Bacheloroppgave

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

Aleksandra Jekic

Stabilitet i internettruting

Bacheloroppgave i Informatikk, drift av datasystemer
Veileder: Jostein Lund
Medveileder: Olav Kvittem
Mai 2021

NTNU
Kunnskap for en bedre verden

UNINETT

Aleksandra Jekic

Stabilitet i internettruting

Bacheloroppgave i Informatikk, drift av datasystemer
Veileder: Jostein Lund
Medveileder: Olav Kvittem
Mai 2021

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

NTNU
Kunnskap for en bedre verden

Sammendrag

Organisasjonen RIPE NCC har en åpen plattform kalt Atlas hvor de lagrer data fra tusenvis av prober som fra over hele verden sender forskjellige typer testtrafikk. Denne oppgaven er et samarbeid med Uninett hvor vi forsøker å bruke resultater fra traceroutes til å oppdage rutingproblemer på det globale Internett. Prosjektet har ledet til et system hvor man i et Linux-miljø med et Python-program automatisk laster ned RIPE Atlas-data, analyserer denne for antatte utfall og laster rapporter om dem inn i Elasticsearch/Kibana. Det gis også eksempler på at man forsøker å bruke systemet til å finne nøyaktige utfall.

Abstract

The organization RIPE NCC has an open platform called Atlas where they store data from thousands of probes which from all over the world send different types of test traffic. This thesis is a collaboration with Uninett where we attempt to use results from traceroutes to discover routing problems on the global Internet. The project has lead to a system which in a Linux environment with the help of a Python program automatically downloads RIPE Atlas data, analyzes it for possible outages and loads reports about them into Elasticsearch/Kibana. Examples are also given of trying to use the system to find specific outages.

Sluttrapport

Innholdsfortegnelse

1. Introduksjon (Hensikten med dokumentet) 5
2. Oppgavebeskrivelse 6
3. Hvordan oppgaven ble løst 7
3.1. Metoder og standarder 7
3.2. Bruk av litteratur og Internett 7
3.3. Oversikt over maskinvare 8
3.4. Programvare som ble brukt 8
3.5. Hvordan arbeidsprosessen har vært 9
4. Resultater 11
4.1. Vurdering ut ifra prosjektmål 11
4.1.1. Effektmål 11
4.1.2. Resultatmål 11
4.1.3. Prosessmål 12
4.2. Egenvurdering av prosjektet 13
5. Videre arbeid 14
6. Referanser 15

1. Introduksjon (Hensikten med dokumentet)

Dette dokumentet er en sluttrapport utarbeidet i forbindelse med bacheloroppgaven til Aleksandra Jekic ved NTNU våren 2021. Arbeidet er gjort i samarbeid med Uninett. Veileder fra NTNU er Jostein Lund mens Olav Kvittem representerer Uninett.

Organisasjonen RIPE, som jobber med det globale Internett, har en åpen plattform hvor de lagrer data fra prober som lager forskjellige typer testtrafikk på Internett. Tanken er at disse dataene kan være nyttig til forskning og analyse av Internett, for eksempel for faginstitusjoner og tilbydere av Internett. I dette prosjektet har oppgaven vært å utvikle et program som kan kjøres på en Linux-tjener, analysere noen av disse dataene og levere rapporter om utfall i det globale Internett. Prosjektet er ment å være en liten del av Dragonlab, et felles forskningsprosjekt mellom nettleverandører for forskning og undervisning hvor Uninett deltar. Man finner tidligere publiseringer knyttet til Dragonlab på Uninetts nettsider. (Uninett, 2019)

Sluttrapporten skal oppsummere arbeidet med prosjektet: Hva prosjektet gikk ut på og problemstillingen var, hvordan arbeidet gikk og hva resultatet ble. Vi skal til slutt også se på hva det eventuelt er å jobbe videre med dersom noen skal ta videre prosjektet. I tråd med at dette prosjektet først og fremst er forskning, er kildekoden offentlig tilgjengelig etter lisensen CC BY-SA 4.0 for de som måtte ønske å ta arbeidet videre. (Creative Commons, 2021)

Dette er den siste rapporten som skrives i prosjektet, men den er plassert først i den endelige prosjektrapporten da den fungerer som en god oversikt og oppsummering av arbeidet. Resten av rapportene knyttet til prosjektet følger i kronologisk rekkefølge. Forstudierapporten beskriver oppgaven og setter rammene rundt den. Designrapporten er en teknisk oversikt av prosjektet. Den lengste delen av prosjektrapporten er Driftsrapporten som i teknisk detalj beskriver hvordan det fullførte systemet er satt opp og fungerer.

2. Oppgavebeskrivelse

Oppgaven er basert på en utlysning Uninett gjorde om en tilgjengelig bacheloroppgave.

Målingsplattformen RIPE Atlas samler inn resultatene fra millioner av traceroutes som utføres over hele Internett. Idéen er å analysere disse dataene for å finne brudd på overføringer.

Slike brudd indikerer problemer med ruting på Internett og skal i utgangspunktet ikke skje dersom alt er satt opp riktig. Disse bruddene er også noe som påvirker og forstyrrer oss alle. Et eksempel er når bilde og lyd i et øyeblikk stopper opp under et videomøte. Hvis man finner slike brudd, kan man også forsøke å gjøre noe med dem og finne ut av årsaken. Over tid er det potensielt mulig å bidra til at kvaliteten på hele det globale Internett blitt bedre.

Veileder fra Uninett, Olav Kvittem, har på forhånd forhørt seg om idéen og kommet frem til at ingen har gjort noe lignende med RIPE Atlas. «Det er ikke så vanskelig å få en helt original idé».

Dersom man skal koke denne oppgaven ned til en problemstilling blir det, «Hvordan kan RIPE Atlas-plattformen brukes til å oppdage rutingproblemer på det globale Internett?»

I praksis har oppgaven gått ut på å skrive et program i Python som laster ned logger fra RIPE Atlas. Disse analyseres for brudd som det skrives ut rapporter om. Programmet blir automatisert og kjørt rutinemessig ved hjelp av skript i et Linux-miljø. Bruddrapportene programmet lager blir også automatisk lastet inn i søkeprogramvaren Elasticsearch. Ved hjelp av Kibana kan man deretter se nærmere på og visualisere dataene man har funnet.

3. Hvordan oppgaven ble løst

I dette kapittelet diskuteres det hvordan arbeidet med oppgaven ble gjennomført.

3.1. Metoder og standarder

Når det kommer til prosjektstyring og dokumentasjon, har oppgaven holdt seg til standardene satt av studiet. I forstudierapporten har prosjektet blitt organisert på forhånd med formelle mål, suksesskriterier og tiltak mot høye risikoer. I tillegg har det under hele prosjekttiden blitt skrevet ukerapporter og møtereferater. En timeliste har også blitt ført.

Når det kommer til det tekniske arbeidet med prosjektet, har man i liten grad forholdt seg til formelle metoder og standarder. På den andre siden kan man på en måte si at dette er en standard og metode i seg selv. Tidlig i prosjektet kom det frem at veileder fra Uninett foretrakk en smidig arbeidsprosess. Dette er en moderne arbeidsmåte hvor man har hyppige leveranser og får raske tilbakemeldinger. Det er vanlig å bruke i prosjekter hvor man må finne ut av mye underveis. Målet er også at man blir mer effektiv fordi man bruker mindre tid på planlegging og at hyppig kommunikasjon fører til at oppdragsgiver får resultatet de ønsker.

Jeg har også på eget initiativ passet på å holde koden modulær. Dette vil si at den deles opp i mange mindre funksjoner som kaller hverandre. Dette skal gjøre koden mer oversiktlig og lettere å endre siden hver del er klart separert. Mot slutten av prosjektet mener jeg dette gav gevinst da det viste seg å være overraskende enkelt å implementere ny uventet funksjonalitet veileder fra Uninett ønsket seg. Dette er trolig også nyttig siden systemet er utviklet med tanke på at noen skal jobbe videre med det. Jeg har også passet på å kommentere koden grundig. Når det gjelder kodestandarder, har jeg ikke fulgt noen spesifikk kodestandard som for eksempel PEP8. Jeg har likevel passet på å holde kodestilen konsistent etter mitt eget skjønn.

3.2. Bruk av litteratur og Internett

I hovedsak har jeg benyttet Internett for å lære meg det jeg trengte som jeg ikke kunne. Eksempler er hvordan man lager separate tråder i Python og hvordan man oppretter en index i Elasticsearch. En fordel med at arbeidet er såpass praktisk, er at det stort sett ikke var så farlig med hvor informasjonen kommer fra siden man bare kan prøve ut det som foreslås og se hva som skjer. Veileder fra NTNU og Uninett har begge vært veldig nyttig når det kommer til programmeringsarkitektur som de har mer erfaring med. I løpet av prosjektet opplevde jeg et par ganger at jeg fullstendig overså enkle løsninger på store problemer jeg hadde.

3.3. Oversikt over maskinvare

I dette delkapittelet beskrives maskinvare som har blitt brukt i løpet av prosjektet.

Maskinvare	Beskrivelse
• Kraftig Linux-tjener hos Uninett med versjon 20.04.2 av Ubuntu, ca. 800 GB RAM og 64 prosessorer	Tjeneren hvor systemet er satt opp og automatisert
• Privat laptop med 16 GB RAM og SSD	Brukt til å skrive mesteparten av koden og gjøre begrensede tester raskt

3.4. Programvare som ble brukt

I dette delkapittelet beskrives programvaren som har blitt brukt under prosjektet.

Programvare	Beskrivelse
Visual Studio Code	Visual Studio Code er en såkalt editor for kildekode som jeg har brukt til å programmere systemet.
glogg	Siden RIPE Atlas sine logger er så store at de krasjer normale programmer for tekstbehandling, har jeg hatt stor nytte av glogg gjennom hele arbeidet. (glogg, 2021)
Elasticsearch	Søkeprogramvare brukt til å lagre og strukturere loggene som kommer ut av programmet mitt.
Kibana	Programvare brukt for å utforske og visualisere data fra Elasticsearch.
Ubuntu	En populær distribusjon av Linux. Brukt på tjeneren til Uninett hvor systemet er satt opp.
Debian	En populær distribusjon av Linux. Brukt i virtuell maskin på egen PC for å gjøre ulike tester av systemet før det ble satt opp.

Windows 10 Education	Operativsystemet på den private PC-en jeg har brukt under store deler av arbeidet.
Microsoft Word	Tekstbehandlingsverktøy. Brukt til å skrive prosjektrapporten.
Microsoft Excel	Program for å lage regneark. Brukt til å føre timelister under prosjektet.
Microsoft Powerpoint	Program for å lage presentasjoner. Brukt til å lage presentasjonen som skal fremføres etter prosjektet.
Microsoft Teams	Samarbeidsverktøy. Brukt til å dele filer og ha videomøter.
Microsoft Project	Planleggingsverktøy. Brukt til å planlegge prosjektet og lage Gantt-diagram.
Zoom	Program for å ha videomøter. Enkelte ganger brukt som alternativ til Teams.
Bash	Kommandospråk i Linux.
Git/GitLab	Versjonskontrollsystem for kildekode med mye ekstra funksjonalitet. Brukt for å dele programmet internt i Uninett.
Pip	Pakkebehandlingsprogram brukt for å installere avhengigheter.
Python3	Populært programmeringsspråk.

I designrapporten er ytterligere moduler og biblioteker brukt i programmet beskrevet.

3.5. Hvordan arbeidsprosessen har vært

Stort sett har det fungert slik at jeg muntlig har fått forklart hva veileder i Uninett ønsker seg. Jeg har deretter forsøkt å lage det slik jeg forstår det og fått nye tilbakemeldinger etterpå som jeg har jobbet videre med. Mens jeg har jobbet med det tekniske, har jeg ut ifra fremdriftsplanen fra forstudierapporten levert rapportene for godkjenning.

Spesielt i begynnelsen syntes jeg det var utfordrende å jobbe på denne måten. Tidvis var jeg usikker på om jeg jobbet i riktig retning og om jeg ville komme til mål til slutt. Jeg har også noen ganger hatt en følelse av at «målstreken flyttes» med idéer om komplisert, ny funksjonalitet som dukket opp frem til siste sekund. Som veileder fra NTNU sa mot slutten av prosjektet, «Det virker som om man kunne fortsatt å jobbe med dette prosjektet til evig tid.»

Det har også noen ganger opplevdes som vanskelig å forene den smidige arbeidsmåten hos Uninett med alle de formelle kravene skolen har satt. En annen utfordring har vært å få denne oppgaven med å passe med rapportmalene som er rettet mot mer driftsrelaterte oppgaver.

Samtidig føler jeg meg veldig fornøyd med prosjektet nå på slutten. Jeg tror denne arbeidsmåten har fått det beste ut av meg siden den hele tiden har blitt tilpasset hvor raskt jeg

jobber og hvor mye jeg får til. Jeg tror virkelig man har fått alt jeg klarer ut av meg. Mitt inntrykk er også at både veileder fra NTNU og Uninett er fornøyd med resultatet.

Underveis føler jeg også at jeg har lært utrolig mye. På grunn av den utforskende måten jeg har jobbet på er det også mye jeg har satt meg inn i og brukt mye tid på som ikke endte opp i det ferdige resultatet. Et eksempel er at jeg har jobbet en del med tråder i Python, som jeg visste svært lite om på forhånd, og Python-biblioteker som Plotly og Pandas.

4. Resultater

I dette kapittelet skal jeg vurdere resultatet av prosjektet. Først skal jeg gjøre det med utgangspunkt i prosjektmålene fra forstudierapporten og så i form av en friere refleksjon.

4.1. Vurdering ut ifra prosjektmål

I dette delkapittelet skal jeg vurdere effektmål, resultatmål og prosessmål satt i forstudierapporten. Grønn status betyr at målet er møtt etter det jeg vet, gul status betyr at det til en viss grad er møtt, rød status betyr at målet ikke er nådd. Blå betyr ukjent.

4.1.1. Effektmål

Nr.	Beskrivelse av mål	Status	Kommentar
1	Uninett sparer tid og penger ved at studenter gjør enkle oppgaver for dem.		Mitt inntrykk er at resultatet av prosjektet er nyttig for Uninett og det de ønsket seg. Jeg har også inntrykk av at prosjektet har kostet dem lite tid og penger.
2	Kunnskap om hva Uninett og RIPE jobber med spres ut i IT- og studentmiljøer.		Jeg har sittet mye hjemme denne våren på grunn av korona, så jeg kan nok ikke si at jeg har snakket med mange om Uninett og RIPE.
3	Uninett får muligheten til å bli kjent med og eventuelt rekruttere en student.		Jeg føler absolutt dette målet er oppnådd.

4.1.2. Resultatmål

Nr.	Beskrivelse av mål	Status	Kommentar
1	Et ferdig program som kan ta inn logger fra RIPE Atlas og analysere disse for utfall.		Som beskrevet i driftsrapporten, har vi definitivt et slikt program.

2	Bacheloroppgaven skal være ferdig med alle krav innfridd innen 20. mai 2021: Forstudierapport, designrapport, driftsrapport, sluttrapport og muntlig presentasjon.		Oppfylt.
3	Bruke et sted mellom 450 og 550 timer på prosjektet i løpet av semesteret.		Brukte ca. 505 timer på prosjektet samlet.
4	En MS Project-fil som beskriver prosjektets forløp samt timelister og ukesrapporter.		Oppfylt.

4.1.3. Prosessmål

Nr.	Beskrivelse av mål	Status	Kommentar
1	Få mer praktisk erfaring med arbeidslivet.		Føler definitivt at jeg har fått mer erfaring med hvordan det er å jobbe med IT ute verden. Opplever at grensene for oppgaver er mye vanskeligere å sette og at det ofte ikke finnes «ett rett svar».
2	Lære mer om Internett, analyse av data og programutvikling.		Føler definitivt jeg har lært utrolig mye om alt mulig.
3	Prøve å ikke jobbe mer enn 8-10 timer hver dag med dette og andre fag.		Har jobbet ca. 8 timer dagen dette semesteret og følt meg veldig bra med dette.
4	Oppnå karakter B eller bedre på oppgaven.		Det får vi nesten se.
5	Få sommerjobb og/eller deltidsjobb hos Uninett.		Har fått sommerjobb hos Uninett.
6	Ha et godt samarbeid med veileder fra både Uninett og NTNU.		Føler samarbeidet har vært godt med begge veiledere.

4.2. Egenvurdering av prosjektet

Inntrykket mitt er at både veileder fra NTNU og Uninett er fornøyd. Jeg mener også jeg formelt har oppfylt kravene til oppgaven og svart på problemstillingen: Oppgaven viser praktisk hvordan man kan bruke RIPE Atlas til å oppdage rutingproblemer. I tillegg er jeg fornøyd med mye ekstra funksjonalitet i programmet som opprinnelig ikke ble planlagt. Det største er kanskje at programmet kartlegger rutene det rapporteres om med RTT-informasjon. Dette er det meningen skal lastes inn i et annet program enn Elasticsearch/Kibana. Jeg er også ganske fornøyd med rapportene og prosjektstyringsdokumentene. Ikke minst føler jeg at jeg har lært utrolig mye i dette prosjektet om utrolig mange emner jeg selv syntes var svært interessante.

5. Videre arbeid

Som veileder fra NTNU sa på slutten av prosjektet, «Det virker som om man kunne fortsatt å jobbe med dette prosjektet til evig tid.» Det tror jeg absolutt på en måte stemmer. Nå har vi et funksjonelt system som kan identifisere potensielle brudd på det globale Internett ved hjelp av RIPE Atlas-plattformen. Jeg tror heller ikke det er mye å vinne på å forsøke å forbedre ytelsen.

Veien videre blir å se nærmere på bruddene man finner og kanskje forsøke å komme i kontakt med de som kan gjøre noe med dem. Etter hvert som man utforsker bruddene, blir det også trolig naturlig å gjøre endringer i systemet underveis. Det kan være alt fra hva slags informasjon programmet henter ut til å lage nye visualiseringer i Kibana.

En annen løs tråd er kartene eller topologiene programmet lager av alle rutene. Den opprinnelige tanken var at disse skulle lastes inn i et eget program som skulle visualisere dem. Det meste av jobben er trolig gjort her, men det gjenstår fortsatt å knytte alt sammen.

6. Referanser

Creative Commons. (2021) Navngivelse-DelPåSammeVilkår 4.0 Internasjonal (CC BY-SA 4.0). Tilgjengelig fra: https://creativecommons.org/licenses/by-sa/4.0/deed.no (Hentet: 16. mai 2021).

glogg (2021) *glogg - the fast, smart log explorer*. Tilgjengelig fra: https://glogg.bonnefon.org/ (Hentet: 5. april, 2021).

RIPE Atlas. (2021) What is RIPE Atlas? Tilgjengelig fra: https://atlas.ripe.net/about/ (Hentet: 16. mai 2021).

Forstudierapport

Innholdsfortegnelse

1. Introduksjon (Hensikten med dokumentet) 18
2. Forkortelser og definisjoner 19
3. Tabelliste 20
4. Bakgrunn for prosjektet 21
5. Prosjektmål 22
5.1. Effektmål 22
5.2. Resultatmål 22
5.3. Prosessmål 23
5.4. Prosjektets omfang 23
5.5. Funksjonelle egenskaper og krav 23
5.6. Ikke-funksjonelle egenskaper og krav 23
5.7. Prosjektets milepæler og hovedaktiviteter 23
6. Interessenter og rammebetingelser 26
6.1. Interessentanalyse 26
6.2. Rammebetingelser 27
7. Kritiske suksessfaktorer 28
7.1. Suksessfaktorer 28
7.2. Informasjonsbehov 29
8. Risikoanalyse 30
8.1. Risikomatrise 30
8.2. Uønskede hendelser 31
8.3. Tiltak 34
9. Kost/nytte-analyse 36
10. Retningslinjer og standarder 37
10.1. Krav til dokumentasjon 37
10.2. Krav til kvalitetsgjennomganger 37
10.3. Krav til standarder og metoder 37
10.4. Endringshåndtering 38
11. Prosjektorganisering 39
12. Anbefaling om videre arbeid 40
13. Referanser 41

1. Introduksjon (Hensikten med dokumentet)

Dette dokumentet er en forstudierapport utarbeidet i forbindelse med bacheloroppgaven til Aleksandra Jekic ved NTNU våren 2021. Arbeidet er gjort sammen med Uninett. Veileder fra NTNU er Jostein Lund mens Olav Kvittem representerer Uninett.

Organisasjonen Ripe, som jobber med det globale Internett, har lagt ut målere som sender ulike typer trafikk ut på Internett. Tanken er at disse dataene kan være nyttig til forskning og analyse av Internett, for eksempel for faginstitusjoner og tilbydere av Internett. Denne oppgaven skal resultere i et program som kan kjøres på en Linux-tjener som er i stand til å analysere disse loggene og levere rapporter om utfall. Det er også mulig programmet senere kan bli brukt til å analysere Uninetts egne logger. Prosjektet er ment å være en liten del av Dragonlab-prosjektet som er et felles forskningsprosjekt mellom nettleverandører for forskning og undervisning.

Forstudierapporten skal gi en beskrivelse av oppgaven og rammene rundt denne. Den skal være en slags avklaring, til hjelp i arbeidet videre og en slags kontrakt mellom de involverte.

Først følger en liste over nyttige forkortelser og definisjoner. Deretter vil vi se på bakgrunnen for prosjektet, det vil si formelt beskrive problemet oppgaven skal løse og hvilke løsninger som kanskje allerede finnes. Under prosjektmål vil vi konkretisere hva man håper man skal oppnå fra dette arbeidet. I interessenter og rammebetingelser vil vi si på hvem som er involvert i prosjektet og hva deres motiver er. I kritiske suksessfaktorer vil vi merke oss faktorer som kan være spesielt viktig å tenke på. Under risikoanalyse vil vi vurdere hva som eventuelt kan gå galt ved prosjektet. Vi skal også gjennomføre en svært enkel kost/nytte-analyse. Vi vil deretter se på retningslinjer og standarder vi skal forholde oss til. Helt til slutt vil vi se på prosjektorganiseringen samt komme med en anbefaling om arbeidet videre.

I den nesten rapporten, designrapporten, skal vi gå nærmere inn på det tekniske rundt oppgaven. I driftsrapporten vil gjennomføringen av selve arbeidet beskrives. I tillegg til disse tre hovedrapportene vil det også være en sluttrapport som vil vurdere resultatet og prosessen.

2. Forkortelser og definisjoner

Begrep	Beskrivelse
REST API	Oppsett hvor en klient henter informasjon fra en tjener ved hjelp av HTTP.
HTTP	Protokollen mest brukt til å utveksle informasjon på Internett.
Flatfil	Data er lagret i en stor fysisk fil slik at det er bare én oppføring på hver linje.
ICMP (Internet Control Message Protocol)	Protokoll som brukes til å teste nettverk og overføre meldinger om feil.
Ping	Hjelpeprogram som sender ut et ekko med ICMP til en bestemt adresse. Mottakeren sender et svar tilbake og Ping registrer hvor lang tid turen frem og tilbake tok.
Traceroute/tracert	Kommando som sender ut ping med stadig høyere «Time-To-Live» (altså antall hopp før pakken skal forkastes). Tanken er at man skal pinge hver maskin til målen, altså «spore ruten», men blir ikke alltid slik i praksis siden veien kan endre seg underveis.
SSL/TLS	Protokoller som tilbyr kryptert kommunikasjon.
DNS	Tjeneste som kobler domenenavn til IP-adresser. Datamaskiner gjør rutinemessig kall til DNS-tjenere for å finne navn.
NTP	Protokoll som brukes til å synkronisere tiden på forskjellige datamaskiner.

Tabell 1: Forkortelser og definisjoner

3. Tabelliste

Tabell 1: Forkortelser og definisjoner 19
Tabell 2: Effektmål 22
Tabell 3: Resultatmål 22
Tabell 4: Prosessmål 23
Tabell 5: Milepæler og hovedaktiviteter 25
Tabell 6: Analyse av interessenter 27
Tabell 7: Rammebetingelser 27
Tabell 8: Suksessfaktorer 29
Tabell 9: Risikomatrise 30
Tabell 10: Analyse av uønskede hendelser 33
Tabell 11: Utfylt risikomatrise 34
Tabell 12: Oversikt over tiltak 35
Tabell 13: Dokumentasjonskrav 37
Tabell 14: Prosjektorganisasjonen 39

Forstudierapporten inneholder ingen figurer, derfor er det kun en tabelliste.

4. Bakgrunn for prosjektet

Uninett er et statlig eid selskap som utvikler og driver nettet som knytter sammen institusjoner innen utdanning og forskning i Norge. De fungerer også som en internettilbyder for disse. I samarbeid med lignende organisasjoner fra andre land er Uninett med på Dragonlab-prosjektet. Dette prosjektet går ut på at man sammen forsker på problemer i nettet mellom. Man finner tidligere forskning knyttet til prosjektet på Uninetts nettsider. (Uninett, 2019)

I dette prosjektet blir RIPE Atlas brukt. (RIPE Atlas, 2021) Dette er en åpen målingsplattform for trafikk på Internett som består av nesten 9000 såkalte prober og 150 såkalte anker. Probene sender ut ulike typer trafikk som de logger resultatene fra. Ved å analysere denne trafikken kan det være mulig å finne problemer på det globale Internett. Anker kan beskrives som mer avanserte prober som i tillegg kan fungere som mål for de vanlige probene.

4.1. Beskrivelse av problemer og behov

I dette prosjektet skal det utvikles et program som er i stand til å ta imot logger fra RIPE Atlas. Programmet skal kunne analysere og gi informasjon om utfall på Internett. Det skal vise hvor mange utfall, hvor lenge og eventuelt om det finnes klynger av problemer.

4.2. Kort om dagens systemer og rutiner

Uninett har allerede et Perl-program som omformer loggene til data som deretter kan vises frem på en nettside i form av grafer. Det er mulig de har mye annet tilknyttet disse loggdataene som også er ferdig utviklet, men det er trolig ikke relevant til dette studentprosjektet.

RIPE tilbyr på dataene sine i flere forskjellige formater. Det er mulig å laste dem ned i form av flatfiler, gjøre kall mot en Google BigQuery-database samt hente dem fra REST API gjennom HTTP. I tillegg tilbyr RIPE at man får laget rapporter gjennom et grafisk grensesnitt. Til denne oppgaven er det i første instans relevant med flatfilene, men programmet som skal bygges bør lages med øye for at det skal kunne ta inn data på forskjellige måter.

5. Prosjektmål

I denne rapporten har vi valgt å skille mellom effektmål, resultatmål og prosessmål som vi skal gå inn på i hver kategori. Det er en prioritet at målene skal være såkalt operasjonelle, det vil si at det er enkelt å vurdere objektivt om de er oppfylt eller ikke. Ved å spikre opp målene såpass konkret blir det også lett å se om man virkelig har forstått hverandre riktig.

5.1. Effektmål

Effektmål beskriver effektene vi ønsker av at prosjektet gjennomføres.

Nr.	Effektmål
1	Uninett sparer tid og penger ved at studenter gjør enkle oppgaver for dem.
2	Kunnskap om hva Uninett og RIPE jobber med spres ut i IT- og studentmiljøer.
3	Uninett får muligheten til å bli kjent med og eventuelt rekruttere en student.

Tabell 2: Effektmål

5.2. Resultatmål

Resultatmål beskriver alt som konkret må foreligge når prosjektet er ferdig.

Nr.	Resultatmål
1	Et ferdig program som kan ta inn logger fra RIPE Atlas og analysere disse for utfall.
2	Bacheloroppgaven skal være ferdig med alle krav innfridd innen 20. mai 2021: Forstudierapport, designrapport, driftsrapport, sluttrapport og muntlig presentasjon.
3	Bruke et sted mellom 450 og 550 timer på prosjektet i løpet av semesteret.
4	En MS Project-fil som beskriver prosjektets forløp samt timelister og ukesrapporter.

Tabell 3: Resultatmål

5.3. Prosessmål

Prosessmål beskriver hva studenten personlig ønsker å oppnå ved å gjennomføre prosjektet.

Nr.	Prosessmål
1	Få mer praktisk erfaring med arbeidslivet.
2	Lære mer om Internett, analyse av data og programutvikling.
3	Prøve å ikke jobbe mer enn 8-10 timer hver dag med dette og andre fag.
4	Oppnå karakter B eller bedre på oppgaven.
5	Få sommerjobb og/eller deltidsjobb hos Uninett.
6	Ha et godt samarbeid med veileder fra både Uninett og NTNU.

Tabell 4: Prosessmål

5.4. Prosjektets omfang

I denne delen av rapporten avgrenser vi prosjektet og hva det skal dreie seg om.

Probene sender ut og samler inn data om ping, traceroute, SSL/TLS, DNS, NTP og HTTP. I første omgang skal oppgaven fokusere på flatfiler med traceroute-logger over en time eller en dag.

Dersom det blir nok tid, er det mulighet for å gjøre programmet mer fleksibelt.

5.5. Funksjonelle egenskaper og krav

Programmet skal i utgangspunktet utvikles i Python og kunne kjøre på en Linux-tjener. Det er selvfølgelig at det kan finne utfall ved å lese loggene og lage rapporter om disse.

5.6. Ikke-funksjonelle egenskaper og krav

Programmet må være mulig for en tredjeårsstudent innen data å utvikle på et halvt år.

5.7. Prosjektets milepæler og hovedaktiviteter

Tabellen under beskriver milepæler samt hovedaktiviteter i prosjektet. Milepæler forventes oppnådd innen sin frist for å vise at prosjektet holder seg innenfor sine tidsrammer.

For å indikere hvilke aktiviteter som er milepæler, er disse markert med blåfarge.

Navn	Beskrivelse	Dato
Jobbe med forstudierapport	Forstudierapporten skal overordnet beskrive oppgaven og dens rammer. Den fungerer som en avklaring av prosjektet og en slags kontrakt. Starter også med MS Project, ukesrapporter og timelister.	11.01.2021 til 22.01.2021
Begynne arbeidet med oppgaven	Aleksandra begynner å prøve seg frem med loggfilene og Python.	11.01.2021 til 22.01.2021
Ferdig forstudierapport	Forstudierapporten er ferdig og godkjent av både veileder fra Uninett og NTNU.	22.01.2021
Jobbe med systemkrav/designrapport	I systemkravrapporten skal de tekniske kravene til programmet beskrives nærmere.	23.01.2021 til 26.02.2021
Første prototype	Programmet skal kunne ta inn logger og behandle disse på en grunnleggende måte.	26.02.2021
Ferdig systemkrav/designrapport	Systemkravrapporten er ferdig og godkjent av både veileder fra Uninett og NTNU.	26.02.2021
Starte arbeidet med driftsdokumentet	Driftsdokumentet skal beskrive hvordan programmet settes opp.	27.02.2021 til 08.05.2021
Kan finne utfall	Programmet klarer kjenne igjen utfall ved hjelp av loggene.	12.03.2021
Kan lage rapporter	Programmet lager interessante rapporter basert på dataene.	16.04.2021
Finpussing	Finner feil og finpusser programmet.	17.04.2021 til 08.05.2021

Ferdig program	Programmet er ferdig og godkjent av veileder fra Uninett.	08.05.2021
Skrive sluttrapport, vurdering av gruppesamarbeidet og lage presentasjon	Avsluttende arbeid på oppgaven.	08.05.2021 til 20.05.2021
Levering	Prosjektet er avsluttet og alle krav er gjennomført.	20.05.2021

Tabell 5: Milepæler og hovedaktiviteter

Dette er en tidlig skisse og kan endres. Det åpnes også for å jobbe videre med ekstra funksjonalitet dersom det skulle bli tid og overskudd til dette. Man kan også finne en tilsvarende versjon av denne planen i en egen MS Project-fil med Gantt-diagram.

6. Interessenter og rammebetingelser

I denne delen skal vi se på interessentene og rammebetingelsene for prosjektet. En interessent defineres som en person, gruppe eller organisasjon som påvirker eller kan bli påvirket av et prosjekt. Rammebetingelser beskriver absolutte forutsetninger og nødvendigheter rundt prosjektet. Ved å forstå interessentene og rammebetingelsene kan vi forstå prosjektet bedre.

6.1. Interessentanalyse

Under følger et skjema med dette prosjektets interessenter. Her vil jeg beskrive kriterier for at hver interessent skal anse prosjektet som en suksess og deres eventuelle bidrag til prosjektet.

Interessent	**Suksesskriterier**	**Bidrag til prosjektet**
Uninett	Samsvarer i stor grad med effektmålene beskrevet i et tidligere kapittel.	Bidrar med ressurser som tid fra en ansatt og eventuell teknologi som må kjøpes.
Olav Kvittem (veileder fra Uninett, «oppdragsgiver»)	▪ Et program som kan analysere logger for utfall på Internett. ▪ Har uttrykket ønske om å ha lite med det formelle rundt oppgaven å gjøre som rapportene.	Veileder prosjektet.
RIPE	Dataene deres blir brukt på en måte som tjener forskning og utvikling av Internett.	Bidrar med data fra sitt Atlas-nettverk.
NTNU	▪ Flest mulig studenter skal fullføre studiene. ▪ Studentene deres representerer dem på en god måte og har gode resultater.	Bidrar med ressurser som tid fra en ansatt og teknologi som Microsoft Teams.

Jostein Lund (veileder fra NTNU)	Samsvarer i stor grad med NTNUs suksesskriterier.	Veileder prosjektet.
Aleksandra Jekic (den som gjennomfører prosjektet, «prosjektgruppen»)	Samsvarer i stor grad med prosessmålene beskrevet i et tidligere kapittel.	Gjennomfører prosjektet.
Internettbrukere	Et stabilt og raskt Internett.	Bidrar ikke.

Tabell 6: Analyse av interessenter

I interessentanalyser snakker man ofte om en sluttbruker, men det avhenger sannsynligvis av det ferdige resultatet hvordan og eventuelt av hvem det ferdige programmet vil bli brukt.

6.2. Rammebetingelser

I denne delen beskrives absolutte og nødvendige krav. Dersom disse rammene ikke kan opprettholdes må det stilles spørsmål til om prosjektet kan gjennomføres slik det er tenkt.

Prosjektet skal ikke kreve mer enn 550 timer arbeid i løpet av semesteret.
Prosjektet skal være ferdigstilt innen 20.05.2021. Da skal det foreligge et ferdig program som oppfyller prosjektets krav samt all dokumentasjon krevd av skolen.
Veileder, både fra NTNU og Uninett, må være tilgjengelig underveis og hjelpe.
De tre hovedrapportene må ferdigstilles innen fristene beskrevet i planen.
Senest i mars foreligger en fungerende prototype med grunnleggende funksjonalitet.

Tabell 7: Rammebetingelser

7. Kritiske suksessfaktorer

I dette kapittelet skal vi beskrive faktorer som anses som kritiske for at prosjektet skal bli en suksess. Man kan tenke på det som en liste over faktorer som må tas spesielt hensyn til. Under informasjonsbehov skal vi se på hvordan informasjon skal formidles mellom de involverte.

7.1. Suksessfaktorer

Under følger en tabell over hensyn som anses som kritiske suksessfaktorer.

Suksessfaktor	**Beskrivelse**
God kommunikasjon	Det sier kanskje seg selv, men forskning viser at god kommunikasjon er svært viktig for at prosjekter skal kunne lykkes. (DeFranco og Laplante, 2017) Spesielt nå som det er korona, og alt blir over epost og videokonferanser, er det viktig at det gjøres en ekstra innsats på dette området.
Tålmodige veiledere	Aleksandra er veldig flink, men det er mye som er nytt i dette prosjektet som hun aldri har jobbet tidligere med – så det er viktig at veilederne tar seg tid til å veilede ved behov og at de gjør det på et passende nivå.
Fleksible arbeidstider	Alle deltakerne i prosjektet jobber også med andre ting. Det er derfor viktig at man tilpasser seg hverandre slik at det er mulig at man får gjennomført alt man jobber med.
Enighet om krav	Noe av det mest utfordrende innen IT-prosjekter, har Aleksandra lært, er å vite nøyaktig hva som skal gjennomføres. Ved hjelp av god kommunikasjon og faste møter med alle parter kan dette innfris.
Jevn fremgang	Det blir ofte best resultat og best stemning hvis man fordeler arbeidsmengden likt over

	lengre tid. Aleksandra regner med å bruke mellom 20 og 30 timer på prosjektet pr. uke.
Rammebetingelsene opprettholdes	I et tidligere kapittel er rammebetingelsene for prosjektet beskrevet. Det er viktig at disse overholdes under prosjektet.

Tabell 8: Suksessfaktorer

7.2. Informasjonsbehov

I utgangspunktet skal kommunikasjonen mellom veilederne og «prosjektgruppen», altså Aleksandra Jekic, foregå over Microsoft Teams. I hvert fall til å begynne med skal det være et møte mellom alle partene annenhver uke på torsdager, klokken 12:30. I tillegg kan partene ta kontakt med hverandre ved behov. Kommunikasjon foregår også over epost. Underveis vil veilederne kunne gi tilbakemelding arbeid og rapporter samt godkjenne milepæler. Microsoft Teams vil dessuten brukes til å dele filer med hverandre. Utkast til rapporter vil lastes opp fortløpende. I tillegg vil MS Project-filer, timelister og ukerapporter være tilgjengelige der.

8. Risikoanalyse

Risikoanalysen handler enkelt sagt om å sette seg ned før arbeidet starter, tenke over hvilke opplagte farer som kan finnes og beskrive tiltak dersom det anses som nødvendig. Dette er en lite kostbar prosess som kan redde et prosjekt eller spare mye tid og frustrasjon senere.

8.1. Risikomatrise

Det er vanlig å definere risiko som produktet av sannsynlighet og konsekvens. Sannsynlighet og konsekvens vurderes på en skala fra 1 til 5. Den høyeste mulige risikoen blir dermed 25. Akseptkriterier defineres som den høyeste risikoen man er villig til å godta. I dette prosjektet har vi satt akseptkriteriet til 4/25. Ved risikoer fra 5 til og med 9 vurderes tiltak. Risikoer som er høyere enn dette krever tiltak. Varmekartet under gir en oversikt av oppsettet.

