NTNU
Det skapende universitet

Kommunikasjon mellom bakkestasjon og ubemannet luftfarkost

Jostein Austvik Jacobsen

Master i teknisk kybernetikk
Oppgaven levert: Juni 2009
Hovedveileder: Sverre Hendseth, ITK
Biveileder(e): Jon Bernhard Hostmark, Kongsberg Defence & Aerospace

Norges teknisk-naturvitenskapelige universitet
Institutt for teknisk kybernetikk

Oppgavetekst

Institutt for Teknisk Kybernetikk ønsker en generell plattform for et autonomt ubemannet fly (UAV) med elektrisk fremdrift. Den skal kunne brukes til for eksempel:

- utføre overvåkningsoppdrag med live video overføring
- relay stasjon for høyhastighets datalink i ulendt terreng
- demonstrasjoner og nyrekruttering for Institutt for Teknisk Kybernetikk

Når flyet er ferdigstilt, skal det inneholde nødvendig instrumentering for å kunne å styre og navigere på egenhånd. I tillegg skal flyet kommunisere trådløst med en bakkestasjon.

En komplett bakkestasjon skal kunne representere all tilgjengelig flydata fra både flysystem og nyttelast, dvs. servoposisjoner, motorpådrag, IMU-posisjoner, GPS-koordinater og bilder for brukeren. Brukeren skal kunne sende kommandoer tilbake til flyet.

Denne oppgaven er et ledd i prosjektet Local Hawk som utføres i et samarbeid mellom NTNU, Kongsberg Defence & Aerospace, Høgskolen i Buskerud og Universitetet i Agder.

Oppgaven består av å lage et rammeverk for kommunikasjon mellom bakkestasjon og UAV. Rammeverket skal også testes ved å sende kamerabilder gjennom det og ned til bakkestasjonen. Det skal dannes et grunnlag for at det senere kan bli implementert ny, og hittil udefinert, funksjonalitet som for eksempel regulatorer, styringsalgoritmer og kunstig intelligens. Hvordan ny funksjonalitet kan legges til skal dokumenteres.

Datalinken mellom UAV og bakkestasjonen har varierende båndbredde og uforutsigbar oppetid. Det skal redegjøres for hvordan dette løses. Brukeren av bakkestasjonen skal slippe å bry seg med hvordan UAVen og bakkestasjonen finner hverandre og hvordan kommunikasjonen mellom dem fungerer ved varierende båndbredde og oppetid, men ønsker på den annen side også å bli informert om slike hendelser på en subtil måte.

Oppgaven gitt: 15. januar 2009
Hovedveileder: Sverre Hendseth, ITK

Forord

En viktig del av UAV-systemet er bakkestasjonen. På dette stadiet i utviklingen av UAVen var det naturlig å ta for seg hvordan bakkestasjonen skal fungere, og hvordan den skal kommunisere med flyet. Oppgaven passer også bra med hva undertegnede ønsker å sette seg dypere inn i, med tanke på oppgavens vekt på embedded, Linux og generelt programmering.

Først vil jeg takke hovedveileder Sverre Hendseth som har vært en pådriver for dette prosjektet så lenge jeg har holdt på med de. Videre vil jeg takke ekstern veileder Jon Bernhard Høstmark ved Kongsberg Defence & Aerospace (KDA) for å ha koordinert arbeidet mellom alle som har jobbet med flyprosjektet ved NTNU, Høgskolen i Buskerud (HiBu) og Universitetet i Agder (UiA).

Jeg vil også rette en takk til Lars Ivar Miljeteig (HiBu) som jeg har hatt en konstruktiv dialog med underveis angående Gumstix-hovedkortet, Paal Alexander Nerholm for kretskortutlegget til 5V/3.3V-omformeren og Christian Kjølseth for hjelp med lodding av nevnte kretskort.

Takk også til Telenor for utlån av USB-modem for mobilt bredbånd.
Og tilslutt vil jeg rette en takk til Kongsberg Defence & Aerospace for finansiell støtte; foruten denne ville ikke prosjektet ha vært mulig å gjennomføre.

Sammendrag

Institutt for teknisk kybernetikk har interesse for å utvikle et autonomt fly (UAV) gjennom prosjekt- og diplom-oppgaver. Det er ønskelig å kunne gi kommandoer til og hente ned data live fra UAVen. I denne oppgaven legges det frem et rammeverk for slik kommunikasjon med flyet. Rammeverket kalles Robot Operating System og ble valgt etter først å ha utredet hva slags funksjonalitet og overordnet design som kreves for dette prosjektet.

Robot Operating System (ROS) er åpen kildekode og er et veldig modulært rammeverk som det er enkelt å benytte seg av. Selve nettverksprotokollene abstraheres vekk av ROS slik at systemutvikleren kan konsentrere seg om hva han/hun vil at systemet skal gjøre, fremfor å gjenoppfinne og reimplementere nettverkskommunikasjonen. ROS har fungert veldig bra under utvikling og bruk av bakkestasjonen.

Det er bygget videre på tidligere arbeid hvor et Gumstix hovedkort med Linux ble valgt som hovedprosesseringsenhet i UAVen. Det er blitt etset et kretskort som kobler Gumstixen opp til I^2C-bussen.

En gruppe studenter koblet Gumstixen til mobilt bredbånd. Det er bygget videre på resultatet deres slik at UAVen kobler automatisk til Internett med VPN og starter alle ROS-programmer.

ROS kan inkluderes og kompileres inn i programmer både på stasjonære maskiner, og på Gumstix-plattformen. Det er produsert en tutorial som beskriver hvordan dette gjøres. I tidligere arbeid fikk Gumstixen installert støtte for webkameraer og dette er bygget videre på ved å lage et program som streamer bilder til bakkestasjonen.

Det er laget en generell bakkestasjon der moduler enkelt kan byttes ut og legges til. Et terminalinterface til denne bakkestasjonen er produsert, sammen med klienter for mottak av video, IMU-data og GPS-data.

Systemet er testet ved å sende testdata for IMU- og GPS-data fra UAV til bakkestasjonen hvor de ble plottet. Video fra webkameraet er også sendt fra flyet over mobilt bredbånd og ned til bakkestasjonen gjennom ROS. Oppsettet egner seg veldig godt til hardware-in-loop testing og utvikling.

Innhold

Kapittel 1

Introduksjon

Initiativet til Local Hawk-prosjektet[8] ble tatt av Kongsberg Defence & Aerospace[6] (KDA) sommeren 2008, med mål om å gi ingeniørstudenter en sjanse til å gjøre noe praktisk. Prosjektet vil gå over flere semestre, der studenter jobber både parallelt og tar over prosjektet etter hverandre. Dette har medført at det har blitt lagt stor vekt på at studentenes bidrag er utformet på en slik måte at de kan brukes til noe av neste student. Prosjektet koordineres av KDA på tvers av utdanningsinstitusjonene NTNU, Høgskolen i Buskerud og Universitetet i Agder.

Denne rapporten er hovedsaklig delt inn i fire deler. Til å begynne med vil det bli laget en prototype av kommunikasjonsrammeverket (kapittel 3). Basert på en evaluering av prototypen blir det bestemt hvordan det endelige systemet skal deles opp i moduler og settes sammen (kapittel 4). Til slutt forklares hvordan modulene til bakkestasjonen og UAVen er implementert (kapittel 5 og 6). Prototypen ble laget først, men ellers følger ikke rapporten noen streng kronologisk struktur.

Det er produsert en gjennomgang av hvordan Robot Operating System[12] (ROS) settes opp for krysskompilering (vedlegg A) samt en gjennomgang av hvordan Gumstix starter Internett, VPN og ROS-programmer automatisk (vedlegg B). En DVD er vedlagt med all kildekode og noen videoer som demonstrerer at ting virker som de er presentert her.

Det ble til å begynne med valgt å gå et steg videre med høstprosjektet. I starten av semestret ble det redegjort for hvordan 3D-rekonstruksjonen kunne gjøres euklidsk korrekt. Resultatet av dette er vedlagt som vedlegg C.

Underveis i prosjektet har det blitt rapportert om fremgang til KDA. Blant annet er det satt sammen to presentasjonsslides som er vedlagt som vedlegg D.

Det gikk med mye tid av denne masteroppgaven på å krysskompilere ROS-noder til Gumstix. Men det er det nettopp denne krysskompileringen som vil være nyttig å lese om for senere års studenter om de velger å bruke ROS og det er den viktigste delen av denne

oppgaven. Arbeidsmengden bak krysskompileringen gjenspeiles best i vedlegg A.

Gjennom semestret har undertegnede veiledet to grupper med studenter i faget *Eksperter i Team*. Den ene gruppen jobbet med I^{2}C-komponentene og den andre gruppen jobbet med å sette i stand mobilt bredbånd for Gumstix. Resultatet fra begge grupper ble koblet sammen i denne diplomoppgaven, men det ble kun tid til å teste det mobile bredbåndet.

For å ta bilder med Gumstixen ble det lenge fastholdt ved at OpenCV[10] var veien å gå. Dette biblioteket gjør det enkelt å ta bilder med alle Video4Linux[7] v1 og v2 enheter. Dessverre ble det problematisk å krysskompilere OpenCV på en måte som fungerte.

OpenCV ble gitt opp og andre løsninger ble undersøkt. En pakke i *sail-ros-pkg*-pakkebrønnen[18] kalt *uvc_cam* virket lovende og krysskompilertes som den skulle. Dessverre virket ikke *uvc_cam* med kameraet som var koblet til.

For å teste at kameraet virker har det tidligere blitt brukt videodog[20]. Det virket dermed som en god idé å pakke videodog inn som en ROS-node og få videodog til å distribuere bildene over nettverket istedenfor å lagre til disk. Dette fungerte til en viss grad, men det oppstod problemer når bildene skulle pakkes ned i ROS sitt bildeformat som resulterte i at bildene som kom frem i andre enden var ubrukelige.

Løsningen ble til slutt å gjøre det samme med fswebcam[1] som ble prøvd med videodog. fswebcam ble altså pakket inn som en ROS-node og sender bilder i ROS sitt bildeformat over nettverket.

På HiBu har studentene klart å koble Gumstixen til I^{2}C-bussen[23] kapittel 4.2. Det er gjort et raskt forsøk på å reprodusere dette, men når det ikke ble oppnådd kontakt med I^{2}C-bussen fra ROS-noden som ble skrevet så var det ikke tid til overs for å debugge dette. Selv om ikke I^{2}C-noden ble ferdig så demonstrerer koden hvordan I^{2}C-bussen kan brukes fra en ROS-node. Det vil være en enkel sak å ferdigstille denne noden når I^{2}C-bussen og Gumstix kommuniserer med hverandre.

Kapittel 2

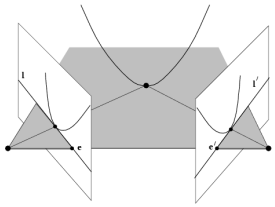
Bakgrunn

2.1 Tidligere arbeid

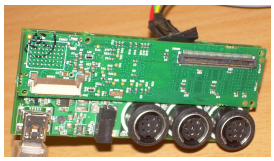
Det har vært studenter på NTNU som har jobbet med utvikling av en UAV siden høsten 2006 / våren 2007 da Jon Bernhard Høstmark, Edgar Bjørntvedt og Mikael Kristian Eriksen skrev sine masteroppgaver. Teoretiske og praktiske aspekter ble vurdert og det ble gjort beregninger og simuleringer. Både flykropp og interne komponenter er byttet ut siden den gang, men GPS-modulen og IMU-modulen fra CyberSwan brukes videre i det nye systemet. Opparbeidet kompetanse og erfaringer fra CyberSwan-prosjektet videreføres ved at Jon Bernhard Høstmark er ekstern veileder og koordinator for det nåværende prosjektet.

Etter CyberSwan ble det lagt stor vekt på modularitet for å tilrettelegge prosjektet å løpe over flere år. Våren 2008 så en gruppe fra Eksperter i Team nærmere på IMU-modulen og lagde sine egne kretskort for kommunikasjon mellom den og en bakkestasjon. Sommeren samme år jobbet en gruppe studenter fra både NTNU og Høgskolen i Buskerud hos Kongsberg Defence & Aerospace. Der ble de veiledet av Jon Bernhard Høstmark gjennom sommerprosjektet *Local Hawk*, hvor de utviklet separate moduler for IMU, GPS, servokontroller og trådløs kommunikasjon. Modulene kommuniserer med hverandre over I^2C, og var første skritt mot et fullstendig modulært system.

Høsten 2008 plukket undertegnede opp stafettpinnen på NTNU og utforsket muligheten for å inkorporere videokameraet i resten av systemet. Det ble utviklet et program for å matche bilder tatt etter hverandre med samme kamera, og hente ut informasjon fra disse om omgivelsene. Det ble bestemt at dagens algoritmer for matching av bilder ikke er gode nok, men at det finnes et stort potensiale for bruk av kamera som sensor. Bildebehandling krever en del prosessorkraft og det ble derfor valgt å bruke et Gumstix hovedkort for dette. Dette hovedkortet kjører Linux og gjør det også enkelt å kjøre andre krevende operasjoner som for eksempel avanserte reguleringsalgoritmer eller kunstig intelligens. Hovedkortet ble konfigurert og testet til å ta bilder.



Figur 2.1: Multiple view geometry[22]. Med flere bilder tatt fra forskjellige vinkler er det mulig å rekonstruere en scene.



Figur 2.2: Gumstix Verdex XL6P-hovedkort montert på et console-VX utvidelseskort. Denne lille datamaskinen ble konfigurert for bruk i en UAV.

Undertegnede har denne våren (2009) veiledet to grupper i emnet *Eksperter i Team* som har jobbet med videreutvikling av flyet. Den ene gruppen har testet og programmert I^2C-modulene (GPS, IMU, servo og trådløs kommunikasjon), mens den andre gruppen har satt opp mobilt bredbånd for Gumstix-hovedkortet. Dette er brukt videre i denne oppgaven.

2.2 Krav til bakkestasjonen og rammeverket for nettverkskommunikasjon

En bakkestasjon er den delen av kommunikasjonslinken som står på bakken. Den er ansvarlig for å ta i mot, arkivere, behandle, analysere og videreformidle innkommende data samt sende tilbake kommandoer til flyet. Den er i det hele tatt bindeleddet mellom brukeren og flyet.

Modularitet har vært et viktig stikkord for alle flyets aspekter. Dette videreføres også når det kommer til software. Prosjektet vil gå over flere semestre og brukes av flere studenter som kan ha nytte av å bytte ut eller legge til moduler. Det vil da være en fordel å sette seg inn i gammel kildekode og kompilere alt om igjen hver gang noe skal modifiseres eller byttes ut.

Det er viktig at kommunikasjonsrammeverket skal være enkelt å ta i bruk for senere års studenter.

Krav til båndbredde og oppetid skal minimeres ettersom kommunikasjonslinken antas å være ustabil. Forfatteren av en ny modul skal ikke trenge å tenke på oppsett av nettverksforbindelser, hvordan finne riktige IP-adresser og lignende.

2.3 Gumstix

Gumstix Inc. er et firma som holder til i USA og produserer forskjellige hovedkort og utvidelseskort. Av hovedkort produserer og selger de Basix, Connex, Verdex XL4P/XL6P/Pro og Overo Earth/Water/Air/Fire. Hovedkortet som brukes i det ubemannede flyet er en Verdex XL6P. Det har 128MB RAM, 32MB on-board minne og 600Mhz klokkefrekvens. Verdex har også USB-host støtte, som gjør det mulig å koble til webkameraer

For å få enkel tilgang til USB-host signaler, I^2C-signaler og tilkobling av strøm brukes et console-VX-utvidelseskort. I figur 2.2 er en Verdex XL6P koblet til en console-VX.

Gumstix bruker OpenEmbedded-rammeverket[11] for å holde orden på avhengigheter og krysskompilere pakker. Linux-distribusjonen som brukes er Ångström Linux.

