```

Kodeutdrag 12: Oppstartsrutine

Joystickverdiene leses, skaleres og oppdateres kontinuerlig så lenge fjernkontrollen har forbindelse med USV-en og joysticken er tilkoblet. Skaleringen sørger for at verdiene som bli sendt er mellom $\pm$ 100 og at aksene følger en logisk bevegelse av joysticken. Dersom joysticken kobles fra vil en feilmelding vises i terminalen og USV-en vil bli satt i *Station Keeping*. Dette er også en sikkerhetsmekanisme for å hindre utilsiktet kjøring med USV-en.

3.5.3 Brukergrensesnitt og feilmeldinger

Fjernkontrollens brukergrensesnitt består en joystick, tre knapper, to dioder, en bryter og terminalvinduet. I programkoden blir GPIO-pinnene satt til input for knappene og output for diodene. Brukeren får tilbakemelding om hvilken modus fjernkontrollen er i gjennom diodene. Statusen på *Station Keeping*, slow/fast modus og drift kan endres ved hjelp av knappene og diodene vil gi tilbakemelding basert på brukerens valg. Brukeren vil også få en tilbakemelding i terminalvinduet. USV parameterene som blir sendt til fjernkontrollen kan bare fremvises i terminalvinduet. Brukeren får informasjon om hastighet [knop], omdreininger per minutt [RPM], retning [grader] og posisjon [grader nord][grader øst].

```
// Displays the USV parameters
void display(Msg::Remoteudp::Feedback &fb)
{
cout << "-----------------------------------------" << endl;
cout << "USV parameters:" << endl;
cout << "Speed over ground: " << fb.sog()*1.9438 << " kn"<< endl;
cout << "RPM: " << fb.rpm() << " rpm" << endl;
cout << "Course over ground: " << fb.cog()*180/M_PI << " deg" << endl;
cout << "Latitude: " << fb.latitude()*180/M_PI << " deg N" << endl;
cout << "Longitude: " << fb.longitude()*180/M_PI << " deg E" << endl;
cout << "-----------------------------------------" << endl;
}
```

Kodeutdrag 13: Tilbakemelding

Det er lagt inn flere sikkerhetsmekanismer i koden for å sørge for at USV-en opereres trygt. Når USV-en står i *Station Keeping* er det ikke mulig å skifte modus før joysticken er i senterposisjon. Det samme er tilfellet for *slow/fast* modus og *drift*. Skiftet mellom forskjellige moduser skjer ved å endre de boolske variablene *Station*, *drift* og *slow*.

```
// As a security measure it is only possible to turn off stationkeeping
    when the joystick is in center position
switch(jv.x || jv.z || jv.y || !jv.station){
```

```
case 0:
    jv.station = false;
    cout << "Stationkeeping is inactive" << endl;
    break;
default:
    if(!jv.station){
        cout << "Stationkeeping is active" << endl;
    }
    jv.station = true;
    break;
```

Kodeutdrag 14: Modusskifte

Flere av funksjonene i programkoden har mulighet til å sette et errorflagg til *true*. Dette vil resultere i at den røde dioden begynner å blinke for å indikere til brukeren at noe er galt.

3.5.4 Ubenyttede funksjoner

Gjennom arbeidet med utviklingen av programkoden har det blitt laget flere funksjoner som ikke ble tatt i bruk i sluttproduktet, men som var svært nyttige under testing og feilsøking. Funksjonene omhandler fremvisning av joystickens posisjon samt alternative måter å sende og motta data. Det ble tidlig utviklet en funksjon for å sende en enkel karakter (*eng. char*) for å bli kjent med "sendto" funksjonen. For å motta parametere fra USV-en ble det laget en funksjon for å motta en struct.

3.5.5 Kjøring av kode

For å kunne kompilere fjernkontroll-koden på RPi-en, må kommandoen i kodeutdrag 15 kjøres i terminalen.

```
$ sudo g++ -o ipRemoteUSV.exe ipRemote.cpp ipRemoteLib.cpp
  MsgRemoteUdp.pb.cc -lwiringPi `pkg-config --cflags --libs
  protobuf`
```

Kodeutdrag 15: Kompilering av fjernkontrollerkode

Sudo, som står for "super user do", gir brukeren lov til å kjøre en kommando som admin eller "root", og må kjøres for å gi protobuf tilgang til de filene den trenger for å fungere som tiltenkt. *IpRemoteUSV.exe* er den kjørbare filen hvor alt av koden blir kompilert til, og som må kjøres når man ønsker å starte koden etter fulført kompilering. Videre legges alle *.cpp*-filene som skal være med inn, slik som *ipRemote.cpp* og *ipRemoteLib.cpp*. *IpRemote.cpp* er hvor *main* funksjonen ligger og fjernkontroller-koden "styres fra", mens *ipRemoteLib.cpp* er hvor alle funksjonene

som brukes i *ipRemote.cpp*. *MsgRemoteUdp.pb.cc* er protobuf filen hvor protobuf structen er deklarert, og må være med for å kunne tolke dataen sendt fra USV-en. For å kunne snakke med GPIO-pinnene må *-lwiringPi* legges ved kommandoen. Siden biblioteket *wiringPi* ikke ligger lokalt med alle andre biblioteker, må denne legges til automatisk ved å legge det ved under kompilering. For å få med alle pakkene og bibliotekene til protobuf, må også disse legges ved manuelt *'pkg-config –cflags –libs protobuf'*. Når kompileringen er ferdig, kjøres kommadoen vist i kodeutdrag 16 i terminalen for at koden skal starte opp.

```
$ ./ipRemoteUSV.exe
```

Kodeutdrag 16: Kjøring av fjernkontrollerkode

3.6 Integrasjon med MR sitt OBS og VCS

En stor del av den avsluttende delen av prosjektet var å integrere det gruppen hadde lært i løpet av testing og undersøkelser inn i MR sitt eget OBS. Dette består av en ganske stor kodebase, og er veldig komplekst for en person å sette seg inn i på tiden gruppen hadde tilgjengelig. Derfor ble MR sitt software-team mer involvert på dette stadiet, selv om gruppen skrev kode og utviklet selv. Denne delen av prosjektet ble igjen mer informasjonshentings- og læringsbasert. Det var mange ukjente programmer og arbeidsområder, og gruppen måtte sette seg inn i mye av dette. Det var i denne delen at en egen klasse for kommunikasjon mellom OBS og i første omgang MR sin VCS-simulator ble opprettet. Denne VCS-simulatoren klarer å simulere de aller fleste av USV-enes funksjonaliteter, og var derfor et nyttig verktøy for å kontinuerlig teste kode som ble lagt til i klassen "UdpRemoteInterface", som var IP-fjernkontrollens klasse i OBS.

Kort fortalt henter denne klassen inn meldinger sendt fra IP-fjernkontollen, utfører nødvendige utregninger, og sender kommandoer videre til USV-ens pådragsorgan, for eksempel motoren. Dette inkluderer mulighet for å ta og gi fra seg kontroll, sette USV i *Station Keeping*, pådragsbegrensning og monitorering av at systemet mottar meldinger. I tillegg integreres funksjonalitet til VCS i denne klassen, og gir VCS-operatøren mulighet til å justere enkelte innstillinger på USV-en.

3.6.1 Kommunikasjon

Denne klassen henter først UDP-meldingen, som har et forhåndsbestemt oppsett med hode, melding og fot. Dette ble gjort ved å opprette et *QUdpSocket* objekt, basert på Qt sitt innebygde nettverksbibliotek. Denne funksjonen leser meldingen som et "bitarray" og sender denne inn i funksjonen *parseData*, som står for

informasjonsuthenting. Dette gjøres ved å deklarere objektet *data*, og deretter benytte innebygde funksjoner for behandling av *QByteArray*-typer.

```
QUdpSocket *udp = dynamic_cast<QUdpSocket *>(m_iodev); // Creating UDP
    socket to recieve communication from configured I/O-device
[...]
while(udp->hasPendingDatagrams())
{
    QByteArray data; // Sends received data into qbytearray sized to
        the message
    data.resize(udp->pendingDatagramSize());
    udp->readDatagram(data.data(), data.size());
    parseData(data);
}
```

Kodeutdrag 17: Kode for å lese UDP-meldinger

Hode og fot, eller "header" og "footer" som de ofte kalles på engelsk, er der for at man skal kunne sjekke at meldingene leses i riktig format. Dette var aldri noe problem for dette bruksområdet, da meldingene er relativt små, men ble lagt inn fordi dette kan anses som god kodeskikk Dette gjelder spesielt hvis det etter hvert vil introduseres flere forskjellige meldingstyper, eller med varierende innhold. Da vil man kunne sjekke hodet på meldingen, sammenligne mot forskjellige gitte hoder, og kunne utføre de riktige kommandoene på denne meldingen. Fordi det bare mottas en meldingstype slik systemet er konstruert, vil lengden på de forskjellige meldingssegmentene være de samme hver gang, og gjør det enkelt å hente ut informasjon, kontra hvis meldingene var av varierende lengde. Dette var et bevisst valg fra gruppens side, nettopp for å gjøre informasjonsuthenting sikrere og mer effektivt. Hvis denne klassen skulle kunne brukes for varierende typer data og lengder, måtte man ha tenkt annerledes og mye mer komplekst. Strukturen på meldingen var den samme som den som sendtes fra RPi-en, dvs:

```
struct UdpMes
{
        uint16_t head;
        bool active;
        bool station;
        bool slow;
        int16_t x;
        int16_t y;
        int16_t z;
        uint16_t feet;
};
```

Kodeutdrag 18: Struct som mottas fra RPi

Structen i kodeutdrag 18 ble kjørt gjennom *parseData*-funksjonen, og lagret i

structen *UdpMes*:

```
const UdpMes *msg = reinterpret_cast<const UdpMes *>(data.constData
    ());
```

Kodeutdrag 19: Kode for å overføre mottatt melding til definert struct

Denne structen har samme oppsett som forklart i 3.5.1. Den mottatte UDP-meldingen ble "castet" inn i en instans av denne structen. Dette betyr at bitverdiene som er mottatt konverteres automatisk til de forskjellige datatypene, f.eks $uint16\_t$, og sikrer da at bit-ene blir tolket riktig. Da vil structen inneholde alle verdiene man ønsker, og hvis denne structen heter f.eks *msg*, som i eksempelet, vil man kunne hente ut informasjon enkelt, ved å skrive $x\_verdi = msg.x$, og få verdiene ut. Ut ifra beskjeden som ble mottatt sendte man enten meldingene *MSG_TAKE_CONTROL*, *MSG_RELEASE_CONTROL* eller *MSG_MODE_MANUAL*. Den første av disse tre tar kontroll over USV-en, den andre gir fra kontroll, og den tredje sender manuelle styringskommandoer mottatt fra joysticken.

De resterende delene av klassen var i hovedsak timere som skal sikre at ting skjer innen rimelig tid. Hvis det ikke skjer skal den varsle bruker, og andre former for tilbakemeldinger til bruker. Dette kan for eksempel være debug-meldinger og informasjon som kan være nyttig, spesielt i testsammenheng. Blant annet mottatte x-, y- og z-verdier ble skrevet til konsollen, og forskjellige beskjeder ble skrevet ut hvis visse betingelser ble oppfylt. Generelt er dette at man trykker på en knapp i VCS, som gjør at debug-meldinger skrives til konsoll.

OBS-klassen sender også en melding tilbake til fjernkontrollen. Dette var mer utfordrende å implementere, fordi RPi-siden slet med å tolke dataene den mottok riktig. Derfor ble det besluttet å benytte seg av protobuf-kompilering av meldinger. Dette ville sikre at meldingene som ble sendt ble tolket på riktig vis, og man fikk den informasjonen man ville ha. Meldingsstrukturen som ble sendt tilbake er som følger:

```
message Feedback
{
        ///Speed over ground
        required float sog = 1;
        //Course over ground
        required float cog = 2;
        ///Latitude value
        required double latitude = 3;
        ///Longitude value
        required double longitude = 4;
        ///Rpm of engine
        required float rpm = 5;
```

```
}
```

Kodeutdrag 20: Protobuf meldingsstrukturen som sendes

Disse meldingene brukes på en litt annen måte enn den tidligere nevnte structen som brukes på den mottatte meldingen. Her lages et meldings-objekt, hvor de forskjellige verdiene defineres som egenskaper ved dette objektet. Dette kan sammenlignes med at en bil er objektet, og egenskapen for eksempel er at den har hjul, motor eller lignende. Alle disse har "required" foran, dvs. at disse må sendes med i meldingen. Informasjonen som sendes tilbake er USV-ens fart, retning, nåværende lengde og breddegrad og turtallet til motoren. Disse meldingene kan enkelt utvides til å inkludere flere eller andre data, ved å endre strukturen til meldingen i .proto filen hvor disse meldingene er derfinert. I tillegg må man hente og legge til dette aktivt i meldingen i OBS-koden.

Funksjonen for å sende tilbake til RPi-en heter *reportBack*. Denne vises under:

```
QUdpSocket    udpSocketSend; // Creates UDP socket object
QHostAddress hostIp;
hostIp.setAddress(m_address);                    // RPi IP
udpSocketSend.connectToHost(hostIp, m_port); // Connects socket to host
    (RPi remote)

Msg::Remoteudp::Feedback msgFeedback;          // Creates feedback
    message
msgFeedback.set_cog(m_nav.cog());              // 0-2*pi (radians)
msgFeedback.set_sog(m_nav.sog());              // SOG
msgFeedback.set_rpm(m_rpm);                    // RPM of engine
msgFeedback.set_latitude(m_nav.latitude());    // Latitude
msgFeedback.set_longitude(m_nav.longitude()); // Longitude

QByteArray byteArray;
byteArray.resize(msgFeedback.ByteSize());
msgFeedback.SerializeToArray(byteArray.$\color{clr-key3}data$(),
    byteArray.size());

udpSocketSend.writeDatagram(byteArray); // Sends bytearray packed into
    protobuf message
```

Kodeutdrag 21: Funksjon for å sende tilbakemeldinger til fjernkontrollen

Originalt ble det forsøkt å sende meldinger tilbake på samme måte som RPi sender meldinger til OBS, som er utdypet i 3.5. Dette fungerte ikke, og verdiene som ble mottatt var som regel ikke i nærheten av verdiene som ble sendt. Det ble forsøkt flere løsninger på dette, blant annet å undersøke om det ble lagt til ekstra bit etter

hver verdi som ble sendt, eller om noen av bit-ene ble flyttet om på under sending, og man måtte om-rokere disse på mottakersiden. Ingen av disse forsøkene ledet noen sted, og det ble derfor bestemt at protobuf skulle benyttes. Dette fungerte på første forsøk, og gjorde meldingssending mye enklere. En stor fordel med dette er at det, som tidligere nevnt, er relativt lett å øke størrelsen på meldingene. Dette er en fordel med tanke på videreutvikling eller andre ønsker fra kunde eller utvikler.

3.6.2 Funksjonsutførelse

Forflytning av USV

OBS-klassen har flere forskjellige funksjonaliteter som den utfører på grunnlag av meldingene den mottar fra både VCS og fjernkontrollen. Hovedfunksjonen er å ta imot manøvreringsmeldinger, sende disse inn til et meldingsobjekt, og deretter sende denne meldingen av gårde. Et eksempel på dette er vist under:

```
Msg::Modes::MsgManual msgMan; // If remote slow not active, multiply x2
if (msg.slow)
    multiplier = 1.0; // Gives max output = 50%
else
    multiplier = 2.0; // Gives max output = 100%
msgMan.set_force_x(multiplier * m_throttleScale * msg.x / 100.0);
// Calculates throttlescaled force in x-direction
msgMan.set_force_y(0);
msgMan.set_torque_z(multiplier * m_throttleScale * msg.z / 100.0);
// Calculates throttlescaled torque in z-direction

auto message = Conpago::Message(MSG_MODE_MANUAL, msgMan);
message.setSource(m_ipVcsId); //VCS client with id m_ipVcsId
sendMessage(message);
```

Kodeutdrag 22: Kode som regner ut pådrag og sender ut på aktuatorer

Her regnes pådraget mellom -100 og 100 ut for kraft i x-retning og dreiemoment i z retning, dvs. sving til venstre eller høyre. Først sjekker den om mottatt melding fra joystick har slow aktivert. Hvis den har dette, settes multiplikasjonsfaktoren, "multiplier" til 1, hvis ikke settes denne til 2. Deretter benytter den seg av tidligere nevnte multiplier, ganger med *m_throttleScale* og deretter ganger med pådraget i x eller z retning, før alt deles på 100. Da ender man opp med et tall mellom -1.0 og 1.0, som sendes til styresystemet. *m_throttleScale* er en variabel man kan stille i VCS, og den kan settes mellom 0.2 og 0.5. Denne benyttes for å begrense maks pådrag, og gjelder både når USV er i slow og fast modus. Formelen for utregning av faktisk pådrag er vist under:

$$Padrag = \frac{innverdi \cdot skaleringsfaktor \cdot slowMultiplikator}{100} \tag{4}$$

$innverdi = [-100, 100]$
$skaleringsfaktor = [0.2, 0.5]$
$slowMultiplikator = 1 \ eller \ 2$

Det lages også to objekter i denne kodesnutten, *msgMan* og *message*. Disse er predefinerte strukturer, og inneholder informasjonsfelter hvor man stiller inn forskjellige variabler, for eksempel "force_x", altså kraft i x-retning. Når disse er fylt inn, vil styresystemet selv kunne hente ut nødvendig informasjon for å styre USV-en i riktig retning og med riktig pådrag. Meldingene for å sette pådrag på baugthrusterene fungerer veldig likt som eksempelet vist over, bortsett fra at det ikke skaleres ned, men utnytter det fulle intervallet fra -100 til 100.

Denne måten å definere objekter, gi disse nødvendig informasjon og sende disse objektene som en melding benyttes i store deler av OBS-koden, og fungerer på omtrent samme måten som eksempelet vist over. Disse meldingsobjektene var i hovedsak allerede definert fra MR sin side, så det var bare å ta disse i bruk og legge inn informasjon. Forskjellene på forskjellige meldingsobjekter er kun hva slags informasjon objektet krever for å kunne utføre ønsket arbeid. Det er også enkelt å sjekke hva slags informasjon et objekt kan inneholde underveis i programmeringsprosessen, noe som gjør det lett å utvide for en eventuell tredjepart.

Sette USV i *Station Keeping*

En annen funksjonalitet var å kunne sette USV-en i *Station Keeping*. Meldingene sendes lignende kodesutdrag 23 for å sette kraft og dreiemonent, og vises under:

```
if (msg.station && !stationActive)
{ // Can manually activate station mode
    Msg::Modes::MsgStationHighRes msg;
    // Setting mode to Station with WOPC as the specification
    msg.set_latitude(m_nav.latitude());
    // Sets latitude for stationpoint as current latitude
    msg.set_longitude(m_nav.longitude());
    // Sets longitude for stationpoint as current longitude
    msg.set_velocity(m_nav.sog());
    // Informs station-function of current speed-over-ground, so it can
        calculate best station point
    msg.set_station_choice(Msg::Modes::StationChoice_WOPC);
    // Sets station to WOPC, can change this to f.ex. virtual anchor
    auto message = Conpago::Message(MSG_MODE_STATION_HIGH_RES, msg);
    message.setSource(m_ipVcsId); // Sets source of message
    sendMessage(message);
```

