Sensur av hovedoppgaver

Universitetet i Sørøst-Norge
Fakultet for teknologi og maritime fag

Prosjektnummer: **2020-01**
For studieåret: **2019/2020**
Emnekode: **SFHO3201-1 19H Bacheloroppgave**

Prosjektnavn
Aegir (Coastal Shark)

Utført i samarbeid med: Kongsberg Defence & Aerospace

Ekstern veiledere: Alexander Gosling, Børge Selbæk

Sammendrag: Gruppen har fått oppgave av KDA om videreutvikling av et av deres mangeårige prosjekt kalt Coastal Shark. Omfanget er å bygge videre på datakommunikasjonen. Her har gruppen sett på slikt som kontroll og oversikt av dataflyt, prioritering av meldinger og struping av nettverket.

Stikkord:
- datakommunikasjon
- prioritering
- kontroll

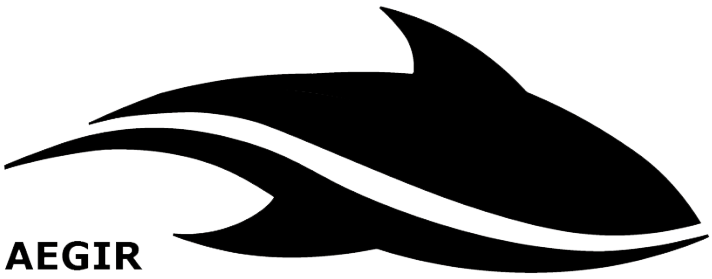
Tilgjengelig: JA

Prosjekt deltagere og karakter:

Navn	Karakter
Daniel Skryseth	
Jannicke Løkaas	
Tomas Endsjø Jacobsen	
Øzlem Tuzkaya	
Bente Hestnes	
Idar Carlsen	

Dato: 15. juni 2020

_____	_____	_____
Sigmund Gudvangen	Karoline Moholth	Alexander Gosling
Intern Veileder	Intern Sensor	Ekstern Sensor



AEGIR

COASTAL SHARK

25. mai 2020

KONGSBERG

Universitetet i Sørøst-Norge

Sammendrag

Denne rapporten ble skrevet av en bachelorgruppe med seks ingeniørstudenter fra data. Bachelorgruppen fikk et prosjekt fra Kongsberg Defence & Aerospace med problemstillingen rundt dataoverføring i et system over et ustabilt nettverk. Systemet er bestående av en landstasjon og et delvis autonomt fartøy. Fartøyet er ment å kunne kjøre i farvann hvor radioforbindelsen kan bli dårlig. Det ble derfor bachelorgruppens oppgave å finne ut hvordan man kan løse denne problemstillingen med tanke på at det skal sendes data mellom stasjonene også under dårlige forhold. Når det sendes mange meldinger kontinuerlig, vil ikke all data komme frem hvis bitraten er lav. I slike tilfeller vil det være nødvendig å prioritere de viktigste meldingene, slik at det er mindre viktige meldinger som går tapt fremfor essensielle data. Det vil i så måte være fornuftig å forkaste utdatert informasjon, slik at nødvendig informasjon kan komme frem til rett tid.

I forbindelse med meldingsoverføringer i systemet, vil det være en fordel med god oversikt over datakommunikasjonen, slik at det blir enklere å foreta vurderinger i forhold til prioriteringer av meldingene. Det var derfor gruppens oppgave å se på løsninger for å lage en slik oversikt. Gruppen fikk også oppgaven med å se på muligheten for å simulere nettverksytelsen i form av et verktøy som kunne strupe nettverket. Med dette verktøyet skal det være mulig å endre båndbredden for å teste at selve prioriteringen fungerer i henhold til intensjonen, og at de prioriterte meldingene blir sendt over nettverket slik det var tenkt.

I tillegg var det krav fra oppdragsgiver om å se på løsninger for å logge- og lagre dataene som sendes over nettverket, og innad i systemet. Loggingen tar for seg meldinger som sendes i begge retninger over nettverket. Det var også ønskelig at dataene som sendes over nettveket lagres, slik at de kan spilles av i en test senere.

Innholdsfortegnelse

Figurliste

Tabelliste

Forkortelser og akronymer

Forkortelse	Beskrivelse
KDA	Kongsberg Defence and Aerospace
USN	Universitet i Sørøst-Norge
Gruppen	Bachelorgruppen
C2	Kontrollstasjon på land
Aegir	Ubemannet overflatekjøretøy
Coastal Shark	Prosjektet som en helhet, inkluderer både C2 og Aegir
ROS	Robot Operating System

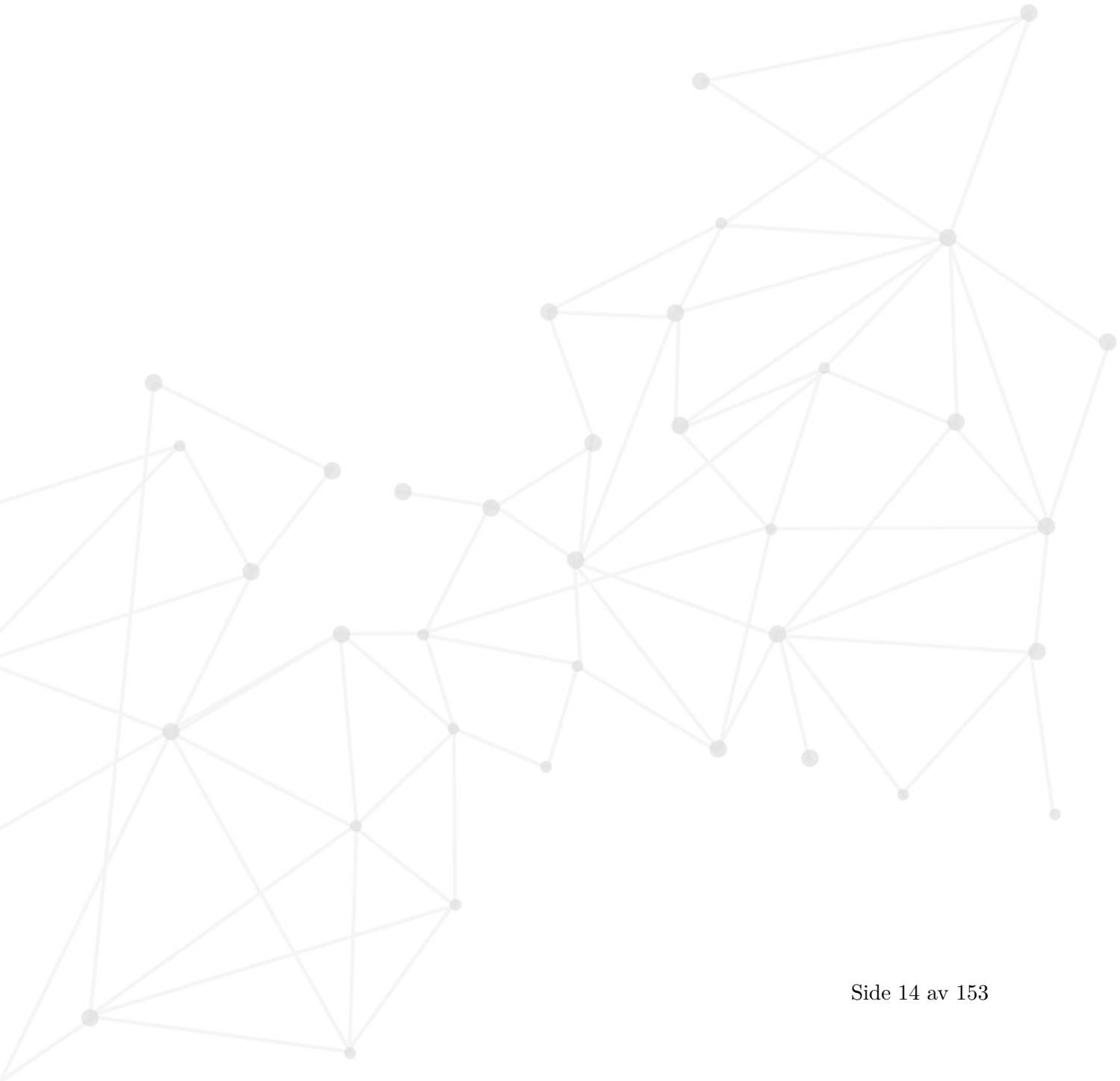
Revisjonstabell

Versjon	Kapnr.	Endringer	Dato
1.1	3.9	Delkapittel om prioritering av oppgaver ble lagt til.	2020-02-17
	8	Nytt kapittel om systemoversikt ble lagt til.	2020-03-10
	7	Ny introduksjon til kapittelet Krav- og testspesifikasjon lagt til	2020-03-20
	5	Nye figurer til risikotabell Rev. 1 lagt til	2020-03-20
	5	Ny introduksjon til kapittelet Risikostyring lagt til	2020-03-20
	1	Nytt kapittel om Revisjonstabell ble lagt til.	2020-03-22
	4	Delkapittel om Unit testing ble lagt til.	2020-03-22
	2	Delkapittel om Forkortelser og akronymer ble lagt til.	2020-03-23
	3	Ny introduksjon til kapittelet oppgavebeskrivelse lagt til	2020-03-23
	9	Nytt kapittel om Prioritering lagt til	2020-03-23
	10	Nytt kapittel om Datainnsamling og loggesystem lagt till	2020-03-23
	11	Nytt kapittel om Loggesystem lagt til	2020-03-23
	12	Nytt kapittel om Strupingsverktøy lagt til	2020-03-23
	6.8	Nytt delkapittel om Utfordringer lagt til i kapittel fremdriftsplan	2020-03-23
	4.6.1	Kanbantavle ble oppdatert med en ekstra kolonne	2020-03-24
	6.1	Ganttdiagrammet er byttet ut med et online-basert tidsestimeringsverktøy	2020-03-24
	Vedlegg A	Vedlegg om alternative løsningsforslag iht. prioritering lagt til	2020-03-25
	Vedlegg C	Vedlegg Gantt-diagram	2020-03-25

Versjon	Kapnr.	Endringer	Dato
2.1	Innhold	Omstrukturerte rekkefølgen på kapitler	2020-05-15
	Tabelliste	Lagt til Tabelliste	2020-05-15
	Alle	Lagt til forside med introduksjon	2020-05-15
	1	Slått sammen kapittel 2 og 3 til kapittel 1	2020-05-15
	2	Slått sammen kapitel 4,5,6 til kapittel 2	2020-05-15
	4	Oppdatert krav- og testspesifikasjon	2020-05-15
	6	Kapittelet om prioriteringssystemet har blitt skrevet om.	2020-05-15
	7	Lagt til delkapittel 7.1 Problemstilling og 7.4 Testing	2020-05-15
	7.2	Lagt til figur 7.1 - 7.3	2020-05-15
	7.4	Lagt til tabell 7.1	2020-05-15
	8.3	Lagt til delkapittel 8.3.1 Datainnsamling på Aegir og 8.3.2 Datainnsamling på C2	2020-05-15
	8.4	Lagt til delkapittel 8.4.1 Loggesystemet for sendt og mottatt på Aegir, 8.4.2 Design	2020-05-15
	10	Lagt til kapittel 5 Dynamisk oversikt	2020-05-15
	1	Lagt til ny revisjonstabell.	2020-05-18
	9	Lagt til kapittel 9 Konklusjon	2020-05-18
	Sammendrag	Lagt til sammendrag før innhold	2020-05-18
	Vedlegg B	Vedlegg om alternative løsningsforslag iht. dataoversikt	2020-05-19
	1.1	Nytt delkapittel om Lisenser lagt til i kapittel Introduksjon	2020-05-19
	Vedlegg D	Vedlegg lisenser lagt til	2020-05-19
	Vedlegg E	Vedlegg økonomi lagt til	2020-05-22
	8.3	Lagt til figur 8.1 - 8.10	2020-05-22
	5	Lagt til figur 5.2 - 5.4	2020-05-23

1. Introduksjon

Dette kapittelet vil gi en introduksjon til dokumentstrukturen, gruppemedlemmene og prosjektoppgaven.



1.1 Lisenser

Lisenser brukt til utviklingen av prosjektet er lagt med som vedlegg D.

1.2 Bakgrunn til prosjektet

Bachelorgruppen bestod av seks datastudenter ved Universitetet i Sørøst-Norge, campus Kongsberg. Fire av medlemmene hadde studieretning innen *embedded systems*, mens to av medlemmene hadde *virtual systems*.

I jakten på et interessant bachelorprosjekt kom gruppen i kontakt med Kongsberg Defence & Aerospace (KDA). De var positivt innstilt og ville gjerne jobbe med gruppen, og foreslo en oppgave som var en videreutvikling av deres mangeårige sommerprosjekt som startet i 2015.

1.3 Gruppemedlemmene

Jannicke Løkaas
Utvikler og gruppeleder

Idar Carlsen
Hovedutvikler

Daniel Skryseth
Hovedutvikler

Bente Hestnes
Utvikler og
dokumentansvarlig

Tomas Endsjø Jacobsen
Utvikler og
hardwareansvarlig

Øzlem Tuzkaya
Utvikler og
økonomiansvarlig

1.4 Rolletitler

Gruppemedlemmene tok på seg forskjellige rolletitler, og disse rollene krevde et visst ansvar gjennom prosjektet. Rollene er beskrevet nedenfor.

1.4.1 Gruppeleder

Gruppelederen skulle i all hovedsak ta seg av administrativt arbeid. Dette innebar å være et kontaktledd mellom gruppa og eksterne personer. Gruppa var for øvrig demokratisk, slik at gruppelederen ikke var en leder i den forstand at gruppeleder skulle ta avgjørelser på vegne av gruppa. Det var gruppeleders ansvar å ta opp eventuelle interne problemer og utfordringer gruppa møtte på underveis.

1.4.2 Hovedutvikler

Hovedutviklerne skulle bidra til å finne relevante oppgaver som alle utviklerne kunne arbeide med. Det var også hovedutviklernes oppgave å hjelpe til å få en dypere forståelse av systemet og oppbyggingen av kildekoden. På denne måten trengte ikke alle gruppemedlemmene å bruke like mye tid på slike undersøkelser hver for seg, hvorav gruppen sparte tid på å få en felles gjennomgang av dette materialet.

1.4.3 Utvikler

En utvikler skulle være med på å utvikle det tekniske resultatet som presenteres for oppdragsgiver. Denne rollen innebar at man måtte sette seg inn i utviklingssystemene som ble brukt, og bidra til kildekoden som førte frem til sluttresultatet.

1.4.4 Dokumentansvarlig

Den som hadde dokumentansvar skulle, så langt det lot seg gjøre, passe på at all dokumentasjon var ferdig utfylt til gitt tid og levert til rett(e) person(er) innen fristen. Det var likevel ikke dokumentansvarlig som selv skulle sende dokumentasjon, ettersom gruppeleder skulle ta på seg slike oppgaver. Men det var dokumentansvarlig sitt ansvar å minne om forefallende dokumentasjonsarbeid og tidsfrister med hensyn til dette.

1.4.5 Hardewareansvarlig

Hardwareansvarlig hadde ansvar for å holde orden på, fikse og sette opp hardware. Ved eventuelle hardwarefeil som oppstod underveis var det hardwareansvarlig sin oppgave å prøve å hjelpe og rette dette.

1.4.6 Økonomiansvarlig

Økonomiansvarlig skulle føre regnskap og sørge for at alle kvitteringer ble håndtert og ført opp på rett sted. Regnskapet til gruppen er lagt med i vedlegg E.

1.5 Oppgavebeskrivelse

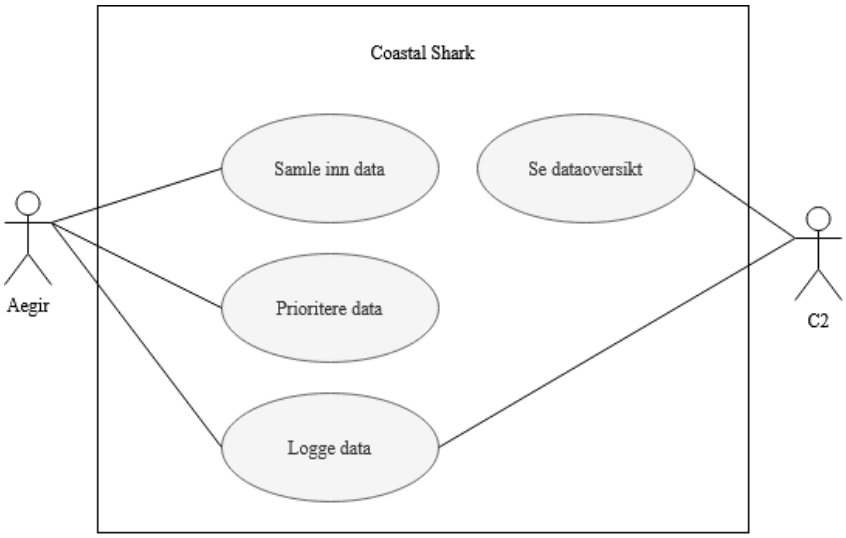
Gruppens oppdragsgiver i dette bachelorprosjektet var KDA. Bachelorgruppen ble tildelt en oppgave av KDA for videreutvikling av et av deres systemer. Dette systemet heter Coastal Shark og har vært et tverrfaglig sommerprosjekt mellom data-, kybernetikk-, elektro- og mekanikkstudenter siden 2015. Siden dette prosjektet har pågått over flere år er det allerede mye arbeid og dokumentasjon fra forskjellige prosjektgrupper. Det var stor variasjon i hvor godt prosjektet var dokumentert underveis. Det vil si at på enkelte områder var informasjonen mangelfull.

Prosjektet Coastal Shark består av to hovedkomponenter, C2 og Aegir. Aegir er en Sea Doo Spark 3-up vannskuter som har blitt modifisert for å operere autonomt ved hjelp av en rekke sensorer. Aegir er delvis autonom med målet om å bli et selvregulerende overflatefartøy. Den kan bli tildelt en posisjon og utifra denne finne en rute til den gitte posisjonen. C2 er en kontrollstasjon plassert på land som brukes til å kontrollere og overvåke Aegir. Disse komponentene kommuniserer gjennom en trådløs forbindelse via radio.

Formålet med Coastal Shark er å kunne videreføre konseptet med et autonomt vannfartøy til båter. Båter med denne funksjonaliteten vil kunne bli brukt i oppdrag som minesøking og redningsoperasjoner, for å unngå risiko for tap av menneskeliv.

I det nåværende systemet er det ingen oversikt eller kontroll på dataflyt mellom Aegir og C2. Det har til nå blitt regnet med at nettverksytelsen har vært den samme på dataoverføringene til enhver tid. Ettersom det er et realistisk scenario at denne ytelsen vil endres, har gruppens hovedoppgave vært å kartlegge datakommunikasjonen og prioritere dataoverføringene ut fra nettverksytelsen. I tillegg til dette var det ønskelig å ta frem en dynamisk dataoversikt, samt implementere et logging- og datainnsamlingssystem, slik at det skal være mulig å ta opp og spille av testsett. Dette vil hjelpe med fremtidig utvikling av prosjektet, samt muligheten for å teste og justere prioritering.

For å illustrere en oversikt over de tre funksjonene i prosjektet og se hvordan systemet samhandler, laget gruppen en *use case* diagram som kan bli sett fra figur 1.1.



Figur 1.1: *Use case* diagram for Coastal Shark.

Scenario for *use case* dataoversikt

Det skal være en dynamisk dataoversikt som gjør det mulig fra C2 å se på datakommunikasjonen i sanntid.

Scenario for *use case* logging

Logging skjer på to steder, både på Aegir og på C2. Alle meldinger som går over radiolinken blir logget.

Scenario for *use case* datainnsamling

Datainnsamlingssystemet ser etter meldinger som blir sendt, og skriver innholdet fra meldingene til en fil. Denne filen kan brukes til testsett.

Scenario for *use case* prioritering

Prioriteringsystemet ble gitt muligheten til å prioritere hvilke meldinger som sendes, bestemt ut fra nettverksytelsen.

2. Prosjektstyring

I dette kapittelet vil gruppens arbeidsverktøy og arbeidsmetode bli forklart. Kapittelet vil også ta for seg håndtering av risiko rundt prosjektet.

Vedlegg C tilhører dette kapittelet.

2.1 Arbeidsverktøy

Som en bakgrunn til prosjektet vil dette kapittelet gå gjennom diverse arbeidsverktøy og utstyr som er blitt brukt av bachelorgruppen. Noen endringer ble gjort rundt bruken av verktøyene grunnet covid-19, disse endringene blir beskrevet i 2.3.8

Fra KDA ble det gitt ikke-funksjonelle krav og begrensninger assosiert med programvare, teknikker, utstyr og bruk av utstyr tilknyttet prosjektet. Dette er noe som blir gjennomgått i dette kapitellet og kapitellet om krav- og testspesifikasjon.

2.1.1 Utstyr

KDA har utstedt alt av utstyr for bachelorprosjektet. Dette inkluderer laptop og stasjonære PCer, Git server, skjermer, svitsjer, flere tastaturer og datamuser, en rekke kabler, en LIDAR sensor og et Jetson kort [1]. Utstyret skulle til enhver tid være stasjonert på gruppen sitt grupperom under hele prosjektet.

Maskinene bruker Ubuntu operativsystem, som er en Linux-distribusjon. Ubuntu var gruppens primære plattform. Sammen var maskinene og lokal Git server koblet opp mot en svitsj for å danne et internt nettverk. Dette nettverket ble brukt til all utvikling av kildekoden og skulle ikke kobles opp mot noe annet nettverk.

I tillegg til dette ble det satt opp en egen Cisco-svitsj i nettverket. Denne skulle bli brukt til kontrollering og simuleringen av nettverket mellom landstasjonen og Aegir.

2.1.2 Git

Programvaren Git [2] ble brukt som versjonskontroll av kildekoden. Gjennom Git ville hvert gruppemedlem kunne jobbe med sin egen *branch* av kildekoden som blir skrevet for prosjektet. Dette gjorde det mulig for hvert enkelt medlem å jobbe uav-

hengig av hverandre helt til det blir gjort en sammenslåing inn mot *master*.

Med andre ord, når et gruppemedlem hadde sin egen *branch* og jobbet med kode, ville man trygt kunne endre på kildekoden uten å skape konflikter med noen andres arbeid. I tillegg ville man kunne gjøre et *commit* av bidraget og dermed lagre forskjellige stadier gjennom et snapshot av prosjektet.

2.1.3 Doxygen

Doxygen [3] er en dokumentasjonsgenerator som ble brukt for dokumentasjon av kildekode. Det praktiske med dette verktøyet var at det ble skrevet direkte inn i kildekoden, noe som gjorde det enkelt å vedlikeholde dokumentasjonen.

Med dette verktøyet var det mulig for leseren å kryssereferere mellom de forskjellige delene av kildekoden. Doxygen har nemlig muligheten til å produsere et HTML og CSS format på dokumentasjonen.

2.1.4 Qt

I prosjektet brukte gruppen Qt til å modifisere det grafiske brukergrensesnittet (*GUI*) til systemet. Dette er fordi systemet allerede hadde et *GUI* implementert som var basert på Qt.

Qt er et rammeverk skrevet i C++ og gir nye egenskaper som egner seg til utvikling av *GUI*. For utvikling med Qt brukte gruppen Qt Creator som er Qt sin egen *Integrated Development Environment* (*IDE*). Denne *IDE*-en har en integrert funksjonalitet til visualisering av *GUI* designet som gjør det lettere å gjøre og evaluere endringer for gruppen som ikke hadde tidligere erfaring med Qt [4].

2.1.5 Kommunikasjonskanaler

I prosjektet valgte gruppen å bruke Messenger [5] og Discord [6] for intern kommunikasjon. Discord var hovedkanalen, mens beskjeder som ikke var direkte knyttet til prosjektet ble gitt gjennom Messenger. Kontakt med KDA og intern veileder skjedde via e-post og Skype.

2.2 Arbeidsmetode

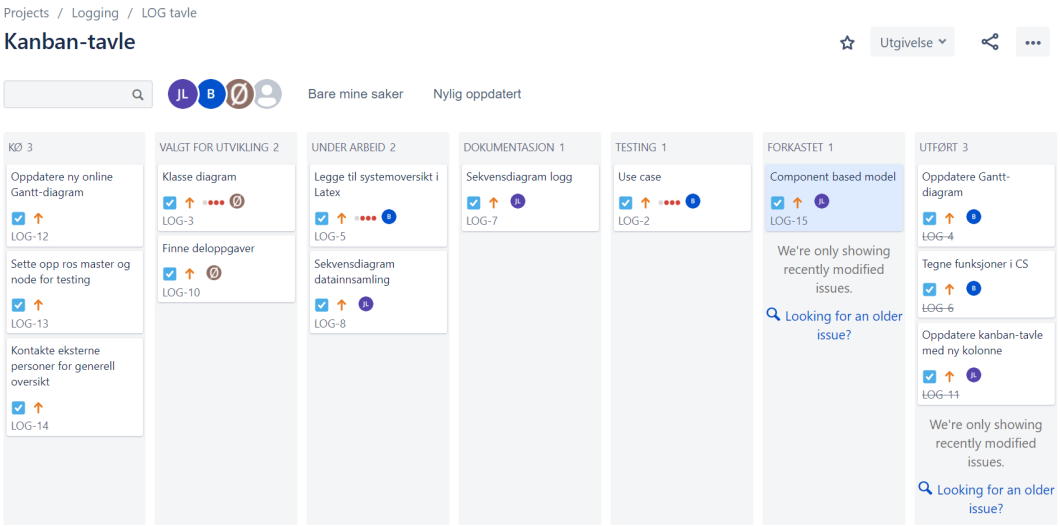
Her følger en beskrivelse av hvilke arbeidsmetoder som ble brukt for å få god arbeidsflyt i prosjektet.

2.2.1 Agil prosjektmodell

Gruppen valgte en Agil arbeidsmetode med utgangspunkt i Kanban [7]. Kanban visualiserer oppgavene ved hjelp av et tavlesystem med små kort, vanligvis en oppgave per kort [7]. Kanban ga gruppen stor frihet til å jobbe selvstendig og det ga en god arbeidsflyt gjennom hele prosjektet ved at hvert enkelt gruppemedlem kontinuerlig kunne tilegne seg oppgaver selv. Denne personen var da ansvarlig for at oppgaven blir løst på en tilfredsstillende måte. For å unngå flaskehals og ineffektivitet i prosjektet hadde gruppen valgt å sette en maksgrense på to oppgaver under utførelse per person. I tillegg til den rene Kanban rammen valgte gruppen å legge til daglige "god-morgenmøter", samt evalueringsmøter hver 14. dag.

2.2.2 Kanban-tavle

Gruppen valgte å bruke en online basert Kanban-tavle kalt Jira fra Atlassian Corporation. Dette er et gratis verktøy og Atlassian tilbyr også en applikasjon for mobilen slik at gruppen kunne holde seg oppdatert til enhver tid. Utover i prosjektet når gruppen delte oppgaven i flere delsystemer, ble det også besluttet å bruke en egen kanban-tavle for hver deloppgave. Et eksempel på en av gruppens kanban-tavler vises i figur 2.1.



Figur 2.1: En illustrasjon av gruppens Kanban-tavle.

Backlog

Backlog består av en prioritert liste med oppgaver som ble utarbeidet fra kravspesifikasjonen. Dette var alle typer oppgaver som skulle utføres for å få et vellykket resultat av prosjektet, både tekniske og ikke-tekniske oppgaver. Fra denne listen plukket hvert enkelt gruppemedlem oppgaver som vedkommende ville stå ansvarlig for. Oppgavene ble plukket fritt, ut fra kapasitet, ønske og kunnskap.

Valgt for utvikling

Når gruppemedlemmer hadde tilegnet seg en oppgave, ville denne oppgaven flyttes til listen over oppgaver valgt for utvikling. Her ville oppgavene ligge og vente til gruppemedlemmet hadde ledig kapasitet til å jobbe med oppgaven, og dermed kunne flytte den til listen med oppgaver under arbeid.

Under arbeid

Alle oppgaver det jobbes med ble lagt i denne listen. En Kanban arbeidsmetode har alltid en grense for maks antall tillatte oppgaver under utførelse [8]. Gruppen hadde valgt maks to oppgaver per person. Det betød at hvis en person jobbet med to oppgaver parallelt ville han eller hun ikke ha mulighet til å flytte en ny oppgave inn i listen.

Dokumentasjon

Alle oppgaver gruppen jobbet med, skal dokumenteres. For å sikre dette, hadde gruppen laget en egen kolonne for dokumentasjon. Det betyr at ingen oppgaver kunne testes eller forkastes uten at det er innom dokumentasjonskolonnen.

Testing

Før oppgaven ble ansett som ferdig utført, ville det i de fleste tilfeller kreves en form for testing eller klarering. Det kunne for eksempel være en gjennomgang og sjekk av dokumentasjon eller kode, eller mer tekniske oppgaver som for eksempel simulering.

Forkastet

Oppgaver som underveis i prosjektet ble forkastet havner i denne listen. Årsaken kunne være at gruppen fant andre og bedre løsninger, eller at oppgaven ikke lenger ble ansett som nødvendig for et vellykket sluttresultat.

Utført

I denne listen ligger alle utførte oppgaver gjennom hele prosjektet.

2.2.3 God-morgenmøter

Hver morgen ble det utført et 15 minutters stående møte der gruppemedlemmene planlegger dagen sin. Under møtet var det tre spørsmål som stod sentralt; hva man gjorde i går, hva man skal gjøre i dag, og om eventuelle utfordringer står i veien for videre arbeid. Møtet ble holdt uansett om gruppemedlemmer var fraværende eller ikke, og var ment for å oppdatere alle gruppens medlemmer på hva hver enkelt jobbet med og hva status var for øyeblikket. Om gruppemedlemmer trengte hjelp for videre framgang i arbeidet, skjedde dette etter møtet.

2.2.4 Evalueringsmøter

I tillegg til de daglige møtene ble det også holdt møter hver 14.dag. Disse møtene var ment for å gå dypere inn i arbeidsprosessen til gruppen som helhet, og for å evaluere hva som har gått bra og mindre bra i perioden som var gått, og hva som kunne gjøres bedre for neste periode. Evalueringsmøtet ga også en status i forhold til tidsrammen. Det ble tatt en avgjørelse av gruppen om å ta interne møter som en erstatning for evalueringsmøter. Disse interne møtene ble tatt ved behov og ble

avtalt gjennom morgenmøtene.

2.2.5 Prioritering av oppgaver

Siden Kanban ikke er avgrenset av tidsintevaller benyttet gruppen Gantt diagrammet for å visualisere avhengigheter og rekkefølgen av oppgavene. Diagrammet ble en estimering som inkluderer milepæler, aktiviteter og leveranser.

2.2.6 Unit testing

Som en del av testingen av systemet ble det brukt *unit tests*. Her ble det brukt en *divide-and-conquer* taktikk der hver komponent i kodebasen deles opp i mindre, individuelle deler som testes hver for seg [9]. Bruken av *unit tests* gjorde det enklere å verifisere at funksjonaliteten til systemet stemmer med forventningene, og gjør det enklere å vedlikeholde og ta høyde for fremtidige endringer av koden.

Testen skjer gjennom automatiserte oppgaver, og resultatet sammenlignes med en kjent eller forventet verdi. Sammenligningen skjer gjennom *assert* og *assert_eq* predikater [10]. Dersom en test mislykkes, vil programmet kaste en *error* og avslutte.

Hver del av prosjektet hadde sine egne *unit tests* som ble skrevet fortløpende under utviklingen. Det kreves at hver komponent har *unit tests* som dekker mesteparten av funksjonaliteten for at koden blir godkjent under *code review*.

2.3 Fremdriftsplan

For å planlegge prosjektet og være sikker på at gruppen var innenfor tidsfrister og for å få oversikt over både deloppgaver og hele prosjektet, var det viktig med en god tidsestimering. En slik oversikt var til nytte for både gruppen og oppdragsgiver for

å kunne følge med på fremdriften i prosjeketet.

2.3.1 Gantt-diagram

Det ble bestemt å bruke et Gantt-diagram for et grovt tidsestimat av oppgavene. Videre ut i prosjektet ble det behov for å bryte oppgavene ned i mindre deloppgaver og få et mer nøyaktig tidsestimat. Det onlinebaserte Gantt-diagrammet Instagantt fra Asana ble da et nyttig verktøy. Her valgte gruppen å dele prosjektet inn i delprosjekter slik arbeidsoppgavene ble fordelt innad i gruppen. Gantt-diagrammet ligger i vedlegg C.

2.3.2 Dokumentasjon

All dokumentasjon ble skrevet fortløpende under hele prosjektet. Relevant dokumentasjon ble samlet og ført inn i en sluttrapport som leveres til sensorene til en gitt dato mot slutten av prosjektet. Underveis ble det samlet aktuell dokumentasjon og innlevert før delpresentasjonene.

2.3.3 Undersøkelser

Det gikk med mye tid på undersøkelser for å bli kjent med nødvendige verktøy og systemer, som gruppen trengte for å få kunnskapen og forståelsen av oppgaven. Gruppen trengte å sette seg inn i følgende:

- Hvordan ROS er bygget opp.
- Brukergrensesnittet Qt.
- Arbeidsverktøyet Jira, som brukes til Kanban-tavle.

- Kildekoden til systemet Coastal Shark.
- Kravene fra oppdragsgiver og spesifisere disse.
- Kontrollversjon av kildekode med Git
- Dokumentasjon av kildekode med Doxygen

2.3.4 Planlegging

For å få prosjektet til å flyte mest mulig gjennom hele semesteret, var det viktig å planlegge fremdriften godt. Gruppen hadde i så måte evalueringsmøte hver 14. dag for å kartlegge status og veien videre. I tillegg ville det være et kort møte hver morgen som kan hjelpe oss til å holde oversikt fra dag til dag.

2.3.5 Prosjektutvikling

Etter hvert som gruppen forsto oppgaven og systemet det er bygget på, var prosjektet i en utvikling- og implementeringsfase.

2.3.6 Sluttrapport og sluttprodukt

Sluttrapporten skulle ferdigstilles over en fire ukers periode mot innspurten av prosjektet. Det endelige resultatet ville først bli vist på stand under Expo på Krona, og presentert i den siste presentasjonen som ville finne sted medio juni.

Grunnet covid-19 ble ikke Expo gjennomført som planlagt, men ble derimot en digital fremvisning.

2.3.7 Milepæler

Underveis var det noen milepæler å merke seg. Det var fire store milepæler som fant sted utover i semesteret. Disse var som følger:

- 1. presentasjon tidlig i februar.
- 2. presentasjon i løpet av mars.
- Expo utstillingen i slutten av mai.
- 3. presentasjon i begynnelsen av juni.