2.4 Rammeverket Robot Operating System

ROS (*Robot Operating System* eller også *Robot Open Source*) ble opprinnelig utviklet av Stanford Artificial Intelligence Laboratory i 2007, men har siden 2008 hovedsaklig blitt utviklet av Willow Garage. Willow Garage er et firma som ble grunnlagt av Scott Hassan (best kjent som grunnlegger av Yahoo! Groups), i 2006 for å fremme utvikling av roboter til ikke-militære formål. I tillegg til å utvikle ROS, støtter Willow Garage også opp om prosjektene OpenCV (bildebehandling og datasyn), Player (ROS sin forgjenger) og TREX (A.I.).

ROS er en distribuert meldingsbasert plattform som oppnår høy granularitet ved å dele opp systemet i noder, og gjøre det enkelt for disse å kommunisere med hverandre. Dette er praktisk om man i fremtiden velger å bruke flyet som en relaystasjon. Om det i tillegg skal koordineres med andre autonome rom-, luft-, bakke- eller undervanns-fartøy er det veldig praktisk å ha tilgjengelig en ferdig plattform for kommunikasjon dem imellom.

ROS er et operativsystem i den forstand[13] at det tilbyr hardware-abstraksjon, lavnivå kontroll av enheter, implementasjoner av mye brukt funksjonalitet, overføring av meldinger mellom prosesser og pakkehåndtering.

2.4.1 Noder og meldinger

Med ROS settes hver del av systemet opp som separate noder. Dette gjør det enkelt å modifisere, bytte ut, eller legge til noder. Hver del av systemet kan utvikles for seg selv og kan til og med skrives i et annet programmeringsspråk om det er ønskelig. For eksempel kan det være en node for kamera, en for GPS, en for IMU, en for servo, en for A.I. og en for styresystemet. I vårt tilfelle vil det nok være mer fornuftig å slå sammen GPS, IMU og servo til en felles I^2C-node. På bakken kan hver programmodul være en egen node.

Noder skilles fra hverandre ved at alle har sitt eget navn, og alt som trengs å vite om en node for å få tak i den, er dens navn. Noder kommuniserer med hverandre ved hjelp av meldinger, som er datastrukturer ikke ulike structs i C. Meldinger utveksles enten gjennom *emner* eller *tjenester*.

I kulissene kjører hver node sin egen XML/RPC-tjener som brukes for å kommunisere med andre noder. Tjeneren bindes til en tilfeldig ledig port på vertsmaskinen.

2.4.2 Emner

Emner egner seg til streaming av data og er basert på *publish/subscribe* paradigmet. Meldinger publiseres på et emne (engelsk: *topic*). Noder må abonnere på emnet for å få disse meldingene. Flere noder kan utgi et emne og flere noder kan abonnere på et emne samtidig. For eksempel kan data fra IMU både abonneres på av en annen node på UAVen som driver med overordnet navigasjon, og av bakkestasjonen.

Utgivere og abonnenter er ikke klar over hverandre. Idéen er å gjøre produksjon og forbruk av informasjonen uavhengig av hverandre. Emner kan tenkes på som en meldingsbuss med et unikt navn som alle kan koble seg til og sende eller motta meldinger av en gitt type.

2.4.3 Tjenester

I tilfeller der informasjonen skal gå begge veier er *request/response* paradigmet mer praktisk. Tjenester (engelsk: *services*) er rett og slett ROS sin implementasjon av remote procedure calls (RPC). Det er (i likhet med emner) elegant implementert og gjør det raskt og enkelt å lage og kalle funksjoner på tvers av noder.

Tjenester vil i praksis brukes til å sette parameterverdier, som for eksempel å slå av og på kamera-filtere, eller oppdateringsfrekvens for GPS- og IMU-målinger. Det vil også brukes til å sette waypoints eller gi nye oppdrag.

2.4.4 Master-noden

Master-noden er noden som gjør at andre noder finner hverandre. Alle noder må vite om denne noden ved oppstart så den har typisk en fast IP (eller et fast domenenavn) og port. Noder må si ifra om at de vil utgi eller abonnere på et emne, eller tilbyr visse tjenester, til denne noden. Noden står kun for koordinering og videreføring av metadata. Det er altså ikke noe faktisk data som går gjennom noden.

2.4.5 Datatransport

Fra nodeskribentens ståsted er det ukjent hvordan kommunikasjonen foregår. ROS gjør alt det kan for å velge den optimale metoden med tanke på båndbredde. TCP er utbredt fordi det tilbyr en enkel og pålitelig informasjonsstrøm; pakkene ankommer i riktig rekkefølge, og blir sendt om igjen hvis de ikke kommer fram. Dette er greit for kablede

nettverk, men for trådløse nettverk eller mobilt bredbånd kan det fort bli mye pakketap, i hvilket tilfelle UDP kan være et bedre alternativ. Om begge nodene befinner seg på samme maskin er det unødvendig å dele opp meldingene i pakker og sende de gjennom nettverksstacken; de kan heller kommunisere ved hjelp av en IPC-mekanisme som for eksempel et delt minneområde. Hvis flere av nodene befinner seg i samme subnet som utgiveren kan det være fornuftig å benytte UDP broadcast eller til og med *Spread*[19].

2.4.6 ROS som utviklingsmiljø

Utvikling av ROS-noder foregår normalt ved hjelp av ROS sin egen toolchain. ROS noder er organisert i pakker. En pakke kan inneholde flere programmer, biblioteker, meldingsdefinisjoner og tjenestedefinisjoner.

Hver pakke har en manifest.xml-fil som inneholder navn, en kort beskrivelse av pakken og en liste over alle andre pakker som pakken avhenger av. For eksempel inneholder pakken *image_msgs* definisjonen av meldingsformatet *Image*. Noder som sender eller mottar bilder i ROS sitt standard bildeformat må inkludere *image_msgs*, og om *image_msgs* er listet opp i manifest.xml vil ROS også sørge for at *image_msgs* er kompilert på forhånd når kommandoen *rosmake* kjøres.

Systemavhengigheter i Ubuntu Linux kan håndteres enkelt med *rosapt-get*, og manifest.xml kan hjelpe til i disse tilfeller også ved å spesifisere hvilke pakker som trengs for hvilke versjoner av operativsystemet.

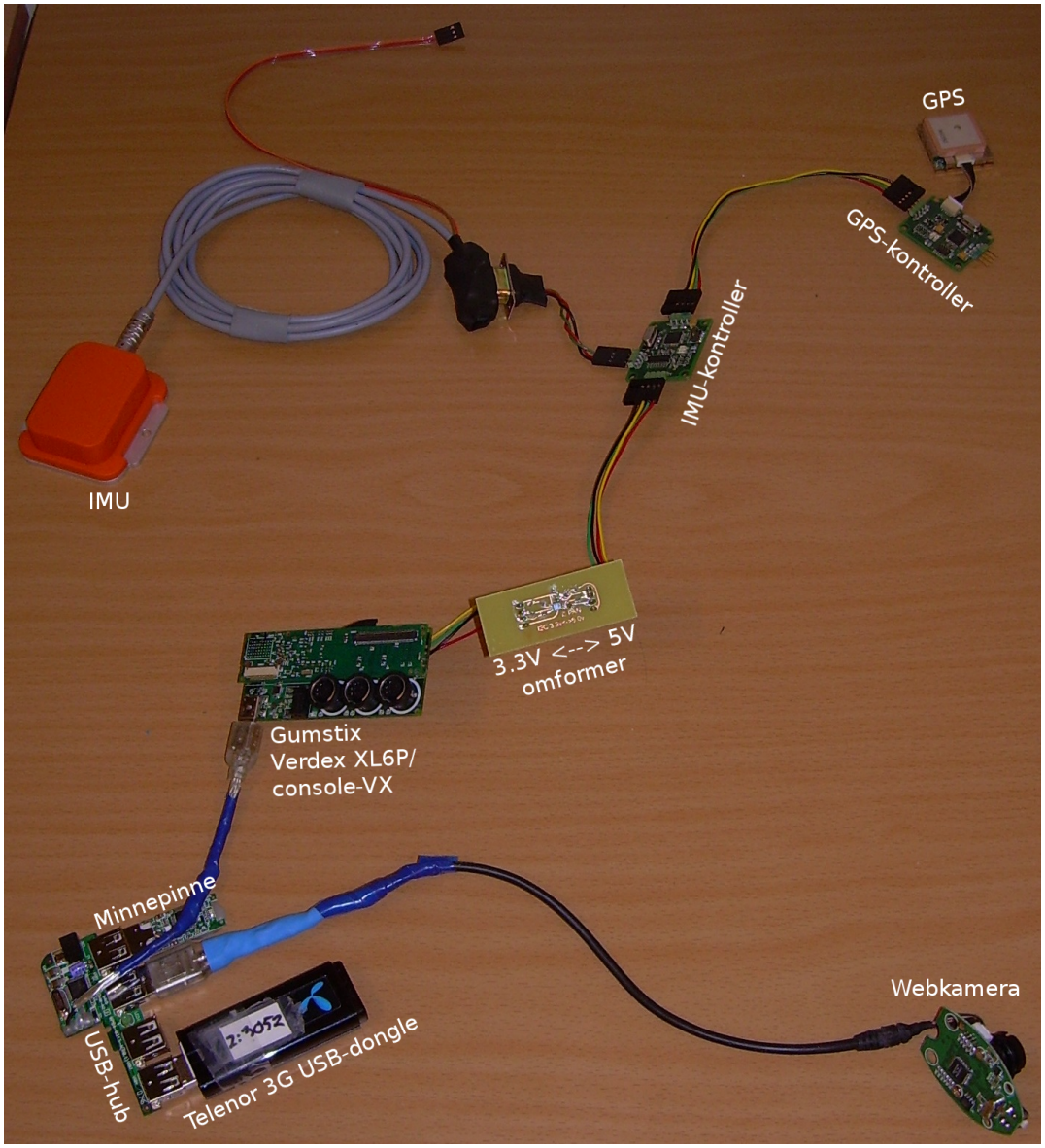
rosmake, som brukes for å kompilere nodene, er bygget rundt CMake-kompileringsrammeverket. Å skrive en CMakeLists.txt-fil er både enklere og mer kompatibelt enn å skrive Makefile-filer eller tilsvarende for hver ny pakke[13].

2.4.7 Andre features

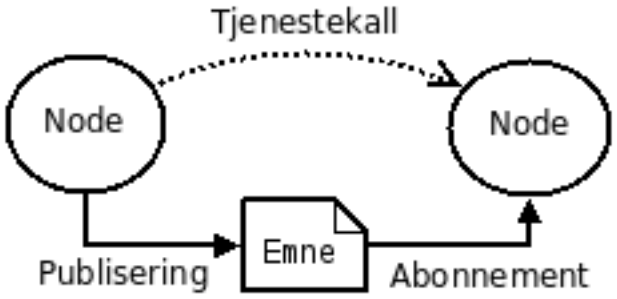
ROS kan brukes fra Octave[9] og utviklerne jobber i øyeblikket med automatisk .mex-fil-kompilering rett fra ROS-utviklingsverktøyene. Dette er praktisk for videre bearbeidelse og plotting av data.

ROS fungerer også med iPod Touch/iPhone (kalt *ROSpod*). Bruksnytten av dette kan stilles spørsmålstegn ved, men det kunne iallefall vært interessant å styre flyet med en slik.

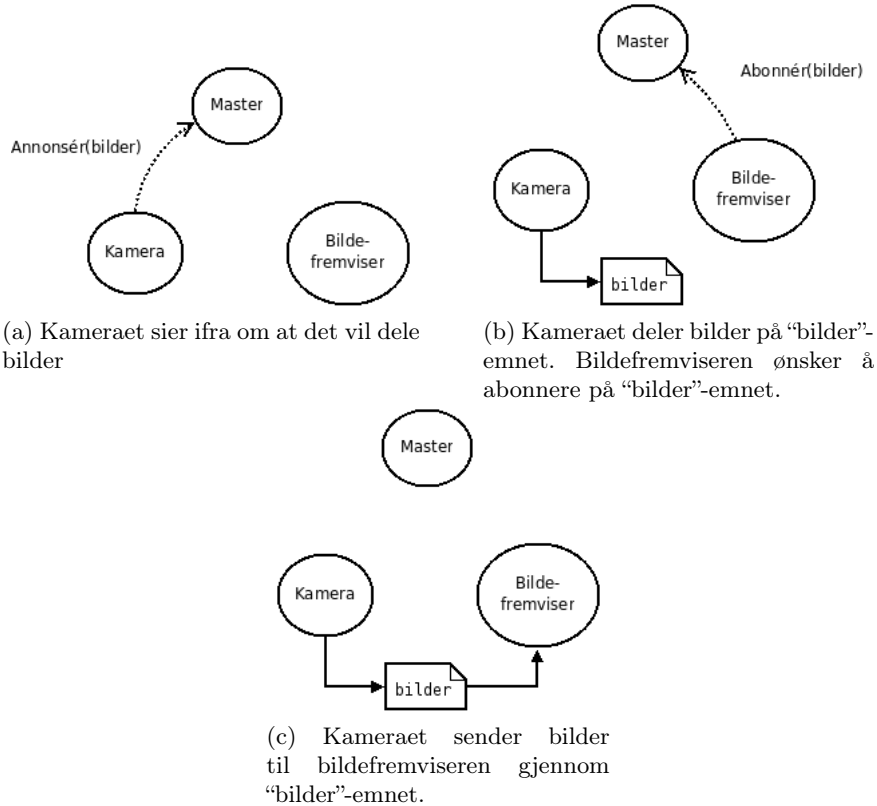
ROS er heller ikke bundet til et spesielt programmeringsspråk, og i skrivende stund er både C++ og Python støttet, med planlagt støtte for Java.



Figur 2.3: Her er de fleste sentrale komponentene koblet sammen.



Figur 2.4: Symbolforklaring for grunnleggende konsepter innen ROS.



(a) Kameraet sier ifra om at det vil dele bilder

(b) Kameraet deler bilder på "bilder"-emnet. Bildefremviseren ønsker å abonnere på "bilder"-emnet.

(c) Kameraet sender bilder til bildefremviseren gjennom "bilder"-emnet.

Figur 2.5: Et typisk hendelsesforløp for publish/subscribe-paradigmet. Kameraet melder ifra om at det har bilder å tilby og bildefremviseren melder ifra om at det ønsker å motta bilder. Når masteren får vite at bildefremviseren vil ha bilder så forteller den til bildefremviseren at det i øyeblikket finnes én node, kameraet, som tilbyr bilder. Bildefremviseren kontakter så kameraet direkte (ikke vist) og spør etter bilder. Bildene strømmer så direkte fra kameraet til bildefremviseren (det går aldri faktisk data gjennom masteren).

Kapittel 3

Prototyping av testdesign

3.1 Introduksjon

Før funksjonelle programmer kan utvikles må selve nettverkskommunikasjonen være på plass som et gjenbrukbart bibliotek/rammeverk. Til å begynne med kan kravene som er spesifisert virke forholdsvis greie å oppfylle så et enkelt bibliotek for nettverkskommunikasjon mellom bakkestasjon og UAV ble skrevet med en gang og er lagt fram og evaluert her.

3.2 Designvalg og virkemåte

Kravene til rammeverket nevner modularitet, enkel nettverksoppkobling og god utnyttelse av båndbredde som viktige momenter. For å oppnå god modularitet må biblioteket vite hvilken modul hver innkommende melding tilhører så rammeverket vet hva det skal gjøre med meldingen. Altså må hver melding merkes med en identifikator som identifiserer den tilhørende modulen.

Enkel nettverksoppkobling innebærer at IP-adresser skal kunne endres underveis og abstraheres vekk fra brukeren av biblioteket.

Hvordan bruk av båndbredde best reguleres avhenger av hva slags informasjon som overføres. For eksempel kan en kameramodul endre komprimeringsinnstillinger i tillegg til antall bilder i sekundet den sender, mens en GPS-modul kun kan endre antall målinger den sender per sekund. Det er altså opp til hver modul å optimalisere bruken av tilgjengelig båndbredde.

Biblioteket skal ha funksjoner for initialisering av nettverksforbindelsen og sending av meldinger, i tillegg til at det skal videreformidle innkomne meldinger til definerte håndtererfunksjoner tilhørende hver modul.