```
    // Sends message for stationkeeping, with lat and long values and
       velocity
    qInfo() << "send msg MSG_MODE_STATION_HIGH_RES";
}
```

Kodeutdrag 23: Kode for å aktivere Station Keeping

For å sette USV i *Station Keeping* på riktig måte, må man først gi den informasjon om hvor den ønsker å ha *Station Keeping* punkt, og nåværende hastighet. Ut ifra dette regnes et optimalt punkt og måte å komme seg til dette punktet på. *Station Keeping* funksjonaliteten er allerede definert fra MRs side, og denne klassen kaller da denne funksjonen ved å sende *MsgStationHighRes*. I tillegg spesifiseres hvilken type *Station Keeping* man skal benytte seg av. Her er "WOPC" (Weather-Optimal Positioning Control) valgt, men man har også mulighet for å velge "Virtual Anchor" eller "Drifting" ved å endre *set_station_choice* til en av de andre mulighetene. En mulig videreutvikling er å kunne endre *Station Keeping* type dynamisk ved hjelp av fjernkontrollen. Et viktig element ved denne koden er bruken av et "if"-argument, som er første linje i kodesnutten. Denne sikrer at *Station Keeping* kun aktiveres hvis man får melding av fjernkontrollen om å gjøre det, og *Station Keeping* ikke allerede er aktivert. Grunnen til at det må gjøres slik, er at man ellers ville redefinert koordinatene til *Station Keeping* punktet hver gang koden ble kjørt gjennom. Derfor har man en boolsk variabel som blir satt til høy når *Station Keeping* er aktivert på USV-en, og kallet til *Station Keeping* vil kun kunne kjøres en gang. Hvis dette ikke hadde vært implementert, ville *Station Keeping* punktet flyttet seg kontinuerlig med USV-en, og den ville ikke arbeidet for å bli på punktet hvor *Station Keeping* først ble aktivert. Denne sjekken etter om *Station Keeping* er aktivert, som så setter *stationActive* til høy, utføres et annet sted i koden.

Tidtakere

I tillegg til disse operasjonelle funksjonalitetene, er det også innebygd noen sikkerhetsfunksjoner i denne klassen. Man har flere forskjellige "timere", som aktiverer forskjellige funksjoner. Det er en "timer", *m_feedbackTimer*, som aktiveres hvert sekund, og sikrer at en informasjonsmelding sendes tilbake til fjernkontrollen. Dette kan være nyttig for en operatør, da man kan se status på hvordan USV-en beveger seg, og dermed ha en ide om alt er som det skal. I tillegg er det to "Watchdog-timere", som kjører kontinuerlig og skal aktiveres hvis det er noe galt med kommunikasjonen. Den første timeren er *m_reopenTimer*, og kjører hvert 5. sekund. Denne timerens oppgave er å åpne mottaksporten igjen, hvis man ikke har mottatt meldinger i løpet av det tidsrommet. Dette er for å sikre at det ikke er en blokkert port på USV-ens ende som forhindrer kommunikasjon.

Den andre timeren er *m_timeoutTimer*, og denne aktiveres hvert sekund hvis man

ikke har mottatt pakker innen dette tidsrommet. Denne timerens oppgave er å sette USV-en i *Station* modus, og holde den der helt til kommunikasjonen er gjenopprettet og man mottar navigasjonsmeldinger fra fjernkontrollen. Grunnen til at man valgte å sette USV-en i *Station* fremfor å be den f.eks. holde konstant pådrag og retning, er at denne fjernkontrollen gjerne skal benyttes til finmanøvrering i områder med lite plass eller andre fartøyer og gjenstander i vannet. Det ville da kunne skapt farlige situasjoner hvis USV-en fortsatte å bevege seg. Gruppen kom derfor frem til at *Station Keeping* var det tryggeste valget for et kommunikasjonsbrudd. Grunnen til at timeren ble satt på 1 sekund var at dette var ansett som et kort nok tidsrom til at alvorlig skade mest sannsynlig ikke kunne inntreffe, samtidig som det ikke var så kort at timeren kun registrere et lite tap av kommunikasjon som kom av pakketap eller tilfeldigheter, og ikke et faktisk kommunikasjonstap. Samtidig som denne USV-en settes i *Station*, settes en variabel som sier at fjernkontrollen har styringen til lav, slik at koden for styring ikke kjører. Dette er fordi det kan ligge igjen meldinger på porten som USV-en kan lese, selv om disse meldingene er gamle.

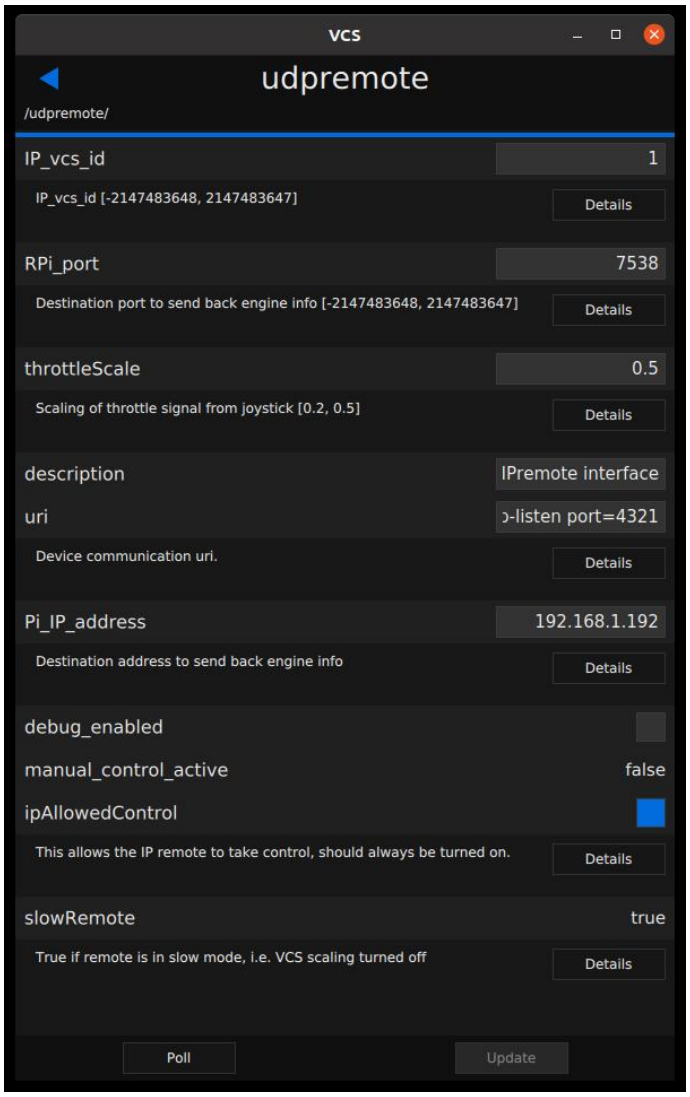
3.6.3 VCS-integrasjon

I tillegg til å få meldinger fra fjernkontrollen, er det lagt til en VCS-meny for UdpRemoteInterface-klassen, hvor man kan stille forskjellige parametere. Dette er for å lettere kunne justere hvordan USV-en oppfører seg under drift, og inneholder også litt grunnleggende informasjon om de forskjellige valgene man ha der. Disse legges til i VCS fra OBS ved å først deklarere typen til disse som *Property<type>*, deretter gi informasjonen som trengs til disse, før man til slutt benytter seg av *addProperty(&variabel)*. Et eksempel på dette for RPi-ens IP-adresse er vist i kodesnutt 24.

```
Property<QString> m_address;   // RPi remote IP
[...]
m_address = "192.168.1.192"; // Sets IP address for Pi joystick
    controller to send info
m_address.setName("Pi_IP_address");
m_address.setInfo("Destination address to send back engine info");
addProperty(&m_address);
```

Kodeutdrag 24: Kode for å legge til verdier i VCS-menyen

Den resulterende F9-menyen ser slik ut:



Figur 22: F9-menyen i VCS for UdpRemoteInterface-klassen

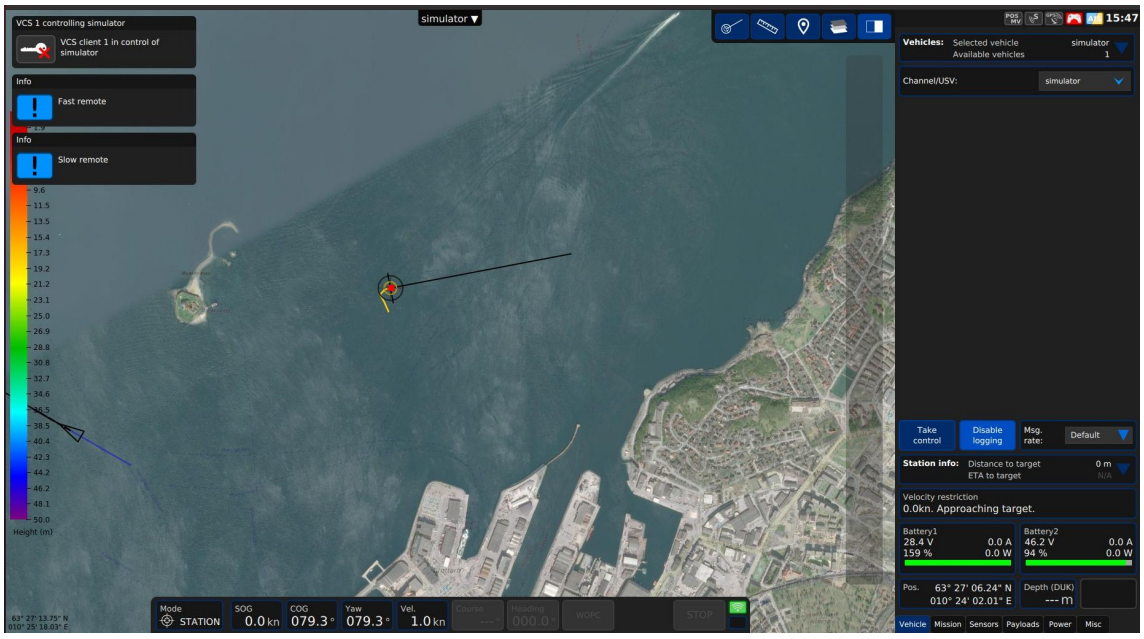
Her kan man se og endre en del forskjellig informasjon. Det er også skrevet inn informasjon under hvert punkt, hvis brukeren er usikker på hva et felt gjør. Fra øverst til nederst vil man kunne endre eller se følgende:

- *IP_vcs_id*, klient-id til IP-fjernkontrollen, her satt til 1.
- *RPi_port*, dvs. hvilken port USV-en skal sende informasjon til på RPi-en.
- *throttleScale*, skaleringsfaktoren som brukes til å regne ut pådrag i både "Slow" og "Fast"-modusene. Denne er minimum 0.2 og maksimum 0.5.
- *Description*, en valgfri beskrivelse av hva denne klassen er.
- *uri*, en tekststreng som brukes til å definere lytteporten OBS skal benytte for å finne mottatte meldinger fra RPi-en. Her er denne satt til "udp-listen port=4321".

- *Pi_IP_address*, IP-adressen til RPi-en som er koblet til joysticken.
- *debug_enabled*, skrur av og på feilsøkings-meldinger, dvs. informasjonsmeldinger som skrives til konsoll.
- *manual_control_active*, viser om fjernkontrollen har tatt kontroll over USV-en. Denne står som "true" hvis fjernkontrollen har tatt kontroll.
- *ipAllowedControl*, gi IP-fjernkontroll tillatelse til å ta kontroll over USV-en. Denne kan skrus av og på, men burde som regel alltid stå på, for å sikre at fjernkontrollen kan gripe inn og styre.
- *slowRemote*, viser om fjernkontrollen står i "slow" eller "fast"-modus, og er enten "true" eller "false".

Det som vises i denne menyen ble valgt både ut fra hva som viste seg å være nødvendig, og ting gruppen tenkte var nyttig å vite for en operatør. Dette ble bestemt løpende, ettersom gruppen kontinuerlig testet klassen i simulator, og senere ute i havnebasseng på fysisk USV. Enkelte funksjoner ble gjort tilgjengelig for å kunne justere variablene uten å måtte gå inn i kildekoden. Eksempler på dette er at man kan justere port og IP til RPi-en og mottaksport på USV rett i F9-menyen. Andre ting, som at man kan definere klient-id, er der for å lettere kunne identifisere at fjernkontrollen har tatt kontroll. Da kan man gjerne bruke et universelt nummer på USV-en slik at man raskt kan forstå at det er IP-fjernkontrollen som styrer. Feilsøking er i hovedsak lagt til for utvikling og simuleringsbruk, ikke for bruk av kunde. Denne bør eventuelt fjernes før produktet gjøres tilgjengelig kommersielt.

I VCS får man også en del tilbakemeldinger på USV-en og meldingene den mottar. Et skjermbilde av VCS med noen meldinger er vist under:



Figur 23: VCS med meldinger

Her ser man forskjellige "popups" i øvre venstre hjørne. Den øverste varsler om at VCS klient 1 har tatt kontroll over USV-en. De to neste varsler om at fjernkontrollen først er satt i "Fast"-modus, deretter i "Slow"-modus.

3.7 Docker

3.7.1 Oppgave

Bruk av Docker baserte seg på et ønske om å kjøre MR sin avanserte OBS kode på WAGO PFC200 kontrolleren. Dette var både for å ha et reservesystem på PFC-en, men også for å kunne samle alt av utregninger og operasjonsutførelse på en enhet. Dette minker kommunikasjonsproblemer mellom styringsenhetene på USV-en, ettersom alt av OBS kan styres fra PFC-en. Docker er da et nyttig verktøy, fordi man kan utvikle for mange forskjellige plattformer samtidig. På denne måten kan man kombinere høynivåkoding med en kontroller som er godt egnet for styringen av aktuatorer og annet på USV-en.

3.7.2 Mulige løsninger

Det var flere årsaker til at Docker ble valgt for å løse oppgaven, fremfor andre løsninger. Hovedgrunnen til at Docker ble valgt kommer av at det er et moderne program, som blir tatt i bruk av mange som jobber med utviklig. I tillegg kommer

programmet med mange fordeler, se 2.6.8 for eksempler. Det som er krevende med Docker er å sette seg inn i programmet, for å kunne dra best mulig nytte av dens funksjoner og nytteområder.

WAGO sin PFC200 kontroller har C integrasjon, som gjør det mulig å kode C++ via CODESYS (CODESYS Online Help, udatert). Dette var en mulig konkurrent for Docker løsningen, men ble fort lagt til side. En av årsaken var at løsningen med integrasjon av C moduler krevde at hele OBS koden måtte bli tilpasset bruk innenfor rammene til CODESYS, eller bli bygd på nytt. Det hadde blitt veldig tidskrevende og var ikke en løsning MR hadde dratt nytte av å ha, siden den kompliserer videreutvikling av systemet. En annen hindring var at bruk av C++ i CODESYS er en relativt manuell prosess, som betyr at man må kopiere over filer til PFC-en, deretter bygge disse pakkene (WAGO, 2022). Hovedgrunnen til at denne metoden ble valgt bort var ulempen med at det ville blitt to OBS, hvorav endring i den ene ville krevd ekstern endring av den andre. I tillegg ville det krevd mer arbeid på MR sin side å oppdatere disse på hver enkelt USV, enn å bare kunne hente ned et Image fra nettet, og starte opp Containeren. Med to OBS blir utvikling og oppdateringer av systemet mer innviklet, selv om disse er basert på de samme kildefilene. Løsningen krever flere PFC-er for testing av ny funksjonalitet, noe som er lite lønnsomt for bedriften.

Docker implementasjonen løser dette enkelt ved at det ikke er nødvendig å ha to OBS. Istedet kan man hente ned oppdatert versjon av OBS og laste opp OBS etter eventuelle utviklinger. Dette kan man gjøre med kommandoene "pull" og "push" (Docker Inc, udatert[a]). Docker gjør det også enkelt å gjøre endringer fra egen PC i forhold til CODESYS C integrasjonen. Programmering av Docker var også enklere å sette seg inn i på grunn av god dokumentasjon av programmets kommandoer, og muligheter.

En siste konkurrent var å bygge en enkel variant av OBS direkte på PFC-en. Her kunne man hentet ut de mest nødvendige delene av OBS til MR, og sette sammen et forenklet system. Denne løsningen kunne vært tidsbesparende i forhold til å bygge hele OBS med C integrasjonen på CODESYS. Løsningen var mindre ønskelig for MR, ettersom en forenklet OBS har mangler i forhold til det MR ønsket. Docker løsningen er altså mer aktuell enn denne, fordi den er godt egnet til storskala bruk. I tillegg er måten OBS oppdateres bedre egnet for videreutvikling av systemet.

3.7.3 Planlagt løsning

For å løse oppgaven var planen å pakke inn høynivåkoden til OBS på WAGO PFC-en, og deretter starte en Container på PFC-en. De første forsøkene på løsningen besto

av en Dockerfil med et Raspbian Image som "Base Image". Relevante filer ble kopiert over og lagt ved dette nye Imaget som ble bygget. Docker Imaget ble så publisert til skytjenesten Docker Hub, i et privat oppbevaringssted, og lastet ned på WAGO PFC200. Basisen for denne Dockerfilen var MR sin egen Dockerfil, som brukes til testing og utvikling på RPi internt. Denne ble tilpasset ved å importere hele OBS-prosjektet til MR, som ligger i en mappe ved navn *obs-qt*. Fremover vil *obs-qt* referere til hele prosjektet med filer, og OBS vil referere til det kjørende programmet OBS. Denne Dockerfilen beskrives i kodesnutt 25 i form av pseudokode.

```
FROM debian:buster-slim
USER root
#Imports MRs list of libraries and installs these
#Sets up system logs
COPY obs-qt obs-qt
RUN cd obs-qt/build && CMD ["./obs-qt", "-b", "/etc/obs/boot.js"]
#Runs the binary obs-qt file, the program version of OBS
```

Kodeutdrag 25: Planlagt løsning for Dockerfil, pseudokode

Det var strukturen i kodeutdrag 25 som ble brukt for å teste at et Docker Image kunne kjøre gjennom en Container på PFC-en. Det ble oppdaget at Containeren kjørte fint på gruppemedlemmenes egne PC-er, men klarte ikke å starte på PFC-en. Gruppen forsøkte også å bygge OBS med forskjellige "Base Images", som Buster varianten av Debian, Raspbian og Ubuntu. Disse ble forsøkt brukt fordi MR hadde spesifikke versjoner av OBS for disse Imagene. Dette fungerte heller ikke, og gruppen måtte finne ut hvorfor.

Forsøkene på å kjøre OBS på PFC-en gjennom Docker ble derfor satt på vent, og man gikk tilbake til å hente inn mer informasjon om Docker og PFC-en. På grunn av dette gikk de neste forsøkene på å gå fra konvensjonelt program til Containerløsning ble gjort sent i prosessen. I tillegg endret gruppen fokus over til å utvikle en variant av OBS som fungerte sammen med IP-fjernkontrollen. Dette fordi man i utgangspunktet skal kunne sende et program rett i Docker, uten store modifikasjoner, så fort det fungerte på en valgfri plattform. Slik hadde gruppen også sikret at man ihvertfall klarte å kommunisere mellom RPi-en og OBS. Hvis man fikk kommunikasjon til å fungere uten bruk av en Container, ville det være raskere og enklere å teste og feilsøke under utviklingen av en Container.

Gruppen opplevde flere feilmeldinger. Etter grundig feilsøking ble det oppdaget at dette kom av arkitekturforskjeller mellom WAGO PFC-en og alle tilgjengelige versjoner av OBS. Arkitekturen til WAGO PFC-en var ARM32v7, altså en 32-bits versjon av ARM sin v7 arkitektur. Alle byggene av OBS var bygd for 64-bit kompatibilitet. Dette forårsaket at Docker kommandoen for kjøring av Dockerfilen

med OBS, returnerte en Docker *exec format error*. Denne feilmeldingen hindret at Containeren med OBS kunne startes på PFC-en i det hele tatt.

Etter mye feilsøking fant gruppen ut at Raspbian-bygget av OBS hadde ARM kompatibilitet. Det ble derfor forsøkt å lage en Dockerfil med Raspbian-image. Dette imaget, med MR sitt Raspbian bygg av OBS gjorde det mulig å kjøre Containeren. Da kunne man slå fast at feilen kom av uforenlighet mellom arkitekturene til OBS og PFC-en. Raspbian bygget av OBS var for 64-bits arkitektur, mens PFC-ens arkitektur var begrenset til 32-bit kompatibilitet.

Kompatibilitetsproblemstillingen ble diskutert innad i gruppen og med flere fra Softwareavdelingen til MR. Det ble bestemt at den eneste løsningen for å kunne kjøre OBS på PFC-en, var å bygge OBS på nytt for riktig arkitektur. Det var flere som tvilte på at denne løsningen ville virke, og hvor tidkrevende denne løsningen ville bli. To mulige løsninger ble diskutert mellom gruppen og softwareavdelingen. Den første og mest lovvende løsningen var å bygge OBS for riktig arkitektur på egen PC, ved hjelp av en maskinvareemulator. Den andre og siste løsningen var å bygge OBS på PFC-en, noe som ville kreve mye tid og være veldig innviklet. Det var usikkerhet rundt om Linux utgaven på PFC-en var kraftig nok til å bygge, og kjøre OBS direkte på PFC-en. Dockerfilene var trege å bygge på vanlig PC, med en bygge-tid på omtrent 20 til 30 minutter. Prosessoren til PFC-en, ville brukt lenger tid på å bygge, ettersom den er tregere enn prosessoren til PC-ene til gruppen.

3.7.4 Endelig løsning og fremgangsmåte

Det ble bestemt at det var mest gunstig å bygge OBS for PFC-ens arkitektur på egen PC. Denne løsningen ble valgt istedenfor å bygge OBS direkte på PFC-en, grunnet fordelene med å kunne bruke Docker til utvikling og oppdatering av systemet. For å kunne bygge OBS på egen PC måtte gruppen få oversikt over alle bibliotekene OBS var avhengig av og installere de i Dockerfilen. Alle bibliotekene støttet ARMhf arkitekturen. Denne arkitekturen virket ikke å samsvare med arkitekturen til WAGO PFC200, ut ifra informasjon gruppen fant på internett. Ettersom de nødvendige bibliotekene ikke var tilgjengelige for arkitekturen man trodde det var, *ARMEL*, ble det tenkt at MR sitt OBS ikke kunne bygges på PFC-en. Etter mye diskusjon med MR og innad i gruppen, så det ut som det ikke var mulig å løse oppgaven på tenkt måte. Videre søking og tester tydet på at ARMhf arkitekturen hadde potensiale til å virke på PFC-en, selv om det ikke stod spesifisert at den hadde denne arkitekturen. Gruppen var nødt til å finne ut av dette for å være sikre om Docker løsningen var mulig å implementere. Etter en samtale med WAGO ble det formidlet at PFC-en hadde ARM32v7 arkitektur, samtidig som den hadde "hardware floating point"

støtte, altså det "hf" i ARMhf står for.

For å kunne bygge til riktig arkitektur på PFC-en, måtte aktuell arkitektur emuleres på egen PC gjennom Docker. For bruk i oppgaven ble QEMU installert på PC-en gjennom terminalen. PC-ens arkitektur var AMD x86_64, mens ønsket arkitektur for Containeren som skulle kjøres på PFC-en var ARM32v7. Ved å følge Nvidias brukerveileding (NVIDIA Corporation, 2023), ble ARM32v7 arkitekturen emulerbar på PC-en, og bygging av Dockerfilen til riktig arkitektur ble gjennomførbart. Uten QEMU ville ikke PC-en ha klart å kjøre en Docker Container ment for en annen datamaskinarkitektur. Med de to linjene i kodeutdrag 26 fra Nvidia ble plattform spesifiseringen kjørbar på PC-en.