Sannsynlighet	5					
	4					
	3					
	2					
	1					
		1	2	3	4	5
	Konsekvenser					

Tabell 9: Risikomatrise

De grønne områdene er innenfor akseptkriteriet (altså >5), det gule området betyr at trusler som havner der kanskje krever tiltak (>10) mens det røde området betyr at tiltak kreves (<10).

8.2. Uønskede hendelser

I denne delen skal vi beskrive mulige uønskede hendelser og plassere dem ut på matrisen.

Navn	Uønsket hendelse	Sanns.	Kons.	Risiko	Vurdering
A	Langvarig sykdom	2	4	8	Det er liten sjanse for at en enkeltperson får korona eller andre alvorlige sykdommer slik ting er nå, men langvarig fravær fra prosjektet ville vært problematisk.
B	Samarbeidet fungerer overhodet ikke	2	5	10	Det er liten grunn til å tro samarbeidet ikke skulle fungere, men dersom det ikke skulle gjøre det vil det gjøre prosjektet vanskelig å gjennomføre.
C	Oppgaven er for vanskelig	1	4	4	Det er mye tid og Aleksandra er flink så oppgaven burde ikke være for vanskelig, men dette vil selvfølgelig kunne være et stort problem.
D	Aleksandra blir overarbeidet	2	4	8	Aleksandra tar også andre fag og risikerer å bli overarbeidet.

E	Manglende kommunikasjon	3	5	15	Spesielt med koronasituasjonen kan det være vanskelig å kommunisere godt mens prosjektet foregår. Dette kan ha svært alvorlige konsekvenser for resultatet.
F	Tap av arbeid	2	4	8	Sannsynligheten for at filer skal bli ødelagt eller lignende vurderes som lav, men det er noe som skjer og man kan miste mye arbeid slik.
G	Ujevn arbeidsinnsats	1	3	3	Aleksandra er veldig strukturert så sannsynligheten for at det blir ujevn arbeidsinnsats er veldig lav.
H	Manglende fremgang	2	4	8	Milepælene er satt for at de skal være mulige å nå, men dersom dette av ulike grunner ikke skulle skje er det problematisk.
I	Problemer med å finne nødvendig lærestoff	1	3	3	Det anses som usannsynlig at det vil være problemer med å finne lærestoff til oppgaven fordi det er ganske populært stoff.
J	Mangler nødvendig teknologi	1	4	4	Både NTNU og Uninett bidrar med nødvendig programvare o. l. ved behov så dette burde ikke utgjøre et problem. Dessuten virker det ikke som om oppgaven krever mye komplisert teknologi.

K	Oppgaven er uklar	3	4	12	Nå som det er korona, og litt vanskeligere å kommunisere enn vanlig, kan det bli vanskelig å bli enig om kravene til prosjektet. Dette kan gjøre det vanskelig å få en tilfredsstillende løsning.
L	Det ferdige programmet er ikke nyttig	2	2	4	Uninett styrer prosjektets innhold. Som jeg vil diskutere kort i kost/nytte-analysen er det også et prosjekt som koster få ressurser så konsekvensene er lave dersom resultatet blir unyttig.
M	Ikke nok tid til å gjennomføre oppgaven	1	4	8	Et semester er ganske lenge, så sannsynligheten anses som lav, men det ville være et problem hvis det ikke var nok tid.

Tabell 10: Analyse av uønskede hendelser

På neste siden har matrisen blitt fylt med de forskjellige uønskede hendelsene etter deres navn.

Sannsynlighet						
	5		B	E		
	4	J, C, M	A, D, F, H	K		
	3	G, I				
	2		L			
	1					
		1	2	3	4	5
	Konsekvenser					

Tabell 11: Utfylt risikomatrise

8.3. Tiltak

Under er det beskrevet tiltak for risikoene der det blir ansett som nødvendige.

Navn		**Risiko**	**Tiltak**
A	Langvarig sykdom	8	For å unngå langvarig sykdom anbefales det at man vasker hender, holder sosiale distanse osv.
B	Samarbeidet fungerer overhodet ikke	10	Viktig at man kommuniserer for å få best mulig forutsetning for samarbeid. Dersom det ikke fungerer i det hele tatt, er det viktig at dette merkes tidlig slik at man kan avslutte prosjektet tidligst mulig.
D	Aleksandra blir overarbeidet	8	Unngå å jobbe mer enn 8-10 timer hver dag.

E	Manglende kommunikasjon	15	Ha faste møter annenhver uke. Være mest mulig tilgjengelig digitalt og på epost. Avtale rammer for kommunikasjon.
F	Tap av arbeid	8	Lagre filer på Microsoft Teams.
K	Oppgaven er uklar	12	Bruke rapportene for å bli enige om krav.

Tabell 12: Oversikt over tiltak

9. Kost/nytte-analyse

Denne delen anses ikke som særlig relevant til dette prosjektet og jeg vil derfor ikke gå mye i detalj. Det er et ganske beskjedent prosjekt som hovedsakelig koster den ekstra tiden Uninetts veileder bruker til å hjelpe prosjektgruppen. Som et lite studentprosjekt er det også lite trolig det vil tjene penger. Det er dessuten primært et prosjekt rettet mot forskning.

10. Retningslinjer og standarder

I dette kapittelet beskriver hvordan man overordnet skal jobbe med prosjektet.

10.1. Krav til dokumentasjon

Tabellen under beskriver dokumentasjonen prosjektet skal inneholde.

Forstudierapporten
MS Project-fil med Gantt-diagram
Timeliste
Ukerapporter
Design/systemkravrapport
Driftsrapport
Sluttreport
Refleksjon over arbeidet
Presentasjon av arbeidet

Tabell 13: Dokumentasjonskrav

10.2. Krav til kvalitetsgjennomganger

Det holdes møte med begge veiledere annenhver uke for å sikre at prosjektet går fremover. Veiledere vil fortløpende gi tilbakemeldinger på og godkjenning av milepæler og rapporter etter hvert som de blir ferdige. Aleksandra vil sende ukesrapporter til veileder fra Uninett hver fredag. I tillegg vil hun holde oppdatert alle prosjektstyringsfilene på Teams.

10.3. Krav til standarder og metoder

Dokumentasjonen vil ta utgangspunkt i malene vi har fått i løpet av studiet. Den kommer til å skrives ved hjelp av Office-produkter som NTNU tilbyr gratis til sine studenter. Kommunikasjon

vil foregå ved hjelp av Microsoft Teams. Når det gjelder selve prosjektet vil teknologien som vil brukes til dette spesifiseres nærmere i systemkrav/designrapporten.

10.4. Endringshåndtering

Ved behov for mindre endringer kan disse endringene uformelt avtales og gjennomføres. Man kan også bruke endringsloggene til dokumentene for å se hva som har blitt endret dersom det skulle være problemer rundt dette. Ved større endringer er det nødvendig å skrive en såkalt endringsmelding. En mal til en slik endringsmelding ligger tilgjengelig på Teams.

11. Prosjektorganisering

Her beskrives deltakerne i prosjektet og deres oppgaver og ansvar.

Navn	Rolle	Oppgaver og ansvar
Jostein Lund	Veileder fra NTNU	▪ Veilede arbeidet og gi hjelp der han kan ▪ Gi tilbakemeldinger på gjennomført arbeid
Olav Kvittem	Oppdragsgiver	▪ Definere oppgaven ▪ Veilede arbeidet ▪ Gi faglig hjelp
Aleksandra Jekic	Prosjektgruppen	▪ Gjennomføre prosjektet ▪ Skrive alle rapporter og dokumentasjon ▪ Be om hjelp ved behov ▪ Holde alle oppdatert om fremdriften i prosjektet

Tabell 14: Prosjektorganisasjonen

12. Anbefaling om videre arbeid

Prosjektet slik det er beskrevet i denne rapporten tyder på at interessene til de ulike partene i stor grad sammenfaller. Det er enkelte risikoer forbundet med prosjektet, for eksempel i forbindelse med kommunikasjon under koronabegrensninger. Samtidig er det foreslått tiltak mot disse og de virker ikke uoverkommelige. Det er dessuten et ganske beskjedent prosjekt hvor det vil ha forholdsvis lave konsekvenser dersom det ikke blir særlig vellykket. Prosjektet anbefales derfor videreført på den måten det er beskrevet i denne rapporten.

13. Referanser

DeFranco, J. F. og Laplante, P. A. (2017) Review and Analysis of Software Development Team Communication Research, *IEEE Transactions on Professional Communication*, 60(2), s. 165-182. doi: 10.1109/TPC.2017.2656626.

RIPE Atlas. (2021) What is RIPE Atlas? Tilgjengelig fra: https://atlas.ripe.net/about/ (Hentet: 16. mai 2021).

Uninett. (2019) *Dragonlab*. Tilgjengelig fra: https://www.uninett.no/node/1819 (Hentet: 16. mai 2021).

Designrapport

Innholdsfortegnelse

1. Introduksjon 45
2. Forkortelser og definisjoner 46
3. Figur- og tabelliste 49
4. Om oppdragsgiver og behov 50
4.1. Bakgrunn for prosjektet 50
4.2. Om oppgaven 50
5. Den overordnede løsningen 51
5.1. Enkel forklaring av systemet 51
5.2. Eksisterende oppsett 52
5.2.1. RIPE Atlas 52
5.2.2. JSON 53
5.2.3. Linux 53
5.2.4. GeoIP2 54
5.2.5. cron 54
5.2.6. Elasticsearch 54
5.2.7. Kibana 55
5.3. Krav til programvaren 55
5.4. Valg og begrunnelser 57
5.4.1. Python 57
5.4.2. Flatfiler 57
6. Detaljert løsningsbeskrivelse 58
6.1. Finne brudd hjelp av traceroutes 58
6.1.1. Hvordan traceroute virker 58
6.1.2. Hvordan en RIPE Atlas-logg ser ut 59
6.1.3. Hvordan man registrer et brudd på en rute 59
6.2. Innhenting av informasjon 60
6.3. Lagring og sortering 60
6.4. Skalering og ytelse 61
6.5. Å skrive ut informasjon 62
6.6. Biblioteker og oppsett 62

6.6.1. Pip 62
6.6.2. Ujson 62
6.6.3. Urllib 62
6.6.4. Math, Time, Datetime 63
6.6.5. bz2, shutil 63
6.6.6. NumPy 63
6.6.7. GeoIP2, Pygeoip 63
6.6.8. Sys, OS 63
6.6.9 cProfile 64
6.6.10 memory-profiler 64
7. Referanser 65

1. Introduksjon

Dette dokumentet er en designrapport utarbeidet i forbindelse med bacheloroppgaven til Aleksandra Jekic ved NTNU våren 2021. Arbeidet er gjort sammen med Uninett. Veileder fra NTNU er Jostein Lund mens Olav Kvittem representerer Uninett.

Oppgaven er å utvikle et program som hjelper med å forske på problemer med videresending av data over det globale Internett. Programmet skal laste ned logger etter millioner av traceroutes fra den åpne målingsplattformen RIPE Atlas. Det skal deretter bruke disse til å identifisere brudd på ruter og rapportere om disse i en ny logg. Disse loggene skal videre lastes inn i Elasticsearch, en søkeprogramvare allerede satt opp på Uninetts tjener. Ved hjelp av Elasticsearch og visualiseringsprogrammet Kibana skal man kunne studere dataene.

Designrapporten skal gi en teknisk oversikt av prosjektet. Den skal beskrive strukturen programmet skal inn i og komponentene som inngår. I tillegg skal de overordnede tekniske valgene begrunnes. Designrapporten skal konkretisere prosjektet og være en rettesnor. Det er også en anledning for prosjektgruppe og oppdragsgiver å bli enig om arbeidet som skal utføres.

I den neste rapporten, Driftsrapporten, vil løsningen og selve arbeidet med prosjektet beskrives i detalj. Dette vil være den største rapporten. For et mindre teknisk, større overblikk av prosjektet og gangen i dette, bør man se tilbake til Forstudierapporten.

2. Forkortelser og definisjoner

Begrep	**Beskrivelse**
REST API	Oppsett hvor en klient henter informasjon fra en tjener ved hjelp av HTTP.
HTTP	Protokollen mest brukt til å utveksle informasjon på Internett.
Flatfil	Data er lagret i en stor fysisk fil slik at det er bare én oppføring på hver linje.
ICMP (Internet Control Message Protocol)	Protokoll som brukes til å teste nettverk og overføre meldinger om feil.
Ping	Hjelpeprogram som sender ut et ekko med ICMP til en bestemt adresse. Mottakeren sender et svar tilbake og Ping registrer hvor lang tid turen frem og tilbake tok.
Traceroute/tracert	Kommando som sender ut ping med stadig høyere «Time-To-Live» (altså antall hopp før pakken skal forkastes). Tanken er at man skal pinge hver maskin til målen, altså «spore ruten», men blir ikke alltid slik i praksis siden veien kan endre seg underveis.
SSL/TLS	Protokoller som tilbyr kryptert kommunikasjon.
DNS	Tjeneste som kobler domenenavn til IP-adresser. Datamaskiner gjør rutinemessig kall til DNS-tjenere for å finne navn.
NTP	Protokoll som brukes til å synkronisere tiden på forskjellige datamaskiner.
RIPE	Réseaux IP Européens Network Coordination Centre. Forvalter IP-nummerserier i Europa, Vest-Asia og den tidligere Sovjetunionen.
RIPE Atlas	Åpen målingsplattform som samler inn data fra det globale Internett ved hjelp av prober.

Probe	Innretning som kobles til en ruter og sender ut traceroutes og andre kall over Internett.
Anker	Store prober som også kan fungere som mål for andre prober og anker.
Elasticsearch	Søkeprogramvare for store tekstfiler.
Kibana	Grensesnitt for Elasticsearch som brukes til å gjøre søke og lage visualiseringer av dem.
SQL	Structured Query Language. Kommandospråk som brukes til å hente, endre, legge inn osv. data i en klassisk relasjonsdatabase.
NoSQL	Database som ikke benytter seg av tabeller eller relasjoner. Elasticsearch kan brukes som en NoSQL-database.
JSON	Javascript Object Notation. Filformat som brukes til å sende data mellom ulike systemer.
cron	Linux-program brukt til å planlegge faste oppgaver.
GeoIP2	System for å hente stedsinformasjon o. l. om IP-adresser fra egne databaser.
atlas-gap-ana.py	Programmet som utvikles i dette prosjektet.
Google BigQuery	Database man kan ha i skyen. Støtter mye avansert funksjonalitet som maskinlæring.
XML	eXtensible Markup Language. Filformat brukt til å utveksle informasjon mellom forskjellige systemer. JSON brukes mye istedet i dag.
Elasticsearch: «Documents»	En enkel porsjon med informasjon i Elasticsearch. For eksempel en enkelt bruddbeskrivelse.
Elasticsearch: «Mappings»	Forteller Elasticsearch hvordan det skal lese attributtene i et «Document».

Elasticsearch: «Index»	Samling «Documents». Kan også bestå av forskjellige typer «Documents».
Elasticsearch: «Index Pattern»	Brukes av Kibana til å vite hvilken informasjon det skal hente ut fra en «Index» i Elasticsearch.
Kibana Query Language (KQL)	Eget språk brukt til å gjøre søk i Kibana.
Unix-tid	Antall sekunder siden 1. januar 1970 UTC. Kommer også i en millisekundversjon.
Autonomt system (AS)	En samling med IP-adresser under en Internettilbyder eller organisasjon som deler rutingregler for disse til resten av Internett.
Numpy	Python-bibliotek som lar deg lagre informasjon I tabeller med svært høy ytelse.
Bibliotek	Ferdig funksjonalitet som kan lastes inn i Python og gjenbrukes til å lage nye programmer. For eksempel «Math» som tilfører Python mer matematisk funksjonalitet.
Dictionary	Datastruktur i Python. Består av attributter og data.
Array	Også kalt tabell på norsk. Enkel datastruktur som brukes til å lagre informasjon.

Tabell 1: Forkortelser og definisjoner

3. Figur- og tabelliste

Tabell 1: Forkortelser og definisjoner 48
Figur 1: Enkel skisse av prosjektet 51
Figur 2: Fysisk eksempel på en RIPE Atlas-probe. Bildet er offentlig eiendom. 52
Figur 3: Eksempel på JSON-formatet 53
Figur 4: Forklaring av crontab 54
Figur 5: Kibana når man åpner det i nettleseren 55
Tabell 2: Eksempel på hvordan de nye loggene kan se ut 56
Figur 6: Eksempel på traceroute fra vanlig datamaskin 58
Tabell 3: Eksempel på logg fra traceroute fra RIPE Atlas 59

4. Om oppdragsgiver og behov

I dette kapittelet skal jeg diskutere bakgrunnen for prosjektet og grunnleggende behov. Disse aspektene av prosjektet er for øvrig mer utfyllende dekket i forstudierapporten.

4.1. Bakgrunn for prosjektet

Uninett er et statlig eid selskap som utvikler og driver nettet som knytter sammen institusjoner innen utdanning og forskning i Norge. De fungerer også som en slags internettilbyder for disse. I samarbeid med lignende organisasjoner fra andre land som NORDUnet og CESNET er Uninett med på noe som heter Dragonlab-prosjektet. Dette prosjektet går ut på at man sammen skal forske på nettet mellom og finne problemer. (Uninett, 2019)

I dette prosjektet skal vi bruke den åpne målingsplattformen RIPE Atlas. RIPE Atlas består av over 10 000 prober og hundrevis av såkalte anker over hele verden. (RIPE Atlas, 2021) Probene sender ut ulike typer trafikk som de sammen logger resultatene fra. Blant annet SSL/TLS, DNS, traceroute og NTP. Såkalte ankere er mer avanserte prober med større kapasitet som også kan fungere som mål for andre prober. Tanken bak denne oppgaven er at man ved å analysere trafikken fra denne plattformen skal klare å identifisere problemer på det globale Internett.

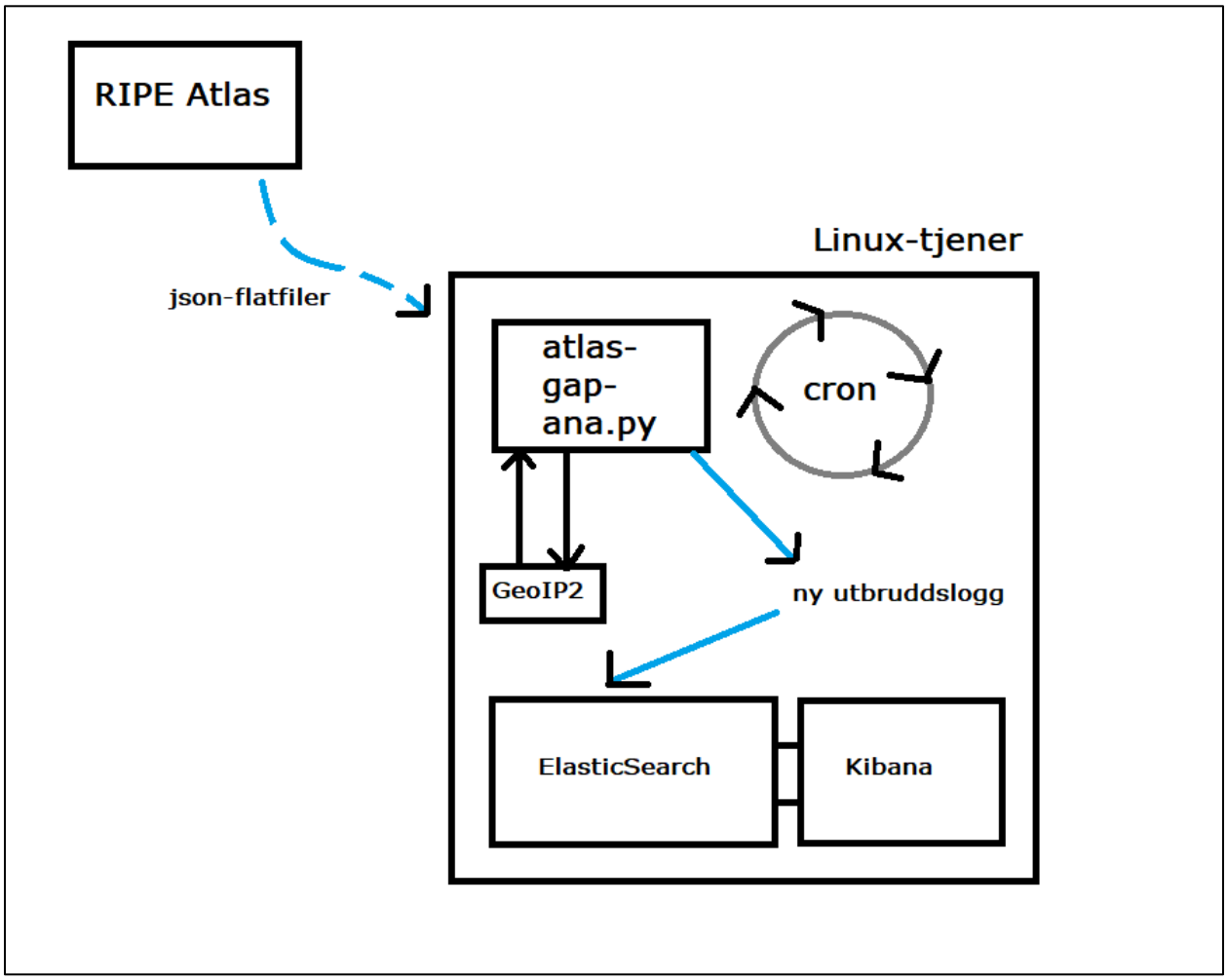
4.2. Om oppgaven

I denne oppgaven skal prosjektgruppen utvikle et program som kan hjelpe med å finne og studere brudd på det globale Internett. Programmet skal kunne settes opp på en Linux-maskin hvor det automatisk kan laste ned og prosessere millioner av traceroute-logger fra målingsplattformen RIPE Atlas. Med dette menes at programmet klarer å bruke informasjonen fra traceroute-resultatene til å finne og beskrive trekk ved overføringsbrudd på rutene. I senere deler av rapporten vil jeg forklare nærmere hvordan dette er mulig.

Programmet skal deretter skrive ut en ny, kortere logg som skal kunne lastes inn i søkeprogramvaren Elasticsearch. Denne er allerede satt opp på Uninetts tjenere. Elasticsearch kan minne litt om SQL, bare at man gjør søkene mot tekstfiler. Hensikten er at man til slutt skal kunne visualisere og studere dataene fra RIPE Atlas i tilleggsprogrammet Kibana.

5. Den overordnede løsningen

I denne delen av oppgaven skal jeg gi en overordnet beskrivelse av oppsettet som programmet skal være en del av. Jeg skal deretter beskrive kravene til selve programvaren og løsningene jeg har valgt. I slutten av kapittelet vil jeg begrunne disse. Under følger en liten skisse av systemet.



Figur 1: Enkel skisse av prosjektet

5.1. Enkel forklaring av systemet

Først har vi en Linux-tjener, avbildet over, med programmet cron som brukes til å få programmet atlas-gap-ana.py til å kjøre med faste mellomrom. Programmet laster millioner av logger ned fra målingsplattformen RIPE Atlas og analyserer disse. For deler av analysen, henter den også inn informasjon fra GeoIP2 som kan knytte IP-adresser til steder, navn og koordinater.

Programmet skriver en utbruddslogg ut når det er ferdig som hentes inn i søkeprogramvaren Elasticsearch. Kibana brukes til å visualisere og analysere informasjonen.

5.2. Eksisterende oppsett

Før arbeidet starter tilbyr RIPE Atlas data fra det globale måleutstyret sitt i JSON-format. Uninett har en Linux-tjener med cron tilgjengelig og GeoIP2 installert. De har allerede Elasticsearch og Kibana gående på Linux-tjeneren jeg får adgang til som de har brukt til å analysere andre data. I hvert underkapittel skal jeg forklare hver del av oppsettet nærmere.

5.2.1. RIPE Atlas

RIPE (Réseaux IP Européens Network Coordination Centre) er organisasjonen som forvalter serier av IP-nummer for Europa, Vest-Asia og den tidligere Sovjetunionen. Et av sideprosjektene deres er RIPE Atlas, en åpen og global målingsplattform for trafikk over Internett. Over 9000 prober og hundrevis av ankere samler inn resultater fra Ping, Traceroute, DNS-søk, SSL-kall, NTP og flere. Det finnes mye gjemt informasjon om Internett i de enorme datamengdene.

Figur 2: Fysisk eksempel på en RIPE Atlas-probe. Bildet er offentlig eiendom.

Per i dag tilbyr RIPE Atlas i hovedsak dataene sine i fire forskjellige former. Fra et grafisk grensesnitt kan man hente ut mye spesifikk informasjon, for eksempel om overføringen mellom en bestemt rute. Det er også mulig å få visualisert dette på nettsidene deres. Videre tilbyr RIPE Atlas også et REST API som man kan bruke til å hente ut spesifikk informasjon. De to siste

alternativene er såkalte Google BigQuery hvor man kan skreddersy søk i SQL. I tillegg finnes flatfiler. Hver morgen legges det ut 15-20 GB for hver av gårsdagens timer.

5.2.2. JSON

JSON (JavaScript Object Notation) er et språkuavhengig standard format som brukes til å sende informasjon mellom ulike programmer. Det har tatt mye av rollen XML hadde tidligere. (Patrizio, 2016) Grunnen til at JSON har JavaScript i navnet er bare fordi strukturen til formatet er inspirert av hvordan data lagres i JavaScript. RIPE Atlas tilbyr dataene sine i form av JSON. De nye loggene programmet mitt skal lage kommer også til å sende informasjonen sin i JSON-format til Elasticsearch. I figuren under kan man se et eksempel på data formatert som JSON.


```
{
    "total_traceroutes":6697844,
    "start_time":1607731200,
    "end_time":1607737189,
    "total_succeeded":5155104,
    "total_failed":1542740,
    "total_probes":11376

}
```


Figur 3: Eksempel på JSON-formatet

En JSON-fil kan bestå av flere slike objekter. Man kan også plassere objekter i objekter. «total_probes» kunne for eksempel inneholdt «yesterday» og «today» med hver sine summer.

5.2.3. Linux

I likhet med Windows, er Linux et operativsystem. Det er kort sagt programvaren som brukes til å forhandle mellom maskinvaren og alle de andre programmene på datamaskinen. Hvis alle programmene prøvde å bruke prosessor samtidig ville det for eksempel blitt krasj. Det ville også vært et sikkerhetsproblem om alle programmer kunne lese hverandres minne.

Mens Windows er utviklet av kommersielle hensyn og på en lukket måte, har Linux blitt bygget sakte opp av frivillige over lang tid. Mange anser løsningene som mer elegante og

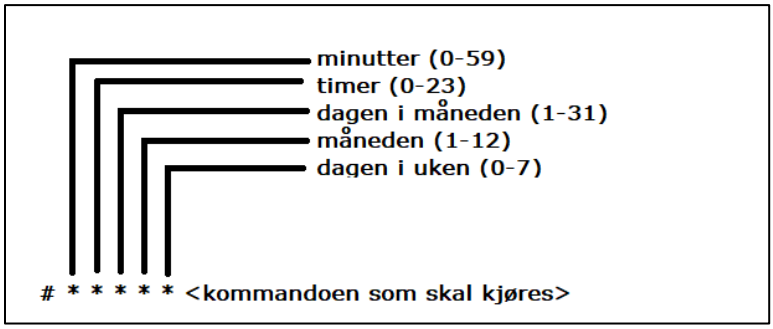
gjennomtenkte. Som bruker har man også større innsyn. Linux har også rykte på seg for å være stabilt og forutsigbart. Linux brukes mye i tjenere, men sjeldent av vanlige brukere.

5.2.4. GeoIP2

GeoIP2 er ulike grensesnitt for å kommunisere med databaser fra det private selskapet Maxmind. Selskapet tilbyr ulike databaser, noen gratis og noen for penger, som lar deg finne ut informasjon om IP-adresser. For eksempel geografiske koordinater. Dette systemet brukes av alt fra å spore opp kriminelle til å skreddersy reklamer for brukere på Internett. I dette prosjektet skal det brukes til å legge inn informasjon om hvor brudd på ruter skjer.

5.2.5. cron

cron et program som brukes til å sette opp andre programmer eller skript til å kjøres med jevne mellomrom. Det finnes i de fleste distribusjoner av Linux. De tidsbestemte oppgavene beskrives i en såkalt crontab. På hver linje i en slik fil beskriver man hvor ofte man vil oppgaven skal gjenta seg på minuttnivå, timenivå, dagsnivå osv. Deretter følger kommandoen som skal kjøres.



Figur 4: Forklaring av crontab

I dette prosjektet er målet å bruke cron til å kjøre programmet med faste mellomrom og deretter laste de nye loggene inn i Elasticsearch og Kibana uten at man må gjøre noe.

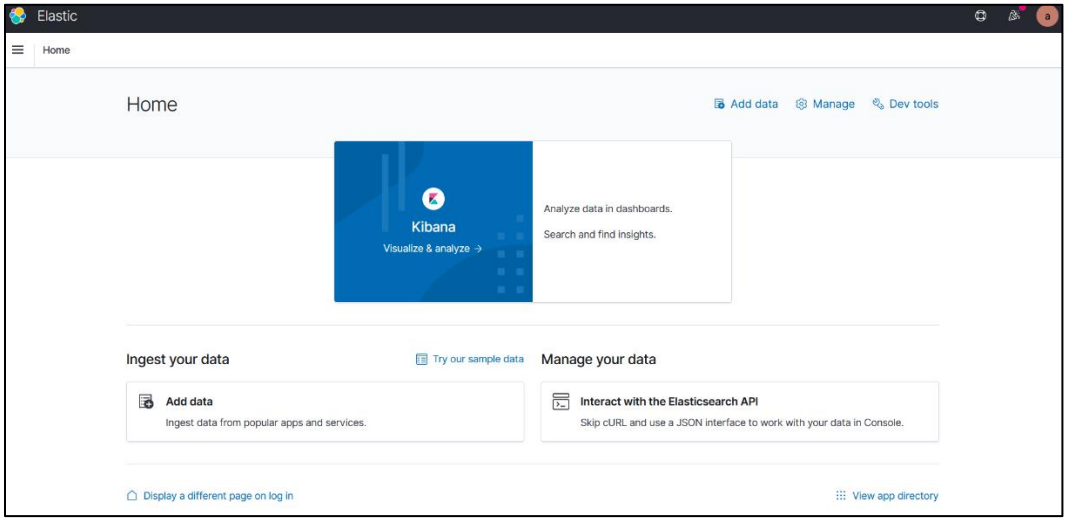
5.2.6. Elasticsearch

Elasticsearch er enkelt sagt søkeprogramvare for store tekstfiler. Det kan minne om en SQL-database fordi man gjør kall mot store mengder lagret informasjon. I motsetning til en SQL-database trenger man ikke lagre informasjonen i tabeller eller ha faste relasjoner mellom informasjonen. Dette betyr at Elasticsearch kan brukes som en såkalt NoSQL-database.

Istedenfor tabeller lagres hver samling med informasjon i et enkelt JSON-objekt kalt «Documents» som består av flere attributter. Dette kan for eksempel være en rapport om et enkelt utfall med attributter som «start_time» og «end_time». Såkalte «mappings» brukes til å definere datatypene i forskjellige attributter. En «Index» utgjør en samling med «Documents». Ikke alle «Documents» i en «Index» trenger å være av samme type. Dette betyr at man kan ha JSON-objekter om både enkelte brudd og alle brudd på en rute i samme «Index». Siden hver «Index» kan bli uendelig stor, kan disse deles opp i flere «Shards» som fordeles på flere tjenere for å få nok plass. Dette kan også øke ytelsen fordi hver tjener søker i sin del samtidig.

5.2.7. Kibana

Kibana er et grensesnitt til Elasticsearch som brukes til å visualisere data. For å kunne gjøre dette lager man såkalte «Index Pattern» som forteller Kibana hva det skal hente fra en Index. Dette kan være en begrenset del av en «Index» eller hele «Index». Når man har et «Index Pattern» kan man gjøre søk i Kibana. Dette kan gjøres både ved å trykke på ting i det grafiske grensesnittet og ved å skrive kall i KQL (Kibana Query Language) eller Apache Lucene. Når man har lagret et søk, for eksempel de lengste bruddene, kan man enkelt få dette visualisert.



Figur 5: Kibana når man åpner det i nettleseren

5.3. Krav til programvaren

Arbeidsgiver har ikke satt noe krav til hvilken teknologi som skal brukes til å utvikle programmet. Det viktige er at det kan kjøres på en Linux-tjener hvor programmet selvstendig kan laste ned logger med traceroutes fra RIPE Atlas. Deretter skal disse loggene analyseres for brudd og resultatet skal skrives ut til en ny JSON-fil. Så langt har det kommet forespørsel om tre

forskjellige typer rapporter som skal skrives ut til JSON-filen. Den ene er en rapport som gir en generell beskrivelse av hele loggfilen, for eksempel traceroutes totalt og antall prober. Den andre rapporten skal fortelle om antall brudd på en rute over en lengre periode. Den siste typen rapport skal beskrive unike brudd. Under kan man se et eksempel under arbeid på hver. Først den totale rapporten, deretter rutesammendraget og til slutt en unik bruddrapport.

{"total_traceroutes":6697844,"start_time":1607731200,"end_time":1607737189,"total_succeeded":5155104,"total_failed":1542740,"total_probes":11376}
{"id":"173.248.145.27-to-147.87.5.254","record_type":"Summary","dst_addr":"147.87.5.254","src_addr":"173.248.145.27","prb_id":6835,"traceroutes":4,"incomplete":0,"outage_count":1,"total_outage_duration":894,"cause":[[["No reply"],["No reply"],["No reply"]]]}
{"id":"173.248.145.27-to-147.87.5.254","record_type":"Detail","order":1,"start_time":1612362907,"end_time":1612363801,"failures":1,"duration":894,"last_reply_from":"64.124.44.234","AS_number":"AS6461","AS_name":"Zayo Bandwidth","location":"United States, Ward","latitude":40.0726,"longitude":-105.508,"error":[[["No reply"],["No reply"],["No reply"]]]}

Tabell 2: Eksempel på hvordan de nye loggene kan se ut

Her kan man blant annet se at loggene henter informasjon fra GeoIP2 og regner ut lengden på brudd. Oppdragsgiver har understreket at det er vanskelig å si på forhånd nøyaktig hva det ferdige resultatet blir og hver detalj som skal være med. Arbeidet er en del av forskningsprosjekt og man vet ikke hva som potensielt ligger i disse dataene før man har jobbet med dem. Man må rett og slett prøve seg frem og se hva som gir interessante funn. Arbeidsgiver har også påpekt at det er vanlig å jobbe på denne måten i arbeidslivet, såkalt smidig arbeidsmetodikk, hvor man har mange leveranser og fortløpende får tilbakemeldinger.

Et annet viktig krav er at programmet skal kunne tåle de store datamengdene og skalere bra hvis man f. eks. ønsker seg data fra 24 timer. En enkelt time med traceroute-logger fra RIPE Atlas er 15-20 GB. Det er et pluss om programmet benytter lite minne og prosessorkraft. Et ekstra ønske fra oppdragsgiver et at programmet skal kunne kartlegge veien en pakke tar på en rute og eventuelle sideveier som oppstår i forbindelse med avlastning, brudd, o. l. Disse kartene skal deretter kunne lastes inn fra Kibana inn i eget program som kan visualisere dette.

5.4. Valg og begrunnelser

I denne delen skal jeg til slutt beskrive valgene jeg selv har tatt i prosjektet og hvorfor.

5.4.1. Python

Jeg har valgt å skrive programmet i Python. Det er for det første fordi det er et språk jeg har brukt mye før og er komfortabel med. For det andre er det et ganske populært språk som det finnes mye informasjon om på nettet og mange biblioteker til. Den tredje grunnen er at det er et veldig populært språk å bruke spesifikt til å analysere store datamengder. Syntaksen gjør det svært godt egnet til å manipulere data. Det er også et raskt språk sammenlignet med mange av konkurrentene som MatLab. Fordi det er mye brukt til å jobbe med store datamengder, finnes det også mange biblioteker og mye informasjon om hvordan man gjør dette. (Zhidkov, 2020)

5.4.2. Flatfiler

I løsningen min har jeg valgt å ta utgangspunkt i flatfilene RIPE Atlas legger ut når jeg skal lage logger til brudd. Jeg tror ikke det grafiske grensesnittet deres er særlig fleksibelt eller godt egnet til å grave i dybden på dataene deres. REST API-et virker også ganske stivbeint ut ifra det jeg har sett. Det virker best egnet til å hente ut mindre, helt spesifikke målinger. Flatfilene deres lar meg på den andre siden hente ut en håndterlig mengde data om gangen (15-20 GB) og behandle denne på en fri måte. Det virket godt egnet til å løse oppgaven. Google BigQuery virker interessant å utforske videre som en erstatning til flatfilene dersom det skulle bli tid. Hvis dataene kan hentes ut mer fleksibelt kan dette potensielt gi nye muligheter.

6. Detaljert løsningsbeskrivelse

I dette kapittelet skal jeg forklare og diskutere løsningene programmet krever i større detalj.

6.1. Finne brudd hjelp av traceroutes

Sentralt for programmet er at det skal klare å finne brudd ved å lese og sammenligne informasjon fra traceroute-logger. I dette underkapittelet vil jeg forklare hvordan.

6.1.1. Hvordan traceroute virker

Når man kjører traceroute fra en maskin, sendes det treping-pakker mot måladressen med beskjed om at pakkene skal droppes etter ett hopp. Når den første maskinen dropper pakken, sender den vanligvis en melding tilbake om at dette ble gjort. På denne måten får man svar fra den første ruteren på veien til målet med informasjon om denne. Deretter øker man hoppgrensen med én og sender en ny pakke. Denne gangen får man svar fra den andre ruterenpå veien. Slik fortsetter man til man kommer frem til målet eller det oppstår en feil. På denne måten kan man bruke traceroutes til å kartlegge en rute over Internett.

```
C:\Users\aleks>tracert nrk.no

Tracing route to nrk.no [84.208.4.147]
over a maximum of 30 hops:

  1     2 ms     1 ms     1 ms  INTENOSMB [192.168.0.1]
  2     3 ms     2 ms     2 ms  pc206-1.ktv.no [109.108.206.1]
  3     2 ms     2 ms     2 ms  82.194.192.5
  4     3 ms     2 ms    16 ms  82.194.192.11
  5     3 ms     3 ms     2 ms  cm-84.208.27.1.getinternet.no [84.208.27.1]
  6     9 ms     9 ms     9 ms  a23-5-48-1.deploy.static.akamaitechnologies.com [23.5.48.1]
  7    10 ms    11 ms     9 ms  cm-84.208.4.147.getinternet.no [84.208.4.147]
```

Figur 6: Eksempel på traceroute fra vanlig datamaskin

Over kan man se et eksempel på en traceroute kjørt fra en vanlig maskin. Hver linje representerer ett hopp med en beskrivelse av hvor lang tid det tok å få svar. Å registrere et svar fra den fjerde ruteren på veien, 82.194.192.11, tok for eksempel 3 ms, 2 ms og 16 ms.

