NTNU
Institutt for informatikk og e-læring

Olav Alexander Mjelde

Kontekst

Norsk
Utvikling av en kontekstuell Android applikasjon som gjennom egenutviklet tjenestebuss i PHP med MySQL geospatiale spørringer finner riktige datakilder og mellomlagrer det i Redis på en Virtuell Privat Server som kjører LEMP-stack med A+ HTTPS sertifisering.

English
The development of a contextual Android application that through a custom service bus in PHP with MySQL geospatial queries finds the right data sources and cache it in Redis on the Virtual Private Server which is running LEMP stack with A+ HTTPS certification.

Bacheloroppgave, Bachelor i informatikk
Bergen, mai 2016

Spesialiseringsretning: Informasjonsbehandling
Veileder: Mildrid Ljosland
Samarbeidsbedrift: Mjelde IKT

NTNU
Institutt for informatikk og e-læring

Bacheloroppgave *2016*

Studium: *Informatikk, Informasjonsbehandling*

Tittel – Kontekst - Utvikling av en kontekstuell Android applikasjon som gjennom egenutviklet tjenestebuss i PHP med MySQL geospatiale spørringer finner riktige datakilder og mellomlagrer det i Redis på en Virtuell Privat Server som kjører LEMP-stack med A+ HTTPS sertifisering. Tittel - engelsk: Kontekst - The development of a contextual Android application that through a custom service bus in PHP with MySQL geospatial queries finds the right data sources and cache it in Redis on the Virtual Private Server running LEMP stack with A+ HTTPS certification.	Oppgave nr.: 54E

Oppgavestiller: Mjelde IKT v / Olav Alexander Mjelde (egendefinert)		
	Kontaktperson: Olav Alexander Mjelde	
	Telefon: 98425342	E-postadresse: *olavamjelde@gmail.com*
	Postadresse: O.J. Brochs Gate 18, 5006 Bergen	
Studenter: Olav Alexander Mjelde		
Veileder ved IIE,NTNU: Mildrid Ljosland		

Sammendrag: Bachelorsoppgaven tar for seg utvikling av en prototype Android applikasjon med tilhørende tjenestebuss for å kunne vise relevante data for brukerens kontekst.
Tid og sted blir premissleverandører for kontekstuelle data i form av vær og luftkvalitet. Bruker av applikasjon kan melde tilbake faktisk opplevde vær- og luftkvalitetsforhold.

Abstract in English: The bachelors assignment aims to develop a prototype Android application and a service bus for displaying contextual relevant data based on the users context.
Time and place will be the entry point into weather and air-quality. The application user can report back individual findings about the local variations of weather or air quality.

Forord

Bacheloroppgaven ble gjennomført våren 2016 ved Norges teknisk-naturvitenskapelige universitet (NTNU) som en del av Informatikk med spesialisering i informasjonsbehandling.

Oppgaven ble tatt ved siden av fulltidsjobb, oppdrag på enkeltmannsforetak og styrejobb i et sameie. Dette medførte at gjennomføringsevnen var avhengig av å holde fokus på å ikke drive med prokrastinering [1], samt behovspyramiden til Abraham Maslow [2] ble sett i sammenheng med mine personlige motivasjonsfaktorer.

Dette ble aktualiserert gjennom studieleder Svend Andreas Horgen sin video [3] hvor det poengteres at studenten må dra prosjektet, og viktigheten av å føle eierskap kommer frem.

Min motivasjon skapes i stor grad av interesse for teknologi, nysgjerrighet, kreativitet og faglig utfordring. Ved å ha fokus på dette kunne jeg snevre inn valget av en oppgave.

På fritiden deltok jeg i FriskBy [4], hvor vi lagde luftkvalitetssensorer av mikrokontrollere. Dette inspirerte på en måte som skulle vise seg å bli utslagsgivende for idefasen.
Åpne data hos Norsk institutt for luftforskning (NILU), gjennom luftkvalitet.info [5] gjorde meg oppmerksom på offentlig tilgjengelige data jeg kunne bruke i prosjektet.

For å lage noe som ikke bare er aktuelt når inversjonsfenomenet [6] oppstår, ble oppgaven å lage en kontekstavhengig applikasjon som skulle munne ut i en fungerende prototype for vær og luftkvalitet.

Jeg vil takke veileder Mildrid Ljosland v/ NTNU for god veiledning, studieveileder Svend Andreas Horgen v/ NTNU for gode tilbakemeldinger og konsulent Harald S. Ulriksen for verdifulle tilbakemeldinger på mellomlagringskonseptet i den tjenestebaserte arkitekturen. Takk til YR og NILU for at de fostrer innovasjon ved å tilgjengeliggjære sine data.

Sist men ikke minst, takk til samboeren min Ingvild Sundal for korrekturlesing av hovedrapporten, samt tålmodighet når jeg har jobbet i hvert eneste ledige tidsrom.

Bergen, 25/05-2016
Olav Alexander Mjelde

_____

Oppgavetekst

Oppgaven omhandler utviklingen av prototypen for en kontekstuell applikasjon med navn "Kontekst", som ut fra konteksten tid og sted, skal være premissleverandør for relevant innhold i form av sammenstilling av data på mobile enheter med Android operativsystem.

Prototypen skal levere luftkvalitet og vær i brukerens kontekst (der og da), så vel som kart.

Prototypen skal utvikles i Android Studio, og det skal være mulighet for metadataberikelse (innrapportering av opplevd vær- og luftkvalitet).

Oppgaven er ikke bare praktisk med programmering og lagdeling i arkitektur og kobling mot applikasjonstjenestegrensesnitt, men det er også en del valg som må tas vedrørende infrastruktur, lagringsstrategier og arkitektur.

Øvrige krav omfatter blant annet forstudierapport, kravdokument og sluttrapport med tilhørende vedlegg som Gant-diagram for aktiviteter som skulle utføres, timeføringsrapporter og ukesrapporter.

Kravene til systemet er nærmere belyst i forstudierapporten [7] og Kravdokumentet [8].

Sammendrag

Bacheloroppgaven «Kontekst» ble utarbeidet i 2016 og bygger i stor grad på over 11 års erfaring med utvikling, forvaltning og drift, så vel som kunnskap tilegnet under utdanning ved Høyskolen i Sør-Trøndelag (nå NTNU) og under tidligere studier. Denne kunnskapen og erfaringen materialiserer seg som bacheloroppgaven «Kontekst» ved NTNU våren 2016.

Oppgaven er egendefinert, og har som har som formål å utvikle en kontekstavhengig applikasjon for å vise data som er relevant i brukerens kontekst (tid og sted). For å få til dette er man avhengig av en plattform, som i dette tilfellet ble Android. Plattformen ble valgt med tanke på markedsandeler for Android, sett i sammenheng med synergieffekten sensorer og vedvarende oppkobling mot Internettet gir for en kontekstuell applikasjon.

For å ha noe å presentere i applikasjonen var det viktig å velge seg ut noen datakilder som var tilgjengelig, og kunne være hensiktsmessig å benytte i brukerens kontekst – lavt hengende frukter. Dette er to kilder man kan si er lavt hengende frukter ved at det er lett å få tilgang til de, og ubegrenset antall oppslag er gratis. Luftkvalitetsdata og værdata var to kilder som utfyller hverandre ved at luftkvaliteten påvirkes av værforhold [6], og i motsetning til luftkvaliteten er de fleste opptatt av været hele året.

Forstudierapporten [7] viste at dette konseptet ikke er gjennomført på samme måte for Android plattformen fra før, kontekstavhengige vær og luftkvalitetsdata i én applikasjon.

Kravdokumentet [8] materialiserte ett av flere vilkår fra YR som tjenesteleverandør [9], kravet om mellomlagring av data i 20 minutter. Dette utløste behovet for å utvikle en tjenestebuss mellom den mobile enheten og tjenestetilbyderne slik at data kunne gjenbrukes.

Tjenestebussen ble laget som en tjenestebasert arkitektur som oppfører seg agnostisk overfor data. Det vil i praksis si at tjenestebussen ikke tolker data den mottar fra tjenestetilbyderne, men den oversetter kall fra applikasjonen mot tjenestetilbyderne, mellomlagrer svarene og komprimerer dem før data sendes over kryptert kommunikasjon som nyttelast. Tjenestebussen sparer båndbredde, tid brukt på å slå opp domener og den muliggjør å slå opp posisjon mot YR og NILU som ikke har dette innebygd i sine applikasjonsprogrammeringsgrensesnitt.

Tjenestebussen ble valgt å kjøre på en såkalt LEMP-stakk [10] hvor man benytter seg av modulære komponenter som baserer seg på åpen kildekode uten lisenskostnader. Dette gir muligheter for å levere god ytelse på virtuelle tjenere med lave driftsutgifter.

Innholdsfortegnelse

Figur- og tabelliste

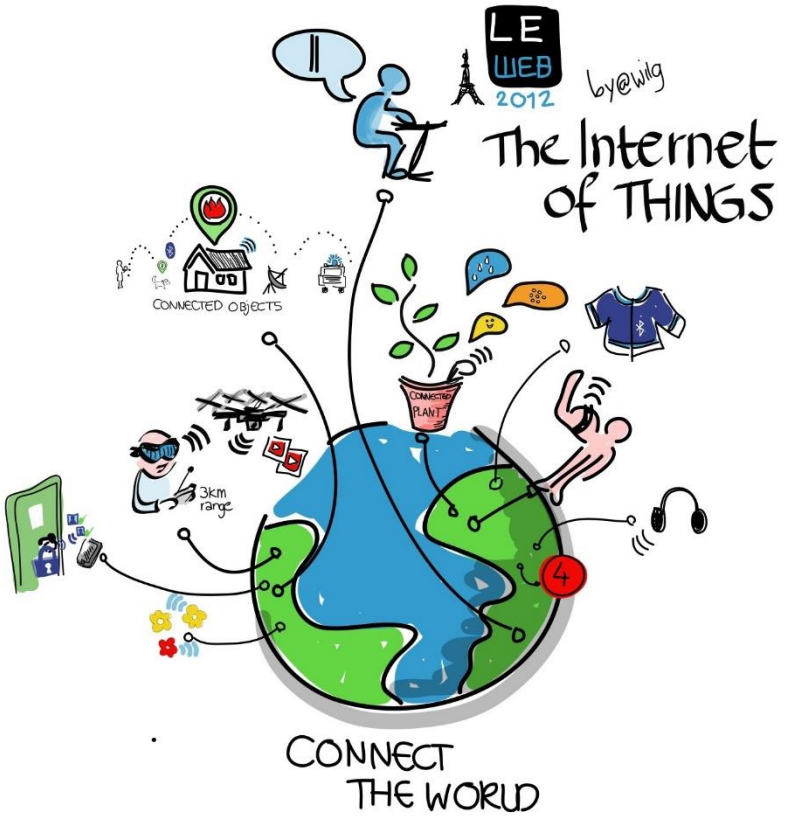
1. Innledning

Dette kapittelet gjør rede for behovet som skal dekkes og hvilke mål som skal oppnås. Rapportens struktur, akronymer og forkortelser forklares i egne underkapitler.

1.1 Introduksjon

Det eksisterer i dag tilkoblede enheter i form av vaskemaskiner, kjøleskap, robotstøvsugere, termostater, værstasjoner, biler og andre produkter, men Tingenes Internett er for det meste et velkjent begrep for teknologer og andre teknologiinteresserte.

Innen få år må derimot folk flest ta stilling til og interagere med tilkoblede enheter.



Figur 1 Connect the world av Wilgengebroed på Flickr

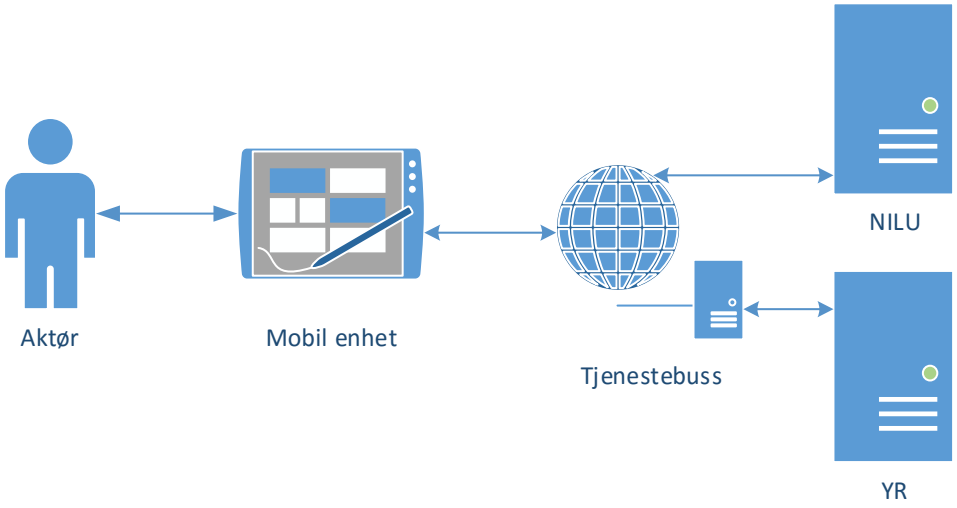
Antall tilkoblede enheter er spådd å nå 6,4 milliarder i 2016 av Computer World (CW) [11]. En enorm mengde tilkoblede enheter vil ifølge en rapport fra Aberdeen Group [12] medføre flere problemstillinger som.

- Den som ikke evner å tilpasse seg, vil drukne i data
- Heterogene datakilder gir utfordringer

For de fleste vil nok dette høres ut som påstander som kan virke logisk ved at flere og flere enheter blir tilkoblet og produserer data kontinuerlig. Denne dataen blir gjerne kalt en informasjonsjungel ved at man kan drukne i data. Informasjonsjungelen gir altså utfordringer for brukerne, hvis informasjonen ikke er tilrettelagt på en slik måte at brukeren ikke drukner i informasjon.

Prototypen «Kontekst» vil bygge på konsepter som kan brukes for å unngå de to problemstillingene, ved at man setter brukerens kontekst som utgangspunktet for å ikke drukne i data. I praksis lar man altså konteksten tid og sted være premissleverandør for hvilke datakilder og hvilke data som er relevant for brukerens kontekst.

Tjenestebussen som kobler sammen heterogene datakilder er også en del av prosjektet.



Figur 2 Tjenestebussarkitektur av Olav Alexander Mjelde

Med tjenesteorientert arkitektur trenger ikke den mobile enheten å forholde seg til heterogene oppslag i API-er fra tjenestetilbydere. Den den mobile enheten kan be om data fra flere kilder gjennom parametere mot tjenestebussen.

Konseptet «Kontekst» som munner ut i en prototype vil derfor kunne videreutvikles til flere heterogene datakilder, alt fra personlige værstasjoner til kjøleskap, robotstøvsugere og annet.

1.2 Behov som skal dekkes

Det er behov for å kunne gjøre oppslag mot heterogene datakilder på en enkel måte, ved å benytte seg av tjenesteorientert arkitektur med en tjenestebuss i senter av løsningen. Denne tjenestebussen må utjevne forskjellige måter man må spørre tjenestetilbyderne sine API-er for data. Tjenestebussen må også mellomlagre data i henhold til YR sine retningslinjer, for å forhindre at tjenestetilbyderne blir overbelastet. Det er også et behov for å komprimere data før det sendes mot mobile enheter for å minimere båndbreddebruken. Dette behovet er både for å øke hastigheten for overføringen og minimere båndbreddekrav til både mobile brukerne og tjenestebussen.

