Hovedprosjekt 08

M Byhring, T E Iversen, L M Bredal
Høgskolen i Oslo, avdeling for ingeniørutdanning

26 februar, 2008

Innhold

1 Innledning

Her vil det komme en innledning etter hvert

LCFG er et system ++++

1.1 Sammendrag

Her kommer abstract.

1.2 Forord

Vi skriver her noen velvalgte og vakre ord, og husker å takke alle som takkes bør, samt de vi har glemt.

2 Hoveddel

2.1 Innledning

Innledning. Hente en del fra forprosjektrapporten..

2.2 Planlegging og metode

2.3 Prosjektets start

Prosjektet begynte den... bla bla bla.. Satte opp hjemmesiden og prosjektdagbok.

2.3.1 Profilfilene

Datagrunnlaget for prosjektet er et sett XML-filer (Extensible Markup Language), der hver fil representerer konfigurasjonen til en maskin på et gitt tidspunkt. Første datasett bestod av 1060 XML-filer med en gjennomsnittlig størrelse på 1 MB. Fase en av prosjektet gikk i stor grad ut på å få en oversikt over strukturen på disse filene og hvilke data vi hadde tilgjengelig. Oppdragsgiver hadde gitt oss en viss pekepinn på hvilke data som ikke var interessante, blant annet feilsøkingsinformasjon. Dette ble fjernet, noe som halverte filstørrelsen og gjorde det lettere å lese filene manuelt.

Neste skritt i prosessen var å velge ut hvilke data vi ville jobbe videre med. I dette arbeidet benyttet vi oss av dokumentasjonen [2] til LCFG for å finne ut hva de enkelte verdiene representerte. Denne dokumentasjonen var forholdsvis mangelfull, men den var likevel til stor hjelp. Det viste seg blant annet at en del datafelter vi i utgangspunktet så på som interessante, likevel ikke hadde den betydningen vi trodde. En oppsummering av oppbygningen til XML-filene:

- profilen består av to hovedseksjoner: components og packages
- seksjonen components inneholder en rekke underseksjoner som representerer konfigurasjonen av et program, eller tjeneste (komponent) på maskinen
- en komponent kan inneholde både andre underseksjoner og bladnoder som inneholder informasjon.
- det er tilsammen 111 forskjellige 'components' i datamaterialet vårt

- det er kun 'profile'-komponenten som må være med i componentsseksjonen

2.3.2 Verktøy

Tidlig i prosjektprosessen hadde vi møte med veileder hvor vi diskuterte hvilke teknologier vi burde bruke til å gjennomføre vårt prosjekt. Vi trengte et programmeringsspråk som raskt kunne tolke XML-data, kommunisere med en database og generere tekstfiler. Vi trengte også et språk som skal brukes til å visualisere hundrevis av noder i et tredimensjonalt rom. Da XML-filene er svært store, vil vi også legge inn ønsket informasjon inn i en database, både for å spare plass og tid. Etter å ha lest om forskjellige teknologier og vurdert alternativene, bestemte vi oss for å bruke Perl, VRML (Virtual Reality Modeling Language) og mySQL.

VRML er et Markup-Language med en syntaks som ligner på HTML (HyperText Markup Language) og XML.

Gruppa har valgt å benytte mySQL som databasemotor, da vi tidligere har jobbet med denne. Som programmerings-IDE har vi valgt å bruke eclipse, med tilleggsmodulene EPIC (Eclipse Perl Integration) og subclipse for versjonskontroll (Subversion). Dette fordi det er et godt utviklingsmiljø til Perl, med integrert støtte for subversion.

Som dokumentasjonsverktøy har vi valgt å bruke LaTeX.

2.3.3 Faglige forutsetninger

AlgDat - algoritmer [TODO: Finne fram noen fine algoritmer fra prosjektet, og relatere dem til algoritmer og datastrukturer]

Lineær Algebra / Matte 200 — vektorer og lineære transformasjoner..

Systemutvikling –skrive noe om at vi har fått kjennskap til områder innenfor systemutvikling, samt progvamvaredesign.

Siden vi har måttet tilegne oss mye ny kunnskap om forskjellige språk og moduler, og vi har måttet lære oss syntaks så fort som mulig har alle tidligere programmeringsfag vi har hatt hjulpet oss med dette. Siden vi da har vært borti flere typer språk, har vi en lettere forståelse for generell programmering og dette har hjulpet oss i å forstå Perl, VRML og de respektive modulene mye kjappere enn hvis vi ikke hadde hatt noen av disse fagene. Fagene Operativsystemer og Unix og Nettverks- og Systemadministrasjon har gitt oss innblikk i Perl-programmering og administrasjon av Unix-tjenester.

Dette har hjulpet oss i programmeringen, samt til å forstå mange av konfigurasjonsparametrene som forekommer i profilene. Dermed har det blitt lettere å velge ut komponenter til visualisering. For eksempel forteller noen av profilene at en maskin er Apache- eller PostgreSQL-servere, og uten fagene hadde vi hatt liten kunnskap om hva dette sto for.