3.3 Implementasjon

Det ble laget et enkelt rammeverk i C. Dette rammeverket legger til rette for modularitet og abstraherer til en viss grad vekk selve nettverksforbindelsen. All kommunikasjon går over UDP, delvis med tanke på at gammel data stort sett ikke er interessant og trenger dermed ikke sendes på nytt, IP-adressen kan endres raskt slik at forbindelsen må settes opp på nytt, men også fordi det rett og slett er enklere å bruke UDP til å begynne med. UDP gir i det hele tatt mindre overhead, og som en connectionless protokoll oppfører den seg mer oversiktlig i tilfelle brudd på forbindelsen. Dette systemet skal fungere i sanntid. Hvis en modul mener det er viktig at data skal komme fram må den drive feilsjekk og re-sende meldinger selv.

Rammeverket for nettverkskommunikasjon (libuav.c) er satt sammen som vist på figur 3.1. Én tråd er dedikert til sending av meldinger ("sender"), én tråd er dedikert til mottak ("receiver") og én tråd er dedikert til å behandle innkomne meldinger ("handler"). UAV og bakkestasjon fungerer på stort sett samme måte, men blir initialisert til å være en av delene ved initialisering ("initComm").

Alle meldinger som sendes ("sendMessage") blir lagt i kø ("sendBuffer") og sendt av gårde ved første anledning. Alle mottatte meldinger blir lagt i kø ("recvBuffer") og behandlet ved første anledning. Hvis meldingen krever et svar, så er det en innebygget funksjon ("requestResponse") som ikke returnerer før et svar har blitt mottatt - eller etter en gitt tid. Når en mottatt melding er merket som et svar, så havner den i et eget mellomlager ("responseBuffer") hvor den blir plukket opp av denne funksjonen.

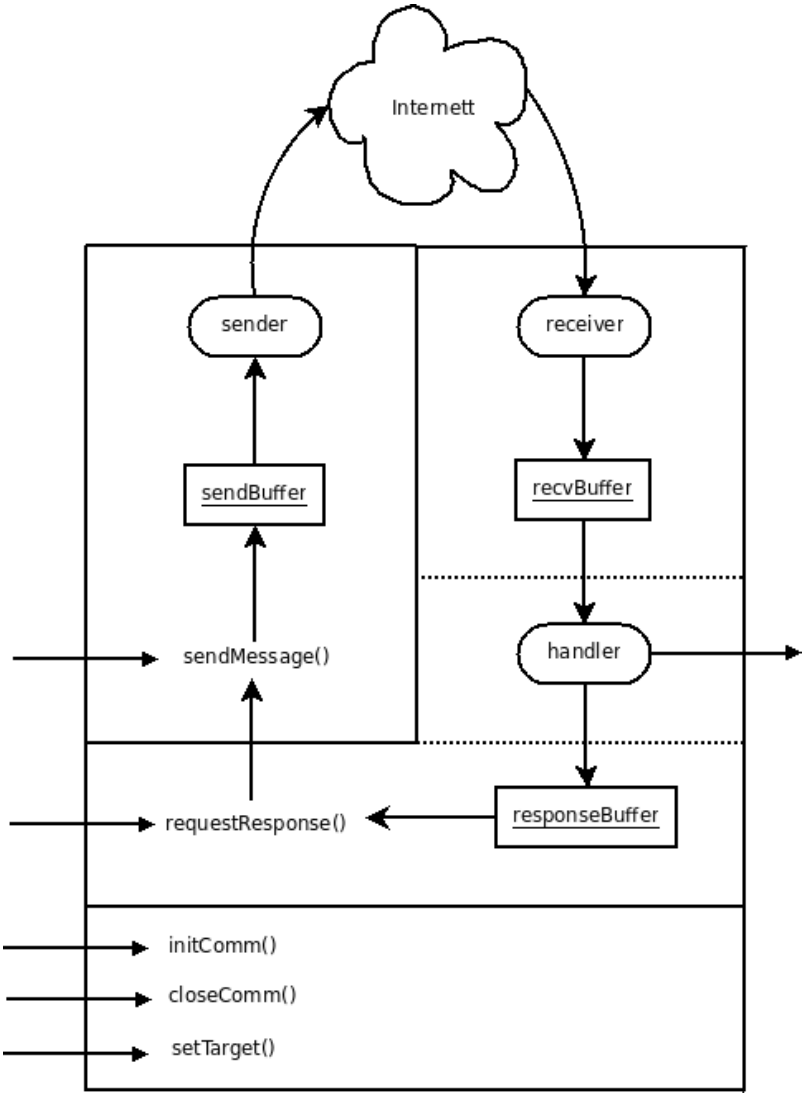
Nettverksadressen til den andre noden kan oppdateres når som helst ("setTarget"), og mellomlagrene kan tømmes, samt trådene avsluttes når forbindelsen ikke skal brukes mer ("closeComm").

3.3.1 Håndterere

Moduler, som for eksempel "ping", "camera", "imu", "error", "info", etc. må registrere seg selv ("registerHandler") i rammeverket for å kunne motta meldinger. De kan også velge å melde seg ut av rammeverket om de vil det ("unregisterHandler"). De registrerer seg ved å fortelle hvilken funksjon som skal kalles når en melding adresseres til dem.

3.3.2 Moduler

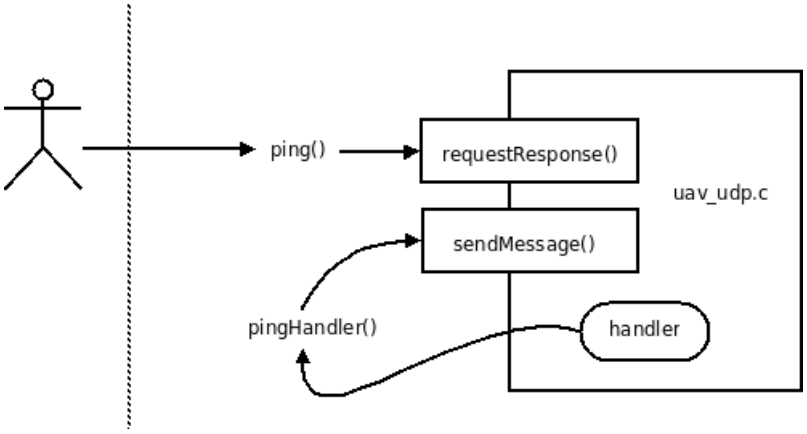
Moduler abstraherer vekk ned- og opp-pakking av meldinger, og tilbyr praktisk funksjonalitet til brukeren. Det ble skrevet ferdig en modul for pinging mellom nodene og en halvferdig modul for overføring av bilder.



Figur 3.1: Konseptuell fremstilling av libuav. Pilene indikerer informasjonsflyt / hvilken vei meldinger beveger seg, rundinger symboliserer tråder og firkanter symboliserer mellomlager. Rammene rundt indikerer hvilke funksjoner/mellomlagere/tråder som funksjonelt hører sammen.

Ping-modulen tilbyr to funksjoner til brukeren; ping() og loadTest().

ping(int timeout_us) gjør nytte av rammeverkets funksjonalitet for svar-på-forespørsel. I den andre enden blir meldingen videresendt til modulens håndterer ("pingHandler"). Her bestemmes det at ettersom dette var en ping-forespørsel, så skal det umiddelbart sendes et ping-svar. Når ping-svaret mottas tilbake på den første noden blir det lagt i mellomlageret hvor svar-meldinger legges. Derifra plukkes det opp igjen (med mindre dette har



Figur 3.2: Konseptuell fremstilling av ping-modulen. Når ping()-funksjonen kalles, bruker den rammeverkets requestResponse()-funksjon for å sende en henvendelse til den andre maskinen. Når den andre maskinen mottar og håndterer henvendelsen blir den sendt til ping-modulens tilsvarende håndterer, pingHandler(), som umiddelbart svarer på henvendelsen. Når dette svaret når den første maskinen igjen, returnerer requestResponse(). ping() regner ut latency basert på hvor lang tid det tok å kalle requestResponse(). Kort fortalt tar ping() tiden på hvor lang tid det tar å gjøre et RPC-kall.

tatt mer enn timeout_us) av svar-på-forespørsel-funksjonen ("requestResponse"), som i sin tur returnerer det til ping()-funksjonen.

loadTest(int payload_size, int *count, int timeout_us) kan brukes for å teste båndbredde, eller for å redusere tilgjengelig båndbredde for andre moduler. Den sender count meldinger av størrelse payload_size, og tidsavbrudd timeout_us.

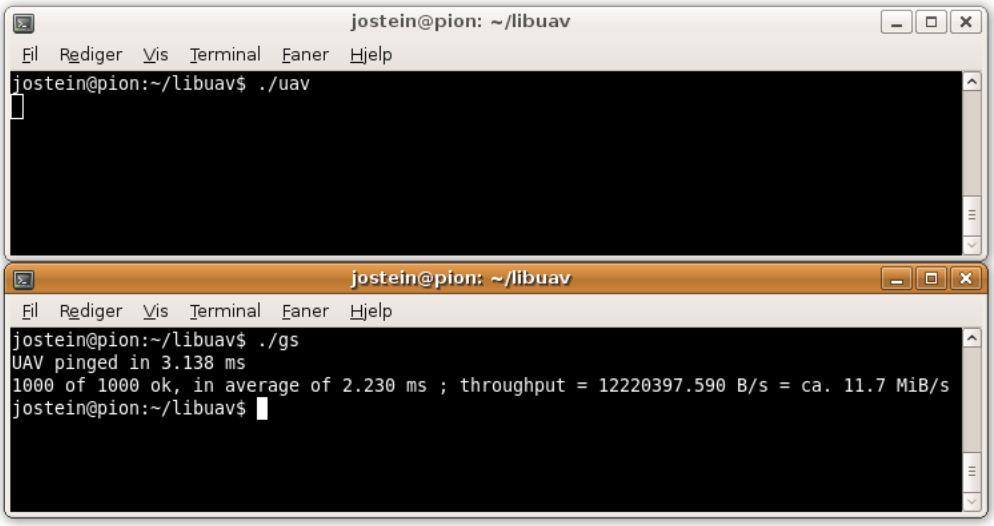
3.4 Resultater og testing

Det er skrevet et program som tester rammeverket og ping-modulen ved å først kalle ping() og skrive ut latency; og deretter sende ut 1000 store meldinger vha. loadTest for å regne ut båndbredde (se figur 3.3).

3.5 Diskusjon

"requestResponse"-funksjonen i biblioteket oppstod fra behovet for å få et svar på sendt melding. Det har vist seg i praksis å fungere som et Remote Procedure Call (RPC). Denne RPC-funksjonaliteten er veldig nyttig og bør inkluderes videre.

Problemet med å få bakkestasjonen til å finne UAVen når IP-adressene settes dynamisk i



Figur 3.3: Først startes programmet ment å kjøre på UAVen, og deretter programmet ment å kjøre på bakkestasjonen. Bakkestasjonen pinger så UAVen, skriver ut litt info, og avslutter seg selv.

begge ender er blitt åpenbart. UAV og bakkestasjon vil trenge en tredjepart med statisk IP-adresse som begge vet om for å finne hverandre.

For å skrive nye moduler til dette biblioteket kreves en forståelse av hvordan libuav fungerer internt. Dette er uheldig med tanke på lærekurven for nye utviklere. Bedre abstraksjon er nødvendig.

Kapittel 4

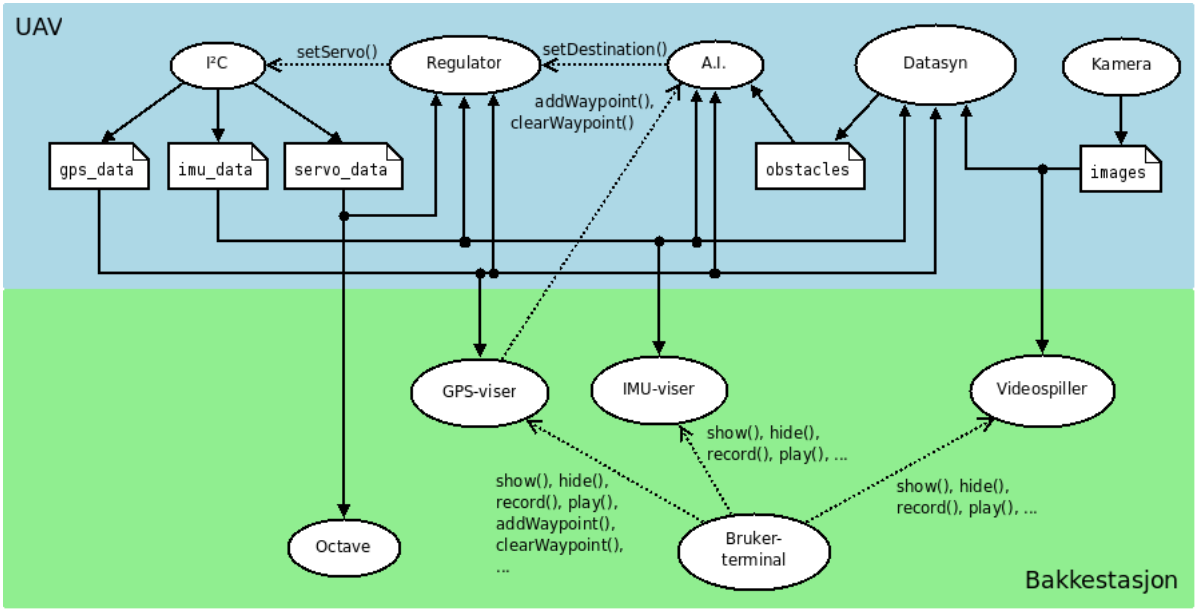
Overordnet design av systemet

All funksjonalitet, både på selve UAVen og på bakkestasjonen, skilles ut i egne noder.

I^2C-enhetene får sin egen felles node ettersom det ikke er mye som skiller de sett fra Gumstixen sitt perspektiv. Det kan også potensielt bli problematisk å ha flere noder som bruker I^2C-bussen samtidig.

Foruten noder for webkamera og I^2C er det naturlig å skille ut flyets autopilot (regulatoren), kunstig intelligens (A.I.) og visuell deteksjon av hindere (datasyn) som egne noder.

Samhandling mellom nodene, inkludert Octave-bindinger om Octave vil brukes, er illustrert i figur 4.1.



Figur 4.1: Hvordan alt skal henge sammen. Navn på noder er kun for illustrasjonsformål og kan få andre navn i den ferdige implementasjonen.

Kapittel 5

Implementasjon av bakkestasjon

5.1 Bakkestasjonen

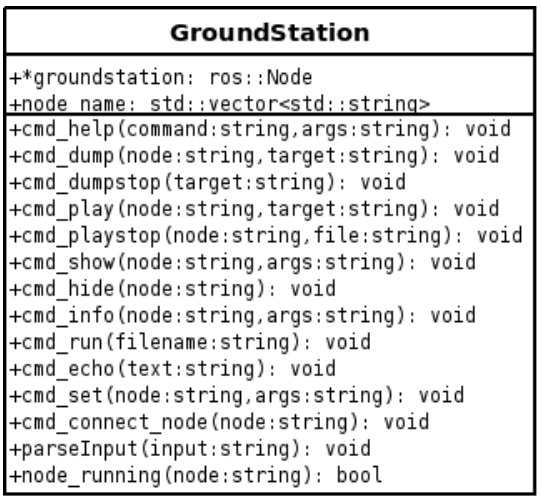
Bakkestasjonen er implementert som en C++-klasse (se figur 5.1) som opererer som en ROS-node. Implementasjonen er i utgangspunktet beregnet på å være en backend for et tekstbasert brukergrensesnitt. Men ettersom bakkestasjonen er separert ut i en egen klasse og fordi måten den er skrevet på ikke forutsetter et tekstbasert brukergrensesnitt, så går det fint å anvende klassen som backend i et grafisk brukergrensesnitt også.

Klassen tar inn kommandoer gjennom parseInput(string input)-funksjonen, som i første omgang omfatter følgende kommandoer: help, exit/quit, dump, dumpstop, play, playstop, show, hide, info, set, connect, run. Kommandoene er forholdsvis selvforklarende (*show webcam*, *play imu imurecording.dat*, *dump gps gpsrecording.dat*, og så videre) men en mer utdypende forklaring av hver kommando kan fåes med *help*-kommandoen. Disse forklaringene er gjengitt i vedlegg E.

