Benjamin Alexander Whittaker
Karl Andreas Wik Opheim
Simen André Dahl Jensen

Bacheloroppgave

Automatisk loggføring av veivedlikehold

Bacheloroppgave i Ingeniørfag, Data
Veileder: Tom Røise
Mai 2021

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

NTNU
Kunnskap for en bedre verden

Benjamin Alexander Whittaker
Karl Andreas Wik Opheim
Simen André Dahl Jensen

Automatisk loggføring av veivedlikehold

Bacheloroppgave i Ingeniørfag, Data
Veileder: Tom Røise
Mai 2021

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Kunnskap for en bedre verden

Sammendrag av Bacheloroppgaven

Tittel: Automatisk loggføring av veivedlikehold

Dato: 20.05.21

Deltakere: Simen Andre Dahl Jensen
Benjamin Alexander Whittaker
Karl Andreas Wik Opheim

Veiledere: Tom Røise

Oppdragsgiver: Electric Time Car

Kontaktperson: Dag Solhaug, dag.solhaug@eletrictimecar.com

Stikkord/Nøkkelord: IoT, CAN, Java, SQL, Backend

Antall sider: 81 **Antall vedlegg:** 8 **Publiseringsavtale inngått:** Åpen

---

Sammendrag: De siste årene har samfunnet utviklet seg mer og mer mot automatisering og effektivisering. Samtidig har interessen for IoT (Internet of Things) økt betraktelig. Electric Time Car (ETC) er et selskap, som ønsker å gi kunder mer oversikt over kjøretøysflåtene sine. På oppdrag fra ETC, skulle vi undersøke om automatisk loggføring av veivedlikeholdsoppgaver, er mulig ved hjelp av å lese CAN-data fra ulike nyttekjøretøy. Dette gjorde vi ved å utvikle en prototype som kunne fungere som utgangspunkt for en backend løsning for denne type loggføring.

Summary of Graduate Project

Title:	Automatic logging of road maintenance
Dato:	20.05.21
Participants:	Simen Andre Dahl Jensen Benjamin Alexander Whittaker Karl Andreas Wik Opheim
Supervisor:	Tom Røise
Employer:	Electric Time Car
Contact:	Dag Solhaug, dag.solhaug@eletrictimecar.com
Keywords:	IoT, CAN, Java, SQL, Backend

Number of pages: 81 **Number of appendix:** 8 **Availability:** Open

---

Abstract:

In the last decades, society has increasingly been moving towards automation and efficiency. Concurrently, the interest in the IoT (Internet of Things) has increased significantly. Electric Time Car (ETC) is a company which seeks to give their customers an improved overview of their vehicle fleets. On their behalf, we were to research whether automatic logging of road maintenance tasks is feasible by reading CAN data from the service vehicles. We solved this by developing a prototype software which may work as a backend solution to this kind of data extraction.

Forord

Vi vil gjerne takke Tom Røise, for god veiledning, tilgjengelighet og alt annen hjelp vi har fått.
Vi vil også takke Dag Solhaug og Øyvind Flatval i ETC for muligheten til å jobbe med et spennende prosjekt, og for å ha hjulpet oss i gang.

Innholdsfortegnelse

1 Innledning 7
1.1 Fagområde 7
1.2 Rapporten 9
1.3 Prosjektmål 10
1.4 Hvorfor valgte vi denne oppgaven 11
1.5 Målgruppe 11
1.6 Rammer 12
1.7 Kompetanse 12
1.8 Prosjektorganisering 14
2 Utviklingsprosessen 15
2.1 Utviklingsmodell 15
3 Kravspesifikasjon 21
3.1 Funksjonelle krav 21
3.2 Ikke-funksjonelle krav 26
4 Valg av teknologi 29
4.1 Valg av verktøy 29
4.2 Valg av enheter 31
5 Teknisk design 35
5.1 Overordnet Arkitektur 35
5.2 Design 36
5.3 Databaseoppsett 38
6 Implementering 41
6.1 Nettverk og sending av data 41
6.2 Protokoller og tolkning 49
6.3 Database 58
6.4 Ekstra kode og verktøy 64
6.5 Sikkerhet 68
6.6 Testing, kvalitetssikring 69
6.7 Installasjon av enhet 71
7 Diskusjoner og resultater 72
7.1 Resultater 72
7.2 Kritikk av oppgaven 74
7.3 Videre arbeid 76
7.4 Evaluering av gruppas arbeid 78

8 Konklusjon ... 79
9 Litteraturliste ... 80
10 Vedlegg ... 82
A. Ord Beskrivelse ... 82
B. Prosjekt Avtale ... 83
C. Prosjektplan ... 86
D. Prosjektbeskrivelse ... 99
E. Use Case ... 100
F. Referat til første møte med Arbeidsgiver 15.01.21 ... 103
G. Utdrag fra dokumentasjon generert av JavaDocs ... 104

Figurer:

- Figur 1: Skisse av systemet ... 8
- Figur 2: Organisasjonskart ... 14
- Figur 3: «Kanban»-tavle datert 19.03.2021 ... 19
- Figur 4: Gantt-diagram ... 20
- Figur 5: Use Case-diagram ... 22
- Figur 6: «Happy Day» scenario for normal flyt i programmet ... 25
- Figur 7: Enhetsløsning ... 34
- Figur 8: Dataflyt gjennom «Pipelines» ... 35
- Figur 9: Programmet sine moduler ... 36
- Figur 10: Resultatskisse av normaliserte tabeller ... 39
- Figur 11: Diagram av database ... 40
- Figur 12: Filstruktur i "Networking" Java-pakke ... 42
- Figur 13: Filstruktur i "Dataprocessing" Java-pakke ... 49
- Figur 14: Sekvensdiagram av TeltonikaFMC-protokoll ... 57
- Figur 15: Filstruktur av databasemodulen ... 58
- Figur 16: Den ferdigkonstruerte databasen ... 58
- Figur 17: Tabell med rader for IO-hendelser ... 59
- Figur 18: Fremmednøkkelverktøy i HeidiSQL ... 60
- Figur 19: Event i HeidiSQL ... 60
- Figur 20: Test data i GPS ... 61
- Figur 22: Skjermbilde av AVLCreator ... 64
- Figur 23: Skjermbilde av klientapplikasjon ... 65

- Figur 24: Eksempel på utforming av logg-fil ... 66
- Figur 25: Filstruktur av Unit-tester ... 69
- Figur 26: Skjermbilde av konfigurasjonsprogram ... 71

Tabeller:

- Tabell 1: «Scrum» og «Kanban»: Positivt og negativt ... 16
- Tabell 2: «Kanban»-tavle ... 18
- Tabell 3: Prioritetsnivåer ... 18
- Tabell 4: Trusler som programmet kan utsettes for ... 28
- Tabell 5: Lagret data ... 38
- Tabell 6: Codec 8E format ... 54
- Tabell 7: Codec 8E forklaring ... 54

Kodesnutter:

- Kodesnutt 1: Packet-klasse ... 43
- Kodesnutt 2: Utkast av Klient-klasse ... 44
- Kodesnutt 3: Utkast av Lytter-klasse ... 45
- Kodesnutt 4: Utkast av Server-klasse ... 46
- Kodesnutt 5: Utkast av TeltonikaFMCPacket-subklasse ... 47
- Kodesnutt 6: Utkast av TeltonikaFMCClient-subklasse ... 48
- Kodesnutt 7: Utkast av TeltonikaFMCListener-subklasse ... 48
- Kodesnutt 8: Protocol-Interface ... 51
- Kodesnutt 9: ProtocolHandler-klasse ... 52
- Kodesnutt 10: «processHandshake»-funksjonen ... 55
- Kodesnutt 11: Ekempel på dekoder-funksjoner ... 57
- Kodesnutt 12: Eksempel på SQL-spørring ... 62
- Kodesnutt 13: Logger-klasse ... 67
- Kodesnutt 14: Unit-test av Teltonika-protokollen ... 69

1 Innledning

1.1 Fagområde

1.1.1 Bakgrunn

Tingenes internett (Internet of Things - IoT) er et nettverk av gjenstander, der gjenstandene er utstyrt med sensorer, programvarer og andre teknologier som gjør de i stand til å kommunisere med hverandre og utveksle data [1].

Controller Area Network (CAN) er en standard designet for å la mikrokontrollere kommunisere gjennom en felles bus uten en sentral datamaskin. De fleste kjøretøy i dag har et slikt nettverk, og det er mulig å plukke opp signal fra nettverket og tolke signalene som virkelige hendelser gjort av kjøretøyet [2].

Electric Time Car AS (ETC) er et IT-selskap som leverer totalløsninger for helhetlig kjøretøyoppfølgning kalt CarAdmin. CarAdmin tilbyr flere moduler og funksjonaliteter for enkel drift av en organisasjon sine kjøretøy, alt fra bilnøkkelhåndtering til flåtestyring [3].

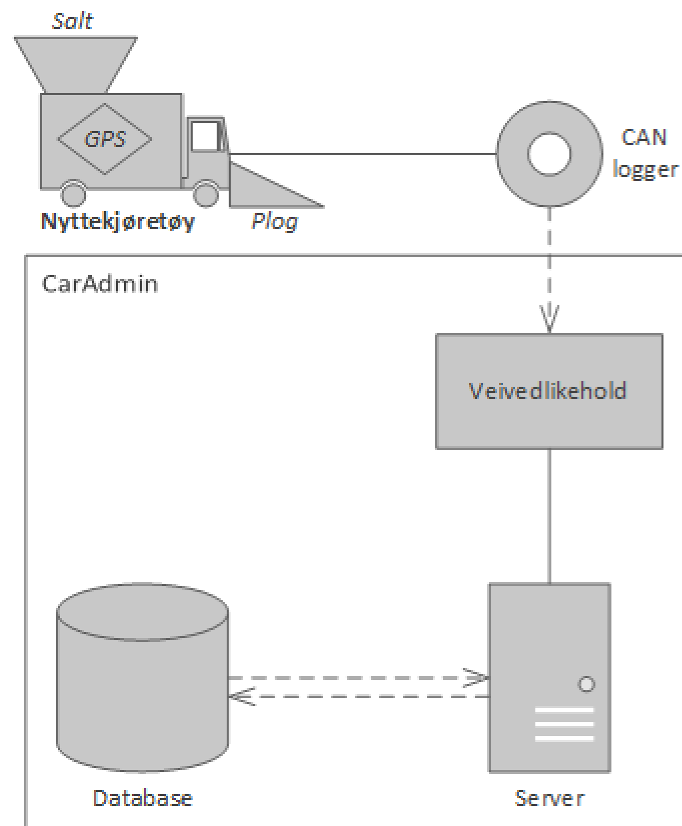
I denne bacheloroppgaven skulle vi utvikle en prototype som skulle kunne være et utgangspunkt for en backendløsning til veivedlikeholdsmodulen i CarAdmin[1]. Vi skulle innhente data fra kjøretøy ved lesing av kjøretøyenes CANbus-system, for så å tolke og lagre disse dataene i CarAdmins systemer for videre utnyttelse.

1.1.2 Oppgavebeskrivelse

Oppgavens omfang var dynamisk i størrelse og kunne gi muligheter til utvidelser hvis prosjektperioden tillot det. Ifølge prosjektbeskrivelsen i vedlegg D skulle vi levere og/eller vise til en prototype av et system som leser av data fra nyttekjøretøy og sender dette til en sentral database. Dette skulle fungere som en backendløsning til veivedlikeholdsmodulen i CarAdmin-løsningen til ETC.

[1] https://caradmin.no/veivedlikehold/

Basert på første utkast av oppgavebeskrivelsen fra ETC, så vi at prosjektet kunne deles opp i tre hoveddeler. Først måtte vi velge enhetsteknologien og enhetsmodellene som oppfylte kravene til ETC. Etter disse var bestilt og på vei, måtte vi utvikle programvare som kunne koble seg mot enhetene over mobilnettet, motta, tolke og formatere dataen, og lagre denne dataen i en sentral database. Til slutt måtte vi sørge for at enhetene ble installert i et eller flere kjøretøy, slik at vi kunne teste programvaren opp mot et reelt testmiljø. Av disse 3 delene så var det del 2 vi brukte mest tid på.



Figur 1: Skisse av systemet

Figur 1 viser en enkel skisse av systemet. På toppen er det avbildet et nyttekjøretøy, som har ulike typer utstyr. I dette eksempelet er det en brøytebil med plog og saltspreder. Igjennom en CAN-logger og evt. flere enheter på kjøretøyet, plukkes det opp data fra CANbus-systemet til kjøretøyet som f.eks. om det brøytes og hvor mye salt det er igjen i saltbeholderen. GPS og tidsstempelet blir også logget, og dette i sammen med CAN-dataen blir sendt over internett til en server, i en viss oppdateringsfrekvens. Her blir dataen tolket og verifisert på serveren, hvor den så blir formatert til et mer leselig format. Dette blir til slutt lagret på en sentral database, der dataen kan bli benyttet til ETCs formål.

1.1.3 Avgrensning

Vi skulle kun fokusere på nyttekjøretøyene som drev med sporing, salting og brøyting. Vi skulle kun lage en prototype som kunne kommunisere, tolke og lagre dataen. Det skulle ikke utvikles noe frontendløsning. Kommunikasjonen med den valgte enheten skulle kun implementeres ved hjelp av TCP. Databasen skulle opprettes og lagres lokalt hos oss, med ingen tilknytting ETC infrastruktur.

1.2 Rapporten

For å ha en god forståelse av rapporten bør leser ha grunnleggende kompetanse innen informatikk, systemutvikling og de tekniske begrepene som benyttes. Dette er for å forstå de teknologiene og verktøyene som er valgt, samt arbeidsmetodikkbegrepene som blir nevnt.

1.2.1 Merknader om prosjektet

Vi vil gjennom rapporten referere til en enhet installert på et kjøretøy som en «Enhet». Dette er en forenkling av virkeligheten, der det ofte er én enhet som leser CANbus-dataen og en annen som sender dataen til en server. For å slippe forvirring rundt dette bruker vi kun entall for å omtale en klynge av enheter som jobber i lag om oppgavene.
Andre begreper som er viktige for prosjektet er listet i ordlisten som er under Vedlegg A.
Vi vil også nevne at prosjektplanen i Vedlegg C er utdatert, og eventuell informasjon vi bruker fra denne er oppdatert i selve rapporten.

1.2.2 Rapportstruktur

Rapporten er delt inn i 8 hovedkapitler.

1. **Innledning**: Forteller om oppgavebeskrivelse, mål, rammer og bachelorgruppen.
2. **Utviklingsprosessen**: Beskriver hvordan vi valgte og skal bruke utviklingsmodellen.
3. **Kravspesifikasjon**: Inneholder krav og andre beskrivelser fra arbeidsgiver.
4. **Valg av teknologi**: Våre valg av verktøy, produsent og enheter
5. **Teknisk design**: Forklaring av modeller, systemarkitekturer og databasestruktur.
6. **Implementering**: Beskriver utviklingen av prosjektet. Koding, testing og installasjon.
7. **Diskusjon og Resultater**: Resultatet vårt. Kritikk av oppgaven og videre arbeid
8. **Konklusjon**: Konklusjonen på om målet til oppgaven er nådd

1.3 Prosjektmål

Formålet med prosjektet fra ETC sitt perspektiv, var å få innsikt i hvordan de automatisk kunne hente og lagre veivedlikeholdsdata fra en flåte med kjøretøy. Dette skulle kunne potensielt være en utvidelse til CarAdmin-tjenesten som ETC leverer i dag. For oss som studenter er prosjektets formål å levere et produkt som følger ETCs kravspesifikasjon, men har også som formål å gi oss læringsutbytte fra utviklingsprosessen. Dette i form av å praktisk bruke arbeidsmetodikken vi har lært gjennom studiet, samt å få utvidet kompetanse ved å lære prosjektspesifikk kunnskap. Siden vi kun skulle utvikle en prototype, vil vi ikke se prosjektets direkte effekter for ETC sine gevinster. Vi endret derfor på vinklingen til effektmålene til å fokusere på hva ETC kunne videreutvikle ut ifra vår prototype.

Effektmål

- Kunne visualisere sjåførens aktivitet på kjøretøyet under arbeidsdriften.
- Overvåke nyttekjøretøysstatus og vedlikehold i sanntid.
- Levere et flåtestyringsverktøy for nyttekjøretøy i både privat og offentlig sektor.
- Gjøre veivedlikeholdsdata tilgjengelig for offentligheten, gjennom et publikumskart.

Resultatmål

- Demonstrere hvordan data fra et kjøretøys CANbus-system kan formateres til et lesbart format og videre sendes til ETCs database til videre bruk.
- Levere et system som kan være et utgangspunkt for en backendløsning for veivedlikeholdsmodulen i CarAdmin.
- Konkludere om lesing av CANbus-systemet er en brukbar løsning for loggføring av vedlikeholdsoppgaver.

Læringsmål

- Erfare å lage et system som består av flere ulike teknologier, deriblant IoT-teknologi, dataprotokoller og database-teknologier.
- Erfare å jobbe med et større prosjekt, hvor man bruker moderne systemutviklingsmetodikk, sammen med en reell arbeidsgiver.
- Få praktiske erfaringer ved å jobbe med maskinvare som kommuniserer over internett og få en dypere forståelse for feltet «Internet of Things».

1.4 Hvorfor valgte vi denne oppgaven

Når gruppen for første gang satt seg sammen for å velge mellom de aktuelle bacheloroppgavene som universitetet hadde mottatt fra utvalgte arbeidsgivere, så hadde vi en felles konsensus i arbeidstype og fagområde som de prioriterte oppgavene skulle ha.

- Dette var blant annet **programvareutvikling**, ettersom vi ønsket å utvikle et produkt gjennom programmering.
- Oppgaven skulle være **studierelevant**, slik at vi i arbeidsprosessen kunne benytte kunnskap og erfaringer som vi har opparbeidet oss i løpet av utdanningen.
- Arbeidsgiver skulle være **nyskapende**, valg av prosjektrammer, som verktøy, skulle være etter moderne standarder og at arbeidsprosessen ville gi oss erfaringer som vi kunne ta med oss ut i arbeidslivet.

Av disse grunnene satt vi ETC sin IoT-oppgave høyt på lista, fordi den traff godt på alle de egenskapene vi ønsket, samtidig som arbeidsgiver var i nærmiljøet som gav gode muligheter for et tett samarbeid.

1.5 Målgruppe

1.5.1 Målgruppe for produktet

Produktets målgruppe var ansatte i ETC. Prototypen vilel bli hovedsakelig brukt av selskapet som et konseptbevis og en demonstrasjon av en mulig løsning på automatisk loggføring av vedlikeholdsoppgaver. Produktet skulle også eventuelt videreutvikles av ETC sine utviklere for å støtte flere enhetsfamilier og protokoller enn de som skulle utvikles i dette prosjektet.

1.5.2 Målgruppe for rapporten

Rapporten er rettet mot personer som er interessert i temaet og teknologien som er benyttet eller personer som ønsker å utvikle tilsvarende produkter. Rapporten kan også være til interesse for framtidige studenter som skal gjennomføre en lignende bacheloroppgave, eller trenger inspirasjon til rapportoppsett og prosjektutførelse.

1.6 Rammer

1.6.1 Fremdriftsplan

NTNU satt fristen på levering av rapporten til 20. mai. Innen den tid måtte kildekode, dokumenter og andre krav være oppfylt.

1.6.2 Teknologisk

Enheter

I starten av prosjektet fikk vi i oppgave å finne passende enheter som skulle installeres i diverse kjøretøy for å samle data og sende dette til en server. Enhetene skulle bestilles, leveres, og installeres i passende kjøretøy. Dette betydde at vi ikke kunne teste programmet på reel data i sanntid før dette var gjort ferdig.

Programmeringsspråk og database

Som et utviklingskrav fra ETC så ønsket de at vi skulle bruke Java som programmeringsspråk og MariaDB som database. Dette er fordi da speiler den teknologien som de selv bruker som gjør gjenbruk enklere for dem. Som versjonskontrollverktøy så var det også satt krav å bruke et som de selv brukte, dette endte opp med å bli GitLab-serveren deres.

1.7 Kompetanse

1.7.1 Egen bakgrunn og kompetanse

Gruppen bestod kun av dataingeniørstudenter så alle hadde en felles grunnkompetanse i de samme fagområdene. Områdene hvor ekspertisen varierte, stammet fra ulikt valg av valgfag og kunnskaper innhentet gjennom hobbyprosjekter, som f.eks. spillutvikling.

Programmeringskompetansen i gruppen kom ifra flere emner vi har hatt opp igjennom studiet. I disse fagene har vi fått kompetanse i flere ulike programmeringsspråk som f.eks. C++, Python og Java. Samtidig har vi hatt flere fag som har fokusert på forståelsen av algoritmer og metoder for programmeringsarkitekturer.

Databasekunnskapen i gruppen kom ifra et databaseemne som alle hadde vært igjennom, noe som gjorde det lettere å få en felles forståelse av hvordan tabeller og relasjoner skulle struktureres. Vi hadde også erfaringer med å koble og skrive til database ved bruk av SQL-spørringer i Java fra noen av programmeringsfagene.

Git som versjonskontrollverktøy hadde vi benyttet oss av i de fleste programmeringsfagene igjennom plattformene «GitHub», «GitLab» og «Bitbucket». Dette ble brukt for å programmere i sammen med flere studenter i samme prosjekt, og vi har god erfaring med dette i både skolesammenheng, og gjennom hobbyprosjekter.

1.7.2 Kunnskap som måtte innhentes

Produkter og protokoller fra selskapet Teltonika måtte vi få kjennskap til tidlig i prosjektet, ettersom vi valgte dette selskapet som produsent, som beskrevet i 4.2. Vi måtte lese oss opp på produktene, for å finne riktige enheter. Kunnskapen om protokollene opparbeidet vi gjennom Teltonikas nettsider og ved å få tilsendt et utviklerverktøy og dokumentasjon fra en av deres kundebehandlere. Vi trengte denne kunnskapen for å kunne lage funksjoner som tolket Teltonika sine datapakker, og for å lage verktøy som kunne lage datapakker for å teste programmet.

Cyclic Redundancy Check (CRC) var et nytt konsept for oss som vi måtte få en dypere forståelse av når vi skulle lage gyldighetstester for innkommen data. CRC er en metode for å sjekke om data er endret eller ødelagt i løpet av en dataoverføring [4]. I Teltonika sitt tilfelle, presiserer dokumentasjon at datapakkene bruker CRC16/IBM [5]. Vi måtte lære oss hvordan CRC fungerte, slik at vi kunne sjekke om dataen vi mottok fra enheten, var lik som da den ble sendt. Vi måtte også gjenskape CRC-metoden i verktøyet vi utviklet for generering av test-datapakker.

Controller Area Network (CAN) måtte vi få kunnskap til ettersom vi ikke hadde erfaring på dette området fra før. Det finnes flere standarder for dette, men i vårt tilfelle var det SAE J1939 som var aktuell for vårt prosjekt, ettersom den er ment for store kjøretøy som buss eller lastebil [6]. Informasjonen om standarden, gav oss grunnlaget for å kunne velge en egnet enhet under fasen der vi valgte enheter til prosjektet.

1.8 Prosjektorganisering

1.8.1 Prosjektroller

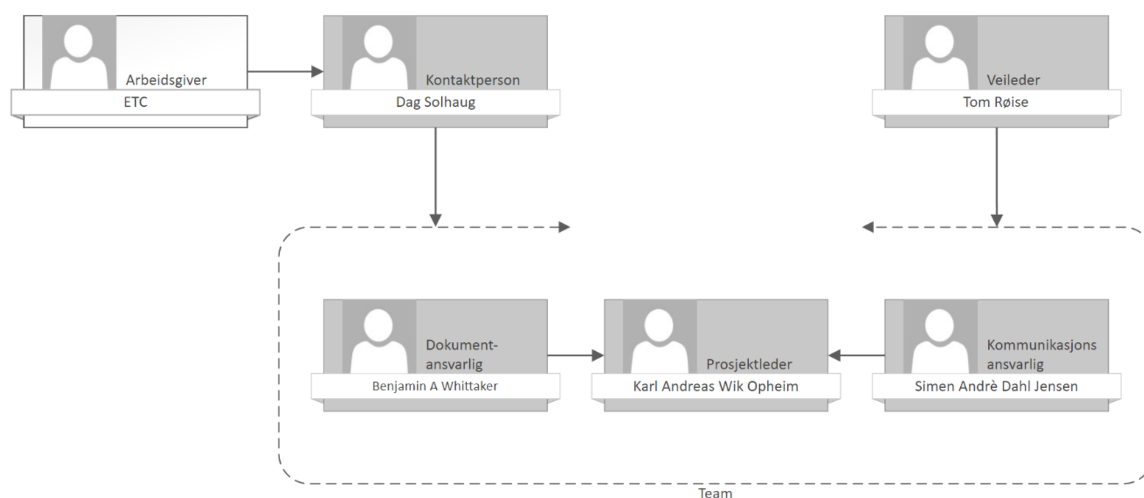
Karl Andreas ble utpekt som prosjektleder, ettersom han hadde ledererfaring i fra flere fritidsaktiviteter. Vi ble enige om at prosjektlederen hadde hovedansvar for at alle i gruppa fulgte gruppereglene og passet på at det ble jobbet målrettet frem mot leveringsfrist.

Benjamin fikk hovedansvaret for dokumentasjon. Han skulle passe på at alle fulgte kodekonvensjonen, brukte Git- og Google Disk-verktøyene korrekt og at rapporten ble skrevet i riktig format og i samsvar med planlagt oppsett.

Simen hadde hovedansvaret for møteinnkalling med veileder og arbeidsgiver, skrive referat om det var noen som ikke kunne møte på avtalt tidspunkt, og hadde ansvaret for å kontakte tredjepartsbedrifter om dette viste seg å være nødvendig.

Selv om hvert medlem i gruppa hadde fått en rolle, så betydde ikke dette at de hadde ansvaret alene om oppgavene sine. Vi hadde kun hovedansvar for vår oppgave, men alle de andre kunne komme med forslag, varsle og passe på at alle utførte oppgavene korrekt.

Vi lagde et organisasjonskart for å vise et syn over prosjektrollene i prosjektet. Se Figur 2.



Figur 2: Organisasjonskart

1.8.2 Utviklingsroller

Vi delte ikke ut noen utviklingsroller, men ønsket at alle var med på alt, slik at vi kunne få en god forståelse for programmet, oppgavene og dokumentasjonen. På denne måten ville alle få god kunnskap om emnene, og gjøre slik at alle kunne ta over en oppgave om noen skulle bli syke.

2 Utviklingsprosessen

2.1 Utviklingsmodell

Når vi skulle vurdere hvilken utviklingsmodell vi ville følge i utviklingsprosessen, så fant vi fort ut at vi ville bruke en smidig utviklingsmodell. I samtaler med arbeidsgiver og veileder, var de enige i dette. Smidig metodikk kjennetegnes med at det som utvikles blir utviklet i flere inkrementer, og kravene blir vurdert kontinuerlig [7]. Som beskrevet i 1.1.2, så kunne vi dele prosjektet i 3 hoveddeler. Disse delene kunne vi dele inn i flere deler igjen, som blir mer beskrevet senere i 2.1.2. Vi hadde derfor naturlig inkrementer i utviklingen. I tillegg til dette, så var kravene fleksible, og mye var opp til oss som utviklere å bestemme. Vi leste oss opp på flere smidige utviklingsmodeller, og bestemte oss for å gå for «Scrum» eller «Kanban». «Scrum» fordi utviklingsmodellen fokuserer mye på samarbeid, og organisering. «Kanban» fordi vi liker å kunne se den visuelle fremgangen ved å bruke ei «Kanban»-tavla, og at det passer for små grupper. Vi leste oss opp på hver av modellene, og identifisert noen fordeler og ulemper med hver av dem, i forhold til vårt prosjekt [8] [9]. Disse er vist i Tabell 1.