Relasjonsdatabasefaget har gitt oss grunnleggende kunnskap om databaser, da i sær MySQL og databasedesign.

2.3.4 Hva måtte vi lære oss

Gruppa hadde på forhånd ingen erfaring med visualisering eller 3d-programmering. Så vi måtte bruke mye tid på å lære VRML og 3d-tankegang, samt få en bedre forståelse av visualiseringsteknikker og utnytte dette i vårt prosjekt. Vi har heller ikke brukt Latex som dokumentasjonsverktøy før, og ser på denne prosjektperioden som en fin anledning til å lære oss dette.

2.3.5 VRML og tredimensjonal geometri

**TODO: Finne bedre tittel på avsnittet Koordinatsystemer og vektorer En 'verden' i VRML har en struktur der alle objekter er plassert i et globalt koordinatsystem. I tillegg kan det defineres transformasjonsobjekter, som danner lokale koordinatsystemer for de objektene som tilhører dette. Den globale posisjonen til et objekt kan dermed avhenge av plasseringen i flere lokale koordinatsystemer.

Muligheten til å bruke lokale referanser gjør det enklere å plassere grupper av objekter i forhold til hverandre siden man først kan plassere objekter som hører sammen i forhold til hverandre for deretter å plassere gruppen av objekter globalt.

TODO**:Figurer???

2.3.6 Beregning av 3D-posisjoner

En viktig del av visualiseringen har vært å finne algoritmer som kan brukes til beregning av posisjoner i det tredimensjonale rommet. Siden posisjonene i VRML er oppgitt i kartesiske koordinater, har det vært hensiktsmessig å benytte vektorer for å beregne plasseringen av objektene.

I visualiseringen har vi funnet det nødvendig å ha to metoder for å generere villkårlige koordinater. Den ene genererer en villkårlig posisjon innenfor en boks med gitte dimensjoner, og den andre genererer en posisjon mellom to

sfærer som begge har sentrum i origo. Begge metodene returnerer en array på tre elementer som representerer en tredimensjonal vektor.

Metoden for beregning av posisjon innenfor en boks returnerer en array med villkårlige verdier mellom null og angitt maksimalverdi for henholdsvis bredde, høyde og dybde.

Metoden for beregning av koordinater mellom to sfærer er litt mer komplisert. Den trenger to parametere som angir radius på den indre og den ytre sfæren. I tillegg tar den en parameter som angir hvilken avstand som kan brukes til å gi en skalering av avstanden mellom to posisjoner. Metoden genererer først en tilfeldig vektorlengde som ligger mellom de to grensene angitt. Denne representerer radius r i likningen for en kule med senter i origo der x, y og z er aksene:

$$r^2 = x^2 + y^2 + z^2 \tag{1}$$

X-verdien settes først til et tilfeldig tall slik at $x \in [\, 0, r \,]$. Deretter settes y tilfeldig slik at $y \in [\, 0, \sqrt{r^2 - x^2} \,]$ før z-verdien til slutt beregnes ut fra r, x og y:

$$z = \sqrt{r^2 - (x^2 + y^2)} \tag{2}$$

2.3.7 Visualisering

En oppgave vi hadde var å prøve ut ulike visualiseringsteknikker, for å se hvilke som kunne passe til forskjellige data.

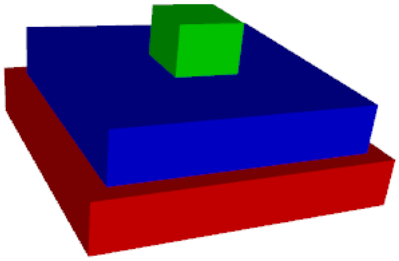
Resultatet vil bli en visualisering av forskjellige grupper (clustere) av datamaskiner, heretter kalt noder, der nodenes konfigurasjons(u)likhet kommer klart frem, for eksempel gjennom nodenes form, farge og posisjoner i et tredimensjonalt rom.

For å få litt inspirasjon, pekte vår veileder oss til en masteroppgave [3] der det var presentert mange forskjellige måter å visualisere data på. Noen teknikker vi ønsket å prøve var blant annet:

- **Information pyramids** (fig. 1)

  Her vises informasjon i flere lag som et hierarki. Eksempelvis kan man tegne opp et lag underst som representerer alle noder (A), trinn 2 består av noder fra A som oppfyller et bestemt kriterie B, slik at Trinn 2 $= A \cap B$.

  Det er da mulig å få et inntrykk av hvor mange noder som oppfyller forskjellige kriterier.



Figur 1: Information pyramid

- **Scatter plot** Dette består av noder som er spredt rundt i et område. Posisjonen til en node kan fortelle noe om nodens egenskaper, litt som en graf.
- **Heatmaps** Ved å legge noder i et plan, og deretter fargelegge de delene som oppfyller et kriterie, vil man kunne få noe som minner om et kart, der interessante områder blir uthevet.
- **Tree visualization** Kan vise nodens indre struktur, mulig å sammenligne to eller flere trær. Jfr. nodenes struktur, vil dette ikke være et typisk binærtre. En mulighet vil også være å forsøke å visualisere en eller flere standard-noderbasert på statistisk analyse av dataene, og så sammenligne enkeltnoder opp mot standarden.