Mottak, visning etc. av bilder, IMU-, GPS- og servo-data er skilt ut i egne noder/klienter som kjører lokalt. Dette gjør det lett å legge til, fjerne eller bytte ut funksjonalitet. For eksempel kan GPS-klienten byttes ut med en ny en uten at resten av bakkestasjonen vet noe om det. En ny node som for eksempel en servo-klient kan også legges til uten at det krever modifikasjon av resten av systemet.

5.1.1 Terminalgrensesnitt

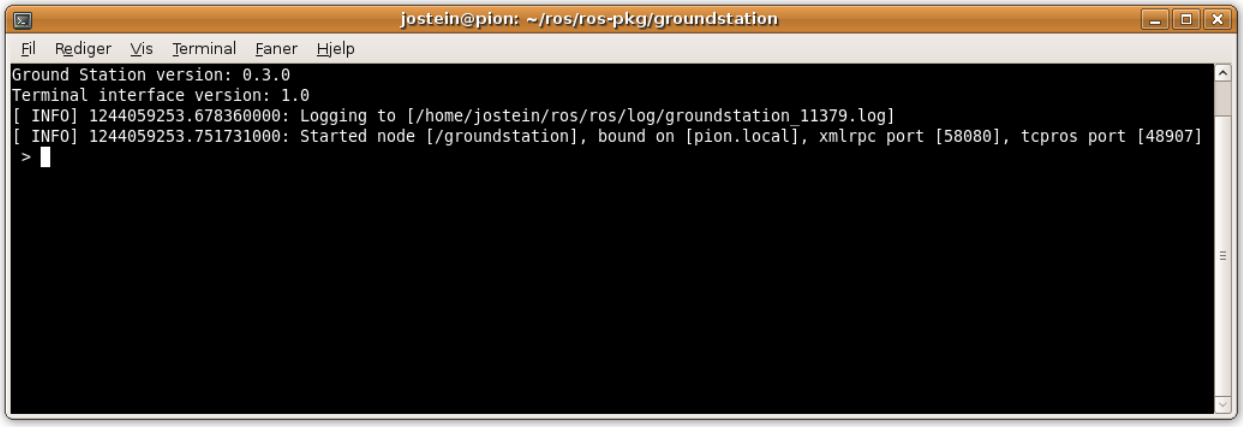
Sammen med bakkestasjonsklassen er det også skrevet et enkelt tekstbasert brukergrensesnitt for å interagere med den. Ved oppstart kjører terminalen et skript som brukeren kan fylle med kommandoer. For eksempel kan det defineres at det finnes lokale noder på bakkestasjonen for å håndtere *gps*, *imu* og *webcam*, og at *webcam*-behandleren skal starte å vise bilder umiddelbart ved oppstart.



Figur 5.1: UML-diagram av GroundStation-klassen.

Kildekode 5.1 main() i terminal.cpp

```
int main(int argc, char** argv) {
        printf("Ground Station version: %d.%d.%d\n"
               "Terminal interface version: %d.%d\n",
                GS_VERSION_MAJOR,GS_VERSION_MINOR,
                GS_VERSION_SUBMINOR,TERM_VERSION_MAJOR,
                TERM_VERSION_MINOR);

        ros::init(argc, argv);

        GroundStation *gs = new GroundStation();
        gs->cmd_run("init.gs"); // oppstartsskript
        string input;
        while (gs->groundstation->ok()) {
                cout << " > ";
                getline(cin, input);
                gs->parseInput(input);
        }

        return 0;
}
```



Figur 5.2: Skjermbilde av bakkestasjonens terminal-interface.

5.2 Kamera

Kamera-klienten tar imot bilder i ROS sitt eget bildeformat, gjør de om til OpenCV sitt IplImage-format og viser de på skjermen. Klienten har også innebygd histogram equalization som gir økt kontrast i tilfelle det er vanskelig å se motivet.

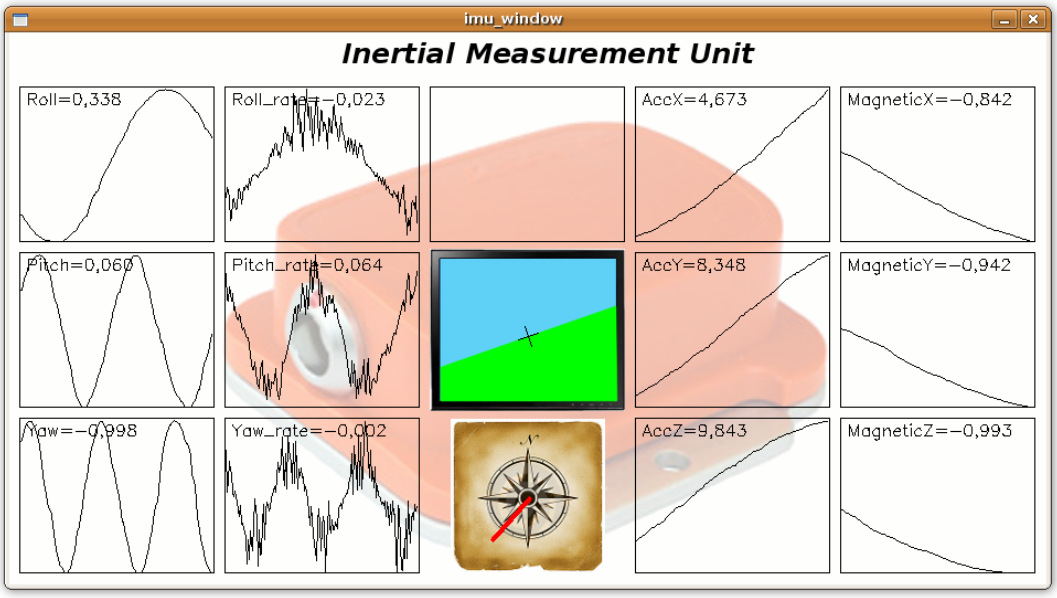
5.3 IMU

Meldinger til IMU-klienten inneholder roll/pitch/yaw, tidsderiverte av roll/pitch/yaw, akselerasjon i X/Y/Z-retningene og magnetfeltstyrke i X/Y/Z-retningene. IMU-klienten plotter alle disse dataene i egne plot, i tillegg til å tegne et kompass for magnetfeltet og et enkelt HUD (Heads-Up Display) for roll/pitch/yaw.

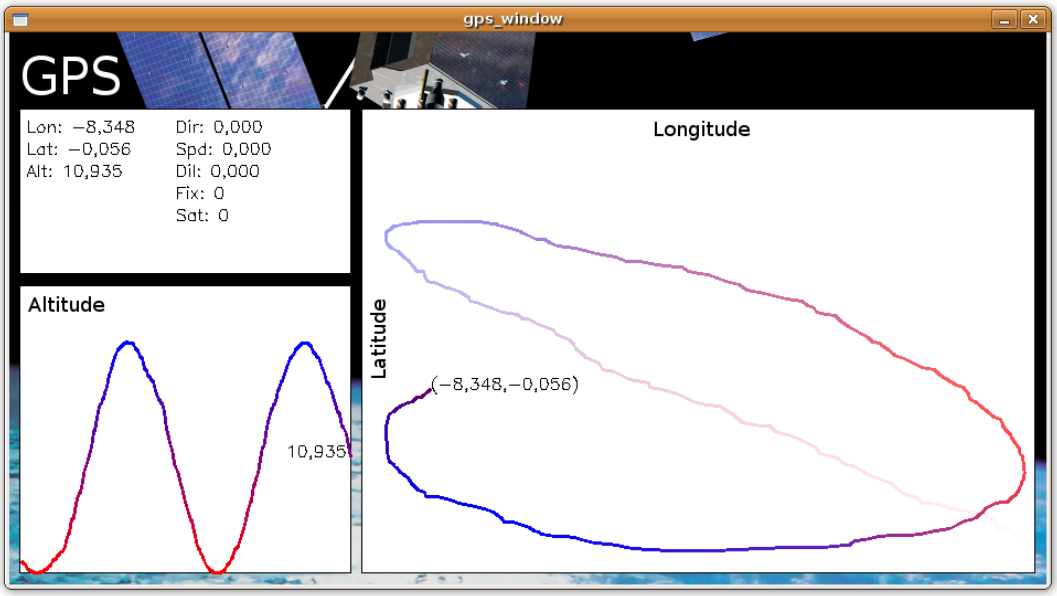
5.4 GPS

GPS-klienten plotter lengdegrad mot breddegrad i et kart med fargekoding for å vise høyden (rødt for lavt og blått for høyt). Høyden plottes også i et eget plott mot tid i tillegg til at høyde, lengdegrad, breddegrad samt diverse annen info som følger med GPS-meldingene. Hver GPS-måling/melding inneholder i tillegg til høyde/lengdegrad/breddegrad også retning, fart, målefeil og antall satellitter brukt.

Figur 5.3: Skjermbilde av bakkestasjonens bildeviser. Dette bildet er streamet over mobilt bredbånd.



Figur 5.4: Skjermbilde av bakkestasjonens IMU-klient. Det var nødvendig å konvertere mellom ROS sitt eget bildeformat og OpenCV sitt IplImage-format. Dette var ikke ferdig implementert i ROS, men med god hjelp fra mailinglisten ble dette løst [17]. Dataene som er plottet her er kun testdata.



Figur 5.5: Skjermbilde av bakkestasjonens GPS-display. Dataene som er plottet her er kun testdata.

Kapittel 6

Implementasjon av noder for Gumstix

6.1 Kamera

Programmet som tar bilder fra webkameraet, konverterer bildene til ROS sitt *Image*-meldingsformat og publiserer disse meldingene på *images*-emnet er basert på fswebcam[1]. Denne omskrivningen av fswebcam er dubbet *rosfswebcam* og mye av den avanserte funksjonaliteten til fswebcam er kommentert ut til å begynne med.

rosfswebcam støtter både Video4Linux v1 og v2 enheter. Altså er det kompatibelt med ganske mange webkameraer. Innstillinger for rosfswebcam lastes fra en eksempel.conf-fil som kan se ut for eksempel som i kildekode 6.1. fswebcam støtter blant annet skalering av bildet og omgjøring til gråtoner. Dette ville gjort bildene mindre og mer passende for overføring over mobilt bredbånd, men rosfswebcam støtter enda ikke dette.

6.2 I^2C

6.2.1 Fysisk oppkobling til I^2C-bussen

Gumstix bruker 3,3V på I^2C-bussen, mens GPS-/IMU-/servo-kortene bruker 5V. Et kretskort (designet av studenter på HiBu[23]) for å gjøre om mellom spenningsnivåene ble produsert. Kretskortet er vist i figur 6.1.

6.2.2 Implementasjonen

I^2C-noden er basert på kildekode som studentene på HiBu skrev for å teste kommunikasjon mellom IMU og Gumstix. Koden deres ble modifisert og gjort om til en ROS-node. Når det ikke ble oppnådd kommunikasjon med I^2C-bussen fra Gumstixen, ble heller ikke koden for noden ferdigstilt. Se figur 6.2 for

Kildekode 6.1 eksempel.conf

```
# fswebcam configuration for ROS / Gumstix setup
# [2009-05-23: josteinaj@gmail.com]

quiet

# Alt som er kommentert ut finner fswebcam av seg selv.
# Denne konfigurasjonen er testet at fungerer med flyet.
#device      "v4l2:/dev/video0"
#input       0
#palette     YUV420P
resolution  320x240

skip    2         # Skip the first two frames.
frames  1         # And capture one.

scale 160x240
```

6.3 Automatisk oppstart

Studentene som jobbet med mobilt bredbånd til Gumstix fikk den koblet på nett, men oppkoblingen deres skjer manuelt via et skript fra kommandolinjen. I tillegg gikk det ikke an å koble til Gumstixen utenfra på grunn av en brannmur hos Telenor. Det ble nødvendig å finne en vei rundt brannmuren og automatisere tilkobling til Internett for at ROS-noder på Gumstix skulle kunne kommunisere med ROS-noder på bakkestasjonen.

Skriptet som kobler til mobilt bredbånd er her automatisert til å kjøre ved oppstart av Gumstix. For å tillate bakkestasjonen å ta initiativ og koble seg til flyet ble det i tillegg valgt å sette opp en VPN-forbindelse til NTNU (*login.stud.ntnu.no*). Programmet *vpnc* ble brukt til dette.

En *cron*-jobb starter skriptet beskrevet i kildekode 6.2 en gang hvert minutt.

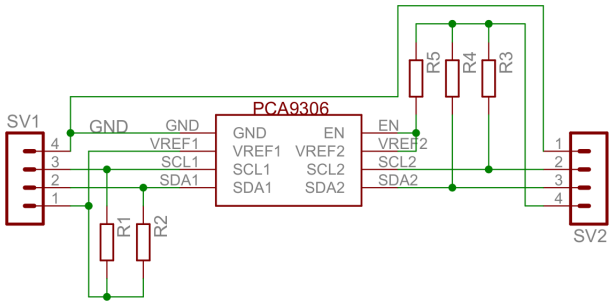
En Eksperter i Team-gruppe fikk Gumstix koblet til mobilt bredbånd, men det må kobles til manuelt ved hjelp av et skript. I tillegg gikk det ikke an å koble til Gumstixen fra Internett på grunn av en brannmur fra Telenors side. Det var nødvendig å finne en vei rundt brannmuren og automatisere tilkobling til Internett for at ROS-noder på Gumstix skulle kunne kommunisere med ROS-noder på bakkestasjonen.

Kildekode 6.2 pseudokode for hspda.sh. Når Gumstixen slåes på kjøres kalles up().