```
$ sudo apt-get install -y qemu  binfmt-support  qemu-user-static

$ sudo docker run --rm --privileged multiarch/qemu-user-static \
--reset -p yes
```

Kodeutdrag 26: Installering av QEMU

Med QEMU installert, og kunnskapen om støtte for "hardware floating point" ble det mulig å bygge OBS for PFC-en. Det var noen bibliotek som hadde blitt satt sammen internt av MR. For å installere disse bibliotekene ble det først lagd .deb pakker på PC-en. Deb er filformatet til pakker laget for Debian Linux-distribusjonen (Wikipedia, udatert[b]).

Etter at alle de riktige bibliotekene var installert i Dockerfilen, kunne *obs-qt* legges inn som en del av Dockerfilen. Da dette var klart kunne man gå videre til å bygge selve OBS. Arbeidet ble gjort med bruk av VS Code, denne programvaren gjorde det enkelt å holde oversikt over prosjektet, kode og aksessere terminalen. for kjøring og kompilering av Dockerfilen. Dockerfilen inneholder alle kommandoer, for å kunne bygge det ønskede Docker Imaget og deretter kjøre Containeren.

3.7.5 Bygging og kjøring av Dockerfiler

For å bygge et Docker Image eller kjøre en Container, må man starte Docker daemon hvis denne ikke allerede kjører. Dette gjøres ved å skrive følgende i terminalen:

```
$ sudo systemctl start docker
```

Kodeutdrag 27: Start Docker daemon

i terminalvinduet, dersom Docker daemon ikke kjører fra før. Kommandoen som brukes for å bygge imaget og samtidig laste det opp til Docker Hub er:

```
$ docker buildx build --platform linux/arm/v7 -t \
*username*/*repository name*:*tag* --push .
```

Kodeutdrag 28: Bygging av Dockerfil

For å hente ned imaget brukes kommandoen "docker pull". Dette krever at man er logget inn på riktig bruker på enheten der Imaget skal hentes ned. For å logge inn på brukeren brukes "docker login." Imaget som hentes ned tilsvarer imaget som tidligere ble lastet opp til "repository" på Docker Hub. I gruppens tilfelle ble det lagd et privat "repository", slik at ingen utenom de med tilgang til brukeren kan finne og laste ned prosjektet, og Image fra dette. Det var viktig å ikke publisere arbeidet som ble gjort offentlig.

For å kjøre et Docker Image, kan man i et terminalvindu kjøre Docker kommandoen "docker run" med noen ekstra spesifiseringer:

```
$ docker run --platform linux/arm/v7 --rm -it --name obstestcontainer
    \
--network host *username*/*repository name*:*tag*
```

Kodeutdrag 29: Kjøring av Dockerfil

Kodeutdrag 29 henter ned Imaget fra Docker Hub dersom det ikke allerede ligger tilgjengelig på PC-en. Etter nedlasting kjører den Containeren og gir brukeren tilgang til kommandolinjen inne i Containeren. "-it" spesifiseringen i "docker run" kommandoen fra kodeutdrag 29, gjør at Containeren blir interaktiv.

3.7.6 Introduksjon til Dockerfilene

Prosessen for byggingen av OBS på PC for PFC krevde 3 ulike Dockerfiler, med forskjellige oppgaver. For å lage 3 ulike Dockerfiler måtte prosjektet ha flere mapper for hver del av prosessen i byggingen av OBS. Den første Dockerfilen ble lagd for å bygge Debian-pakker av ulike bibliotek fra MR sine prosjekter. Disse pakkene er designet og eid av MR. Det er også flere andre programmer og biblioteker som ligger åpent på internett, som OBS avhenger av for å kunne kjøre. De installeres i den andre Dockerfilen for å bygge OBS. Den tredje og siste Dockerfilen er for å kjøre OBS på PFC-en, her kopieres over alle nødvendige filer for simulering og OBS. Avsnittene under beskriver Dockerfilene mer detaljert.

3.7.7 Dockerfil for bygging av Debian-pakker

Dockerfilen begynner med å spesifisere plattform, samt "Base Imaget" som skal brukes. Brukerettighetene til brukeren av Docker Containeren spesifiseres i begynnelsen av Dockerfilen. Spesifiseringen av "Root" gir brukeren alle tillatelser.

```
FROM --platform=linux/arm/v7 arm32v7/debian:buster-slim
USER root
```

Kodeutdrag 30: Docker FROM og USER

I Dockerfilen installeres de avhengige programmene for bygging av .deb pakkene. Bibliotekene fra prosjektene makeVer, libqtlogutils og libais var ikke tilgjengelige uten tillatelse fra MR. Etter en brifing fikk gruppen tilgang til prosjektene hvor pakkene var tilgjengelige. Herfra ble de to første Dockerfilene lagd for å bygge de nødvendige pakkene. På denne måten kunne man lage Debian-pakker for alle de nødvendige avhengighetene.

Bygging av Debian-pakkene begynte med at de aktuelle bibliotekenes kildefiler ble kopiert til Docker Container, som en mappe med tilsvarende navn. Kodeutdrag 31 viser blant annet at de forskjellige bibliotekene blir kopiert fra PC-en til en ny mappe i Docker Container.

```
RUN apt update
RUN apt -y upgrade
RUN apt install -y devscripts
RUN apt install -y git
RUN apt install -y debhelper
RUN apt install -y qtbase5-dev
RUN apt install -y cmake
RUN apt install -y doxygen

COPY libqtlogutils libqtlogutils
COPY makeVer makeVer
COPY aisparser aisparser
```

Kodeutdrag 31: Kopierer internavhengige bibliotek

I Docker Containeren må bruker navigere til den nye *libqtlogutils*-mappen. "Devscripts" er nødvendig for å kjøre buildDeb, som lager .deb pakker av libqtlogutils- og makeVer bibliotekene. Med Docker kommandoen "docker cp" brukes for å kopieres Debian-pakkene fra Docker Containeren til ønsket plassering. Her var ønsket plassering prosjektmappen, og kopieringen med "docker cp" kommandoen, måtte utføres frakoblet Containeren, i et nytt terminalvindu. Det siste biblioteket

aisparser er avhengig av "doxygen" for å lage en Debian-pakke. Da alle Debian-pakkene var lag kunne prosessen gå videre til neste Dockerfil.

Byggingen av disse tre bibliotekene er ikke noe som må gjøres flere ganger, med mindre disse bibliotekene blir oppdatert av MR. Det er derfor ikke nødvendig å automatisere denne prosessen. Pakkene er en del av prosessen for å kunne lage en Debian-pakke av OBS, som brukes for å kjøre OBS. Etter .deb pakkene er lagd fra bibliotekene kan de kopieres over til et minnekort eller lagres i en av skylagringstjenestene til MR, for tilgjengelighet ved senere bruk.

3.7.8 Dockerfil for bygging av OBS binærfil og Debian-pakke

Den andre Dockerfilen ble lagd for bygging av en binærfil av OBS. En binærfil av OBS er en kjørbar OBS, som gruppen deretter kunne implementere i den siste Dockerfilen. Dockerfilen har lik struktur og oppbygging som den første Dockerfilen. Plattform og "Base Image", samt brukerettighetene til brukeren spesifiseres i starten av Dockerfilen, se kodeutdrag 30.

Videre installeres alle avhengighetene som kreves for å lage en kjørbar OBS-fil eller Debian-pakke. Først installeres de åpent tilgjengelige bibliotekene og programmene.

```
RUN apt-get update
RUN apt-get install -y debhelper
RUN apt-get install -y qtbase5-dev
RUN apt-get install -y qt5-qmake
RUN apt-get install -y libprotobuf-dev
RUN apt-get install -y libqt5serialbus5-dev
RUN apt-get install -y libqt5serialbus5-plugins
RUN apt-get install -y libqt5serialport5-dev
RUN apt-get install -y libarmadillo-dev
RUN apt-get install -y libproj-dev
RUN apt-get install -y libgeographic-dev
RUN apt-get install -y libssl-dev
RUN apt-get install -y qtdeclarative5-dev
RUN apt-get install -y libconfig++-dev
RUN apt-get install -y libxml2-dev
RUN apt-get install -y libmodbus-dev
RUN apt-get install -y libeigen3-dev
RUN apt-get install -y libnlopt-dev
RUN apt-get install -y qtscript5-dev
RUN apt-get install -y libsnmp-dev
RUN apt-get install -y cmake
RUN apt-get install -y git
RUN apt-get install -y devscripts
```

```
RUN apt-get install -y protobuf-compiler
```

Kodeutdrag 32: Installering av nøvendige avhengigheter

I kodeutdrag 32 ble det ikke brukt lagoptimalisering, som forklart i 2.6.7. Koden ble skrevet med flere lag og ikke bare ett, fordi det egnet seg bedre for feilsøking. I tillegg gir flere lag bedre oversikt over hvilke avhengigheter som blir installert i Dockerfilen. Det var under utviklingen av Dockerfilen, for å finne ut hva som gikk feil under bygging, at lagoptimaliseringen ble valgt bort. På denne måten var det enkelt å se hvilke pakker som ikke ble installert, samt tiden hver av pakkene brukte på å installere. Senere i Dockerfilen blir Debian-pakkene installert med lagoptimaliseringsmetoden.

```
COPY libqtlogutils-dev_0.5.2-1_armhf.deb .
COPY libqtlogutils0_0.5.2-1_armhf.deb .
COPY makever_0.8.0-1_all.deb .
COPY libais1_1.9.0-5_armhf.deb .
COPY libais-doc_1.9.0-5_all.deb .
COPY libais-dev_1.9.0-5_armhf.deb .

RUN apt-get update && apt-get install -y \
./libqtlogutils-dev_0.5.2-1_armhf.deb \
./libqtlogutils0_0.5.2-1_armhf.deb \
./makever_0.8.0-1_all.deb \
./libais1_1.9.0-5_armhf.deb \
./libais-doc_1.9.0-5_all.deb \
./libais-dev_1.9.0-5_armhf.deb

COPY obs-qt obs-qt
```

Kodeutdrag 33: Kopiering og installering av Debianpakker

For å installere de avhengige Debian-pakkene måtte alle de eksternt avhengige bibliotekene og filene til OBS kopieres til prosjektmappen. Alle avhengighetene kan ses i kodeutdrag 32. Til slutt måtte *obs-qt* prosjektet, som ble lastet ned internt fra MR, legges inn i prosjektmappen. Når alle avhengighetene var installert på riktig plass, og prosjektmappen hadde alle nødvendige filer kunne binærfilen og Debian-pakken lages.

Stegene for å lage binærfilen og Debian-pakken av OBS er forskjellige. For binærfilen lages en ny mappe som kalles *build* i *obs-qt*-mappen, og bruker navigeres inn i mappen for å kjøre CMake. CMake kompilerer og skriver byggefiler til mappen. Deretter kjøres kommandoen "make", som bruker en spesifisert mengde av prosessorens kjerner for å bygge binærfilen av OBS. Dette er filen man trenger for å kjøre OBS i siste Dockerfil. Kompileringen og byggingen av OBS binærfilen

tok omtrent en halvtime, noe som kan skyldes mengden avhengige bibliotek og filer. Kodeutdrag 34 viser kommandoen for bygging av .deb pakken. Bruker må plasseres i mappen som inneholder alle nødvendighetene for å bygge Debian-pakke av OBS. Debian-pakken som lages, kopieres til mappen hvor den siste Dockerfilen befinner seg.

```
$ debuild -uc -us -b -i
```

Kodeutdrag 34: Bygging av Debianpakke

Det er .deb pakken eller binærfilen av OBS som skal brukes på maskiner med ARM32 v7 arkitektur med ARMhf støtte. Binærfilen er lik MR sine versjoner av OBS, forskjellen ligger i programvarenes arkitektur. Den nye binærfilen har nå støtte for arkitekturen på PFC-en.

I likhet med den første Dockerfilen, er denne Dockerfilen en byggestein på veien mot målet. Binærfilen som blir lagd i Containeren er den kjørbare versjonen av OBS og ligger under *build* mappen til prosjektet. Da ligger binærfilen tilgjengelig for kjøring av OBS i neste Dockerfil.

Figur 24: Resulterende binærfil og Debian-pakke av OBS

3.7.9 Dockerfil for kjøring av OBS

Den siste Dockerfilen er den som brukes for å kjøre OBS på PFC-en og er den siste Dockerfilen i prosjektet. Første gang denne ble tatt i bruk, var det nødvendig å kopiere over alt fra mappen hvor forrige Dockerfil ble skrevet. For at OBS skal kjøre, må alle nødvendige avhengigheter ligge i mappen med denne Dockerfilen, se kodeutdragene 32 og 33.

Dockerfilen begynner med å spesifisere et "Base Image" man skal hente fra, i likhet med de tidligere Dockerfilene er dette debian:buster-slim for C++. I tillegg spesifiseres ønsket byggeplattform for Dockerfilen. Dette ble gjort slik at OBS kunne kjøre på en enhet med samme plattform. Brukeren ble satt til "Root", grunnet fordelene som følger.

Videre i Dockerfilen installeres alle tilgjengelige og nødvendige avhengigheter, som trengs for å kjøre en fullverdig versjon av OBS. Etter kopieres og installeres de essensielle filene for å kjøre OBS. Forskjellen fra forrige Dockerfil er at kun nødvendige filer for kjøring av OBS kopieres inn i Containeren. I tillegg kopieres *obs-qt* mappen i kodeutdrag 35. Den inneholder *build* mappen med OBS-binærfilen, men ikke noe av kildekoden.

```
COPY obs-qt obs-qt
COPY boot.js /etc/obs/
COPY speed2force.csv /etc/obs/
COPY properties.conf /etc/obs/
COPY /obs /var/lib/obs
```

Kodeutdrag 35: Kopierer filer for simulering og kjøring av OBS

Når Dockerfilen blir kjørt, kjøres OBS automatisk. Hvis det er ønskelig kan man kjøre OBS manuelt i Containeren. For å gjøre dette må man kommentere ut koden i kodeutdrag 36 og navigere til "build"-mappen der binærfilen ligger. Derfra kan man kjøre kommandoen spesifisert i CMD fra kodeutdraget.

