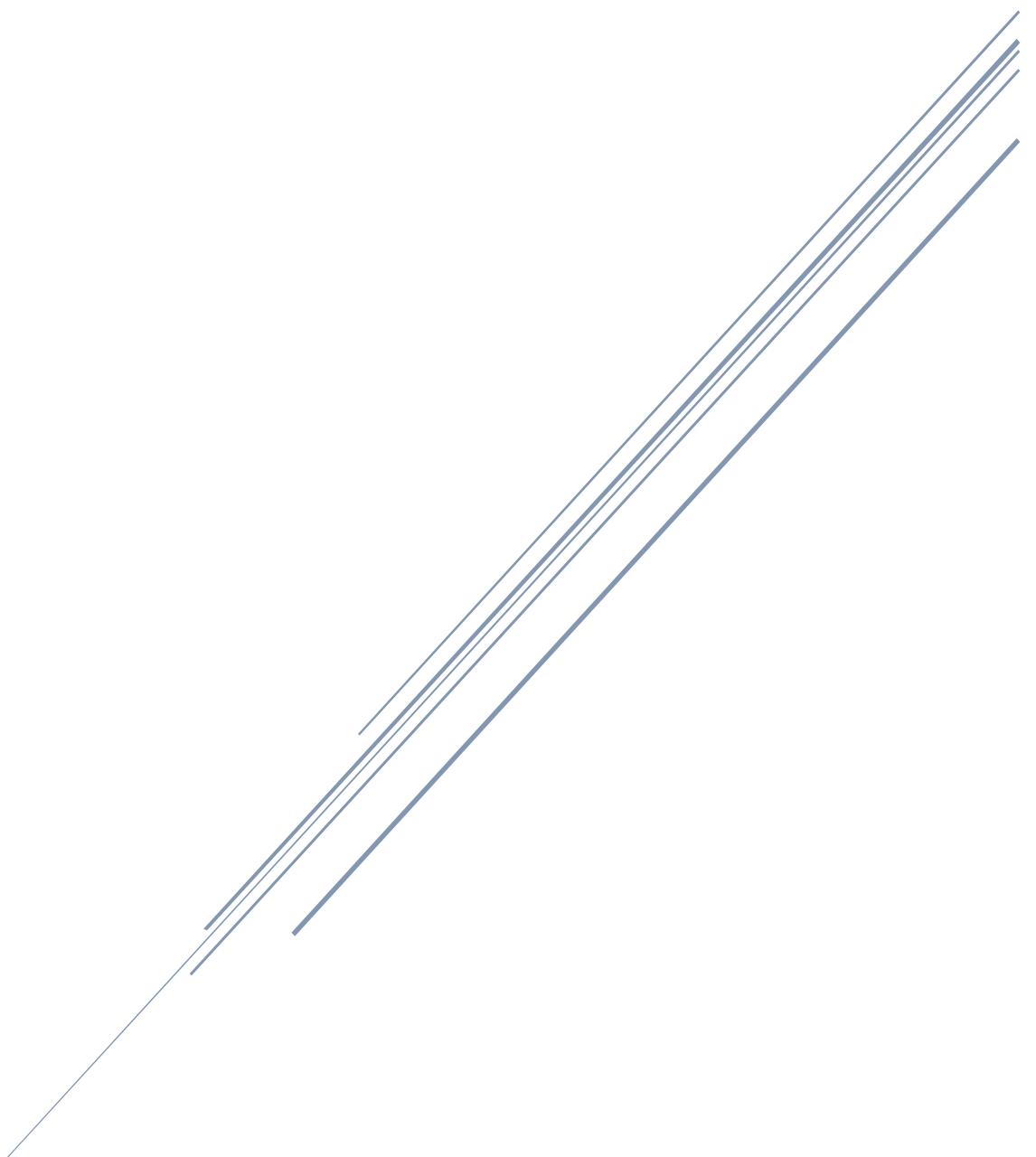


# PROJEKT GRUPPERUM

Teknologirapport



University College Nordjylland  
Gruppe 2 – David, Dina, Emil & Mark

## Titelblad



**Sted:** University College Nordjylland, UCN

**Uddannelse:** Datamatiker 3. semester i Aalborg

**Emne:** Teknologirapport

**Gruppe:** 2

**Titel:** Projekt Grupperum

**Afleveringsdato:** 16. december 2015

**Vejleder:** Rasmus Christiansen Knap

**Sider:** 28

**Normalsider:** 20

### Abstract:

At UCN, students have the opportunity to use group rooms for a great environment with little to no disruption. Unfortunately, UCN provides only a few couple group rooms for rent, and they do this manually. That is a problem: Many groups cannot get a group room. We want to find a better solution to the current problem with .NET technology. We envision people renting group rooms through a website, leaving UCN personal to only administrate special cases. This report suggests a solution to this problem, even if there is no group rooms left. We developed an algorithm to sort remaining group into empty classrooms.

**David Mathias Kolind, 1033937:** \_\_\_\_\_

**Dina Sandvad, 1034352:** \_\_\_\_\_

**Emil Andersen, 1034514:** \_\_\_\_\_

**Mark Hjorth Sørensen, 1033933:** \_\_\_\_\_

## Indhold

<b>Titelblad.....</b>	<b>2</b>
<b>Abstract:.....</b>	<b>2</b>
<b>Forord .....</b>	<b>5</b>
<b>Indledning .....</b>	<b>6</b>
<b>Problemformulering .....</b>	<b>6</b>
<b>Krav .....</b>	<b>6</b>
F for functional .....	6
U for usability .....	7
R for reliability.....	7
P for performance .....	7
S for supportability .....	7
<b>Arkitektur .....</b>	<b>7</b>
<b>Domænemodel.....</b>	<b>7</b>
Tannenbaums definition .....	8
Lag.....	9
<b>Klient lag .....</b>	<b>11</b>
<b>Middleware .....</b>	<b>13</b>
<b>Valg af middleware .....</b>	<b>13</b>
<b>WCF.....</b>	<b>14</b>
<b>Algoritme til fordeling .....</b>	<b>17</b>
<b>Algoritmen i C# .....</b>	<b>22</b>
<b>Test .....</b>	<b>24</b>
<b>Samtidighed .....</b>	<b>25</b>
<b>ACID .....</b>	<b>25</b>
Atomicity.....	25
Consistency.....	25
Isolation .....	25
Durability .....	25
<b>Håndtering af samtidighed.....</b>	<b>25</b>
<b>Atomicitet.....</b>	<b>26</b>
<b>Sikkerhed .....</b>	<b>27</b>
<b>Firewall .....</b>	<b>27</b>
<b>Router .....</b>	<b>27</b>
<b>Authentication.....</b>	<b>28</b>
<b>Kryptering.....</b>	<b>28</b>
<b>SSL (Secure socket layer).....</b>	<b>28</b>
<b>VPN.....</b>	<b>28</b>
<b>Konklusion.....</b>	<b>28</b>
<b>Kildehenvisning.....</b>	<b>29</b>

**Bilag.....29**

### **Forord**

Denne rapport er skrevet af datamatiker studerende på UCN, David Mathias Kolind, Dina Sandvad, Emil Andersen og Mark Hjorth Sørensen, som et eksamensprojekt på 3. semester.