I tillegg var det små milepæler som forekom hyppigere. Hvert evalueringsmøte skulle være en liten milepæl, og hvert delmål underveis ville være små milepæler. For eksempel ville første test som viste at et krav ble oppfylt være en milepæl.

2.3.8 Utfordringer

Bachelorgruppen fikk noen nye utfordringer pga. covid-19 viruset. Gruppen jobbet på KDA sine stasjonære PC-er, disse maskinene og alt av utstyr var opprinnelig ment å stå innelåst på bachelor-rommet på universitetet. PC-ene var koblet i et lokalt nettverk med hverandre, men det var et krav fra KDA at disse ikke skulle kobles på Internett. Da universitetet stengte pga. covid-19 viruset, fikk gruppen likevel lov til å ta PC-ene med hjem og jobbe fra hjemmekontor.

Men det var fremdeles viktig at de ikke ble koblet til Internett. Gruppen ble nå avskåret fra hverandre og måtte finne nye former for kommunikasjon. Løsningen på dette ble å benytte private PC'er til tale-kommunikasjon over Discord. Likevel var det en ekstra utfordring når den nære kommunikasjonen og samværet ble borte. Det ble også vanskeligere å få ekstern hjelp.

Siden PC-ene heller ikke lenger var i et lokalt nett med hverandre var også kodegjennomgang og godkjenning et problem. Noe av denne utfordringen ble løst ved at KDA hadde gitt gruppemedlemmene tillatelse til å opprette private Git *repositories*, hvor kode som utvikles kan lastes opp og deles med hverandre. Siden noen av gruppemedlemmene hadde barn som var hjemme fra barnehage og skole grunnet den spesielle situasjonen samfunnet var oppe i, ble arbeidsdagen ekstra krevende og ikke like effektiv som ellers.

Møter med veiledere og KDA, som tidligere ble holdt på bachelor-rommet på universitetet, ble nå holdt via Skype. For å løse noe av utfordringen med at den nære kontakten var borte, ble det bestemt å holde disse møtene hyppigere.

På tross av alle utfordringer og mulige forsinkelser og nedprioriteringer av enkelte oppgaver, så gruppen likevel på dette som god læring.

2.4 Risikostyring

Risiko er noe som kan ha en negativ innvirkning på systemet. Eksempler på dette kan være en feil som har oppstått gjennom en dårlig avgjørelse gjort av et gruppemedlem, feil på teknisk utsyr eller svikt av en type programvare.

For bachelorgruppen inkluderer risikoer en rekke scenarioer som kan gi en hindring for utviklingen av prosjektet. Dette inkluderer blant annet aktiviteter, situasjoner og tiltak som er knyttet til prosjektet, i tillegg til den tidligere versjonen av systemet til Coastal Shark. Det er derfor viktig å både identifisere og vurdere risikoer utifra disse faktorene.

Med en risikostyringsprosess har bachelorgruppen benyttet seg av en analyse av mulige risikoer. Følger av dette er også håndtering og overvåkning av fremdriften i prosjektet, samt benyttet seg av risikoreduserende tiltak. På denne måten er risikoer redusert til et akseptabelt nivå, hvor formålet var å få et sluttresultat som ansees

vellykket.

Risikostyringsprosessen er splittet opp i fem faser og inkluderer:

- Risikoplanlegging
- Risikoidentifikasjon
- Risikoanalyse
- Risikohåndtering
- Risikoovervåking

2.4.1 Risikoplanlegging

Risikoplanlegging inneholder en strategi for å identifisere, analysere, håndtere og overvåke risikoer i prosjektet. Dette er listet i risikostyringsprosess.

2.4.2 Risikoidentifikasjon

Prosjektrisiko har blitt identifisert med kategorier for at hver enkelt skal forstå årsaken til risikoen og dens betydning. Tre kategorier er blitt brukt for å identifisere risiko:

- Maskinvarefeil (risiko forårsaket av maskinvare)
- Programvarefeil (risiko forårsaket av programvare)
- Menneskelige feil (risiko forårsaket av prosjektmedlemmer og veiledere)

2.4.3 Maskinvarerisiko

Gruppen har tenkt over mulige maskinvarefeil som er viktig å jobbe med gjennom prosjektet og hvordan det kan påvirke prosjektmedlemmene, se tabell 2.2. Det er

også skrevet forebyggende tiltak for å redusere påvirkningen av risikoen eller hindre det.

2.4.4 Programvarerisiko

Programvarefeil er risiko som kan oppstå på programvare, som f.eks. program kræsj. Dette har blitt vurdert og gruppen har kommet med forebyggende tiltak, se tabell 2.3.

2.4.5 Menneskelig risiko

Menneskelige feil er ting som kan oppstå ubevisst, noe gruppen ikke kan unngå. Det kan også være at oppdragsgiver kommer med nye krav. Gruppen har forebyggende tiltak som kan hjelpe oss gjennom prosjektet, se tabell 2.4.

2.4.6 Risikoanalyse

Prosessen med å evaluere hver identifiserte risiko for å estimere sannsynligheten for forekomst og konsekvens av forekomst og deretter se risikonivået ut ifra resultatet.

2.4.7 Risikomatrise

Gruppen har brukt risikomatrise for å vise alvorlighetsgraden til risikoene slik det framkommer av tabell 2.1. Den er fargekodet. Grønn farge symboliserer lav prioritet, risiko som er nummerert fra 0 til 6. Oransje farge er medium prioritert risiko som er nummerert fra 8 til 12 og som gruppen bør være forberedt på. Rød farge har høy prioritet og er nummerert fra 15 og oppover. Rød risiko bør gruppen etterstrebe å redusere.

Tabell 2.1: Risikomatrise

Sannsynlighet						
	5	5	10	15	20	25
	4	4	8	12	16	20
	3	3	6	9	12	15
	2	2	4	6	8	10
	1	1	2	3	4	5
		1	2	3	4	5
		Konsekvens				

2.4.8 Risikohåndtering

Hver risiko gruppen har identifisert et risikoforebyggende tiltak. Det er for å redusere risiko til et akseptabelt nivå eller hindre risikoen. Det er viktig å planlegge for mulige risikoer, ved å ha risikoforebyggende tiltak er gruppen mer forberedt.

2.4.9 Risikoovervåkning

Overvåkning av risiko gir gruppen en oversikt over identifiserte risiko og nye risikoer. Gruppen har på "god-morgenmøter" stadig nevnt viktige risikoer og notert nye risikoer.

Tabell 2.2: Maskinvare risiko.

Maskinvare							
Risko Nr.	**Type**	**Beskrivelse**	**Risikoanalyse**	**Sannsynlighet**	**Konsekvens**	**Nivå**	**Forebyggende tiltak**
R1	Maskinvarefeil	Beskrivelse av feilen	Risikoen knyttet til feilen				
R1.1	PC kræsj	Systemkræsj på en datamaskin (f.eks. tomt batteri eller blue screen).	Eventuell tap av data og tid.	3	2	6	Gjevnlig backup både lokalt på datamaskinen(e) og skyen.
R1.2	Feil/skade på ytterlig utstyr.	Tilbakeslag av hvordan oppgavene blir løst gjennom begrenset tilgang til ytterlig utstyr knyttet til prosjektet (ved feil/skade).	Kan resultere i små til middels store forsinkelser.	1	2	2	Sette opp ny oppfølgingsplan for å jobbe med noe annet midlertidlig. Eventuelt se på muligheten for å erstatte del(ene) med noe annet. Kontakt av veileder for nytt utstyr. Følg retningslinjer fra KDA.
R1.3	Feil/skade på utvikler PC.	Personen(e) det gjelder får implikasjoner med å utvikle tekniske løsninger til prosjektet fordi de ikke har tilgang til utvikler PC grunnet feil/skade på PCen.	Kan skape store forsinkelser og eventuelle hindringer for utvikling av systemkravene. Dette har mindre påvirkning på dokumentasjon av prosjektet.	1	5	5	Push ut mot Git løsning med jevnlige mellomrom. Kontakt av veileder for nytt utstyr. Følg retningslinjer fra KDA.
R1.4	Begrenset eller ingen tilgang på nødvendig maskinvare	Oppdagelse av begrenset tilgang på ressurser knyttet til utviklingen av systemkrav	Kan skape små forsinkelser og/eller komplisere hvordan oppgavene blir løst (f.eks. utvikle digital simulator).	1	3	3	Kontakt veileder angående systemkravene og muligheten for utstyret. I tillegg kan en finne andre oppgaver til person(ene) det gjelder i mellomtiden.

Tabell 2.3: Programvare risiko.

Programvare							
Risko Nr.	**Type**	**Beskrivelse**	**Risikoanalyse**	**Sannsynlighet**	**Konsekvens**	**Nivå**	**Forebyggende tiltak**
R2	Programvarefeil	Beskrivelse av feilen	Risikoen knyttet til feilen				
R2.1	Kode kompilerer ikke	Koden funker ikke	Tap av tid. I tillegg kan det ha en liten innvirkning på annen kildekode	4	2	8	"Code Reviews" under testing fasen av den agile arbeidsmodellen
R2.2	Kode kompilerer men er ufullstendig	Koden passerer "Code Review" men har uoppdagede feil som kan ha en innvirkning på hele eller deler av systemet	Tap av større mengder tid. Muligens store påvirkninger på funksjonaliteten til både området til koden og eventuelt andre systemfunksjoner	2	5	10	Testspesifikasjonene benyttes som en muligheten for feildeteksjon og korreksjon
R2.3	Jira feil	Feil med Jira som blir brukt som en oversikt over tasks i sammenheng med prosjektet	Tap av dokumentasjon og detaljert planlegging	1	4	4	Ha backup av Jira på OneDrive
R2.4	Ubuntu feil	Feil ved en ellere flere utvikler PCer som bruker Ubuntu som operativsystem	Gruppemedlemmet kan ikke jobbe og kan eventuelt bli noe forsinket	1	5	5	Passe på ekstern harddisk gitt av KDA. Denne oppbevarer tidligere dokumentasjon og filer for utvikler PCene som kan bli installert på ny Ubuntu versjon
R2.5	Feil ved kildekoden til systemet	Hele eller deler av kildekoden til systemet kompilerer ikke	Gruppemedlemmet kan ikke jobbe og kan eventuelt bli litt forsinket	1	1	1	Hent inn den tidligere versjonen av kildekoden som fungerte via Git

Tabell 2.4: Menneskelige risiko.

Menneskelige feil							
Risko Nr.	Type	Beskrivelse	Risikoanalyse	Samsynlighet	Konsekvens	Nivå	Forebyggende tiltak
R3	Menneskelige feil	Beskrivelse av feilen	Risikoen knyttet til feilen				
R3.1	Fravær hos gruppemedlemmene.	Fravær hos et eller flere gruppemedlemmer på grunn av sykdom eller ingen deltagelse	Forsinkelser i utvikling av prosjektet i sin helhet	3	3	9	Ved lengre perioder kan bachelorgruppen finne oppgaver som kan gjøres hjemmefra
R3.2	Fravær hos veiledere.	Fravær hos en eller flere veiledere på grunn av sykdom eller ingen deltagelse	Små problemer tilknyttet spørsmål og oppfølging	1	3	3	Send møteinnkalling med agenda tidlig nok for veiledere. Ved eventuelt fravær kan mail med spørsmål sendes
R3.3	Tap av kode	Et eller flere gruppemedlemmer overskriver og/eller taper kode	Tap av tid. Koden må skrives på nytt	1	5	5	Vi bruker Git for å kode parallellt og eldre versjoner av koden lagres både på server og lokale PCer
R3.4	Tap av dokumentasjon	Et eller flere gruppemedlemmer overskriver og/eller taper dokumentasjon	Tap av tid. Dokumentasjonen må skrives på nytt	2	5	10	Lokal backup av OneDrive på egne PCer og i skyen. Lagring av revisjoner og e-mail

3. Systemoversikt

I dette kapittelet blir det gitt en kort oversikt over den nåværende systemetoversikten til Coastal Shark. I tillegg til dette, blir problemstillingen til gruppen sett på fra et systemperspektiv.

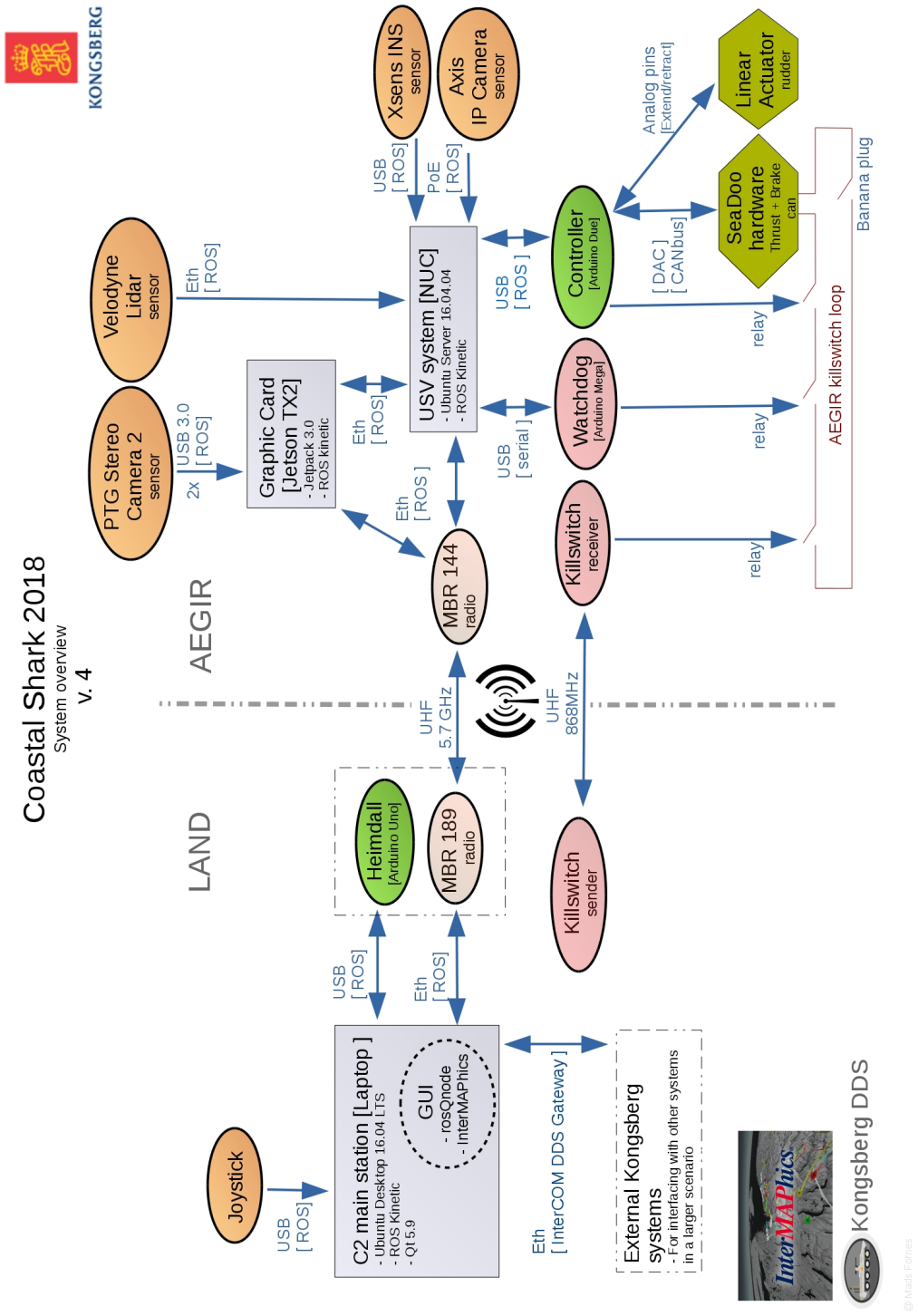
Oversikten fra figur 3.1 viser både systemoversikten og kommunikasjonsforbindelsene til Coastal Shark. Figuren er en kopi av systemoversikten laget for prosjektet i 2018.

For bachelorgruppen sin del var fokuset på kommunikasjonsforbindelsene til prosjektet, spesifikt på kommunikasjonen mellom land og Aegir. Denne koblingen skjer over en radiolink forbindelse hvor det benyttes to Maritime Broadband Radioer (MBR). Linken mellom Aegir og C2 er IP-basert og har i teorien en total kapasitet på 15Mbps.

MBR 189 befinner seg på landstasjonen og egner seg for kommunikasjon for land-til-sjø [11]. MBR 144 er installert på Aegir og er designet for små bemmannede- eller ubemannende fartøy [12]. Mellom seg vil de overføre flere forskjellige typer data i en full duplex kobling.

Programvaren til systemet er basert på Robot Operating System (ROS) som er et middleware for Unix-baserte plattformer. Det er designet for å standardisere og forenkle utviklingen av roboter, hvor hensikten er å modulisere et system for å gjøre det mulig å gjenbruke så mye kode som mulig. ROS består av tusenvis av biblioteker og andre verktøy. Blant annet tilbyr det maskinvare-abstraksjoner, drivere, meldingsutveksling mellom noder og pakkehåndteringer. I tillegg har ROS en plattformuavhengig *runtime* og vil dermed fungere likt uavhengig av hvilken maskinvare som har blitt brukt [13].

For bachelorgruppen betød dette at programvaren måtte videreutvikles med hensyn av dens fundament i ROS. Det ville si at etableringen av en kartlegging av datakommunikasjonen og prioritering av dataoverføringene enten måtte bli utviklet gjennom ROS eller konstruert som en egen tilpasset løsning utenfor *middlewaret*. Datainnsamlingsverktøyet derimot var nødt til å bli skrevet i ROS.



Figur 3.1: Systemoversikt Coastal Shark

3.1 Systemfunksjoner

Systemfunksjonene til bachelorprosjektet, som kort nevnt tidligere, omfattet styring, representasjon og regulering av datakommunikasjonen mellom kontrollstasjonen på land (C2) og Aegir. I tillegg til dette ville det dannes både testsett og testlogger gjennom et datainnsamlingssystem, samtidig som at det ville bli sett på muligheten til å innføre et sikkerhetaspekt til denne kommunikasjonen. Oppgaven innebar da at bachelorgruppen blant annet så på slikt som kontroll av dataflyt, prioritering av meldinger og struping av nettverket.

3.1.1 Styring og regulering av datakommunikasjon

For å kunne styre og regulere datakommunikasjonen var det nødvendig at det ble tatt frem en oversikt over alt av datakommunikasjonen. Denne oversikten over dataene ville blant annet kunne inkludere slikt som innhold, destinasjon og fra-via-til interface. I henhold til programvaren omfattet dette kommunikasjonen mellom to forskjellige ROS nettverk, nemlig C2 og Aegir.

Hver av nettverkene har sine egne ROS noder, som i all hovedsak er en oppdeling av systemet. Hvor nodene er mindre, separate deler og selvstendige prosesser som gjør det enklere å forstå, vedlikeholde og gjenbruke deler av systemet.

Fordelen med et slikt system som ROS og dens noder, er at alle nodene kjører uavhengig av hverandre. Det vil si at hvis en node svikter så kan resten av systemet fortsette dersom det er mulig. I tillegg vil koden for diverse utstyr enkelt kunne bli gjenbrukt, et eksempel på dette kan være en sensor.

Nodene i et ROS nettverk vil også kunne utveksle meldinger mellom seg. For at disse prosessene skal kunne kommunisere mellom hverandre blir det benyttet noe som kalles *topics* og *services*.

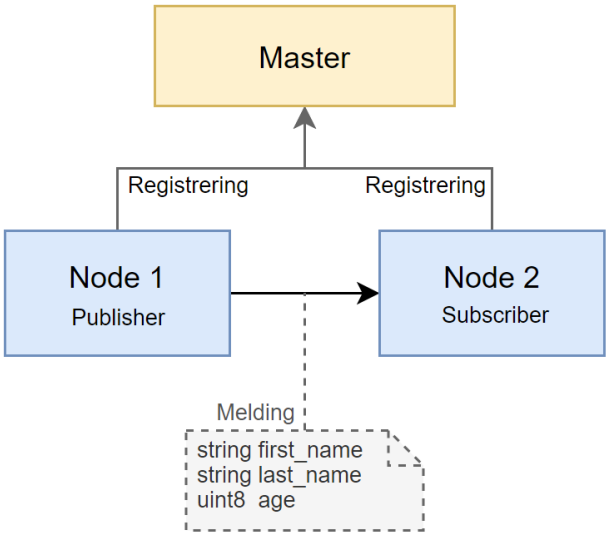
Topics

Topics blir brukt som en meldingskanal. Meldingssystemet bruker en *publish* og *subscribe* semantikk, der hver node har mulighet til å både sende og motta meldinger. Noder har mulighet til å sende og motta meldinger over kanaler, som gjør de til *publisher* og *subscriber* henholdsvis. Et *topic* kan ha en-til-en, mange-til-en, eller mange-til-mange forhold, men vil alltid være en enveiskommunikasjon.

Hver meldingstype er forhåndsdefinert i sin egen meldingsfil (.msg fil). Meldingsstrukturen er definert av dens typer, i likhet med en vanlig datastruktur.

Services

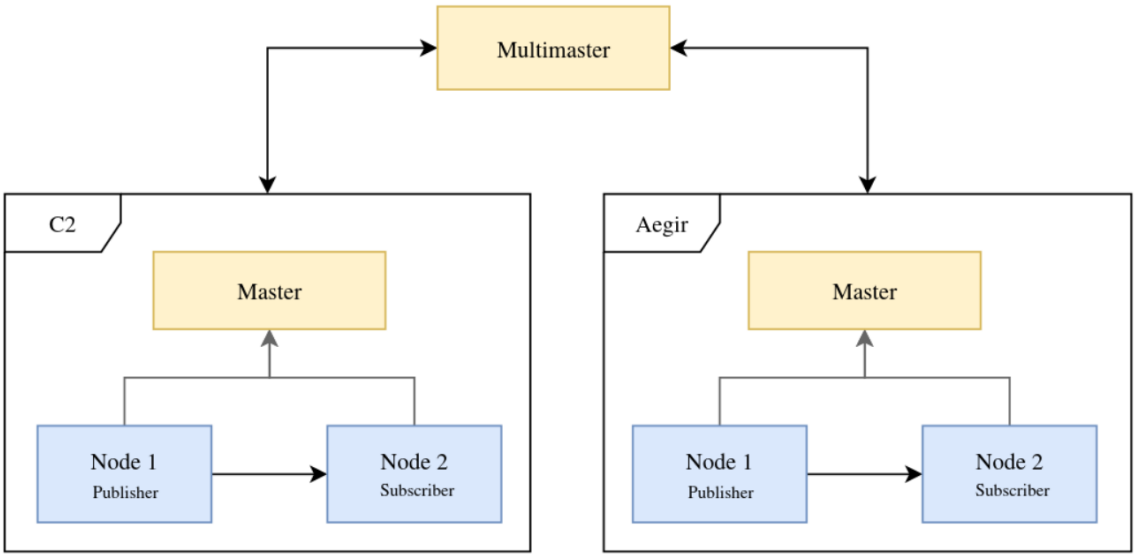
Services er annen type meldingskanal. I motsetning til *Topics* bruker *services* en *request-reply* paradigme, der hver etterspørsel forventer en respons tilbake. En hver service er definert som et par meldinger som inneholder semantikken til både etterspørselen og responsen. *Service*-kall er synkrone og blokkerende.



Figur 3.2: Illustrasjon av meldingsflyten over topics.

For å gjøre det mulig for noder å finne andre noder å utveksle informasjon til, benytter ROS seg av noe kalt en *master*. En *master* fungerer som en navnetjeneste/DNS server. Hver *topic* og/eller *service* blir registrert hos en *master*. Hvis en annen node vil motta meldinger fra en av disse må den registrere dette hos *masteren*, deretter vil *masteren* sette opp en tilkobling direkte mellom nodene. Denne kommunikasjonen foregår som regel over TCP/IP hvor den mest vanlige protokollen er TCPROS [14]. Figur 3.2 er en visuell illustrasjon på kommunikasjonen mellom to noder.

Om derimot noder skal sende meldinger på tvers av nettverk blir det benyttet en *multimaster*. En *multimaster* er hovedsakelig ansvarlig for å publisere og synkronisere *topic* og *services* over flere nettverk. Noe som gjør det enkelt å koble flere lokale nettverk sammen, hvor hvert nettverk vil ha sin egen *master*.



Figur 3.3: Illustrasjon av meldingsflyten over topics.

Systemet bruker et *multimaster*-oppsett via ROS. Det har to *mastere*, en for C2 og en for farkosten Aegir, figur 3.3 viser en abstrakt versjon av dette.

3.1.2 Prioritering av datakommunikasjon

Bachelorgruppens hovedoppgave var å finne en løsning for prioritering av meldinger fra Aegir til C2. Dette systemkravet gikk ut på at nettverket til Coastal Shark kunne både endre båndbredde og/eller oppleves som et *lossy* nettverk.

I tidligere prosjekt har det vært lite hensyn til hvordan Aegir kan bevege seg i forskjellige territorium og hvordan dette påvirker kommunikasjonen. Prioriteringen av meldinger ville dermed ta utgangspunkt i dette. Med det ville det bli tatt hensyn til hvordan det skal prioritere meldinger mellom seg når nettverksytelsen svekkes.

For å teste dette ble en strupingsmekanisme utviklet og så brukt til å simulere kommunikasjonen. Prioriteringen av datakommunikasjonen vil da gå på premisset om at enkelte meldinger har mer verdi enn andre utifra forskjellige scenarioer. For eksempel, det kan være at meldinger i form av bilder fra et kamera kan prioriteres som lite viktig hvis nettverksytelsen er lav. Et annet er at *killswitch* til enhver tid vil være viktig å prioritere som den viktigste faktoren i systemet.

En hvilken som helst operatør av C2 skal kunne manuelt prioritere meldinger bort fra et standard sett skapt av prioriteringsfunksjonen. Gjennom et *GUI* for C2 kan operatøren gjøre dette.

3.1.3 Styring av nettverksytelse

For å kunne teste ut prioriteringssystemet, ble det benyttet en PC mellom de to kommunikasjonskanalene, med mulighet for å endre båndbredden.

3.1.4 Innsamling av data

En deloppgave bestod i å implementere et innsamlingssystem som ser på alle meldinger som sendes fra Aegir, og skrive innholdet i meldingene til en fil. Gruppen hadde først en ide om at innsamlingssystemet skulle befinne seg på en egen pc for styring av nettverksytelse. Det ble imidlertid klart, etter litt diskusjon, at innsamlingssystemet måtte implementeres på Aegir ettersom meldingene skal prioriteres før de sendes fra Aegir.

3.1.5 Logging av meldinger

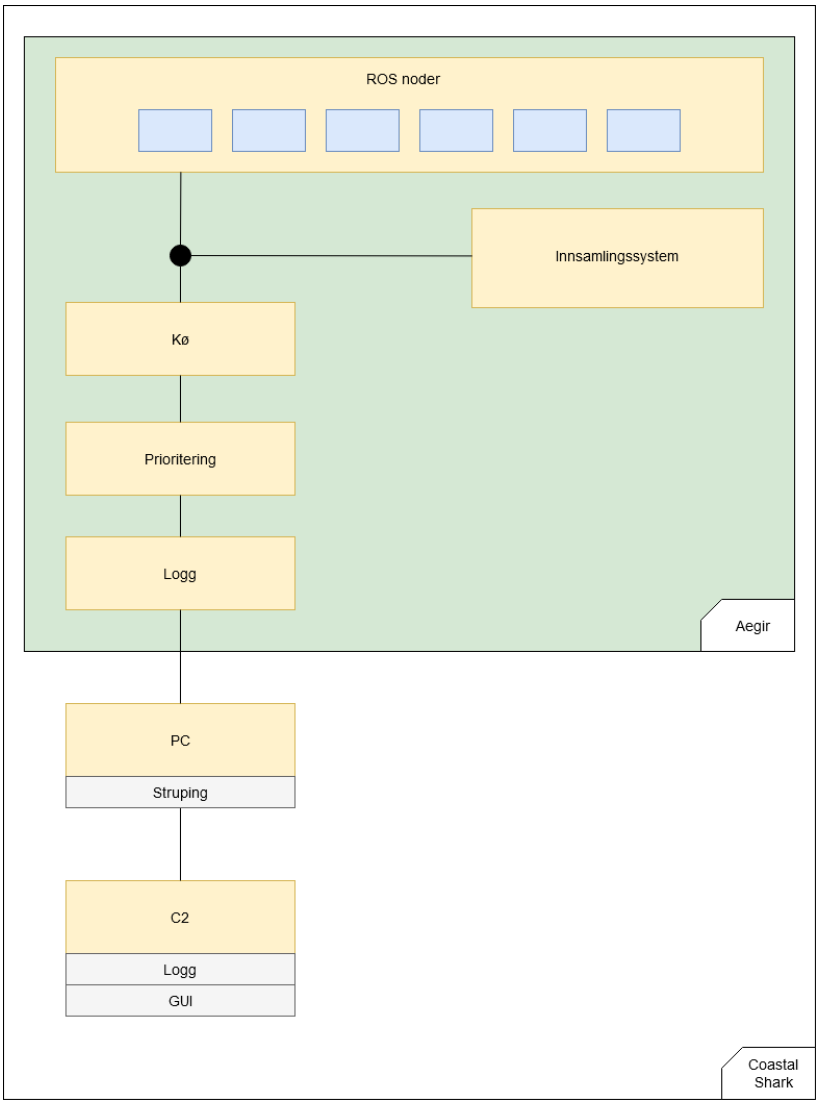
En annen deloppgave var å logge all datatrafikken over nettverket. Alle meldinger som sendes og mottas skal logges, både på Aegir og C2.

Når meldinger er ferdig prioritert skal loggesystemet logge hvilken type data som blir sendt til C2 og skrive disse meldingene til en fil. Senere, om det blir tid, kan dette endres til at ikke bare data type, men også innhold av meldinger blir skrevet til filen.

3.1.6 Oversikt systemfunksjoner

Som det fremkommer av figur 3.4, vil meldinger som sendes fra sensorene på Aegir, leses av innsamlingssystemet og legges i en kø som venter på å bli sendt over nettverket. Meldingene vil deretter prioriteres utifra nettverksytelsen, og til slutt logges før

de sendes til C2. På C2 vil det være en GUI som viser dataene fra innsamlingssystemet og loggen, samt en logg over meldinger som sendes og mottas på landstasjon. Figuren viser også en PC som brukes mellom de to kommunikasjonskanalene for å kunne simulere nettverksytelsen.



Figur 3.4: Systemfunksjoner

3.1.7 Sikkerhetsaspekt

Et av B-systemkravene til prosjektet var å vurdere om det burde innføres et sikkerhetsaspekt mot avlytting og innbrud på kommunikasjonen mellom Aegir og C2. Dette var med hensyn til at datautvekslingen mellom disse består av en rekke komponenter som kan karakteriseres som privat data. Herav blant annet data om miljøet fra eksempelvis LIDAR sensor og kameraer, samt informasjon om Aegir gjennom dens ruter og posisjon.

Siden systemet Coastal Shark er bygget på ROS ville den originale overføringen av meldinger fra *publisher*- til *subscriber* node vært i klartekst [15]. Det ville dermed vært en risiko for at eksterne kilder enkelt kunne fått tilgang på data sendt mellom noder i et ROS nettverk, eller på tvers av nettverket. I tillegg var det en risiko for at eksterne kilder kunne koblet seg opp til nettverket og endret atferden mellom nodene. Et eksempel på sistnevnte ville vært et Man-In-The-Middle angrep, hvor et ondsinnet parti kan etterligne tilgangspunktene i nettverket [16] for å oppnå dette.

I tillegg til dette ville andre faktorer som eventuelt ubeskyttede TCP porter, *legacy* systemer og kryptering av lokal data [17] vært noe som kunne blitt vurdert som sikkerhetstiltak. Hvor tanken og målet med en vurdering av disse sikkerhetsaspektene ville vært at de vil satt et grunnlag for hvordan bachelorgruppen valgte å utvikle systemet videre.

For eksempel ville en kryptering av meldinger muligens hatt en innvirkning på nettverket. Ved at enkelte kryptosystemer kan øke datamengden, ville dette resultert i en større bitrate over nettverket. Siden et av kravene for prosjektet var å se på hvordan man kan prioritere meldinger utifra endringer i bitraten, ville det vært gunstig å tatt dette i betrakning. Et annet eksempel ville vært å se på om det fantes grunnlag for å kryptere data som ble samlet inn og/eller logget. Hvis det var en risiko for at denne dataen kunne blitt komprimert og havnet på avveie, så kunne det gitt en viss trygghet å implementert en form for kryptertering.

4. Krav- og testspesifikasjon

Dette kapittelet tar for seg håndtering av krav og test spesifikasjoner assosiert med prosjektet. Her blir også alle krav og de tilsvarende testene listet opp.

Ved starten av prosjektet fikk bachelorgruppen oppgitt en liste med kravspesifikasjoner fra arbeidsgiver. Disse ble senere brutt ned i mindre underkrav.

Malen nedenfor ble brukt for alle funksjonelle- og ikke-funksjonelle krav, i tillegg til begrensningene prosjektet har:

[X.Y]	[Dato]	[Tittel]	[Status]	[Prioritet]
	[Opphav]	[Test ID]		
	Beskrivelse	[Beskrivelse]		
	Kommentarer	[Kommentarer]		

For testspesifikasjonene har bachelorgruppen spesifisert tester som tar for seg underkravene. Gruppen tenkte også på å ha mindre tester som eksempelvis unit test for å underbygge testene. Malen nedenfor ble brukt for alle testspesifikasjonene:

[TN]	[Dato]	[Status]	[Prioritet]
	Krav	[Krav som blir testet]	
	Beskrivelse	[Beskrivelse av testen som skal bli utført]	
	Kriterier	[Kriterier for at testen skal godkjennes]	

Status i forbindelse med krav informerte om kravet har blitt godkjent av arbeidsgiver. I forbindelse med test ga status en indikasjon på tilstanden til de tillhørende kravene.

Prioriteten er fordelt inn i tre kategorier:

- **A-krav:** Krav som absolutt måtte bli oppfylt.
- **B-krav:** Krav som burde oppfylles. Viderebygget gjerne på funksjonaliteten fra A-krav.
- **C-krav:** Krav som kunne bli oppfylt, men som ikke var en prioritet. Ville kun bli oppfylt dersom det ble nok tid mot slutten av prosjektet.