Vi ønsker å forsøke å kombinere flere av disse teknikkene der dette er mulig, for å kunne trekke ut informasjon og sammenhenger som er vanskelig å finne ut av ellers.

2.3.8 Fargeproblematikk

17/4-08 Vi ønsker å generere X antall farger, der X er de forskjellige verdiene til et kriterie.
I VRML kan fargene skrives som en tredimensjonal vektor der rød, grønn og blå komponentene har verdier mellom 0 og 1.
Utfordringer: Menneskeøyet er ikke spesielt godt egnet til å se forskjell på farger, slik at vi bør tilstrebe at fargene som brukes er nokså forskjellige. Dette blir naturligvis vanskeligere dersom vi har svært mange forskjellige kriterier.
En løsning: Filtrere ut informasjon – ta bort noen grupper når man studerer visualiseringen, for å ikke ta feil. Vi bør også sette en max-grense for antall

farger – Over et visst punkt er det ikke lenger praktisk mulig å se forskjell på fargene, og da er det ikke hensiktsmessig å bruke farger på kriteriet. CA 20 - 30 stk???

Da vi har svart bakgrunn, kan vi heller ikke bruke veldig mørke toner, fordi disse ikke er tydelige nok.

Første utkast : Satte statiske farger og brukte de.

Deretter: Generering av farger: Bruke tre nestede for-løkker, øke en komponent med step"0.2...

Regnbue – generere roggbif – deretter øke en ting"og legge til ,gjenta loop.

TODO: sette inn algoritmen for fargevalg via vektorer og dens lengde...

2.3.9 Hva skal vi visualisere?

Dette var noe vi måtte bruke en del tid på å finne ut. For det første trengte vi å få oversikt over dataene, for å se hva de representerte. Etter hvert ble det klart at for å avgrense oppgaven, begrenset vi oss til components-seksjonen i filene. **TODO: Flytte dette opp i XML-fil delen?** Til å hjelpe oss å analysere, laget vi et perl-script som ved hjelp av libXML gikk gjennom alle profil-filene, og fant antallet forskjellige komponenter, samt hvor mange maskiner som hadde disse. Dette gjorde det enklere å få oversikt over hvilke komponenter som var i bruk. Etter at vi hadde lest gjennom dokumentasjonen over komponentene (link?) og sett hvilke parametere som var vanlige, satte vi opp en liste over hvilke aktuelle felter i samarbeid med oppdragsgiver / veileder. Disse var i første omgang: **Note to self: Skal denne listen være her???**

- inv / os -> Operativsystem
- inv / manager -> Ansvarlig administrator
- profile / group -> Gruppen som maskinen hører til. Kan si noe om funksjonen til maskinen
- inv / location -> Fysisk lokasjon
- xinetd / enableservices -> Tjenester som benyttes av xinet.d. Ikke mange profiler som har denne.
- apache -> Betyr at maskinen kjører en webservertjeneste. Kun 17 av 43 har apacheconfig
- mysql -> Betyr at maskinen har mysql daemon kjørende

- dhcpd -> Betyr at maskinen kjører en dhcpdaemon
- postgresql -> Betyr at maskinen har en postgresql daemon kjørende
- rsync -> Betyr at maskinen har en rsync tjeneste oppe
- subversion/cvs -> subversion tjeneste kjører
- samba -> sambaserver kjører
- inv / model -> forteller om maskinmodellen (eks: Dell OptiPlex 9700)
- network / gateway -> default gateway for maskinen.

2.3.10 XML-parser

For å tolke XML-filene, trenger vi moduler som kan hjelpe Perl til å skjønne XML-struktur slik at vi får de dataene vi vil ha. Vi valgte først å bruke DOM (Document Object Model) til å tolke disse filene. Denne modulen er noe vi hadde liten forkunnskap om, så vi måtte tilbringe en god stund foran manualer for å forstå hvordan vi skulle ekstrahere data med den. Når vi ble godt kjent med syntaksen lagde vi et testskript i Perl, som vi testet på et lite sett med filer. Da vi senere skulle parse alle filene, viste DOM seg å være veldig treg, og minneforbruket var så stort at våre maskiner kræsja. Vi løste midlertidig problemet med minneforbruket ved å kalle en doc-¿dispose() metode for hver fil vi hadde lest inn, fordi garbage collector ikke selv gjorde dette. Men fortsatt var vi ikke fornøyd med hastigheten. Vi søkte etter en ny modul, og fant LibXML som innfridde de forventninger vi har til en XML-parser. Dette er en XML parser til Gnome biblioteket (et Unix-grensenitt), og viste seg å være utrolig kjapp. Syntaksen på XML-spørringene er litt annerledes fra DOM, så noe omskriving måtte til. Som nevnt trenger vi å parse et stort antall XML-filer og LibXML viste seg å være raskere og mer effektiv enn DOM. Vi testet de to forskjellige modulene på vårt datasett, og det viste seg at LibXML er ca ti ganger kjappere, og brukte ca. en prosent av systemminne, mens DOM forsynte seg av rundt 70 - 80 prosent.