Det er også et behov for persistent lagring av data som meldes tilbake av applikasjonsbrukere. Dette betyr at tjenestebussen må lagre data på en hensiktsmessig måte når det gjelder integritet og ytelse, sett i sammenheng med ressursbruk, lås og GEO-spatiale data.

Kommunikasjon mellom klient og tjenestebuss må gjøres gjennom sikret kommunikasjon ved at det sendes brukerens koordinater for å finne riktige data, samt man ønsker ikke at andre skal bruke tjenestebussen til å lansere egne tjenester.

1.3 Effektmål

Effektmål man sitter igjen etter gjennomføring av prosjektet er:

- Økt fokus på luftkvalitet
- Nye markeder og kanaler gjennom applikasjonsutvikling
- Økt goodwill ved å lansere en samfunnsnyttig tjeneste
- Økt kompetanse ved å tilegne kompetanse og erfaring rundt bruk, konsumering og sammenstilling av tredjeparts datakilder, mellomlagringsmetodikk og applikasjonsutvikling.

1.4 Resultatmål

Gjennomføringen av oppgaven vil resultere i følgende resultater:

- Kontekst – en kontekstavhengig Android applikasjon
- Prosjektrapport
- Vedlegg
- Forstudierapport

- Kravdokument
- Konseptskisse av løsningen
- Fremdriftsplan
- Timelister
- Møtereferater
- Kontrakt
- Dokumentasjon i kildekode
- Dokumentasjon av grensesnitt (bruksanvisning)
- Video av applikasjonen i bruk

1.5 Prosessmål

Følgende prosessmål er definert for oppgavens gjennomføring:

- Kompetansebygging
- Nettverksbygging
- Levere et godt produkt

1.6 Rapportens struktur

Rapporten er bygd opp for å kunne leses fra start til slutt, men kan også brukes for å slå opp på relevante kapitler uavhengig av hverandre. Sammendraget vil fortelle hva prosjektet omfatter og kan være greit for å sette seg kjapt inn i rapporten uansett om man har som formål å lese fra start til slutt, eller bare slå opp på deler av rapporten.

Oppbygningen videre i rapporten foregår under kapitlene:

- Innholdsfortegnelse, figur- og tabelliste
- Innledning
- Teori
- Valg av teknologi og metode
- Resultater
- Diskusjon
- Konklusjon og videre arbeid
- Referanser
- Vedlegg

Respektive kapitler har underkapitler for logisk inndeling.

1.7 Definisjoner og forkortelser

En omfattende liste over definisjoner og forkortelser er definert i kravdokumentet [8] kapittel 1.3, *«Definisjoner og forkortelser»* i tillegg til forkortelser i vedlegget.

2. Teori

Underkapitlene viser teori for relevant teknologi som måtte vurderes for å kunne implementere løsningen for å kunne etterleve kravspesifikasjonene i størst mulig grad.

Kilder er primært fra Internett grunnet dette prosjektet i stor grad går utenfor pensum.

2.1 Tjenesteorientert arkitektur

Den tjenesteorienterte arkitekturen med en tjenestebuss i sentrum understøtter homogene kall mot heterogene datakilder. Dette betyr at tjenestetilbyderne som i dette prosjektet er definert som YR og NILU kan kalles på homogen måte fra applikasjonen.

2.1.1 Heterogene datakilder

NILU sin API på luftkvalitet.info har mange måter den kan kalles og ikke alle har tilgang til alle kallene, samt parameterne er annerledes enn YR sine. Formålet med en tjenestebuss er altså å ha ett sted man kan be om data, altså i tjenestebussen.

Eksempelkall	
YR	/stad/Noreg/Telemark/Sauherad/Gvarv/varsel.xml /sted/Norge/Telemark/Sauherad/Gvarv/varsel.xml /place/Norway/Telemark/Sauherad/Gvarv/forecast.xml
NILU	/onlinedata/timeserie/v2/?id=3&format=xml&from=201603312000&to=201603312200&key=<nøkkel>

Eksempelkall ovenfor viser et vesentlig problem man har uten tjenestebussen. Det er mange måter å kalle heterogene API-er på, samt hvordan kan en applikasjon omgjøre koordinater som `60.392130, 5.328050` til de respektive API-kallene? YR bygger sin API rundt hierarkiet til geografiske målepunkter, dette i kontrast til NILU som har en tabell over identifikatorer som ikke ligger tilgjengelig på nett.

Dette er noe som må bygges inn i tjenestebussen for å utjevne forskjellene i kall mot API-er.

2.1.2 Tjenestebussen

Applikasjonen kan be tjenestebussen om data for en gitt lokasjon og tjenestebussen finner relevante datapunkter fra kilder den har, og sender dette i retur. Dette er ikke en

hyllevarekomponent, men noe som må utvikles skreddersøm for oppgaven, kapittel 2.2, 2.3 og 2.4 omhandler teori som er nødvendig for å kunne designe en tjenestebuss.

2.2 Tjenertyper

I en tidsepoke hvor virtuell tjenerteknologi er tilgjengelig hos mange aktører er det fremdeles en del valg man må ta før man startet et prosjekt. Valg av tjenertype er et grunnleggende valg.

2.2.1 Plattform som en tjeneste (PaaS)

Metoden PaaS baseres på skytjeneste med abstraksjonslag mot underliggende arkitektur. Denne plattformen inneholder ofte tjenester som autentisering mot sosiale medier, funksjoner for å lagre bilder og andre binære data. Man har ikke noen form for tilgang eller oversikt over underliggende arkitektur, og forholder seg derfor ikke til teknisk drift.

Fordeler	Man kan begynne utvikling med en gang, og kan spare mye tid ved at tjenester er laget fra før i PaaS. Tidsbesparelser kommer i form av at man ikke må drifte en tjener, samt mange tjenester leveres ferdig som moduler. PaaS er ofte bygd på skyteknologi som skalerer effektivt og automatisk. Det kan defineres at den får lov å bruke en gitt mengde ressurser og den vil automatisk tilpasse seg.
Ulemper	Kan bli dyrt hvis man har behov for mye trafikk eller mange tjenester. Utviklingen av egen applikasjon/tjeneste kan bli avhengig av PaaS-tilbyderen man valgte. Dette kan gi utfordringer hvis behov eller tilbud endres.
Prismodell	Priser varierer basert på antall tjenester. Alt fra relasjonsdatabaser til NO-SQL databaser, kryptert kommunikasjon og annet er priset. Prisene er ofte utregnet basert på bruk av diskplass, båndbredde, CPU-tid og allokert minne. Hvor mye hver tjeneste er aktiv vil altså påvirke sluttregningen. Det kan være vanskelig å estimere driftsutgiftene.
Passer best for	De som vil ha en løsning uten å ha ansvar for drift, samt ha fordelene skyen kan gi med umiddelbar skalering når en tjeneste har mange brukere.

Et godt eksempel på en slik tjeneste er Parse [13] som leverte tjenester for utallige applikasjonsutviklere. De ble kjøpt opp av FaceBook som blant annet teknologinettstedet Tech Crunch trodde ville styrke utviklernes muligheter [14]. Mange var positive til dette oppkjøpet frem til nyheten om at Parse skulle legge ned tjenesten ble kjent i et blogginnlegg [15]. Løsningen ble gjort tilgjengelig i åpen kildekode, men dette som ble sett på som en enkel måte å komme i gang med applikasjonsutvikling viste seg senere å bli en uventet utfordring for utviklerne.

2.2.2 Mykvare som en tjeneste (SaaS)

Tjenesten har tradisjonelt sett vært en måte å lisensiere programvare på, gjerne brukt for mindre kommuner når de skal ha sakarkiv løsninger eller andre hyllevareløsninger. Tjenesten kjøres sjeldent på egne tjenere, men noen leverandører har begynt med løsninger for privatsky.

Fordeler	Fleksibel metode for å betale for antall aktive brukere eller annen måte som ofte rettferdiggjør størrelse av utgifter opp mot bruk. Løsning driftes av leverandør, men store leverandører kan også tilby privatsky til firmaer som vil ha kontroll på data. Det kalles hybridsky når tjeneren plasseres i eget lokale men drift og forvaltning er satt ut. Mange ser også fordelen med skreddersøm at man ikke trenger å involvere driftspersonell for å oppgradere til siste versjon. Forvaltning skjer tilsynelatende av seg selv ved at alle kundene kjører samme versjon.
Ulemper	Skreddersøm utvikling utgår.
Prismodell	Ofte er det en grunnpris med påslag per aktive bruker, samt eventuelt påslag for lagret datamengde.
Passer best for	De som ikke vil, eller kan drifte selv og kan bruke hyllevareløsninger.

2.2.3 Dedikert tjener

Dedikert tjener er den mest tradisjonelle måten man kan tilby sine tjenester, hvor en tjener står i et rom eller anlegg dedikert for dette.

Fordeler	Dedikerte tjener gir dedikerte ressurser, som medfører mulighet for å utnytte tjenerens maskinvare til det man ønsker.
Ulemper	Med dedikert tjener må man ta hånd om sikkerhetskopiering, overvåkning, testing og generelle driftsavvik. Skalering er vanskeligere og dyrere enn de virtuelle løsningene ved at man må gå til innkjøp av maskinvare og sette dette opp, og tilpasse tjenestene for skalering. Fysisk defekt på tjeneren vil være direkte knyttet til maskinvaren uten noen form for abstraksjon eller skyløsning, som ofte medfører at tjenesten går ned frem til man har fått tak i kompatibel maskinvare.
Prismodell	Kunden betaler som regel en relativt høy engangssum for innkjøp, samt nettlinjen er ofte prissatt per måned. Det er ofte forutsigbare priser, men en dedikert tjener kan være kostbart for en nyetablert bedrift.
Passer best for	De som vekter høyt å sikre data, krav til ytelse eller behov for å drifte tjeneren selv.

Hvis man betaler for driftstjenester er det viktig å sette seg inn i og eventuelt forhandle en Service Level Agreement, en såkalt SLA-avtale. SLA-avtaler regulerer faktorer som forventet tjenestekvalitet, oppetidsmålinger og andre vesentlige faktorer for driftssituasjonen [16].

2.2.4 Virtuell Privat Server (VPS)

VPS gir et abstraksjonslag mellom fysisk maskinvare og operativsystem ved å emulere en eller flere tjenere på en fysisk tjener. I dag kalles en virtuell tjener ofte for infrastruktur som en tjeneste (IaaS).

Dette muliggjør kjøring av en eller flere VPS på en eller flere dedikerte tjenere, uten at det medfører andre sikkerhetsrisikoer enn man hadde hatt på en dedikert tjener. Prinsippet kalles for *sandkasse* ved at man gir full tilgang innenfor et kontrollert område. Med dette konseptet har man tilgang til å ødelegge for seg selv, men i teorien skal det ikke gå ut over andre.

Fordeler	Ingen ventetid når man går til anskaffelse. Kan ofte settes opp på 1-2 minutter hos de mest kjente leverandørene. Mulighet for å kopiere hele tjeneren for å teste den med oppgraderinger eller andre tjenester, og mulighet for å skalere enkelt og kjapt. De største tilbyderne har brukerforum og instrukser for mange typer tjenesteoppsett.
Ulemper	Krever samme kompetanse som en dedikert tjener - i noen tilfeller litt mer kompetanse. Ofte vet man ikke hvilken maskinvare man får, bare antall prosessorkjerner, mengde primærminne, lagringsplass og trafikk.
Prismodell	Ofte basispris basert på kapasitet man har til rådighet i form av lagringsplass på disk, minne, antall prosessorkjerner og båndbredde. Forutsigbar prismodell.
Passer best for	De som vil ha samme muligheter som en dedikert tjener gir, men har mindre krav til ytelse og ønsker lavere driftsutgifter.

Det fins driftstjenester for VPS på lik linje med en dedikert tjener. Dette kan være hensiktsmessig hvis man har behov for en driftstjeneste som avlaster kravene om kompetanse og ressurser drift av en tjener krever.

På lik linje med en dedikert tjener er det i så tilfelle like viktig å sette seg inn i og forhandle vilkårene for tjenestekvaliteten, og hva som tilbys og forventes av driften. Dette defineres i en SLA-avtale hos bedrifter som har en ITIL V3 tilnærming til drift [17].

2.3 Operativsystem

Så fremt det velges en tjener i form av dedikert tjener eller VPS, vil det måtte velges et operativsystem. Operativsystemet vil ha som ansvar å kommunisere mellom underliggende maskinvare, og tjenester som kjører på maskinen.

Dagens operativsystemer kan kjøre mange av de samme tjenestene som webtjenere, NO-SQL databaser, relasjonsdatabaser, domenetjenester, e-post og annet. Forskjellene ligger altså på andre områder enn hva man kan gjøre med operativsystemene. Faktorer som pris, driftsutgifter og kontinuitet i tjenesten vil derfor være vesentlige for valget i kapittel 3.3

Det man ender opp med vil ofte være omtalt som en stakk med akronymer som LEMP, LAMP og WAMP hvor første bokstaven er operativsystemet, andre bokstav er webtjener, tredje bokstav er databasemotor og fjerde bokstav er skripspråket.

2.4 Tjenester

For å ha en tjenestebuss er det viktig at den kjører noen tjenester som muliggjør kommunikasjon, mellomlagring, komprimering, kryptering og persistent lagring.

2.4.1 Webtjener

Det må installeres en webtjener for å muliggjøre kommunikasjon over HTTP-protokollen. Her må det velges en som er svært effektiv og kan håndtere mange tråder for å understøtte mange sanntidsbrukere fra applikasjonsbrukere. Webtjener må installeres i valgt operativsystem, og må derfor velges etter operativsystem er valgt.

2.4.2 Mellomlagring

Mellomlagring på tjeneren kan forekomme på flere teknologier:

Teknologi	Beskrivelse
Tradisjonell harddisk	Harddiskens plater må spinnes opp, og data er spredd rundt på tilfeldige områder. Billig måte å lagre mye data, på bekostning av ytelse.
Minnebasert disk (SSD)	Datamaskinen kan hente det den trenger ved at det er lagret i minnebrikker som holder på data også ved strømbrudd. Vesentlig dyrere enn tradisjonell harddisk for høyvolumlager, men vesentlig raskere.
RAM-disk	RAM-disk baserer seg på at datamaskinen oppretter virtuelle disker på minnebrikker som er montert i datamaskinen. Dette kan forekomme i form av fysiske kretskort dedikert for formålet, eller gjennom programvare. Ved strømbrudd vil en maskinvarebasert RAM-disk ha batteri for å holde data, og en mykvarebasert RAM-disk vil ofte lagre kopier mot en tradisjonell harddisk underveis, og må gjenopprette ved neste oppstart.

	Mykvarebaserte RAM-disker har av denne årsaken større sannsynlighet for tap av data ved strømbrudd eller krasj. En RAM-disk er veldig kjapp, men samtidig sårbar og den dyreste måten man kan lagre data.
Øvrig RAM-basert lagring	Programvare som kan holde data persistent i minnet fungerer i praksis ikke helt ulikt mykvarebasert RAM-disk, med unntak av at man ikke kan se området, og derfor ikke kan benytte det gjennom en filutforsker. I likhet med en RAM-disk vil man her kunne miste data ved uventede strømbrudd, eller hvis datamaskinen krasjer. Ytelsen er av høy grad, samtidig som at man er mer sårbar for tap av data.