6.1.2. Hvordan en RIPE Atlas-logg ser ut

Hver flatfil fra RIPE Atlas inneholder en linje for hver traceroute utført av probene og ankrene på målingsplattformen deres. En vanlig loggfil er 15-20 GB og inneholder 7 millioner traceroutes. Under kan man se et eksempel på en enkelt linje fra en slik fil.

{"fw":5020,"mver":"2.2.1","lts":44,"endtime":1612778430,"dst_name":"5.20.0.41","dst_addr":"5.20.0.41","src_addr":"212.122.95.149","proto":"ICMP","af":4,"size":48,"paris_id":8,"result":[{"hop":1,"result":[{"from":"212.122.95.190","ttl":255,"size":28,"rtt":0.516},{"from":"212.122.95.190","ttl":255,"size":28,"rtt":0.498},{"from":"212.122.95.190","ttl":255,"size":28,"rtt":0.501}]},{"hop":2,"result":[{"from":"84.240.1.241","ttl":254,"size":28,"rtt":1.125},{"from":"84.240.1.241","ttl":254,"size":28,"rtt":1.176},{"from":"84.240.1.241","ttl":254,"size":28,"rtt":1.034}]},{"hop":3,"result":[{"from":"84.240.1.239","ttl":253,"size":28,"rtt":0.826},{"from":"84.240.1.239","ttl":253,"size":28,"rtt":0.67},{"from":"84.240.1.239","ttl":253,"size":28,"rtt":0.719}]},{"hop":4,"result":[{"from":"77.241.206.10","ttl":61,"size":76,"rtt":1.001},{"from":"77.241.206.10","ttl":61,"size":76,"rtt":0.935},{"from":"77.241.206.10","ttl":61,"size":76,"rtt":1.223}]},{"hop":5,"result":[{"x":"*"},{"x":"*"},{"x":"*"}]},{"hop":6,"result":[{"x":"*"},{"x":"*"},{"x":"*"}]},{"hop":7,"result":[{"x":"*"},{"x":"*"},{"x":"*"}]},{"hop":8,"result":[{"x":"*"},{"x":"*"},{"x":"*"}]},{"hop":9,"result":[{"x":"*"},{"x":"*"},{"x":"*"}]},{"hop":255,"result":[{"x":"*"},{"x":"*"},{"x":"*"}]}],"msm_id":24442628,"prb_id":19789,"timestamp":1612778358,"msm_name":"Traceroute","from":"212.122.95.149","type":"traceroute","group_id":24442628}

Tabell 3: Eksempel på logg fra traceroute fra RIPE Atlas

Det er veldig mye informasjon pakket inn her. Et viktig attributt er «timestamp» på slutten som forteller når tracerouten ble startet. Tiden er beregnet i UNIX-tid, som er antall sekunder etter midnatt 1. januar 1970 UTC. I «endtime» kan vi se når tracerouten var ferdig. Også interessant er «from» som forteller oss avsenderen 212.122.95.149 og «dst_addr» som forteller oss mottakeren 5.20.0.41. Mesteparten av loggen er «result» som er delt opp i «hop 1», «hop 2» osv. Hvert hopp tilsvarer resultatet fra en linje i en vanlig traceroute.

6.1.3. Hvordan man registrer et brudd på en rute

I logglinjen over kan man se at tracerouten aldri rakk frem til målet. Den ble sist sett ved 77.241.206.10. Alle svar i etterkant er «"x":"*"» som betyr at man aldri fikk svar. Hvordan kan man bruke en slik logg til å si noe om statusen til en rute? Det man må gjøre er å samle sammen flere traceroute-logger mellom to adresser. For eksempel 212.122.95.149 og 5.20.0.41. Dette definerer som en rute. Hver gang en traceroute kommer frem til målet, kan man gjette på at

ruten er oppe i dette tidsrommet. Dersom man etter flere vellykkede traceroutes får en serie mislykkede traceroutes som den over, kan vi gjette på at det har vært et brudd i perioden.

Når programmet har identifisert et slikt brudd kan det hente ut informasjon, som for eksempel når man først registrerte at bruddet startet og når man registrerte at det var over, og skrive dette ut til en ny logg. Fra 15-20 GB og i underkant av 7 millioner linjer bør dette bli redusert til en logg på et par MB og i underkant av 100 000 linjer som kan lastes inn i Elasticsearch.

6.2. Innhenting av informasjon

Det finnes ikke én måte å hente ut all informasjonen man trenger for de nye loggene. For å telle antallet linjer, kan man for eksempel ha en variabel som man inkrementerer hver gang man leser inn en ny linje. Skal man finne ut hvor mange forskjellige prober logger kommer fra, er man nødt til å huske hvilke identifikasjonsnummer man har sett før.

Mer komplisert blir det for eksempel å finne ut i hvilket AS en traceroute sist ble sett før et brudd. Internett er delt opp i såkalte autonome systemer (AS) etter hvilke Internettleverandører som har hovedansvar for å fortelle andre AS-er hvordan man ruter til adressene som er deres ansvar. (Rekhter m. fl., 1995) For å finne AS-et hvor traceroute-pakken sist ble sett må man først finne bruddet og den siste registrerte IP-en. Deretter må man sjekke denne IP-adressen mot GeoIP2 og skrive AS-nummeret ut riktig sted i den nye loggen.

Når det gjelder for eksempel feilene bak et brudd, må man samle inn disse og skrive dem ut både til bruddrapportene og rutesammendragene. Feilene må også oversettes fra koder til noe som er lesbart av mennesker. Koden «H» står for eksempel for «Destination unreachable». (RIPE Atlas Documentation, 2021) Her er det mange feller. Det kommer ofte «late replies» og man må passe på at det ikke er de man henter feilene fra. Det kan også være såkalte «duplicates» hvor den samme feilmeldingen blir sendt flere ganger.

All informasjonen som skal vises i loggene er ikke definert på det nåværende tidspunktet. En del av den praktiske utføringen av oppgaven, som jeg kommer tilbake til i Driftsrapporten, blir også å finne løsninger på alle disse problemene. Jeg kommer derfor ikke til å prøve å gå gjennom hvordan man samler inn all informasjonen programmet trenger. I stedet håper jeg dette underkapittelet har gitt en viss idé om hvordan disse problemene må løses.

6.3. Lagring og sortering

Som diskutert i kapittel 5.1. kan man ikke si noe om et brudd basert på ett traceroute-resultat. I stedet er man nødt til å sammenligne resultatet fra flere traceroutes over tid for å kunne gjette på brudd. Jo flere traceroutes man har i en periode, jo sikrere er man på statusen til ruten.

En utfordring er imidlertid at traceroutes ligger hulter til bulter i flatfilene som skal analyseres. Et eksempel på dette finner man i flatfilen med loggen fra kl. 9 til kl. 10, 8. februar 2021. Linje 112 684 i loggen beskriver hva som skjer fra ca. kl. 9:49 til 9:50. Linje 140 025 (som altså kommer lenge etterpå) beskriver hva som skjer ca. kl. 9:04 til kl. 9:05. For å være helt sikker på at man har fått med alle traceroutes fra en rute, må man altså lagre alle traceroutes for en rute fra hele loggfilen. Deretter er man nødt til å sortere disse på tid. Først når man har gjort dette kan man med sikkert begynne å beregne brudd på ruten og trekke ut informasjon fra disse.

6.4. Skalering og ytelse

Ifølge veileder fra Uninett har det ikke noe å si for Elasticsearch/Kibana om ruteoppsummeringene beskriver 1 time eller 24 timer. Det finnes fordeler og ulemper med begge deler, så man kan like gjerne ta det som er enklere. Flatfilene inneholder heller ikke logger fra andre timer enn sin egen. Med andre ord er det aldri behov for å ta med seg informasjon fra en annen time når man skal analysere en time i programmet. Hvis man ber programmet analysere 24 timer, gjør det seg bare ferdig med 1 time før det går til den neste.

Selv om man bare skal analysere 15-20 GB om gangen, er det likevel ganske store data. Når jeg har testet med å lagre informasjonen som trengs i Python-dictionaries, bruker fort programmet mer enn 3,5 GB minne før det blir ferdig med én flatfil. Det virker også som om det bruker ganske mye prosessorkraft. En mulighet er å endre datastruktur til såkalte NumPy-arrays. Denne datastrukturen burde være langt raskere og bruke mindre minne. (W3schools, 2021)

En utfordring er at NumPy-arrays kun kan ha tall som indeks. Det gjør det vanskelig å finne akkurat den arrayen med informasjonen man trenger. Løsningen på dette kan være at man har en dictionary som oversetter mellom rutenavnet og indeks. En annen mulighet er at man dynamisk navngir NumPy-arrayen etter rutenavnet. Problemet med dette er at det visstnok er utrolig tregt for et kjørende program å lage variabler, men det kan være verd å utforske.

Andre muligheter er en løsning som bruker harddisken mer. Man kan for eksempel skrive ruter ut fra minnet når det er lenge siden de har behøvd å bli oppdatert. Enda en ting man kan gjøre er å skrive all informasjon om en rute til samme plass på harddisken. Deretter kan man til slutt hente inn en rute om gangen og analysere med svært lav minnebruk. En siste mulighet som bør nevnes er at det kan være gjemt en struktur i flatfilene vi kan utnytte. Det finnes indikasjoner på at de fleste traceroutes til en rute dukker opp innenfor en viss mengde linjer selv om det er i feil rekkefølge. Hvis jeg kan finne ut sikkert at alle traceroutes for en bestemt rute dukker opp innen 100 000 linjer, kan jeg gjøre meg ferdig med ruter fortløpende og tømme minnet.

Det finnes også mange andre store og små måter å optimalisere programmet på. For eksempel kan man få til at programmet laster ned nye logger samtidig som det jobber med gamle logger. På denne måten slipper man vente på at programmet skal bli ferdig med å få ned og pakke ut

filene mellom hver logg som skal analyseres. Det er heller ikke sikkert den første løsningen man finner på for å sortere traceroutes eller finne feil i brudd, er de mest effektive. Et program kan alltid yte bedre og det er et spørsmål og tid og prioritering hvor langt man vil gå.

6.5. Å skrive ut informasjon

Når man har funnet all informasjonen om en rute og sortert dem, kan man gå gjennom informasjonen som finnes og trekke ut de nødvendige dataene. For eksempel lengde på bruddene. Deretter bruker man bare et bibliotek til å skrive ut de forskjellige rapportene på en linje hver. Her er det også mulig å foreta filtrering. Det er for eksempel kanskje ikke så interessant å ta med seg ruter med bare 1-3 traceroutes eller uten noen brudd. Dette kan utgjøre en veldig stor forskjell for hvor mye data Elasticsearch/Kibana får inn. Riktignok er det også mulig dette er oppgaver Elasticsearch og Kibana kan løse like greit.

6.6. Biblioteker og oppsett

I dette underkapittelet vil jeg beskrive støtteprogramvare og biblioteker jeg vil bruke.

6.6.1. Pip

Pip er det standard pakkesystemet til Python. Det gjør det enkelt å laste ned, installere og avinstallere biblioteker man vil bruke i Python. Dersom biblioteket man ønsker å bruke har avhengigheter, holder også Pip styr på dette for deg. Pip kan i tillegg brukes til å lage virtuelle miljøer slik at det er lett å ha kontroll på hva et enkelt Python-prosjekt benytter seg av pakker og få dette overført for eksempel fra en arbeidsmaskin til en tjener.

6.6.2. Ujson

Ujson, eller UltraJSON, er et Python-bibliotek skrevet i C som er spesiallaget for å kunne lese inn informasjon fra JSON-filer eller skrive informasjon til JSON-filer så raskt som mulig.

6.6.3. Urllib

Urllib er et lite bibliotek som kan brukes til å laste ned, åpne og lese filer fra Internett. Tanken er å bruke dette til å la programmet laste ned flatfiler fra RIPE Atlas selv.

6.6.4. Math, Time, Datetime

Diverse pakker som brukes til datoer, tid og mattefunksjoner som ikke er med i Python i utgangspunktet. For eksempel funksjonen «ceiling» og «floor» som runder tall opp og ned.

6.6.5. bz2, shutil

Flatfilene til RIPE Atlas er komprimert i bz2-format. For å kunne pakke ut disse i koden trenger jeg bz2-biblioteket, i tillegg til shutil som brukes til å skrive innholdet til en ny fil.

6.6.6. NumPy

NumPy er et bibliotek som legger til støtte for store, flerdimensjonale arrays og matriser i Python sammen med mange matematiske funksjoner man kan bruke på disse. Fordelen med NumPy er et at det er raskt og bruker lite ressurser sammenlignet med alternativene. Ulempen er at det er ment for tallanalyse og derfor på mange områder er svært begrenset.

6.6.7. GeoIP2, Pygeoip

GeoIP2 er et offisielt bibliotek som brukes til å kommunisere med GeoIP-databaser i MMDB-format. Pygeoip lar meg hente AS-informasjonen fra DAT-filene hvor Uninett har dette lagret.

6.6.8. Sys, OS

Sys og OS er biblioteker som brukes til å hente inn parameter i programmet fra Linux og skrive ut feilmeldinger. For eksempel hvis man kjører programmet med «python3 atlas-gap-ana.py 2021-02-03T1400» kan man bruke sys til å hente ut «2021-02-03T1400». Dersom dette blir skrevet feil, kan man bruke OS til å fortelle brukeren om dette i terminalen.

6.6.9. cProfile

cProfile er en modul som brukes til å analysere kjøretiden til Python-kode. Når man kjører en funksjon gjennom cProfile (eller hele koden) forteller den oss hvor mange sekunder det tok å komme til slutten. cProfile lager også en rapport over hvor mye tid som ble brukt i de forskjellige delene av programmet. Dette kan hjelpe oss forbedre programmets ytelse.

6.6.10. memory-profiler

Modulen memory-profiler fungerer ganske likt som cProfile, bortsett fra at det analyserer minnebruken istedenfor tidsbruken. Dette kan også hjelpe oss forbedre ytelsen og få programmet til å skalere bedre. En utfordring med slike moduler er at de ikke nødvendigvis greier ta hensyn til minnebruken til importerte moduler i C som ujson.

7. Referanser

Patrizio, Andy. (2016) XML is toast, long live JSON. CIO. Tilgjengelig fra: https://www.cio.com/article/3082084/xml-is-toast-long-live-json.html (Hentet: 19. februar).

Rekhter, Y., Li, T. (1995) A Border Gateway Protocol 4 (BGP-4). Tilgjengelig fra: https://tools.ietf.org/html/rfc1771 (Hentet: 19. februar).

RIPE Atlas. (2021) Global RIPE Atlas Network Coverage. Tilgjengelig fra: https://atlas.ripe.net/results/maps/network-coverage/ (Hentet: 19. februar 2021).

Ripe Atlas Documentation. (2021) RIPE Atlas – Raw data structure documentation. Tilgjengelig fra: https://atlas.ripe.net/docs/data_struct/ (Hentet: 19. februar 2021).

Uninett. (2019) *Dragonlab*. Tilgjengelig fra: https://www.uninett.no/node/1819 (Hentet: 16. mai 2021).

W3schools. (2021) NumPy Introduction. Tilgjengelig fra: https://www.w3schools.com/python/numpy_intro.asp (Hentet: 19. februar 2021).

Zhidkov, Roman. (2020) Why Python is Essential for Data Analysis. Tilgjengelig fra: https://www.rtinsights.com/why-python-is-essential-for-data-analysis/ (Hentet: 19. februar 2021).

Driftsrapport

Innholdsfortegnelse

1. Om driftsrapporten og prosjektet 69
2. Forkortelser og definisjoner 70
3. Figur- og tabelliste 73
4. Oversikt: Prosjektet i flere deler 77
5. Første del: Selve programmet 78
   5.1 Forutsetninger og begrensninger 78
   5.2 Installasjon av programmet 78
   5.3 Kjøre programmet 79
   5.4 Hva programmet gjør 80
      5.4.1 Om loggene fra RIPE Atlas 80
      5.4.2 Loggene atlas-gap-ana.py lager 84
         5.4.2.1 Om loggoppsummeringene 84
         5.4.2.2 Om ruteoppsummeringene 85
         5.4.2.3 Om topologiene i ruteoppsummeringene 87
         5.4.2.4 Om utfallsbeskrivelsene 88
   5.5. Hvordan programmet virker 90
      5.5.1 Hvordan RIPE Atlas-loggene leses inn i programmet 90
         5.5.1.1 Den tidligere versjonen av main() som brukte tråder 90
         5.5.1.2 Den nye versjonen av main() 94
      5.5.2 Hvordan den nødvendige informasjonen lagres 95
      5.5.3 Hvordan informasjonen analyseres og skrives ut 105
      5.5.4 Hvordan programmet samler inn og bruker topologiene 119
6. Andre del: Kibana og Elasticsearch 129
   6.1. Lage «index» og «index pattern» 129
   6.2. Bruke «Discover» 132
   6.3. Bruke «Visualize» og «Dashboard» 138
7. Tredje del: Oppsett i Linux 147
   7.1. Automatiske skript 147
   7.2. Bruk av «crontab» 150
8. Fjerde del: Eksempel på analyse av utfall 152

8.1. Et mislykket forsøk 152
8.2. Et mer vellykket forsøk 153
9. Referanser.....155

1. Om driftsrapporten og prosjektet

Dette dokumentet er en driftsrapport, utarbeidet i forbindelse med Aleksandra Jekic sin bacheloroppgave ved NTNU våren 2021. Arbeidet er gjort i samarbeid med Uninett. Veileder fra NTNU er Jostein Lund mens Olav Kvittem representerer Uninett.

Oppgaven er å sette opp et system for å forske på problemer med videresending av data over det globale Internett. Dette er noe som påvirker de fleste daglig. For eksempel når video eller lyd stopper opp i noen sekunder under en nettkonferanse. I teorien burde dette aldri skje. Oppgaven er en del av DragonLab-prosjektet hvor Uninett i samarbeid med andre organisasjoner fra hele verden som tilbyr Internett til forskning og skole jobber med å forstå og forbedre det globale Internett. Tidligere forskning finnes på Uninetts nettsider. (Uninett, 2021)

Systemet skal laste ned logger fra millioner av traceroutes lagret på den åpne, globale målingsplattformen RIPE Atlas. Disse loggene skal deretter brukes til å identifisere brudd på ruter og rapportere om disse i en ny logg. Videre skal de nye loggene lastes inn i Elasticsearch, en søkeprogramvare allerede satt opp på en av Uninetts tjenere. Ved hjelp av Elasticsearch og visualiseringsprogrammet Kibana skal man deretter kunne studere dataene.

Driftsrapporten skal forklare hvordan prosjektet er satt opp og løst i detalj. Det skal kunne fungere som en bruksanvisning. Det skal også kunne brukes av de som eventuelt skal jobbe videre med systemet. For eksempel vil man i dette dokumentet kunne lese og få en oversikt av begrunnelser for ulike tekniske valg som vil kunne være til hjelp i videre arbeid.

For en mer generell, overordnet beskrivelse av prosjektet henvises man til designrapporten. For en mer organisatorisk beskrivelse av prosjektet henvises man til forstudierapporten.

2. Forkortelser og definisjoner

Begrep	Beskrivelse
REST API	Oppsett hvor en klient henter informasjon fra en tjener ved hjelp av HTTP.
HTTP	Protokollen mest brukt til å utveksle informasjon på Internett.
Flatfil	Data er lagret i en stor fysisk fil slik at det er bare én oppføring på hver linje.
ICMP (Internet Control Message Protocol)	Protokoll som brukes til å teste nettverk og overføre meldinger om feil.
Ping	Hjelpeprogram som sender ut et ekko med ICMP til en bestemt adresse. Mottakeren sender et svar tilbake og Ping registrer hvor lang tid turen frem og tilbake tok.
Traceroute/tracert	Kommando som sender ut ping med stadig høyere «Time-To-Live» (altså antall hopp før pakken skal forkastes). Tanken er at man skal pinge hver maskin til målen, altså «spore ruten», men blir ikke alltid slik i praksis siden veien kan endre seg underveis.
SSL/TLS	Protokoller som tilbyr kryptert kommunikasjon.
DNS	Tjeneste som kobler domenenavn til IP-adresser. Datamaskiner gjør rutinemessig kall til DNS-tjenere for å finne navn.
NTP	Protokoll som brukes til å synkronisere tiden på forskjellige datamaskiner.
RIPE	Réseaux IP Européens Network Coordination Centre. Forvalter IP-nummerserier i Europa, Vest-Asia og den tidligere Sovjetunionen.
RIPE Atlas	Åpen målingsplattform som samler inn data fra det globale Internett ved hjelp av prober.
Probe	Innretning som kobles til en ruter og sender ut traceroutes og andre kall over Internett.
Anker	Store prober som også kan fungere som mål for andre prober og anker.
Elasticsearch	Søkeprogramvare for store tekstfiler.
Kibana	Grensesnitt for Elasticsearch som brukes til å gjøre søke og lage visualiseringer av dem.

SQL	Structured Query Language. Kommandospråk som brukes til å hente, endre, legge inn osv. data i en klassisk relasjonsdatabase.
NoSQL	Database som ikke benytter seg av tabeller eller relasjoner. Elasticsearch kan brukes som en NoSQL-database.
JSON	Javascript Object Notation. Filformat som brukes til å sende data mellom ulike systemer.
cron	Linux-program brukt til å planlegge faste oppgaver.
GeoIP2	System for å hente stedsinformasjon o. l. om IP-adresser fra egne databaser.
atlas-gap-ana.py	Programmet som utvikles i dette prosjektet.
Google BigQuery	Database man kan ha i skyen. Støtter mye avansert funksjonalitet som maskinlæring.
XML	eXtensible Markup Language. Filformat brukt til å utveksle informasjon mellom forskjellige systemer. JSON brukes mye istedet i dag.
Elasticsearch: «Documents»	En enkel bit med informasjon i Elasticsearch. For eksempel en enkelt bruddbeskrivelse.
Elasticsearch: «Mappings»	Forteller Elasticsearch hvordan det skal lese attributtene i et «Document».
Elasticsearch: «Index»	Samling «Documents». Kan bestå av forskjellige typer «Documents».
Elasticsearch: «Index Pattern»	Brukes av Kibana til å vite hvilken informasjon det skal hente ut fra en «Index» i Elasticsearch.
Kibana Query Language (KQL)	Eget språk brukt til å gjøre søk i Kibana.
Unix-tid	Antall sekunder siden 1. januar 1970 UTC. Kommer også i en millisekundversjon.
Autonomt system (AS)	En samling med IP-adresser under en Internettilbyder eller organisasjon som deler rutingregler for disse til resten av Internett.
Numpy	Python-bibliotek som lar deg lagre informasjon I tabeller med svært høy ytelse.
Bibliotek	Ferdig funksjonalitet som kan lastes inn i Python og gjenbrukes til å lage nye programmer. For eksempel «Math» som tilfører Python mer matematisk funksjonalitet.
Dictionary	Datastruktur i Python. Består av attributter og data.

Array	Også kalt tabell på norsk. Enkel datastruktur som brukes til å lagre informasjon.

Tabell 1: Forkortelser og definisjoner

3. Figur- og tabelliste

Figur 1: Enkel skisse av systemet 77
Figur 2: Programmet vil ha problemer med å hente inn parameteret fra Windows 78
Figur 3: Innholdet i requirements.txt 79
Figur 4: Standard stier brukt av programmet 80
Figur 5: Tjeneren med noen av loggene åpnet i en nettleser 80
Figur 6: Forklaring av traceroute 81
Tabell 3: Eksempel på en traceroute lagret av RIPE Atlas 82
Tabell 4: Eksempel på RIPE Atlas-innpakning av traceroutes 82
Tabell 5: En fullstendig RIPE Atlas-logg med både traceroute og innpakning 83
Figur 7: Bruk av programmet glogg 83
Figur 8: Den originale loggen og loggen programmet lager 84
Tabell 6: Eksempel på en "Log Summary" 84
Tabell 7: Forklaring av attributtene i "Log Summary" 85
Tabell 8: Eksempel på en "Summary" 86
Tabell 9: Forklaring av attributtene i "Summary" 87
Tabell 10: Utdrag fra en "Summary"-topologi 88
Tabell 11: Eksempel på en "Detail"-logg 88
Tabell 12: Forklaring av attributtene i "Detail" 89
Figur 9: Forenklet main() 90
Figur 10: Mer forenklet main() 90
Figur 11: Analyse av én logg med tidligere versjon av main() 91
Figur 12: Kodesnutt fra getLogName() 91
Figur 13: Kodesnutt fra downloadLog() 91
Figur 14: Tidligere versjon av main() med flere tråder 92
Figur 15: Kodesnutt fra downloadLogs(), ikke det samme som downloadLog() 93
Figur 16: Den forenklede versjonen av main() igjen 94
Figur 17: Hvordan analyzeLog() åpner linken som en fil 94
Figur 18: Linjene lastes inn og dekomprimeres, én om gangen 94
Figur 19: Kjøretiden til dekomprimering målt med cProfile 95
Figur 20: Kjøretiden til ujson.load() målt med cProfile 95
Figur 21: Teller antall traceroutes i hele RIPE Atlas-loggen 95
Figur 22: Holder styr på antall unike prober 95
Figur 23: getUniquePair() og checkForProbeFail() 96
Figur 24: Funksjonen getUniquePair() 96
Figur 25: Sjekker om proben klarte starte tracerouten 97
Tabell 12: Eksempel på en feil I startproben 97
Tabell 13: Eksempel på en feil etter ett hopp 97
Figur 26: analyzeLog() sjekker hvordan hver logglinje skal lagres 98
Figur 27: Funksjonen initialize() 98
Figur 28: Memory initialiseres av analyzeLog() før den begynner å lese linjer 99
Figur 29: checkSuccess() sjekker om en traceroute nådde målet sitt 99

Figur 30: Andre del av initialize()-funksjonen 100
Figur 31: findLastReplyIP() finner den siste IP-en en mislykket traceroute fikk svar fra 100
Figur 32: Første del av checkForErrors() 101
Tabell 14: Eksempel på svar vi henter feilmeldinger fra 102
Figur 33: explainError() gjør feilkoder til tekst som er mer lesbar for mennesker 102
Figur 34: Andre del av checkForErrors() 103
Tabell 15: Eksempel på svar uten feilmeldinger 103
Tabell 16: Eksempel på feil i ruterne underveis 103
Tabell 17: Eksempel på svar med altfor mange "No Reply" 104
Figur 35: Funksjonen updateMemory() 104
Figur 36: Funksjonen sortMemoryByTimestamps() 105
Figur 37: Kjøretiden til sortMemoryByTimestamps beregnet av cProfile() 105
Figur 38: Koden i analyzeLog() etter at hver linje har blitt lest 105
Figur 39: Koden til cleanUp() 106
Figur 40: countOutages() teller antall utfall på en rute 107
Figur 41: analyzeEachOutage() initialiseres 107
Figur 42: analyzeEachOutage() jobber seg gjennom ett og ett brudd 108
Figur 43: Blar gjennom traceroutes til vi finner starten på utfallet 108
Figur 44: Lagrer informasjon om starten på utfallet 108
Figur 45: Lagrer informasjon om bruddet mens det foregår 109
Figur 46: Setter duration til "Unknown" hvis utfallet ikke har en slutt 109
Figur 47: Teller antall ufullstendige traceroutes 110
Figur 48: Plusser sammen varighet 110
Figur 49: Lagrer alt vi vet om det enkelte utfallet et sted hvor vi finner det igjen 110
Figur 50: Lagrer alt vi vet om alle utfallene knyttet til denne ruten 111
Figur 51: Lager ruteoppsummeringen som skal skrives ut som JSON 111
Figur 52: Sjekker at en probe kun har brukt én offentlig IP 112
Figur 53: JSON-loggen begynner å ta form 112
Figur 54: Funksjonen countErrorTypes() 113
Tabell 18: Typisk output fra countErrorTypes() 113
Figur 55: Lager utfallsoppsummeringene 114
Figur 56: Henter ut GeoIP-informasjon 114
Figur 57: Henter ut navn på sted og koordinater 115
Figur 58: AS-informasjonen hentes fra en seperat database 115
Figur 59: Utfallsoppsummeringen fylles ut 116
Figur 60: Antall unike prober blir telt 116
Figur 61: Sletter koordinatattributtet dersom det er ukjent 116
Figur 62: Første del av classifyOutage() 117
Figur 63: Andre del av classifyOutage() 117
Figur 64: Tredje del av classifyOutage() 118
Figur 65: Logg for hele timen 118
Figur 66: Kommer tilbake til topologi i neste kapittel 118
Figur 67: Funksjonen som skriver ut JSON 119
Figur 68: Loggene i ferdig form. Merk at bildet er fra en gammel versjon av programmet. 119

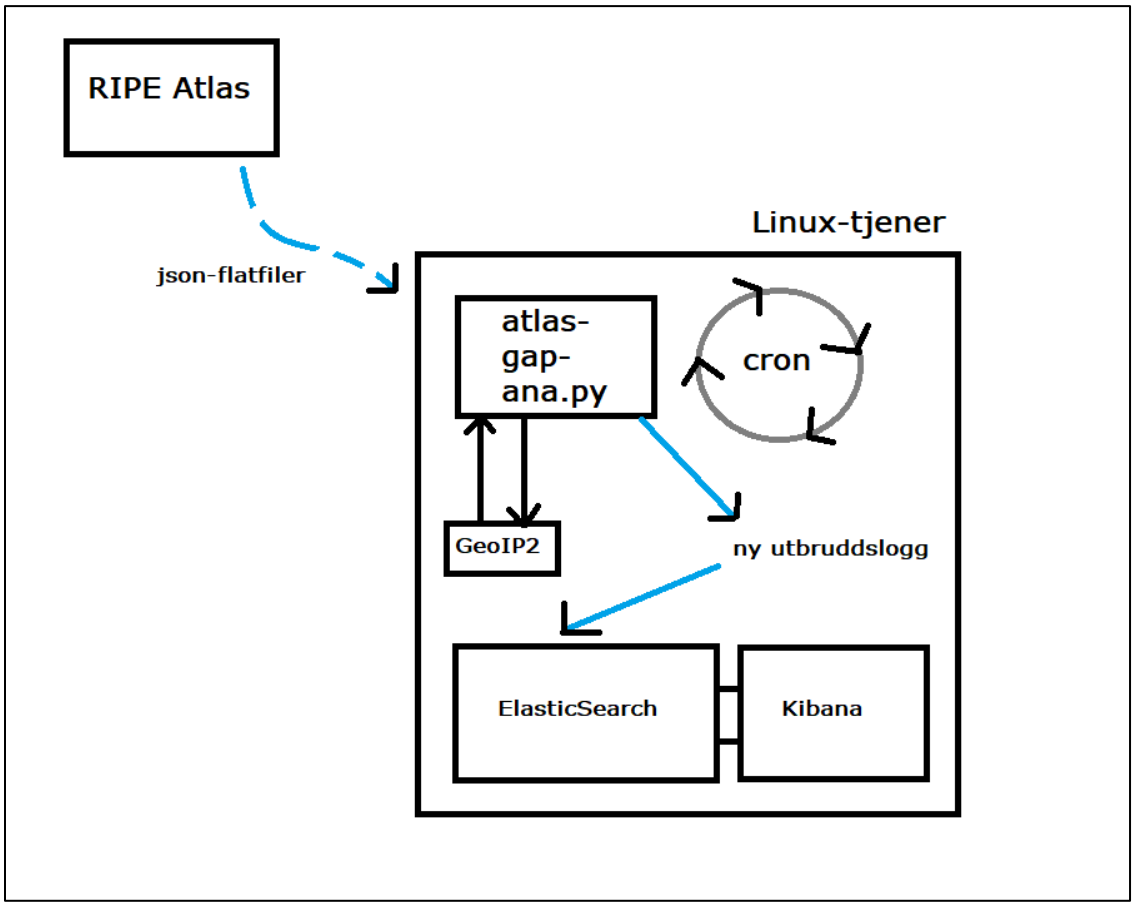
Figur 69: Kobler oss til databasefilen 120
Figur 70: Prøver å droppe en gammel tabell først for sikkerhets skyld 120
Figur 71: Søkehastigheten blir drastisk høyere med indeksering 120
Figur 72: Topologiene lagres fortløpende i databasen 121
Figur 73: Vi lagrer bare nøkkelen til topologien i minnet 121
Figur 74: printRoute() skriver ut ruten en traceroute tar 122
Figur 75: collectRtt() henter ut RTT-informasjonen 122
Figur 76: Topologien til tracerouten som startet utfallet blir lagret 123
Figur 77: Topologiene til hver traceroute som er med i utfallet blir lagret 123
Figur 78: Topologiene til alle de mislykkede traceroutene i utfallet lagres 123
Figur 79: Funksjonen getRtt() 124
Figur 80: Topologiene fra traceroutene som nådde frem hentes ut fra databasen 124
Figur 81: Topologiene blir gjort klar til å skrives ut som JSON 125
Figur 82: Begynnelsen av analyzeTopolog() 125
Figur 83: Finner antall identiske steg på hvert hopp 126
Tabell 19: Eksempel på en ferdig analysert topologi 127
Figur 84: Nå trenger vi de andre topologiene 127
Figur 85: Topologiene til de mislykkede traceroutene hentes ut fra databasen 128
Figur 86: Gjør de mislykkede topologiene klare for JSON 128
Figur 87: De mislykkede rutene gjøres klar til JSON 128
Figur 88: Finner "Dev Tools" i Kibana 129
Figur 89: Oppretter "index" i Elasticsearch 130
Figur 90: Åpner "Stack Management" i Kibana 131
Figur 91: Finner "Index Pattern" 131
Figur 92: Knappen "Create index pattern" 131
Figur 93: Gir "index pattern" et navn 131
Figur 94: Velger standard dato fra loggene 132
Figur 95: Legger loggene inn i index 132
Figur 96: Finner "Discover" i Kibana 132
Figur 97: Finner vårt "index pattern" i "Discover" 133
Figur 98: Velger riktig tidsrom i "Discover" 133
Figur 99: Antallet logger lagt inn 134
Figur 100: Kan utforske alle loggene i "Discover" 134
Figur 101: Velger "duration" i "Discover" 135
Figur 102: Logger listet med "duration" 135
Figur 103: Sortering etter "duration" 135
Figur 104: Loggene sortert etter tid 136
Figur 105: Loggene sortert etter lengde på utfallene 136
Figur 106: Detaljert visning av en logg 136
Figur 107: Den unike ruten knyttet til loggen 137
Figur 108: Søke på rutenavnet 137
Figur 109: Alle loggene knyttet til denne ruten 137
Figur 110: Finne "Dashboard" i Kibana 138
Figur 111: "Create dashboard"-knappen 138

Figur 112: Tomt "Dashboard" 138
Figur 113: Opprette et søylediagram 139
Figur 114: Valg av index pattern til visualisering 139
Figur 115: Y-aksen til søylediagrammet 140
Figur 116: Søylediagram uten x-akse 140
Figur 117: Legger inn x-aksen 141
Figur 118: Søylediagram med både x- og y-akse 141
Figur 119: Legger inn "split series" 142
Figur 120: Splitter på unike ruter 142
Figur 121: Søylediagram med rutenavn 142
Figur 122: Finne "Metrics & axes" 143
Figur 123: Finner "custom extents" 143
Figur 124: Setter minimum til 1500 143
Figur 125: Ferdig søylediagram 144
Figur 126: Lagre visualisering 144
Figur 127: Lage "Coordinate map" 144
Figur 128: Finne summen av varighet 145
Figur 129: Valg av geokoordinater 145
Figur 130: Ferdig koordinatkart 146
Figur 131: "Dashboard" med visualiseringer 146
Figur 132: Styre tidspunktene som visualiseres 146
Figur 133: Rotmappen på Linux-tjeneren 147
Figur 134: Innholdet i "log"-mappen 147
Figur 135: Første del av skriptet som analyserer 24 timer 147
Figur 136: Linux-skript sjekker om flatfilene er klare 148
Figur 137: Linux-skriptet kjører programmet 24 ganger 148
Figur 138: Linux-skript sjekker om utfallsloggene er klare 149
Figur 139: Linux-skript laster loggene inn i Elasticsearch/Kibana 149
Figur 140: Linux-skriptet avslutter 150
Figur 141: Tom crontab 150
Figur 142: Ferdig crontab 151
Figur 143: Stedet med mest utfallstid 152
Figur 144: Mange utfall knyttet til én plass 152
Figur 145: Mistenkelige logger 153
Figur 146: Får alltid problemer samme sted 153
Figur 147: Eksempel på normal ruting 154
Figur 148: Enda et eksempel på normal ruting 154

4. Oversikt: Prosjektet i flere deler

Systemet som skal settes opp i prosjektet består av flere ulike deler som er tydelig avgrenset fra hverandre. For oversiktens skyld har jeg derfor valgt å dele rapporten opp i disse delene.

Først skal vi se på selve programmet som laster ned data fra RIPE Atlas og analyserer dette. Deretter skal vi se på hvordan de nye dataene lastes inn i Elasticsearch og Kibana. I den tredje delen skal vi se på hvordan man i Linux kan få systemet til å gå av seg selv. I den fjerde og siste delen av prosjektet skal vi se noen enkle eksempler på hvordan vi kan analysere utfall.



