Traffic Shaping

Mads Ritter Nørskov
Thomas Lützen

Kongens Lyngby 2014
IMM-B.Sc-2014

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk IMM-B.Sc-2014

Summary (English)

The goal of this thesis is to provide insight into the development of an API that should be able to communicate with a newly implemented traffic shaper solution. The API should have an adjacent switchdatabase and should be able to synchronize with a billing database. The first chapter introduced contains descriptions of the state of the art in networking, traffic shaping algorithms and database normalization. Different concepts like virtual lan's, queue in queue and quality of service and the differences between traffic shaping and traffic policing are outlined. After that comes an analysis chapter which looks at the current traffic shaper solution and what the options for replacing it are. Furthermore the analysis contains a section on appropriate programming methods and design patterns for this project. After the analysis chapter the choice of database structure and the different choices of programming methods and what language to use are presented in a design chapter. Based on the choices made in the design chapter the most important methods and how the communication with the different databases works is described in a implementation chapter. The solution is then evaluated by a series of tests to see if it satisfies the demands. Furthermore a list of possible extensions for future work is made. In the end is a conclusion is made on the project as a whole.

Summary (Danish)

Målet for denne afhandling er, at give et indblik i udviklingen af et API, der kan kommunikere med en nyimplementeret traffic shaper løsning. Dette API skal have en tilhørende switchdatabase og skal kunne synkronisere med en faktureringsdatabase. Først præsenteres state of the art indenfor netværk, traffic shaping algoritmer og databasenormalisering. Der beskrives forskellige koncepter som virtuelle lan, queue in queue, quality of service og forskellene mellem shaping og policing opridses. Dernæst kommer en analyse af de forskellige krav til løsningen, den nuværende traffic shaper løsning beskrives og der kigges på hvilke muligheder der er for at erstatte den. Analysen indeholder desuden et afsnit om hensigtsmæssige programmeringsmetoder og design patterns. Efter analyseafsnittet fremlægges valget af databaseopbygning og de forskellige valg af programmeringsmetode og sprog i et design afsnit. Ud fra designet beskrives implentationen i et implementationsafsnit der går i dybden med de vigtigste metodekald og hvordan der kommunikeres med de forskellige databaser. Efter dette evalueres der i et evalueringsafsnit. Dette undersøger via en række test om løsningen opfylder de opstillede krav. Derudover opstilles der en liste af mulige udvidelser for fremtidigt arbejde. Til sidst konkluderes der på projektet som en helhed.

Preface

Denne afhandling er udarbejdet under Institut for Matematik og Computer science hos Danmarks Tekniske Universitet med opfyldelse af de opstillede krav for erhvervelse af en B.Sc i softwareteknologi. Afhandlingen behandler problemstillinger som:

- Hvilke aspekter skal beskrives for at kunne forstå begrebet traffic shaping?
- Hvad er Quality of service og hvad kan det bruges til?
- Hvordan man vil opbygge et API der kommunikerer med flere forskellige databaser og services?

Afhandlingen består af en gennemgang af de teoretiske begreber der er nødvendige for at forstå hvad en traffic shaper er og hvorfor denne er vigtig. Samtidig beskæftiger rapporten sig også med databasenormalisering og programmeringssproget Scala, især med henblik på det asynkrone toolkit Akka. Vi valgte netop dette emne da vi fandt hele ideen omkring implemenentation af en traffic shaper interessant.

Vi har hver især arbejdet med support af kunder der har været berørt af en traffic shaper, men har aldrig opnået den fulde forståelse af, hvorfor en traffic shaper er vigtig, hvad en sådan præcist gør, og hvilken teori der ligger til grund for en sådan?

Lyngby, 31-January-2014

Mads Ritter Nørskov
Thomas Lützen

Acknowledgements

Vi vil gerne takke vores vejleder Christian Damsgaard Jensen for godt feedback til vores projekt samt at opfordre os til at planlægge forløbet og følge den plan.

Derudover vil vi gerne takke Cirque for opbakning, brug af lokaler og hjælp til forklaring af netværksstruktur. Især Martin Schiøtz, Jacob Vous Petersen og Anders Ørsted Petersen for stor hjælp i forbindelse med tekniske spørgsmål. Derudover vil vi gerne takke Jan Munkhammer hos IPNett for hjælp til at få en funktionel SOAP forbindelse til at fungere.

Til sidst vil vi gerne takke vores kærester, venner og familie for støtte igennem projektet og ikke mindst at lytte til os når vi forsøgte at forklare projektets detaljer.

Contents

Summary (English) **i**

Summary (Danish) **iii**

Preface **v**

Acknowledgements **vii**

1 Introduktion **1**
1.1 Indledning 1
1.2 Traffic shaper skift hos Cirque 2
1.2.1 Problemstilling 2
1.2.2 Problemformulering 3
1.3 Programmeringsindledning 4
1.4 Rapportstruktur 4

2 State of the Art **7**
2.1 Indledning 7
2.2 Quality of Service 8
2.2.1 Indledning 8
2.2.2 Overprovisioning 9
2.2.3 QoS 9
2.3 Virtual local area network 10
2.3.1 Indledning 10
2.3.2 QinQ 11
2.4 Traffic shaping 13
2.4.1 Indledning 13
2.4.2 Shaping vs policing 13
2.4.3 Shaping metoder 15

2.5 Database 17
2.5.1 Behovet for databaser 17
2.5.2 Database Normalformer 17

3 Analyse **23**
3.1 Indledning 23
3.2 Eksisterende traffic shaping løsning 24
3.3 Forskellige Traffic shaper løsninger 26
3.3.1 Udskiftning til 2x Juniper MX480 27
3.3.2 Udskiftning til 2x Juniper MX10 27
3.3.3 Udvidelse af eksisterende Linux traffic shaper 27
3.3.4 Ny selvudviklet løsning 28
3.3.5 3. part udviklet Linux server med specialbyggede netkort 28
3.4 Programmeringsanalyse 28
3.4.1 Asynkronitet 29
3.4.2 Forskellige programmeringsmuligheder 32
3.5 Databasebehov 33
3.6 Design patterns 34
3.6.1 Model View Controller 34
3.6.2 Data Access Object Pattern 34
3.7 Opsummering 35

4 Design **37**
4.1 Indledning 37
4.2 Valg af traffic shaper løsning 38
4.3 Database Design 39
4.4 Design patterns 40
4.4.1 MVC Design 40
4.4.2 DAO 41
4.5 Programmeringsdesign 42
4.5.1 Indledning 42
4.5.2 Scala 43
4.5.3 Play framework 43
4.5.4 Actors 46
4.5.5 Struktur 49
4.6 Opsummering 49

5 Implementation **51**
5.1 Indledning 51
5.2 Databaseimplementation 52
5.3 Implementation af program 54
5.3.1 Indledning 54
5.3.2 Design patterns 54
5.3.3 Actorsystem 65

5.4 Opsummering 74

6 Evaluering **75**
6.1 Indledning 75
6.2 Tests 75
6.3 Diskussion 80
6.4 Udvidelsespunkter 81
6.5 Opsummering 82

7 Konklusion **85**

A Traffic Shaping for boligforeninger **87**
A.0.1 Baggrund 87
A.0.2 Formål 87
A.0.3 Succes kriterier 87
A.0.4 Løsningsforslag 88
A.1 Udskiftning af eksisterende core til 2 x MX480 – Metro Activate fra IPNett 88
A.1.1 Fordele 88
A.1.2 Ulemper 88
A.1.3 Økonomi 89
A.1.4 Tidsperspektiv for implementering 89
A.2 2 x MX10 – Metro Activate fra IPNett 89
A.2.1 Fordele 89
A.2.2 Ulemper 89
A.2.3 Økonomi 89
A.2.4 Tidsperpektiv 89
A.3 Udvidelse af eksisterende linux traffic shaping 90
A.3.1 Fordele 90
A.3.2 Ulemper 90
A.3.3 Økonomi 90
A.3.4 Tidsperpektiv 90
A.4 Ny egen udviklet løsning (BSD/Linux) 91
A.4.1 Fordele 91
A.4.2 Ulemper 91
A.4.3 Økonomi 91
A.4.4 Tidsperpektiv 91
A.5 3 part udviklet linux server med specialbygget netkort 91
A.5.1 Fordele 91
A.5.2 Ulemper 92
A.5.3 Økonomi 92
A.5.4 Tidsperpektiv 92

B Kommunikationsdiagram **93**

C Soap API dokumentation **95**

D SwitchDok Datatypes **123**

E SwitchDAOActor case classes **125**

F SyncActor.scala **129**

Bibliography **135**

Chapter 1

Introduktion

1.1 Indledning

Langt de fleste danske hjem har i dag en internetforbindelse. Denne internetforbindelse er leveret af en internet leverandør (På engelsk Internet Service Provider eller ISP). Disse ISP'ere leverer internet til hvert enkelt hjem alt efter det abonnement som kunden betaler for. De fleste internetabonnementer er i dag begrænset til en bestemt øvre hastighed, f.eks. 20mbit internetforbindelse. For at alle kunder i landet kan få leveret den hastighed de ønsker skal der opbygges en tilstrækkelig netværksinfrastruktur som de forskellige ISP'er betaler for. Det er derfor i ISP'ens interesse at de ikke har brugere der får en højere hastighed end de betaler for, og samtidig er brugerne interesseret i at de får den hastighed de betaler for. Hvordan sikrer man så, som ISP, at ens brugere får den hastighed de betaler for, og samtidig minimerer antallet af sortseere? Svaret er en server type der populært bliver kaldt for en traffic shaper.

Denne rapport vil beskæftige sig med den tekniske forklaring om; Hvad en traffic shaper er, hvordan en sådan fungerer og hvorfor en traffic shaper er vigtig for et netværk med en større mængde brugere. Vores rapport vil derudover se på et integrationsmodul mellem Cirque's nye traffic shaper og deres eksisterende faktureringsdatabase. Dette integrationsmodul vil fungere som programmeringsdelen af vores projekt. Grunden til at en ny traffic shaper er nødvendig

er at det nuværende udstyr ikke længere kan følge med behovet, dette vil blive forklaret i detaljer senere [Se afsnit 3.2].

1.2 Traffic shaper skift hos Cirque

1.2.1 Problemstilling

Cirque A/S er en virksomhed der leverer telefoni og bredbåndsløsninger til flere store boligforeninger i Danmark samt samlede teleløsninger til erhvervslivet. Grundet overbelastning på deres gamle traffic shapere(TS) har Cirque valgt at udskifte deres nuværende TS'ere til et nyt system.

En traffic shaper er en server der står for at styre netværkstrafikken for en række IP-adresser i et netværk i forhold til en tildelt hastighed til hver IP-adresse. Dette gør det muligt at styre enkelte kunders hastigheder og lukke for dem i forhold til manglende betaling. Den nye løsning tilbyder queue in queue(QinQ) hvilket muliggør indpakningen af virtual local area network(VLAN) i et ydre VLAN. Derved kan validering ske på et højere niveau og fjerne denne belastning i forbindelse med shaping af enkelte pakker.

QinQ er et netværks koncept der bygger på netværks standarden 802.1ad. Konceptet går ud på at fange de enkelte brugeres private VLAN og indkapsle det i et ydre VLAN for at tilføje et ekstra lag til hver enkelt brugers VLAN. Dette giver en række fordele i forhold til administration af brugere samt deres sikkerhed; blandt andet at alle brugeres VLAN bliver fuldstændig beskyttet af det ydre VLAN lag, således at de enkelte private VLAN ikke kan se, eller tilgå, andre VLAN i samme netværk. Dette har hidtil været et problem i ældre opsætninger hos Cirque hvor kunder teoretisk set kunne tilgå naboens netværksprinter. Der har dog aldrig været registreret at det reelt er forsøgt.

Den nye løsning forventes at have et API til styring af hastigheder for de enkelte brugere således at hastighederne kan styres både automatisk, f.eks. via synkronisering fra den eksisterende kunde database, og manuelt, så kundeservice kan justere hastigheder i forbindelse med fejlsøgning. Derudover åbner den nye løsning op for mulighed for packet capturing der for eksempel kan gøre at ikke tilmeldte kunder kan tilmelde sig med det samme via et selvbetjeningssite. Ligeledes tilbyder den nye løsning dynamiske IP-tabeller så den ikke skal huske og kunne validere alle IP-adresserne fra VLAN's, men kun disse der har en aktiv tildelt IP adresse.

Vi vil derfor starte med at se på grundlaget for skiftet af den gamle løsning, hvad gjorde den utilstrækkelig i forhold til traffic shaping af en kundebase af firmaets størrelse, og derefter se på forcerne i den nye løsning, samt perspektivere på om den bedst mulige løsning er valgt. Derudover vil vi sammen med Cirque restrukturere traffic shaper løsningen og kundebasen for automatisk styring af abonnementshastigheder.

1.2.2 Problemformulering

Vi inddeler problemstillingen i to hovedspørgsmål med hver nogle underspørgsmål.
Hvad er en TS?

- Hvilke aspekter indgår i en TS løsning?
- Hvad gør den nuværende løsning utilstrækkelig og opfylder den nye TS de nødvendige krav til Cirque?
- Er der nogen begrænsninger på den nye TS? Hvilke?
- Er den valgte løsning optimal for Cirque? Hvilke andre løsninger kunne eventuelt have været interessante?

Hvordan udfører man programmeringsmæssigt en opgradering af en TS, med henblik på at styre hastigheder for den eksisterende kundebase?

- Hvordan fungerer QinQ?
- Hvordan berører QinQ kundebasen, hvordan skal den ændres?
- Hvilke muligheder åbner packet capturing op for i forhold til automatisering og selvbetjening af kundebasen?
- Hvordan sammenkodes TS API'et med kundedatabasen?
- Hvordan muliggøres automatiseringen af kundernes abonnementer i forhold til systemerne, i forhold til at forhindre ikke betalende kunder og at levere hastigheden som kunden er berettiget til?

1.3 Programmeringsindledning

Til den nye traffic shaping løsning Cirque køber, ønsker de mulighed for nemt at kunne redigere i de shaping regler der er gældende for deres brugere, således at de kan levere stabile internetløsninger til deres betalende kunder. Samtidig ønsker de at systemet skal automatiseres så meget som muligt, så nye kunder automatisk bliver oprettet og gamle kunder automatisk bliver lukket. Dette sker ved at skabe kommunikation mellem Cirque's nuværende faktureringsdatabase, hvor informationer om kundernes abonnementer findes, og den nye traffic shaper.

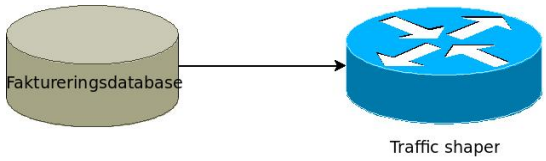


Figure 1.1: Et indledende overblik over strukturen af vores program hvor vi ønsker at synkronisere mellem den eksisterende faktureringsdatabase og den nye traffic shaper

Derudover ønsker Cirque at kunne kommunikere med traffic shaperen via et API således at de kan supportere deres kunder der skulle opleve problemer.

Vi vil løbende igennem rapporten iterere på ovenstående diagram hvor vi vil præcisere detaljerne i vores program.

1.4 Rapportstruktur

Rapporten er opbygget efter følgende rapport struktur:

Indledning Indeholder en introduktion til rapporten og vores program samt problemformuleringen.

State of the art Beskriver de netværkstekniske aspekter der er nødvendige for forståelse af hvorfor dette projekt er nødvendigt. Det har desuden en beskrivelse af databaser, da dette vil vise sig at blive en vigtig del i vores program.

Analyse I analyse afsnittet begynder vi at se på de forskellige muligheder og begrænsninger vi har i forhold til vores program. Deriblandt mulige traffic

shaper løsninger, mulige valg af programmeringssprog og beskrivelse af behovet for databaser.

Design Vores design kapitel vil gå i dybden med de valg vi, og Cirque, tager i forbindelse med dette projekt. Dette vil blandt andet inkludere valg af traffic shaper, design af databaser, valg af design patterns og design af program.

Implementation Dette kapitel vil primært beskæftige sig med den programmeringsmæssige del af vores projekt. Nemlig hvordan vi har implemeteret de forskellige aspekter af programmet og der tilhørende databaser.

Evaluering Indeholder tests og diskussion af vores program, samt udvidelsespunkter for denne.

Konklusion Afrundelse af rapporten hvori vores diskussion bliver opsummeret og hvor vi bedømmer om vores program lever op til de krav der er blevet sat i indledningen.

Chapter 2

State of the Art

2.1 Indledning

I dette kapitel vil vi se på den teori der ligger bagved en traffic shaper, samt nogle strukturelle software aspekter til vores program. Som det blev nævnt i vores programmeringsindledning[Afsnit 1.3], ønsker vi at lave et synkroniseringsmodul mellem Cirque's faktureringsdatabase og den nye traffic shaper til automatisk oprettelse og nedlæggelse af kunder. Som vist i introduktionen [Figur 1.1] skal faktureringsdatabasen sende oplysninger om; Nye kunder der skal oprettes, ændringer i eksisterende kunder og nedlæggelser af gamle kunder til traffic shaperen. En kunde bliver i faktureringsdatabasen identificeret ud fra deres navn og adresse samt et unikt kundenummer. I den eksisterende traffic shaper bliver en internetforbindelse identificeret udfra deres IP adresse, og noget lignende vil højst sandsynligt gælde for den nye traffic shaper. For at disse to kan snakke sammen, skal vi derfor opsætte en database der kan sammenkoble kunders fysiske adresser med deres IP adresser og dette vil få vores program struktur til at se således ud:

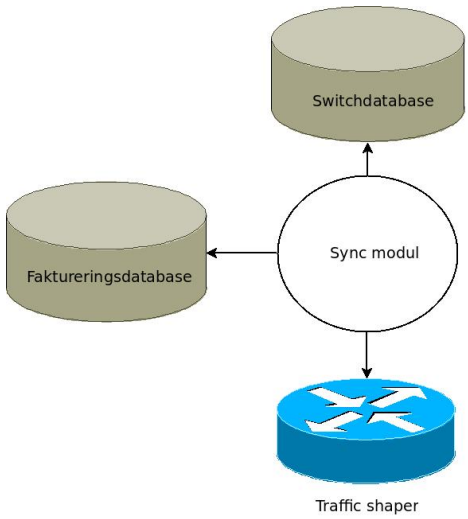


Figure 2.1: Her ses tilføjelsen af et mellemliggende styrings lag til strukturen i form af vores sync modul. Derudover ses en ny database ved navn switchdatabase der håndterer koblingen imellem kunders adresser og deres IP adresser

De af Cirque's kunder der bliver berørt af denne løsning er alle beboere i boligforeninger rundt omkring i Danmark. Hver boligforening har et switch system der sørger for leveringen af internet til de enkelte boliger. For at kunne adskille de enkelte boligforeninger fra hinanden vil der være oplysninger om switche at finde i vores switchdatabase. Da det er vigtigt at databasen fungerer hurtigt, effektivt og overskueligt vil vi bestræbe os på at vores switchdatabase er normaliseret tilstrækkeligt. Database normalisering vil derfor blive forklaret mere dybdegående senere i dette kapitel.

2.2 Quality of Service

2.2.1 Indledning

Basale netværks løsninger arbejder ofte mod at forbedre netværkets ydeevne, og derved levere den nødvendige datapakke så hurtigt som muligt, alt efter hvor stort pres der er på systemet. Da man typisk ikke ønsker at have et system med overprovisioning [Se afsnit 2.2.2], og dog stadig kan være udsat for situationer

hvor belastning af systemet ikke må foresage en forsinkelse for enkelte pakker. Dette sker f.eks. i forbindelse med VoIP telefoni hvor man skal have konstant mulighed for 64kbps data [AST10, Section 5.4]. Hvis en VoIP telefon ikke kan opnå den nødvendige dataforbindelse konstant vil brugeren opleve udfald, eller i værste fald opleve at opkaldet bliver droppet. Da dette langt fra er hensigtsmæssigt er man nødt til at finde forskellige løsninger på dette problem. Disse løsningsmuligheder vil blive beskrevet nedenfor.

2.2.2 Overprovisioning

Som beskrevet af Tanenbaum [AST10, Section 5.4] er en meget simpel løsning til ovenfor beskrevne problem at bygge et netværk, hvor man aldrig kan nå den øvre grænse for trafik som netværket kan håndtere, og derved ikke opleve flaskehalse i netværket. En sådan løsning kaldes for overprovisioning. Der vil derfor ikke være behov for at behandle trafikken, efter en eller flere af de metoder der vil blive forklaret nedenfor. Der er derfor ikke noget pakketab, eller nogen signifikant forsinkelse af alle former for trafik i netværket. Dette vil naturligvis være et perfekt netværk, da det hverken har pakketab, høj latency eller jitter (udsvingninger i latency) [Cis06]. Overprovisionering er dog typisk ikke en foretrukken løsning, da hvis man betaler sig til at være forberedt på alle situationer, vil dette naturligvis vil være meget dyrt. Man har derfor fundet på andre måder, hvor en af disse er Quality of Service (QoS), der bliver beskrevet nedenfor.

2.2.3 QoS

QoS er en metode hvorpå man kan sikre at visse applikationer altid kan opnå et minimum eller maksimum trafik igennem netværket. Man kan derfor spare på størrelsen af netværket, og samtidig være sikker på at f.eks. tele-trafik kommer igennem en eventuel flaskehals med det samme, uanset belastningen på denne flaskehals. QoS kan derfor fungere som et nødspor på en motorvej for bestemte typer af data. Et nødspor sikrer blandt andet at udrykningskøretøjer kan komme frem så hurtigt som muligt, uanset belastningen af motorvejen på det pågældende tidspunkt. Et netværk har fire spørgsmål der skal addresseres for at kunne sikre QoS [AST10, Section 5.4]:

- Hvor meget data de forskellige applikationer skal bruge fra netværket?
- Hvordan bliver den indgående trafik reguleret?

- Hvordan kan man reservere tilstrækkelige ressourcer ved routerne for at sikre ydeevnen?
- Kan netværket optimeres til at acceptere mere trafik?

For at indføre QoS på et netværk er det derfor nødvendigt at kortlægge hvilke kilder og destinationer et netværk består af, hvad størrelsen af flowet (flow beskriver en strøm af pakker mellem en kilde og en destination) er og hvad behovet for den enkelte applikation er. Som nævnt tidligere vil et perfekt netværk være uden delay, jitter og pakketab, samtidig med at netværket har en båndbredde der er tilstrækkelig stor til at alle applikationer har nok i alle situationer. Da dette er en meget dyr og derfor typisk utopisk løsning indeler man alle applikationer der findes på dette netværk efter deres behov for netop disse fire kriterier (båndbredde, delay, jitter og pakketab). Udfra disse kan man derfor kortlægge en prioritet for alle applikationer efter deres behov. Ifølge Tanenbaum [AST10, Section 5.4] er de typer af applikationer der kan laves QoS på, applikationer der har et af følgende fire krav:

- Konstant bit rate (f.eks. VoIP telefoni)
- Real tid variabel bit rate (f.eks. webcam chat)
- Ikke real tid variabel bit rate(f.eks. streaming af film og musik)
- Tilgængelig bit rate (f.eks. fil overførsler)

QoS for en applikation er typisk kun i effekt når applikationen forsøger at afsende data, f.eks. har en telefon kun reserveret den konstante båndbredde den behøver når et opkald bliver foretaget. Dette er i kontrast til vores eksempel med motorvejen ovenfor hvor QoS for udrykningskøretøjer er i funktion hele tiden. Hvis man derimod ændrede motorvejen til at almindelige biler godt måtte køre i nødsporet, med mindre en udrykning var på vej, ville man have et mere fuldendt eksempel på QoS. Dette er på nuværende tidspunkt implementeret som forsøg på Hillerødmotorvejen [And13].