Tabell 1: «Scrum» og «Kanban»: Positivt og negativt

«Scrum»	«Kanban»
Hyppige møter Mindre sjanser for misforståelser	**Vi kan velge oppgaver som passer fra «Kanban»-tavla** Vi kan selv velge oppgaver som passer vår kompetanse
Lett å se hvor man er i prosjektet Oversikt over hvor vi ligger an i prosjektet, grunnet flere møter	**Ser visuelt arbeidsflyten** «Kanban»-tavla viser hva som skal gjøres og hva som er gjort
Kan bli for mye «Scrum» er beregnet for grupper på 5-7 medlemmer, og det kan bli litt mye opplegg for liten gruppe på 3	**Ikke veldig organisert** Har ikke klare roller og ansvarsområder
Drukning i møter Mange møter, men vi setter pris på å effektivisere utviklingen og ikke bruke for mye tid på møter, og heller ta møter som det kommer	**Vanskelig å se hvor langt man er i prosjektet** Siden prosjektet sakte bygges opp av «User Stories», er det vanskelig å se hvor mye som mangler

Etter å ha satt «Scrum» og «Kanban» opp mot hverandre kom vi fram til at begge hadde egenskaper som var svært ettertraktet i arbeidsprosessen, men de hadde også egenskaper som nesten utelukket dem som alternativ.

«Scrum» har den gode egenskapen at det er lett å følge planen og ikke grave seg ned i en grop, ettersom man har mange milepæler og møter. Likevel kan mengden med møter og organisasjonsarbeid gjøre det vanskelig å fordele tid og arbeid for en gruppe på 3 personer.

«Kanban» har den nyttige egenskapen at utviklingsprosessen blir effektiv, ettersom vi kan ta oppgaver som passer oss til enhver tid fra «Kanban»-tavla. Derimot er det lett å miste helhetsbilde, og man kan grave seg ned i en grop.

Etter diskusjon og diverse undersøkelser, fant vi ut at det gikk an å bruke en utviklingsmodell, som er en slags kombinasjon av «Scrum» og «Kanban», kalt «Scrumban». Og vi vil nå gå igjennom hvordan vi planla å ta i bruk denne i prosjektet.

2.1.1 «Scrumban»

«Scrumban» kombinerer strukturen til «Scrum», og arbeidsflyten til «Kanban» [10]. Vi bestemte oss for å plukke det beste av hver av dem. Likevel gikk vi til noen kilder, for å se hva som var lurt å ha med i "Scrumban". I følge «Kanban and Scrum - Making the Most of Both» og «What is Scrumban?», så legger begge tekstene vekt på at det er viktig med «Retrospectives» [10] [11]. Vi ville ha så få interne møter som mulig, men tenkte det var bra å ha et «Retrospective Meeting» hver uke for å se hva som var gjort bra og hva vi kunne gjøre bedre.

Utenom dette så ville vi bruke «Scrum» sin inkrement-struktur, der vi planla en periode der vi skulle jobbe med en del av prosjektet.

En forskjell på «Scrum» og «Kanban» er at i «Scrum» så blir arbeids-tavla, tømt for hver iterasjon [11]. Her valgte vi å heller bruke «Kanban»-tavla fordi vi ønsket å se alle oppgavene som var gjort for bedre oversikt hva som er gjort, og gjøre det mulig å åpne gamle oppgaver. Når det gjelder oppgaver i tavla, så tenkte vi å legge til flere oppgaver når vi så at det var lite i backloggen (3-4 oppgaver).

Vi skrev i tabellen at det var negativt at «Kanban» ikke var så organisert, men ettersom vi var få på gruppen, så vi ikke nødvendigheten med å spesifisere den individuelle rollen til hver deltager, ettersom dette er en typisk «Scrumban»-organisering.

Oppsummert tenkte vi en arbeidsuke slik:

Mandag morgen: «Retrospective Meeting» for å se hva som er bra og hva som kan bli bedre
Mandag-fredag: Faste arbeidstider, der vi jobber i sammen, og velger oppgaver fra «Kanban»-tavla.

«Kanban»-tavle

Etter å ha definert hva vi ønsket å bruke i utviklingsmodellen, definerte vi hvordan oppsettet av «Kanban»-tavla skulle være. Vi ville dele opp tavla i flere deler, slik at vi kunne få oversikt hvor en ppgave lå i prosessen. Disse er listet i Tabell 2.

Tabell 2: «Kanban»-tavle

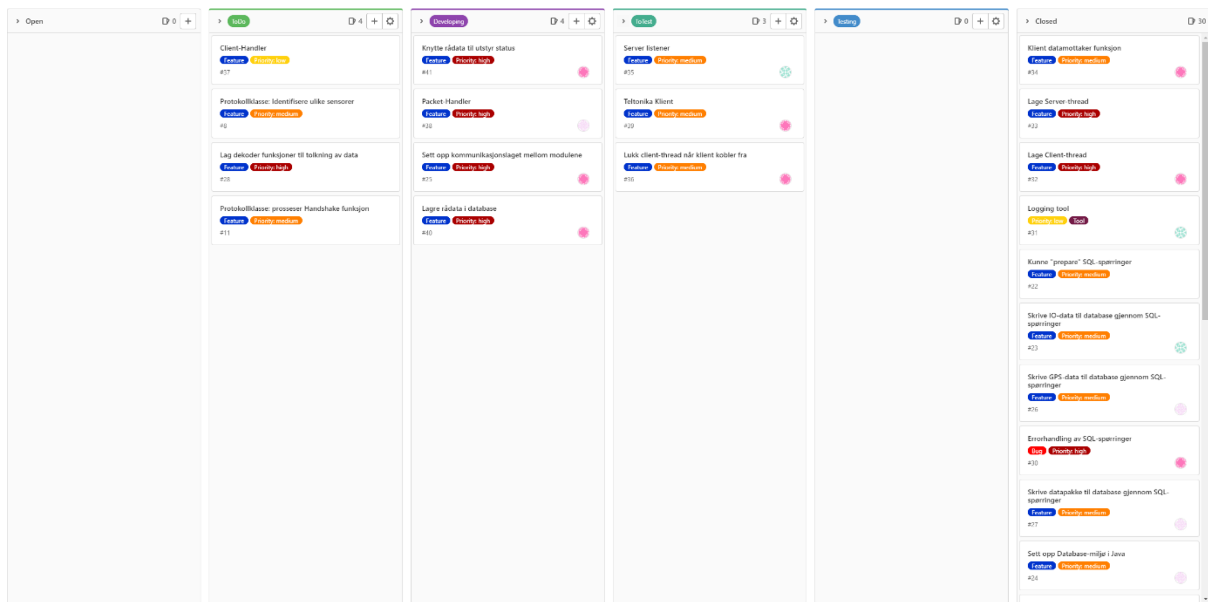
Open	**ToDo** (maks 10)	**Developing** (maks 4)	**ToTest** (maks 4)	**Testing** (maks 2)	**Closed**
Oppgaver som skal implementeres	Oppgaver vi skal jobbe med i nåværende periode	Oppgaver vi har startet å implementere	Oppgaver som er klare for testing	Oppgaver vi har startet å teste	Ferdige oppgaver

Vi satte grenser på hver del, slik at vi ikke kunne flytte oppgaver til neste kolonne, før det var ledig plass. På denne måten, vil ikke oppgaver hope seg opp i en kolonne, og det blir da forsikret at oppgaver kontinuerlig blir fullstendig ferdig.

Hver av oppgavene ble gitt et prioriteringsnivå som var enten lavt, medium eller høyt. Dette var kun et hjelpemiddel for oss, for å gjøre det lettere å velge viktige oppgaver. Vi lagde en oversikt, som skulle gjøre det lettere å vurdere hvilket prioriteringsnivå en oppgave hadde, som kun skulle være en pekepinn vi kunne gå etter. Oversikten vises i Tabell 3.

Tabell 3: Prioriteringsnivåer

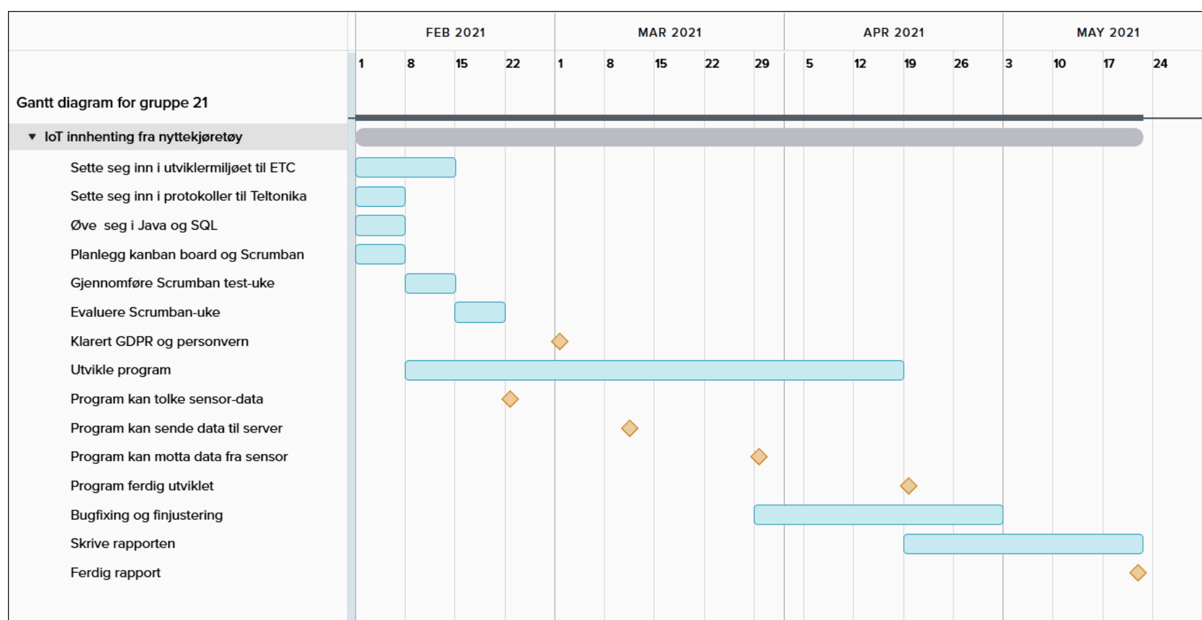
Lav prioritet	**Middels prioritet**	**Høy prioritet**
Oppgave som mangler kommentarer eller dokumentasjon Små hjelpefunksjoner	Bugs som ikke påvirker program-flyten, men som kan inntreffe Oppgaver som mangler feilhåndtering	Kommunikasjon mellom to store moduler Fundament som må utvikles før videre utvikling Bugs som påvirker program-flyten, og Bugs som omhandler sikkerhet



Figur 3: «Kanban»-tavle datert 19.03.2021

2.1.2 Milepæler

I planleggingsfasen laget vi et Gantt-diagram, der vi skrev ned ulike hendelser og milepæler som vi forestilte oss ville være aktuelle for vårt prosjekt. Vi hadde 3 milepæler relatert til utviklingen av programmet. Først skulle vi utvikle Dataprosesseringsmodulen. Så skulle vi utvikle Databasemodulen. Til slutt skulle vi utvikler Nettverksmodulen. Grunnen til at vi valgte å gjøre dette i den rekkefølgen, var at det kunne ta en del tid før enhetene var installert og klar til å koble seg til serveren, derfor valgte vi å utvikle Nettverksmodulen til slutt. Vi valgte heller å lage Dataprosesseringsmodulen først, slik at vi kunne teste og tolke og formatere dataen før lagring i Databasemodulen. Gantt-diagrammet med milepæler og hendelser vises i Figur 4.



Figur 4: Gantt-diagram

Hendelser for hver måned

Første måned (Februar) av prosjektet fulgte vi planen nøye. Her fikk vi gjort mye av planleggingen med ETC, valgte enhet og satt opp utviklingsmodellen som vi skulle bruke. De to milepælene som er en del av fasen, (Avklare GDPR, Tolke data) møtte vi også innen den fastsatte tiden. Utvikling av Dataprosesseringsmodulen ble gjort ferdig 1 uke før fristen.

Andre måned (Mars) var mest programmeringsorientert, hvor tiden gikk til utvikling av Databasemodulen og Nettverksmodulen. I tillegg gjorde vi mye innsamling av informasjon. Her forandret milepælen til Nettverksmodulen seg i forhold til Gantt-diagrammet, hvor vi utsatte målet med å motta data fra enhet, siden enhetene ikke var installert enda og ble erstattet med å utvikle en datapakkegenerator. Ellers ble resten av målene møtt i god tid for fristene og programmet ble ferdigstilt, også Nettverksmodulen, men den kunne ikke testes.

Tredje måned (April) skulle være den avsluttende fasen for programmering og opptrapping på rapportskriving, men på grunn av utsettelsen av enhetsinstallasjonen så måtte vi lage en ny milepæl, som gikk ut på å gjøre datapakkegeneratoren om til en kjøretøysimulator, som vi måtte utvikle så fort som mulig. I tillegg til å jobbe med dette, fikset vi mye på koden, dokumentasjonen og lagde tester til programmet.

Siste måned (Mai) var hovedsakelig kun rapportskriving som beskrevet i Gantt-diagrammet. I tillegg var det noen få justeringer i koden, men ikke så store at det gikk utover planen.

2.1.3 Møter

I starten av prosjektet ble vi enig om å ha møte med arbeidsgiver og veileder på ulike tidspunkt hver tirsdag. Her kom vi forberedt med en saksliste hvis det var noe vi lurte på. Ellers gikk møtene mye ut på å snakke om prosjektet for å få en felles forståelse hva som måtte gjøres. Vi fikk mye god hjelp fra veileder med rapporten. Ifølge timeplanen hadde vi 10 møter med veileder og 12 møter med arbeidsgiver i løpet av prosjektet.

3 Kravspesifikasjon

3.1 Funksjonelle krav

Selv om ETC var åpne for benyttelse av ulike teknologier og enheter, ble de funksjonelle kravene tidlig definert og ble ikke endret gjennom utviklingen av prosjektet:

- Det skulle automatisk loggføres veivedlikeholdsoppgaver.
- Det skulle loggføres tidsstempel og GPS-data, hvor det ble utført veivedlikeholdsoppgaver.
- Det skulle være mulig å jobbe mot flere typer enheter for avlesning av CANbus-systemet
- Ren data mottatt over nettet skulle behandles og formaters til leselig format på en server
- Mottatt data skulle lagres i en database for videre benyttelse.

Eksempler på benyttelse av lagret data kan være å kartlegge hvor det er brøytet og saltet, som kan gjøres tilgjengelig for kunder og publikum.

3.1.1 Use Case-diagram

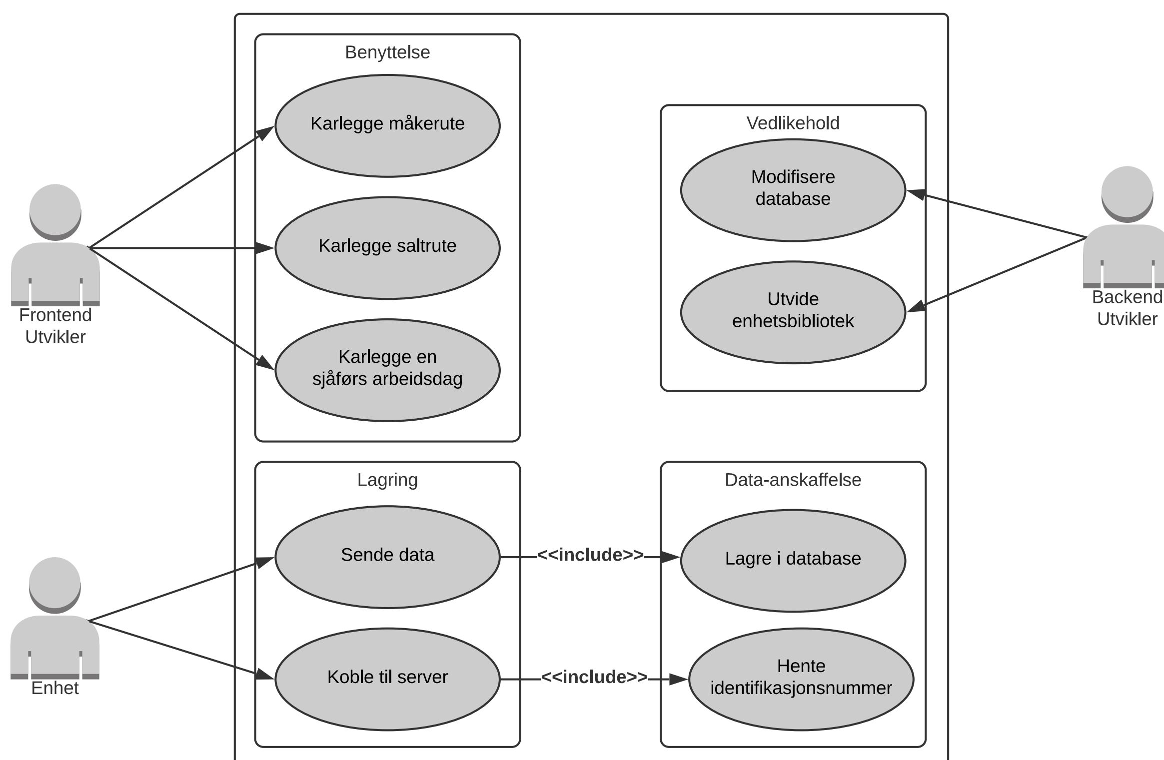
Use Case-diagrammet i Figur 5, viser benyttelser av funksjonaliteter og data i programmet vårt. Dette kan være bruksområder som å hente data om kjøreruter for å kartlegge en måkerute, eller at en backend utvikler legger til protokoller i systemet. Vårt Use Case-diagram har en litt annen vinkel enn et vanlig Use Case-diagram. Blant annet ser vi på hva frontend- og backend utviklere kan gjøre med programmet vårt, som ikke nødvendigvis går ut på bruk av programmet vi har utviklet, men at de kan videreutvikle det ved å bl.a. bruke de abstrakte metodene våre.

Aktører i diagrammet

Frontend Utvikler: En ansatt i ETC som vil sende spørringer til databasen for å hente ulike data, for å danne grafiske visualiseringer av bl.a. kjøreruter.

Backend Utvikler: En ansatt i ETC som modifiserer systemet for å bl.a. legge til flere enheter.

Enhet: Enhet som er installert på et kjøretøy. Kobler seg til ETCs server, og sender data.



Figur 5: Use Case-diagram

3.1.2 Høynivå Use Case

Her er 3 høynivå Use Case på de viktigste Use Casene til programmet vårt. Resten ligger i Vedlegg E.

Use case	Koble til server
Aktør	Enhet
Formål	Koble seg til serveren for å sende data fra kjøretøy til server
Beskrivelse	Enheten kobler seg opp til serveren gjennom enhetsfamilien sitt riktige portnummer, og initierer en «Handshake» med serveren. Serveren responderer med å godkjenne eller ikke godkjenne enheten.

Use case	Sende data
Pre-betingelse	Enheten har koblet seg til serveren og gjennomført godkjent «Handshake»
Aktør	Enhet
Formål	Hente data fra CANbus-systemet og sende til serveren
Beskrivelse	Enheten leser data fra kjøretøyet sitt CANbus-system og sender dette til serveren for videre tolkning, formatering og lagring.

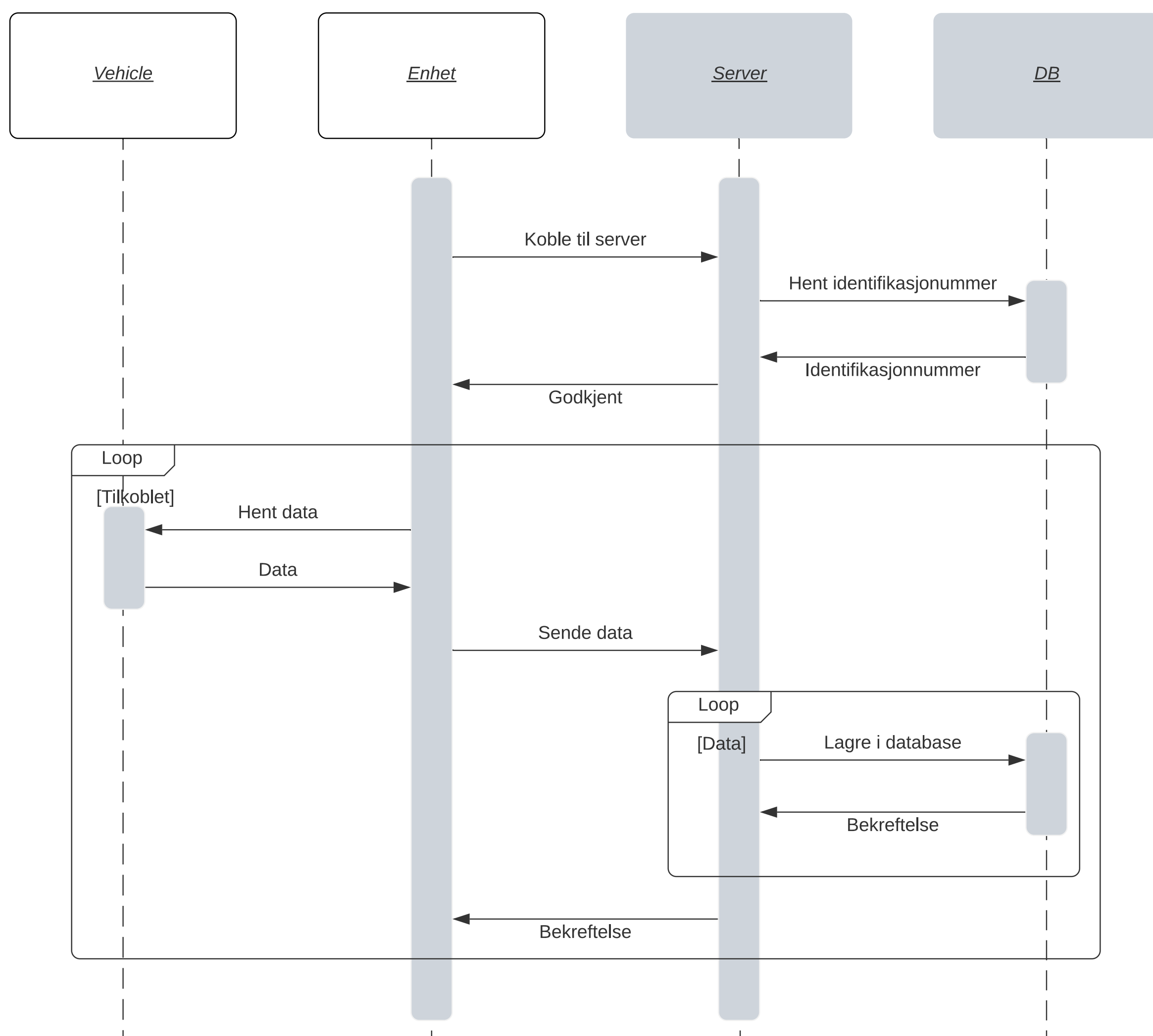
Use case	Kartlegge en sjåførs arbeidsdag
Aktør	Frontend Utvikler
Formål	Visualisere oppgaver en sjåfør har utført en spesifikk dag
Beskrivelse	En frontend utvikler, bruker innhentet data fra databasen, til å visualisere arbeidsruter. Ved å bruke en kombinasjon av data, som f.eks. GPS-data, tidsstempel og «Har plog vært nede»-data, kan utvikleren danne grafiske bilder av hvor det er brøytet på et publikumskart.

3.1.3 Expanded Use Case

Use case	Utvide enhetsbibliotek
Aktør	Backend Utvikler
Formål	Utvikle protokoller og andre funksjoner, for å legge til nye enheter i systemet.
Beskrivelse	En backend utvikler skal kunne utvide programmet for å støtte nye enhetsfamilier. Dette inkluderer å gjøre det mulig å koble seg til enhetsfamilien, gjøre det mulig tolke dataen riktig og registrere enhetsfamilien i databasen.
Post-Betingelse	Enheter fra den nye enhetsfamilien kan koble seg til serveren, og sende data til serveren der dataen blir prosessert og lagret i databasen.
Detaljert Hendelsesforløp ("Main Success Scenario"):	1. Utvikler setter seg inn i den nye enhetsfamilien sin protokoll 2. Nettverksdelen til programmet må bli utvidet slik det kan motta datapakker fra den nye enhetsfamilien. 3. Prosesseringsdelen i systemet utvides til å kunne tolke datapakkene etter enhetsfamilien sin protokoll og formaterer til et universalt dataformat. 4. Enheter i enhetsfamilien blir registrert i databasen med sitt unike identifikasjonsnummer
Ulike feilsituasjoner:	2a. Enheten støtter ikke TCP 1. Programmet må utvides for å støtte andre nettverksprotokoller 4a. Enheten har ingen måte å bekrefte et unikt identifikasjonsnummer 1. Må finne enhetsfamilie som kan gi bekreftelse

3.1.4 Sekvensdiagram

For å få en bedre forståelse av systemet lagde vi et sekvensdiagram, som viser den normale flyten til programmet. Diagrammet viser en «Happy Day»-scenario fra det dannes tilkobling mellom server og enhet, til data tolkes, formateres og lagres i databasen. Først vil en enhet prøve å koble seg til serveren. Serveren vil da sjekke om enheten er registrert i databasen. Er den registrert får enheten godkjennelse og får en tilkobling. Enheten vil da forsøke å lese data fra CANbus-systemet og sende dette til serveren, som vil tolke, formatere og lagre den relevante dataen i databasen. Diagrammet er vist i Figur 6.



Figur 6: «Happy Day» scenario for normal flyt i programmet

3.2 Ikke-funksjonelle krav

3.2.1 Design

Programmet skulle kunne koble seg til mange forskjellige enheter fra ulike produsenter der alle kunne ha forskjellige protokoller. Programmet skulle også kunne utvides for innhenting av annen data enn de oppgavene ETC gav oss som eksempel (Brøyting og salting). Det ble derfor stilt krav om at programmet skulle være skalerbart slik at det kunne utvides for nye enheter og innhenting av annen type data.

3.2.2 Drift

Krav til drift av server var delt inn i to kategorier, oppetid og responstid. Siden noen elementer som infrastruktur, nettverk og prosjektambisjoner var ukjent og dynamisk for både oss og ETC, så brukte vi noen tilnærmede krav for å dekke et minimumsbehov.

Oppetid: Som leverandør til flere kunder, ønsket ETC en oppetid på tilnærmet 100%, for å tilby en stabil tjeneste til kundene sine. Det er urealistisk å få en oppetid på 100% i starten av en tjeneste, ettersom man i løpet av oppstartsfasen sannsynligvis må endre på flere elementer i program og server, som vil senke oppetiden. Vi satte krav på 99% oppetid ettersom det ikke er et urealistisk mål og ville dekke de bruksområdene forventet av en prototype, og for å gi oss noe å strekke oss etter.

Responstid: Er tiden fra enheten sender en pakke, til den mottar en respons. Denne tiden ville være påvirket av internetthastigheten og behandlingstiden til serveren. Vi kan ikke kontrollere internetthastigheten, så det var kun behandlingstiden til serveren vi kunne påvirke. Etter samtaler med ETC, så kom det fram at serveren skulle kunne håndtere opptil flere titusener av kjøretøy (Klienter). Hvis hver av disse sender en pakke hvert 120. sekund, må serveren kunne ha en behandlingstid på under 12 millisekund.

3.2.3 Database

Et krav fra ETC var at databasen skulle være av typen MariaDB. Vi bestemte oss for at utformingen av databasen skulle følge normaliseringsprinsippene for relasjonelle databaser for å unngå duplisering av data som kan føre til inkonsistente tabeller, tregere databasesøk og unødvendig lagringsplass [12]. Det var mulig at ville bli lagret data kontinuerlig i databasen, ettersom det kom til å være mange enheter som ville koble seg opp til programmet, hvor de hver for seg ville sende data hvert andre minutt. Det var derfor viktig å gjøre det slik at gammel data ble jevnlig slettet for å minimere mengden data, slik at databasen ikke ville bli full av unødvendige informasjon og at slik at den ikke ville bli treig.

3.2.4 Dokumentasjon

ETC ønsket at all kode skulle dokumenters, med unntak av selvforklarende kode. Vi bestemte oss for å bruke Javadoc, som er en generator som gjør det lett å automatisk lage dokumentasjon i fra kildekoden. Dette er en industristandard for dokumentering av Java-klasser som følger med de fleste moderne Java-IDEer bl.a. IntelliJ [13]. Dokumentasjonen skulle skrives på norsk, imens variabler, metoder og klasser skulle skrives på engelsk. Vi bestemte oss for å kreve at vi fulgte kodekonvensjonene definert i Google Java Style Guide[2], ettersom dette var en konvensjon der kodestilen lignet på det vi hadde brukt i tidligere prosjekter, og følte oss derfor trygge på den.