Figur 1: Enkel skisse av systemet

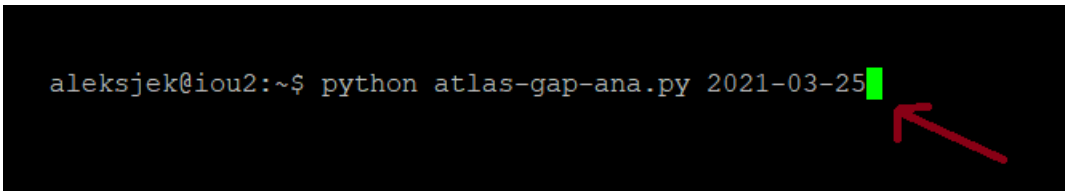
Over kan man se en liten skisse av systemet. Programmet mitt, atlas-gap-ana.py, laster ned flatfiler fra RIPE Atlas. Disse analyseres for brudd som det lagres informasjon om. Det meste av informasjonen regnes ut fra loggene, men vi henter også ut geografisk informasjon og lignende ved å sjekke IP-adressene mot GeoIP2. De nye loggene lastes inn i Elasticsearch ved hjelp av et Linux-skript. I Kibana kan man visualisere og forske på loggene. Ved hjelp av cron kan man i tillegg få Linux til automatisk å lage og laste inn nye logger etter hvert som de legges ut.

5. Første del: Selve programmet

I denne delen vil jeg først vise hvordan man installerer programmet på en fersk datamaskin. Deretter vil jeg forklare hvordan man bruker programmet og hva det gjør. I den lengste delen vil jeg gå gjennom koden i programmet, hvordan den fungerer og hvorfor ulike valg er tatt.

5.1 Forutsetninger og begrensninger

Programmet er tilpasset bruk på en Linux-maskin og hovedsakelig testet på versjon 20.04 av Ubuntu. Det er likevel mulig å få programmet til å kjøre på Windows med små tilpasninger, det meste av problemet ligger i hvordan parameter hentes fra kommandolinjen ved kjøring. Man må også regne med at installasjonen blir litt annerledes enn beskrevet i denne rapporten.



Figur 2: Programmet vil ha problemer med å hente inn parameteret fra Windows

Man bør ha en god Internett-forbindelse og en moderne prosessor. Man trenger 4 GB med plass til mellomlagring, og bør regne med 8 GB minne for hver kjøring av programmet. Hvis man har tenkt å kjøre programmet 24 ganger slik at man parallelt analyserer data fra en hel dag, blir kravene 24 ganger større: I underkant av 100 GB lagringsplass og 200 GB minne. Merk at dette ikke er nøyaktige angivelser, men er omtrentlig basert på erfaring med prosjektet.

Hvor lang tid programmet bruker varierer fra maskin til maskin. På Uninetts tjenere tar det ca. 40-50 minutter å analysere 24 timer samtidig mens én time alene tar ca. 20-30 minutter. Kjøretiden domineres av utpakkingen av de komprimerte bz2-loggene og innlesningen av JSON-dataene i dem. Dette arbeidet blir gjort av høyt optimaliserte moduler importert inn. Så vidt prosjektgruppen kan se er det tvilsomt det vil gi mye gevinst å bruke mer tid på ytelsen.

5.2 Installasjon av programmet

Både versjon 2 og 3 av Python kommer installert med de vanligste distribusjonene av Linux. Vi bruker Python 3. For å installere programmet trenger vi i tillegg et program for pakkebehandling. Jeg har brukt Pip. Dette brukes til å installere moduler, for eksempel den vi skal bruke til å lese JSON-filer inn i Python og til å holde styr på avhengigheter mellom dem. For å installere Pip skriver man bare «sudo apt install python3-pip».

Før vi installerer programmet bruker vi pip til å installere alle avhengighetene til programmet. Dette kan gjøres enten manuelt eller ved hjelp av en såkalt «requirements.txt»-fil som følger med programmet. En requirements.txt-fil lages ved å skrive «pip freeze > requirements.txt». Siden jeg har laget et virtuelt miljø for programmet med programmet pipenv, vil bare modulene vi trenger skrives til requirements.txt og ikke alt jeg har installert på maskinen min.

```
aiohttp==3.7.4.post0
async-timeout==3.0.1
attrs==20.3.0
certifi==2020.12.5
chardet==4.0.0
geoip2==4.1.0
idna==2.10
maxminddb==2.0.3
multidict==5.1.0
pkg-resources==0.0.0
pygeoip==0.3.2
requests==2.25.1
typing-extensions==3.7.4.3
ujson==4.0.2
urllib3==1.26.4
yarl==1.6.3
```

Figur 3: Innholdet i requirements.txt

For å bruke requirements.txt skriver man «pip install -r requirements.txt» i mappen med txt-filen. Da vil alt man trenger bli installert av Pip. Den alternative måten er at man forsøker å kjøre programmet. Man vil deretter får advarsler om hvilke moduler man mangler og kan installere disse etter tur ved hjelp av «sudo apt install ujson» osv.

5.3 Kjøre programmet

Nå er vi klare til å kjøre programmet. For å gjøre dette skriver man simpelthen «python atlas-gap-ana.py» og timen man ønsker å analysere i det samme formatet som 2021-03-15T1400 som er det samme som 15. mars 2021, klokken 14. Eksempel: «python atlas-gap-ana.py 2021-0315T1400». Hvis alt er greit vil programmet skrive ut, «Started analyzing log: 2021-03-15T1400» i konsollen. Ved slutt skrives «Completed analyzing log: 2021-03-15T1400» ut.

Underveis vil programmet mellomlagre opp mot 4 GB i filen 2021-03-15T1400.db som er en SQLlite3-database. Programmet trenger ikke laste ned noen filer fra RIPE Atlas siden bare én logglinje blir lastet inn om gangen. Når det er ferdig skrives den nye loggen ut til mappen log i

samme mappe som programmet ligger i med f. eks. navnet 2021-03-15T1400-finished. Merk at dette kun er satt som standard og at man kan endre stiene lett øverst i koden.

```
#Filepaths used by the program

CityDatabase = "/usr/share/GeoIP/GeoLite2-City.mmdb"
ASDatabaseIPv4 = "/usr/share/GeoIP/GeoIPASNum.dat"
ASDatabaseIPv6 = "/usr/share/GeoIP/GeoIPASNumv6.dat"
LogPath = "log/" #The folder where you want the completed new logs stored
DBPath = "" #Where you want the temporary database stored
```

Figur 4: Standard stier brukt av programmet

5.4 Hva programmet gjør

Enkelt sagt laster programmet ned enorme logger på ca. 20 GB fra RIPE Atlas som simpelthen inneholder resultatene fra millioner av traceroutes utført over en time. Programmet bruker deretter informasjon fra disse til å identifisere brudd på ruter over Internett og beskrive disse bruddene. Beskrivelsene blir deretter skrevet ut til en ny, mer kompakt logg.

5.4.1 Om loggene fra RIPE Atlas

Plassert utover hele verden har RIPE Atlas over 10 000 innretninger kalt prober og ankere som konstant sender ut traceroutes. Til sammen utføres det ca. 7 millioner traceroutes hver time. Resultatene fra disse lagres hos RIPE Atlas og skrives ut til loggfiler på en time hver. Disse loggfilene er ca. 15-20 GB store og blir gjort tilgjengelig fra RIPE Atlas sine tjenere. Merk at loggene på bildet under er komprimert til 1.5 GB og må pakkes ut for å leses.

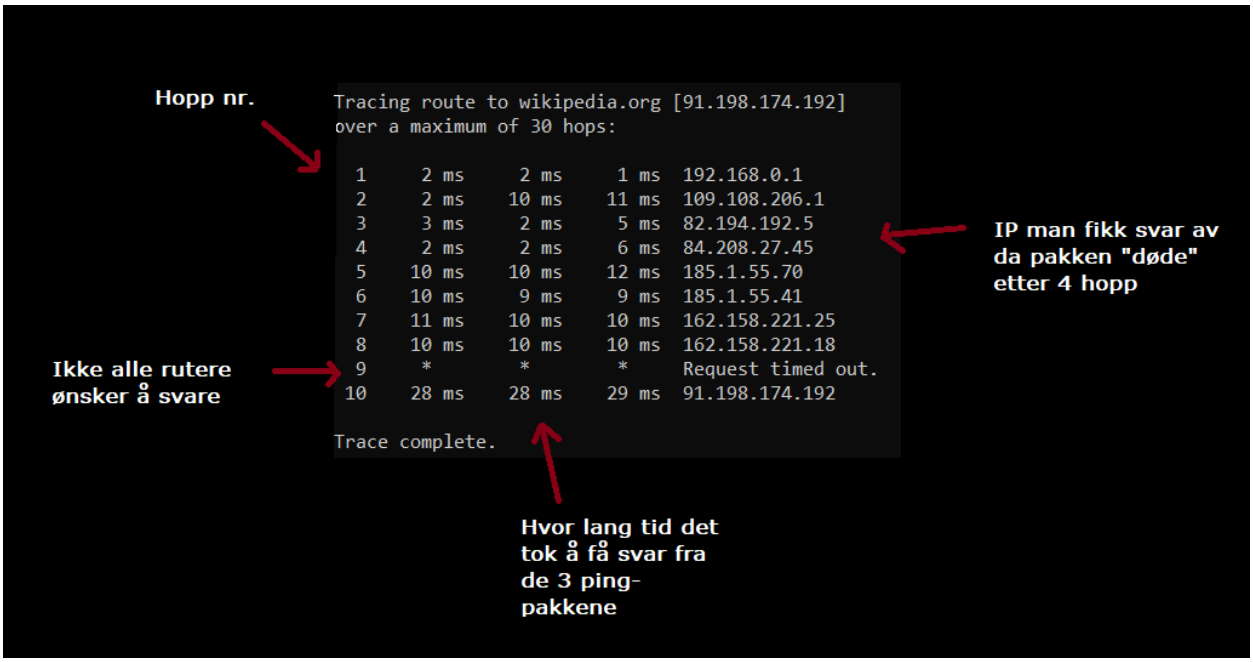
traceroute-2021-03-27T0300.bz2	28-Mar-2021 07:28	1.4G
traceroute-2021-03-27T0400.bz2	28-Mar-2021 07:27	1.4G
traceroute-2021-03-27T0500.bz2	28-Mar-2021 07:30	1.4G
traceroute-2021-03-27T0600.bz2	28-Mar-2021 07:27	1.4G
traceroute-2021-03-27T0700.bz2	28-Mar-2021 07:27	1.4G
traceroute-2021-03-27T0800.bz2	28-Mar-2021 07:28	1.4G
traceroute-2021-03-27T0900.bz2	28-Mar-2021 07:27	1.4G
traceroute-2021-03-27T1000.bz2	28-Mar-2021 07:29	1.4G
traceroute-2021-03-27T1100.bz2	28-Mar-2021 07:28	1.4G
traceroute-2021-03-27T1200.bz2	28-Mar-2021 07:28	1.4G
traceroute-2021-03-27T1300.bz2	28-Mar-2021 07:28	1.4G
traceroute-2021-03-27T1400.bz2	28-Mar-2021 07:41	1.4G

Figur 5: Tjeneren med noen av loggene åpnet i en nettleser

Før vi ser på innholdet i disse loggene tror jeg det er ryddig med en kort gjennomgang av hva en traceroute er. Traceroute er et verktøy som brukes til å finne veien informasjon tar når det sendes over Internett. For å kartlegge dette, sender traceroute ping-pakker. For at man skal få svar fra hvert hopp på veien, starter man med en «Time-To-Live» på ett hopp på ping-pakkene.

På denne måten «dør» pakken etter ett hopp og den første destinasjonen på veien er nødt til å fortelle avsender at den droppet pakken. Slik får traceroute vite hvor lang tid det tar å få svar fra den første destinasjonen og hva IP-en til den første destinasjonen er.

Deretter økes «Time-To-Live» med én slik at neste ruter også svarer og man kan skrive ned informasjon om denne. Slik fortsetter man til man når målet. På denne måten får man kartlagt hele veien fra avsenderen til den endelige mottakeren. Grunnen til at man ser 3 tidsangivelser for hvert hopp, er at traceroute sender 3 ping-pakker på hvert steg.



Figur 6: Forklaring av traceroute

RIPE Atlas sine logger inneholder enkelt sagt resultatet fra ca. 7 millioner slike spørringer. Under kan vi som i tracerouten over se tre svar for hvert hopp. Vi ser også her at ikke alle rutere svarer. Noen av ping-pakkene får bare tilbake resultatet «{"x":"*"}».

"result":[{"hop":1,"result":[{"from":"2a0b:97c0:0:110::1","ttl":64,"rtt":0.651,"size":96},{"fro
m":"2a0b:97c0:0:110::1","ttl":64,"rtt":0.633,"size":96},{"from":"2a0b:97c0:0:110::1","ttl":64,
"rtt":0.509,"size":96}]},{"hop":2,"result":[{"from":"2a0b:97c0:0:1::6","ttl":63,"rtt":4.123,"size
":96},{"from":"2a0b:97c0:0:1::6","ttl":63,"rtt":1.94,"size":96},{"from":"2a0b:97c0:0:1::6","ttl"
:63,"rtt":1.674,"size":96}]},{"hop":3,"result":[{"from":"::ffff:172.16.176.2","ttl":62,"rtt":12.84
3,"size":96},{"from":"::ffff:172.16.176.2","ttl":62,"rtt":10.515,"size":96},{"from":"::ffff:172.16.
176.2","ttl":62,"rtt":7.7,"size":96}]},{"hop":4,"result":[{"from":"2a03:5f80:4::224:229","ttl":6
1,"rtt":97.062,"size":96},{"from":"2a03:5f80:4::224:229","ttl":61,"rtt":97.092,"size":96},{"fro
m":"2a03:5f80:4::224:229","ttl":61,"rtt":97.642,"size":96}]},{"hop":5,"result":[{"from":"2001:
470:0:3aa::1","ttl":60,"rtt":115.247,"size":96},{"from":"2001:470:0:3aa::1","ttl":60,"rtt":115.
246,"size":96},{"from":"2001:470:0:3aa::1","ttl":60,"rtt":115.074,"size":96}]},{"hop":6,"result
":[{"x":"*"},{"x":"*"},{"x":"*"}]},{"hop":7,"result":[{"x":"*"},{"x":"*"},{"from":"2a01:5a8::1:1","
ttl":59,"rtt":142.22,"size":96}]},{"hop":8,"result":[{"from":"2a01:5a8::1:4","ttl":58,"rtt":142.6
4,"size":48},{"from":"2a01:5a8::1:4","ttl":58,"rtt":142.724,"size":48},{"from":"2a01:5a8::1:4",
"ttl":58,"rtt":142.639,"size":48}]}]

Tabell 3: Eksempel på en traceroute lagret av RIPE Atlas

I tillegg blir hver av disse resultatene pakket sammen med informasjon om selve tracerouten. For eksempel forteller «fw» oss hvilken versjon av probene som stod bak. Attributtene «timestamp» og «endtime» forteller oss henholdsvis når tracerouten ble startet og når den sluttet i UNIX-tid. UNIX-tid er det samme som antall sekunder siden 1. januar 1970 UTC. Hos RIPE kan man finne en forklaring av alle attributtene. (RIPE Atlas, 2021)

{"fw":5020,"mver":"2.2.1","lts":44,"endtime":1607731572,"dst_name":"2a01:5a8::1:4","dst_
addr":"2a01:5a8::1:4","src_addr":"2a0b:97c0:0:110::2","proto":"ICMP","af":6,"size":48,"paris
_id":8,"msm_id":28236165,"prb_id":6624,"timestamp":1607731550,"msm_name":"Tracerou
te","from":"2a0b:97c0:0:110::2","type":"traceroute","group_id":28236165}

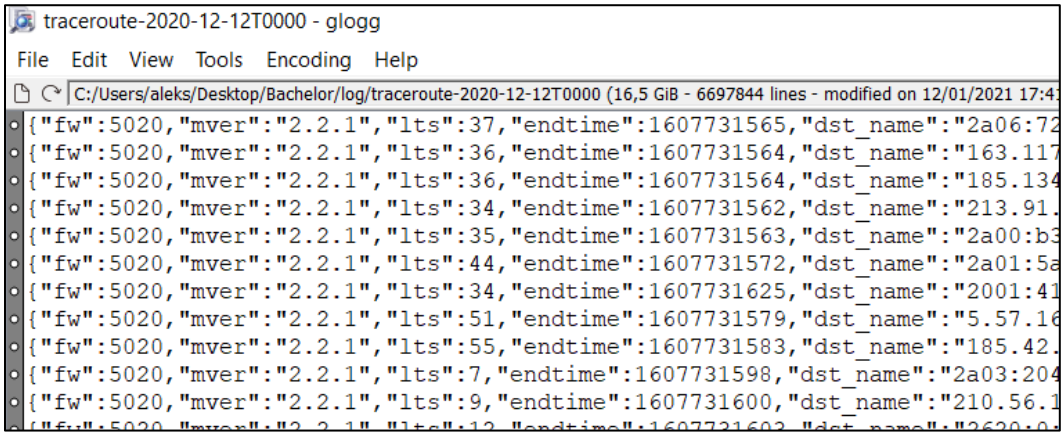
Tabell 4: Eksempel på RIPE Atlas-innpakning av traceroutes

Sammen ser en enkelt linje i en av disse loggene med andre ord slik ut:

{"fw":5020,"mver":"2.2.1","lts":44,"endtime":1607731572,"dst_name":"2a01:5a8::1:4","dst_addr":"2a01:5a8::1:4","src_addr":"2a0b:97c0:0:110::2","proto":"ICMP","af":6,"size":48,"paris_id":8,"result":[{"hop":1,"result":[{"from":"2a0b:97c0:0:110::1","ttl":64,"rtt":0.651,"size":96},{"from":"2a0b:97c0:0:110::1","ttl":64,"rtt":0.633,"size":96},{"from":"2a0b:97c0:0:110::1","ttl":64,"rtt":0.509,"size":96}]},{"hop":2,"result":[{"from":"2a0b:97c0:0:1::6","ttl":63,"rtt":4.123,"size":96},{"from":"2a0b:97c0:0:1::6","ttl":63,"rtt":1.94,"size":96},{"from":"2a0b:97c0:0:1::6","ttl":63,"rtt":1.674,"size":96}]},{"hop":3,"result":[{"from":"::ffff:172.16.176.2","ttl":62,"rtt":12.843,"size":96},{"from":"::ffff:172.16.176.2","ttl":62,"rtt":10.515,"size":96},{"from":"::ffff:172.16.176.2","ttl":62,"rtt":7.7,"size":96}]},{"hop":4,"result":[{"from":"2a03:5f80:4::224:229","ttl":61,"rtt":97.062,"size":96},{"from":"2a03:5f80:4::224:229","ttl":61,"rtt":97.092,"size":96},{"from":"2a03:5f80:4::224:229","ttl":61,"rtt":97.642,"size":96}]},{"hop":5,"result":[{"from":"2001:470:0:3aa::1","ttl":60,"rtt":115.247,"size":96},{"from":"2001:470:0:3aa::1","ttl":60,"rtt":115.246,"size":96},{"from":"2001:470:0:3aa::1","ttl":60,"rtt":115.074,"size":96}]},{"hop":6,"result":[{"x":"*"},{"x":"*"},{"x":"*"}]},{"hop":7,"result":[{"x":"*"},{"x":"*"},{"from":"2a01:5a8::1:1","ttl":59,"rtt":142.22,"size":96}]},{"hop":8,"result":[{"from":"2a01:5a8::1:4","ttl":58,"rtt":142.64,"size":48},{"from":"2a01:5a8::1:4","ttl":58,"rtt":142.724,"size":48},{"from":"2a01:5a8::1:4","ttl":58,"rtt":142.639,"size":48}]}],"msm_id":28236165,"prb_id":6624,"timestamp":1607731550,"msm_name":"Traceroute","from":"2a0b:97c0:0:110::2","type":"traceroute","group_id":28236165}

Tabell 5: En fullstendig RIPE Atlas-logg med både traceroute og innpakning

Disse loggfilene er såpass store at de for eksempel vil krasje Notepad. For å kunne lese dem ubehandlet har jeg derfor brukt et gratis program som heter glogg som tåler såpass store filer. Dette programmet kan lastes ned fra utviklernes hjemmesider. (glogg, 2021)

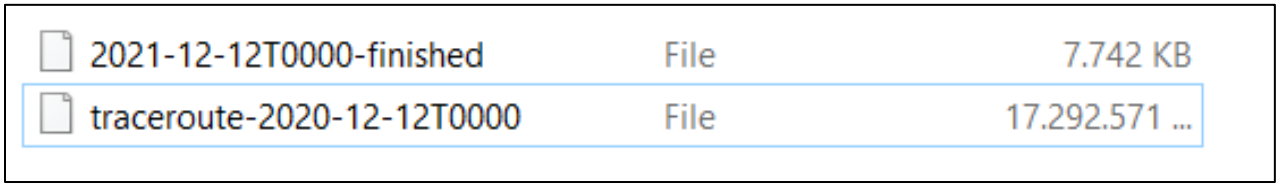


Figur 7: Bruk av programmet glogg

5.4.2 Loggene atlas-gap-ana.py lager

Det programmet enkelt sagt gjør er at det kombinerer informasjon fra store mengder traceroutes for å prøve å fortelle oss om hva som skjer på forskjellige ruter. Et eksempel er at man har 5 traceroutes etter hverandre hvor pakken når det endelige målet sitt. Deretter har man 2 traceroutes hvor pakken ikke når det endelige målet sitt. Til slutt følger 7 traceroutes hvor pakken når målet igjen. Da vil programmet anta at det har skjedd et brudd i periodene traceroutene ikke nådde og lage en beskrivelse av dette bruddet.

Den vil også hente ut informasjon om disse bruddene, for eksempel når de startet og sluttet samt hvilke feil som eventuelt ble rapportert i perioden. Fra 20 GB og ca. 7 millioner linjer med logg får vi 20 MB og rundt 15 000 linjer med informasjon som er mye mer interessant for oss. Merk at programmet kun rapporterer om ruter med minst én negativ traceroute og én positiv. Dette er for å få en mer håndterlig datamengde på slutten og fordi man helst bør ha litt mer å gå på enn dette om en rute dersom informasjonen skal være interessant for oss.



Figur 8: Den originale loggen og loggen programmet lager

5.4.2.1 Om loggoppsummeringene

Den endelige filen som programmet skriver ut består av tre typer logger. Den første loggtypen skrives bare ut én gang og inneholder informasjon beregnet fra hele kjøringen.

{"total_traceroutes":7634825,"event_type":"log_summary","start_time":1621119600,"end_time":1621123199,"total_succeeded":5876397,"total_failed":1758428,"failed_at_probe":77353,"total_probes":11544,"version":"1.63 atlas-gap-ana aleksandra jekic"}

Tabell 6: Eksempel på en "Log Summary"

De fleste av disse attributtene er selvforklarende, men under følger en enkel forklaring av hver.

Attributt	**Forklaring**
total_traceroutes	Oppgir den totale mengden traceroutes i loggen. Her var det altså 7,6 millioner logglinjer. De fleste loggene har rundt 7 millioner traceroutes.

event_type	Forteller oss hvilken type logg det er. «Log Summary» betyr at det er en oppsummering av hele timesloggen. De andre typene er «Summary» og «Outage» som jeg skal forklare etterpå.
start_time	Forteller oss hva det tidligste tidspunktet en traceroute i loggen ble startet.
end_time	Forteller oss hva det seneste tidspunktet en traceroute i loggen ble startet.
total_succeeded	Antall traceroutes som nådde frem til det endelige målet sitt til slutt. Vi ser her at det er ca. 5,8 millioner som er en ganske stor andel.
total_failed	Antall traceroutes i loggen som aldri nådde frem til destinasjonen. Det er disse antakelsene om brudd baserer seg på.
failed_at_probe	Antall traceroutes som sviktet før de klarte gjøre det første hoppet.
total_probes	Antall unike prober de forskjellige traceroutene kom fra.
version	Hvilken versjon av programmet man har brukt. Her kan man også se at Aleksandra har vært litt jålete og lagt navnet sitt inni.

Tabell 7: Forklaring av attributtene i "Log Summary"

5.4.2.2 Om ruteoppsummeringene

Den andre typen logg, «Summary», inneholder informasjon som gjelder for hver unike rute hvor det har blitt registrert minst én vellykket og én mislykket traceroute.

{"pair_id":"29278-to-2a03:300:3::1","event_type":"summary","dst_addr":"2a03:300:3::1","src_addr":"2a00:6020:13be:3600:ee08:6bff:fe73:2f7e","prb_id":"29278","start_time":1621123171,"last_response":1621123179,"traceroutes_recorded":8,"failed_traceroutes":"12%","outage_count":1,"outage_at_log_start":false,"outage_at_log_end":false,"incomplete_outages":0,"total_dupes":0,"total_late":0,"total_outage_errors":{"No reply":2,"Destination unreachable":1},"total_outage_duration_in_seconds":8,"max_hop":13,"min_hop":11,"topology":[{"hop":0,"ip":"2a00:6020:13be:3600:3681:c4ff:fe3b:7405","rtt_count":18,"portion":"86%","maxrtt":0.993,"minrtt":0.628,"rtt_std_dv":0.08},{"hop":2,"ip":"2a00:6020:0:a::1","rtt_count":5,"portion":"24%","maxrtt":6.609,"minrtt":6.31,"rtt_std_dv":0.11},{"hop":3,"ip":"2001:7f8:1::a503:1027:1","rtt_count":4,"portion":"19%","maxrtt":23.946,"minrtt":23.509,"rtt_std_dv":0.16},{"hop":3,"ip":"2001:2000:3080:197d::1","rtt_count":6,"portion":"29%","maxrtt":6.199,"minrtt":5.895,"rtt_std_dv":0.12},{"hop":4,"ip":"2a01:7e8:1:200::9aa","rtt_count":15,"portion":"71%","maxrtt":24.412,"minrtt":22.27,"rtt_std_dv":0.91},{"hop":4,"ip":"2001:2034:1:6b::1","rtt_count":6,"portion":"29%","maxrtt":10.031,"minrtt":9.669,"rtt_std_dv":0.13},{"hop":5,"ip":"2a01:7e8:1:200::9a9","rtt_count":6,"portion":"29%","maxrtt":24.562,"minrtt":24.229,"rtt_std_dv":0.11},{"hop":5,"ip":"::ffff:93.176.90.112","rtt_count":9,"portion":"43%","maxrtt":25.734,"minrtt":25.347,"rtt_std_dv":0.14},{"hop":5,"ip":"2001:2034:0:90::1","rtt_count":6,"portion":"29%","maxrtt":10.891,"minrtt":9.916,"rtt_std_dv":0.4},{"hop

":6,"ip":"2a01:7e8:1:200::9a6","rtt_count":4,"portion":"19%","maxrtt":22.315,"minrtt":21.927,"rtt_std_dv":0.16},{"hop":6,"ip":"::ffff:194.182.97.108","rtt_count":9,"portion":"43%","maxrtt":24.425,"minrtt":23.674,"rtt_std_dv":0.21},{"hop":6,"ip":"2001:2000:3080:116::2","rtt_count":6,"portion":"29%","maxrtt":9.35,"minrtt":8.029,"rtt_std_dv":0.55},{"hop":7,"ip":"2001:1900:5:2:2::3bdd","rtt_count":6,"portion":"29%","maxrtt":19.501,"minrtt":19.217,"rtt_std_dv":0.11},{"hop":7,"ip":"2a01:7e8:1:200::8f1","rtt_count":9,"portion":"43%","maxrtt":23.829,"minrtt":23.414,"rtt_std_dv":0.14},{"hop":7,"ip":"2001:6c8:40::1f3","rtt_count":6,"portion":"29%","maxrtt":21.276,"minrtt":20.102,"rtt_std_dv":0.4},{"hop":8,"ip":"2001:6c8:81:100::15a","rtt_count":6,"portion":"29%","maxrtt":19.629,"minrtt":19.29,"rtt_std_dv":0.11},{"hop":8,"ip":"2a01:7e8:1:200::372","rtt_count":9,"portion":"43%","maxrtt":26.697,"minrtt":24.154,"rtt_std_dv":0.77},{"hop":8,"ip":"2001:6c8:41:100:0:8e:0:2","rtt_count":6,"portion":"29%","maxrtt":20.3,"minrtt":20.138,"rtt_std_dv":0.06},{"hop":9,"ip":"2001:6c8:40::1f3","rtt_count":6,"portion":"29%","maxrtt":23.92,"minrtt":23.419,"rtt_std_dv":0.18},{"hop":9,"ip":"2a01:7e8:1:800::2f9","rtt_count":1,"portion":"5%","maxrtt":24.484,"minrtt":24.484,"rtt_std_dv":0.0},{"hop":9,"ip":"2a03:300:2::3","rtt_count":6,"portion":"29%","maxrtt":21.848,"minrtt":21.58,"rtt_std_dv":0.1},{"hop":10,"ip":"2001:6c8:41:100:0:8e:0:2","rtt_count":6,"portion":"29%","maxrtt":23.664,"minrtt":23.481,"rtt_std_dv":0.07},{"hop":10,"ip":"2a03:300:3::1","rtt_count":5,"portion":"24%","maxrtt":21.187,"minrtt":20.722,"rtt_std_dv":0.2},{"hop":11,"ip":"2a03:300:2::3","rtt_count":6,"portion":"40%","maxrtt":25.191,"minrtt":24.959,"rtt_std_dv":0.09},{"hop":12,"ip":"2a03:300:3::1","rtt_count":10,"portion":"67%","maxrtt":24.497,"minrtt":18.843,"rtt_std_dv":2.16}]}

Tabell 8: Eksempel på en "Summary"

Denne er litt mer komplisert enn det «Log Summary» var. Under er hvert attributt forklart igjen.

Attributt	**Forklaring**
pair_id	Inneholder id-en til den beskrevne ruten som blir laget av probe ID til avsender og IP til mottaker. I eksempelet over er avsenderproben «29278» og mottakeren «2a00:6020:13be:3600:ee08:6bff:fe73:2f7e». IP-en til avsender og mottaker er bundet sammen med teksten «-to-» for å lage den unike id-en.
event_type	Hvilken type logg det er. «Summary» betyr at det loggen sier gjelder for hele denne unike ruten for denne timen.
dst_addr	IP-en hvor denne ruten slutter.
src_addr	IP-en hvor denne ruten starter.
prb_id	ID-en til proben som utførte tracerouten.
start_time	Når den første tracerouten mellom disse adressene ble sendt.
last_response	Når den siste tracerouten mellom disse adressene ble sendt.
traceroutes_recorded	Antall traceroutes knyttet til denne ruten.

failed_traceroutes	Antall prosent av traceroutene som var mislykket og aldri nådde målet sitt.
outage_count	Antallet brudd registrert på denne ruten i løpet av timen.
outage_at_log_start	Hvorvidt den første tracerouten på denne ruten var mislykket. Brukes for å knyttes sammen med logger fra andre timer.
outage_at_log_end	Hvorvidt den siste tracerouten på denne ruten var mislykket. Brukes for å knyttes sammen med logger fra andre timer.
incomplete_outages	Enkelte av traceroutene er tilsynelatende ufullstendige og kan tolkes mange veier. Denne attributten forteller antallet tilsynelatende brudd som består av slike ukomplette traceroutes.
total_dupes	Antall duplikatsvar på denne ruten.
total_late	Antall sene svar på denne ruten.
total_outage_errors	Samler alle feilene fra alle bruddene på denne ruten. Angir alle feiltypene og antallet av hver.
total_outage_duration_in_seconds	Antall sekunder alle bruddene varte til sammen.
max_hop	Den lengste vellykkede ruten fra avsender til destinasjon.
min_hop	Den korteste vellykkede ruten fra avsender til destinasjon.
topology	Her finner vi et kart over ruten når den fungerer. Under forklarer jeg dette litt nærmere siden det skjer litt mye her.

Tabell 9: Forklaring av attributtene i "Summary"

5.4.2.3 Om topologiene i ruteoppsummeringene

«Topology» er kanskje litt enklere å forstå når det blir formatert pent som i utdraget under.


```
[…],
{
  "hop": 6,
  "rtt_std_dv": 0.16,
  "ip": "2a01:7e8:1:200::9a6",
  "maxrtt": 22.315,
  "minrtt": 21.927,
  "portion": "19%",
  "rtt_count": 4
},
{
```

```
  "rtt_std_dv": 0.21,
  "hop": 6,
  "ip": "::ffff:194.182.97.108",
  "maxrtt": 24.425,
  "portion": "43%",
  "minrtt": 23.674,
  "rtt_count": 9
},
[…]
```


Tabell 10: Utdrag fra en "Summary"-topologi

Her ser man to mulige eksempler på det sjette hoppet i denne ruten. Det kan være til adressen «2a01:7e8:1:200::9a6» eller adressen «::ffff:194.182.97.108». Attributtet portion forteller oss hvor ofte dette hoppet gjøres i prosent basert på de traceroutene vi har for denne timen. Noen ganger vil ikke prosentene summere til 100. Grunnen til det er at vi dropper å registrere hopp med «No Reply». Disse inneholder likevel ingen nyttig informasjon og vi kan regne oss til at de eksisterer ved at de mangler. Attributtet rtt_count forteller oss antall treff vi har til hvert hopp, maxrtt og minrtt er den høyeste og laveste rtt vi har registrert til hoppet og rtt_std_dv er standardavviket som forteller oss hvor mye rtt har variert. RTT er for øvrig «round-trip time» og er antall sekunder det tok å sende ping-pakken til hoppet og få et svar tilbake.

5.4.2.4 Om utfallsbeskrivelsene

Mesteparten av den nye loggen beskriver hvert enkelt registrerte utfall.


```
{"pair_id":"6307-to-
2001:500:200::b","event_type":"outage","order":2,"dst_addr":"2001:500:200::b","src_addr":
"2001:67c:2e8:3::c100:a4","prb_id":"6307","start_time":1621123193,"last_response":16211
23193,"outage_at_log_start":false,"outage_at_log_end":true,"failure_count":1,"polling_freq
uency":220,"duration":0,"last_reply_from":"2001:500:209:5::2","AS_number":"AS394353","
AS_owner":"B.Root-Server-OPS","location":"United States","coordinates":"37.751,-
97.822","outage_errors":{"No reply":3},"error_type":"Stopped
route","max_hop":12,"min_hop":12,"topology":[{"hop":1,"ip":"2001:67c:2e8:101::1:1","rtt_c
ount":3,"portion":"100%","maxrtt":0.907,"minrtt":0.648,"rtt_std_dv":0.12},{"hop":2,"ip":"20
01:7f8:1::a500:2914:1","rtt_count":3,"portion":"100%","maxrtt":7.841,"minrtt":2.589,"rtt_st
d_dv":2.16},{"hop":3,"ip":"2001:728:0:2000::1e9","rtt_count":3,"portion":"100%","maxrtt":9
.853,"minrtt":4.215,"rtt_std_dv":2.57},{"hop":4,"ip":"2001:728:0:2000::106","rtt_count":3,"p
ortion":"100%","maxrtt":4.032,"minrtt":2.902,"rtt_std_dv":0.52},{"hop":5,"ip":"2001:500:20
9:5::2","rtt_count":3,"portion":"100%","maxrtt":2.32,"minrtt":1.664,"rtt_std_dv":0.29}]}
```


Tabell 11: Eksempel på en "Detail"-logg

Under har jeg igjen forklart hver av attributtene man finner i disse loggene.

Attributt	Forklaring
pair_id	Inneholder id-en til den beskrevne ruten som blir laget av probe ID til avsender og IP til mottaker. I eksempelet over er avsenderproben «29278» og mottakeren «2a00:6020:13be:3600:ee08:6bff:fe73:2f7e». IP-en til avsender og mottaker er bundet sammen med teksten «-to-» for å lage den unike id-en.
event_type	Hvilken type logg det er. «Outage» betyr at det er en utfallsbeskrivelse.
order	Unik identifikasjon for hvert unike utbrudd innenfor en rute den timen. Bruddene er nummerert etter rekkefølgen.
dst_addr	IP-en hvor denne ruten slutter.
src_addr	IP-en hvor denne ruten starter.
prb_id	ID-en til proben som utførte tracerouten.
start_time	Når den første tracerouten mellom disse adressene ble sendt.
last_response	Når den siste tracerouten mellom disse adressene ble sendt.
outage_at_log_start	Hvorvidt den første tracerouten på denne ruten var mislykket. Brukes for å knyttes sammen med logger fra andre timer.
outage_at_log_end	Hvorvidt den siste tracerouten på denne ruten var mislykket. Brukes for å knyttes sammen med logger fra andre timer.
failure_count	Antallet mislykkede traceroutes knyttet til utfallet.
polling_frequency	Gjennomsnittlig tid mellom traceroutes knyttet til bruddet i sekunder.
duration	Sekunder fra ruten tilsynelatende var nede til den var oppe igjen.
last_reply_from	Siste IP som gav svar, hentet fra første mislykkede traceroute i utfallet.
AS_number	AS-nummeret til den siste IP-en som gav svar, hentet fra GeoIP2. Autonome soner (AS) er sett med IP-adresser forvaltet av én nettverksforvalter som har ansvaret for å fortelle andre autonome soner hvordan de skal nå disse IP-adressene. (Rekhter, 1995)
AS_owner	Navnet på eieren av AS-et til den siste IP-en som gav svar.
location	Byen/landet IP-en som sist gav svar er knyttet til.
coordinates	Kartkoordinatene IP-en som sist gav svar er knyttet til.
outage_errors	Alle feilmeldinger samlet inn fra de mislykkede traceroutene.
error_type	Sannsynlig hovedårsak for utfallet.
max_hop	Den lengste mislykkede ruten registrert.
min_hop	Den korteste mislykkede ruten registrert.
topology	Samme struktur som i ruteoppsummeringene, bare at man her tar for seg mislykkede ruter som aldri kommer frem til målet.

Tabell 12: Forklaring av attributtene i "Detail"

5.5. Hvordan programmet virker

5.5.1 Hvordan RIPE Atlas-loggene leses inn i programmet

Når funksjonen main() kjøres ved start av programmet, hentes forespørselen inn. Forespørselen er det man skriver etter «python atlas-gap-ana.py». For eksempel «2021-03-03T1400». Dette styrer hvilken time programmet skal analysere data for fra RIPE Atlas sine tjenere.

```
def main():

    #Checks the nature of the request, sends
    #error message if it doesn't make sense

    try:
        request = str(sys.argv[1])
    except:
        print("Please execute program using pattern 'python atlas-gap-ana.py" +
        " 2021-03-03T1400'.")
```

Figur 9: Forenklet main()

Dersom parameteret ikke finnes, vil ikke Python klare å finne noe i sys.argv[1] og man vil få en beskjed om hvordan programmet startes riktig. Dersom parameteret finnes, blir det gjort til en tekststreng med str() som vi kan bruke videre. Merk at sys.argv[0] er navnet på programmet.