Projektarbejdet er påbegyndt i efteråret 2015 og afleveret 16. december 2015.

Denne rapport har til formål at samle 3. semester fagene "Programmering" og "Teknologi" om en opgave der udfordrer den akademiske forståelse der er lagt til grund for semesteret.

Projektets sværhedsgrad opnås ved udfordringen i at bygge et distribueret system til web og dedikeret klient indeholdende komplekse problemstillinger som samtidighed og atomicitet.

Rapporten beskriver projektets formål, det system som vil kunne løse formålet og den del af systemet som blev implementeret. Med udgangspunkt i det byggede, forklares og diskuteres teknologier, deres fordele og ulemper samt de valg vi har foretaget. Til slut fremlægges konklusionen i forhold til problemformuleringen.

Hensigten er at synliggøre den viden gruppen har gjort brug af til at bygge et system i C#, .NET og SQL.

## Indledning

Projektet bygger på ideen om et system til udlejning af grupperum, som kan forbedre måden hvorpå det foregår på UCN nu, hvor studerende skal møde op og bede en medarbejder manuelt undersøge mulighederne for at få et grupperum og foretage registreringen.

På grund af manglen på grupperum, er der et stort pres på denne service. Studerende skal kunne møde tidligt mandag morgen og stå i kø, for at have en chance for at få et lokale i den kommende uge. Medarbejderne bruger en del tid på registreringen og presset på medarbejderne er unødigst stort da de også har opgaven med at afvise de mange, som ikke kan få grupperum.

## Problemformulering

Hvordan kan vi lave et system, som de studerende selv kan tilgå, oprette grupper og leje lokale igennem, så udlejningen bliver lettere tilgængelig og forhåbentlig mere tilfredsstillende for både studerende og ansatte? Hvordan kan et system give de studerende mulighed for, at vælge lokaler ud fra bestemt inventar i lokalerne og også tage hensyn til at en gruppe kan leje et bestemt lokale i f.eks. 5 dage i træk, så de kan indrette rummet efter deres behov?

Hvordan kan de UCN-ansatte administratorer få et overblik over hvem der har lejet hvilke lokaler og få rettet til at slette en gruppe fra et lokale, hvis gruppen misligholder aftalen for lokalebrugen?

## Krav

Som sagt er den manuelle grupperumsregistrering uhensigtsmæssig. En IT-løsning er en opslagt mulighed for forbedring. I dette projekt lægges der i høj grad vægt på bestemte funktioner, men det skal ikke overses, at der ved planlægning af ethvert system, bør tages hensyn til en mængde krav. Af den grund benyttes herunder metoden FURPS, som også vil bidrage til et overblik over relevante krav og overvejelser omkring dette system.

### F for functional

Omhandler de funktionelle krav. Af funktionelle krav kan nævnes, at de studerende, over web, selv skal kunne logge på, oprette en gruppe af klassekammerater ud fra deres klasse og ud fra denne gruppe, booke grupperum. Dermed har systemet behov for en database, som bl.a. opbevarer data om de studerende, deres klasser, deres grupper og de bookninger de foretager.

Bookning af grupperum skal kunne foretages ud i fremtiden, men dog inden for en begrænset tidshorisont som f.eks. en måned frem. Det skal være muligt at booke et bestemt lokale i flere dage i træk.

Er der ikke gruppelokaler nok til alle de grupper som ønsker et, er der mulighed for at blive skrevet på en venteliste. Ventelisten tages op en gang om ugen og fordeler grupper i klasselokaler, med op til 3 grupper pr lokale, og fordelingen gælder kun en uge frem.

For både grupperum og klasselokaler gælder det, at de har forskelligt inventar, som de studerende kan gøre brug af. Inventar som whiteboards, monitors og projectors.

Systemet skal håndtere gruppernes ønsker til inventar og sammenholde dem med lokalernes inventar. Et sådant system rummer udfordringer i forhold til at bygge et fornuftigt distribueret system, som kan tilgås over web samt fra en dedikeret klient og det rummer udfordringer hvad angår samtidighed og atomicitet.

Desuden skal ansatte kunne oprette lokaler mv. og administrere hele systemet via en dedikeret klient. Denne del af systemet nedprioriteres kraftigt i dette projekt aht. de studiemæssige krav og tiden til rådighed.

Systemets funktionelle krav og håndteringen af information belyses nærmere med en domænemodel senere.

#### **U for usability**

Omhandler brugervenligheden. Som altid, med systemer som henvender sig til en stor bred brugerskare, bør det tilstræbes at skabe et intuitivt "selvførlærende" program. Der må ikke være behov for en detaljeret guide for at forstå, hvordan programmet fungerer.

#### **R for reliability**

Omhandler pålidelighed. Et system der gemmer data på hardware, kræver samtidig en funktion der rydder automatisk op i data, for ikke at oversvømme hardwaren med unødig tung data.

#### **P for performance**

Omhandler ydeevnen. Igen et vigtigt krav at tage hensyn til. Derudover skal der være en indikation af om at programmet arbejder.

#### **S for supportability**

Omhandler vedligeholdelse og muligheder for at udvide og udvikle på programmet. Sigende klasse og metodenavne samt relevant dokumentation i interface gør det nemmere for andre programmører at overtage arbejdet. Desuden er det målet at skrive god overskuelig objektorienteret kode.

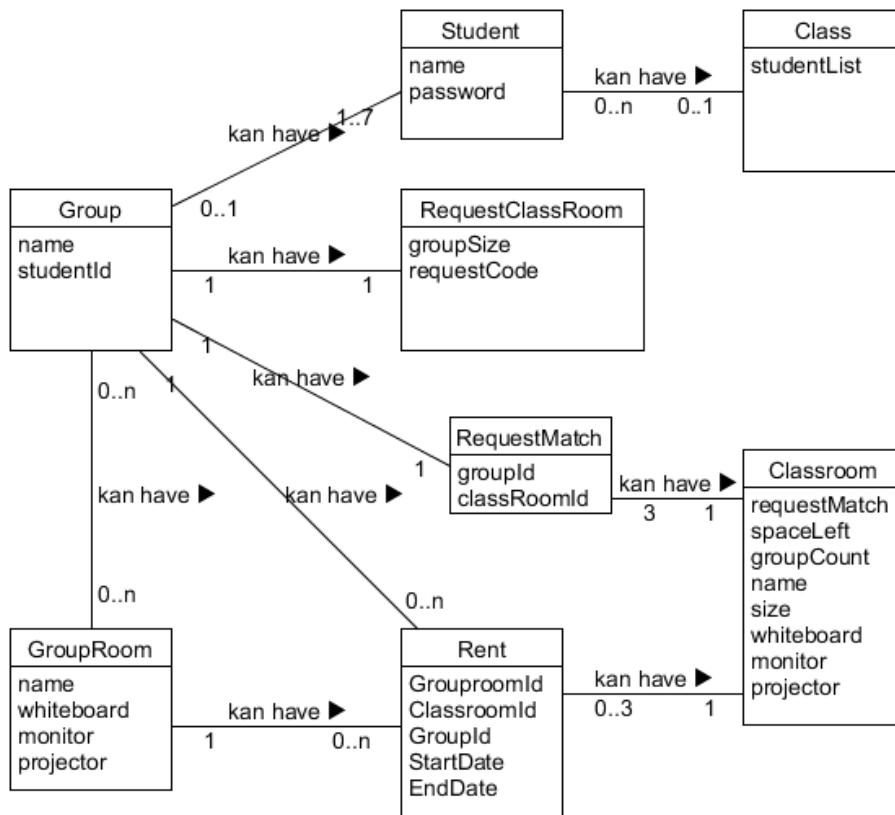
Arkitekturen er desuden væsentlig i forhold til at sikre et godt grundlag for vedligeholdelse og udvidelsesmuligheder.