4.1 Funksjonelle krav

F1.1	2020-01-01	Oversikt over datakommunikasjon	Godkjent	A
	KDA			
	Beskrivelse	Ta frem en oversikt over all datakommunikasjon mellom Aegir og kontrollstasjonen på land.		
	Kommentarer			

F1.2	2020-02-03	Oversikt over datakommunikasjon	Godkjent	A
	Gruppen			
	Beskrivelse	Systemet skal få en oversikt over alle nodene.		
	Kommentarer			

F1.3	2020-02-03	Oversikt over datakommunikasjon	Godkjent	A
	Gruppen			
	Beskrivelse	Systemet skal observere meldingsoverføringer mellom nodene.		
	Kommentarer			

F1.4	2020-02-03	Oversikt over datakommunikasjon	Godkjent	B
	Gruppen			
	Beskrivelse	Systemet skal kunne representere en oversikt over datakommunikasjon i GUI.		
	Kommentarer			

F2	2020-01-01	Oversikt over datakommunikasjon	Godkjent	A
	KDA			
	Beskrivelse	Ta frem en oversikt av hva slags type data og datamengde som skal prioriteres hvis nettverksytelsen mellom Aegir og kontrollstasjonen på land endres (reduserer eller økes).		
	Kommentarer			

F2.1	2020-02-03	Oversikt over datakommunikasjon	Godkjent	A
	Gruppen			
	Beskrivelse	Systemet kan kategorisere de forskjellige meldingstypene til nodene.		
	Kommentarer			

F2.2	2020-02-03	Oversikt over datakommunikasjon	Godkjent	A
	Gruppen			
	Beskrivelse	Systemet skal observere nettverksytelsen kontinuerlig.		
	Kommentarer			

F2.3	2020-02-03	Oversikt over datakommunikasjon	Godkjent	A
	Gruppen			
	Beskrivelse	Systemet skal ha funksjonaliteten til å prioritere meldinger.		
	Kommentarer			

F2.4	2020-02-03	Oversikt over datakommunikasjon	Godkjent	A
	Gruppen			
	Beskrivelse	Systemet skal ha en forhåndsdefinert prioritering av meldingene.		
	Kommentarer			

F2.5	2020-02-03	Oversikt over datakommunikasjon	Godkjent	A
	Gruppen			
	Beskrivelse	Systemet skal prioritere meldinger ut ifra nettverksytelsen.		
	Kommentarer			

F2.6	2020-02-03	Oversikt over datakommunikasjon	Godkjent	A
	Gruppen			
	Beskrivelse	Systemet skal kunne presentere en oversikt over prioriteringene i GUI.		
	Kommentarer			

F3	2020-01-01	Kontrollsystem	Godkjent	A
	KDA			
	Beskrivelse	Lage et kontrollsystem i applikasjonen hvor man kan endre prioritering på gitt sette meldinger for å styre og regulere datakommunikasjonen.		
	Kommentarer			

F3.1	2020-02-03	Kontrollsystem	Godkjent	A
	Gruppen			
	Beskrivelse	Systemet skal ha funksjonaliteten til å kunne endre prioriteringer.		
	Kommentarer			

F3.2	2020-02-03	Kontrollsystem	Godkjent	A
	Gruppen			
	Beskrivelse	Operatøren skal kunne overstyre systemets forhåndsdefinerte prioriteringer manuelt.		
	Kommentarer			

F3.3	2020-02-03	Kontrollsystem	Godkjent	A
	Gruppen			
	Beskrivelse	Systemets prioriteringskontroll skal implementeres i GUI.		
	Kommentarer			

F4	2020-01-01	Nettverkskontroll	Fjernet	A
	KDA			
	Beskrivelse	Nettverkstrafikken/datakommunikasjonen skal reguleres med en Cisco router.		
	Kommentarer	Kravet har blitt erstattet med F6.		

F5	2020-01-01	Nettverkskontroll	Fjernet	A
	KDA			
	Beskrivelse	Ta frem en applikasjon med brukergrensesnitt hvor man kan styre, kontrollere og vise nettverkstrafikken på Cisco routeren.		
	Kommentarer	Kravet har blitt erstattet med F6.		

F6	2020-02-07	Nettverkskontroll	Godkjent	A
	Gruppen			
	Beskrivelse	Ta frem en virtuell NIC med sitt eget brukergrensesnitt for å regulere nettverkstrafikken.		
	Kommentarer			

F6.1	2020-05-15	Nettverkskontroll	Godkjent	B
	Gruppen			
	Beskrivelse	Den virtuelle NICen skal ha en GUI applikasjon for å regulere nettverkstraffiken.		
	Kommentarer			

F7	2020-05-22	Loggingssystem	Godkjent	A
	KDA			
	Beskrivelse	Loggesystemet skal logge alle meldinger som blir sendt og mottatt.		
	Kommentarer			

F8	2020-05-22	Loggingssystem	Godkjent	A
	Gruppen			
	Beskrivelse	Loggesystemet skal logge nettverksytelsen.		
	Kommentarer			

F9	2020-05-22	Loggingssystem	Godkjent	A
	Gruppen			
	Beskrivelse	Operatør kan lese logger.		
	Kommentarer			

F10	2020-05-22	Loggingssystem	Godkjent	B
	KDA			
	Beskrivelse	Operatør kan åpne logger i et GUI som en visuell fremstilling.		
	Kommentarer			

F11	2020-05-22	Datainnsamlingssystem	Godkjent	A
	Gruppen			
	Beskrivelse	Datainnsamlingsystemet skal samle inn all meldingsdata.		
	Kommentarer			

F12	2020-05-22	Datainnsamlingssystem	Godkjent	A
	Gruppen			
	Beskrivelse	Operatør skal ha mulighet til å velge hva som skal samles.		
	Kommentarer			

F13	2020-05-22	Datainnsamlingssystem	Godkjent	A
	KDA			
	Beskrivelse	Datainnsamlingsystemet skal generere testsett/testlogger.		
	Kommentarer			

F13.1	2020-05-22	Datainnsamlingssystem	Godkjent	A
	Gruppen			
	Beskrivelse	Systemet skal kunne lese testsett/testlogger		
	Kommentarer			

F13.2	2020-05-22	Datainnsamlingssystem	Godkjent	A
	KDA			
	Beskrivelse	Systemet skal kunne sende data basert på testsett/testlogger.		
	Kommentarer			

F14	2020-05-22	Datainnsamlingssystem	Godkjent	A
	Gruppen			
	Beskrivelse	Operatør kan åpne og lese testsett i et GUI som en visuell fremstilling.		
	Kommentarer			

F15	2020-01-01	Kryptering	Godkjent	B
	KDA			
	Beskrivelse	Innføre et security aspect på datakommunikasjonen. Vurdere om å innføre kryptering, sjekksum på hele eller deler av datakommunikasjonen.		
	Kommentarer			

F15.1	2020-01-01	Kryptering	Godkjent	B
	Gruppen			
	Beskrivelse	Meldinger som går mellom Aegir og landstasjonen skal krypteres.		
	Kommentarer			

F15.2	2020-01-01	Kryptering	Godkjent	B
	Gruppen			
	Beskrivelse	Sjekksum på krypterte meldinger skal implementeres.		
	Kommentarer			

4.2 Ikke-funksjonelle krav

IF1	2020-01-01	Nettside	Godkjent	B
	KDA			
	Beskrivelse	Vedlikeholde nettsiden til Coastal Shark: www.coastalshark.no		
	Kommentarer			

IF2	2020-01-01	Nettside	Godkjent	A
	KDA			
	Beskrivelse	Kildekoden skal være dokumentert med doxygen		
	Kommentarer			

4.3 Begrensninger

B1	2020-01-01	Begrensning	Godkjent	A
	KDA			
	Beskrivelse	Systemet skal ikke publiseres eller gjøres kjent for allmenheten.		
	Kommentarer			

B2	2020-01-01	Programmeringsspråk	Godkjent	A
	KDA			
	Beskrivelse	Programvaren skal være utviklet i programmeringsspråket C++.		
	Kommentarer			

B3	2020-01-01	Operativsystem	Godkjent	A
	KDA			
	Beskrivelse	Operativsystem som skal brukes: Linux Ubuntu.		
	Kommentarer			

B4	2020-01-01	Versjonskontroll	Godkjent	A
	KDA			
	Beskrivelse	Software versjonskontroll som skal brukes: GIT		
	Kommentarer			

B5	2020-01-01	Utvikling	Godkjent	A
	KDA			
	Beskrivelse	All utvikling av programvare skal skje på datamaskiner, skjermer, utstyr levert fra KDA.		
	Kommentarer			

4.4 Testspesifikasjon

T1	05.05.2020	Godkjent	Høy
	Krav	F1.1 - 1.3	
	Beskrivelse	• Sett opp en simulering av Coastal Shark • Kjør “roslaunch spotter spotter“ programmet i terminalen • Se på de forskjellige variablene til noden som Spotter programmet konstruerer	
	Kriterier	Verifiser at programmet tar frem en oversikt over datakommunikasjonen og lagrer dette i en dataflow.csv fil. Radene skal inneholde informasjon om de forskjellige nodene i henhold til aktivitet, publications, subscriptions, services og nettverksforbindelsene.	

T2	05.05.2020	Ikke godkjent	Middels
	Krav	F1.4	
	Beskrivelse	• Start opp C2 GUI • Trykk på “Communications“ knappen i menyen • Velg “dataflow sheet“	
	Kriterier	Verifiser at QT applikasjonen dukker opp som en liste hvor det blir ført opp navn, publications, subscriptions, services, nettverksforbindelser og aktivitet på alle nodene.	

T3	2020-02-04	Godkjent	Høy
	Krav	F2.3, F2.5	
	Beskrivelse	• Start systemet • Strup nettverskytelsen • La systemet kjøre en liten periode • Stopp systemet og inspiser generert loggfil	
	Kriterier	Datakommunikasjonen ble endret i henhold til forhåndsbestemte prioritet.	

T4	2020-02-04	Godkjent	Høy
	Krav	F2.1, F2.4	
	Beskrivelse	• Start opp C2 GUI • Trykk på “Communications“ i menyen • Velg “Prioritering“	
	Kriterier	Verifiser at et vindu dukker opp som viser prioriteten til de forskjellige meldingstypene, gruppert etter topics.	

T5	2020-02-04	Godkjent	Høy
	Krav	F2.6 - F3.3	
	Beskrivelse	• Start opp C2 GUI • Trykk på ”Communications”i menyen • Velg Prioritering” • Endre prioriteten til diverse meldingstyper.	
	Kriterier	Verifiser at et vindu dukker opp som viser prioriteten til de forskjellige meldingstypene, og at det er mulig å endre på prioriteten. Lagre deretter innstillingene, og verifiser gjennom loggfilen av riktig prioritet ble satt.	

T6	2020-02-04	Godkjent	Høy
	Krav	F2.2	
	Beskrivelse	• Start opp Aegir og C2 slik at de vil kommunisere med hverandre. • Etter å ha kjørt systemet, ta frem loggfilen som ble generert.	
	Kriterier	Verifiser at båndbredden ble logget sammen med sendte meldinger.	

T7	2020-02-04	Fjernet	Høy
	Krav	F4	
	Beskrivelse	• Koble to maskiner sammen gjennom Cisco-svitsjen • Strup båndbredden gjennom kontrollpanelet til svitsjen	
	Kriterier	Verifiser at båndbredden ble strupet til den gitte hastigheten ved bruk av 'netcat' verktøyet.	

T8	2020-04-02	Godkjent	Høy
	Krav	F6-F6.1	
	Beskrivelse	• Start strupingsenhet, enhet S. • Koble en enhet på hver av Ethernet-portene til S. Enhet A og B. (A—S—B) • Kjøre Iperf på maskin A og B. A som server og B som klient som sender pakker i 3 minutter med TCP som har en window size på 85kbyte. • Kjøre 2 sendinger fra klient på B, og notere båndbredden på klienten uten struping. • Bytte server til B og klient til A. • Kjøre 2 sendinger fra klient på A, og notere båndbredden på klienten uten struping. • Strupe båndbredden til 10mbps, med burst 32kbit og latency 500ms på S. • Kjøre 2 sendinger fra klient på A, og notere båndbredden på klienten. • Bytte server til A og klient til B. • Kjøre 2 sendinger fra klient på B, notere båndbredden på klienten. • Bytte server til B og klient til A. • Strupe båndbredden til 1mbps, med burst 32kbit og latency 500ms på S, og gjenta steg 8-11. • Strupe båndbredden til 500kbps, med burst 32kbit og latency 500ms på S, og gjenta steg 8-11. • Strupe båndbredden til 50kbps, med burst 32kbit og latency 500ms på S, og gjenta steg 8-11. • Strupe båndbredden til 10kbps, med burst 32kbit og latency 500ms på S, og gjenta steg 8-11.	
	Kriterier	Den gjennomsnittlige båndbredden på klienten skal tilsvare den gitte båndbredden på strupingsverktøyet.	

T9	2020-05-22	Godkjent	Høy
	Krav	F7-F9	
	Beskrivelse	• Start talker- og listenernode. • Kjør i 10 sekunder. • Stopp prosessen i listenernode. • Observer CoastalShark_tidspunkt.txt på Aegir. • Start Excel og åpne den angitte filen.	
	Kriterier	CoastalShark_tidspunkt.txt på Aegir skal fylles med valgte data for logging. Innspillingsfilen skal kunne åpnes direkte fra Excel.	

T10	2020-05-22	Ikke godkjent	Høy
	Krav	F10	
	Beskrivelse	• Åpne GUI applikasjonen for loggingen. • Åpne en logg fil i GUI og excel.	
	Kriterier	Observer at GUI åpner. Observer at filen åpnes og at GUI og excel har den samme informasjonen.	

T11	2020-05-22	Godkjent	Høy
	Krav	F11-F14	
	Beskrivelse	• Start en talker node som publiserer til 10 forskjellige topics. • Kjør launch filen til Aegir noden og start opp GUI kontroll for datainnsamlingen. • Åpne bagfiles mappen i hjem mappen. Observer bagfilene underveis. • Start innsamling av alle topics, kjør i 30 sek, stop innsamling. Gjenta 2 ganger. • Start innsamling av 5 topics, kjør 30 sek, stop innsamling. Gjenta 2 ganger. • Start innsamling av alle topics, kjør i 30 sek, Start innsamling av 5 topics, kjør 30 sek, stop innsamling. Gjenta 2 ganger. • Åpne bag filene med rqt_bag.	
	Kriterier	Skal lage bagfiler. Data og topics i bag filene skal samsvare med innsamlingen som er valgt.	

T12	2020-02-04	Godkjent	Middels
	Krav	F15-F15.2	
	Beskrivelse	• Start systemet • Send krypterte meldinger mellom kontrollstasjonen og Aegir • Inspiser meldingene som blir sendt og mottatt gjennom en packet sniffer	
	Kriterier	Verifiser at dataen er kryptert, og at ingenting blir sendt i plaintext.	

5. Dynamisk dataoversikt

Dette kapittelet går gjennom systemkravene rundt å ta frem en dynamisk oversikt over all datakommunikasjon i Coastal Shark, samt utviklingen av et brukergrensesnitt som gjør det mulig for en operatør å lese av informasjonen.

Vedlegg B tilhører dette kapittelet.

5.1 Problemstilling

Siden Coastal Shark er bygget på ROS vil informasjon om datakommunikasjonen kunne hentes inn på flere forskjellige måter, men samtidig bli låst til bruken av ROS. Dette innebærer at gruppen har evaluert flere ulike metoder for å nå målet med å produsere informasjonen, hvor metodene har blitt veid opp mot hverandre og sammenlignet for best mulig resultat. Det samme gjelder et brukergrensesnitt, hvor det finnes flere forskjellige typer løsninger som kan bli brukt til å fremstille informasjonen over den dynamiske dataoversikten.

Informasjon som må hentes ut av Coastal Shark vil være de grunnleggende egenskapene til hver ROS node. Dette vil inkludere navnene på nodene, deres ROS *publications*, ROS *subscriptions*, ROS *services*, nettverkskoblinger og aktivitet. Hvor informasjonen om nettverkskoblingene vil inneholde hvilke *topics* som blir sendt, deres retning og på hvilket transportlag. Når det gjelder aktiviteten til nodene, så vil dette kunne uttrykkes i form av et boolsk uttrykk som holder oversikt over om noden er aktiv eller ikke på informasjonen den er registrert under.

Som en visualisering av informasjonen vil en type brukergrensesnitt måtte bli utviklet. Et eksempel på dette ville vært å ha utviklet en graf som illustrerer nodene og deres egenskaper. Et annet ville vært å ha lagd en liste som representerer informasjonen. Den eneste begrensningen til et brukergrensesnitt er at gruppen må benytte seg av Qt hvis en grafisk representasjon blir brukt.

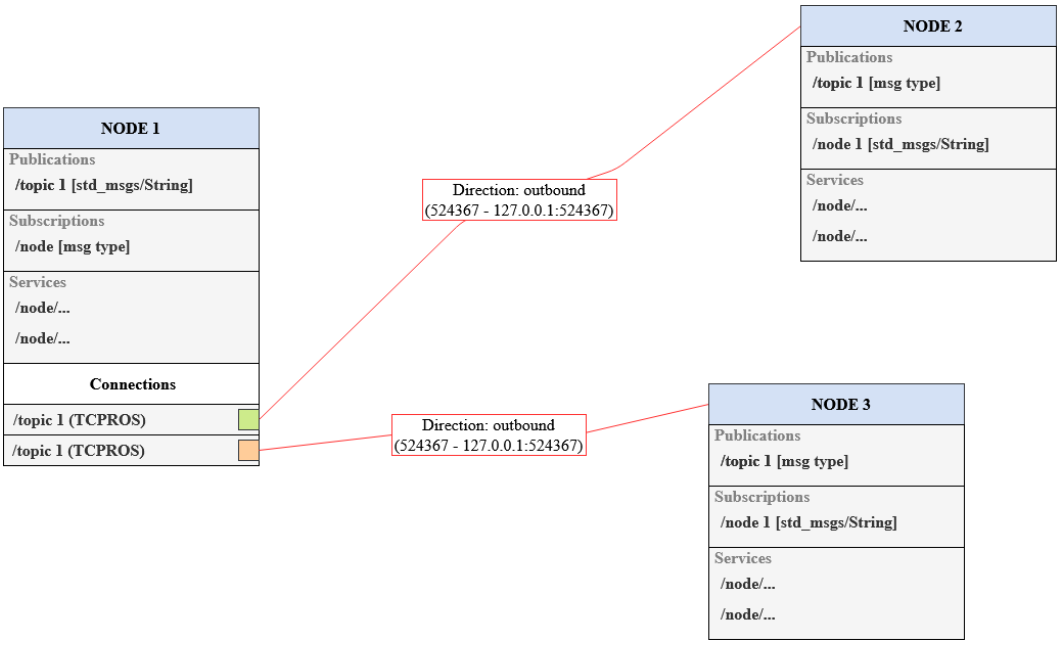
Det endelige målet med å skape en oversikt og representere denne, vil være å la en operatør av C2 kunne benytte seg av informasjonen fra den dynamiske oversikten til å foreta viktige prioriteringer gjennom prioriteringssystemet.

I tillegg må oversikten være dynamisk, noe som vil si at den må oppdateres kontinuerlig. For dette har gruppen tatt utgangspunkt i at dataoversikten skal bli oppdatert en gang i sekundet.

5.2 Løsningsforslag

Det første gruppen begynte med var å visualisere seg både hva slags informasjon som var gjeldende og hvordan denne skulle bli fremstilt. En av de originale tankene var å benytte seg av en alt i alt graf. Denne grafen ville da kunne tegnet nodene med deres respektive egenskaper, for å ha så tegnet relasjonene mellom nodenene. Sammen med prioriteringssystemet ville grafen kunne ha gitt en operatør en komplett oversikt over den dynamiske dataflyten og muligheten til å prioritere.

Formålet gruppen hadde med en slik graf var at dette ville gjort det lett for en operatør av C2 å lese den dynamiske dataoversikten. Med andre ord, det var et design som ble foreslått med operatøren i tankene. Siden grafen ville hatt en større grad av brukervennlighet en mange andre alternativer. Figur 5.1 viser et eksempel på hvordan en graf for den dynamiske dataoversikten kunne ha sett ut.



Figur 5.1: Grafisk fremstilling av dynamisk dataoversikt graf

En annen foreslått løsning for representasjonen av informasjonen var å lage en liste som oppsummerte dataoversikten som en tabell. En slik fremvisning ville ikke ha vært like estetisk og brukervennlig som en graf, men ville fortsatt ha representert informasjon om systemet på en oversiktlig måte. Tabell 5.1 viser hvordan en slik tabell ville kunne sett ut uten å ha integrert en løsning med prioriteringssystemet.

Tabell 5.1: Tabell alternativ for dynamisk dataoversikt

Name	Publications	Subscriptions	Services	Connections	Alive
/node	/topic	/node	/node/... /node/...	topic: /topic to /node Direction: outbound (524367 - 127.0.0.1:524367) Transport: TCPROS	True
...	...	...	...	...	...

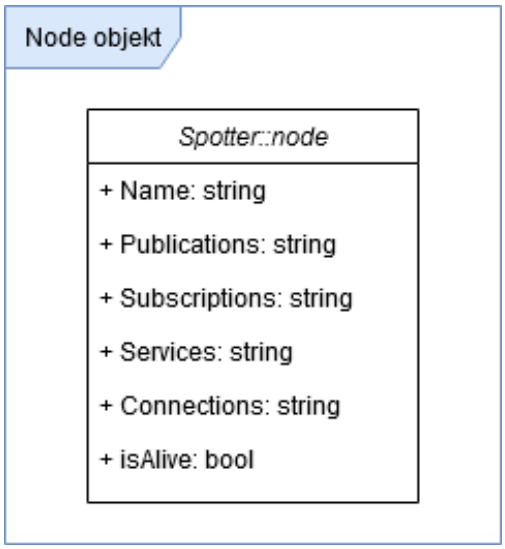
Vedlegg B går gjennom detaljene på de ulike løsningsforslagene for så til slutt sammenligne disse i form av en vekting. Løsningen som ble valgt er å hente inn og vise frem den dynamiske oversikten ved hjelp av ROS C++ API, samt utvikling av en Qt applikasjon som er bygd med QTableWidget og som illustrerer en grafisk tabell.

Grunnlaget for valget var at en grafisk tabell utviklet som en Qt applikasjon ville gi en nokså god oversikt, men ikke ta for mye tid å lage. Samtidig ville biblioteket ROS C++ API eller *roscpp* [18], gjøre det mulig å hente inn essensiell informasjon over ROS nodene i systemet gjennom kildekoden.

Med en slik løsning vil informasjon om nodene kunne bli brukt andre steder i systemet om det er ønskelig. I tillegg vil det legge et grunnlag for å kunne skalere systemet opp eller ned ut ifra behov. Et eksempel på å skalere systemet opp er å legge til informasjon om frekvensbåndbredde på nodene.

5.3 Arkitektur

Løsningsens arkitektur benytter seg av en Spotter ROS node som er basert på en Spotter-klasse. Denne består av funksjoner som kommuniserer med *roscpp* for å hente inn informasjon om nodene i systemet. Deretter blir informasjonen splittet opp ved hjelp av *regular expression* (regex) og lagret som variabler på struct objekter kalt node. Det vil si at hver ROS node i systemet blir sitt helt eget objekt med forskjellige variabler som inneholder informasjon. Klassediagrammet fra figur 5.2 viser til hva slags informasjon hvert objekt vil inneholde.



Figur 5.2: Spotter node for dynamisk dataoversikt

Sammen med Spotter-klassen er node objekter med i et Spotter navnrom. På denne måten kan node objektene, som kan bli kalt via sitt navn og navnrom, kunne bli benyttet i alt fra andre systemfunksjoner til å representere informasjon om nodene. Det samme gjelder for å bygge videre på Spotter-klassen og dens funksjoner.

Med andre ord, Spotter-klassen er ansvarlig for kommunikasjonen med *roscpp* gjennom funksjonene sine. Hvor det endelige målet er å bygge opp separate node entiteter med et format som kan gjøre det mulig å bruke disse andre steder i systemet.

5.3.1 Regex funksjoner

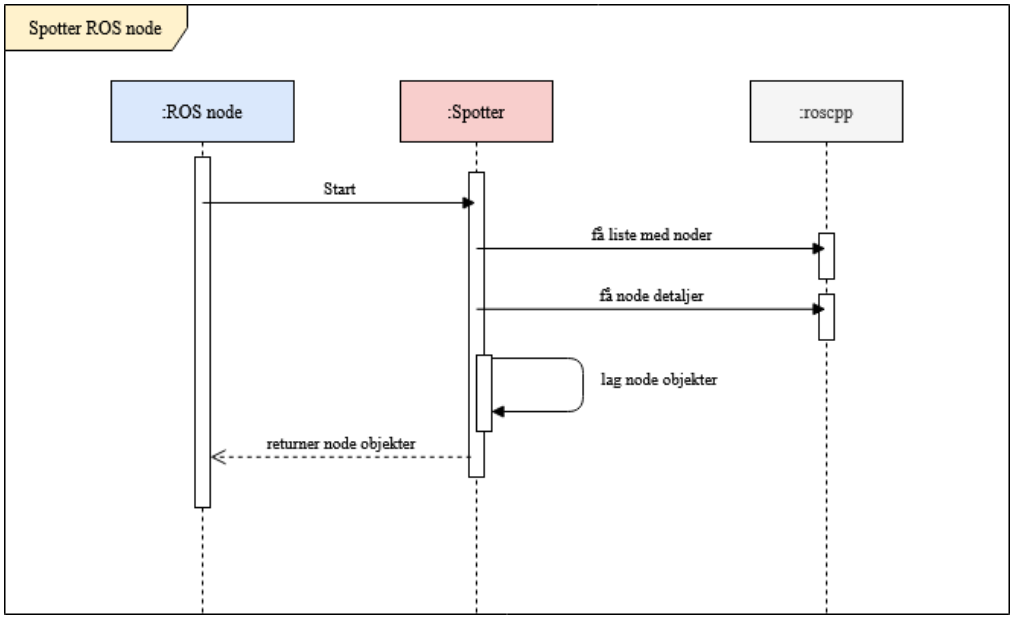
For Spotter-klassen har det blitt skrevet to funksjoner basert på *regular expression* (regex). Disse funker som strengsøkealgoritmer som har et mål om å få frem den spesifiserte oversikten til node objektene basert på ulik informasjon fra ROS nodene i systemet. Den ene funksjonen har blitt utviklet til å være både dynamisk og basert på flere grenseverdier. Slik at den kan ta en start- og slutt grenseverdi ut fra en hvilken som helst streng, for så å returnere leddet i midten. Den andre strengsøkealgoritmen har blitt utviklet til å erstatte bestemte bokstaver fra en streng, som vil hjelpe med å fjerne unødvendige bokstaver fra enkelte strenger.

Grunnen til at det ble utviklet strengsøkealgoritmer er fordi informasjonen hentet fra *roscpp* var ustrukturert i henhold til node objekt strukturen. Med det så ble det vanskelig å finne de forskjellige typene med variabler, som skulle bli lagret til node objektene. Uten å ha en form for filtrering og organisering av data.

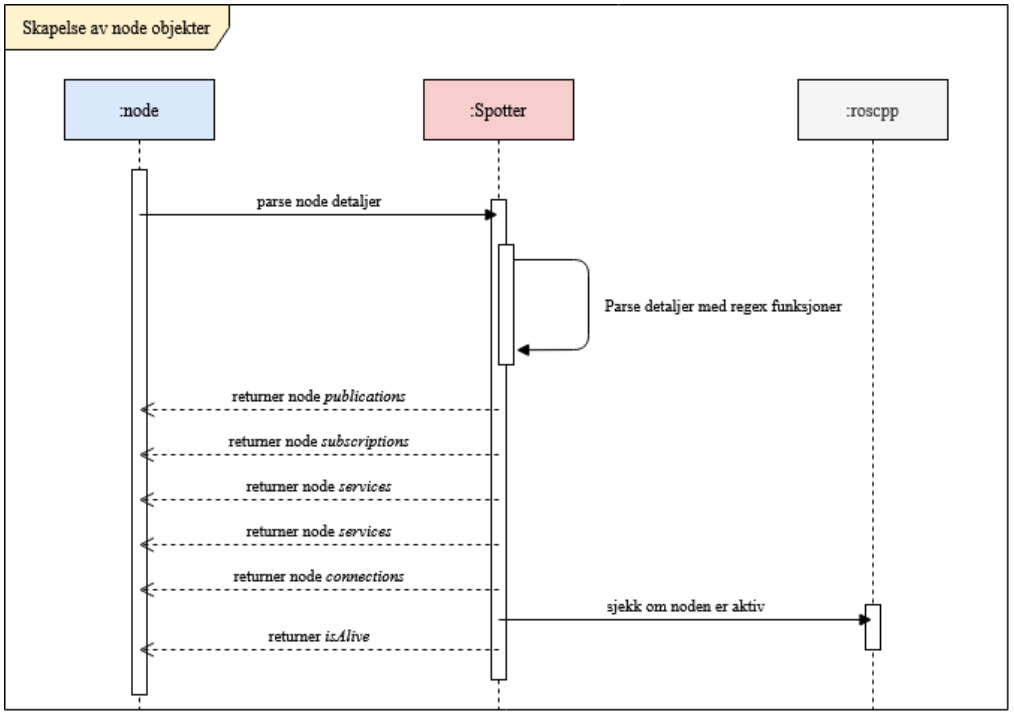
For å ta dette i bruk har Spotter-klassen en funksjon som gjør det mulig å hente ut en liste over alle ROS nodene i systemet. Deretter blir en funksjon fra Spotter-klassen kalt for å hente ut informasjon om ROS nodene gjennom *roscpp*. Som til slutt vil bli kjørt gjennom strengsøkealgoritmen basert på forskjellige grenseverdier som har blitt definert i node objektene sin konstruktør.

Når dette skjer, vil variablene til node objektene bli fylt med informasjon om ROS nodene og holdt oppdatert en gang i sekundet. Disse kan da bli tatt i bruk i andre deler av systemet ved enkle funksjonkall.

Sekvensdiagrammene fra figur 5.2 og 5.4 viser en abstrakt oversikt over hvordan arkitekturen til den dynamiske dataoversikten ser ut.



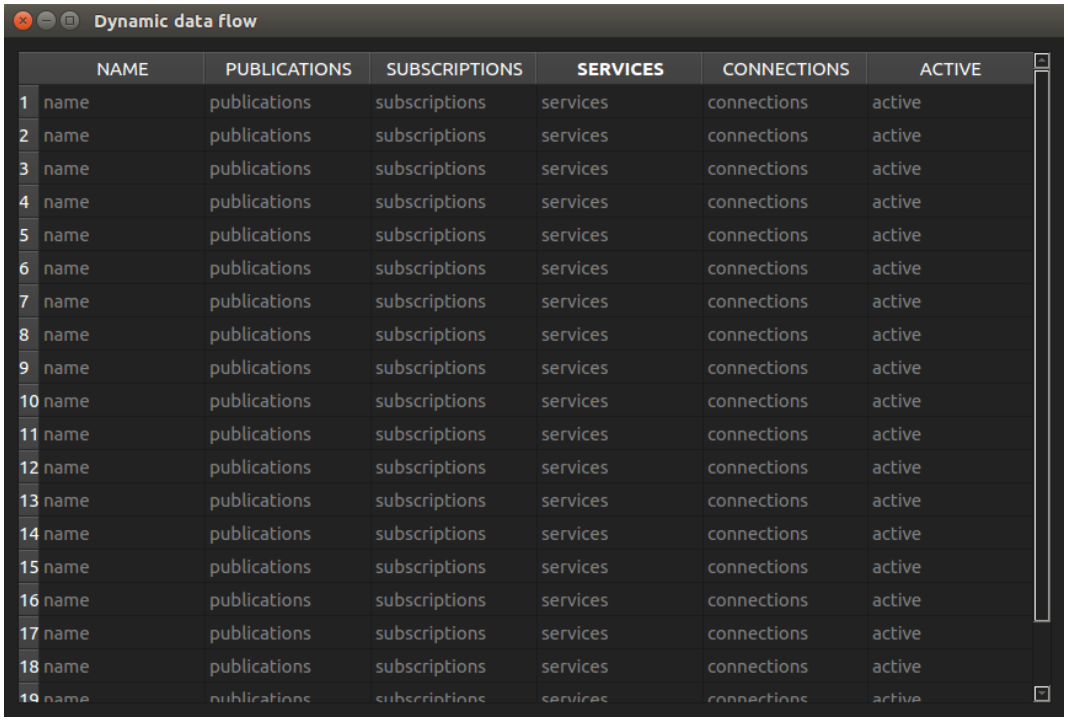
Figur 5.3: Sekvensdiagram for hvordan Spotter ROS noden lager objekter



Figur 5.4: Sekvensdiagram for hvordan node objekter blir skapt

5.4 Brukergrensesnitt

Tabell 5.2: QT applikasjon med grafisk tabell for dataoversikt



Brukergrensesnittet, sett fra tabell 5.2, består av 6 kolonner og et generert antall med rader. Antallet vil bli automatisk skapt ut fra størrelsen på listen fra Spotter funksjonen. Deretter blir radene oppdatert kontinuerlig. Designet har blitt tilpasset Coastal Shark sitt mørk modus tema. Kolonnene inneholder:

- **Name**: Navnet på noden.
- **Publications**: ROS Publications og meldingstype assosiert med noden.
- **Subscriptions**: ROS Subscriptions og meldingstype assosiert med noden.
- **Services**: ROS Services assosiert med noden.
- **Connections**: ROS topic fra til, retning og transportlag.
- **Active**: boolsk verdi (true/false) på om noden er aktiv.

5.5 Konklusjon

Prosessene med å ta frem en dataoversikt, gjøre den dynamisk og fremstille den i et brukergrensesnitt har for det meste vært suksessfult. Det eneste unntaket er forbindelsen mellom C2 sin Qnode og Spotter ROS noden. Hvor det er slik at brukergrensesnittet for dataoversikten er integrert inn i C2 sitt GUI, men funksjonaliteten som binder brukergrensesnittet og Spotter ROS noden sammen ikke er etablert.

Derimot har Spotter ROS noden blitt gitt funksjonaliteten til å konstruere og oppdatere en .csv fil som har blitt kalt dataflow.csv. Denne filen vil ha et likt oppsett som brukergrensesnittet, men vil ikke kunne nåes gjennom C2 sitt GUI. Istedenfor dette, vil filen kunne bli åpnet med et regneark verktøy som for eksempel Excel.

Et forslag for fremtidig arbeid vil dermed være å knytte C2 sin Qnode og Spotter ROS noden. I tillegg til dette, kan det være verdt å se mer inn på hvordan prioriteringssystemet kan bli integrert inn i den dynamiske dataoversikten.