Deretter sjekker vi om parameteret faktisk passer med forventningene våre.

```
if len(request) == 15 and len(sys.argv) == 2:

    analyzeLog(request)

else:
    print("Please execute program using pattern 'python atlas-gap-ana.py" +
    " 2021-03-03T1400'.")
```

Figur 10: Mer forenklet main()

Hvis det gjør det, sender vi det videre til analyzeLog()-funksjonen som vi kommer tilbake til.

5.5.1.1 Den tidligere versjonen av main() som brukte tråder

Opprinnelig var main() mye mer komplisert. Da var tanken at programmet både kunne analysere 1 time om gangen og 24 timer om gangen avhengig av forespørselen.

```
#If only one hour is requested, that is easy

if len(request) == 15 and len(sys.argv) == 2:

    link = getLogName(request)
    downloadLog(link)
    analyzeLog(request)
```

Figur 11: Analyse av én logg med tidligere versjon av main()

Her ser vi at dersom forespørselen var 15 tegn lang og dermed inneholdt klokkeslettet også, for eksempel «2021-03-03T1400», så var det ganske enkelt å analysere bare den. Først fant vi ut hvilken link denne loggen befant seg på ved hjelp av getLogName().

```
def getLogName(logsNotAnalyzed):

    day = logsNotAnalyzed[0:10]

    link = "https://data-store.ripe.net/datasets/atlas-daily-dumps/" + day \
        + "/traceroute-" + logsNotAnalyzed + ".bz2"

    return link
```

Figur 12: Kodesnutt fra getLogName()

Det første getLogName() gjør er å klippe ut de ti første tegnene for å få datoen «2021-03-03». Deretter lager man linken av https://data-store.ripe.net/datasets/atlas-daily-dumps/ + datoen (for eksempel 2021-03-03) + /traceroute- + datoen og klokkeslettet («2021-03-03T1400)» og legger så til «.bz2» bak. Da får man for eksempel linken «https://data-store.ripe.net/datasets/atlas-daily-dumps/2021-03-03/traceroute-2021-03-03T1400.bz2».

Tilbake i main() ble linken sendt videre til downloadLog().

```
def downloadLog(link):

    print("Downloading ...", link)

    #Downloads bz2 from RIPE Atlas website
    urllib.request.urlretrieve(link, 'log/compressedlog.bz2')

    print("Unpacking log ...")

    #Unpacks log to a file in log folder with the date and time of the log
    with bz2.BZ2File("log/compressedlog.bz2") as fr, open("log/" + str(link[-19:-4]),"wb") as fw:
        shutil.copyfileobj(fr,fw)
```

Figur 13: Kodesnutt fra downloadLog()

Den komprimerte loggen på ca. 2 GB ble lastet ned med urllib.request til «log/compressedlog.bz2». Deretter ble hele filen pakket opp med bz2-modulen og lagret med navnet på forespørselen. For eksempel «log/2021-03-03T1400».

Når hele filen på 15-20 GB var pakket opp, ble navnet på forespørselen sendt videre til analyzeLog() som ved hjelp av navnet på forespørselen visste hvor filen ville ligge.

Dersom man på den andre siden ville laste ned 24 timer på en gang, var ting mer komplisert.

```
#If an entire day is requested, we download that
#After we make a seperate thread to continue downloading
#Meanwhile we start analyzing the logs as they come

elif len(request) == 10 and len(sys.argv) == 2:

    oldrequest = request
    request = request[0:10] + "T0000"
    link = getLogName(request)
    downloadLog(link)

    #Used to log the progress of the download thread
    finished = ["Unfinished"] * 23
    #Start the thread which downloads the remaining threads
    thread = Thread(target=downloadLogs, args=(finished, oldrequest))
    thread.start()

    analyzeLog(request)
    print("Completed log")
    print(str(count) + " logs left")

    count = 23

    while count > 0:

        #Waits if the the download thread hasn't finished with the next download yet
        while finished[23-count] == "Unfinished":
            sleep(10)

        analyzeLog(finished[23-count])
        count = count - 1
        print("Completed log")
        print(str(count) + " logs left")
```

Figur 14: Tidligere versjon av main() med flere tråder

Da lastet man ned og pakket som normalt ut den første loggen. Deretter ble det laget en dictionary med 23 plasser hvor det stod «Unfinished» samtidig som man startet en ny tråd med

thread.start(). Denne tråden brukte funksjonen downloadLogs(), som ikke må forveksles med downloadLog(), til å laste ned, pakke ut alle loggene og merke dette i dictionarien.

```
#Downloads every log in an hour and writes its
#name to a dictionary when it's done

#The idea is to run this function on a seperate
#thread so that the program is downloading later
#logs while analyzing others.

#Python runs on only one thread, but
#for IO operations it can still pay with threading.

def downloadLogs(finished, request):

    count = 23

    while count > 0:

        time = 24 - count

        if time < 10:
            time = str("0") + str(time)

        request = request[0:10] + "T" + str(time) + "00"

        link = getLogName(request)
        downloadLog(link)
        finished[23-count] = request

        count = count - 1
```

Figur 15: Kodesnutt fra downloadLogs(), ikke det samme som downloadLog()

I mellomtiden startet jeg analyzeLog() for den første loggen. Når analyzeLog() da ble ferdig for den første loggen, gikk den inn i en løkke hvor den jobbet seg ferdig med resten av loggene etter hvert som de ble ferdige. Når en logg ikke var ferdig, ventet den i 10 sekund før den prøvde igjen. Slik holdt programmet på til det ble ferdig med 24 timer med loggfiler.

Dette var en veldig dårlig løsning av to grunner. For det første er det veldig unødvendig å laste ned og dekomprimere loggene på hele 15-20 GB hver før man begynner å analysere. I stedet kan man laste ned én linje fra loggen om gangen og bli ferdig med den før man går videre. Begge alternativene tar ca. like lang tid, men det ene alternativet sparer mye lagringsplass.

Den andre grunnen til at det var en dårlig løsning, er at Python bare kan kjøre én faktisk tråd om gangen. (Tayal, 2019) Dette kalles «Global Interpreter Lock» (GIL). Det vil si at tråder i Python er tråder styrt av programmet som operativsystemet bare ser som én tråd. Dette gir en viss effekt når man jobber med input/output slik man gjør når man laster ned og dekomprimerer. Ofte var programmet halveis ferdig med den neste loggen når analysen var

gjort. Men det gir lite effekt sammenlignet med å starte 24 instanser av programmet samtidig med hver sin tråd som operativsystemet kan se. Dette kommer jeg tilbake til i kapittel 7.

5.5.1.2 Den nye versjonen av main()

Den nyeste versjonen av programmet kan altså bare ta imot forespørsler på en bestemt time. Programmet må kjøres med mønsteret «python atlas-gap-ana.py 2021-03-03T1400». Programmet tar deretter som sagt bare ut forespørselen og sender den til analyzeLog().

```
if len(request) == 15 and len(sys.argv) == 2:

    analyzeLog(request)

else:
    print("Please execute program using pattern 'python atlas-gap-ana.py" +
    " 2021-03-03T1400'.")
```

Figur 16: Den forenklede versjonen av main() igjen

AnalyzeLog() henter deretter ut navnet med getLogName(). Så åpnes linken med den komprimerte filen som et dokument istedenfor at det lastes ned.

```
link = getLogName(request) #Creates the url where the log data for the requested day exists
req = urllib.request.Request(link)
bz2log = urllib.request.urlopen(req) #Opens the url as a file
```

Figur 17: Hvordan analyzeLog() åpner linken som en fil

Deretter dekomprimerer bz2 hver linje i dokumentet etter hvert som de lastes inn. Biblioteket ujson, som jeg har kalt json, oversetter JSON-linjene om til noe Python kan jobbe med.

```
#Loads each line of the hourly log from RIPE Atlas servers
for line in bz2.BZ2File(bz2log, 'rb'):
    data = json.loads(line)
```

Figur 18: Linjene lastes inn og dekomprimeres, én om gangen

Denne delen av programmet er faktisk det som bruker det meste av kjøretiden til programmet. Nedlastning, dekomprimering og innlasting av JSON. Siden dette blir gjort med svært optimaliserte biblioteker, er det lite trolig det finnes særlig forbedringspotensial her. Her ser vi for eksempel at innlasting og dekomprimering av hver linje tar over 600 sekunder.

```
ill)
       1    0.725    0.725 1515.636 1515.636 atlas-gap-ana.py:961(main)
       1    0.000    0.000    0.000    0.000 bz2.py:112(close)
 7011382    3.040    0.000    3.040    0.000 bz2.py:137(closed)
 7011380    4.750    0.000   12.691    0.000 bz2.py:151(readable)
 7011380   22.497    0.000  637.186    0.000 bz2.py:206(readline)
       1    0.000    0.000    0.000    0.000 bz2.py:41(__init__)
       1    0.000    0.000    0.168    0.168 client.py:1001(_send_output)
       1    0.000    0.000    0.000    0.000 client.py:1055(putrequest)
       1    0.000    0.000    0.000    0.000 client.py:1182(_encode_request)
```

Figur 19: Kjøretiden til dekomprimering målt med cProfile

I tillegg bruker ujson til sammen over 350 sekunder på å tolke hver linje.

```
    6922    0.191    0.000    0.191    0.000 {built-in method ujson.dumps}
 7011379  367.772    0.000  367.772    0.000 {built-in method ujson.loads}
       1    0.000    0.000    0.000    0.000 {function HTTPResponse.flush at 0x
f9da36ee0d0}
       3    0.000    0.000    0.000    0.000 {function SocketIO.close at 0x7f9d
```

Figur 20: Kjøretiden til ujson.load() målt med cProfile

Den samlede kjøretiden var ca. 1500 sekunder fra dette eksempelet. Tidsangivelsene er hentet ut ved hjelp av cProfile som er en modul som brukes til å måle kjøretiden i programmer.

5.5.2 Hvordan den nødvendige informasjonen lagres

Noe av informasjonen hentes ut fortløpende etter hvert som linjene leses inn. Dette gjelder spesielt informasjonen som brukes i «Log Summary». Et eksempel er variabelen numberOfTraceroutes som brukes til å telle antall traceroutes i hele loggen.

```
    #Counts total traceroutes
    numberOfTraceroutes += 1
```

Figur 21: Teller antall traceroutes i hele RIPE Atlas-loggen

Et annet eksempel er «dictionary»-en som holder styr på hvilke unike prober vi har sett før.

```
#Checks if a probe has been counted before.
current_probe_id = data["prb_id"]
if current_probe_id not in uniqueProbeDictionary:
    uniqueProbeDictionary[current_probe_id] = 0
```

Figur 22: Holder styr på antall unike prober

På slutten kan vi simpelthen ta len(uniqueProbeDictionary) for å finne ut antall unike.

Dette er ganske enkelt så jeg skal i stedet her fokusere på hvordan man lagrer større mengder informasjon man trenger for å finne ut av ting som ikke kan fastslås på bakgrunn av enkelte

traceroutes. For eksempel om det har vært et brudd. Dette krever at man går gjennom alle traceroutes i loggen slik at man vet at man har alle som gjelder for en enkelt rute. Deretter må disse sorteres etter tid før man kan se om det har vært mulige brudd.

Jeg skal likevel ikke se på hvordan informasjon samles inn til topologier med RTT siden dette fungerer litt annerledes. Det kommer jeg tilbake til i kapittel 5.5.4.

Før jeg eventuelt lagrer informasjon om en rute, sjekker jeg først at tracerouten har en til/fra-adresse og at den faktisk ble sendt ut. Hvis det skjedde en feil med DNS-en proben bruker kan man for eksempel ende opp uten en destinasjonsadresse. På samme måte kan proben svikte fordi den ikke er koblet til Internett når tracerouten blir startet. Til tross for dette vil resultatet bli lagret og sendt videre til Atlas RIPE. Slike traceroutes er lite nyttige for oss.

Dette gjør vi ved hjelp av funksjonene getUniquePair() og checkForProbeFail().

```
#Gets identifier for route
uniquePair = getUniquePair(data)

#Makes sure that traceroutes that fail in the starting probe are
#skipped by making their uniquePair identifier say "None"
uniquePair = checkForProbeFail(data, uniquePair)
```

Figur 23: getUniquePair() og checkForProbeFail()

getUniquePair() er ganske enkel og ser ut som følger.

```
#Checks for source and destination address.
#Marks log entries without it which are DNS failures etc.
#These log entries are dropped by the function calling this one.
#Otherwise returns address in format "srcaddress/destaddress".

def getUniquePair(data):

    try:

        dst = str(data["prb_id"]) + "/" + data["dst_addr"]

        if data["from"] == "" or data["dst_addr"] == "" or str(data["prb_id"]) == "":
            dst = None

    except:
        dst = None

    return dst
```

Figur 24: Funksjonen getUniquePair()

Etter at den har blitt lastet inn med json.load() vil linjen være en Python-ordbok hvor vi kan hente ut attributter. Fra loggene beskrevet i kapittel 5.4.1 kjenner vi igjen «from» og «dst_addr». Dersom disse mangler eller er tomme, setter vi uniquePair til «None». Senere

kommer vi til å forkaste pakken dersom den har «None» i dette feltet. Merk at hvis alt går bra velger vi likevel å kalle den unike ruten opp etter probe ID og mottakerens IP.

Deretter sjekker vi om proben mislyktes ved det første hoppet.

```
#Checks if the probe failed to even start the traceroute
#Typical errors are "name resolution failed: non-recoverable
#failure in name resolution" and "connect failed: Network is unreachable"

def checkForProbeFail(data, uniquePair):

    if "error" in data["result"][0]:
        return None

    else:
        return uniquePair
```

Figur 25: Sjekker om proben klarte starte tracerouten

Merk at denne ikke vil slå ut dersom det første ruteren på veien gir en feilmelding siden den i så fall ville ligget i noe sånt som «data["result"][0]["result"][0]["err"]». En annen ting er at det kun er feil i maskinvaren som rapporteres som «error» ut ifra min erfaring.

Under kan man se et eksempel på den typen probefeil jeg sjekker etter.

{'fw': 5020, 'mver': '2.2.1', 'lts': 48, 'endtime': 1616627813, 'dst_name': 'hitlist-rr6.hitlist.sdstrowes.co.uk', 'ttr': 1.943565, 'proto': 'ICMP', 'af': 6, 'size': 48, 'paris_id': 0, 'result': [{'error': 'name resolution failed: non-recoverable failure in name resolution'}], 'msm_id': 24304869, 'prb_id': 12722, 'timestamp': 1616627813, 'msm_name': 'Traceroute', 'from': '2001:628:21d0:1:fa1a:67ff:fe4d:6a0d', 'type': 'traceroute', 'group_id': 24304869}

Tabell 12: Eksempel på en feil I startproben

Mens det følgende er en feil rapportert tilbake fra det første hoppet på ruten.

{'fw': 5020, 'mver': '2.2.1', 'lts': 54, 'endtime': 1616629865, 'dst_name': '2001:4248:f:0:102:163:100:12', 'dst_addr': '2001:4248:f:0:102:163:100:12', 'src_addr': '2a02:4680:3:b203:a62b:b0ff:fef1:596a', 'proto': 'ICMP', 'af': 6, 'size': 48, 'paris_id': 12, 'result': [{'hop': 1, 'result': [{'from': '2a02:4680:3:b203::1', 'ttl': 64, 'rtt': 0.396, 'size': 96, 'err': 'N'}, {'from': '2a02:4680:3:b203::1', 'ttl': 64, 'rtt': 0.361, 'size': 96, 'err': 'N'}, {'from': '2a02:4680:3:b203::1', 'ttl': 64, 'rtt': 0.353, 'size': 96, 'err': 'N'}]}], 'msm_id': 22332298, 'prb_id': 30525, 'timestamp': 1616629865, 'msm_name': 'Traceroute', 'from': '2a02:4680:3:b203:a62b:b0ff:fef1:596a', 'type': 'traceroute', 'group_id': 22332298}

Tabell 13: Eksempel på en feil etter ett hopp

Her kan man for øvrig merke seg at jeg for getUniquePair() har brukt try/except mens jeg for checkForProbeFail() har brukt «if x in y» for å sjekke om noe eksisterer. Grunnen til det er fordi try/except er mye raskere hvis try i de fleste tilfeller lykkes mens «if x in y» er raskere hvis

sjekken oftere er feil. (Gopakumar, 2020) Dette har jeg prøvd å ta hensyn til i hele programmet. Her vil man i de fleste tilfeller finne fra/til-adresser mens probefeil er relativt sjeldne.

Hvis en traceroute ikke passerer testene over vil den bli navngitt med «None» og da vil den altså bare bli forkastet. Hvis den passerer testene, sjekker vi om vi har lagret informasjon om den fra før av eller om vi nå trenger å sette opp lagring av informasjon om en ny rute.

```
if uniquePair is None:

    pass

else:


    #Initializes the memory if we have never seen this route before
    if uniquePair not in memory:
        cursor.execute(('INSERT INTO topologies (key, topology) values (?, ?)'), (marker, json.dumps(printRoute(data))))
        cursor.execute(('INSERT INTO rtt (key, rtt) values (?, ?)'), (marker, json.dumps(collectRtt(data))))
        initialize(data, memory, uniquePair, marker)
        marker = marker + 1

    #Otherwise adds it to existing memory
    else:
        cursor.execute(('INSERT INTO topologies (key, topology) values (?, ?)'), (marker, json.dumps(printRoute(data))))
        cursor.execute(('INSERT INTO rtt (key, rtt) values (?, ?)'), (marker, json.dumps(collectRtt(data))))
        updateMemory(data, memory, uniquePair, marker)
        marker = marker + 1
```

Figur 26: analyzeLog() sjekker hvordan hver logglinje skal lagres

Her ser man også linjene «cursor.execute()» og «marker = marker + 1». Dette er knyttet til topologidelen av programmet som jeg kommer tilbake til i kapittel 5.5.4 i stedet.

Under kan man se koden for initialize() som lager lagringsplass for en ny rute.

```
def initialize(data, memory, uniquePair, marker):

    memory[uniquePair] = {}
    memory[uniquePair][0] = {}
    memory[uniquePair][0]["ip_ver"] = data["af"]
    memory[uniquePair][0]["timeStamps"] = data["timestamp"]
    memory[uniquePair][0]["routeStatus"] = checkSuccess(data)
    memory[uniquePair][0]["from"] = data["from"]
    memory[uniquePair][0]["topology"] = marker
    memory[uniquePair][0]["rtt"] = marker

    if memory[uniquePair][0]["routeStatus"]:
        memory[uniquePair][0]["errors"] = "No error"
        memory[uniquePair][0]["dups"] = 0
        memory[uniquePair][0]["lates"] = 0
        memory[uniquePair][0]["lastReply"] = "Destination IP"
    else:
        lates, dups, errors = checkForErrors(data)
        memory[uniquePair][0]["errors"] = errors
        memory[uniquePair][0]["dups"] = dups
        memory[uniquePair][0]["lates"] = lates
        memory[uniquePair][0]["lastReply"] = findLastReplyIP(data)
```

Figur 27: Funksjonen initialize()

Vi ser først at det blir lagt en ny dictionary i «memory» med samme navnet som det unike ruteparet. Dette kan for eksempel være «6307/2001:500:200::b». Merk at dictionarien «memory» først blir opprettet i starten av analyzeLog før den første linjen analyseres. Dette er data som blir lagret direkte i minnet mens programmet kjører.

```
def analyzeLog(request):

    #Initializes data
    numberOfTraceroutes = 0
    successfulTraceroutes = 0
    failedTraceroutes = 0
    uniqueProbeCount = 0
    uniqueProbeDictionary = {}
    current_probe_id = 0
    lowestTimestamp = float('inf')
    highestTimestamp = 0
    memory = {}
    pairData = {}
    outageData = {}
    marker = 0
    probeFailCount = 0
```

Figur 28: Memory initialiseres av analyzeLog() før den begynner å lese linjer

I dictionarien for denne ruten lager vi deretter en tredje dictionary på posisjon 0. Dette er den første tracerouten vi har for denne ruten. Senere traceroutes vil lagres på posisjon 1, 2, 3, osv. Her kan vi begynne å lagre informasjon den spesifikke tracerouten. I «ip_ver» legger vi inn IP-versjon, altså om det er IPv4 eller IPv6. Grunnen til at vi trenger å vite dette er fordi vi må vite hvilken type IP det er når vi skal gjøre kall mot GeoIP-databasen senere.

I «timeStamps» lagrer vi når tracerouten ble startet. I «routeStatus» lagrer vi om tracerouten faktisk nådde frem til destinasjonen sin eller om det var en mislykket traceroute. For å gjøre dette bruker vi checkSuccess() som er en ganske enkel funksjon.

```
def checkSuccess(data):

    try:

        destination = data["dst_addr"]
        lastResult = data["result"][-1]["result"]

    except:
        return False

    for packet in lastResult:
        try:

            if packet["from"] == destination:
                return True

        except:
            pass

    return False
```

Figur 29: checkSuccess() sjekker om en traceroute nådde målet sitt

Først henter vi ut den siste gruppen med ICMP-svar. Deretter går vi gjennom hvert av de tre ping-svarene. Dersom en av dem kommer fra destinasjonen regnes tracerouten som vellykket. Vi må også lagre IP-en til avsenderen av tracerouten siden vi kun har probe ID i rutenavnet.

```
if memory[uniquePair][0]["routeStatus"]:
    memory[uniquePair][0]["errors"] = "No error"
    memory[uniquePair][0]["dups"] = 0
    memory[uniquePair][0]["lates"] = 0
    memory[uniquePair][0]["lastReply"] = "Destination IP"
else:
    lates, dups, errors = checkForErrors(data)
    memory[uniquePair][0]["errors"] = errors
    memory[uniquePair][0]["dups"] = dups
    memory[uniquePair][0]["lates"] = lates
    memory[uniquePair][0]["lastReply"] = findLastReplyIP(data)
```

Figur 30: Andre del av initialize()-funksjonen

Dersom checkSuccess() gav «true» lagrer vi bare en tom liste i «errors» og «Destination IP» i «lastReply». Dersom den er mislykket, må vi finne feilene og den siste IP-en vi fikk svar fra. Dette gjør vi ved hjelp av funksjonene checkforErrors() og findLastReplyIP(). Ved hjelp av checkForErrors() teller vi også antall eventuelle duplikater og sene svar.

```
def findLastReplyIP(data):

    #Grabs the set of last ICMP replies received

    result = data["result"][-1]["result"]

    notfound = True
    x = -1

    while notfound:

        #Searches each packet

        for packet in result:
            try:
                return packet["from"]
            except:
                pass


        #If an IP is not found, we jump one set of answers back
        #If we run out of sets of ICMP replies, then
        #there were no IPs in any of the replies received.
        #This does really occur sometimes.

        x = x - 1

        try:
            result = data["result"][x]["result"]
        except:
            return "No IP found in results"
```

Figur 31: findLastReplyIP() finner den siste IP-en en mislykket traceroute fikk svar fra

I likhet med checkSuccess() tar findLastReplyIP() tak i den siste pakken med resultater. Deretter sjekker den om svarene har en «from»-adresse. Ofte står det nemlig bare «no reply». Dersom det siste settet med tre svar ikke inneholder en IP, blar vi oss tilbake. Slik fortsetter vi til det eventuelt ikke finnes flere pakker å sjekke igjen. Dette hender faktisk noen ganger. I så fall skrives bare «No IP found in results» der det skulle vært en IP-adresse.

checkForErrors() er litt mer komplisert så vi skal se på den i flere deler.

```
for packet in lastResult:

    packetErrors = []


    #If a reply is late, it's discarded
    #since we can't tell what hop it is from (?)
    #We still choose to count it though

    if "late" in packet:
        lates = lates + 1
        continue


    #I believe dupes may just add confusion
    #when looking at the output of the program,
    #so I've chosen to discard those as well
    #We still choose to count it though

    if "dup" in packet:
        dups = dups + 1
        continue


    #If everything is all right so far, the error
    #message of the given reply is recorded

    try:
        packetErrors.append(explainError(packet["err"]))

    except:
        pass
```

Figur 32: Første del av checkForErrors()

Først lager vi en liste, packetErrors, hvor feilene fra de tre siste pakkene skal lagres. Vi lager også tellevariabler for duplikater og for sene meldinger vi ikke vet hvor hører til. Deretter sjekker vi om det er et svar som har kommet sent eller en såkalt «dup», altså et duplikat. Dersom det ikke er noen av delene, henter vi ut feilmeldingen.

Under kan man se et eksempel på hvordan et siste sett med pakker ser ut og hva vi henter ut.

```
{'hop': 13, 'result': [{'from':
```

'2001:470:1:cf::2', 'ttl': 57, 'rtt': 3482.009, 'size': 96, 'err': 'H'}, {'from': '2001:470:1:cf::2', 'ttl': 57, 'rtt': 3215.713, 'size': 96, 'err': 'H'}, {'from': '2001:470:1:cf::2', 'ttl': 57, 'rtt': 3310.423, 'size': 96, 'err': 'H'}]}

Tabell 14: Eksempel på svar vi henter feilmeldinger fra

Her ville vi hentet ut «H», «H» og «H». Siden dette er vanskelig for mennesker å forstå kjøres innholdet, som man ser over, gjennom en ny funksjon kalt explainError().

```
#Translates error codes into their corresponding textual meaning.
#Based on various RFC and documentation from RIPE Atlas
#https://atlas.ripe.net/docs/data_struct/#v5000_traceroute

def explainError(name):

    if name == "N":
        return "Network unreachable"
    elif name == "H":
        return "Destination unreachable"
    elif name == "A":
        return "Administratively prohibited"
    elif name == "P":
        return "Protocol unreachable"
    elif name == "p":
        return "Port unreachable"
    elif name == 6:
        return "IPv6: Route to destination rejected"
    elif name == 10:
        return "IPv4: Administratively prohibited"
    elif name == 5:
        return "IPv6: Source address not allowed"
    elif name == "h":
        return "Beyond scope"
    elif name == 9:
        return "IPv4: Destination network administratively prohibited"
    else:
        return(name)
```

Figur 33: explainError() gjør feilkoder til tekst som er mer lesbar for mennesker

Her ville «Destination unreachable» blitt sendt tilbake og lagret for hver av disse. Oversettelsene er hentet fra RIPE Atlas sin dokumentasjon og forskjellige RFC-dokumenter.

Men nå må vi tilbake til checkForErrors() igjen. Vi har bare sett på halve funksjonen.

```
    #If there was no error message, checks for other conditions

    if len(packetErrors) == 0:

        #Checks if we received a "No reply" in the format 'x':'*'

        if "x" in packet:
            packetErrors.append("No reply")

        #Check if there was a failure in the router

        elif "error" in packet:
            packetErrors.append(packet["error"])

        #If no explanation has yet been found, then the
        #traceroute is labeled as incomplete

        else:
            packetErrors.append("Incomplete")

    errors.append(packetErrors)

    #Sometimes the last set of ICMP replies is something like 5 times 'x':'*'.
    #This may be mostly lates and dupes from unknown hops.
    #This is commented out here, so that we get each reply

    #if len(errors) > 3:
    #    errors = [['Incomplete'], ['Incomplete'], ['Incomplete']]

return lates, dups, errors
```

Figur 34: Andre del av checkForErrors()

Dersom vi etter dette skal ha funnet 0 feil må vi selv prøve å finne en forklaring på hvorfor denne tracerouten aldri nådde hele veien til destinasjonen. Først sjekker vi etter «x: *» som betyr at ingen av disse pakkene fikk svar fra det siste stedet de kom til.

{'hop': 9, 'result': [{'x': '*'}, {'x': '*'}, {'x': '*'}]}

Tabell 15: Eksempel på svar uten feilmeldinger

Hvis det ikke er tilfellet, sjekker vi etter «error». Dette er ulike typer feil med ruteren på veien.

{'hop': 2, 'result': [{'from': '41.221.32.129', 'ttl': 254, 'size': 28, 'rtt': 1.726}, {'from': '41.221.32.129', 'ttl': 254, 'size': 28, 'rtt': 1.458}, {'error': 'sendto failed: Network is unreachable'}]}

Tabell 16: Eksempel på feil i ruterne underveis

«Sendto failed: Network is unreachable» indikerer for eksempel etter det jeg forstår at det er en feil med nettverksinnstillingene i maskinen som forsøker å videresende.

Hvis ingen av disse mulighetene viser seg å være riktige blir pakken merket som «Incomplete». Det kan for eksempel være at grensen for antall hopp en traceroute kan gjøre er overskredet. Det virker å være tilfellet noen ganger siden de ofte stopper på pene tall som 30 og 32.

Det kan også være proben ble slått av mens den var midt i en traceroute. Dette virker sannsynlig fordi det ofte tar lang tid før det kommer nye traceroutes etter en «Incomplete».

Til slutt hadde jeg en litt kontroversiell funksjonalitet som nå er kommentert ut.

Dersom det har blitt registrert noe på mer enn 3 forskjellige pakker, så slettet jeg disse og skrev bare «Incomplete», «Incomplete», «Incomplete» for tre pakker. Grunnen til dette er at det etter det jeg har sett alltid er snakk om 5-6 «No Replies» eller lignende. Min teori er at dette er en blanding av «dupes» og «lates» fra maskiner som ikke vil svare. Det gjør det vanskelig å si sikkert hva feilen fra de tre siste pakkene faktisk var. Eksempel under.

{'hop': 10, 'result': [{'x': '*'}, {'x': '*'}, {'x': '*'}, {'x': '*'}, {'x': '*'}]}

Tabell 17: Eksempel på svar med altfor mange "No Reply"

Med dette har vi kommet oss gjennom og forklart initialize()-funksjonen. Nå skal vi se på updateMemory() som i stedet brukes hvis vi allerede har informasjon om en rute. updateMemory() er relativt lik initialize() så dette burde gå mye raskere.

```
def updateMemory(data, memory, uniquePair, marker):

    memory[uniquePair][len(memory[uniquePair])] = {}
    memory[uniquePair][len(memory[uniquePair])-1]["timeStamps"] = data["timestamp"]
    memory[uniquePair][len(memory[uniquePair])-1]["routeStatus"] = checkSuccess(data)
    memory[uniquePair][len(memory[uniquePair])-1]["from"] = data["from"]
    memory[uniquePair][len(memory[uniquePair])-1]["ip_ver"] = data["af"]
    memory[uniquePair][len(memory[uniquePair])-1]["topology"] = marker
    memory[uniquePair][len(memory[uniquePair])-1]["rtt"] = marker

    if not checkSuccess(data):
        lates, dups, errors = checkForErrors(data)
        memory[uniquePair][len(memory[uniquePair])-1]["dups"] = dups
        memory[uniquePair][len(memory[uniquePair])-1]["lates"] = lates
        memory[uniquePair][len(memory[uniquePair])-1]["errors"] = errors
        memory[uniquePair][len(memory[uniquePair])-1]["lastReply"] = findLastReplyIP(data)

    else:
        memory[uniquePair][len(memory[uniquePair])-1]["errors"] = "No error"
        memory[uniquePair][len(memory[uniquePair])-1]["dups"] = 0
        memory[uniquePair][len(memory[uniquePair])-1]["lates"] = 0
        memory[uniquePair][len(memory[uniquePair])-1]["lastReply"] = "Destination IP"


    sortMemoryByTimestamps(data, memory, uniquePair)
```

Figur 35: Funksjonen updateMemory()

Alt er faktisk helt likt, bare at vi ikke trenger lage en dictionary i memory[uniquePair] for ruten. I tillegg skal vi ikke legge informasjonen inn i indeks 0. I stedet legger vi den inn i

len(memory[uniquePair]) som er lengden på memory[uniquePair], altså antall traceroutes. Siden indeksen teller fra null og len() teller fra 1, så vil len() alltid være den første ledige indeksen. Etterpå har vi imidlertid økt lengden og for at ikke len(memory[uniquePair]) bare skal vokse og vokse så legger vi informasjonen inn i indeksen len(memory[uniquePair]) minus 1.

En annen forskjell er at vi på slutten sorterer traceroute-informasjonen basert på tid med sortMemoryByTimestamps() slik at den kronologisk første tracerouten havner i indeks 0, den kronologisk andre tracerouten havner i indeks 1, osv.

```
#Since RIPE Atlas logs are not sorted by time,
#we must do the sorting ourselves to determine outages

def sortMemoryByTimestamps(data, memory, uniquePair):

    for x in range(len(memory[uniquePair])-1, -1, -1):

        if data["timestamp"] < memory[uniquePair][x]["timeStamps"]:

            temp = memory[uniquePair][x]

            memory[uniquePair][x] = memory[uniquePair][x+1]
            memory[uniquePair][x+1] = temp
```

Figur 36: Funksjonen sortMemoryByTimestamps()

Dette er en variant av Insertion Sort som er en veldig enkel sorteringsalgoritme som er rask på små lister som allerede er ganske sortert fra før. (Cormen, m. fl, 2009, s. 16-19) cProfile() viser at denne sorteringen utgjør en liten del av den totale kjøretiden til programmet.

```
  2173304    8.849    0.000   [illegible]    0.000 atlas-gap-ana.py:250(initialize)
  4784105   32.336    0.000   32.790    0.000 atlas-gap-ana.py:273(sortMemoryByT
imestamps)
  1701572    5.818    0.000    6.262    0.000 atlas-gap-ana.py:290(countOutages)
```

Figur 37: Kjøretiden til sortMemoryByTimestamps beregnet av cProfile()

5.5.3 Hvordan informasjonen analyseres og skrives ut

Etter at programmet har kjørt ferdig kan vi analysere og skrive ut de nye loggene.

```
#Prepares final data for writing to JSON
memory = cleanUp(memory, cursor) #Creates outage reports and deletes unused data
pairData = preparePairData(memory, topology, cursor) #Creates the JSON files for each unique pair report
memory = exchangeTopology(memory, cursor) #Deletes successful route topologies and gets unsuccessful from SQLite database
outageData = prepareOutageData(memory) #Creates the JSON reports for each outage
uniqueProbeCount = len(uniqueProbeDictionary)
totalData = prepareTotalData(numberOfTraceroutes, successfulTraceroutes,
failedTraceroutes, uniqueProbeCount, lowestTimestamp, highestTimestamp)
```

Figur 38: Koden i analyzeLog() etter at hver linje har blitt lest

Her kan vi igjen se bort fra alt som har med topologiene å gjøre som vi kommer tilbake til.

Det første vi gjør her er at vi kjører cleanUp() som lager informasjonen vi trenger om hvert utfall ved hjelp av analyzeEachOutage(). Deretter kvitter den seg med den overflødige informasjonen.

```
def cleanUp(memory, cursor):
    newmemory = {}
    for x in memory.copy():
        if len(memory[x]) > 1:
            if countOutages(memory, x) > 0:
                newmemory[x] = analyzeEachOutage(memory, x, cursor)
    return newmemory
```

Figur 39: Koden til cleanUp()

Her er det to ting jeg gjerne vil forklare først. Det første er grunnen til at jeg bruker memory.copy(). Hvis man endrer på en dictionary mens man går gjennom den i en løkke, så stopper Python. Det andre er hvorfor jeg kvitter meg med så mye informasjon. For det første tenker jeg at det er unødvendig å bruke mer RAM enn man må til enhver tid. Det andre er at vi nå lager informasjon om hvert utfall. I tillegg skal vi straks lage dictionariene som skal skrives ut. Derfor kan det være greit å rydde frem litt plass til denne nye dataen.

Før vi ser på analyzeEachOutage(), kan vi raskt se på countOutages(). Denne funksjonen brukes til å sjekke fort om det finnes utfall i en rute før vi eventuelt analyserer dem.

```
def countOutages(memory, uniquePair):

    currentState = memory[uniquePair][0]["routeStatus"]
    outage = not currentState
    posfound = False

    if outage:
        outages = 1
    else:
        outages = 0

    for x in range(1, len(memory[uniquePair]), 1):

        if currentState != memory[uniquePair][x]["routeStatus"]:
            outage = not memory[uniquePair][x]["routeStatus"]
            currentState = memory[uniquePair][x]["routeStatus"]

            if outage == True:
                outages += 1
            else:
                posfound = True

    if not posfound:
        outages = 0

    return outages
```

Figur 40: countOutages() teller antall utfall på en rute

Denne funksjonen teller antall ganger ruten har vært nede etter definisjonen vår. Det vil si at det er minst én traceroute som blir mislykket og minst én traceroute som når målet.

Nå kan vi analysere hvert brudd ved hjelp av analyzeEachOutage(). Merk at det følgende sannsynligvis er den mest komplekse delen av programmet og at vi tar det i deler.

```
def analyzeEachOutage(memory, uniquePair, cursor):

    outages = countOutages(memory, uniquePair)
    index = 0 #Used to track position in the traceroutes as they are read for outages
    newmemory = {} #We'll store everything here and drop the old memory file to save RAM.
    totalOutageDuration = 0 #Used to track total duration of all outages for a route.
    incomplete = 0 #Used to track total amount of incomplete traceroutes in a route
```

Figur 41: analyzeEachOutage() initialiseres

Vi starter her med å telle antall utbrudd nok en gang samt lage variabler hvor vi skal lagre informasjon om alle utfallene og ikke om individuelle utfall. For eksempel total varighet.

Deretter jobber vi oss gjennom ett og ett utbrudd om gangen.

```
for r in range(0, outages):

    newmemory[r] = {}

    markers = [] #Saing the SQLlite ids of failed routes so they can be retrieved
    #after we have used and deleted the successful routes.
    #They are not immediately retrieved to save RAM.
    failrtts = [] #Does the same for their RTTs.
    srcadr = [] #Collects src adresses of each traceroute in the outage in the very unlikely case they differ
    outageErrors = [] #Used to track errors per outage
    failures = 0 #Used to track amount of failed traceroutes in an outage
    frequency = 0 #Used to calculate polling frequency
    begins_in_other_hour = False
    end_in_other_hour = False
    freqchange = 0 #Counts amount of time differences recorded, used to find polling frequency
```

Figur 42: analyzeEachOutage() jobber seg gjennom ett og ett brudd

Her ser vi at vi starter med å lage variabler for hvert enkelt utbrudd.