3.2.5 Sikkerhet

Programmet skulle lytte åpent på nettet etter nye tilkoblinger. Det kunne føre til at uautoriserte personer, med ondsinnede hensikter, ville prøve å koble seg til serveren. Det var derfor viktig å implementere tiltak for å minimere angrepsoverflaten og eventuelle skader som kunne forekomme av et angrep. All innkommen data skulle valideres før videre prosessering og inaktive tilkoblinger skulle stenges. For å kunne skille mellom ønskede og uønskede tilkoblinger så skulle en «whitelisting»-funksjon være til stede.

Trusler

I Tabell 4 er de mest aktuelle truslene som programmet kan møte under oppetid. Hensikten med dette er å identifisere og aktivt planlegge mottiltak for å minimere risiko og konsekvens.

[2] https://google.github.io/styleguide/javaguide.html

Tabell 4: Trusler som programmet kan utsettes for

Trussel	**Beskrivelse**	**Konsekvens**
DDoS angrep	Et systematisk angrep mot serveren ved å lage massive mengder med fiktive dataforespørsler.	Vil kunne begrense eller stoppe serverens mulighet til å prosessere datapakker, og vil påvirke oppetiden til programmet.
SQL-injeksjon	Ved å misbruke data som sendes til en SQL-spørring, kan en person, påvirke SQL-spørringen, om det ikke er implementert metoder for å motvirke en slik trussel.	Gir en person som har fått muligheten til påvirke SQL-spørringene, muligheten til å hente ut eller slette data fra databasen.
Sabotasje fra ansatt i ETC	Lekkasje av data eller sabotasje av systemet fra en ansatt i ETC, med monetær gevinst eller personlig revansje som motivasjon.	Alvorlig datalekkasje kan gi bøter og ødelegge offentlig omtale for ETC, samt misbruk av data. Sabotasje av systemet kan gi systemkrasj og stoppe tilgjengeligheten for kunder.

3.2.6 Personvern

Siden prosjektet går ut på å loggføre data fra et kjøretøy, vil det potensielt bli lagret store mengder med personopplysninger om sjåførene og hendelser knyttet til arbeidet deres. For at behandlingen av disse opplysningene skal være lovlige, må ETC har rettslig grunnlag, og det har dem igjennom en berettiget interesse for flåtestyring. Selv om ETC har rettslig grunnlag, må de følge flere krav. Ifølge Datatilsynet, så må krav innen arbeidsmiljøloven og personvernforordningen (GDPR) være oppfylt, for at et slikt overvåkingstiltak skal være lovlig [14]. I vårt prosjekt er mye av disse kravene ETC sitt ansvar, blant annet kravet i GDPR om å informere de ansatte om formålet med overvåkingen. Som utviklere har vi mer ansvar for å følge personvernprinsippene, og da spesielt punktet om dataminimering. Vi må prøve å lagre *kun* den nødvendige dataen, og minimere hvor mye data som blir lagret, spesielt når det gjelder data som kan kobles mot en person. Det gjelder også lagringsbegrensning, der personvernprinsippene sier at når data ikke lenger er nødvendig for formålet, skal den slettes eller anonymiseres [15].

4 Valg av teknologi

For at utviklingen av prosjektet skulle gå effektivt, og likevel ha høy kvalitet, var det viktig å velge riktig teknologi. For det førte måtte vi velge gode verktøy, slik at selve utviklingsdelen av prosjektet ville bli effektiv og likevel gi høy kvalitet. For det andre måtte vi også velge riktig enhet som skulle installeres på et kjøretøy, og sørge for at denne var av høy kvalitet, og forsikre oss om at den var kompatible med kjøretøyet og programmet.

4.1 Valg av verktøy

ETC anbefalte oss å bruke Intellij som vårt integrerte utviklingsmiljø til prosjektet. Vi hadde brukt dette tidligere og hadde derfor god erfaring med å konfigurere prosjekter med programmets innstilinger. Vi fikk også vite at utviklerne hos ETC brukte Intellij, så om det skulle oppstå noen problemer knyttet til programmet, så kunne vi høre med dem.

I begynnelsen av utviklingen måtte vi bestemme oss hvilken versjon av Java vi skulle bruke. Vi ble enige med ETC om å bruke Java SDK 15, ettersom det var en av de nyeste versjonene. Vi trengte ikke bekymre oss for at programmets SDK-versjon skulle ha kompatibilitetsproblemer med ETC sine moduler, siden prototypen skulle være frittstående, uten å være koblet mot de andre CarAdmin-modulene.

JUnit Jupiter er et testrammeverk for Java, som vi brukte til å lage enhetstester for de ulike funksjonene og klassene i programmet. En på gruppen hadde jobbet med JUnit Jupiter før, og anbefalte oss å bruke dette.

HeidiSQL er et brukergrensesnittverktøy til databasehåndtering, der man kan konfigurere alle aspektene knyttet til en database på en oversiktlig måte. Programmet fulgte med når vi nedlastet MariaDB, ettersom det er en del av MariaDB-installasjonen. Vi hadde ikke brukt dette før, men det ville gjøre det mye enklere å konstruere databasen og sette inn test-data, ettersom vi hadde et brukergrensesnitt vi kunne bruke og slapp å bruke konsollkommandoer.

GitLab er et versjonskontrollsverktøy, som vi brukte etter krav fra ETC. Her hadde ETC sin egen instans som vi skulle bruke. Til prosjektstyring i GitLab benyttet vi de fleste verktøyene som vi mente var hensiktsmessig som f.eks. «Kanban»-tavla og milepælstyring.

Google Disk brukte vi som et felles dokumentdelingsverktøy, hvor vi lagret viktige filer som arbeidstimeliste, møtenotater, sakslister, prosjektplanen og andre dokumenter som krevde felles tilgang av alle i gruppen. Google Disk er nyttig, i den grunn at det er lett tilgjengelig overalt ved å bruke nettleser, og vi kan jobbe i sammen med flere typer dokumenter i sanntid. Google Disk lagrer dokumentene i skyen, så hvis en av datamaskinene våre krasjer, så vil ikke noen dokumenter gå tapt.

Discord er kommunikasjonsverktøyet som vi brukte for interne møter, fellesarbeidsøkter og intern fildeling mellom gruppemedlemmene. Dette gjorde vi ved å sette opp en egen Discord-server, som gjorde at dokumentering av kilder, lenker og avtaler ble lettere å finne igjen for medlemmene på gruppa. Discord lar også medlemmene dele skjerm, slik at det er lettere å vise ulike diagrammer og koder mellom hverandre.

Microsoft Teams var også et sentralt verktøy i prosjektet. Det var her vi avtalte og hadde møter med veileder og ETC. Det var også der vi endte opp med å skrive rapporten. Programmet er koblet mot Microsoft Word, og gir verktøy for å jobbe med samme dokument i sanntid. I tillegg blir dokumentet også lagret i skyen.

Microsoft Word brukte vi som tekstprogram for Bacheloroppgaverapporten. Det er det tekstprogrammet gruppemedlemmene hadde mest erfaring i å bruke. Programmet har flere funksjoner som var gunstige for rapporten vår. Blant annet et automatisk referanseverktøy, som ville gjøre jobben med referanser mye enklere. Word har også stavekontroll. Vi brukte også mye «Merknad»-verktøyet, for å legge til notater på ting som burde endres, eller om det manglet en kilde osv.

4.2 Valg av enheter

For å velge riktig enhet hadde vi flere samtaler med ETC. Her kom vi fram til ulike egenskaper som var av interesse for systemet, hvor noen av dem var krav fra ETC, mens andre var sterkt ønsket. ETC ville at kommunikasjonen til enheten skulle gå over 4G-nettet. Dette er ettersom 3G-nettet begynte å bli gradvis slukket i 2020 og 2G-nettet legges ned rundt 2025 [16]. Vi hadde også satt en avgrensning i 1.1.3 for nettverksprotokollen, og enheten måtte derfor støtte TCP. ETC ønsket også at det skulle være enkelt å installere enheten i kjøretøyene. Dette for å forhindre komplikasjoner og kostnader knyttet til kutting av ledninger og andre tiltak som må til for å installere enheten. I mange tilfeller vil dette ødelegge garantien på kjøretøyet. Enhetene måtte også følge SAE J1939 standarden. Med disse kravene og ønskene i bakhodet, begynte vi å se etter enheter.

4.2.1 Valg av produsent

Da vi begynte prosessen der vi skulle velge enheter til prosjektet, startet vi med å undersøke de mulige produsentene av slike produkter. Mange av kandidatene fant vi ved å undersøke selskaper og produsenter som leverte flåtestyringstjenester, ved å søket på nettet.

Teltonika

I starten av prosjektet foreslo ETC en produsent som de hadde jobbet med før, som en mulig kandidat. Dette var Teltonika, som er et selskap som produserer diverse enheter. Blant annet produserer de enheter som henter og tolker CAN-data fra kjøretøy, og som kan sende denne dataen over nettet [17].

Teltonika
Fordeler med å bruke Teltonika
• ETC har jobbet med selskapet før, og har derfor erfaringer med selskapet • ETC har allerede flere enheter fra Teltonika på lager, og vi kan derfor teste på disse mens vi venter på de bestilte enhetene • Anerkjent selskap med kontor i hele verden • God og gratis dokumentasjon på grensesnittet til enhetene
Ulemper med å velge Teltonika
• Potensiell lang leveringstid ifølge ETC • Økonomisk konsekvens/risiko

CSS Electronics

Etter å ha gjort litt nettsøk, fant vi en annen potensiell produsent, CSS Electronics. Dette selskapet leverer CAN-loggere, og holder til i Danmark [18].

CSS Electronics
Fordeler med å bruke CSS Electronics
• Anerkjent selskap. Selskapet supplerer over 1000 selskaper i over 80 land og brukt i flere store selskaper som bl.a. Volkswagen og Toyota • God dokumentasjon på grensesnitt til enhetene
Ulemper med å velge CSS Electronics
• Ingen tidligere erfaringer • CSS Electronic krever en ekstra database for å gi innhentet data fra kjøretøyet meningsfulle verdier • Enhetene har ikke innebygd GPS-modul eller sender for sending av data over mobilnettet, kun over WLAN

Sammenligning av CSS Electronics og Teltonika

Hvis vi ser på fordelene, så går det igjen i begge produsentene at de er anerkjente selskap og har god dokumentasjon på grensesnittet til enhetene. Begge følger også SAE J1939. En ekstra fordel med Teltonika er at ETC har arbeidet med Teltonika før. De har derfor god kontakt, og har også en del innkjøpte enheter fra før. Disse kan vi teste på, hvis leveringstiden blir lang. I tillegg til å ha erfaring med Teltonika, så har ETC også sagt at de har utviklet en testprotokoll for å tolke data fra Teltonika-enheter, som vi kunne få tilgang til. Ser vi på ulempene til produsentene, så ser vi at vi mangler erfaring med CSS Electronics, og det ekstra arbeidet med å tolke den rene mottatte dataen kan bli et problem. Andre ulemper som produsentene deler er at det er en viss økonomisk risiko involvert, der man kan kjøpe en enhet som ikke tilfredsstiller kravene eller som ikke er kompatibel med kjøretøyene, og at leveringstiden fort kan bli lang, ifølge ETC sine erfaringer. CSS Electronics har en spesiell økonomisk risiko ettersom det må betales mye for ekstra komponenter, og andre verktøy.

Resultat

Vi endte opp med å velge Teltonika. Dette gjorde vi med en totalvurdering, etter å ha sett på hvilke fordeler som veide mest. ETC hadde gode erfaringer med Teltonika, hadde kode-prototyper, og i tillegg hadde enheter liggende, gjorde at det veide mest i Teltonika sin retning.

4.2.2 Enhetskandidater

Etter å ha bestemt oss for å bruke Teltonika som produsent, var neste steg å finne riktig enhet. Som beskrevet i 4.2 satte ETC opp noen ønsker og krav. Teltonika har flere enhetsfamilier og vi vurderte enten FMB- eller FMC-familien. Enhetene i disse familiene har innebygd GPS og tidsmåler, og har som formål å sende dette og CANbus dataen til serveren. Enhetene leser ikke CAN-dataen selv, og for å følge opp kravene trenger vi også en enhet som kan lese CAN-dataen. Her var det ALL-CAN300 eller LV-CAN200 som er kandidatene, men siden LV (Light Vehicle)-CAN200 er beregnet for lette kjøretøy, og har mindre kompatible kjøretøytyper, så velger vi ALL-CAN300. ETC ønsket også at det skulle være lett å installere enhetene, og unngå å ødelegge garantien med å klippe ledninger i kjøretøyet. Her var det Simple-CAN eller Mini-CAN som var de aktuelle kandidatene. Vi valgte Mini-CAN ettersom det ikke var noe signifikant forskjell mellom enhetene, og Mini-CAN var mindre og ville derfor ta mindre plass i kjøretøyet. Videre vil vi nå diskutere hvilken enhet vi vil velge som sender.

Valg av sender

Etter å ha bestemt oss for ALL-CAN300 og Mini-CAN, sendte vi en e-post til en ansatt i Teltonika, som vi hadde fått anbefalt i fra ETC. Her skrev vi kravene fra ETC. Først fikk vi anbefalt FMB140. FMB140 har en innebygd ALL-CAN300-komponent, som ville gjøre installasjon mye enklere. Dessverre benyttet FMB140 seg av 2G-nettet og ikke 4G-nettet som ETC ønsket, og vi spurte om andre anbefalinger som brukte 4G-nett. Da fikk vi anbefalt FMM130 eller FMC130. FMC130 var kompatibel med 4G-nettet mens FMM130 var kompatibel med både «LTE M1» og «NB IoT». Vi var usikre på om «LTE M1» og «NB IoT» ville fungere ordentlig, så vi valgte FMC130 som vår sender, ettersom vi visste den oppfylte kravet om 4G og vi var usikre på om de andre nett-metodene ville være kompatible nok, og ville unngå den risikoen.

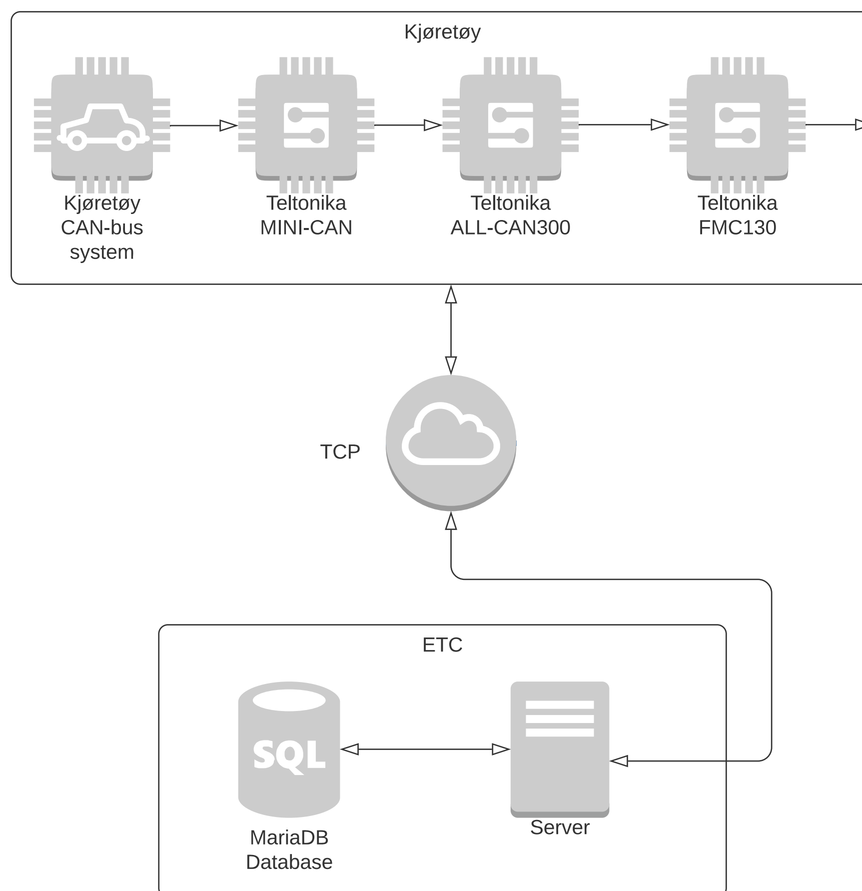
4.2.3 Oppsummert

ALL-CAN300 er enheten som skal lese CAN-dataen, fra kjøretøyet.

Mini-CAN er en enhet som kobles til ALL-CAN300-enheten. Den leser signaler fra ledninger i kjøretøyet uten å måtte klippe ledningene og ødelegge garantien.

FMC130 er en enhet som hente CAN-data fra ALL-CAN300-enheten og sender dette i sammen med GPS-data og tidsstempel over 4G til en mottaksserver.

Dette enhetsoppsettet er vist i Figur 7.



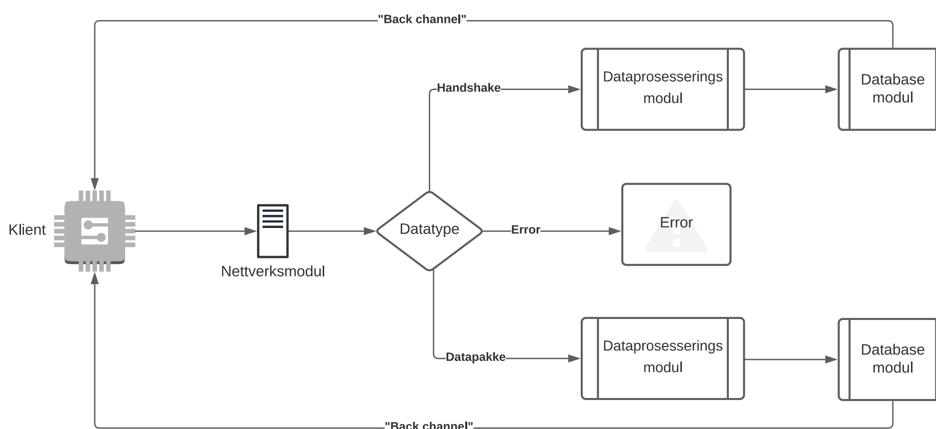
Figur 7: Enhetsløsning

5 Teknisk design

5.1 Overordnet Arkitektur

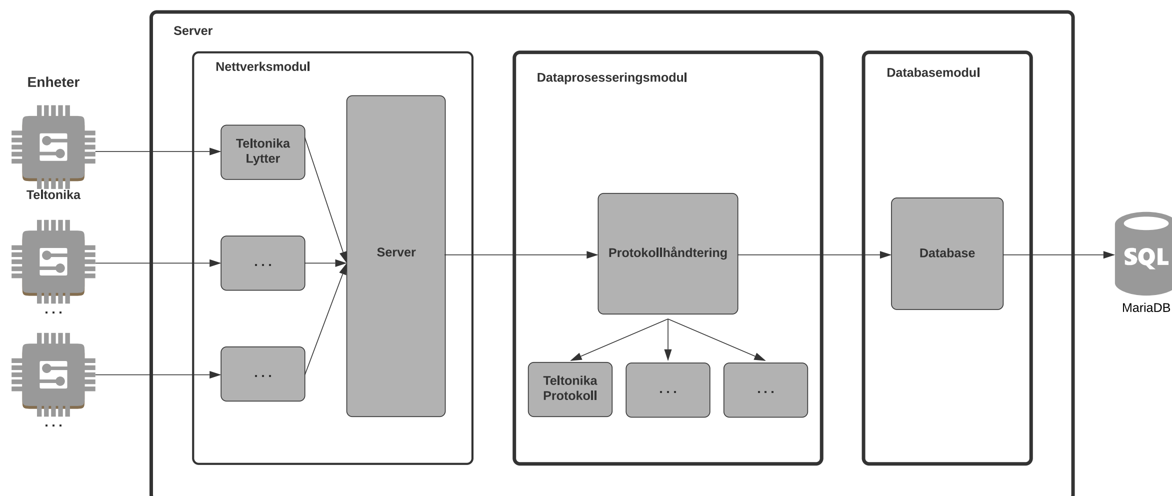
Som systemets arkitektur valgte vi å følge «Pipe and filter»-strukturen. Datastrømmen i programmet vil være av sekvensiell natur, der dataen ble sendt fra en enhet, prosessert på en server, og til slutt lagret i en database. I løpet av denne sekvensen, ville dataen bli tolket og formatert. Dette passet bra med «Pipe and filter»-prinsippet, siden dataen blir behandlet sekvensielt gjennom flere trinn [19].

I starten av prosjektet så vi at programmet kunne være modulært, ettersom vi kunne se tydelig ansvarsområder programmet skulle ha, som kunne jobbe selvstendig av hverandre. Vi kunne tenke på et filter som en modul i programmet. Disse filtrene skulle bli koblet sammen i en pipeline, hvor et filter sitt «Output»-objekt skulle være neste filters «Input». Ved slutten på en «Pipeline» skulle responsen sendes tilbake til enheten gjennom en «Back Channel», dersom det var nødvendig. Denne dataflyten er illustrert i Figur 8.



Figur 8: Dataflyt gjennom «Pipelines»

Vi bestemte at modulene som programmet skulle bestå av var en Nettverksmodul, en Dataprosesseringsmodul og en Databasemodul. Nettverksmodulen skulle sørge for å innhente data fra tilkoblede enheter og lytte etter nye tilkoblinger. Dataprosesseringsmodulen skulle tolke den innhentede dataen fra Nettverksmodulen, og formatere den til et universalt format. Så skulle Dataprosesseringsmodulen sende den prosesserte dataen videre til Databasemodulen, som skulle lagre dataen i en database. Denne blandingen av «Pipe and filter» og modulær arkitektur, gav oss god kontroll og oversikt over både dataflyten gjennom programmet, men også utviklingsprosessen. Ved å la hver modul være en milepæl i utviklingsfasen, fikk vi en oversiktlig arbeidsplan. Figur 9 viser en illustrasjon over modulene.



Figur 9: Programmet sine moduler

5.2 Design

Når vi planla og designet de individuelle modulene, kartla vi de ulike elementene som vi forutså modulene ville inneholde. Dette var innhold som klasser og funksjoner.

5.2.1 Nettverksmodulen

Nettverksmodulen var modulen som skulle ha kontroll på alle tilkoblinger og klienter som serveren skulle kommunisere med.

Som en grovbeskrivelse, skulle modulen inneholde en «Lytter» for hver enhetsfamilie, som skulle lytte etter enheter på portnummeret til enhetsfamilien. En «Lytter» skulle iverksette en oppkobling til en enhet ved å lage en «Klient»-tilkobling som skulle motta data fra enheten. Alt dette skulle bli kontrollert i en «Server»-instans.

Siden serveren skulle ha kontroll på alt, og siden mottakelse av data og lytting etter nye tilkoblinger ville blokkere på «I/O», så måtte vi unngå dette, og vi tenkte å løse dette ved å la «Klientene» motta data og «Lytterne» lytte etter tilkoblinger på egne tråder, for å hindre blokade i systemet.

5.2.2 Dataprosesseringsmodulen

Dataprosesseringsmodulen var modulen som skulle stå for klassifisering av datapakker, velge riktig protokoll etter enhetsfamilie og til slutt tolke og formatere dataen til et universelt format.

Dataprosesseringsmodulen skulle inneholde flere funksjoner og klasser. Den skulle ha flere protokollklasser, der hver protokollklasse skulle tolke dataen til sin egen enhetsfamilie. Tolkingen av data ville variere fra enhet til enhet, men den resulterende formaterte dataen skulle bli lik for hver eneste enhet. En annen sentral del av modulen vil være en protokollbehandler som skulle delegere datapakker til validering og prosessering.

For å gi protokollklassene et felles grensesnitt, tenkte vi å ha et abstrakt protokoll-objekt som hver protokollklasse arvet ifra. På denne måten ville det bli lettere å lage protokollklassene og samtidig ville klassene da ha mange fellestrekk, som gjorde at man ikke trengte å gjøre store endringer i grunnsystemet.

5.2.3 Databasemodulen

Databasemodulen skulle sørge for å koble seg til og sende data til databasen gjennom SQL-spørringer. Oppsettet av selve databasen beskrives i 5.3 nedenfor. Selve Databasemodulen skulle ha en statisk klasse, som ville inneholde funksjoner som skulle utføre SQL-spørringer. Den skulle også ha et grensesnitt som ville sørge for tilkoblingen til databasen.
Som beskrevet i 3.2.5, skulle vi forsøke å hindre SQL-injeksjon. Derfor var det viktig at klassen som stod for SQL-spørringene, sørget for å prøve å forhindre dette.

5.3 Databaseoppsett

Når vi skulle lage oppsettet for databasen, satt vi oss først ned og skrev opp all potensiell data som ETC ønsket å lagre. Her gikk vi en del ut ifra TeltonikaFMC sine sendingsparametere.

Tabell 5: Lagret data

Info om Kjøretøy	
Kjøretøytype	Type Kjøretøy (Traktor, Lastebil, Personbil)
Merke	Merke på kjøretøyet (Eks Toyota Avensis, Zetor 7745)
Info om Enhetsfamilien	
Enhetsfamilie	En serie med enheter med mange fellestrekk
Produsent	Hvem som står for produksjon av enheten (selskapsnavn, e-postadresse)
Info fra Enheter	
Tidsstempel	Tidsstempel for dataen
GPS-data	GPS-data (Lengdegrad, Breddegrad, Høyde over havet, Vinkel, Antall Satellitter)
IO-hendelser	Hendelser skjedd i forbindelse med et kjøretøy (Navn på hendelsen, Verdi på hendelsen)

5.3.1 Normalisering av tabeller

Etter å ha fått oversikt over hva som skulle i databasen, begynte vi oppgaven med å normalisere tabellen [20]. Vi delte tabellen opp i flere tabeller slik de 3 normalformene i normalisering av database blir oppfylt. Tabellene fikk da primærnøkler, fremmednøkler og fikk nye relasjoner med hverandre, for å hindre duplisering av data og få bedre integritet. Vi endte opp med resultatet vist som en skisse i Figur 10. Det var et sted i normaliseringen at vi var usikre på om vi skulle bruke 3. normalform, og det var når vi skulle konstruere tabellen for GPS-dataen. Alle datapakkene skulle lagres med GPS-data, som forteller hvor hendelsene som er knyttet til datapakken ble utført. Vi var usikre på om denne GPS-dataen skulle lagres i en egen tabell med en fremmednøkkel til datapakken dataen kom fra, eller om det skulle lagres i samme tabell som datapakken. Hvis vi lot GPS-dataen ligge i samme tabell som Datapakkene, ville tabellen hatt 10 kolonner, og ville blitt vanskelig å lese, og samtidig føles det ikke riktig å lagre GPS-dataen i en tabell kalt datapakker. Derfor tok vi valget i å la GPS-dataen ligge i en egen tabell med en fremmednøkkel til datapakken. Dette strider egentlig med normaliseringsprinsippene ettersom det blir en unødvendig tabell, men vi ønsket en leselig og strukturert database.