2.3 Virtual local area network

2.3.1 Indledning

Et Virtual Local Area Network (VLAN) er et netværk af computere der opfører sig som om de er koblet på det samme lokalnetværk selvom de kan være ad-

skilt. [AST10] Dette er styret i softwarelaget på de forskellige switche i f.eks et virksomhedsnetværk. Der kan nemlig være et behov for at have de forskellige afdelinger adskilt af hensyn til ydeevne og sikkerhedshensyn. HR afdelingen kan have fortrolige dokumenter som udviklingsafdelingen ikke skal have adgang til og vice versa. Måden hvorpå et VLAN implementeres er at tilføje et VLAN tag bestående af et 2 x 2 byte felt efter afsender mac-adressen i en mac ramme. Dette foregår i mac sublaget i Data link laget i forhold til OSI modellen. De første 2 bytes indeholder VLAN protocol id'et der altid er 0x8100. Da dette er større end 1500 bytes som er den største data enhed der kan sendes over et ethernet netværk bliver det opfattet som en type og ikke en længde. De næste 2 bytes indeholder 3 felter. Det første felt bestående af 3 bits prioritet der kan bruges til at implementere QoS. Det andet felt bestående af 1 bit er Canonical Format Identifier der bruges for kompatibilitet mellem ethernet og token ring netværk. Det sidste felt bestående af de resterende 12 bits er VLAN id'et hvilket giver værdier mellem 0 og 4095. Dog bruges 0 typisk til user priority packets og 4095 fungerer som default VID [oIT07]

Hos Cirque benyttes VLAN's i forbindelse med switchene hos de forskellige boligforeninger som Cirque leverer internet til. Hver lejlighed/kunde har sit eget VLAN id der giver dem hver deres lokalnetværk selvom de er koblet på samme switch. Dog er der en lille sikkerhedsrisiko ved denne opsætning og dette går ud på at en hacker får adgang til hovedswitchen i en boligforening. Hvis der opnås adgang til denne ville personen kunne tilgå alle de forskellige VLAN's i foreningen. En af grundende til udskiftningen af traffic shaperne er at forhindre denne potentielle sikkerhedsbrist.

2.3.2 QinQ

Da behovet for at levere bredbånd til flere kunder stiger i fremtiden vil der kunne opstå en situation hvor antallet af brugere i en boligforening overstiger 4094 der er det maksimale antal VLAN ID'er man kan have i et netværk. Dette problem kan dog løses med QinQ som er en del af 802.1ad standarden [IEE05]. QinQ fungerer ved at tilføje et ydre VLAN tag til en brugers pakke inden den forlader hovedswitchen i en boligforening. Når pakken rammer traffic shaperen fjernes begge VLAN tags hvorefter der udføres shaping på pakken og brugeren får derved den hastighed der svarer til deres abonnement. Da QinQ indeholder både et ydre og indre VLAN tags gøres derved muligt at opnå 4094*4094 = 16760836 unikke VLAN ID'er.

Måden hvorpå en ethernet frame indpakkes er som følger:

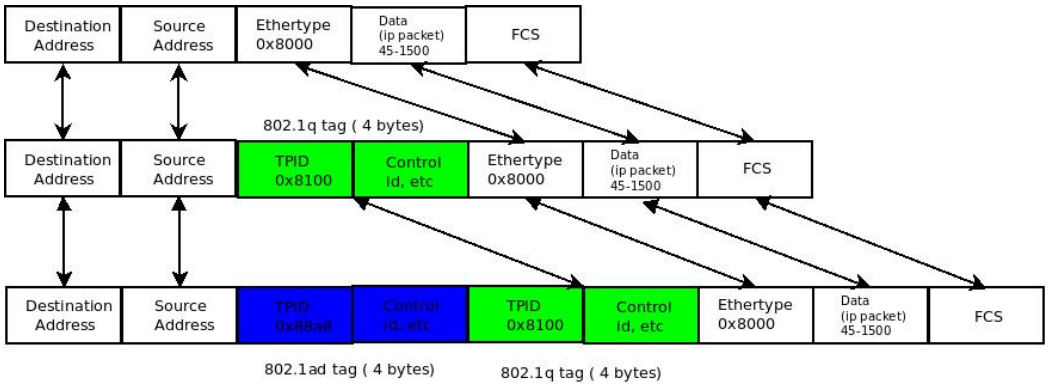


Figure 2.2: En visuel repræsentation af hvordan QinQ virker.

Det indre VLAN tag indsættes efter kilde adressen i mac-rammen inden pakken sendes afsted fra boligforeningens switch. Dernæst tilføjes det ydre tag når pakken rammer hovedswitchen inden pakken sendes ind til Cirques core netværk. VLAN id'et for det ydre tag er 0x88a8 for at kunne adskille dem.

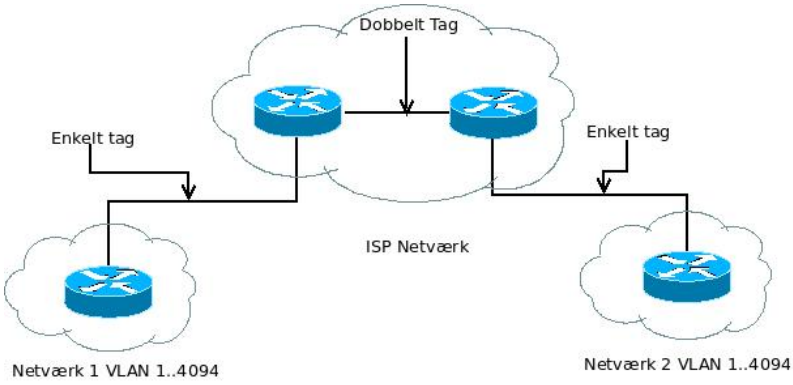


Figure 2.3: En visuel repræsentation af hvordan QinQ virker.

Figuren viser et eksempel på QinQ hvor netværk 1 sender en pakke med et VLAN tag op til internet service provideren (ISP) hvorefter den internt sendes med dobbelt tag indtil den pakkes ud til et enkelt tag inden pakken ankommer til modtageren i netværk 2.

2.4 Traffic shaping

2.4.1 Indledning

For at kunne beskrive hvad traffic shaping er det først nødvendigt at foreklare hvorfor der er et behov for traffic shaping i et større netværk. Trafik i den generelle forstand, dette kunne være på en motorvej, trafik igennem et computernetværk eller kunder igennem et supermarkeds kasselinje og hvis man forestiller sig at trafikken bevæger sig uniformt vil trafikken, i alle eksempler, løbe igennem jævnt og stabilt. Dette er at mængden af trafikken ikke overgår den maksimale behandlingsrate af trafikken. Som man kan forestille sig er det kun den færreste form for trafik der har en uniform bevægelse over tid, men der findes typer af trafik hvor dette er et krav. Sådan et jævnt eksempel kan være telefon trafik i et computer netværk, der har behov for præcis 64 kbps, der består af at sende en 8 bit pakke hver μsec [AST10, Section 5.4]. Det er dog de færreste trafik eksempler der har jævn trafik, faktisk kan stort set alle trafik systemer komme i en situation hvor trafikken kommer i ryk. Dette forårsager kø, og kan ses meget visuelt med trafikpropper på motorvejen og kø i kasselinjen. Her kommer traffic shaping så ind i billedet. Traffic shaping er en metode, hvorpå man kan behandle trafik der til tider overgår kapaciteten af en flaskehals. I kasselinje eksemplet kan dette betyde at åbne en kasse til, hvis køen bliver for lang, men i motorvejs og netværks eksemplerne er det typisk ikke muligt at forstørre gennemgangsevnen, og man må derfor tænke i andre baner. Traffic shaping er derfor en metode der kan behandle overflow situationer, sørge for udjævning af traffik m.m. I de følgende afsnit vil der blive set mere på måden de forskellige metoder traffic shaping kan implementeres på, fordele og ulemper med disse, samt se på yderligere detaljer indenfor netværks traffic shaping.

2.4.2 Shaping vs policing

Traffic policing er en anden form for traffic management end traffic shaping. De bruges begge til at udjævne trafikken der opstår i et datanetværk, men de har dog nogle forskellige måder at gøre dette på. Traffic shaping bruger forskellige metoder til at forsinke trafik således at trafik flowet bliver mere jævnt [AST10, Section 5.4]. Dette kan gøres ved hjælp af f.eks. leaky bucket eller token bucket som bliver forklaret nedenfor. Traffic policing er en overvågnings police af et netværk, der derfor overvåger et givent netværk og først tager aktion når en fastsat situation opstår. En traffic policy vil derfor sende al trafik igennem en flaskehals indtil peak raten (den maksimale mængde data der kan komme igennem på en gang) er opnået. Når der bliver forsøgt at sende data over peak

raten, smider en traffic policy metode typisk denne data væk [KD11], hvorimod en traffic shaper typisk vil sætte den overflødne trafik til at have en lavere prioritet. Forskellen mellem de to metoder er derfor at traffic policing typisk smider alt overfløden trafik væk, hvorimod traffic shaping forsinker traffikken. Traffic policing er derfor typisk ikke en specielt funktionel løsning for almindelig data trafik, da man nemt kan få meget pakketab, men traffic policing kan være meget nyttig til f.eks. at sikre et netværk mod Denial of Service (DoS) angreb, da alle data fra en enkelt kilde der overstiger peak raten vil blive smidt væk. Cisco bruger en control-plane policy i deres Catalyst 6500 switche mod netop DoS der er et godt eksempel på traffic policing [Cis05]. Nedenfor ses et eksempel på opførslerne af policing og shaping i forbindelse med trafik der overgår peak raten i et netværk. Traffic shaping bruger i figuren tilfælde en token bucket metode [Som beskrevet i afsnit 2.4.3.2].

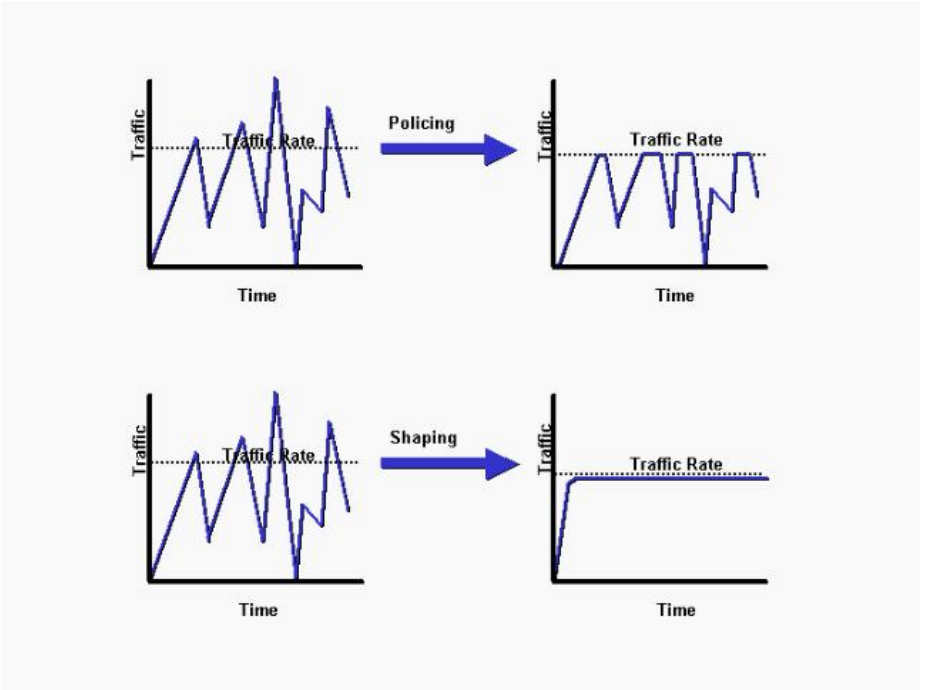


Figure 2.4: En visuel representation af en shaper policy's opførsel kontra en traffic shapers[mcm]. Her ses det tydeligt at ved policing bliver alt trafik over grænsen blot skåret væk, og fører til pakketab mens ved traffic shaping er trafikken fordelt mere jævnt. Dette giver naturligvis en generel forsinkelse for den shapede trafik, men vil ikke opleve pakketab.

2.4.3 Shaping metoder

2.4.3.1 Leaky bucket

For at forstå leaky bucket algoritmen skal man forestille sig en spand, hvori der er et hul hvor vand kan løbe ud med en konstant hastighed, ligemeget hvor meget vand der hældes i spanden. Hvis spanden er fyldt vil det overskydende vand løbe ud over siderne og gå tabt. Den konstante hastighed R hvormed vandet løber ud af spanden udgør bitraten i bytes per sekund hvormed en pakke kan sendes. Leaky bucket kan derfor bruges til traffic shaping, da der skal være plads til mere vand i spanden for at sende en pakke i netværket. Spandens kapacitet betegnes B. Den maksimale forsinkelse der kan opstå for en pakke er derfor B/R. Hvis spanden er fuld når en pakke ankommer bliver den enten kasseret eller sat i en kø indtil der er mindre vand i spanden. I situationer hvor en Leaky bucket smider overskydende pakker væk vil den ikke være så effektiv til traffic shaping, men vil være mere funktionel som en traffic policy [Se afsnit 2.4.2]. Leaky bucket begrænser flow på længere sigt, men tillader korte ryk i pakketrafikken op til en vis grænse.

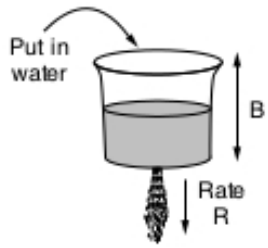


Figure 2.5: En visuel repræsentation af hvordan leaky bucket fungerer.

2.4.3.2 Token bucket

En algoritme der ligner leaky bucket men dog er lidt forskellig er token bucket. Her skal man forestille sig en spand uden hul hvori der løber vand. Spanden har en kapcitet på B og for at sende pakker skal man tage vand ud af spanden i stedet [AST10, Section 5.4]. Vandet i spanden udgør det for tokens som kan bestå af en størrelse på et par bytes eller en fastlagt pakkestørrelse. For at sende et givent antal pakker skal spanden have nok tokens lagret for at pakkerne kan sendes. Et link med kapaciteten C bits per sekund har en token bucket tilknyttet

og token bucketen har, som før nævnt, kapaciteten B tokens. Der genereres R tokens per sekund når spanden ikke er fuld. Linket kan kun sende en pakke med størrelsen L hvis der er L tokens tilgængelige fra spanden. Hvis der opstår et ryk hvor der sendes et antal pakker af størrelsen L og spanden er fuld når dette sker vil man kunne sende $k = \frac{B/L}{1-R/C}$ det antages at B er et multiplum af L. Efter de k pakker er sendt vil linket sende pakker med hastigheden R.

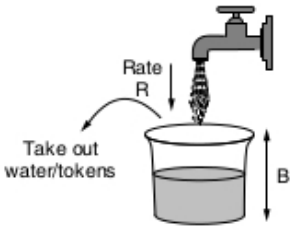


Figure 2.6: En visuel repræsentation af hvordan token bucket fungerer.

2.4.3.3 Yderligere shaping metoder

Traffic shaping har mange applikationer der er nyttige, og nogle endda nødvendige for at kunne opbygge et tilfredsstillende netværk af en større grad. Traffic shaping kan bruges som en implementation af en bucket til en specifik ip eller port, således at en bruger på den ip/port kan modtage den hastighed brugeren bør have. Man kan også bruge shaping til at begrænse mængden af data der bliver brugt af en bestemt applikation (f.eks. BitTorrent) for at sikre QoS for andre applikationer. Dette har en konsekvens i forhold til internet neutralitet debatten [GA06] hvor sådanne restriktioner på brugeres enkelte applikationer betegnes som data diskrimination og har været forslået ulovliggjort i USA. Derudover kan traffic shaping også bruges til at sikre de forskellige former for QoS ved hjælp f.eks. af prioritering i buckets [Se afsnit 2.2.3]. Sidst men ikke mindst bruges traffic shaping til at håndtere en mængde brugeres hastigheder, der derved sikrer at brugerene ikke får højere hastigheder end de betaler for, samt forhindrer sortseerer på netværket.

Hvis en bruger betaler for en 20mb download er det hensigtsmæssigt for en internet service provider (ISP) at brugeren ikke modtager højere hastighed end dette, for at spare belastning på det totale netværk. Dette gøres ved at implementere en form for bucket for hver enkelt bruger der betaler til netværket med

en peak rate på brugerens højeste hastighed. Denne form for traffic shaping er den vores projekt beskæftiger sig med.

2.5 Database

2.5.1 Behovet for databaser

Inden vi går igang med at beskrive opbygningen af en database ved hjælp af normalformer og hvordan man opbygger et program omkring en database vil vi først forklare hvorfor en database er nødvendig for et program af denne type. Dette gøres ved at besvare følgende spørgsmål:

- Hvorfor overhovedet have en database?
- Hvorfor bruge tid og besværet på at opbygge en database når man alligevel skal opbygge en datastruktur til at bruge internt i programmet?

Det primære svar vil være at adskildelsen af data fra selve programmet vil være at foretrække hvis man skulle havne i den situation at noget går galt og programmet, eller den server programmet ligger på, går ned. Hvis man har valgt at gemme alt ens data i en datastruktur i ens program, ville alt dataen da være gået tabt. Hvis man istedet vælger at gemme alt data i en database, der ligger på en ekstern server vil man blot kunne genstarte programmet og alt vil være tilbage til normalen.

Derudover kan en database sikre at man nemt kan spørge, søge og indsætte i sin data, uden behov for iterationer eller på anden måde gå unødvendig data igennem hver gang. Derudover vil en database langt bedre kunne håndtere de store datamængder da den skalerer bedre end langt de fleste datastrukturer.

2.5.2 Database Normalformer

For at gøre kontrollen af kundernes hastigheder lettere vil en switchdatabase blive tilknyttet traffic shaper løsningen. Da der er behov for at databasen skal køre effektivt og præcist er det nødvendigt at udføre normalisering på tabellerne. Dette forhindrer desuden redundans i data. [Dat03] Vi vil i dette afsnit beskrive og definere de forskellige relevante normalformer.

2.5.2.1 Første normalform

Hvis en tabel skal være på første normalform skal følgende krav være opfyldt:

- En relation R er på første normalform, hvis det for enhver attribut A i R gælder, at enhver tupel har en og kun en værdi for A.
- Der må ikke være kolonner der gentager sig.

Et eksempel på brud af første normalform kan være en kolonne for farver der indeholder attributten rød,grøn. En løsning vil være:

id	farve
1	rød
2	grøn

Et andet eksempel på et brud af første normalform er en tabel indeholdende studerende der tager kurser på et universitet:

s_id	s_name	kursus
401	Adam	Fysik
401	Adam	Matematik
402	Alex	Biologi
403	Bob	Biologi

Denne tabel bryder med første normalform da der er gentagelser af navnet Adam og kurset Biology. Måden hvorpå dette løses vil være at splitte tabellen op i 2 tabeller hvor den ene indeholder de studerendes id og navn og den anden indeholder kurser samt hvilke studerende der er på kurset.

s_id	s_navn
401	Adam
402	Alex
403	Bob

k_id	studerende_id	kursus
10	401	Fysik
11	401	Matematik
12	402	Biologi
12	403	Biologi

2.5.2.2 Anden normalform

For anden normalform gælder der at:

- En relation r er på anden normalform hvis den er på første normalform
- Der findes ikke en attribut A som er funktionelt afhængig af en del af primærnøglen

Et eksempel på brud af anden normalform er en relation som matrikel(vej,nr,postnr,ejernavn,bynavn
med primærnøglen (vej,nr,postnr) hvor den funktionelle afhængighed postnr->bynavn bryder med anden normalform. Dette skyldes at et bynavn der hører til et postnummer gentages en gang for hver matrikel indenfor samme postnummer. En løsning til dette problem ville være at separere postnr -> bynavn i en seperat tabel. Et andet eksempel på en tabel der bryder anden normalform er:

Kunde_tabel				
kunde_id	kunde_navn	ordre_id	ordre_navn	salgsnr
101	Alice	10	ordre1	salg1
101	Alice	11	ordre2	salg2
102	Bob	12	ordre3	salg3
103	Chris	13	ordre4	salg4

I denne tabel udgør kunde_id og ordre_id primærnøglen og tabellen er på første normalform. Dog brydes anden normalform da kunde_navn kun er afhængigt af kunde_id og ordre_navn kun er afhængigt af ordre_id. Desuden er der ingen sammenhæng mellem salgsnr og kunde_navn. For at opnå anden normalform splittes tabellen op i tre seperate tabeller:

kunde	
kunde_id	kunde_navn
101	Alice
102	Bob
103	Chris

ordre	
ordre_id	ordre_navn
10	ordre1
11	ordre2
12	ordre3
13	ordre4

salg		
kunde_id	ordre_id	salg
101	10	salg1
101	11	salg2
102	12	salg3
103	13	salg4

2.5.2.3 Tredje normalform

For tredje normalform gælder der at:

- En relation r er på tredje normalform hvis den er på anden normalform.
- Alle attributter afhænger af primærnøglen og ikke 2 attributter Z og A hvor Z-> A ikke indgår i primærnøglen.

Et eksempel på brud af tredje normalform vil være en relation som matrikel(matrikelnr,vej, nr, postnr, ejernavn,bynavn) med primærnøgle matrikelnr, hvor der igen gælder en funktionel afhængighed postnr-> bynavn. Problemet opstår da oplysningen om, at et givet bynavn der hører til et postnr bliver gentaget en gang for hver matrikel inden for samme postnr. For at løse dette splittes tabellen op i to tabeller hvor den ene indeholder matrikel(matrikelnr,ejernavn,postnr,) hvor primærnøglen er matrikelnr. Den anden indeholder så adresse(postnr, vej,nr,bynavn) med postnr som primærnøgle. Derved afhænger alle attributter af primærnøglerne og tredje normalform opnåes.

Et andet lignende eksempel er denne tabel:

Studerende					
s_id	s_navn	fødselsdato	vejnavn	by	postnr

For at opnå tredje normalform skal denne splittes op på samme måde som matrikel tabellen.

studerende			
s_id	s_navn	fødselsdato	postnr

adresse		
postnr	vejnavn	by

2.5.2.4 Boyce-Codd normalform

Boyce-Codd normalformen minder meget om tredje normalform men er mere restrektiv. Hvis en relation skal være på Boyce-Codd normalform skal der gælde at enhver funktionel afhængighed Z -> A skal Z være entydig i relationen. Et eksempel på en relation som bryder Boyce-Codd er:

Forelæsere		
kursus	dag	forelæser
programmering	mandag	alice
matematik	onsdag	bob
fysik	fredag	chris

I den ovenstående tabel findes den funktionelle afhængighed forelæser -> kursus men forelæser er ikke den eneste kandidatnøgle og derfor brydes Boyce-Codd. En løsning på dette vil være at splitte tabellen op i 2.

Forelæser_ekspertise	
kursus	forelæser
programmering	alice
matematik	bob
fysik	chris

Forelæser_ekspertise		
kursus	dag	forelæser
programmering	mandag	alice
matematik	onsdag	bob
fysik	fredag	chris

Relationen forelæser-> kursus er dermed på Boyce-Codd normalform.

2.5.2.5 Fjerde normalform

For fjerde normalform gælder der at

- Hvis der for enhver ikke triviel flerværdiafhængighed X-»Y gælder at X er entydig i relationen R

Et brud på fjerde normalform kan være en relation som Kursusplan(kursus,lærer,tekst) der til at starte med har en tupel (k1,l1,l2,l3,t1,t2,t3) vil når den kommer på første normalform blive splittet op til 9 tupler. I denne relation vil der gælde de ikke trivielle flerværdiafhængigheder kursus -» lærer og kursus -» tekst. Da kursus ikke er entydig bryder relationen kursusplan fjerde normalform. Løsningen vil være at splitte kursusplan tabellen op i 4 tabeller som følger:

kursusplan		
kursus_id	lærer_id	tekst_id

kursus	
kursus_id	kursus_navn

lærer	
lærer_id	lærer_navn

tekst	
tekst_id	tekst_navn

De nye tabeller opfylder dermed kravene for fjerde normalform.