Her kan det også for eksempel bli sett mer inn på hvordan dataoversikten kan gå fra en tabell, til å bli transformert inn i en graf sammen med prioriteringen. Hvor en operatør vil både ha full oversikt og kontroll over alle nettverksforbindelsene i systemet. Ved at grafen tillater enkel navigering og markeringer gjennom et teknisk men fortsatt brukervennlig brukegrensesnitt.

Dette er noe som også kan bli utvidet ytterligere, ved for eksempel at grafen inkluderer slikt som injeksjon tester, et loggesystem med tilpassbare opptak, testsett, søkefunksjoner og andre typer verktøy. Med det så vil den dynamiske dataoversikten forhåpentligvis legge grunnlaget for noe som dette.

6. Prioriteringssystemet

Dette kapittelet forklarer den tekniske bakgrunnen til meldingsutveksling i ROS, og presenterer utfordringene med prioritering av meldinger og andre problemer som eksisterer i systemet. Til slutt blir løsningen og systemarkitekturen som gruppen endte opp med forklart i detalj.

Vedlegg A tilhører dette kapittelet.

For å kunne prioritere hvilke data som blir sendt må man på et punkt ha oversikt og kontroll over all dataen som blir sendt og hvor den kommer fra. Dette åpner opp for flere løsninger, noen som bygger på ROS og noen som fungerer utenom. Ved å utforske flere områder innenfor nettverksoverføring, så har det kommet frem fire løsningsforslag. Disse ligner på hverandre, men er implementert på fire forskjellige lag. Løsningsforslagene er forklart og sammenlignet med hverandre i vedlegg A. Der blir også løsningsvalget resonert og begrunnet.

Det første gruppen begynte med var å få en oversikt over hvordan ROS og dens meldingsutveksling fungerte. Dette ga en forståelse om hvilke data som blir sendt, hvordan den blir sendt, og mulige punkter i kjeden hvor prioritering av meldinger kan bli implementert. Ettersom all dataen fra systemet blir sendt gjennom ROS åpnet dette for muligheter å implementere prioriteringen rett i ROS.

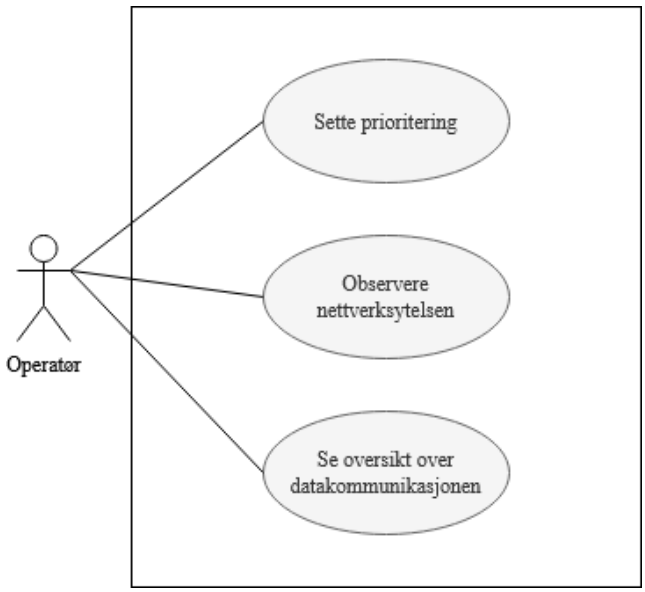
Gruppen prøvde å finne lignende løsninger, men ingen av disse tok for seg prioritering. De fleste tok for seg en form for justering av hvor ofte meldinger blir sendt. Et eksempel på dette er *dynamic bandwidth manager* (DBM), et rammeverk for ROS som prøver å maksimere bruken av tilgjengelig båndbredde [19]. Dette gjøres ved å regulere hvor ofte en node får lov til å sende meldinger – dette gjelder også internt i systemet. Ulempen med DBM er at den forhindrer at meldinger blir sendt, så det ville vært vanskelig å legge meldinger i en kø for å sende disse senere. Den krever også en instans per node i systemet, og tar ikke for seg hvordan meldingene blir overført til mottaker.

6.1 Problemstilling

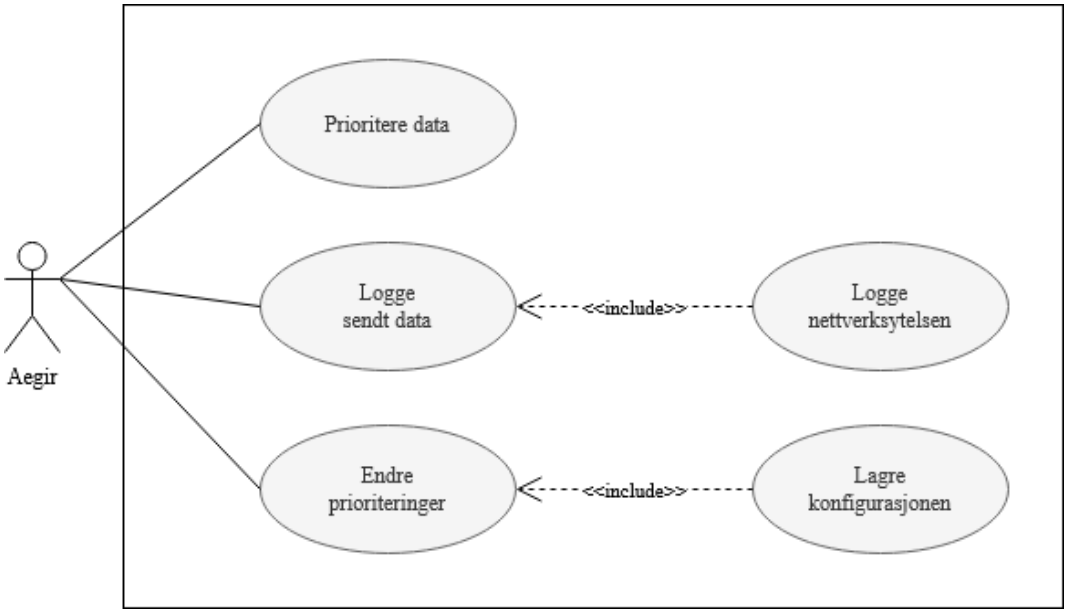
Et av kravene til oppgaven var å implementere et prioriteringssystem for datakommunikasjonen mellom C2 og Aegir. Dette innebærer å ta for seg all dataen som blir sendt, finne ut hva som er viktig, og til slutt bestemme hva som faktisk skal bli sendt mellom systemene. Systemet skulle være dynamisk slik at den dataen som blir sendt vil avhenge av den tilgjengelige båndbredden: dersom det er tilstrekkelig med båndbredde skal all data sendes, men hvis tilkobling blir dårlig og bitraten går ned, skal prioriteringssystemet kunne bestemme hva slags data som skal bli sendt først. Hva som skjer med den usendte dataen er også en del av oppgaven.

Et eksempel på dette er hvis det er en ekstremt dårlig tilkobling mellom C2 og Aegir, så ville det vært uønskelig å måtte vente flere minutter på at et stort bilde som ble tatt skal bli overført før noen annen data kan komme frem. Dermed skal prioriteringssystemet, etter operatørens ønsker, kunne favorisere noen typer data over andre slik at viktig informasjon kan komme frem.

Systemet skal også ha et brukergrensesnitt der operatøren skal kunne sette prioritet på de forskjellig meldingstypene. Figur 6.1 og 6.2 viser *use case* diagrammene for prioriteringssystemet.



Figur 6.1: *Use-case* diagram for operatøren.



Figur 6.2: *Use-case* diagram for Aegir.

6.2 Underliggende konsepter

Master og multimaster

En master i ROS er hovedsakelig ansvarlig for å holde styr på hvilke noder som finnes i systemet, og opprette koblinger mellom disse nodene dersom det er nødvendig. Hvis en node ønsker å kommunisere med en annen, må begge nodene registrere seg hos masteren som deretter vil opprette en direkte kobling mellom nodene. Meldingsutvekslingen vil dermed gå direkte fra en node til en annen, uten noen form for mellomledd.

Disse koblingene er vanlig TCP/IP sockets og bygger på ROS sin interne protokoll, kalt TCPROS. Det finnes støtte for UDP gjennom UDPROS, men denne er ikke vedlikeholdt, og – etter gruppens erfaring – fungerer ikke per dags dato.

Dersom systemet består av flere maskiner er det vanlig å bruke et multimasteroppsett. Da vil hver maskin ha sin egen ROS master som holder styr på sine lokale noder. Multimasteren er en applikasjon som prøver å synkronisere de forskjellige masterne slik at *topics* fra en maskin kan bli tilgjengeliggjort på andre maskiner. Ved å fortelle masteren at et *topic* av en viss type er tilgjengelig gjennom en viss IP og port, vil den kunne sette opp direktekoblinger på lik måte som med lokale *topics* og noder.

TCPROS

Meldingsutveksling er grunnlaget for ROS, og den skjer hovedsakelig over TCP. ROS har sin egen TCP-protokoll for dette som kalles TCPROS. Siden ROS var opprinnelig designet for systemer som opererer på samme maskin og/eller samme nettverk, så har protokollen en del overhead som gjør den uegnet for overføringer over dårlige forbindelser. Siden hver node vil ha en direktekobling til andre noder som den kommuniserer med, så krever det å opprette en TCP-kobling for hvert nodepar som ønsker å kommunisere med hverandre. I tillegg, siden *topics* er en enveis kommunikasjonskanal, vil det være nødvendig å sette opp to koblinger per node dersom toveiskommunikasjon er ønskelig.

Hver gang masteren setter opp en tilkobling mellom to noder vil den forhandle med de to nodene slik at de blir enig om hvilket format og hvilken protokoll meldingene skal sendes med. Dette skjer hver gang en tilkobling opprettes, og må gjøres på nytt dersom en eksisterende tilkobling droppes – noe som fort kan skje ved dårlig forbindelser. Det fører dermed til en del overhead, og kommer til å føre til forsinkelser i systemet.

Dette, kombinert med vanlige TCP *handshakes*, fører til at det totalt sett blir overført 50 forespørsler frem og tilbake kun for å opprette en enkel kobling mellom to noder, som sett i tabell 6.1.

Tabell 6.1: Forespørsler som blir sendt mellom maskinene for å opprette en kobling mellom to noder ved bruk av TCPROS.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	02:42:ac:11:00:03	Broadcast	ARP	42	Who has 172.17.0.2? Tell 172.17.
2	0.000038311	02:42:ac:11:00:02	02:42:ac:11:00:03	ARP	42	172.17.0.2 is at 02:42:ac:11:00:
3	0.000106824	172.17.0.3	172.17.0.2	TCP	74	45212 → 11311 [SYN] Seq=0 Win=64
4	0.000137790	172.17.0.2	172.17.0.3	TCP	74	11311 → 45212 [SYN, ACK] Seq=0 A
5	0.000194141	172.17.0.3	172.17.0.2	TCP	66	45212 → 11311 [ACK] Seq=1 Ack=1
6	0.000317260	172.17.0.3	172.17.0.2	HTTP/XML	396	POST /RPC2 HTTP/1.1
7	0.000332242	172.17.0.2	172.17.0.3	TCP	66	11311 → 45212 [ACK] Seq=1 Ack=33
8	0.002144205	172.17.0.2	172.17.0.3	HTTP/XML	433	HTTP/1.1 200 OK
9	0.002219523	172.17.0.3	172.17.0.2	TCP	66	45212 → 11311 [ACK] Seq=331 Ack=
10	0.003430481	172.17.0.3	172.17.0.2	TCP	66	45212 → 11311 [FIN, ACK] Seq=331
11	0.003638869	172.17.0.2	172.17.0.3	TCP	66	11311 → 45212 [FIN, ACK] Seq=368
12	0.003726938	172.17.0.3	172.17.0.2	TCP	66	45212 → 11311 [ACK] Seq=332 Ack=
13	0.172400165	172.17.0.3	172.17.0.2	TCP	74	45214 → 11311 [SYN] Seq=0 Win=64
14	0.172453239	172.17.0.2	172.17.0.3	TCP	74	11311 → 45214 [SYN, ACK] Seq=0 A
15	0.172543711	172.17.0.3	172.17.0.2	TCP	66	45214 → 11311 [ACK] Seq=1 Ack=1
16	0.172702349	172.17.0.3	172.17.0.2	HTTP/XML	622	POST /RPC2 HTTP/1.1
17	0.172726212	172.17.0.2	172.17.0.3	TCP	66	11311 → 45214 [ACK] Seq=1 Ack=55
18	0.175759882	172.17.0.2	172.17.0.3	HTTP/XML	571	HTTP/1.1 200 OK
19	0.175878668	172.17.0.3	172.17.0.2	TCP	66	45214 → 11311 [ACK] Seq=557 Ack=
20	0.178140429	172.17.0.3	172.17.0.2	HTTP/XML	478	POST /RPC2 HTTP/1.1
21	0.178163513	172.17.0.2	172.17.0.3	TCP	66	11311 → 45214 [ACK] Seq=506 Ack=
22	0.180074621	172.17.0.2	172.17.0.3	HTTP/XML	467	HTTP/1.1 200 OK
23	0.180157235	172.17.0.3	172.17.0.2	TCP	66	45214 → 11311 [ACK] Seq=969 Ack=
24	0.183346376	172.17.0.3	172.17.0.2	HTTP/XML	659	POST /RPC2 HTTP/1.1
25	0.183364174	172.17.0.2	172.17.0.3	TCP	66	11311 → 45214 [ACK] Seq=907 Ack=
26	0.185041548	172.17.0.2	172.17.0.3	HTTP/XML	525	HTTP/1.1 200 OK
27	0.185127417	172.17.0.3	172.17.0.2	TCP	66	45214 → 11311 [ACK] Seq=1562 Ack
28	0.186872952	172.17.0.3	172.17.0.2	HTTP/XML	664	POST /RPC2 HTTP/1.1
29	0.186889853	172.17.0.2	172.17.0.3	TCP	66	11311 → 45214 [ACK] Seq=1366 Ack
30	0.188515144	172.17.0.2	172.17.0.3	HTTP/XML	530	HTTP/1.1 200 OK
31	0.188591840	172.17.0.3	172.17.0.2	TCP	66	45214 → 11311 [ACK] Seq=2160 Ack
32	0.190194164	172.17.0.3	172.17.0.2	TCP	74	45216 → 11311 [SYN] Seq=0 Win=64
33	0.190231132	172.17.0.2	172.17.0.3	TCP	74	11311 → 45216 [SYN, ACK] Seq=0 A
34	0.190282847	172.17.0.3	172.17.0.2	TCP	66	45216 → 11311 [ACK] Seq=1 Ack=1
35	0.190384419	172.17.0.3	172.17.0.2	HTTP/XML	403	POST /RPC2 HTTP/1.1
36	0.190396795	172.17.0.2	172.17.0.3	TCP	66	11311 → 45216 [ACK] Seq=1 Ack=33
37	0.193735929	172.17.0.2	172.17.0.3	HTTP/XML	1005	HTTP/1.1 200 OK
38	0.193813288	172.17.0.3	172.17.0.2	TCP	66	45216 → 11311 [ACK] Seq=338 Ack=
39	0.195359117	172.17.0.3	172.17.0.2	TCP	66	45216 → 11311 [FIN, ACK] Seq=338
40	0.195509759	172.17.0.2	172.17.0.3	TCP	66	11311 → 45216 [FIN, ACK] Seq=946
41	0.195584531	172.17.0.3	172.17.0.2	TCP	66	45216 → 11311 [ACK] Seq=339 Ack=
42	0.196004580	172.17.0.3	172.17.0.2	HTTP/XML	478	POST /RPC2 HTTP/1.1
43	0.197160292	172.17.0.2	172.17.0.3	HTTP/XML	467	HTTP/1.1 200 OK
44	0.197233630	172.17.0.3	172.17.0.2	TCP	66	45214 → 11311 [ACK] Seq=2572 Ack
45	0.198609721	172.17.0.3	172.17.0.2	HTTP/XML	483	POST /RPC2 HTTP/1.1
46	0.200137153	172.17.0.2	172.17.0.3	HTTP/XML	472	HTTP/1.1 200 OK
47	0.201717939	172.17.0.3	172.17.0.2	HTTP/XML	618	POST /RPC2 HTTP/1.1
48	0.203386618	172.17.0.2	172.17.0.3	HTTP/XML	526	HTTP/1.1 200 OK
49	0.240140779	02:42:ac:11:00:03	Broadcast	ARP	42	Who has 172.17.0.1? Tell 172.17.
50	0.249781435	172.17.0.3	172.17.0.2	TCP	66	45214 → 11311 [ACK] Seq=3541 Ack

En annen ulempe er overføringen av dupliserte meldinger. Dersom to noder fra et system får meldinger fra en node fra et annet system, så vil meldingen sendes to ganger – en for hver *subscriber*. Dette kaster bort båndbredde som kunne ha blitt brukt til andre ting, og er strengt tatt unødvendig: dataen trenger kun å sendes en gang, og kan heller bli duplisert på mottakeren sin side.

6.3 Løsning

Ved å erstatte multimaster-oppsettet med en lignende tjeneste som også tar for seg overføringen av meldinger, vil det være mulig å forhindre problemene som meldingsutvekslingen i ROS introduserer. Siden systemet dermed tar for seg overføringen av meldinger mellom de forskjellige maskinene, vil den til enhver tid ha full kontroll av hvilke meldinger som blir sendt og hvordan de blir sendt. Dette åpner dermed opp for å implementere prioritering av meldinger.

Ideen er dermed at utgående meldinger kan bli lagt til i en kø, og når senderen får mulighet, kan den hente en melding ut av denne køen. Hvordan denne køen blir sortert vil avhenge av prioriteten til de individuelle meldingene.

Hovedfokuset med denne løsningen er å gi systemet mulighet til å overføre meldinger med UDP for å forhindre *handshakes* og bekreftelser, samt eliminere behovet for overføringen av dupliserte meldinger. I tillegg, siden systemet vil både være ansvarlig for å sende og motta pakker, vil det ikke være behov for å forhandle mellom partene om hvilket format eller protokoll de skal bruke da dette er fastsatt ved compile-time.

Systemet kommer fortsatt til å støtte TCP siden det kan være ønskelig for å passe på at viktig data kommer frem i sin helhet. Istedenfor å opprette en TCP-kobling mellom hver node, vil det kun opprettes en kobling mellom sender og mottaker. All data som skal bli overført mellom to systemer vil dermed bli sendt over denne koblingen. TCP *handshakes* vil fortsatt være nødvendig, men her trengs det heller ikke å forhandle om format eller protokoll.

For å gi operatøren mest mulig kontroll vil det være mulig å la operatøren bestemme hvilken protokoll som skal bli brukt for hvert *topic*. Det vil si at data som ansees som viktig kan sendes med TCP, mens mindre viktig data kan sendes med UDP. Sistnevnte tar ikke for seg retransmisjon av tapte pakker, men er noe som kan bli implementert senere dersom det er ønskelig.

Her er det kun data som blir sendt fra Aegir som blir prioritert. All data som kommer fra C2 ansees som viktig, og blir derfor ikke prioritert. Dersom det er ønskelig å implementere prioritering fra C2 sin side så er det tatt høyde for dette i systemarkitekturen, slik at det enkelt kan bli lagt til.

Systemet vil fungere som en multiplekser i den forstand at den tar data fra flere kilder – i dette tilfelle noder – og sender det over en enkel link. Mottakeren vil ta i mot dataen og distribuere den til riktig plass.

6.3.1 Kjernebuffer og prioritering

Linux tilbyr diverse abstraksjoner for socketprogrammering. Blant disse er *send* og *sendto*, som brukes til å sende data over TCP og UDP henholdsvis.

Som en del av kernelen finnes det et internt nettverksbuffer for hver socket som er åpen. Når de sistnevnte abstraksjonene blir brukt, så legges meldingen til i dette bufferet. Fra kallerens side vil det se ut som at meldingen har blitt sendt, selv om den fortsatt ligger i bufferet. Når bufferet er fullt, så vil fremtidige *send*-funksjonskall bli blokkerende – det vil si at programmet som sender data ikke får lov til å fortsette før det er plass i bufferet [20]. Dette er grunnlaget for hvordan prioriteringen skjer: hver gang programmet får lov til å sende en melding, så vil den velge den meldingen som ansees som viktigst der og da.

Et problem er at dette fungerer kun for TCP. I motsetning til *send*, så vil *sendto* forkaste nye meldinger som blir sendt dersom bufferet er fullt [20]. For å unngå dette vil man regelmessig sjekke tilstanden til dette bufferet for å finne ut hvor mye ledig plass det har. Først når bufferet har plass til en melding vil man hente en melding fra prioritetskøen.

Størrelsen på bufferet vil dermed være proporsjonal med unøyaktigheten til prioriteringen. Så fort en melding har blitt lagt til i bufferet så kan den ikke fjernes.

Det medfører at en melding av lavere prioritet kan ha blitt lagt til i bufferet før en melding av høyere prioritet kom fram i køen uten at dette kan endres. Størrelsen på bufferet kan derimot konfigureres manuelt, og har en standardstørrelse på 16 kB [21].

6.3.2 Jetson-kortet

Ombord på Aegir finnes det to maskiner: et Jetson-kort og en Intel *Next Unit of Computing* (NUC). Per i dag er disse koblet direkte opp til radiosambandet mellom C2 og Aegir. Dette skaper noen problemer med tanke på prioritering av meldinger. Hvis begge maskinene har fri tilgang til radiosambandet vil det ikke være mulig å garantere noen form for prioritering, da en melding av lavere prioritet fra en maskin kan bli sendt før en annen melding av høyere prioritet fra en annen maskin.

En mulighet var å introdusere en form for kommunikasjon mellom systemene for å bestemme hvilken maskin som skulle få tilgang til radioen til enhver tid. Etter å ha diskutert mulighetene med oppdragsgiver ble det enighet om å dirigere meldingene fra Jetson-kortet gjennom NUC-en. Dermed kommer kun NUC-en til å være koblet til radiosambandet. Den enkleste måten å få dette til er å sette opp "relay"-noder som henter data fra *topics* fra en maskin og republiserer de på den andre. Kommunikasjonssystemet som kjører på NUC-en vil dermed kunne ta denne dataen og sende det til C2. Relays finnes allerede i standardbiblioteket til ROS.

Et annet alternativ er å sette opp disse maskinene slik at de har en felles ROS master. Da vil det ikke være et behov for relay noder siden de vil ha direkte tilgang til hverandres *topics*.

6.3.3 Sjekksummer

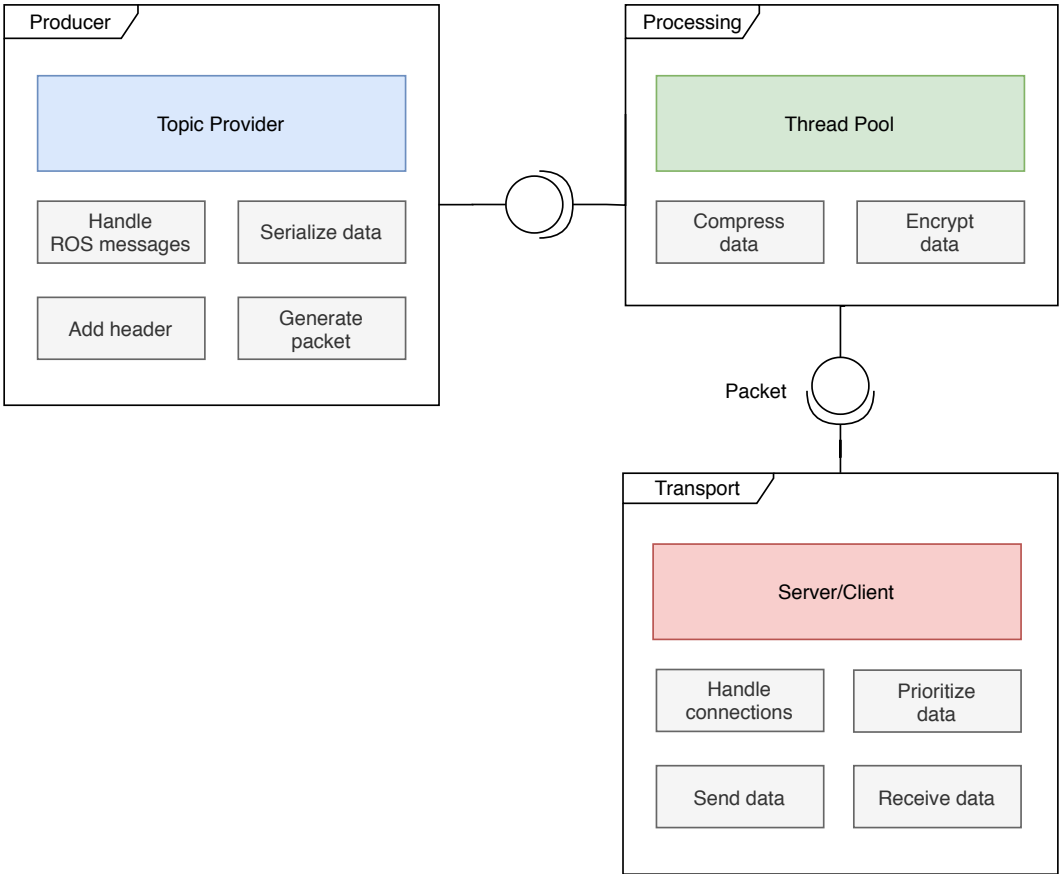
Et av kravene var å implementere kryptering og/eller sjekksummer for hver melding som blir sendt. Kryptering har blitt implementert og er beskrevet i 6.5.6, men sjekksummer blir overlatt til transportlaget. Både TCP og UDP har støtte for sjekksummer. Bruken av sjekksummer for UDP pakker er valgfritt for IPv4 og må aktiveres eller deaktiveres gjennom operativsystemet. Dersom sjekksummen til UDP-pakken ikke stemmer vil kun den pakken bli forkastet.

6.4 Arkitektur

Systemet bruker en komponentbasert arkitektur, som sett i figur 6.3. Programmet er delt opp i flere komponenter, der hver komponent gjør kun en ting. Dette er for å introdusere en separasjon mellom delene slik at det er tydelig hvilken rolle hver komponent har, og fungerer som en skille mellom de forskjellige lagene. Siden funksjonaliteten er delt opp og fokusert vil det være lettere å gå inn og endre på ting i senere tider. Hver komponent er løst koblet sammen med hverandre dersom det er nødvendig. Mellom hver komponent er det definert et interface slik at komponentene kan enkelt byttes ut dersom det er ønskelig.

Komponentene er designet som mindre, generiske biblioteker som opprinnelig ikke har noen sammenkobling med hverandre. Intensjonen var at hver komponent skulle være selvstendig. For å få til det, vil hver komponent bli koblet sammen gjennom callbacks og delte køer. I noen tilfeller brukes det *polling* for å sjekke tilstanden til køene.

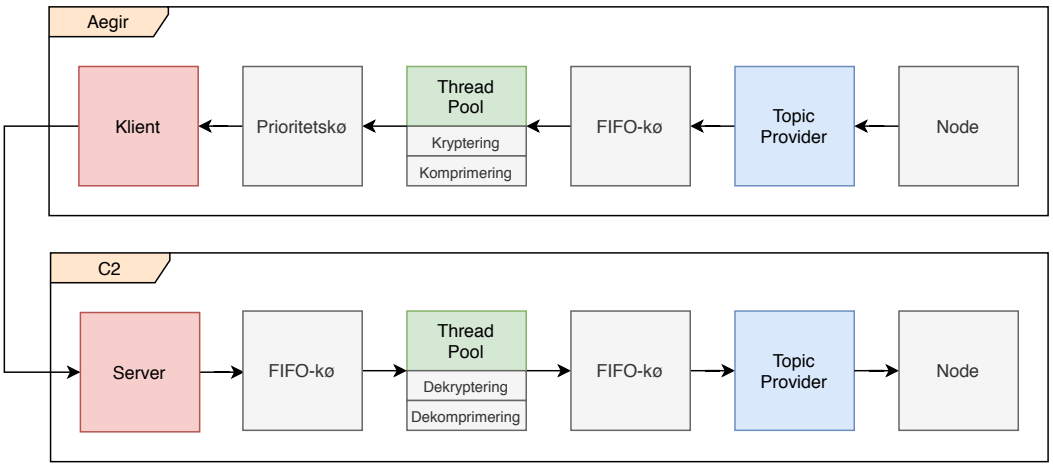
Det finnes tre hovedkomponenter i systemet. Den første komponenten er en produsent, som er ansvarlig for å generere data som serveren skal sende. Den andre komponenten er en behandler, som vil manipulere dataen fra produsenten i form av kryptering og komprimering. Den siste komponenten er ansvarlig for selve transporten av dataen.



Figur 6.3: Komponentgrafen til systemet.

Topic Provider er ansvarlig for å interagere med ROS, og konvertere ROS-meldinger til rådata som kan bli overført over nettverk; trådbassenget vil kryptere og komprimere inngående og utgående data; og serveren og klienten vil ta imot og sende pakker mellom hverandre. Dette er gjort for å modularisere systemet slik at det er enkelt å sette seg inn i koden, og for å gjøre koden så gjenbrukbar som mulig. Eksempelvis kan hele trådbassenget bli fjernet uten at det vil stoppe systemet fra å fungere.

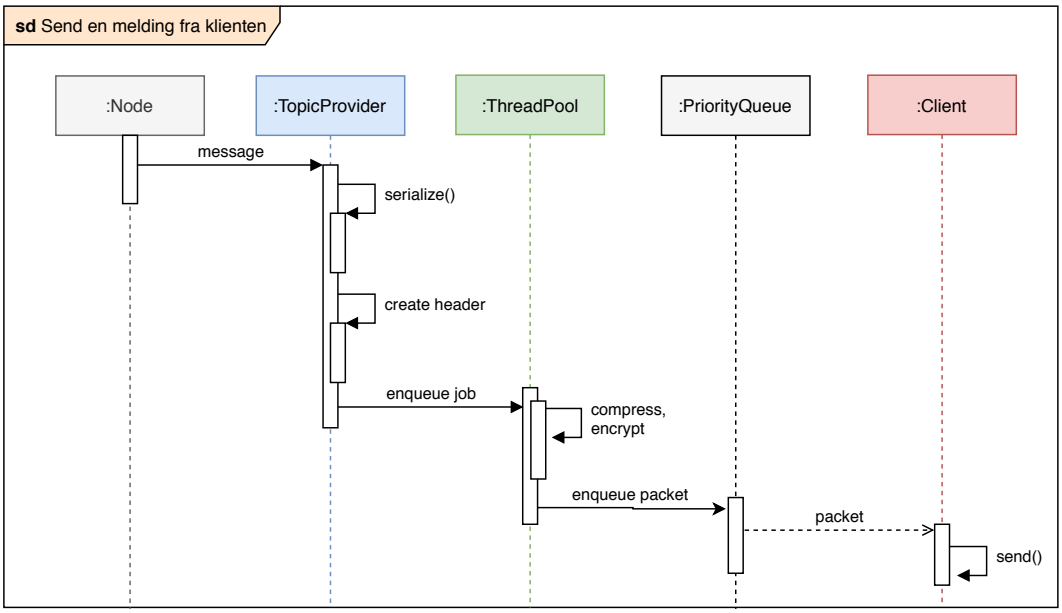
For at hver maskin skal kunne kommunisere med hverandre trengs det en sender og mottaker på hvert system. Her er det brukt en klient-server arkitektur: du har en eller flere klienter som kobler seg opp mot en server. Siden ansvaret til C2 er å være et kontrollsystem for Aegir, så vil den være ansvarlig for å kjøre serveren. Aegir vil kjøre en klientapplikasjon som vil koble seg opp mot serveren. Kommunikasjonssystemet støtter dermed flere klienter slik at prosjektet har muligheten til å bli utvidet senere.



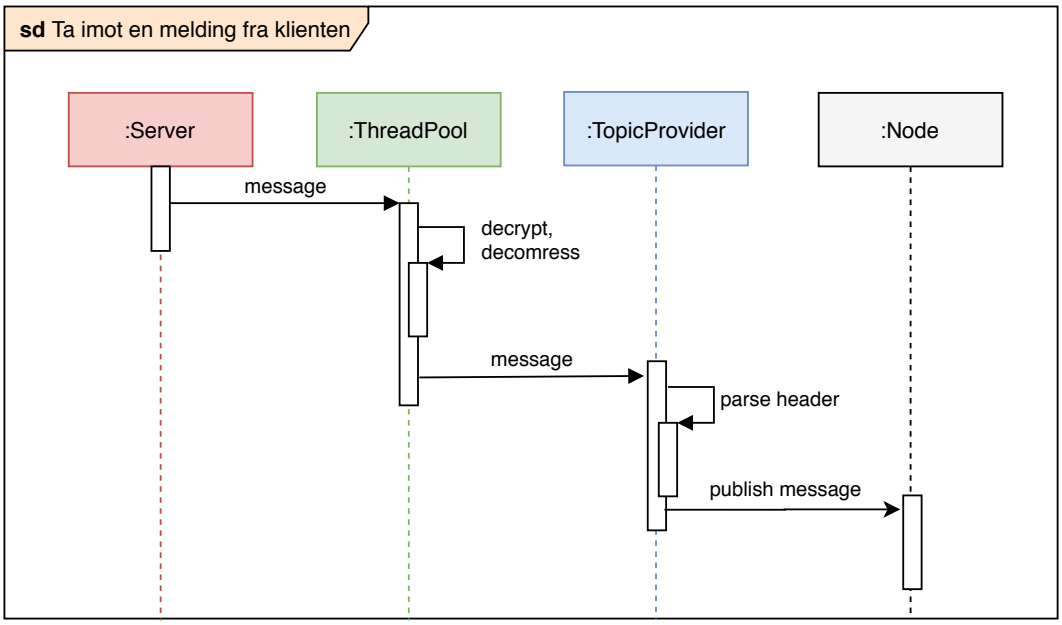
Figur 6.4: Flytskjema for overførselen av en ROS-melding fra Aegir til C2.

Serveren tar kun for seg det å lese og skrive over sockets. Den prøver ikke tolke eller dirigere dataen; den leser til en kø, og skriver fra en annen kø. På grunn av det asynkrone opphavet til systemet vil funksjonskall mellom komponentene aldri returnere noen verdier siden de opererer uavhengig av hverandre. Funksjonskallene vil heller ikke være blokkerende med mindre de venter på en lås for å få tilgang til en av køene.

Figur 6.4 viser prosessen for hvordan en melding blir sendt fra Aegir til C2. Prosessen for å sende og motta meldinger er ganske lik, med unntak at meldinger som blir mottatt hos Aegir ikke kommer til å bli prioritert – de vil derimot bli lagt i en vanlig FIFO-kø som trådbassenget har tilgang til. Figur 6.5 og 6.6 viser sekvensdiagrammene for hvordan en klient vil sende og ta imot data.