## **Arkitektur**

I et distribueret system som dette er arkitekturen aldeles vigtig for, at det kan yde optimalt. F.eks. er det vigtigt, at 3-delingen er i orden og at 3-lags arkitekturen overholderes, for at give kommende programmører en lettere tilgang til programmet.

## **Domænemodel**

Domænemodellen for projektet kan ses nedenfor på Figur 1.



Figur 1 – Domænemodel.

På domænemodellen, som ses på Figur 1 ovenfor illustreres domænet ud fra en logisk forretningsmæssig tilgang som også en productowner kan forholde sig til. Almindeligvis vil de klasser der er associeret og som "kan have" hinanden, også have en instans af den anden klasse som attribut. Grunden til at dette ikke forekommer mange steder på vores denne domænemodel er, at når en gruppe booker et lokale bliver dette gemt direkte på databasen, og derfor får et objekt af Group ikke som objekt tilknyttet noget GroupRoom men logisk set hænger de alligevel sammen.

I det tilfælde at en gruppe ikke kan få et grupperum, kan de i stedet blive skrevet op til en plads i et klasselokale. Når dette sker bliver der oprettet en RequestClassRoom (dvs. en forespørgsel om plads i et klasselokale), som har en Group tilknyttet.

På et givet tidspunkt hver uge køres algoritmen igennem, som fordeler klasselokalerne ud fra forespørgslerne, som bliver hentet fra en venteliste på databasen. Når der findes et match imellem en gruppens request og et lokale, oprettes et nyt objekt RequestMatch, som har lokals ID samt gruppens ID. Det vil aldrig forekomme at man søger på et klasselokale for at finde en gruppe som arbejder der. En sådan søgning vil i stedet skulle gennemgå listen af Rents.

#### Tannenbaums definition

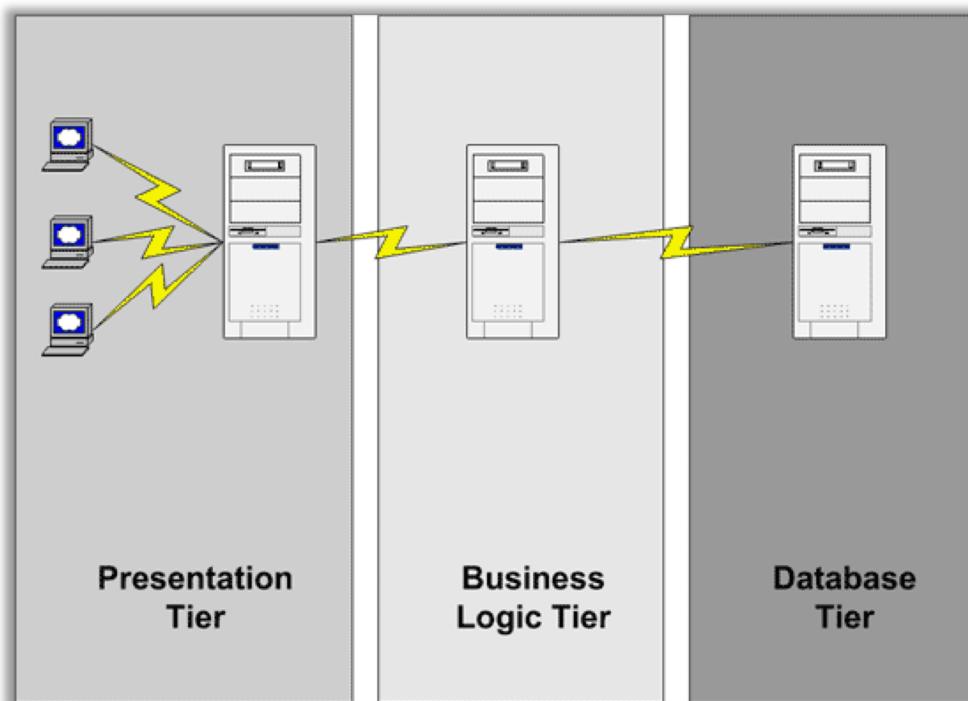
Et distribueret system defineres af Andrew S. Tannenbaum som:

*"A collection of independent computers that appears to its users as a single coherent system"<sup>1</sup>*

Hermed siger Tannenbaum, at det distribuerede system fungerer på en mængde selvstændige maskiner, som ikke har delt hukommelse, men at det alligevel fremstår som det ene og samme system.

### Lag

I et distribueret system er arkitekturen aldeles vigtig af hensyn til flere ting. Bl.a. er det relevant at forholde sig til, at systemet skal kunne renoveres, og med en 3-delning i opbygningen er fremtidige programmører godt hjulpet. På Figur 2 nedenfor ses en typisk 3-delning. Ligesom en trelagsarkitektur er et fornuftigt valg for opbygningen af selve programmet, som ligger på Grupperums-Serveren (se Figur 3, side 10), sådan er det også fornuftigt at adskille andre dele i systemet fra hinanden. I et godt distribueret system, kan "presentation tier" udskiftes uden større ændringer i "business logic tier". Ligeledes skal databasen kunne skiftes ud, uden at kræve væsentlige ændringer på "business logic tier".