Den raskeste mellomlagringen får man ved å gå bort i fra tradisjonelle diskbaserte løsninger. Tall fra jboner på GitHub [18] tilsier at tilfeldig lesing fra en tradisjonell harddisk er hele 100.000 ganger mer tidskrevende enn tilfeldig lesing fra datamaskinens primærminne. Minnebasert lagring i form av SSD-disker har bedre forutsetninger, men er ifølge samme kilde 1.500 ganger mer tidskrevende enn oppslag fra primærminnet.

2.4.3 Komprimering

I tjenestebussen vil det være hensiktsmessig å komprimere data for å kunne oversende dette raskere til applikasjonen, og minimere båndbreddebruk. Komprimering skjer teknisk sett ved at tjenestebussen bruker logaritmer til å effektivisere lagringsplassen for data, og fotavtrykket blir derfor mindre. Vanlige datakilder som XML og JSON kan ofte komprimeres opp mot 80-90 % ifølge publikasjonen «*XML compression techniques: A survey and comparison*» [19].

Komprimering i denne oppgaven vil gjennomføres i form av en tapsfri komprimering ved at logaritmen analyserer data og ser at det eksisterer visse mønster, eksempelvis: 0000000000. For å spare plass kan den under komprimering si at 0000000000 er det samme som 10 x 0. Jo mer repeterende data man har, jo bedre vil forutsetningen være for å minske fotavtrykket for data.

Komprimeringen i dette prosjektet går ut på å bevare opprinnelig struktur og lesbarhet, altså gjennomføres det ikke erstatning av tagger og verdier, men en generell komprimering av data som skal overføres.

Klienten som i dette tilfellet vil være en Android applikasjon vil lese i hodet til data som overføres fra tjenestebussen, der det fremkommer at data er komprimert. Deretter fjernes komprimeringen av data uten tap, slik at den blir leselig for programkoden.

Dette er en velutprøvd teknologi som nettlesere og webtjenere benytter for overføring. Konseptet implementeres enkelt i tjenestebussen ved at den tar i bruk HTTP-protokollen.

2.4.4 Kryptering

Kryptering i dette prosjektet vil være sikkerhet over transportlaget som forklart hos The Internet Engineering Task Force (IETF®) for RFC2818 [20], ofte omtalt som HTTPS.
I praksis fungerer dette som vanlig kommunikasjon i en nettleser, med unntak av at kommunikasjonen er kryptert med nøkler og derfor ikke er enkel å overvåke.

For å opprettholde en kryptert kommunikasjon på denne måten må man ha et sertifikat som utstedes av et firma, eller en organisasjon som går god for den det er utstedt til. Levetiden til sertifikatet vil variere ut ifra hvor man anskaffer sertifikatet. Sertifikatene må anskaffes hos en sertifikatutsteder.

2.4.5 Persistent lagring

Persistent lagring kan forekomme på flere måter, alt fra disk til relasjonsdatabaser.
I en relasjonsdatabase vil det være mulig at flere aktører kan aksessere den samme dataen på en gang, samt databasemotoren vil ha effektive metoder for å håndtere ytelse i form av unike nøkler og annet som kan påføres i et databasedesign.

NoSQL-databaser vil ha noen av fordelene til relasjonsdatabasen, og vil for ustrukturerte data ofte være vesentlig raskere enn både relasjonsdatabasen og flate filer.

En databasemotor basert på NoSQL eller en av flere relasjonsdatabaser kan utføre mange operasjoner effektivt, som beregning av avstander, sortering, summering, aggregering og annet som vil være viktig for tjenestebussen.

2.4.6 Sikkerhet og kommunikasjon

Sikkerhet og kommunikasjon er et sammensatt bilde, men formålet med tjenestebussen aktualiserer sikring i form av:

- Brannmur for å begrense tilganger til porter
- Antivirus for å detektere og stanse uønsket kode
- Rettigheter og grupper for å begrense tilgang til ressurser
- Nøkkelgenerering for pålogging for å sikre autorisasjon
- Deaktivere administratorpålogging for å hindre automatiserte angrep
- PDO kobling mellom PHP og MySQL for å hindre SQL-injeksjon
- SSL (HTTPS) for å hindre overvåking av trafikk
- Filoverføring over kryptert kanal
- Rutiner for oppdatering av tjener og tjenester

Beste praksiser gjennomført i faget *Linux Systemdrift* ved HiST (Nå NTNU) er i stor grad bakgrunnen, men mange av prinsippene er også nevnt i artikkelen fra DO *«An Introduction to Securing your Linux VPS»* [21].

Når det gjelder kommunikasjon må det også velges kommunikasjon internt i tjenestebussen mellom de forskjellige tjenestene, der vil det variere hva man kan og bør velge basert på operativsystem. Windows har *Sockets* og *Named Pipes* som nevnt i *«Named Pipes vs. TCP/IP sockets»* på TechNet [22] , Linux har *TCP Sockets* og *Unix Domain Sockets (UDS)*. Manualen til Linux [23] viser at hovedforskjellen er at Unix Domain Socket går rett mot bufferen og TCP sockets går gjennom en nettverksbasert loopback. Dette betyr at for tjenester som kjører på samme tjener, vil domain sockets være teoretisk overlegen i ytelse.

Implementering vil være aktuelt å ytelsesteste på mellomlagringstjenesten som velges.

2.5 Applikasjonsutvikling

Før man kan starte utviklingen av en applikasjon er det viktig å velge noen prinsipper å følge for å opprettholde ønsket tjenestekvalitet, og gjennomføre prosjektet på en hensiktsmessig måte.

2.5.1 Prosesser og tråder

Tråder, som forklart av Google sine utviklersider for Prosesser og Tråder [24], belyser en aktuell problemstilling rundt hovedtråden applikasjonen kjører i. Hovedtråden tegner grensesnittet, sender hendelser til grensesnitt og interagerer med komponenter fra den grafiske verktøykassen.

I utgangspunktet kjører alle komponenter på samme tråd, og alt av hendelser og operasjoner legges i en kø med en livssyklus. Et aktualisert problem i denne sammenhengen er at intensive oppgaver vil gi følelsen av at applikasjonen henger. Hvis applikasjonen henger i over 5 sekunder vil man også få feilmeldingen om at applikasjonen ikke svarer, som gjerne medfører at brukerne lukker applikasjonen.

Applikasjonsutvikling som understøtter flere tråder foreslås av samme utviklerressurs å baseres på enten *Worker threads* eller *Async Task* som jeg oversetter til arbeidstråder og Asynkrone oppgaver.

Hovedforskjellen på arbeidstråder og de asynkrone oppgaver er hvordan de interagerer med grensesnittet uten å potensielt medføre uventede programstopp hvis de ikke er trådsikre. Asynkrone oppgaver er vesentlig mer lesbare, og kan enklere oppdatere grensesnittet ved at

de har automatisk håndtering av tråder som nevnt av utviklersidene for asynkrone oppgaver [25].

2.5.2 Grensesnitt

Minimalisme er et ord som brukes innen designtrender ikke bare hos utviklere, men alt fra møbeldesignere til arkitekter som designer hus. Minimalismen på Internett og i applikasjoner er derimot relativt ny. Artikkelen *«Google/Bin: Minimalism vs. Maximalism»* [26] viser til hvordan Microsoft sin søkemotor Bing og Google sin søkemotor Google så ut i 2009, hvor Google gikk for minimalisme og Microsoft gikk for maksimalisme.

Google deler sine prinsipper gjennom nettstedet *Google Design* [27]. Prinsippene omtales som *Material Design Guidelines* som på eget nettsted har en full gjennomgang av hva *Material Design* er, og hvordan det implementeres [28]. Nettstedet går gjennom hvordan man kan få et interaksjonsdesign til å fungere best mulig for alle typer brukere, og hvordan man kan etterstrebe implementeringen av en moderne og minimalistisk stil.

Grensesnittet baserer seg på bruk av C*ards* som i prinsippet er kort med informasjon som flyter i en strøm, det er rene farger, bruk av marg, fontstørrelser, kontraster og andre prinsipper som medfører en helhetlig praksis for god interaksjonsdesign og høy grad av universell utforming.

2.5.3 Lister

Listene i Android kan være basert på komponenten *List View* [29] eller komponenten *RecyclerView* [30] som er en liste som kan gjenbruke grensesnittet og bytte ut data når brukeren navigerer i listen. *RecyclerView* er smart nok til å selv laste det den trenger og kan holde på grafiske elementer i *ViewHolders* [31] . En *View* er i praksis et grensesnitt og en *ViewHolder* sørger for å holde det grensesnittet som gjerne er et *Card*. Styrken til *RecyclerView* er at man kan gjenbruke visuelle komponenter og får derfor veldig god ytelse i listene.

2.5.4 Databinding

Når data skal vises i grensesnittet må det bindes gjennom en eller flere måter Google nevner for å binde data i dokumentasjonen for *«Data Binding Guide»* [32]. Prinsippet databinding er velkjent innen IT-miljøer som *Model-view-controller* (MVC). Prinsippet går ut på en seperasjon av *Model* (data) og *View* (grensesnitt), som foregår ved hjelp av en kontroller som flytter og oppdaterer endringene.

Dette er en tilnærming utviklere gjør innen alle områder i applikasjonsutvikling, alt fra klientprogrammer til websider. MVC-prinsippet forenkler endring av designet, og gjør det enklere å gjenbruke komponenter ved at det ikke er hardkodet inn i grensesnittet. Dette medfører også en forenkling når man skal implementere internasjonalisering.

Prinsippet MVC ble først omtalt av nordmannen Trygve Reenskaug som et notis når han jobbet for Xerox PARC i 1978 [33].

2.5.5 Dekoding av Extensible Markup Language (XML)

Data fra tjenestebussen må dekodes før den kan implementeres i en databinding ved at en XML [34] i utgangspunktet vil være en datastrøm som må tolkes, og lastes inn i en klasse før den kan bindes til en datamodell i henhold til Google sine prinsipper for databinding [32].

Dekoding kan gjøres på flere måter i Android. De tre mest omtalte metodene er XMLReader [35] som ikke lenger anbefales brukt, DocumentBuilder [36] som leser XML inn i et dokument-objekt og XMLPullParser [37] som på effektivt vis kan tolke XML etter hvert som den itererer strukturen i XML-dokumentet. En av de tre metodene må velges.

2.6 Utviklingsmetoder

Utviklingsmetode definert i forstudierapporten [7] kapittel 3.5, ble gjort med hensyn til at det var mange ukjente faktorer i oppgaven, og at man ved å velge prototyping derfor enklere kunne spisse seg inn mot et utviklingsløp som er hensiktsmessig.

Det er flere typer prototyping [38] og i kapittel 3.6 må det velges en metode

Metode	Kjennetegnes av
Throwaway Prototyping	Jobber med en modell som etter hvert blir forkastet. Det viktigste er å utvikle noe raskest mulig som viser konseptet.
Evolusjonær prototyping	Målet er å lage en mest mulig robust prototype med en strukturert tilnærming og konstant forbedre prototypen. Dette medfører kontinuerlige endringer og rekompileringer. Resultatet blir en fungerende applikasjon.
Inkrementell prototyping	Sluttproduktet er bygd som seperate prototyper. Ved fullførelse settes prototypene sammen i ett design. Dette er effektivt hvor man har større team, og flere kan jobbe på hver sin del for å få ned tiden det tar å levere til markedet. Man utvikler da i parallelle løp fremfor å vente på andre.
Extreme prototyping	Særlig utbredt for webapplikasjoner hvor man i praksis deler det i tre: 1) Statisk grensesnitt utvikles 2) Utvikling og programmering på statisk grensesnitt, simulering av datakilder 3) Implementering av tjenester og datakilder

	Denne metoden er derfor veldig godt egnet når man har en kunde som er opptatt av et grafisk grensesnitt, eksempelvis en reklamekampanje, nettsted eller noe annet som er designorientert.

3. Valg av teknologi og metode

Siden prototypen kontekst er avhengig av en tjenestebuss med mellomlagring, GEO-spatiale søk, mellomlagring på effektiv måte og mange valg i utviklingen i Android, vil det være viktig å sammenligne alternative komponenter for bruk i applikasjon og tjenestebuss.

De største komponentene ble sammenlignet i den grad det var mulig å vekte de mot hverandre. Ved sammenligninger måtte de være utført på samme versjon av programvare eller tjeneste for at det skulle være hensiktsmessig å vekte valg. Årsaken til dette er at funksjoner i programvare og operativsystemer ofte endres mye under nye versjoner.

De fleste teknologiske implementeringene eksisterer på flere plattformer. En stor utfordring er tiden til rådighet i prosjektet. Før man i det hele tatt kunne begynne med utviklingen, måtte det gjøres en del valg, oppsett, installasjon og kompetanseheving.

Dette betyr at mange av komponentene er valgt basert på artikler som omhandler teknologien.

Undertegnede har erfaring med flere plattformer, og hadde et åpent sinn når det skulle velges teknologi.

Metoden ble altså å sammenligne først valg av tjenertype, og hvilken tjenertype som best støtter formålet til en best mulig pris/ytelse. Deretter valg av plattform og til slutt valg av tjenerlokasjon som derfor effektivt snevret inn valg av operativsystem.
Videre fortsatte valg av skriptspråk og mellomlagringsteknologi. Ikke alt lot seg sammenligne med A-B sammenligninger eller artikler, grunnet at artiklene var for gamle eller sammenlignet basert på egenskaper som ikke var relevant.

3.1 Tjenesteorientert arkitektur

Selv om løs kobling gir en enorm fordel når det gjelder vedlikehold, utvikling, strømforbruk og ytelse ved at man flytter prosesser til tjenestebussen, er det YR sitt krav om mellomlagring av data som var det viktigste for valget. Den kollektive hukommelsen i form av en sentralisert mellomlagring ville ikke vært mulig uten tjenestebussen, og da ville applikasjonen gått vesentlig tregere og belastet yr med utallige repeterende tjenestekall.

3.1.1 Heterogene datakilder

Valgt metode for implementering baseres på tjenestebussen, hvor oppslag mot heterogene datakilder deretter kan utføres på en homogen måte. Applikasjonen kan bruke parameterverdier mot tjenestebussens grensesnitt for å hente data fra flere kilder.

3.1.2 Tjenestebussen

Tjenestebussen er implementert som kode med tilhørende mellomlagring og persistent lager i henhold til kapittel 3.4. Den tjenesteorienterte arkitekturen materialiseres som tjenestebussen.

I kapittel 4 vil rapporten gå mer i dybden på tjenestebussen.

3.2 Valg av tjenertype

Felles for samtlige tjenertyper er at man må ha en utgangsstrategi som tar høyde for eventualiteter som endrer forutsetningene for fortsatt drift. Dette kan være økonomiske rammer, endring i tjenester, endring i vilkår, endring i lovverket, krig, terror eller andre uønskede hendelser.

Før man velger tjenertype må man vurdere om det er sensitive data, eller data som ikke bør havne på avveie av annen årsak. Det må vurderes om man har kompetanse, tid og ikke minst motivasjon til å forvalte tjeneren på en hensiktsmessig måte i tiden som kommer.

De forskjellige tjenertypene krever forskjellig kompetanse, og det er viktig å vite at skytjenester i utlandet kan være ulovlig [39]. Når det gjelder denne bacheloroppgaven er det ingen restriksjoner på bruk av skytjenester.