```
WORKDIR /obs-qt/build
CMD ["./obs-qt", "-b", "/etc/obs/boot.js"]
```

Kodeutdrag 36: Kjører OBS automatisk ved åpning av Containeren

3.7.10 Effektivisering underveis

I forståelse- og utviklingsfasen så gruppen på hvordan "Base Image" kan utgjøre en stor forskjell på størrelsen, og "build-hastigheten" til det Imaget. Ved å bruke Alpine som "Base Image" istedenfor GCC, ble størrelsen på imaget betydelig mindre og "build-hastigheten" gikk ble betydelig raskere. For å teste optimalisering ble et Docker image hentet på PFC200 fra Docker Hub, hvor en besparelse på 1GB kom av å endre "Base Image" fra GCC til Alpine. Alpine er et "Base Image" med minimal størrelse, noe som gjør den lett å kompilere og bygge, i tillegg til at den gir større besparelser ved lagringskapasitet.

Senere ble det gjort klart at MR sitt OBS tidligere har vært bygget på et "Debian-image" for testbruk, og det var derfor ønskelig å benytte dette i denne brukssituasjonen også. Grunnen til det er at denne varianten allerede var bekreftet å fungere, og derfor ville være enklere å feilsøke i hvis det ble nødvendig. Dette førte til litt økning i størrelse på imaget, men det var fortsatt mye mindre enn ved de tidligste testene der GCC-image ble brukt.

I Docker Containeren

I Docker Containeren kan man navigere seg gjennom ulike filer og mapper, og kjøre det man ønsker. Containeren har like egenskaper som man finner i terminalen på Linux OS. Målet var at OBS binærfilen skulle kjøres, dersom det ikke ble gjort automatisk med CMD i Dockerfilen. Ved manuell kjøring av OBS må man navigere inn i *ob-qt* mappen og videre inn i "build" mappen, deretter kan man kjøre OBS med "./obs-qt -b /etc/obs/boot.js". Dette ble automatisert i hoved Dockerfilen, slik at OBS automatisk kjøres ved oppstart av Containeren.

3.8 Testing

3.8.1 PFC

Tidlig i prosjektfasen fikk gruppen tilgang til en testbenk med flere av WAGOs PFC200 (se figur 15) som ble brukt til å teste kommunikasjon mellom RPi og PFC, i tillegg til muligheten for å laste ned OBS på en PFC ved hjelp av Docker. Oppsettet av denne testbenken er gjort i henhold til WAGO Norge sin support nettside (Jansrud, 2021). Både ethernett-kabel og Wifi er benyttet under testingen.

For å opprette kommunikasjon mellom RPi og PFC ble det først utviklet et enkelt program i Codesys hvor meldingen "Hello world!" kunne sendes direkte mellom enhetene. Målet var å få sendte meldinger fra fjernkontrollen til OBS via PFC-en. Utover i prosjektet viste det seg at fjernkontrollen kunne kommunisere direkte med Docker Containeren hvor OBS-et kjører. Dette fjernet et ledd i kommunikasjonen som gjorde at Codesys programmet ikke lenger var nødvendig.

3.8.2 USV

Underveis i prosjektarbeidet har det vært behov for å teste koden som ble utviklet for styring av USV-en. Det har hovedsaklig blitt gjort med MRs simuleringsprogram. Fordelen med å simulere er at kode kan testes raskt og enkelt, i tillegg til at det er null risikoen for skade av matriell. Den klare ulempen er at det ikke har vært mulig å teste baugthruster siden den ikke er implementert i simuleringsprogrammet. I tillegg er ikke simulatoren helt lik virkeligheten, der man har usikkerhetsmoment som vind, bølger og strømmer. Simulatoren fungerer derimot for å teste at det meste av funksjonalitet er implementert og fungerer som det skal. Fysisk testing er da mer nyttig for å justere og tilpasse ettersom man oppdager utfordringer med å styre en fysisk USV.

Det ble gjennomført tester på USV-en mot slutten av prosjektarbeidet når gruppen var trygge på at koden fungerte som tiltenkt. Før egenprodusert programvare ble testet på USV-en fikk gruppen en gjennomgang av oppstartsrutiner og styring med MRs fjernkontroll. Dette ble gjort av sikkerhetshensyn, ettersom den fjernkontrollen kan overstyre og ta kontroll over USV-en ved eventuelle problemer. I tillegg fikk gruppen kjennskap til hvordan USV-en opptrådte i virkeligheten.

Første test av egenprodusert kode ble gjennomført med USV-en fortøyd. Det ble kontrollert at fjernkontrollen kunne skru av og på motoren, samt at manøvreringen fungerte. Det ble oppdaget at pådraget til motoren måtte skaleres ned for å kunne operere trygt i havneområdet. Lite utslag med joysticken resulterte i større pådrag enn forventet. I tillegg til dette ble det oppdaget at baugthrusteren ikke fungerte.

Etter at koden var revidert ble en ny test gjennomført hvor manøvreringen samt supplerende funksjonalitet ble testet med positive resultater. USV-en beveger seg nå med en hastighet på under 10 knop (18.52 km/t) i slow modus med skaleringsfaktoren *m_throttleScale* satt lik 0.2, og baugthrusteren fungerer som ønsket. Det ble også kontrollert at tilleggsfunksjonen *Station Keeping* fungerte likt som i simuleringen.

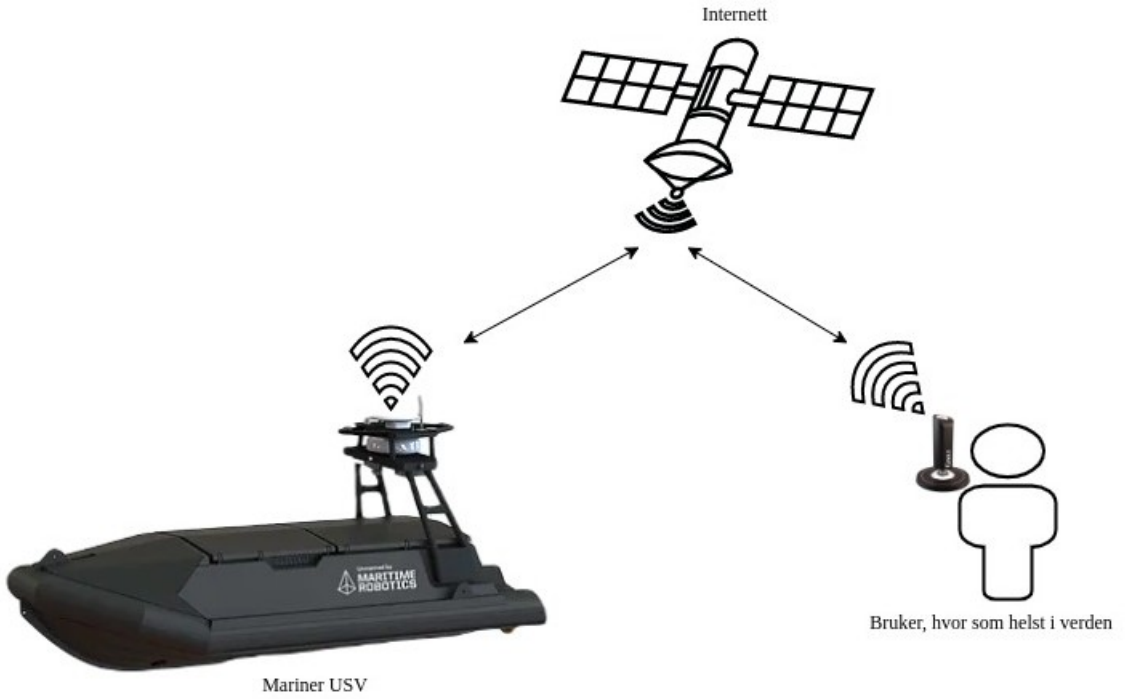
4 Resultater

Resultatene presenterer den endelige løsningen av prosjektoppgaven. Det ble utviklet og implementert en IP-basert fjernkontroll for styring av USV. I tillegg har gruppen kommet frem til at OBS kan kjøres på en PFC. Videre blir det gjennomgått hvordan fjernstyringssystemet er satt opp.

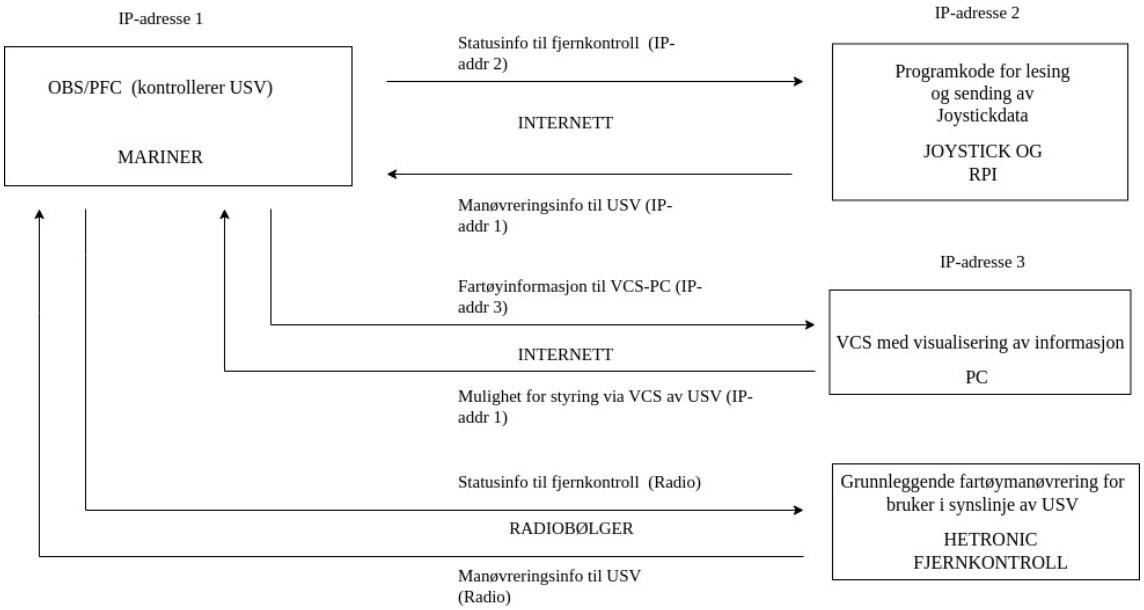
4.1 Ny systemarkitektur

Det nye systemet gruppen har utviklet er todelt. Det er utviklet en IP-basert fjernkontroll, samt lagt til rette for å kjøre OBS rett på PFC-en gjennom en Docker Container, i stedet for at dette kjører på en PC som deretter sender meldinger til PFC-en. Den nye systemarkitekturen er i grove trekk nokså lik den forrige. Man har fortsatt en PC og en PFC, og PC-en er den som står for vanlig, daglig drift. I tillegg er det tilrettelagt for å kjøre OBS på PFC-en, men det gjenstår mer arbeid før dette er gjennomførbart i praksis, og ikke bare i et simulert miljø hvor PFC-en er koblet til en PC. Et annet moment er at den utviklede fjernkontrollen koster mindre enn Hetronicen. Den totale komponentkostnaden for denne joysticken er omtrent NOK 8000. Prisen er bare en approksimasjon og den baserer seg på kostnadene av en RPi, joystick, diverse komponenter og produksjon av et kretskort. En rimeligere fjernkontroll kan være en fordel for en kunde som ønsker å ha flere av disse tilgjengelig, da terskelen for å kjøpe en ny eller en ekstra er mye lavere enn for Hetronicen. Det er viktig å påpeke at denne nye fjernkontrollen ikke nødvendigvis er tenkt som erstatning for Hetronicen, men en tredje mulighet i tillegg til denne og VCS.

Det mangler noen steg før dette systemet med Docker er fullt implementert på en USV, da dette per nå kun fungerer på testbenk. PFC-ens styreprogram, altså programmet som sender kommandoer ut til motorer og andre aktuatorer, må tilpasses slik at det kan lese meldinger som sendes lokalt fra PFC-en, i stedet for å få meldinger fra en annen enhet med en egen IP-adresse. Antagelig må man også skrive om litt av denne PFC-koden, da denne er basert på CODESYS 2.3 og ikke CODESYS 3.5, som er versjonen WAGO PFC200 i utgangspunktet har støtte for. I tillegg må man montere en WAGO PFC200 på en USV, da PFC-en som sitter på USV-ene i dag ikke har støtte for Docker. Derimot er det bevist at OBS kan kjøre på PFC ved å benytte Docker. En Docker Container som inneholder OBS er bygget og lastet ned på PFC-en på testbenken, og koblet til RPi med joystick og PC med VCS-simulering.



Figur 25: Den endelige kommunikasjonsarkitekturen mellom USV og den nye fjernkontrollen.



Figur 26: Flytskjema for kommunikasjon mellom forskjellige styringsenheter for Mariner USV med den nye fjernkontrollen inkludert.

Flytskjema for kommunikasjon er vist i figur 26. Her kan man se tre generelle styrings- og kommunikasjonsmuligheter med USV-en, alt ettersom hvor operatøren er. Alternativ 1 er at man bruker VCS for å legge inn en rute, sette kurs og

lignende, forutsatt at man har en form for internettilkobling. Alternativ 2 er å bruke Hetronic-fjernkontrollen, som krever sikt til fartøyet, hvor man kan detaljstyre RPM, retning, baugthrustere og pådrag. Alternativ 3 er gruppens nyutviklede IP-baserte fjernkontroll, hvor man, i likhet med VCS, trenger internett, og kan styre fra hvor som helst. Denne fjernkontrollen gir detaljstyring på nesten samme nivå som Hetronic-fjernkontrollen, hvor man kan styre retning, pådrag og baugthrustere, i tillegg til å sette USV i *Station Keeping* og endre maks- og minimumspådrag med joysticken. Disse tre styringsmulighetene supplerer hverandre, dvs. at de skal kunne brukes på samme USV ut ifra hva situasjonen krever.

5 Refleksjon og diskusjon

5.1 Fremtidige forbedringer og tillegg

Til tross for at det endelige produktet er en fullverdig joystick med nytt funksjonalitet, kan det fortsatt gjøres forbedringer. Det samme gjelder for Docker-delen av prosjektet, hvor det ligger fremtidig arbeid i å implementere det på en USV.

5.1.1 Forbedringer for fjernkontrollen

Først vil forbedringer på fjernkontroll og OBS-kode diskuteres. En klar forbedring vil være å legge til en skjerm på fjernkontrollen. Dette ville gjort at operatøren kan se informasjon i sanntid, hvis operatøren ikke har tilgang på en PC med VCS. Der kan man både vise meldinger mottatt fra OBS-et på USV-en, men også informasjon fra fjernkontrollen selv. Dette kan for eksempel være motorstatus og posisjon, som allerede sendes fra USV-en, men ikke vises annet enn i terminalen. I tillegg kan man også gi mer detaljerte feilmeldinger, for eksempel om man ikke får opprettet kommunikasjon, eller om innleste verdier er utenfor det man forventer. Det kan også være nyttig å legge til flere LEDs, slik at man kan ha flere varierte varslingssignal, og ikke bare må benytte seg av to LEDs for all varsling visuelt.

En annen forbedring ville vært å introdusere flere kommunikasjonslinjer som kjører samtidig. Dette kan for eksempel være at man sender over to frekvenser eller med to forskjellige antenner samtidig. Da ville man ha en sikkerhet hvis man skulle få et problem med maskinvare. Dette ville redusert sjansen for at en annen operatør må gripe inn, eller at USV-en går i *Station Keeping* som følge av kommunikasjonstap. Dette er lignende hvordan kommunikasjon med VCS allerede foregår, hvor man kan velge forskjellige kanaler å kommunisere med USV-en over, f.eks. 4G, Wifi og MBR. Dette ville gjort selve fjernkontrollen enda mer robust i bruk.

Det hadde også vært nyttig å sende enda mer informasjon mellom USV og fjernkontroll, for at operatøren skal ha enda bedre kontroll og oversikt. Dette kan for eksempel være hva slags type *Station Keeping* USV-en er i, batteristatus, dybdeinformasjon, eventuell integrasjon med påmonterte sensorer og lignende. Her er det egentlig bare fantasien og kundens ønsker som setter grenser. Det kunne også vært nyttig for operatøren å få et estimat på hvor lenge USV-en kan kjøre med nåværende drivstoff og batterinivå, og eventuelt også hvor lenge USV-en har vært på sitt nåværende oppdrag. Samtidig er det viktig å huske på at for mye informasjon som sendes vil kunne gi mer treghet i systemet, og potensielt føre til at USV-en

bruker flere sekunder på å reagere på en kommando. Derfor er det viktig å finne en balanse mellom hva operatøren bør eller vil vite, og hvor mye man maksimum vil sende. Man kunne også definert forskjellige meldingsformer, og basert på meldingens hode og fot bestemme hvilke funksjoner som skal kjøres på meldingen. Dette ville gjort integrasjon med andre USV-er som har andre pådragsorgan lettere, da man kan gi info i meldingen om man styrer en Mariner, Otter eller noe annet.

Til slutt er det noen forbedringer på hvordan fjernkontroll-koden er skrevet som kan gjøre videreutvikling lettere. Den første er å flytte koden over til et QT utviklingsmiljø, likt hvordan OBS-delen av prosjektet er programmert. Da får man tilgang til mer avanserte biblioteker. I tillegg blir det mer likt resten av MRs eksisterende kodebase, og vil nok derfor være lettere å sette seg inn i for de som allerede kan OBS og annen MR kode. I tillegg kan det vært lurt å benytte seg av protobuf-sending av meldinger fra fjernkontrollen, slik som man allerede bruker på å sende fra OBS til fjernkontrollen. Dagens løsning med å sende uten protobuf fungerer uten at man får feil, men hvis meldingene som sendes skal utvides, vil det være lurt å gjøre dette til protobuf. Dette er fordi man da vil ha et ekstra sikkerhetslag for å motta korrekte meldinger, i tillegg til at man lett kan endre meldingsstrukturen ved å modifisere .proto filen som benyttes av både fjernkontroll og OBS.

Avslutningsvis vil enkelte fysiske forbedringer på fjernkontrollen diskuteres. En stor forbedring for brukervennlighet ville vært å gjøre fjernkontrollen lettere å håndtere. I sin nåværende form består den av 3 forskjellige deler hvis man vil ha den med seg ut i felt. Man har selve joysticken, RPi-en med HAT og powerbank til å gi det hele strøm, i tillegg til alle ledninger mellom disse. En ideell forbedring hadde vært å samle strømforsyning og RPi med HAT til en enhet, som for eksempel kan festes til et belte eller lignende. Da trenger man ikke bære disse, og man kan lettere bevege armen til siden for å trykke på knapper og skyve på brytere på den festede enheten. Hvis man i tillegg får laget et slags feste til joysticken å man kan ha den på midjen og operere joystick med en hånd og knapper og lignende med den andre, har man et produkt som kan benyttes av en enkeltperson. Denne operatøren vil da også ha relativt stor frihet til å bevege seg rundt hvis situasjonen krever det.

I tillegg vil det være nyttig å ha større knapper og brytere på enheten, da dagens knapper og brytere er ganske små, og ikke enkle å bruke hvis man ikke ser på HAT-en med disse på. I tillegg burde hele enheten gjøres vann og støvtett, slik at man kan bruke dette i all slags vær. Dagens variant har RPi-en så og si helt åpen for elementene, og faren for at noe blir ødelagt under dårlig vær er stor. Her ville det vært naturlig å trekke inspirasjon fra fjernkontrollen, som har et mye mer robust design. Det ville også vært naturlig å ha større LEDs, disse kunne eventuelt vært festet på eller nære joysticken, slik at de er lett synlige for operatøren. Dette er

spesielt viktig hvis RPi med knapper skal festes i belte eller lignende, da dette ville gjort at dagens løsning med toppmonterte LEDs på RPi ikke ville vært særlig synlige. Det er også mulig å designe to varianter av denne IP-fjernkontrollen, en for bruk inne i kontrollrom, og en for bruk ute i felt. Da kan man tilpasse med for eksempel strømforsyning, festemetode og lignende, alt etter bruksområde.

5.1.2 OBS

Det er også litt tilleggsfunksjonalitet som kunne vært nytting å legge til, både på OBS-siden og på fjernkontrollsiden. På fjernkontrollsiden kunne det vært nyttig med mer detaljstyring, slik at funksjonalitetsnivået nærmer seg eller går forbi Hetronic fjernkontrollen. Dette kan for eksempel være at man kan endre hvilken type *Station Keeping* USV-en går i, og at man ikke bare kan gå i WOPC. I tillegg ville det vært nyttig å kunne styre RPM fra fjernkontrollen, da dette vil gjøre det lettere å svinge, uten at man må ha særlig pådrag fremover. Styring av skaleringsfaktoren i slow og fast modus kunne også vært nyttig funksjonalitet, slik at man ikke må inn i VCS for å endre denne i intervallet 0.2 til 0.5. Det kunne også vært nyttig å kunne skru av og på motoren, uten at det skrur av og på hele fjernkontrollen. Selv om dette kan oppnås til en viss grad med den nåværende versjonen av fjernkontrollen, ved å sette USV-en i "drift"-modus, der joysticken ikke vil styre, og man ikke har noe pådrag.

Det er også et par forbedringer på VCS og OBS-delen som kunne vært nyttig. Slik det er med den nåværende løsningen kan man ikke skru av og på slow modus med VCS, kun justere skaleringsfaktoren og se om slow modus er av eller på. Dette er en funksjonalitet som ikke er kritisk, siden operatøren med fjernkontrollen har muligheten til å skru slow modus av og på. Det er kun når fjernkontrollen brukes at slow vil kunne skrus på, ikke hvis VCS tar kontrollen. I tillegg hadde det vært nyttig for den som sitter ved VCS å få vite om USV-en og fjernkontrollen mister kommunikasjon. Dette kan man delvis se i dag, ved at USV-en automatisk går i *Station Keeping*, men man får ingen andre indikasjoner enn det. Man kan derfor implementere en "popup" som forteller VCS-operatøren at USV-en ikke har mottatt meldinger innenfor bestemt tidsintervall, og at USV-en går i *Station Keeping*.

5.1.3 Docker

Når det kommer til Docker delen av prosjektet, er vegen videre litt annerledes. Det viktigste som må gjøres der, er å gjøre klar en USV for bruk av Docker. Dette innebærer å tilpasse kode som kjører på PFC til å lese meldinger fra OBS som kommer fra samme enhet, og ikke fra en annen PC over Modbus TCP. Dette vil

antagelig medføre at hele kodebasen må løftes fra CODESYS V2.3 til CODESYS V3.5. CODESYS V2.3 er det nåværende programmet, mens CODESYS V3.5 er den versjonen som er støttet på PFC-ene som også har Docker støtte. For å utføre dette må man antagelig justere mange av funksjonene for V3.5, og det må da testes grundig at disse oppfører seg som tiltenkt. Den beste løsningen vil antagelig være å fortsatt benytte seg av TCP-tilkobling mellom OBS og PFC-kode, for å sikre at pakker blir overført riktig og i riktig rekkefølge. Dette er viktig for at USV-en ikke begynner å oppføre seg rart hvis man skulle få pakketap eller feil rekkefølge på pakkene. Samtidig må man da lytte på "localhost" på maskinen, i stedet for en ekstern tilkobling til en annen IP-adresse. Så fort dette mellomleddet mellom OBS og aktuatorer er satt opp og testet, vil OBS fungere akkurat som det gjør i dag, bare at det kjøres i en Docker Container, og ikke på en ekstern PC.

En annen mulig forbedring er å effektivisere OBS for bruk på PFC i Docker Container. Dette vil bety å fjerne overflødig funksjonalitet, som ikke er essensielt å ha i dette sekundærsystemet, og gjøre denne varianten av OBS så rask som mulig. Grunnen til at man eventuelt vil gjøre dette, må være at PFC-en sliter med mengden utregninger og kommunikasjon inn og ut som kreves av et ordinært OBS. Under testing ble det oppdaget at man fikk lang forsinkelse hvis man sendte mange pakker i sekundet inn og ut av OBS-et som kjørte i Docker Container på PFC-en. Dette skjedde for eksempel hvis feilsøkings-meldinger ble sendt, og forsinkelse opp mot 15 sekunder ble observert. Når man derimot skrudde av at feilsøkingsmeldinger skulle sendes, var responsen tilnærmet øyeblikkelig. Dette er et godt eksempel på at man tjener på å være sparsom på kommunikasjon til og fra slike systemer, som tross alt må ha lav responstid for å potensielt forhindre ulykker og lignende. Slik OBS-et i Docker endte opp på slutten av dette prosjektet, virker det som man kan benytte seg av fullt OBS med all funksjonalitet i en Container uten problem. Dette vil derimot ikke være helt sikkert før man har fått testet dette på en fysisk båt, hvor man gjerne vil få mer forsinkelse og lignende som følge av at det ikke er et simulert miljø.

Docker delen av prosjektet har flere automatiseringsforbedringer, spesielt når det gjelder Dockerfilene. Det ble til slutt tre Dockerfiler for å gjennomføre kjøringen av OBS på PFC-en. Løsningen var litt mer komplisert enn nødvendig, og innebærer mange steg. Noe som kunne forenklet dette ville vært å brukt Dockerkommandoer for å instruere Dockerfilene, i stedet for at bruker må skrive i terminalen i Containeren. De to første Dockerfilene som er til for å gjøre klart alt som er nødvendig for å kjøre OBS er ønskelig å automatisere. For eksempel kunne byggingen av Debian-pakkene blitt gjort direkte i Dockerfilen ved å spesifisere "WORKDIR" og ulike "CMD". For å kunne bruke Debian-pakkene i neste steg av prosessen må man finne navnet til filene som produseres, og kopiere de fra Containeren til riktig mappe. Dette er en større hindring, og det er ikke helt sikkert om det er mulig å fullstendig automatisere

begge prosessene i Dockerfilene. Dette er fordi disse Debian-pakkene ikke har samme navn hver gang de bygges. Uten kjent navn på .deb filene, samt at man må koble fra Containeren for å kopiere filene til riktig mappe på en server eller vertsmaskin vil det være vanskelig. Hvis man får til en automatisert løsning trenger brukeren kun å bygge og kjøre Docker Containeren, og resten skal ordnes av seg selv. Forbedringen sparer tid for utvikling og testing av nye versjoner. Automatisering denne delen vil gjøre det lettere for MR å implementere løsningen på USV, fordi man da kan laste ned automatisk bygd OBS.

Docker løsningen kom med sine fordeler og ulemper underveis i arbeidet. Det var bra dokumentasjon rundt å skrive Dockerfiler, og hvilke kommandoer og parametere som var tilgjengelig for bruk var lett å finne på Docker sine nettsider. På den andre siden var det ikke så mye god dokumentasjon på hvordan implementere Docker på PFC-en. Det var et par videoer som kom til nytte i utviklingen av den tenkte løsningen. Fordelene med å bruke Docker er mange, så selvom det kan ta litt tid å sette seg inn i programvaren, kommer det til god nytte. Det vil også være en stor fordel å kunne utnytte PLS-er med Docker støtte, ettersom innpakkingen av avansert kode på PLS-er kan gjøre maskinene mer selvstendige og kraftfulle. Nytten av løsningen øker når den blir mer automatisert. En av ulempene med løsningen kommer av mengden avhengige steg for å nå det ønskede målet. I tillegg til dette gikk mye av tiden til feilsøking rundt oppsettet av Containeren, innpakkingen av OBS og innholdet i Dockerfilene. Til tross for dette ble OBS kjørt via Docker Container på PFC-en. Når man setter seg godt inn i Docker, ser man fort at mulighetene for videreutvikling er mange, dette kan også åpne opp for nye idéer som kan føre til ny utvikling.

Et annet viktig moment ved implementeringen av OBS i en Docker Container er at all data som ligger i en Docker Container blir slettet hvis Containeren slettes (Copes, 2020). Dette kan være problematisk, for eksempel for konfigurasjonsfiler og loggfiler, som man gjerne vil ha tilgang på i ettertid, eller ikke vil endre hver gang man starter en Container. Slik dagens løsning er, vil ikke disse filene lagres hvis man endrer dem. Dette kan gå fint hvis man setter opp alt helt riktig før man bygger Docker imaget, men dette fungerer dårlig når man skal bruke samme Docker image på flere forskjellige USV-er, med forskjellige sensorer og spesifikasjoner. Løsningen på dette vil være å benytte seg av "bind mounts" eller "volumes", som er en måte å knytte en filplassering på vertsmaskinen til Docker Containerens egen filstruktur (Copes, 2020). Da vil alt leses og skrives til en permanent plassering på vertsmaskinen, og ikke slettes hver gang Docker Containeren fjernes og startes på nytt. Dette kan også være nyttig hvis man må endre ting underveis, som hvis man har en annen IP for fjernkontrollen, og vil at dette skal huskes selv om PFC-en som kjører Docker Containeren skrus av og på.

5.2 Arbeidsprosessen og refleksjoner rundt denne

Det ble tidlig i arbeidsprosessen klart at bacheloroppgaven ville kreve mye tilegning av kunnskap som lå utenfor det gruppen hadde lært i løpet av studiet. Gruppen hadde god grunnkunnskap om PLS-er og deres anvendelsesområder i tradisjonell industri, og hadde også kjennskap til datakommunikasjon, systemarkitekturer og litt C++-programmering. Dette la grunnlaget for å kunne tilegne seg ytterligere kunnskap om disse fagområdene, og bruken av disse til å designe et produkt. Det var derimot også en del nytt, da spesielt på programmeringssiden. Det krevde mye arbeid å sette seg inn i MRs kodebase, og utvide kunnskapen om C++. I tillegg til hvordan man kan benytte seg av dette programmeringsspråket for kommunikasjon og interaksjon med eksisterende kode. Før man kunne begynne arbeidet med å lage fjernkontrollen og integrere denne inn i OBS, var det nødvendig å forstå hvordan OBS fungerte. Dette var tidskrevende, men ga også nyttig erfaring i hvordan slike systemer bygges opp, og hvordan gruppen burde bygge sin løsning.

Et annet aspekt ved oppgaven som var ukjent for gruppen, var Docker. Slik kryssplattform-kompilering og programmering var ukjent for gruppen, i tillegg til at Docker i seg selv er et ganske abstrakt konsept for noen som ikke er utdannet innenfor dataingeniør-faget. Arbeidsoppgavene relatert til dette var preget av kontinuerlig informasjonsinnhenting og testing. Dette har vært veldig lærerikt, både Docker spesifikt, men også den mer dataorienterte delen av elektroingeniøryrket. Dette er kunnskap gruppen ser for seg vil være meget nyttig både for videre studier og i arbeidslivet. I tillegg har det også vært interessant og lærerikt å se andre bruksområder for PLS enn de tradisjonelle man har hørt om i undervisningssammenheng, og viser også potensialet PLS-er har.