Figur 6.5: Sekvensdiagram for hvordan en melding blir sendt.



Figur 6.6: Sekvensdiagram for hvordan en melding blir mottatt.

6.4.1 Konfigurering

Operatøren skal kunne endre på konfigurasjonen til prioriteringen gjennom et brukergrensesnitt. For å redusere mest mulig "proprietær" kommunikasjon mellom klienten og serveren ble det valgt å bruke vanlig ROS *topics* og meldinger for å konfigurere systemet. Dette skjer gjennom en selvstendig konfigurasjonsnode som kjører på hver maskin. Klienten vil abonnere på dette *topic*-et slik at den alltid vil ta imot disse meldingene. Dersom konfigurasjonsnoden ikke kjører, så vil systemet operere med forhåndsdefinerte innstillinger.

Brukergrensesnittet trenger derfor kun generere ROS-meldinger for å konfigurere kommunikasjonssystemet.

6.4.2 Topic-synkronisering

For å gjøre det lettere å sette opp og videreutvikle systemet vil klienten og serveren automatisk dele hvilke *topics* som er tilgjengelige på hver maskin. Denne funksjonaliteten ble sammensveiset med konfigurasjonsnoden. En liste med *topics* vil bli hentet fra en maskin og overført til den andre, der de vil bli tilgjengeliggjort. Først når en node prøver å *subscribe* til et av disse vil dataoverføringen av meldingene starte.

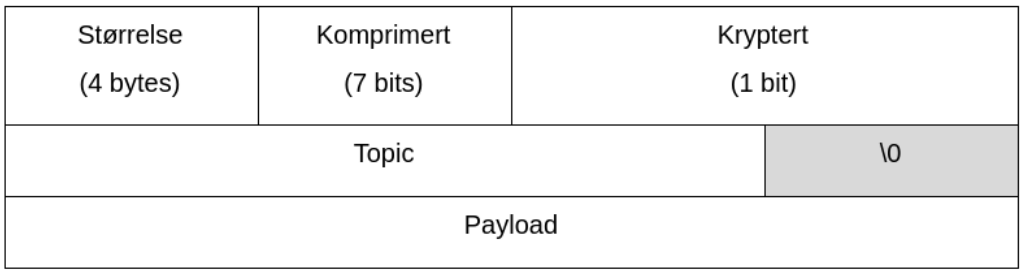
I tillegg støttes det et statisk oppsett, hvor hvilke *topics* som skal bli overført er forhåndsbestemt. Dette krever manuell konfigurering på både sender og mottaker sin side før oppstart av systemet.

6.4.3 Protokoll

Hver melding som blir sendt mellom serveren og klienten blir prefikset med en header, som vist i figur 6.7. Det finnes kun en type melding som blir overført mellom systemene. Headeren inneholder følgende:

- **Størrelse**: Størrelsen på meldingen, oppgitt i bytes. Størrelsen inkluderer ikke sitt eget felt.
- **Komprimert**: En 7-bits verdi som tilsier hvilken komprimeringsalgoritme som ble brukt.
- **Kryptert**: 1-bit verdi som sier om meldingen er kryptert.
- **Topic**: Navnet på *topic*-et som meldinger gjelder. Denne avsluttes med en null-verdi for å skille mellom navnet og dataen som kommer etter.
- **Payload**: ROS-meldingen som blir overført. Denne er av variabel størrelse, og blir avgrenset av størrelsen som ble oppgitt i headeren.

Størrelsen på de første tre feltene er statiske.



Figur 6.7: Meldingsformatet som systemet bruker.

6.5 Implementasjon

Her blir det forklart hvordan de forskjellige delene av systemet ble implementert. Både serveren og klienten tar i bruk *Boost ASIO*-biblioteket, som tilbyr abstraksjoner for socketprogrammering. ASIO tilbyr asynkrone I/O-operasjoner, men disse har ikke blitt tatt i bruk.

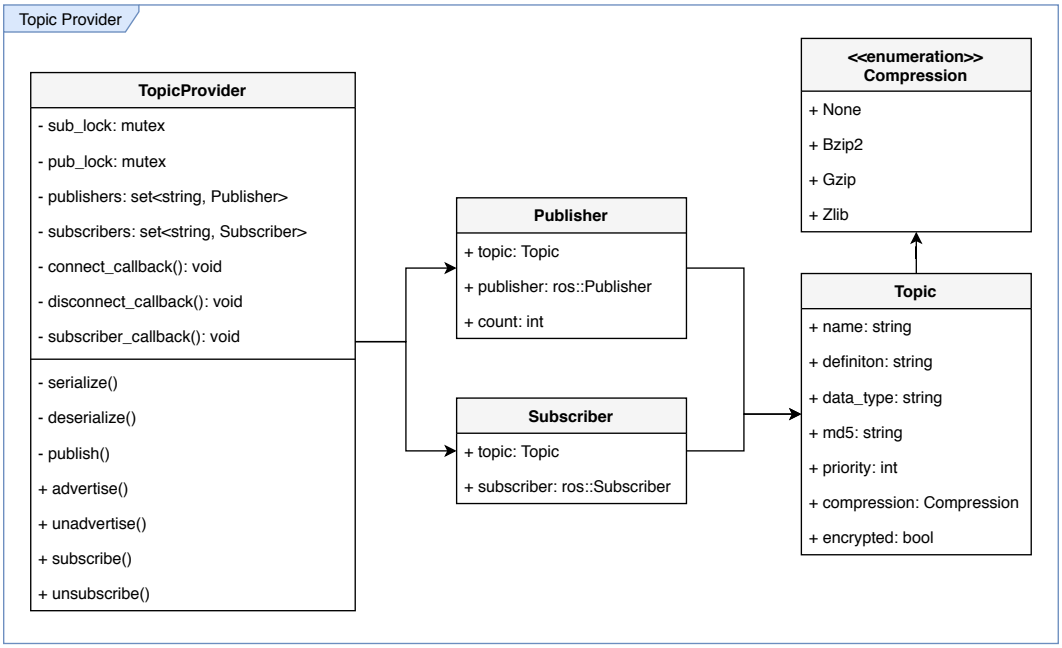
6.5.1 Topic Provider

Oppgaven til TopicProvider-klassen er å interagere med ROS. Den er hovedsakelig ansvarlig for å *subscribe* til ønskede *topics* slik at den kan ta imot interne meldinger, og publisere innkommende meldinger fra andre systemer slik at de blir tilgjengeliggjort for andre ROS noder.

For hver melding som skal bli sendt vil den serialisere innholdet og lage en pakke ut av det. Denne pakken inneholder informasjon om hvilket *topic* meldinger gjelder, hvilken prioritet den har, og om meldingen er kryptert og/eller komprimert.

ROS prøver å utnytte typesystemet til C++ for å legge til en slags sikkerhet slik at kun meldinger av riktig type kan sendes over et *topic*. Denne typen må normalt være fastdefinert ved kompileringstid, siden både *Publisher*- og *Subscriber*-objektene til ROS bruker templates av meldingstypen for å bli instansiert.

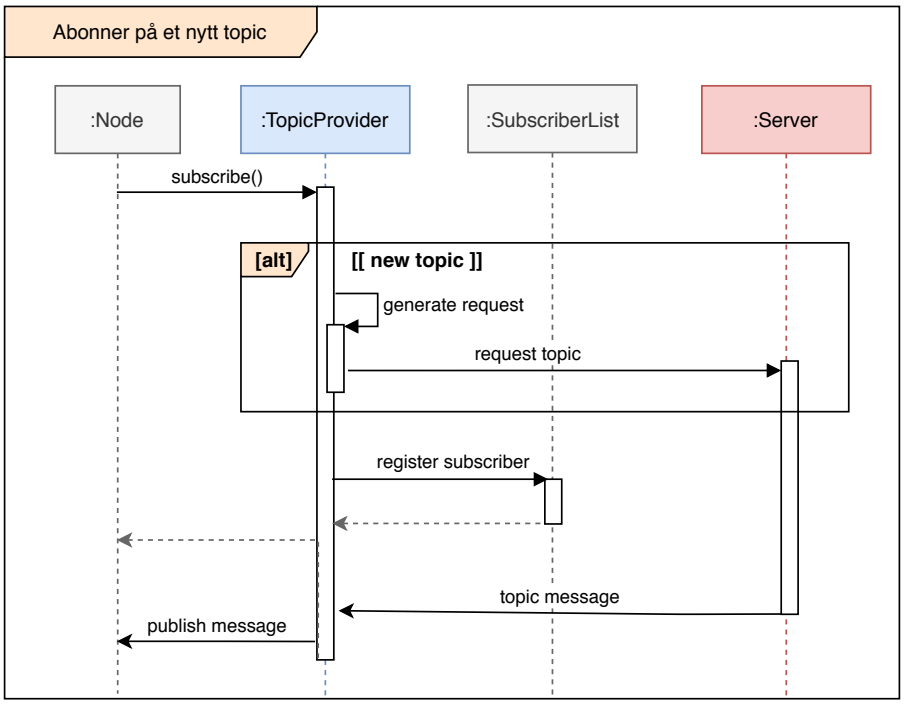
En meldingstype består av tre deler: meldingsdefinisjonen, navnet på meldingstypen, og en MD5-sjekksum av meldingsdefinisjonen. For at systemet skal kunne fungere uavhengig av hva slags type data blir sendt, så ble *ShapeShifter*-klassen fra standardbiblioteket i ROS tatt i bruk. Den gjør det mulig å lage nye meldingstyper i sanntid, som gjør at systemet kan *subscribe* til hvilket som helst *topic* uten å vite typen på forhånd.



Figur 6.8: Klassediagram for TopicProvider.

TopicProvider-klassen inneholder en liste med ROS *publishers* og en liste med aktive *subscribers*, som sett i figur 6.8. Hver gang et topic som klassen abonnerer på får en melding, vil et *callback* bli eksekvert som starter prosessen å lage en pakke ut av meldingen.

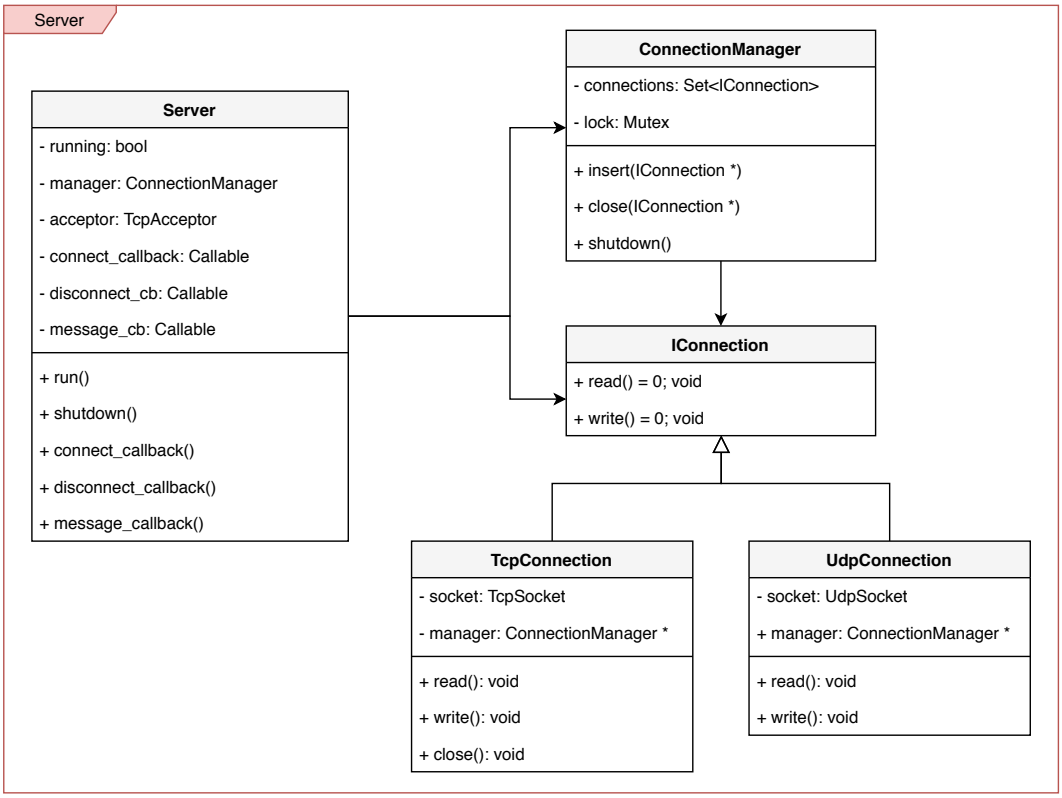
Hvert *topic* som blir tilgjengeliggjort av *TopicProvider*-klassen vil være koblet til en *callback*-funksjon som blir eksekvert hver gang en node velger å starte å motta meldinger av denne typen. Ved å holde styr på hvor mange noder som abonnerer på *topic*-et er det mulig å automatisk starte og stoppe overføringen av dataen som blir publisert over dette *topic*-et.



Figur 6.9: Sekvensdiagram for registrering av en ny *subscriber*.

Sekvensdiagrammet i figur 6.9 viser prosessen for å *subscribe* til et nytt *topic*. Dersom meldinger av denne typen allerede blir mottatt vil noden kun bli lagt til som en mottaker av innkommende meldinger. Hvis det derimot er en ny meldingstype vil klassen sende en melding til klienten – gjennom serveren – for å be den om å starte overføringer av disse meldingene.

6.5.2 Server



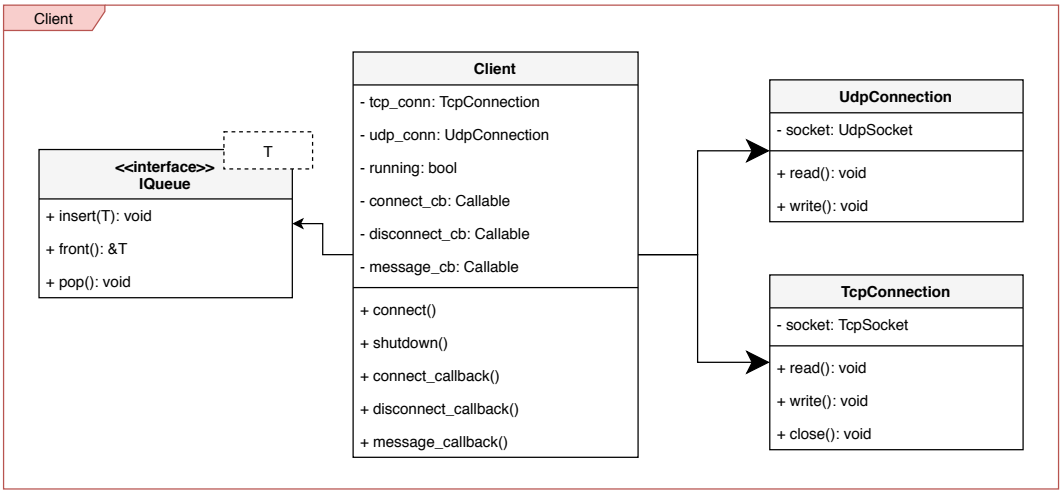
Figur 6.10: Klassediagram for serveren.

Oppgaven til serveren er å kun ta imot og sende meldinger, og holde styr på aktive forbindelser. Så fort en melding har blitt tatt i mot, så vil den legge meldingen i en kø som blir hentet av trådbassenget. Serveren vil regelmessig lese fra en annen kø for utgående meldinger.

Serveren vil lage to nye tråder for hver TCP-forbindelse som blir opprettet, slik at den kan håndtere flere klienter til enhver tid og både ta imot og sende data samtidig.

Selv om UDP er en forbindelsesløs protokoll, så er det en mulighet å lagre endepunktet til mottaker slik at man kan behandle det som en forbindelse. Sender vil derimot ikke vite om det finnes en mottaker på andre enden, eller om meldingen kommer frem.

6.5.3 Klient



Figur 6.11: Klassediagram for klienten.

Klienten er en forenklet versjon av serveren. Den vil kun prøve å etablere en tilkobling til serveren. Dersom tilkobling mislykkes eller blir frakoblet vil den regelmessig prøve å etablere en ny tilkobling.

I likhet med serveren bruker den callbacks hver gang en tilkobling blir opprettet, hver gang en melding mottas, og hver gang tilkoblingen droppes.

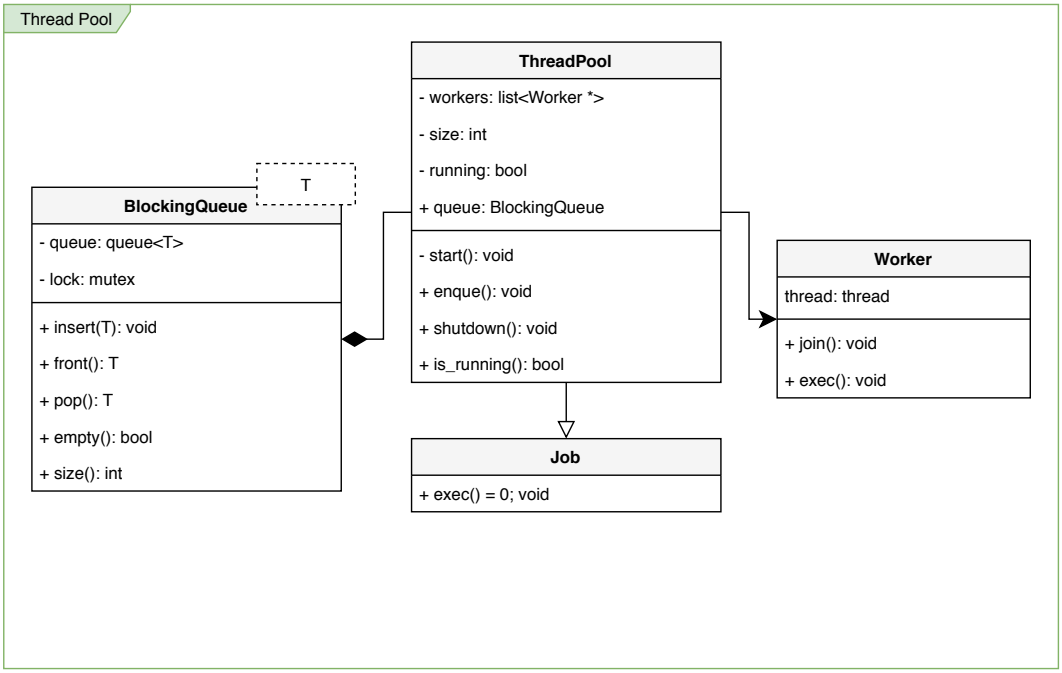
Klient-klassen inneholder en kø som er hvor selve prioriteringen skjer. Denne er nærmere forklart i avsnitt 6.5.7.

6.5.4 Trådbasseng

Komprimering og kryptering av større filer kan være tidkrevende. Dette kan innføre en del forsinkelser hos mottaker og sender, siden hver pakke må krypteres/dekrypteres og komprimeres/dekomprimeres. For å unngå slike forsinkelser, som kan føre til pakketap, brukes det et trådbasseng. Et trådbasseng er et sett med tråder som blir laget kun en gang, der hver tråd vil, så fort den har mulighet, hente en jobb

fra en kø og utføre den jobben. Etter det vil prosessen gjentas. Å lage en tråd er en ganske dyr prosess, som er grunnen til at et trådbassenget blir brukt istedenfor. Antall tråder som skal brukes kan bli spesifisert manuelt, men vil ellers bruke en tråd per fysiske kjerne.

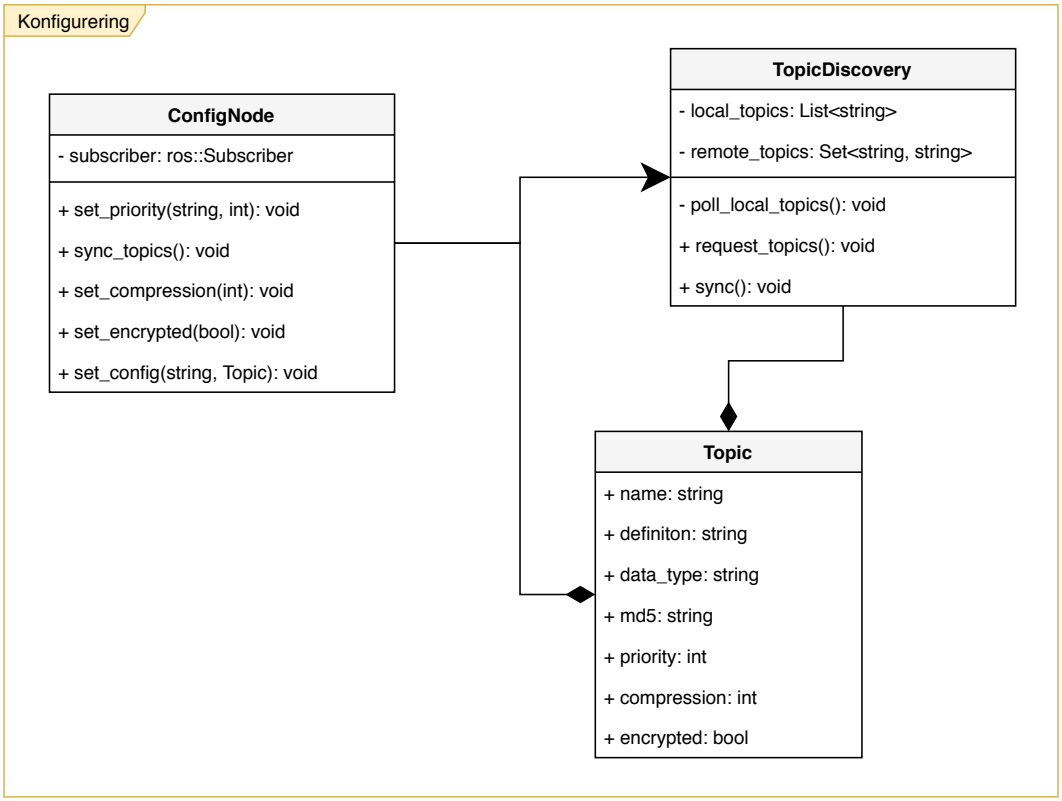
Hver melding som skal bli sendt, og de som har blitt mottatt, kommer derfor til å bli lagt til i en kø som trådbassenget har tilgang til. Ut fra den køen vil en melding hentes, krypteres/dekrypteres og komprimeres/dekomprimeres, og deretter vil de bli publisert til riktig *topic* eller sendt til mottaker. Alt dette vil skje fra trådbassenget, og ansees som en jobb fra dens perspektiv.



Figur 6.12: Klassediagram for et trådbassenget.

6.5.5 Konfigurering

Dersom operatøren gjør endringer og velger å lagre innstillingene vil en ROS melding bli generert i bakgrunn som blir sendt over til klienten. Der vil den tas i mot som en del av TopicProvider-klassen, som vil lese innholdet og lagre konfigurasjonen.



Figur 6.13: Klassediagram for konfigureringsnoden.

ROS parameter server

Når systemet starter vil den først ta utgangspunkt i innstillingene som er satt via ROS sin parameter server. Disse blir lest fra en YAML-fil der ting som IP-addresse og port har blit fastsatt. Hvis et statisk oppsett er ønskelig, så må informasjonen om hvilke *topics* som skal overføres og lignende bli definert her, med tilhørende prioritet og transportmetode.

6.5.6 Komprimering og kryptering

Systemet støtter både komprimering og kryptering av meldinger som blir sendt, uavhengig av hvilken nettverksprotokoll som blir brukt. I første omgang ble det implementert SSL-støtte for koblingen mellom C2 og Aegir. Dette ble i senere tider fjernet, da SSL kun støtter TCP, og går i mot hensikten med å redusere antall *handshakes*. Hver gang tilkoblingen blir droppet ville det vært nødvendig med et nytt SSL handshake. For å unngå dette ble det valgt å kryptere individuelle ROS meldinger. Det er kun meldingen som blir komprimert – headeren sendes i ukryptert form da den vil aldri inneholde sensitiv informasjon. Meldingene krypteres med AES-256-CBC gjennom OpenSSL-biblioteket. AES ble valgt siden den vil ikke endre på størrelsen til meldingen. Systemet tar ikke for seg utveksling av kryptografiske nøkler; disse må på forhånd være utvekslet slik at begge systemene har tilgang til hverandres private nøkler.

Systemet støtter flere komprimeringsalgoritmer. Hvilke av disse som er best egnet vil avhenge av hva slags type data som blir sendt. Eksempelsvis er *lz4* en ekstremt rask og effektiv komprimeringsalgoritme, men fungerer kun for tekst som inneholder mye repetisjon [22]. Dersom den blir brukt for annen type data, for eksempel på bilder, vil den komprimerte versjonen være større enn den ukomprimerte. Valget av hvilken algoritme som brukes blir dermed overlatt til operatøren. Systemet bruker kun tapsfri komprimeringsalgoritmer for å unngå tap av data.

Dersom både kryptering og komprimering er aktivert, vil alltid komprimeringen skje først. Dette er fordi komprimering bruker statistisk redundans for å minske størrelsen, noe som gjør det vanskelig å komprimere kryptert data [23].

6.5.7 Prioriteringsalgoritme

Utgående meldinger blir lagt til i en prioritetskø hvor selve prioriteringen skjer. Hver gang klienten har mulighet til å sende en melding – det vil si så fort det er plass i kjernebufferet – så vil den hente en melding ut fra køen.

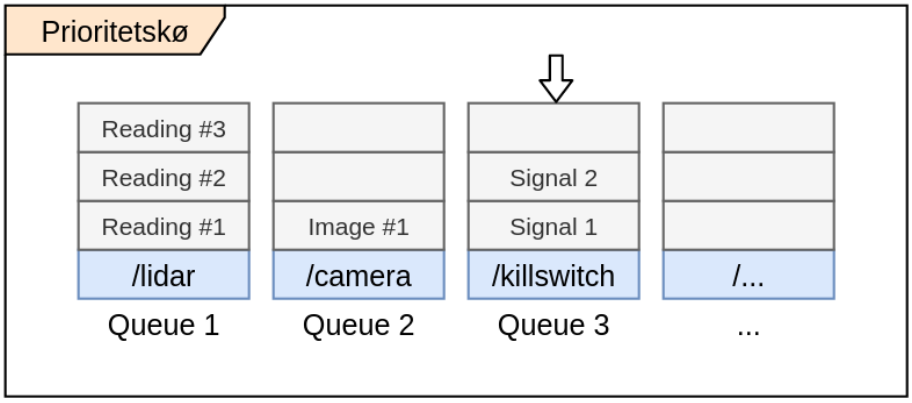
Ønsket fra oppdragsgiver var at det skulle være mulig å sette en prioritetsverdi for hver meldingstype i systemet. Meldingstyper med høyere prioritet skal bli favorisert over andre typer, men sistnevnte skal innimellom få muligheten til å sende en melding. Med andre ord, å fullstendig kutte ut overførselen av meldingstyper med lavere prioritet er ikke alltid ønskelig.

En ren tids- eller rate-basert metode vil dermed ikke fungere, siden det kan føre til at en meldingstype som består av store mengder data, for eksempel bilder fra et kamera, kan innimellom få lov til å bli sendt. Slikt kan være problematisk for svært begrensede forbindelser. Det hadde vært en mulighet å ta utgangspunkt i den nåværende nettverksytelsen og sammenlignet dette med størrelsen for hver melding, men dette kunne igjen vært problematisk dersom operatøren virkelig ønsket å motta data av en type til tross for meldingsstørrelsen.

For å unngå dette ble det lagt til en boolsk verdi for hver meldingstype som indikerer om meldingen skal innimellom få lov til å bli sendt dersom nettverkshastigheten er lav. Hvis verdien er falsk, så vil meldinger av den typen *kun* bli sendt dersom båndbredden er tilstrekkelig og andre meldinger av høyere prioritet er allerede sendt.

Hver meldingstype vil, i prioritetskøen, få sin egen interne *first-in, first-out* (FIFO) kø, som sett i figur 6.14. Størrelsen på de interne FIFO-køene er svært begrensede da det ikke er ønskelig å sende gammel data. Gamle verdier vil bli overskrevet av nye hvis køen er full. Størrelsen er lik for hver kø, og er fastsatt ved kompileringstid. En verdi på 5 ble valgt som utgangspunkt for størrelsen til køene, som vil si at en kø kan maksimalt inneholde fem elementer. Dette er hovedsakelig for å ivareta data som eventuelt har blitt fordelt over flere ROS-meldinger.

De interne FIFO-køene er sortert etter prioritet. Når prioritetskøen blir bedt om å hente ut en verdi, vil den velge en av disse køene for å hente ut en verdi. Hvilken kø som blir valgt baseres på to faktorer: tildelt prioritet, og tid siden sist en melding av denne typen ble sendt.



Figur 6.14: Visualisering av de interne FIFO-køene i prioriteringsalgoritmen. Pilen viser køen som ble valgt av algoritmen.

Ideen er at hvis en meldingstype har fått en prioritet på 100, så skal denne *alltid* bli sendt først. Dette skal derimot ikke være tilfelle hvis den har en prioritet på 95 – da skal andre meldingstyper få mulighet til å innimellom sende en melding. Dette gjøres gjennom aldring: hver meldingstype vil få høyere prioritet desto lenger de har ventet på å få sende en melding. Hver gang en melding hentes fra en kø vil verdien bli resatt. Aldringen vil kun skje dersom den boolske verdien er sann.

Dermed, hvis nettverksytelsen er tilstrekkelig, vil alle meldingene bli sendt. Hvis forbindelsen er dårlig vil kun de viktigste meldingene bli sendt, og – hvis aldring er aktivert – meldingstyper av lavere prioritet vil innimellom komme frem og få sendt en melding.

Funksjonen for prioritet med aldring er gitt ved

$$P_S(t) = \begin{cases} P_0, & P_0 \geq 99 \\ \min\{P_0 + \frac{P_0 t}{C}, P_0 + M\}, & P_0 < 99 \end{cases} \tag{6.1}$$

der,

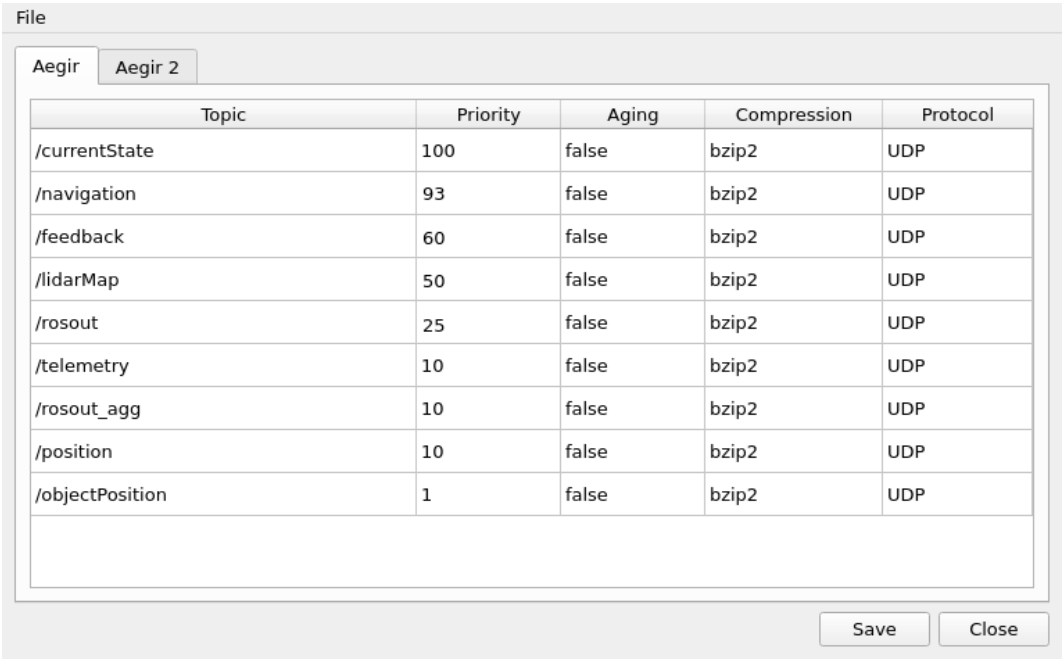
- P_0 er den gitte prioriteten;
- t er tiden siden en melding av denne typen var sist sent;
- C er vektningsfaktoren til veksten; og
- M er maksgrensen for hvor mye en verdi kan øke.

Meldingstyper av høyere prioritet vil dermed vokse raskere enn de med lavere prioritet. Veksten vil kun skje frem til maksgrensen $P_0 + M$ er nådd, som i implementasjonen ble definert slik at $P_0 + M = 99$. Vektningsfaktoren bestemmer vekstraten til funksjonen. Gjennom testing har verdien blitt fastsatt til $C = 60$, men er noe som muligens burde finjusteres. Et alternativ er å få verdien til å vokse invers proporsjonalt med dataraten.

Lagring av usendt data

Fersk data kommer til å bli prioritert over gammel data. For å unngå at gammel data dermed blir liggende i et buffer, var det tenkt at det skulle være mulig å skrive disse meldingene til en fil slik at dataen kunne innhentes senere. Dette skulle sammenkobles med datainnsamlingsdelen av prosjektet, men det ble det ikke tid til. Per nå vil gammel, usendt data bli forkastet og overskrevet av nyere data.

6.5.8 Brukergrensesnitt



Figur 6.15: Brukergrensesnittet for prioriteringssystemet.

Brukergrensesnittet er delt inn i faner, der hver fane representerer en klient. Hver klient har sine egne innstillinger. Innstillingene endres på en per-*topic* basis, og kan endres på i sanntid.

Grensesnittet lar operatøren stille på følgende parametere:

- **Prioritet**: Hvilken prioritet denne meldingstypen skal ha. Spesifisert med en verdi mellom 0 og 100.
- **Komprimeringsalgoritme**: Om meldingene skal bli komprimert, og isåfall Hvilken komprimeringsalgoritme som skal tas i bruk. Operatøren kan velge mellom *Gzip*, *Zlib*, og *Bzip2*.
- **Kryptering**: Om meldingstypen skal bli kryptert (ja/nei).
- **Protokoll**: Hvilken kommunikasjonsprotokoll som skal tas i bruk.

Brukergrensesnittet er tar i bruk *QComboBox*-objektet for å vise de forskjellige innstillingene.

6.5.9 Datarate

For å måle bitraten vil systemet se hvor lang tid det bruker på å sende en viss mengde data, og deretter dele mengden på tiden for å finne et estimat for tilgjengelig båndbredde i form av bytes per sekund. På grunn av forsinkelser og lignende kommer ikke dette til å være en nøyaktig verdi, men er derimot så nøyaktig målingen kan bli gjort uten å sende unødvendig data. For små mengder data vil verdien være veldig unøyaktig, mens ved store mengder vil måling bli mer og mer nøyaktig.