Figur 2 - 3 Tiers arkitektur.<sup>2</sup>

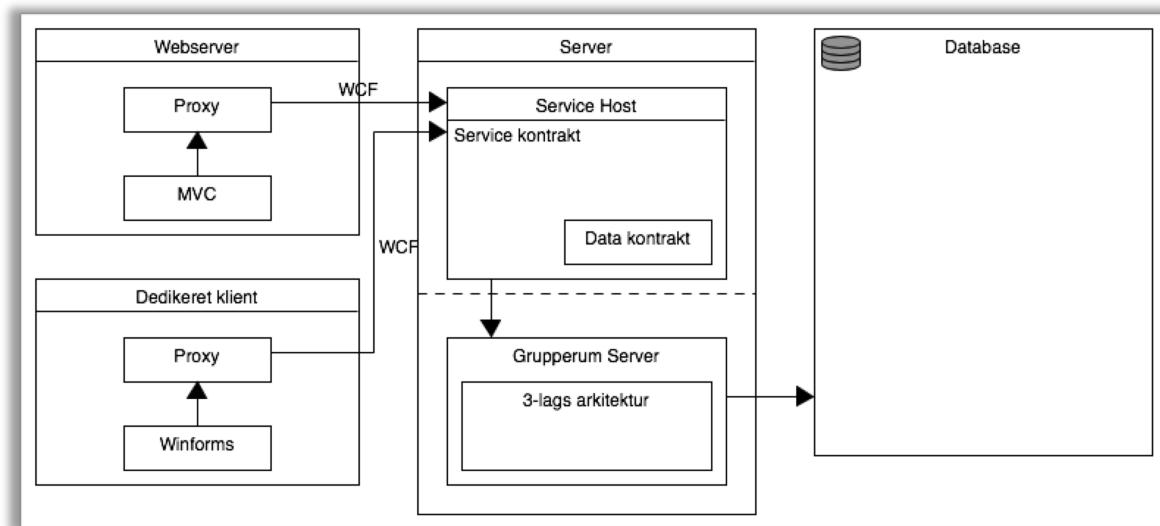
"Presentation tier", er i dette projekt ækvivalent med webserveren der håndterer klienternes forespørgsler på hjemmesiden, og den dedikerede klient, der ligeledes håndterer forespørgsler i programmet. "Business Logic tier" kaldes i dette system for "Grupperum server" og håndterer logikken og algoritmen. "Database Tier" skaber forbindelse til lagring af informationer om brugerne, klasselokaler og grupperum på databasen.

<sup>1</sup> Kilde: Distributed Systems: Principles and Paradigms, 2007 (Se kildehenvisningen)

<sup>2</sup> <http://www.techrepublic.com/article/architecting-a-cms-in-aspnet-one-server-or-a-bunch-of-servers/>

Projektet er blevet bygget op med tanke på denne 3-deling af systemet. Af hensyn til lettere fejlfinding i systemet kører "presentation" og "business logic tier" på den samme fysiske computer. Dog er systemet bygget op med tanke på, at webserveren og grupperum serveren ikke nødvendigvis kører på den samme computer. I projektet har UCN stillet en database til rådighed, som er blevet benyttet.

I det følgende afsnit vil der være en mere detaljeret visning af arkitekturen i projektet.



Figur 3 - Systemets overordnede arkitektur.

Som det kan ses på Figur 3 ovenfor, er systemet opdelt i 3 overordnede lag. I den venstre kolonne ses webserveren. Når en klient besøger hjemmesiden, vil denne sende en forespørgsel til webserveren. Webserveren er opbygget ved hjælp af MVC, hvilket står for Model View Controller. MVC kan anses som en form for 3-delt arkitektur der sender Html til klienten. MVC vil blive forklaret nærmere i afsnittet "Klient lag" på side 11.

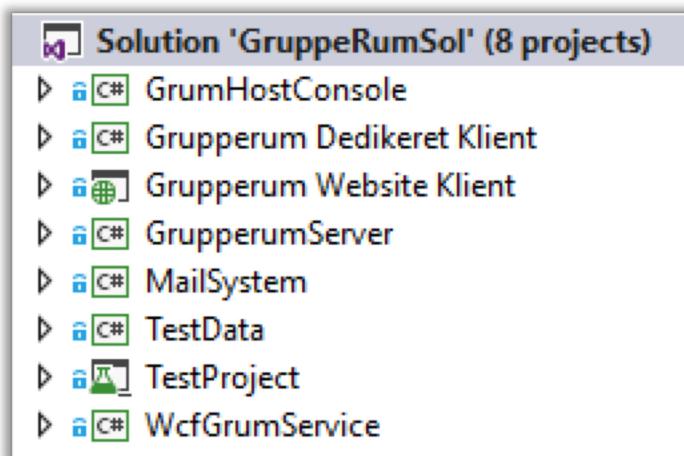
Den midterste del, kaldet "Server", på Figur 3, kan ses som en form for controller lag. Det er her logikken i programmet ligger. "Server-laget" er delt op i 2 dele, den egentlige Grupperum server (GrupperumServer og MailSystem på Figur 4) og Service host (GrumHostConsole og WcfGrumService). Service Host sørger for, at starte en WCF<sup>3</sup> service. WCF benyttes til, at webserveren kan kalde metoder i serveren, uden nødvendigvis at være på samme fysiske lokalitet. WCF vil blive gennemgået nærmere i afsnittet "Middleware" på side 13.

Grupperum serveren er ligeledes delt op i en 3-lagsarkitektur. Grupperum serveren består af et service lag, et controller lag og et modellag samt database-forbindelse. I grupperum serveren udføres udlejningen af et grupperum. Det er fra dette lag, at systemet kontrollerer om brugeren allerede har lejet et grupperum i den givne periode, og det er herfra systemet kontrollerer om det ønskede grupperum stadig er ledigt på det givne tidspunkt. Det er også her algoritmen som fordeler ventelistegrupperne på klasselokaler kører. Kort sagt er det her systemets logik befinner sig, også kaldet forretningslogikken.

<sup>3</sup> WCF: Windows communication foundation.

I højre side af Figur 3 er databasen, som udelukkende fungerer som lager. Databasen holder de studerende, klasserne, grupperne, lokalerne og mange andre informationer som skal lagres i længere tid. Når en gruppe har lejet et lokale vil udlejningen blive gemt i databasen under "Rent". På den måde kan grupperum serveren slå op i databasen for, at se om lokalet allerede er udlejet i denne periode, og ellers sende en fejlmeddeelse op til webserveren, der kan præsentere det for brugeren.

På Figur 4 nedenfor ses et billede af projektets solution. I dette projekt består "presentation tier" af "Grupperum Dedikeret Klient" og "Grupperum Website Klient".



Figur 4 - Vores forskellige projekter som de ser ud i Visual Studio.

### Klient lag

I et distribueret system kan man have mange forskellige klienter, der tilgår de samme services og servere. Man kan for eksempel have en service, som skal kunne bruges både af dedikerede klienter på et kontor, og fra en hjemmeside som skal være tilgængelig fra diverse enheder, som Android eller iOS telefoner og tablets mv.

Projektets distribuerede system er tilgængeligt over web og for dedikerede klienter. Den dedikerede klient del er lavet med Winforms, som er en del af Microsofts .Net framework til at lave desktop applikationer. Winforms har rigtig mange muligheder indbygget som forenkler processen med at danne forskellige gængse elementer som knapper og tekstbokse mv.

I dette projekt er den dedikerede klient blevet kraftigt nedprioriteret ad hensyn til opfyldelsen af de faglige mål for projektet. Den dedikerede klient er tiltænkt UCN personel og med denne kan administrationen oprette klasselokaler og fordele grupper, der har skrevet sig op, til et klasselocale ved hjælp af en knap. Herefter simuleres afsendingen af mails ud til de studerende i form af en tekstfil på C drevet. Al funktionalitet ud over dette er blevet tiltænkt til potentiel senere sprints.