5.2.1 Arbeidspakker

Arbeidspakker	Sum antall timer	% av planlagte timer
1. Forprosjekt	127.00	105.83
2. Skrive rapport	354.00	136.15
3. PLS og CODESYS	27.00	108.00
4. Forstå Docker	54.00	135.00
5. Kom. mellom fjernkontroll og RPi	42.00	105.00
6. Kom. mellom RPi og PLS	184.00	99.46
7. I/O fra RPi til PLS	152.50	84.72
8. Docker sammen med CODESYS	113.50	94.58
9. C++ kode fra MR i Docker	231.50	77.17
10. I/O Docker Container	87.00	48.33
11. Testbenk / simulering	284.00	88.75
12. Testskjema	54.00	54.00
13. Test på USV	140.50	87.81
14. Lese rapport	92.00	115.00
15. Presentasjonsarbeid	0.00	0.00
16. Admin	153.50	100.00
Totalt	2096.50	95.30

Tabell 1: Oversikt over fordelingen av timer til hver arbeidspakke

I tabell 1 ligger en oversikt over timetallet som ble brukt på de forskjellige arbeidspakkene, og hvor mange prosent av planlagte timer som ble benyttet. Man kan se at en del arbeidspakker ble overskredet, mens andre ikke hadde behov for alle timene. Dette var forventet fra gruppens side, da det er vanskelig å planlegge store prosjekter, spesielt fordi man ikke har gjort så mange før. Dette viser også tydelig at man sjelden vet arbeidsmengden på enkelte arbeidsoppgaver, og man må forvente avvik. Det totale timetallet ble derimot nærme det forventede tallet på 2200 timer. Man klarer altså lettere å tilnærme tidsbruken til et helt prosjekt, enn enkeltoppgaver.

Gruppen har også tatt mye lærdom av å måtte arbeide mer selvstendig, og uten tidligere eksempler å lære fra. Tidligere i utdanningsløpet har man som regel hatt tilgang på like eller lignende problemer og løsninger som det man selv har jobbet med. Under utførelsen av oppgaven har gruppen måtte vært nyskapende og oppfinnsomme.

6 Konklusjon

Bacheloroppgaven har resultert i et komplett IP-basert fjernstyringssytem for en USV og den beviser at MR sitt ombordsystem (OBS) kan kjøres på en PFC ved hjelp av Docker. Fjernstyring av ubemannede overflatefartøy over internett har skapt nye muligheter siden fartøyet nå kan styres fullstendig fra et kontrollrom. Maritime Robotics sitt OBS er implementert i en Docker Container-løsning, hvor OBS kan kjøres på PFC-en og slik samle all oppgaveutførelse på en maskin.

Fjernstyringssystemet for USV-en er implementert på en adekvat måte som utfyller de eksisterende styringsmulighetene. Den nye fjernkontrollen kan benytte seg av avanserte funksjoner fra OBS, noe som ikke har vært mulig tidligere. Gruppens løsning med Docker åpner mulighetene for å kjøre OBS på alle maskiner med støtte for Docker. Dette i tillegg til friheten i arbeidslokasjon gjør arbeidet innovativt.

I tillegg til alle fordelene med sluttproduktet av bacheloroppgaven peker også rapporten på framtidige muligheter for videreutvikling av systemet. Viktige element som er nevnt her er blant annet styring av RPM og tillegg av skjerm for å forbedre tilbakemeldingene til operatør.

Gruppen er fornøyd med resultatet av prosjektoppgaven og har fått svært gode tilbakemeldinger fra veiledere og andre involverte. Arbeidet har vært spennende, utfordrende og lærerrikt.

Kildeliste

ankita_saini (2021). *Introduction to Linux Operating System*. URL: https://www.geeksforgeeks.org/introduction-to-linux-operating-system/. (hentet: 01.04.2023).

Arora, Simran (2022). *Docker vs. Virtual Machines: Differences You Should Know*. URL: https://cloudacademy.com/blog/docker-vs-virtual-machines-differences-you-should-know. (hentet: 22.02.2023).

Bikram (2021). *Docker Objects*. URL: https://bikramat.medium.com/docker-objects-e561f0ce3365. (hentet: 21.05.2023).

Cegal (udatert). *Operativsystem*. URL: https://www.cegal.com/no/ordbok/operativsystem. (hentet: 17.02.2023).

Choudhary, Ashish (2021). *Understanding Docker Networking*. URL: https://earthly.dev/blog/docker-networking/. (hentet: 05.03.2023).

CODESYS Group (2023). *THE COMPREHENSIVE SOFTWARE SUITE FOR AUTOMATION TECHNOLOGY*. URL: https://www.codesys.com/the-system.html. (hentet: 04.04.2023).

CODESYS Online Help (udatert). *Integrating C modules*. URL: https://help.codesys.com/api-content/2/codesys/3.5.13.0/en/_cds_integrating_c_code/. (hentet: 02.05.2023).

Copes, Flavio (2020). *How to access files outside a Docker container*. URL: https://flaviocopes.com/docker-access-files-outside-container/. (hentet: 15.05.2023).

Dictionary.com (udatert). *virtual machine*. URL: https://www.dictionary.com/browse/virtual-machine. (hentet: 20.05.2023).

Digi-Key (udatert). *SML-LXT0805GW-TR*. URL: https://www.digikey.no/en/products/detail/lumex-opto-components-inc/SML-LXT0805GW-TR/304367. (hentet: 15.05.2023).

Docker Inc (udatert[a]). *docker pull*. URL: https://docs.docker.com/engine/reference/commandline/pull/. (hentet: 08.05.2023).

— (udatert[b]). *Docker run*. URL: https://docs.docker.com/engine/reference/commandline/run/. (hentet: 18.05.2023).

— (udatert[c]). *Dockerfile reference*. URL: https://docs.docker.com/engine/reference/builder/. (hentet: 18.05.2023).

— (udatert[d]). *Multi-platform images*. URL: https://docs.docker.com/build/building/multi-platform/. (hentet: 18.05.2023).

— (udatert[e]). *Use containers to Build, Share and Run your applications*. URL: https://www.docker.com/resources/what-container/. (hentet: 18.05.2023).

— (udatert[f]). *Use the Docker command line docker*. URL: https://docs.docker.com/engine/reference/commandline/cli/. (hentet: 18.05.2023).

Exterman, Dori (2021). *Top 7 Open Source C++ Build Systems*. URL: https://www.incredibuild.com/blog/top-7-open-source-c-build-systems. (hentet: 12.04.2023).

Flade, Joerg (2020). *Docker — What it is, How Images are structured, Docker vs. VM and some tips (part 1)*. URL: https://ragin.medium.com/docker-what-it-is-how-images-are-structured-docker-vs-vm-and-some-tips-part-1-d9686303590f. (hentet: 17.03.2023).

GeeksforGeeks (2021). *TCP 3-Way Handshake Process*. URL: https://www.geeksforgeeks.org/tcp-3-way-handshake-process/. (hentet: 08.04.2023).

— (2023a). *TCP/IP Model*. URL: https://www.geeksforgeeks.org/tcp-ip-model/. (hentet: 16.05.2023).

— (2023b). *Types of Network Protocols and Their Uses*. URL: https://www.geeksforgeeks.org/types-of-network-protocols-and-their-uses/. (hentet: 09.04.2023).

Google LLC (udatert). *Protocol Buffers*. URL: https://protobuf.dev/. (hentet: 20.05.2023).

gurukiranx (2021). *How to use make utility to build C projects?* URL: https://www.geeksforgeeks.org/how-to-use-make-utility-to-build-c-projects/. (hentet: 06.04.2023).

IBM (2021). *Docker base image*. URL: https://www.ibm.com/docs/en/order-management-sw/9.5.0?topic=docker-base-image. (hentet: 02.04.2023).

Inductive Automation (2020). *What is a PLC?* URL: https://www.inductiveautomation.com/resources/article/what-is-a-PLC. (hentet: 27.03.2023).

Jansrud, Thorgrim (2021). *Oppdatering av 750-8xxx firmware via minnekort*. URL: https://www.wagonorgesupport.com/post/750-8x0x_firmware_upgrade_note. (hentet: 11.05.2023).

Kasireddy, Preethi (2016). *A Beginner-Friendly Introduction to Containers, VMs and Docker*. URL: https://www.freecodecamp.org/news/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b/. (hentet: 05.03.2023).

Kerner, Sean Michael (2021). *Internet Protocol (IP)*. URL: https://www.techtarget.com/searchunifiedcommunications/definition/Internet-Protocol. (hentet: 16.05.2023).

Kisller, Edward (2023). *A Beginner's Guide to Understanding and Building Docker Images*. URL: https://jfrog.com/knowledge-base/a-beginners-guide-to-understanding-and-building-docker-images/. (hentet: 17.03.2023).

Lenka, Chinmoy (2023). *Introduction to C++ Programming Language*. URL: https://www.geeksforgeeks.org/introduction-to-c-programming-language/. (hentet: 06.04.2023).

Maritime Robotics (udatert). *The Mariner*. URL: https://www.maritimerobotics.com/mariner. (hentet: 14.05.2023).

Mosaic Documentation Web (udatert). *GPIO Electrical Specifications*. URL: http://www.mosaic-industries.com/embedded-systems/microcontroller-projects/raspberry-pi/gpio-pin-electrical-specifications#rpi-gpio-input-voltage-and-output-current-limitations. (hentet: 15.05.2023).

NVIDIA Corporation (2023). *Running Cross-Architecture Containers*. URL: https://docs.nvidia.com/datacenter/cloud-native/playground/x-arch.html. (hentet: 12.05.2023).

Palak Jain 5 (2022). *Differences between TCP and UDP*. URL: https://www.geeksforgeeks.org/differences-between-tcp-and-udp/. (hentet: 02.03.2023).

pp_pankaj (2022). *Kernel in Operating System*. URL: https://www.geeksforgeeks.org/kernel-in-operating-system/. (hentet: 05.03.2023).

Raspberry Pi Foundation (udatert). *What is a Raspberry Pi?* URL: https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/. (hentet: 06.04.2023).

Singh, Abhishek (2021). *Raspberry Pi 4 Specifications Pin Diagram and Description*. URL: https://www.hackatronic.com/raspberry-pi-4-specifications-pin-diagram-and-description/. (hentet: 19.05.2023).

SUSE (udatert). *QEMU Overview*. URL: https://documentation.suse.com/sles/12-SP4/html/SLES-all/cha-qemu-overview.html. (hentet: 18.05.2023).

Taylor, Craig (2020). *User Datagram Protocol (UDP)*. URL: https://cyberhoot.com/cybrary/user-datagram-protocol-udp/. (hentet: 11.05.2023).

WAGO (2022). *WAGO/pfc-firmware-sdk*. URL: https://github.com/WAGO/pfc-firmware-sdk. (hentet: 21.05.2023).

— (udatert). *Control Included: Embedded Linux*. URL: https://www.wago.com/global/embedded-linux. (hentet: 20.05.2023).

Wago (2018). *Embedded Linux*. URL: https://www.wago.com/global/open-automation/modular-software/linux. (hentet: 13.05.2023).

— (udatert). *Controller PFC200; 2nd Generation; 4 x ETHERNET*. URL: https://www.wago.com/global/plcs-%5C%E2%5C%80%5C%93-controllers/controller-pfc200/p/750-8210#&gid=1&pid=2. (hentet: 12.05.2023).

Wikimedia Commons (2017). *File:ISO C++ Logo.svg*. URL: https://commons.wikimedia.org/wiki/File:ISO_C%2B%2B_Logo.svg. (hentet: 20.05.2023).

Wikipedia (2012). *File:Qemu logo.svg*. URL: https://en.wikipedia.org/wiki/File:Qemu_logo.svg. (hentet: 20.05.2023).

— (2023). *UDP*. URL: https://no.wikipedia.org/wiki/UDP. (hentet: 19.04.2023).

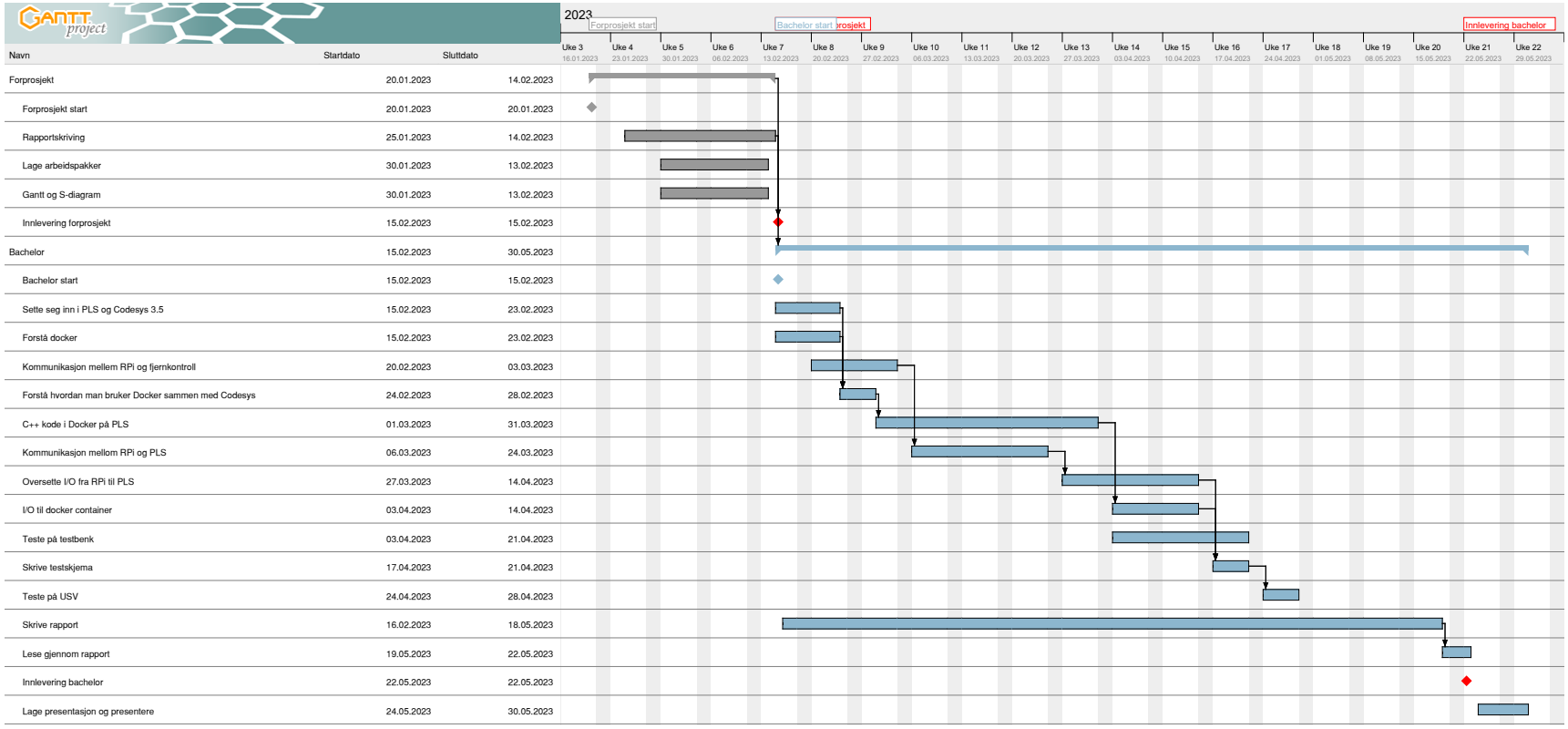
— (udatert[a]). *CMake*. URL: https://en.wikipedia.org/wiki/CMake. (hentet: 03.05.2023).

— (udatert[b]). *deb (file format)*. URL: https://en.wikipedia.org/wiki/Deb_(file_format). (hentet: 19.05.2023).

— (udatert[c]). *Make (software)*. URL: https://en.wikipedia.org/wiki/Make_(software). (hentet: 03.05.2023).

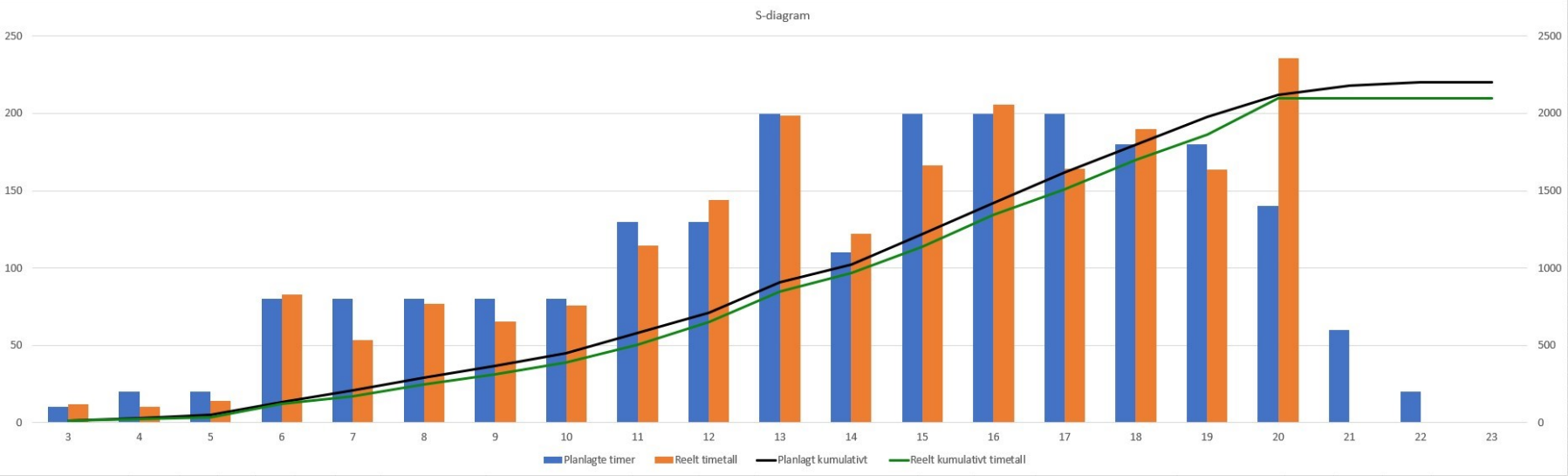
Wikipedia (udatert[d]). *QMake*. URL: https://en.wikipedia.org/wiki/Qmake. (hentet: 07.05.2023).
— (udatert[e]). *Qt*. URL: https://no.wikipedia.org/wiki/Qt. (hentet: 10.05.2023).
— (udatert[f]). *Qt Creator*. URL: https://en.wikipedia.org/wiki/Qt_Creator. (hentet: 10.05.2023).
— (udatert[g]). *Transmission Control Protocol*. URL: https://en.wikipedia.org/wiki/Transmission_Control_Protocol. (hentet: 20.05.2023).
Wilson, B. og Khandelwal, S. (2021). *How to Reduce Docker Image Size: 6 Optimization Methods*. URL: https://devopscube.com/reduce-docker-image-size/. (hentet: 07.04.2023).

A Gantt-diagram



Navn	Startdato	Sluttdato
Forprosjekt	20.01.2023	14.02.2023
Forprosjekt start	20.01.2023	20.01.2023
Rapportskriving	25.01.2023	14.02.2023
Lage arbeidspakker	30.01.2023	13.02.2023
Gantt og S-diagram	30.01.2023	13.02.2023
Innlevering forprosjekt	15.02.2023	15.02.2023
Bachelor	15.02.2023	30.05.2023
Bachelor start	15.02.2023	15.02.2023
Sette seg inn i PLS og Codesys 3.5	15.02.2023	23.02.2023
Forstå docker	15.02.2023	23.02.2023
Kommunikasjon mellem RPi og fjernkontroll	20.02.2023	03.03.2023
Forstå hvordan man bruker Docker sammen med Codesys	24.02.2023	28.02.2023
C++ kode i Docker på PLS	01.03.2023	31.03.2023
Kommunikasjon mellom RPi og PLS	06.03.2023	24.03.2023
Oversette I/O fra RPi til PLS	27.03.2023	14.04.2023
I/O til docker container	03.04.2023	14.04.2023
Teste på testbenk	03.04.2023	21.04.2023
Skrive testskjema	17.04.2023	21.04.2023
Teste på USV	24.04.2023	28.04.2023
Skrive rapport	16.02.2023	18.05.2023
Lese gjennom rapport	19.05.2023	22.05.2023
Innlevering bachelor	22.05.2023	22.05.2023
Lage presentasjon og presentere	24.05.2023	30.05.2023

B S-diagram


S-diagram
250
200
150
100
50
0
2500
2000
1500
1000
500
0
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
Planlagte timer
Reelt timetall
Planlagt kumulativt
Reelt kumulativt timetall

C Testskjema

Test IP fjernkontroll og OBS for Mariner, OBS på Docker

Testskjema E2312

Innholdsfortegnelse

Innholdsfortegnelse 1
1 Utstyrsliste 2
2 Kort forklaring av funksjonalitet 2
3 Klargjøring før fysisk test 4
4 Tester i havnebasseng 5
4.1 Ta over kontroll av USV 5
4.2 Manøvrering med Joystick 6
4.3 Stationkeeping 7
5 Tester i fjorden 8
5.1 Test av slow/fast modus 8
6 Tester i simulator 9
6.1 Kommunikasjonsbrudd ved nettverksproblemer 9
6.2 Kommunikasjonsbrudd ved tap av joystick 10

1 Utstyrsliste

Utstyret som trengs for å gjennomføre testene er:

1. RPI med PCB
2. USV (Unmanned Surface Vehicle) av typen Mariner
3. Joystick (Lilaas)
4. VCS (Vehicle Control Station)
5. Egenutviklet OBS

2 Kort forklaring av funksjonalitet

Her vil fjernkontrollens funksjonalitet forklares kort, og enkelte momenter som er nyttige å vite om.

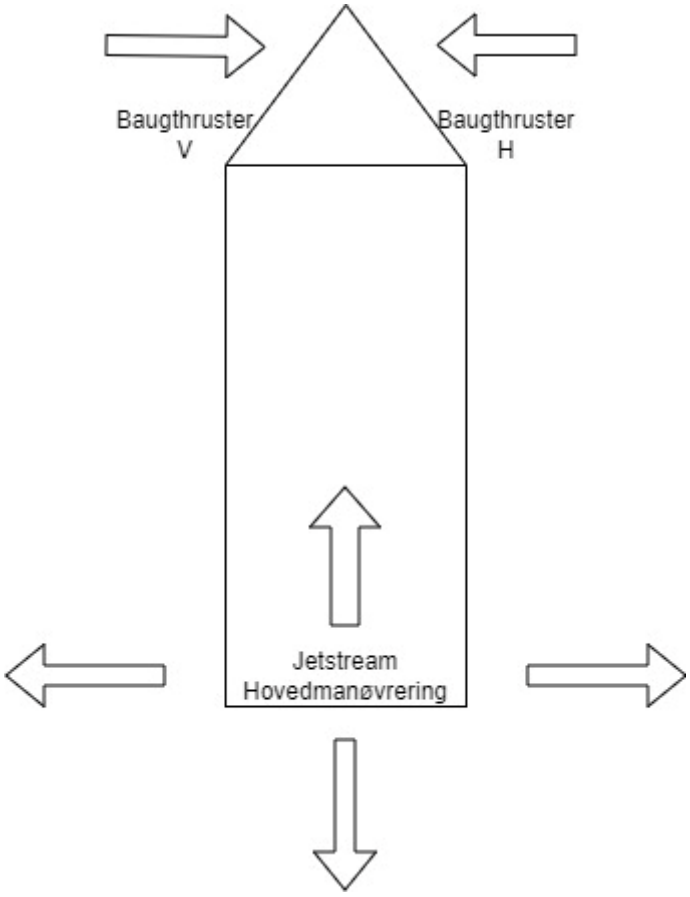
Station Mode, Station Keeping: USV settes til å ha en fast posisjon, ut ifra nåværende posisjon og fart. Dette gjøres generelt ikke i havnebasseng, da det er optimalt for USV å ha litt areal for å manøvrere. Det er mulig å gjøre det i havnebasseng, men da burde man ha god plass og utvise aktsomhet. Dette er fordi USV regner ut plassering og pådrag basert på vær og egen hastighet, og beveger seg på egen hånd når posisjonen stilles inn.

Manøvrering: USV får kommandoer fra joystick for pådrag fra $-100 \rightarrow 100$, hvor 0 er stillestående. Bevegelse mot/fra kroppen med joystick tilsier bevegelse bakover/fremover. Bevegelse venstre/høyre med joystick tilsier sving «på stedet» mot venstre/høyre. En kombinasjon av disse bevegelsene vil gi summert bevegelse i begge retninger. Rotasjon av joystick, såkalt «yaw», vil aktivere baugthruster til i venstre/høyre retning ved rotasjon i samme retning. Bevegelse av joystick venstre/høyre vil bevege USV-ens bakende mot denne retningen. Rotasjon av joystick, dvs. aktivering av baugthruster, vil bevege USV-ens front til venstre/høyre. Viktig: baugthruster burde ikke brukes mye, og ikke i høy fart.

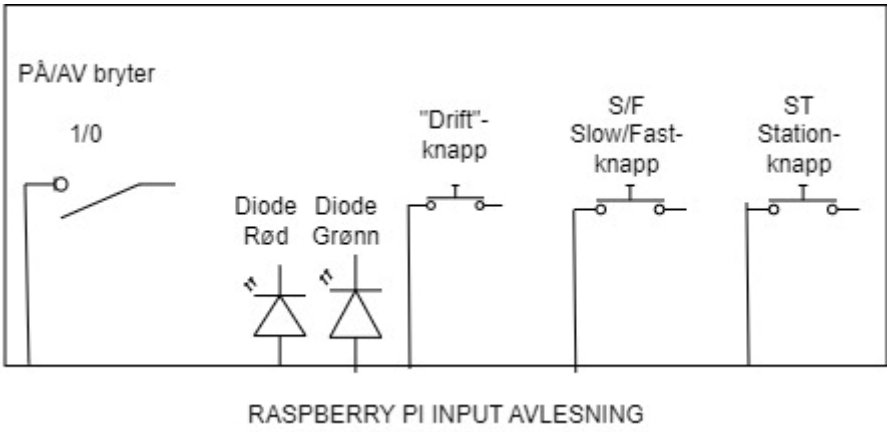
Slow/Fast mode: Kontrolleren er utstyrt med en knapp markert med S/F. Dette betyr slow og fast, og sier noe om maks mulig pådrag ut av $0 \rightarrow \pm 100$ i hver retning. Hvis fast mode er på er regnestykket som følger: $Skaleringsfaktor \cdot mottatt\ verdi \cdot 2$. Maks mulig pådrag her er 100%. For slow på har man: $Skaleringsfaktor \cdot mottatt\ verdi \cdot 1$. Maks mulig pådrag her er 50%. Skaleringsfaktoren stilles i VCS vha. F9 menyen «udp-remote-interface», og kan ha en verdi mellom 0.2 og 0.5. Dvs. at minimumspådrag i slow er 20% og i fast er 40%.

Dioder:

- Konstant rødt lys: Kontroller av
- Blinkende rødt lys: Alarm
- Blinkende grønt lys: “Slow” modus aktivert
- Konstant grønt lys: “Fast” modus aktivert
- Både rødt og grønt lys: Station keeping aktivert



Oversikt over bevegelsesaktuatorer og deres mulige bevegelsesretninger



Oversikt over PCB og dens funksjoner

3 Klargjøring før fysisk test

Klargjøring for test	
Steg/punkt	Kommentar
1. Daglig sjekk av USV.	Sjekke at alt på båten er i orden og at den er klar for drift.
2. Fjernkontroll og USV må kobles til samme nettverk.	Kontroller zerotier, ping Mariner 6 fra RPi.
3. Riktig IP-adresse til fjernkontroll og USV må legges inn i programkoden som kjøres på de ulike enhetene. Det samme gjelder for VCS IP.	Sjekk kontroller-kode og F9 menyen på VCS.
4. Sørg for at programmet kjøre som normalt og at joystick ikke er koblet til RPI.	Ved å koble til joystick, vil fjernkontrollen kunne påvirke Mariner. Dette er ikke ønsket før alt er klart til demo.
5. Sikkerhetsbrief.	Alle deltakere må en innføring i trygg bruk av systemet, og de faremomentene som kan oppstå.
6. Joystick kobles til Rpi og må stå i senterposisjon (alle aksene på joysticken er i 0).	For at fjernkontrollen skal ha mulighet til å ta kontroll over USV må joystick stå i startposisjon. Dette er en sikkerhetsmekanisme for å forhindre at USV får instruks om å flytte seg med en gang brukeren skrur på fjernkontrollen.

4 Tester i havnebasseng

Enkelte tester passer seg best i havnebassenget for godt overblikk.

4.1 Ta over kontroll av USV

Test: Ta over kontroll av USV		
Beskrivelse: Det skal fremstå av testen at brukeren av systemet kan overta kontroll over USV-en. PS! VCS kan alltid overta kontroll fra denne fjernkontrollen, men dette må være et aktivt valg. I tillegg kan Hetronic fjernkontrollen ta over, dette er i tilfelle noe skulle gå galt under testing og man må ta styring fra systemet. Hetronic fjernkontrollen har høyest prioritet og vil kunne overstyre all kontroll.		
Utførelse	**Forventet resultat**	**Godkjent**
1. Gjennomfør klargjøring for test		
2. Skyv bryteren på fjernkontrollen til posisjon "1". Sørg for at Joystick er i senterposisjon.	Den grønne dioden på fjernkontrollen vil blinke for å indikere at fjernkontrollen er på, i "slow-modus" og brukeren har kontroll. I terminalvinduet står det «Controller is active». En melding i VCS viser at fjernkontrollen USV-en. Man kan høre at motor starter, hvis denne ikke allerede går.	
Kommentarer:		

4.2 Manøvrering med Joystick

Test: Manøvrering med joystick		
Beskrivelse: Testen skal vise at USV kan styres med joystick. Her er det viktig med forsiktighet og at risikovurderinger følges. All kjøring skal utføres på en sikker måte for å unngå materielle skader.		
Utførelse	**Forventet resultat**	**Godkjent**
1. Gjennomfør klargjøring for test.	VIKTIG: Sikre at joystick er i «slow» modus og at skalering av «throttle» er satt til 0.2.	
2. Påse at det ikke befinner seg noe rundt Mariner når du utfører de neste stegene. Utvis forsiktighet når du opererer USV-en.	Unngå skader/ulykker.	
3. Før joystick få grader rett fremover.	USV beveger seg fremover med gitt fart. Gjenta frem til ønsket fart.	
4. Returner joystick til senterposisjon.	USV drifter fremover til vannet stopper den. Uten utslag med joysticken vil motor gi null pådrag og USV vil drifte med vannet.	
5. Før joystick bakover.	USV beveger seg bakover med gitt fart.	
6. Før joystick til venstre.	USV vil rotere/svinge mot venstre.	
7. Før joystick til høyre.	USV roterer/svinge mot høyre.	
8. Roter joystick med klokken.	Baugthruster skyver baugen mot høyre.	
9. Roter joystick mot klokken.	Baugthruster skyver baugen mot venstre.	
10. Forsøk å føre USV sidelengs ved å balanser sving og baugthruster.	USV beveger seg rett sidelengs.	
11. Prøve pådrag i forskjellige retninger samtidig.	USV-en vil oppføre seg likt som joystick-bevegelsene.	
Kommentarer:		

4.3 Stationkeeping

Test: Stationkeeping		
Beskrivelse: Brukeren av systemet kan avgjøre om USV skal stå i stationkeeping og holde posisjonen sin på vannet, eller ta over kontroll på USV og styre manuelt.		
Utførelse	**Forventet resultat**	**Godkjent**
1. Gjennomfør klargjøring for test.		
2. Trykk på knappen markert ST.	Begge lysdiodene på fjernkontrollen lyser for å indikere at USV er i stationkeeping. USV vil holde sin posisjon i vannet. Fjernkontrollen har kontakt med USV, men utslag med joystick vil ikke gi noe resultat.	
3. Forsøk å styre USV med joystick.	Du skal ikke kunne styre USV-en.	
4. Trykk på knappen markert ST og sørg for at joystick **ikke** er i senterposisjon.	Begge lysdiodene på fjernkontrollen skal fortsette å lyse, og indikere at stationkeeping enda er aktiv.	
5. Trykk på knappen markert ST og sørg for at joystick er i senterposisjon.	Kun den grønne lysdioden på fjernkontrollen vil lyse for å indikere at manuell kontroll er aktivert og fjernkontrollen har kontakt med USV. USV-en vil drifte med vannet i påvente om utslag fra joystick.	
6. Beveg på joysticken i en sikker retning.	USV-en vil bevege seg i den retningen joysticken blir beveget i.	
Kommentarer		

5 Tester i fjorden

Av praktiske og sikkerhetsmessige årsaker er det enkelte tester som må gjennomføres i fjorden.

5.1 Test av slow/fast modus

Test: Test av slow/fast modus		
Beskrivelse: En test for å påse at kontrolleren skal kunne settes over i "fast" modus		
Utførelse	**Forventet resultat**	**Godkjent**
1. Påse at bryteren er skjøvet over til posisjon "1", og at kontrolleren har kontroll. Ha joysticken i senterposisjon.	Den grønne dioden vil blinke og signalisere at den er i "slow" modus. VCS vil signalisere om at fjernkontrolleren har kontroll.	
2. Ha USV pekende bort fra land eller langs land, og gi fult utslag fremover på joysticken.	USV vil holde lav fart (under 10 kn).	
3. Trykk inn knappen merket med S/F.	Den grønne dioden vil lyse konstant, og signaliserer at kontrolleren er i "fast" modus. VCS vil gi beskjed om at fjernkontrolleren er i "fast" modus.	
4. Ha USV pekende bort fra land eller langs land, og gi fult utslag på joysticken fremover.	USV vil kunne komme opp i høyrere fart (rundt 23 kn).	
Kommentarer		

6 Tester i simulator

Dette foregår i en Docker-container på Wago PFC200 PLS, som kobles til Raspberry Pi og en PC med VCS. Her vil kommandoer sendes fra joystick til PLS, og deretter til VCS PC, som visualiserer det hele.

6.1 Kommunikasjonsbrudd ved nettverksproblemer

Test av kommunikasjonsbrudd		
Beskrivelse: Testen skal vise hvordan USV og fjernkontroll opptrer ved et kommunikasjonsbrudd. Det skal fremstå av testen at systemet er trygt å operere. Systemet skal oppdage feil automatisk og gi tilbakemelding til bruker.		
Utførelse	**Forventet resultat**	**Godkjent**
1. Gjennomfør klargjøring for test.	Det skal være mulig å styre USV i henhold til testskjema 2. manøvrering med joystick.	
2. Skyv bryteren på fjernkontrollen til posisjon «1». Sørg for at Joystick er i senterposisjon.	Den grønne dioden på fjernkontrollen vil blinke for å indikere at fjernkontrollen er på, i "slow-modus" og brukeren har kontroll. I terminalvinduet står det «Controller is active». En melding i VCS viser at fjernkontrollen har kontroll over USV-en. Man kan høre at motor starter, hvis denne ikke allerede går.	
3. Skyv joysticken fremover.	USV vil bevege seg fremover med gitt pådrag.	
4. Fjern nettverkskoblingen fra RPi.	Den røde dioden på fjernkontrollen blinker for å indikere at det har oppstått et problem med systemet. USV går over i stationkeeping.	
5. Forsøk å styre USV med joystick.	Ingen bevegelse i USV.	
6. Koble til nettverkskabelen.	Den røde dioden på fjernkontrolleren vil slutte å blinke og lyse konstant når nettverkskommunikasjonen er på plass igjen.	
7. Forsøk å styre USV med joystick, uten å sette joystick i senterposisjon.	Du skal ikke kunne styre USV-en.	
8. Sett joystick i senterposisjon.	Den grønne dioden på fjernkontrollen vil blinke for å indikere at fjernkontrollen er på, i "slow-modus" og brukeren har kontroll.	
9. Forsøk å styre USV med joystick.	Bruker har kontroll, og USV beveger seg i ønsket mønster etter input fra joystick.	
Kommentarer		

6.2 Kommunikasjonsbrudd ved tap av joystick

Test: Kommunikasjonsbrudd ved tap av joystick		
Beskrivelse: Testen skal vise hvordan systemet opptrer dersom fjernkontrollen mister kontakt med joysticken.		
Utførelse	**Forventet resultat**	**Godkjent**
1. Fjern USB-kabel til joystick fra RPi.	Den røde dioden på fjernkontrollen blinker for å indikere at det har oppstått en feil på systemet. USV går over i stationkeeping.	
2. Forsøk å styre USV med joystick.	Ingen bevegelse i USV.	
3. Koble til USB-kabel tilbake i RPi.		
4. Forsøk å styre USV med joystick, uten å sette joystick i senterposisjon.	Du skal ikke kunne styre USV-en.	
5. Sett joystick i senterposisjon.	Den grønne dioden på fjernkontrollen vil blinke for å indikere at fjernkontrollen er på, i "slow-modus" og brukeren har kontroll.	
6. Forsøk å styre USV med joystick.	Bruker har kontroll, og USV beveger seg i ønsket mønster etter input fra joystick.	
7. Skyv bryteren på fjernkontrollen til posisjon "0".	Den røde dioden vil lyse konstant og kontrolleren vil miste kontrollen over USV. I terminalvinduet står det «Controller is inactive», og en melding i VCS viser at fjernkontrollen har mistet kontrollen over USV-en.	
Kommentarer		

Dato 05.05.2023

Sted: Brattørekaia 11, 7010 Trondheim

_____	_____
Intern veileder (NTNU)	Ekstern veileder (Maritime Robotics)

_____

Representant for bachelorgruppe

D ipRemote.cpp

ipRemote.cpp