Det første vi gjør er å bla oss frem til en første traceroute som ikke når målet sitt. Det er her det første utbruddet per definisjon er nødt til å begynne.

```
#Goes through log entries until the status changes to false again.
#The other possibility is that we run out of log entries to check.

while (memory[uniquePair][index]["routeStatus"] == True) and index < len(memory[uniquePair]):
    index = index + 1
```

Figur 43: Blar gjennom traceroutes til vi finner starten på utfallet

Når vi finner en mislykket traceroute etter en vellykket, lagrer vi informasjon om starten.

```
#If the status is false, this is recorded as the start of an outage.
#All information needed is collected.

start = memory[uniquePair][index]["timeStamps"]
markers.append(memory[uniquePair][index]["topology"])
failrtts.append(getRtt(memory[uniquePair][index]["rtt"], cursor))
srcadr.append(memory[uniquePair][index]["from"])

if index > 0:
    frequency = memory[uniquePair][index]["timeStamps"] - memory[uniquePair][index-1]["timeStamps"]
    freqchange += 1

outageErrors.append(memory[uniquePair][index]["errors"])
finalReply = memory[uniquePair][index]["lastReply"]
failures = 1
index = index + 1

#If the log started on an outage we want to make a note of that

if index == 0:
    begins_in_other_hour = True
```

Figur 44: Lagrer informasjon om starten på utfallet

Her kan vi se at vi lagrer feil, setter antallet traceroutes bruddet består av til 1 osv. Det neste vi gjør er at vi blar oss gjennom traceroutene. For hver traceroute som fortsatt er mislykket, lagrer vi mer informasjon om dette utfallet. Dersom vi finner en slutt på det, gjør vi opp regnskap og lagrer den siste informasjonen. Blant annet lengden på utfallet.

```
while notfound and (index in memory[uniquePair]):

    #While the outage is still ongoing, we keep adding information about it.

    if (memory[uniquePair][index]["routeStatus"] == False):
        outageErrors.append(memory[uniquePair][index]["errors"])
        markers.append(memory[uniquePair][index]["topology"])
        failures = failures + 1
        frequency = (memory[uniquePair][index]["timeStamps"] - memory[uniquePair][index-1]["timeStamps"]) + frequency
        freqchange += 1
        failrtts.append(getRtt(memory[uniquePair][index]["rtt"], cursor))
        srcadr.append(memory[uniquePair][index]["from"])
        index = index + 1

    #When the outage ends, we record information about this.

    elif (memory[uniquePair][index]["routeStatus"] == True):

        end = memory[uniquePair][index]["timeStamps"]
        frequency = (memory[uniquePair][index]["timeStamps"] - memory[uniquePair][index-1]["timeStamps"]) + frequency
        freqchange += 1
        frequency = frequency/freqchange #Calculates average polling
        duration = (end - start) #Calculates duration of each outage
        notfound = False
```

Figur 45: Lagrer informasjon om bruddet mens det foregår

Det hender imidlertid at vi går tom for traceroutes og at det et brudd fortsatt foregår. Dette betyr at bruddet ikke avsluttes innenfor timen vi holder på å analysere.

```
#If we ran out of traceroutes to check while the outage was still ongoing,
#we have to record the time it ends and the duration as unknown.

if not (index in memory[uniquePair]):
    end_in_other_hour = True
    end = memory[uniquePair][index-1]["timeStamps"]
    duration = (end - start) #Calculates duration of each outage
```

Figur 46: Setter duration til "Unknown" hvis utfallet ikke har en slutt

Det neste vi gjør er å sjekke antall «Incomplete» traceroutes i dette utfallet.

```
#We count the amount of incomplete traceroutes.

for msg in outageErrors[0]:
    if str(msg) == "['Incomplete']":
        incomplete = incomplete + 1
        break
```

Figur 47: Teller antall ufullstendige traceroutes

Vi må også plusse sammen varigheten for å finne total varighet.

```
#We add the duration from the last
#outage to the total duration of outages.

totalOutageDuration = totalOutageDuration + duration
```

Figur 48: Plusser sammen varighet

Nå kan vi skrive all informasjonen vi har samlet inn om det enkelte utfallet.

```
newmemory[r]["startTime"] = start
newmemory[r]["endTime"] = end
newmemory[r]["failures"] = failures
newmemory[r]["duration"] = duration
newmemory[r]["fromfalse"] = srcadr
newmemory[r]["lastReply"] = finalReply
newmemory[r]["errors"] = outageErrors
newmemory[r]["markers"] = markers
newmemory[r]["polling_time"] = int(frequency)
newmemory[r]["begins_in_other_hour"] = begins_in_other_hour
newmemory[r]["end_in_other_hour"] = end_in_other_hour
newmemory[r]["failrtts"] = failrtts
```

Figur 49: Lagrer alt vi vet om det enkelte utfallet et sted hvor vi finner det igjen

Dersom dette er det siste utfallet, må vi også lagre informasjonen om alle utfall. Vi lagrer denne sammen med det første utfallet (0) hvor vi lett finner frem til det igjen.

```
if r+1 == outages:

    newmemory[0]["incomplete"] = incomplete
    newmemory[0]["ip_ver"] = memory[uniquePair][0]["ip_ver"]
    newmemory[0]["total_outage_duration"] = totalOutageDuration
    newmemory[0]["length"] = len(memory[uniquePair])

    #We also need to get all the probe_ids before we delete the old memory
    #And get the successful route topologies from the SQLlite database with their RTT data
    #Additionally we want to get the total dup/late count

    newmemory[0]["from"] = []
    newmemory[0]["whole_routes"] = []
    newmemory[0]["total_dups"] = 0
    newmemory[0]["total_lates"] = 0
    newmemory[0]["rtt"] = []


    for x in memory[uniquePair]:

        newmemory[0]["from"].append(memory[uniquePair][x]["from"])
        newmemory[0]["total_dups"] = newmemory[0]["total_dups"] + memory[uniquePair][x]["dups"]
        newmemory[0]["total_lates"] = newmemory[0]["total_lates"] + memory[uniquePair][x]["lates"]
```

Figur 50: Lagrer alt vi vet om alle utfallene knyttet til denne ruten

Nå som vi har all informasjonen vi trenger kan vi skrive ut ruteoppsummeringene.

```
def preparePairData(memory):

    pairData = {}

    for x in memory:

        pairname = x.replace("/", "-to-")
        pair = x.split("/")

        allerrors = []
        failed = 0

        for y in memory[x]:
            allerrors.append(memory[x][y]["errors"])
            failed = failed + memory[x][y]["failures"]

        if memory[x][0]["whole_routes"] != []:
            maxm, minm, top = analyzeTopology(memory[x][0]["whole_routes"], memory[x][0]["rtt"])
        else:
            maxm = 0
            minm = 0
            top = "No successful routes known"
```

Figur 51: Lager ruteoppsummeringen som skal skrives ut som JSON

Her lager vi en dictionary for hver ruteoppsummering og blar oss gjennom rutene. Vi starter hver rute med å formatere navnet fra deling med «/» til «-to-» fordi det ellers blir formateringsproblemer i JSON. Vi samler også inn alle feilene fra hver enkelt feil i hver traceroute. Igjen kan vi se bort fra topologiene som vi kommer tilbake til i det neste kapittelet.

Vi må også hente ut IP-adressen til avsenderen. Vi sjekker først at alle avsenderadressene er like, det er de i de fleste tilfeller, og legger i så fall denne inn i oppsummeringen. Dersom en probe ID har brukt flere offentlige IP-er legger vi dem alle med i kronologisk rekkefølge.

```
if len(set(memory[x][0]["from"])) > 1:
    srcadr = memory[x][0]["from"]
else:
    srcadr = memory[x][0]["from"][0]
```

Figur 52: Sjekker at en probe kun har brukt én offentlig IP

Nå kan vi skrive all informasjonen ut til en «dictionary» vi skal oversette til JSON.

```
entry = {
"pair_id": pairname,
"event_type": "summary",
"dst_addr": pair[1],
"src_addr": srcadr,
"prb_id": pair[0],
"start_time": memory[x][0]["startTime"],
"last_response": memory[x][len(memory[x])-1]["endTime"],
"traceroutes_recorded": memory[x][0]["length"],
"failed_traceroutes": str(round((failed/memory[x][0]["length"]) * 100)) + "%",
"outage_count": len(memory[x]),
"outage_at_log_start": memory[x][0]["begins_in_other_hour"],
"outage_at_log_end": memory[x][len(memory[x])-1]["end_in_other_hour"],
"incomplete_outages": memory[x][0]["incomplete"],
"total_dupes": memory[x][0]["total_dups"],
"total_late": memory[x][0]["total_lates"],
"total_outage_errors": countErrorTypes(allerrors, True),
"total_outage_duration_in_seconds": memory[x][0]["total_outage_duration"],
"max_hop": maxm,
"min_hop": minm,
"topology": top
}
```

Figur 53: JSON-loggen begynner å ta form

Vi kunne fortløpende skrevet disse ut og deretter slettet all informasjonen, men hvis jeg venter litt med det kan jeg skrive ut utfallsoppsummeringene etter hver ruteoppsummering. Når man da ser direkte på de nye loggfilene synes jeg det er litt mer oversiktlig.

Her kan vi også nevne countErrorTypes() som brukes for å finne total_outage_errors. Dette er en enkel funksjon som bare går gjennom alle feilmeldingene og teller antallet typer.

```
def countErrorTypes(errors, mode):

    arr = {}

    if mode:
        for x in errors:
            for y in x:
                for z in y:
                    if str(z[0]) in arr:
                        arr[str(z[0])] += 1
                    else:
                        arr[str(z[0])] = 1
    else:
        for y in errors:
            for z in y:
                if str(z[0]) in arr:
                    arr[str(z[0])] += 1
                else:
                    arr[str(z[0])] = 1

    return arr
```

Figur 54: Funksjonen countErrorTypes()

Mode (som enten er «True» eller «False») brukes til å skille om vi teller en serie lister med feil fra forskjellige utbrudd som i ruteoppsummeringen eller om vi bare teller én liste som i utbruddsoppsummeringene. Et typisk eksempel på output fra funksjonen ser man under.

{"No reply":3}

Tabell 18: Typisk output fra countErrorTypes()

Utfallsoppsummeringene lages på en ganske lik måte.

```
def prepareOutageData(memory):

    outageData = {}

    for y in memory:

        #Checks if there are any outages for this route
        #using data made by preparePairData()

        pairname = y.replace("/", "-to-")
        pair = y.split("/")
        ipv = memory[y][0]["ip_ver"] #Gets the IP version for use by GeoIP2
        collection = {} #Stores data for all outages in each route, added to outageData in the end


        for x in memory[y]:

            #Gets the min and max lengths of the failed routes
            listlen = [len(i) for i in memory[y][x]["topology"]]
            maxm = max(listlen)
            minm = min(listlen)

            #Gets the GeoIP2 information
            AS_number, AS_name, location, latitude, longitude = useGeoIP(memory[y][x]["lastReply"], ipv)

            maxm, minm, top = analyzeTopology(memory[y][x]["topology"], memory[y][x]["failrtts"])

            if len(set(memory[y][x]["fromfalse"])) > 1:
                srcadr = memory[y][x]["fromfalse"]
            else:
                srcadr = memory[y][x]["fromfalse"][0]
```

Figur 55: Lager utfallsoppsummeringene

En viktig forskjell er at vi her også må hente GeoIP-informasjon med useGeoIP().

```
#Gets AS and location information from GeoIP2

def useGeoIP(lastReply, ipv):

    #In case the lookups fail and also so the program will still
    #otherwise work if it's not run on a server with GeoIP2 setup like this

    AS_number = "Unknown"
    AS_name = "Unknown"
    location = "Unknown"
    latitude = "Unknown"
    longitude = "Unknown"
```

Figur 56: Henter ut GeoIP-informasjon

Vi setter først «Unknown» som standard i tilfelle vi ikke skulle finne informasjon.

```
#Mostly doing try/exceptions here because it's faster
#when the lookups are mostly successful

try:

    with geoip2.database.Reader('/usr/share/GeoIP/GeoLite2-City.mmdb') as reader:
        response = reader.city(lastReply)

        #In many cases there is no city name
        try:
            location = response.country.name + ", " + response.city.name
        except:
            location = response.country.name

        try:
            latitude = response.location.latitude
            longitude = response.location.longitude
        except:
            pass

except:
    pass
```

Figur 57: Henter ut navn på sted og koordinater

Vi bruker try/except fordi databasen ikke alltid inneholder informasjonen vi trenger. Spesielt mangler den «city» ofte, men navnet på landet pleier alltid å ligge inne.

AS-numrene fant jeg på en annen database hos Uninetts-tjenere som har endelsen «.dat». For å kunne hente informasjon fra denne var jeg nødt til å bruke et eget program kalt pygeoip.

```
#AS Numbers from seperate database

try:
    if ipv == 4:
        gi = pygeoip.GeoIP('/usr/share/GeoIP/GeoIPASNum.dat')
    elif ipv == 6:
        gi = pygeoip.GeoIP('/usr/share/GeoIP/GeoIPASNumv6.dat')

    result = gi.org_by_addr(lastReply)
    AS_number = result.split(' ', 1)[0]
    AS_name = result.split(' ', 1)[1]

except:
    pass

return AS_number, AS_name, location, latitude, longitude
```

Figur 58: AS-informasjonen hentes fra en seperat database

Nå kan vi fylle ut oppsummeringene for hvert av utfallene.

```
outage = {
    "pair_id": pairname,
    "event_type": "outage",
    "order": x+1,
    "dst_addr": pair[1],
    "src_addr": srcadr,
    "prb_id": pair[0],
    "start_time": memory[y][x]["startTime"],
    "last_response": memory[y][x]["endTime"],
    "outage_at_log_start": memory[y][x]["begins_in_other_hour"],
    "outage_at_log_end": memory[y][x]["end_in_other_hour"],
    "failure_count": memory[y][x]["failures"],
    "polling_frequency": round(memory[y][x]["polling_time"]),
    "duration": memory[y][x]["duration"],
    "last_reply_from": memory[y][x]["lastReply"],
    "AS_number": AS_number,
    "AS_owner": AS_name,
    "location": location,
    "coordinates": str(latitude) + "," + str(longitude),
    "outage_errors": countErrorTypes(memory[y][x]["errors"], False),
    "error_type": classifyOutage(memory[y][x]["errors"]),
    "max_hop": maxm,
    "min_hop": minm,
    "topology": top
}
```

Figur 59: Utfallsoppsummeringen fylles ut

Før vi kan lage oppsummeringen for hele loggen må vi merke oss lengden på dictionarien vi har brukt til å holde styr på hvilke prober vi har sett. Denne gir oss antall unike prober.

```
uniqueProbeCount = len(uniqueProbeDictionary)
totalData = prepareTotalData(numberOfTraceroutes, s
```

Figur 60: Antall unike prober blir telt

Merk at vi også til slutt sletter coordinates dersom disse er «Unknown». Dette er for å unngå problemer med Elasticsearch som forventer å se koordinatnumre her.

```
#Returns log without coordinates if latitude and longitude is unknown
if latitude == "Unknown" or longitude == "Unknown":
    del outage["coordinates"]
```

Figur 61: Sletter koordinatattributtet dersom det er ukjent

Her kan vi også nevne en ny funksjon som blir brukt, classifyOutage().

Først teller den antallet ganger de forskjellige feilmeldingene dukker opp.

```
def classifyOutage(errors):

    cause = "Unknown"
    arr = {}

    #Counts each error type

    for x in errors:
        for y in x:
            if str(y) in arr:
                arr[str(y)] += 1
            else:
                arr[str(y)] = 1
```

Figur 62: Første del av classifyOutage()

Deretter sjekker den om det er én tydelig feilmelding som dominerer eller om det ikke er det. Hvis det ikke er det, blir utfallet kategorisert som «various causes». Dette er sjeldent.

```
#Makes sure there are not more than
#one "most common" error message

if len(arr) == 1:
    key = next(iter(arr))
else:
    key = max(arr, key=arr.get)
    count = 0
    for x in y:
        if x == arr[key]:
            count += 1
    if count > 1:
        return "Various causes"
```

Figur 63: Andre del av classifyOutage()

Vi skriver ellers ut navnet på det vi mener er hovedfeilen. Hvis det stort sett er «No reply», har vi valgt å kalle dette «Stopepd route». Incompletes gir «Not enough information». Ellers klipper man bare feilmeldingen ut. Dette kan for eksempel være «Administratively prohibited».

```
if key == "['No reply']":
    cause = "Stopped route"
elif key == "['Incomplete']":
    cause = "Not enough information"
else:
    cause = key[2:-2]
```

Figur 64: Tredje del av classifyOutage()

Nå kan vi kjøre prepareTotalData() og lage den siste typen loggen vi skal ha med oss.

```
def prepareTotalData(numberOfTraceroutes, successfulTraceroutes, failedTrac

    totalData = {
        "total_traceroutes": numberOfTraceroutes,
        "event_type": "log_summary",
        "start_time": lowestTimestamp,
        "end_time": highestTimestamp,
        "total_succeeded": successfulTraceroutes,
        "total_failed": failedTraceroutes,
        "failed_at_probe": probeFailCount,
        "total_probes": uniqueProbeCount,
        "version": "1.63 atlas-gap-ana aleksandra jekic"
    }

    return totalData
```

Figur 65: Logg for hele timen

Denne bruker bare informasjon analyzeLog() har lagret underveis.

Det siste programmet gjør er å slette sql-databasen vi har brukt til topologiene, som vi kommer til i neste kapittel, og skrive ut alle loggene vi har forberedt til JSON.

```
#Finally writes all reports to JSON
writeToJSON(totalData, pairData, outageData, request)
#Deletes log file we've finished with
os.remove(str(request) + ".db")
print("Completed analyzing:", request)
```

Figur 66: Kommer tilbake til topologi i neste kapittel

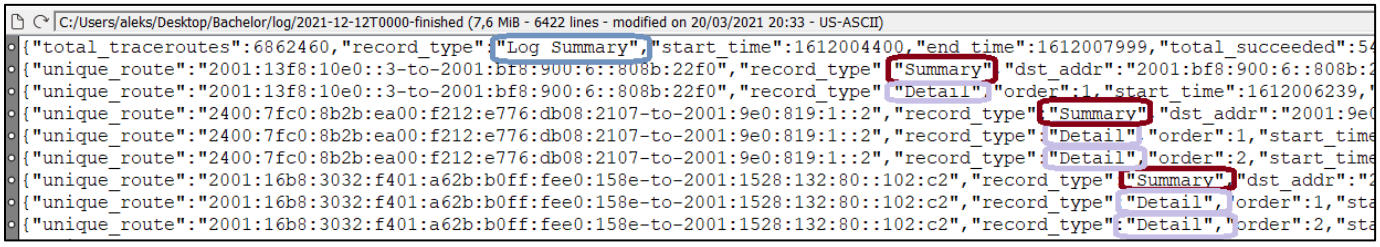
Vi kan kort se på writeToJSON() som er en ganske enkel funksjon her.

```python
#Writes all information prepared to a JSON file

def writeToJSON(totalData, pairData, outageData, request):

    path = "log/" + str(request) + "-finished"

    with open(path, "w") as outfile:
        outfile.write(json.dumps(totalData) + "\n")
        for x in pairData:
            outfile.write(json.dumps(pairData[x]) + "\n")
            try:
                for y in outageData[x]:
                    outfile.write(json.dumps(outageData[x][y]) + "\n")
            except:
                pass
```

Figur 67: Funksjonen som skriver ut JSON

Den lager en fil i mappen «log» der hvor programmet kjøres med samme navn på forespørselen med «-finished» lagt til. For eksempel «2021-03-17T0100-finished».

Først skrives oppsummeringen for den totale loggen ut. Deretter skrives en ruteoppsummering ut etterfulgt av en oppsummering av hvert utfall som er knyttet til denne ruten. Hver logg skrives ut på sin egne linje. Her kan man se hvordan det ser ut i glogg.



Figur 68: Loggene i ferdig form. Merk at bildet er fra en gammel versjon av programmet.

5.5.4 Hvordan programmet samler inn og bruker topologiene

Som jeg flere ganger har nevnt er topologiene lagret litt annerledes for å spare plass. Grunnen til dette er at topologiene er det som bruker mest minne i hele programmet. Samtidig skal vi bruke veldig få av disse topologiene når det kommer til stykket. Det finnes for eksempel veldig mange traceroutes som er alene som det ikke er vits å lage bruddrapporter om eller serier med traceroutes hvor det ikke finnes noen utfall som vi dermed ikke er interessert i.

Derfor har jeg gjort en liten byttehandel. Istedenfor å lagre direkte i minnet, som koster oss mye, slik at vi kan hente informasjon raskt ut har jeg i stedet lagret på harddisk. Dette tar lengre tid å hente ut igjen, men fordi vi skal hente relativt få topologier ut skader dette oss ikke.

```
connection = sqlite3.connect(str(request) + ".db")
cursor = connection.cursor()
```

Figur 69: Kobler oss til databasefilen

Før vi begynner å laste inn linjer fra RIPE Atlas i analyzeLog()-funksjonen, begynner vi med å opprette en forbindelse til en sqllite3-database som vi kaller opp etter forespørselen. Det blir for eksempel «2021-03-12T0100.db». Dersom databasen ikke eksisterer fra før av vil sqlite3-modulen automatisk opprette en database i samme mappe som programmet nå.

Deretter kobler vi oss til med connection.cursor(). Tilkoblingen vår ligger nå i «cursor». Vi begynner med å prøve å droppe tabellen i tilfelle databasen allerede har blitt brukt. Dette kan skje hvis det forrige forsøket til programmet ikke avsluttet.

```
try:
    cursor.execute("DROP TABLE topologies")
    cursor.execute("DROP TABLE rtt")
    cursor.execute("CREATE TABLE topologies (key TEXT, topology TEXT)")
    cursor.execute("CREATE TABLE rtt (key TEXT, rtt TEXT)")
except:
    cursor.execute("CREATE TABLE topologies (key TEXT, topology TEXT)")
    cursor.execute("CREATE TABLE rtt (key TEXT, rtt TEXT)")
```

Figur 70: Prøver å droppe en gammel tabell først for sikkerhets skyld

Hvis vi ikke får til å slette tabellen, oppretter vi den bare. Hver enkeltlagring består av en unik nøkkel vi kan bruke til å finne frem til topologien og RTT-informasjonen dens.

Det siste vi gjør er å lage en indeks for tabellen. Om vi ikke gjør dette blir søkene vi skal gjøre helt til slutt i programmet såpass trege at programmet blir helt ubrukelig.

```
#Makes lookups much faster
cursor.execute("CREATE UNIQUE INDEX idx_key1 ON topologies (key)")
cursor.execute("CREATE UNIQUE INDEX idx_key2 ON rtt (key)")
```

Figur 71: Søkehastigheten blir drastisk høyere med indeksering

Hvis vi nå husker tilbake til dette veiskillet, burde vi forstå mye mer.

```
#Initializes the memory if we have never seen this route before
if uniquePair not in memory:
    cursor.execute(('INSERT INTO topologies (key, topology) values (?, ?)'), (marker, json.dumps(printRoute(data))))
    cursor.execute(('INSERT INTO rtt (key, rtt) values (?, ?)'), (marker, json.dumps(collectRtt(data))))
    initialize(data, memory, uniquePair, marker)
    marker = marker + 1

#Otherwise adds it to existing memory
else:
    cursor.execute(('INSERT INTO topologies (key, topology) values (?, ?)'), (marker, json.dumps(printRoute(data))))
    cursor.execute(('INSERT INTO rtt (key, rtt) values (?, ?)'), (marker, json.dumps(collectRtt(data))))
    updateMemory(data, memory, uniquePair, marker)
    marker = marker + 1
```

Figur 72: Topologiene lagres fortløpende i databasen

Før vi kjører initialize() eller updateMemory(), legger vi et unikt identifikasjonsnummer inn i databasen vår sammen med selve topologien vår og RTT-informasjonen. Vi lagrer informasjonen som JSON for å gjøre det lettere å pakke den ut igjen senere. Ellers kan det bli veldig komplisert å oversette strenger til lister som ligger i lister.

I initialize() eller updateMemory() lagrer vi så identifikasjonsnummeret med tracerouten.

```
def initialize(data, memory, uniquePair, marker):

    memory[uniquePair] = {}
    memory[uniquePair][0] = {}
    memory[uniquePair][0]["ip_ver"] = data["af"]
    memory[uniquePair][0]["timeStamps"] = data["timestamp"]
    memory[uniquePair][0]["routeStatus"] = checkSuccess(data)
    memory[uniquePair][0]["from"] = data["from"]
    memory[uniquePair][0]["topology"] = marker
    memory[uniquePair][0]["rtt"] = marker

    if memory[uniquePair][0]["routeStatus"]:
        memory[uniquePair][0]["errors"] = "No error"
        memory[uniquePair][0]["dups"] = 0
```

Figur 73: Vi lagrer bare nøkkelen til topologien i minnet

Før vi går videre må vi likevel se på printRoute(). Hvordan finner vi topologiene?

```python
def printRoute(data):

    result = data["result"]
    arr = []

    for x in result:

        hop = []

        for y in x["result"]:
            if "from" in y:
                hop.append(y['from'])
            elif "x" in y:
                hop.append(y['x'])
            elif "error" in y:
                hop.append(y["error"])

        arr.append(hop)

    return arr
```

Figur 74: printRoute() skriver ut ruten en traceroute tar

Først henter vi ut resultatene fra selve tracerouten. Deretter går vi gjennom hver trio med svar fra ping-pakker. Vi forsøker å hente ut en svaradresse, en «No Reply» og til slutt en feil.

Vi bør også se kort på collectRTT som brukes til å hente all RTT-informasjonen. Denne fungerer ganske likt ved at den blar seg gjennom og samler inn all RTT informasjon. Dersom den er «Late» eller den ikke har noe, for eksempel fordi det er «No Reply», registrer vi «Unknown».

Merk at strukturen til printRoute()-dataene og collectRtt()-dataene matcher hverandre. Dette skal vi bruke senere når vi skal slå sammen informasjonen i topologiene våre.

```python
def collectRtt(data):

    rtt = []

    for x in range(0, len(data["result"])):

        rtt.append([])

        for packet in data["result"][x]["result"]:

            try:
                rtt[x].append(packet["rtt"])
                continue
            except:
                pass

            if "late" in packet:
                rtt[x].append("Late")
            else:
                rtt[x].append("Unknown")

    return rtt
```

Figur 75: collectRtt() henter ut RTT-informasjonen

Vi kommer først tilbake til topologier og RTT i analyzeEachOutage().

Når vi finner starten på et utfall, lagrer vi nøkkelen til topologien knyttet til tracerouten som identifiserte utfallet. Deretter lagrer vi nøkkelen for hver mislykkede topologi. Vi lagrer også RTT-informasjonen til de mislykkede rutene som vi straks skal bruke.

```
#If the status is false, this is recorded as the start of an outage.
#All information needed is collected.

start = memory[uniquePair][index]["timeStamps"]
markers.append(memory[uniquePair][index]["topology"])
failrtts.append(getRtt(memory[uniquePair][index]["rtt"], cursor))
srcadr.append(memory[uniquePair][index]["from"])
```

Figur 76: Topologien til tracerouten som startet utfallet blir lagret

```
if (memory[uniquePair][index]["routeStatus"] == False):
    outageErrors.append(memory[uniquePair][index]["errors"])
    markers.append(memory[uniquePair][index]["topology"])
    failures = failures + 1
    frequency = (memory[uniquePair][index]["timeStamps"] - memory[uniquePair][index-1]["timeStamps"]) + frequency
    freqchange += 1
    failrtts.append(getRtt(memory[uniquePair][index]["rtt"], cursor))
    srcadr.append(memory[uniquePair][index]["from"])
    index = index + 1
```

Figur 77: Topologiene til hver traceroute som er med i utfallet blir lagret

Disse blir så lagt inn sammen med informasjonen om hvert utfall.

```
newmemory[r]["startTime"] = start
newmemory[r]["endTime"] = end
newmemory[r]["failures"] = failures
newmemory[r]["duration"] = duration
newmemory[r]["fromfalse"] = srcadr
newmemory[r]["lastReply"] = finalReply
newmemory[r]["errors"] = outageErrors
newmemory[r]["markers"] = markers
newmemory[r]["polling_time"] = int(frequency)
newmemory[r]["begins_in_other_hour"] = begins_in_other_hour
newmemory[r]["end_in_other_hour"] = end_in_other_hour
newmemory[r]["failrtts"] = failrtts
```

Figur 78: Topologiene til alle de mislykkede traceroutene i utfallet lagres

Vi kan kort se på getRtt() som henter ut RTT-informasjonen vi lagrer her og formaterer den.

```
def getRtt(marker, cursor):

    k = str(cursor.execute("SELECT rtt FROM rtt WHERE key = ?", (marker,),).fetchone())
    k = k[2:-3]
    k = json.loads(k)

    return k
```

Figur 79: Funksjonen getRtt()

Men vi må også ta med oss de vellykkede traceroutene. Merk at vi her faktisk henter topologiene ut igjen fra databasen. Siden er lagret som tekststrenger må de også formateres tilbake igjen. Grunnen til at vi henter rutene i sin helhet her er fordi vi straks skal bruke dem. Vi henter også ut RTT-informasjonen om de vellykkede rutene samtidig. Topologiene til de mislykkede traceroutene kommer vi tilbake til litt senere igjen.

```
if r+1 == outages:

    newmemory[0]["incomplete"] = incomplete
    newmemory[0]["ip_ver"] = memory[uniquePair][0]["ip_ver"]
    newmemory[0]["total_outage_duration"] = totalOutageDuration
    newmemory[0]["length"] = len(memory[uniquePair])

    #We also need to get all the probe_ids before we delete the old memory
    #And get the successful route topologies from the SQLlite database with their RTT data
    #Additionally we want to get the total dup/late count

    newmemory[0]["from"] = []
    newmemory[0]["whole_routes"] = []
    newmemory[0]["total_dups"] = 0
    newmemory[0]["total_lates"] = 0
    newmemory[0]["rtt"] = []


    for x in memory[uniquePair]:

        newmemory[0]["from"].append(memory[uniquePair][x]["from"])
        newmemory[0]["total_dups"] = newmemory[0]["total_dups"] + memory[uniquePair][x]["dups"]
        newmemory[0]["total_lates"] = newmemory[0]["total_lates"] + memory[uniquePair][x]["lates"]

        if memory[uniquePair][x]["routeStatus"] == True:
            k = str(cursor.execute("SELECT topology FROM topologies WHERE key = ?", (memory[uniquePair][x]["topology"],),).fetchone())
            k = k[2:-3]
            k = json.loads(k)
            newmemory[0]["whole_routes"].append(k)
            newmemory[0]["rtt"].append(getRtt(memory[uniquePair][x]["rtt"], cursor))
```

Figur 80: Topologiene fra traceroutene som nådde frem hentes ut fra databasen

Disse topologiene skriver vi så ut sammen med ruteoppsummeringene.

```
entry = {
"pair_id": pairname,
"event_type": "summary",
"dst_addr": pair[1],
"src_addr": srcadr,
"prb_id": pair[0],
"start_time": memory[x][0]["startTime"],
"last_response": memory[x][len(memory[x])-1]["endTime"],
"traceroutes_recorded": memory[x][0]["length"],
"failed_traceroutes": str(round((failed/memory[x][0]["length"]) * 100)) + "%",
"outage_count": len(memory[x]),
"outage_at_log_start": memory[x][0]["begins_in_other_hour"],
"outage_at_log_end": memory[x][len(memory[x])-1]["end_in_other_hour"],
"incomplete_outages": memory[x][0]["incomplete"],
"total_dupes": memory[x][0]["total_dups"],
"total_late": memory[x][0]["total_lates"],
"total_outage_errors": countErrorTypes(allerrors, True),
"total_outage_duration_in_seconds": memory[x][0]["total_outage_duration"],
"max_hop": maxm,
"min_hop": minm,
"topology": top
}
```

Figur 81: Topologiene blir gjort klar til å skrives ut som JSON

Vi bruker analyzeTopology() til å hente ut lengden på den korteste og lengste ruten. I tillegg lager funksjonen de pene topologiene vi så på i kapittel 5.4.2.3 med prosentangivelser.

Dette er en av de vanskeligste funksjonene i programmet, så vi tar den i deler.

```
#First finds the max and min length of the routes

listlen = [len(i) for i in topology]
maxm = max(listlen)
minm = min(listlen)

#Makes a post for each possible hop
#based on topology and RTT data

charts = {}
pos = -1

for x in topology:
    pos = pos + 1
    hop = 0
    for y in x:
        counter = 0
        hops = []
        for z in y:
            item = {}
            item["hop"] = hop
            item["ip"] = z
            item["rtt"] = rtt[pos][hop][counter]
            counter += 1
            hops.append(item)
        if hop not in charts:
            charts[hop] = []
        charts[hop].append(hops)
        hop += 1
```

Figur 82: Begynnelsen av analyzeTopolog()

Først regner vi ut den korteste og lengste ruten. Det neste vi gjør er at vi jobber oss gjennom rutene og lager en post hvert mulige hopp med RTT-informasjonen om hoppet. Her utnytter vi at RTT-informasjonen og ruteinformasjonen har en parallell struktur.

Det neste vi gjør er at vi slår sammen poster på samme hopp til samme destinasjon. Vi regner også ut hvor stor andel som er på hvert hopp, maks RTT og alt annet vi trenger.

```
newchart = []
finalchart = []

for x in charts:
    hop = {}
    hopcount = 0
    totcount = 0
    for y in charts[x]:
        for z in y:
            cur = z["ip"] + "hop" + str(z["hop"])
            if cur not in hop:
                hop[cur] = {}
                hop[cur]["hop"] = z["hop"]
                hop[cur]["ip"] = z["ip"]
                hop[cur]["rtt"] = [z["rtt"]]
                hop[cur]["rtt_count"] = 1
            else:
                hop[cur]["rtt"].append(z["rtt"])
                hop[cur]["rtt_count"] += 1
            totcount += 1
    for z in hop:
        hop[z]["portion"] = str(round(hop[z]["rtt_count"]/totcount * 100)) + "%"
        hop[z]["maxrtt"] = findMaxRtt(hop[z]["rtt"])
        hop[z]["minrtt"] = findMinRtt(hop[z]["rtt"])
        hop[z]["rtt_std_dv"] = top_std_dev(hop[z]["rtt"])

        if hop[z]["ip"] != "*":
            del hop[z]["rtt"]
            finalchart.append(hop[z])

    hopcount += 1
    newchart.append(hop)
```

Figur 83: Finner antall identiske steg på hvert hopp

Resultatet følger, som diskutert i kapittel 5.4.2.3, det følgende mønsteret.

```
[…],
{
```

```
  "hop": 6,
  "rtt_std_dv": 0.16,
  "ip": "2a01:7e8:1:200::9a6",
  "maxrtt": 22.315,
  "minrtt": 21.927,
  "portion": "19%",
  "rtt_count": 4
},
{
  "rtt_std_dv": 0.21,
  "hop": 6,
  "ip": "::ffff:194.182.97.108",
  "maxrtt": 24.425,
  "portion": "43%",
  "minrtt": 23.674,
  "rtt_count": 9
},
[…]
```


Tabell 19: Eksempel på en ferdig analysert topologi

Nå som vi er ferdige med preparePairData(), kjører vi exchangeTopology().

```
#Prepares final data for writing to JSON
memory = cleanUp(memory, cursor) #Creates outage reports and deletes unused data
pairData = preparePairData(memory, topology, cursor) #Creates the JSON files for each unique pair report
memory = exchangeTopology(memory, cursor) #Deletes successful route topologies and gets unsuccessful from SQLlite database
outageData = prepareOutageData(memory) #Creates the JSON reports for each outage
uniqueProbeCount = len(uniqueProbeDictionary)
totalData = prepareTotalData(numberOfTraceroutes, successfulTraceroutes,
failedTraceroutes, uniqueProbeCount, lowestTimestamp, highestTimestamp)
```

Figur 84: Nå trenger vi de andre topologiene

Det exchangeTopology() gjør er at vi kvitter oss med alle de vellykkede topologiene som vi la inn i ruteoppsummeringene. Disse tar nemlig veldig mye plass og vi er nødt til å hente de mislykkede topologiene nå for å kunne skrive ut utfallsoppsummeringene.

```
def exchangeTopology(memory, cursor):

    for x in memory:
        del memory[x][0]["whole_routes"]

    for x in memory:
        for y in memory[x]:
            memory[x][y]["topology"] = []
            for z in memory[x][y]["markers"]:
                k = str(cursor.execute("SELECT topology FROM topologies WHERE key = ?", (z,),).fetchone())
                k = k[2:-3]
                k = json.loads(k)
                memory[x][y]["topology"].append(k)

    return memory
```

Figur 85: Topologiene til de mislykkede traceroutene hentes ut fra databasen

Dette er en ganske enkel funksjon som bare sletter de mislykkede topologiene. Deretter lastes rutene til de mislykkede traceroutene inn ganske likt som de vellykkede ble lastet inn.