DBI (Database Interface) er et databasegrensesnitt for Perl, som vi valgte å bruke for å lage og eksekvere spørringer til en database. Vi hadde liten forkunnskap om modulen, men syntaksen var lett å skjønne. Alle på gruppen har hatt et relasjonsdatabasefag, og vi ble enige om at MySQL (My Structured Query Language) skal være vårt databasespråk. Dette fordi vi er familiære med syntaksen og skolen tilbyr gratis MySQL-servere. DBI-modulen kobler opp mot en MySQL-database på en enkel måte, og spørringer og svar er enkelt å fremstille og hente.

2.3.11 Databasestruktur

Etter å ha analysert XML-filene, kom vi fram til at det ville være hensiktsmessig å opprette en tabell for hver konfigurasjonsdel som vi ønsker å bruke i components-delen. Et eksempel: i XML-profile-filene har vi som oftest en konfigurasjonsdel $< network >$. Vi genererer da en tabell network, med de feltene vi ønsker å benytte oss av. Som oftest er dette string-felter eller boolske verdier. Primærnøkkelen i disse komponent-tabellene vil være maskinnavn, samt dato, da konfigurasjonen kan endres over tid og vi ønsker å ta vare på endringer i konfigurasjon.

I noen tilfeller vil denne strukturen ikke være gunstig, slik at noen tabeller vil være bygget opp på en annen måte. TODO: sette inn eksempel på dette...

Vi ønsker også at systemet skal være utvidbart, så det bør være relativt enkelt for sluttbruker å velge nye felter i XML-profilene som utvider eksisterende tabeller, eller oppretter nye dersom tabellen ikke eksistererer.

2.3.12 Hendelsesflyt for import til DB

- Leser inn cfg fil
- cfg-fila skal inneholde:
  - Databaseinformasjon (username, pass, host, port, type of base)
  - Tabell- og kolonnenavn (inv / os) - attributt
  - Namespace over xml-filene
  - Path til zip-fil over profiler
- Tester databasetilkobling med parametre fra cfg
- Zipper opp profiler og legger de i en temporær katalog

Deklarer en array over tabell (hovedkomponenter - eksempel inv) Og i denne arrayen må det legges flere nye hasher over kolonnenavn (eksempel inv/os) Deklarer også en boolsk array over tabellene, som senere vil si i fra om operasjonenene har gått bra eller ikke Hvis det er noen som ikke har gått OK, så kopiereres ikke tabellene tilbake.

Opprett tabeller med kolonnenavn hentet fra cfg Hvis tabell eksisterer fra før av og kolonnenavnene ikke samsvarer: Kopier tabell og alter. Denne nye tabellen skal det legges data i, og senere skal den overskrive den gamle Hvis tabell eksisterer fra før av og kolonnenavnene samsvarer: Append table.

Løpe i gjennom filer (foreach ¡¿) Deklarer man rotnode, etc Finne llast_modified"og legge den i en variabel sub getLastModified(maskinnavn)

Hvis last_modified fra xml er nyere enn den i DB, fortsett Opprett en boolsk variabel som skal holde rede på om det blir gjort noen forandringer, $changed

Løpe i gjennom tabellparametre (hovedkomponenter)

Sjekke om parameteren eksisterer i xml-filen Bør sjekke nivået på parameteren som kommer inn, og at det i hele tatt er verdi der.

Finne ut verdien til parameteren, og legger denne som en value til tilhørende key i hash (f.eks. fc6til inv/os) Ikke glem attributten hvis vi skal ha den.

Hvis databasen allerede eksisterte, må vi selecte fra databasen alle dataene fra den maskinen vi holder på med og sjekke mot hashen om det er noen nye verdier. Hvis det er det, så må vi inserte en ny rad og gi $changed verdien true".

Sjekke $changed Hvis true" Oppdatere tabell last_modified med last_modified

Sjekke om det er noen false-verdier i den boolske arrayen Overskrive de gamle tabellene hvis alle er OK.

```
sub getLastModified
{
    my \$tempMachine = shift  ;

     Hente ut fra DAL last modified fra   denne maskinen
     Returnere denne
}
```

TODO: Lag en tabell last_modified , som inneholder id, maskinnnavn og lastmodified Denne må oppdateres hver gang det blir gjort en endring

Logge alle aktiviteter/forandringer som skjer ved hjelp av dette scriptet For eksempel, antall rader forandret, mange filer som var helt like, etc etc

*TODO: Sette inn ER-diagram.***

2.4 Utviklingsprosessen

2.4.1 Metodikk / prosessmodell

Gruppen bestemte seg tidlig for å benytte seg av en smidig utviklingsprosess, med mange iterasjoner og konstant utvikling av kravspesifikasjonene. Vi valgte å bruke RUP (Rational Unified Process) som prosessmodell, med innslag av XP (eXtreme Programming). XP-elementer ble valgt blant annet fordi vi måtte komme raskt i gang med programmeringen, for å gjøre oss kjent med språkene, vi synes også parprogrammering kan være gunstig innimellom.