Linux tilbyr en fil, */proc/net/dev*, som inneholder informasjon om hvor mange bytes har blitt sendt siden systemet ble startet. Det hadde vært mulig å regelmessig lese denne verdien og estimere båndbredden ut fra det, men dette ville vært basert på totalforbruket og ikke tilgjengelig kapasitet siden systemet vil ikke alltid sende data til enhver tid.

6.6 Konklusjon

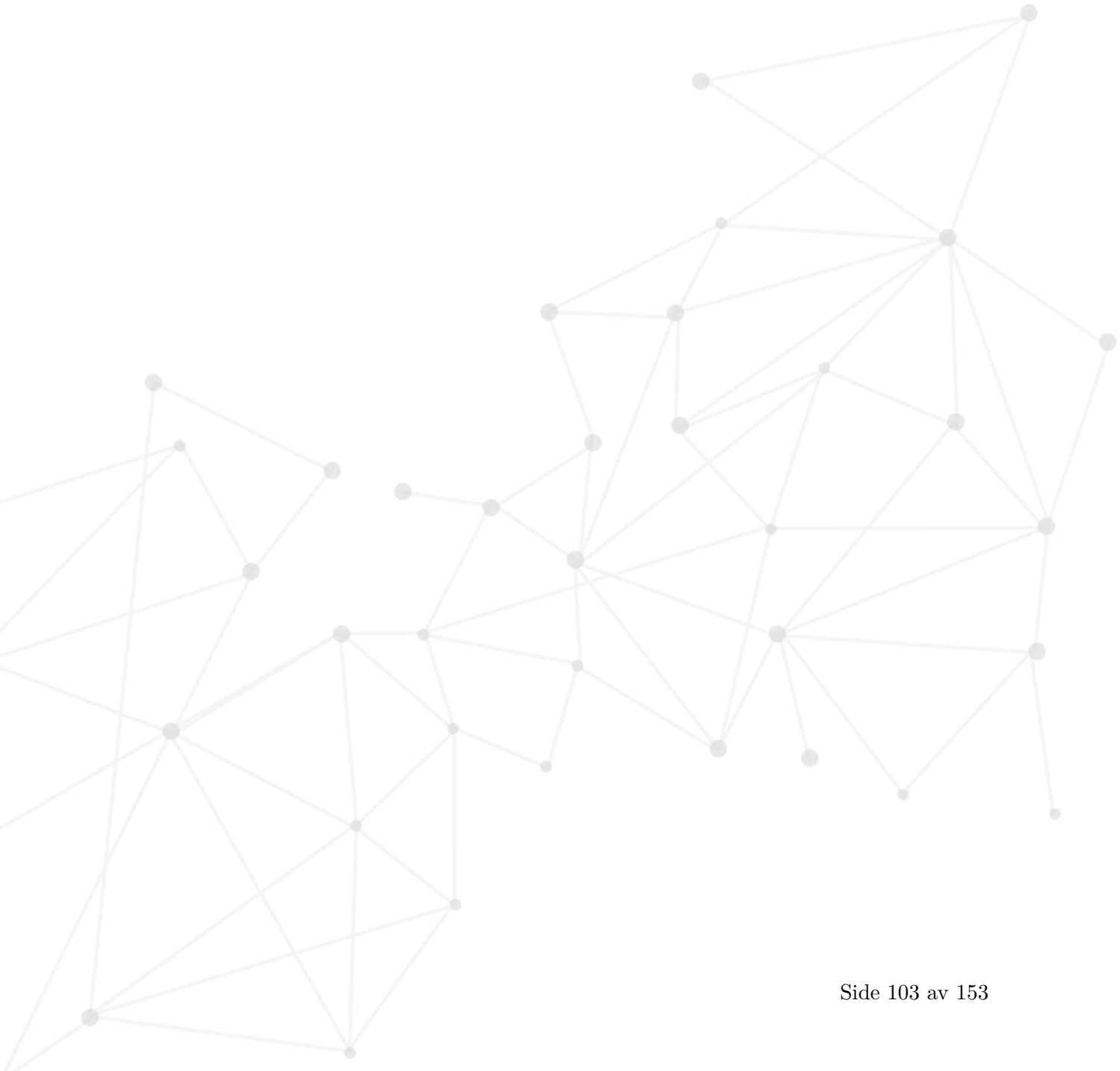
Ved å lage et system som tar for seg overføringen av ROS-meldinger på tvers av systemer og nettverk var det mulig å kutte ned på den totale datamengden som blir sendt, redusere forsinkelser ved dataoverføringene, og introdusere en form for prioritering av utgående data.

Løsningen gir operatøren muligheten til å sette prioriteringer på diverse meldingstyper som blir sendt mellom C2 og Aegir, slik at viktige meldinger kan komme fortere frem. I tillegg ble det introdusert komprimering og kryptering av datakommunikasjonen. Løsningen har et tilhørende brukergrensesnitt som lar brukeren endre på konfigurasjonen mens systemet kjører, eller manuelt fastsette innstillingene ved oppstart av systemet.

I tillegg tar løsningen for seg automatisk synkronisering av ROS *topics* slik at utviklingsprosessen til systemet fremover vil bli enklere.

7. Strupingsverktøy

Dette kapittelet vil ta for seg gruppens resonering rundt deloppgaven om oppsetting av et strupingsverktøyet. Løsningene gruppen kom frem til vil bli beskrevet kort og konsist, mens den valgte løsningen vil bli forklart i detalj. Testing av løsningen, med resultater, blir også beskrevet i kapittelet.



7.1 Problemstilling

Ved kommunikasjon mellom C2 og Aegir kan det i en realistisk situasjon bli store endringer i nettverksytelsen på forbindelsen. Oppgaven gikk derfor ut på å gjøre det mulig å teste handlingene til C2 og Aegir under dårlige eller endrende nettverkforhold. For å gjøre dette måtte gruppen ha et verktøy som kunne manipulerer forbindelsen mellom C2 og Aegir for å simulerer en spesifikk realistisk situasjon. Verktøyet kunne da bli brukt til feilsøking og testing av Coastal Shark og andre systemer.

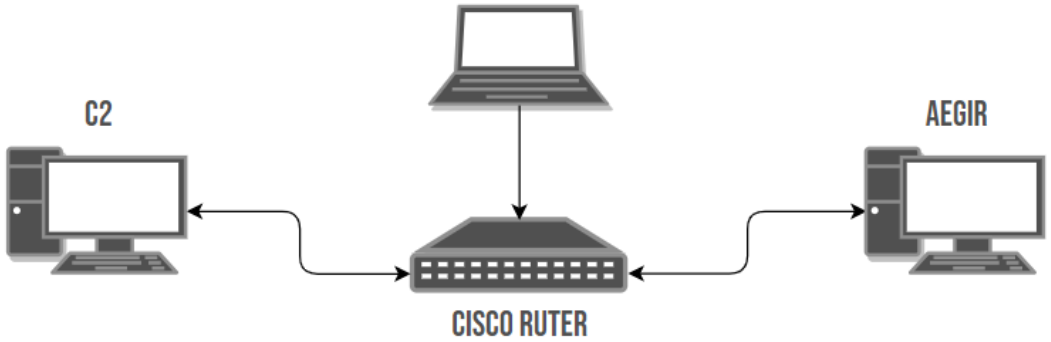
7.2 Løsningsforslag

Gruppen har vært innom to mulige løsninger for dette strupingsverktøyet.

- En Cisco ruter
- En virtuell ruter

7.2.1 Løsningsforslag 1: Cisco ruter

Den første løsningen var et forslag gitt av KDA gjennom et spesifisert krav. Kravet sa at gruppen skulle bruke en Cisco ruter til å kontrollere nettverksytelsen på på forbindelsen mellom Aegir og C2. Denne ruteren har allerede innebygde funksjoner for å kunne håndtere denne type oppgaver, og den måtte da kobles opp til en laptop eller desktop for å kunne bli kontrollert gjennom en terminal interface. Dette er visualisert i 7.1.



Figur 7.1: Visualisert løsning 1

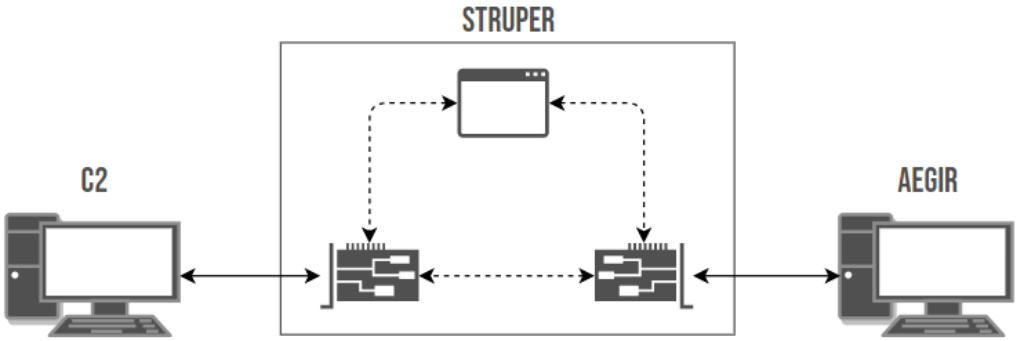
7.2.2 Løsningsforslag 2: Virtuell ruter

Den andre løsningen var et forslag gruppen kom med til arbeidsgiver om å lage en virtuell løsning for å styre nettverksytelsen. Dette kom av at en slik løsning kan bli satt opp på samme maskin som gjør en operasjon og dermed spare mye flytting og oppsett av hardware. Den kunne også få en GUI som ble spesifisert for sitt bruksområde, som vil si at verktøyet kunne få en mer oversiktlig interface. Denne løsningen er visualisert i 7.2.

Figur 7.2: Visualisert løsning 2

7.2.3 Valgt løsning

Gruppen diskuterte med arbeidsgiver om valg av løsning. Løsningen ble en hybrid av den virtuell løsning basert på funksjonaliteten *Traffic Control* [24], forkortet TC, som er innebygget i pakken *IProute2* [25] for Linux. Denne ville bli installert på en maskin med to nettverkskort, og på den måten bli en egen enhet slik at den enkelt kan kobles direkte inn i en annen forbindelse senere. Denne maskinen kunne dermed kontrollere nettverksytelsen med TC-kommandoer til Linux terminalen. For at strupingen skulle bli lettere å kontrollere besluttet gruppen å lage en GUI-applikasjon som ville bruke TC-biblioteket som erstatning for Linux-terminalen. Figur 7.3 er en visualisering av den valgte løsningen.



Figur 7.3: Visualisert valgt løsning

7.3 Konklusjon

For at maskinen med to nettverkskort kunne installeres mellom hvilken som helst forbindelse, måtte kortene i maskinen kobles sammen. Dette var for at maskinen skullle fungere som en ruter hvor maskinene som var tilkoblet ruteren kunne kommunisere med hverandre. For å koble disse sammen ble det satt opp en *bridge* [26] mellom kortene. Denne bridge-forbindelsen ville gjøre at kortene videresender pakkene de mottar til det andre kortet. Denne forbindelsen kan bli satt opp med bruk av kommandoer i terminalen eller endring av nettverksfiler i Linux. Hvis man bruker terminal kommandoer kan det bli brukt flere forskjellige kommandoer, avhengig av hvilke verktøy man har og vil bruke. Denne bridgen blir ikke permanent siden nettverket blir resatt når maskinen restarter. For å få en permanent bridge må man redigere nettverksfilene i Linux.

Det beste for denne maskinen var å ha en permanent forbindelse slik at det ikke blir nødvendig å sette den opp hver gang maskinen starter. Derfor satte gruppen opp en permanent forbindelse ved å redigere filen for nettverksoppkobling i Linux. Grupppen laget en oversikt over hvordan dette kan settes opp slik at det kan bli flyttet til en ny maskin uten problemer.

Strupingsverktøyet ble basert på TC, som er en funksjonalitet som kan manipulere nettverksforbindelser. Den gjør dette ved å manipulere køene som mottar og sender pakker. Disse køene blir kalt for *qdisc* [24] og kan bli laget, manipulert og slettet ved hjelp av TC.

Ved å lage en applikasjon bygget på biblioteket til TC kunne en bruker, gjennom et GUI, manipulere qdiscer og dermed nettverksytelsen. I en dialog med arbeidsgiver senere i prossesen tok gruppen en avgjørelse på at implementasjonen av et GUI for strupingen av nettverkytelsen skulle nedprioriteres til et B krav. Dette var grunnet at funksjonaliteten var allerede brukbar ved å bruke kommandoer i Linux-terminalen og at det trengtes mer arbeidskraft i andre deler av prosjektet. Det ble

derfor laget en brukermanual for hvordan kommandoer i terminalen brukes for manipulering av nettverkskortene som erstatning for et GUI. Denne ble slått sammen med instruksjonen for oppsetting av en *bridge*.

7.3.1 Testing

Gruppen valgte å bruke verktøyet *Iperf* [27] for Linux til testing av strupingen av nettverksytelsen. Dette verktøyet vil kunne vise båndbredden ved å overføre pakker i en gitt tid og gi en gjennomsnittlig overføringshastighet. Ved å observere resultatene som dette verktøyet gir kan man se om nettverksytelsen faktisk blir endret. Gruppen observerte at overføringen får en økning i overføringshastigheten ved oppstart før den stabiliserer seg. For å få en riktig gjennomsnittshastighet måler gruppen hastigheten over en lenger periode på 3 minutter.

Gruppen kjørte testen som er spesifisert i testspesifikasjon T8. Denne ga følgende resultater som er representert i 7.1

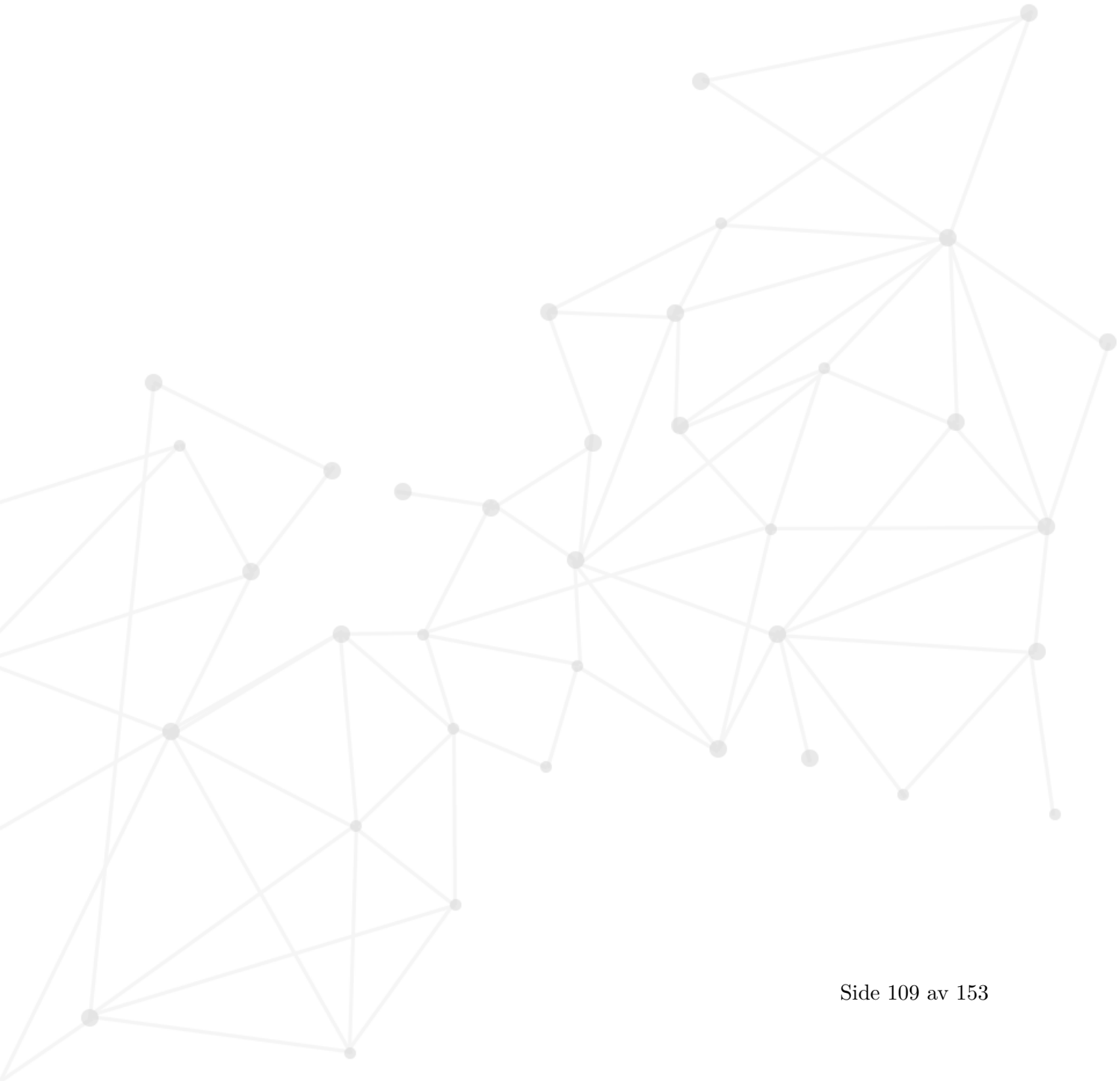
Tabell 7.1: T8 test resultat

	Klient på B		Klient på A	
Begrensning	Test 1	Test 2	Test 1	Test 2
Ingen	79.6mbps	83.0mbps	56.4mbps	55.4mbps
10mbps	9.13mbps	9.13mbps	9.13mbps	9.13mbps
1mbps	0.92mbps	0.92mbps	0.92mbps	0.92mbps
500kbps	473kbps	473kbps	473kbps	473kbps
50kbps	47.9kbps	47.4kbps	47.8kbps	47.3kbps
10kbps	9.80kbps	9.71kbps	9.69kbps	9.89kbps

Disse resultatene viser gruppen at verktøyet struper hastigheten til tilnærmet den gitte hastigheten, selv ved veldig lave hastigheter som kan være vesentlig å teste for noen systemer. Strupingen har derfor bestått testen og kravet er blitt godkjent.

8. Datainnsamling og loggesystem

Følgende kapittel forklarer deloppgavene rundt datainnsamling og loggesystem, med forskjellige løsningsforslag og konklusjoner.



8.1 Problemstilling

Det var et krav fra oppdragsgiver å lage et system som samler inn alle meldinger som sendes fra Aegir. Systemet skal registrere et lokalt timestamp, datatype og innhold i meldingen. Disse dataene skulle skrives til en eller flere filer, som senere skal kunne hentes opp til bruk i testsett. Loggingen skulle registrere meldinger som sendes og mottas på både Aegir og landstasjon, samt logge nettverksytelsen. Dette systemet skulle også ha et brukergrensesnitt, slik at de lagrede dataene er lesbare for operatøren på landstasjon. Det ble tenkt at filene med de lagrede dataene befinner seg på Aegir, slik at den må kjøres til land for å hente ut dataene.

For å kunne lese meldingene som sendes fra nodene på Aegir, må det være en funksjon som registrer meldingene og lagrer dem slik at det er mulig å hente ut dataene. Systemet må også ha en funksjon som overvåker nettverkstrafikken og registrerer alle meldinger som faktisk blir sendt og mottatt, både fra Aegir og C2.

8.2 Løsningsforslag

For å forstå dataflyten mellom nodene på Aegir og Coastal Shark systemet forøvrig, var det nødvendig å gjøre seg kjent med ROS konseptet. Etter mange undersøkelser og vurderinger kom gruppen frem til flere alternative løsninger på hvordan oppgaven kunne løses.

8.2.1 Løsningsforslag 1: C++ bibliotek

I begynnelsen ble det tenkt at datainnsamling og logging skulle være ett system. Det ble sett på mulige løsninger med tcpdump [28], som overvåker pakker som sendes over nettverket, og libpcap [29] som er et bibliotek i C++ som fanger opp pakkene. Det ble imidlertid konkludert med at det kunne være utfordrende å lese pakker,

ettersom en melding kan bestå av flere pakker. Dermed kan det være vanskelig å finne ut hvilke pakker som hører til samme melding og hvor pakkene kommer fra.

8.2.2 Løsningsforslag 2: ROS-biblioteker

Ettersom Coastal Shark er bygget opp på ROS, og det ble fastslått at alle meldinger skulle sendes med ROS, ble det avgjort at den enkleste og beste løsningen ville være å bruke biblioteker som allerede eksistererer i ROS. Derfor ble det fokus på å lage dette innsamlingssystemet med rosbag [30], et bibliotek som lagrer meldinger i bagfiler, og rosout [31] som lagrer meldinger i loggfiler.

8.2.3 Løsningsforslag 3: Separat datainnsamling- og loggesystem

Etter nøyere undersøkelser kom gruppen fram til at den beste løsningen var å dele dette i to separate systemer, et for logging og et for datainnsamling. Dette fordi datainnsamlingen fanger opp alle meldinger som sendes fra nodene, og loggingen logger de meldingene som er ferdig prioritert og faktisk sendes over nettverket. Derfor skjer datainnsamlingen før prioriteringen og loggingen skjer etter at meldingene er prioritert.

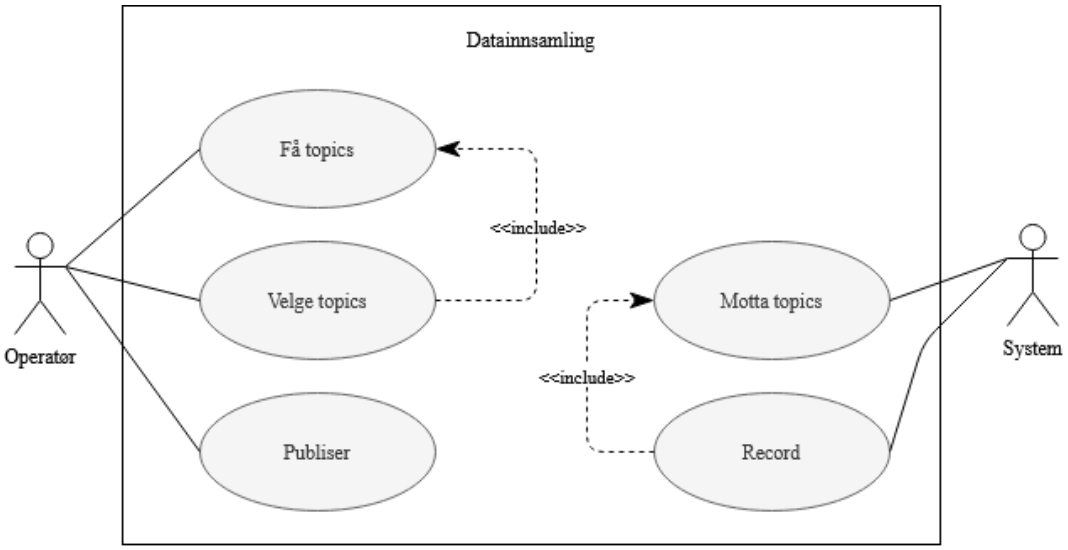
8.3 Datainnsamlingssytem

Datainnsamlingssystemet ble nå et eget delsystem. Dette delsystemet ble fordelt på en kontrollstasjon på C2 og selve datainnsamlingen på Aegir. Med denne løsningen valgte gruppen å bruke ROS sitt ferdige bibliotek, rosbag. rosbag vil gjøre det mulig for datainnsamlingssystemet å abonnere til alle *topics* fra nodene på Aegir, slik at alle meldinger som sendes fra Aegir fanges opp av innsamlingssystemet. Disse meldingene inneholder data som skrives til en bagfil med et lokalt timestap.

Løsningen har en node på Aegir som bruker rosbag for starting av datainnsamlingen og tar opp meldingene som sendes. Den vil da danne bagfiler som kan brukes til å lese meldingen eller spille dem tilbake som et testsett. Dette er noe som er mulig gjennom ROS applikasjonen rqt_bag [32]. Denne applikasjonen har et GUI som representerer dataen fra bagfilen som en tidslinje med datainnhold. Den gjør det også mulig å publisere meldinger tilbake til *topics* og dermed kjøre dem som et testsett. Denne applikasjonen kjøres på C2 for å se over filene og på en maskin som simulerer Aegir for å kjøre testsett.

Use case for datainnsamling

For å illustrere en oversikt over datainnsamlingen i prosjektet og hvordan systemet samhandler, lagde gruppen et *use case* diagram, figur 8.1.



Figur 8.1: *Use case* diagram for datainnsamling.

8.3.1 Datainnsamling på Aegir

Datainnsamlingssystemet kunne løses på flere måter. Det ble som nevnt tidligere bestemt å bruke ROS biblioteker, men implementeringen av disse kunne brukes på

mange forskjellige måter. I likhet med mye annet i ROS, inneholder rosbag en del statiske funksjoner. Dermed ville noden som håndterte datainnsamlingen kjøre til den ble avsluttet. Dette fører til at endringer av innstillinger på datainnsamlingen ikke ble mulig mens noden kjører. Gruppen måtte derfor finne en løsning på hvordan innstillingene kunne endres uten å måtte restarte Aegir for hver gang.

Løsningsforslag 1: To noder

En måte å løse det på er å lage to noder, som eksempel A og B. Node A benytter seg av klassen rosbag::Recorder for å starte datainnsamlingen og lagre data til bagfiler. Når det ønskes å endre innstillingene for datainnsamlingen, vil node B restarte node A. Dette fører imidlertid til mye kommunikasjon mellom nodene og krever tid og kapasitet, ettersom noden må restarte hver gang det gjøres endringer.

Løsningsforslag 2: Node med launch fil

Med denne løsningen er det en launch fil [33] som starter opp noden for datainnsamlingen. Denne noden er tilsvarende node A i løsningsforslag 1. I launch filen er det mulig å sette en variabel slik at i tilfeller hvor noden avsluttes, vil filen automatisk restarte noden helt til filen blir avsluttet. Med denne løsningen er det ikke kommunikasjon mellom noder for å kontrollere omstart, som vil føre til mindre risiko for uforutsette feil. Likevel vil det fortsatt kreve tid og kapasitet når noden avsluttes og restartes.

Løsningsforslag 3: Modifisere rosbag::Recorder

Denne løsningen er å lage en node som bruker en modifisert versjon av klassen rosbag:Recorder [34]. Det blir lagt til en modifisert versjon av start funksjonen som gjør det mulig å stoppe den uten å skru av noden. Ved bruk av denne modifiserte klassen

i en node, vil den ha en funksjonalitet som gjør det mulig å stoppe datainnsamlingen og endre innstillingene. Med denne løsningen vil noden kunne kjøre hele tiden, slik at den ikke bruker tid og kapasitet på kommunikasjon og restarting for endringer av innstillinger.

Konklusjon

Etter mye undersøkelser og testing av konseptene, valgte gruppen å gå for løsningsforslag 2 med en launch fil som starter noden. Dette var grunnet med at løsningsforslag 3 ga gruppen store vanskeligheter. Disse vanskelighetene var knyttet til at etter den første undersøkelsen av løsning 3 hadde gruppen inntrykk av at det var en funksjon som måtte modifiseres. Det viste seg i videre undersøkelser at funksjonen i rosbag::Recorder tok i bruk flere funksjoner som var fordelt over flere tråder og flere av disse funksjonene var låst, inntil noden ble slått av. Dette førte til at gruppen måtte modifisere større deler av klassen enn først antatt, som ville føre til forsinkelser i prosjektet. Gruppen tok derfor i samarbeid med arbeidsgiver en beslutning om å gå tilbake til løsning 2 da tiden og kapasiteten som krevdes for omstart av noden ble vurdert til minimal og ville derfor ikke påvirke systemet. Løsningsforslag 2 hadde gruppen allerede i en testfase fra tidligere undersøkeler, som gjorde at det var et utgangspunkt man kunne bygge videre på. Denne løsningen viste seg å være en gylden middelvei mellom tid og funksjonalitet.

Noden baserte seg på den originale klassen rosbag::Recorder. Ved å lage et objekt av denne klassen kunne noden starte innspilling av data til bagfiler. Objektet av denne klassen krever en datastruktur parameter av typen rosbag::Recorder::RecorderOptions. Denne datastrukturen inneholder all informasjon relatert til innstillingene for innspillingen av data, som gruppen kaller datainnsamlingen. Det er ingen måte å sette disse innstillingene utenfor konstruktøren av rosbag::Recorder objektet. Dette ble løst ved at noden blir omstartet av launch filen, som vil føre til at objektet blir destruert og det blir dannet et nytt objekt for hver omstart. Dette gjør det mulig å

endre datastrukturen før det nye objektet dannes og starter innsamlingen. Det store problemet som blir løst, er kjør funksjonen til objektet som blir låst ved å kjøre et kall på ros::spin() funksjonen som vil kjøre til noden blir omstartet. For hver gang innsamlingen starter blir det dannet en ny bagfil.

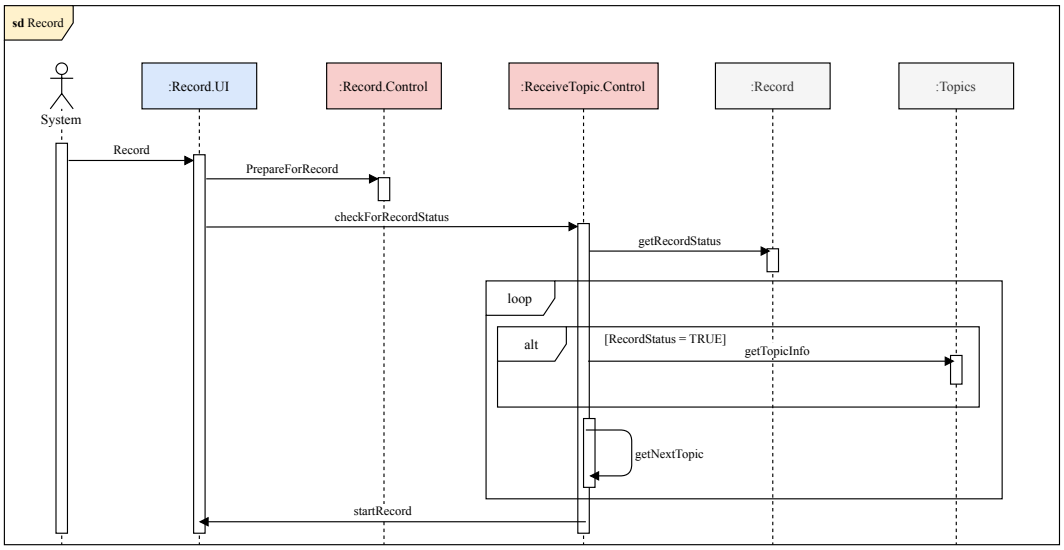
For kontroll av datainnsamlingen mottar noden en ROS melding fra C2 som inneholder en vektor. Denne vektoren leses av noden og bestemmer om noden skal starte innsamling av data, og hvilke *topics* som eventuelt skal innsamles.

Gruppen laget noden slik at den genererer en mappe i hjem folderen på maskinene, hvis den ikke finnes fra før. Her vil bag filene fra innsamlingen lagres med prosjektnavn, tid og dato. Dette er noe som lett kan bli modifisert ved endringen av en variabel som angir veien for lagring og prosjektnavn.

rosbag::Recorder gir en funksjonalitet for å kryptere deler av data som blir lagret. Noden bruker denne funksjonaliteten til å komprimere deler av innsamlet data med *Bzip2* som vil redusere den totale filstørrelsen for å spare lagringsplass på Aegir.

Sekvensdiagram for innsamling på Aegir

Gruppen utviklet et sekvensdiagram, figur 8.2, for å vise den sekvensielle handlingen for datainnsamlingen på Aegir. Her illustreres hvordan datainnsamlingssystemet på Aegir mottar *topics* og starter innsamling av data.



Figur 8.2: Sekvensdiagram for datainnsamling på Aegir.

8.3.2 Datainnsamling på C2

Datainnsamlingen har et GUI på C2 hvor operatøren på kontrollstasjonen kan velge om man ønsker opptak av dataene som sendes fra nodene på Aegir. Her kan operatøren også velge om det er enkelte *topics* man ikke ønsker å ta opp, for å begrense lagringsomfanget.

QNode

På C2 er det en node som er inspirert av klassen QNode [35] for å opprette et grensesnitt mellom Qt og ROS. QNode klassen ble modifisert slik at den tar imot *topics* fra ROS masteren og brukes i Qt. QNoden mottar en vektor fra Qt og publiserer til et ROS *topic* som fanges opp av noden på Aegir. Denne vektoren inneholder et første element med verdi 1 eller 0. Noden som befinner seg på Aegir vil lese dette elementet først, hvis verdien er 1 betyr det at de neste elementene inneholder data fra *topics* som er publisert. Dersom verdien i det første elementet i verktoren er 0, betyr det at opptak av data er stoppet. Coastal Shark systemet har sin egen QNode, slik at

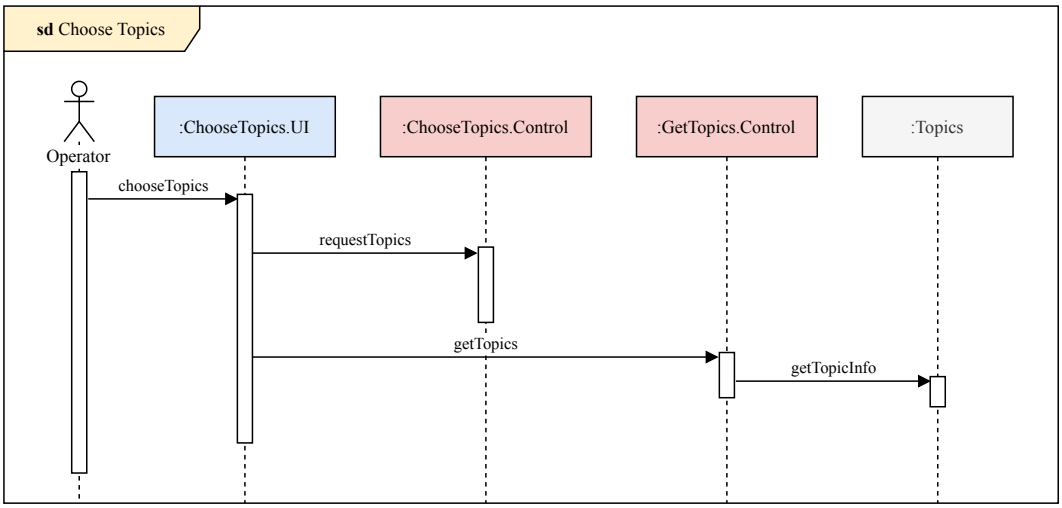
funksjonene i QNoden som gruppen har laget blir implementert i det eksisterende systemet.