Vi får rutene analyser på samme måte som sist.

```
maxm, minm, top = analyzeTopology(memory[y][x]["topology"], memory[y][x]["failrtts"])
```

Figur 86: Gjør de mislykkede topologiene klare for JSON

Deretter legger vi også disse klar til å skrives ut som JSON.

```
outage = {
    "pair_id": pairname,
    "event_type": "outage",
    "order": x+1,
    "dst_addr": pair[1],
    "src_addr": srcadr,
    "prb_id": pair[0],
    "start_time": memory[y][x]["startTime"],
    "last_response": memory[y][x]["endTime"],
    "outage_at_log_start": memory[y][x]["begins_in_other_hour"],
    "outage_at_log_end": memory[y][x]["end_in_other_hour"],
    "failure_count": memory[y][x]["failures"],
    "polling_frequency": round(memory[y][x]["polling_time"]),
    "duration": memory[y][x]["duration"],
    "last_reply_from": memory[y][x]["lastReply"],
    "AS_number": AS_number,
    "AS_owner": AS_name,
    "location": location,
    "coordinates": str(latitude) + "," + str(longitude),
    "outage_errors": countErrorTypes(memory[y][x]["errors"], False),
    "error_type": classifyOutage(memory[y][x]["errors"]),
    "max_hop": maxm,
    "min_hop": minm,
    "topology": top
}
```

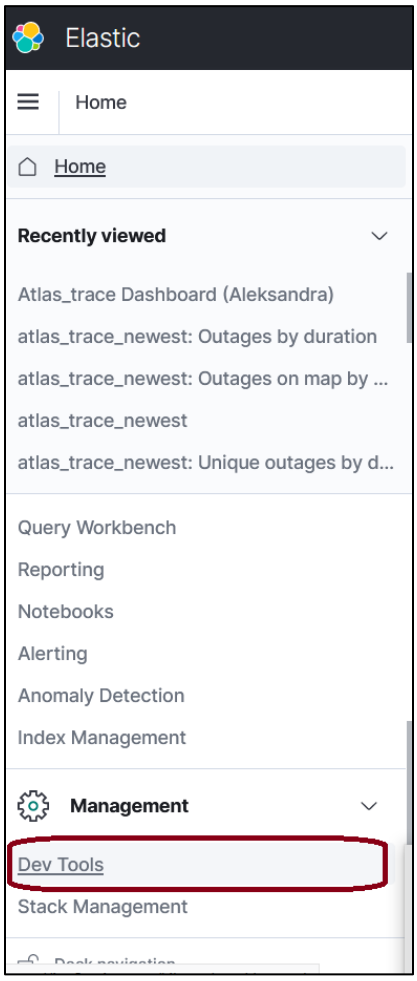
Figur 87: De mislykkede rutene gjøres klar til JSON

6. Andre del: Kibana og Elasticsearch

6.1. Lage «index» og «index pattern»

For at Elasticsearch skal kunne lese dataene fra loggene riktig, er vi nødt til å legge inn en indeks. Det vanlige er å gjøre dette med curl, men jeg liker veldig godt konsollen som er bygget inn Kibana, så da kommer jeg til å opprette indeksen med denne i stedet. For å åpne konsollen, åpner man navigasjonen på venstre og finner «Dev Tools» under «Management».

Merk at en del av disse bildene er utdatert og ikke passer med innleverte versjon av systemet.



Figur 88: Finner "Dev Tools" i Kibana

For å lage indeksen kjører jeg den følgende koden i konsollen.

```
PUT /atlas_trace
{
  "mappings": {
    "properties": {
      "total_traceroutes": { "type": "long" },
      "start_time": { "type": "date", "format": "epoch_second" },
      "end_time": { "type": "date", "format": "epoch_second" },
      "last_response": { "type": "date", "format": "epoch_second" },
      "total_succeeded": { "type": "long" },
      "total_failed": { "type": "long" },
      "failed_at_probe": { "type" : "long" },
      "total_probes": { "type": "long" },
      "version": { "type": "text"},
      "pair_id": { "type" : "text", "fields" : { "raw": { "type" : "keyword" }}},
      "event_type": { "type" : "text", "fields" : { "raw": { "type" : "keyword" }}},
      "dst_addr": { "type" : "text", "fields" : { "raw": { "type" : "keyword" }}},
      "src_addr": { "type" : "text", "fields" : { "raw": { "type" : "keyword" }}},
      "prb_id": { "type" : "text"},
      "traceroutes_recorded": { "type": "long"},
      "failed_traceroutes": { "type": "text"},
      "outage_count": { "type" : "long" },
      "outage_at_log_start": { "type" : "boolean" },
      "outage_at_log_end": { "type" : "boolean" },
      "incomplete_outages": { "type" : "long" },
      "total_dupes": { "type": "long" },
      "total_late":{ "type": "long" },
      "total_outage_duration_in_seconds": { "type" : "long" },
      "max_hop": { "type": "long" },
      "min_hop": { "type": "long" },
      "topology": { "enabled": false},
      "order": { "type" : "long" },
      "failure_count": { "type" : "long" },
      "polling_frequency": { "type": "long" },
      "duration": { "type" : "long" },
      "last_reply_from": { "type" : "text", "fields" : { "raw": { "type" : "keyword" }}},
      "AS_number": { "type" : "text", "fields" : { "raw": { "type" : "keyword" }}},
      "AS_owner": { "type" : "text", "fields" : { "raw": { "type" : "keyword" }}},
      "location": { "type" : "text", "fields" : { "raw": { "type" : "keyword" }}},
      "coordinates" : { "type" : "geo_point" },
      "error_type": { "type" : "text" }
    }
  }
}
```

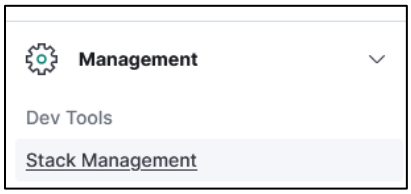
Figur 89: Oppretter "index" i Elasticsearch

Det meste er ganske selvsagt. Det er mye «type: text» for tekststrenger som AS_owner osv. og «type: long» for nummer som for eksempel «max_path» og «min_path».

Unntaket er topology som bare er lagret med «enabled: false». Det dette gjør er at det ber Elasticsearch la være å forsøke å tolke dataene som krever ganske mye arbeid å få til å fungere. Foreløpig er tanken at disse skal bare skal leses av mennesker og eventuelt lastes inn i et annet program. Det er ikke praktisk å gjøre søke på tvers av ulike topologier.

Vi kan også merke oss at datoene er lagret som «dates» og da spesifikt «epoch_seconds» som er det samme som Unix-tid i millsekunder. I tillegg har jeg lagt «raw: type: keyboard» inn på de fleste av tekststrengene siden dette gir flere muligheter til å søke på disse verdiene.

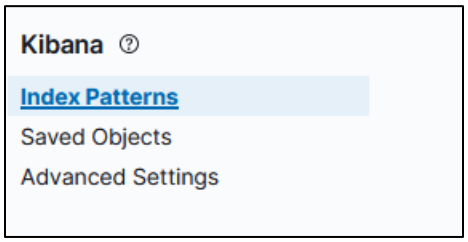
Når vi nå har opprettet en index for Elasticsearch, trenger vi videre et «index pattern». Dette styrer hvordan Kibana henter inn informasjon fra Elasticsearchs indekser. For å lage en «index pattern» åpner vi igjen navigasjonsmenyen og velger «Stack Management».



Figur 90: Åpner "Stack Management" i Kibana

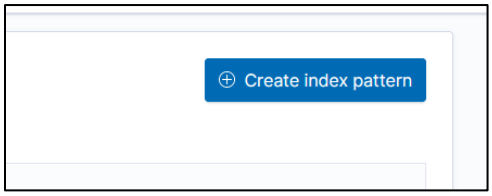
Dette ligger like under der vi fant konsollen vi brukte til å lage indeksen.

Her trykker vi oss bare videre inn på «Index Pattern».



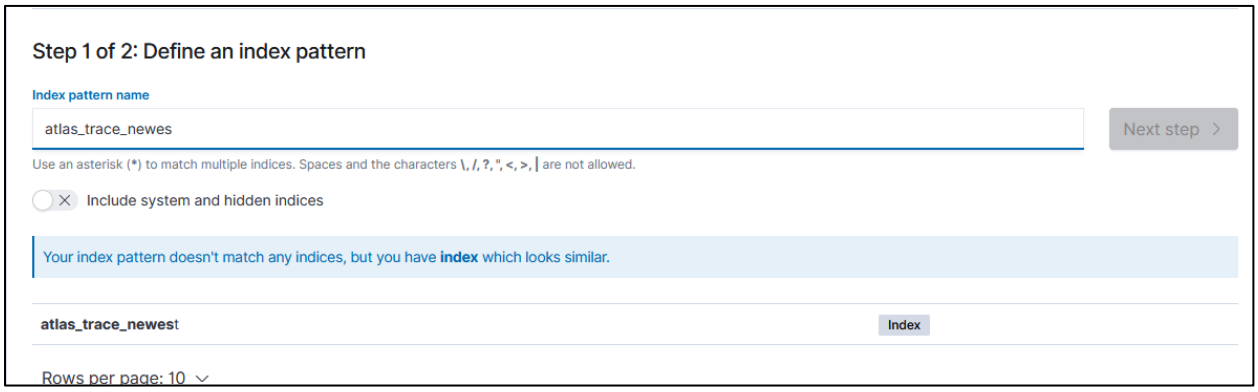
Figur 91: Finner "Index Pattern"

Vi velger «Create index pattern» på siden som kommer opp.



Figur 92: Knappen "Create index pattern"

Vi søker på indeksen vi nettopp laget og legger denne inn.

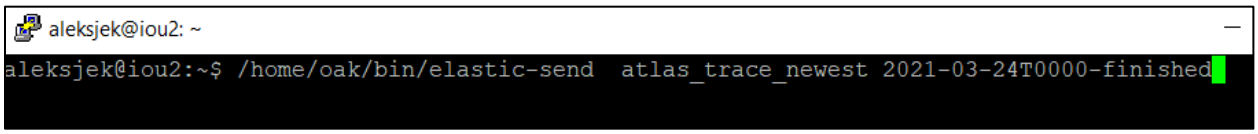


Figur 93: Gir "index pattern" et navn

Vi velger «start_time» som tidsfeltet som er ganske viktig for å kunne finne frem til loggene etterpå siden de blir ordnet etter tid. Alle loggtypene programmet lager har «start_time».

Figur 94: Velger standard dato fra loggene

Nå som alt er klart kan vi laste inn loggene vi ønsker i Kibana/Elasticsearch.

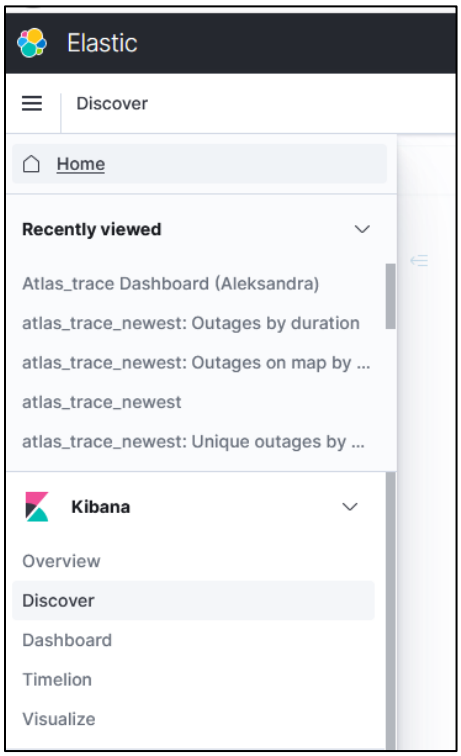


Figur 95: Legger loggene inn i index

Jeg skal komme nærmere tilbake til dette i kapittel 7 som handler om Linux-delen av systemet.

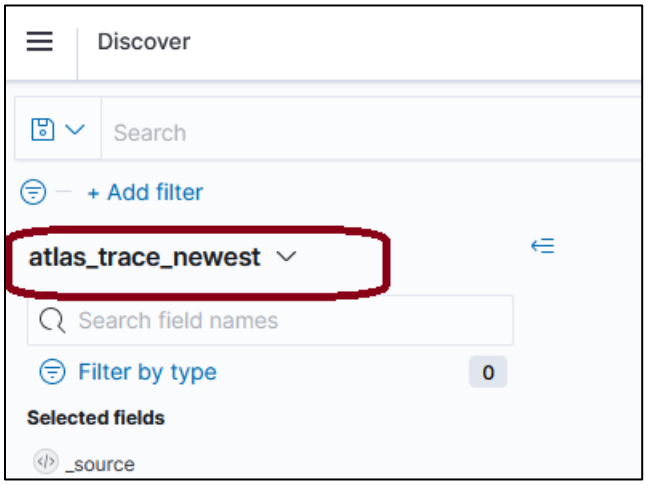
6.2. Bruke «Discover»

Nå kan vi gå i navigasjonsmenyen igjen og velge «Discover».



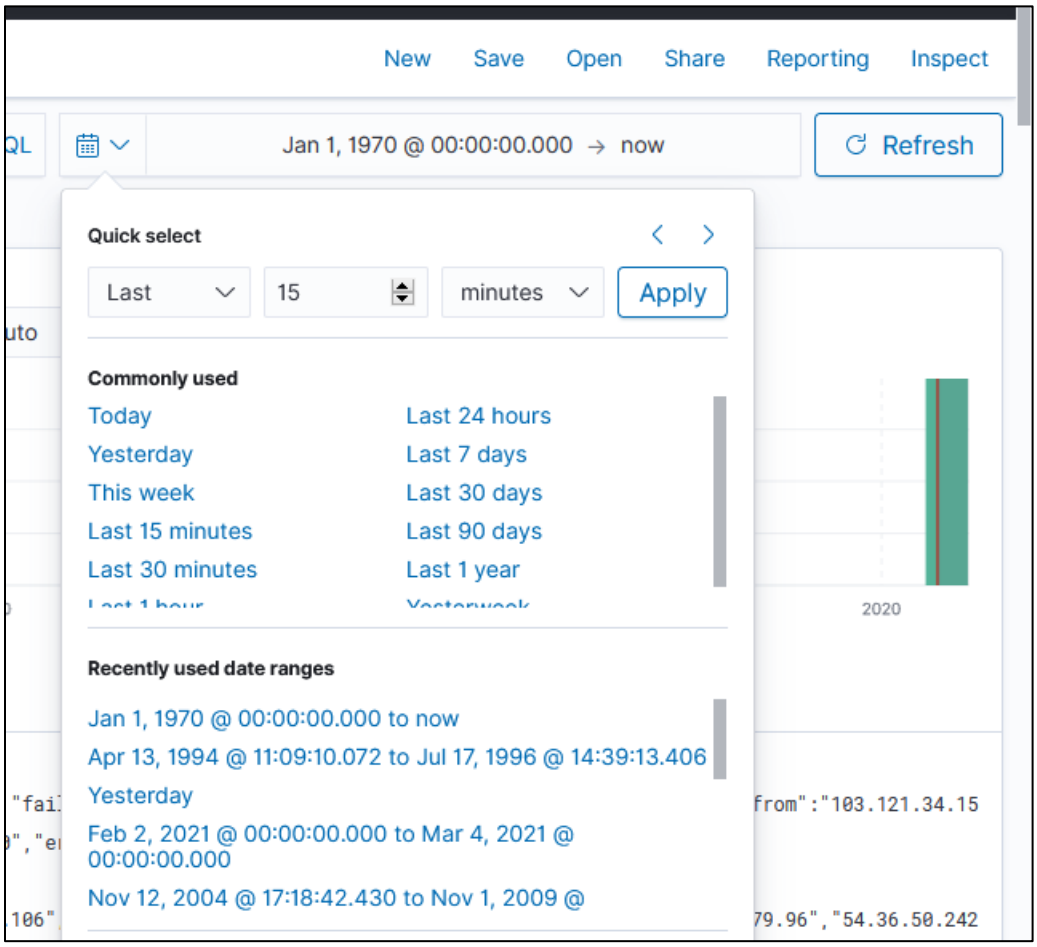
Figur 96: Finner "Discover" i Kibana

Her velger vi «index pattern» som vi nettopp opprettet.



Figur 97: Finner vårt "index pattern" i "Discover"

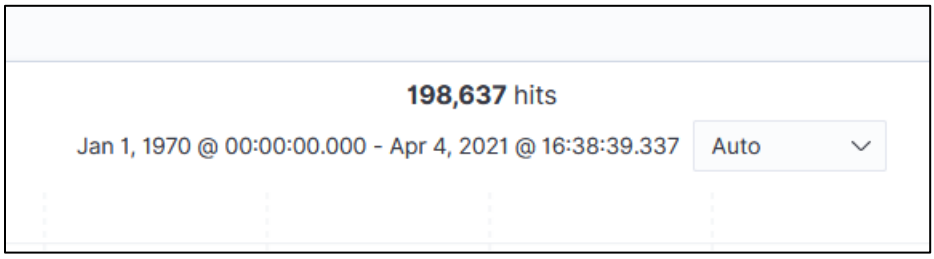
Deretter må vi velge tidsrommet vi ønsker å se logger fra.



Figur 98: Velger riktig tidsrom i "Discover"

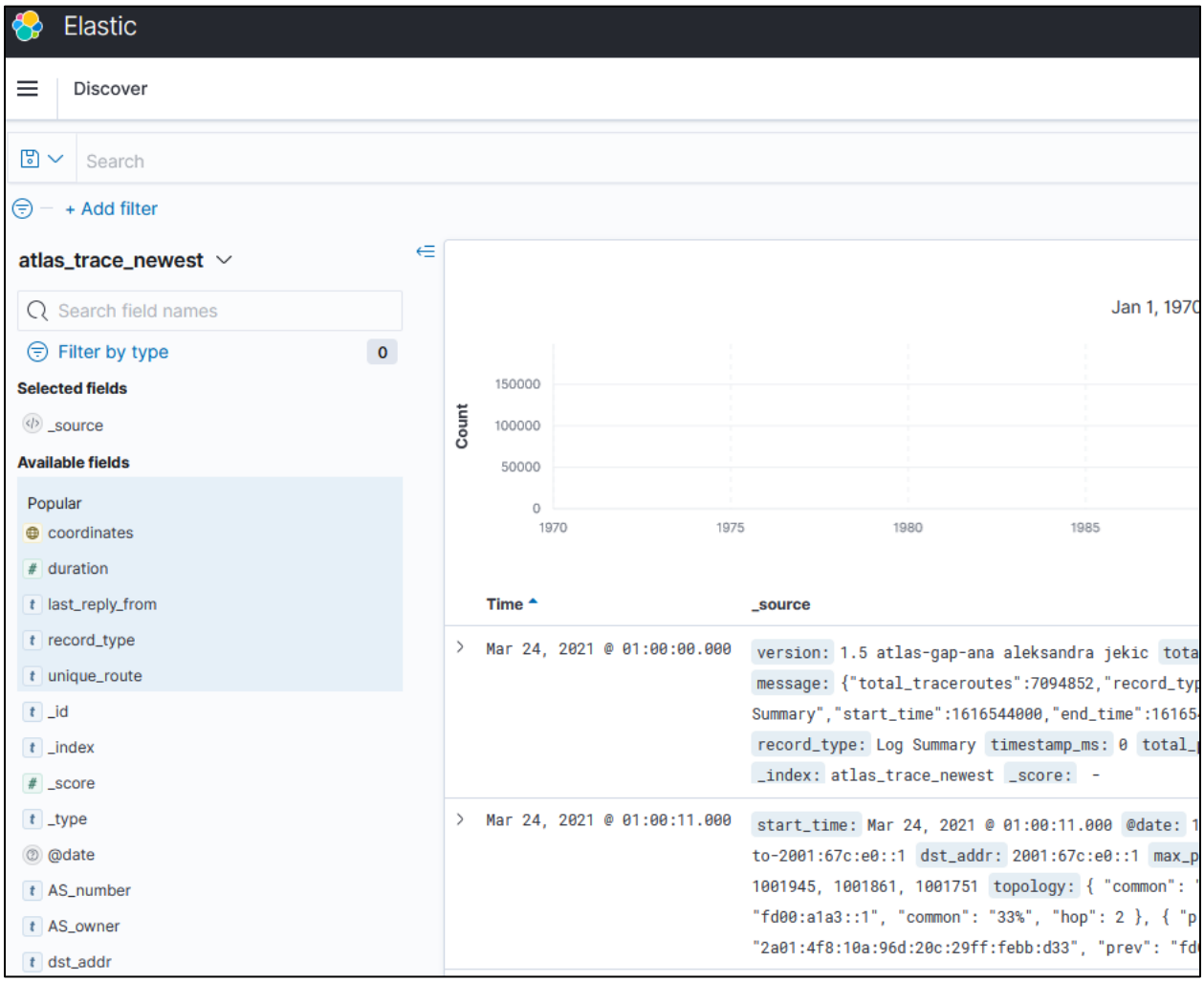
Her har jeg valgt tidsrommet fra 1. januar 1970 midnatt frem til nå.

Da får vi opp de nesten 200 000 loggene jeg har lagt inn fra 24. mars 2021.



Figur 99: Antallet logger lagt inn

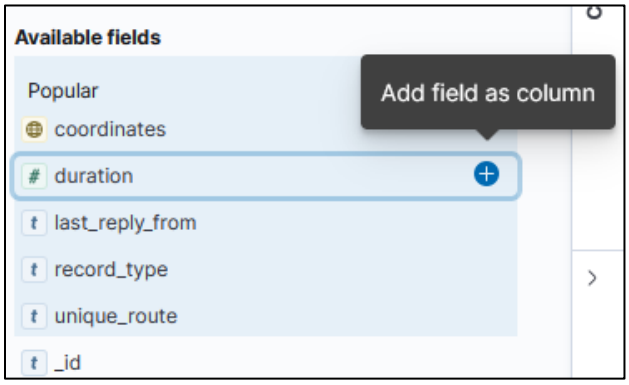
Nå er det veldig lett å utforske loggene ved hjelp av Kibana.



Figur 100: Kan utforske alle loggene i "Discover"

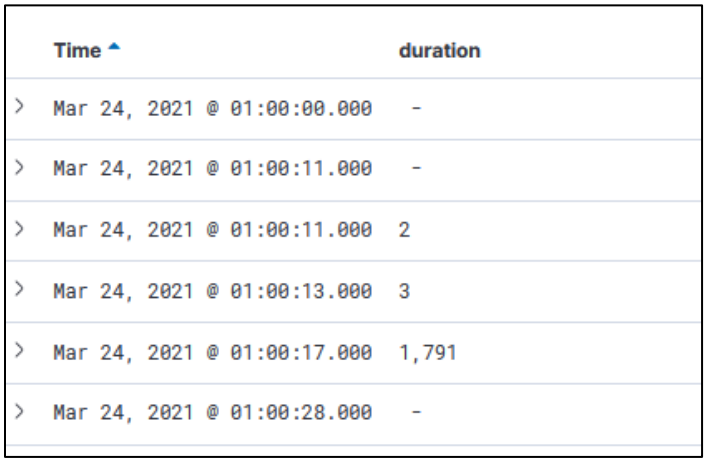
På den ene siden har man de forskjellige feltene og på den andre vises loggene.

For å gi en demonstrasjon av hvordan man kan søke, kan jeg trykke på «duration».



Figur 101: Velger "duration" i "Discover"

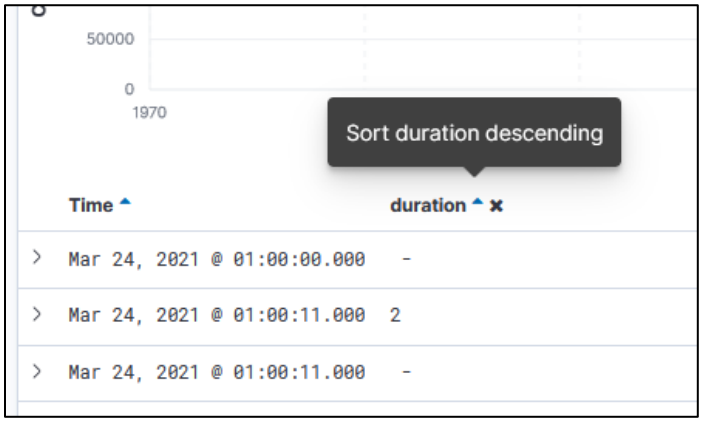
Da kommer alle loggene opp igjen, men med duration tydelig vist.



Time	duration
Mar 24, 2021 @ 01:00:00.000	-
Mar 24, 2021 @ 01:00:11.000	-
Mar 24, 2021 @ 01:00:11.000	2
Mar 24, 2021 @ 01:00:13.000	3
Mar 24, 2021 @ 01:00:17.000	1,791
Mar 24, 2021 @ 01:00:28.000	-

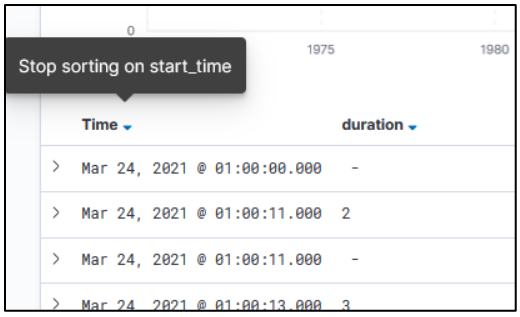
Figur 102: Logger listet med "duration"

Nå kan jeg for eksempel trykke «sort duration descending».



Figur 103: Sortering etter "duration"

I tillegg kan jeg trykke «stop sorting on start_time».



Figur 104: Loggene sortert etter tid

Da kommer alle loggene sortert etter lengde på utbruddene.

Time	duration
Mar 24, 2021 @ 05:14:50.000	2,707
Mar 24, 2021 @ 15:14:50.000	2,705
Mar 24, 2021 @ 19:14:54.000	2,704
Mar 24, 2021 @ 19:14:54.000	2,704

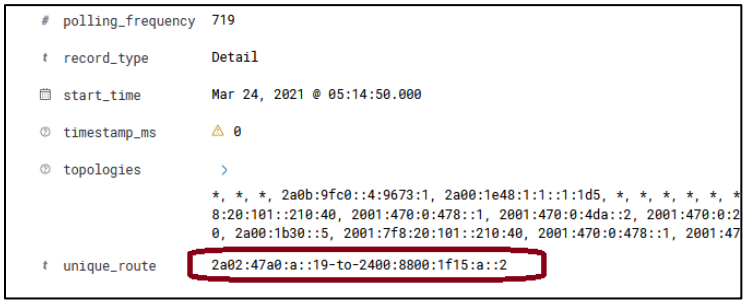
Figur 105: Loggene sortert etter lengde på utfallene

Hvis jeg vil kan jeg åpne det lengste bruddet og studere dette nærmere.

Figur 106: Detaljert visning av en logg

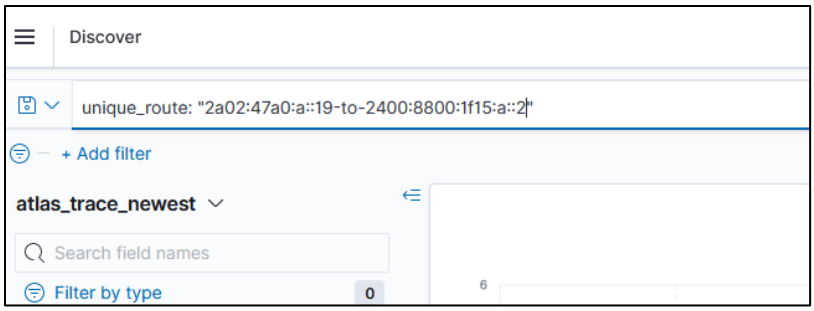
Her ser man for eksempel at det stammer fra Russland og feilene knyttet til bruddet.

Hvis jeg syntes dette var et spennende utfall, kan jeg prøve å finne ut mer om det. Hvis jeg blar lengre ned finner jeg det unike rutenavnet som jeg kan bruke til å finne andre logger fra ruten.



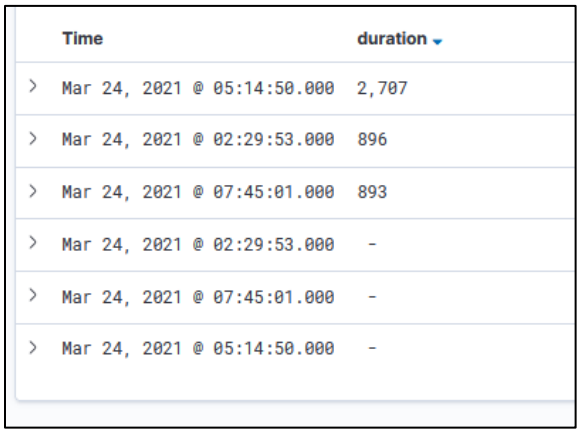
Figur 107: Den unike ruten knyttet til loggen

Nå skriver jeg unique_route: «2a02:47a0:a::19-to-2400:8800:1f15:a::2» inn i søkefeltet.



Figur 108: Søke på rutenavnet

Da vises alle loggene som sier noe om den samme unike ruten.



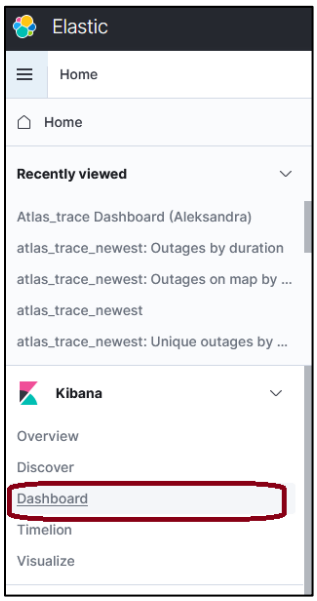
Time	duration
Mar 24, 2021 @ 05:14:50.000	2,707
Mar 24, 2021 @ 02:29:53.000	896
Mar 24, 2021 @ 07:45:01.000	893
Mar 24, 2021 @ 02:29:53.000	-
Mar 24, 2021 @ 07:45:01.000	-
Mar 24, 2021 @ 05:14:50.000	-

Figur 109: Alle loggene knyttet til denne ruten

Merk at de tre nederste ikke har «duration» siden det er «record type: Summary». Jeg kommer nærmere tilbake til «Discovery» i kapittel 8, dette var bare en grunnleggende innføring.

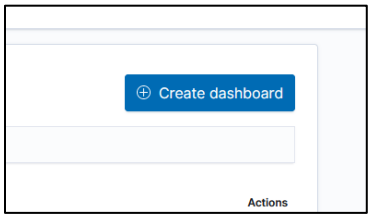
6.3. Bruke «Visualize» og «Dashboard»

Videre kan man i Kibana lage visualiseringer basert på dataene. Vi kan starte med å lage et Dashboard hvor vi kan samle flere visualiseringer på en oversiktlig måte.



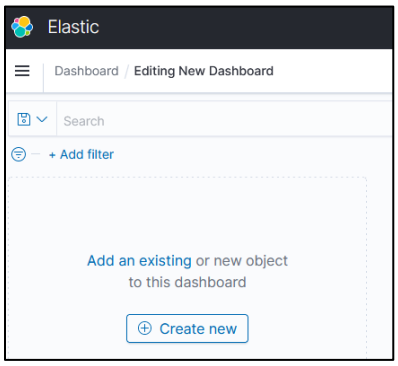
Figur 110: Finne "Dashboard" i Kibana

Finner «Dashboard» i navigasjonsmenyen og velger «Create New Dashboard».



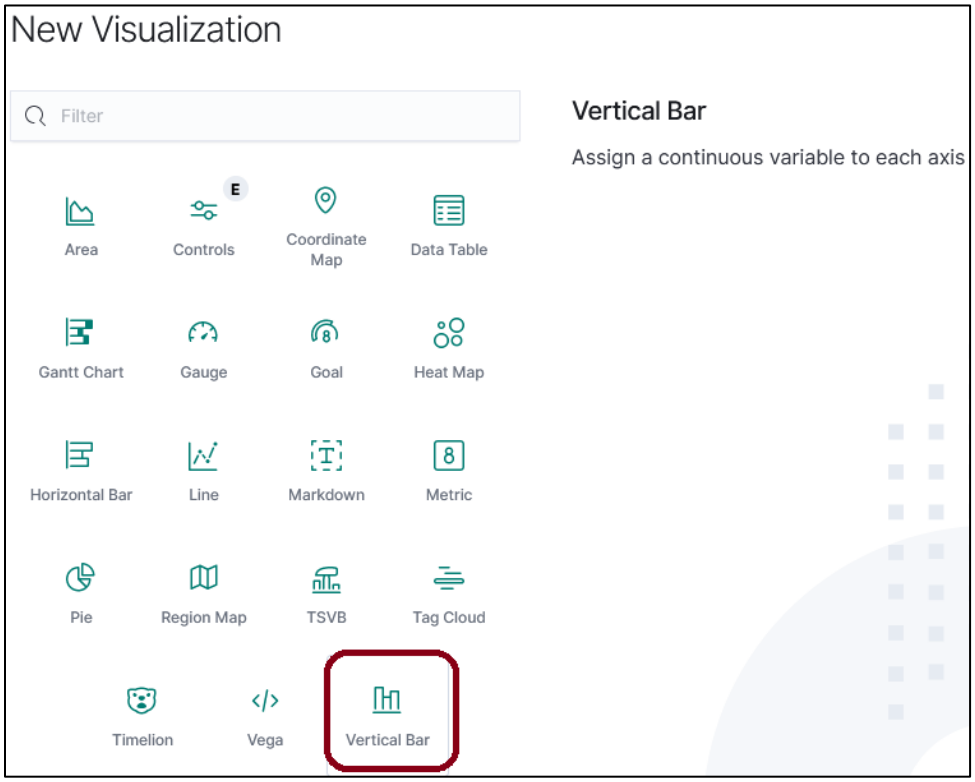
Figur 111: "Create dashboard"-knappen

Da får man opp en ganske tom flate.



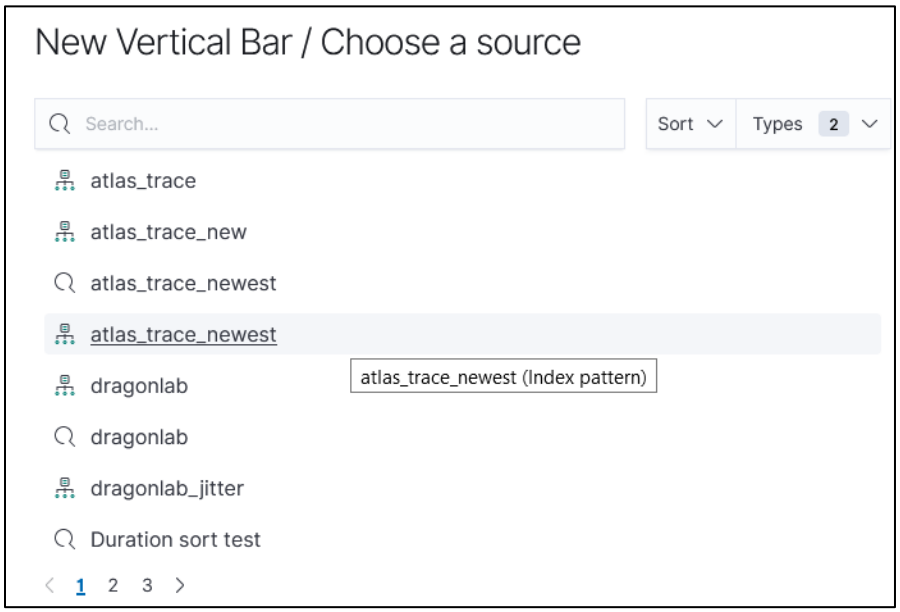
Figur 112: Tomt "Dashboard"

Vi kan begynne enkelt med å lage et søylediagram som viser de lengste bruddene.



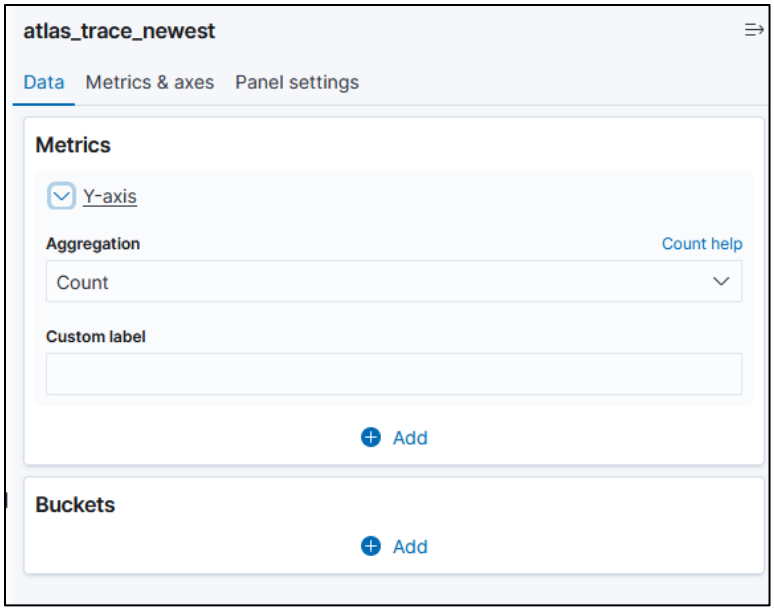
Figur 113: Opprette et søylediagram

Vi kan ta utgangspunkt i «index pattern» når vi skal lage diagrammet.



Figur 114: Valg av index pattern til visualisering

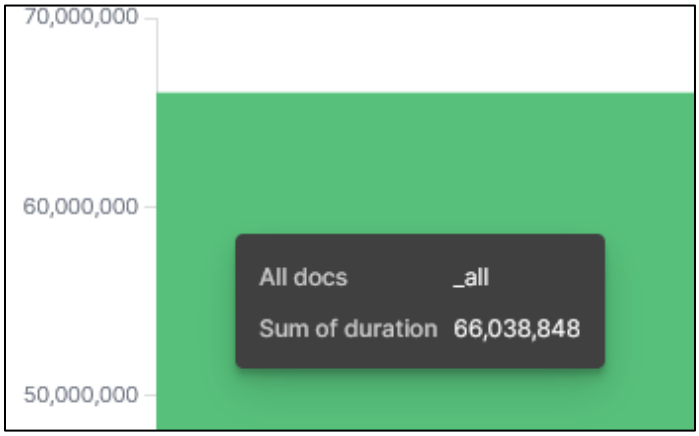
Til venstre må vi nå bestemme hva slags data søylediagrammet skal vise.