GPS-DATA

gps_id (INT)	DataPack_id (INT)	Longitude (DECIMAL)	Latitude (DECIMAL)	Altitude (SMALLINT)	ANGLE (SMALLINT)	NoOfSatellites (TINYINT)	Speed (SMALLINT)

SENSOR

Sensor_id (INT)	DeviceFam_id (INT)	Name VARCHAR(50)

DEVICE-FAMILIES

DeviceFam_id (INT)	Name (VARCHAR(50))	Producer_id (INT)	Port (INT)

PRODUCER

Producer_id (INT)	Name (VARCHAR(50))	Mail (VARCHAR(255))

DATA-PACKET

DataPack_id (INT)	Sensor_id (INT)	GPS_id (INT)	Timestamp (TIMEDATE)

IO-EVENT

IO_id (INT)	DataPack_id (INT)	Event-Name (VARCHAR(50))	VALUENUM (DOUBLE)	ASCIIVALUE	MULTIPLIER (DECIMAL)

VEHICLE

Vehicle_id (INT)	VehicleType_id (INT)	Modell (VARCHAR(50))	LicensePlate(gdpr?) (VARCHAR(45))

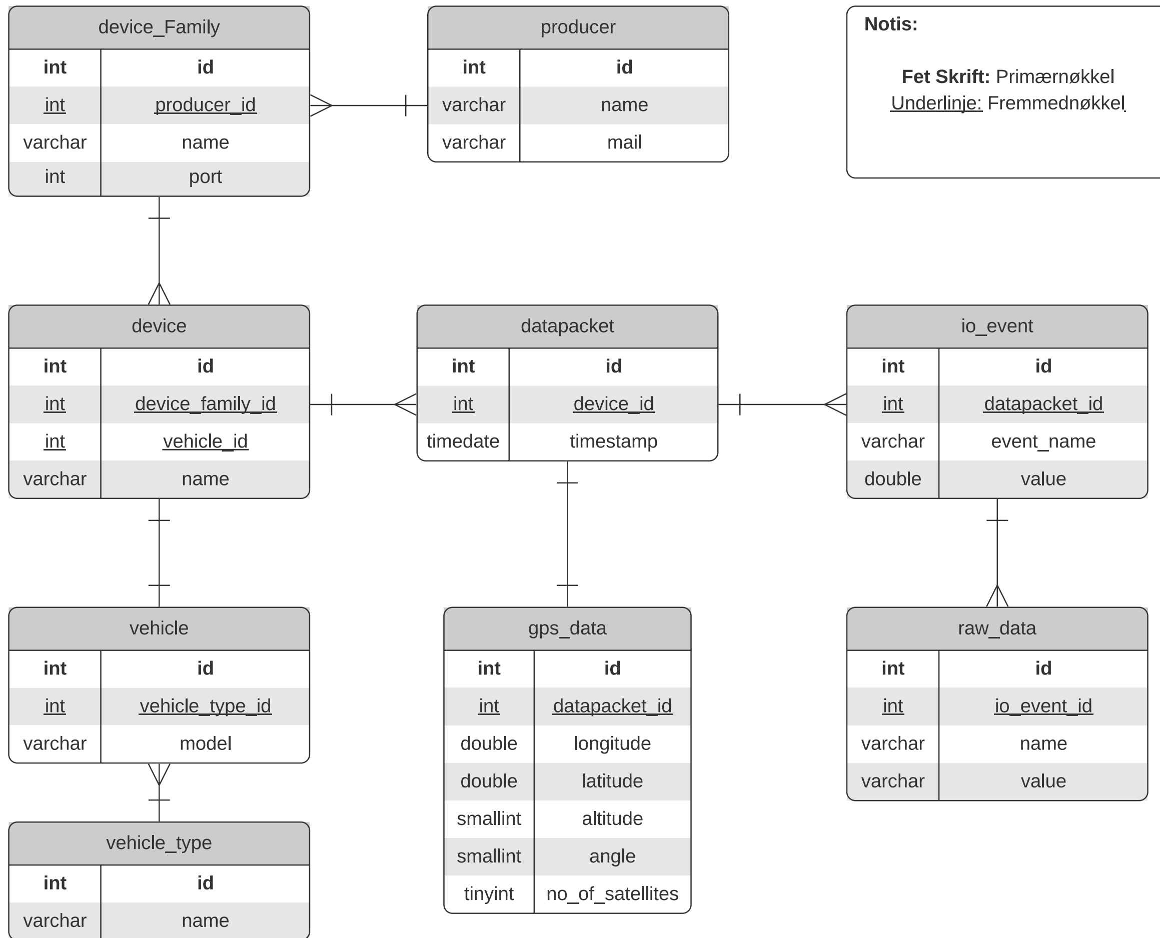
VEHICLE_TYPE

VehicleType_id (INT)	Name (VARCHAR(50))

Figur 10: Resultatskisse av normaliserte tabeller

5.3.2 Konstruksjon av databasediagram

Etter å ha normalisert tabellene, ville vi også legge resultatet inn i et databasediagram, for å få oversikt over tabellene, radene og relasjonene imellom dem. Dette ville gjøre det lettere å konstruere selve databasen i etterkant, ettersom vi kan lage databasetabellene rett i fra tabellene i diagrammet. I tillegg til å vise primærnøkler og fremmednøkler viser diagrammet også hvilke datatyper de ulike radene skal ha, som gjør konstruksjonen av databasen mer effektiv. Dette diagrammet ble endret flere ganger i løpet av utviklingsperioden, grunnet endringer i datatyper, endring av rader, og justering av navn. Den nyeste versjonen er avbildet i Figur 11.



Figur 11: Diagram av database

6 Implementering

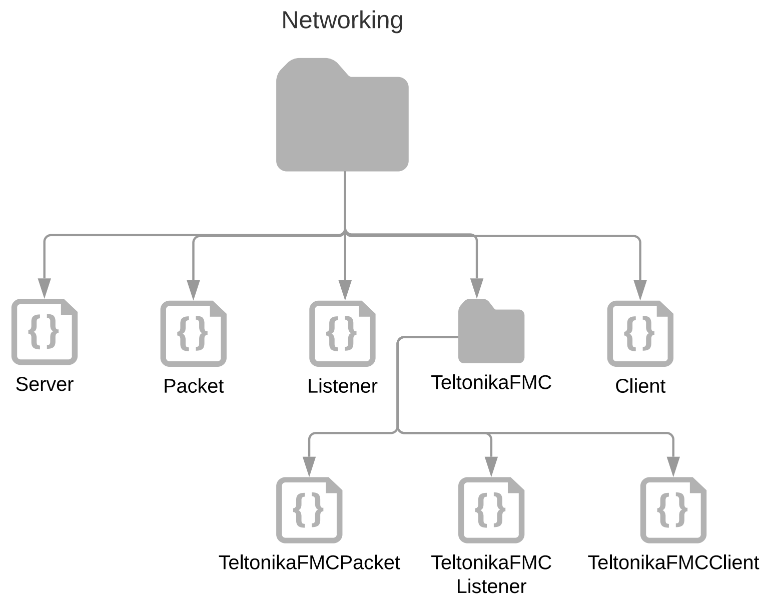
Etter at vi bestemte oss for hvilken enhet vi ville bruke i prosjektet og ETC hadde satt inn bestilling til produsenten, så startet vi utviklingen av programvaren. Implementeringen av koden, skjedde i tre steg, som vi vil beskrive i detalj i de neste tre delkapitlene. De var ikke implementert i rekkefølgen som er beskrevet under, men kunnskapen om implementeringen av en modul, har en naturlig overgang til neste modul.

I hvert kapittel, vil vi gå igjennom hvordan vi har implementert metoder og strukturer for hver modul, hvor vi til slutt beskriver hvordan dette blir brukt for å tilpasse modulen til å fungere med TeltonikaFMC-familien. Vi følte vi kunne implementere metodene i modulene uten å ha enheten til stede, eller installert, ettersom det var god dokumentasjon på hvordan dette gjøres.

Alle kodesnutter vist i dette kapittelet har blitt justert i forhold til den virkelige koden. Dokumentasjon, kommentarer og mye av innmaten er fjernet for å kun vise de viktigste funksjonene.

6.1 Nettverk og sending av data

Den første modulen i programmet vi vil gå igjennom, var delen der det skulle kommuniseres og sendes data mellom enhetene og programmet. Vi måtte utvikle et system der ulike typer enheter kunne koble seg til samme server, og modulen måtte derfor være skalerbar og kunne utvides. Som beskrevet i 1.1.3, avgrenset vi programmet til å kun implementere kommunikasjon ved hjelp av TCP. Vi lagde en Java-pakke som vi kalte «Networking» der kommunikasjonen og sending av datapakker skulle bli gjennomført. Figur 12 viser filstrukturen til Java-pakken.



Figur 12: Filstruktur i "Networking" Java-pakke

6.1.1 Datapakker

Når data fra et kjøretøy skal bli sendt over nettet, vil vi i de fleste tilfeller motta dette som en ren datastrøm. Denne dataen kan være utformet på ulike måter, og vi ønsket å legge dataen fra de ulike enhetene i ulike klasser, slik at det ble mer oversiktlig og slik at vi kunne kontrollere hvordan data fra ulike enheter ble håndtert i programmet. For å oppnå dette, lagde vi en abstrakt klasse som vi kalte «Packet». Et slikt objekt inneholder enhetens familienavn og en referanse til klienten til enheten som sendte dataen (Dette blir beskrevet ytterligere i klient-delkapittelet). Fra denne klassen kan man danne spesifiserte pakker til en enhet, der pakken vil inneholde data lagret i enhetens dataformat, som returneres i en abstrakt funksjon.

```
public abstract class Packet {
    private final Client client;
    private final DeviceFamily deviceFamily;

    public Packet(Client client, DeviceFamily deviceFamily) {
        this.client = client;
        this.deviceFamily = deviceFamily;
    }

    public Client getClient() {
        return client;
    }

    public DeviceFamily getDeviceFamily() {
        return deviceFamily;
    }

    public abstract <T> T getData();
}
```

Kodesnutt 1: Packet-klasse

6.1.2 Klient

Når det skal dannes kommunikasjon mellom enhetene og programmet er det viktig at denne kommunikasjonen holdes åpen, slik at serveren og enheten kan fortsette å utveksle data. Vi lagde derfor en abstrakt klasse som vi kalte «Client», som kan spesifiseres for hver enhetsfamilie. En klient har et «Socket»-Java-objekt, som sørger for å holde tilkoblingen åpen. For å minimere sannsynligheten for overbelastning av systemet vil programmet jevnlig frakoble inaktive klienter. Dette skjer om ingen data mottas på serveren innen en halvtime, dette er en tid som vi har tilfeldig valgt og kan endres i «Client»-klassen.

Klassen har også et «DataInputStream»-Java-objekt og et «DataOutputStream»-Java-objekt som sørger for at det kan dannes en datastrøm inn og ut av programmet. «DataInputStream»-Java-objektet blokkerer på «I/O», til det mottas data. Som beskrevet i 5.2.1, så kunne vi forhindre at dette midlertidig stopper programmet, ved å la «Client»-instansene kjøre på sine egne tråder. Dette gjorde vi ved å la klassen implementere en Interface i Java kalt «Runnable». Et alternativ til dette kunne være å benytte «Non blocking I/O», men dette hadde vi ingen tidligere erfaringer med, og ETC sa at bruk av tråder er tilstrekkelig.

Klienten har noen abstrakte funksjoner som overstyres i subklassene. Dette er blant annet en funksjon som konstruerer og returnerer en «DataPacket»-subklasse til riktig enhetsfamilie. Klassen har responsfunksjoner ved mottakelse av data, som også overstyres i subklassene.

En ulempe med måten vi har gjort implementasjonen av klienter på, er at det er vanskelig å teste tråder, og det kan oppstå flere problemer rundt dem. Selv om vi har brukt synkroniserte metoder, er det vanskelig å vite med 100% sikkerhet at alt fungerer problemfritt. Ettersom dette skal være en prototype, har vi ikke dedikert mye tid til å teste dette, og det kan være aktuelt i en fremtidig versjon av programmet.

```
public abstract class Client implements Runnable {
    protected final Server handler;
    protected final Socket socket;
    protected final DataInputStream input;
    protected final DataOutputStream output;

    private long id = -1;
    private int packetCount = 0;
    private final int bufferSize;

    public Client(Server packetList, Socket socket,
                  DataInputStream input, DataOutputStream output,
                  int bufferSize) {}

    public void acceptConnection(long id) {}

    protected abstract void onAcceptConnection(boolean flag);

    public void sendReceipt(ParsedData data) {}

    protected abstract void onSendReceipt(ParsedData data);

    protected Packet receive() throws IOException{}

    protected abstract Packet onReceive(byte[] data);

    public void send(byte[] data) {}

    public void disconnect() {}

    @Override
    public void run() {}

    public long getId() {}
}
```

Kodesnutt 2: Utkast av Klient-klasse

6.1.3 Lytter

For å danne kommunikasjon mellom enhetene og programmet, trengte vi en metode for å kunne lytte etter nye tilkoblinger over nettet og initiere nye «Client»-instanser. Siden vi skulle kommunisere med flere enhetsfamilier, var det viktig å kunne skille mellom dem, slik at programmet visste hvilken type klient som skulle lages. Vi lagde derfor en abstrakt klasse som vi kalte «Listener». Klassen har et «ServerSocket»-Java-objekt, og funksjonen vi bruker fra dette objektet, blokkerer også på «I/O». For å hindre blokade i programmet, implementerer også denne klassen Java sin «Runnable»-Interface slik at instanser kan kjøre på sine egne tråder. Den har funksjoner som er like for alle enhetsfamiliene, men forskjellen defineres med hvilket portnummer som gis til konstruktøren og hvilken type klient som blir initiert. Ulempene her er like som i «Client»-klassen, med at det er vanskelig å teste trådene.

```
public abstract class Listener implements Runnable {
    private final Server handler;
    private ServerSocket serverSocket;
    private boolean listening;
    private final int port;

    public Listener(Server handler, int port) {}

    private void startListening() {}

    private void stopListening() {}

    @Override
    public void run() {}

    protected abstract Client createClient(Server handler,
                Socket socket, DataInputStream input,
                DataOutputStream output);

    private void connect() {}
}
```

Kodesnutt 3: Utkast av Lytter-klasse

6.1.4 Server

For å knytte klassene i Nettverksmodulen i sammen og for å lage et grensesnitt for kjerneprogrammet, lagde vi en klasse som vi kalte «Server». Den skulle sørge for at lytterne ble iverksatt ved programstart. Den har en synkronisert liste med pakker mottatt fra klientene, slik at programmet kan hente dem for videre behandling. For å sørge for at det kun initieres én «Server»-klasse, lar vi konstruktøren være privat, slik at man kun kan hente en statisk instans gjennom funksjonen «getInstance()». En ulempe med denne klassen er at man ikke kan hente klienter fra klientlisten ut ifra identifikasjonsnummeret, bare gjennom plasseringen i listen.

```
public class Server {

    private static Server instance;

    private final List<Client> clients =
                Collections.synchronizedList(new ArrayList<>());
    private final List<Packet> packets =
                Collections.synchronizedList(new ArrayList<>());
    private final HashMap<String, Listener> listeners;

    public static Server getInstance() {}

    private Server() {}

    public void init() {}

    public void close() {}

    public void addPacket(Packet packet) {}

    public Packet getPacket() {}

    private void addListener(String name, Listener listener) {}

    protected void addClient(Client client) {}

    protected void removeClient(Client client) {}

    public int getClientCount() {}

    public Client getClient(int idx) {}
}
```

Kodesnutt 4: Utkast av Server-klasse

6.1.5 Implementasjon av TeltonikaFMC-familien

I dette delkapittelet vil vi forklare hvordan vi satt opp kommunikasjonsklassene for enhetsfamilien som vi valgte i Kapittel 4.2, TeltonikaFMC-familien.

TeltonikaFMC-pakke

Vi måtte lage en egen «Packet» for TeltonikaFMC-familien sin data, som vi gjorde ved å arve fra «Packet»-klassen. Siden TeltonikaFMC-familien sin data blir sendt som en liste med bytes, lagde vi en objekt-variabel med riktig datatype som vi kalte «data», og overstyrte funksjonen som hentet denne dataen.

```
public class TeltonikaFmcPacket extends Packet {
    private final byte[] data;

    public TeltonikaFmcPacket(TeltonikaFmcClient client, byte[] data) {
        super(client, DeviceFamilies.TELTONIKA_FMC);
        this.data = data;
    }

    @Override
    public byte[] getData() {
        return data;
    }
}
```

Kodesnutt 5: Utkast av TeltonikaFMCPacket-subklasse

TeltonikaFMC-klient

Vi måtte også lage en egen «Client»-klasse for TeltonikaFMC-familien. Den holder på maksstørrelsen til TeltonikaFMC-pakkene og bruker innkommende data til å konstruere TeltonikaFMC-pakker gjennom «onReceive»-funksjonen. I TeltonikaFMC-familien sin kommunikasjonsprotokoll, så forventes det en respons når serveren mottar data. Ved mottak av første datastrøm (handshake) så skal serveren respondere tilbake til enheten med 1 (godkjent) eller 0 (ikke godkjent). Ved mottak av ordinær datastrøm, så responderer klassen med antall interne TeltonikaFMC-pakker.

```java
public class TeltonikaFmcClient extends Client {

    public TeltonikaFmcClient(Server handler, Socket socket,
                           DataInputStream input, DataOutputStream output) {
        super(handler, socket, input, output, 1280);
    }

    @Override
    public void onAcceptConnection(boolean flag) {
        send(DataConverter.int8ToByte((byte) (flag ? 1 : 0)));
    }


    @Override
    public void onSendReceipt(ParsedData data) {
        send(DataConverter.int32ToByte(data != null ?
                                             data.getLength() : 0));
    }

    @Override
    protected Packet onReceive(byte[] data) {
        return new TeltonikaFmcPacket(this, data);
    }
}
```

Kodesnutt 6: Utkast av TeltonikaFMCClient-subklasse

TeltonikaFMC-lytter

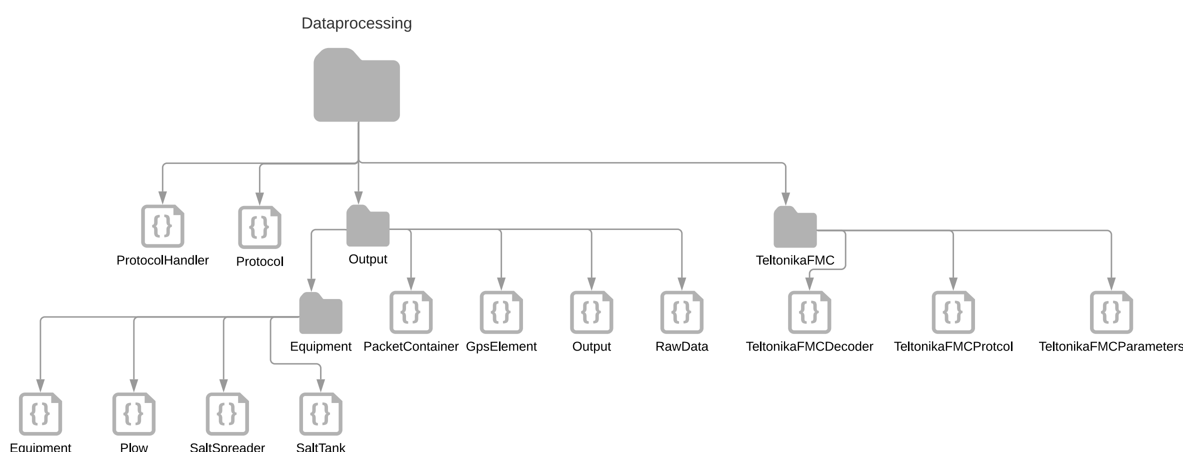
TeltonikaFMC-familien trengte en egen lytter, som kan lytte på portnummeret som er valgt i konfigurasjonen av enhetene i 6.7. Vi konstruerte en subklasse fra «Listener»-klassen som vi kalte «TeltonikaFMCListener». Når det dannes en ny tilkobling med en enhet, sørger TeltonikaFMC-lytteren for å lage en ny «TeltonikaFMCClient»-instans, som holder tilkoblingen åpen på en egen tråd i programmet.

```java
public class TeltonikaFmcListener extends Listener {

    public TeltonikaFmcListener(Server handler) {
        super(handler, 8160);
    }

    @Override
    protected Client createClient(Server handler, Socket socket,
                           DataInputStream input, DataOutputStream output) {
        return new TeltonikaFmcClient(handler, socket, input, output);
    }
}
```

Kodesnutt 7: Utkast av TeltonikaFMCListener-subklasse

6.2 Protokoller og tolkning

Når programmet har mottatt datapakker gjennom Nettverksmodulen, må disse prosesseres, tolkes og formateres. Som beskrevet i kravene skulle det være mulig å utvide programmet slik at det kunne motta og tolke data fra ulike produsenter og enhetsfamilier. Vi måtte derfor strukturere programmet slik at det kunne utvides og være skalerbart. Vi lagde en ny Java-pakke som vi kalte «Dataprocessing» der prosessering av datapakker skulle bli gjort i henhold til enhetens protokoll. Figur 13 viser filstrukturen til Java-pakken. Dette er den delen i prosjektet som vi har jobbet mest med, og som vi har hatt mest fokus på.



Figur 13: Filstruktur i "Dataprocessing" Java-pakke

6.2.1 Protokoll

En protokoll, i datamaskinsammenheng, er formater og fremgangsmåter som må følges for å få datamaskiner til å kommunisere [21]. I vårt tilfelle er en protokoll en fremgangsmåte for å tolke dataen fra ulike enheter, og formatere den til et universalt format. Ettersom hver enhetsfamilie ofte har sin egen protokoll definert av produsenten, måtte vi utvikle en grunnleggende struktur for protokollene. Denne strukturen skulle gjøre det mulig å utvide programmet til å støtte flere enhetsfamilier, ved å lage variasjoner av strukturen. Vi identifiserte fellesfunksjoner som vi ønsket at alle protokoller skulle ha, disse fellesfunksjonene vil nå bli beskrevet ytterligere.

Klassifisere og validere data

Vi forventet at dataen som skulle bli sendt fra enhetene, kunne tolkes som én av to typer. De to typene er «Handshake» og «Data», og har forskjellig struktur og hensikt. Vi har også en tredje klassifikasjon for når det er noe galt med dataen; «Error».

- **Handshake:** Den første pakken mottatt fra en enhet er som oftest en «Handshake». Den har som formål å identifisere enheten og innlede videre kommunikasjon.
- **Data:** Mesteparten av dataen mottatt fra en enhet faller i denne kategorien, ettersom en «Handshake» som oftest blir inngått én gang. Det er denne datatypen som har et meningsfullt innhold, ettersom en «Handshake» kun er til for å opprette kommunikasjonsleddet, og data-kategorien inneholder målingene enheten sender fra kjøretøyet.
- **Error:** Er det noe galt med dataen som blir mottatt blir denne kategorien returnert. Det kan være feil størrelse på dataen, eller feilformatert data, som f.eks. byte endret under sending.

Prosessere håndtrykk

Etter den mottatte dataen har blitt identifisert som en «Handshake», skal denne funksjonen tolke datastrømmen for å identifisere enheten ved å finne enhetens identifikasjonsnummer. Dette skal gjøres ved å implementere enhetsfamiliens protokoll for «Handshake». Ved godkjent «Handskake» skal identifikasjonsnummeret sendes som returverdi, ellers er returverdi -1. Identifikasjonsnummeret blir brukt for å knytte en enhet til mottatt data.

Prosessere data

Denne funksjonen har som formål å validere og tolke dataen fra enheten ut ifra enhetsfamiliens protokoll. Etter å ha tolket relevant data skal dataen formateres til et universalt format som blir likt for alle enheter. Dette blir beskrevet ytterligere i 6.2.2.

Disse funksjonene ble deklarert i en Java-Interface for å oppnå abstraksjon, slik at det er lett å utvide til nye enheter ved å implementere metodene fra «Protocol».

```
public interface Protocol {

    enum Codes{
        Error,
        Data,
        Handshake
    }

    Codes validateData(Packet packet);

    long processHandshake(Packet packet);

    ParsedData processData(Packet packet);
}
```

Kodesnutt 8: Protocol-Interface

6.2.2 Output

Vi ønsket at uavhengig av enhetens enhetsfamilie, dataprotokoll og dataformat, så skulle all dataen fra enhetene formateres i til et universalt format. Dette var for å gjøre det lettere å lagre dataen uavhengig av enheten senere i programmet. Vi lagde derfor en ny Java-pakke som vi kalte «Output». Her skal den tolkede og formaterte dataen lagres. Disse dataene er blant annet GPS-data, enhetsidentifikasjonsnummer og «I/O hendelser».

Dette er egentlig et unødvendig ledd, ettersom protokollklassene som tolker dataen, kunne ha sendt dette til databasen direkte, uten å bli formatert til det universale formatet. Vi bestemte likevel for å gjøre dette slik at modulene blir uavhengig av hverandre. Dette gjør systemet mindre innviklet, og siden all dataen skal bli på det samme universale formatet, er det lett å teste akkurat denne dele av koden. En annen fordel med å formatere til universalt format før dataen lagres i databasen, var at man da kun trengte å tolke dataen og sette den inn i det universelle formatet når man lagde en ny protokoll, og slapp å skrive egne SQL-spørringer i hver «Protocol»-implementering. Likevel er det en ulempe med dette, og det er at siden alt må bli sendt i samme universale format, kan det gjøre at noen enheter ikke blir kompatible med programmet.

6.2.3 Protokollbehandler

Vi trengte et objekt som kunne kontrollere hvilken protokoll som skulle velges. Vi lagde derfor en klasse som vi kalte «ProtocolHandler». Dette er grensesnittet til kjerneprogrammet for å styre Dataprosesseringsmodulen. Den har som formål å distribuere datapakkene til de relevante protokollene og returnere protokollens resulterende data. Protokollene blir lagret i et «EnumMap», der «ProtocolHandler»-klassen kan finne riktig protokoll til riktig enhet, ut ifra enhetsfamilien sin «Enum»-verdi.

```
public class ProtocolHandler {

    private static ProtocolHandler instance;

    private final EnumMap<DeviceFamily, Protocol> protocols = new
                                          EnumMap<>(DeviceFamily.class);

    private ProtocolHandler(){}

    public static ProtocolHandler getInstance() {
        if (instance == null) instance = new ProtocolHandler();
        return instance;
    }

    public void init(){
        protocols.put(DeviceFamilies.TELTONIKA_FMC,
                        new TeltonikaProtocol());
    }

    private Protocol getProtocol(DeviceFamilies id) {
        return protocols.get(id);
    }

    public Protocol.Codes validateData(Packet packet) {
        return getProtocol(packet.getDeviceFamily())
                        .validateData(packet);
    }

    public long processHandshake(Packet packet) {
        return getProtocol(packet.getDeviceFamily())
                        .processHandshake(packet);
    }

    public ParsedData processData(Packet packet) {
        return getProtocol(packet.getDeviceFamily())
                        .processData(packet);
    }
}
```

Kodesnutt 9: ProtocolHandler-klasse

6.2.4 Implementasjon av TeltonikaFMC-protokollen

I dette delkapittelet vil vi forklare hvordan vi satt opp protokollen for enhetsfamilien som vi valgte i 4.2; TeltonikaFMC-familien. Dette er et av de viktigste bidragene til løsningen, ettersom den viser hvordan man kan legge til ulike protokoller, uten å måtte endre på noe fundamentalt i programmet.

Det første vi gjorde var å implementere «Protocol»-Interface, i en klasse vi kalte «TeltonikaFMCProtocol». Her måtte vi definere hvordan dataen fra TeltonikaFMC-enheter skulle valideres og tolkes. Før vi forklarer hvordan vi implementerte metodene, skal vi forklare hvordan TeltonikaFMC-familien strukturerer dataen sin.

TeltonikaFMC har to forskjellige pakkestrukturer. En for «Handshake» og en for «AVL data-pakker». Begge disse kommer i form av en rekke heksadesimaler, som man kan dele inn i flere byte av ulike størrelse, der hver av disse delene beskriver en del av dataen.

«Handshake»

De to første bytene i TeltonikaFMC sin «Handshake» forteller hvor mange byte som er skrevet til pakken, som etterfølges av enhetens IMEI-nummer representert som en String.

Lengde	IMEI
2 byte	n byte

For eksempel kan en «Handshake» se slik ut:

000F333536333037303432343431303133

000F forteller oss at IMEI-en består av 15 byte og disse bytene oversettes til Tekst-format: 356307042441013

«AVL datapakker»

I «AVL datapakker» er datastrukturen satt av en kodek. Teltonika sine produkter har flere kodeker, men i vår oppgave har vi kun implementert «Codec 8» og «Codec 8E». Her nevner vi kun «Codec 8E», ettersom disse kodekene er svært like [5].

«Codec 8E» er en kodek utviklet av Teltonika, og gjør det mulig å dekode enhetens «AVL datapakker». Disse pakkene blir sendt fra enhetene og kan holde flere «AVL records», som er data fra kjøretøyet som vi er interessert i. All denne dataen, pluss flere egenskaper, blir lagret i en lang liste med byte. Tabellene nedenfor beskriver hver del av en slik liste [5].