Basert på fordeler og ulemper i kapittel 2.1 ble det valgt å satse på en virtuell tjener (VPS), altså en infrastruktur som en tjeneste (IaaS). Dette gir forpliktelser for drift og forvaltning, men samtidig fleksibilitet for skalering og lave driftsutgifter.

En synergieffekt ved valg av VPS var også at nettstedet til Mjelde IKT kunne flyttes til samme tjeneren og på den måten ble det en gevinstrealisering med VPS fremfor et webhotell.

3.2.1 Valg av tjenestetilbyder

Valg av tjenestetilbyder for VPS er en minst like stor utfordring som valg av tjenestetype. Det er tilsynelatende uendelig med leverandører og spørsmålet er hvordan du vet at leverandøren du velger gir deg ytelsen du trenger, samtidig som at man ikke betaler for mye.

Prisene hos de største aktørene er relativt like, og det viste seg å være praktisk talt umulig å sammenligne de forskjellige tilbyderne sine ytelser grunnet at man trenger å utføre interne

ytelsestester. Tilbyderne har forskjellige serverparker rundt om i forskjellige deler av verden, og det fins ikke en standard måte å gjennomføre ytelsestester på. For å gjennomføre en slik sammenligning må man i praksis kjøpe en konto hos hver tilbyder.

Artikkelen «*Comparing Cloud Compute Services*» [40] tok for seg en grundig gjennomgang av forskjellige tilbydere av skytjenester. Alt fra ytelse, standardavvik på ytelser og overraskende funn som tilsa at det ikke alltid ble bedre ytelse med en kraftigere VPS. En av de største utfordringene er at man ikke vet hvilken underliggende teknologi man får, årsaken er at man kjøper en tjeneste med en eller flere kjerner og en gitt mengde minne, samt lagringsplass. Man vet ikke om det er AMD, Intel, om det er årets modell, fjorårets eller eldre. Dette betyr at selv om man kjøper to VPS samtidig, kan man i teorien få forskjellig ytelse.

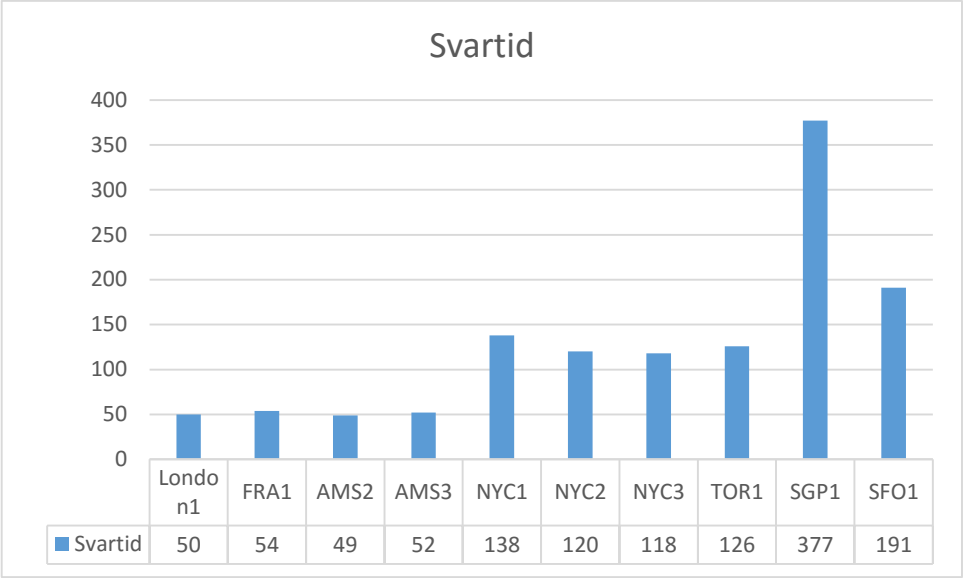
Ytelsestestene i artikkelen viste at Digital Ocean (DO) hadde gode ytelser. Det sett sammen med at de har gode priser og et veldig aktivt og godt brukerforum gjorde at det var interessant å studere DO nærmere. Valget av VPS var fremdeles ikke gjort, men etter å ha studert det nøyere viste det seg at DO er en av svært få aktører som leverer minnebaserte harddisker (SSD) til en overkommelig pris. Minnebaserte disker understøtter høy ytelse på tjenestebussen ved at lastetider til og fra SSD er vesentlig kjappere enn tradisjonelle harddisker.

Valget av VPS-tilbyder falt derfor på DO, basert på en helhetlig vurdering av ytelse og pris, så vel som nettsamfunnet rundt tjenesten. Det å ha et aktivt nettsamfunn kan være en viktig faktor når man skal høste erfaringer og inspirasjon for oppsett.

3.2.2 Valg av tjenerlokasjon

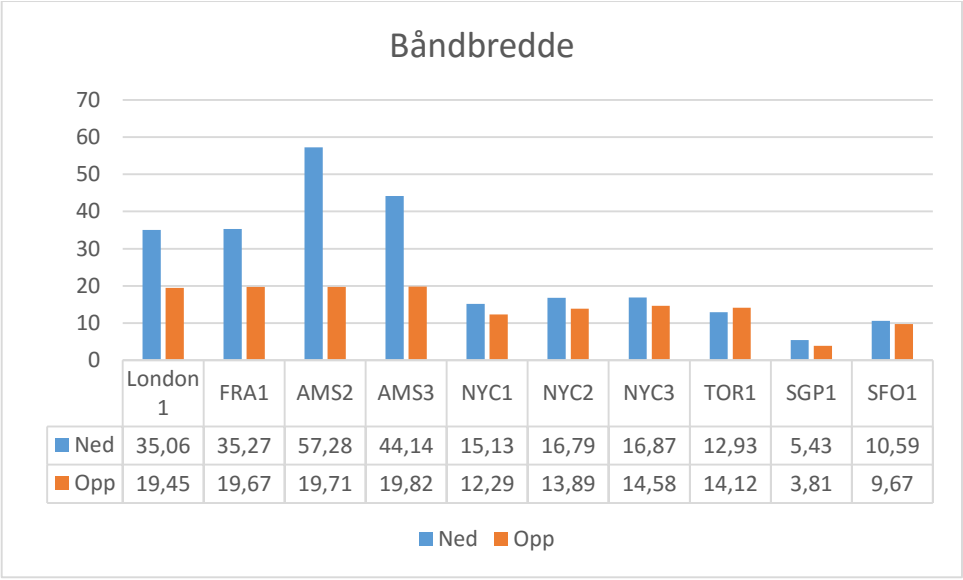
Etter valg av tjenesteleverandør var gjort, gjenstod det å velge datasenter. Valg av datasenter innenfor en leverandør bør gjøres basert på valg som er hensiktsmessig for bruken.

Det ble gjennomført komparative ytelsestester mot de forskjellige datasentrene for å kunne sammenligne de forskjellige egenskapene som var relevant for tjenestebussen. Testene ble kjørt 3 ganger, snittverdiene ble notert.



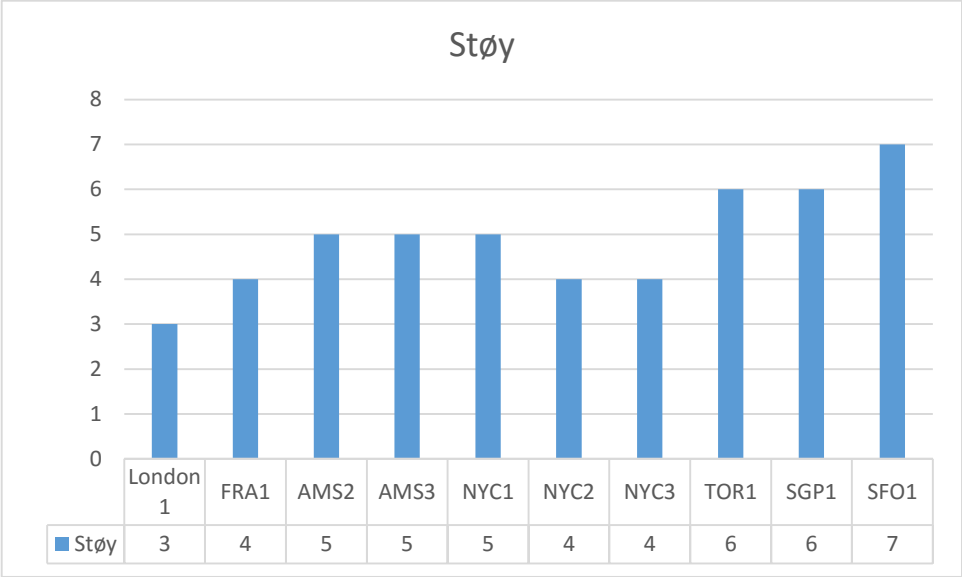
Figur 3 Svartider målt mot datasentre (lavere tall er bedre)

Amsterdam 2 (AMS2) oppnådde best resultat hvis man ser på svartiden alene i figur 3, men samtlige lokasjoner i Europa var relativt jevn i responstidytelse. AMS2 hadde vesentlig raskere ned-hastighet som fremkommer av figur 4. Opp-hastighet er ikke vektlagt av den enkle årsaken at undertegnede under testing hadde en nettlinje med 100 Mbit ned og 20 Mbit opp, altså var det ikke mulig å måle opp-kapasitet. Man kan anta at verdiene vil være relativt like på de synkrone fiberlinjene hos datasentrene.



Figur 4 Båndbredde målt mot datasentre (høyere er bedre)

Støy på linjen var best for London som det fremkommer av figur 5.



Figur 5 Målt støy mot datasentre (lavere er bedre)

Hvis man skal lansere en båndbreddeintensiv-tjeneste som videostrømming vil tjenerens båndbredde være viktigst. Video og talekommunikasjon (VOiP) vil ikke tåle tap av pakker grunnet at VOiP er sanntidsdata og bør derfor velges basert på lavest mulig støy.

For tjenestebussen er det små pakker med data som sendes og derfor vil responstid være viktigere enn maksimal overføringshastighet. Tap av et par pakker vil for tjenestebussen bare marginalt øke responstid og reduserer i så måte ikke kvaliteten av tjenesten.

Datasenteret ble derfor valgt primært basert på svartid ved testing. Vektingsfaktoren ble valgt basert på teorien til Jakob Nielsen sin artikkel *«Website Response Times»* [41] hvor det synliggjøres 3 responstider:

- 0.1 sekunder – gir følelsen av umiddelbar respons
- 1 sekund – gir følelsen av at man er i kontroll over applikasjonen fremfor at de venter
- 10 sekunder – brukeren tåler det men føler at de er på nåde av teknologien

Artikkelen omhandler websider men prinsippet vil være det samme for applikasjoner, ved at man risikerer å miste brukere hvis responstiden er for lang. Før en applikasjon kan presentere data i grensesnittet må applikasjonen hente inn data, tolke data, bearbeide data, laste det inn i datamodellen og til slutt vise det i grensesnittet.

Applikasjonen i prosjektet utvikles for å gi kontekstavhengige data, og tjenestebussens formål er å spare tid på domeneoppslag og hindre at brukerne gjør gjentagende oppslag mot tjenestetilbyderne i de tilfeller andre har lastet de samme data fra før. Derfor er responstider noe av det viktigste vedrørende valg av datasenter i denne oppgaven, ved at responstidene vil komme på toppen av tolking, bearbeiding og visning av data i applikasjonen. Hvor god ytelse den mobile enheten har, hvor rask nettlinjen er og andre faktorer er utenfor kontroll, men alt til og med tjenestebussen må bygges opp for å være raskest mulig.

Valget falt derfor på AMS2 hos tilbyderen DO for VPS.

3.3 Valg av operativsystem

Det er mange operativsystemer å velge mellom hvis man ser på alt som er tilgjengelig, men dersom man skal innsnevre valgene til noe som kan kjøre på tjenerplattformene nevnt i underkapitlene til kapittel 2.1 vil man i stor grad stå igjen med to valg: Linuxbasert eller Windows. I kapittel 3.2.3 vil valg av operativsystem begrunnes.

3.3.1 Windows

Windows er et tradisjonelt operativsystem for tjenere, særlig dedikerte tjenere. Windows er generelt sett et trygt valg, og operativsystemet administreres primært gjennom grafiske grensesnitt og oppdateringer krever som regel omstart med tilhørende nede-tid.
Tradisjonelt sett har ikke Windows eksistert i stor grad for virtuelle tjenere, men nylig har Amazon begynt å tilby virtuelle tjenere med Windows [42], det samme har Google [43] og andre store aktører.

3.3.2 Linux

Linux kommer i mange distribusjoner. Artikkelen fra linux.org sammenligner de mest populære for tjenerbruk ifølge artikkelen fra linux.org [44] og grupperer de etter typer: Debian basert, Diverse, Slackware eller RedHat.

Artikkelen kommer med tre anbefalinger [44]

- Ubuntu for stabilitet, langtidsstøtte, rykte, hvor utstrakt den er i bruk, store pakkebrønner, samt enkel pakkehåndtering
- Arch for hastighet, ytelse, portabilitet og dokumentasjon
- CentOS for gratisutgave av RedHat

3.3.3 Valgt operativsystem

Bacheloroppgavens premisser med å holde utgiftene nede samtidig som at ytelsene holdes oppe, understøtter operativsystemvalg som bruker minst mulig ressurser, sjeldent trenger omstart og er kjent for lang oppe-tid og stabil drift.

Styrkene til Linux for denne bruken vil av overnevnte grunner være viktige nok til å velge vekk Windows uavhengig av datasentervalg. En VPS med Windows vil være betraktelig mer kostbar i drift, både ved at man må betale for mer ressurser, samt betaling for lisenser.

Ubuntu 14.04 LTS ble valgt av Linuxdistribusjonene med tanke på stabilitet, langtidsstøtte, rykte, utstrakt av bruk og enkel pakkehåndtering.

3.4 Valg av tjenester

Tjenestene ble valgt etter valg av operativsystem var definert, og utgjør sammenlagt tjenestebussens stakk som muliggjør kjøring av tjenester og interaksjon.

3.4.1 Valg av webtjener

Webtjener ble valgt etter å ha sammenlignet flere faktorer ved Apache og Nginx som er to av de mest kjente webtjenerne på markedet. Ytelsestesting utført av *Speedemy* [45] viste at Nginx kan levere data til dobbelt så mange, med mindre minneforbruk.

Valgt tjenertype VPS medfører at det er begrensede ressurser tilgjengelig og de må brukes så effektivt som mulig. Derfor ble Nginx valgt og med en versjon som understøttet protokollen HTTP/2 for parallelle overføringer, samt HTTPS for kryptert kommunikasjon.

Når det gjelder skriptspråk på tjenestebussen var naturlig å velge *PHP: Hypertext Preprocessor (PHP)* siden dette er skriptspråket som de fleste kjører på Linux for webtjenester. Den dagsaktuelle versjonen PHP7 lovet betraktelige ytelsesforbedringer, opp mot dobbel hastighet sammenlignet med eldre PHP-versjoner ifølge en artikkel hos DO [46]. PHP ble derfor valgt i form av *PHP7-FPM (Fast Processing Module)*, som er ytterligere optimalisert for ytelse.