Måten vi jobbet på for hver visualisering: Til å begynne med laget vi en

prototyp kun med VRML-editor på hva vi ønsket oss. Så gikk vi over til å lage et perl-script der vi laget metoder for å generere ulike vrml-elementer og skrev så resultatet fra kjøringen ut til fil. Deretter inspiserte vi den genererte koden og sjekket at den virket i en vrml-viewer. Ved syntax-feil rettet vi opp vrml'en og la evt. til kode manuelt, testet dette og gikk så tilbake til scriptet og endret / la til metoder her som så genererte ny vrml-kode.

2.4.2 Faser

Fasene i prosjektet fulgte RUP-fasen, med mange iterasjoner for å forbedre og utvikle koden.
Funksjonaliteten ble utvidet gjennom en inkrementell prosess, der vi startet med kun en visualiseringsteknikk og gjorde denne ferdig, før vi startet på en ny.

Innledning
Utforming
Bygging
Overgang

2.4.3 visualisering i praksis

Det første vi prøvde å visualisere var maskinnoder og hvilket os de har. For at vår visualisering skulle passe til forskjellig antall maskiner, telte vi opp det totale antall maskiner, m. Ved å ta kvadratroten av dette: $n = \sqrt{m}$ kunne vi lage en grid med n * n noder. Størrelsen på nodene ble av praktiske hensyn satt til 1 * 1 * 1 , og hver node ble satt inn med et mellomrom på 1. Dette gjorde at vi fikk en formel for høyde og bredde på denne visualiseringen, basert på antall maskiner m: $h = w = 2 * \sqrt{m}$

Dette trengte vi for å kunne beregne posisjonen for et viewpoint, altså hvor kameraetskal stå for å kunne vise hele scenen. Et viewpoint i VRML har som standard et synsfelt på $\pi/4$ grader, men kan settes fra alt til 0 - 2 π. Vi valgte å beholde standard, blant annet for å unngå distortion- dersom man øker eller minker for mye vil man få sammentrekninger eller fisheyeeffekt.

Med en vinkel på 90 grader var det enkelt å beregne hvor kameraet skulle stå. (Vi begynner i punkt 0,0,0). TODO: Illustrasjon??? Med en høyde h og bredde b, kunne vi bruke trigonometri til å beregne kameraets x,y, og z posisjoner. x og y settes til hhv b / 2 og h /2. z kan regnes ut ved å beregne $x/\tan 45 = x$ og for å få litt luft på hver side setter vi z til 3 * x.

Dette resulterte i en visualisering som den i fig.: **illustrasjon?**

Neste oppgave ble nå å lage en annen visualiseringsmetode for et kriterie. Vi valgte å ta utgangspunkt i maskinens gateway, fordi det var et felt som nokså mange maskiner hadde, og det var ikke så altfor mange forskjellige. For å vise tilhørighet, bestemte vi oss for å la maskinnodenes posisjon si noe om hvilken gateway de tilhørte. Sagt tydeligere: Vi tegnet opp gateways og posisjonerte nodene i nærheten av sin respektive gateway. Ved å ta utgangspunkt i det første scriptet, endret vi det til å hente inn de forskjellige gateway-addressene. Så lagde vi en grid basert på dette antallet og genererte en kule for hver gateway. Deretter hentet vi inn alle maskinnavn sortert på tilhørende gateway fra databasen. Nå kunne vi løpe gjennom listen over maskiner og tegne en firkant for hver maskin, og sette posisjonen til området rundt den respektive gatewayen.

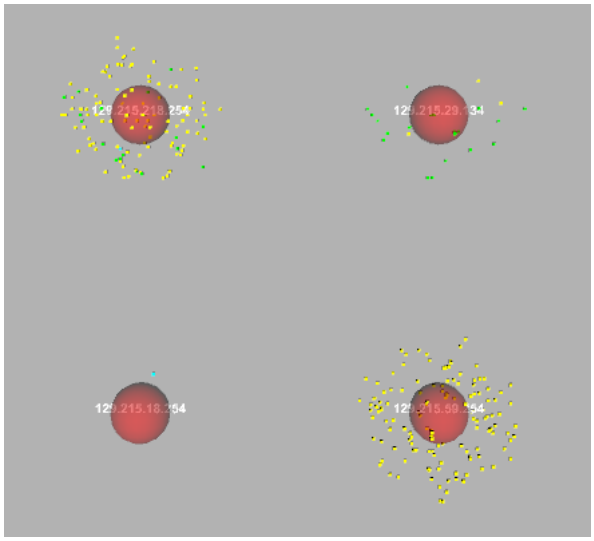
Da dette virket, endret vi det til å sette en tilfeldig posisjon for hver maskin, og deretter flytte det mot gatewayen som en animasjon, både for en penere effekt, men også fordi vi ønsket å prøve ut bevegelse i visualisering, da dette er noe menneskeøyet oppfatter spesielt godt. Så ved å prøve ut dette her, kunne vi bruke metodene senere i andre visualiseringer. For å få til dette, måtte vi opprette grupper (transforms) for hver gateway. Denne gruppen fikk så en posisjonsbeskrivelse med start og slutt-koordinater, hvor sluttkoordinatene er de samme som den tilhørende gateway-nodens koordinater. Alle maskinnoder som hadde denne gatewayen ble lagt inn i denne gruppen på et tilfeldig sted. En animasjon er bevegelse over tid, så vi måtte også lage en klokke (timer), og rute klokkens tikktil posisjonsbeskrivelsen. Så må denne rutes videre til sin respektive gruppe, som så flytter alle nodene i gruppen mot gatewayen. Maskinnodene selv ble også tildelt en lokal. tilfeldig posisjon rundt gatewayen, så ikke alle skulle bli liggende i en klump.