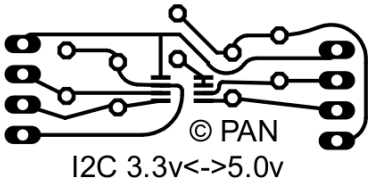
```
DEVICE=/dev/ttyHS0
...
up() {
  start modeswitch
  installer driver
  koble til Internett
  start vpnc
  start watchdog.sh
}

down() {
  koble fra vpnc
  koble fra Internett
}
...
```

Kildekode 6.3 watchdog pseudokode

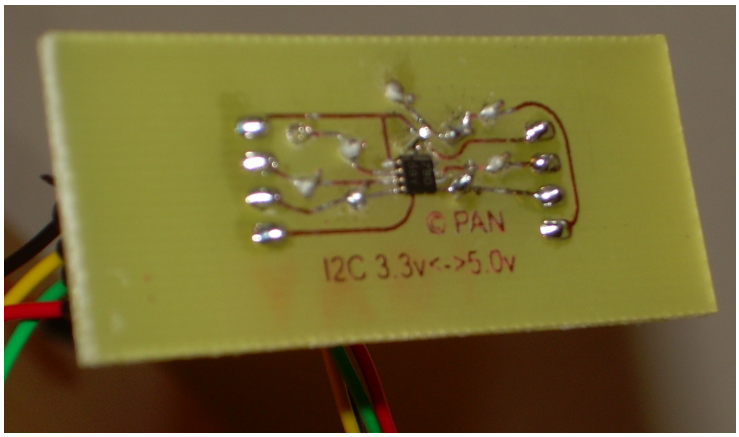
```
- Bestem IP-adressen tilordnet VPN-forbindelsen
- Finn USB-minnepinnen
- For hver node listet opp i 'launch'-filen fra minnepinnen:
    - Hvis IP-adressen har forandret seg, start noden om igjen
    - Hvis noden ikke er i gang, start den
```



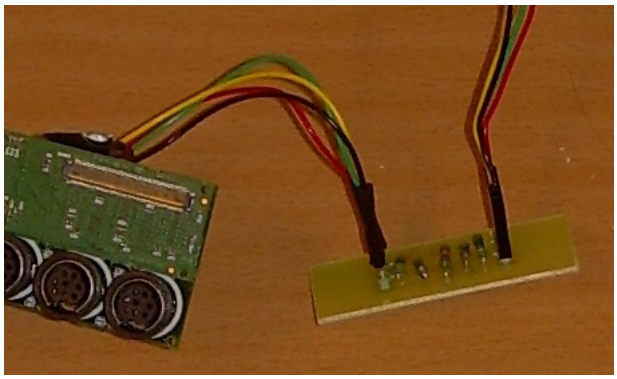
(a) I²C kretsskjema



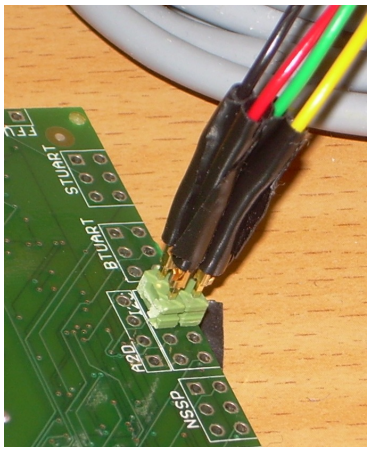
(b) I²C kretsutlegg



(c) I²C kretskort - kretser

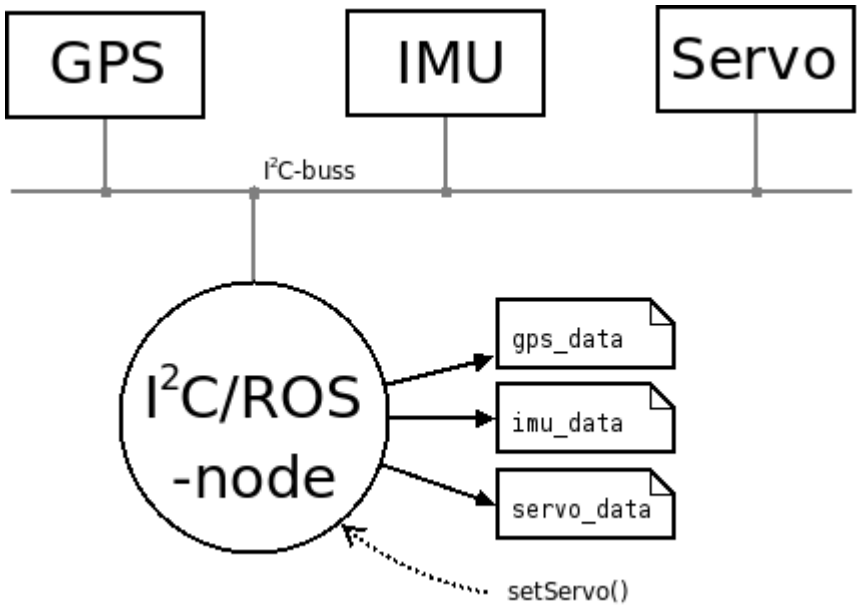


(d) I²C kretskort - tilkobling til kretskortet

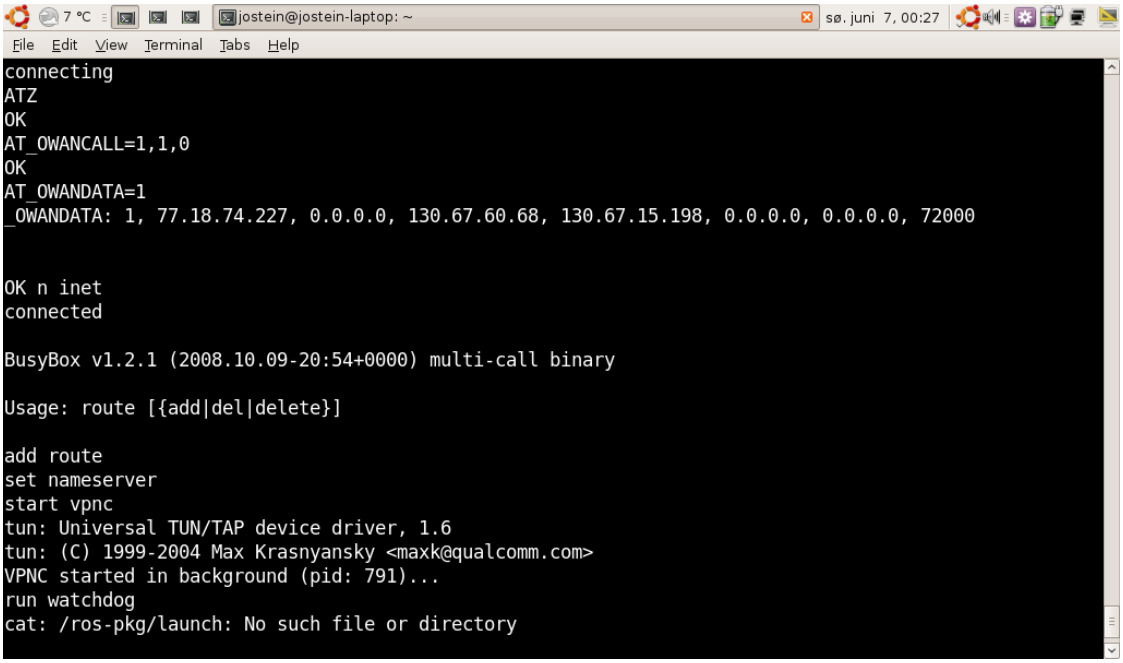


(e) I²C kretskort - tilkobling til console-VX/Gumstix

Figur 6.1: I²C-kretskortet som brukes for å koble sammen Gumstix og GPS-/IMU-/servo-enhetene. Kretsskjema og utlegg er designet av studentene ved HiBu, mens selve kortet er produsert ved NTNU.



Figur 6.2: Data fra I^{2}C-bussen (i grått) videreformidles på de virtuelle bussene (emnene) i ROS. Disse emnene kan abonneres på av for eksempel av reguleringsalgoritmer, logge-programmer eller visualiserings-/plotte-programmer som vist i figur 4.1 på side 18.



Figur 6.3: Skjermbildet viser den delen av oppstarten som kobler til Internett og VPN i tillegg til å starte alle nodene via oppstartsskriptet watchdog.sh. På akkurat dette skjermbildet finner ikke watchdog.sh-skriptet *launch*-filen ettersom minnepinnen ikke er koblet til.

Kapittel 7

Resultater

Testnoder ble skrevet for å teste GPS- og IMU-plotting på bakkestasjonen. Nodene sender ut sinus-bølger kombinert med litt hvit støy for hver enkelt verdi i GPS-/IMU-meldingene. Disse testnodene ble kompilert og testet for å kjøre både fra bakkestasjonen og flyet.

Både streaming av testdata fra test-IMU-noden til IMU-klienten, fra test-GPS-noden til GPS-klienten og webkamera-noden til webkamera-klienten fungerte. Disse virket både i en testkonfigurasjon der alt kjørte på samme PC eller mellom to PCer i tillegg til mellom Gumstix og PC.

Det har vist seg at når kun kameramodulen og kameraklienten er aktivert, det vil si at det går veldig lite data opp til flyet mens båndbredden ned til bakkestasjonen brukes for fullt, så oppnås en hastighet på ca. 44KiB/s (downlink til bakkestasjon).

På vedlagt CD er det lagt ved én video som demonstrerer kameraet, én video som demonstrerer IMU- og GPS-klientene, og én video som ved hjelp av ROS sin innebygde *rxgraph* viser hvordan noder kobles til og fra systemet.

Det ble totalt skrevet 1528 linjer kildekode (talt opp ved hjelp av David A. Wheeler sitt program *SLOCCount*, skript for utregning er vedlagt på DVD).

Kapittel 8

Diskusjon

Det ligger til rette for hardware-in-loop-testing av utstyr. For eksempel kan kunstig generert eller tidligere logget GPS- og IMU-data sendes ut på deres respektive emner, og en regulator- eller en kunstig intelligens-modul vil ikke kunne se forskjell på disse og virkelige data.

Videooverføringen slik den er nå har to problemer; den går for sakte, og det er for stor forsinkelse. Det er i øyeblikket ikke mulig å styre kun basert på disse bildene. Det er flere sekunder mellom hvert bilde, og bildene kommer flere sekunder for sent. Dette er en konsekvens av at det tar for lang tid å sende bildene fra flyet til bakkestasjonen. Hvis vi får senket tiden det tar å sende et bilde vil både bildefrekvensen og bildeforsinkelsen forbedres.

Testingen viste at vi får en båndbredde på 44KiB/s. For å streame bilder med rimelig hastighet må bildene komprimeres. For eksempel vil et bilde på 320x240 piksler komprimert med JPEG kvalitet 20 bruke ned mot 4KiB. Det vil være rimelig å anta at vi kan oppnå fem bilder i sekundet og allikevel ha båndbredde til overs for andre formål dersom JPEG-komprimering blir brukt. Det var under prosjektet ingen støtte for JPEG-komprimering i ROS.

Kapittel 9

Konklusjon

Robot Operating System er et meget nyttig verktøy når det skal distribueres informasjon både på tvers av programmer og på tvers av maskiner. Tiden det tar å sette seg inn i og ta i bruk ROS er mindre enn å lage et egne funksjoner for nettverkskommunikasjon når det trengs.

Den eneste ulempen med å bruke ROS er i øyeblikket er at de statisk lenkede programmene blir for store til å oppbevares på Gumstixens interne flash-minne. Det ville vært bedre om bibliotekene var lenket til programmene dynamisk. I så fall er det mulig at de får plass på Gumstixen, men om de fortsatt ikke gjør det så vil det frigjøre en del plass i minnet.

For å oppnå en fornuftig bildefrekvens og bildeforsinkelse ved streaming av bilder over mobilt bredbånd bør det byttes til JPEG-komprimerte bilder[14]. Implementasjon av dette bør integreres med ROS sitt allerede eksisterende bildeformat for å unngå at forskjellige bildeformat brukes på tvers av noder. I det dette prosjektet ble avsluttet ble det sluppet en pakke som gjør nettopp dette. Den heter *image_publisher*[15]

ROS er under konstant utvikling og det ble nylig lansert en ROS-pakke som tar bilder fra vanlige digitalkameraer (gphoto2[16]). Det kan være interessant å teste denne for å ta høyoppløselige stillbilder.

Når det gjelder VPN og *vpnc* på Gumstix så er det en ulempe at brukernavn og passord lagres i klartekst. Det ville vært nyttig om prosjektet hadde en egen brukerkonto hos NTNU, eller et tilsvarende sted som tilbyr en unik IP-adresse ut mot Internett.

En interessant mulighet vil være å koble flyet opp til en flysimulator (for eksempel FlightGear[2]). En ROS-node kan fungere som bindeledd mellom flysimulatoren og flyets hardware. Dette hardware-in-loop-oppsettet[3] vil lette arbeidet med utvikling av regulatorer og kunstig intelligens betraktelig. Det vil ikke lenger risikeres skade på flyet, og i tillegg vil tiden mellom implementasjon, testing og forbedring av implementasjon kortes

ned. Flysimulatoren, gjerne med oppdaterte kart og live oppdatering av vær, kan også brukes som en del av bakkestasjonen når flyet er ute og flyr.

Vedlegg A

GumROS

A.1 Installasjon av GumROS

Dette er en gjennomgang av hvordan man setter opp ROS for Gumstix ("GumROS"). Guiden er myntet på brukere av Ubuntu, men det er ikke store modifikasjoner som må til før det virker med andre distribusjoner av Linux. Dette er første versjon av tutorialen. En mer oppdatert versjon ligger på `http://pr.willowgarage.com/wiki/gumros`

Til å begynne med, sørg for at Gumstix OpenEmbedded er installert. gcc og g++ er i så fall plassert i ${GUMSTIXTOP}/tmp/cross/arm-angstrom-linux-gnueabi/bin/. En vanlig installasjon av ROS for ikke-krysskompilering er også nødvendig.

A.1.1 Avhengigheter

ROS avhenger av to biblioteker; Boost 1.35 (eller nyere) og Apache log4cxx 0.10. log4cxx avhenger i sin tur av APR og APR-util. Det beste ville vært å kompilere disse avhengighetene med bitbake, men ettersom det ikke finnes ferdige bitbake-oppskrifter for dette (APR og APR-util finnes, Boost finnes i for gammel versjon og log4cxx ikke i det hele tatt), er det greiere å gjøre det manuelt.

Det er mulig å sette CXX, CC, CPP og så videre, men det er enklere å rett og slett legge til ${GUMSTIXTOP}/tmp/cross/arm-angstrom-linux-gnueabi/bin/ i starten av $PATH-variabelen. Fjern dette igjen etter at APR, APR-util og log4cxx er installert, før Boost kompileres. For eksempel:

```
OLD_PATH="$PATH"
PATH="${GUMSTIXTOP}/tmp/cross/arm-angstrom-linux-gnueabi/bin:$PATH"
... kompiler APR, APR-util og log4cxx her ...
PATH=${OLD_PATH}
```

APR

Last ned APR 1.3.3, pakk ut og konfigurer:

```
mkdir -p ~/gumros/ros-deps
cd ~/gumros/ros-deps
wget http://www.powertech.no/apache/dist/apr/apr-1.3.3.tar.gz
tar -xzf apr-1.3.3.tar.gz
cd apr-1.3.3
./configure --prefix=/opt/gumros/
            --disable-shared
            --enable-static
            --host=arm-linux
            ac_cv_file__dev_zero=yes
            ac_cv_func_setpgrp_void=yes
            apr_cv_tcp_nodelay_with_cork=no
            apr_cv_process_shared_works=no
            apr_cv_mutex_robust_shared=no
            ac_cv_sizeof_struct_iovec=8
            apr_cv_mutex_recursive=yes
```

Før APR kan kompileres må det gjøres en liten endring. For å unngå en feilmelding om at PATH_MAX ikke er definert kan dette legges til i ~/gumros/ros-deps/apr-1.3.3/include/apr.h ved linje 407:

```
#if !defined(PATH_MAX)
#define PATH_MAX 4096
#endif
```

Deretter er det bare å kompilere og installere:

```
make
sudo make install
```

APR-util

Last ned, konfigurer, kompiler og installer APR-util på denne måten:

```
cd ~/gumros/ros-deps
wget http://www.powertech.no/apache/dist/apr/apr-util-1.3.4.tar.gz
tar -xzf apr-util-1.3.4.tar.gz
cd apr-util-1.3.4
./configure --prefix=/opt/gumros
            --disable-shared
            --enable-static
            --without-pgsql
            --without-sqlite2
            --without-sqlite3
            --host=arm-linux
            --with-apr=/opt/gumros
            --with-expat=builtin
            ac_cv_file__dev_zero=yes
            ac_cv_func_setpgrp_void=yes
            apr_cv_tcp_nodelay_with_cork=no
            apr_cv_process_shared_works=no
            apr_cv_mutex_robust_shared=no
            ac_cv_sizeof_struct_iovec=8
make
sudo make install
```

log4cxx

Last ned, konfigurer, kompiler og installer log4cxx på denne måten:

```
cd ~/gumros/ros-deps
wget http://pr.willowgarage.com/downloads/apache-log4cxx-0.10.0-wg_patched.tar.gz
tar -xzf apache-log4cxx-0.10.0-wg_patched.tar.gz
cd apache-log4cxx-0.10.0
./configure --prefix=/opt/gumros
            --host=arm-linux
            --with-apr=/opt/gumros
            --with-apr-util=/opt/gumros
            ac_cv_file__dev_zero=yes
            ac_cv_func_setpgrp_void=yes
            apr_cv_tcp_nodelay_with_cork=no
            apr_cv_process_shared_works=no
            apr_cv_mutex_robust_shared=no
            ac_cv_sizeof_struct_iovec=8
make
sudo make install
```

Boost

Boost bruker sitt eget system for konfigurering, kompilering og installasjon som kalles bjam og må være installert på maskinen. Begynn med å installere bjam, laste ned Boost og pakke ut Boost:

```
sudo apt-get install bjam
cd ~/gumros/ros-deps
wget http://heanet.dl.sourceforge.net/sourceforge/boost/boost_1_38_0.tar.gz
tar -xzf boost_1_38_0.tar.gz
```

Boost må konfigureres for kryss-kompilering. Legg til en linje til tools/build/v2/usr-config.jam med innholdet "*using gcc : din-g++-versjon : sti-til-g++ ;* ". Det kan for eksempel gjøres som dette:

```
GPP_PATH=${GUMSTIXTOP}/tmp/cross/arm-angstrom-linux-gnueabi/bin/g++
GPP_VER=`${GPP_PATH} -v 2>&1 | tail -1 | awk '{print $3}'`
echo "using gcc : ${GPP_VER} : ${GPP_PATH}  ; " > tools/build/v2/user-config.jam
```

Nå er Boost klar for krysskompilering. Konfigurer, kompiler og installer på denne måten:

```
sudo bjam --toolset=gcc-${GPP_VER} --prefix=/opt/gumros install
```

Det kommer til å stå at ikke alle biblioteker blir kompilert, men så lenge libboost_date_time, libboost_signals og libboost_thread kompileres (se i /opt/gumros/lib) bør det være bare å gå videre.