Chapter 3

Analyse

3.1 Indledning

Vores analyse kapitel vil beskæftige sig i dybden med den eksisterende traffic shaper og beskrive de forskellige erstatninger Cirque har overvejet som deres nye traffic shaper. Derudover vil vi også se på yderligere aspekter af vores synkroniserings program imellem den nye traffic shaper, og den eksisterende faktureringsdatabase via vores egen udviklede switchdatabase. Synkroniseringsmodulet vi selv udvikler skal kunne kontaktes via en http service, således at Cirque's teknikere og supportere kan fejlsøge på enkeltstående brugeres forbindelser og manuelt styre de forskellige aspekter af synkroniseringen. Derfor vil vores programmerings struktur se således ud:

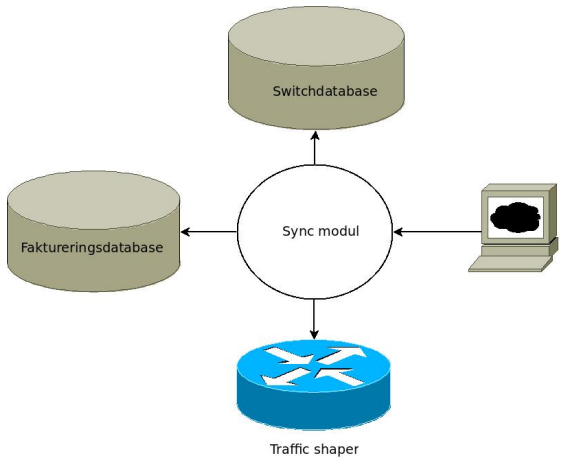


Figure 3.1: Diagrammet her ligner meget figuren i vores State of the art indledning [Figur 2.1], men har fået tilføjet muligheden for ekstern kommunikation igennem en http client.

I dette kapitel vil vi ligeledes nærstudere de forskellige relevante design patterns til vores struktur således at vi kan ankomme til vores design afsnit.

3.2 Eksisterende traffic shaping løsning

Cirque har på nuværende tidspunkt en linux baseret traffic shaper der primært er af eget design.

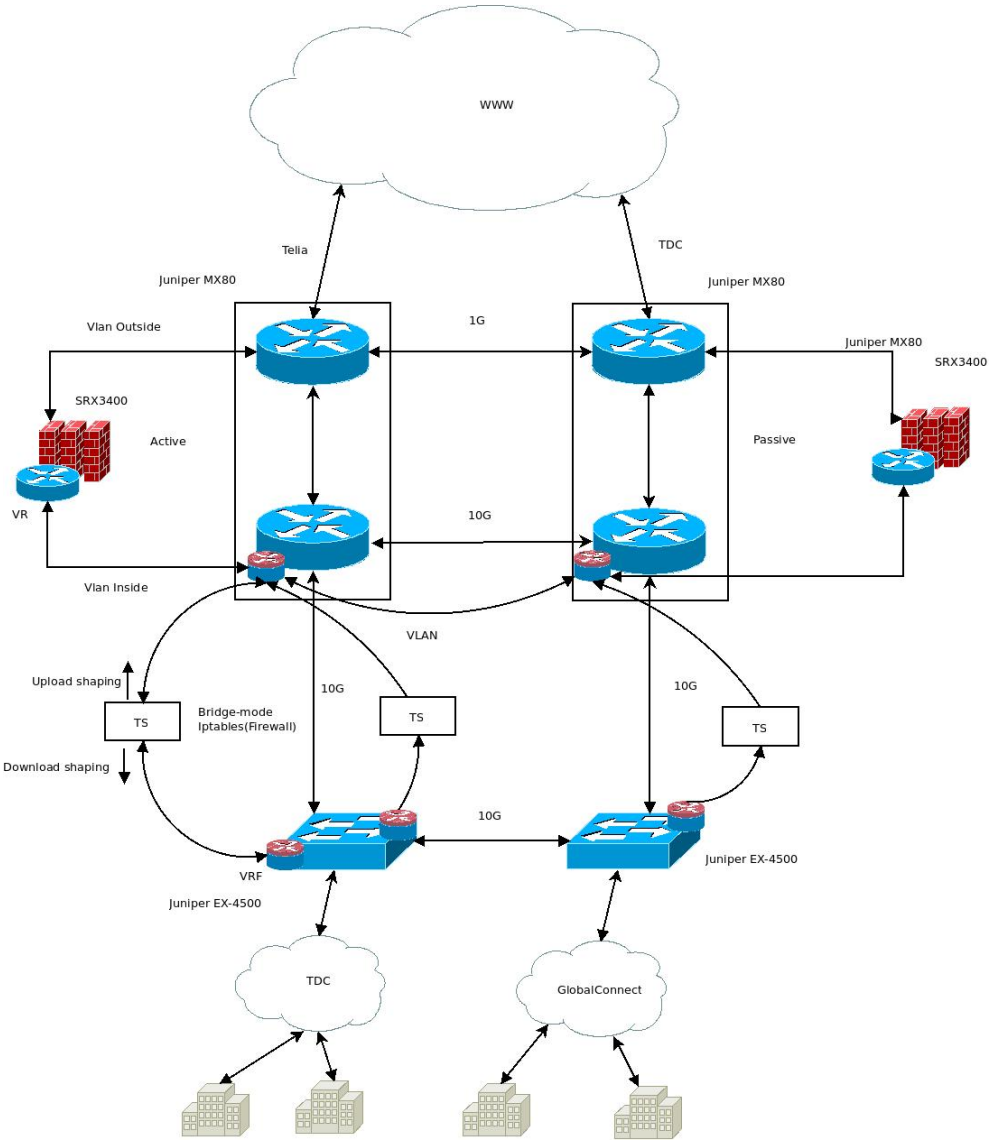


Figure 3.2: En visuel repræsentation af Cirques nuværende core netværk. Figuren viser nederst boligforeningerne der er koblet op via TDC eller globalconnect til Cirque's interne netværk. Inden datapakkerne bliver sendt ud på internettet rammer de trafficshaperne der dernæst sender dem op til de virtuelle firewalls der bagefter sender dem videre ud på internettet. Venstre del af netværket er aktivt og højre del er passivt og bliver sat i drift hvis venstre skulle fejle. VR er en virtuel router og en VRF er en Virtuel Router Firewall

Traffic shaper systemet består på nuværende tidspunkt af 3 linux servere der hver håndterer sin egen forudbestemte kundegruppe. Disse servere fungerer ved at have en bridge mellem to netværkskort, en til at håndtere alt indgående, en alt udgående traffik for denne servers kundegruppe. Når en datapakke bliver modtaget for et af disse netværkskort bliver pakken sendt op til CPU'en for at blive traffic shapet, når dette er sket bliver den sendt tilbage til samme netværkskort, der derefter sender den videre til det andet netværkskort og derfra videre i systemet. Når en pakke bliver sendt til CPU'en bliver der derfor sendt mere data med end der kunne være nødvendigt, hvilket skaber problemer med at gemme al dataen i et tilstrækkeligt lavt cache layer for at forhindre at bruge for lang tid på at swappe imellem de forskellige cache layers. For at undgå swapping imellem kerner i CPU'en har serveren en speciel schedulering der arbejder på en pakke per kerne og ikke swapper til en ny pakke før denne er behandlet færdig. Optimalt ville netværkskortet kun sende til/fra IP adressen samt størrelsen på pakken, da andre informationer ikke er nødvendige for traffic shaping. Netværkskortet skal derfor huske de resterende oplysninger i en seperat cache og sende den fulde pakke videre når der kommer svar fra CPU'en. Med den nuværende løsning har Cirque regnet sig frem til at deres servere hver især kan håndtere 500mb traffik uden at brugere oplever pakketab eller for høj latency. Ønsket om at skifte til et mere fremtidssikret system kommer derfor af at deres brugere får så høje hastigheder at det ikke vil være rentabelt at tilføje flere servere.

3.3 Forskellige Traffic shaper løsninger

Cirque har i forbindelse med deres indkøb af nye traffic shapere undersøgt flere forskellige løsninger. Dette afsnit vil beskæftige sig med disse løsninger, gennemgå deres fordele og ulemper, i forhold til det formål og de kriterier Cirque har opstillet i forbindelse med processen.

Cirques formål med udskiftningen af deres traffic shapere er at finde et fremtidsikret system således at deres system kan følge med stigningen i internethastigheder og stadig flere kunder. Der er på baggrund af dette opstillet følgende successkriterier:

- Skalerbarhed - Løsningen skal forholdsvis simpelt kunne forstørres for at skabe plads til flere kunder og øget båndbredde
- Kunne håndtere eksisterende kundebases behov
- Minimal tid på drift
- Stabilitet - høj oppetid

Successkriterierne er derfor forholdsvis simple, og har umiddelbart ingen krav omkring platform, teknologi eller andre begræsninger [Den oprindelige formålsbeskrivelse kan ses i bilag A].

3.3.1 Udskiftning til 2x Juniper MX480

Denne løsning omhandler at udskifte alt eksisterende server udstyr til to Juniper MX480 servere. Disse servere bliver solgt og leveret af et eksternt firma ved navn IPNett der ligeledes leverer serverne med et API til styring af kunder på disse servere. Denne løsning kan nemt opgraderes da løsningen kører på udskiftelige kort istedet for at man skal tilføje og skifte server bokse når en opgradering er nødvendig. Samtidig er det en løsning der allerede er gennemtestet på op til 30.000 brugere i Sverige, hvilket mindsker nødvendigheden for at teste på systemet. Denne løsning kræver dog at den eksisterende netværks core skal omlægges, samt at flere fiber linjer skal omlægges for at denne løsning kan fungere.

3.3.2 Udskiftning til 2x Juniper MX10

Ved at vælge at udskifte til Juniper MX10 fremfor nuværende løsning fåes mange af samme fordele som ved udskiftning til Juniper MX480. De er begge udbudt af IPNett og er derfor allerede testet til brug af 30.000 kunder, samtidig har denne løsning også det selvsamme API som ovennævnte løsning og har derfor mange af de samme løsninger. Dog er forskellen at denne løsning kan fungere adskilt fra den eksisterende netværks core og en omlægning af denne er derfor ikke nødvendig. Samtidig er den væsentlig billigere end ovenstående løsning.

3.3.3 Udvidelse af eksisterende Linux traffic shaper

Et andet forslag vil være at beholde den nuværende traffic shaper løsning [Som beskrevet i afsnit 3.2], og blot udvide på den som nødvendigt. Den kræver derfor ingen væsentlig omlægning i netværksstruktur, ej heller at skifte noget internt betjeningsværktøj. Med denne løsning vil detnaturligvis være billigt og hurtigt at tilfredsstille det nuværende behov. Dog har har den problemer i forhold til skalerbarhed, kapacitet per boks samt at den er dyr i drift. Udover dette vil det ikke vides hvor mange flere servere der skal til for at tilfredsstille et fremtidigt behov, hvilket gør det til en meget lidt fremtidssikret løsning.

3.3.4 Ny selvudviklet løsning

Ved at selv udvikle sin egen løsning kan denne formes som ønsket, og man kan derfor tilgodese eventuelle unikke behov enkelte grupper af kunder måtte have. Dog vil denne løsning sandsynligvis tage lang tid uden garanti om fuldførelse og pris.

3.3.5 3. part udviklet Linux server med specialbyggede netkort

En 3. parts udviklet løsning åbner ligeledes op for muligheder om at specialdesigne løsningen specifikt til Cirque, således at nødvendigheden for indordning af andre systemer bliver minimal. Derudover giver det også mulighed for at købe support hvis noget går galt, hvilket ikke kan lade sig gøre hvis man selv udvikler løsningen. De specialbyggede netkort der bliver nævnt i titlen er nogle kort der er designet til kun at sende størrelsen på en modtaget datapakke og afsender adressen til CPUen for udregning af den hastighed pakken bør sendes igennem med, altså om afsenderens bucket er fuld eller ej [Se afsnit 2.4.3.2]. Hvis man opsætter et normalt netkort til at fungere som en del af en traffic shaper løsning vil netkortet sende hele datapakken op til CPU'en når denne bliver modtaget. Dette skaber derfor et hurtigt et stort lagringsbehov for CPU'en og kan derfor føre til at pakkerne tager for lang tid igennem CPU'en, hvilket fører til højere ping tider. Ydermere kan mængden af pakker der venter på at blive beregnet blive så stor at layer 1 og eventuelt layer 2 cachen bliver opbrugt, hvilket vil føre til ekstremt store forsinkelser da det nu er nødvendigt at swappe til RAMene hvilket vil foresage mærkbart store forsinkelser. Der er naturligvis forskellige metoder til at undgå dette, f.eks. kan man overprovisionere CPU kraften og cachen, således at dette aldrig kan blive et problem, men en anden løsning kan være at bruge disse speciel designede netkort der som nævnt tidligere sender langt færre data afsted til CPU'en, og derfor drastisk mindsker sandsynligheden for at RAM swapping finder sted. Denne løsning vil langt hen af vejen kunne fremtidssikre løsningen, men har samtidig stor usikkerhed, da det er uvist omkring en sådan løsning overhovedet kan lade sig gøre, samtidig med at både tiden for udvikling og implementering samt prisen er ikke fastlagt.

3.4 Programmeringsanalyse

Til den nye traffic shaper har det været et ønske fra Cirque at have et API til at håndtere oprettelse/ændringer/nedlæggelser af kunder, samt have et system

der kan håndtere en database over switche og leveringsadresser. Systemet skal samtidig kunne automatisk mindst en gang i døgnet køre en algoritme der opretter kunder efter de abonnementer der findes i faktureringsdatabasen, og sætte deres hastigheder efter det abonnement de har.

Derudover er det vores eget ønske at lave et skalerbart RESTful API der kommunikerer med JSON således at det vil være simpelt at lave et GUI til vores program. Programmet skal derfor både kunne snakke med den nuværende kunde database, indsætte i en ny seperat database der håndterer alle switch oplysninger [Se afsnit 3.5 for mere information om denne] og håndtere kald til den nye traffic shaper igennem det API denne måtte gøre tilgængeligt.

Vi ønsker at gøre programmet asynkront så vidt muligt, da vi ønsker at vores API ikke skal fryse eller blokere ved flere klienter, eller ved mange requests. Dette kan gøres på forskellige måder. Faktisk har de fleste programmeringssprog muligheder for asynkron programmering, men dog er der nogle der har en nemmere tilgang til asynkron programmering således at man kan undgå mange af de faldgrupper der er at finde ved asynkron programmering. Inden vi går yderligere i dybden med de forskellige muligheder vil vi først forklare omkring asynkron programmering.

3.4.1 Asynkronitet

For at kunne forklare asynkronitet vil vi først beskrive synkronitet og derved adskillelsen til asynkronitet.

3.4.1.1 Synkron programmering

Synkronitet bliver typisk betragtet som en simpel og nemt overskuelig måde at programmere på, da alt sker sekventielt og hver opgave i program flowet blokerer således at en opgave bliver lavet fuldstændig færdig før den næste kan begynde.

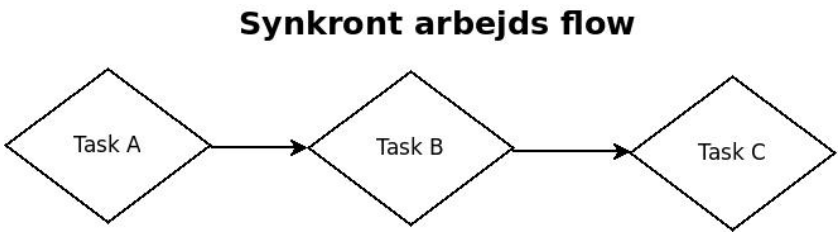


Figure 3.3: Her ses et simpelt flow chart der beskriver en synkron process

Dette vil i mange situationer være tilstrækkeligt, men hvis et program skal kunne håndtere input fra flere forskellige kilder, og især hvis det skal håndtere brugerinput, vil denne model skabe unødvendig ventetid og derved vil programmet kunne optimeres til at køre hurtigere. Hvis man som nævnt implementerer et UI program, vil programmet sidde og vente på opgaver fra brugeren, og vil håndtere dem så snart brugeren f.eks. trykker på en knap. Sådanne programmer fungerer typisk ved hjælp af en event handler der lytter på alle events brugeren kan lave, og sætter disse events i en kø til programmet, i den rækkefølge brugeren trykker på knapperne. Hvis køen derfor bliver for lang kan en bruger reelt vente lang tid på ting der burde være hurtigt behandlet fordi køen ligger og blokerer.

3.4.1.2 Asynkron programmering

Asynkronitet kan tilbyde en løsning til ovenstående problem. Asynkronitet går nemlig ud på at ens program aldrig blokerer, og derfor altid vil være i stand til at modtage events samtidig. Hvis programmet modtager et event der ellers ville ende op med at blokere, vil et asynkront program kunne skifte midlertidigt til et andet event således at alle events venter mindst muligt. Asynkronitet fungerer derfor godt til programmer der har flere aspekter i sig, som f.eks. et program med et GUI, et API der kan have op til flere samtidige klienter eller et program hvor man ønsker flere beregninger sideløbende, til f.eks. algoritmer. Derudover kan et asynkront program med fordel gøre brug af parallelitet for at åbne op for endnu mindre ventetid, da man derved får mulighed for at eksekvere opgaver sideløbende på flere forskellige kerner.

Parallelitet er når et program kan eksekvere flere dele sideløbende, hvilket er meget tiltalende når langt de fleste computere i dag har to eller flere kerner. Ved parallelitet kan der dog opstå en række problematikker hvis f.eks. forskellige dele af programmet forsøger at tilgå data eller services samtidig, for hvad bliver

så behandlet først, og hvad sker der med ens data? Dette vil blive vist med nedenstående eksempel:

Parallel eksempel. Hvis man forestiller sig, for enkelthedens skyld, en enkelt kerne der har to tråde kørende og begge tråde samtidig bliver sat til at sætte en variabel, indeholdende et tal, til en højere end den nuværende værdi. Lad os forestille os at tallet som trådene forsøger at forøge med et, er 4 og har navnet v. Da vi sætter to tråde til at forøge forventer vi derfor at tallet til slut er 6, men det vil vise sig at slut tallet vil kunne blive 5. Flowet for de to tråde bliver vist i nedenstående trin. Imellem hvert trin vil der forekomme et swap i scheduleringen, der gør at den nuværende tråd ryger tilbage i køen. Dette sker ved f.eks. round-robin schedulering [And00].

1. Begge tråde bliver sat i kø til schedulering.
2. Tråd 1 henter variablen. Variablen v er her 4
3. Tråd 2 henter variablen. Variablen v er her 4
4. Tråd 1 forøger variablen med 1. Den lokale variabel hos tråd 1 er her 5. Variablen v er dog stadig 4.
5. Tråd 2 forøger variablen med 1. Den lokale variabel hos tråd 2 er her 5. Variablen v er dog stadig 4.
6. Tråd 1 overskriver variablen v med sin lokale variabel. Variablen v er nu 5.
7. Tråd 2 overskriver variablen v med sin lokale variabel. Variablen v er nu 5.

Denne problematik kaldes en race condition og kan undgås ved at bruge mutexes eller flags hvilket populært kaldes for blocking [And00].

Dog hvis man indfører blocking til at håndtere alle situationer for race conditions kan man risikere det der hedder deadlock [And00]. En deadlock er når en, eller flere, processorer forsøger at tilgå noget data der er låst med et flag der aldrig vil blive frigivet. Dette gør at alle processorer der forsøger at tilgå og arbejde med værdien der er låst af dette flag vil stå i scheduleringskøen uendeligt og kan før eller side, føre til et systemnedbrud.

3.4.2 Forskellige programmeringsmuligheder

Java har mulighed for asynkron programmering både ved at man selv kan udpege hvilke dele af ens kode der skal fungere asynkront ved hjælp af await og async funktioner. På den måde kan man nemt lave en løsning der ikke er fuldt asynkron hvis man ønsker det, men man skal selv sørge for thread safety [And00]. Thread safety er en betegnelse der dækker over deadlocks og race conditions, altså om disse kan eksistere. Et asynkront program kan være i følgende thread safety tilstande:

- *Ikke thread safe* race conditions kan forefinde
- *delvis thread safe* Alt delt data mellem tråde er beskyttet af race conditions og tråde kan tilgå egen data. Deadlocks kan forekomme.
- *thread safe* Data er fuldstændig thread safe når ingen race conditions kan forekomme men at alle tråde kan tilgå al data samtidig.

I Java kan man derfor selv styre hvilken grad af asynkronitet og thread safety man ønsker, men samtidig er systemet meget besværligt at gå til og det er derfor nemt at overse en race condition. Der findes derfor et plugin til Java der hedder Akka. Akka introducerer en klasse type der hedder Actors[Gup12, Chapter 3]. Actors består af en receive funktion og en mailbox. Alle actors har sin egen thread safe tråd og behandler beskeder til sig selv som first in, first out i forhold til actorens mailbox. Hvis en actor modtager en besked løber den sin receive funktion igennem for at bestemme hvad den skal gøre med denne. Hvis den ikke ved hvad den skal gøre ryger beskeden i systemets dead letters kasse. For yderligere asynkronitet introducerer Akka også Promises og Futures[Gup12, Chapter 3] hvilket fungerer som asynkrone svar. Altså at man beder om noget, og at man så på et tidspunkt får et svar, men at man kan begynde at arbejde på det allerede inden man får det fulde svar.

Scala er et både et funktions og objekt orienteret sprog på samme tid, der eksekverer på Java Virtual Machine. Det kan derfor køre på alle platforme som Java kan køre på. Det er både funktions og objekt orienteret på den måde at det blandt andet har pattern matching, tail recursion, immutability og lazy evaluation hvilket er ting typiske for funktionsorienterede sprog.

- *Pattern matching*: Man kan matche hvilken slags data mod cases med en første-match politik. Hvis data matcher den første case eksekveres denne.

- *Tail recursion*: Et rekursivt kald til en metode der ikke opbruger plads på memorystacken.
- *Immutability*: Gør at et objekt ikke kan ændres efter det er oprettet.
- *Lazy evaluation*: evalueringen af et expression foregår først når der er brug for en værdi.

Samtidig bygger det på Java, så man kan programmere rent objekt orienteret hvis man ønsker dette og det gør op med noget af det der bliver ment som design fejl i Java sproget såsom:

- Type erasure
- Checked exceptions (try/catch)
- Non-unified type system [Mal, Se følgende artikel for info om scalas type system]

Da scala bruger Java's VM virker alle Java plugins også til Scala, og man kan derfor få de samme fordele i forhold til asynkronitet som Akka tilbyder.

C# fungerer på mange måder på samme måde som Java uden Akka når det gælder asynkronitet, og da vi samtidig ønsker en løsning der er så lidt properitær som muligt er dette ikke umiddelbart en oplagt løsning.

3.5 Databasebehov

Den tidligere traffic shaper løsning indeholder en tabel hvor switchene og hvilke kunder der er tilknyttet en given port er dokumenteret. Derudover indeholder den ip-adresser, hastigheder, og hvilket VLAN en kunde er tilknyttet. Denne tabel er integreret i faktureringsdatabasen hvilket vil besværliggøre en eventuel udskiftning af systemet i fremtiden. Der vil derfor være et behov for at have en ekstern switchdatabase som kan synkronisere med faktureringsdatabasen. Desuden vil der sættes fokus på at den nye løsning indeholder et api der hurtigt kan omstilles hvis en ny faktureringsløsning vælges. Desuden er det besværligt at oprette nye foreninger med den gamle løsning hvilket der også vil blive sat fokus på ved brug af api'et. Dette vil lette arbejdet med databasen i fremtiden både ved oprettelse og fjernelse af foreninger.

Den nye løsning vil benytte sig af QinQ [Som beskrevet i afsnit 2.3.2]. Derfor vil det være nødvendigt for den nye database at indeholde en tabel med outer VLAN for hver forening.

3.6 Design patterns

Denne del af analysen vil beskæftige sig designpatterns der kan have relevans for programmeringen af API'et.