Ligeledes kan man benytte WebForms (endant til WinForms) til udviklingen af webklienten. I Webforms har man, ligesom i Winforms, en designer-mulighed hvor man kan drag and drop prædesignede standardelementer ind, og definere funktionaliteten. Det kan selvfølgelig udbygges en hel del, men er en oplagt mulighed hvor designets look ikke har strengt definerede krav eller er høj prioritet.

En anden mulighed er at benytte MVC. Som sådan er MVC også bare et udtryk for en arkitektur af tre lag, ud fra samme logik som tidligere beskrevet. Lagene er "Model", "View" og "Control". I selve MVC .NET frameworket kan man i højere grad styre elementerne selv. I MVC har man ingen designer, men bygger alt selv. Model laget består af forskellige objekter ligesom på serverniveau. På den måde kan man samle data om objekter på et sted.

På Control laget har man alt forretningslogikken. Controlleren styrer også hvilket view der bliver returneret eller redirected til. På Figur 5 nedenfor er et eksempel på to controller metoder. Metoden med overskriften [HttpGet] bliver kaldt når siden CreateGroup er requested.

```
[HttpGet]
public ActionResult CreateGroup()
{
    CreateGroupModel model = new CreateGroupModel();
    using (GrumServiceClient client = new GrumServiceClient())
    {
        model.Students = client.hentDataTilModel();
    }
    return View(model);
}

[HttpPost]
public ActionResult CreateGroup(CreateGroupModel formModel)
{
    using (GrumServiceClient client = new GrumServiceClient())
    {
        client.SendDataNedTilService(formModel.Name);
    }
    return Redirect("Rent");
}
```

Figur 5 – Pseudo kodeudsnit af [HttpPost] og [HttpGet]

Når brugeren er færdig på siden og trykker videre til næste step i transaktionen, vil controlleren kalde [HttpPost] metoden til Viewet. Her vil metoden udføres ved hjælp af vores service reference og derefter sende brugeren videre til Viewet "Rent".

I View laget laves de sider som brugeren skal se, og her bruges information fra modellerne til at bygge siderne op og præsentere for brugeren.

Fordelen ved at bruge MVC frem for Webforms er, at man kan have flere modeller at arbejde i. Det vil sige der kan bruges forskelligt data på de enkelte views, alt efter hvilken data der opbevares i modellerne, og hvilke modeller man beder om at bruge i Viewet.

Til gengæld medfører de større muligheder i MVC ligeledes større udfordringer i forhold til f.eks. at bruge, håndtere og gemme modeller. WebForms er måske en anelse mere intuitivt at benytte, men en oplagt ulempe ved WebForms er, at det er i udfasning. Alene på det grundlag har MVC en stor fordel da det holder på længere sigt end Webforms.

## Middleware

En computer er på sin vis delt op i en intern trelags arkitektur. Det øverste lag er applikationslaget, hvor brugeren kan se og interagere med systemet. Det nederste lag er operativsystemet og hardwaren. Det er her alt logikken og intelligensen i applikationerne håndteres. Imellem de to lag er der middleware, som er det software der styrer forbindelsen imellem applikationen og operativsystemet samt hardware.

Middlewares opgave er at håndtere, at den samme applikation/software kan fungere på forskellige enheder på trods af diversitet i hardware og operativsystemer. Det kan for eksempel være, man skal bruge det samme program på forskelligt producerede computere.

I et distribueret system er der også et middleware lag som skal håndtere at forskellige enheder skal tilgå den samme server eller service over internettet, hvor internettet så vil være en del af den middleware, som sørger for at sende de rigtige beskeder videre i det rigtige format.

En af de problemstillinger middleware skal håndtere er "Little endian ctr big endian"-problemet. Dybest set bliver alt digitalt information bygget op af strøm og spænding hvor det er rækkefølgen på spændingerne som udgør det data der kan sendes og modtages. Derved opstår en problematik, når forskellige enheder skal sende information til hinanden, uden at have den samme lingvistik på deres bytes. En maskine med big endian teknologi vil have en anden rækkefølge på dens bytes i en digital besked, end en maskine med little endian, og det kan være svært at oversætte imellem de to versioner. Derfor skal middleware tage højde for hvilke teknologier der er i spil og oversætte, så man ikke får unødige misforståelser i kommunikationen. Kort fortalt hjælper middleware med at oversætte forskellige måder at kommunikere på, så slutresultatet bliver det, der var hensigten.

## Valg af middleware

Når man skal vælge hvilken middleware man vil benytte i et distribueret system er der en række ting man er nødt til at overveje i forhold til funktionaliteten i systemet.

Først skal man overveje heterogeniteten i systemet, altså om forskellige platforme skal kunne tilgå de samme services. Hvis der er tale om et internt system, i en virksomhed, er det ikke så vigtigt, da man så kan målrette systemet til de ønskede kunder, men skal man lave et system som kan bruges fra forskellige computere, webklienter og mobile enheder er det vigtigt, at middlewaren kan håndtere dette.

Derefter skal man overveje åbenheden i systemet (Openness). Mht. åbenhed skal man overveje om systemet skal være åbent så mange kan benytte og ændre i det, eller det skal være lukket så der ikke kan foretages ændringer fra for mange forskellige steder.

En vigtig ting at overveje er skalerbarheden i systemet. Det kan blive meget relevant om et system kan udvides til at håndtere mange flere samtidige brugere, end det først var designet til, eller om man ved voldsom ekspansion bliver nødt til at skaffe et helt nyt system til udfordringen.

Sikkerhed er også et vigtigt punkt at tage til overvejelse når man udvikler et system. Hvis systemet skal behandle personfølsom data eller kodeord og andre brugeroplysninger er det vigtigt at disse data bliver sikret ordentligt. Hvis man derimod kun skal håndtere offentligt tilgængelig information er det ikke så vigtigt at det er sikret.

En anden ting man skal overveje er fejlhåndtering. Hvor alvorligt er det hvis der opstår fejl? Hvis systemet skal håndtere funktionaliteten i en pacemaker eller bremserne på et tog kan det gå alvorligt galt, hvis fejl der opstår, ikke bliver håndteret korrekt. Skal systemet derimod stå for at tænde pyntelys på et bestemt klokkeslæt, kan indsatsen, for fuldstændig at undgå nedbrud, nedprioriteres.

Det næste man skal overveje er håndteringen af samtidighed i systemet. Hvis systemet skal have mange samtidige brugere er det vigtigt at vide hvordan man håndterer samtidighedsproblematikker. I et biograf booking system er det for eksempel relevant om to personer kan komme til at booke det samme sæde, eller om hele den valgte sal låses så kun en bruger kan se salen ad gangen.

Til sidst skal man gøre sig overvejelser i forhold til transparency. Transparency omhandler, om forskellige funktioner i systemet kører, uden at brugeren mærker det. Det kan for eksempel være om en fil eller noget data som brugeren arbejder med, bliver flyttet til en anden placering, eller det kan være om en resurse bliver delt imellem flere brugere.