```
// Including selfmade library
#include "ipRemoteLib.h"


// Declering variables and constants
int sendSock, recvSock, js;
bool stationPressed = false, slowPressed = false, errorFlag = false, ledState = false,
driftPressed = false, drift = false;
const char *device = "/dev/input/js0";
struct joystickValues joystickValues, emptyValues;
sockaddr_in sendAddr, recvAddr;
js_event event, *ev;
time_t previousTime;
Msg::Remoteudp::Feedback Feed;


// Starting the main function
int main(){
    cout << "Starting program" << endl;

    // Setting up the GPIO-pins
    setGPIO();

    // Checing if the remote is on or off and continues when remote is off
    // This is a security measure to avoid starting the remote in the on position
    while(digitalRead(switchOn)){
        if(!errorFlag){
            cout << "Turn off controller to continue" << endl;
        }
        error(errorFlag);
    }
    errorFlag = false;

    // Setting the client socket, which is used to send the joystick values to USV
    if(setSocket(sendSock, sendAddr, inet_addr(USV_IP), SEND_PORT, 0)){ // Inet_addr()
converts IP-string to IPv4 dotted decimal notation
        return 1;
    }

    // Setting the server socket, which is used to receive USV values
    if(setSocket(recvSock, recvAddr, INADDR_ANY, RECV_PORT, 1)){
        return 1;
    }

    // Opening the device path and makes it possible to read the connected joystick
    while(jsOpen(js, device, errorFlag)){
        error(errorFlag);
    }

    // Sends the default joystick values to USV
    while(sendStruct(sendSock, joystickValues, sendAddr, errorFlag)){
        error(errorFlag);
    }

    cout << "Entering while loop" << endl;
    previousTime = time(NULL);  // Time used for blinking LEDs
```

```
    while(true){

        // Checks and updates the joystick values
        if(jsUpdate(event, js, joystickValues)){
            joystickValues.station = true;
            while(jsOpen(js, device, errorFlag)){
                sendStruct(sendSock, joystickValues, sendAddr, errorFlag);
                error(errorFlag);
            }
        }

        // Updating lights and checks if any buttons or switches have been pressed
        joystickOnOff(joystickValues);
        stationOnOff(stationPressed, joystickValues, errorFlag);
        slowOnOff(slowPressed, joystickValues);
        feedbackLEDs(joystickValues, ledState, previousTime);

        // Sends the joystick values to OBS
        while(sendStruct(sendSock, joystickValues, sendAddr, errorFlag)){
            error(errorFlag);
        }

        // Checks if OBS has sent any messages back
        if(checkForMsg(recvSock)){
            receiveProtobuf(recvSock, Feed, recvAddr);
            display(Feed);
        }

        // Checks if drift is selected
        driftOnOff(joystickValues, driftPressed, drift);
        while(drift){
            sendStruct(sendSock, emptyValues, sendAddr, errorFlag);
            jsUpdate(event, js, joystickValues);
            usleep(500000);
            driftOnOff(joystickValues, driftPressed, drift);
        }

        emptyValues.active = joystickValues.active;
        emptyValues.slow = joystickValues.slow;

    }
    return 0;
}
```

E ipRemoteLib.cpp

ipRemoteLib.cpp

```
#include "ipRemoteLib.h"


//*************************
//    UDP FUNCTIONS
//*************************


// Checks for messages on the server socket
// If there are messages ready to receive, it returns 1, if not it returns 0
int checkForMsg(int sock)
{
    timeval timeout;
    fd_set rfds;

    timeout.tv_sec = 0;
    timeout.tv_usec = 10000; // Sets the wiating time to 0.01 sec
    int read_result = 0;

    FD_ZERO(&rfds);
    FD_SET(sock, &rfds);
    read_result = select(sock + 1, &rfds, NULL, NULL, &timeout);

    if (read_result > 0){
        return 1;
    }
    else{
        return 0;
    }
}