3.4.2 Valg av mellomlagring

Tjenestebussen må mellomlagre data for å understøtte kravene til YR, hvor definert levetid av data er 20 minutter. Verdien av mellomlagringen er primært å gi god ytelse og avlaste tjeneren til YR og NILU. Hvis data går tapt i mellomlageret vil dette hentes ved neste forespørsel gjennom tjenestebussen, noe som betyr at tap av data i denne sammenhengen ikke er kritisk. Med andre ord trenger man ikke persistent lagring.

Valget av mellomlagring baseres på en mest mulig kostnadseffektiv lagringsstrategi i form av ytelse og mulighet for skalering sett i sammenheng med pris. Valg måtte utføres basert på valgt operativsystem i kapittel 3.2.3 (Ubuntu) og valgt tjenertype i kapittel 3.1 (VPS).

Mellomlagring på minnebaserte tjenester eksisterer i flere former og det kan være vanskelig å gjøre et valg når det eksisterer så mange tjenester at man ikke har tidsvindu til å teste alle. Funksjonene de tilbyr er svært like, derfor er konsulenten Kristof Kovacs sin sammenligning [47] nyttig. Hovedstyrkene til Redis sammenfaller best med tjenestebussens behov for hurtig mellomlagring i primærminnet.

Redis ble etter en helhetlig vurdering valgt for mellomlagringen, altså en NoSQL løsning.

3.4.3 Komprimering

Komprimeringen ble konfigurert i webtjeneren på en slik måte at også andre ressurser kan komprimeres, uten at man må inn i hvert PHP-skript. Komprimeringen foregår gjennom bruken av *gzip* som er en standard metode i HTTP-protokollen forklart av *IETF RFC6713* [48].

3.4.4 Kryptering

Tjenestebussen hadde behov for minst 3 sertifikater som i utgangspunktet så ut til å bli en vesentlig kostnadsdriver i prosjektet. Derfor ble letsencrypt [49] valgt av den enkle årsaken at de tilbyr gratis sertifisering og fremstår som en innovativ aktør i sterk vekst.

3.4.5 Persistent lagring

MySQL database ble valgt basert på at dette er en databaseteknologi som er kjent for å være stabil på Linux, samt det faktum at undertegnede har hatt vesentlig erfaring med MySQL gjennom studier og over 11 år i arbeidslivet, så vel som på fritiden.

MySQL 5.7+ måtte velges med tanke på støtte for spatiale utvidelser [50] som muliggjør avstandssøk i en databaseindeks. Det er mulig å kjøre avstandssøk uten indekser ved hjelp av den såkalte haversine formelen [51], men det krever mer minne og CPU-tid som derfor vil øke responstiden til tjenestebussen. Databasemotoren i MySQL ble definert som MyISAM med tegnsett UTF-8 MB4 for å understøtte alle tegnene i kallene som måtte kjøres mot YR sitt applikasjonstjenestegrensesnitt.

YR sine kall kjøres mot geografiske lokasjoner og inneholder derfor tegn som ligger utenfor det vanlige UTF-8 tegnsettet til MyISAM, men UTF8 MB4 dekker alle tegnene som behøves.

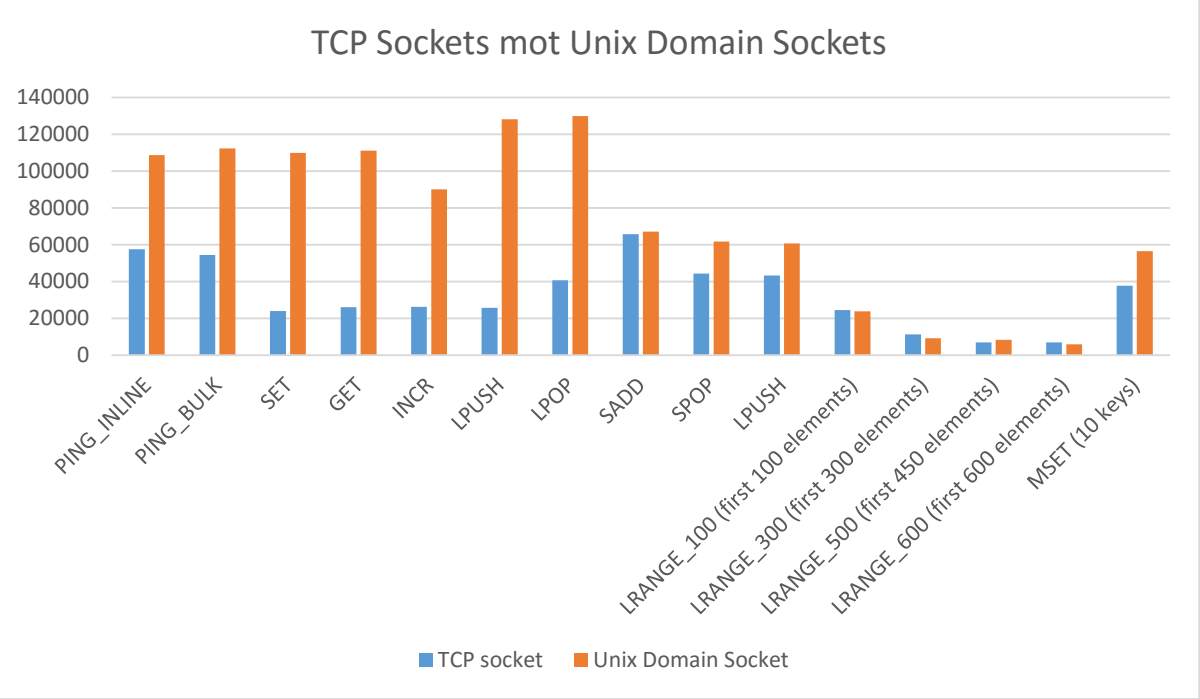
Med valgt relasjonsdatabase er stakken definert som LEMP (Linux, Nginx, MySQL og PHP).

3.4.6 Valg av sikkerhet og kommunikasjon

Brannmuren UFW fulgte linuxdistribusjonen og ble satt opp med portene som var nødvendig å åpne (443 for HTTPS, 80 for HTTP og 22 for SSH). Tjenestebussen kjører på egen bruker som har egen gruppe med rettigheter som er innsnevret til eget område med egen brukergruppe. ClamAV ble valgt for antivirus, av den enkle årsaken at den anbefales av Ubuntu [52], i tillegg til at egne efaringer med ClamAV tilsier den er lett å sette opp og drifte.

Pålogging mot rootbruker ble deaktivert slik at man må logge inn gjennom en annen bruker. Dette gjør at det ikke er mulig å automatisere angrep mot root-bruker. Fail2ban [53] vil overvåke loggene og blokkere kjente angrep som eksempelvis kan være gjentatte forsøk på root-pålogging, DOS-angrep eller andre kjente angrepssignaturer.

Før det ble valgt kommunikasjon for mellomlagring mot øvrige komponenter i tjenestebussen ble ytelsestester kjørt som det fremkommer av figur 6.



Figur 6 Komparativ sammenligning av ytelsen til TCP sockets og Unix Domain Sockets i NoSQL Redis

Ved at tjenestebussen primært vil bruke SET og GET på nøkkel/verdi vil Unix Domain Sockets gi en vesentlig økning i ytelse fremfor TCP Sockets. Denne implementeringen ble derfor valgt.

3.5 Valg av applikasjonsutvikling

Applikasjonen ble valgt utviklet med *RecylerView* komponenten [30] som gir god ytelse for flere datakilder ved at grensesnitt *View* kan gjenbrukes. Dette foregår ved hjelp av et seperasjonsprinsipp mellom grensesnitt og data gjennom en *Controller*.

Alle kall mot tjenestebussen ble valgt å gjøre asynkrone i henhold til kapittel 2.4 for å hindre at grensesnittet låser seg. Dette gjelder også tilbakemeldingsfunksjonaliteten for vær- og luftkvalitet. Protokollen HTTPS benyttes for sikker kommunikasjon mot tjenestebussen i henhold til tidligere krav.

XML tolking ble valgt å gjøre gjennom XML Pull Parser som anbefalt av Google [54].

Seperasjonsprinsippet ble etterlevd ved å benytte seg av grensesnitt basert på *View* og *ViewHolders* som beskrevet i Google sin utviklermanual for *RecyclerView* [31] som holdt data og variabler. Dette gjør at applikasjonen er bygget på en slik måte at den kan håndtere mange datakilder og effektivt bla i dataene uten at det vil oppleves som tregt.

Grensesnittene er valgt å utvikles etter Google sine Material Design prinsipper [28] implementert med støttebiblioteker for kompatibilitet bakover til Android 2.3.

3.6 Valg av utviklingsmetode

Av de tre typene prototyping faller throwaway prototyping bort ved at ett av effektmålene i forstudierapporten [7], kapittel 3.1 er *«Nye markeder og kanaler gjennom applikasjonsutvikling»*. Hvis man hadde valgt throwaway ville resultatet blitt kastet. Dette på grunn av at kvaliteten ville blitt vesentlig redusert ved at tiden brukt i utvikling hadde vært den primære faktoren. Inkrementell prototyping faller også bort grunnet at å utvikle på flere prototyper som skal slås sammen vil være vanskelig og tidkrevende å håndtere for en person.

Extreme prototyping er primært brukt for webapplikasjoner og kunne vært brukt for oppgaven hvis flere faktorer hadde vært kjent. Da ville man i praksis først bygd opp grensesnittet, deretter emulert svar fra tjenestebussen og så laget tjenestebussen og implementert API-er.

Prosjektets kompleksitet hvor alt i praksis var ukjent, måtte angripes fra motsatt hold, og med metoden evolusjonær prototyping kunne man ha fokus på best mulig kvalitet i gjennomførelse og inkrementelle endringer underveis.

Aktiviteter ble i stor grad gjennomført i henhold til følgende plan

- Problemanalyse hvor man skal forstå problemene som skal løses

- Kravarbeid hvor kravene til systemet defineres
- Utforming definerer systemarkitektur, moduler og komponenter
- Implementasjon hvor systemet realiseres
- Testing for å validere at systemet fungerer etter kravspesifikasjon
- Innføring som er å ta systemet i bruk
- Drift som vil være normal bruk av systemet

Ved at oppgaven innehar mange komponenter som skal utvikles av en person, og de respektive komponentene må fungere sammen, er valgt utviklingsmetode hensiktsmessig. Dette gir kontinuitet på delleveranser og mulighet for å justere seg inn underveis i prosjektet, etter hvert som det skulle vise seg å være nødvendig for ønsket måloppnåelse.

4. Resultater

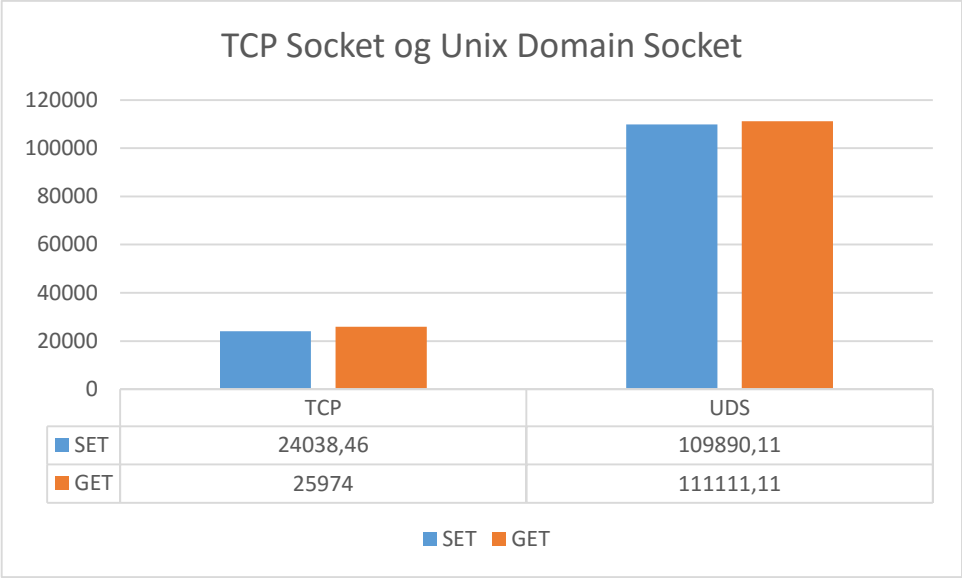
Kapittelet tar for seg resultatene prosjektets gjennomføring har ført til.

4.1 Vitenskapelige resultater

Dette kapittelet vil gå gjennom underlagene til problemstillingen fra kapittel 1. Empiri fra tidligere kapitler som var nødvendig for valg av tjenerlokasjon og annet vil i stor grad ikke være hensiktsmessig å repetere i kapittel 4 i sin helhet.

4.1.1 Data / Empiri av tjenestebussens ytelsestesting

Intern ytelsestesting som vist i kapittel 3.4.6, figur 6 er i figur 7 filtrert til å bare inkludere GET og SET som er funksjonene som brukes av tjenestebussen for mellomlagring. Testene ble kjørt 3 ganger og snittresultatene ble notert.



Figur 7 - Ytelsestesting av TCP mot UDS i tjenestebussen

Det å drive intern ytelsestesting er en del av den evolusjonære prototypmetoden som på denne måten alltid vil prøve å utelukke og optimalisere de svakeste leddene.

4.1.2 Data / Empiri av applikasjonens ytelsestesting

Testing av ytelse på hele tjenestebussen medregnet applikasjonen og GPS-posisjonering i sin helhet ble gjort på Sony Z5 Compact som var en av de raskeste telefonene i 2015 og er fremdeles er kapabel i 2016. Testene ble utført 3 ganger og snittet ble notert.

Test	Mellomlager	Tid
1 Første start av applikasjon, henting av data	Ingen	5,4 sekunder
2 Hente ny data		3,5 sekunder
3 Første start av applikasjon, henting av data	Tjenestebussen (Redis)	2,1 sekunder
4 Hente ny data	Tjenestebussen (Redis)	0.002 sekund
5 Hente ny data	Tjenestebussen (Redis) Lokalt mellomlager	0,001 sekund

Ved å ta resultatet fra test 1 og trekke fra resultatet av test 3, ser vi at tjenestebussens mellomlager er verdt rundt 3,3 sekunder i tidsbesparelse for 2 datakilder.

Det vil aldri være like tall hvis man tester på ulike enheter, med lavt batteri, dårlig GPS-dekning eller på tregt nettverk. Men selv om usikkerheten rundt testingen rår så viser det at man har høy effekt av å bruke tjenestebussen, ikke minst avlaster den også tjenestetilbyderne.

4.1.3 Design av database

Tjenestebussen er bygd opp for å fungere til dels agnostisk over data som er i kildene, ved at den behandler svar fra API-er som en nyttelast som returneres til Android-applikasjonen.

For YR og NILU som er i prototypen til kontekst er det ≈ 40.000 datapunkter over hele verden, som viser datatetthet. I tillegg vil brukerskapt data også berike tjenestebussen.

For å håndtere dette benytter tjenestebussen seg av en database med flere tabeller for oppslag mot API-er og tilbakemeldingsfunksjoner. Ved leveranse av oppgave er det 4 tabeller.

Tabelldesignet er laget utelukkende for maksimal ytelse. Hvor stor plass det tar i databasen er ikke viktig siden data hentes fra tjenestetilbyderne og mellomlagres i Redis. Det viktigste er derfor hurtige spørringer mot geometriske indekser i relasjonsdatabasen MySQL.