### **WCF**

I projektet bruges WCF (Windows Communication Foundation), som er en service der håndterer at flere klienter på forskellige platforme tilgår den samme server. WCF sørger for at mange forskellige klienter kan kontakte serveren igennem forskellige protokoller. Det kan for eksempel være en webklient som kommunikerer over en HTTP forbindelse, eller en dedikeret klient som kommunikerer over en TCP eller UDP protokol.

En WCF service opbygges, ved hjælp af en række skridt. Først oprettes WCF servicen, som kommer til at håndtere kommunikationen imellem klienten og serveren. På servicen tilføjes så et interface der skal fungere som servicekontrakt. Servicekontrakten indeholder en række operationskontrakter, som er de metoder en klient kan tilgå på servicen. Selve servicen oprettes så, med de metoder der er defineret i servicekontrakten og operationskontrakterne.

Herunder ses et billede af et udsnit af vores service interface (Figur 6). Som det ses er denne defineret som servicekontrakt, med operationskontrakter.

```
[ServiceContract]
public interface IGrumService
{
    [OperationContract]
    Class getClassById(int id);

    [OperationContract]
    bool CreateGroupRoom(string name, bool whiteboard, bool monitor);

    [OperationContract]
    bool Authenticate(int user, string password);

    [OperationContract]
    bool CreateGroup(string name, List<int> studentId);

    [OperationContract]
    bool UpdateGroupRoom(string name, bool whiteboard, bool monitor);
}
```

Figur 6 - Udsnit af GrumService interface. Eksempel på Service kontrakt og Operationskontrakter.

De model klasser, på serveren, som skal være tilgængelige for servicen, og derigennem klienterne, skal laves serialiserbare, i form af datakontrakter. De properties og attributter der skal være tilgængelige på modellerne skal laves til data medlemmer (datamembers).

På Figur 7 nedenfor ses vores "Group"-model. Den har fåetDataContract og DataMembers, da den skal sendes igennem vores server, ud til vores webklient.

```
[DataContract]
public class Group
{
    [DataMember]
    private String name;
    [DataMember]
    public int Id { get; set; }
    [DataMember]
    private List<int> StudentId;
```

Figur 7 - Udsnit af klassen 'Group' hvor det ses at den har en Data Kontrakt og Data Medlemmer.

Når man skal bruge WCF servicen på klienten, giver man denne en service reference til servicen. Referencen sørger så for at oprette en proxy klasse, som svarer til servicekontrakten. Det er denne proxy klasse, som klienten benytter til at kalde serveren.

Når en WCF service sættes op, giver man den forskellige 'end points' og 'bindings', som fortæller servicen hvilke protokoller den kan forvente på forskellige porte. På den måde sørger WCF servicen for at håndtere de indkommende forespørgsler, og sende dem videre til serveren i et format den forstår, uanset hvilket format de kommer ind i.

Nogle af de forskellige protokoller man kan bruge til at sende information igennem WCF er TCP, UDP og HTTP (HTTPS). TCP/IP (Transmission Control Protocol / Internet Protocol) er internet protokol-stakken, som enten bruger TCP eller UDP protokollen.

TCP og UDP protokollerne ligger i OSI<sup>4</sup> modellens fjerde lag, Transportlaget.

TCP protokollen er "Forbindelses orienteret" (connection oriented) hvilket vil sige at der oprettes en forbindelse imellem klienten og serveren, over en fast defineret rute, som benyttes til at sende al dataen frem og tilbage. Derudover afventer TCP protokollen svar (Acknowledgement) fra modtageren, for at bekræfte at forbindelsen er oppe. Denne transaktion kaldes et TCP Handshake, og den sørger for at alt data sendes og modtages i den rigtige mængde og rækkefølge.

UDP protokollen er derimod " Forbindelsesløs" (Connectionless) hvilket vil sige at den sender data afsted uden at kende en fast route til modtageren. Den sender i stedet pakkerne ud på netværket hvor de så enkeltvis bliver routet videre til modtageren over forskellige ruter ved hjælp af IP-adresser.

HTTP<sup>5</sup> og HTTPS protokollerne er internet protokoller som bruges til at vise websider. HTTPS er en SSL krypteret version af HTTP. HTTP bruger en af TCP eller UDP protokollerne til at overføre data.

En fordel ved TCP protokollen, frem for UDP, er at den sender og modtager alt i den rigtige rækkefølge og derfor ikke skal bruge så meget tid på at validere data. En ulempe er til gen gæld, at hvis en af de routere der er på ruten imellem serveren og klienten går ned, så mistes forbindelsen, og der skal oprettes en ny, hvilket, i nogle tilfælde, kan skabe problemer.



Figur 8 - En visuel repræsentation af kommunikation over TCP protokollen. Ruten er fast defineret, og må ikke brydes.<sup>6</sup>

En fordel ved UDP er at man er ligeglads med om en enkelt router går ned imellem serveren og klienten, da de forskellige pakker alligevel routes i forskellige retninger. Skulle dette ske sendes pakkerne bare en anden vej til deres destination. En ulempe er at protokollen skal bruge en del tid på at validere at alle pakker er nået frem og er intakte.

<sup>4</sup> Se Bilag 1

<sup>5</sup> http: Hyper text transfer protocol

<sup>6</sup> Kilde: <https://www.quora.com/Why-do-the-different-body-processes-such-as-digestion-respiration-urination-etc-produce-sound>



Figur 9 - En visuel repræsentation af kommunikation over UDP protokollen. Ruten er ukendt og påvirkes af eksterne faktorer.<sup>7</sup>

På Figur 10 nedenfor kan man se service modellen fra vores webklient. Der ses først vores Binding som er en Basic HTTP binding til vores service, som heder IGrumService. Derefter ses vores endpoint. Adressen på endpointet er localhost, efterfulgt af en port og vores service reference. Vores system kører kun på localhost, men i det tilfælde at det skal distribueres skal vi naturligvis ændre adressen på endpointet fra localhost til IP adressen på den server hvor servicen findes.

```
<system.serviceModel>
  <bindings>
    <basicHttpBinding>
      <binding name="BasicHttpBinding_IGrumService" />
    </basicHttpBinding>
  </bindings>
  <client>
    <endpoint address="http://localhost:8733/WcfGrumService/" binding="basicHttpBinding"
      bindingConfiguration="BasicHttpBinding_IGrumService" contract="GrumService.IGrumService"
      name="BasicHttpBinding_IGrumService" />
  </client>
</system.serviceModel>
```

Figur 10 - Billede af binding og endpoint på webklienten.