3.6.1 Model View Controller

Model View Controller(MVC) er et design pattern der går ud på at dele ens applikation op i 3 komponenter. En model der indeholder programmets logik og data, et view der beder modellen om data som det så fremviser til brugeren og en controller der sørger for at sammenkoblingen mellem viewet og modellen.

Først og fremmest vil den udviklede løsning basere sig på model og controlleren hvor der senere kan tilkobles et view. Modellens logik vil basere sig på Data Access Objects som vil blive beskrevet i følgende afsnit.

3.6.2 Data Access Object Pattern

Data Access Object Pattern(DAO)[Wie13] er et design pattern der bruges til at oprette en abstract metode til at manipulere vedholdende data. Dette kan være en database, XML filer, webservices m.m. Hvilken type service som DAO'et skal oprette forbindelse til sættes ved oprettelse af metoden. Så hvis DAO'et skal oprette forbindelse til f.eks. en Oracle database vil det indeholde de specifikke datatyper for forbindelsen. Fra applikationens synspunkt er det ligemeget hvilken type data der oprettes forbindelse til. Et typisk DAO indeholder metoder der understøtter create, read, update og delete(CRUD). DAO'et er en del af data access laget hvor man typisk isolerer al logikken der skal interagere med vedholdende data i en eller flere DAO klasser. Dette gør at man hurtigt kan skifte til en anden type data uden at skulle ændre for meget i data logikken. Dette design pattern er derfor oplagt at bruge i vores implementation da vi skal udveksle data med en ekstern database og en traffic shaping webservice.

3.7 Opsummering

Vi har i dette kapitel fået beskrevet de nødvendige behov for både vores switch-database og hvilke muligheder for programmeringssprog og teknologier der vil være nærliggende for os. Det er derfor tydeligt i forhold til vores ønske omkring asynkronitet at vi bør vælge et programmeringssprog, der yder gode muligheder for parallelitet og asynkronitet, da vi ikke ønsker situationer hvor vi kan overse deadlocks eller race conditions. For at opsummere de forskellige parallel termer:

Race condition er når to tråde forsøger at arbejde på samme data og derved opnår et resultat der ikke var efter hensigten [Se eksempel 3.4.1.2]. Dette bliver typisk løst ved blocking.

Deadlock er når en tråd eller flere tråde forsøger at tilgå noget blokeret data, der aldrig vil blive frit.

Derudover har vi beskrevet hvilke design patterns der vil være nærliggende at benytte og problematikken med den nuværende traffic shaper og forklaret de forskellige løsning som Cirque har overvejet i forhold til deres nye traffic shaper.

Chapter 4

Design

4.1 Indledning

Dette design kapitel vil omhandle hvilke valg vi og Cirque foretager sig, samt begrundelserne for disse. I dette afsnit vil vi derfor komme ind på strukturen af vores switchdatabase, samt at vi vil gå yderligere i dybden med vores programstruktur, hvilket bringer os til nedenstående figur:

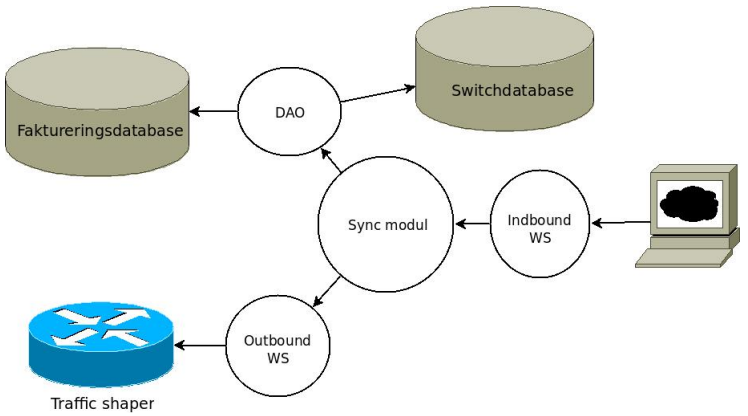


Figure 4.1: Her ses en udpensling af vores program struktur, hvor der er blevet tilføjet et Data Access Object (DAO) der bruges til at kommunikere med databaserne, samt en inbound og outbound Web Service (WS) til at håndtere requests fra brugere, og til at sende anmodninger til traffic shaperen.

Præcis hvordan vores DAO og WS bør se ud vil vi gå mere i detaljer med senere i dette kapitel, i vores afsnit omkring design patterns. Derudover vil vi beskrive vores design valg med hensyn til vores programmeringsprojekt, der blandt andet vil indeholde detaljer omkring vores valg af programmeringssprog.

4.2 Valg af traffic shaper løsning

Den løsning Cirque har valgt at bruge som deres nye traffic shaper, er ikke en vi har haft indflydelse på, men dette afsnit vil præsentere den nye løsning og forsøge at kommentere på den i forhold til de alternativer der tidligere er blevet præsenteret [Se afsnit 3.3].

Cirque har valgt at udskifte deres traffic shaper udstyr til 2x Juniper MX10 traffic shapere. Disse er udbudt af et firma ved navn IPNett, som Cirque har valgt at købe udstyret og supporten af. Denne Juniper traffic shaper løsning er af IPNett testet til brug af op til 30.000 brugere. Dette er langt større end Cirque's nuværende kundebase der ligger under 10.000 brugere på landsplan. Løsning er derfor yderst fremtidssikret, og udover at den kan håndtere langt flere brugere end Cirque har på nuværende tidspunkt er løsningen skalerbar ved at man kan tilføre flere kort. IPNett har desuden programmeret et SOAP API

til traffic shaperen således at man eksternt kan oprette/nedlægge kunder samt konfigurere og overvåge serveren [Se bilag C for API dokumentation].

Løsningen lever derfor op til Cirques krav omkring stabilitet, skalerbarhed og håndtering af kundebasen. Samtidig tilføjer den også en masse ekstra i form af SOAP API'et, som Cirque ikke direkte har haft som krav, men generelt vil lette at arbejde på og drive traffic shaperen.

4.3 Database Design

Den nye databasestruktur vil bygge videre på den gamle samt sørge for at de forskellige normalformer [Som beskrevet i afsnit 2.5.2] er opfyldt.

Tabellerne for den nye database vil oprettes som følger:

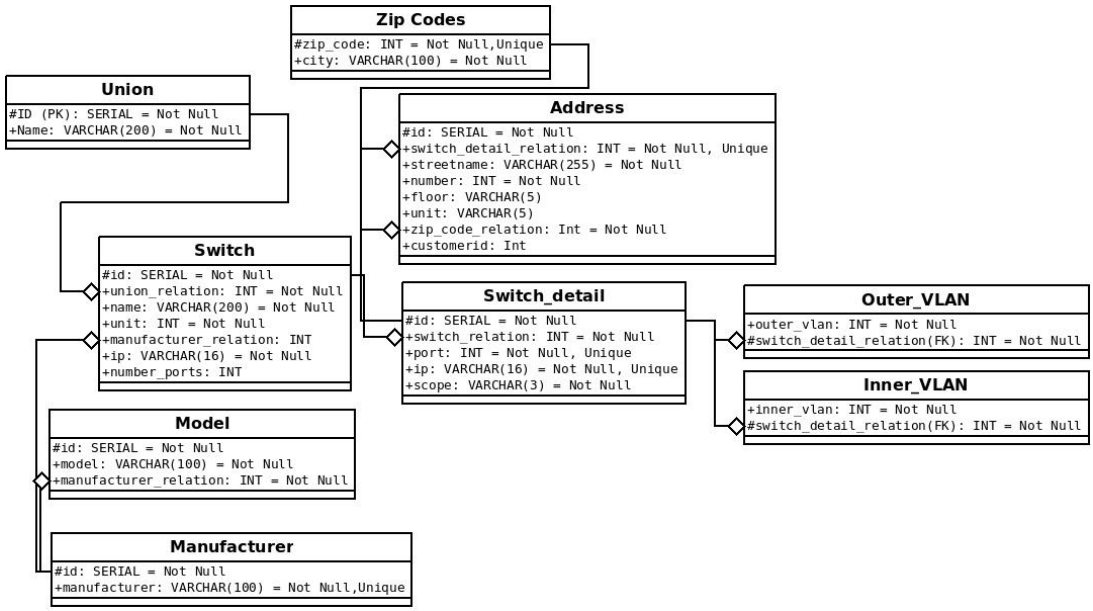


Figure 4.2: Database tabellerne for switchdatabasen.

På figuren indikerer # at attributten er en primary key. Pilene imellem tabellerne indikerer at attributten er en foreign key. Det ses at databasetabellerne udgør en

træstruktur med en union(boligforening) som roden. Et union id fungerer som foreign key til en tilknyttet switch. En switch har så tilknyttet alle de forskellige adresser på lejligheder der er koblet op til de forskellige porte. Hver switch har via switch_detail en oversigt over de forskellige ip adresser og ydre VLAN, samt hvilke indre VLAN der er for hver port på switchen. Derudover er der også tilknyttet hvilken fabrikant og model switchen har. En kunde kan tilknyttes en given adresse i address tabellen ved hjælp af deres kundenummer som det ses ud fra customer tabellen. Da en adresse har en switch_detail_relation er kunden tilknyttet porten på switchen der er tilknyttet adressen.

Da alle attributterne i enhver tupel kun har én værdi er tabellerne på første normalform.

Union, Inner_VLAN, Outer_VLAN, zip_code og manufacturer tabellerne er på 2. normalform da de kun har en attribut for deres primærnøgle. I de andre tabeller er der fuld funktionel afhængighed af primærnøglen så disse er også på 2. normalform.

Derudover er tabellerne på 3. normalform da der ikke opstår transitive afhængigheder mellem primærnøglerne og relationerne.

Da model, manufacturer, Outer_VLAN, Inner_VLAN og zip_code har hver deres tabel i databasen opstår der ingen flerværdiafhængigheder og tabellerne er derfor på fjerde normalform.

4.4 Design patterns

4.4.1 MVC Design

API'et bliver delt op i en model, et view og en controller [Som nævnt i afsnit 3.6]. Modellen vil indeholde de forskellige DAO's beskrevet i efterfølgende afsnit. Controlleren vil indeholde de forskellige kald der gør brug af DAO's, hvilket i vores tilfælde vil være http requests fra en browser. Da løsningen fokuserer på et API vil den ikke i første omgang indeholde et view.

Nedenfor ses et overblik over vores model-view-controller struktur og som nævnt er programmet kun et API og det returnerer derfor tilbage direkte igennem controlleren. Hvis man ønsker at programmet ligeledes skal kunne levere HTML sider, eller andre former for et view, på et senere tidspunkt kan dette også lade sig gøre.

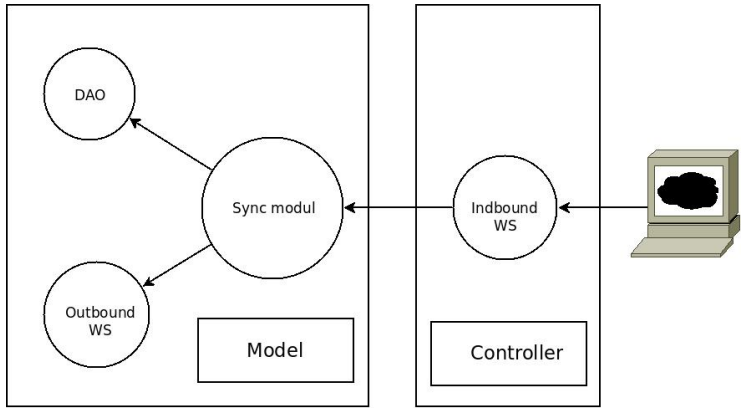


Figure 4.3: Et groft overblik over vores design pattern.

4.4.2 DAO

Som beskrevet tidligere i analysen af DAO vil DAO implementationen indeholde CRUD funktionalitet for switchdatabasen [Se afsnit 3.6.2]. Derudover vil der også være funktionalitet for at kommunikere med traffic sharperen igennem DAO's.

De forskellige DAO's der skal oprettes i API'et til switchdatbasen for slutbrugeren bliver:

- createUnion: opretter en ny boligforening.
- createCustomer: opretter en kunde på en given port på en switch.
- createSwitch: opretter en switch tilknyttet en forening.
- search: søger efter et kundenummer, en adresse eller en ipadresse.

For traffic shaping webservice API'et skal følgende DAO's oprettes:

- getSubscriber: finder en kunde baseret på outer og inner VLAN.
- addSubscriber: tilføjer en kunde baseret på outer og inner VLAN.
- deleteSubscriber: fjerner en kunde baseret på outer og inner VLAN.

Andre DAO's der skal oprettes for at give mulighed for at oprette/ændre i switchdatabasen.

- createSwitchDetail: opretter en switchdetail tilknyttet en given switch.
- createAddress: opretter en adresse tilknyttet en given switch.
- createZipCode: opretter et postnr med en tilknyttet by.
- createSwitchModel: opretter modellen for en switch.
- createSwitchManufacturer: opretter en producent for en given switch.
- createVlan: opretter et vlan til en given adresse.

Ligeledes skal der bruges følgende DAO operationer for at kunne hente de nødvendige data fra faktureringsdatabasen:

- getUsers: henter alle brugere, deres hastigheder og adresser.
- getUser: henter en enkelt bruger på baggrund af et givent kundenummer.
- getUsersByUnion: henter brugere i en forening på baggrund af en given forening id.
- getUnionIds: henter alle forenings id'er.

4.5 Programmeringsdesign

4.5.1 Indledning

Som beskrevet i vores analyse af programmeringen [Se afsnit 3.4] ønsker vi at implementere et program, der så vidt muligt kan fungere asynkront, således at programmet kan bruges af flere klienter samtidig, uden at de skal opleve uhensigtsmæssige forsinkelser. På baggrund af dette har vi besluttet at vores program skal skrives i Scala. Scala har i forvejen en masse værktøjer indbygget, således at vi nemt kan implementere programmet efter vores ønsker. Nedenfor vil vi beskrive de værktøjer, vi har tænkt os at gøre brug af i vores program.

4.5.2 Scala

Til vores projekt har vi valgt at programmere rent i Scala. Som nævnt i afsnittet omkring programmeringsanalyse [Se afsnit 3.4.2], er Scala et program der lægger op til at man programmerer både funktionsorienteret og objekt orienteret, og at man kan have begge metoder stående side om side [Ale13, Preface]. Dog vil vi primært programmere så funktionsorienteret som muligt ved at bruge Scala's indbyggede funktioner til dette. Scala forsøger desuden at gøre op med Java's try/catch kultur ved at introducere alternative former for error handling, dette er primært smeltet sammen med Scala's ønske om at gøre op med Java's null casting. Scala har som alternativ til disse to introduceret en Option. En Scala Option i scala dokumentationen defineret som

```
Sealed Abstract Class Option[+A]
```

Det er derfor en abstrakt klasse der kan instantieres som en Option af en hvilken som helst underklasse af A, hvor A værende en hvilken som helst klasse i Scala's typesystem.

En Option har to mulige resultater, En Option[+A] kan enten være Some(+A) altså have en værdi af typen +A, eller også kan det have værdien None gældende for ingen værdi. Options kan bruges til parametre i metoder der enten skal eller ikke skal være der, og man så ved hjælp af pattern matching få metoden til at opføre sig forskelligt alt efter om paramteren er til stede eller ej.

Til Scala har vi tænkt os at bruge et framework der hedder Play. Play åbner op for muligheder for at lave et asynkront RESTful API og bygger tungt ovenpå Akka's actor struktur [Som beskrevet i afsnit 3.4.2]. Play vil desuden blive nærmere uddybet nedenfor.

4.5.3 Play framework

Play frameworket er et open source framework skrevet i Java og Scala der følger MVC design pattern. Play tilbyder et HTTP-centrisk API med metoder hvor man nemt kan udvikle stateless webservices og RESTful API'er.

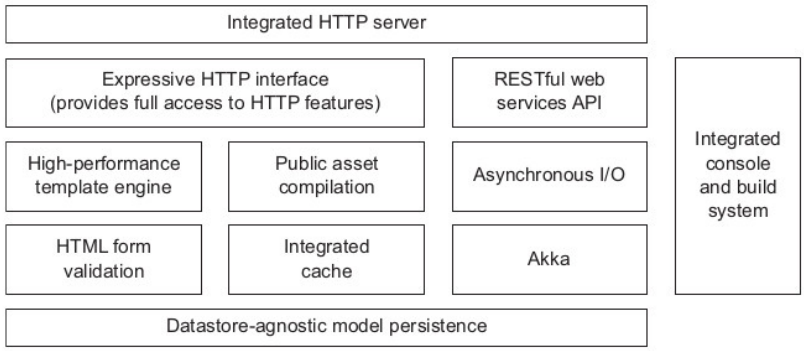


Figure 4.4: Play Frameworket[BC13]

Som det ses på ovenstående figur tilbyder play frameworket den nødvendige funktionalitet, for at kunne udvikle et platformsuafhængigt API til styring af switchdatabasen og traffic shaperen. Play indeholder en bl.a. en indbygget webserver der står for håndtering af client requests, for at sende dem videre til selve play applikationen og for at sende et svar tilbage til klienten. Play's webserver er JBoss Netty der indeholder asynkron I/O hvilket gør det muligt for serveren at behandle flere requests i samme tråd.

Play benytter sig, som før nævnt, af MVC pattern og frameworket er opdelt som følger:

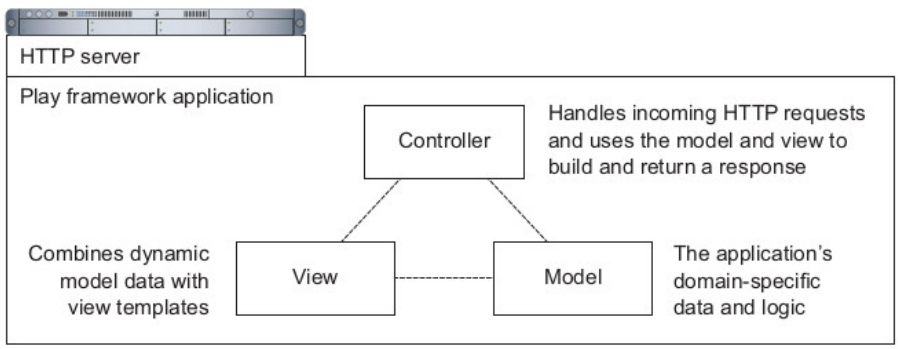


Figure 4.5: Play Framework MVC [BC13]

Det ses at der er en løs kobling imellem komponenterne. Viewet håndterer generering af HTML templates der sendes til klientens browser sammen med Controlleren der står for modtagelsen af HTTP requests. Al logikken for applikationen ligger i modellen og det er også her man vil placere eventuelle DAO's. Modellen har intet kendskab til viewet eller controlleren men disse kan hente data fra modellen.

Da der skal opbygges et API til en switchdatabase vil det være nærliggende at tage udgangspunkt i database-centrisk design hvor man opretter data modellen og finder typer, deres attributter og forhold imellem dem. Når dette er på plads kan der opbygges DAO's for at tilgå disse objekter og manipulere databasetabellerne via HTTP requests igennem controllerne.

En Controller i Play består af en Scala klasse hvori der implementeres et interface der er defineret i routes konfigurationsfilen. Når der modtages en HTTP request vil routeren i Play sørge for at udføre den controller handling der er defineret i routes konfigurationsfilen og giver det tilsvarende svar genereret i HTML i samarbejde med Viewet. Dog behøver svaret ikke være HTML men kan også være XML, JSON eller ren tekst.

Som det ses ud fra figur 4.4 er Akka integreret i Play. Dette giver mulighed for at benytte actors. Actors er objekter der indkapsler tilstand og adfærd. Hver actor kører på en enkelt tråd i systemet og har en intern postkasse hvor de udveksler beskeder om hvilken opgaver der skal udføres. Sådanne udvekslinger kan både ske ved at en forælder actor fortæller en eller flere af sine børn hvad de skal gøre, eller andre actors kan sende beskeder over en eventstream. Flere dele af programmet kan derfor arbejde sammen uden at vide at hinanden eksisterer. Kommunikation imellem actors foregår asynkront, som beskrevet i afsnittet om programmeringsmuligheder [Afsnit 3.4.2], og actor systemet står selv for håndtering af hvilken rækkefølge beskederne håndteres. En analogi for actors er at forestille sig en gruppe hvori der uddelegeres opgaver og underopgaver til hver person. En actor kan skabe flere child actors og agere som vejleder for disse. Hvis en child actor fejler sender den en besked til vejlederen der bagefter kan udføre en passende handling som f.eks. at genstarte child-actoren. Alle actors opbygger herved et hieraki hvor der kan eskaleres opad når fejl opstår og en passende handling kan udføres for at udbedre fejlen. Actors er derfor en meget simpel måde at implementere et asynkront program på, da de enkelte actors automatisk har deres egen tråd, og man derfor ikke behøver at bekymre sig om mange af de faldgruber der findes ved paralellitet [Se afsnit 3.4.1].

4.5.4 Actors

Som det kort har været berørt tidligere er en Akka actor en type af en Scala klasse. Der er helt præcist tale om et trait man forlænger sin klasse med, og dette trait implementerer så størstedelen af den funktionalitet der er nødvendige for alle actors. Nemlig at klassen bliver oprettet som sin egen seperate tråd, og at den bliver oprettet med en postkasse til at tage imod beskeder. Enhver klasse af typen actor skal have en metode til at håndtere de beskeder actoren modtager i sin postkasse. Enhver actor skal derfor have en implementation af en modtage metode der på baggrund af en pattern matching håndterer de forskellige beskeder denne actor kan modtage. Hvis man forsøger at sende en besked til en actor som den ikke er i stand til at håndtere vil en exception blive smidt, og dette skal man derfor kunne håndtere.

En actor har en predefineret livscyklus, der er beskrevet af overvågningsactoren, samt en preRestart, postStop, preRestart og postRestart metode der kan defineres i actor klassen [Se figur 4.6].

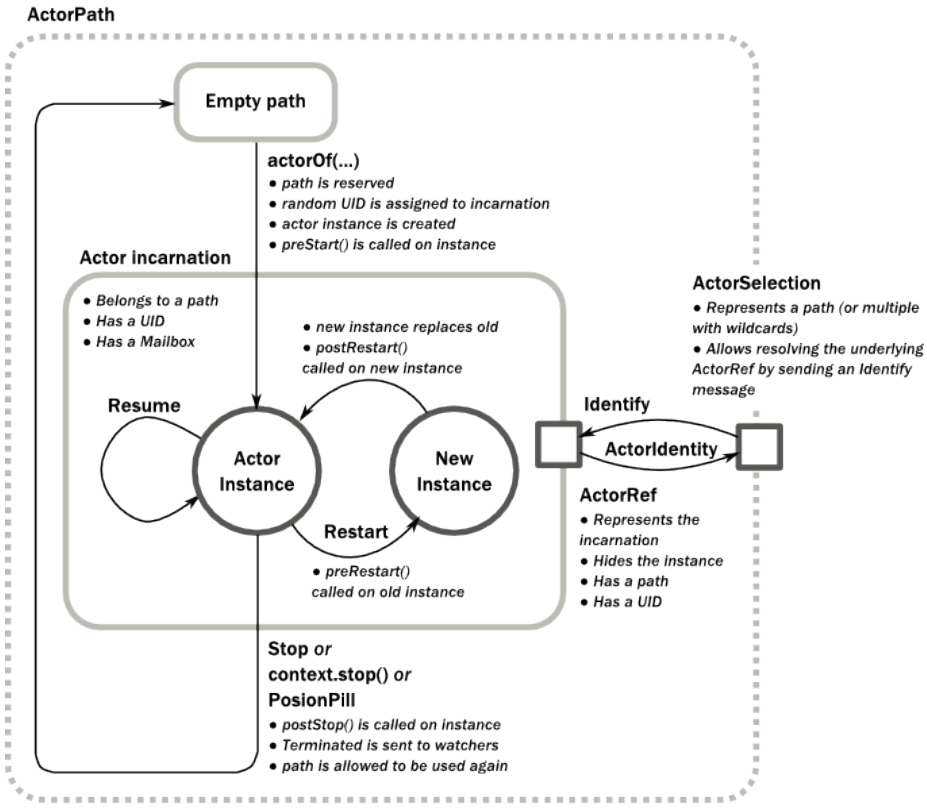


Figure 4.6: Her ses en actors livscyklus [akk, Actor Lifecycle]

En actor bliver startet ved hjælp af en actorOf(...) funktion og det første der derfor sker er at actorens preStart metode bliver kaldt. Når en actor er oprettet kan den restartes eller stoppes fra det sted den er blevet oprettet. En actor kan dog nå at foretage operationer efter den har fået besked på f.eks. restart i preRestart metoden.