A.1.2 ROS

.bashrc.gumros

Lag først en GumROS-spesifik variant av den allerede eksisterende ~/.bashrc.ros og kall den *~/.bashrc.gumros*. Fyll den med disse linjene:

```
export ROS_ROOT=~/gumros/ros
export ROS_PACKAGE_PATH=~/gumros/ros-pkg
export ROS_MASTER_URI=http://localhost:11311/
export ROS_BINDEPS_PATH=/opt/gumros
export ROS_BOOST_PATH=/opt/gumros
export PYTHONPATH=$PYTHONPATH:$ROS_ROOT/core/roslib/src
export OCTAVE_PATH=$OCTAVE_PATH:$ROS_ROOT/core/experimental/rosoct
source $ROS_ROOT/tools/rosbash/rosbash
```

Pass på at også .bashrc.ros definerer ROS_BINDEPS_PATH og ROS_BOOST_PATH, ellers vil de krysskompilerte variantene brukes til vanlig kompilering. Merk også at til forskjell fra .bashrc.ros så blir ikke $ROS_ROOT/bin lagt til $PATH, ettersom programmene uansett ikke vil kjøre lokalt. De som ligger i den vanlige ROS-installasjonen (typisk ~/ros/ros/bin) brukes istedenfor.

Laste ned

Sjekk ut ROS fra svn:

```
cd ~/gumros
svn co ros.svn.sourceforge.net/svnroot/ros/tags/stable ros
svn co personalrobots.svn.sourceforge.net/svnroot/personalrobots/pkg/trunk ros-pkg
```

Før kompilering må igjen PATH_MAX defineres for å unngå feilmeldinger. Legg disse tre linjene inn ved linje 48 i ~/gumros/ros/core/roscpp/include/ros for å fikse dette:

```
#if !defined(PATH_MAX)
#define PATH_MAX 4096
#endif
```

Python vil gi feilmeldinger om det blir forsøkt å generere meldinger eller tjenester for det, så det må deaktiveres:

```
echo "#!/bin/sh" > ~/gumros/ros/core/rospy/scripts/genmsg_py
echo "#!/bin/sh" > ~/gumros/ros/core/rospy/scripts/gensrv_py
```

rostoolchain.cmake

Filen ~/gumros/ros/rostoolchain.cmake er hvor GumROS blir fortalt hvordan den skal krysskompilere. Opprett filen og fyll den med disse linjene:

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_C_COMPILER ${GUMSTIXTOP}/tmp/cross/arm-angstrom-linux-gnueabi/bin/gcc)
set(CMAKE_CXX_COMPILER ${GUMSTIXTOP}/tmp/cross/arm-angstrom-linux-gnueabi/bin/g++)
set(CMAKE_FIND_ROOT_PATH ${GUMSTIXTOP}/tmp/cross)
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM BOTH)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

Kompilering

Kompilering er nesten likt det som står i ~/gumros/ros/Makefile foruten at etter at genmsg_cpp må alle programmene dens erstattes av de tilsvarende ikke-krysskompilerte variantene. GumROS bør kompileres i denne rekkefølgen:

```
cd ~/gumros/ros/tools/rospack && make
cd ~/gumros/ros/3rdparty/gtest && make
cd ~/gumros/ros/core/genmsg_cpp && make
cd ~/ros/ros/core/genmsg_cpp
cp genmsg genmsg_lisp genmsg_oct genmsgtest ~/gumros/ros/core/genmsg_cpp/
cp gensrv gensrv_lisp gensrv_oct submsgs ~/gumros/ros/core/genmsg_cpp/
cd ~/gumros/ros/core/roslib && make
cd ~/gumros/ros/core/rospy && make
cd ~/gumros/ros/3rdparty/pycrypto && make
cd ~/gumros/ros/3rdparty/paramiko && make
cd ~/gumros/ros/3rdparty/xmlrpc++ && make
cd ~/gumros/ros/tools/roslaunch && make
cd ~/gumros/ros/test/rostest && make
cd ~/gumros/ros/core/rosconsole && make
cd ~/gumros/ros/core/roscpp && make
cd ~/gumros/ros/core/rosout && make
```

Nå skal alt være klart og GumROS står klar til bruk. Bare husk å kjøre *source ~/.bashrc.gumros* når en ny terminal åpnes, ellers vil rosmake forsøke å kompilere alt ved hjelp av de vanlige kompilatorene. For å bytte tilbake til vanlig (ikke-kryss-)kompilering, bruk *source ~/.bashrc.ros*.

Faktisk så kan det være ganske praktisk å lage enkle snarveier for å bytte mellom vanlig kompilering og krysskompilering:

```
ln -s ~/.bashrc.ros ~/ros/ros/bin/rospc ;
ln -s ~/.bashrc.ros ~/gumros/ros/bin/rospc ;
ln -s ~/.bashrc.gumros ~/ros/ros/bin/rosgum ;
ln -s ~/.bashrc.gumros ~/gumros/ros/bin/rosgum ;
```

Med dette kan man skrive *source rosgum* istedenfor *source ~/.bashrc.rosgum* og *source rospc* istedenfor *source ~/.bashrc.ros* som er en fordel på norske tastatur.
For å teste at alt virker kan det være lurt å prøve å kompilere de medfølgende eksemplene:

```
rosmake roscpp_tutorials
```

A.1.3 OpenCV

Det er enda ikke helt klart hvordan man får OpenCV til å virke med kompilering av statiske programmer. Standardoppsettet tilbyr heller ingen enkel måte å kompilere OpenCV som et statisk bibliotek. Når dette blir mulig vil det beskrives online på `http://pr.willowgarage.com/wiki/gumros`.

A.2 Utvikling av ROS-noder

Det kan være en fordel å utvikle og kompilere for PC til å begynne med, og teste på Gumstix senere. Online finnes det gode gjennomganger av eksempler på bruk av tjenester og emner, og derfor gjengis de ikke her.

For å krysskompilere og overføre de kompilerte programmene til Gumstix, er det enklest å kompilere statiske programmer. Dessverre er ikke kompilering av statiske programmer

støttet i ROS helt enda (se `https://prdev.willowgarage.com/trac/ros/ticket/912`), så det må hackes til.

Heretter antas det at ROS sine environment variables peker på GumROS (i praksis at *source rosgum* er kjørt i terminalvinduet).

- Først er det lurt å ha kompilert programmet dynamisk med rosmake for å være sikker på at alle avhengigheter er kompilert og virker.
- Videre må alle ROS-bibliotekene være kompilert statisk. Legg til "set(ROS_BUILD_STATIC_LIBS true)" i *~/gumros/ros/rosconfig.cmake* (opprett rosconfig.cmake hvis den ikke allerede finnes).
- Kjør *cd ~/gumros/ros && make*
- Endre ~/ros/ros/bin/rosboost-cfg (legg merke til ~/ros og ikke ~/gumros) og legg til " flags += lpthread -lros -llog4cxx -lexpat -laprutil -lapr"" som linje 214 (nøyaktig to mellomrom før 'flags' for å få riktig innrykk - som har mye å si i Python skript). Så lenge denne linjen finnes i rosboost-cfg vil ikke rosmake virke skikkelig. rosmake vil faktisk ødelegge avhengighetene slik at de må må kompileres på nytt igjen, så ikke bruk rosmake heretter. Er det behov for å kjøre rosmake, kommenter ut denne linjen fra rosboost-cfg.
- I pakken som skal kompileres statisk, endre CMakeLists.txt og legg til *set(ROS_BUILD_STATIC_EXES true)* etter inkluderingen av rosbuild.cmake. Legg gjerne inn *set(ROS_BUILD_TYPE Release)* eller set(ROS_BUILD_TYPE MinSizeRel)" for å få ned størrelsen.
- Nå er det bare å kjøre *make* i pakkens mappe for å kompilere. De endelige programmene blir ganske store så det kan være en idé å stippe de ned med *~/gumstix/gumstix-oe/tmp/cross/bin/arm-angstrom-linux-gnueabi-strip*.

Dersom vi ønsker å kompilere statiske programmer med ROS for vanlige PCer så er det et par ekstra ting som må gjøres først (Husk å ha kjørt *source rospc* først):

- Rekompiler først apr, apr-util og log4cxx som både statiske og delte biblioteker.
- Lenk alle *apr-1.** til *apr.** og *aprutil-1.* til aprutil.** i */opt/ros/lib*.
- Videre må det gjøres stort sett det samme som for krysskompilering (bortsett fra at /usr/bin/strip brukes istedenfor *.../arm-angstrom-linux-gnueabi-strip*.

Ferdig kompilerte noder plasseres på et minnekort som nevnt i B.4.

Vedlegg B

Oppsett av automatisk oppstart

B.1 Internett

Opprett */etc/init.d/hspda.sh* og fyll den med dette skriptet. Skriptet er utviklet av studentgruppen som ordnet mobilt bredbånd[21], men er modifisert for å la det starte automatisk.

```
#!/bin/sh

DEVICE=/dev/ttyHS0
TMPFIL=/tmp/connect.$$

up() {
  # kjorer modeswitch
  /usr/hspda/usb_modeswitch_arm_static
  sleep 5

  # installerer driver
  /sbin/modprobe hso

  echo connecting
  rm -f $TMPFIL
  (
    /usr/sbin/chat -E -s -V -f /usr/hspda/dial.cht <$DEVICE > $DEVICE
  ) 2>&1 |tee $TMPFIL
  echo " "n inet
  echo connected
  echo " "
  PIP="`grep '^_OWANDATA' $TMPFIL | cut -d, -f2`"
  NS1="`grep '^_OWANDATA' $TMPFIL | cut -d, -f4`"
  NS2="`grep '^_OWANDATA' $TMPFIL | cut -d, -f5`"

  # lagt til
  ifconfig hso0 $PIP netmask 255.255.255.255 up

  # find the old default route and replace it
  ORT="`route | grep default | awk '{printf $8}'`"
  route delete default dev $ORT #lagt til
  echo "add route"
  # lagt til
  route add default dev hso0
```

```
  echo "set nameserver"
  (             # update the DNS
    echo "nameserver  $NS1"
    echo "nameserver  $NS2"
  ) > /etc/resolv.conf

  rm -f $TMPFIL

  echo "start vpnc"
  /usr/sbin/vpnc /etc/vpnc/vpnc.conf
  echo "run watchdog"
  /home/root/.ros/watchdog.sh
}

down() {
  echo disconnecting

  /usr/sbin/vpnc-disconnect

  /usr/sbin/chat -V -f /usr/hspda/stop.cht <$DEVICE >$DEVICE
  ifconfig hso0 down
  echo "reset nameserver"
}

init() {
  echo init

  /usr/sbin/chat -E -s -V -f /usr/hspda/init.cht <$DEVICE >$DEVICE
}

usage() {
  echo Usage: $0 \(start\|stop\|restart\|init\)
}

case "$1" in
  start)
    up
    ;;
  stop)
    down
    ;;
  restart)
    down
    up
    ;;
  init)
    init
    ;;
  *)
    usage
    ;;
esac
```

For at dette skriptet skal kjøres automatisk ved oppstart brukes *update-rc.d*[4]. Kjør:

```
update-rc.d hspda.sh defaults
```

Opprett mappen */usr/hspda*.

```
mkdir /usr/hspda
```

Skaff programmet *usb_modeswitch_arm_static* fra vedlagt DVD, eller som beskrevet i fagrapporten til studentgruppen som fikk Gumstixen opp på mobilt bredbånd[21], og legg

den i */usr/hspda*.

Opprett filen */usr/hspda/dial.cha* og fyll den med:

```
ABORT    ERROR
TIMEOUT  10
""       ATZ
OK       "AT_OWANCALL=1,1,0^m"
OK       "\d\d\d\d\d\d\dAT_OWANDATA=1^m"
OK       ""
```

Nå skal Internett koble seg til automatisk.

B.2 VPN

For å få VPN til å starte automatisk, må *vpnc* installeres. På utviklingsmaskinen, kjør *bitbake vpnc*. Finn frem til *vpnc_ 0.3.3-r1_ armv5te.ipk* i *~/gumstix/gumstix-oe/tmp/deploy/glibc/ipk/armv5te* og flytt den over til Gumstixen. Installer vpnc med:

```
ipkg install vpnc_0.3.3-r1_armv5te.ipk
```

Opprett filen */etc/vpnc/vpnc.conf*. Husk å bruke ditt eget brukernavn og passord og fyll den med dette[5]:

```
Interface name vpn0
IKE DH Group dh2
Perfect Forward Secrecy nopfs
IPSec gateway vpn.ntnu.no
IPSec ID alle
IPSec secret passord
Xauth username josteija
Xauth password ********
```

Nå skal også VPN fungere som det skal.

B.3 Oppstart av ROS-noder

Videre må også ROS-nodene starte automatisk. Dette er det watchdog.sh-skriptet som tar seg av. Opprett */home/root/.ros/watchdog.sh* og fyll filen med dette (filen inneholder lange linjer, derfor er linjene nummerert):

```
1  #/usr/bin/sh
2
3  IP_FILE=/home/root/.ros/ip
4  ROS_MASTER_URI=`/bin/cat /home/root/.ros/master`
5  NET_IF=vpn0
6
7  # Determine IP
8  ROS_HOSTNAME=`/sbin/ifconfig $NET_IF | /bin/grep inet | /usr/bin/awk -F: '{ print
       $2 }' | /usr/bin/awk '{ print $1 }'`
9  OLD_IP=`/bin/cat $IP_FILE`
10 if [ "$ROS_HOSTNAME" == "" ]; then
```

```
11   ROS_HOSTNAME=$OLD_IP
12 fi
13 if [ "$OLD_IP" != "$ROS_HOSTNAME" ]; then
14   echo "$ROS_HOSTNAME" > $IP_FILE
15 fi
16
17 # Locate USB-stick. (will usually be /media/hdd, but seems to vary)
18 ROS_PACKAGE_PATH=`/bin/mount | /bin/grep vfat | /usr/bin/head -n 1 | /usr/bin/awk
       '{ print $3 }'`/ros-pkg
19
20 # Make sure all nodes listed in $ROS_PACKAGE_PATH/launch are up and running and
       configured correctly
21 export ROS_HOSTNAME=$ROS_HOSTNAME
22 export ROS_MASTER_URI=$ROS_MASTER_URI
23 for node in `/bin/cat ${ROS_PACKAGE_PATH}/launch`; do
24   if [ "$OLD_IP" != "$ROS_HOSTNAME" ]; then
25         /usr/bin/killall $ROS_PACKAGE_PATH/$node
26   fi
27   if [ `/bin/ps | /bin/grep $ROS_PACKAGE_PATH/$node | /bin/grep -v grep | /usr/bin
         /wc -l` -lt 1 ]; then
28     echo starting $node
29     /usr/bin/nohup $ROS_PACKAGE_PATH/$node ROS_HOSTNAME=$ROS_HOSTNAME
           ROS_MASTER_URI=$ROS_MASTER_URI &> /dev/null < /dev/null &
30   fi
31 done
```

Vi vil at watchdog.sh skal kjøres en gang hvert minutt slik at noder som har kræsjet blir startet igjen. watchdog.sh starter ikke noder som allerede er startet. Vi sette opp watchdog.sh som en cron-jobb:

```
* * * * * /home/root/.ros/watchdog.sh
```

IP/hostname til masternoden defineres i en egen fil */home/root/.ros/master*:

```
echo "http://medea.austad.us:11311/" > /home/root/.ros/master
```

Nå er systemet konfigurert, men enda er ingen noder definert.