### Algoritme til fordeling

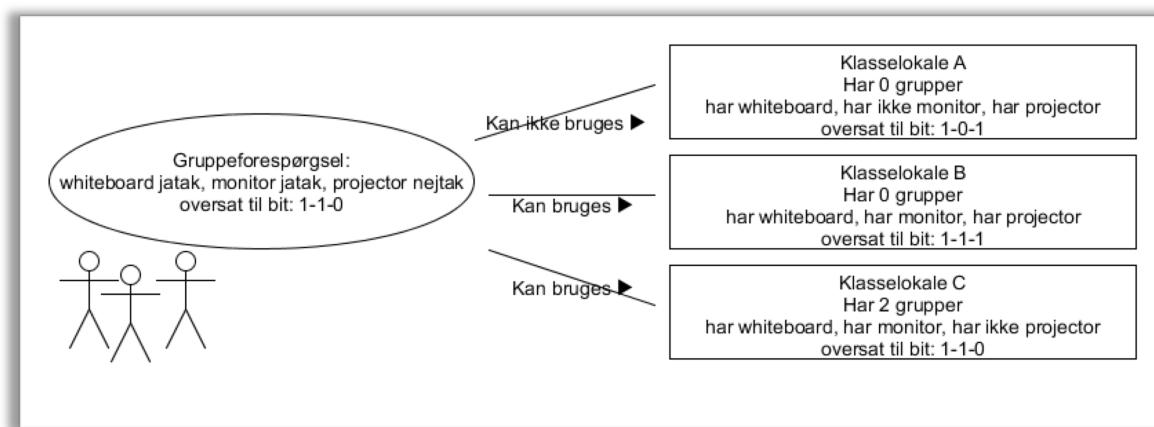
Når en gruppe ikke kan få et grupperum, kan man skrive sig på en venteliste til at få gruppeplads i et klasselocale. Algoritmen står for at tage ventelisten med gruppernes ønsker og fordele dem optimalt på lokaler.

Fordelingen foregår ud fra gruppernes krav til inventar (whiteboard, monitor, projector) og ud fra et fastsat krav til, at der max må være 3 grupper pr lokale aht. plads og koncentration. Der er forskellige udfordringer i denne fordeling, bl.a. at hvis 3 grupper ønsker en projektor skal de ikke have tildelt det samme lokale, selvom det er et lokale med projektor, for kun en af grupperne kan benytte den. Dermed må lokalelets tilgængelige inventar ændres undervejs at

<sup>7</sup> Kilde: <http://www.vox.com/2015/4/28/8505377/paper-airplanes-history>

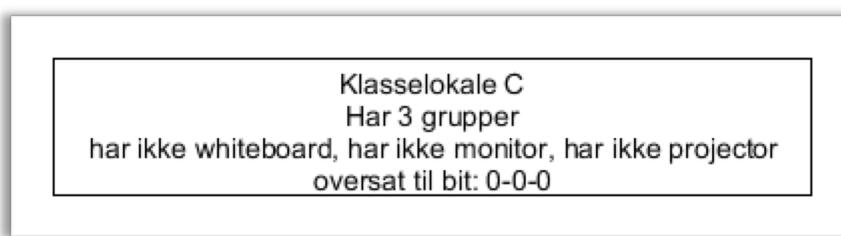
grupperne bliver fordelt. Både gruppernes krav kan være forskellige og lokalernes inventar kan være forskelligt.

I databasen bliver lokalet (Classroom) oprettet med bit-værdier for de 3 forskellige inventaregenskaber. Et lokale med både whiteboard, monitor og projector vil på de 3 felter i databasen have 1-tal. Ligeledes vil gruppens forespørgsel også registreres på databasen med 1-tal for det inventar de ønsker og 0-tal for det inventar de ikke har brug for. Der er naturligvis behov for at kunne sammenligne de to bit-rækkefølger og tage højde for mere end at de er identiske. Der er jo den mulighed at en gruppe som ønsker en projector og ikke andet (0-0-1) bør kunne få plads i lokaler uanset om de også har monitor og whiteboard og vil derfor skulle kunne få plads i både lokaler med bitrækkefølgen (0-0-1), (1-0-1), (0-1-1) og (1-1-1).



Figur 11 - En gruppens forespørgsel forsøges matchet.

På Figur 11 illustreres et eksempel på en forespørgsel overfor nogle forskellige lokaler; Gruppens forespørgsel vil kunne blive matchet af lokale B og C som begge har det forespurgt inventar til rådighed og desuden stadig plads til flere grupper (op til 3 grupper pr lokale). Hvis gruppen bliver koblet på klasselokale C, så vil klasselokale C ikke længere indeholde ledigt inventar, men der vil desuden heller ikke være plads til flere grupper i dette lokale, da gruppemax på 3 er nået, se Figur 12.



Figur 12 - Klassenlokale C modereret i forhold til i Figur 11.

En enkel måde at løse denne bit-rækkefølge-sammenligning på, er at omregne bitrækkefølgen binært til int. Ved at give whiteboard det første tal i bit-rækkefølgen, et 4-tal ( $1 \cdot 2^0 = 1$ ), monitor et 2-tal ( $0 \cdot 2^1 = 0$ ) og projector et 1-tal ( $0 \cdot 2^2 = 0$ ) vil alle muligheder for inventar kunne holdes i intværdierne 0-7. Således vil et lokale med 3 positive bit få koden 7 ( $4+2+1$ ). Et lokale uden whiteboard, men med monitor og projector får koden 3 ( $0+2+1$ ). Se yderligere muligheder i Figur 13.

Whiteboard	Monitor	Projector	omregnet til int
1	1	1	7
1	1	0	6
1	0	1	5
1	0	0	4
0	1	1	3
0	1	0	2
0	0	1	1
0	0	0	0

Figur 13 - Oversigt over de forskellige muligheder for dannelsen af int-koderne.