Tabell 6: Codec 8E format

Innledning 0x00000000	**Datastørrelse**	**Codec ID**	**Antall «AVL records» 1**	**«AVL records»**	**Antall «AVL records» 2**	**CRC-16**
4 byte	4 byte	1 byte	1 byte	n byte	1 byte	4 byte

Tabell 7: Codec 8E forklaring

Innledning	«Hode» til AVL-pakken er alltid fire byte med verdi 0. Dette er en god måte å skille mellom en «Handshake» og en «AVL data-pakke».
Datastørrelse	Er antall byte i pakkene som er mellom 'Codec ID' og 'Antall «AVL records» 2'
Codec ID	Er en enkel byte som beskriver Codecen på pakken. I vårt eksempel vil byten være «0x8E».
Antall «AVL records» 1	Er en byte som beskriver antall «AVL records» som pakken inneholder.
«AVL records»	Er hendelsesdata som blir sendt med pakken. Denne inneholder alltid posisjon- og tidsdata. Den kan også ha «I/O element», som endringer i «Plog nede» eller «Bil på». Hver av disse kan ha ulike antall byte, og ulike måter å være formatert på.
Antall «AVL records» 2	En gjentagelse av 'Antall «AVL records» 1' dette må være samme verdi.
CRC-16	Et tall som genereres ut ifra dataen fra 'Codec ID' til 'Antall «AVL records» 2'**.** Tallet skal sjekke om noe form for datakorrupsjon av dataen har tatt sted under sending og følger CRC-16/IBM standarden [4].

Videre til hvordan vi implementerte metodene i TeltonikaFMCProtocol-klassen:

Validere dataen

Som beskrevet i 6.2.1, trengte vi en funksjon for å klassifisere og validere dataen før videre prosessering. Denne funksjonen skulle se etter kjennetegn i dataen for å klassifisere pakken mellom en «Handshake» og en «AVL datapakke». Dette gjøres ved å sjekke de 4 første bytene. Hvis funksjonen ikke klarer å finne kjennetegnene, eller at lengden ikke er som forventet eller at «CRC»-sjekkverdien ikke samsvarer med dataen, er det noe galt og funksjonen sender «Error» som returverdi, ellers returneres pakketypen.

«Handshake»

Når pakker blir klassifisert som en «Handshake», trenger vi å identifisere enheten. I TeltonikaFMC sitt tilfelle sendes enhetens IMEI-nummer som vi kan bruke som identifikasjonsnummer for fremtidige mottatte pakker. Dataen leses ved bruk av implementerte hjelpefunksjoner, som leser av data fra bytelisten, og gjør det om til tall av datatypen «Long».

```
public long processHandshake(Packet packet) {
    byte[] data;
    try {
        data = packet.getData();
    } catch (ClassCastException e) {
        return -1;
    }

    ByteStreamReader reader = new ByteStreamReader(data);

    int expectedLen = reader.readInt16();

    try {
        return Long.parseLong(reader.readString(expectedLen));
    } catch (NumberFormatException e) {
        return -1;
    }
}
```

Kodesnutt 10: «processHandshake»-funksjonen

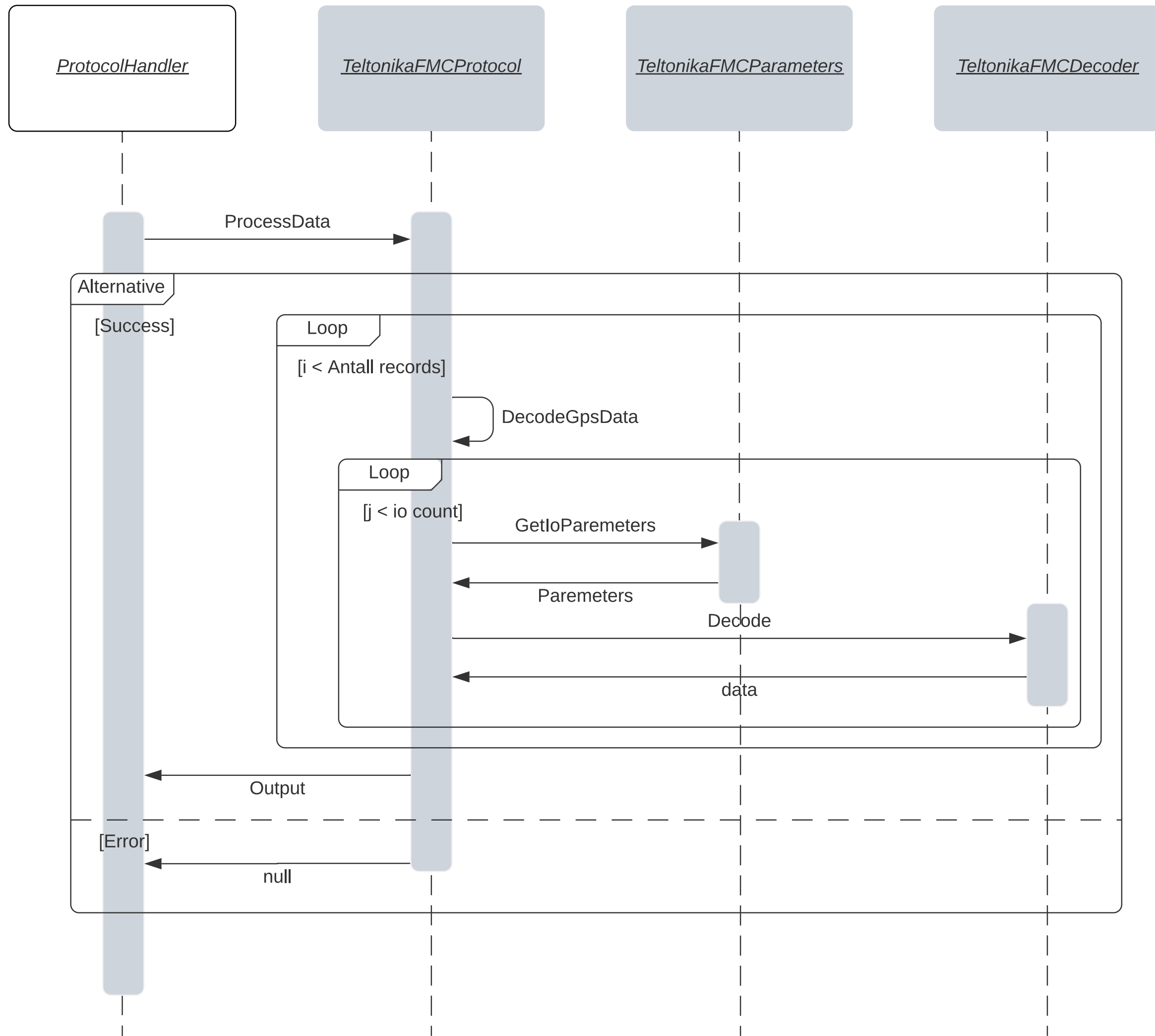
Prosessere Data

Når funksjonen for prosessering av data blir kjørt, hentes bytelisten fra «TeltonikaFMCPacket» og lagres i et hjelpeobjekt. Dette hjelpeobjektet er en egenprodusert hjelpeklasse «ByteStreamReader». Denne hjelpeklassen leser av bytelisten på riktige steder, og omformer dette til ønsket type data i returverdi.

Først blir det initiert en instans av «Output»-klassen for å lagre den tolkede dataen. Så blir det lest hvilken type kodek det er, for å vite hvordan dataen er formatert og hvilke funksjoner som må kalles. Etterpå leses det inn hvor mange «AVL record» som finnes i bytelisten. Så itererer funksjonen gjennom alle «AVL record»-elementene. Hver «AVL record» har et tidsstempel og GPS-data som er det første som blir lagret i «Output»-klassen. Hver «AVL record» kan også ha flere «I/O element». Det er disse som beskriver hendelsene til kjøretøyet.

Teltonika har en oversikt over alle «I/O element»-ene som den valgte enheten leser fra kjøretøyet. Denne oversikten inneholder bl.a. identifikasjonsnummeret til «I/O elementet», navnet på «I/O elementet» og antall byte som må leses fra datastrømmen for å tolke verdien. Denne oversikten blir brukt i en klasse som vi har kalt «TeltonikaFMCParameters» som har funksjoner for å hente slik data om «I/O element».

Når vi skulle loggføre dataen, var det viktig at vi kun loggførte den relevante dataen, og hoppe over irrelevante hendelser. Vi lagde en «TeltonikaFMCDecoder»-klasse. Den har dekoder-funksjoner lagret i et «HashMap» som er indeksert til et «I/O element» sitt identifikasjonsnummer. Dette gjør det lett å legge inn et nytt «I/O element» i fremtiden. Programmet forsøker å hente riktig dekoder-funksjon, ved å sende inn «I/O elementet» sitt identifikasjonsnummer til dekoder-klassen sitt «HashMap». Finnes det ingen dekoder-funksjon, blir standardfunksjonen kalt, som hopper over bytene i listen. Ellers blir den leste dataen lagret i «Output»-instansen. Disse dekoder-funksjonene skulle vi implementere etter vi fikk test-data fra en installert enhet, ettersom det ikke var god nok dokumentasjon på hva verdiene representerte. Oppstår det noen feil under lesing av verdiene fra datapakken, for eksempel i form av ugyldig GPS-data eller at et «I/O element» ikke finnes, avbrytes tolkningen og funksjonen returnerer «Null» (ingen verdi). Se sekvensdiagrammet i Figur 14 for et detaljert hendelsesforløp av «ProcessData»-funksjonen.



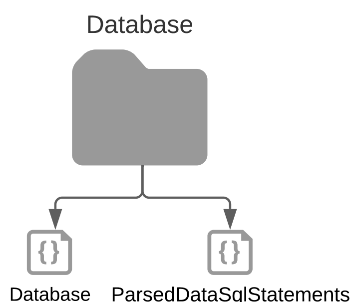
Figur 14: Sekvensdiagram av TeltonikaFMC-protokoll

```
decoders.put(DEFAULT, (reader, packet, parameters) -> {
    int bytes = parameters.getBytes();
    if (bytes != 1 && bytes != 2 && bytes != 4 && bytes != 8)
        bytes = reader.readInt16();
    reader.skip(bytes);
});

decoders.put(161, (reader, packet, parameters) -> {
    var value = reader.readInt16() * parameters.getMultiplier();

    Plow plow = packet.getEquipment(Plow.class);
    plow.addRawData(parameters.getName(), String.valueOf(value));

    plow.down = value < parameters.getLimit();
});
```

Kodesnutt 11: Ekempel på dekoder-funksjoner

6.3 Database

Når programmet har tolket og formatert dataen fra kjøretøyet, så skal den lagres i en database. Programmet må koble seg til databasen, og lagre data i riktig format og på riktig sted. Vi lagde en ny Java-pakke som vi kalte «Database» der lagring av data i databasen skulle bli gjort. Figur 15 viser filstrukturen til Java-pakken. Før vi går igjennom hvordan vi programmerte programmet til å skrive SQL-spørringer, vil vi gå igjennom hvordan vi konstruerte databasen i MariaDB/HeidiSQL.



Figur 15: Filstruktur av databasemodulen

6.3.1 Konstruksjon av database med MariaDB/HeidiSQL

Som beskrevet i 5.3.2 satt vi opp et databasediagram for å normalisere databasen. Etter dette var gjort laget vi databasen i MariaDB. Vi brukte HeidiSQL, som var koblet mot MariaDB, ettersom det var lettere å sette opp tabeller i dette, og lett å sette inn test-data. I Figur 16 kan man se hvordan den ferdige databasen så ut i HeidiSQL.

File Edit Search Tools Go to Help

Database filter | Table filter | Host: 127.0.0.1 | Database: caradmin_iot_data | Query*

Unnamed
caradmin_iot_data 272.0 KiB

Name	Rows	Size	Created	Updated	Engine
datapacket	0	32.0 KiB	2021-05-13 13:26:43		InnoDB
DeleteEvent		112 B	2021-05-13 13:26:43	2021-05-14 13:39:19	
device	6	48.0 KiB	2021-05-13 13:26:43		InnoDB
device_family	3	32.0 KiB	2021-05-13 13:26:43		InnoDB
gps_data	0	32.0 KiB	2021-05-13 13:26:43		InnoDB
io_event	0	32.0 KiB	2021-05-13 13:26:43		InnoDB
producer	2	16.0 KiB	2021-05-13 13:26:43		InnoDB
raw_data	0	32.0 KiB	2021-05-13 13:26:44		InnoDB
vehicle	4	32.0 KiB	2021-05-13 13:26:44		InnoDB
vehicle_type	3	16.0 KiB	2021-05-13 13:26:44		InnoDB

Figur 16: Den ferdigkonstruerte databasen

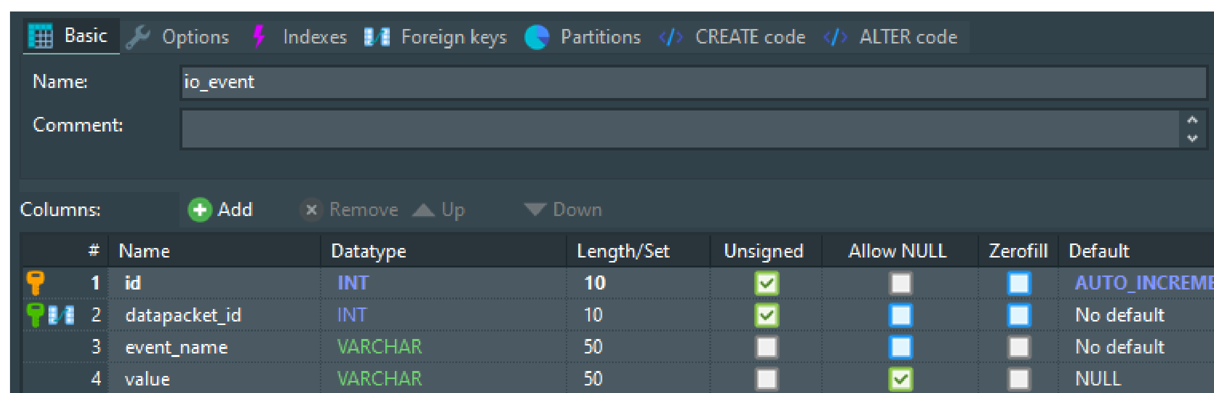
For hver entitet i diagrammet, så lagde vi en tilsvarende tabell i databasen vår, og hvert element innenfor entiteten skulle ha en rad i tabellen. MariaDB tilbyr flere nyttige egenskaper for radene i en tabell. Det var derfor viktig å følge disse slik at det ikke ble kluss når data skulle settes inn i tabellen.

Auto-inkrement

Hvis entiteten skulle ha et unikt identifikasjonsnummer, og som ikke hadde dette fra en verdi i seg selv, skulle vi bruke auto-inkrement funksjonen i MariaDB. En ny oppføring i databasen ville da automatisk få et unikt identifikasjonsnummer. Dette er nyttig ettersom vi slapp å finne et unikt tall når vi skal legge til elementer i databasen, ettersom vi bare kunne la feltet for identifikasjonsnummeret stå tomt.

Ikke «Null»

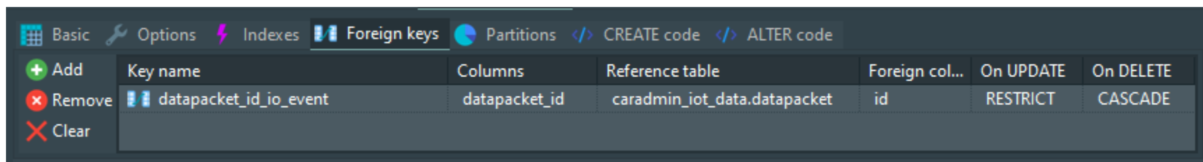
Vi brukte flere metoder for å forsikre oss at all data ble satt riktig inn i tabellen. En av disse var å la rader ha egenskapen at verdien ikke kunne være «Null», altså et tomt felt. Dette gjorde at det måtte være data i disse feltene i riktig format, og man får en feilmelding om man prøver å legge til tom data i disse feltene.



Figur 17: Tabell med rader for IO-hendelser

Nøkler

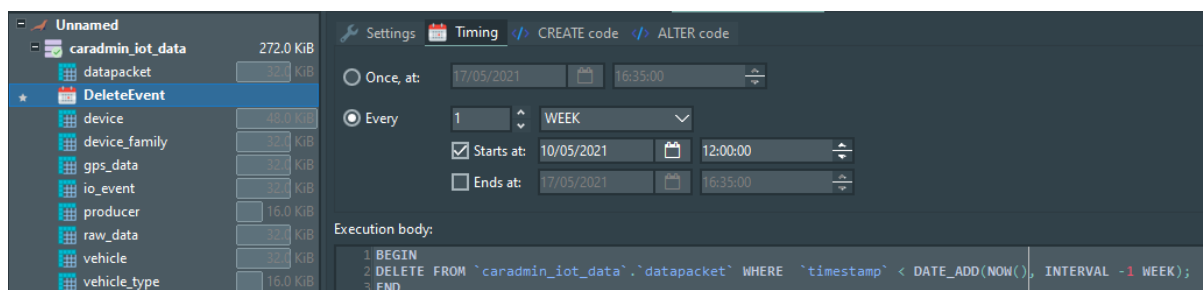
Alle tabellene skulle ha en primærnøkkel som forsikret at verdiene i tabellene hadde et unikt identifikasjonselement. Dette uttrykkes gjennom en gul nøkkel i HeidiSQL. I noen tilfeller skulle vi også ha fremmednøkler koblet på rader mellom flere tabeller. Dette gjorde vi ved å bruke HeidiSQL sine verktøy for fremmednøkler. I Figur 18 kan du se et eksempel på dette.



Figur 18: Fremmednøkkelverktøy i HeidiSQL

Minimering av data

I kapittel 3.2.3 stilles det krav til jevnlig sletting av data for å unngå lagring av unødvendig informasjon og for å tilfredstille kravene for personvern nevnt i 3.2.6. Dette løste vi ved å bruke MariaDB sin innebygde «Event Scheduler». Den kjører «Events» som er objekter som inneholder SQL-spørringer som kan bli kjørt på et senere tidspunkt eller i et gitt intervall [22]. Vi lagde en «Event» som slettet all data som var en uke gammel, ettersom ETC ikke hadde satt noe krav på tidsintervallet, men dette kan endres i databasen på et senere tidspunkt. Figur 19 viser hvordan dette ble gjort i HeidiSQL. I standardinnstillingene til en MariaDB-database så er «Event Scheduler» skrudd av, og for at denne funksjonalieten skal kjøres må man følge instruksene definert i MariaDB sin dokumentasjon[3] for å skru på funksjonaliteten.



Figur 19: Event Scheduler i HeidiSQL

Problemløsning med sletting av subelementer

Det kan oppstå problemer når elementer i databasen skal bli slettet. For eksempel: Man sletter et pakkelement fra databasen. Det tilhørende I/O-elementet og GPS-elementet vil fortsatt være lagret i databasen, med en referanse til en pakke som ikke finnes lenger. Dette kan løses ved å bruke MariaDB sine On DELETE funksjoner for fremmednøkler. Ved å endre denne fra RESTRICT til CASCADE for GPS-element og I/O-element, vil disse automatisk bli slettet når pakken de er koblet til gjennom fremmednøkkelen blir slettet. Se kolonnen helt til høyre i Figur 18.

[3] https://mariadb.com/docs/reference/mdb/system-variables/event_scheduler/

id	datapacket_id	longitude	latitude	altitude	angle	no_of_satellites
1	1	14.48	11.355	66	6	2
2	2	38.973	32.6	585	168	7
3	3	83.16	16	889	187	0
4	4	23.703	15.486	147	357	8
5	5	84	89.998	661	261	11
6	6	80.85	82.7	413	112	7
7	7	74.2194	70.847	365	221	9
8	8	21.1	18.6	29	107	5
9	9	19.5	81.26	278	28	1
10	10	18.79	42.1	17	199	11
11	11	68.02	8.9244	983	276	3
12	12	4.916	61.31	660	294	1
13	13	74.05	55.2	333	165	5

Figur 20: Test data i GPS

6.3.2 Tilkobling til database

For å kunne koble til en database og utføre spørringer i Java, trenger programmet en driver. Vi lastet derfor ned mysql-connector-java[4] som er en driver for bruk av MySQL og MariaDB i Java, og la den til som et bibliotek i prosjektet. For å kunne etablere tilkoblinger og utføre spørringer til databasen lagde vi en ny klasse som vi kalte «Database». Når programmet skal forsøke å koble seg til databasen, trenger vi IP-adressen, brukernavnet og passordet til databasen. Denne informasjonen valgte vi å lagre i en «properties»-fil slik at det er lett å endre informasjonen, dersom det ble nødvendig. Denne filen blir lest når Databasemodulen initieres, og verdien lagres i database-klassen som medlemmer. For å hindre at en tilkobling til databasen blir stående og kjøre uten å stoppe, så gjorde vi det slik at når programmet skal utføre spørringer, så lages det en ny tilkobling som stenges etter spørringen er utført. I ettertid så vi at vi heller burde ha implementert en «Connection Pool», der tilkoblingen går i dvalemodus mellom hver spørring, og man slipper å koble opp på ny for hver spørring, som tar unødvendig tid. Vi valgte å ikke fokusere på dette, ettersom programmet kun skulle være en prototype, og det ville ta tid og gjøre om på det vi allerede hadde implementert når vi oppdaget dette.

[4] https://dev.mysql.com/downloads/connector/j/

6.3.3 SQL-spørringer til databasen

Etter å ha satt opp tilkoblingen til databasen, implementerte vi funksjoner som skulle sette inn og fjerne rader fra databasen. Dette blir gjort gjennom SQL-spørringer som blir sendt til databasen gjennom programmet. Her valgte vi å bruke «Prepared Statements». Etter en totalvurdering, kom vi fram til at SQL-injeksjon er lite sannsynlig. For det første må SQL-injeksjoner bli sendt som en ASCII-verdi for at den skal kunne påvirke SQL-spørringene våre, og på nåværende tidspunkt aksepterer vi ingen verdier av denne typen. For det andre må personen som vil utføre injeksjonen, måtte få tilgang til en enhet og kontrollere hva den ville sende. Vi velger likevel å motvirke dette ved å bruke Prepared Statements for å følge standarden og i tilfelle programmet skulle utvides i fremtiden, og det da ville bli aktuelt.

«PreparedStatement» er en klasse i Java som gjør det mulig å lage forberedte spørringer. Dette fungerer slik at i elementene i spørringen som skal settes inn blir satt som spørsmålstegn i selve spørringen, for så å bli satt inn som verdier med å bruke klassens funksjoner. Eksempel: I Kodesnutt 12 under er en funksjon som legger til et GPS-element i databasen. Funksjonen tar inn et «Connection»-objekt som er tilkoblingen til databasen, «GpsElement»-dataen og IDen på datapakken «GPS-dataen» kommer ifra. Vi vet hvilken tabell og hvilke typer verdier som skal settes inn, og skriver spørringen, men setter spørsmåltegn der de faktiske verdiene skal sette. Så setter vi verdiene til spørringen, ved å kalle diverse «set»-funksjoner, for å sette inn riktig verdi. Til slutt kalles funksjonen «executeUpdate» som vil prøve å sende spørringen til databasen, slik at GPS-elementet blir lagt til.

```
public static void addGpsElement(Connection connection, GpsElement gps,
                                 int dataPacketID) throws SQLException {
    PreparedStatement ps = connection.prepareStatement
        ("INSERT INTO gps_data (`datapacket_id`, `longitude`,
      `latitude`, `altitude`, `angle`, `no_of_satellites`)
      VALUES (?,?,?,?,?,?)");

    ps.setInt(1, dataPacketID);
    ps.setDouble(2, gps.getLongitude());
    ps.setDouble(3, gps.getLatitude());
    ps.setShort(4, gps.getAltitude());
    ps.setByte(5, gps.getAngle());
    ps.setShort(6, gps.getNoOfSatellites());
    ps.executeUpdate();
}
```

Kodesnutt 12: Eksempel på SQL-spørring

Vi implementerte disse SQL-spørringsfunksjonene:

Legg til datapakke som sender pakkeinformasjon til `datapacket`-tabellen i databasen. Dette er pakkedata som f.eks. enhetens ID og tidsstempel.
Fjern datapakke som fjerner data fra `datapacket`-tabellen ved bruk av datapakkeID. Denne blir kalt hvis det skjer noe galt under innleggelse av data fra denne pakken.
Legg til «I/O event» som legger inn eventID, evenetverdi og datapakkeID inn i «I/O event»-tabellen.
Legg til rådata som legger til utolket data inn i rådata tabellen, dette er da data som eventID, eventnavn og verdi.
Legg til Gps legger inn GPS relatert data in i GPS data tabellen. Dette er da data som pakkeID, lengdegrad, breddegrad, høyde, vinkel og antall satellitter.
Enhet ID-sjekker som leter etter en ID til en enhet i databasen og returnerer den vis funnet.

6.4 Ekstra kode og verktøy

6.4.1 AVLCreator verktøy

Vi trengte en måtte å få test-data mens vi ventet på installasjonen av enheten, så vi lagde en AVL pakke generator som lot oss konstruere egne pakker til testing av programmet. Verktøyet ble programmert i Javascript og HTML fordi dette lot oss raskt kjøre og teste programmet. Programmet følger «Codec 8E» formatet beskrevet i 6.2.4.

Figur 21: Skjermbilde av AVLCreator

Ved å bruke dette verktøyet kunne vi lage test-data, som vi kunne bruke til å lage valideringstester, og for å teste programmet mot realistiske datapakker.

6.4.2 Klientapplikasjon

Når vi lagde Nettverksmodulen lagde vi en enkel klientapplikasjon som kunne kobles til programmet for å teste kommunikasjon og håndtering av klienter på serveren. Her brukte vi et program som en av oss hadde utviklet fra grunnen av under en eksamen som et utgangspunkt. Klientapplikasjonen kunne sende AVL-pakker til programmet og skrive ut tilbakemeldingen som serveren sendte tilbake.

DebugClient

000000000000000298E010000000000000000000000000000000000000000000000000000000000

Connect

Disconnect

Send

Connecting to localhost on port:8160
Connected

MESSAGE SENT:
[B@48287d6b

MESSAGE FROM SERVER:
[1]

MESSAGE SENT:
[B@168f6dc1

MESSAGE FROM SERVER:
[0, 0, 0, 1]

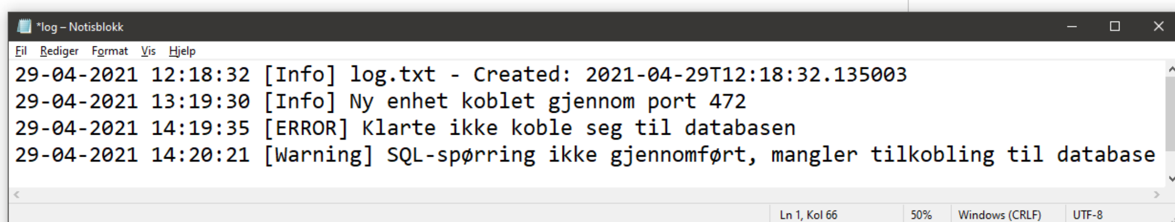
Figur 22: Skjermbilde av klientapplikasjon

6.4.3 Simulasjon

Når det oppstod komplikasjoner med installasjonen av enhetene, (beskrevet ytterligere i 6.7), så vi et behov for å videreutvikle klientapplikasjonen for å kunne simulere en enhet. Vi oversatte AVLCreatoren fra Javascript til Java for å kunne integrere funksjonaliteten inn i klientapplikasjonen. Simulatoren danner «AVL datapakker», med tilfeldig data, i et gitt intervall og sender dataen til serveren. Ved oppstart av simulasjonen tar programmet inn IP-adressen som skal tilkobles, IMEI-nummert simulasjonen skal bruke og hvor ofte pakker skal bli sendt til serveren. I etterkant ser vi at vi skulle ha implementert muligheten til å påvirke dataen som ble generert med et enkelt brukergrensesnitt for å ha mer kontroll på pakkene som blir sendt. Dette hadde gitt oss mer muligheter til testing av dekoder-funksjonene i TeltonikaFMC-protokollen, men hadde ikke vært nok til å erstatte virkelig data ettersom dokumentasjonen til Teltonika er ikke helt klar på hva verdiene betyr.