B.4 Hvordan definere programmer som skal startes opp

watchdog.sh, som starter alle nodene, leter fram den første FAT32-partisjonen som er montert (altså minnepinnen som er plugget i), og antar at alle nodene ligger i en mappe *ros-pkg* under denne. Inni ros-pkg vil det ligge en *launch*-fil som definerer alle nodene som skal startes automatisk. For eksempel, om minnepinnen er montert som */media/hdd* så vil filen */media/hdd/ros-pkg/launch* definere hvilke noder som skal startes.

Nodene bør legges i egne mapper under ros-pkg for å holde dem adskilt. Katalogstrukturen kan for eksempel se slik ut:

```
ros-pkg
|-- launch
|-- dummy
|   |-- dummy_gps
|   `-- dummy_imu
`-- rosfswebcam
    `-- rosfswebcam
```

For å fortsette eksempelet kan launch-filen inneholde:

```
rosfswebcam/rosfswebcam -c /etc/rosfswebcam.conf
dummy/dummy_gps
dummy/dummy_imu
```

watchdog.sh går gjennom launch linje for linje og kjører programmene som er definert på hver linje som egne prosesser. dummy_gps vil for eksempel kjøres med denne kommandoen:

```
/usr/bin/nohup dummy/dummy_gps &> /dev/null < /dev/null &
```

Vedlegg C

Videreutvikling av høstprosjektet

C.1 Introduksjon

Prosjektet til undertegnede høsten 2008 endte med en implementasjon av *The Optimal Triangulation Method* i C++ ved hjelp av biblioteket OpenCV. Denne algoritmen korrigerer for målefeil i to bilder. Det som ikke ble oppnådd i høstprosjektet var en euklidsk rekonstruksjon, det vil si at resultatene var kun riktig opp til en projektiv transformasjon.

Jeg vil rette en stor takk til Richard Hartley, forfatter av boken Multiple View Geometry som var hovedreferansen i prosjektoppgaven, for å ha hjulpet undertegnede med videre utvikling av prosjektet.

C.2 Euklidsk korrekt 3D-rekonstruksjon

Ved testing av funksjonen for *The Optimal Triangulation Method* fra høstprosjektet kom det til å begynne med ut merkelige resultater. Det ble bestemt at dette var på grunn av en uforutsett tredimensjonal projektiv transformasjon av rom-koordinatene. Når de romlige koordinatene er funnet vil de avvike fra de faktiske koordinatene med denne transformasjonen.

Ved å vite kameramatrisene på forhånd kan dette unngåes. I all hovedsak innebærer det å vite kameraets interne parametere slik som brennvidden, og eksterne parametere som hvor bildene ble tatt ifra. Hvis kameramatrisen P angir sammenhengen mellom et koordinat $\mathbf{X}$ i rommet og et koordinat i bildet $\mathbf{x}$ så er

$\mathbf{x} = P\mathbf{X} = KR[\ I\ |\ -\tilde{\mathbf{C}}\ ]\mathbf{X}$ der $K = \begin{bmatrix} fm_x & & x_0 \\ & fm_y & y_0 \\ & & 1 \end{bmatrix}$

K-matrisen inneholder her de interne parametrene. $\tilde{\mathbf{C}}$ angir kameraets posisjon i forhold til origo i "verdenskoordinater" mens R angir kameraets rotasjon, og tilsammen utgjør disse to kameraets eksterne parametere. f er brennvidden, m_x og m_y er antall piksler per

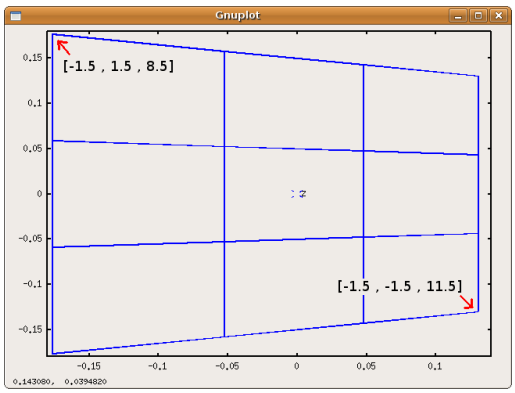
distanseenhet i x- og y-retning og x_0 og y_0 er bildets origo.

C.3 Testing og konklusjon

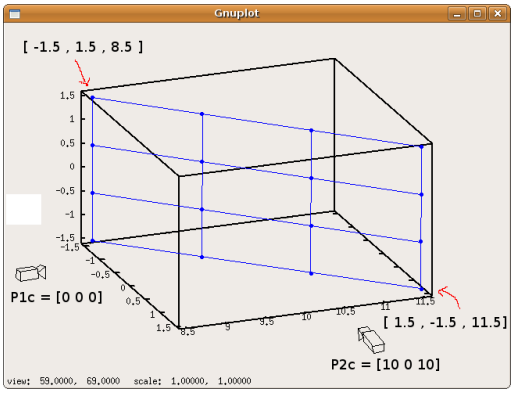
Hvis de interne parametrene er kjent, for eksempel ved hjelp av kalibrering, og de eksterne parametrene regnes ut basert på GPS og IMU, så er det mulig å oppnå en korrekt 3D-rekonstruksjon av scenen.

Og enda bedre, hvis tre bilder istedenfor to brukes i trianguleringsprosessen vil resultatet bedres drastisk, noe som anbefales til de som får lyst til å jobbe videre med denne form for 3D-rekonstruksjon.

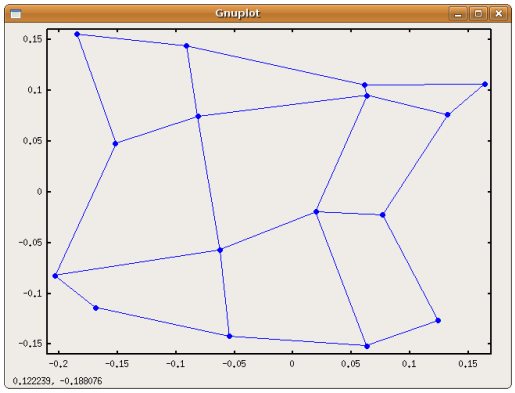
Implementasjonen av *The Optimal Triangulation Method* er nå å finne i OpenCV-biblioteket som funksjonen *cvCorrectCorrespondences*.



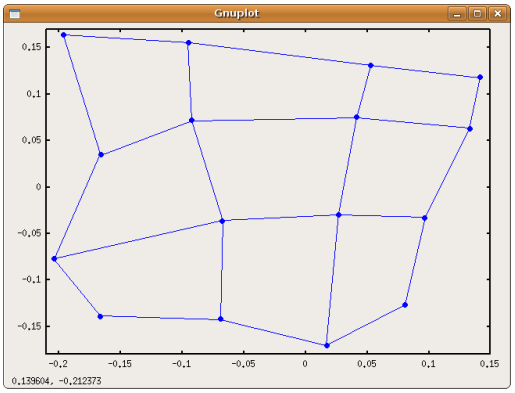
(a) Et rutenett av 16 punkter ble generert for å teste algoritmene. Her ser man rutenettet fra kamera #1 som er sentrert i origo.



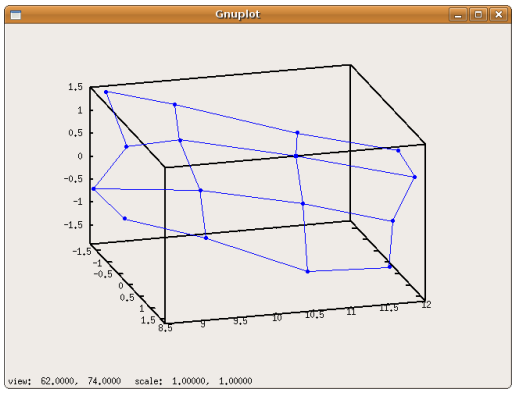
(b) Dette er de samme punktene som i (a), der kameraenes posisjon er indikert.



(c) Her er det introdusert betydelig støy i hvert av kameraene. Her vises kun rutenettet sett fra kamera #1.

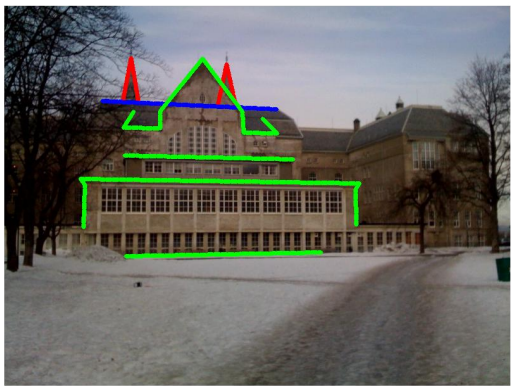


(d) Dette er resultatet etter at *Direct Linear Triangulation* og påfølgende *The Optimal Triangulation Method* er kjørt på målingene.

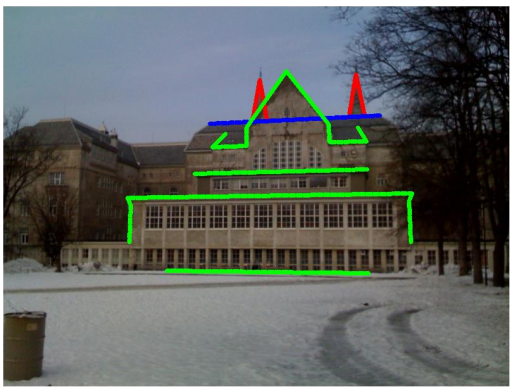


(e) Samme punkter som i (d) fra samme vinkel som i (b).

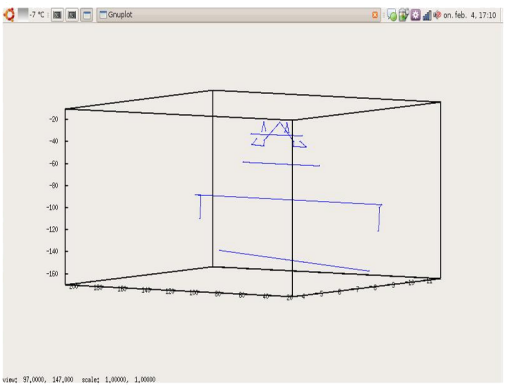
Figur C.1: Demonstrasjon av *The Optimal Triangulation Method*. I (a) og (b) vises de faktiske punktene. Når kamera #1 og #2 tar bilder vil det introduseres støy (vist i (c)), hovedsaklig på grunn av diskretiseringen og feil i modellen. Punktene blir deretter rekonstruert i 3D basert kun på bildene med målefeil. I (d) og (e) er det tydelig at *The Optimal Triangulation Method* har dempet støyen betydelig. I praksis vil det heller ikke være så mye støy i målingene som her.



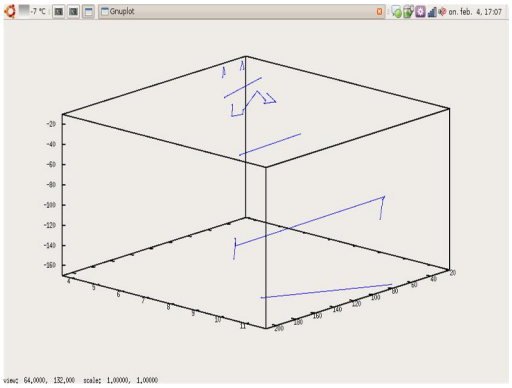
(a) Bilde av hovedbygget fra venstre med påtegnede features. Linjer er trukket mellom features for illustrasjonens skyld. De forskjellige fargene er kun for å skille de forskjellige linjene fra hverandre.



(b) Bilde av hovedbygget fra høyre med påtegnede features.

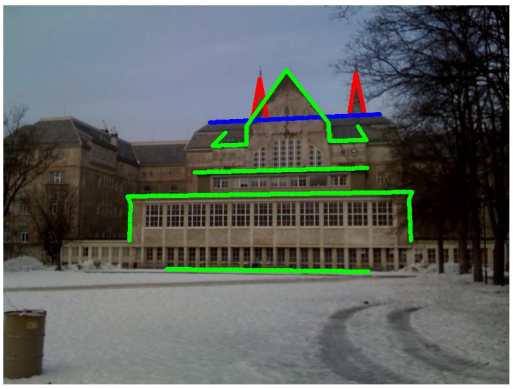


(c) 3D-rekonstruksjon av hovedbygget sett fra omtrent samme perspektiv som i (a).

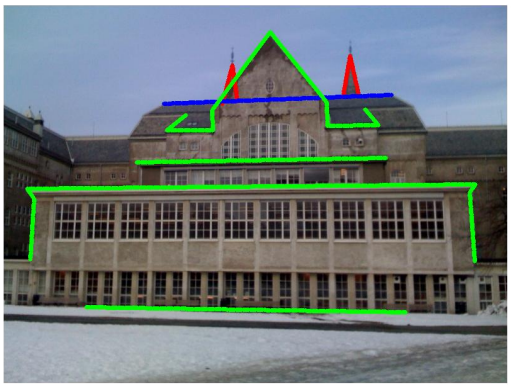


(d) 3D-rekonstruksjon av hovedbygget sett fra luften et sted over elektrobygget

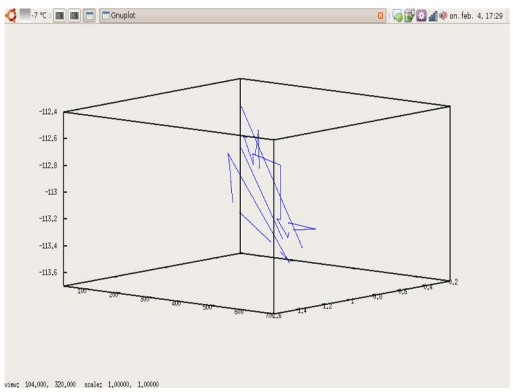
Figur C.2: Demonstrasjon av 3D-rekonstruksjon ved hjelp av *Direct Linear Triangulation* og *The Optimal Triangulation Method*. Bildene er tatt med god vinkel mellom seg og gir dermed et godt resultat. Feature-detektorene som ble testet i prosjektet var ikke gode nok så punktene her er tegnet inn manuelt. Kameramatrisene er regnet ut basert på kvalifisert gjetning av de interne og eksterne parametrene.



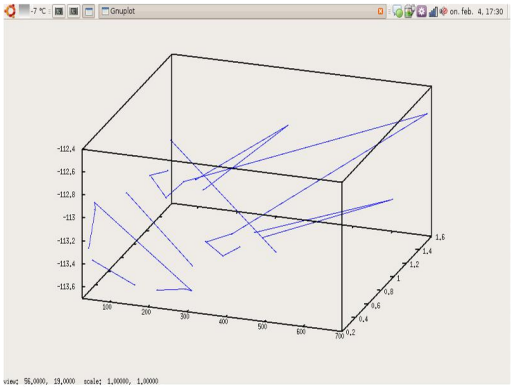
(a) Samme bildet som i C.2b.



(b) Bilde tatt nærmere hovedbygget enn i (a) men i omtrent samme retning.



(c) 3D-rekonstruksjon av hovedbygget forsøkt sett fra omtrent samme sted som i (a).