Når også dette var i orden, kombinerte vi os-visualisering og gateways, slik at nodene fikk farge etter hvilket os de hadde og ble posisjonert etter hvilken gateway de tilhørte, se eksempel figur 2.

Denne visualiseringen har vi valgt å kalle en gruppe-visualisering med to kriterier", fordi noder blir gruppert etter kriterier.

Etter dette ønsket vi også å ha mulighet til å filtrere bort uønsket informasjon, for eksempel ved at man kan slå visning av grupper av operativsystem av og på. På dette tidspunktet var ikke gruppene organisert på annen måte enn gateway, så for å få til å velge alle noder med et bestemt operativsystem, ble det nødvendig å organisere nodene annerledes.

Løsningen ble å lage nøstede grupper av maskiner, slik at vi opprettet en gruppe for hvert operativsystem. Inne i denne gruppen opprettet vi så nye



Figur 2: Group Visualisation

grupper over forskjellige gateways, og inne i disse ble maskinene plassert. Det er da en smal sak å få manipulert en gruppe med maskiner, og vi kan da filtrere ut maskiner basert på hvilket operativsystem de har.

2.4.4 Oppbygging av programmet

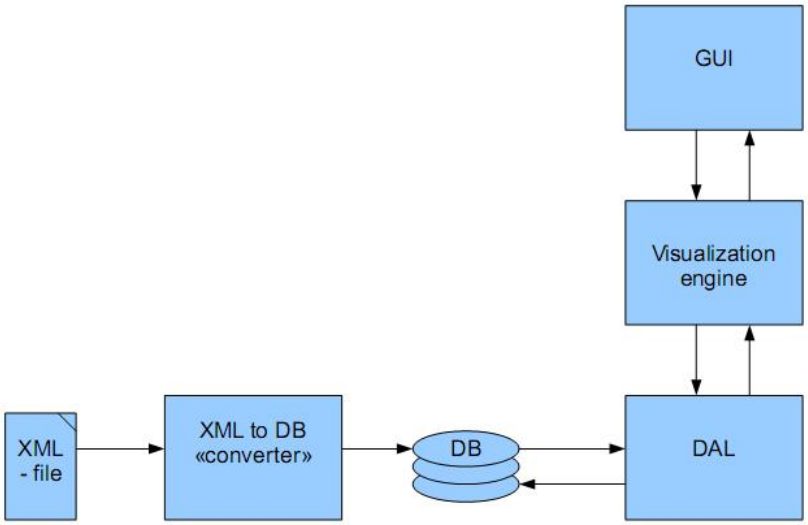
Programmet består av to hoveddeler: dataImport og dataVisualiserer. (fig. 3)

Vi har valgt å implementere en 3-lags struktur med DAL (Data Access Layer) for database-tilkobling, BLL (Business Logic Layer) for generering av VRML, og GUI (Graphical User Interface) for kommunikasjon / presentasjon for sluttbruker. Dette er gjort for å gjøre systemet utvidbart og generisk - eksempelvis vil det være mulig å gå over til en annen databasemotor ved kun å endre / bytte ut DAL. Videre har vi valgt å implementere løsningen som en Web-applikasjon, da kombinasjonen *AMP (Apache, MySQL, Perl) er en god og plattformuavhengig kombinasjon, samt at det ikke stiller store krav til klienten, som egentlig kun trenger å kunne vise HTML, javascript og VRML.