// Receives USV paramters, parse the data and saves them in the protobuf message
// Returns 1 if an error occurs, or 0 if a message was received
int receiveProtobuf(int sock, Msg::Remoteudp::Feedback &fb, sockaddr_in &addr)
{
    int receive_result;
    socklen_t len = sizeof(addr);
    char buffer[MAXLINE];

    receive_result = recvfrom(sock, (char *)buffer, MAXLINE, 0, (struct sockaddr *) &addr,
&len);
    if(receive_result < 0){
        cout << "Error reciving data" << endl;
        return 1;
    }

    // Parse the received data to the protobuf message
    fb.ParseFromArray(buffer, receive_result);
    return 0;
}

// Initiating the socket and socket address
// Takes into account if it is a server or client socket
// Returnes 0 on success, or 1 if an error occurs
int setSocket(int &sock, sockaddr_in &addr, in_addr_t IP, int PORT, int server)
{
```

```
    int bind_result, connect_result;

    // Initiating the socket
    sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if(sock < 0){
        cout << "Socket creation failed" << endl;
        return 1;
    }

    // Makes the socket address
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = IP;
    addr.sin_port = htons(PORT);

    if(server){
        // If socket is a server, it will bind the socket to the IP and port
        bind_result = bind(sock, (const struct sockaddr *)&addr, sizeof(addr));
        if(bind_result < 0){
            cout << "Bind failed" << endl;
            return 1;
        }
    }
    else{
        // If socket is a client, it will connect the socket to the IP and port
        connect_result = connect(sock, (sockaddr *)&addr, sizeof(addr));
        if(connect_result < 0){
            cout << "Connection failed" << endl;
            return 1;
        }
    }

    return 0;
}

// Sends the joystick values to USV
// Returnes 0 on success, or 1 if an error occurs
int sendStruct(int sock, struct joystickValues &jv, sockaddr_in &addr, bool errorFlag)
{
    int send_result;

    send_result = sendto(sock, (char *)&jv, sizeof(jv), 0, (const struct sockaddr *) &addr,
sizeof(addr));
    if((send_result < 0) && !errorFlag){
        cout << "Error sending data" << endl;
        return 1;
    }
    else if(send_result < 0){
        return 1;
    }

    return 0;
}



// *************************
//   JOYSTICK FUNCTIONS
// *************************
```

```
// Scales the joystick values to the desired range
// Returns the scaled value
int mapping(float val, int axis)
{
    int value, mappingResolution = 100;

    if (axis < 2){  // z- and x-axis
        value = ((val/15700) * mappingResolution);
        if (value > mappingResolution){
            value = mappingResolution;
        }
        if (value < -mappingResolution){
            value = -mappingResolution;
        }
    }
    else{   // y-axis
        value = ((val/20900) * mappingResolution);
        if (value > mappingResolution){
            value = mappingResolution;
        }
        if (value < -mappingResolution){
            value = -mappingResolution;
        }
    }

    if(axis == 1){
        return value*-1;    // Inverts the axis to match desired movement
    }

    return value;
}

// Opening the device path and makes it possible to read the connected joystick
// Returnes 0 on success, or 1 if an error occurs
int jsOpen(int &js, const char* device, bool &errorFlag)
{
    js = open(device, O_RDONLY);
    if(js < 0 && !errorFlag){
        cout << "Error connecting to joystick" << endl;
        return 1;
    }
    else if(js < 0){
        return 1;
    }

    cout << "Joystick connected" << endl;
    return 0;
}

// Turnes the remote on or off
// When turnig the remote on, the engine will turn on
void joystickOnOff(struct joystickValues &jv){

    // As a security measure, the remote will only turn on if the joystick is in center
position
    if(digitalRead(switchOn) && (jv.x == 0) && (jv.z == 0) && (jv.y == 0)){
        //
        if(!jv.active){
            cout << "Controller is active" << endl;
```

```
            jv.active = true;
        }
    }
    else if(!digitalRead(switchOn)){
        if(jv.active){
            cout << "Controller is inactive" << endl;
        }
        jv.active = false;
        jv.station = false;
    }
}

// Reads the joystick position and updates the joystick values
// Returns 1 if connection to joystick is lost
int jsUpdate(struct js_event event, int js, struct joystickValues &jv)
{
    struct js_event *ev;
    ev = &event;
    timeval timeout;
    fd_set rfds;

    timeout.tv_sec = 0;
    timeout.tv_usec = 10000; // 0.01 sec
    int read_result = 0;

    while(true){
        FD_ZERO(&rfds);
        FD_SET(js, &rfds);
        read_result = select(js + 1, &rfds, NULL, NULL, &timeout);

        if (read_result > 0){
            read_result = read(js, ev, sizeof(*ev));
            if(read_result == -1){
                return 1;
            }
            if(event.type == JS_EVENT_AXIS){
                switch(ev->number){
                // Change in z-axis
                    case 0:
                        jv.z = mapping(ev->value, ev->number);
                        break;

                // Change in x-axis
                    case 1:
                        jv.x = mapping(ev->value, ev->number);
                        break;

                // Change in y-axis
                    case 2:
                        jv.y = mapping(ev->value, ev->number);
                        break;

                // Error when reading
                    default:
                        cout << "Error with reading axis" << endl;
                        jv.x = 0;
                        jv.y = 0;
                        jv.z = 0;
                }
            }
```

```
        }
        else{
            return 0;
        }
    }
}



// *************************
//  User interface functions
// *************************


// Setting up the GPIO-pins on Raspberry Pi
void setGPIO()
{
    wiringPiSetupPhys();
    pinMode(redLED, OUTPUT);
    pinMode(greenLED, OUTPUT);
    pinMode(switchOn, INPUT);
    pinMode(button1, INPUT);
    pinMode(button2, INPUT);
    pinMode(button3, INPUT);
    digitalWrite(greenLED, LOW);
    digitalWrite(redLED, HIGH);
}

// Changes between slow and fast mode
void slowOnOff(bool &buttonPressed, struct joystickValues &jv)
{
    if(digitalRead(button2) && !buttonPressed && jv.active){
        buttonPressed = true;
        // As a security measure it is only possible to change mode when the joystick is in
center position
        switch(jv.x || jv.z || jv.y){
            case 0:
                if(jv.slow){
                    jv.slow = false;
                    cout << "Slow mode is inactive" << endl;
                }
                else{
                    jv.slow = true;
                    cout << "Slow mode is active" << endl;
                }
                break;
            default:
                break;
        }
    }

    if(!digitalRead(button2) && buttonPressed){
        buttonPressed = false;
    }
}

// Turns on and off the drift function
// Returns the boolean drift state
bool driftOnOff(struct joystickValues &jv, bool &button, bool &drift)
{
```

```
    if(digitalRead(button3) && !button && !drift){
        button = true;
        drift = true;
        jv.x = 0;
        jv.y = 0;
        jv.z = 0;
        digitalWrite(redLED, HIGH);
        digitalWrite(greenLED, LOW);
    }
    else if(!digitalRead(button3) && button){
        button = false;
    }
    // As a security measure it is only possible to change state when the joystick is in
center position
    else if(digitalRead(button3) && !button && drift && jv.x == 0 && jv.y == 0 && jv.z == 0){
        button = true;
        drift = false;
    }

    return drift;
}

// Manages the LED outputs
void feedbackLEDs(struct joystickValues &jv, bool &ledState, time_t &previousTime)
{
    time_t currentTime;
    int interval = 1;
    currentTime = time(NULL);

    switch(jv.active && jv.station){
        // Controller is on and stationkeeping is active
        case 1:
            digitalWrite(greenLED, HIGH);
            digitalWrite(redLED, HIGH);
            break;
        default:
            // Controller is on and slowmode is active
            if(jv.active && jv.slow && (currentTime - previousTime >= interval)){
                if(ledState){
                    digitalWrite(greenLED, HIGH);
                    digitalWrite(redLED, LOW);
                }
                else{
                    digitalWrite(greenLED, LOW);
                    digitalWrite(redLED, LOW);
                }
                ledState = !ledState;
                previousTime = time(NULL);
            }
            // controller is on and slowmode is inactive
            else if(jv.active && !jv.slow){
                digitalWrite(greenLED, HIGH);
                digitalWrite(redLED, LOW);
            }
            // Controller is off
            else if(!jv.active){
                digitalWrite(greenLED, LOW);
                digitalWrite(redLED, HIGH);
            }
            break;
```

```
    }
}

// Turns on and off the stationkeeping function
void stationOnOff(bool &buttonPressed, struct joystickValues &jv, bool &errorFlag)
{
    // If an error has occurred, stationkeeping is activated
    if(errorFlag){
        jv.station = true;
        // To reset the error flag the joystick has to be in the center position
        if(jv.x == 0 && jv.y == 0 && jv.z == 0){
            errorFlag = false;
            cout << "All is good" << endl;
        }
    }

    if(digitalRead(button1) && !buttonPressed && jv.active){
        buttonPressed = true;
        // As a security measure it is only possible to turn off stationkeeping when the
joystick is in center position
        switch(jv.x || jv.z || jv.y || !jv.station){
            case 0:
                jv.station = false;
                cout << "Stationkeeping is inactive" << endl;
                break;
            default:
                if(!jv.station){
                    cout << "Stationkeeping is active" << endl;
                }
                jv.station = true;
                break;
        }
    }

    if(!digitalRead(button1) && buttonPressed){
        buttonPressed = false;
    }
}

// Displays the USV parameters
void display(Msg::Remoteudp::Feedback &fb)
{
    cout << "-------------------------------------" << endl;
    cout << "USV parameters:" << endl;
    cout << "Speed over ground:    " << fb.sog()*1.9438 << " kn"<< endl;             // From
m/s to kn
    cout << "RPM:                  " << fb.rpm() << " rpm" << endl;
    cout << "Course over ground:   " << fb.cog()*180/M_PI << " deg" << endl;         // From
rad to deg
    cout << "Latitude:             " << fb.latitude()*180/M_PI << " deg N" << endl;  // From
rad to deg
    cout << "Longitude:            " << fb.longitude()*180/M_PI << " deg E" << endl; // From
rad to deg
    cout << "-------------------------------------" << endl;
}



// *************************
//  ERROR FUNCTIONS
// *************************
```

```
// Blinks the red LED if an error has occurred
void error(bool &errorFlag)
{
    errorFlag = true;
    digitalWrite(greenLED, LOW);
    digitalWrite(redLED, LOW);
    usleep(200000);
    digitalWrite(redLED, HIGH);
    usleep(200000);
}



// *************************
//  NOT IN USE
// *************************


// Sends a const char array to USV
// Returnes 0 on success, or 1 if an error occurs
int sendMsg(int &sock, const char* &msg, sockaddr_in &addr)
{
    int send_result;

    send_result = sendto(sock, msg, strlen(msg), 0, (const struct sockaddr *) &addr,
sizeof(addr));
    if (send_result < 0){
        cout << "Error sending data" << endl;
        return 1;
    }

    cout << "Message sent" << endl;
    return 0;
}

// Receives USV paramters as a struct
// Returnes 0 on success, or 1 if an error occurs
int receiveStruct(int sock, struct usvValues &usv, sockaddr_in &addr)
{
    int receive_result;
    socklen_t len = sizeof(addr);

    receive_result = recvfrom(sock, (char *)&usv, sizeof(usv), 0, (struct sockaddr *) &addr,
&len);
    if(receive_result < 0){
        cout << "Error reciving data" << endl;
        return 1;
    }
    else if(receive_result < 0){
        return 1;
    }

    return 0;
}

// Displays the joystick values
// Used during testing and troubleshooting
void displayJs(struct joystickValues &jv)
```

```
{
    cout << "----------------------" << endl;
    cout << "Joystick values:" << endl;
    cout << jv.x << endl;
    cout << jv.y << endl;
    cout << jv.z << endl;
    cout << "----------------------" << endl;
}
```

F ipRemoteLib.h

ipRemoteLib.h

```
#ifndef REMOTELIB_H_
#define REMOTELIB_H_

// Including necessary libraries
#include <iostream>             // Standard library for interface with terminal
#include <string.h>             // Library for manipulating arrays of characters
#include <sys/socket.h>         // Internet protocols
#include <arpa/inet.h>          // Definitions for internet operations
#include <fcntl.h>              // File control operations
#include <unistd.h>             // Accessing I/O primitivies and standard symbolic types
#include <cmath>                // Common mathematical functions
#include <ctime>                // Converts time and date formates
#include <linux/joystick.h>     // Capture Joystick inputs
#include <wiringPi.h>           // Controlling GPIO pins
#include "MsgRemoteUdp.pb.h"    // Protobuf message library


// UDP-comunication
#define USV_IP "10.206.6.2"
#define SEND_PORT 4321
#define RECV_PORT 7538
#define MAXLINE 1024

// GPIO pins
#define redLED 31
#define greenLED 29
#define switchOn 37
#define button1 11
#define button2 13
#define button3 15

// using namespace called std - avoiding to have to write "std::" everywhere
using namespace std;



// Struct for saving joystick values
struct joystickValues{
    uint16_t msg_header = 0x45; //0x45
    bool active = false;
    bool station = false;
    bool slow = true;
    int16_t x = 0;
    int16_t y = 0;
    int16_t z = 0;
    uint16_t msg_tail = 0x1A4; //Ox1A4
};


// Struct for saving USV parameters
struct usvValues{
    uint16_t msg_header;
    int16_t latitude;
    int16_t longitude;
    int16_t sog;
    int16_t rpm;
```

```
    int16_t cog;
    uint16_t msg_tail;
};


// UDP functions

int checkForMsg(int sock);
int receiveProtobuf(int sock, Msg::Remoteudp::Feedback &fb, sockaddr_in &addr);
int setSocket(int &sock, sockaddr_in &addr, in_addr_t IP, int PORT, int server);
int sendStruct(int sock, struct joystickValues &jv, sockaddr_in &addr, bool errorFlag);



// Joystick functions

int mapping(float val, int axis);
void joystickOnOff(struct joystickValues &jv);
int jsOpen(int &js, const char* device, bool &errorFlag);
int jsUpdate(js_event event, int js, struct joystickValues &jv);


// User interface functions

void setGPIO();
void slowOnOff(bool &buttonPressed, struct joystickValues &jv);
bool driftOnOff(struct joystickValues &jv, bool &button, bool &drift);
void feedbackLEDs(struct joystickValues &jv, bool &ledState, time_t &previousTime);
void stationOnOff(bool &buttonPressed, struct joystickValues &jv, bool &errorFlag);
void display(Msg::Remoteudp::Feedback &fb);


// Error functions

void error(bool &errorFlag);



// NOT IN USE

int sendMsg(int &sock, const char* &msg, sockaddr_in &addr);
int receiveStruct(int sock, struct usvValues &usv, sockaddr_in &addr);
void displayJs(struct joystickValues &jv);


#endif
```

G MsgRemoteUdp.proto

MsgRemoteUdp.proto

```
syntax = "proto2";
package Msg.Remoteudp;

message Feedback
{
        ///Speed over ground
        required float sog = 1;
        //Course over ground
        required float cog = 2;
        ///Latitude value
        required double latitude = 3;
        ///Longitude value
        required double longitude = 4;
        ///Rpm of engine
        required float rpm = 5;
}
```

H Dockerfil for bygging av Debian-pakker

Docker dependencies/Dockerfile

```
# Three of the core (build) dependencies are installed privately from Maritime Robotics, making them not available without MR permission.
# This Dockerfile builds .deb packages for the dependencies.

# Creates a base layer from the arm32v7/debian:buster-slim Docker Image. Also defines which platform the image is built from.
FROM --platform=linux/arm/v7 arm32v7/debian:buster-slim

# Specifies the user inside the Container.
USER root

# Installs core (build) dependencies and tools necessary for OBS build and run:
RUN apt update
RUN apt -y upgrade
RUN apt install -y devscripts
RUN apt install -y git
RUN apt install -y debhelper
RUN apt install -y qtbase5-dev
RUN apt install -y cmake
RUN apt install -y doxygen

# Creating .deb packages to build OBS
# Copies the project libraries to the docker container, for building the packages
COPY libqtlogutils libqtlogutils
COPY makeVer makeVer
COPY aisparser aisparser

# MAKING THE PACKAGES FROM INSIDE THE CONTAINER
# In the container, enter the specific folders and build the .deb packages needed for creating OBS:

# Commands for creating libqtlogutils .deb packages:
# cd /libqtlogutils
# git checkout no-cmake
# ./buildDeb
# cd ..
# ls libqtlogutils*
# detach from the container "Ctrl P + Ctrl Q"
# docker cp *container name*:/*deb-packages* .

# Commands for creating makeVer .deb packages:
# cd /makeVer
# git checkout *branch* # if necessary
# ./buildDeb
# cd ..
# ls makeVer*
# detach from the container "Ctrl P + Ctrl Q"
# docker cp *container name*:/*deb-packages* .

# Commands for creating aisparser .deb packages:
# cd /aisparser
# git checkout *branch* # if necessary
# debuild -uc -us -b -i # doxygen needed for build
# cd ..
# ls libais*
```

```
53 # detach from the container "Ctrl P + Ctrl Q"
54 # docker cp *container name*:/*deb-packages* .
55
56 # TO BUILD THIS DOCKERFILE:
57
58 # Start the Docker daemon:'sudo systemctl start docker'. If Docker desktop is
   installed, open the app instead.
59 # cd *folder with Dockerfile*
60 # docker buildx build --platform linux/arm/v7 -t *username*/*repository*:*tag*
   --push .
61
62 # TO RUN THIS DOCKERFILE:
63
64 # docker run (--rm) --platform linux/arm/v7 -(d)it --network host --name
   *container name* --restart always *username*/*repository*:*tag*
65 # Container will also always restart. Could swap always for unless-stopped, then
   it doesnt restart if stopped manually or otherwise,
66 # even if Docker daemon restarts.
```

I Dockerfil for bygging av OBS

Build OBS/Dockerfile

```
# Creates a base layer from the arm32v7/debian:buster-slim Docker Image. Also defines which platform the image is built from.
FROM --platform=linux/arm/v7 arm32v7/debian:buster-slim

# Specifies the user inside the Container.
USER root

# Installs core (build) dependencies and tools necessary for OBS build and run:
RUN apt-get update
RUN apt-get install -y debhelper
RUN apt-get install -y qtbase5-dev
RUN apt-get install -y qt5-qmake
RUN apt-get install -y libprotobuf-dev
RUN apt-get install -y libqt5serialbus5-dev
RUN apt-get install -y libqt5serialbus5-plugins
RUN apt-get install -y libqt5serialport5-dev
RUN apt-get install -y libarmadillo-dev
RUN apt-get install -y libproj-dev
RUN apt-get install -y libgeographic-dev
RUN apt-get install -y libssl-dev
RUN apt-get install -y qtdeclarative5-dev
RUN apt-get install -y libconfig++-dev
RUN apt-get install -y libxml2-dev
RUN apt-get install -y libmodbus-dev
RUN apt-get install -y libeigen3-dev
RUN apt-get install -y libnlopt-dev
RUN apt-get install -y qtscript5-dev
RUN apt-get install -y libsnmp-dev
RUN apt-get install -y cmake
RUN apt-get install -y git
RUN apt-get install -y devscripts
RUN apt-get install -y protobuf-compiler
# These packages were earlier installed for debugging:
# RUN apt-get install -y iproute2
# RUN apt-get install -y iputils-ping

# Copies the build dependent packages that were created on PC in the Dockerfile for building library packages.
COPY libqtlogutils-dev_0.5.2-1_armhf.deb .
COPY libqtlogutils0_0.5.2-1_armhf.deb .
COPY makever_0.8.0-1_all.deb .
COPY libais1_1.9.0-5_armhf.deb .
COPY libais-doc_1.9.0-5_all.deb .
COPY libais-dev_1.9.0-5_armhf.deb .