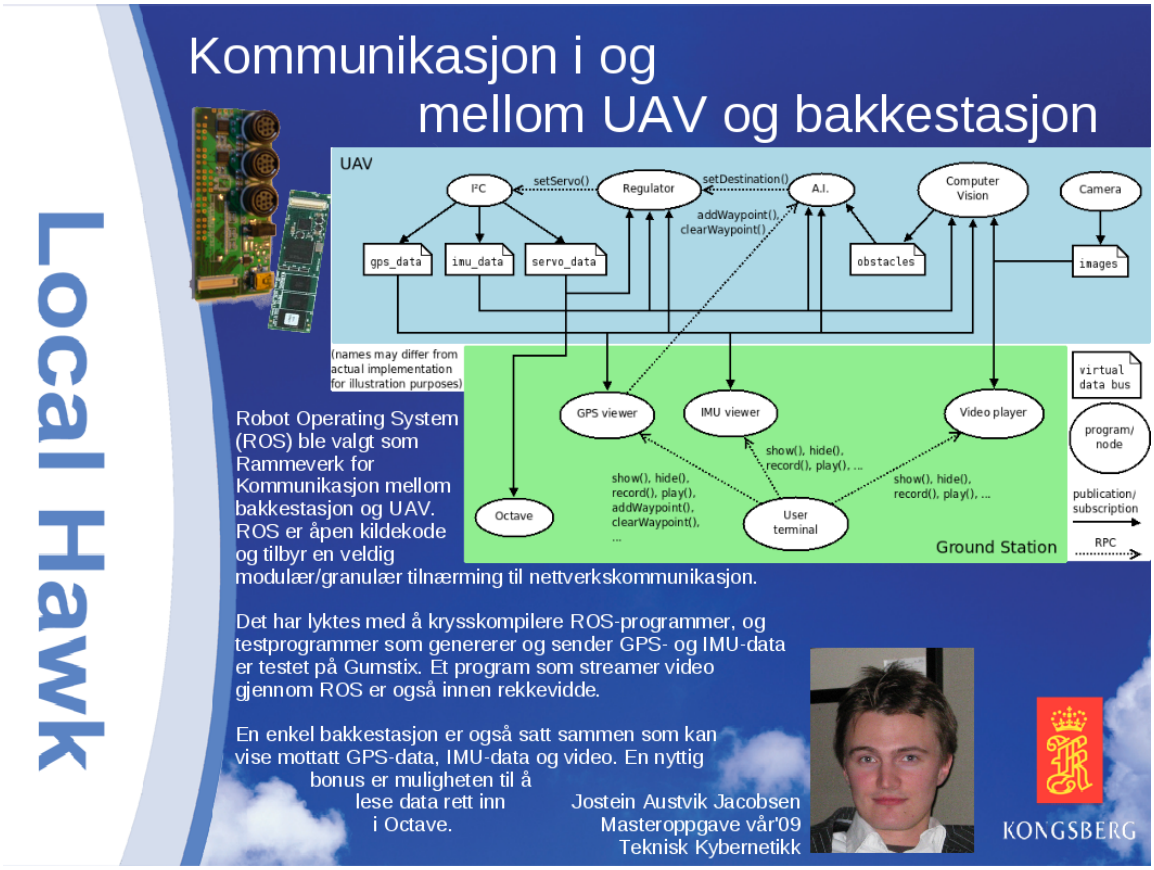
(d) 3D-rekonstruksjon av hovedbygget sett fra et sted over kjel-bygget.

Figur C.3: Her er det prøvd å gjøre det samme som i C.3 men med mye spissere vinkel mellom retningen de to bildene er tatt mot. Det er tydelig hvordan rekonstruksjonen blir dårligere for punkter der "trianguleringsvinkelen" er liten (kameramatrisene ble forsøkt justert uten hell).

Vedlegg D

Presentasjonsslides for høstprosjekt og masteroppgave

Figur D.1: Presentasjonsslide for høstprosjektet produsert for Kongsberg Defence & Aerospace.



Figur D.2: Presentasjonsslide for diplomoppgaven produsert for Kongsberg Defence & Aerospace.

Vedlegg E

Dokumentasjon av kommandoer for bakkestasjonen

```
jostein@pion:~/ros/ros-pkg/groundstation$ ./terminal
Ground Station version: 0.3.0
Terminal interface version: 1.0
 > help
available commands:
    help                          : this help listing
    connect                       : connect to a channel client
    exit, quit                    : exit program
    dump <channel> <file>         : dump channel data to a file
    dumpstop <channel>            : stop dumping channel data
                                                    to file
    play <channel> <file>         : playback a stored data file
    playstop <channel> <file>     : stop playback of data
    show <channel> <args>         : visualize channel
    hide <channel>                : close the visualization
    info                          : show general or specific info
                                                  about something
    run                           : run a script
    echo                          : print text to screen
    set, setting                  : set a setting on the channel

use 'help <command>' for details about a command
 > help help
help <command>

    Display list of available commands.
    If <command> is specified, shows a
    detailed explaination of that command.
```

```
 > help connect
connect <name>

    name : the name of the node

Connects to a node/channel so that other commands
can be used on the channel. When for instance
'show gps' is called, the ground station must
first be connected to the gps.

example usage:
    connect webcam
    connect gps
 > help exit
exit
quit

    Exits the program
 > help dump
dump <channel> <file>

    channel : optional data channel to record from
    file    : optional filename of the file to
              record data to

If no channel is specified, starts dumping all
channels with automatically generated filenames.
If only the channel is specified, the channel
defines its own way of storing the data, for
instance, image data are stored as a movie file,
while most numerical data like IMU and GPS data
are stored as comma separated values (CSV-format).
If the filename is not specified, a filename will
be automatically generated.
No files should be overwritten without your
approval.

example usage:
    dump imu
    dump camera atlantic_voyage_june2011.avi

see also: dumpstop
 > help dumpstop
```

```
dumpstop <channel> <file>

    channel : channel to stop recording from
    file    : file to stop recording to

Use this command to stop recording to a file.
If no channel is specified, will stop all
recordings. If no file is specified, will stop
all recordings being done on the specified
channel.

example usage:
    dumpstop
    dumpstop gps
    dumpstop imu imudata2009.csv

see also: dump
 > help play
play <channel> <file>

    channel : channel to play recording from
    file    : file to play recording from

Use this command to play back a recording from
a file. If no file is specified, the most recent
file is used. If no channel is specified either
then the latest recording from all channels are
played back. When multiple recordings are playing,
they are synchronized in time with respect to
when they were recorded. For instance, if GPS-
data was logged for ten minutes, and camera-
images was logged for the late five minutes of
those, then a playback will first consist of
five minutes of pure GPS-data, followed by a
synchronized playback of GPS-data along with video

example usage:
    play
    play gps
    play imu imudata2009.csv

see also: playstop
 > help playstop
```

```
playstop <channel> <file>

    channel : channel to stop playback from
    file    : file to stop playback from

Use this command to stop playback from a file.
If no file is specified, will stop playback
from all logs being played back from the
specified channel. If no channel is specified,
will stop all playback.

example usage:
    playstop
    playstop gps
    playstop imu imudata2009.csv

see also: play
 > help show
show <channel> <args>

    channel : the channel to visualize
    args    : channel specific visualization
              settings

Visualize a channel.

example usage:
    show webcam
    show gps

see also: hide
 > help hide
hide <channel>

    channel : the channel to stop visualizing

Stop visualizing the channel.

example usage:
    hide gps

see also: show
 > help info
```

```
info <channel>

    channel : the channel (optional)

Display info about the channel such as
a description, state and capabilities.
If no channel is specified, displays a
list of available channels as well as
connection status

see also: channels, channel
 > help run
run <file>

    file : filename of the sript to run

Lets you run a script. The script is simply a set
of commands that will be run sequentially. If a
line cannot be interpreted, the script will abort.
Whitespaces at the start of a line is ignored.
Lines beginning with a '#' is ignored.

example script:

# start logging immediately
dumpall
# start camera
show camera

example usage:
    run ~/start.gs
 > help echo
echo <text>

    text : the text to echo back

Prints some text to the screen. Useful in scripts.
 > help set
set <channel> <options>
setting <channel> <options>

    channel : channel to set setting for
    options : options to set
```

```
This will send updated parameters to a channel.
For instance, the webcam channel could enable
histogram equalization or the gps channel could
update its publish rate.

example usage:
    setting webcam histogramequalization on
    set imu opticalflowcorrection=off
```

Vedlegg F

DVD

På den vedlagte DVDen finnes all kildekode som er produsert sammen med nødvendige filer. Det er også noen videoer som demonstrerer resultatet.

Bibliografi

[1] FireStorm.cx - fswebcam. Nettside. http://www.firestorm.cx/fswebcam/.

[2] FlightGear Flight Simulator. Nettside. http://www.flightgear.org/.

[3] Hardware-in-the-loop simulation - Wikipedia, the free encyclopedia. Nettside. http://en.wikipedia.org/wiki/Hardware-in-the-loop_simulation.

[4] How-To: Managing services with update-rc.d. Nettside. http://www.debuntu.org/how-to-manage-services-with-update-rc.d.

[5] Infoweb: VPN på Linux. Nettside. http://infoweb.ntnu.no/nettverk/vpn/network-manager-vpnc.html%5C#i__137317704_407.

[6] Kongsberg Defence & Aerospace. Nettside. http://www.kongsberg.com/eng/KDA/.

[7] LinuxTVWiki. Nettside. http://www.linuxtv.org/wiki/index.php/Main_Page.

[8] Local Hawk Project. Nettside. http://www.localhawk.net/.

[9] Octave. Nettside. http://www.gnu.org/software/octave/.

[10] OpenCV - a library of programming functions mainly aimed at real time computer vision. Nettside. http://opencv.willowgarage.com/.

[11] OpenEmbedded. Nettside. http://www.openembedded.org/.

[12] ROS - Wiki. Nettside. http://pr.willowgarage.com/wiki/ROS.

[13] ROS (Robot Operating System) - Wikipedia, the free encyclopedia. Nettside. http://en.wikipedia.org/wiki/ROS_(Robot_Operating_System).

[14] ros-users mail archive: Convert JPG to ROS image format. Nettside. http://sourceforge.net/mailarchive/forum.php?thread_name=e31f09810904090028v1c7cd1e1x344c4a07d6567f64%40mail.gmail.com&forum_name=ros-users.

[15] ros-users mail archive: New package image_publisher. Nettside. http://sourceforge.net/mailarchive/forum.php?thread_name=62051ad90906101412x65abe456yef5026696df2e749%40mail.gmail.com&forum_name=ros-users.

[16] ros-users mail archive: Packages used for consumer cameras. Nettside. http://sourceforge.net/mailarchive/forum.php?thread_name=e31f09810906011700g162da708h3e67d7e8ad945015%40mail.gmail.com&forum_name=ros-users.

[17] ros-users mail archive: Problem with CvBridge - going from OpenCV image to ROS image. Nettside. http://sourceforge.net/mailarchive/forum.php?thread_name=b4428cfe0903161615o5fa301b5jdd4e5a3a1dd58a53%40mail.gmail.com&forum_name=ros-users.

[18] SourceForge.net: sail-ros-pkg. Nettside. http://sail-ros-pkg.sourceforge.net/.

[19] The Spread Toolkit. Nettside. http://www.spread.org/.

[20] videodog(1) - Linux man page. Nettside. http://linux.die.net/man/1/videodog.

[21] Odin Hammer Eliassen, Øyvind Risan, Bjørn Isak Håkonsen, Peder Wenstad Haug og Jon Øyvind Vold. LocalHawk. Fagrapport, 2009.

[22] R. I. Hartley og A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second utgave, 2004.

[23] Lars Ivar Miljeteig, Paal Alexander Nerholm, Sumit Mohindra og Vegar Heggen Hedenstad. Local Hawk. Hovedrapport, 2009.

Begrepsoversikt

A.I.	Artificial Intelligence. Norsk: kunstig intelligens.
Abonnent	Program som abonnerer på og mottar data fra emne (engelsk: subscriber).
Bakkestasjon	Viser en representasjon av tilgjengelig informasjon om flyet. Lar brukeren sende kommandoer til flyet.
CMake	Kryssplattformssystem for automatisk kompilering og lenking av programmer.
Console-VX	Utvidelseskort til Gumstix Verdex-hovedkortene som gir tilgang til blant annet RS-232 signaler, USB host og strømtilkobling.
cron / crontab	Gjør at Linux kan starte programmer ved spesifiserte tider.
CyberSwan	Local Hawk sin forgjenger, utviklet av masterstudenter ved NTNU.
Datasyn	Teknologi som gjør at maskiner kan se.
downlink	Forbindelsen fra UAV til bakkestasjon.
EiT	se: Eksperter i Team.
Eksperter i Team	Interdisiplinært kurs for alle masterstudenter på NTNU. To grupper med fem studenter jobbet med Local Hawk.
Emne	Et emne (engelsk: topic) er en virtuell databuss i ROS.

fswebcam	Et tekstbasert program for å ta bilder med webkamera og lagre bildene til disk. Tilsvarende videodog.
GPS	Global Positioning System. GPS-enheter bestemmer sin absolutte posisjon basert på målinger av en konstellasjon av GPS-satellitter.
Ground Station	Norsk: Bakkestasjon.
GumROS	ROS for Gumstix.
Gumstix	Gumstix Incorporated produserer Gumstix-hovedkort. Verdex XL6P, som ble brukt i dette prosjektet er et eksempel på et slikt.
Hardware-in-loop	Hvis flyet lures til å tro at det er ute og flyr, mens det egentlig er koblet til en flysimulator, så er det et eksempel på hardware-in-loop utvikling.
HiBu	Høgskolen i Buskerud.
HIL	se: Hardware-in-loop.
HUD	Heads-up-display. Fremvisning av informasjon som en del av kamerabildet eller brukergrensesnittet.
I^2C	Inter-Integrated Circuit. I^2C-bussprotokollen tilbyr en enkel måte å koble sammen innebygde systemer på.
IMU	Inertial Measurement Unit. Enhet med kombinert gyroskop, akselerometer, rotasjon og kompass.
IPC	Inter-Process Communication. Kommunikasjon mellom programmer på samme maskin.
KDA	Kongsberg Defence & Aerospace AS.
Krysskompilering	Kompilering for en annen maskin enn den det kompileres på.
Local Hawk	Navn på det overordnede flyprosjektet koordinert av KDA.

Master-node	Noden som gjør at andre noder finner hverandre.
Mobilt bredbånd	USB-modemene på utlån fra Telenor brukes for å koble til Internett fra Gumstixen.
Node	Når det snakkes om en node i ROS så er det i praksis snakk om et program.
Octave	Brukes for numerisk analyse og er et åpen kildekode-alternativ til Matlab.
OpenCV	Bibliotek for datasyn og bildebehandling.
Pakkebrønn	Oppbevaringssted for programpakker (engelsk: repository).
Publisher	Norsk: Utgiver.
ROS	Robot Operating System, eller alternativt Robot Open Source. Rammeverk/bibliotek for nettverkskommunikasjon i og mellom roboter.
rosfswecam	Versjonen av fswebcam som er omskrevet for ROS refereres til som rosfswebcam.
rosmake	Verktøy for ROS som håndterer avhengigheter når pakker kompileres.
ROSpod	ROS for iPod Touch og iPhone.
RPC	Remote Procedure Call. I ROS tilsvarer dette en tjeneste.
Service	Norsk: Tjeneste.
Servo	En servokontroller mottar et settpunkt og regulerer motorhastigheten deretter.
Spread	The Spread Toolkit er et bibliotek for høyytelses meldingstjenester.
Subscriber	Norsk: Abonnent.

TCP	Transmission Control Protocol. TCP er en tilkoblingsbasert (engelsk: connection-oriented) nettverksprotokoll.
Tjeneste	En tjeneste (engelsk: service) er ROS sin implementasjon av RPC.
Toolchain	Settet med verktøy som brukes, gjerne etter hverandre, for å lage et program.
Topic	Norsk: Emne.
Tutorial	En tutorial er en guide eller veiledning som går gjennom et oppsett av noe steg for steg.
UAV	Unmanned Aerial Vehicle. Norsk: Ubemannet luftfartøy.
Ubuntu	Linux distribusjon.
UDP	User Datagram Protocol. UDP er en tilkoblingsløs (engelsk: connectionless) nettverksprotokoll.
UDP broadcast	UDP-funksjonalitet som tillater nettverkspakker å bli sendt til alle i samme subnet samtidig.
UiA	Universitetet i Agder.
USB	Universal Serial Bus. Utbredt standard for sammenkobling av enheter.
Utgiver	Programmet som annonserer og sender avgårde data på et emne (engelsk: publisher).
Verdex XL6P	Et Gumstix-hovedkort med 600Mhz klokkefrekvens, 128MB RAM og 32MB flash minne.
videodog	Et tekstbasert program for å ta bilder med webkamera og lagre bildene til disk. Tilsvarende fswebcam.
Waypoint	Et sted som flyet er bedt om å kjøre innom.