4.1.4 Design av tjenestebussen

Tjenestebussen legges ved prosjektet ved leveranse og vil ikke vises i sin helhet her, men funksjonen for å hente riktig værkilde vises her, ut av sammenheng i figur 8.

```
function getYr($redis, $pdo, $latitude, $longitude, $lng="en") {

// prepare SQL
$sth = $pdo->prepare("SELECT *, (ST_Distance(yr.geometry, ST_GeomFromText(:point, 4326)) * 111195) as dist from yr order by dist ASC LIMIT 0,1;");

// bind variables
$sth->execute(array( ':point' => "POINT($latitude $longitude)") );

$row = $sth->fetch(PDO::FETCH_ASSOC);
// for debug $sth->debugDumpParams();


// Code for finding correct weather URL based on language input
switch ($lng) {
        case "nb":
                $myWeather = $row['api_nb'];
        break;
        case "nn":
                $myWeather = $row['api_nn'];
        break;
        default:
                $myWeather = $row['api_en'];
        break;
}

// return weather
return getDataFromWS($redis, $myWeather, 600);
}
```

Figur 8 funksjon for å hente riktig værkilde fra YR

Når applikasjonen ber tjenestebussen om data vil koden først sjekke hvilken kilde man ber om, deretter vaskes inndata og sendes videre til riktig funksjon. I figur 8 vises det å hente riktig datakilde for YR. Mellomlagringstjenesten Redis sendes også med som et objekt inn i funksjonen. Instansen av Redis er opprettet utenfor funksjonens virkeområde slik at instansen kan gjenbrukes i de forskjellige funksjonene for maksimal ytelse.

Som det fremkommer av figur 8 er det en relativt enkel funksjon med en GEO-spatial spørring mot den geometriske kolonnen i tabellen til YR. En tilsvarende eksisterer for luftkvalitet.

Etter den har funnet riktig målepunkt sjekker den hvilket språk applikasjonen ba om. Med informasjon nok til å kunne kontakte tjenestetilbyder, går resultatene sammen med Redis-instansen videre inn i funksjonen *getDataFromWS* som vises av figur 9.

```
function getDataFromWS($redis, $wsURL, $ttl = 600) {
        $rdData = "";

        // check if stored in redis
        if($redis->exists($wsURL)) {
                $rdData = $redis->get($wsURL);
        }

        // get from source
        else {
                // Get data via CURL
                for ($i=0;$i<=10;$i++){
                    $ch = curl_init();
                    curl_setopt($ch, CURLOPT_URL, $wsURL);
                    curl_setopt($ch, CURLOPT_HEADER, 0);
                    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
                    $rdData = curl_exec($ch);
                }

                // set data to key
                $redis->set($wsURL, $rdData, Array('nx', 'ex'=>$ttl));
        }

        // return data
        return $rdData;
}
```

Figur 9 Tjenestebussens logikk for mellomlagring og henting av data

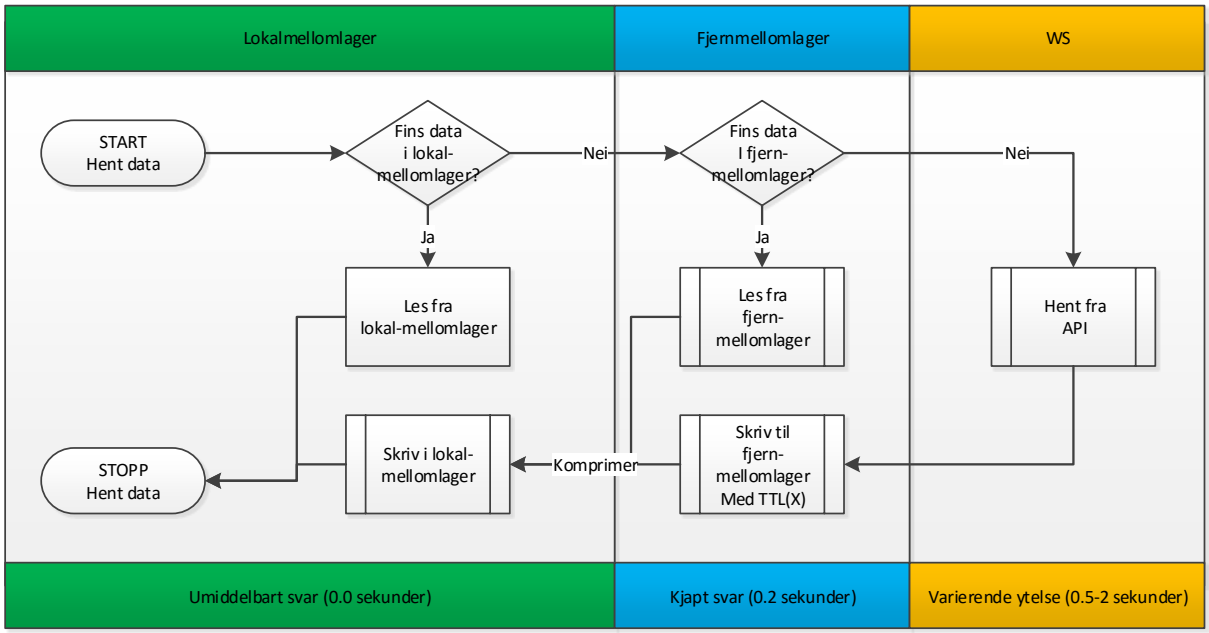
Som det fremkommer av figur 9 vil tjenestebussen se om data applikasjonen ønsker, eksisterer i mellomlageret Redis. Eksisterer data der, returneres data fra Redis. Hvis data ikke eksisterer vil tjenestebussen hente data og lagre denne i Redis med en levetid definert tidligere i tjenestebussen som 20 minutter.

Når dette er gjort, vil tjenestebussen returnere data oppover i logikken hvor data skrives ut av PHP-skriptet og på den måten returneres til applikasjonen som det fremkommer av figur 10.

```
case "yr":
    header("Content-type: text/xml");
    echo getYr($redis, $pdo, $_GET['latitude'], $_GET['longitude'], $_GET['lng']);
    break;
case "nilu":
    header("Content-type: text/xml");
    echo getNilu($redis,$pdo, $_GET['latitude'], $_GET['longitude']);
    break;
```

Figur 10 Data oversendes applikasjonen fra tjenestebussen med riktig hode

I praksis ser man på konseptet som vises av figur 11. Tjenestebussen opererer i den blå og oransje seksjonen av figuren. Den grønne seksjonen er lokalt i applikasjonen.



Figur 11 Prinsippet for mellomlagring i tjenestebussen

Styrken med tjenestebussens design er at den fungerer agnostisk overfor innholdet og databasen med geometriske spørringer vil finne riktige datakilder. Resultatene hentes enten fra mellomlageret, eller fra API-er. Det gjør at å legge til flere tjenere er svært enkelt.

4.1.5 Design av applikasjonen «Kontekst»

Applikasjonen «Kontekst» er utformet med tanke på MVC-prinsippet hvor data og grensesnitt er separert gjennom en kontroller. Ut over dette er også applikasjonen designet for å bruke flere tråder i form av asynkrone tråder for henting av data, så vel som sending av data.

De grafiske grensesnittene er basert på kort som Google kaller Material Design Cards og etterstreber prinsippene til Material Design gjennom alle sine grensesnitt. Ved at man etterstreber de premissene ovenfor, er applikasjonen også tilgjengelig i flere språk som velges automatisk av applikasjonen basert på enhetens innstillinger:

Tilgjengelige språk i applikasjonen				
Norsk nynorsk	Norsk bokmål	Engelsk	Dansk	Svensk

Oversettelse av applikasjonen er med dette abstraksjonsprinsippet vesentlig forenklet.

Applikasjonens klasser og beskrivelse av dem er vedlagt i systemdokumentasjon [55].

4.2 Ingeniørfaglige resultater

De ingeniørfaglige resultatene tar for seg målene som ble definert under prosjektets oppstart som beskrevet i forstudierapporten [7]. Status for hvert av målene ved leveringstidspunkt er beskrevet i underkapitlene.

4.2.1 Effektmål

Effektmål	Status
Økt fokus på luftkvalitet	Prosjektet omtales ofte i egne kretser og oppgaven produserer en applikasjon som viser vær og luftkvalitet er dette punktet realisert. En ytterligere gevinstrealisering kan forekomme hvis applikasjonen lanseres i Google sin nettbutikk.
Nye markeder og kanaler gjennom applikasjonsutvikling	Prosjektet har tilført erfaring og kompetanse samt gjenbrukbar kode som kan bygges videre på er dette målet nådd.
Økt Goodwill ved å lansere en samfunnsnyttig tjeneste	Goodwill er økt i nære kretser i Bergen, men en bredere høsting av dette vil være mulig ved å lansere applikasjonen i applikasjonsbutikken til Google.
Økt kompetanse ved å tilegne erfaring rundt bruk, konsumering og sammenstilling av tredjeparts datakilder, mellomlagringsteknikk og applikasjonsutvikling	Status er at prosjektet er gjennomført med effektmålene knyttet til tilegning av erfaring, og dermed er dette målet realisert.

4.2.2 Resultatmål

Resultatmål	Status
Kontekst – en kontekstavhengig Android applikasjon	Utviklet prototype, ønsket resultat oppnådd.
Prosjektrapport	Leveres i hovedleveranse innen 25. mai.
Vedlegg til rapporter	Utført, vedlegg til hovedleveransen.
Forstudierapport	
Kravdokument	
Konseptskisse av løsningen	
Fremdriftsplan	Utarbeidet i prosjektets start og revidert noe underveis med nye aktiviteter. Brukt aktivt mot timeføring. Det er brukt gantdiagram.

Timelister	Timelister er ført hver dag og summert ukentlig i statusrapporter som knyttes til aktivitetene i gantdiagrammet.
Møtereferater	Referater fra møter er vedlagt hoved-leveransen.
Kontrakt	Lagt inn i kravdokumentet [8] under kapittel 4.5, leveres som vedlegg til hoved-leveransen.
Dokumentasjon i kildekode	Dokumentasjon i kildekode er fyldig utført og leveres som en del av kildekoden i hovedleveransen.
Dokumentasjon av grensesnitt	Bruksanvisning vedlagt hoved-leveransen.
Video av applikasjon i bruk	Vedlagt hovedleveransen og tilgjengelig på youtube [56].

4.2.3 Prosessmål

Prosessmål	Status
Kompetansebygging	Bred kompetanse er tilegnet under prosjektets gjennomføring, særlig innen utvikling av Android applikasjoner samt forskjellige måter å lansere en tjenestearkitektur for mobil.
Nettverksbygging	Det ble utført nettverksbygging i miljø for IT-spesialister ved å delta på seminar for mellomlagring hvor erfaringsutveksling og tilbakemeldinger på konseptet ble gitt.
Levere et godt produkt	Produktet fungerer i henhold til kravene og fremstår som hurtig i bruk, med pent og moderne grensesnitt. Øvrige leveranser er gjennomført uten å ta snarveier. Maler og metoder er brukt.

4.2.4 Status på systemet ved leveranse

Utvikling er gjennomført i henhold til kravene og fungerer som tiltenkt. Alle leveranser er gjort i henhold til planer. Status på prosjektet i sin helhet er at alt er utført med et fokus på kvalitet under den evolusjonære prototypingen.

Statusen er altså at det er en velfungerende prototype og alle leveranser er realisert.

4.3. Administrative resultater

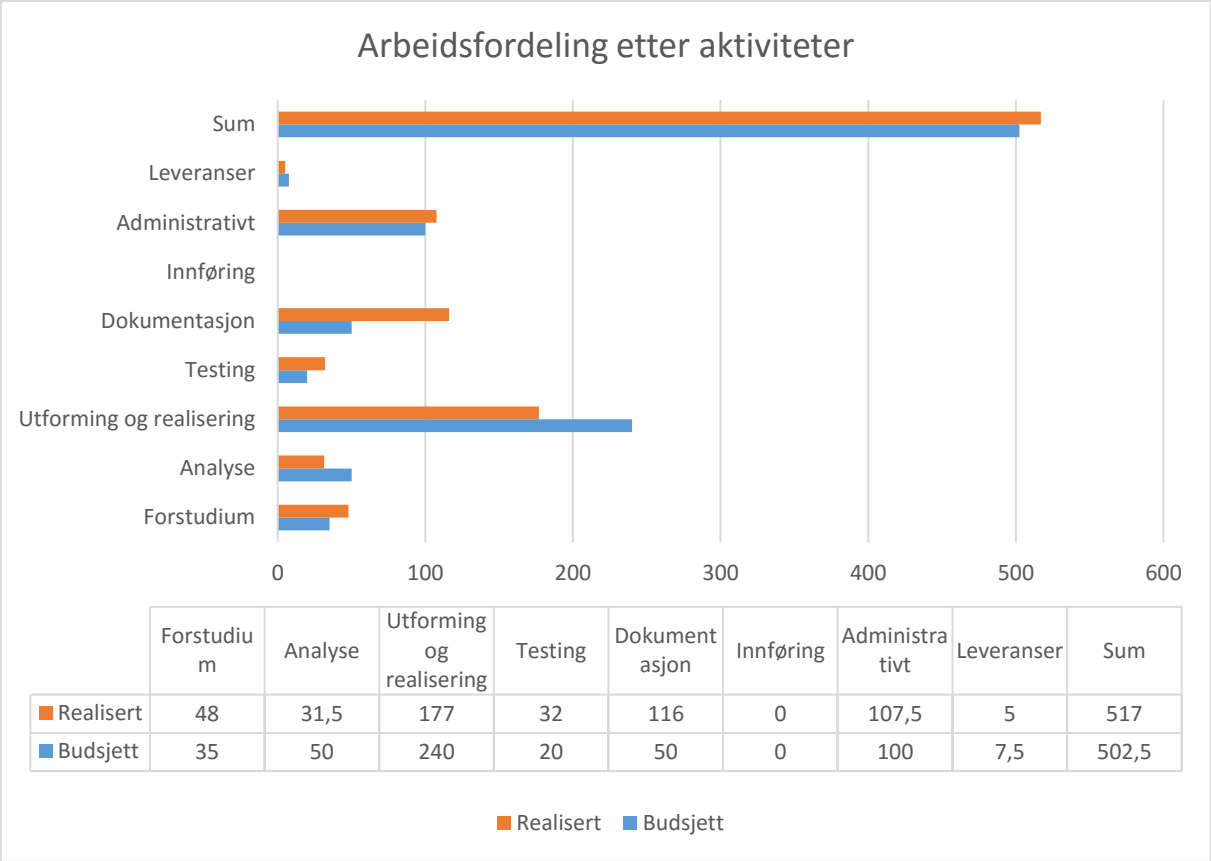
De administrative resultatene fremkommer av dette kapittelets respektive underkapitler.

4.3.1 Prosjekthåndboken

- Se vedlagt prosjekthåndbok

4.3.2 Måloppnåelse i forhold til fremdriftsplan

Prosjektets fremdriftsplan ble laget i gant i Microsoft Project. Vurdering av måloppnåelse grupperes på hovedpunktene for å vurdere fremdriftsplan i henhold til virkeligheten i figur 12.