# Update and installs all the build dependent packages that are copied by the commands over.
RUN apt-get update && apt-get install -y \
./libqtlogutils-dev_0.5.2-1_armhf.deb \
./libqtlogutils0_0.5.2-1_armhf.deb \
./makever_0.8.0-1_all.deb \
./libais1_1.9.0-5_armhf.deb \
./libais-doc_1.9.0-5_all.deb \
./libais-dev_1.9.0-5_armhf.deb

```

```
53 # Copies the project library obs-qt to the docker container, for building the
   OBS binary file or OBS .deb package.
54 COPY obs-qt obs-qt
55
56 # MAKING THE BINARY FILE/PACKAGE FROM INSIDE THE CONTAINER
57 # In the container, enter the specific folder and build the binary / .deb
   packages of OBS:
58
59 # Making the binary file from the Docker container
60 # Important to checkout the right branch and update submodules
61 # cd obs-qt
62 # git checkout origin/enh-532/docker-container-implementation-of-obs-for-mariner
63 # git submodule update --init --recursive
64 # mkdir build && cd build
65 # cmake -DCMAKE_BUILD_TYPE=Release ..
66 # make -j$(nproc)
67 # detach from the container "Ctrl P + Ctrl Q"
68 # docker cp *container name*:/*binary file* .
69
70 # Alternatively or in addition:
71 # Making the debian package of OBS from the container
72 # # Important to checkout the right branch and update submodules
73 # cd obs-qt
74 # git checkout origin/enh-532/docker-container-implementation-of-obs-for-mariner
75 # git submodule update --init --recursive
76 # debuild -uc -us -b -i
77 # cd ..
78 # ls obs*
79 # detach from the container "Ctrl P + Ctrl Q"
80 # docker cp *container name*:/*deb-packages* .
81
82 # TO BUILD THIS DOCKERFILE:
83
84 # Start the Docker daemon:'sudo systemctl start docker'. If Docker desktop is
   installed, open the app instead.
85 # cd *folder with Dockerfile*
86 # docker buildx build --platform linux/arm/v7 -t *username*/*repository*:*tag*
   --push .
87
88 # TO RUN THIS DOCKERFILE:
89
90 # docker run (--rm) --platform linux/arm/v7 -(d)it --network host --name
   *container name* --restart always *username*/*repository*:*tag*
91 # Container will also always restart. Could swap always for unless-stopped, then
   it doesnt restart if stopped manually or otherwise,
92 # even if Docker daemon restarts.
93
```

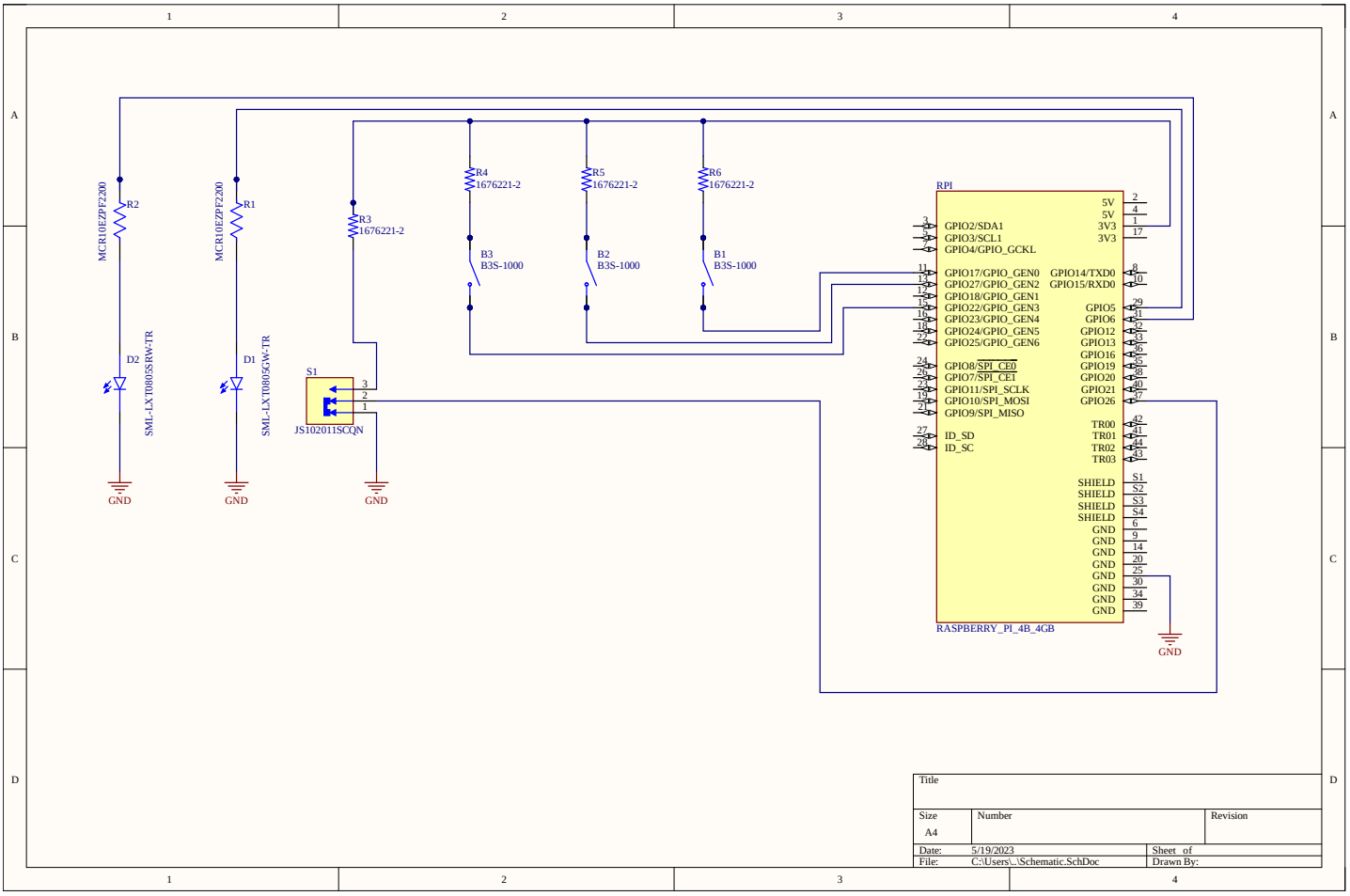
J Dockerfil for kjøring av OBS

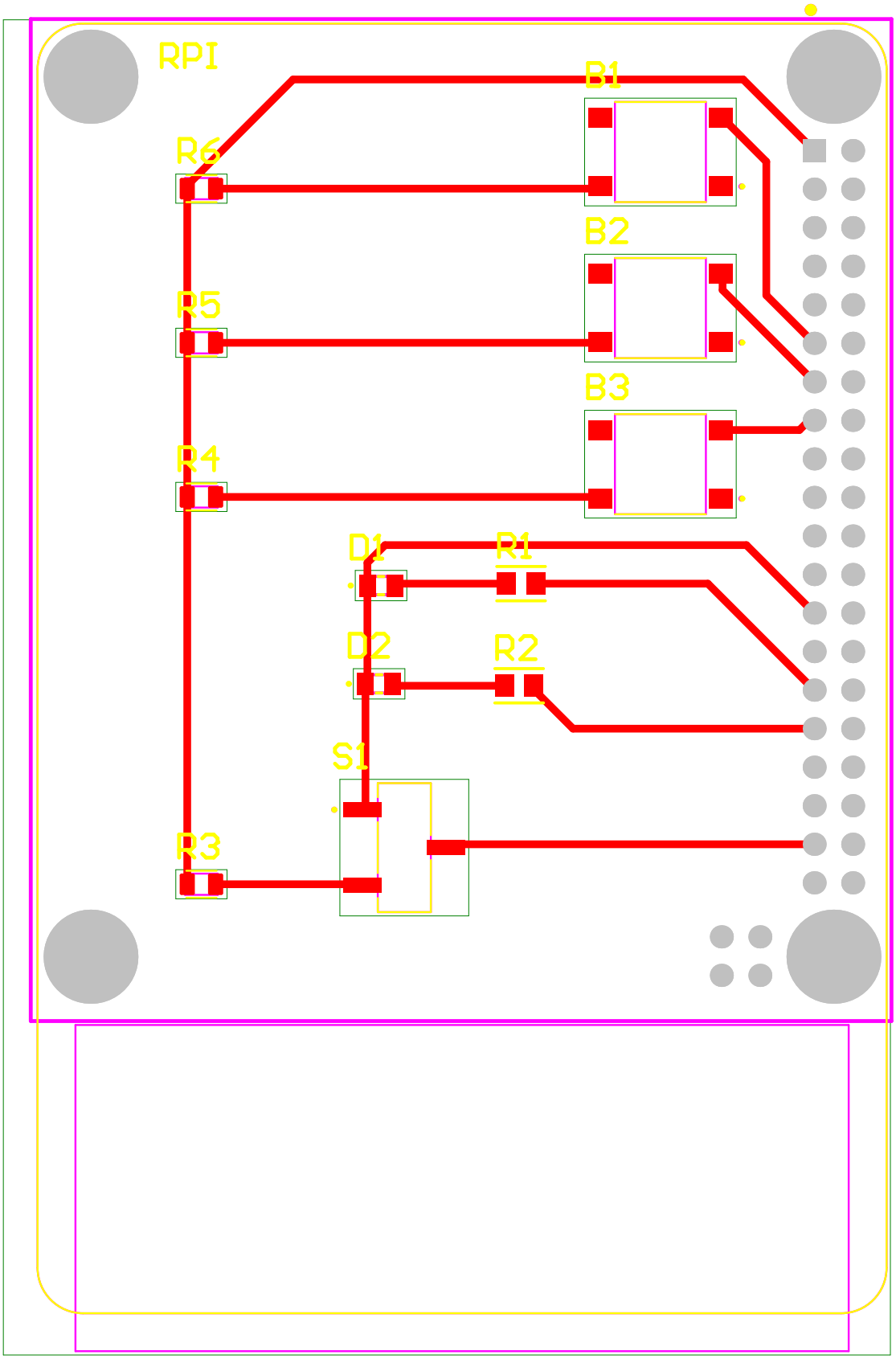
Docker-Oppgave/Dockerfile

```
# Creates a base layer from the arm32v7/debian:buster-slim Docker Image. Also defines which platform the image is built from.
FROM --platform=linux/arm/v7 arm32v7/debian:buster-slim

# Specifies the user inside the Container.
USER root

# Installs core (build) dependencies and tools necessary for OBS build and run:
RUN apt-get update
RUN apt-get install -y debhelper
RUN apt-get install -y qtbase5-dev
RUN apt-get install -y qt5-qmake
RUN apt-get install -y libprotobuf-dev
RUN apt-get install -y libqt5serialbus5-dev
RUN apt-get install -y libqt5serialbus5-plugins
RUN apt-get install -y libqt5serialport5-dev
RUN apt-get install -y libarmadillo-dev
RUN apt-get install -y libproj-dev
RUN apt-get install -y libgeographic-dev
RUN apt-get install -y libssl-dev
RUN apt-get install -y qtdeclarative5-dev
RUN apt-get install -y libconfig++-dev
RUN apt-get install -y libxml2-dev
RUN apt-get install -y libmodbus-dev
RUN apt-get install -y libeigen3-dev
RUN apt-get install -y libnlopt-dev
RUN apt-get install -y qtscript5-dev
RUN apt-get install -y libsnmp-dev
RUN apt-get install -y cmake
RUN apt-get install -y git
RUN apt-get install -y devscripts
RUN apt-get install -y protobuf-compiler
# These packages were installed for debugging:
# RUN apt-get install -y iproute2
# RUN apt-get install -y iputils-ping

# Copies the build dependent packages that were created on PC in the Dockerfile for building library packages.
COPY libqtlogutils-dev_0.5.2-1_armhf.deb .
COPY libqtlogutils0_0.5.2-1_armhf.deb .
COPY makever_0.8.0-1_all.deb .
COPY libais1_1.9.0-5_armhf.deb .
COPY libais-doc_1.9.0-5_all.deb .
COPY libais-dev_1.9.0-5_armhf.deb .
COPY obs-qt_2.19.0~alpha-1_armhf.deb .

# Update and installs all the build dependent packages that are copied by the commands over.
RUN apt-get update && apt-get install -y \
./libqtlogutils-dev_0.5.2-1_armhf.deb \
./libqtlogutils0_0.5.2-1_armhf.deb \
./makever_0.8.0-1_all.deb \
./libais1_1.9.0-5_armhf.deb \
./libais-doc_1.9.0-5_all.deb \
./libais-dev_1.9.0-5_armhf.deb \
```

```
./obs-qt_2.19.0~alpha-1_armhf.deb

# Copies over the needed files for VCS and OBS to the container.
COPY obs-qt obs-qt
COPY boot.js /etc/obs/
COPY speed2force.csv /etc/obs/
COPY properties.conf /etc/obs/
COPY /obs /var/lib/obs

# To get the newest update and UDP-remote-interface, git checkout the 532 branch. To do this comment out the code under and enter the container.
# OBS runs on the develop branch, if UDP-remote-interface is commented out in nano boot.js.
# This code runs OBS on startup from the binaryfile. To run with .deb package, comment out the two lines of code under.
WORKDIR /obs-qt/build
CMD ["./obs-qt", "-b", "/etc/obs/boot.js"]

# TO BUILD THIS DOCKERFILE:

# Start the Docker daemon:'sudo systemctl start docker'. If Docker desktop is installed, open the app instead.
# cd *folder with Dockerfile*
# docker buildx build --platform linux/arm/v7 -t *username*/*repository*:*tag* --push .

# TO RUN THIS DOCKERFILE:

# docker run (--rm) --platform linux/arm/v7 -(d)it --network host --name *container name* --restart always *username*/*repository*:*tag*
# Container will also always restart. Could swap always for unless-stopped, then it doesnt restart if stopped manually or otherwise,
# even if Docker daemon restarts.

```

K Skjematikk




RPI
B1
R6
B2
R5
B3
R4
D1
R1
D2
R2
S1
R3

Comment	Description	Designator	Footprint	LibRef	Quantity
B3S-1000	Tactile Switch SPST-NO Top Actuated Surface Mount	B1, B2, B3	SW_B3S-1000	B3S-1000	3
SML-LXT0805GW-TR	Green LED Indication - Discrete 2V 0805 _2012 Metric_	D1	LEDC2012X120N	SML-LXT0805GW-TR	1
SML-LXT0805SRW-TR	LED RED DIFFUSED 0805 SMD	D2	LEDC2012X120N	SML-LXT0805SRW-TR	1
MCR10EZPF2200	Thick Film Chip Resistor, 0805, 220Ω, 1%, 100ppm/°C, 0.125W, 150V	R1, R2	FP-MCR10-IPC_A	CMP-08839-025901-1	2
1676221-2	RES SMD 1K OHM 0.1% 1/10W 0805	R3, R4, R5, R6	RESC2012X65N	1676221-2	4
RASPBERRY_PI_4B_4GB	BCM2711 Raspberry Pi 4 Model B 4GB - ARMÂ® CortexÂ®-A72 MPU Embedded Evaluation Board	RPI	MODULE_RASPBERRY_PI_4B_4GB	RASPBERRY_PI_4B_4GB	1
JS102011SCQN	Slide Switch SPDT Surface Mount	S1	SW_JS102011SCQN	JS102011SCQN	1

Implementering av et PLS-basert fjernstyringssystem for ubemannede overflatefartøy (USV)

Trym Helle, Isaac Aninmpong, Martin Wester Hagen og Sindre Vatnamot Amble

Introduksjon

Denne bacheloroppgaven tar for seg utviklingen og implementering av et internet protocol (IP) basert fjernstyringssytem for et ubemannet overflatefartøy (USV), i tillegg til mulighetene for å kjøre Maritime Robotics egenutviklede ombordsystem (OBS) på en PLC ved hjelp av programmet Docker. Arbeidet har resultert i en fjernkontroll som kan styre en USV, samt en fullverdig versjon av OBS som kjører via en container løsning på en WAGO PLS med navn PFC200.

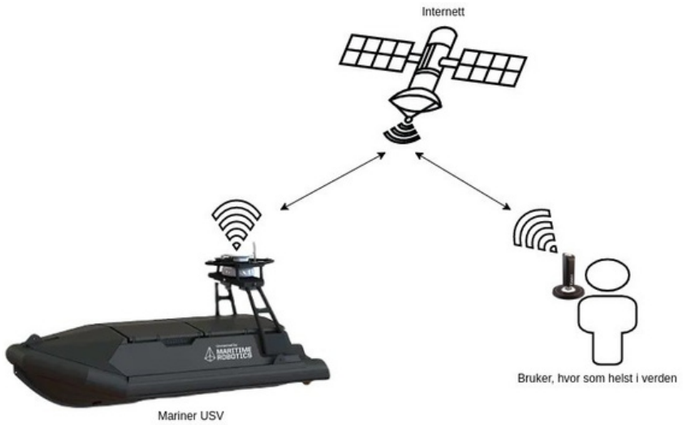
Maritime Robotics

Maritime Robotics A/S er en Trondheimsbasert bedrift som utvikler og produserer avanserte USV-er og droner (UAS). USV-ene deres går fra Otter, på 2 meter, til Mariner X, på 9 meter. Maritime Robotics har USV-er for bruk i både kystnære strøk og offshore, og deres USV-er brukes blant annet av Universitetet i Tromsø, og den Spanske Marinen.

Originalt system

Systemets bestod opprinnelig av en Mariner USV, som kommuniserer med Maritime Robotics styreprogram for PC (VCS) i tillegg til en radiofrekvensfjernkontroll som kan brukes til finmanøvrering på nært hold. Fjernkontrollen, Hetronic, har kort rekkevidde, og mangel på smartposisjoneringsfunksjoner. VCS gir gode ruteplanleggings og styringsmuligheter, men mangel på god manuell styring. USV-ens funksjonsutførelse skjer på en PC som sender kommandoer til WAGO PFC.

Ny løsning

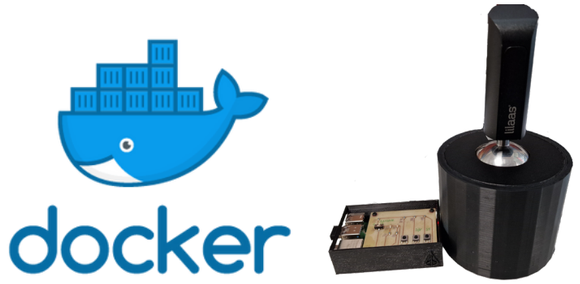


Nå kan USV-en styres med joystick over internett, fra hvor som helst i verden og benytte seg av avanserte funksjoner fra OBS.

Sluttprodukt

Den nye løsningen var todelt:

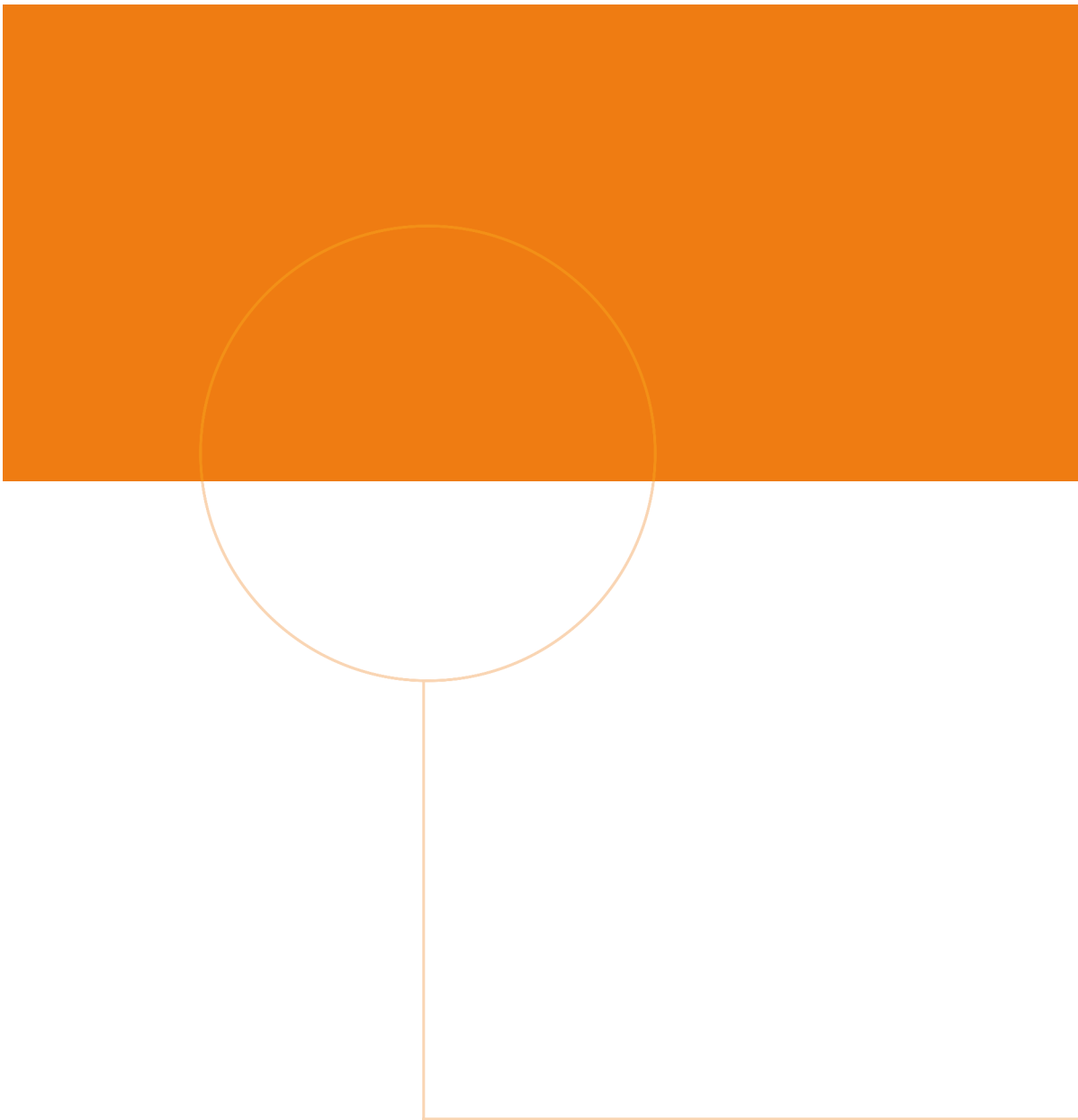
- En IP-basert fjernkontroll, som har rekkevidde over hele verden, og gir god finstyring og tilgang til avansert funksjonalitet
- Maritime Robotics OBS implementert i en Docker container-løsning, hvor OBS kan kjøres på PFC-en, og slik samle all oppgaveutførelse på en maskin



Prosjektets to bestanddeler: Docker-container og IP-fjernkontrollen

Dockeriseringen av OBS betyr at man kan bygge og kjøre programmer for andre typer datamaskiner på sin egen datamaskin ved hjelp av QEMU. Dette gjør at man enkelt kan utvikle nye versjoner uten å måtte ha fysisk hardware å teste på, og det gjør også at en WAGO PFC-200 kan ha avansert C++-bygget kode, OBS-et, kjørende på seg.

IP-fjernkontrollen gir en USV-operatør stor frihet i arbeidslokasjon, da man kun trenger internett for å operere USV-en. I tillegg har man tilgang til funksjonalitet som "Station Keeping", altså statisk posisjon til tross for vær og vind, og man kan sette pådragsbegrensninger på USV-en. Man mottar også visuelle tilbakemeldinger fra systemet i form av en rød og en grønn LED, som lyser for å indikere forskjellige beskjeder.



NTNU
Kunnskap for en bedre verden

MARITIME ROBOTICS