6.4.4 Logger

Vi ønsket en metode å loggføre feil som ville oppstå, og ettersom programmet vil kjøre på en server er det ikke alltid nok å skrive dette ut i konsollen. Hvis programmet startes på nytt, blir konsollen nullstilt og tidligere feil vil bli umulig å finne igjen. Vi lagde derfor en «Logger»-klasse som lagrer feil, advarsler og info i en tekstfil, slik at man kan gå igjennom dette og se etter feilmeldinger og annen info. Når programmet starter, vil klassen sjekke om det finnes en fil med navn «log.txt». Finnes den, skjer det ingenting, ellers lages en ny tekstfil. Siden klassen har statiske funksjoner, er det da enkelt å skrive feil til filen ettersom man kun trenger å skrive Logger.writeError() hvor som helst i koden. Vi valgte å la det skrives 3 ulike type meldinger. Feilmelding, Advarsel og Info. Disse blir kalt med hver sin funksjon, og det markerer meldingen i logg-filen med kategorien som er valgt. I hver av disse funksjonene kan du legge inn hva meldingsdetaljene er som en String-variabel. Meldingen blir da lagret, med et tidsstempel for hendelsen. En logg-fil kan se slik ut som i Figur 23.



Figur 23: Eksempel på utforming av logg-fil

```java
public class Logger {
    private static final String fileName = "log.txt";
    private static boolean isInitalized = false;

    public static void init() {
        try {
            File logFile = new File(fileName);
            if (logFile.createNewFile()) writeInfo(fileName +" Created:"
                                              + LocalDateTime.now());
            isInitalized = true;
        } catch (IOException e) {
            System.out.println("Feil i Logger klasse");
            e.printStackTrace();
        }
    }
    public static void writeWarning(String logDetails) {
        writeToLog(logDetails, "Warning");
    }

    public static void writeError(String logDetails) {
        writeToLog(logDetails, "ERROR");
    }

    public static void writeInfo(String logDetails) {
        writeToLog(logDetails, "Info");
    }

    private static void writeToLog(String logDetails, String type) {
        if (!isInitalized) init();
        try {
            LocalDateTime date = LocalDateTime.now();
            DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-
                                                  MM-yyyy HH:mm:ss");
            String formattedDate = date.format(format);

            FileWriter writerObj = new FileWriter(fileName, true);
            writerObj.write(formattedDate + " [" + type + "] " +
                                                  logDetails);
            writerObj.write(System.getProperty("line.separator"));
            writerObj.close();

        } catch (IOException e) {
            System.out.println("Klarte ikke skrive til " + fileName);
            e.printStackTrace();
        }
    }
}
```

Kodesnutt 13: Logger-klasse

6.5 Sikkerhet

Autorisering av datapakker gjøres først ved å autorisere avsender og godkjenne tilkoblingen. Dette må gjøres gjennom en «Handshake», hvor innholdet identifiserer avsender mot de registrerte avsenderne i databasen. Et system som dette har som formål å avise uønskede pakker og stenge avsender sin tilkobling umiddelbart hvis avsender ikke har identifisert seg eller ikke er registrert.

I Teltonika sin «Handshake» så er det et IMEI-nummer som sendes. Dette er et unikt identifiserende nummer for mobile enheter, og enhetene som skal brukes får IMEI-nummeret sitt lagret i databasen som en godkjent enhet.

Etter planen skulle vi implementere en del funksjoner som forbedret sikkerheten, men ut i utviklingen, måtte vi se bort fra dem ettersom det var problemer med installering av enheten. Vi måtte da fokusere på å lage et alternativt test-miljø for resten av programmet, og fikk mindre tid til å tenke på sikkerheten. I tillegg krevde noen av funksjonene vi planla å implementere en tilkobling til en reel enhet.

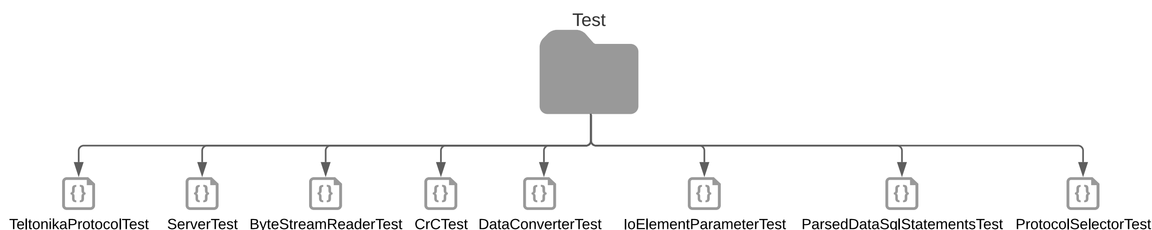
En av løsningene vi planla å implementere var å hvitvaske IP-adresser for å kun lytte etter tilkoblinger innenfor Norge, som minsker angrepsoverflaten betraktelig. En annen løsning var å begrense antall datapakker en klient kunne motta i minuttet for å redusere konsekvensen av et DDoS-angrep.

I vår oppgave har vi forhold oss til å lage en prototype for en bedrift som skal utforske teknologien vi har benyttet. Vi har ikke lagt til rette for intern sabotasje, der en ansatt saboterer prosjektet ved å f.eks. gå inn å slette/endre data i databasen, dele tilkoblingsinformasjon til andre bedrifter/personer eller å dele persondata. Dette er tiltak som utenfor våre prosjektrammer, ettersom fokuset vårt ikke var å lage et ferdig produkt, men å utforske teknologien, å lage en prototype.

6.6 Testing

6.6.1 Unit-testing

Ved starten av utviklingen ble det planlagt at hver klasse og deres funksjoner skulle testes ved hjelp av automatiserte Unit-tester. Disse måtte bli bestått før klassen ble regnet som ferdig i «Kanban»-tavla. Vi lagde en rekke Unit-tester, for de mest kritiske klassene og funksjonene som stod for klassifisering, validering, tolkning og lagring av de ulike datapakkene som blir sendt igjennom programmet. Et eksempel på formen til Unit-testene kan man finne i Kodesnutt 14, som viser hvordan vi tester resultatet fra «ValidateData»-funksjonen i Teltonika-protokollen. En oversikt over alle Unit-testene er å finne i Figur 24.



Figur 24: Filstruktur av Unit-tester

```
public class TeltonikaProtocolTest {


    @Test
    void testValidateData() {
        logger.log(Level.INFO, "Test 1: Data Packet ");
        assertEquals(Protocol.Codes.Data,
                      protocol.validateData(packetData));
        logger.log(Level.INFO, "Test 2: Error 1 Packet ");
        assertEquals(Protocol.Codes.Error,
                      protocol.validateData(packetError));
        logger.log(Level.INFO, "Test 3: Error 2 Packet ");
        assertEquals(Protocol.Codes.Error,
                      protocol.validateData(packetError2));
        logger.log(Level.INFO, "Test 4: Handshake Packet ");
        assertEquals(Protocol.Codes.Handshake,
                      protocol.validateData(packetHandshake));
    }
}
```

Kodesnutt 14: Unit-test av Teltonika-protokollen

6.6.2 Nettverkstesting

Under utviklingen av Nettverksmodulen støttet vi på problemer med å lage Unit-tester. Dette var på grunn av vanskelighetsgraden og den komplekse strukturen som må til for å lage automatiserte tester ved bruk av «Socket»-Java-objekter og objekter som kjører på forskjellige tråder. Det var her klientapplikasjonen gav oss muligheten for å teste Nettverksmodulen manuelt ved å sende test-data og se hvordan dataen flyter gjennom programmet. Dette hjalp oss med å oppdage synkroniseringsfeil og implementere funksjoner for håndtering av uforventete situasjoner.

6.6.3 Stresstesting

Programmet ble stresstestet ved å koble til en simulasjonsklient som sendte «AVL datapakker» med varierende lengde til programmet hvert 60. millisekund. Vi lot denne testen kjøre i en time uten noen avbrudd. Det oppstod ingen feil eller opphoping av data under testingen og vi så på dette som en vellykket test. Under stresstestingen loggførte vi også behandlingstiden til hver pakke som serveren mottok. Med denne dataen regnet vi ut en gjennomsnittlig behandlingstid på omtrent 38 millisekund. Ut ifra kravene satt i kapitel 3.2.2, så visste vi at enhetene burde sende datapakker hvert 120. sekund og kunne derfor regne ut maks antall tilkoblede enheter før det blir for mye datatrafikk for systemet. Vi regnet ut at grensen for maks antall klienter var omtrent 3000, før det tok for lang tid å behandle dataen og den vil da hopes opp i Nettverksmodulen. Denne testen ble kjørt på datamaskinen til en på gruppen med begrenset prosesseringskraft[5] og vi kunne derfor antakeligvis ha fått bedre resultater ved bruk av en dedikert server.

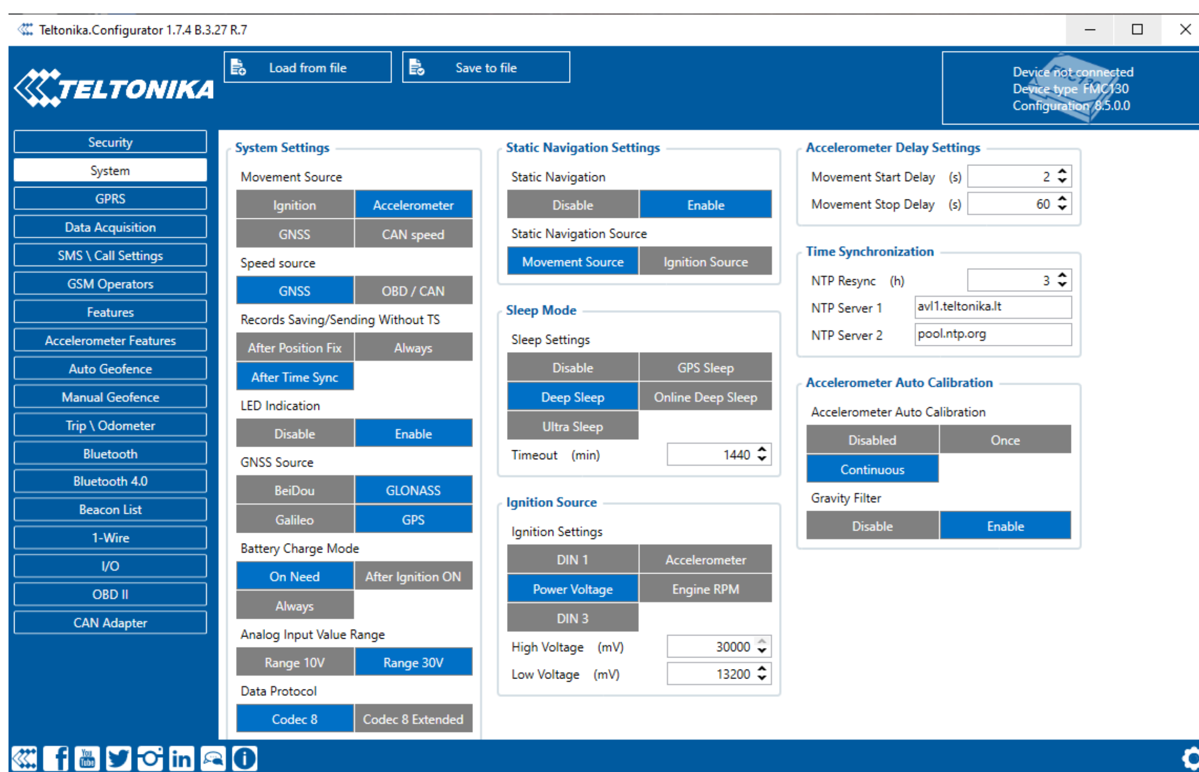
6.6.4 Server-hosting

Server-hosting testet vi ved bruk av en servertjeneste som NTNU har som tilbud til sine studenter og ansatte, kalt «Skyhigh» som kjører en instans av «Openstack». Vi opprettet en dedikert Windows-server i tjenesten deres, som kjørte kontinuerlig. Gjennom dette fikk vi testet hvordan det settes opp tilkobling mellom server og klientverktøyet over internett. Resultatet av dette gav oss en bekreftelse på at Nettverksmodulen sin kode fungerte og at det ikke oppstod noen problemer etter lengre oppetider.

[5] AMD Ryzen 7 1800x

6.7 Installasjon av enhet

Etter hvert som enhetene ble mottatt og programmets utviklingsfase hadde kommet så langt at vi nå burde teste programmet på virkelig datastrøm fra enheten, konfigurerte vi enheten til å følge kravene vi satt opp, i sammen med ETC. Dette gjorde vi ved å bruke konfigurasjonsprogrammet til Teltonika. Programmet lar oss bestemme et bredt spekter av innstillinger, som blant annet kodek valg, hvilken IP-adressen den skal koble seg til, hvilket portnummer den skal koble seg til med, datasendingsintervall og datainnhold. Innstillingene vi satt ble lagret i en nedlastbar konfigurasjonsfil, som vi sendte til ETC for opplasting til enheten. Se Figur 25 for skjermbilde av konfigurasjonsprogrammet.



Figur 25: Skjermbilde av konfigurasjonsprogram

Imens vi ventet på installasjonen av enheten, jobbet vi i mellomtiden med å utvikle et enkelt datapakkeprogram som beskrevet i 6.4.1, for å lage test-data vi kunne sette inn i programmet. Etter hvert viste det seg vanskelig å finne kompatible kjøretøy til enhetene. Etter at ETC hadde flere tilbakefall med å finne et kjøretøy som passet, bestemte vi oss for å utvikle et klientprogram for å sende data til serveren lokalt på maskinen, som gav oss bedre testmuligheter av programmet.

I sluttperioden av prosjektperioden, så konkluderte vi at sannsynligheten for at enheten ble installert på et kjøretøy som særlig usannsynlig. Dette er et utfall som vi allerede forutså i risikotabellen i prosjektplanen, men løsningen til denne konsekvensen, ble noe annet enn planlagt, ettersom prosjektet var i en tidsklemme. Vi kom fram til at å videreutvikle klientprogrammet til et simuleringsprogram var den beste løsningen for å kunne vise til en fungerende løsning som ble beskrevet i 6.4.3.

7 Diskusjoner og resultater

7.1 Resultater

For å sammenligne prosjektets resultater mot prosjektmålene og oppgavebeskrivelsen tar vi opp målene som er beskrevet i kapittel 1.3.

7.1.1 Prosjektmål

En sentral del av prosjektprosessen var å få installert enheten på et kjøretøy, og ettersom det ikke ble oppnådd og virkelige enhetsdatapakker aldri ble mottatt så vil målene bli drøftet med tanke på om programmet kunne tolke de simulerte datapakkene som vi genererte.

Resultatmål

Et av resultatmålene som vi satte tidlig i prosessen var å kunne demonstrere et helhetlig system, fra å motta en datapakke fra et kjøretøy, til dataen blir lagret i en database. Sammenligner vi resultatet vårt med dette målet, så ser vi at det er delvis nådd. Programmet vårt viser at data blir sendt inn, blir tolket, formatert og lagret i en database. Et avvik fra målet var datapakkeopprinnelsen, hvor vi måte erstatte kjøretøyet med en simulert pakkesender, der pakkenes innhold ikke har meningsfull data. Vi hadde heller ingen måte å vite hva verdiene representerer ved tolkning, ettersom vi var avhengige av en installert enhet for å finne ut av hvordan dataen samsvarer med kjøretøyets virkelige handling. For eksempel kunne en enhet sende data om at plogen på kjøretøyet har en vinkel på 90 grader, vi vet ikke om dette er 90 grader opp eller 90 grader ned, vi vet kun at det har skjedd en endring i vinkelen. Vi hadde planlagt å bruke vinkelen for å finne ut om plogen var nede eller oppe for å sjekke om kjøretøyet brøytet eller ikke, og er derfor avhengig av å teste dette på et ekte kjøretøy.

Et annet mål var å kunne levere et system som kunne være utgangspunktet til en backend-løsning for veivedlikeholdsmodulen i CarAdmin. Vi har ikke fått noe innblikk i CarAdmin sitt system, så vi kan ikke svare på om dette målet er nådd. Likevel har vi med tanke på målet, utviklet programmet slik at alt ligger i backend, og ikke krever noe kobling til andre systemer, som gjør det selvstendig. Siden vi har lagd en fungerende prototype, så vil vi likevel si at vi har et utgangspunkt som ETC kan bruke for å videreutvikle en løsning i CarAdmin.

Det siste målet var å kunne finne ut om lesing av CANbus-systemet var en brukbar løsning for å loggføre vedlikeholdsoppgaver. I løpet av prosjektet forstod vi gradvis at samspillet mellom kjøretøy og enheter, var mer komplisert enn vi først trodde. Enhetene må passe til kjøretøyet, ha riktig protokoll og det kan oppstå andre kompatibilitetsproblemer. For eksempel er det ikke alle kjøretøy som lar enheten lese dataen som er av interesse. Gjennom samtaler med ETC så fikk vi forståelsen av at kompatibilitet med enheter og kjøretøy er uforutsigbart. Teltonika sine enheter var ikke kompatible med kjøretøyene ETC hadde til rådighet, og dette gjorde at vi måtte ta en helomvending av oppgaven, mot slutten av prosjektperioden. Om målet er oppnådd er vanskelig å si. I vårt prosjekt viste det seg komplisert og vanskelig å få en brukbar løsning, ettersom enhetskompatibiliteten er uforutsigbart. Om dette er på grunn av enhetsvalget vårt, må videre utvikling og undersøkelser svare på. Programmet vårt har lagt til rette for å fungere med flere enheter, og bruke CANbus-systemet til å loggføre vedlikeholdsoppgaver. Hvis vi hadde riktig enheter til kjøretøyene, ville kanskje programmet vist at å lese CANbus-systemet er en brukbar løsning, men det kan vi ikke si noe om i denne rapporten.

Effektmål

Når det gjelder effektmål, som beskrevet i 1.3 var det vanskelig for oss å se om disse målene er oppnådd, ettersom vi kun har lagd en prototype, som ikke er iverksatt som et produkt.
Vi endret som sagt derfor på vinklingen fra langtidseffekter, til hva ETC kan videreutvikle fra vårt produkt.

Utvidelsene vi skulle tilrettelegge ble alle delvis løst gjennom innhenting av data fra enheter og loggføring av denne dataen i en database. Ved hjelp av denne informasjonen kan ETC implementere frontend-løsninger for visualisering av nyttekjøretøyets posisjon og handling.

Som beskrevet ovenfor, hadde vi ingen måte å vite hva den innhenta dataen representerte og derfor ville ikke den tolkede handlingen til kjøretøyet nødvendigvis representere kjøretøyets virkelige handling. Derfor ser vi på effektmålene som kun delvis løst.

Læringsmål

Læringsmålene satt vi som mål på hva vi skulle lære i prosjektutviklingen. Det ene målet var å erfare å lage et system som bestod av flere ulike teknologier, deriblant IoT-teknologi, dataprotokoller og database-teknologier. Igjennom prosjektet har vi utviklet flere protokoller knyttet til IoT, protokoller knyttet til spesifiserte enheters datastrøm og teknologi innenfor databaser. Med dette har vi lært mye om disse temaene og har blitt erfarne på området IoT.

Et annet læringsmål var å erfare å jobbe med et større prosjekt, hvor man bruker moderne systemutviklingsmetodikk, sammen med en reell arbeidsgiver. Vi jobbet med et stort prosjekt, der vi i kommunikasjon med arbeidsgiver oppnådde dette læringsmålet.

Det siste læringsmålet var å få praktiske erfaringer ved å jobbe med maskinvare som kommuniserer over internett og få en dypere forståelse for feltet «Internet of Things». Ved å jobbe med et stort system, med flere ledd av moduler, har vi fått flere erfaringer rundt teknologien. Vi har erfart, at å jobbe med «Internet of Things» er et komplisert arbeid, der det er mange deler av et system som må samhandle problemfritt for å ha en god løsning. Dette fikk vi god erfaring med i selve utviklingsdelen, der vi så hvordan forskjellige enheter hadde store forskjeller innen bl.a. protokoller og dataformat. Ettersom et system kan ha flere enheter, er det da viktig at selv om protokollene varierer, så skal det samhandles mellom de ulike delene.

7.2 Kritikk av oppgaven

Mot slutten av prosjektet satt vi oss ned for å se på hva som kunne bli gjort bedre i oppgaven, og vi kom fram til flere punkter som vi nå vil gå dypere innpå.

7.2.1 Enheter

Da det kom fram at det var problemer med å installere enheten i kjøretøyet, burde vi prøvd funnet en ny enhet. Siden dette kom frem sent i utviklingen av prosjektet, hadde vi ikke nok tid og ressurser til å lese opp og implementere de nye protokollene, og det kunne være de nye enhetene ikke kom fram i tide. I stedefor lagde vi en simulasjon som beskrevet i 6.4.3. I starten av prosjektet anbefalte ETC oss å se på flere enheter, i tilfelle noe sånt ville skje. Vi så på flere enheter, men bestilte ingen, fordi det virket som Teltonika sine enheter tilfredsstilte alle kravene. I etterkant ser vi at vi burde ha bestilt inn flere enheter fra ulike produsenter i tilfelle noe sånt ville skje.

7.2.2 Tidsbruk

I prosjektplanen skrev vi i gruppereglene at vi forventet at hver deltager jobbet 30 timer i uken. Hvis vi regner fra levering av prosjektplanen, 1. Februar, så tilsvarer dette 13 arbeidsuker, og omtrent 390 timer per deltager, utenom prosjektplanleggingen. I slutten av prosjektet så vi at dette målet ikke ville bli oppnådd. Grunner til dette er at vi flere dager var nødt til å jobbe med gruppeprosjekter i et annet fag vi hadde samme semester som prosjektet. En annen grunn var også at vi ikke fulgte strengt de 30 timene, siden vi så at vi ville komme i mål med milepælene i god tid før fristen. Se Vedlegg H for detaljert tidsbruk per deltager.

7.2.3 «Scrumban»

I starten av prosjektet fulgte vi «Scrumban», vi hadde et «Retrospective Meeting» i starten av uken for å se hvor vi var i prosjektet og hva vi kunne gjøre bedre, og vi brukte «Kanban»-tavla hyppig. Etter hvert sluttet vi å ha disse «Retrospective Meeting». Dette var fordi vi jobbet i sammen hver dag, og følte vi ikke trengte de formelle møtene, ettersom vi hele tiden så hvor vi lå an og visste hvor vi var i prosessen. Vi hjalp hverandre hele tiden og fortalte hverandre hva som måtte gjøres bedre hver dag.

7.2.4 Løsningen

Selv om vi er stort sett fornøyde med løsningen vår, så er det enkelte deler som kunne forbedres eller gjøres annerledes. Dette er blant annet en «whitelisting»-løsning som også baserte seg på IP-adresser, og ikke bare IMEI-nummeret, ettersom det tillater å avise tilkobling før den oppstår. Andre forbedringsmomenter ville være kontroll på antall klienter som er tilkoblet og muligheten til å stoppe nye klienter som en beskyttelse mot DDoS angrep.

Vi ser også at koden kan optimaliseres på enkelt områder og på den måten føre til at prosesseringstiden optimaliseres, dette ettersom ETC forstilte seg 10,000+ klienter, men etter stresstestingen på vår maskinvare, viste at vi kunne maks håndtere 3000 tilkoblinger.

Etter å ha undersøkt på nettet og tatt kontakt med andre bedrifter som leverer tilsvarende produkter som CarAdmin, så fant vi ut at flere har brukt en annen løsning hvor logging av GPS og arbeidsdata blir gjort manuelt via et nettbrett eller andre mobile enheter. Ettersom ETC hadde problemer med å finne kompatible kjøretøy til enhetene våre så er det mulig at andre bedrifter har opplevd det samme og heller gått for en lettere og mindre avansert løsning. Vi tror og håper at med riktig ressurser og tid, vil det gå an å få en god løsning, som gjør flåtestyringen automatisk, uten manuell input fra sjåfør.

En annen ting som kunne ha blitt gjort annerledes er oppsettet av databasen og strukturen til «Output»-klassen beskrevet i 6.2.2. I etterkant ser vi at valgene som vi gjorde ved konstruksjon av disse har vært veldig preget av strukturen til Teltonika sine «AVL-datapakker». Dette kan gjøre det vanskelig å implementere nye enhetsfamilier, ettersom det er mulig at de nye enhetene ikke sender de samme verdiene som de verdiene det universale formatet vi lagde, krever.

7.3 Videre arbeid

Siden oppgaven kun gikk ut på å lage en prototype, er det mange muligheter for videre arbeid. Dette gjelder blant annet å forbedre den nåværende løsningen, men også å videreutvikle den.

7.3.1 Forbedring av løsningen

Når det gjelder forbedring av løsningen vår, så er det flere ting som det kan jobbes videre med.

Effektivitet og struktur av programmet

Som en forbedring av løsningens effektivitet og et steg i retningen mot ETC ønske å kunne håndtere mer enn 10 000 klienter, så er optimalisering av kode en mulighet. Dette kunne innebære å lese seg opp på de Java-biblioteksfunksjonene som ble brukt og undersøke om tilsvarende funksjoner er mer effektive.

En annen løsning var å kunne bruke tråd-deling under dataprosesserings og database spørringene. Dette innebærer å la hver tråd prosessere data og ha sin egen kobling til databasen. En slik løsningen har muligheten til å åpne prosesseringsflaskehalsen betrakting og muligens fjerne «Maks Klient» problemet. En mulig ulempe med dette er synkroniseringsproblemer som kan oppstå når det skal skrives mye data til databasen.

Det er også flere sikkerhetstiltak som kan implementeres, men dette er forklart tidligere i rapporten.

7.3.2 Videreutvikling av programmet

Siden programmet kun er en prototype, er det mye som kan videreutvikles i løsningen.

For det første så kan det legges til flere enhets- og nettverksprotokoller for å gjøre programmet kompatibelt med flere enheter. Blant annet så støtter kun løsningen vår TCP for øyeblikket, og det er derfor gunstig og utvikle en løsning for UDP også, slik at flere enheter kan koble seg til løsningen.

For det andre så kan det være at databasen må oppdateres og endres for å tilpasse seg ETC sitt interne system. Man må da også oppdatere Databasemodul-koden for å utføre riktige spørringer.

Noe av det viktigste som må videreutvikles er å faktisk installere en enhet på et kjøretøy og prøve å koble seg til denne. Man må så teste hva de ulike dataen fra kjøretøyet betyr, og endre på koden sånn at man tolker dataen riktig fra kjøretøyet.

7.4 Evaluering av gruppas arbeid

Under bacheloroppgaveprosessen fikk vi gjort oss erfaringer å jobbe som gruppe om et større prosjekt. Vi vil nå gå igjennom og evaluere om måten vi har organisert oss på har vært en god løsning, og eventuelt hva som kunne bli gjort bedre.

7.4.1 Arbeidsdagen

Igjennom hele prosjektet har vi nærmest jobbet i sammen hver ukedag. Kommunikasjonen ble gjort over Discord, der vi har jobbet i sammen, parprogrammert, stilt spørsmål, og diskuterte løsningene til oppgaven. Dette fungerte bra, ettersom alle var med å gjorde valg underveis, og det ble lett å se hvor vi var i prosessen til enhver tid. Av og til jobbet vi individuelt med oppgaver. Da gikk vi sammen igjennom det som var gjort neste gang vi skulle samles. I slutten av hver arbeidsdag fylte vi inn i timeplanen og så over «Kanban»-tavla hvor vi var i prosessen. Som beskrevet i 7.2.3 gikk vi bort i fra «Retrospective Meetings», men vi mener dette var riktig valg, ettersom det ikke gav oss noe nytt, ettersom vi hele tiden evaluerte prosjektet.

7.4.2 Arbeidsfordeling

I 1.8.2, skrev vi at alle skulle være med å jobbe med alt. Dette fulgte vi opp i starten, men etter hvert så vi at det var mer effektivt å jobbe med forskjellige deler av programmet. Fordeling av arbeidet varierte mye. I programmeringsfasen, tok vi ledige oppgaver i «Kanban»-tavla, og utførte disse. I store oppgaver, og større tekster i rapporten, skrev alle 3 samlet, og jobbet for å kvalitetssikre og få en samlet forståelse for emne.

7.4.3 Kritikk

I etterkant ser vi at arbeidsorganiseringen vår ikke nødvendigvis er den mest effektive. Siden vi jobbet mye i lag, og det ble diskusjoner og mye prat, tok teksting og koding lenger tid enn det egentlig krevde, men vi satt stor verdi av å skape en enstemmig forståelse av prosjektet, og kvalitetssikre kode og tekst, slik at det ble så god kvalitet som mulig.

8 Konklusjon

Gjennom arbeidsprosessen av bacheloroppgaven har vi fått muligheten til å bruke mange av de kunnskapene og erfaringer som vi har fått igjennom utdanningen, innenfor problemløsning, design, database-systemer og programmering. Vi har også innhentet mye ny kunnskap, spesielt innen fagområdet IoT.