	Forstudium	Analyse	Utforming og realisering	Testing	Dokumentasjon	Innføring	Administrativt	Leveranser	Sum
Realisert	48	31,5	177	32	116	0	107,5	5	517
Budsjett	35	50	240	20	50	0	100	7,5	502,5

Figur 12 Fremdriftsplan i forhold til virkeligheten gruppert på hovedaktiviteter

Detaljert oversikt over aktiviteter er vedlagt i gantdiagram, samt prosjekthåndboken og timeliste med detaljert beskrivelse av aktiviteter ned på dagsnivå.

Det totale timeavviket ble for prosjektet 14,5 timers overskridelse. Det ble stipulert mer tid på utforming og realisering og mindre på dokumentasjon. I ettertid er det lett å se at dokumentasjon ble sprengt, men årsaken for at det totale budsjettet ble mindre overskredet var at fokus ble skjøvet over på å fullføre dokumentasjon siste måneden, etter prototypen fungerte. Analyse tok mindre tid enn forventet, mest sannsynlig på grunn av godt forstudium som derfor belyste mye av det man trengte å vite.

Kapittel 5: Diskusjon

5.1 Årsaker til at resultatene ble som de ble

Det er ikke nok å angripe en slik oppgave med timer alene. Man må jobbe strukturert og målrettet. De ukjente faktorene i starten av prosjektet medførte kartleggingsbehov og

vurdering av hvilke teknologier som skulle vurderes, samt hvordan de kunne implementeres for å i best mulig grad understøtte kravene i oppgaven.

Det å følge utviklingsmodellen, ha kontroll og orden på filområder, bruke maler, kode etter god skikk, lage databaser med tanke på ytelse, finne, vurdere og implementere teori, lage modeller og grafiske elementer, grensesnitt og andre komponenter er alle faktorer som må gjøres mest mulig strukturert og med et åpent sinn for å få et godt resultat.

Man kan sånn sett si at resultatene ble som de ble ved å jobbe strukturert, jobbe jevnlig, selvoppofrelse når det gjaldt sosiale hendelser men samtidig tillate seg selv litt fri innimellom.

Effektiv kommunikasjon med faglærer var nok også en medvirkende faktor for resultatet.

5.1.2 Hvorfor holdt hypotesene

Hypoteser rundt mellomlagringsprinsippet viste seg å fungere i henhold til forventet effekt.

Interne ytelsestester i tjenestebussen og valgene som hele tiden ble gjort med tanke på ytelse har realisert en tjenestebuss som fungerer etter teorien. Tjenestebussen forenklet også utviklingen. Implementeringen av luftkvalitet var lite krevende når arkitekturen med løs kobling var til stede.

Det var altså vesentlig at jeg ikke bare vurderte hvilken programvare jeg skulle bruke, men også hvordan programvaren skulle snakke sammen, videre at jeg lagde databasetabeller for best mulig ytelse med GEO-spatiale indekser og en mest mulig effektiv lokasjon for tjeneren, samt en tjener med SSD-lagring og annet som ble valgt i kapittel 3. Ved å utføre ytelsestester på hver komponent, hver for seg, maksimerte jeg sannsynligheten for en rask tjenestebuss.

Testing på flere enheter (4 telefoner, 1 nettbrett og 1 emulert telefon) var nok også en medvirkende faktor til at alt fungerte. Virtuelle enheter medfører en forenkling når man vil teste forskjellige konfigurasjoner, som var en stor fordel for funksjonstesting.

5.1.3 Drøft hvordan resultatene kan forstås i forhold til problemstillingen

Problemstillingen «*Hvordan kan man gjennom brukerens kontekst vise data som er relevant for brukeren, ved hjelp av en mobilapplikasjon?*» besvares gjennom resultatene ved at prototypen og tjenestebussen fungerer som tiltenkt og viser relevante data i henhold til konteksten.

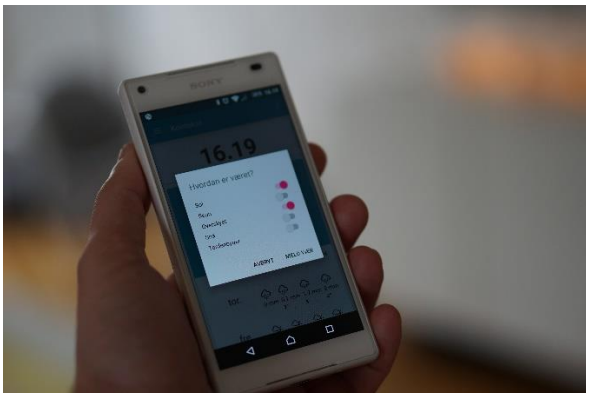
Resultatet er derfor oppnådd med de to kildene som skulle være med i prototypen: vær og luftkvalitet, men det er et bevis på mye mer enn vær og luftkvalitet – altså en infrastruktur

som muliggjør homogene kall mot heterogene datakilder for så å filtrere ut de datakildene som er relevant for brukeren, før dette vises i applikasjonen.

5.1.4 Hvordan ble sluttproduktet

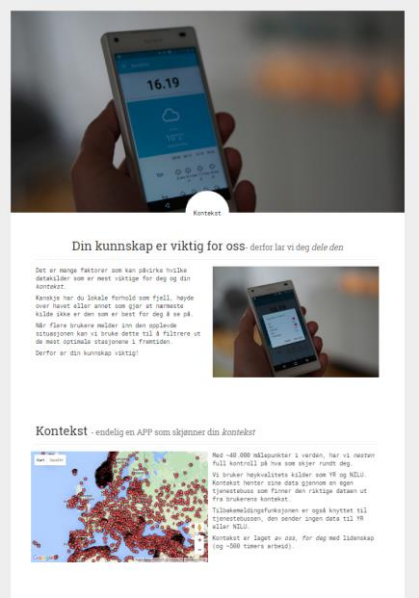
Applikasjonen henter relevante data som fremkommer av figur 13 og muliggjør berikelse av data som fremkommer av figur 14 hvor brukeren rapporterer inn opplevd værfenomen.

Figur 13 Visning av vær i brukerens kontekst



Figur 14 Rapportering av værfenomen i brukerens kontekst

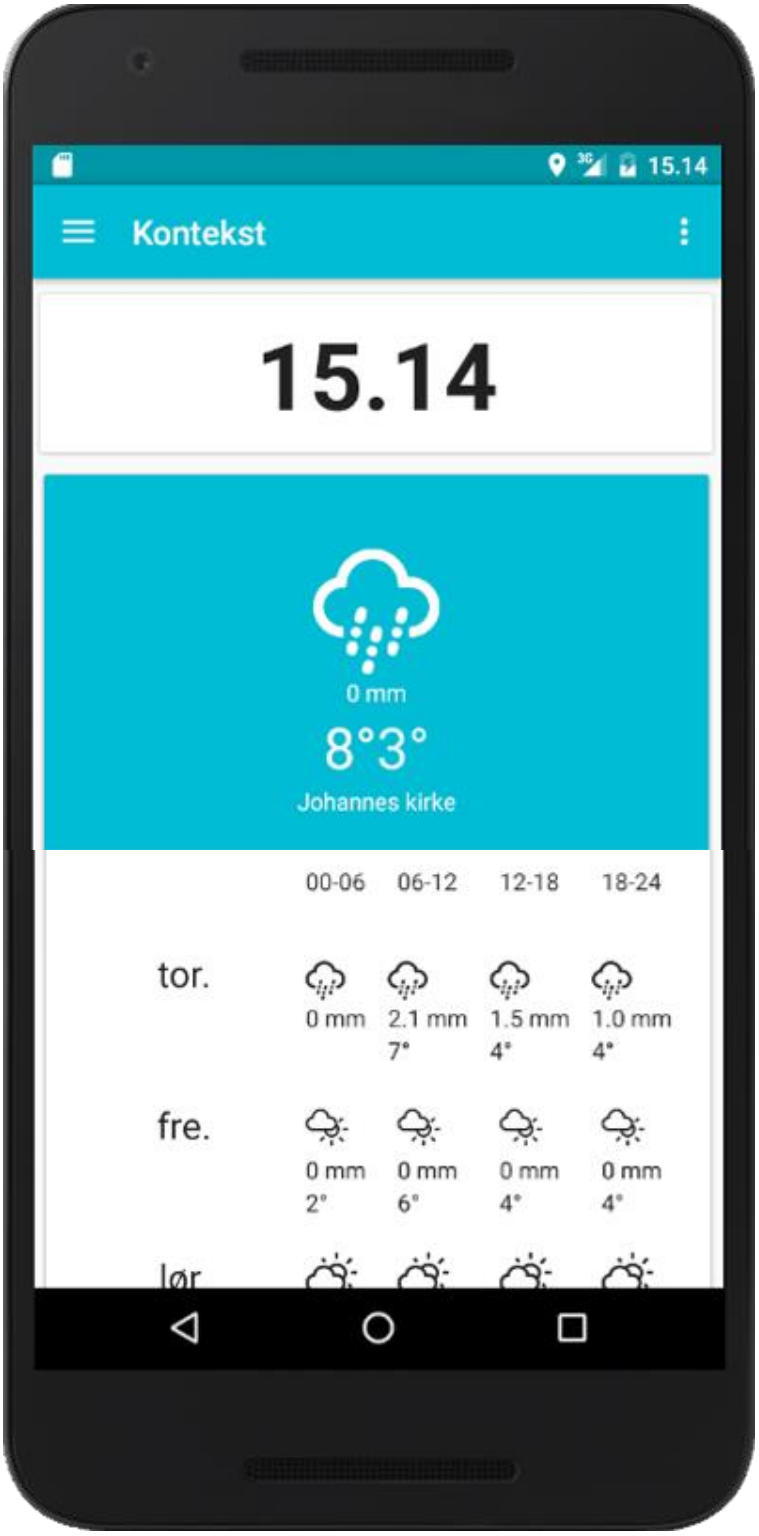
Det var ingen krav om hvordan data skulle vises, med unntak av skulle å etterstrebe material design prinsippene med rene, minimalistiske designelementer som vises effektuert.



Figur 15 Applikasjonens landingsside

Landingssiden vist av figur 15 fungerer som en interaktiv reklameplakat for applikasjonen med kart som viser målepunkter, lettfattelig informasjon om applikasjonen og demonstrasjonsvideo.
Siden er utviklet i *Adobe Brackets* basert på det responsive rammeverket *Bootstrap*, javascriptbiblioteket *JQuery* og kartløsningen *Google Maps* mot Google sine *Fusion Tables*.

Landingssiden nås fra applikasjonen, som beskrevet i brukerdokumentasjonen [57].



Figur 16 Applikasjonens grensesnitt

Applikasjonens grensesnitt benytter seg av material design prinsippene med kort. Øverst er det et kort for klokke etterfulgt av vær som fremkommer av figur 16. Brukeren blar forbi været for å vise luftkvaliteten i eget kort.

De grafiske elementene brukt for værfenomener er hentet fra Erik Flowers på github [58] for å understøtte Material Design prinsippene til Google.

Brukeren kan klikke på kortene for å melde tilbake, brukeren kan også rotere telefonen eller trekke ned listen for å spørre tjenestebussen om nyere data eksisterer.

5.1.5 Fikk oppdragsgiver det som var forventet

Oppdragsgiver fikk det som var forventet ved gjennomføringen av prosjektet, ved at resultatene og konseptet fungerer etter kravene.

5.1.6 Hvilke krav ble oppfylt

Alle krav ble oppfylt, se respektive kapittel 4.2.1, 4.2.2, 4.2.3 og 4.2.4. Kravet om kart ble valgt å løse på landingssiden fremfor i applikasjonen ved at landingssiden fungerer som en reklameplakat for applikasjonen.

5.1.7 Hva var bra

Gjennomføringen av prosjektet gav kompetanse innen utvikling av Android applikasjon med tilhørende tjenestebuss, samt nye kanaler for markedsføring av tjenester.

Samarbeid med faglærer fungerte godt, og bruk av maler, rapporter og filområde fungerte bra. Kommunikasjonskanalen Lync Professional fungerte også bra, med unntak av ett nettmøte hvor vi måtte bruke Google Hangouts.

Det at jeg testet applikasjonen på 3 mobiltelefoner, 1 nettbrett og 1 virtuell mobiltelefon. Testet applikasjonen forskjellige steder i Bergen, både å hente ut data og sende inn data gav en god måte å avdekke potensielle feilkilder på. Fra emulatoren kunne jeg også teste steder utenfor Bergen ved å sette lokasjon i emulatoren.

Det å kunne logge seg på Redis konsollet og på redis-cli monitor gjorde at man enkelt kunne følge med på monitor hva som var og hva som ble mellomlagret, samt hvilke kommandoer som ble kjørt. I tillegg kunne man da tømme mellomlageret i redis-cli ved hjelp av flushall. Dette viste seg å vesentlig forenkle testing, ved at man da effektivt sett kunne omgå mellomlageret i tjenestebussen når det var behov for dette.

5.1.8 Hva var ikke så bra

Den største utfordringen var det å jobbe alene som en student over fjernundervisning. Man har ikke et team man kan snakke med om ideer, kvalitetssikre hverandres arbeid og utfylle hverandres egenskaper slik et velfungerende team gjør.

Selv om jeg har følt litt på dette som sikkert alle studenter gjør, så vet jeg med meg selv at jeg har gjort så god innsats som jeg kunne gjøre. Jeg har stått på ved siden av fulltidsjobb og styreoppgaver i sameiet.

5.1.9 Hva ble bra på grunn av valgt prosess, fremgangsmåte og teknologi

Tilnærmingen evolusjonær protyping satset på et godt resultat som kan gjenbrukes til en endelig kode, i motsetning til de øvrige metodene som var vurdert i kapittel 3.6. Prosessen sikret en god gjennomføring av prosjektet ved at man fikk en strukturert og helhetlig gjennomføring med riktig nivå av dokumentasjon.

Teknologivalg ble gjort uten å være forutinntatt, men ble gjort med fokus på hver hovedkomponent for å velge de komponentene som på best mulig måte understøttet kravene om ytelse og interoperabilitet i tjenestebussen.

Krypteringen ble gjennom valgt utviklingsmetode optimalisert frem til resultatet ble en A+ sertifisering som er det beste resultatet man kan få til. Mozilla sin konfigurasjonsgenerator for SSL [59] ble brukt for å lage konfigurasjonen, og Qualys SSL Labs sitt testverktøy [60] for å validere kvaliteten på krypteringen. Dette ble repetert i iterasjoner frem til sertifikatene hadde A+ sertifisering.

Valget av Redis som mellomlager med mulighet for å definere levetid når data mellomlagres medførte at man ikke trenger å lage ryddeskript.

5.1.10 Hva ble ikke bra på grunn av valgt prosess, fremgangsmåte og teknologi

På grunn av at undertegnede har en asynkron nettlinje (100 / 20 mbit) var det ikke mulig å teste opp-kapasitet til de forskjellige tjenerlokasjonene. Men det skal sies at det er synkrone fiberlinjer hos lokasjonene og antageligvis vil det være mye den samme kapasiteten ned.

5.1.11 Hva ble bra eller dårlig uavhengig av valgt prosess, fremgangsmåte og teknologi

Noe som var utslagsgivende var deltakelsen på et BitShift mellomlagringsseminar i Bergen 27/01-2016 [61] som omhandlet hvordan NRK bruker Redis for mellomlagring. Konseptet å kunne vokse i skyen med slike tjenester var noe som ble relevant for valget.