QThread

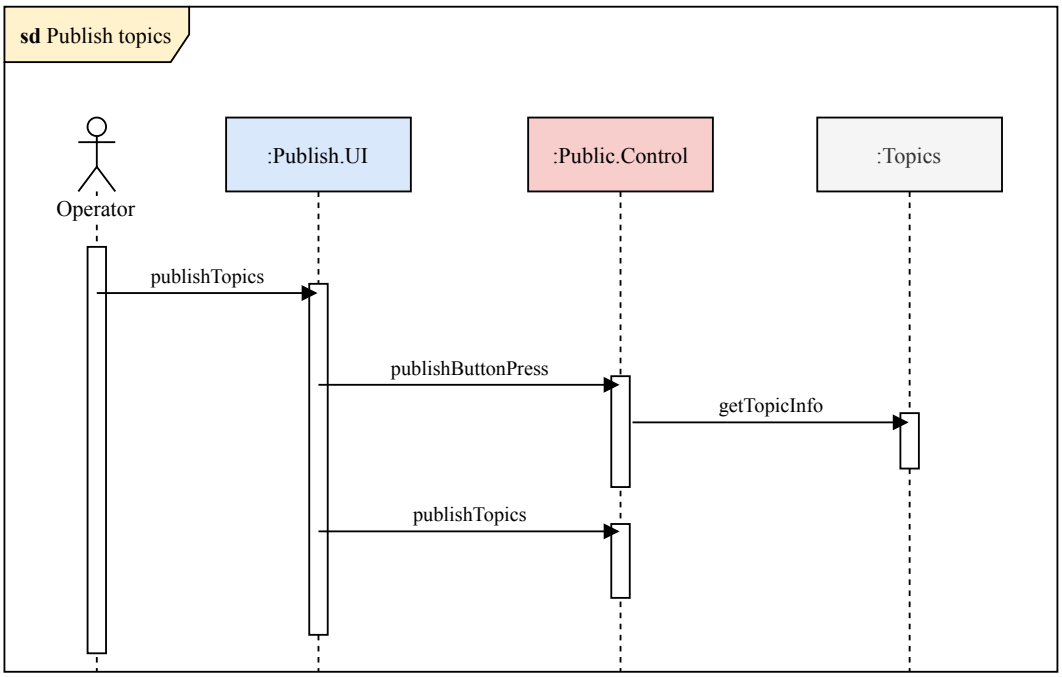
QNoden bruker QThread [36] for å ha muligheten til å kjøre Qt og ROS på seperate tråder. Den deler data med de andre trådene i systemet, men kjører samtidig uavhengig. Istedenfor å starte i main(), så arves en run() funksjon som inkluderes i koden .

Sekvensdiagram for datainnsamling på C2

Gruppen utviklet sekvensdiagrammer, som viser den sekvensielle handlingen for datainnsamlingen på C2. Her illustreres hvordan datainnsamlingssystemet henter og velger *topics*8.3, og publiserer fra noden8.4.



Figur 8.3: Sekvensdiagram for datainnsamling på C2.

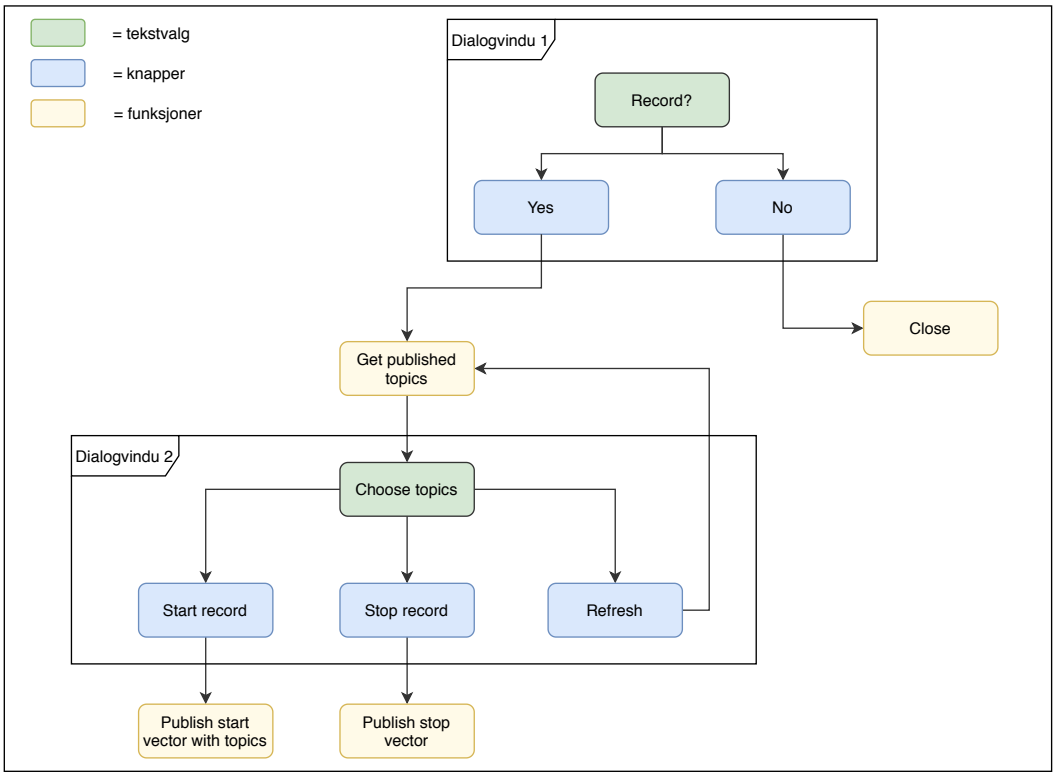


Figur 8.4: Sekvensdiagram for datainnsamling på C2.

Brukergrensesnitt

Brukergrensesnittet på C2 ble laget i Qt og vil gi operatøren på kontrollstasjonen enkelte valg i forhold til opptak av dataene som sendes fra nodene på Aegir.

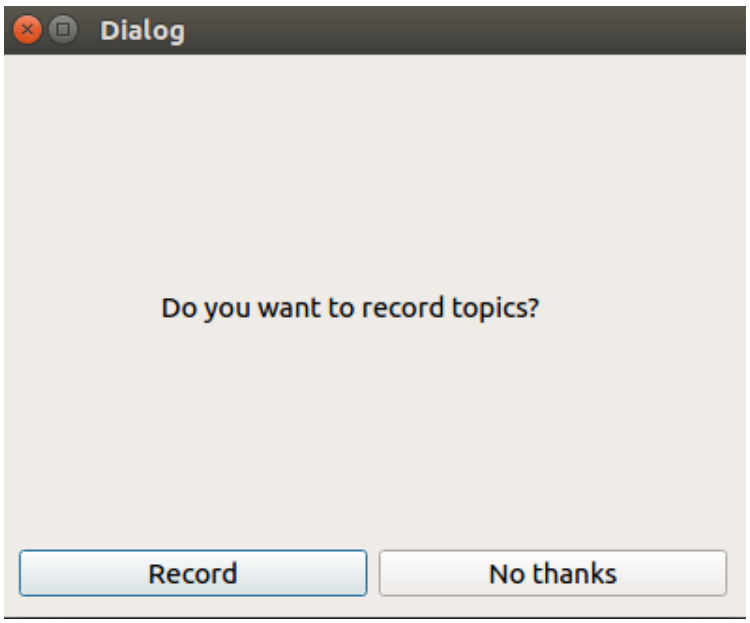
Som figur 8.5 viser, vil man først få et spørsmål om å ta opp *topics*, dernest vil noden se etter tilgjengelige topics og operatøren kan bestemme hvilke topics han ønsker opptak av. Deretter kan man velge å starte opptak til noden på Aegir, som vil sende vektoren med de valgte topics. Hvis man ønsker å endre innstillingene på valgte topics, er det et valg om å oppdatere listen og sende en ny vektor til noden. Ved å velge å stoppe opptak vil det sendes en vektor til noden på Aegir med kun elementet 0.



Figur 8.5: Brukergrensesnitt for opptak av data.

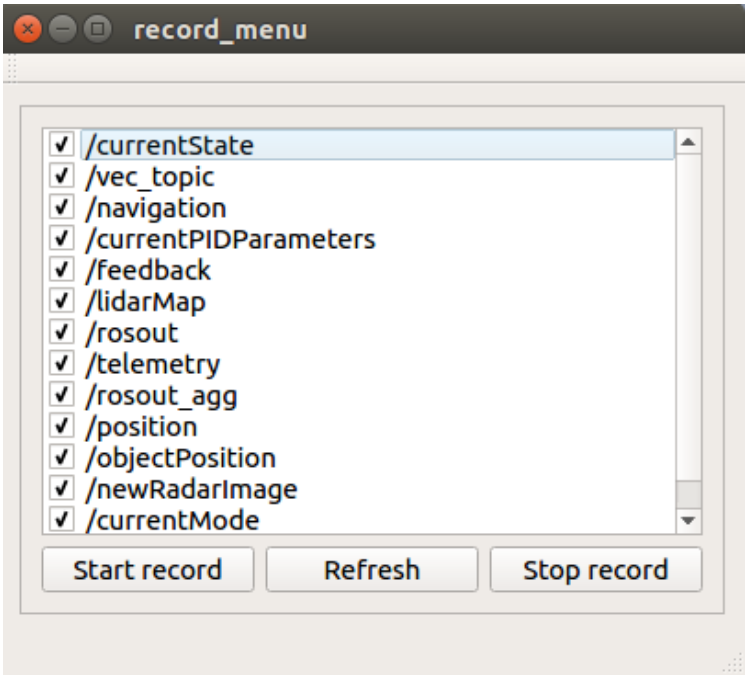
Dialogvindu

Det første dialogvinduet kommer opp med et spørsmål om å ta opp *topics* eller ikke, se figur 8.6



Figur 8.6: Dialogvindu for opptak av data.

De tilgjengelige *topics* som publiseres vil komme opp som en liste med en sjekkboks for hver *topic*, se figur 8.7. Disse sjekkboksene er som standard huket av, slik at hvis man ikke ønsker opptak av enkelte *topics* kan operatøren krysse ut disse før han trykker "Start record", for å starte opptak av dataene. Som forklart tidligere, har operatøren også mulighet for å oppdatere listen over tilgjengelige *topics*, samt stoppe opptak av data.



Figur 8.7: Dialogvindu med liste over topics.

Alternativ løsning

I stedet for å få opp et spørsmål om å ta opp *topics*, er det en mulighet å lage en konfigurasjonsfil som enten starter, eller ikke starter opptak av *topics* når systemet startes opp.

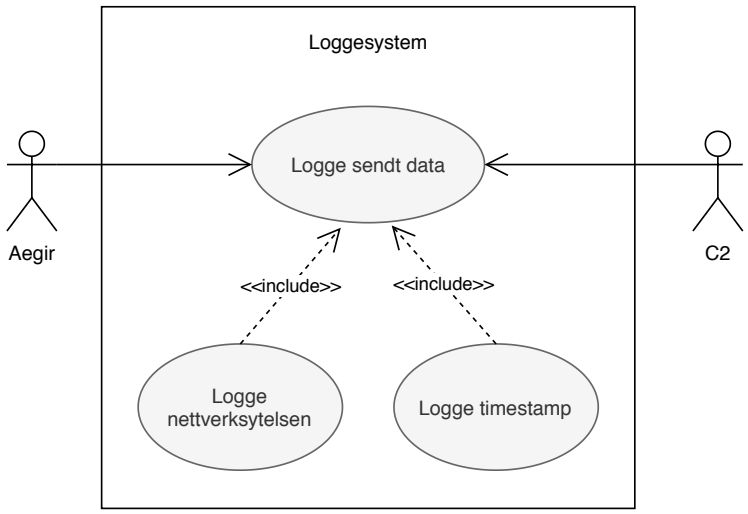
Gruppen har valgt å ha et *pop-up* vindu for å velge om man ønsker å ta opp *topics*, av flere grunner. Det ene argumentet er at operatøren på C2 blir minnet om å ta dette valget, slik at det ikke blir glemt og dermed ikke ha mulighet for å spille av testsett senere, som er ønskelig i tilfeller hvor noe går galt for eksempel. I dette tilfelle ville det vært natulig å ha en konfigurasjonsfil som satte i gang opptak når systemet starter. Men det tar oss til neste argument, nemlig at det kan ta opp mye plass hvis all data blir lagret til enhver tid. Derfor er det mer gunstig å få opp dette valget, slik at man har mer kontroll på hva og hvor mye data som blir lagret.

8.4 Loggesystem

Loggesystemet er nå sitt eget delsystem. Loggesystemet sin oppgave er å logge alle meldinger som sendes over radiolinken mellom Aegir og C2 samt skrive dette til en fil. Filen skal inneholde *timestamp*, båndbredde, *topics*, om meldingen er sendt fra eller til Aegir og C2, meldingstype, navn på meldingstype og størrelse på meldingen.

Use case diagram for loggesystemet

For å illustrere en oversikt over loggingen og hvordan det samhandler med systemet, laget gruppen et *use case* diagram, i figur 8.8.



Figur 8.8: *Use case* diagram for loggesystemet.

8.4.1 Loggesystemet for sendt og mottatt på Aegir og C2

Loggesystemet for sendte og mottatte meldinger på Aegir og C2 kunne løses på flere måter. Som nevnt tidligere var det tenkt å bruke ROS sitt ferdige bibliotek, rosout, men etter nøyere gjennomgang kom gruppen fram til to løsninger.

Løsningsforslag 1: ROS-implementasjon

Ved en ROS implementasjon ville loggesystemet vært sin egen node. Det hadde da vært mulig å implementere loggingen ved å bruke ROS sitt ferdige bibliotek rosout, men samtidig ville en ROS-implementasjon betydd flere kjeder i systemet som kunne feilet. ROS-meldingene måtte blitt sendt over nettet og mottatt før de kunne blitt printet til fil, og denne overføringen ville gjort denne løsningen litt tregere enn en evt. annen løsning. For å motta meldinger ville systemet dessuten vært avhengig av andre noder, som igjen ville betydd at begge nodene måtte kjørt samtidig og det ville oppstått en avhengighet.

Løsningsforslag 2: Implementeres som en del av prioriteringssystemet

Ved å gjøre loggesystemet til en del av prioriteringssystemet ville loggingen kunne mottatt meldinger direkte fra prioriteringsdelen. Det hadde betydd at tiden for selve overførselen av meldinger ikke ville påvirket loggesystemet. En annen fordel ved å implementere det på denne måten er at det ikke hadde blitt en helt ny kjede i systemet, og kompileringen kunne kjørt direkte uten annen avhengighet enn koblingen til selve prioriteringsdelen.

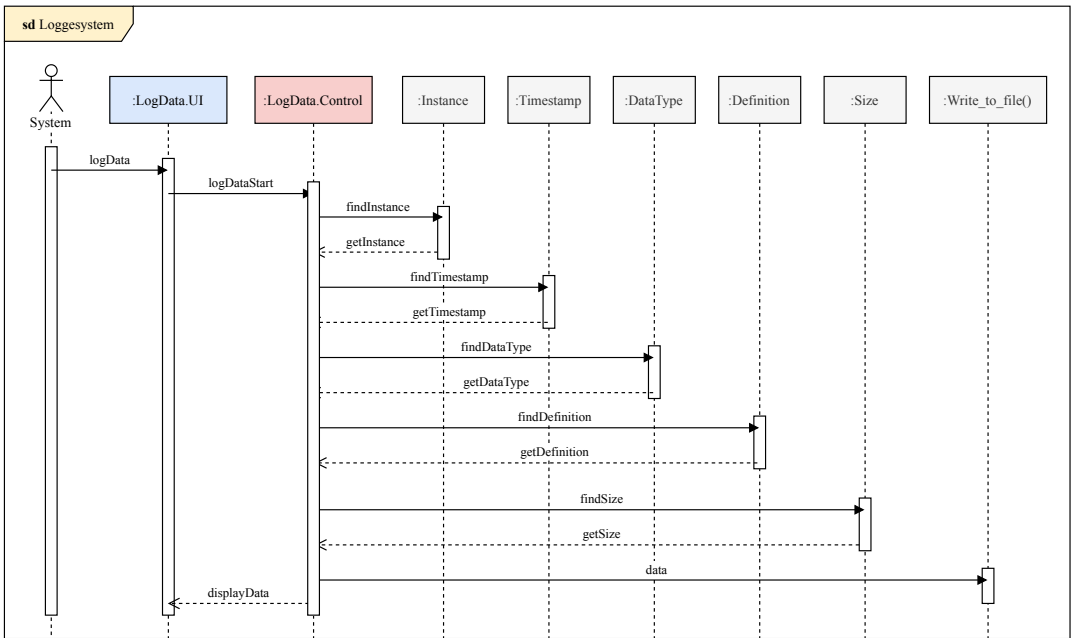
Konklusjon

Etter å ha vurdert begge løsningsalternativene kom gruppen fram til at det ikke var nødvendig å implementere loggingen i ROS. Loggingen blir kun brukt av prioriteringssystemet, så det var enklere å implementere det der istedet for å separere det til sin egen node. Systemet vil også være raskere siden meldingene allerede er overført til prioriteringsdelen, og koden kan kompileres direkte uten å være avhengig av andre deler enn selve koblingen mellom logging og prioritering. Loggesystemet er derfor nå en del av prioriteringssystemet, hvor prioriteringssystemet kaller direkte

på loggefunksjonen. Det ble bestemt at loggingen skulle være generisk og derfor ikke bruke ROS-spesifikke funksjoner. Det betyr at loggingen tar i mot meldinger i et string-format fra prioriteringen på Aegir, og skriver dette til en fil. Koden ble derfor skrevet i C++, utenfor ROS.

Sekvensdiagram for loggesystemet

Gruppen laget et sekvensdiagram som viser den sekvensielle samhandlingen mellom loggesystemet og Aegir eller C2. Her illustreres hvordan all meldingsinfo hentes når meldingene går via prioriteringssystemt, se figur 8.9. Når systemet kaller på instance-objektet, vil den peke på seg selv dersom objektet allerede er instansiert. Om objektet enda ikke eksisterer, vil det instansierers i dette øyeblikket. Når all informasjon er innhentet skrives det til fil.



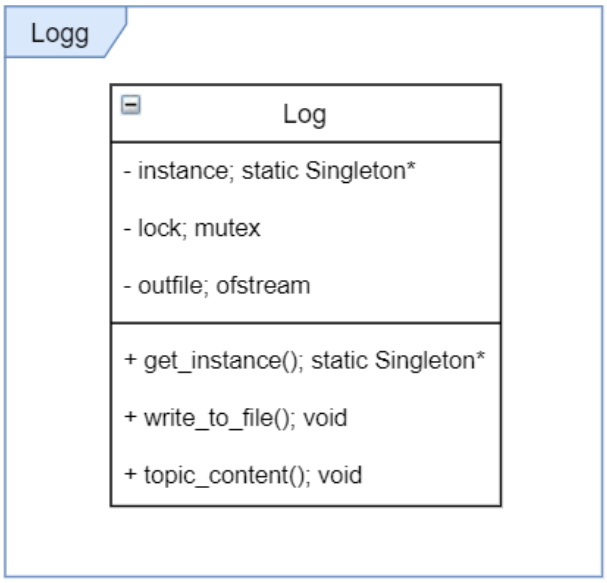
Figur 8.9: Sekvensdiagram for logging.

8.4.2 Design

Her følger en forklaring på designet av loggesystemet.

Singleton

På loggingen ble det brukt en Singleton-klasse [37]. Singleton-klasse ble valgt fordi det bare skal instansieres ett objekt av denne typen. Dette er for å sikre at det bare finnes en instanse av klassen, slik at det kun er en loggfil som blir generert. For at dette skal la seg gjøre må konstruktøren være privat og kan da bare kalles via en *public* klassefunksjon som returnerer en peker til et globalt objekt.



Figur 8.10: Klassediagram for en Singleton-klasse.

ShapeShifter

Loggingen skal være generisk og derfor ikke bruke ROS-spesifikke funksjoner. I koden er det likevel inkludert en ROS klasse, topic_tools/shapeshifter.h [38]. Den tillater

oss å ta imot informasjon om hvilken som helst ROS melding, uavhengig av type. I dette tilfellet vil Singleton-klassen Logg, ved hjelp av ShapeSifter, hente informasjon om *topics*, meldingstype, størrelse og definisjon. Informasjonen blir slått sammen til en *string*, sammen med en *timestamp*, retning på melding og nettverksytelse, og deretter skrevet til en fil.

Mutex-lås

Å skrive til fil er en kritisk del av koden. Det er viktig at handlingen ikke blir avbrutt, samt at hvis flere skriver til samme fil samtidig vil løsningen bli korrupt. Det ble derfor klart at gruppen måtte finne en løsning for å sikre denne prosessen.
Gruppen fant to alternativer til dette problemet, lås-fri programmering eller mutex-lås. Lås-fri programmering går i hovedsak ut på å fordele endring av kode over flere tråder uten å bruke lås for å låse en prosess til en tråd av gangen. Dette lar seg gjøre ved å bruke forskjellige algoritmer. Fordelen er at parallelliteten øker fordi en unngår kø, men ulempen er at koden er vanskelig å implementere. På grunn av tidsmangel valgte gruppen å gå for mutex-lås [39] som synkroniserer trådene. Når en tråd tar i bruk prosessen for å skrive til fil, låses prosessen med mutex, og de andre trådene må vente inntil den første tråden er ferdig med prosessen og låser opp mutexlåsen igjen. Ulempen med mutex er at det kan skje en *deadlock* [40] eller det kan oppstå kø og hele prosessen vil da ta lengre tid enn ved en lås-fri kode.

9. Konklusjon

I denne bacheloroppgaven, som tar utgangspunkt i et allerede eksisterende system, er hovedutfordringen å ta høyde for hva som skjer med meldinger over nettverket når båndbredden blir lav. Oppgaven har også vært problemstillinger rundt det å kunne logge og lagre dataene som sendes, samt teste båndbredden.

Hele systemet er bygget på ROS, det har derfor vært nødvendig å forstå dette konseptet for å kunne komme frem til aktuelle løsninger. Ved å sette seg inn i ROS og dens funksjonaliteter, ble det klarere hvordan meldingsutvekslingen i dette systemet fungerte. Det ble bekreftet at all data som sendes i systemet går gjennom ROS. Det var derfor tenkelig at all datahåndtering i forhold til oppgavene ville bli implementert i ROS. Løsningene endte imidlertid opp med å være både innenfor og utenfor ROS.

9.1 Dynamisk dataoversikt

Det var et krav å få oversikt over all datakommunikasjon i hele systemet. Målet med denne oversikten er å bruke denne informasjonen til å foreta prioriteringer av gjennom prioriteringssystemet. For å få en oversikt over hele systemet i Coastal Shark, er det nødvendig å hente informasjon fra hver node. Oversikten må også være dynamisk, slik at informasjonen oppdateres kontinuerlig.

Løsningen ble å bruke ROS C++ API, som gjør det mulig å hente inn nødvendig informasjon over nodene i systemet og en Qt applikasjon som illustrerer en grafisk tabell. Med denne løsningen vil det være mulig å bruke informasjonen andre steder i systemet og eventuelt legge til informasjon for å utvide oversikten etter behov.

9.2 Prioritering

Som nevnt i begynnelsen av dette kapittelet, består dette prosjektet av flere deloppgaver. Den ene handler altså om å finne en løsning for hvordan meldingene håndteres når båndbredden endres. Kravet var å implementere et prioriteringssystem for datakommunikasjonen mellom C2 og Aegir. Det betyr at meldingene som sendes mellom disse stasjonene, må prioriteres i henhold til kapasitet på båndbredden og prioriteres på en slik måte at de til enhver tid viktigste dataene kommer frem, fremfor mindre viktige data. Det er imidlertid kun data som sendes fra Aegir som blir prioritert, ettersom all data fra C2 anses som like viktig. Løsningen ble å erstatte *multimaster*-oppsettet med en tilsvarende tjeneste som tar hånd om overføringen av meldinger, slik at den har full oversikt over meldingene som sendes. Løsningen åpner for muligheten å sende meldinger med UDP, slik at det ikke oppstår forsinkelser med opprettelse av forbindelser og handshakes. Ettersom det kan være ønskelig å sørge for at enkelte data kommer frem i sin helhet, støtter løsningen også TCP. I det tilfellet vil pakkene gå over en kobling som er opprettet mellom sender og mottaker, i motsetning til ROS sin måte, som ville vært en kobling mellom hver node.

9.3 Strupingsverktøy

Siden nettverksforbindelsen i systemet skal gå mellom en landstasjon og et fartøy som beveger seg i vann, er det et realistisk scenario at nettverksytelsen vil endre seg til stadighet. For å kunne teste nettverksytelsen, var det behov for et verktøy som kunne simulere dette. Løsningen her ble en virtuell tilnærming hvor èn maskin har to nettverkskort. Denne maskinen er da en egen enhet som kan ha forskjellige bruksområder.

Strupingsverktøyet er basert på biblioteket Traffic Control (TC) som er en funksjonalitet for å manipulere nettverksforbindelser på Linux kjerner. Dermed kan ma-

skinen kontrollere nettverksytelsen med TC kommandoer i Linux terminalen. De to nettverkskortene er koblet sammen med en bridge for å kunne kommunisere med hverandre. På denne måten vil forbindelsen gjøre at pakker blir sendt fra det ene kortet til det andre. Forbindelsen ble satt opp ved å endre nettverksfiler i Linux. Strupingsverktøyet er satt opp slik at det enkelt kan tas i bruk i andre systemer og andre maskiner.

9.4 Datainnsamling

Et annet krav var å lagre all data som sendes fra Aegir og skrive dette til en fil, som senere kunne brukes til et testscenario. Løsningen ble en node på Aegir som tar opp meldinger som publiseres fra andre noder på Aegir, altså diverse sensorer, og skriver innholdet til en fil. På C2 vil det være et brukergrensesnitt som gjør det mulig å velge om man ønsker å ta opp *topics* som publiseres fra nodene, og hvilke *topics* man eventuelt ønsker å ta opp. Når man velger å starte opptak av *topics*, vil noden på Aegir motta en vektor fra C2 og starte datainnsamlingen av de aktuelle *topics*. Innholdet i filen disse dataene lagres i, kan åpnes i et brukergrensesnitt på C2 og spilles av som et testsett sammen med en pc som simulerer Aegir.

9.5 Logging

Et av kravene var å kunne logge all datatrafikk som går over nettverket. Nettverket mellom C2 og Aegir er koblet sammen med en radiolink. Det er prioriteringssystemet som sender meldingene over denne linken, slik at loggesystemet må ta imot meldingene fra prioriteringen før de skrives til en fil. Den beste løsningen ble derfor å gjøre loggingen til en del av prioriteringssystemet, slik at det ikke er nødvendig å ha en egen node som må kobles sammen med resten av systemet. På den måten utelukkes eventuelle forsinkelser med ekstra oppkoblinger. Loggingen skal logge alle meldin-

ger som sendes og mottas både på C2 og Aegir, samt gjeldende nettverksytelse. Informasjon om meldingene og nettverksytelsen blir lagret i en fil.

9.6 Gruppens bidrag

Bachelorgruppen sitt bidrag i dette prosjektet har vært en utvidet funksjon av et allerede eksisterende system. Med dette bidraget vil systemet være mer pålitelig i forhold til meldingsoverføringer og det er utvidet med funksjoner som gjør det mulig å kjøre test scenarier, slik at man f.eks. kan oppdage hva som gjorde at noe feilet. Dessuten vil det være mer oversiktlig i forhold til å kunne se dataflyten i systemet.

Det bør nevnes at bachelorgruppen ikke har hatt tilgang til hele systemet i operativ drift, slik at det har ikke vært mulig å teste alle funksjoner i en reell situasjon. Det vil si at det ikke har vært tilgang til de faktiske meldingene som blir sendt fra sensorene på Aegir og dermed heller ikke muligheten til å teste funksjonene som håndterer disse meldingene, annet enn i simulerte tester.

9.7 Fremtidig arbeid

Det er nå tenkt at dataene som lagres hentes ut med USB-tilkobling. For fremtidig arbeid kan det være en fordel å se på mulige løsninger med en overføringsalgoritme som gjør det mulig å sende dataene direkte over radiolinken, slik at man ikke behøver å stoppe Aegir og koble seg til manuelt for å hente ut ønsket datainnhold.

På grunn av tidsbegrensning gjenstår også et GUI for strupingsverktøyet, loggingen og dataoversikten. Men dette ble ikke prioritert da alt har brukbare grensesnitt slik at det er fullt mulig å lese de dataene man er interessert i. All nedprioritert data blir også per i dag forkastet – fremtidig arbeid kan endre på dette for å lagre dataen til en fil slik at det kan bli hentet inn igjen senere.

10. Referanser

[1] *DATA SHEET [PRELIMINARY] NVIDIA Jetson TX2 System-on-Module*, NVIDIA Corporation, 2017. [Online]. Available: https://download.kamami.pl/p569306-DATASHEET-NVIDIAJetsonTX2System-on-Module.pdf

[2] "Git," https://git-scm.com/.

[3] "doxygen," http://www.doxygen.nl/.

[4] Q. Wiki, "Qt Creator," 2016. [Online]. Available: https://wiki.qt.io/Qt_Creator

[5] "Messenger," https://www.messenger.com/.

[6] "Discord," https://discord.com/.

[7] Atlassian. What is Kanban. Atlassian. [Online]. Available: https://www.youtube.com/watch?v=iVaFVa7HYj4

[8] ——. Kanban WIP limits. Atlassian. [Online]. Available: https://www.youtube.com/watch?v=zEJn6eQO6FE

[9] R. Osherove, "The art of unit testing," 2011. [Online]. Available: https://www.artofunittesting.com/definition-of-a-unit-test/

[10] "Programming languages — C++," International Organization for Standardization, Standard, ISO/IEC 14882:2017, Dec. 2017.

[11] "Maritime broadband radio 189 - spesifikasjoner," Kongsberg Seatex AS, Desember 2017.

[12] "Maritime broadband radio 144 - spesifikasjoner," Kongsberg Seatex AS, Desember 2017.

[13] L. Joseph, *Mastering ROS for Robotics Programming*, 2nd ed. Packt Publishing, 2018.

[14] R. Wiki, "ROS/TCPROS," 2013. [Online]. Available: http://wiki.ros.org/ROS/TCPROS

[15] ——, "ROS/technical overview," 2014. [Online]. Available: http://wiki.ros.org/ROS/Technical%20Overview

[16] A. M. W. Le Wang, "Detection of man-in-the-middle attacks using physical layer wireless security techniques," 2014. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1002/wcm.2527

[17] J. B.-C. M. G.-H. C. F.-L. Francisco J. Rodríguez-Lera, Vicente Matellán-Olivera, "Message encryption in robot operating system: Collateral effects of hardening mobile robots," 2018. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fict.2018.00002/full

[18] R. Wiki, "roscpp," 2015. [Online]. Available: http://wiki.ros.org/roscpp

[19] G. S. B. Ricardo Emerson Julio, "Dynamic bandwidth management library for multi-robot systems," pp. 2585–2590, 2015. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7353729

[20] B. Hubert, *send(2) - Linux man page*. [Online]. Available: http://man7.org/linux/man-pages/man2/send.2.html

[21] ——, *tcp(7) - Linux man page*. [Online]. Available: http://man7.org/linux/man-pages/man7/tcp.7.html

[22] "lz4 - extremely fast compression," https://lz4.github.io/lz4/.

[23] M. Johnson, P. Ishwar, V. Prabhakaran, D. Schonberg, and K. Ramchandran, "On compressing encrypted data," *IEEE Transactions on Signal Processing*, vol. 52, no. 10, pp. 2992–3006, 2004.

[24] B. Hubert, *tc(8) - Linux man page*. [Online]. Available: https://linux.die.net/man/8/tc

[25] X. V. Perez. Introduction. The Linux Fundation. [Online]. Available: https://wiki.linuxfoundation.org/networking/iproute2

[26] D. Wiki, "Bridgenetworkconnections," 2019. [Online]. Available: https://wiki.debian.org/BridgeNetworkConnections

[27] J. Dugan, *iperf(1) - Linux man page*. [Online]. Available: https://linux.die.net/man/1/iperf

[28] S. M. Van Jacobson, Craig Leres, "Manpage of tcpdump," 2020-03-02. [Online]. Available: https://www.tcpdump.org/manpages/tcpdump.1.html

[29] ——, "Manpage of pcap," 2020-01-29. [Online]. Available: https://www.tcpdump.org/manpages/pcap.3pcap.html#lbAS

[30] R. Wiki, "rosbag," 2015. [Online]. Available: http://wiki.ros.org/rosbag

[31] ——, "rosout," 2018. [Online]. Available: http://wiki.ros.org/rosout

[32] ——, "rqt_bag," 2015. [Online]. Available: http://wiki.ros.org/rqt_bag

[33] ——, “roslaunch,” 2019. [Online]. Available: http://wiki.ros.org/roslaunch

[34] J. B. Tim Field, Jeremy Leibs, “rosbag::recorder class reference,” 2020-03-22. [Online]. Available: http://docs.ros.org/melodic/api/rosbag/html/c++/classrosbag_1_1Recorder.html

[35] D. Stonier, “Qnode class reference,” 2015. [Online]. Available: http://docs.ros.org/hydro/api/qt_tutorials/html/classQNode.html

[36] Qt, “Qthread class,” 2020. [Online]. Available: https://doc.qt.io/qt-5/qthread.html

[37] YoLinux, “C++ singleton design pattern,” 2013. [Online]. Available: http://www.yolinux.com/TUTORIALS/C++Singleton.html

[38] R. Wiki, “topic_tools::shapeshifter class reference,” 2019-06-06. [Online]. Available: http://docs.ros.org/indigo/api/topic_tools/html/classtopic__tools_1_1ShapeShifter.html

[39] Cplusplus, “mutex::lock,” 2020. [Online]. Available: http://www.cplusplus.com/reference/mutex/mutex/lock/

[40] GeeksforGeeks. Introduction of deadlock in operating system. GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/introduction-of-deadlock-in-operating-system/

[41] OpenWRT. Linux packet scheduler. OpenWRT. [Online]. Available: https://openwrt.org/docs/guide-user/network/traffic-shaping/packet.scheduler.theory

[42] R. Wiki, “rqt graph,” 2018. [Online]. Available: http://wiki.ros.org/rqt_graph

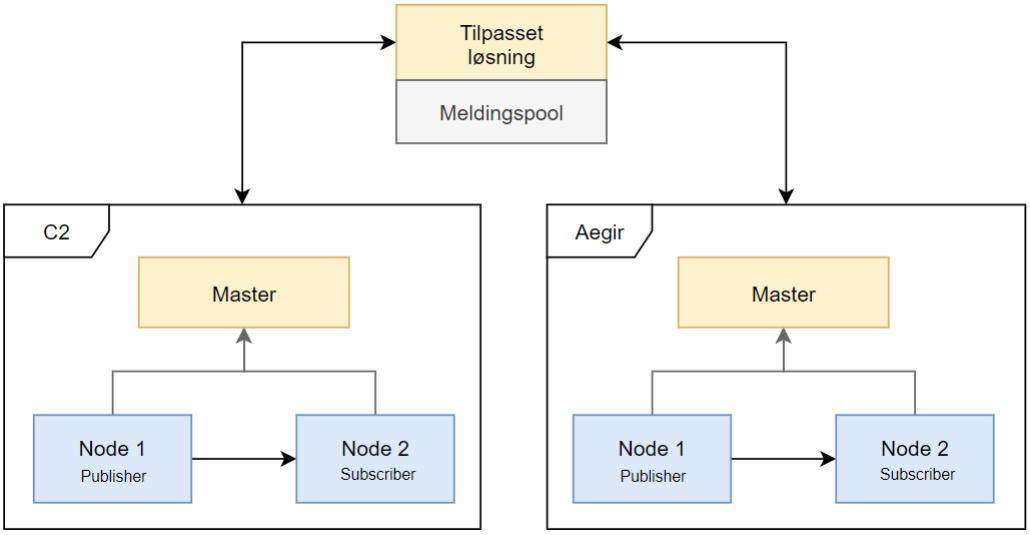
A. Løsningsforslag, prioritering

Formålet med dette dokumentet er å forklare de forskjellige løsningsforslagene til prioriteringssystemet som kom frem under prosjektet. Hver løsning blir forklart i detalj med sine fordeler og ulemper. De alternative løsningene fokuserer hovedsaklig på mer generiske implementasjoner som ikke bygger på ROS, hvor man prioriterer individuelle nettverkspakker istedenfor ROS-meldinger. Dette fører til andre implementasjonsproblemer, da det kreves en måte å markere pakker på slik at systemet vet hvilken prioritet de skal få.