En typisk actor vil derfor se således ud:

```
class testActor extends Actor {
        override def preStart() = {...}
        override def postStop() = {...}
        override def preRestart() = {...}
```

```
        override def postRestart() = {...}

        def receive = {
         case 1 => ...
         case 2 => ...
         case 3 => ...
         case other => ...
        }
}
```

Denne actor modtager derfor tal og laver matching på den i receive metoden. Hvis tallet er enten 1,2 eller 3 er der defineret en special case for dem. For alle andre tal, eller andre typer for den sags skyld vil actoren altid gøre det samme.

Da alle actors har deres egen tråd og postkasse er de bygget til at fungere asynkront i forhold til andre actors. Der er dog et problem, hvis man ønsker at få et svar fra en actor, vil man derfor typisk være nødt til at blokere, mens man venter på svar. Hvis man dog ønsker asynkronitet er det typisk ikke en særlig funktionel metode, at vente på et svar ved at blokere. På grund af dette introducerer Akka en abstrakt klasse der hedder en Future og deres superklasse et Promise. En future er et svar fra en ikke blokerende asynkron kilde der helt præcist betyder at man før eller siden vil modtage et svar. Futures kan have to resultater, et Success(A) hvor A er en værdi eller en Failure(E) hvor E er en exception. Fordelen ved en Future er at afsender af en besked ikke behøver at vente på svar, men kan istedet for fortsætte med at foretage andre opgaver. Når der så kommer svar til afsenderen kan den så håndtere den så hurtigt som muligt.

4.5.4.1 Actorsystem

Et Actorsystem er et en gruppering af actors der hænger sammen omkring at løse en mængde relaterede opgaver. Et sådan system har typisk også en politik for hvordan de forskellige actors bør reagere, hvis den skulle gå hen at dø, hvordan nedlukning og opstart af actors skal foregå. En sådan politik bliver typisk håndteret i en såkaldt overvågnings actor der sørger for at starte alle nødvendige actors ved opstart og sørger for at håndtere en eventuelt genstartnings strategi.

Et actorsystem har ligeledes en eventstream, hvor alle actors i systemet kan sende case classes til andre actors i systemet. Dette fungerer ved hjælp af en publish/subscribe metode, hvor en actor abonnerer på de case classes de ønsker at modtage. Andre actors kan derfor sende en case class til den actor uden

direkte at kalde til den. Dette giver mulighed for et mere løst koblet program, hvor hele systemet kun er kendt af overvågnings actoren.

4.5.5 Struktur

Størstedelen af strukturen af vores program er allerede præsenteret tidligere [Se figur 4.1], men dette afsnit vil præsentere en yderligere detaljeret struktur. For bedst muligt at beskrive vores actorstruktur, har vi valgt at lave et UML kommunikationsdiagram. Vi har valgt ikke at lave et klassediagram da den asynkrone struktur af vores actors gør det yderst besværligt at beskrive programmet overskueligt ved hjælp af et klassediagram.

Grundet størrelsen af vores kommunikationsdiagram er figuren at finde i bilag B

Der er to aktører til stede, nemlig en bruger af systemet der igennem en hjemmeside opretter og nedlægger kunder, her repræsenteret som en browser, og vores synkroniseringsmodul der med jævne mellemrum opretter og nedlægger alle ændringer i forhold til faktureringsdatabasen. Synkroniseringsmodulet fungerer ved først at spørge faktureringsdatabasen efter alle kunder og for hver af disse bliver switchdatabasen opdateret, således at nye kunder bliver oprettet, gamle bliver nedlagt osv. Til sidst bliver vores TSActor bedt om at oprette/opdatere eksisterende kunder og nedlægge gamle.

4.6 Opsummering

Dette afsnit har fremvist de forskellige designvalg der er truffet med hensyn til switchdatabasen, API design, programmeringssprog og hvilke design patterns der bruges. Valget af Play frameworket i kombination med Scala giver mulighed for at have flere klienter, der benytter systemet samtidigt og brug af DAO's gør det lettere at styre databasen programmeringsmæssigt. Derudover har vi ved en gennemgang af Play framework, kunne præcisere strukturen i vores program således at vi så nemt som muligt kan implementere programmet.

Chapter 5

Implementation

5.1 Indledning

I dette kapitel vil vi beskrive vores implementation af henholdsvis switchdatabasen, hvordan vi har overholdt vores design patterns og hvordan programmet så til slut ser ud. Derudover vil vi beskrive implementationen af vores forskellige DAO's, vores actorsystem og de individuelle actors. På baggrund af vores design valg har vi yderligere beskrevet vores program og opbygningen af dette, som ses i nedenstående figur:

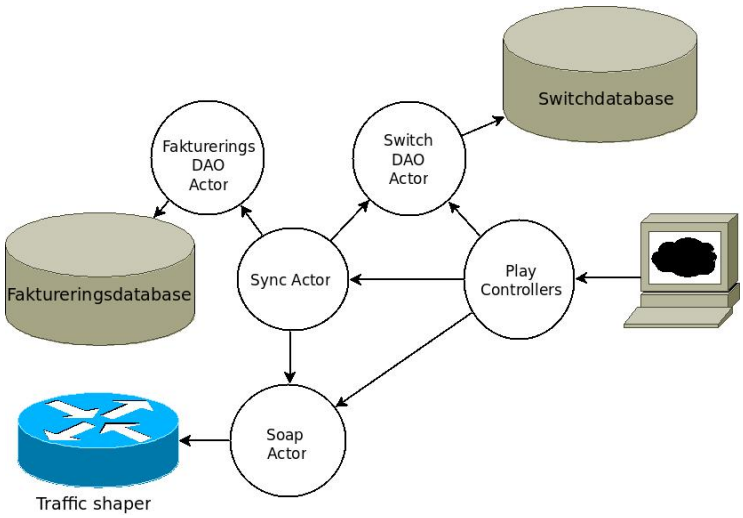


Figure 5.1: Her ses vores programstruktur inddelt i actors

Vi har derfor en række actors der har hver deres tråd [Som beskrevet i afsnit 4.5.3] . Ved at gøre dette får vi derfor muligheder for at f.eks hente i faktureringsdatabasen og switchdatabasen samtidig, og vi opnår derfor mulighed for stor grad af asynkronitet igennem vores program.

5.2 Databaseimplementation

Implementationen af switchdatabasen foregår på en POSTgreSQL 9.3 server der er uafhængig af traffic shaper serveren. Der er oprettet en metode i API'et der kan håndtere oprettelse af databasestrukturen via en selvudviklet SQL scala pakke der indeholder definitioner for de forskellige tabeller og søjle restriktioner. Derudover er de forskellige datatyper også inkluderet. Det skal dog nævnes at denne pakke ikke er komplet men den indeholder det nødvendige for at oprette de tabeller som er beskrevet i database design delen [Se figur 4.2]. Måden hvorpå API'et kommunikerer med databaseserveren er ved brug af Anorm, der er en del af Play frameworket, der kan kommunikere direkte ved brug af SQL statements. For at oprette switchdatabasen kaldes CreateDatabase der benytter sig af en createTable metode som modtager de forskellige parametre for hver tabel der ønskes oprettet. Et eksempel kan være at oprette en boligforening(Union):

```
createTable(
      Table(
        "Union",
        List(
          SQLColumn("id", sql.DataType.Serial,
          Some(SQLDefinitions.notNull),None
          ,Some(SQLDefinitions.autoIncrement),
          Some(SQLDefinitions.unique)),
          SQLColumn("name",sql.DataType.VarChar(200)
          ,Some(SQLDefinitions.notNull),None,None,
          Some(SQLDefinitions.unique))
        ),
        Some(Primary(List("id")))
      )
```

Her oprettes et table med et navn "Union" og de forskellige columns tilføjes med et kald til SQLColumn. En SQLcolumn indeholder felterne name, datatype, null, default, autoincrement, unique og comment.

- Name angiver hvilket navn den givne søjle har i tabellen.
- Datatype er hvilken SQL datatype søjlen indeholder.
- Null angiver om det er tilladt at søjlen kan indeholde null værdier.
- Default angiver om tabellen skal indeholde default værdier.
- Autoincrement adderer automatisk 1 til værdien i søjlen.
- Unique angiver at al data i søjlen skal være unikt.
- Comment tilføjer en kommentar til søjlen.

Alle variable undtagen name og datatype er Options hvor feltet enten har en værdi eller ingen værdi angivet ved None. De andre tabeller fra database diagrammet [Se diagrammet 4.2] oprettes på samme fremgangsmåde. SQL kaldet for createTable for en union ender med at være:

```
CREATE TABLE ts."union"
(
  id serial NOT NULL,
  name character varying(200) NOT NULL,
  CONSTRAINT union_pkey PRIMARY KEY (id),
  CONSTRAINT union_name_key UNIQUE (name)
)
WITH (
  OIDS=FALSE
);
ALTER TABLE ts."union"
  OWNER TO dbuser;
```

Parametrene i createTable metoden ændres til en streng med SQL syntax og sendes videre i SQL queries, efter en konvertering via Anorm, til databasen.

5.3 Implementation af program

5.3.1 Indledning

Dette afsnit vil først beskrive implementationen af vores design pattern, derunder vores forskellige DAO's, hvordan disse opererer med de forskellige databaser og systemet. Derefter vil vi beskrive vores actorsystem, og hvad dette indeholder.

5.3.2 Design patterns

I dette afsnit vil vi beskrive hvordan vi har overholdt model-view-controller i implementationen, samt hvordan vores data-access-objects er implementeret.

5.3.2.1 MVC

Model.
Implementationen følger MVC design patternet på den måde at der er oprettet en model hvori de forskellige DAO's er placeret. Dette bliver beskrevet nærmere i følgende DAO afsnit [Se afsnittet 5.3.2.2].

I enhver applikation der benytter Play frameworket findes der en application.conf fil, der indeholder indstillinger for applikationen. I vores implementation indeholder den opsætningen af forbindelserne til fakturerings og switchdatabasen samt login til API'et på traffic shaperne. Derudover definerer application.conf også hvilken fil der fungrerer som startfil ind i applicationen. I vores tilfælde er det en Global fil, der starter en vejleder actor samt modtager login indstillingerne fra application.conf. Disse indstillinger bruges så af vejleder actoren til at starte de services der opretter forbindelserne.

View.
Der er som sådan ikke defineret noget view til API'et men Play frameworket indeholder et predefineret view der f.eks. kan returnere fejlbeskeder fra Controller handlingerne.

Controller.
Controllerne i API'et benytter sig som beskrevet, af en routes konfigurations fil, der bestemmer hvilken handling der controllen skal udføre. Hovedfunktionerne såsom at oprette en database, tilføje og fjerne kunder og ændre hastigheder for givne kunder. Et eksempel på en route kunne være

```
POST    /createunion    controllers.MainController.createUnion
```

Som det ses fra koden kaldes controlles.TestController.createUnion via /createunion og dette vil svare til at kalde "servernavnet":9000/createunion.

Nedenstående figur viser hvordan et kald routes igennem Play API'et

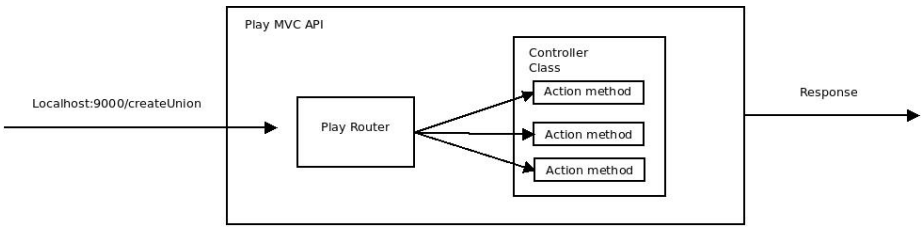


Figure 5.2: En visuel repræsentation af routes i Play frameworket virker.

Controlleren modtager JSON objekter der repræsenterer parametrene i det data der skal indsættes/oprettes. Hvis dette valideres korrekt lægges der en besked

op på eventstreamen der så vil håndteres af en actor der f.eks. opretter en given forening.

5.3.2.2 DAO

Fakturering.
Vores faktureringsdao er et simpel DAO der ikke kan indsætte noget i databasen, men kun hente informationer ud af den. Det er ligeledes der informationer omkring brugeres hastigheder, adresser og kundenumre hentes ud. Vores faktureringsdao har derfor følgende operationer:

- getUsers(): Henter alle aktive brugere i databasen.
- getUser(customerid: Int): forsøger at hente en bestemt bruger ud, hvis denne ikke findes bliver der returneret et tomt resultat.
- getUsersByUnion(unionid: Int): Henter alle aktive brugere for en forening.
- getUnionIds(): Henter alle forenings id'er.

Hvis man f.eks. ønsker at hente informationer omkring en bestemt kunde hentes dette ved følgende SQL query:

```
SELECT customer.customerid, address1, download, upload
FROM privat.customer, privat.vget_traffic_shaping
WHERE customer.customerid = vget_traffic_shaping.customerid
AND customer.customerid = {customerid}
```

Hvor customerid er kundenummeret for den ønskede kunde er givet i parameteren customerid.

Denne query returnerer et resultsæt bestående af en liste af rækker der bliver returneret, på baggrund af disse rækker opbygger vi en SubscriptionInfo for hver række. SubscriptionInfo er vores egen datastruktur bestående af et kundenummer, adresse og internet hastighed.

```
case class SubscriptionInfo(
customerId: Long,
address: String,
speed: Speed
)
```

hvor datastrukuren Speed er defineret som:

```
case class Speed(
download: Option[Int],
upload: Option[Int]
)
```

Alle operationer i vores faktureringsDAO, med undtagelse af getUnionIds, returnerer en liste af unikke SubstriptionInfo's. Denne datastruktur bliver i Scala annoteret som:

```
Set[SubscriptionInfo]
```

Herunder vil vi derfor beskrive de fire DAO operationer der er nødvendige for vores program:

getUsers henter som tidligere nævnt alle aktive brugere ud. Dette gøres ved følgende metode:

```
def getUsers: Set[SubscriptionInfo] = {
        log.info(s"received request on all user info")
        DB.withConnection(this.dbName) { implicit connection =>
                val query = SQL(
                """
                        SELECT customer.customerid,
                        address1,
                        download,
                        upload
                        FROM privat.customer,
                        privat.vget_traffic_shaping
                        WHERE 1=1
                        AND customer.customerid =
                        vget_traffic_shaping.customerid
                """
                )
                query().map{ row =>
                SubscriptionInfo(
                        row[Long]("customerid"),
                        row[String]("address1"),
                        Speed(
                                row[Option[Int]]("download"),
                                row[Option[Int]]("upload")
                        )
```

```
                )
                }.toSet
        }
}
```

Først åbnes en connection til databasen ved DB.withConnection, dernæst bliver vores SQL query defineret i værdien query, denne bliver nu eksekveret ved at sætte et sæt paranteser og bliver mappet over:

```
query().map
```

Et map itererer igennem alle medlemmer i en liste og modulerer hvert medlem på baggrund af den kode der er inde i map funktionen. I dette tilfælde modtager vi en liste af SQLRows og for hvert SQLRow i denne liste opretter vi istedet en SubscriptionInfo.

Metoderne getUsers og getUsersByUnion er ens med undtagelse af en lettere variation i SQL querierne, vi har derfor valgt ikke at vise dem. getUser returner ligeledes et Set[SubscriptionInfo], hvor der enten kan være ingen eller et medlem i sættet.

getUnionIds henter alle forenings id'er fra faktureringsdatabasen. Kildekoden til denne metode ser således ud:

```
def getUnionIds: Set[Long] = {
    log.info(s"Received request for all union ids")
    DB.withConnection(this.dbName) { implicit connection =>
      val query = SQL(
        """
          SELECT DISTINCT taeller
            FROM privat.forening
            ORDER BY taeller;
        """
      )
      query().map{ row =>
        row[Long]("taeller")
      }.toSet
    }
  }
```

Vi kan derfor se at denne metode ikke tager imod nogle parametre, og den returnerer et sæt af Long værdier. Ligesom med getUsers, getUser og getUserByUnion bliver en database forbindelse først givet ved DB.withConnection, derefter

bliver det SQL query der skal bruges defineret. Derefter bliver det eksekveret, mappet over og alle medlemmer i sættet bliver lavet om til et tal af typen Long.

Switch.
Vores Switch DAO står for interaktionen med switchdatabasen og har derfor følgende Scala traits.

```
trait SwitchDokDAO {
  //insert methods
  def insertUnion(union:Union): Long

  def insertSwitch(switch:Switch,
  unionRelation: Option[Long],
  manufacturerRelation: Option[Long]): Long

  def insertSwitchDetail(switchDetail: SwitchDetail,
  switchRelation: Option[Long]) : Long

  def insertAddress(address:SDAddress,
  switchDetailRelation: Option[Long],
  zipCodeRelation: Option[Long]) : Long

  def insertZipCode(zipCode:ZipCode) : Long

  def insertSModel(sModel: SModel,
  manufacturerRelation: Option[Long]) : Long

  def insertManufacturer(manufacturer:Manufacturer) : Long

  def insertCustomer(customer:Customer,
  switchDetailRelation: Option[Long]) : Long

  def insertVlan(vlan: Vlan,
  switchDetailRelation: Option[Long]) : Long

  def createDatabase(): List[Boolean]

  def selectFrom(input:Any) : List[(SwitchDetail)]

  def deleteCustomer(customerID:Int): Boolean

}
```

Traits bruges til at definere objekt typer i Scala ved at specificere en signatur for de supporterede metoder. En tilsvarende Java metode er Interfaces. I ovenstående kode ses de forskellige traits, med tilhørende datatyper, til at indsætte og søge i databasen. Derudover er der også mulighed for at fjerne en kunde fra databasen via deleteCustomer. Datatyperne er en forlængelse af den generelle datatype SDStructure. Denne type overskriver Scala's toString metode for nemt at kunne konvertere til SQL syntaxen brugt af Anorm. Hver datatype der forlænger SDStructure har sin egen toString definition til samme formål. Disse datatyper er implementeret i SwitchDok.scala [En oversigt over de forskelle datatypes findes i bilag D].

Som eksempel på en af metoderne der benytter et DAO ses indsættelse af en boligforening nedenfor. Denne er implementeret i SwitchDokDaoImpl:

```
def insertUnion(union:Union) = this.insertInto(Table("Union",
List(SQLColumn("name"))),union,
List.empty).getOrElse(0)
```

Metoden modtager en union af datatypen Union og denne indeholder et unionnavn der er en streng. Navnet indsættes i union databasetabellen via et kald til insertInto.

```
private def insertInto(table:Table,input:SDStructure,
relations:  List[Option[Long]]): Option[Long] = {
    val column= table.columns.map(_.name).mkString("(",",\n",")")
    val relationValues = if(relations != List.empty)
    relations.map(_.getOrElse(0)).mkString(", ", ", ", ")")
    else ""
    val values = input.toString + relationValues
    log.info(values)

    DB.withConnection(this.dbName){implicit connection =>
      val query = SQL(
        s"""
          INSERT INTO ts.${table.name} $column VALUES $values
        """
      )
      query.executeInsert()
    }

  }
```

Metoden modtager en tabel hvori data skal indsættes, data der består af en SDStructure og relationerne. Derefter konverteres det hele til en streng og der oprettes forbindelse til switchdatabasen og data indsættes.

For at søge efter kundedata i databasen er der oprettet en selectFrom metode der modtager input af typen Any d.v.s. data kan være en integer eller en streng.Nedenfor ses koden til selectFrom:

```
def selectFrom(input: Any): List[SwitchDetail] = {
 DB.withConnection(this.dbName){implicit connection =>

    log.info(s"is input $input an ip address:
     "+this.isIp(input.toString))
    log.info(s"input $input is of type ${input.getClass}")
    val column = input match {
      case int: Int => "a.customer = "
      case ip: String if(this.isIp(ip)) => "sd.ip LIKE "
      case address: String => "a.street_name LIKE "
    }
    val query = SQL(
      s"""
         SELECT *
         FROM ts.switch_detail sd
         LEFT JOIN ts.address a ON
         a.switch_detail_relation = sd.id
         LEFT JOIN ts.outer_vlan ov ON
         ov.switch_detail_relation = sd.id
         LEFT JOIN ts.inner_vlan iv ON
         iv.switch_detail_relation = sd.id
         LEFT JOIN ts.zip_code zc ON
         zc.zip_code = a.zip_code_relation
         WHERE 1=1
         AND ${column}{prepared};
         """
    )
      .on("prepared" -> this.prepareString(input))
    query().map{ row =>
    //(),
      SwitchDetail(
       row[Int]("port"),
        row[String]("ip"),row[String]("scope"),
        OuterVlan(row[Int]("outer_vlan")),
        InnerVlan(row[Int]("inner_vlan")),
        Some(row[Long]("id")),
```

```
        Some(SDAddress(
          row[String]("street_name"),
          row[Int]("number"),
          row[Option[String]]("floor"),
          row[Option[String]]("unit"),
          ZipCode(
            row[Int]("zip_code"),
            row[String]("city")),
          Some(row[Int]("customer"))))
      )
    }.toList

    }
  }
```

Efter input er givet forberedes det via en match case. Hvis input er en integer sendes det videre som et customerid. Hvis det er en streng checkes der på om strengen er en ip-adresse eller en adresse. Hvis det er en adresse fjernes al tegnsætning for at simplificere søgningen da adresser indsættes i databasen uden tegnsætning.

Udover alle insert metoderne og selectFrom metoden er der implementeret en DeleteCustomer metode der fjerner et kundenummer fra Databasen hvis kunden ikke længere har et abonnement.

Traffic shaper.

Vores traffic shaper DAO er et DAO, der kommunikerer med traffic shaperen. Denne er anderledes opbygget end vores to andre DAO, da den ikke snakker direkte med en database, men istedet kommunikerer over en SOAP forbindelse direkte til et traffic shaper API. Som tidligere nævnt skal vores traffic shaper DAO have muligheder for følgende operationer:

- getSubscriber: finder en kunde baseret på outer og inner VLAN.
- addSubscriber: tilføjer en kunde baseret op outer og inner VLAN.
- deleteSubscriber: fjerner en kunde baseret på outer og inner VLAN.

På baggrund af dette har vi lavet et trait, der fungerer som et interface til vores soapklient:

```
trait AbstractSoapClient {

  private val log = LoggerFactory
  .getLogger(classOf[AbstractSoapClient])

  def headers: Seq[(String, String)] = Seq(
  ("Content-Type", "text/xml; charset=UTF-8")
  )

  def host: String

  protected def prependedHost(relative: String): String =
   s"urn:$host#$relative"

  def wrap(xml: Elem): String

  def sendMessage(relative: String, xml: Elem):
   Future[Option[Elem]]

  def error(msg: String) = {
  log.error(s"SoapClient error: $msg")}
}
```

Traitet har en sekvens af de nødvendige headers der er de samme for alle operationer. Derudover har den en metode til at finde den korrekte soap-action url i form af prependedHost. PrependedHost tager den soap-action vi ønsker at finde og prepender host adressen på. Wrap er en metode der pakker den givne XML i form af parametrerne til det givne kald, ind som en SOAP pakke som serveren til modtage. Til sidst har vores trait en sendMessage metode der sender en besked over en oprettet soap forbindelse og giver resultatet tilbage som en Future[Option[Elem]] hvor Elem er Scala's XML datatype.