Også DAL og BLL er delt opp i mindre moduler for å gjøre det enkelt å legge til funksjonalitet.

dataImport-delen er ansvarlig for å lese inn nye datasett i form av XML-filer, og ekstrahere de feltene vi ønsker å trekke ut, for så å legge disse inn i database. Denne delen er så delt opp i moduler for hver komponent som skal trekkes ut, dette for å gjøre det enkelt å utvide importen til å gjelde flere komponenter / felter. Forhåpentligvis gjør dette også det lettere å endre

Figur 3: Skisse av systemets oppbygging



konfigurasjonen til å kunne importere andre XML-filer, dersom andre senere vil importere andre typer filer. Vi ville få til import til databasen så generisk som mulig, og fant ut at en konfigurasjonsfil vil erstatte våre dataImport-moduler på en god måte. Modulene vi tenkte oss i første omgang gir oss en god mulighet for å ekstrahere data, men det ville bli en for stor jobb for en potensiell systemadministrator og skreddersy perl-moduler etter behov. Konfigurasjonsfilen er tenkt slik at den vil ta i mot en kriterier fra en web-applikasjon, for eksempel databasetilkoblingsinformasjon, lokasjon og namespace av XML-filene, og hvilke komponenter som systemadministratoren senere vil visualisere. Perl-scriptet XML_to_DB leser inn informasjon fra konfigurasjonsfilene, lokaliserer dem, oppretter databasetilkobling og ekstraherer ut ønsket komponentinformasjon fra filene og legger disse inn i databasen. I DBMETODER-modulen ble det opprettet to nye metoder, én for å opprette en ny tabell og én for å injiserer nye verdier inn i databasen, og disse er begge generiske.

TODO: Kjøre tester som sammenligner de generiske metodene mot de som ikke var det. Bør ta lengre tid for de generiske

De generiske tabellene som blir opprettet er ikke normalisert til høyeste nivå med hensikt. Med tanke på at vi senere ville få nye datasett, måtte vi sette opp en ny datastruktur i databasen. I hver eneste tabell vil det være et kolonnenavn 'sist_oppdatert', som sammen med maskinnavn vil danne primærnøkkelen for de respektive radene. Alle XML-filene har et obliga-

torisk element kalt 'last_modified' , og denne vil vi bruke for å sjekke om filen har grunnlag for å muligens bli importert til databasen eller ikke.

Når vi importerer data til databasen, er vi som oftest kun ute etter noen få komponenter. Selvom 'last_modified' er oppdatert siden sist, betyr ikke det nødvendigvis at de komponenter som det er ønske for skal bli visualisert har blitt forandret. Derfor måtte det også implementeres en kryssjekk, som ser om det er ren redundant data som vil bli injisert. Hvis det er blitt noen forandringer, vil det opprettes en ny rad med nye data.

Etter som vi har blitt mer kjent med datasettene (XML-filene) og vi har fått et klarere bilde over hvordan databasen kommer til å se ut, fikk vi nye datasett fra arbeidsgiver. Disse nye datasettene var over et tidsløp på nesten 3 måneder, og vi trengte nå å oppdatere våre metoder for å injisere nye verdier til databasen.

Tidligere har vi brukt databasemetoder i DBMETODER-modulen, og sendt med verdier til den for at den skal gi omdanne disse til SQL-spørringer. I vårt tilfelle har vi litt over 1000 XML-filer som vi vil ha inn i databasen, og til nå er det to hovedkomponenter vi bruker fra databasen til visualisering (inv og network). DBMETODER er bygd opp slik at den åpner en ny databasetilkobling for hver eneste spørring, og det resulterte i at ved kjøring av XML_to_DB ble det gjort omtrent 2000 databasetilkoblinger under kjøring på vår filbase. XML_to_DB bruker totalt 156 sekunder på å legge inn data til tre tabeller fra 1047 filer. Vi vil redusere denne tiden, og vil prøve oss på å objektorientere DBMETODER-modulen slik at vi har en databasetilkobling som vil være åpen fra scriptets start til slutt. Vi kaller denne nye modulen xtd.pm, og kopierer over de generiske databasemetodene fra DBMETODER-modulen. Det ble gjort tre sammenligninger, én hvor vi satt inn et helt nytt datasett (1047 filer) inn i en ren database med én komponent, én hvor vi satt inn et helt nytt datasett (1047 filer) inn i en ren database med tre komponenter, og én hvor vi la inn et nyere datasett over et gammelt. Det viste seg på de to første testene at tidsdifferansen på sammenligningene var under to sekunder, og det så ut til at det ikke spilte noen rolle om databasetilkoblingen ble åpnet for hver spørring eller om den var åpen hele tiden. På den tredje testen derimot, var det en signifikant tidsdifferanse. I denne testen ble det også sjekket for redundant data, for hver eneste injisering måtte man først kryssjekke om det var noe nytt som skulle inn i databasen. Ved hjelp av DBMETODER-modulen ble det brukt 71.98 sekunder. Ved hjelp av xtd.pm ble det brukt 42.05 sekunder. Da er det tre faktorer som spiller inn for at vi har valgt å bruke en objektorientert modul til importering av data: Det er penere og ryddigere kode, det er kjappere og x. Når dette systemet er i gang, vil scriptet bli brukt mest til å importere et nytt datasett over et annet og derfor er en OO-løsning mest å foretrekke her.