Etter seminaret opprettet jeg dialog med foredragsholderen Harald S. Ulriksen, og fikk gode tilbakemeldinger som forenklet konseptet mitt for lokalmellomlager. Jeg hadde opprinnelig laget en overkomplisert lokalmellomlagringsløsning for de mobile enhetene, men Harald kom med innspillet om at jeg burde bruke HTTP-mellomlager på den mobile enheten.

6. Konklusjon og videre arbeid

6.1. Konklusjon

Det samme året som jeg ble født (i 1992), kom forfatteren John Naisbitt med en god spådom i boken sin «*Megatrends*» [62]: *«Drowning in Information But Starved for Knowledge»*.

Dette setter jeg i sammenheng med den analoge alderen som i større og større grad ble tatt over av den digitale informasjonsalderen. Jeg opplevde selv denne overgangen fra MS-DOS til Windows, og så overgangen fra Usenet til Internett og søkemotorer med positivt blikk.

Internett har gått fra å være statiske sider med kontaktinformasjon til å skape interaktivitet og en Web 2.0 som i større og større grad understøtter interaksjon og brukerskapt materiale. De senere årene har antall tilkoblede enheter som pumper data ut i skyløsninger økt betraktelig i omfang.

Konseptet til kontekst er en prototype applikasjon med en tilhørende tjenestebuss som skal vise data for brukerens kontekst. Dette er en måte å filtrere ut data som er relevant for å unngå å overlesse brukeren. Konseptet ville ikke vært mulig før mobiltelefoner ble smarttelefoner, før de fikk sensorer for posisjon, vedvarende oppkobling for nett og programmeringsspråk som er fullverdig med det man kan gjøre på vanlige datamaskiner.

Konseptet realisert i oppgaven kan brukes for mye mer enn luft- og værkvalitetskilder. Det meste i dag kan knyttes til kontekster og en tjenestebuss med konteksten i sentrum som kan skalere og bygges ut med små utgifter er i så måte en svært fleksibel tjenestebasert arkitektur.

Det samme konseptet kunne blitt brukt i en bil som gjennom sine sensorer registrerte at det var hull i veien og rapporterte det inn automatisk til vegvesenet, den kunne registrert at det var glatt for å rapportere det til bilen bak, man kunne laget en virtuell turguide, eller quiz knyttet til lokasjon.

Konklusjonen er altså at oppgaven og resultatene viser hvordan man kan vise data som er relevant for brukerens kontekst, om det skulle være sosiale medier, reklamekampanjer, ladestasjoner for elbiler, nærmeste offentlige toalett eller hva du har i kjøleskapet ditt.

Det er egentlig bare fantasien som setter grenser, samt tiden man har til rådighet.

Derfor vil jeg anbefale alle med pågangsmot og teknisk interesse til å enten forske eller drive utvikling innen kontekstuelle data, om det er som en produsent eller konsument.

6.2. Videre arbeid

Selv om applikasjonen «kontekst» i dag fungerer etter kravene kom det mange gode tilbakemeldinger fra venner og bekjente som så applikasjonen i bruk. Noen av forslagene bør implementeres før den lanseres i nettbutikken.

Forslag jeg fikk var blant annet en kollega som ville se flo og fjære på hytten, som er både en ny type datakilde og en ny funksjon: å legge til egne kontekster (hytte, hjem, jobb og andre).

Følgende videreutvikling foreslås

- Mulighet for å opprette egne kontekster i en ny liste - Hytten, ferien og annet.

    - Flere datakilder - Flo/fjære, sosiale medier, turguide
- Mulighet for å ta bilder og påføre de med data gyldig for konteksten
    - Eksempel flo/fjære, vær, luftkvalitet
- Deling mot sosiale medier
- Ikoner for luftkvalitet bør lages eller finne noen som kan brukes
- Teste og eventuelt utvikle tilpasset grensesnitt for flere enheter
    - Android TV, Android Auto, Android Wear
- Lage Widgets
- Utvide landingssiden med
    - Visning av heatmaps som viser hvor applikasjonen blir mest brukt.
    - Visning av vær og innrapporterte værfenomen
- Legge applikasjonen ut i Google sin nettbutikk

Referanser

[1] F. Svartdal, «Store Norske Leksikon,» 16 03 2016. [Internett]. Available: https://snl.no/prokrastinering. [Funnet 01 04 2016].

[2] A. Maslow, A Theory of Human Motivation, 1943.

[3] S. A. Horgen, «Youtube.com,» 28 01 2015. [Internett]. Available: https://www.youtube.com/watch?time_continue=1&v=5wcbPiNRvdc. [Funnet 15 01 2016].

[4] «friskby,» [Internett]. Available: http://www.friskby.no. [Funnet 28 3 2016].

[5] «luftkvalitet.info,» [Internett]. Available: http://www.luftkvalitet.info/home.aspx. [Funnet 01 02 2016].

[6] Fort Air Partnership, «Fort Air Partnership,» ukjent år. [Internett]. Available: http://www.fortair.org/articles/how-cold-weather-affects-air-quality/. [Funnet 30 01 2016].

[7] O. A. Mjelde, «Forstudierapport,» Bergen, 2016.

[8] O. A. Mjelde, «Kravdokumentet,» Bergen, 2016.

[9] YR.no, «Vilkår for bruk av gratis data frå Yr,» 15 9 2019. [Internett]. Available: http://om.yr.no/verdata/vilkar/. [Funnet 01 02 2016].

[10] «LEMP Stack Resources and Q&A,» [Internett]. Available: http://www.lemp.io. [Funnet 15 02 2016].

[11] S. Øyvann, «Computerworld,» 11 11 2015. [Internett]. Available: http://www.cw.no/artikkel/tingenes-internett/64-milliarder-ting-til-neste-ar. [Funnet 01 05 2016].

[12] P. Krensky, «pentaho,» 02 2015. [Internett]. Available: http://www.pentaho.com/sites/default/files/uploads/resources/wp_data_manatement_internet_of_things.pdf. [Funnet 01 05 2016].

[13] «Parse,» 2016. [Internett]. Available: https://parse.com/. [Funnet 15 02 2016].

[14] J. Costine, «Tech Crunch,» 25 04 2013. [Internett]. Available: http://techcrunch.com/2013/04/25/facebook-parse/. [Funnet 30 02 2016].

[15] «Moving on,» 28 1 2016. [Internett]. Available: http://blog.parse.com/announcements/moving-on/. [Funnet 01 03 2016].

[16] DiFi, «Tjenestenivåavtale - SLA,» 11 04 2016. [Internett]. Available: http://www.anskaffelser.no/offentlige-anskaffelser-it/it-drift/sla-tjenesteavtale. [Funnet 02 05 2016].

[17] Axelos, «What is ITIL® Best Practice?,» [Internett]. Available: https://www.axelos.com/best-practice-solutions/itil/what-is-itil. [Funnet 02 05 2016].

[18] jboner, «Github - jboner/latency.txt,» 2012. [Internett]. Available: https://gist.github.com/jboner/2841832.

[19] S. Sakr, «XML compression techniques: A survey and comparison,» 14 08 2008. [Internett]. Available: https://www.researchgate.net/publication/222672490_XML_compression_techniques_A_survey_and_comparison. [Funnet 08 05 2016].

[20] «HTTP Over TLS,» The Internet Engineering Task Force (IETF®), [Internett]. Available: https://tools.ietf.org/html/rfc2818. [Funnet 01 02 2016].

[21] J. Ellingwood, «An Introduction to Securing your Linux VPS,» 04 05 2014. [Internett]. Available: https://www.digitalocean.com/community/tutorials/an-introduction-to-securing-your-linux-vps. [Funnet 02 02 2016].

[22] «Named Pipes vs. TCP/IP Sockets,» Microsoft, [Internett]. Available: https://technet.microsoft.com/en-us/library/aa178138(v=sql.80).aspx. [Funnet 08 05 2016].

[23] «unix - sockets for local interprocess communication,» Linux Programmer's Manual, 15 03 2016. [Internett]. Available: http://man7.org/linux/man-pages/man7/unix.7.html. [Funnet 01 04 2016].

[24] «Processes and Threads,» Google, [Internett]. Available: http://developer.android.com/guide/components/processes-and-threads.html. [Funnet 15 01 2016].

[25] «AsyncTask,» Google, 06 05 2016. [Internett]. Available: http://developer.android.com/reference/android/os/AsyncTask.html. [Funnet 11 05 2016].

[26] «Google Pagespeed Insights,» [Internett]. Available: https://developers.google.com/speed/pagespeed/insights/. [Funnet 01 01 2016].

[27] «Google Design,» Google, [Internett]. Available: https://design.google.com/. [Funnet 15 01 2016].

[28] «Material Design,» Google, [Internett]. Available: https://www.google.com/design/spec/material-design/introduction.html. [Funnet 10 01 2016].

[29] «List View,» Google, [Internett]. Available: http://developer.android.com/guide/topics/ui/layout/listview.html. [Funnet 20 01 2016].

[30] «RecyclerView,» Google Develoers, 04 05 2016. [Internett]. Available: https://developer.android.com/reference/android/support/v7/widget/RecyclerView.html. [Funnet 01 03 2016].

[31] «RecyclerView.ViewHolder,» Google Developers, 04 05 2016. [Internett]. Available: https://developer.android.com/reference/android/support/v7/widget/RecyclerView.ViewHolder.html. [Funnet 30 02 2016].

[32] «Data Binding Guide,» Google, [Internett]. Available: http://developer.android.com/tools/data-binding/guide.html. [Funnet 20 01 2016].

[33] «MVC - XEROX PARC 1978-79,» [Internett]. Available: https://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html. [Funnet 11 05 2016].

[34] «Extensible Markup Language (XML),» W3C, 19 05 2015. [Internett]. Available: https://www.w3.org/XML/. [Funnet 14 05 2016].

[35] «XMLReader,» Google, [Internett]. Available: http://developer.android.com/reference/org/xml/sax/XMLReader.html. [Funnet 01 02 2016].

[36] «DocumentBuilder,» Google, [Internett]. Available: http://developer.android.com/reference/javax/xml/parsers/DocumentBuilder.html. [Funnet 10 02 2016].

[37] «XmlPullParser,» Google, [Internett]. Available: http://developer.android.com/reference/org/xmlpull/v1/XmlPullParser.html. [Funnet 15 02 2016].

[38] «Types of prototyping,» Wikipedia, 04 02 2014. [Internett]. Available: https://en.wikipedia.org/wiki/Software_prototyping#Types_of_prototyping. [Funnet 03 02 2016].

[39] M. Jørgenrud, «Skyarkiv i utlandet er ulovlig,» 29 9 2014. [Internett]. Available: http://www.digi.no/bedriftsteknologi/2014/09/29/skyarkiv-i-utlandet-er-ulovlig. [Funnet 02 05 2016].

[40] J. Reed, «Comparing Cloud Compute Services,» 08 06 2014. [Internett]. Available: http://blog.cloudharmony.com/2014/07/comparing-cloud-compute-services.html. [Funnet 01 02 2016].

[41] J. Nielsen, «Website Response Times,» Nielsen Norman Group, 01 06 2010. [Internett]. Available: https://www.nngroup.com/articles/website-response-times/. [Funnet 05 02 2016].

[42] Amazon AWS, «Windows Server on AWS,» [Internett]. Available: https://cloud.google.com/compute/docs/instances/windows/creating-managing-windows-instances.

[43] Google Cloud Platform, «Creating and Managing Windows Instances,» 27 04 2016. [Internett]. Available: https://cloud.google.com/compute/docs/instances/windows/creating-managing-windows-instances. [Funnet 02 05 2016].

[44] D. C. Johnson, «Which Server Distro is Right for Me,» 29 3 2015. [Internett]. Available: http://www.linux.org/threads/which-server-distro-is-right-for-me.7783/. [Funnet 02 05 2016].

[45] A. Mikalauskas, «APACHE -VS- NGINX. 2015 EDITION,» Speedemy, 07 04 2015. [Internett]. Available: http://www.speedemy.com/apache-vs-nginx-2015/. [Funnet 20 02 2016].

[46] E. Heidi, «Getting Ready for PHP 7,» Digital Ocean, 15 07 2015. [Internett]. Available: https://www.digitalocean.com/company/blog/getting-ready-for-php-7/. [Funnet 02 03 2016].

[47] K. Kovacs, «Cassandra vs MongoDB vs CouchDB vs Redis vs Riak vs HBase vs Couchbase vs OrientDB vs Aerospike vs Neo4j vs Hypertable vs ElasticSearch vs Accumulo vs VoltDB vs Scalaris vs RethinkDB comparison,» [Internett]. Available: http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis. [Funnet 01 03 2016].

[48] Internet Engineering Task Force (IETF) , «The 'application/zlib' and 'application/gzip' Media Types,» 08 2012. [Internett]. Available: https://tools.ietf.org/html/rfc6713. [Funnet 08 05 2016].

[49] «Let's Encrypt,» [Internett]. Available: https://letsencrypt.org/. [Funnet 2016 02 02].

[50] «Extensions for Spatial Data,» MySQL, [Internett]. Available: http://dev.mysql.com/doc/refman/5.7/en/spatial-extensions.html. [Funnet 15 02 2016].

[51] «Haversine formula,» Wikipedia, 08 07 2013. [Internett]. Available: https://en.wikipedia.org/wiki/Haversine_formula. [Funnet 02 01 2016].

[52] «Antivirus,» Ubuntu, 31 12 2015. [Internett]. Available: https://help.ubuntu.com/community/Antivirus. [Funnet 08 05 2016].

[53] «Fail2Ban,» [Internett]. Available: http://www.fail2ban.org/wiki/index.php/Main_Page. [Funnet 01 05 2016].

[54] «Parsing XML Data,» Google Developer, [Internett]. Available: http://developer.android.com/training/basics/network-ops/xml.html. [Funnet 20 01 2016].

[55] O. A. Mjelde, «Systemdokumentasjon,» 2016.

[56] O. A. Mjelde, «YouTube - Prototypen "Kontekst",» 20 05 2016. [Internett]. Available: https://www.youtube.com/watch?v=g71QPqmID-k.

[57] O. A. Mjelde, «Brukerdokumentasjon».

[58] E. Flowers, «Weather Icons,» [Internett]. Available: http://erikflowers.github.io/weather-icons/. [Funnet 20 02 2016].

[59] «Mozilla SSL Configuration Generator,» Mozilla, [Internett]. Available: https://mozilla.github.io/server-side-tls/ssl-config-generator/. [Funnet 03 03 2016].

[60] Qualys SSL Labs, «SSL Server Test,» [Internett]. Available: https://www.ssllabs.com/ssltest/. [Funnet 15 03 2016].

[61] O. A. Mjelde, «Bitshift mellomlagringseminar,» Bergen, 2016.

[62] J. Naisbitt, Megatrends. Ten New Directions Transforming Our Lives., 1982: Warner Books.

[63] «YSlow,» [Internett]. Available: http://yslow.org/. [Funnet 01 01 2016].

[64] «ssdb.io,» [Internett]. Available: http://ssdb.io/. [Funnet 25 02 2016].

[65] «Google/Bing: Minimalism vs. Maximalism,» Technologizer, 12 03 2009. [Internett]. Available: http://www.technologizer.com/2009/12/03/googlebing-minimalism-vs-maximalism/. [Funnet 11 05 2016].

[66] «Material Design - Introduction,» Google, [Internett]. Available: https://www.google.com/design/spec/material-design/introduction.html. [Funnet 15 02 2016].

Vedlegg