I løpet av arbeidsprosessen dukket det opp flere problemer, som vi måtte ta stilling til. Disse var ofte problemer som typisk oppstår ved større prosjekt, og vi hadde derfor lite erfaring med dette fra før, og måtte innhente kunnskap for å ta stilling til problemene.

Målet med prosjektet var å lage en prototype som skulle være et konseptbevis på et program som automatisk kunne loggføre veivedlikeholdsoppgaver gjort av nyttekjøretøy. Selv om prosjektet hadde problemer med innstallering av enhetene, har vi utviklet en løsning som vi mener er en velfungerende prototype for videre utvikling.

Vi skulle finne ut om lesing av CAN-data var en brukbar løsning. I løpet av prosjektet og problemene som oppstod, kan vi ikke konkludere om dette er en brukbar løsning, men kan konkludere med at det å få dette systemet til å fungere, ikke er en enkel oppgave, og som vil kreve mye tid og ressurser, men vi har troen på at dette er mulig, ut ifra våre erfaringer gjennom prosjektet.

9 Litteraturliste

[1] W. Bidragsytere, «Tingenes internett,» [Internett]. Available: https://no.wikipedia.org/wiki/Tingenes_internett. [Funnet 26 01 2021].

[2] W. Bidragsytere, «CAN bus,» [Internett]. Available: https://en.wikipedia.org/wiki/CAN_bus. [Funnet 20 01 2021].

[3] ETC, «Hjem - Caradmin,» [Internett]. Available: https://caradmin.no/. [Funnet 08 04 2021].

[4] GeeksforGeeks, «Cyclic Redundancy Check and Modulo-2 Division - GeeksforGeeks,» [Internett]. Available: https://en.wikipedia.org/wiki/Cyclic_redundancy_check. [Funnet 18 05 2021].

[5] Teltonika, «Codec,» [Internett]. Available: https://wiki.teltonika-gps.com/view/Codec. [Funnet 19 03 2021].

[6] CSS Electronics, «CAN Bus Explained - A Simple Intro (2021),» [Internett]. Available: https://www.csselectronics.com/screen/page/simple-intro-to-can-bus/language/en. [Funnet 24 03 2021].

[7] Atlassian, «What is Agile? | Atlassian,» [Internett]. Available: https://www.atlassian.com/agile. [Funnet 18 05 2021].

[8] C. Drumond, «Scrum - what it is, how it works, and why it's awesome,» [Internett]. Available: https://www.atlassian.com/agile/scrum. [Funnet 18 05 2021].

[9] D. Radigan, «Kanban - A brief introduction | Atlassian,» [Internett]. Available: https://www.atlassian.com/agile/kanban. [Funnet 17 05 2021].

[10] R. Lynn, «What is Scrumban? | Planview,» [Internett]. Available: https://www.planview.com/no/resources/guide/what-is-scrum/lkdc-what-is-scrumban/. [Funnet 18 05 2021].

[11] M. Skarin og H. Kniberg, Kanban and Scrum - Making the Most of Both, Lulu.com, 2010.

[12] JavaTpoint, «Normalization,» [Internett]. Available: https://www.javatpoint.com/dbms-normalization. [Funnet 09 04 2021].

[13] W. Bidragsytere, «Javadoc,» [Internett]. Available: https://en.wikipedia.org/wiki/Javadoc. [Funnet 5 5 2021].

[14] Datatilsynet, «GPS og sporing av yrkesbiler | Datatilsynet,» Datatilsynet, [Internett]. Available: https://www.datatilsynet.no/personvern-pa-ulike-omrader/personvern-pa-arbeidsplassen/overvaking-kjoretoy/. [Funnet 19 5 2021].

[15] Datatilsynet, «Personvernprinsippene | Datatilsynet,» Datatilsynet, [Internett]. Available: https://www.datatilsynet.no/rettigheter-og-plikter/personvernprinsippene/. [Funnet 19 5 2021].

[16] Telenor, «Slukker 3G for bedre M2M og IoT-tjenester på 4G - Telenor,» [Internett]. Available: https://www.telenor.no/bedrift/iot/m2m/3g/. [Funnet 30 04 2021].

[17] Teltonika, «GPS Trackers, Accessories, Solutions | Teltonika Telematics,» Teltonika, [Internett]. Available: https://teltonika-gps.com/. [Funnet 25 01 2021].

[18] CSS Electronics, «CAN Bus Data Loggers - Simple. Pro. Interoperable,» CSS Electronics, [Internett]. Available: https://www.csselectronics.com/. [Funnet 20 01 2021].

[19] I. Sommerville, Software Engineering, Global Edition, Pearson Education Limited, 2015.

[20] Microsoft, «Database normalization description - Office | Microsoft Docs,» Microsoft, [Internett]. Available: https://docs.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description. [Funnet 5 9 2021].

[21] H. Bothner-By, «protokoll (IT),» 12 01 2021. [Internett]. Available: https://snl.no/protokoll_-_IT. [Funnet 26 04 2021].

[22] MaridBD bidragsytere, «Events Overview - MariaDB Knowledge Base,» [Internett]. Available: https://mariadb.com/kb/en/events/. [Funnet 17 05 2021].

10 Vedlegg

A. Ord Beskrivelse

AVL-datapakke	En datapakke brukt av Teltonika, som kan inneholde data om hendelser fra kjøretøy.
CRC-16	CRC eller Cyclic redundancy check har som formål å sjekke om data som er mottatt er lik som da den ble sendt
Kodek	En Kodek er en algoritme som har som formål å fortelle hva ulike deler av en datastrøm, inneholder.
Enhetsfamilie	En enhetsfamilie refererer til en gruppe enheter som bruker en felles protokoll for tilkobling og tolking av data.
Handshake	Er pakker som enheten sender for å identifisere seg og få tilkobling til serveren.
TCP	TCP står for «Transmission Control Protocol» og er en nettverksprotokoll for dataoverføring mellom endepunkter.
IO Element	Et IO-element er en datastruktur som inneholder info om hendelser som har skjedd på kjøretøyet. F.eks. Plog nede.
IMEI	IMEI er et identifikasjonsnummer for mobil og kommunikasjonsutstyr som er globalt brukt. Dette er et tall med 15 siffer.
GDPR	Ellers kalt Personvernforordningen er en forordning som skal styrke og beskytte personvern for personer i EU og EØS.

B. Prosjekt Avtale

Vår dato Vår referanse

Norges teknisk-naturvitenskapelige universitet

Prosjektavtale

mellom NTNU Fakultet for informasjonsteknologi og elektroteknikk (IE) på Gjøvik (utdanningsinstitusjon), og

ETC, Electrictime Car AS

(oppdragsgiver), og

Benjamin Alexander Whittaker
Karl Andreas Wik Opheim
Simen Andrè Dahl Jensen
(student(er))

Avtalen angir avtalepartenes plikter vedrørende gjennomføring av prosjektet og rettigheter til anvendelse av de resultater som prosjektet frembringer:

1. Studenten(e) skal gjennomføre prosjektet i perioden fra **JAN 2021** til **JUN 2021** .

Studentene skal i denne perioden følge en oppsatt fremdriftsplan der NTNU IE på Gjøvik yter veiledning. Oppdragsgiver yter avtalt prosjektbistand til fastsatte tider. Oppdragsgiver stiller til rådighet kunnskap og materiale som er nødvendig for å få gjennomført prosjektet. Det forutsettes at de gitte problemstillinger det arbeides med er aktuelle og på et nivå tilpasset studentenes faglige kunnskaper. Oppdragsgiver plikter på forespørsel fra NTNU å gi en vurdering av prosjektet vederlagsfritt.

2. Kostnadene ved gjennomføringen av prosjektet dekkes på følgende måte:
   - Oppdragsgiver dekker selv gjennomføring av prosjektet når det gjelder f.eks. materiell, telefon, reiser og nødvendig overnatting på steder langt fra NTNU i Gjøvik. Studentene dekker utgifter for ferdigstillelse av prosjektmateriell.
   - Eiendomsretten til eventuell prototyp tilfaller den som har betalt komponenter og materiell mv. som er brukt til prototypen. Dersom det er nødvendig med større og/eller spesielle investeringer for å få gjennomført prosjektet, må det gjøres en egen avtale mellom partene om eventuell kostnadsfordeling og eiendomsrett.

3. NTNU IE på Gjøvik står ikke som garantist for at det oppdragsgiver har bestilt fungerer etter hensikten, ei heller at prosjektet blir fullført. Prosjektet må anses som en eksamensrelatert oppgave som blir bedømt av intern og ekstern sensor. Likevel er det en forpliktelse for utøverne av prosjektet å fullføre dette til avtalte spesifikasjoner, funksjonsnivå og tider.

4. Alle beståtte bacheloroppgaver som ikke er klausulert og hvor forfatteren(e) har gitt sitt samtykke til publisering, kan gjøres tilgjengelig via NTNUs institusjonelle arkiv NTNU Open.

Tilgjengeliggjøring i det åpne arkivet forutsetter avtale om delvis overdragelse av opphavsrett, se «avtale om publisering» (jfr Lov om opphavsrett). Oppdragsgiver og veileder godtar slik offentliggjøring når de signerer denne prosjektavtalen, og må evt. gi skriftlig melding til studenter og instituttleder/fagenhetsleder om de i løpet av prosjektet endrer syn på slik offentliggjøring.

Den totale besvarelsen med tegninger, modeller og apparatur så vel som programlisting, kildekode mv. som inngår som del av eller vedlegg til besvarelsen, kan vederlagsfritt benyttes til undervisnings- og forskningsformål. Besvarelsen, eller vedlegg til den, må ikke nyttes av NTNU til andre formål, og ikke overlates til utenforstående uten etter avtale med de øvrige parter i denne avtalen. Dette gjelder også firmaer hvor ansatte ved NTNU og/eller studenter har interesser.

5. Besvarelsens spesifikasjoner og resultat kan anvendes i oppdragsgivers egen virksomhet. Gjør studenten(e) i sin besvarelse, eller under arbeidet med den, en patentbar oppfinnelse, gjelder i forholdet mellom oppdragsgiver og student(er) bestemmelsene i Lov om retten til oppfinnelser av 17. april 1970, §§ 4-10.

6. Ut over den offentliggjøring som er nevnt i punkt 4 har studenten(e) ikke rett til å publisere sin besvarelse, det være seg helt eller delvis eller som del i annet arbeide, uten samtykke fra oppdragsgiver. Tilsvarende samtykke må foreligge i forholdet mellom student(er) og faglærer/veileder for det materialet som faglærer/veileder stiller til disposisjon.

7. Studenten(e) leverer oppgavebesvarelsen med vedlegg (pdf) i NTNUs elektroniske eksamenssystem. I tillegg leveres ett eksemplar til oppdragsgiver.

8. Denne avtalen utferdiges med ett eksemplar til hver av partene. På vegne av NTNU, IE er det instituttleder/faggruppeleder som godkjenner avtalen.

9. I det enkelte tilfelle kan det inngås egen avtale mellom oppdragsgiver, student(er) og NTNU som regulerer nærmere forhold vedrørende bl.a. eiendomsrett, videre bruk, konfidensialitet, kostnadsdekning og økonomisk utnyttelse av resultatene. Dersom oppdragsgiver og student(er) ønsker en videre eller ny avtale med oppdragsgiver, skjer dette uten NTNU som partner.

10. Når NTNU også opptrer som oppdragsgiver, trer NTNU inn i kontrakten både som utdanningsinstitusjon og som oppdragsgiver.

11. Eventuell uenighet vedrørende forståelse av denne avtale løses ved forhandlinger avtalepartene imellom. Dersom det ikke oppnås enighet, er partene enige om at tvisten løses av voldgift, etter bestemmelsene i tvistemålsloven av 13.8.1915 nr. 6, kapittel 32.

Norges teknisk-naturvitenskapelige
Fakultet for informasjonsteknologi og elektroteknikk

NTNUs veileder (navn): ______________________

Oppdragsgivers kontaktperson (navn): ______________________

tur): Karl Anton Nik Ekeland dato 21.01.21

Benjamin A Whittaker dato 26.01.21

Student(er) (signatur): Simen D Jensen dato 28.01.21

______________________ dato ________

Oppdragsgiver (signatur): Dag L Solberg dato 29.01.21

Signert avtale leveres digitalt i Blackboard, romfor bacheloroppgaven.
Godkjennes digitalt av instituttleder/faggruppeleder.

Om papirversjon med signatur er ønskelig, må papirversjon leveres til instituttet i tillegg. Plass for evt sign:

Instituttleder/faggruppeleder (signatur): ______________________ dato ________

C. Prosjektplan

Prosjektplan

Karl Andreas Wik Opheim, Benjamin A Whittaker, Simen Andrè Dahl Jensen

Januar 2020

Innholdsfortegnelse:

1 Mål og Rammer **2**
1.1 Bakgrunn 2
1.2 Prosjektmål 2
1.2.1 Effektmål 2
1.2.2 Resultatmål 3
1.2.3 Læringsmål 3
1.3 Rammer 4
1.3.1 Tid 4
1.3.2 Sensorer 4
1.3.3 Utvikling 4

2 Omfang **4**
2.1 Fagområder 4
2.2 Avgrensning 5
2.3 Oppgavebeskrivelse 5

3 Prosjektorganisering **6**
3.1 Ansvarsforhold og roller 6
3.2 Rutiner og regler i gruppa 7
3.2.1 Rutiner 7
3.2.2 Regler 7

4 Planlegging, oppfølging og rapportering **8**
4.1 Hovedinndeling av prosjektet 8
4.1.1 Valg av modell 8
4.1.2 Metode 9
4.2 Plan for statusmøter og beslutningspunker i perioden 9

5 Organisering av kvalitetssikring **10**
5.1 Dokumentasjon, standardverk og kildekode 10
5.1.1 Utvikling 10
5.1.2 Rapportskriving 10
5.2 Konfigurasjonsstyring: Git-Repository og regler 10
5.3 Risikoanalyse 11

6 Plan for gjennomføring **13**
6.1 Gantt-skjema for utviklingsperioden 13
6.2 Timeplan 13

Referanser **14**

1 Mål og Rammer

1.1 Bakgrunn

Internet of Things (IoT) er et nettverk av gjenstander, der gjenstandene er utstyrt med sensorer, programvarer og andre teknologier som gjør de i stand til å kommunisere med hverandre og utveksle data [1].

Controller Area Network (CAN) er en protokoll designet for å la mikrokontrollere kommunisere gjennom en felles bus uten en sentral datamaskin. CAN ble i utgangspunktet utviklet for bruk i kjøretøy, men har siden spredt seg videre til automasjon[2].

Electric Time Car AS (ETC) er et IT-selskap som leverer totalløsninger for helhetlig kjøretøy oppfølgning kalt CarAdmin[1]. CarAdmin tilbyr flere moduler og funksjonalitet for enkel drift av organisasjonens kjøretøy, alt fra nøkkelhåndtering til flåtestyring.

I denne bacheloroppgaven skal vi utvikle backend-løsningen til veivedlikeholds modulen i CarAdmin. Vi skal innhente data fra kjøretøy, gjennom bruk av CAN, som skal tolkes og lagres i CarAdmins systemer for videre utnyttelse.

1.2 Prosjektmål

1.2.1 Effektmål

- Kunne loggføre sjåførens aktivitet på kjøretøyet under arbeidsdriften.
- Overvåke nyttekjøretøys status og vedlikehold i sanntid og loggføre dette.
- Levere et flåtestyrke-verktøy for nyttekjøretøy i både privat og offentlig sektor.
- Gjør veivedlikeholds-data tilgjengelig for offentligheten, gjennom et publikumskart.

1.2.2 Resultatmål

- Demonstrere hvordan data fra kjøretøys CAN bus system kan formateres til et lesbart format og videre sendes til ETCs database til videre bruk.
- Levere et system som kan fungere eller være et utgangspunkt for en backend løsning for nyttekjøretøy modulen til CarAdmin.

[1] https://caradmin.no/

1.2.3 Læringsmål

- Erfare å lage et system som består av flere ulike teknologier, deriblant sensor, internett protokoller og database.
- Erfare å jobbe med et større prosjekt hvor man bruker moderne programvare utvikling arbeidsmetodikk sammen med en “klient”.
- Få kjennskap til teltonika produktene vi bruker og hvordan utnytte deres egenskaper.
- Forstå moderne kjøretøys CAN system, dens ulike layer standarder(SAE J1939) og protokoller.
- Utenom dette gjelder læringsmålene i emnebeskrivelsen for bacheloroppgaven

1.3 Rammer

1.3.1 Tid

NTNU har satt fristen på levering av rapporten til 20. mai. Innen den tid må kode, dokumenter og andre krav være oppfylt og ferdiggjort.

1.3.2 Sensorer

I starten av prosjektet fikk vi i oppgave å finne passende sensorer som skal installeres i diverse kjøretøy for å samle data og sende dette til serveren. Sensorene må bestilles, bli levert, og bli installert i passende kjøretøy, før vi kan teste og utvikle store deler av prosjektet.

1.3.3 Utvikling

Løsningen skal utvikles i Java, siden serveren til ETC er programmert i dette, og det skal benyttes SQL (MariaDB) for loggføring av data fra sensor.

2 Omfang

2.1 Fagområder

Flåtestyring omfatter flere funksjoner, f.eks. overvåkning, tilrettelegging for transportrelaterte oppgaver og vedlikehold av kjøretøy. Gjennom disse funksjonene effektiviserer flåtestyring produktivitet og reduserer de samlede kostnadene til aktiviteter som er avhengig av transport. Effektiv flåtestyring har som mål å redusere kostnader gjennom kostnadseffektiv utnyttelse av ressurser som biler, drivstoff, reservedeler osv. Flåtestyring forbedrer også kundeservice

og kundetilfredshet ved å redusere ventetid og leveransetid. Kunden får også tilgang til mer informasjon om leveranse status gjennom varsler og påminnelser. Fra lesing av CAN data kan det måles drivstoff-forbruk, advare om kjøretøyets status osv. Med veksten av IoT har innhenting av data og kommunikasjon mellom kjøretøy blitt lettere, som skaper nye muligheter innenfor flåtestyring.

2.2 Avgrensning

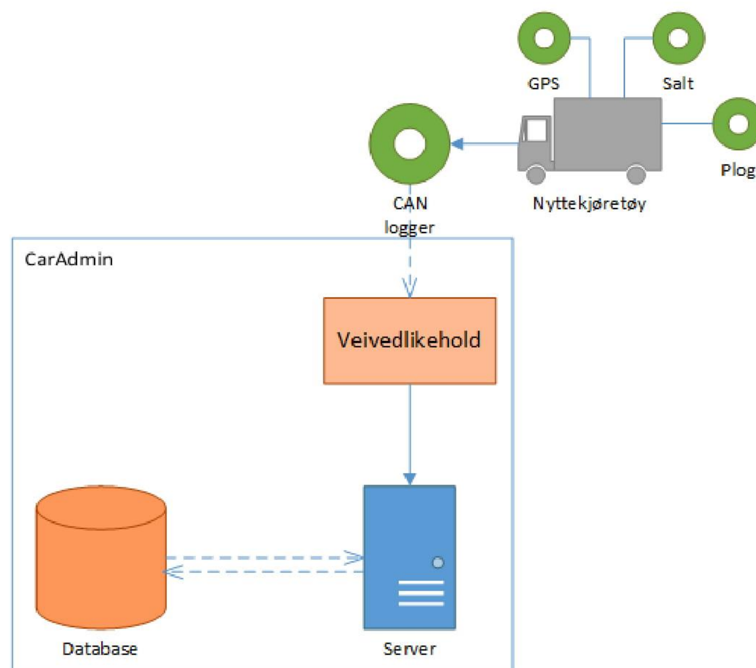
Oppgaven går ut på å utvikle et system for flere kjøretøy som skal sende en rekke data til serveren. Vi vil fokusere mest på kjøretøyene som driver med salting og ploging. Vi skal også kun fokusere på å tolke og lagre dataen i back-end, ikke front-end. På nåværende punkt i prosjektet er det vanskelig å si om det vil komme flere avgrensninger.

2.3 Oppgavebeskrivelse

Oppgavens omfang er dynamisk i størrelse og kan gi muligheter til utvidelser hvis prosjektperioden tillater det. Prosjektbeskrivelsen ønsker at vi skal levere og/eller vise til et "proof of consept" av et system som leser av data fra nyttekjøretøy og sende dette til ETCs databaser. Dette skal fungere som en backend til en modul i CarAdmin løsningen til ETC.

Basert på første utkast av produktbeskrivelse fra ETC, så skal vi velge ut sensor-teknologien og sensor-modeller som oppfyller kravene. Sensorene vi velger skal bestå av en CAN BUS sensor som jobber med en lednings sensor som kan lese data gjennom ledningene uten å koble seg inn i bilens ledningsnett. Disse skal være seriekoblet med en GPS sensor som kan sende dens AVL datapakker over 4G nettet i TCP protocol. Disse må så mottas av (server?) hvor vårt program tolker og omgjør pakkene til menneskelig leselig data.

Ved å gå gjennom dokumentasjonen til en av CAN BUS leserne så kan vi se at utvalget av datasorter som kan logges ut av bilen er godt over hundre ulike typer, så må finne de som er mest interessante for arbeidsgiver. Etter samtaler med ETC så er data som angår salting, måking og vedlikehold av bil som er høyest på prioriteringslisten.



Figur 1: Skisse av systemet

3 Prosjektorganisering

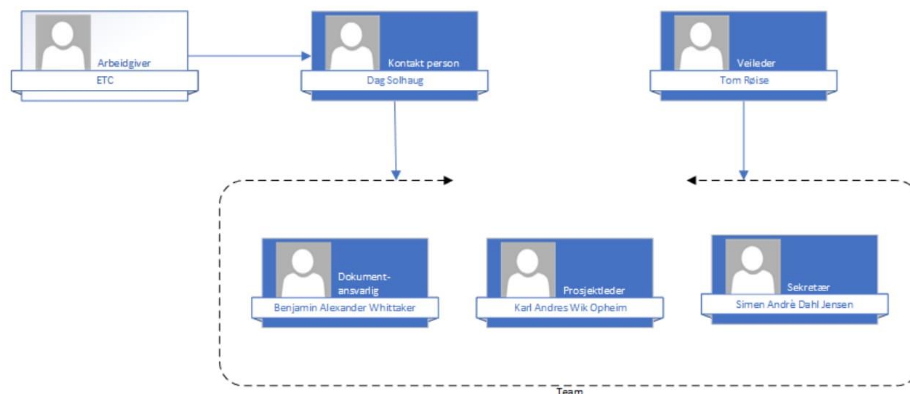
3.1 Ansvarsforhold og roller

Karl Andreas ble utpekt som prosjektleder. Vi ble enige om at prosjektlederen har hovedansvar for at alle i gruppa følger gruppe reglene, passer på at det jobbes målrettet frem mot leveringsfrist og at gruppa følger planen som er lagt.

Benjamin har hovedansvaret for dokumentasjon. Han skal passe på at alle følger kode konvensjonen, bruker Git og Google Disk verktøyene korrekt og at rapporten blir skrevet korrekt og i samsvar med planlagt oppsett.

Simen har hovedansvar for møteinnkalling med veileder og arbeidsgiver, skrive referat om det er noen som ikke kan møte på avtalt tidspunkt, og har ansvaret for å kontakte tredjeparts bedrifter om dette viser seg å være nødvendig.

Selv om hver i gruppa har fått en rolle, så betyr ikke dette at de har ansvar alene om oppgavene sine. De har kun hovedansvar for sin oppgave, men alle de andre kan komme med forslag, tipse, varsle og passe på at alle gjør sine oppgaver.



Figur 2: Organisasjonskart

3.2 Rutiner og regler i gruppa

3.2.1 Rutiner

Det er et stort arbeid som skal gjøres, og vi har blitt enige om at vi bør jobbe minst 30 timer i uken. Dette blir da 6 timer hver dag i hverdagene. En vanlig jobbdag blir da fra 10.15 til 16.15.

Hver fredag legger vi en plan for den neste uka. Møtetider, hva som skal bli gjort, hvordan vi ligger an. Mer om dette i kapittel 4 av prosjektplanen.

3.2.2 Regler

Gruppen satt seg i sammen og kom fram til disse 10 reglene, som skal i sammen med rutinene, gjelde fra prosjektplanen er levert.

1. *Alle møter på avtalt tid, og gir beskjed i god tid hvis det ikke passer.*

2. *Hvis man ikke kan delta på et møte, skriver referatansvarlig et referat av møte og hva som ble gått igjennom, som blir sendt til den som ikke kan delta. Hvis ikke referatansvarlig er tilgjengelig, må en av de andre skrive.*
3. *Alle skal være hjelpsomme og imøtekommende.*
4. *Er det uenigheter, så har vi en formell diskusjon, hvor vi til slutt stemmer over problemet. Hvis ikke dette fungerer, kontakter vi veileder for tips.*
5. *Alle engasjerer seg, og kommer med positive innspill.*
6. *Alle bruker timeplanen godt, og fører inn hver dag man jobber med oppgaven.*
7. *Alle setter seg inn i og benytte verktøyene vi velger, på en ordentlig måte.*
8. *Det forventes at det jobbes minst 30 timer i uken per deltaker. Ligger man bak dette forventes det at man jobber mer for å ta igjen timene som er mistet.*
9. *Være åpen for at arbeidet sitt, blir utfordret konstruktivt av andre i gruppa.*
10. *Ha det gøy!*

4 Planlegging, oppfølging og rapportering

4.1 Hovedinndeling av prosjektet

4.1.1 Valg av modell

Etter å ha snakket med arbeidsgiver og veileder, viste det seg gunstig å velge en smidig systemutviklingsmodell. Etter å ha diskutert, gjort research og snakket med veileder, fant vi ut at vi vil bruke scrum eller kanban. Her er noen fordeler og ulemper mellom dem:

Tabell 1: Scrum vs. Kanban

Scrum	**Kanban**
+ Hyppige møter	+ Utviklere velger oppgaver som passer fra kanban-tavla
+ Lett å se hvor man er i prosjektet	+ Ser visuelt arbeidsflyten på kanban-tavla
- Kan bli for mye når gruppen er liten	- Ikke så organisert
- Drukning i møter	- Vanskelig å se hvor langt man er i prosjektet

Scrum har den gode egenskapen, at det gjør det lettere å følge planen og ikke grave seg ned i en grop, ettersom man har flere milepæler og møter. Likevel kan det bli for mye møter og organiseringsarbeid som bruker tid og ressurser, spesielt for små grupper.

Kanban derimot, gjør det lettere å ta tak i oppgaver når det passer, ved hjelp av kanban-tavla, der utviklerene i mellom seg bestemmer hvilke oppgaver som skal gjøres til en hver tid. Likevel er det lett å miste helhetsbilde, og grave seg ned i en grop.

Vi vil ha det administrative i fra scrum og det effektive med kanban, og velger da å bruke Scrumban som vår modell. Scrumban er en smidig modell som kombinerer forutsigbarheten fra Scrum med fleksibiliteten til Kanban.

4.1.2 Metode

Vi skal ha en kanban-tavle. Vi har bestemt at den skal ha fasene:

Backlog	ToDo (max 10)	Developing (max 4)	ToTest (max 4)	Testing (max 2)	Done
Issues som skal implementeres	Issues vi skal jobbe med i nåværende periode	Issues vi har startet å implementere	Issues som er klare for testing	Issues vi har startet å teste	Ferdige issues

På hver kolonne i tavla, setter vi en max-grense på hvor mange oppgaver det kan være i hver av dem. Dette betyr du ikke kan flytte en issue til neste kolonne før det er ledig plass i denne. På denne måten vil det ikke bygge seg opp en haug med uferdige issues, men vi blir tvunget til å fullføre issues før vi begynner på nye.

4.2 Plan for statusmøter og beslutningspunker i perioden

4.2.1 Veileder Møte

Gruppen møter med veileder Tom Røise ukentlig der prosjektets status og retning blir diskutert. Her vil vi kunne få rådgivning til hvordan vi innfører pensum relaterte elementer inn i rapporten og hvordan unngå eventuelle arbeids fallgroper ved innspill fra hans erfaringer.