Noen komponentverdier i XML-filene inneholder tegn som ødelegger SQL-spørringene i xtd-modulen, og det har vi måttet vise hensyn til. Ved at vi har prøvd nå til sammen rundt 80 sett med XML-filer (rundt 80 000 filer) inn til databasen er det to tegn som kan gå igjen, ' og ; . Løsningen til nå er å bruke substitusjon i xtd-modulen, og fjerne disse tegnene. Forhåpentligvis vil vi senere utvikle en bedre metode for dette, som vil sjekke for andre ødeleggende tegn slik at vi unngår SQL-injeksjon.

Datavisualizer:

dataVisualiserer-delen har ansvaret for å hente data fra databasen og generere VRML som så blir presentert for brukeren.

***TODO: Klassediagram GUI:

Vi ble enige om at vi ville ha et brukergrensesnitt som er enkelt og funksjonabelt, hvor hovedfokuset skal være på visualiseringsfilen. Med en browserbasert løsning har vi muligheten til å gi et enkelt grensesnitt til bruker, og vrml fila kan legges inn og oppdateres kontinuerlig. Vi valgte å bruke en av Perl's moduler, CGI (Common Gateway Interface), til å lage dette. CGI kommuniserer med Perl og Apache på en god måte, og siden det er laget i Perl trenger vi ikke introdusere nye språk / teknologier i brukergrensesnittet.

Hendelsesforløpet ved å få fram en visualisering i GUI er tenkt slik (gruppevisualisering her):

- Velge visualiseringsteknikk
- Velge hvor mange kriterier man har lyst til å legge grunnlag for
- Velge tabeller og kriterier
- VRML-filen blir lagt til, og kommer opp på siden

Det har oppstått et par utfordringer underveis. Et problem ved utvikling av førsteukastet til websiden er at vi må skrive koden med absolutte stier for at Apache skal forstå hvor de respektive filene ligger. Websiden trenger flere moduler som vi har skrevet, og disse krever igjen andre moduler vi har skrevet - og for å få de rette modulene implementert i websiden har vi måttet skrive inn absolutte stier. Dette håper vi senere vil la seg løse ved at hele systemet blir gitt i en virtuell maskin, slik at de absolutte stiene faktisk er absolutte hvor enn systemet kjøres fra.

Selve VRML-filen (.wrl) som ble lagt til i websiden ble ikke oppdatert av browseren eller apache. Ved første kjøring av websiden på en ren Apache-server fungerte det å hente opp den VRML-fila som akkurat ble generert

av GUI, men ved neste visualisering (med helt andre kriterier) hentet den fortsatt opp den første genererte VRML-fila. Ved rensking av cache og temporært minne håpet vi det ville la seg ordne, men samme problem oppsto fortsatt. For å løse dette, må vi lage en ny VRML-fil for hver gang den blir generert. Dette er ingen god løsning, kun en workaround, og vi ser helst at den blir løst på en annen måte i neste utkast.

2.4.5 Strukturen i gruppa

- miljø, arbeidsgiver, veileder

2.4.6 Hva var X-tra vanskelig?

2.5 Kravspesifikasjon

Hovedmålet med prosjektet er å ekstrahere data og visualisere grupper av maskiner basert på konfigurasjons(u)likhet. Vårt system skal kunne:

- Lese inn de dataene vi har spesifisert fra XML-profilene og legge inn i en database.
- Hente de dataene vi har spesifisert fra databasen.
- Visualisere data på en eller flere hensiktsmessige måter.
- Systemet skal være tilgjengelig som åpen kildekode.
- Visualiseringen vil gjøres med VRML i kombinasjon med javascripts.
- Programmeringsspråk vil være Perl for ekstrahering av data og generering av VRML.
- Brukerdokumentasjon, kildekode samt deler av sluttdokumentasjonen må være tilgjengelig på engelsk.
- Det er ønskelig at systemet er plattformuavhengig.
- Systemet bør være generisk og utvidbart.
- Hva har den betydd, og hvilken rolle har den spilt for utviklingen.
  I begynnelsen hadde vi en svært løs kravspesifikasjon. Underveis i prosjektperioden la vi til nye krav og raffinerte de eksisterende kravene, i samarbeid med oppdragsgiver. Dette ble gjort fordi vi trengte kunnskap om både verktøyene og konfigurasjonsfilene, for å kunne sette realistiske krav, som vi så forsøkte å innfri så raskt som mulig, før vi gikk en ny runde med videreutviklingen av kravene våre.

  – gruppa

  - produkt

- Endringer ....

2.6 Resultat

Tolking av resultatet.

3 Avslutning

test av kilde: [1]

- Eget utbytte
- Oppsummering
- Konklusjoner
- Refleksjoner - Hva kunne vært gjort annerledes
- Hva kan det brukes til? Fremtid
- Hva syns oppdragsgiver om produktet vårt?
- Skal det brukes videre?

Referanser

[1] *VRML International Standard ISO/IEC 14772-1:1997.* http://www.web3d.org/x3d/specifications/vrml/ISO-IEC-14772-VRML97/.

[2] Paul Anderson. The complete guide to lcfg.

[3] Werner Putz. The hierarchical visualization system. Master's thesis, Graz University of Technology, 2005.