Da det SOAP API vi ønsker at kommunikere med kræver et brugernavn og adgangskode skaber vi yderligere et lag bestående af et trait ved navn AuthSoapClient. Denne forlænger vores trait beskrevet ovenfor og tilføjer nogle nye metoder:

```
  protected def username: String
  protected def password: String

  protected def authenticate(
  relative: String,
  contentLength: Int): WSRequestHolder = {
    val head = this.headers ++
```

```
    Seq(
            ("SOAPAction", this.prependedHost(relative)),
                ("Content-Length", contentLength.toString)
    )
    WS.url(this.host).withHeaders(head: _*)
    .withAuth(
    this.username,
    this.password,
    AuthScheme.BASIC)
  }
```

Det er derfor nu krævet at vi har to strenge, en indeholdende et brugernavn og en et kodeord. Derudover har vi nu fået en authenticate metode der forbereder alle headers nødvendige for SOAP kaldet. Metoden tager i mod en string som er navnet på kaldet og længden af det data der skal sendes med. Det første der bliver gjort er at forberede en sekvens af alle headers i værdien head, derefter bliver den sat på en ny url til den givne hostadresse fra vores AbstractSoapClient. Til sidst bruges metoden .withAuth hvor man bekræfter brugernavn og adgangskoden, samt hvilken krypteringsmetode af disse der skal bruges.

Vi er derfor nu klar til at implementere vores to traits. Dette gøres i klassen IPNettSoapClient:

```
class IPNettSoapClient(
val host: String,
val username: String,
val password: String) extends AuthSoapClient{
def wrap(xml: Elem): String = {
  val buffer = new StringBuilder
  buffer.append("<?xml version=\"1.0\"
  encoding=\"UTF-8\"
  standalone=\"no\"?>\n")
  .append("<soapenv:Envelope " +
    "xmlns:xsi=
    \"http://www.w3.org/2001/XMLSchema-instance\" " +
    "xmlns:xsd=
    \"http://www.w3.org/2001/XMLSchema\" " +
    "xmlns:soapenv=
    \"http://schemas.xmlsoap.org/soap/envelope/\" " +
    "xmlns:ipe=
    \"urn:"+this.host+"\">\n")
    .append("<soapenv:Header/>\n")
  .append("<soapenv:Body>\n")
  .append(xml.toString())
```

```
      .append("\n</soapenv:Body>\n")
      .append("</soapenv:Envelope>\n")
      .toString()
    }

    def sendMessage(
    relative: String,
    xml: Elem): Future[Option[Elem]] = {
      val content = wrap(xml)
      val url = this.authenticate(
      relative,
      content.getBytes().length
      )
        url.post(content).map{ response =>
        Some(response.xml)
      }
    }
}
```

Denne klasse implementerer de to metoder fra AbstractSoapClient der endnu ikke er implementeret, nemlig wrap og sendMessage. Wrap bygger en streng der er bestående af det XML der skal sendes som en besked til SOAP API'et. På baggrund af soap headeren som vi definerer i AuthSoapClient's authenticate metode og denne besked vil API'et svare. SendMessage modtager en soap-action der skal foretages i form af strengen relative, og den XML vi ønsker at bruge som input parametre. SendMessage opretter nu en forbindelse til vores host og sender beskeden afsted. Svaret vi får tilbage sikrer vi os at det er af typen Future[Option[Elem]] ved at mappe over den Future vi får tilbage og gøre følgende:

```
.map{ response =>
    Some(response.xml)
  }
```

Med disse beskrevne metoder kan vi derfor implementere alle de DAO operationer som vi ønsker. Disse er dog blevet implementeret i en TSSoapActor og vil blive beskrevet senere [Se afsnit 5.3.3].

5.3.3 Actorsystem

Vores actorsystem er opbygget således at det overordnet følger vores program design [Se Figur5.1]. Vi har dog tilføjet nogle flere komponenter der ikke vises

i dette diagram, blandt andet en overvågningsactor, eller supervisor actor, og nogle objekter til at håndtere vores datatyper.

Dette afsnit vil beskrive de forskellige actors, og primært deres hvordan de agerer i forhold til modtagne beskeder fra andre steder i systemet.

5.3.3.1 SupervisorActor

Vores SupervisorActor er den actor der overvåger alle andre actors, sørger for korrekt opstart, nedlukning og håndterer hvis en anden actor dør før tid. Hvis en actor dør før tid er vi interesseret i at få denne genstartet hurtigst muligt, vi har derfor implementeret en supervisorStrategy der genstarter alle actors der dør:

```
override def supervisorStrategy:
SupervisorStrategy = OneForOneStrategy() {
    case e: Throwable =>
      log.error(
      "Child escalated exception, restarting it: {}",
      e
      )
      Restart
  }
```

Hvis en underliggende actor i dette system derfor smider en exception e, bliver denne logget og actoren bliver genstartet.

Det eneste andet vores Supervisor skal kunne er at starte de nødvendige underliggende actors i actorsystemet. Dette sker ved at ved opstart af programmet bliver der sendt en InitSystem kommando til denne actor der derefter gør følgende:

```
  def receive = {
    case InitSystem(config, promise) =>

      context.actorOf(
        Props(classOf[BillingDAOActor], "billing")
      )
      context.actorOf(
        Props(classOf[SwitchDokDAOActor], "switch")
      )

      val soapClient = new IPNettSoapClient(
```

```
        this.stringValue("ipnett.soap.host", config),
        this.stringValue("ipnett.soap.username", config),
        this.stringValue("ipnett.soap.password", config))

      context.actorOf(Props(new TSSoapActor(soapClient)))

      val syncActor = context.actorOf(
      Props(classOf[SyncActor], "sync")
      )

      context.system.scheduler.schedule(
        nextExecutionInSeconds(0,0,0) seconds,
        24 hours,
        syncActor,
        DoSync
      )
      promise.success({})

      context.become({
        case msg =>
        throw new RuntimeException(
        s"Can no longer accept messages: $msg"
        )
      })
  }
```

Her ses at vores SupervisorActor først opretter de actors der er nødvendige for systemet, nemlig:

- BillingDAOActor: Håndterer al kommunikation med vores faktureringss-DAO.
- SwitchDokDAOActor: Håndterer al kommunikation med vores switch-DAO.
- TSSoapActor: Håndterer al kommunikation til traffic shaperen igennem vores SOAP client.
- SyncActor: Denne actor sørger for at automatisk oprette og nedlægge brugere en gang i døgnet.

Når disse er oprettet bliver vores SyncActor scheduleret til at lede efter nye brugere og nedlægge gamle ved midnat hver dag. Dette sker ved kommandoen:

```
context.system.scheduler.schedule(
        nextExecutionInSeconds(0,0,0) seconds,
        24 hours,
        syncActor,
        DoSync
      )
```

Hvor nextExecutionInSeconds er en metode der finder antallet af sekunder til midnat.

Når alle actors nu korrekt er oprettet svarer vores SupervisorActor tilbage ved at færdiggøre det promise den fik som parameter. Da vi ikke ønsker at man kan oprette flere instanser af vores actors skifter supervisor actoren herefter natur. Dette sker ved kommandoen:

```
context.become({
        case msg =>
        throw new RuntimeException(
        s"Can no longer accept messages: $msg"
        )
      })
```

Hvilket betyder at herefter vil alle beskeder sendt til denne actor resultere i en exception.

5.3.3.2 SwitchDAOActor

SwitchDAOActoren håndterer, som navnet indikerer, alle henvendelser til vores SwitchDAO [Beskrevet i afsnit 5.3.2.2]. For at vi nemmmere kan holde styr på de forskellige typer af kald til denne actor har vi opbygget en række datastrukturer. disse ovenstående traits er implementeret i SwitchDok modellen og de forskellige metoder for API'ets switchdok traits er implementeret i SwitchDok-DAO actoren. Her findes 4 sealed traits, der alle forlænger et SwitchDokEvent, der ligeledes er et sealed trait. At et trait er sealed betyder at det kun kan extendes i samme fil som deklarationen er i.

```
sealed trait SwitchDokEvent { def promise: Promise[_] }
sealed trait InsertEvent extends SwitchDokEvent {
def data: SDStructure }
sealed trait UpdateEvent extends SwitchDokEvent {
def data: SDStructure }
sealed trait DeleteEvent extends SwitchDokEvent {
```

```
def data: SDStructure}
sealed trait SelectEvent extends SwitchDokEvent {
def data: AnyVal}
```

En metode der indsætter data i swichtdatabasen forlænger så et InsertEvent såsom insertUnion:

```
case class InsertUnion(
        promise: Promise[Map[String, List[Long]]],
        data: Union
) extends InsertEvent
```

Ud fra koden ses det at InsertUnion metoden modtager et promise der består af et map af strenge der angiver hvad der er oprettet og en liste med primary keys for det tabeller der gives med til oprettelse. Derudover modtager den også data for den Union der oprettes. De andre actor metoder der indsætter data i switchdatabasetabellerne har samme opbygning og beskrives derfor ikke her men kan ses i [Se bilag E for disse.].

For at vi kan sikre os at alle beskeder kommer korrekt hen til denne actor har vi implementeret en preStart metode der subscriber til SwitchDokEvents i actorsystemets eventstream.

```
override def preStart() = {
   log.info("Starting a SwitchDok Database Actor")
   context.system.eventStream.subscribe(
            self,
                classOf[SwitchDokEvent]
   )
   super.preStart()
 }
```

På denne måde sikrer vi derfor at alle beskeder af typen SwitchDokEvent, eller under klasser af denne bliver modtaget af denne actor [Se bilag E for alle event typer]. Grundet mængden af case classes som denne actor skal kunne håndtere har vi valgt at ikke vise hele vores receive metode. Vi vil derfor vise hvordan receive metoden vil reagere hvis den modtager et InsertUnion event.

```
case InsertUnion(promise,union) =>
 val unionRelation = dao.insertUnion(union)
 union.switchList.map{ switch =>
  switch.manufacturer match {
   case Some(manufacturer) =>
    val manufacturerPromise = Promise[
```

```
            Map[String, List[Long]]
        ]
        self ! InsertManufacturer(
            manufacturerPromise,
            manufacturer
          )
        manufacturerPromise.future.onComplete{
      case Success(manufacturerResult) =>
        val switchPromise = Promise[
            Map[String, List[Long]]
        ]
        self ! InsertSwitch(
          switchPromise,
          switch,
          Map(
              "unionRelation" -> unionRelation,
              "manufacturerRelation" ->
                manufacturerResult.getOrElse(
                 "manufacturer", List.empty
                ).headOption.getOrElse(0)
            )
          )
        switchPromise.future.onComplete{
          case Success(switchResult) =>
            val map = Map(
             "union" -> List(unionRelation))
              ++ manufacturerResult
              ++ switchResult
            log.info(map.toString)
            promise.success(map)
        }
      }
      case None =>
    }
  }
```

Denne metode er også den mest komplekse i SwitchDAOActoren, da en Union som det ses i datastrukturen, kan indeholde en liste af switche, der ligeledes kan indeholde en liste af leveringsadresser. Det første denne metode derfor gør er at bede den tilknyttede Switch DAO til at oprette en ny forening. Når dette er gjort, bliver der mappet over alle switche der er givet i switch listen. Alle switches kan have en producent og på grund af vores database struktur skal denne oprettes før switch, hvilket actoren beder sig selv om at gøre asynkront

med kommandoen:

```
self ! InsertManufacturer(
          manufacturerPromise,
          manufacturer
        )
```

Først når denne kommando er færdigbehandlet må vi gå igang med at oprette switchen. Derfor er det først når vi modtager en Success fra vores manufacturerPromise, at vi beder vores actor om at oprette en switch. Når alle switches er blevet oprettet sætter vi alle resultaterne sammen og returnerer den i det oprindelige promise.

5.3.3.3 BillingDAOActor

BillingDAOActoren håndterer alle henvendelser til faktueringsDAO'en og videre til faktureringsdatabasen. Den fungerer derfor som et asynkront lag udenom databasen. Det første vores actor gør er at sikre sig at den modtager de beskeder den kan håndtere fra evenststreamen, og kun de beskeder. Dette sker i actorens preStart metode:

```
override def preStart = {
  log.info("Starting a BillingDAOActor")
  context.system.eventStream.subscribe(
   self,
   classOf[BillingCommand]
  )
  super.preStart()
}
```

Denne actor håndterer skal derfor kunne håndtere de samme anmodninger som faktureringsDAO'en kan, og vi har derfor implementeret følgende cases:

```
sealed trait BillingCommand extends {
    def promise: Promise[_]
}
case class GetUser(
       customerid: Long,
       promise: Promise[Set[SubscriptionInfo]]
) extends BillingCommand
case class GetUsers(
  promise: Promise[Set[SubscriptionInfo]]
) extends BillingCommand
```

```
case class GetUsersByUnion(
      promise: Promise[Set[SubscriptionInfo]],
      unionId: Long
) extends BillingCommand
case class GetUnionIds(
  promise: Promise[Set[Long]]
) extends BillingCommand
```

Da datastrukturerne der bliver returneret fra denne actor er sæt er det ikke krævet at actoren modulerer inputtet, som det bliver gjort i switchDAOActoren. Vores receive metode ser derfor således ud:

```
def receive = {
    case GetUser(customerid, promise) =>
     promise.success(dao.getUser(customerid))
    case GetUsers(promise) =>
     promise.success(dao.getUsers)
    case GetUsersByUnion(promise, union) =>
     promise.success(dao.getUsersByUnion(union))
    case GetUnionIds(promise) =>
     promise.success(dao.getUnionIds)
  }
```

Alle kald til denne actor henter derfor de data den skal bruge direkte fra DAO'en og indsætter det i vores promise.

5.3.3.4 TSSoapActor

TSSoapActoren håndterer al kommunikationen til traffic shaperen igennem vores SOAP klient. Alle beskeder som actoren skal kunne håndtere, er samlet i et trait ved navn TSEvent, og det første denne actor gør er derfor at sikre sig at den modtager alle beskeder af denne type:

```
override def preStart(): Unit = {
  log.info("Starting a TSSoapActor")
  context.system.eventStream.subscribe(
    self,
    classOf[TSEvent]
  )
  super.preStart()
}
```

Et TSEvent er et sealed trait, så actoren er nu klar til at modtage alle under cases af dette trait. For at overholde vores design omtalt i afsnittet om DAO design [Se afsnit 4.4.2], skal vi kunne få traffic shaperen til at tilføje, se og fjerne brugere på denne. Vi har derfor følgende cases som vores actor skal kunne håndtere:

```
  case class TSAddSubscriber(
    promise: Promise[Elem],
    xml: Elem
  ) extends TSEvent
case class TSGetSubscriber(
  promise: Promise[Elem],
  xml: Elem
) extends TSEvent
case class TSDeleteSubscriber(
  promise: Promise[Elem],
  xml: Elem
) extends TSEvent
```

vores SOAP klient returnerer altid en Future af Option af typen Elem, så derfor er det nødvendigt at vi pattern matcher på denne option, da vi kun ønsker at returnere en success hvis Optionen har Some værdi. En case i vores receive metode ser derfor således ud:

```
case TSAddSubscriber(promise, xml) =>
 soapClient.sendMessage(
  "IPEaddSubscriber",
  xml
 ).map(_ match {
  case None => promise.failure(
   new Exception("Something went very wrong")
  )
  case Some(x) => promise.success(x)
 })
```

5.3.3.5 SyncActor

SyncActoren er en actor der samler alle DAO's til at håndtere en daglig synkronisering af alle abonnementer i faktureringsdatabasen, og opretter deres tilsvarende hastigheder i traffic shaperen. Den skal ligeledes sikre sig at switchdatabasen er opdateret med hvilke kunder der er opkoblet på hvilke adresser. Alt dette gøres i receive funktionen ved at vi definerer en case ved navn DoSync. DoSync henter

først alle foreningsid'er fra faktureringsDAO'en, for hvert for hvert foreningsid henter vi nu alle aktive abonnementer i denne forening med faktureringsDAO'ens metode GetUsersByUnionId. For hver af disse kunder bliver der nu foretaget en kontrol. Hvis kunden ikke findes i switchdatabasen bliver denne oprettet der og i traffic shaperen. Hvis der findes en anden kunde på abonnentens adresse bliver denne kunde slettet og den nye abonnenent oprettet, og hvis kundenummeret er det samme, bliver kun traffic shaperen opdateret. [Se bilag F for kildekode].

Den nuværende udgave af SyncActoren er ikke i stand til at nedlægge kunder der ikke længere er aktive, da faktureringsDAO'en kun giver os besked på hvilke kunder der er aktive. For at kunne gøre dette, skal man ligeledes have mulighed for at hente alle kunder i switchdatabasen og sammenligne disse to lister. Alle kunder der findes i switchdatabasen, men ikke i faktureringsdatabasen skal da lukkes i traffic shaperen. Alle kunder der findes i faktureringsdatabasen men ikke i switchdatabasen skal oprettes i traffic shaperen.

5.4 Opsummering

Dette afsnit har fremlagt de vigtigste aspekter af API'ets implementation. Dette være sig kommunikation med faktureringsdatabasen, switchdatabasen og traffic shaperen. Der er beskrevet hvordan de vigtigste metoder fungerer og benyttes samt vist hvordan datastrukturen i applikationen er opbygget. Den nuværende opbygning af programmet er opsat således at udvidelser af funktionalitet er ligetil, da hvis man ønsker at tilføje funktionalitet til f.eks. switchdatabasen skal dette blot tilføjes i DAO'en og dens DAOActor. Som det ses i Bilag C har traffic shaperen rigtig stor funktionalitet som vi har valgt ikke at implementere, men kan være brugbar for en fuldstændig løsning, med f.eks. overvågning osv.

Vi har desuden implementeret mulighed for at systemet automatisk opretter nye kunders hastighed, hvilket har været en af Cirque's primære ønsker omkring funktionalitet i forhold til dette program. Dog kan den automatiske synkronisering ikke deaktivere nedlukkede kunder som forklaret tidligere [Se afsnit 5.3.3.5].

Chapter 6

Evaluering

6.1 Indledning

I dette afsnit vises en række tests der viser om implementationen udfører operationerne som forventet. Dernæst vil der diskuteres om resultaterne opfylder kravene i målsætningen. Derudover vil der beskrives en række udvidelsespunkter og tilføjelser til API'et. Til sidst vil der være en opsummering af de vigtigste resultater i rapporten.

6.2 Tests

I denne sektion vil vi beskrive en række testscases for derefter at teste dem op mod API'et.

De følgende tests er defineret:

For switchdatabasen via en browser

- Foretag en søgning efter en ip-adresse.
- Foretag en søgning efter en adresse.
- Foretag en søgning efter et kundenummer

Søgning i API'et foregår ved at man kalder Play med /database/search?"Option"="søgeparameter" hvorefter der returneres en liste med resultatet. Option kan være address, customerid eller ip

- søgning efter ip 192.168.1.10, resultat: List((2,'192.168.1.10','/29'))
- søgning efter adressen "Testgade", resultat: List((2,'192.168.1.10','/29'))
- søgning efter kundenummeret 200000, restultat: List((2,'192.168.1.10','/29'))

Som det ses returnerer metoderne kun det der findes i switch_detail toString metode der forbereder dem til at blive indsat i en SQL query. Der ville det nok være mere hensigtsmæssigt med en seperat metode der forbereder datastrukturen til at blive en SQL query og toString så var bibeholdt, da der så ville kunne printes alt ud fra search statementet.

For datastrukturerne via specs2

- Valideringstest af Switch datatypens toString
- Valideringstest af Switch_detail datatypen toString

En test af datastrukturerne vil indebære brug af specs2. Dette gør det muligt at skrive specifikationer for en klasse og hvad denne skal indeholde. Et eksempel på en specifikation er:

```
class HelloWorldSpec extends Specification {

  "The 'Hello world' string" should {
    "contain 11 characters" in {
      "Hello world" must have size(11)
    }
    "start with 'Hello'" in {
```

```
      "Hello world" must startWith("Hello")
    }
    "end with 'world'" in {
      "Hello world" must endWith("world")
    }
  }
```

Must er en matcher der returnerer succes eller failure alt efter om testen virker eller fejler.

De 2 test for switch og switch_detail bliver derfor:

```
"The Switch.toString" should {
    "return the name, unit,ip,numports" in{

    "('TestUX1',1,'192.168.1.11',4" must
    beEqualTo(switchTest.toString)
            }

          }


"The switchDetailTest.toString" should {
"return the port,ip,scope" in{

 "(4,'10.20.30.41','/27')" must
 beEqualTo(switchDetailTest.toString)
         }
       }
```

Resultaterne for de 2 tests er:

```
[info] DataTypeSpec
[info] The Switch.toString should
[info] + return the name, unit,ip,numports
[info] THE switchDetailTest.toString should
[info] + return the port,ip,scope
[info] Total for specification DataTypeSpec
[info] Finished in 16 ms
[info] 2 examples, 0 failure, 0 error
[info] Passed: Total 2, Failed 0, Errors 0, Passed 2
```

Så toString for switch og switch_detail datastrukturerne returnerer det forventede resultat.

For TS- og BillingActoren via specs2

- GetUsers: får en liste af brugere fra faktureringsdatabasen
- GetUnionIds: får en liste af union ID's fra faktureringsdatabasen
- GetUser: henter en bruger på bagrund af kunde id fra faktureringsdatabasen
- GetIsps: Henter isp'er fra Metro Activate.

Disse tests bruger new WithApplication() til at starte actor systemet så dette kan tilgås og databaserne og traffic shaper API'et kan kaldes. De er defineret på følgende måde:

```
"BillingDaoActor" should {
  "GetUsers" in new WithApplication(){
    val promise = Promise[Set[SubscriptionInfo]]
    Akka.system.eventStream.publish(GetUsers(promise))

    promise.future.onComplete{
      case Success(set) => set.isEmpty must beFalse
      case _ =>
    }

  }
"GetUnion" in new WithApplication(){
    val promise = Promise[Set[Long]]
    Akka.system.eventStream.publish(GetUnionIds(promise))

    promise.future.onComplete{
      case Success(set) => set.isEmpty must beFalse
      case _ =>
    }

   }

"GetCustomerId" in new WithApplication(){
   val promise = Promise[Set[SubscriptionInfo]]
   Akka.system.eventStream.publish(GetUser(300000,promise))
```

```
    promise.future.onComplete{
    case Success(set) => set.isEmpty must beTrue
     }
  }

}

"TrafficShaperActor" should{
   "Get ISPs" in new WithApplication(){
       val promise = Promise[Elem]
       Akka.system.eventStream.publish(TSGetIsps(promise,
         <ipe:IPEgetIsps>
           <ispname xsi:type="xsd:string">cirque</ispname>
         </ipe:IPEgetIsps>
       ))


       promise.future.onComplete{
       case Success(elem) => elem.attribute("return") must beEqualTo(1)
   }
  }
 }
```

Ud fra koden ses det at der tilføjes en besked til eventstreamen, hvorefter der testes på actornes future for at se om de returnerer de rigtige værdier. Kunde id testen er ment til at skulle fejle, da faktureringsdatabasen ikke indeholder kundenumre over 246000 endnu.

Testresultaterne for de forskellige actors er:

```
[info] ActorSpec
[info] BillingDaoActor should
[info] + GetUsers
[info] + GetUnion
[info] + GetCustomerId
[info] TrafficShaperActor should
[info] + Get ISPs
[info] Total for specification ActorSpec
[info] Finished in 9 seconds, 976 ms
[info] 4 examples, 0 failure, 0 error
```

6.3 Diskussion

I forhold til vores problemstilling har vi fået implementeret en første version af et API, der kan kommunikere med Cirque's nye trafficshapere, switchdatabasen og kundedatabasen. Samtidig har vi beskrevet de forskellige tekniske aspekter, der ligger til grund for en traffic shaper.