A.1 Forslag 1: *Multimaster*-løsning

Per i dag bruker systemet et multimaster-oppsett via ROS. Det har en master for hver maskin i systemet: en for NUC-en, en for Nvidia Jetson-kortet ombord på Aegir, og en på landstasjonen, C2. En vanlig multimaster bygger direkte på *TCPROS* og vil dermed sende alle meldinger over *TCP*. I tillegg, hvis det finnes flere noder som abonnerer på samme *topic*, vil det også sendes flere kopier av samme melding. Det gjør den eksisterende løsningen uegnet for kommunikasjon over dårlige eller begrensede forbindelser.

Ved å erstatte dette multimaster-oppsettet med en tilpasset løsning, vil det være mulig å ta høyde for disse problemene. Den tilpassede løsningen vil være ansvarlig for å opprette koblinger mellom de forskjellige maskinene, og å transportere meldingene fra Aegir til C2. Figur A.1 viser et grovt konsept av en tilpasset løsning.



Figur A.1: Illustrasjon av meldingsflyten over topics via en tilpasset løsning.

Den vil dermed være ansvarlig for å velge hvilke meldinger den skal sende og hvordan meldingene skal bli sendt. Prioritet vil derfor være trivielt å implementere. Samtidig vil man ha fullstendig kontroll over hvordan meldingene blir sendt, og om de blir sendt. Dette er dermed en god løsning for å redusere forsinkelser i overføringen. Eksempelvis, så kan meldinger som ikke ansees som viktige bli sendt over *UDP*, mens for viktige meldinger kan *TCP* brukes. Alternativt kan *UDP* med bekreftelse (*ACK*) eller negativ bekreftelse (*NAK*) bli tatt i bruk.

I tillegg vil denne løsningene åpne opp for en dynamisk oversikt over all dataflyten i systemet. Til enhver tid har multimasteren styr på hvilke noder som kommuniserer med hverandre og hvilke type meldinger som blir sendt. Siden det er et av kravene til oppgaven, passer det utmerket.

Hovedfordelen med å gjøre det slik er at dette vil fungere som en *drop-in* erstatning for det nåværende multimaster-oppsettet. Ingen av den eksisterende koden trenger å endres, samt at man kan fortsatt forholde seg til ROS sin meldingsutveksling. Dette vil også gjøre det enklere å implementere et sikkerhetsaspekt på kommunikasjonen, for eksempel kryptering eller lignende.

Siden denne løsning kun fungerer innenfor ROS, må andre applikasjoner i systemet også gå gjennom ROS dersom de ønsker å overføre data. Per i dag skjer all kommunikasjonen gjennom ROS, så dette er ikke et problem nå, men det er noe som må tas hensyn til.

Tabell A.1: Fordeler og ulemper med *multimaster*-løsning.

Fordeler	• *Drop-in* erstatning for multimaster – ingen av den eksisterende koden trenger å endres. • Enkelt å ta frem en dynamisk oversikt over dataflyten. • Full kontroll over hvordan meldingene blir sendt, og kan derfor redusere forsinkelser. • Mulighet for å redusere duplisering av data som blir overført. • Brukeren kan velge mellom *UDP* og *TCP*, eventuelt *UDP* med *ACK* eller *NAK*.
Ulemper	• Fungerer kun innenfor ROS. Eksterne nettverkskoblinger vil derfor ikke kunne bli regulert. Kan være begrensende i fremtiden for prosjektet. • Nye maskiner må settes opp manuelt; eksisterende multimaster verktøy som gjør dette for deg kan ikke brukes.

A.2 Forslag 2: *Kernel Packet Scheduler*

De fleste operativsystemer har en *packet scheduler* som er ansvarlig for å velge hvilke nettverkspakker som blir sendt [41]. Når en applikasjon ønsker å sende data over en kobling, vil denne dataen bli lagt til i en intern kø. Ut fra denne køen er det opp til *scheduler*-en å velge hvilke pakker som skal sendes først. Hvordan denne køen blir organisert bestemmes av hvilken *queuing discipline* (QDisc) som blir brukt.

På mange vis kan det forklares som en multiplekser – flere signaler blir tatt i mot

på tvers av flere koblinger, før de slåes sammen og sendes over ett medium. Ved å implementere prioriteringen her vil man ikke trenge å tenke på hvordan *sockets* blir opprettet eller hvordan dataen blir sendt. Dette er hovedsaklig en ganske god løsning, men har et problem: køen har en fast størrelse. Dersom denne køen blir full, noe som kan skje, så vil påfølgende funksjonskall for å sende meldinger bli blokkerende. I tilfelle med ROS kan noder dermed stoppe opp som kan hindre hvordan resten av systemet fungerer.

Denne løsningen har mulighet til å prioritere all type trafikk, uavhengig om det kommer fra ROS eller ikke. Det krever derimot en måte å markere pakker på: hvis to forskjellige meldinger kommer fra samme node i ROS, så må løsningen ha en måte å kunne differensiere mellom disse. I tillegg har denne løsningen heller ingen måte å redusere forsinkelser i henhold til *TCP handshakes*, og har ikke muligheten til å forhindre overføringen av unødvendige kopier av ROS-meldinger.

Scheduler-en har allerede en prioritetsfunksjon, men denne er basert på *DSCP*-verdier som er satt i *IP-header* til hver pakke. Den tilbyr kun et begrenset antall prioritetsnivåer, og har heller ingen mulighet for å droppe nettverkspakker.

Tabell A.2: Fordeler og ulemper med *Kernel Packet Scheduler*.

Fordeler	• Generisk. Har et større bruksområdet siden det fungerer utenom ROS. Fremtidige prosjekter er dermed ikke avhengig av ROS. • Ingen av den eksisterende koden iht. ROS trenger å endres. • Eksisterende multimaster-oppsett kan bli brukt, samt tilhørende verktøy for konfigurering. • Full kontroll over hvilke meldinger som blir sendt, inkludert det internett bufferet.
Ulemper	• Ingen mulighet for å bytte mellom *TCP* og *UDP*. Dette må isåfall skje i ROS. • Ingen mulighet for å redusere duplisering av data. • Krever en tjeneste som markerer nettverkspakker med en prioriteringsverdi.

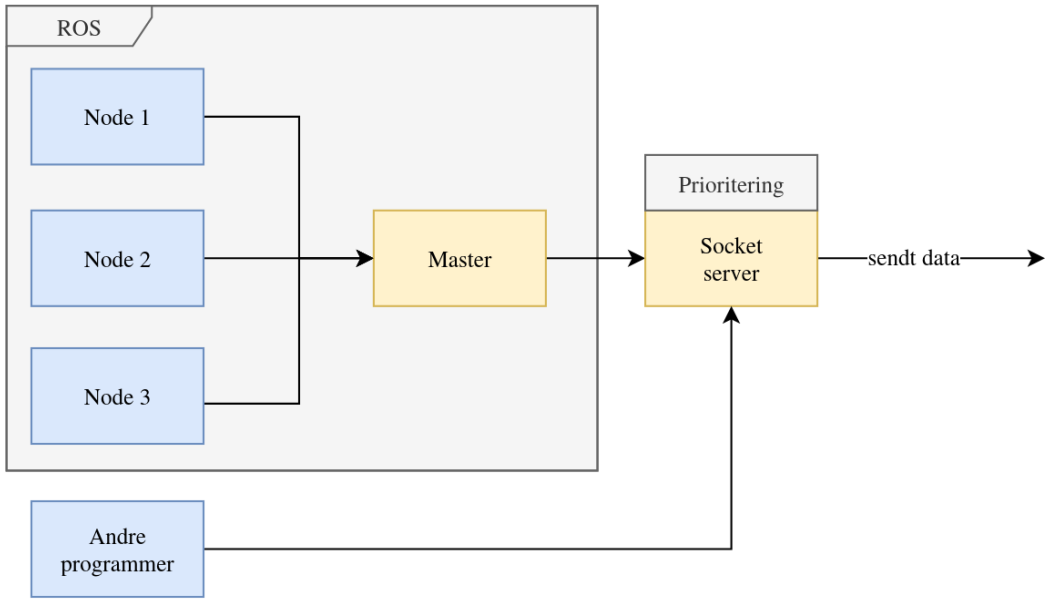
Denne løsningen vil også kreve endringer i kildekoden til kernelen som systemet kjører på. Dette gjør byggeprosessen og oppsettet til systemet ganske mye mer komplisert og tidskrevenede. I tillegg vil det kreve mer arbeid for å vedlikeholde systemet, og man kan ikke garantere at løsningen vil fungere når kernelen blir oppgradert til en nyere versjon. Etter å ha foreslått denne løsningen til oppdragsgiver, ble det gjort klart at de helst ville unngå å gjøre endringer i kildekoden til kernelen.

A.3 Forslag 3: *Proxy* Server

En annen generisk løsning er å lage en *proxy* server som fanger opp alle utgående koblinger. Hver applikasjon som prøver å sende data vil bli dirigert til serveren som mottar nettverkspakkene, legger de i en kø og velger deretter pakkene med høyest prioritet. Det krever at hver applikasjon på systemet kobler seg opp til serveren, noe som krever eksterne applikasjoner for å kunne dirigere hver kobling. Dette kan også

skape noen problemer for diverse applikasjoner. I ROS vil dette fungere fint siden meldingsutvekslingen mellom noder foregår over *sockets*, dog det vil kreve en liten endring på *multimaster*-oppsettet.

I likhet med forrige løsningsforslag krever denne også en måte å markere pakker på. Figur A.2 viser en illustrasjon av denne løsningen. Denne løsningen tilbyr ingen betydelige fordeler over den forrige løsningen.



Figur A.2: Illustrasjon av kommunikasjonen gjennom *socket* serveren.

Tabell A.3: Fordeler og ulemper med *proxy* server.

Fordeler	• Generisk. Har et større bruksområdet siden det fungerer utenom ROS. Fremtidige prosjekter er dermed ikke avhengig av ROS. • Kan fortsette å bruke en vanlig multimaster og tilhørende verktøy. • Full kontroll over hvilke meldinger som blir sendt.
Ulemper	• Krever en tjeneste som markerer nettverkspakker med en prioriteringsverdi, eller en intern mekanisme som analyserer hver pakke. • Ingen kontroll over hvordan meldingene blir sendt. • Ingen mulighet for å redusere duplisering av data. • Ingen kontroll over den interne *network scheduler*-bufferet.

A.4 Forslag 4: Virtuelt nettverkskort

En annen løsning, som ligner på de andre forslagene, er å implementere et virtuelt nettverkskort på hver maskin. Ved å sette dette til det standard nettverksgrensesnittet, så vil hver nettverkspakke bli sendt gjennom dette kortet. Dermed vil man få muligheten til å kontrollere hvilke nettverkspakker som blir videresendt til det virkelige nettverkskortet, som ville vært utgangspunktet for å implementere prioritering. Fordelen med denne løsningen er at den er lettere å bruke, og krever ingen tilpasning av hvordan data blir sendt i fra diverse applikasjoner.

I likhet med de andre løsningene har denne heller ingen måte å kontrollere hvilken protokoll som brukes til å overføre pakken.

Tabell A.4: Fordeler og ulemper med virtuelt nettverkskort.

Fordeler	• Generisk. Har et større bruksområdet siden det fungerer utenom ROS. Fremtidige prosjekter er dermed ikke avhengig av ROS. • Eksisterende *multimaster*-oppsett kan bli brukt, samt tilhørende verktøy for konfigurering. • Full kontroll over hvilke meldinger som blir sendt.
Ulemper	• Krever en tjeneste som markerer nettverkspakker med en prioriteringsverdi, eller en intern mekanisme som analyserer hver pakke. • Ingen kontroll over hvordan meldingene blir sendt. • Ingen mulighet for å redusere duplisering av data. • Manglende kontroll over interne buffere. • Vanskelig å implementere. Tidskrevende.

A.5 Sammenligning

Løsningene ble sammenlignet etter følgende kriterier:

- **Tid**: Hvor lang tid det estimeres at implementasjonen vil ta.
- **Fleksibilitet**: Til hvilken grad løsningen kan bli endret på for å møte andre krav.
- **Generisk**: Til hvilken grad løsningen kan brukes i andre prosjekter eller lignende, og hvor enkelt det ville vært å tilpasse.
- **Risiko**: Hvor stor risiko som er involvert i oppgaven – høy risiko medfører at implementasjonen ikke nødvendigvis blir ferdig i tide.
- **Kompleksitet**: Hvor vanskelig det vil være å implementere løsningen.

Tabell A.5: Sammenligning av de forskjellige løsningene for prioritering.

Kriterier	Vekt	Løsning 1: Multimaster	Løsning 2: Packet scheduler	Løsning 3: Socket server	Løsning 4: Virtuelt nettverkskort
Tid	5	3	2	2	1
Fleksibilitet	2	3	4	3	4
Generisk	1	2	5	5	5
Risiko	4	-2	-2	-3	-4
Kompleksitet	2	-2	-3	-1	-5
Sum		11	9	7	-8

A.6 Konklusjon

Etter å ha vurdert og sammenlignet de forskjellige løsningene, og etter å ha snakket med oppdragsgiver, ble det bestemt å gå videre med løsningen om å lage en tilpasset multimaster. Oppdragsgiver var mer interessert i å fullføre Coastal Shark og mindre interessert i generiske løsninger. Tiden kan dermed bli bedre brukt til å implementere en generisk oversikt over all datakommunikasjonen, og eventuelt fokusere på de andre delene av prosjektet.

De andre løsningene krever en ROS node for å kunne ta i mot konfigurasjonen fra C2, i tillegg til at de krever en måte å markere hver nettverkspakke på. Den enkleste måten å få til sistnevnte er å implementere et tilpasset *multimaster*-oppsett. Siden all datakommunikasjonen skjer gjennom ROS gir det dermed mest mening å implementere prioriteringen i ROS også, siden sluttproduktet blir det samme og det medfører mindre arbeid og vedlikehold.

B. Løsningsforslag, Dataoversikt

Dette vedlegget tar for seg de forskjellige løsningsforslagene som har blitt vurdert i henhold til kapittelet om dynamisk dataoversikt. Her vil hvert løsningsforslag bli forklart med både fordeler og ulemper. Til slutt vil en vekting bli anvendt på de forskjellige løsningsforslagene, hvor det vil bli tatt en beslutning på hvilken løsning som vil bli brukt for den dynamiske dataoversikten.

B.1 Forslag 1: *rqt graph plugin*

Et av forslagene for å fremstille en dynamisk dataoversikt var å bruke en *plugin* til ROS som heter *rqt graph*. Med en slik *plugin* ville gruppen ha oppnådd både en dynamisk grafrepresentasjon og fått muligheten til å presentere informasjon fra Coastal Shark systemet. Siden *rqt graph* har generiske komponenter [42] ville det også vært mulig å splitte opp systemet og grafrepresentasjonen til flere forskjellige ledd, som for eksempel et for Aegir, C2 og Coastal Shark som en helhet. I tillegg til dette, vil dataen kunne eksporteres til flere formater, som for eksempel et bildeformat.

En av ulempene med denne pluginen er at grafrepresentasjonen gir statisike snapshots. Det vil derfor være nødvendig å finne en mulighet for å kjøre en kontinuerlig beregning av grafen. Et annet er at brukergrensesnittet til grafrepresentasjonen er bygget inn i sitt eget brukergrensesnitt, så det vil også være nødvendig å finne en måte å få dette integrert inn i C2 sitt Qt *GUI*. Siden *rqt graph* sin kildekode er skrevet i Python kan det hende at dette byr på problemer, for ikke å nevne mulighetene for en fremtidig skalering av dataoversikten. Selv for at det trolig kan bli gjort mye ved å bygge et slags *interface* i C++ som binder resten av Coastal Shark prosjektet sammen med *plugin*.

Tabell B.1: Fordeler og ulemper med tilpasset rqt graph plugin.

Fordeler	• Grafrepresentasjonen er dynamisk og har sin egne geometriske løsninger for hvordan elementer i brukergrensesnittet ikke kolliderer. • Løsningen er basert på generiske komponenter. • Enkel å legge til på det nåværende systemet. • Kan produsere eksporterbare data (e.g. png filer).
Ulemper	• Produserer statisike snapshots av informasjonen. • Kan være nogenlunde innviklet å bygge den inn i C2 *GUI*. • Kildekoden til *plugin* er skrevet i Python.

B.2 Forslag 2: Tilpasset graf

Et annet løsningsforslag for fremstillingen av dataoversikten vil være å utvikle en helt egen og tilpasset graf. Ved hjelp av Qt finnes det store muligheter for å utvikle en grafpresentasjon som kan systematisk liste opp all informasjonen for dataoversikten. I tillegg, vil en slik graf også ha et hav av muligheter når det kommer til videreutvikling av systemet.

Ved å legge et slikt grunnlag, vil andre systemfunksjoner i Coastal Shark kunne blitt bygget på dette. Eksempler på dette inkluderer det å integrere andre deler av systemet inn i grafen, som blant annet prioritering-, logging- og strupingssystemet.

Ulempen med å lage en tilpasset graf er at det vil ta mye tid å utvikle den. For eksempel, så vil grafen måtte være dynamisk med at den tilpasser seg alt den skal representere. Det vil si at grafen skal kunne skalere seg opp og ned basert på alt den trenger og romme uten at noe kolliderer med hverandre. I tilegg til dette, må det utvikles en god form for navigasjon av grafen.

Tabell B.2: Fordeler og ulemper med tilpasset graf.

Fordeler	• Alt kan bli tilpasset akkurat slik en måtte ønske. • Etter å ha lagt et grunnlag burde det være nokså greit å skalere grafen. • Grafen ville vært bygget i Qt og med C++.
Ulemper	• Det vil ta mye tid å legge grunnlaget for grafen. Noe som vil inkludere slikt som geometriske løsninger for den dynamiske tilpasningen og navigasjonsmulighetene som grafen må ha.

B.3 Forslag 3: Tilpasset grafisk tabell

Som en erstatning for grafen vil en tilpasset grafisk tabell kunne utvikles. Tabellen vil da inneholde informasjon om dataoversikten og bli oppdatert kontinuerlig. Ved å utvikle dette i Qt burde det ikke ta lang tid å få frem en slik oversikt.

Det som derimot er negativt med en tabell er at den vil være ganske begrenset, med tanke på hva slags informasjon den viser. Den kunne for eksempel blitt utvidet med prioriteringssystemet, men noe utover det kan være vanskelig.

Tabell B.3: Fordeler og ulemper med tilpasset grafisk tabell.

Fordeler	• Vil være enkel og rask å lage. • Kan gi en nokså god dynamisk dataoversikt. • Prioriteringssystemet kan implementeres inn.
Ulemper	• Det kan være vanskelig å få utvidet tabellen til å inkludere andre ting. • Navigasjon av tabellen kan oppfattes som litt slitsomt siden den vil få et stort antall med rader.

B.4 Sammenligning

Løsningene ble sammenlignet etter samme kriterier som prioriteringsystemet, men med en annerledes vekting:

- **Tid**: Hvor lang tid det estimeres at implementasjonen vil ta.
- **Fleksibilitet**: Til hvilken grad løsningen kan bli endret på for å møte andre krav.
- **Generisk**: Til hvilken grad løsningen kan brukes i andre prosjekter eller lignende, og hvor enkelt det ville vært å tilpasse.
- **Risiko**: Hvor stor risiko som er involvert i oppgaven – høy risiko medfører at implementasjonen ikke nødvendigvis blir ferdig i tide.
- **Kompleksitet**: Hvor vanskelig det vil være å implementere løsningen.

Tabell B.4: Sammenligning av de forskjellige løsningene for dataoversikt.

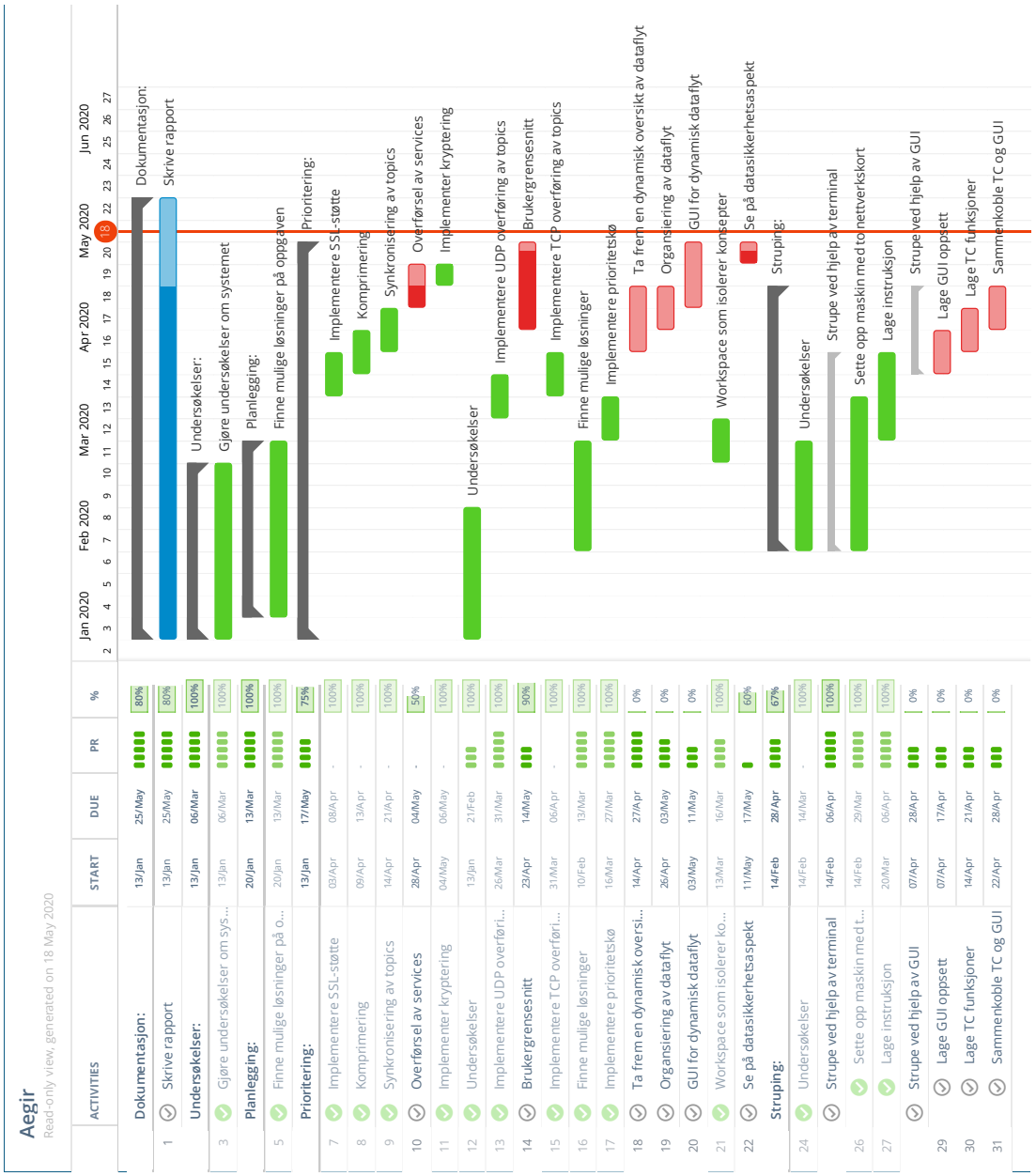
Kriterier	Vekt	Løsning 1: rqt plugin	Løsning 2: Tilpasset graf	Løsning 3: Tilpasset grafisk tabell
Tid	3	5	-4	4
Fleksibilitet	4	-5	5	2
Generisk	2	-4	5	-2
Risiko	5	3	-4	-2
Kompleksitet	2	-3	3	2
Sum		-4	4	10

B.5 Konklusjon

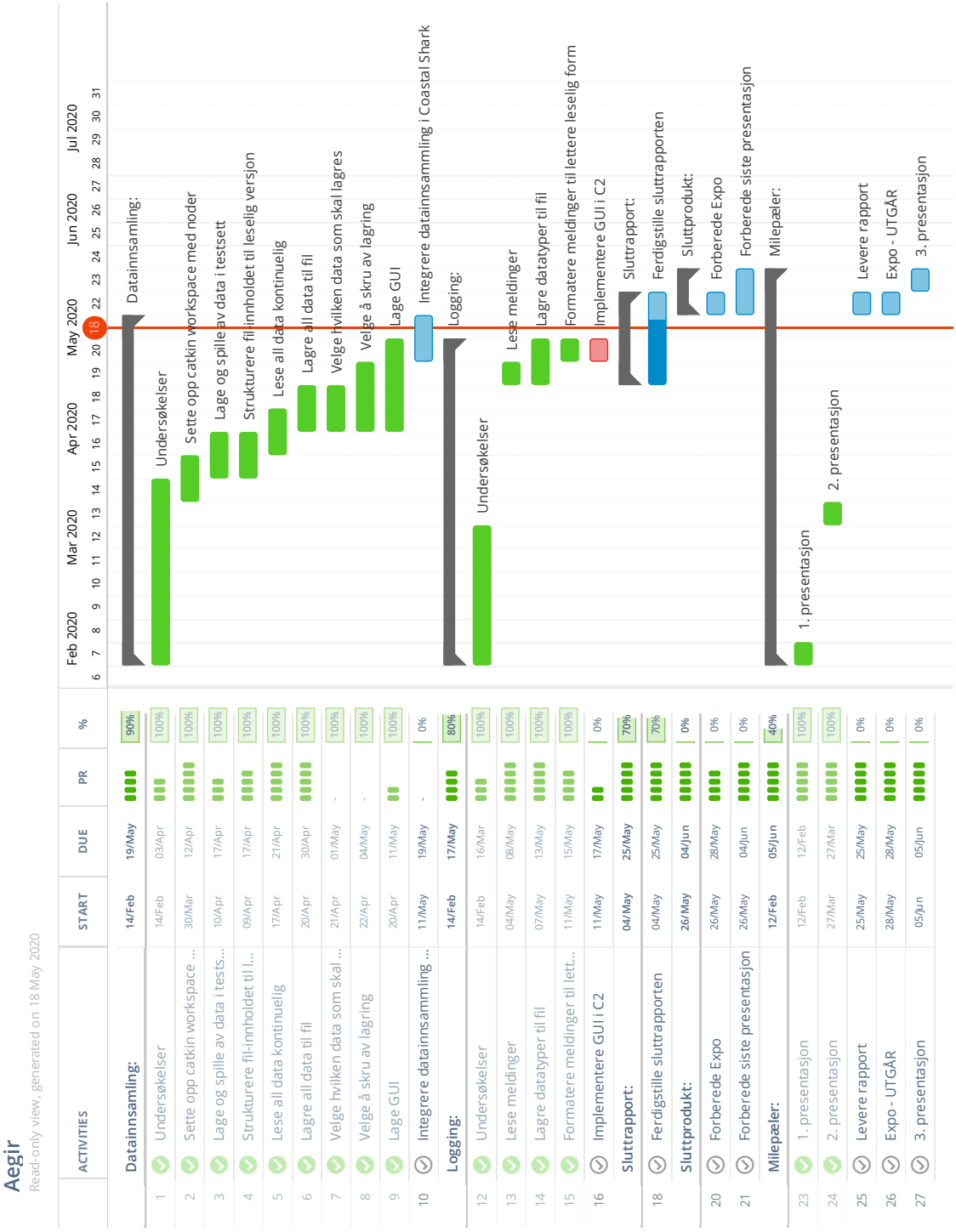
Det var først etter at gruppen hadde startet med å utvikle en tilpasset graf at det ble bestemt at det burde gjøres en sammenligning av forskjellige løsningsforslag på å ta frem dataoversikten. Det visste seg nemlig at det ville bli brukt mye tid på å utvikle denne, så en tidsfaktor ble en stor del dette systemkravet. Selv for at det allerede hadde blitt utviklet en god del av grafen, spesielt med tanke på oppsett og geometriske funksjoner.

Etter sammenligningen av de forskjellige løsningsforslagene ble det slik at vektingen falt i favør på en tilpasset grafisk tabell. Med denne løsningen ville det være mulig å gå bort fra det som hadde blitt gjort på grafen, og fortsatt oppnådd en god oversikt over all kommunikasjon i Coastal Shark. I tillegg, ville det åpnet for å implementere prioriteringssystemet inn og gjøre det mulig å ta med seg konseptene videre. Hvor det eventuelt kan bli brukt til å utvikle en tilpasset graf i en senere anledning.

C. Gantt-diagram



Figur C.1: Gantt-diagram side 1



Figur C.2: Gantt-diagram side 2

D. Lisenser

Navn	**Lisens**	**Offentlig**	**Iht.**
Roscpp	BSD	Ja	Dynamisk dataoversikt
Boost	Boost Software License	Ja	Prioriterings-systemet
OpenSSL	Apache License 2.0	Ja	Prioriterings-systemet
iperf	BSD	Ja	Strupingverktøyet

E. Regnskap

Dette vedlegget inneholder to tabeller som tar for seg regnskapet for alt utstyr som er kjøpt i løpet av dette prosjektet. En tabell for hva gruppemedlemmene har kjøpt og en for hva oppdragsgiver har kjøpt. Gruppen har stort sett hatt små utgifter siden gruppen utviklet et software-produkt og ikke et produkt som krever store materielle kostnader, dermed har vi ikke satt opp et budsjett og har hatt frihet til å handle nødvendige utstyr.

Tabellen inneholder utstyret gruppen har kjøpt så langt, hvilket gruppemedlem som har lagt ut for dette, dato, pris og den totale utgiften.

Tabell E.1: Regnskap for produkter betalt av gruppemedlemmene under prosjektet.

Dato	Vare	Salgssted	Antall	Betalt av	Pris
1/10/2020	Whiteboard tusjer	Brage bokhandel	1	Daniel Skryseth	99.00
1/10/2020	Postit lapper	Brage bokhandel	1	Daniel Skryseth	19.00
1/13/2020	Plastlommer	Brage bokhandel	1	Jannicke Løkaas	25.00
1/13/2020	Smalordner	Brage bokhandel	1	Jannicke Løkaas	39.00
1/13/2020	Hefte	Clas Ohlson	1	Jannicke Løkaas	39.90
1/13/2020	Kulepenn	Clas Ohlson	1	Jannicke Løkaas	39.90
1/13/2020	Grenkontakt 6-Veis. 3M	Biltema	1	Jannicke Løkaas	119.80
1/13/2020	Grenkontakt 5-Veis. 5M	Biltema	1	Jannicke Løkaas	59.90
1/13/2020	Buntebænd 100 stk.	Biltema	1	Jannicke Løkaas	29.90
1/13/2020	Plastbøtte	Biltema	1	Jannicke Løkaas	29.90
1/15/2020	Skjøtekontakt	Biltema	2	Tomas E.Jacobsen	45.80
2/7/2020	Postit lapper	Brage bokhandel	2	Bente Hestnes	30.00
2/7/2020	Postit lapper	Brage bokhandel	1	Bente Hestnes	25.00
2/14/2020	Writelatex Limited	Overleaf Online	1	Idar Carlsen	110.00
2/19/2020	Minnepenn Kingston	Brage bokhandel	1	Bente Hestnes	89.00
2/19/2020	Smalordner EMO 50mm	Brage bokhandel	1	Bente Hestnes	39.00
2/21/2020	Spiralhefte Studer S/Ruter	Brage bokhandel	1	Idar Carlsen	25.00
2/21/2020	Spiralhefte Studer S/Linjer	Brage bokhandel	1	Daniel Skryseth	29.00
3/4//2020	Notatbok Whitelines	Brage bokhandel	1	Øzlem Tuzkaya	45.00
3/14/2020	Writelatex Limited	Overleaf Online	1	Idar Carlsen	110.00
4/2/2020	Instagantt April abonnement	Instagantt Online	1	Bente Hestnes	∼325.44 (30$)
4/14/2020	Writelatex Limited	Overleaf Online	1	Idar Carlsen	110.00
5/2/2020	Instagantt Mai abonnement	Instagantt Online	1	Bente Hestnes	∼318.10 (30$)
5/14/2020	Writelatex Limited	Overleaf Online	1	Idar Carlsen	110.00
	Total sum:				**1,912.64**

Denne tabellen inneholder utgifter som oppdragsgiver har betalt for under hele prosjektet, antall varer som ble kjøpt, pris per vare med og uten Mva, og den totale prisen.

Tabell E.2: Regnskap for produkter betalt av KDA under prosjektet.

Dato	Vare	Salgssted	Antall	Pris per vare uten Mva	Pris per vare med Mva	Total pris
2/7/2020	Genser 4001 Belfast Marine	Kopisenteret Kongsberg As	6	598.00	897.00	4 485.00
	Total sum:					**4 485.00**