Figur 115: Y-aksen til søylediagrammet

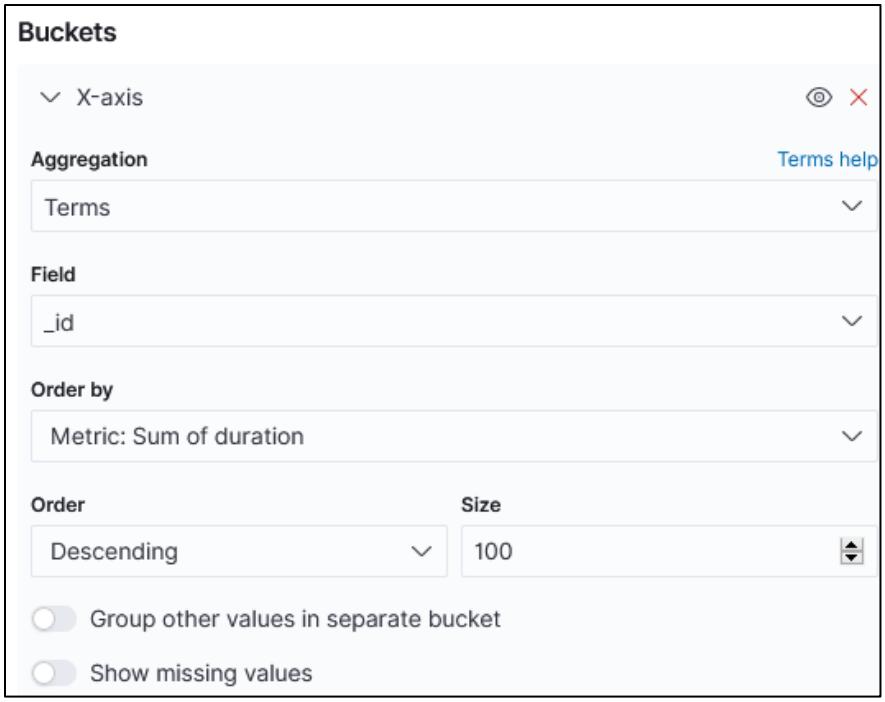
Det kan være litt vanskelig å få Kibana til å vise det man vil siden Kibana elsker å aggregere ting. Vi kan starte med å få Kibana til å vise summen av varighet på y-aksen.

Nå viser Kibana selvfølgelig varigheten fra alt vi har lagret til sammen.



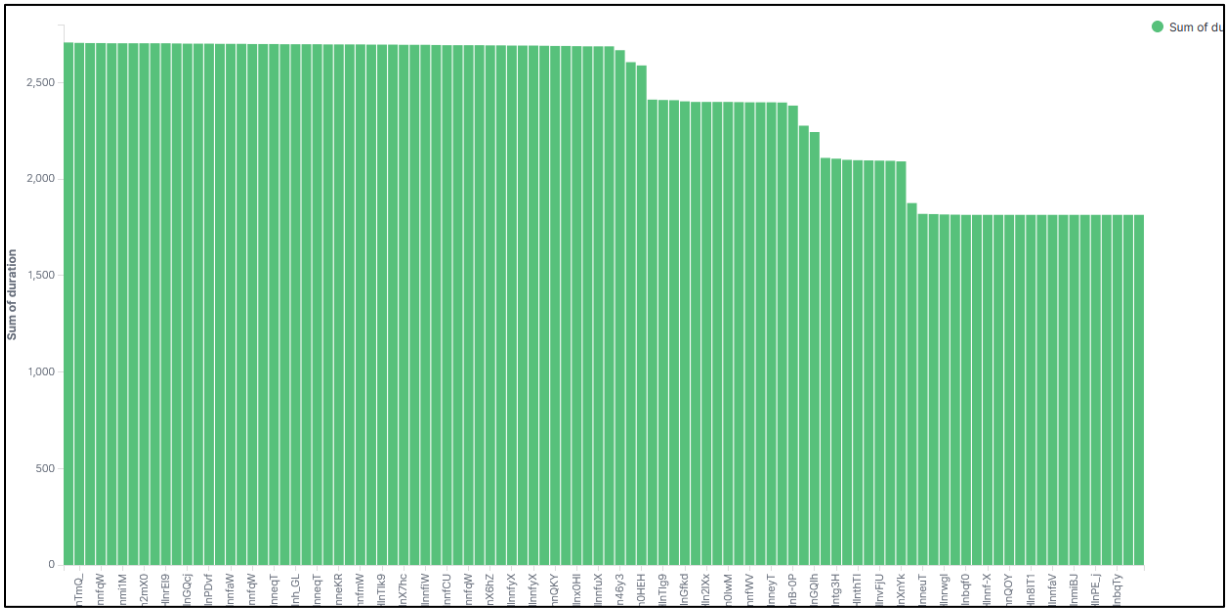
Figur 116: Søylediagram uten x-akse

Det vi må gjøre nå er å legge inn en x-akse. Vi er heldige siden hver loggføring har en unik id.



Figur 117: Legger inn x-aksen

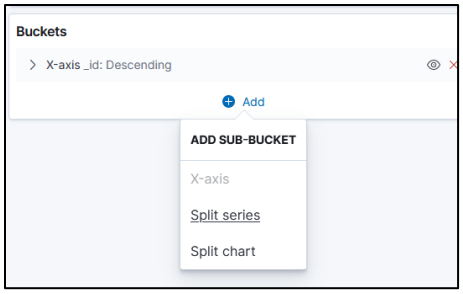
Nå begynner det også å ligne på noe mer.



Figur 118: Søylediagram med både x- og y-akse

Men jeg vil gjerne at diagrammet skal vise navnet på ruten til hvert utfall.

Det jeg da kan gjøre er å lage en «split series» på det unike rutenummeret.

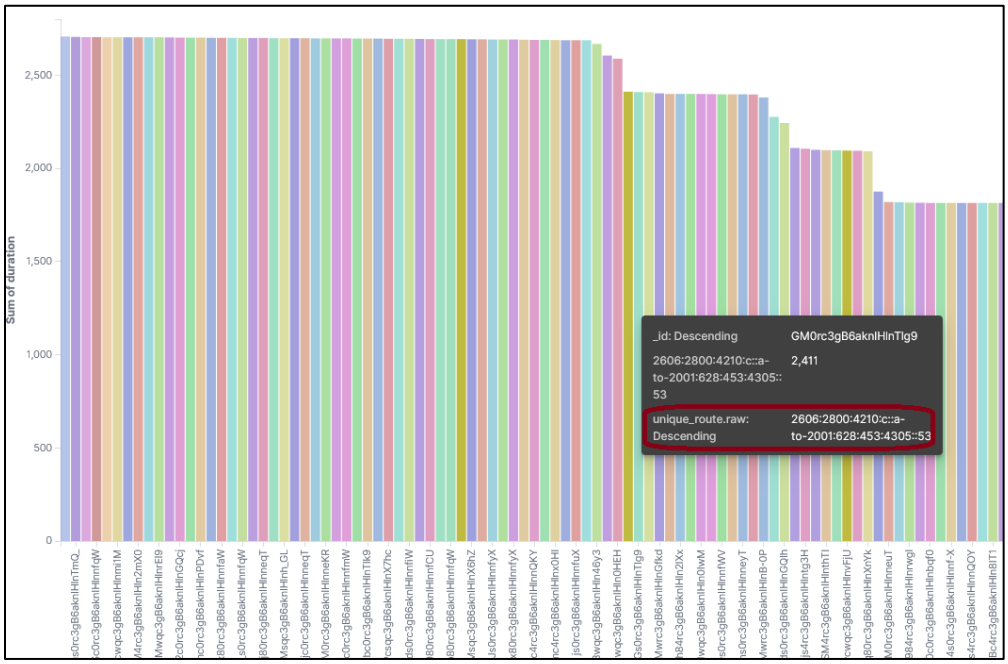


Figur 119: Legger inn "split series"

Deretter velger vi «unique_route» som termen det skal dele på.

Figur 120: Splitter på unike ruter

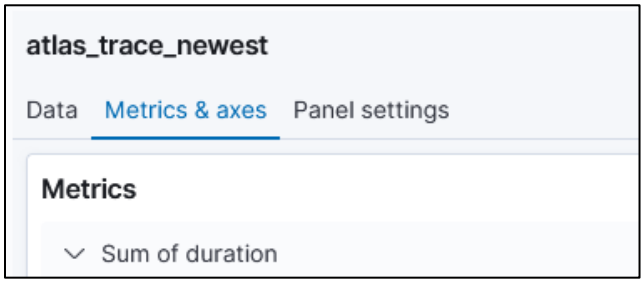
Nå kan vi også se identifikasjonene til rutene på alle utbruddene.



Figur 121: Søylediagram med rutenavn

Dette er egentlig litt juks siden «split series» er laget for å dele opp aggregerte grupper. For eksempel hvis det hadde vært flere logger som delte samme id ville de blitt delt videre opp i unike ruter. Men her bruker jeg det bare for å få navnet på rutene inn i grafen.

En siste ting vi kan gjøre er å gå på «Metrics & axes».



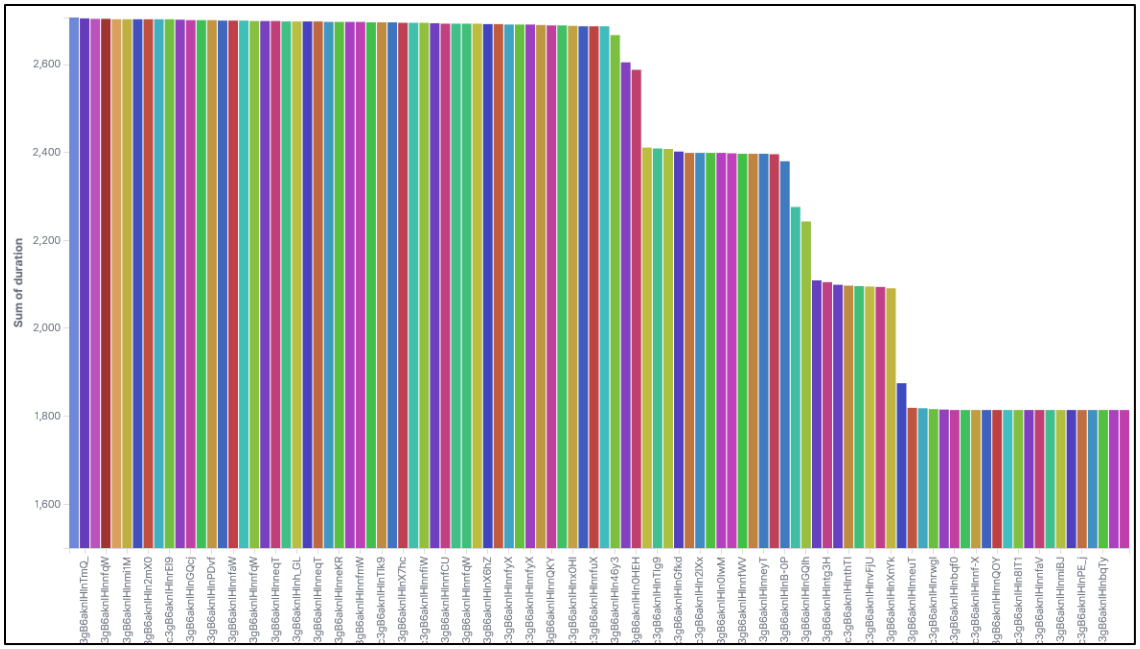
Figur 122: Finne "Metrics & axes"

Her kan vi velge «Custom Extents».

Figur 123: Finner "custom extents"

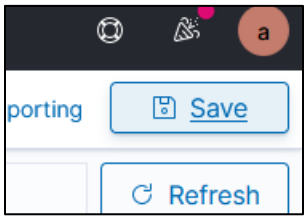
Hvis vi da legger inn 1500 som minimum for y-aksen får vi en litt mer oversiktlig graf.

Figur 124: Setter minimum til 1500



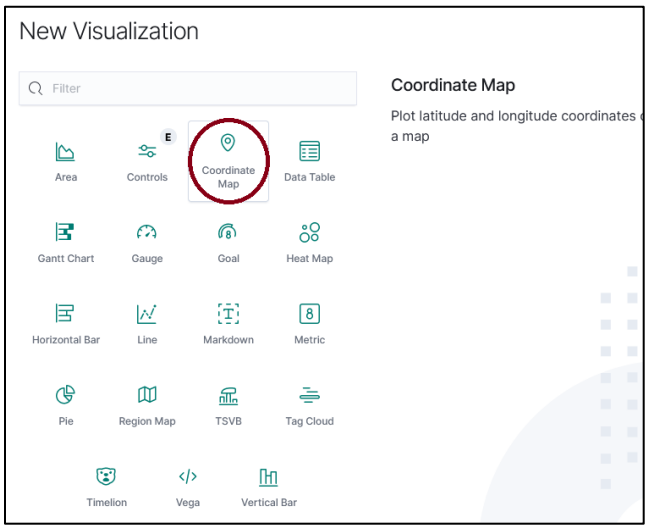
Figur 125: Ferdig søylediagram

Nå må vi huske å lagre visualiseringen oppe i høyre hjørne før vi går videre.



Figur 126: Lagre visualisering

Noe annet vi for eksempel kan gjøre er å lage et kart. Det er mye enklere å få til.

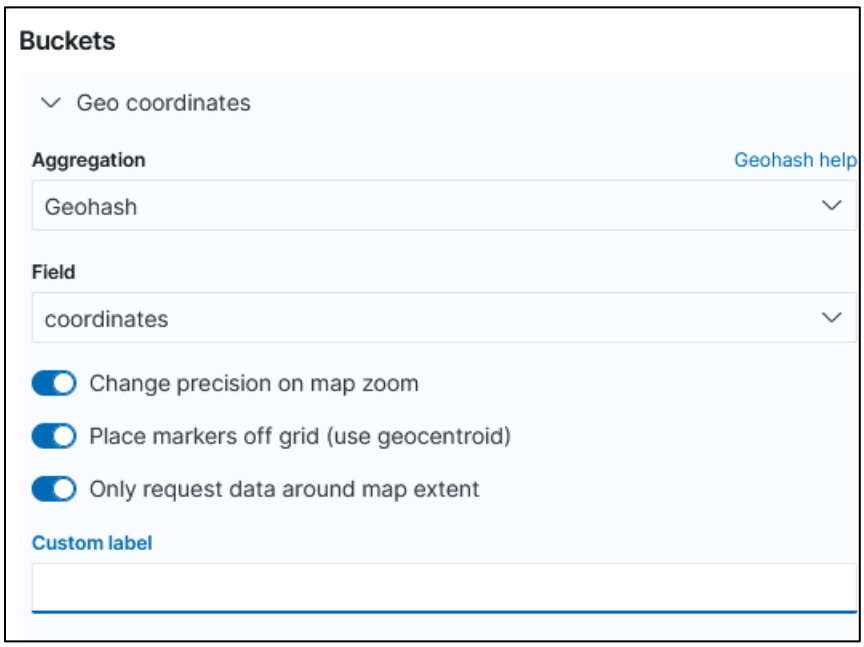


Figur 127: Lage "Coordinate map"

Her kan vi for eksempel legge inn summen av varighet som det vi er interessert i.

Figur 128: Finne summen av varighet

Og deretter velger vi bare å bruke koordinatene fra loggene.



Figur 129: Valg av geokoordinater

Da kan man se på kartet hvor bruddene til sammen er lengst.

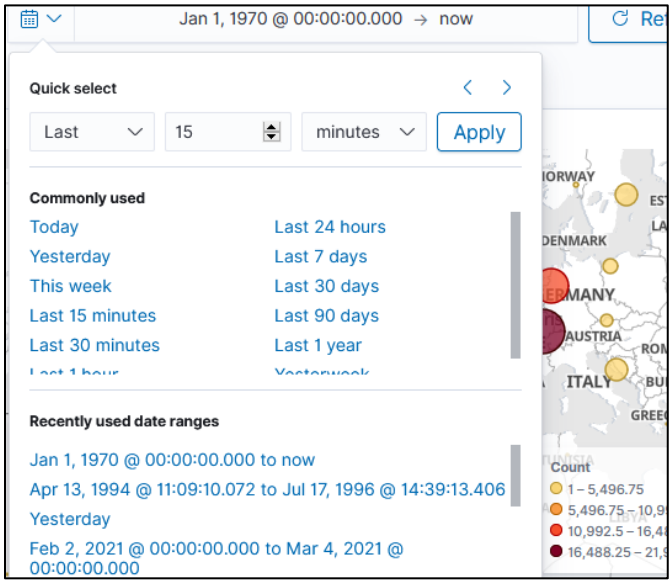
Figur 130: Ferdig koordinatkart

Hvis vi nå lagrer begge disse så kan vi se begge deler på dashboardet.

Figur 131: "Dashboard" med visualiseringer

Visualiseringene blir automatisk oppdatert etter hvert som det kommer nye logger inn.

Man kan også styre hvilke tidspunkter man skal visualisere informasjon fra.



Figur 132: Styre tidspunktene som visualiseres

7. Tredje del: Oppsett i Linux

7.1. Automatiske skript

Denne delen skal handle om hvordan programmet skal settes opp på Linux-tjeneren og hvordan det er knyttet sammen med Kibana/Elasticsearch. Her ser man mappen min på tjeneren.

```
aleksjek@iou2:~$ ls
atlas-gap-ana  atlas-gap-ana.py  log  oldtests
aleksjek@iou2:~$
```

Figur 133: Rotmappen på Linux-tjeneren

Selve programmet er atlas-gap-ana.py som ble gjennomgått i kapittel 5. «atlas-gap-ana» er et eget skript som får programmet til å analysere 24 timer. Vi skal se på dette straks. I «oldtests» har jeg bare gamle versjoner av atlas-gap-ana.py. «log» er mappen hvor de ferdige loggene havner. Det ligger også et skript der, «send-logs», som sender ferdige logger til Elasticsearch.

```
aleksjek@iou2:~$ ls log
2021-03-05T0100-finished  2021-03-18T1500-finished  2021-03-24T0800-finished
2021-03-07T0100-finished  2021-03-18T1600-finished  2021-03-24T0900-finished
2021-03-18T0000-finished  2021-03-18T1700-finished  2021-03-24T1000-finished
2021-03-18T0100-finished  2021-03-18T1800-finished  2021-03-24T1100-finished
2021-03-18T0200-finished  2021-03-18T1900-finished  2021-03-24T1200-finished
2021-03-18T0300-finished  2021-03-18T2000-finished  2021-03-24T1300-finished
2021-03-18T0400-finished  2021-03-18T2100-finished  2021-03-24T1400-finished
2021-03-18T0500-finished  2021-03-18T2200-finished  2021-03-24T1500-finished
2021-03-18T0600-finished  2021-03-18T2300-finished  2021-03-24T1600-finished
2021-03-18T0700-finished  2021-03-24T0000-finished  2021-03-24T1700-finished
2021-03-18T0800-finished  2021-03-24T0100-finished  2021-03-24T1800-finished
2021-03-18T0900-finished  2021-03-24T0200-finished  2021-03-24T1900-finished
2021-03-18T1000-finished  2021-03-24T0300-finished  2021-03-24T2000-finished
2021-03-18T1100-finished  2021-03-24T0400-finished  2021-03-24T2100-finished
2021-03-18T1200-finished  2021-03-24T0500-finished  2021-03-24T2200-finished
2021-03-18T1300-finished  2021-03-24T0600-finished  2021-03-24T2300-finished
2021-03-18T1400-finished  2021-03-24T0700-finished  send-logs
```

Figur 134: Innholdet i "log"-mappen

Vi kan begynne med å se på skriptet atlas-gap-ana i hovedmappen.

```
#!/bin/bash

#Starts analyzing 24 hours of files
#Correct input is ./atlas-gap_ana 2021-03-05
#if you want that particular date

#Checks to see if files are already downloaded or loaded
if [ -f "log/$1-loaded" ];
        then exit
fi

if [ -f "log/$1T0000-finished" ];
        then exit
fi
```

Figur 135: Første del av skriptet som analyserer 24 timer

Den øverste linjen forteller bare skriptet hvilket shell det skal kjøre kommandoene i. Deretter sjekker vi om filene allerede er lastet ned eller om de er lastet inn i Elasticsearch og slettet. Vi trenger bare sjekke en tilfeldig logg (her T0000) siden de alle blir ferdige ca. samtidig. «loaded»-filen lages av «send-logs»-skriptet og forteller oss om loggene allerede er lastet inn og slettet.

```
#Checks to see if files are on RIPE Atlas server
for i in {0..23}
do
    if [ $i -lt 10 ]
    then
        j=0${i}
    else
        j=${i}
    fi

    if ! wget --spider \
"https://data-store.ripe.net/datasets/atlas-daily-dumps/${1}/traceroute-${1}T${j}00.bz2" \
 2>/dev/null;
    then
        exit
    fi

done
```

Figur 136: Linux-skript sjekker om flatfilene er klare

Det neste vi gjør er at vi har en løkke som går 24 ganger. Det vi gjør her er at vi sjekker at alle de 24 loggene for denne dagen har blitt lagt ut av RIPE Atlas. Linken konstrueres av $1 som er datoparameteret som sendes med skriptet og $j som blir laget av $i som sier hvor langt vi har kommet i løkken. Grunnen til at vi ikke bare kan bruke $i er at vi er nødt til å legge til 0 foran tallet hvis det er mindre enn 10 slik at vi får klokkeslett som T0500 og ikke T500. Merk at disse tidspunktene er angitt i UTC siden det er dette RIPE Atlas bruker.

```
#Runs the program for each hourly log
#if the logs were not already downloaded
#and they were available on RIPE Atlas

for i in {0..23}
do
    if [ ${i} -lt 10 ]
    then
        j=0${i}
    else
        j=${i}
    fi

    python atlas-gap-ana.py ${1}T${j}00 &
done
```

Figur 137: Linux-skriptet kjører programmet 24 ganger

Nå kan vi bruke den samme type løkke for å starte en tråd i Linux hver for programmet atlas-gap-ana.py som analyserer hver sin time av datoen vår. På slutten av denne linjen har vi «&» som gjør at man ikke venter på hver kjøring blir ferdig, men starter alle etter hverandre.

Det er mye som er likt i «send-logs»-skriptet som ligger i log. Dette starter med å sjekke at loggene er lastet ned ferdig og at de ikke allerede har blitt lastet inn i Elasticsearch.

```
#!/bin/bash

#Sends 24 hours of logs to Elasticsearch/Kibana
#Correct input is ./send-logs 2021-03-05
#if you want that particular date

#Exits if the files haven't been downloaded
if [ ! -f "${1}T0000-finished" ]; then
        exit
fi

#Exits if the files have already been loaded
if [ -f "${1}-loaded" ]; then
        exit
fi
```

Figur 138: Linux-skript sjekker om utfallsloggene er klare

Dersom alt er greit, lastes loggene inn i Elasticsearch.

```
for i in {0..23}
do
        if [ ${i} -lt 10 ]
        then
                j=0${i}
        else
                j=${i}
        fi

        /home/oak/bin/elastic-send atlas-trace $1T${j}00-finished
        rm /tmp/elastic-send*

done
```

Figur 139: Linux-skript laster loggene inn i Elasticsearch/Kibana

Merk at jeg her ikke har & bak kommandoen. Dette betyr at hver elastic-send blir kjørt ferdig før man begynner på den neste. Jeg har gjort det slik fordi jeg har fått problemer hvis jeg prøver å sende alle samtidig med 2021-03-05T* for eksempel eller kjøre dem samtidig. Det tar likevel veldig kort tid før man finner alle loggene lagt trygt inn i Elasticsearch/Kibana. Merk også at vi etter hver innsending sletter alt i /tmp/-mappen som starter på elastic-send. Grunnen til dette er at tjeneren ellers vil bli overfylt med disse mellomlagringsfilene før Linux sletter dem selv.

```
#Leaves marker that files have been loaded
#so they are not loaded again
#Also removes old markers
rm /home/aleksjek/log/*-loaded
touch ${1}-loaded
rm /home/aleksjek/log/$1T*
```

Figur 140: Linux-skriptet avslutter

Når vi er ferdige legger vi igjen en fil som markerer at filene har blitt lastet inn og ikke trengs lastes ned igjen. I tillegg sletter vi gamle markeringer og loggene vi ikke trenger lenger.

7.2. Bruk av «crontab»

Det følgende er ikke slått på akkurat nå, men jeg kan vise hvordan man setter opp cron til å kjøre disse skriptene med jevne mellomrom slik at nye logger hele tiden blir lastet inn.

Først skriver man crontab -e for å åpne crontaben til brukeren min.

```
# and what command to run for the task
#
# To define the time you can provide concrete values for
# minute (m), hour (h), day of month (dom), month (mon),
# and day of week (dow) or use '*' in these fields (for 'any').
#
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h  dom mon dow   command
#
```

Figur 141: Tom crontab

Nå lager jeg linjer for de to skriptene jeg ønsker skal kjøres hele tiden.

```
# and day of week (dow) or use '*' in these fields (for 'any').
#
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h  dom mon dow   command
DATEVAR='date -d yesterday +%Y-%m-%d'
00 * * * * /home/aleksjek/atlas-gap-ana $($DATEVAR)
59 * * * * /home/aleksjek/log/send-logs $($DATEVAR)
```

Figur 142: Ferdig crontab

Den øverste linjen gir oss gårsdagens dato i formatet skriptene liker.

Nedlastningsskriptet blir forsøkt hver ny time. Dersom det finnes logger som hverken er lastet ned eller lastet inn i Elasticsearch, blir disse lastet ned. Hvis ikke avslutter programmet med en gang. Det andre skriptet kjøres på det 59. minuttet. Man kan regne med at det første skriptet er ferdig innen da fra forrige gang. Hvis nye filer er lastet ned, lastes disse inn i Elasticsearch.

8. Fjerde del: Eksempel på analyse av utfall

Merk at denne analysen er basert på en eldre versjon av programmet enn den som ble innlevert. Det kan derfor være små forskjeller i hva slags data Kibana har.

8.1. Et mislykket forsøk

I denne delen skal jeg forsøke å bruke systemet til å studere utbrudd.

Hvis vi husker tilbake, var det mest utfallstid på et spesielt punkt i USA.

Figur 143: Stedet med mest utfallstid

Dette er jo interessant så vi vil prøve å se nærmere på hva dette er for noe. Jeg søker på koordinaten og finner at ca. 20 000 logger er knyttet til koordinatene. Nesten 10% av loggene.

Time	duration	coordinates	location
Mar 24, 2021 @ 06:14:54.000	2,703	37.751,-97.822	United States
Mar 24, 2021 @ 14:13:20.000	2,701	37.751,-97.822	United States
Mar 24, 2021 @ 08:14:56.000	2,700	37.751,-97.822	United States
Mar 24, 2021 @ 19:14:59.000	2,699	37.751,-97.822	United States
Mar 24, 2021 @ 19:14:58.000	2,697	37.751,-97.822	United States
Mar 24, 2021 @ 19:15:01.000	2,693	37.751,-97.822	United States

Figur 144: Mange utfall knyttet til én plass

Jeg lurer først på om dette kan være et slags enormt knutepunkt for Internett i USA eller hele verden. Jeg begynner å google for å finne ut hva dette stedet i Kansas er.

Det viser seg at svaret på mysteriet er ganske enkelt. Koordinatene 37, -97 er rett og slett standardverdien GeoIP2 oppgir for ukjente koordinater i USA. Visstnok er punktet valgt fordi det ligger midt i USA og familien som bor der har hatt mange problemer. (Denver Post, 2016)

Dette viser at vi ikke kan stole blindt på resultatene vi får, men hele tiden må forsøke å tolke dataene og resultatene vi får inn for å forsøke å forstå hva som ligger bak.

8.2. Et mer vellykket forsøk

Jeg bestemte meg for å søke etter en «lastReplyFrom» som gikk mye igjen og hadde høy «duration». «2001:2034:0:22::1» er en svensk adresse som tilhører Telia AS.

Time	duration ▾	coordinates	location
Mar 24, 2021 @ 22:07:46.000	2,703	59.3247,18.056	Sweden
Mar 24, 2021 @ 12:07:53.000	2,701	59.3247,18.056	Sweden
Mar 24, 2021 @ 15:07:52.000	2,697	59.3247,18.056	Sweden
Mar 24, 2021 @ 02:07:50.000	2,695	59.3247,18.056	Sweden
Mar 24, 2021 @ 21:07:57.000	2,688	59.3247,18.056	Sweden

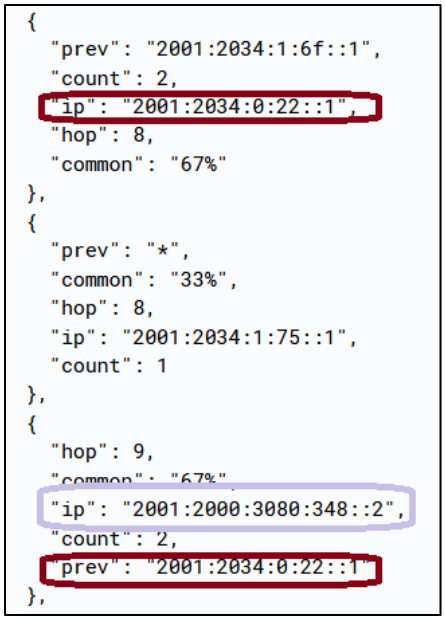
Figur 145: Mistenkelige logger

Vi ser at den alltid mislykkes på samme adresse.


```
{"unique_route":"2001:4644:a25:5:c225:e9ff:fe99:ff58-to-2a00:e400:2:1::a","record_type":"Detail","order":1,"start_time":1616620
066,"end_time":1616622769,"failure_count":6,"polling_frequency":338,"duration":2703,"last_reply_from":"2001:2034:0:22::1","AS_n
umber":"AS1299","AS_owner":"Telia Company AB","location":"Sweden","coordinates":"59.3247,18.056","errors":[[["Destination unrea
chable"],["Destination unreachable"],["Destination unreachable"]],[["No reply"],["No reply"],["Destination unreachable"]],[["No
reply"],["No reply"],["Destination unreachable"]],[["No reply"],["No reply"],["Destination unreachable"]],[["No reply"],["Desti
nation unreachable"],["Destination unreachable"]],[["No reply"],["Destination unreachable"],["Destination unreachable"]]],"max_
path":19,"min_path":10,"topologies":[["2001:4644:a25:5:20c:29ff:fe24:f55","2001:4600:0:200::5c","2001:4600:0:100::2","2001:460
0::e","2001:2000:3080:9af::1","2001:2034:1:75::1","2001:2034:1:6f::1","2001:2034:0:22::1","2001:2000:3080:348::2","*","*","*","
*","*","*","*","*","*","2001:2034:0:22::1"],["2001:4644:a25:5:20c:29ff:fe24:f55","2001:4600:0:200::5c","2001:4600:0:100::2","20
01:4600::e","2001:2000:3080:9af::1","2001:2034:1:75::1","2001:2034:1:6f::1","2001:2034:0:22::1","2001:2000:3080:348::2","*","
*","*","2001:2034:0:22::1"],["2001:4644:a25:5:20c:29ff:fe24:f55","2001:4600:0:200::5c","2001:4600:0:100::2","2001:4600::e","2001:2000:3080:9af::
1","2001:2034:1:75::1","2001:2034:1:6f::1","2001:2034:0:22::1","2001:2000:3080:348::2","*","*","*","2001:2034:0:22::1"],["2001:
4644:a25:5:20c:29ff:fe24:f55","2001:4600:0:200::5c","2001:4600:0:100::2","2001:4600::e","*","*","*","2001:2034:1:75::1","2001:2
034:1:6f::1","2001:2034:0:22::1","2001:2000:3080:348::2","2001:2034:0:22::1"],["2001:4644:a25:5:20c:29ff:fe24:f55","2001:4600:
0:200::5c","2001:4600:0:100::2","2001:4600::e","2001:2000:3080:9af::1","2001:2034:1:75::1","2001:2034:1:6f::1","2001:2034:0:2
2::1","2001:2000:3080:348::2","2001:2034:0:22::1"]}
```


Figur 146: Får alltid problemer samme sted

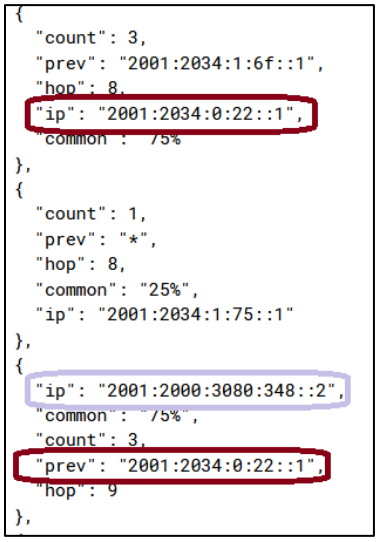
Når ruten fungerer som den skal, ruter maskinen imidlertid til «2001:200:3080:348::2».



Figur 147: Eksempel på normal ruting

Det kan dermed virke som om maskinen «2001:2034:0:22::1» den 24. mars 2021 mellom klokken 22:07 og 22:57 er nede. Dette er vel å merke basert på kun 6 mislykkede traceroutes som det gjennomsnittlig er ca. 300 sekunder mellom. Feilmeldingene vi får fra disse utfallene er «Destination unreachable» og «No reply». Kanskje Telia AS har svar på hvorfor deres maskin i dette tidsrommet plutselig ikke visste hvor den skulle rute pakkene videre.

Her er enda et eksempel, fra tidlig på morgenen den samme dagen, som tyder på at «2001:2034:0:22::1» under normale omstendigheter ruter videre til «2001:2000:3080:348::2».



Figur 148: Enda et eksempel på normal ruting

9. Referanser

Cormen, m. fl. (2009) *Introduction to Algorithms*. 3. utgv. Cambridge, Massachusetts: MIT Press.

Denver Post (2016) *Kansas home wound up with 600 million IP addresses and a world of trouble*. Tilgjengelig fra: https://www.denverpost.com/2016/08/10/lawsuit-kansas-home-600-million-ip-addresses/ (Hentet: 6. april, 2021).

glogg (2021) *glogg - the fast, smart log explorer*. Tilgjengelig fra: https://glogg.bonnefon.org/ (Hentet: 5. april, 2021).

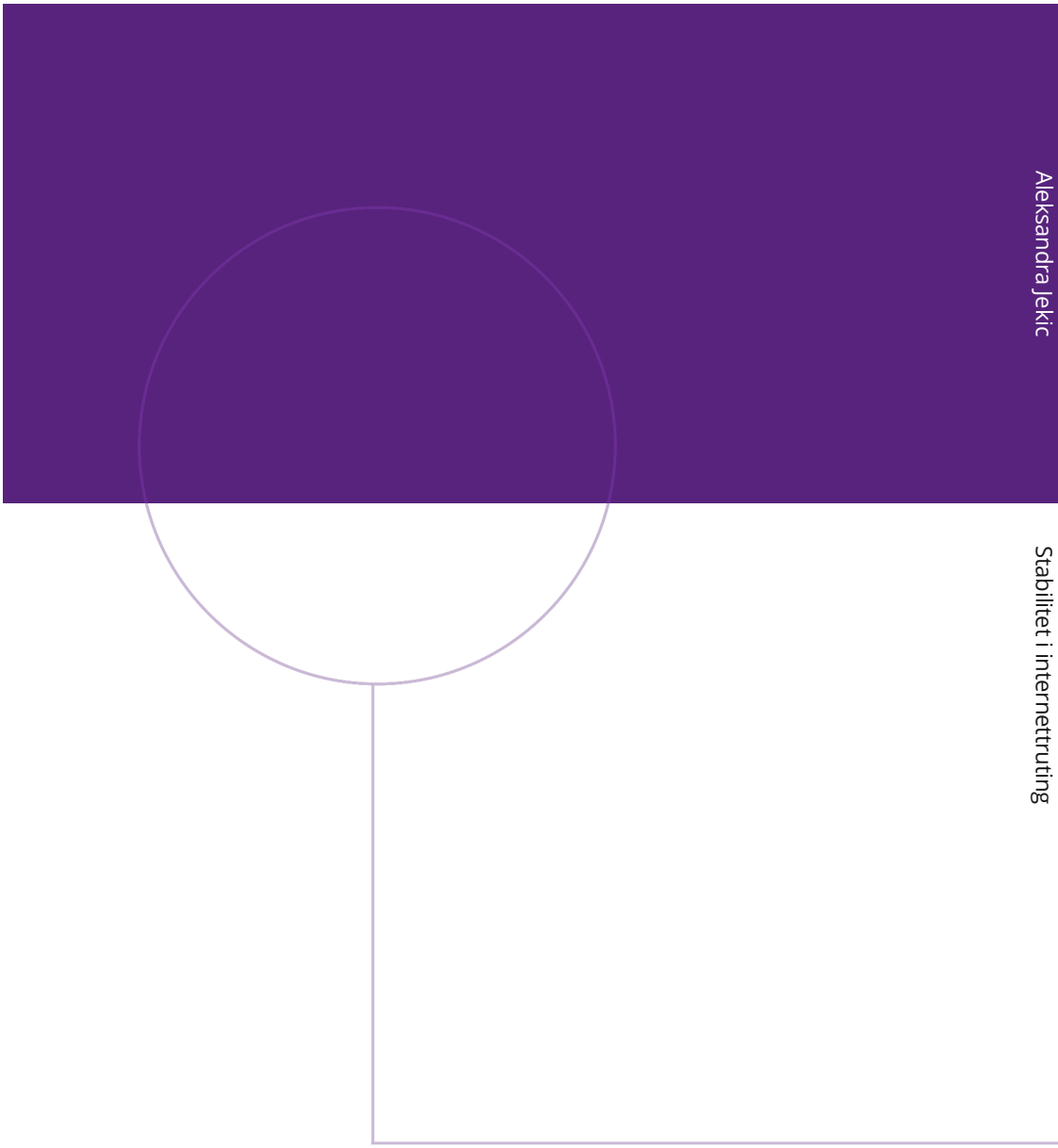
Gopakumar, G. (2020) *try-except vs If in Python*. Tilgjengelig fra: https://www.geeksforgeeks.org/try-except-vs-if-in-python/ (Hentet: 5. april, 2021).

Rekhter, Y. m. fl. (1995) *A Border Gateway Protocol 4 (BGP-4)*. Tilgjengelig fra: https://tools.ietf.org/html/rfc1771 (Hentet: 6. april, 2021).

RIPE Atlas (2021) *RIPE Atlas - Raw data structure documentation*. Tilgjengelig fra: https://atlas.ripe.net/docs/data_struct/ (Hentet: 5. april, 2021).

Tayal, R. (2019) *Python's GIL — A Hurdle to Multithreaded Program*. Tilgjengelig fra: https://medium.com/python-features/pythons-gil-a-hurdle-to-multithreaded-program-d04ad9c1a63 (Hentet: 6. april 2021).

Uninett (2021) *DragonLab*. Tilgjengelig fra: https://www.uninett.no/node/1819 (Hentet: 5. april, 2021).



Aleksandra Jekic

Stabilitet i internettruting

NTNU
Kunnskap for en bedre verden

UNINETT