Vi har valgt at programmere vores API i Scala og har valgt at gøre brug af Play frameworket. Dette valg er primært taget for at vi kan implementere vores program med visionen om at holde det asynkront. Play tilbyder en asynkron http server, så alle requests til vores API vil blive modtaget, da http serveren aldrig blokerer. Play tilbyder ligeledes paralelle klasser i form af Akka Actors, der kan kommunikere asynkront hinanden imellem. En standard actor har en First-In-First-Out (FIFO) postkasse betydende at den tager det job der har ventet i længst tid og behandler det først. Dog har actors også en ikke blokerende natur, således at hvis en besked skal vente på en Future kan actoren gå i gang med at behandle en anden besked. Dette åbner også op for at man kan sende asynkrone beskeder fra actoren til sig selv uden at en deadlock opstår. Dette opfylder vores vision om ,at flere klienter kan tilgå systemet samtidig, uden sandsynlighed for race conditions og deadlocks. De forskellige dele af programmet fungerer desuden asynkront imellem.

Det ses i vores tests at programmet er funktionelt med hensyn til at kommunikere med de forskellige databaser og traffic shaperen. Der er ikke oprettet test til indsættelse i switchdatabasen, da kun første kørsel vil have succes, da data kun kan indsættes en gang i tabellen. Dette gælder kun medmindre man fjerner data manuelt fra tabellerne, hvilket vil besværliggøre testen og underminere tankegangen omkring et DAO, da dette gør sig ud for et abstraktionslag for data i databasen.

Vi har vist igennem teori, at vores egen designede database opfylder fjerde normalform som er vores ønske. Som begrundet tidligere ønsker vi ikke en database med højere normalform end fjerde. Fjerde normalform opfylder standardiserede database krav om ingen dupletter, adskillelse af tabeller og inden unødvendige afhængigheder. At have en database i så høj en normalform, kan dog besværliggøre forespørgsler til databasen da data vil være opsplittet i flere seperate tabeller.

Vil kunderne kunne mærke forskel?
I forhold til den tidligere løsning vil kunderne, på længere sigt, kunne mærke forbedrede supportmuligheder hvis nogle af de præsenterede udvidelsespunkter

også implementeres. Her tænkes især på muligheder for overvågning og diagnosticering af kunders forbindelser og hastigheder [Se afsnit 6.4].

6.4 Udvidelsespunkter

I forhold til at denne udgave af vores program er funktionel, er der dog ting vi har overvejet som mulige tilføjelsespunkter. Vi vil i dette afsnit beskrive de forskellige tilføjelser som vi har overvejet. Tilføjelserne er af variabel størrelse og sværhedsgrad at implementere, og i præsentationen vil der ikke blive skildnet imellem hverken sværhedsgrad, størrelse eller rækkefølge disse kunne implementeres i.

Analysemodul til nye boligforeninger. Cirque sælger fortsat denne løsning til nye kunder og har stor interesse i nemt at kunne implementere disse. Hver boligforening er koblet op på sin egen fiber forbindelse, der går direkte ud til boligforeningen. Størrelsen af denne fiber linje skal vurderes, således at de beboer der er i boligforening kan få den hastighed de betaler for. Derfor ville det være interessant at udregne, hvilken fælles linje der ville være nødvendig på baggrund af antallet af brugere, og deres mulige individuelle internet hastigheder. En mulig løsning på dette, kunne være at man ved hjælp af en Erlang distribution finder det sandsynlige antal af brugere og deres gennemsnitlige hastighed. Man kan derfor ved blot at gange den gennemsnitlige hastighed med det sandsynlige antal brugere, for at få et overslag på den fiber hastighed der skal bruges.

Overvågning af kunder til supportering. Da vores program i længden skal bruges både til implementering af nye brugere, og til support af eksisterende brugere vil nogle yderligere support muligheder være gode. Dette kunne f.eks være:

- Mulighed for at se hastighed i et GUI, heriblandt mulighed for on-the-fly at justere for kundens hastighed.
- Mulighed for at se om kunden har ledige IP adresser i sit vlan's scope, og derved om de har mulighed for at tilgå nettet.
- Overvågning af kundens forbindelse helt ud til opkoblingsadresse såsom kundens pingtider, pakketab m.m.
- Se oppetid for den switch kunden er koblet op igennem.

Opdater SyncActor til også at kunne nedlægge kunder. I den nuværende implementation kan SyncActoren ikke automatisk nedlægge kunder, dette kan give en problematik i forhold til sortseere på netværket. Disse sortseere kan i længden misbruge netværket, således at betalende kunder kan opleve problemer i diverse flaskehalse. Det er derfor både i Cirque's og brugernes interesse at dette bliver implementeret. Dette kan enten gøres ved at vi implementerer en DAO til at hente de kunder der er blevet inaktive siden vi sidst kørte vores synkronisering. En anden løsning [Som er beskrevet i afsnit 5.3.3.5] kunne være at man beder switchdatabasen om alle aktive kunder og sammenligner de to lister.

Mulighed for udvidet søgning i switchdatabasen. På nuværende tidspunkt er der mulighed for at søge efter adresser, kundenumre og IP adresser. Denne søgning kunne dog udvides til f.eks. at kunne:

- Søge efter alle adresser i en switch
- Søge efter alle opkoblinger på opgangs/gadeplan
- Hente alle brugere i en forening
- Prefix søge efter kundenumre og IP adresser
- Hente oplysninger omkring en switch, dens producent og model samt IP adresse.
- Mulighed for at søge på ydre og indre vlan

QoS til VoIP løsninger. Den nye traffic shaper har [Som det ses i bilag C] mulighed for at yde specielt Quality of Service til Voice over IP. Vi kan derfor åbne op for muligheden at kunne yde VoIP løsninger til Cirque's kunder, og kunne oprette dem, samt at supporterene kan sikre sig at de også har den QoS der er krævet.

6.5 Opsummering

Dette kapitel har beskæftiget sig med tests af vores program, samt har diskuteret om vi har opfyldt problemformuleringen. Vi har diskuteret de forskellige aspekter af programmet og switchdatabasen, og har fundet at vi langt hen af vejen har ramt vores mål og vision, fra da vi startede. Vi har opnået et program der kan

kommunikere internt med en grad af asynkronitet, som vi finder tilfredstillende. Programmet kan derudover kommunikeres med udefra ved hjælp af et RESTful API, hvilket fungerer helt asynkront. Vi har designet og implementeret en database efter fjerde normalform, denne er blevet opsat på en ekstern server således at den fungerer adskilt fra programmet. Derudover har vi set på mulige udvidelsespunkter der kunne være interessante retninger for programmet at udvikle sig i. Disse udviklingspunkter vil ligge til grund for fremtidigt udvikling på løsningen.

Chapter 7

Konklusion

Målet med denne rapport har været at implementere en traffic shaper og åbne op for dennes funktionalitet til styring af kunder og deres hastigheder. Derudover har vi også haft henblik på at definere og beskrive relevante netværkstermer. I forbindelse med moderniseringen af traffic shaperen har vi derfor valgt at implementere en ny switchdatabase til at holde informationer om opkoblingsparametre og sikre os at denne overholder en passende normalform.

Med hensyn til vores oprindelige problemformulering har vi opfyldt alle teoretiske spørgsmål i vores state of the art kapitel. Gennem en analyse af hvordan den programmeringsmæssige, udførsel af projektet, skulle foretages har vi ligeledes overholdt vores oprindelige målsætning. Dette gøres ved at have et lille, men funktionelt, API der sammenkobler traffic shaperen med faktureringsdatabasen og gemmer alle nødvendige data i switchdatabasen.

Som det ses i opsummeringen af vores evaluerings kapitel, er vores program blot en begyndelse til et komplet produkt, men der er fuld mulighed for udvidelser i fremtiden. Vi har desuden udført forskellige tests på programmet for at sikre os at vores funktionalitet agerer som ønsket.

Vi har igennem projektet fået dyb insigt i det netværksmæssige aspekt der skaber grund for en traffic shaper og de udfordringer en ISP står overfor. Vi har ligeledes fået en god og stabil forståelse for Scala, Play framework og Akka

actors. Dette vil gøre os i stand til at implementere ligendende API'er og backend-services til andre løsninger. Desuden er vores kendskab til databaser, normalisering af disse og SQL standarder væsentligt blevet forøget, som vi har fået ved at implementere en række forskellige Data Access Objects.

Appendix A

Traffic Shaping for boligforeninger

A.0.1 Baggrund

Vi har i dag en udfordring med båndbreddestyring, vi har ikke den fornødne kapacitet til at styre de mængder trafik der i dag bruges af vores bredbåndskunder.

A.0.2 Formål

Finde en løsning der løser problemet med kapacitet men som også sikrer at vi kan vækste, både på antal kunder og øget båndbredde til de enkelte kunder (100 mbit)

A.0.3 Succes kriterier

- Skalerbarhed – i takt med flere kunder og øget båndbredde
- Håndtere vores eksisterende kunders behov

- Minimal tid på drift af systemet
- Høj oppetid - stabilitet

A.0.4 Løsningsforslag

- Udskiftning af eksisterende core til 2 x MX480 – Metro Activate fra IPNett
- 2 x MX10 – Metro Activate fra IPNett
- Udvidelse af eksisterende linux traffic shaping
- Ny egen udviklet løsning (BSD/Linux)
- 3 part udviklet linux server med specialbygget netkort

A.1 Udskiftning af eksisterende core til 2 x MX480 – Metro Activate fra IPNett

A.1.1 Fordele

- Mulighed for 100 % redundans
- 100 % skalerbar løsning
- Løsningen er afprøvet og bruges for 30.000 kunder i Sverige
- Lav kompleksitet
- Mange udvidelsesmuligheder (tilkøb af kort i stedet for bokse)
- Billig at udvide
- Mulighed for multicast video streaming
- Selvbetjeningsværktøj (API)

A.1.2 Ulemper

- Omlægning af eksisterende Core
- Omlægning af Fiber forbindelser til boligforeninger

A.1.3 Økonomi

1.000.000 kr. (evt. sælge eksisterende MX80'ere).

A.1.4 Tidsperspektiv for implementering

2 måneder

A.2 2 x MX10 – Metro Activate fra IPNett

A.2.1 Fordele

- Mulighed for 100% redundans
- skalerbar løsning (ved tilkøb af flere bokse)
- Løsningen er afprøvet og bruges for 30.000 kunder i Sverige
- Mulighed for multicast video streaming
- Selvbetjeningsværktøj (API)
- Adskillelse fra eksisterende Core

A.2.2 Ulemper

- Omlægning af eksisterende fiber forbindelser

A.2.3 Økonomi

350.000

A.2.4 Tidsperpektiv

2 måneder

A.3 Udvidelse af eksisterende linux traffic shaping

A.3.1 Fordele

- Kendt system
- Stor erfaring med fejlsøgning

A.3.2 Ulemper

- Dårlig skalerbarhed
- Ikke muligt at presse mere trafik igennem
- Kompleks routning af trafik
- Maks kapacitet pr. box er 400 mbit
- Egen udviklet – ikke muligt at købe support
- hos eksterne leverandører
- Tungt at drifte
- Ingen redundans

A.3.3 Økonomi

30.000 kr. for at tilfredsstille det nuværende behov. Ved øgning af trafik og kunder vides ikke hvor mange servere der skal bruges

A.3.4 Tidsperpektiv

2 måned

A.4 Ny egen udviklet løsning (BSD/Linux)

A.4.1 Fordele

- Kan tilrettes som vi vil, da vi selv udvikler det

A.4.2 Ulemper

- Ved ikke om det kan lade sig gøre
- Ved ikke hvor meget hardware der skal bruges
- Ikke muligt at købe support
- Potentielt bruge meget tid på noget der måske aldrig kommer til at virke optimalt

A.4.3 Økonomi

?

A.4.4 Tidsperpektiv

?

A.5 3 part udviklet linux server med specialbygget netkort

A.5.1 Fordele

- Mulighed for at købe support

A.5.2 Ulemper

- Ikke afprøvet
- Softwaren skal først udvikles af 3 part
- Potentielt kan det være spildte kræfter – ved ikke med sikkerhed om det kommer til at virke optimalt

A.5.3 Økonomi

Hardware ca. 140.000 derudover udviklingsomkostninger til 3 part.

A.5.4 Tidsperpektiv

3-4 måneder

Appendix B

Kommunikationsdiagram

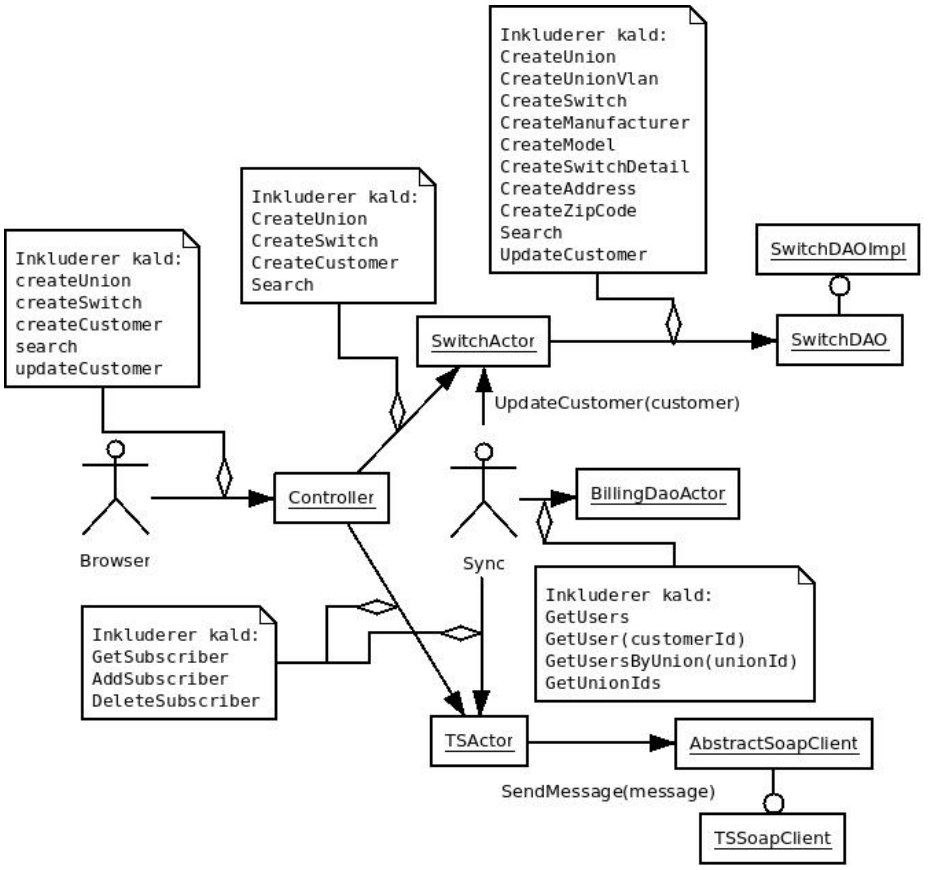


Figure B.1: Dette kommunikationsdiagram beskriver de mulige flows igennem vores program

Appendix C

Soap API dokumentation

SOAP API

IPnett Metro Activate.

Jan Munkhammar

JUNIPER NETWORKS

Table of Contents

The SOAP Server basics **4**
Service access **5**
Useful tools for test/development **7**
Services to consume **7**
IPEaddSubscriber 8
Result 8
IPEaddVoipSubscriber 9
Result 9
IPEdeleteSubscriber 10
Result 10
IPEdeleteVoipSubscriber 11
Result 11
IPEdisconnectSubscriber 12
Result 12
IPEgetIsps 13
Result 13
IPEgetSNMP 14
Result 14
IPEgetSubscriber 15
Result 15
IPEgetSubscriberAccounting 17
Result 17
IPEgetSubscriberOnline 18
Result 18
IPEpoolUtilization 19
Result 19
IPErouterCommand 20
Result 20
IPEserviceStatus 21
Result 21
IPEsubscriberUtilization 22

Result 22
IPEupdateSubscriber 23
Result 23
IPEdbStatus 24
IPEdbVersion 24
IPEdbSchemaChecksum 24
Example nusoap PHP client request 25

Revision History

Rev	Date	Author	Notes
1	2012-05-03	Jan Munkhammar	Initial document
2	2013-09-02	Jan Munkhammar	Update for new services.

The SOAP Server basics

The IPE is built on the Open Source project **NuSOAP - SOAP Toolkit for PHP,** http://sourceforge.net/projects/nusoap/.

It is a set of PHP classes - no PHP extensions required - that allows us to consume web services based on SOAP 1.1, WSDL 1.1 and HTTP 1.0/1.1 to manage and control subscribers dynamically in conjunction with the Juniper MX series routers.

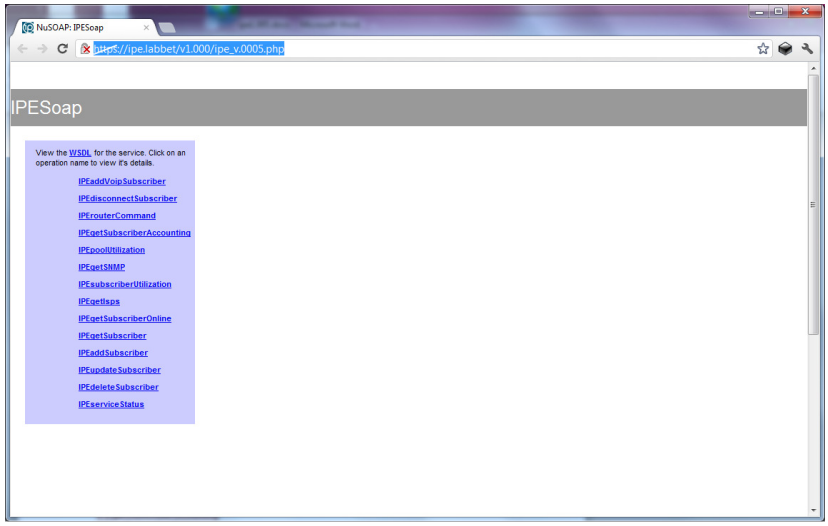
Another source of documentation is
http://www.scottnichol.com/nusoapprogwsdl.htm

Service access

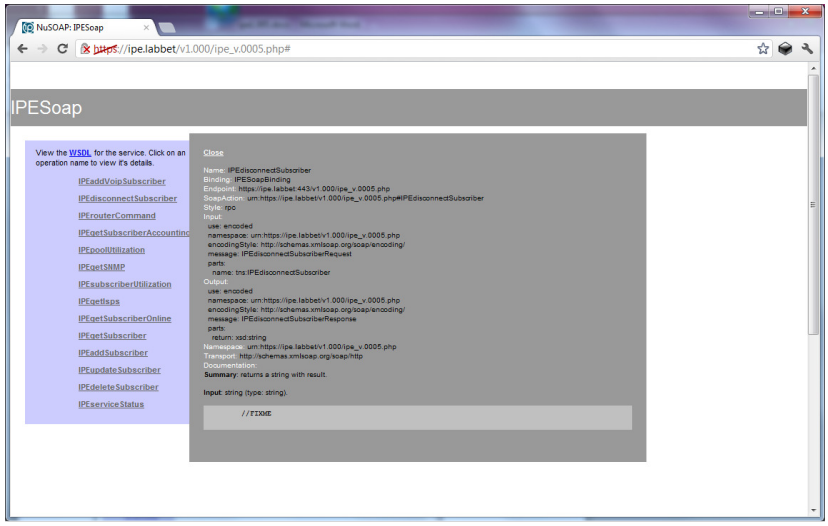
To gain access to the server and account must be created and each ipaddress calls will be made from must be set up. Credentials are static and requires no renewal. Multiple accounts can be made available. Access is only available on HTTPS (*Hypertext Transfer Protocol Secure*). Certificates are self-signed and must be accepted.

An example URL would be something like this, https://ipe.labbet/v1.000/ipe_v.0005.php.

On each new version the number will increase to keep services stable and compatible. Test and development systems can be provided.

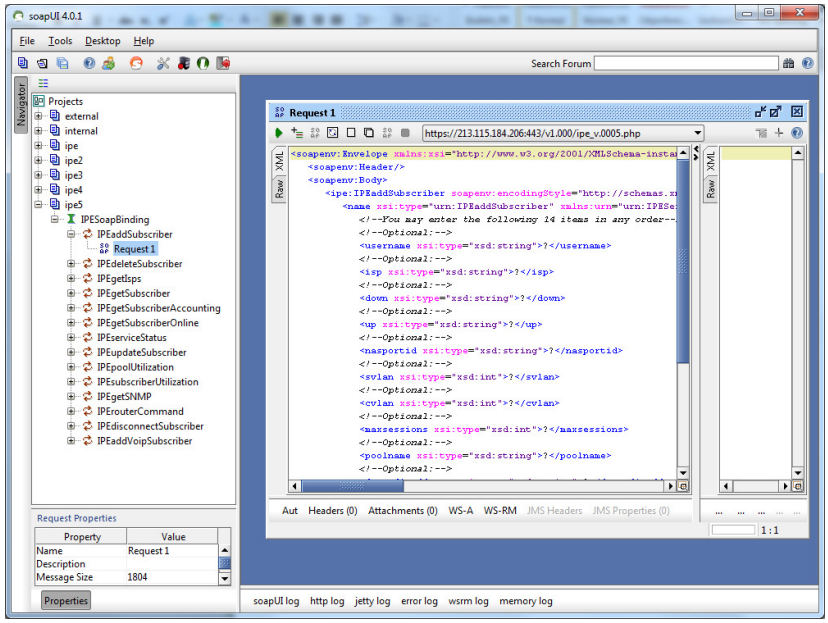


Example of help.



Useful tools for test/development

A very useful tool to test calls without programming is http://www.soapui.org/.



Services to consume

This is a list of all available services in alphabetical order.

IPEaddSubscriber
IPEaddVoipSubscriber
IPEdbSchemaChecksum
IPEdbStatus
IPEdbVersion
IPEdeleteSubscriber
IPEdeleteVoipSubscriber
IPEdisconnectSubscriber
IPEgetIsps
IPEgetSNMP
IPEgetSubscriber
IPEgetSubscriberAccounting
IPEgetSubscriberOnline
IPEpoolUtilization

Services are documented on the server side as well. By accessing the server most information and replies are documented there. All available services are listed and additional help is available on each service by clicking on them. More extensive information is provided in this document.

IPEaddSubscriber

Adds a new subscriber to the system. Most parameters are optional but input is validated before insertion and some have default values. If values already exists they are overwritten without notice. When changing ISP all live sessions are terminated. If speed changes, sessions are updated on the fly.

If no username is provided it will be generated,
Example, ID-y32u807e, where the last part is random.

isp is required.

up- and down-stream speeds are required.
Units are "M" for Megabit, "G" for Gigabit and "k" for kilobit. If no unit is sent "M" is applied as default.

Nasportid
or
svlan and/or cvlan is required.
Nasportid is the combination of outer and inner vlan tags separated by a semicolon. Example, 3431:101.
svlan is the outer vlan. cvlan is the inner vlan. If no outer vlan is used the cvlan should be used for vlan.
This is the key value to identify a subscriber request.

maxsessions has a system default value if not provided.

filter can be either "Disabled" or "Abuse".
filtermessage is an optional personal message for filters.

nasidentifier has a system default value if not provided.

Input: username (type: string).
Input: isp (type: string).
Input: down (type: string).
Input: up (type: string).
Input: nasportid (type: string). Format is svlan:cvlan.
Overrules svlan and cvlan.
Input: svlan (type: int).
Input: cvlan (type: int).
Input: maxsessions (type: int).
Input: macaddress (type: string).
Input: framedipaddress (type: string).
Input: filter (type: string).
Input: filtermessage (type: string).
Input: nasidentifier (type: string).

Result

Returns a string,

1::Action activate SUCCESS
or
0::Action activate FAILED

IPEaddVoipSubscriber

Adds a new VOIP-subscriber to the system. If the values already exists they are overwritten without notice. Shares the same object in the database as the internet service but they can be added and deleted independently.