4.2.2 Oppdragsgiver Møte

Etter planlagt timeplan så møter vi med Dag Solhaug fra ETC ukentlig, hvor vi kan diskutere oppgavens mål, ambisjoner og status. Her kan vi også få veiledning på gjennomføring av oppgaven gjennom Dag og ETCs mange år med bacheloroppgave erfaringer. Vi møter også med en av utviklerne til ETC (Øyvind), hvor vi får en mer programmerings orientert veiledning og rådgivning for å jobbe mot CarAdmin løsningen.

5 Organisering av kvalitetssikring

5.1 Dokumentasjon, standardverk og kildekode

5.1.1 Utvikling

For å sikre lett leselighet og høy kvalitet på koden skal vi følge kode konvensjonene definert i Google Java Style Guide[2]. All kode skal dokumenters ved bruk av JavaDocs, unntak er selvforklarende kode, for å sikre lett tilgjengelig informasjon om klasser og funksjoner. Dokumentasjonen skal skrive på norsk, men variabler, metoder og klasser skal skrive på engelsk. Det er viktig at koden blir testet for å oppdage svakheter og eventuelle feil, derfor skal all kode testes før den blir sendt til release.

5.1.2 Rapportskriving

Alle kritiske dokumenter og filer relatert til rapporten skal lagres i en delt mappe i Google Drive for å forsikre oss mot tap av dokumenter ved eventuelle uhell. Google Drive er enkelt å bruke og lett tilgjengelig, og det er mulig å få tilgang til tapte filer hvis noe skulle skje gjennom Google Drives Skylagring.

5.2 Konfigurasjonsstyring: Git-Repository og regler

Gjennom prosjektet skal vi bruke Git-løsningen som ETC bruker. Her skal vi sette inn kanban-tavla vår, som vi skal forsøke å koble mot en issue tracker. Hyppig bruk av branches vil være nyttig for å jobbe individuelt på en funksjonalitet.

Vi har skrevet ned noen regler rundt bruk av git:

1. *Hyppige commits. Ikke la det bygges opp mye arbeid før du pusher til repo, for å unngå conflicts og se utviklingsprosessen.*

[2] https://google.github.io/styleguide/javaguide.html

2. *Korte beskrivende titler, og kort beskrivelse av hva som er gjort.*
3. *Skriv på norsk.*
4. *Koble commits mot issues og kanban.*
5. *Snakke sammen, før merging av branches for feilhåndtering.*

5.3 Risikoanalyse

Vi setter opp en risiko-tabell med uønskede hendelser som kan inntreffe i løpet av prosjektperioden. Sannsynligheter går fra 1-5 (lite til meget sannsynlig) og konsekvenser fra 1-5 (ufarlig til katastrofal konsekvens). Vi ønsker å tallfeste både når en hendelse har en høy sannsynlighet og/eller en høy konsekvens, og regner derfor ut risikonivået slik:

$$Risikonivå = Sannsynlighet * Konsekvens$$

Vi bruker en risikomatrise hentet fra UiB[3] som beskriver hvilket risikonivå som gir lav, middels og høy risiko.

Tabell 3: Risiko-Tabell

Beskrivelse	Sannsynlighet	Konsekvens	Risikonivå	Kommentar
Deltakere blir syke i kort tid	2	1	2 * 1 = 2	Lite sannsynlig, med liten konsekvens
Sensorer bruker lang tid på å bli levert	2	2	2 * 2 = 4	ETC har sensorer vi kan teste på mens vi venter
Ikke nok kompetanse i gruppen	3	2	3 * 2 = 6	Gruppen er ikke eksperter på området
Tap av dokumenter og filer	1	4	1 * 4 = 4	Lagret i skyen, derfor lite sannsynlig
Sensorer fungerer ikke som planlagt	2	4	2 * 4 = 8	Må kjøpe nye sensorer. Tid og penger går tapt
Deltakere blir langtidssyke	1	5	1 * 5 = 5	Tap av tid. Omfordeling av oppgaver
Prosjektet blir ikke ferdig innen gitt dato	3	5	3 * 5 = 15	Stor konsekvens
Problemer med installering av sensor	2	5	2 * 5 = 10	Garanti utgår. Personskader

[3] https://www.uib.no/hms-portalen/137767/risikomatrise#eksempel

Etter å ha skrevet opp noen uønskede hendelser, ser vi det nødvendig og lage en tabell som viser hvordan vi kan håndtere de viktigste risikoene med diverse sikkerhetstiltak.

Tabell 4: Plan for håndtering av risiko

Beskrivelse	**Sikkerhetstiltak**
Ikke nok kompetanse i gruppen **Risikonivå: 6**	Denne oppgaven tar for seg teknologi og programvare som gruppa ikke har erfaring eller kunnskap med fra før. Det vil derfor være en sjanse for at det blir mangel på kompetanse i gruppen. Et tiltak er at vi gjennom utviklingen vil være i kontakt med en erfaren utvikler fra ETC, som vil hjelpe der kompetansen vår ikke strekker seg til.
Tap av dokumenter og filer **Risikonivå: 4**	Dokumenter lagres i skyen (Google Drive), og filer til koden vil bli lagret i Git, så sannsynligheten for at det blir tap av filer er lav, men konsekvensen er veldig høy, ettersom tap av flere dager med jobbing vil gi oss mye ekstra arbeid. Google har allerede backup løsninger for Google Drive, og Git ligger lokalt på hver PC. Utenom dette, er det viktig å følge gode Git-vaner, som ikke forårsaker feil og mangler i prosjekt-koden.
Sensorer fungerer ikke som planlagt **Risikonivå: 8**	Mye kan gå galt når vi skal bruke sensorene. Det kan forekomme installasjonsfeil som gjør sensorene defekte, de være trege og ikke sende dataen vi ønsker raskt nok, og ikke sende dataen vi ønsker. Konsekvensene er store, siden det må bestilles nye sensorer som både tar tid og penger. Det er mange leverandører av sensorer og disse har ulike protokoller for dataen. Hvis vi i midten av prosjektet finner ut at vi bytte leverandør, kan det være programmet ikke er kompatibelt. Derfor brukte vi mye tid på å lese på sensorene og velge en som tilfredsstiller alle krav. ETC har også noen sensorer fra før, som vi kan teste på, om det må bestilles nye. Vi vil også utvikle programmet med baktanke at protokoller kan endres.
Deltakere blir langtidssyke **Risikonivå: 5**	Det er usannsynlig at denne hendelsen inntreffer, men skulle den inntreffe vil det ha store konsekvenser. Oppgaver må da tildeles på nytt og det må brukes mer tid av de andre deltakerne. Det er derfor viktig at vi ikke lar noen ta fullt ansvar for en del av koden. Alle må få god kompetanse på alle områder, slik at vi kan ta over for den som eventuelt måtte få et illebefinnende. Blir det så alvorlig at deltakeren ikke kan delta resten av perioden, tar vi kontakt med NTNU, for rådgivning, og eventuelt korte ned oppgavens omfang.
Prosjektet blir ikke ferdig innen gitt dato **Risikonivå: 15**	Definitivt den største risikoen i dette prosjektet. Blir vi ikke ferdig med prosjektet/rapporten innen leveringsfrist, står vi uten bachelor og må ta studiet på nytt. Derfor er det lurt å ha gode milepæler slik

	at vi vet hvor vi ligger an i prosjektet. Ukentlige møter er også et tiltak for å passe på at vi jobber framover og ikke ned i en grop.
Problemer med installering av sensor **Risikonivå: 10**	Det må bestilles inn personer med kompetanse, for å installere sensorene i kjøretøyene. Klipping av ledninger kan ødelegge garantien på kjøretøyet, samtidig som det kan forekomme personskader. Derfor er det viktig at personene som skal gjøre jobben har god kompetanse, og vi har i tillegg sett på løsninger for installering av sensor uten klipping av ledninger.

6 Plan for gjennomføring

6.1 Gantt-skjema for utviklingsperioden

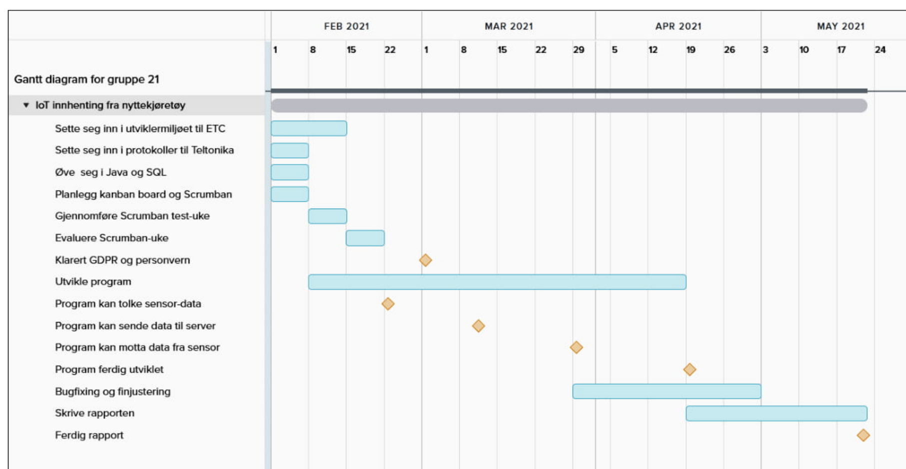


Fig 4: Gantt-skjema

Gantt-skjemaet beskriver hele utviklingsprosessen. De første 2 ukene skal brukes til å sette oss inn i ETC sitt utviklermiljø og bli flinke på Java og SQL. Vi skal også planlegge gjennomføring av Scrumban og lage en kanban-board. I uke 3 evaluerer vi hvorvidt dette fungerte og prøver å se forbedringspotensialer med modellen. I mars og april er det utvikling som er fokuset med diverse milepæler. Midt i april håper vi å ha et ferdig produkt. men har lagt inn 14 buffer dager for videre testing og feilretting. Disse datoene er vanskelig å forutse, og blir nok endret i løpet av utviklingsperioden, men er en skisse på hvordan det vil se ut.

6.2 Timeplan

Vi lagde et excel-regneark i Google Drive. Her legger vi inn hva vi har gjort på ulike dager i prosjektet. Regnearket regner ut totale timer, og vi kan sortere etter kategori av oppgaver.

Dato	Karl Andreas W.O	Start	Slutt	Totalt	Kommentar	Benjamin A.W	Start	Slutt	Total	Kommentar	Simen Jensen	Start	Slutt	Total	Kommentar
12. Jan 2021	Møte	11:00	12:00	01:00	Gruppe	Møte	11:00	12:00	01:00	Gruppe	Møte	11:00	12:00	01:00	Gruppe
13. Jan 2021				00:00					00:00					00:00	
14. Jan 2021	Møte	12:30	13:30	01:00	Gruppe	Møte	12:30	13:30	01:00	Gruppe	Møte	12:30	13:30	01:00	Gruppe
15. Jan 2021	Møte	10:30	12:00	01:30	CarAdmin	Møte	10:30	12:00	01:30	CarAdmin	Møte	10:30	12:00	01:30	CarAdmin
16. Jan 2021				00:00		Research	15:00	15:30	00:30	Lese om CAN				00:00	
17. Jan 2021	Research	17:15	17:45	00:30					00:00		Research	17:00	17:30	00:30	Lese opp på CAN
18. Jan 2021	Møte	11:00	13:00	02:00	Gruppe	Møte	11:00	13:00	02:00	Gruppe	Møte	11:00	13:00	02:00	Gruppe
19. Jan 2021	Møte	13:00	14:30	01:30	CarAdmin, Veileder	Møte	13:00	14:30	01:30	CarAdmin, Veileder	Møte	13:00	14:30	01:30	CarAdmin, Veileder
20. Jan 2021	Møte	10:00	12:00	02:00	Mail til vegvesenet	Møte	10:00	12:00	02:00	Mail til Teltonika	Møte	10:00	12:00	02:00	
21. Jan 2021	Møte	10:00	11:30	01:30	Sensor-research	Møte	10:00	11:30	01:30	Sensor-research	Møte	10:00	11:30	01:30	Sensor-research

Fig 5: Timeplan i ExCel

Referanser

[1] Wikipedia bidragsytere. Tingenes internett - Wikipedia, den frie encyklopedi. https://no.wikipedia.org/wiki/Tingenes_internett

[2] Wikipedia bidragsytere. CAN bus - Wikipedia, den frie encyklopedi. https://en.wikipedia.org/wiki/CAN_bus

D. Prosjektbeskrivelse

Oppdragsgiver

Oppdragsgiver: ETC, Electric Time Car AS
Kontaktperson: Dag L Solhaug
Adresse: Studieveien 2, 2815 Gjøvik
Telefon: +47 901 01 344
Epost: dag.solhaug@electrictimecar.com

IOT-datainnhenting fra kjøretøy

ETC er et lokalt innovativt IT-selskap som leverer nyskapende løsninger for helhetlig kjøretøyoppfølging, kalt CarAdmin. Nå samarbeider vi med NTNU ifm Smarte vinterveier for å vinne en Innovasjonskontrakt med Mjøsbyene.

Innovasjonspartnerskapet Smart vinterveg er et prosjekt hvor vi finner fremtidens løsninger for vinterdrift av veger og eiendommer. https://www.smartemjosbyer.no/

Oppgaven

Finne den beste måten å automatisk kunne loggføre hvilke veivedlikeholdsoppgaver en sjåfør gjør mens han utfører veivedlikehold. Eks når brøytes det, nå saltes det, hvor mye salt er forbrukt osv.

Utfordringen ligger i at det er varierende typer kjøretøy (lastebil, hjullaster, traktor…) som benyttes ift, alder, merke og type. Hvert kjøretøy er også utstyr med ulike typer utstyr, som saltspreder, strøkasser, plog, fres osv.

Det finnes ulike enheter for å avles Can buss i kjøretøyene. Det finnes også egne plattformer/leverandører som er rettet inn mot å levere slike data via API. I tillegg ligger også muligheten for egen sensor til formålet. Finnes det en enhet som leser alle kjøretøy, må det brukes egne sensorer. Eventuelt hvilke miks av enheter og løsninger anbefales? Og hvilke begrensinger finnes?

Det ønskes også at dataprotokollen implementeres i mottaksserver hos leverandør hvor mottatt data også tolkes og oversettes for videre utnyttelse i vårt system.

Avgrensing av oppgavene gjøres sammen med oppdragsgiver ift gruppens antall og kompetanse.

Oppgaven passer best for en gruppe på 3 - 4 personer.

ETC har gjennomført bacheloroppgave ved HiG/NTNU siden 2004 hvor flere av oppgaven har fått topp karakter. Alle bacheloroppgaver har resultert i ansettelser i ETC, eller ansettelser som følge av referanse fra ETC etter endt bacheloroppgave.

Vi er på utkikk etter å ansette Java programmerere, lokal tilhørighet vektlegges.

E. Use Case

Use case	Koble til server
Aktør	Enhet
Formål	Koble seg til serveren for å sende data fra kjøretøy til server
Beskrivelse	Enheten kobler seg opp til serveren gjennom enhetsfamilien sitt riktige portnummer, og initierer en «Handshake» med serveren. Serveren responderer med å godkjenne eller ikke godkjenne enheten.

Use case	Sende data
Pre-betingelse	Enheten har koblet seg til serveren og gjennomført godkjent «Handshake»
Aktør	Enhet
Formål	Hente data fra CANbus-systemet og sende til serveren
Beskrivelse	Enheten leser data fra kjøretøyet sitt CANbus-system og sender dette til serveren for videre tolkning, formatering og lagring.

Use case	Kartlegge en sjåførs arbeidsdag
Aktør	Frontend Utvikler
Formål	Visualisere hvordan en sjåfør har arbeidet en spesifikk dag
Beskrivelse	En frontend utvikler, bruker innhentet data fra databasen, til å visualisere arbeidsruter. Ved å bruke en kombinasjon av data, som f.eks. GPS-data, tidsstempel og «Har plog vært nede»-data, kan utvikleren danne grafiske bilder av hvor det er brøytet på et publikumskart.

Use case	Lagre i database
Aktør	Enhet
Formål	Skrive data hentet fra kjøretøy til databasen
Beskrivelse	Serveren bruker «Prepared Statements» for et element som skal skrives til databasen.

Use case	Hente Identifikasjonsnummer
Aktør	Enhet
Formål	Verifisere identifikasjonsnummeret til en enhet for å vite om den finnes i systemet eller ei
Beskrivelse	Henter identifikasjonsnummeret til enhetene og sjekker om enhet med dette nummeret eksisterer i databasen til systemet

Use case	Kartlegge måkeruter
Aktør	Frontend Utvikler
Formål	Visualisere hvor kjøretøy har måket snø
Beskrivelse	En frontend utvikler bruker innhentet data fra et eller flere kjøretøy, som er lagret i databasen. Visualiserer måkeruter ved å bruke data som kjøretøy id, sensor id, gps data, plog nedetid og tidsstempel.

Use case	Kartlegge salteruter
Aktør	Frontend Utvikler
Formål	Visualisere hvor kjøretøy har saltet
Beskrivelse	En frontend utvikler bruker innhentet data fra et eller flere kjøretøy, som er lagret i databasen. Visualiserer salteruter ved å bruke data som kjøretøy id, sensor id, gps data, saltspreder på, og tidsstempel.

Use case	Utvide enhetsbibliotek
Aktør	Backend Utvikler
Formål	Legge til protokoller for ulike sensorer
Beskrivelse	Skal det legges inn en enhet fra en ny enhetsfamilie, trengs det en ny protokoll til denne enheten. Dette gjøres ved å implementere «Protokoll»-objektet i koden og legge til de nye funksjonene for å lese dataen.

Use case	Modifisere database
Aktør	Backend Utvikler
Formål	Endre tabeller og legge til kategorier
Beskrivelse	En utvikler kan endre elementer i databasen. Legge til kjøretøy, kjøretøytyper, sensorer, produsenter og enhetsfamilier.

F. Referat til første møte med Arbeidsgiver 15.01.21

Dato: 15.01.21

Tid: kl 10.30

Deltagere: Dag, Karl Andreas, Benjamin, Simen

Spørsmål vi hadde til arbeidsgiver:

1. Skal det videreutvikles en modul i CarAdmin? Hva finnes fra før?
2. Hvilket verktøy (API, Språk, IDE, Git-løsning, Server-tools, Sikkerhet)
3. Installasjon av sensor, testing (program til sensor)
4. Hvor skal dataen loggføres? På en server? Skal det være Admin-verktøy? En nettside?
5. Skal sensorene bestilles, hva kan utvikles mens vi venter på denne?
6. Hva slags type sensorer skal brukes, og hvilke kjøretøy? Use cases?
7. Skal dataen bli mottatt på en av ETC sine servere eller på en kunde sin server?
8. Skal vi visualisere data, eller kun loggføre data til videre bruk?
9. Hva tenker dere er avgrensning på oppgaven?
10. Har dere tanker om utviklingsmodellen? Hvilke praksis og erfaringer har dere?
11. Hvor ofte ønsker dere å ha møter?
12. Hvem skal bruke løsningen? Hva er målgruppen?

Utdrag fra møte:

Viktig å ha sensor på plass så fort som mulig, kan ta tid å bestille. Vi skal skrive programmet i Java og bruke IntelliJ som utviklingsverktøy. Databasen er av typen MariaDB. Vi skal også bruke Git. Mens vi venter på sensor, kan vi planlegge prosjektet og gjøre research på sensorer. Vi skal jobbe ut ifra det som lese fra can bus systemet på et kjøretøy, hva kan vi lese, tolke og lagre av denne dataen? Vil ha ukentlige statusmøter, som ikke trenger å være lange, og må passe på at vi ikke står fast. Møte hver tirsdag kl 08.30. Vi kaller inn til møte med evt. saksliste. Når det gjelder utviklingsmodeller så bruker ETC agile metoder.

G. Utdrag fra dokumentasjon generert av JavaDocs

OVERVIEW PACKAGE CLASS TREE DEPRECATED INDEX HELP

SUMMARY: NESTED | FIELD | CONSTR | METHOD   DETAIL: FIELD | CONSTR | METHOD

SEARCH: Search

Package networking

Class Client

java.lang.Object
    networking.Client

All Implemented Interfaces:

`java.lang.Runnable`

Direct Known Subclasses:

`TeltonikaFmcClient`

```
public abstract class Client
extends java.lang.Object
implements java.lang.Runnable
```

Abstarkt klient klasse som kjøres på egen tråd. Sender og mottar data fra sensor. Sørger for at komunikasjon med enhet følger forventet format

Field Summary

Fields

Modifier and Type	Field	Description
protected Server	handler	
protected java.io.DataInputStream	input	
protected java.io.DataOutputStream	output	
protected java.net.Socket	socket	

Constructor Summary

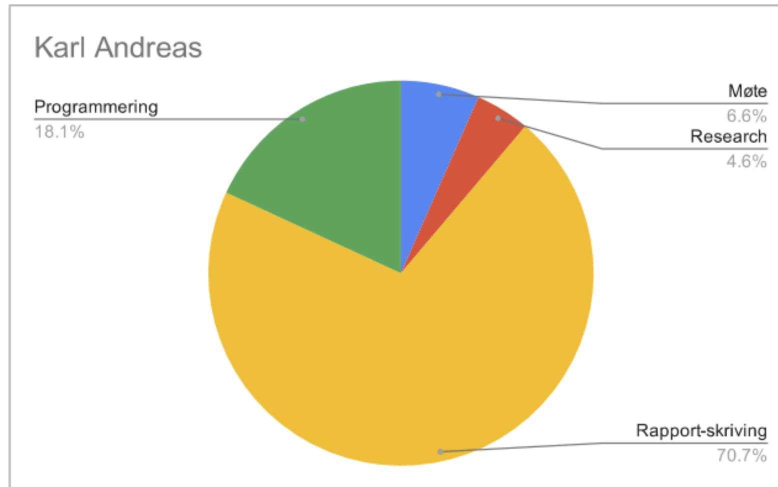
Constructors

Modifier	Constructor	Description
protected	Client(Server handler, java.net.Socket socket, java.io.DataInputStream input, java.io.DataOutputStream output, int bufferSize)	Konstruktøren til klienten.

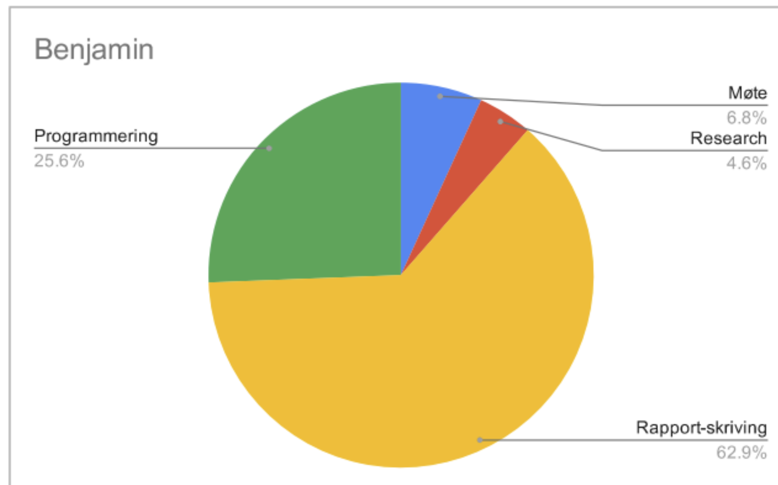
Method Summary

All Methods | Instance Methods | Abstract Methods | Concrete Methods

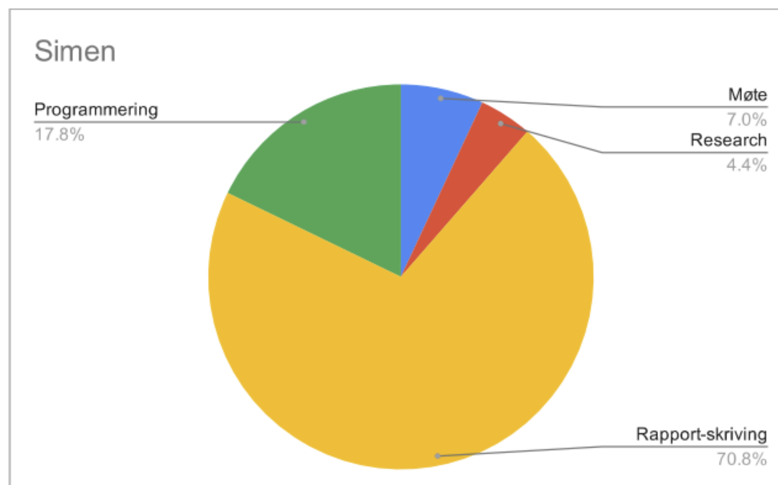
H. Tidsbruk



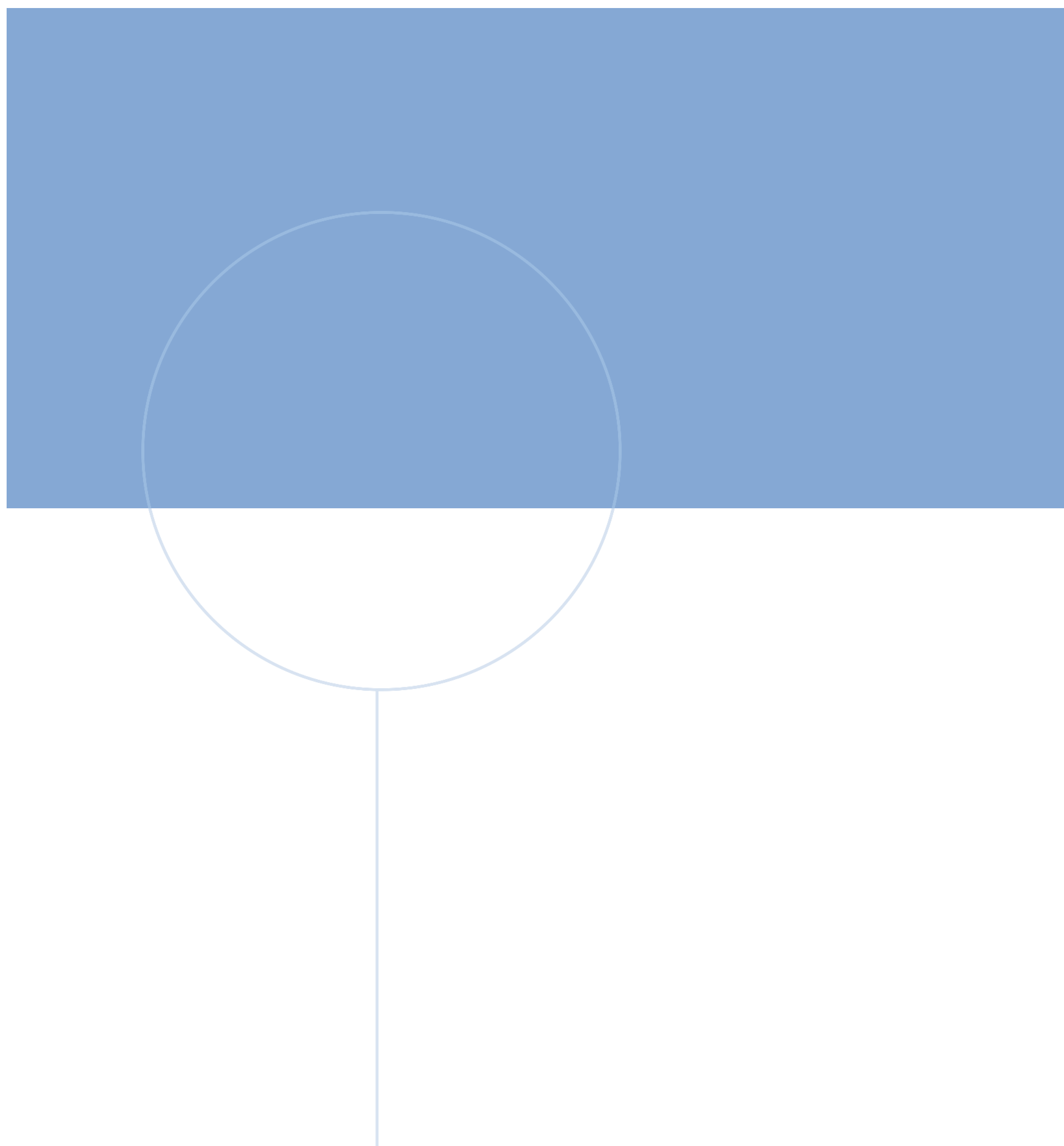
Totall: 382 Timer



Totall: 391 Timer



Totall: 388 Timer



NTNU
Kunnskap for en bedre verden