Nasportid
or
svlan and/or cvlan is required.
Nasportid is the combination of outer and inner vlan tags separated by a semicolon. Example, 3431:101.
svlan is the outer vlan. cvlan is the inner vlan. If no outer vlan is used the cvlan should be used for vlan.
This is the key value to identify a subscriber request.

voippool, voipip and voipisp is used differently depending on customer implementation.

nasidentifier has a system default value if not provided.

Input: nasportid (type: string). Format is svlan:cvlan.
Overrules svlan and cvlan.
Input: svlan (type: int).
Input: cvlan (type: int).
Input: voippool (type: string).
Input: voipip (type: string).
Input: voipisp (type: string).
Input: nasidentifier (type: string). Router name/indentifier

Result

Returns a string,

1::Action activate SUCCESS
or
0::Action activate FAILED

IPEdeleteSubscriber

Deletes a subscriber from the system. All active session are terminated.

Nasportid
or
svlan and/or cvlan is required.
Nasportid is the combination of outer and inner vlan tags separated by a semicolon. Example, 3431:101.
svlan is the outer vlan. cvlan is the inner vlan. If no outer vlan is used the cvlan should be used for vlan.

Input: nasportid (type: string). Format is svlan:cvlan.
Overrules svlan and cvlan.
Input: svlan (type: int).
Input: cvlan (type: int).

Result

Returns a string,

1::Action deactivate SUCCESS
or
0::Action deactivate FAILED

IPEdeleteVoipSubscriber

Deletes a VOIP-subscriber from the system.

Nasportid
or
svlan and/or cvlan is required.
Nasportid is the combination of outer and inner vlan tags separated by a semicolon. Example, 3431:101.
svlan is the outer vlan. cvlan is the inner vlan. If no outer vlan is used the cvlan should be used for vlan.

Input: nasportid (type: string). Format is svlan:cvlan.
Overrules svlan and cvlan.
Input: svlan (type: int).
Input: cvlan (type: int).

Result

Returns a string,

1::Action deactivate SUCCESS
or
0::Action deactivate FAILED

IPEdisconnectSubscriber

This service disconnects one or more subscribers from the system. If more then one separate the values with semicolon. (";")

nasidentifier has a system default value if not provided.
acctsessionid is a dynamic value assigned to the session. It can be retrived by calling ***IPEgetSubscriberOnline.*** This is only valid for live sessions.

Input: nasidentifier (type: string).
Input: acctsessionid (type: string).

Result

Returns a string,

0::Session 4545 did NOT disconnect. Error-Cause = Session-Context-Not-Found
or
1::Session 238 is disconnected.

IPEgetIsps

This service tells if or which isps are available.
If ispname in request 1 or 0 is returned depending result.
If nothing in request a comma separated string with all ISP's is returned.

Input: ispname (type: string).

Result

Returns a string,

alltele,bahnhof,bredband2,isp1,isp2,t3,tele2,uvtc
or
1 or 0

IPEgetSNMP

This service returns the snmp get oid result as a string.

Input: oid (type: string).
Input: nasidentifier (type: string)

Result

Returns a string,

STRING: Juniper Networks, Inc. mx80 internet router, kernel JUNOS 11.4R1.14 #0

IPEgetSubscriber

This service allows search for one or more subscribers based on search input. The reply is either an array with one or more subscribers or, if paging is requestsed, delivered as JSON data. This is tailored for use of jqGrid, http://www.trirand.com/blog/.

Input: username (type: string).
Input: isp (type: string).
Input: down (type: string).
Input: up (type: string).
Input: nasportid (type: string). Format is svlan:cvlan.
Can be used for wildcard searches, %454 returns 1006:454. Overrules svlan and cvlan.
Input: svlan (type: int).
Input: cvlan (type: int).
Input: maxsessions (type: int).
Input: macaddress (type: string).
Input: voippool (type: string).
Input: voipip (type: string).
Input: voipisp (type: string).
Input: framedipaddress (type: string).
Input: filter (type: string).

Optional input for paging support

Input: page (type: string). Returns the input value.
Input: limit (type: string). Numrows to return
Input: sidx (type: string). Orderid
Input: sord (type: string). Sortorder

Result

Returns an array,

```
Array ( [0] => Array (
        [id] => 107
        [username] => jan@isp2
        [isp] => isp2
        [down] => 100M
        [up] => 10M
        [svlan] => 1006
        [cvlan] => 454
        [maxsessions] => 5
        [poolname] =>
        [framedipaddress] => 234.14.34.223
        [macaddress] => 00dd.0101.0203
        [voippool] =>
        [voipip] =>
        [voipisp] =>
        [filter] =>
        [filtermessage] =>
        [nasidentifier] =>
        [date_entered] => 2012-01-31 10:05:58
        [entered_user_id] => 1.1.1.17
        [date_modified] => 2012-02-02 21:41:22
        [modified_user_id] => 1.1.1.17
                        ) )
```

or JSON formatted if paging support is requested.


```
{"page":"1","total":"1","records":"2","rows":[{"id":0,"cell":["377
36","janss@isp1","isp1","1:1","1M","1M","","MX80-
1_BRAS"]},{"id":1,"cell":["107","janne@isp2","isp2","1006:454","1M
","1M","","MX80-1_BRAS"]}]}
```

IPEgetSubscriberAccounting

This service allows search for subscriber accounting data. This service can also return both simple arrays and JSON.
The maximum datespan is three months. This can be changes by configuration if required.

begin and end dates are required.
class is the combined username@isp.

Input: begin (type: string). Start date for first record. yyyy-mm-dd.
Input: end (type: string). End date for last record. yyyy-mm-dd.
Input: class (type: string).
Input: macaddress (type: string).
Input: framedipaddress (type: string).
Input: nasportid (type: string). Format is svlan:cvlan.
Can be used for wildcard searches, %454 returns 1006:454.Overrules svlan and cvlan.
Input: svlan (type: int).
Input: cvlan (type: int).
Input: isp (type: string).
Input: nasidentifier (type: string).Router identifier/name
Input: page (type: int). Page support for grid tables. Turns on JSON reply. Input: rows (type: string).Number of requested rows on page.
Input: sidx (type: string).Sort on column.
Input: sord (type: string).Sortorder

Result

Returns an array,
Array ([0] => Array (
[class] => ID-gqiyozde@isp1
[nasportid] => 1006:11
[status] =>
[eventtime] => 2012-02-28 09:52:13
[macaddress] => 0023.9c22.6b40
[framedipaddress] => 10.254.100.2
[reason] =>
[callingstationid] =>
[nasidentifier] => MX80-1_BRAS

or JSON formatted if paging support is requested.

{"page":"1","total":"682","records":"682","rows":[{"id":0,"cell":[0,"arton@isp1","isp1","18","0000.a212.3486","10.254.99.25","Alive","","MX80-1_BRAS","2012-03-05 10:32:33"]}]}

IPEgetSubscriberOnline

This service allows search in the online session database. Here is all valid sessions stored until the subscriber logs off.

Input: username (type: string).
Input: isp (type: string).
Input: nasportid (type: string). Format is svlan:cvlan.
Can be used for wildcard searches, %454 returns 1006:454. Overrules svlan and cvlan.
Input: svlan (type: int).
Input: cvlan (type: int).
Input: macaddress (type: string).
Input: framedipaddress (type: string).
Input: nasidentifier (type: string).
Input: page (type: int). Page support for grid tables. Turns on JSON replyInput: rows (type: string).Number of requested rows on page.
Input: sidx (type: string).Sort on column.
Input: sord (type: string).Sortorder

Result

Returns an array, Array ([0] => Array (
[username] => jan@isp2
[nasportid] => 1006:454
[svlan] => 1006
[cvlan] => 454
[macaddress] => 00dd.01aa.bb22
[framedipaddress] => 10.254.200.7
[acctsessionid] => 75
[option82] => 06010048@IPnett-iMAP#10m:1M
[timestamp] => 2012-02-03 14:59:36

or JSON formatted if paging support is requested.

{"page":"1","total":"5","records":"67","rows":[{"id":0,"cell":[0,"reconfigure_test@isp2","isp2","1996:72","00dd.01aa.bb22","10.254.200.3","Alive","","06010048@IPnett-iMAP#10m:1M","MX80-1_BRAS","2012-05-03 16:08:49"]}…

IPEpoolUtilization

This service can show the DHCP pool utilization of an requested isp or a summary of the whole router.

Input: ispname (type: string).
Input: nasidentifier (type: string).

Result

Returns a string with "The % usage of pool(s)","Total # of addresses in pool(s)","The isp"
0,256,isp1

IPErouterCommand

This service allows you to send any show command to the router if you have the secret word.

nasidentifier is the requested router.
Command is any JUNOS "show" command, example,
"show dynamic-configuration session information session-id 61939"
token is the secret string to allow this.

Input: nasidentifier (type: string).
Input: command (type: string).
Input: token (type: string).

Result

Returns a string with the result,

```
Session info:
  Accounting session ID: 61939
  IP address: 10.254.200.3
  IP netmask: 255.255.255.0
  Logical system name: default
  Profile name: SVLAN-LOCAL-DEMUX
  MAC address: 00:dd:01:aa:bb:22
  NAS port type: 15
  Routing instance: isp2
  Access Profile: subscribers
  User name: 00dd.01aa.bb22
  Interface name: demux0.1073803744
  Dynamic-configuration state: 2
  Client session type: 1
  IFL type: 2
  Framed Ipv4 Pool: isp2-pool1
  Accounting type: 2
  Accounting interval: 3600
  Underlying logical-interface: ge-1/0/0.1073742034
  Client login time: 2012-05-03 16:08:48 CEST
  DHCP option: 35:01:01:74:01:01
  VLAN tag: 72
  SVLAN tag: 1996
  DSL Forum attributes: \x01\x04\x06\x01
  Agent Circuit ID: \x06\x01
  Agent Remote ID: IPnett-iMAP
  Configuration bits: 0x80097 0 0 0 0
Dynamic configuration:
  junos-cos-shaping-rate: 10m
  junos-input-filter: 1M
  junos-interface-unit: 1073803744
  junos-output-filter: fwd_all
  junos-phy-ifd-name: ge-1/0/0
  junos-underlying-interface: ge-1/0/0.1073742034
```

IPEserviceStatus

This service allows to check if services can be consumed and can also force a SOAP error to allow SOAP error handling. ERRORTEST

Input: returnstring (type: string).

Result

Returns a string
1::test

or a SOAP fault if requested.

```
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode xsi:type="xsd:string">CLIENT_020</faultcode>
      <faultactor xsi:type="xsd:string"/>
      <faultstring xsi:type="xsd:string">ERRORTEST This is a forced
error!</faultstring>
      <detail xsi:type="xsd:string"/>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
```

IPEsubscriberUtilization

This service returns the number if subscribers for a given router.

nasidentifier has a system default value if not provided.

Input: nasidentifier (type: string)

Result

Returns a string,
2234

IPEupdateSubscriber

This service allows updates of a subscriber. This is an replica of IPEaddSubscriber. When changing ISP all live sessions are terminated. If speed changes, sessions are updated on the fly.

If no username is provided the old one is kept,

Nasportid
or
svlan and/or cvlan is required.
Nasportid is the combination of outer and inner vlan tags separated by a semicolon. Example, 3431:101.
svlan is the outer vlan. cvlan is the inner vlan. If no outer vlan is used the cvlan should be used for vlan.
This is the key value to identify a subscriber request.

nasidentifier has a system default value if not provided.

Input: username (type: string).
Input: isp (type: string).
Input: down (type: string).
Input: up (type: string).
Input: nasportid (type: string). Format is svlan:cvlan.
Overrules svlan and cvlan.
Input: svlan (type: int).
Input: cvlan (type: int).
Input: maxsessions (type: int).
Input: macaddress (type: string).
Input: framedipaddress (type: string).
Input: filter (type: string).
Input: filtermessage (type: string).
Input: nasidentifier (type: string).

Result

Returns a string,

1::Action activate SUCCESS
or
0::Action activate FAILED

IPEdbStatus

This service return a checksum for each servers database to validate that they are in sync. Used by the GUI. Please contact IPnett if details are required.

IPEdbVersion

This service return an array of version info. Used by the GUI. Please contact IPnett if details are required.

IPEdbSchemaChecksum

This service return a md5 checksum for the database schema. Used by the GUI. Please contact IPnett if details are required.

Example nusoap PHP client request

```
require_once('lib/nusoap.php');
$client = new
nusoap_client('https://127.0.0.1/v1.000/ipe_v.0005.php?wsdl',
true);
$client->soap_defencoding = 'UTF-8';
$client->setCredentials("username","password");

$result = $client->call('IPEaddSubscriber', array(array(
'username' => 'e123431',
'isp' => 'isp1',
'down' => '100',
'up' => '10',
'svlan' => '3442',
'cvlan' => '101',
)));

print_r(utf8_encode($result));
```

Appendix D

SwitchDok Datatypes

```
package model

object SwitchDok {

  sealed trait SDStructure {
    override def toString: String
    def optionToString(option: Option[Any], prefix: String = "", suffix
      case Some(any) => prefix + any.toString + suffix
      case None => ""
    }
  }

}

import SwitchDok._



case class Union(name:String,switchList:List[Switch]) extends SDStructu
  override def toString = s"('$name')"
}
sealed trait Vlan extends SDStructure {
```

```
  def vlan: Int
}
case class OuterVlan(vlan:Int) extends Vlan{
  override def toString = s"($vlan"
}
case class InnerVlan(vlan:Int) extends Vlan{
  override def toString = s"($vlan"
}
case class Customer(customerid:Long,address:String) extends SDStructure{
  override def toString = s"($customerid"
}
case class Switch(name:String,unit:Int, ip:String, numports:Int, switchDet
  override def toString = s"('$name', $unit, '$ip', $numports"
}
case class SwitchDetail(port:Int, ip:String, scope:String,outerVlan:OuterV
  override def toString = s"($port, '$ip', '$scope')"
}
case class SDAddress(streetname:String, number:Int, floor:Option[String],
 override def toString = s"('$streetname',$number${this.optionToString(floo
}
case class ZipCode(zipCode:Int,city:String) extends SDStructure{
  override def toString = s"($zipCode,'$city')"
}
case class SModel(smodel:String) extends SDStructure{
  override def toString = s"('$smodel'"
}
case class Manufacturer(manufacturer:String, model: Option[SModel]) extend
  override def toString =s"('$manufacturer')"
}
```

Appendix E

SwitchDAOActor case classes

```
sealed trait SwitchDokEvent { def promise: Promise[_] }
case class SwitchDokCreateDatabase(
promise: Promise[List[Boolean]]
) extends SwitchDokEvent
sealed trait InsertEvent extends SwitchDokEvent {
def data: SDStructure }
sealed trait UpdateEvent extends SwitchDokEvent {
def data: SDStructure }
sealed trait DeleteEvent extends SwitchDokEvent {
def data: SDStructure}
sealed trait SelectEvent extends SwitchDokEvent {
def data: Any}

case class InsertUnion(
promise: Promise[Map[String, List[Long]]],
data: Union
) extends InsertEvent
case class InsertSwitch(
promise: Promise[Map[String, List[Long]]],
data: Switch, relations: Map[String, Long]
) extends InsertEvent
```

```
case class InsertSwitchDetail(
promise: Promise[Map[String, List[Long]]],
data: SwitchDetail,
relations: Map[String, Long]
) extends InsertEvent
case class InsertAddress(
promise: Promise[Map[String, List[Long]]],
data: SDAddress,
relations: Map[String, Long]
) extends InsertEvent
case class InsertCustomer(
promise: Promise[Map[String, List[Long]]],
data: Customer,
relations: Map[String, Long]
) extends InsertEvent
case class InsertZipCode(
promise: Promise[Map[String, List[Long]]],
data:ZipCode
) extends InsertEvent
case class InsertSModel(
promise: Promise[Map[String, List[Long]]],
data:SModel,
relations: Map[String, Long]
) extends InsertEvent
case class InsertManufacturer(
promise: Promise[Map[String, List[Long]]],
data:Manufacturer
) extends InsertEvent
case class InsertVlan(
promise: Promise[Map[String, List[Long]]],
data: Vlan,
relations: Map[String, Long]
) extends InsertEvent

case class SelectFrom(
promise:Promise[Any],
data:Any
) extends SelectEvent

case class DeleteCustomer(
promise:Promise[Map[String, List[Long]]],
data:Customer
) extends DeleteEvent
```

```
case class UpdateCustomer(
promise: Promise[Map[String, List[Long]]],
data: Customer,
relations: Map[String, Long]
) extends UpdateEvent
```

Appendix F

SyncActor.scala

```
class SyncActor extends Actor with ActorLogging{

override def preStart(): Unit = {
  log.info("Starting a SyncActor")
  context.system.eventStream.subscribe(self, classOf[SyncMessage])
  super.preStart()
}

def receive: Actor.Receive = {
  case DoSync =>
    val unionPromise = Promise[Set[Long]]
    context.system.eventStream.publish(
     GetUnionIds(unionPromise))
    unionPromise.future.map{_.map{ unionId =>
      val userSetPromise = Promise[Set[SubscriptionInfo]]
      context.system.eventStream.publish(
      GetUsersByUnion(userSetPromise, unionId))
      userSetPromise.future.onComplete{
        case Failure(_) => log.warning(
         "Something went very wrong in the BillingDAOActor"
        )
        case Success(set) => set.map{ subscriptionInfo =>
```

```
        val customerExistsPromise =
         Promise[List[SwitchDetail]]
        context.system.eventStream.publish(
         SelectFrom(
          customerExistsPromise,
          subscriptionInfo.address
         )
        )
    customerExistsPromise.future.map{ list =>
        list match {
         case List.empty => log.warning(
            s"No address found by the name of
            ${subscriptionInfo.address}"
            )
           case valid =>
             val switchDetail = valid.head
             switchDetail.customer match {
               case None =>
               //No customer found, adding one we found.
                    val insertCustomerPromise =
                    Promise[Map[String, List[Long]]]
                    context.system.eventStream.publish(
                    InsertCustomer(
                    insertCustomerPromise,
                    Customer(
                    subscriptionInfo.customerId
                    ),
                    Map(
                    "switchDetailRelation" ->
                     switchDetail.id.get
                    )
                   )
                  )
               insertCustomerPromise.future.onComplete{
               case Success(_) =>
               val xml = <ipe:IPEaddSubscriber
               soapenv:encodingStyle=
               "http://schemas.xmlsoap.org/soap/encoding/">
          <name xsi:type="urn:IPEaddSubscriber"
          xmlns:urn="urn:IPEServiceSoapwsdl">
          <isp xsi:type="xsd:string">cirque</isp>
        <down xsi:type="xsd:string">{
        subscriptionInfo.speed.download}</down>
        <up xsi:type="xsd:string">{
```

```
      subscriptionInfo.speed.upload}
      </up>
      <svlan xsi:type="xsd:int">{
      switchDetail.outerVlan.vlan}
      </svlan>
      <cvlan xsi:type="xsd:int">{
      switchDetail.innerVlan.vlan}
      </cvlan>
      </name>
      </ipe:IPEaddSubscriber>
      val tsAddSubscriberPromise = Promise[Elem]
      context.system.eventStream.publish(
      TSAddSubscriber(tsAddSubscriberPromise,xml))
      tsAddSubscriberPromise.future.onComplete{
       case Success(_) => log.info(
       s"Successfully added customer
        ${subscriptionInfo.customerId}"
        )
         }
       }
case Some(customer) =>
  //A customer exists, checking if
  //its the same customer as the subscriber.
   if (customer.customerid ==
    subscriptionInfo.customerId) {
    //customer is the same, updating traffic shaper.
     val xml = <ipe:IPEaddSubscriber
     soapenv:encodingStyle=
     "http://schemas.xmlsoap.org/soap/encoding/">
     <name xsi:type="urn:IPEaddSubscriber"
     xmlns:urn="urn:IPEServiceSoapwsdl">
     <isp xsi:type="xsd:string">cirque</isp>
     <down xsi:type="xsd:string">{
     subscriptionInfo.speed.download}
     </down>
     <up xsi:type="xsd:string">{
     subscriptionInfo.speed.upload}</up>
     <svlan xsi:type="xsd:int">{
     switchDetail.outerVlan.vlan}
     </svlan>
     <cvlan xsi:type="xsd:int">
     {switchDetail.innerVlan.vlan}
     </cvlan>
     </name>
```

```
    </ipe:IPEaddSubscriber>
  val tsAddSubscriberPromise = Promise[Elem]
  context.system.eventStream.publish(
  TSAddSubscriber(tsAddSubscriberPromise,xml)
  )

tsAddSubscriberPromise.future.onComplete{
    case Success(_) =>
     log.info(
     s"Successfully added customer
     ${subscriptionInfo.customerId}"
       )
      }
    } else {
    val deletePromise = Promise[Boolean]
    context.system.eventStream.publish(
    DeleteCustomerRelation(
    deletePromise,
    switchDetail.id.get
        )
       )
    val insertPromise = Promise[Map[String, List[Long]]]
     context.system.eventStream.publish(
      InsertCustomer(
       insertPromise,
        Customer(
         subscriptionInfo.customerId),
          Map("switchDetailRelation" ->
           switchDetail.id.get
           )
          )
         )

insertPromise.future.onComplete{
case Success(_) =>
 val tsAddSubscriberPromise = Promise[Elem]
 val xml = <ipe:IPEaddSubscriber
 soapenv:encodingStyle="
 http://schemas.xmlsoap.org/soap/encoding/">
 <name xsi:type="urn:IPEaddSubscriber"
 xmlns:urn="urn:IPEServiceSoapwsdl">
 <isp xsi:type="xsd:string">cirque</isp>
 <down xsi:type="xsd:string">{
 subscriptionInfo.speed.download}
```

```
            </down>
            <up xsi:type="xsd:string">{
            subscriptionInfo.speed.upload}
            </up>
            <svlan xsi:type="xsd:int">{
            switchDetail.outerVlan.vlan}
            </svlan>
            <cvlan xsi:type="xsd:int">{
            switchDetail.innerVlan.vlan}
            </cvlan>
            </name>
            </ipe:IPEaddSubscriber>
            context.system.eventStream.publish(
             TSAddSubscriber(tsAddSubscriberPromise, xml)
            )

           tsAddSubscriberPromise.future.onComplete{
             case Success(_) =>
              log.info(
              s"Successfully added customer
              ${subscriptionInfo.customerId}"
                      )
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Bibliography

[akk] Actors.

[Ale13] A. Alexander. *Scala Cookbook: Recipes for Object-Oriented and Functional Programming*. O'Reilly Media, 2013.

[And00] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[And13] Ulrik Andersen. Fra december må du køre i motorvejens nødspor, 2013.

[AST10] David J. Wetherall Andrew S. Tanenbaum. *Computer Networks Author*. Prentice Hall, 2010.

[BC13] Peter Hilton Erik Bakker and Francisco Canedo. *Play for Scala*. Manning, 2013.

[Cis05] Cisco. Protecting the cisco catalyst 6500 series switches against denial-of-service attacks, 2005.

[Cis06] Cisco. Understanding jitter in packet voice networks (cisco ios platforms), 2006.

[Dat03] C.J. Date. *An introduction to Database Systems*. Addison Wesley, 2003.

[GA06] Paul Ganley and Ben Allgrove. Net neutrality debate. *Computers & Law*, 17(3), 2006.

[Gup12] Munish K. Gupta. *Akka Essentials*. PACKT, 2012.

[IEE05] IEEE-2005. Ieee 802.1ad-2005, 2005.

[KD11] Partha Kanuparthy and Constantine Dovrolis. Shaperprobe: end-to-end detection of isp traffic shaping using active methods. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 473–482. ACM, 2011.

[Mal] Konrad Malawski. Scala's types of types.

[mcm] mcmasteruni. Computer network traffic shaping.

[oIT07] Office of Information Technology. Vlan identification number assignments, 2007.

[Wie13] TU Wien. Best-practice software engineering, 2013.