Beregningen fra bit til int i C# ser ud som vist på Figur 14.

```
public int CreateBinaryCode(bool whiteboard, bool monitor, bool projector)
{
    int requestCode = 0;
    using (var con = new DBCon())
    {

        if (whiteboard)
        {
            requestCode += 4;
        }

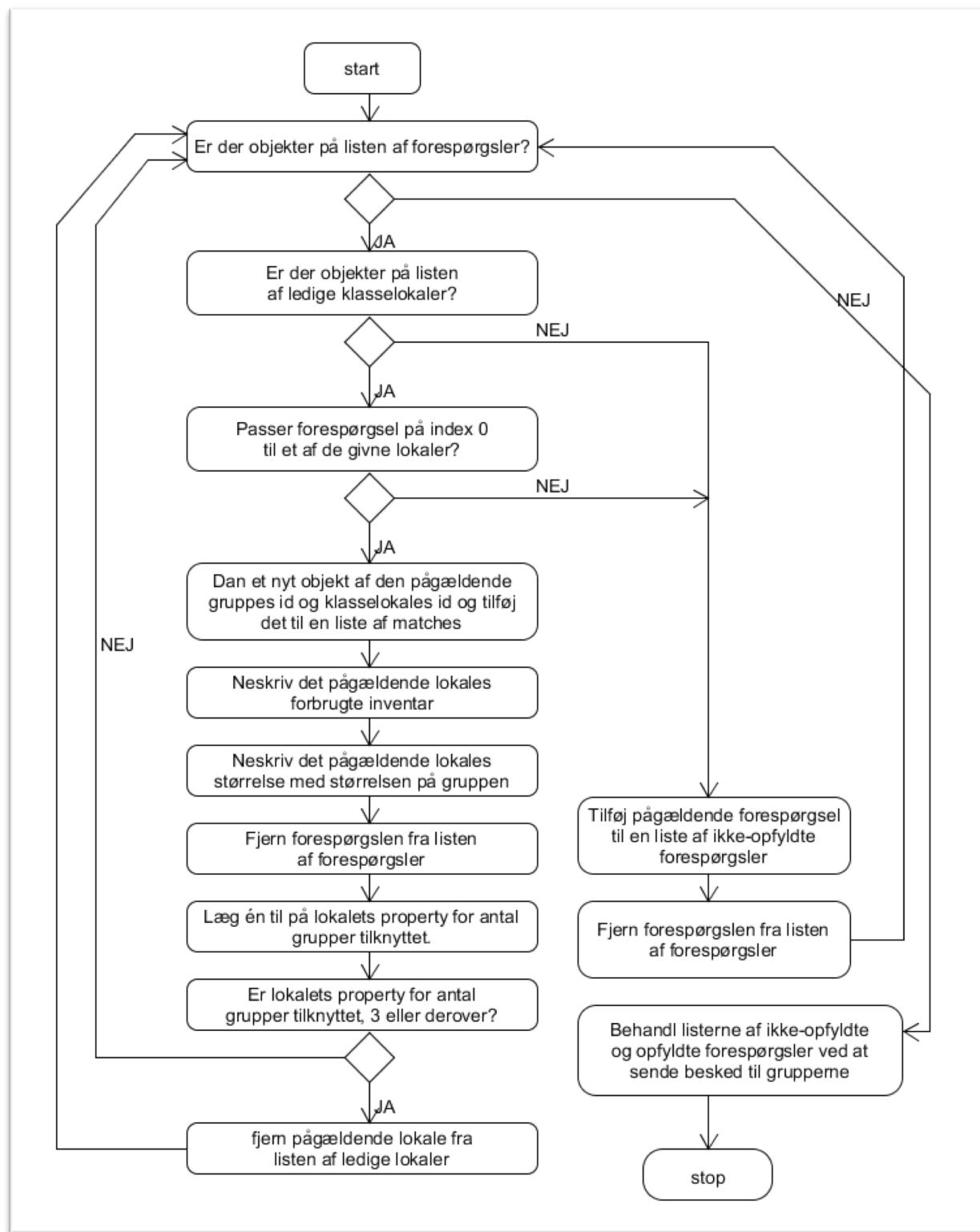
        if (monitor)
        {
            requestCode += 2;
        }

        if (projector)
        {
            requestCode += 1;
        }
    }
    return requestCode;
}
```

Figur 14 - Kodesnip hvor bitinfo fra databasen omdannes til en kode i form af en int.

Dette system rummer samtidig den fordel, at et lokales inventar kan nedskrives, med det der er forbrugt og få en ny kode som passer. Et lokale med koden 3 (0 - 1 - 1) der får forbrugt sin monitor med værdien 2 (0 - 1 - 0) vil have 1 tilbage (0 - 0 - 1), hvilket også stemmer, da projector stadig er ledig.

Næste skridt er at iterere igennem listerne med forespørgsler (RequestClassroom) og lokaler (Classroom). Undervejs i iterationen er det nødvendigt at notere hvilket lokale der bliver koblet på hvilken gruppe, nedskrive klassens inventarkode, og tælle op på lokalet for hver gruppe der tages ind (må ikke overskride 3). Algoritmen kører som illustreret i følgende flowchart se Figur 15.



Figur 15 - Forløbet i algoritmens fordeling af forespørgsler på klasselokaler.

Algoritmen er den opskrift, der sikrer, at vi kommer igennem alle mulige udfald og får behandlet samtlige forespørgsler. Rækkefølgen som forespørgslerne bliver gennemgået i, er dog ikke uden betydning for det samlede udfald. Da det er optimalt med flest mulige matches, bør forespørgsler med færrest matchmuligheder (1-1-1), altså forespørgsler med koden 7, behandles først. Med tanke på at listen itereres igennem fra en ende af, sorteredes listen med

forespørgsler, ud fra kodeproprietien (requestCode), så forespørgsler med de højeste koder ligger på laveste index og så fremdeles. Til det benyttes en metode på List<> som hedder "OrderByDescending". Den benytter en lambda til at sortere listen ud fra en property på objekterne. Den sorterer netop efter højeste først.

```
public List<RequestClassroom> sortRequestList(List<RequestClassroom> stillNotFulfilled)
{
    return stillNotFulfilled.OrderByDescending(x => x(requestCode).ToList();
}
```

Figur 16 - Kodesnip hvor OrderByDescending-metoden er i brug og det fremgår at det er proprietien "requestCode" den sorterer ud fra.

Når listen af klasselokaler gennemgås, er det til gengæld fordelagtigt at finde det laveste match først og bruge det objekt, så der ikke, som det første, bliver "splittet lokaler ad" som ville kunne matche andre krav. En 5'er forespørgsel (4+1) vil f.eks. kunne bruge et 7'er lokale (4+2+1) og i den forbindelse nedskrive det lokale til en 2'er. Senere vil vi skulle finde et lokale til en 3'er forespørgsel (2+1), som også ville kunne have brugt 7'er lokaledet før det blev nedskrevet.

Altså er ideen at opfylde forespørgslen med så lidt som muligt og derfor helst med en klassekode identisk med forespørgselskoden, og først derefter søge i højere matches. Ligesom listen af forespørgsler blev sorteret, kan vi også sortere listen af lokaler, blot med en metode på List<> som hedder OrderBy som også giver os en liste sorteret på en bestemt af objekternes properties, men efter laveste først.

Næste udfordring er at matchene skal laves korrekt af algoritmen og dermed skal det formuleres på en matematisk formel i C#. Når man ser på forespørgselskoderne ctr lokalekoderne i en matrix ser de ud som på Figur 17 nedenfor.

Forespørgsel									
7	7								
6	7	6							
5	7		5						
4	7	6		4					
3	7				3				
2	7	6			3	2			
1	7		5		3		1		
0	7	6	5	4	3	2	1	0	

Figur 17 - Forespørgsler i venstre kolonne og derudfor de lokalekoder som forespørgslen vil være tilfreds med.

Det kan blive lidt indviklet at formulere hvornår forespørgsel(x) skal være tilfreds med lokale(y). Det vil blive noget i retning af:

Når x er større end 0 OG x == y ELLER y==7 ELLER y/x = giver enten 5 eller 3 eller 1.5 så vil det være et matchbart udfald ELLER når

x==0 vil det altid være matchbart udfald da en 0-forespørgsel er tilfreds med alt.

Selvom lignende udregning vil kunne give et korrekt udfald, er den dog kompliceret og tager ikke umiddelbart højde for skalerbarhed, da det i så fald vil skulle tjekkes, hvilke øvrige resultater der vil kunne forekomme af  $y$  divideret med  $x$ . Indtil videre giver  $3/1$  og  $3/2$  og  $5/1$  og  $6/2$  og  $6/4$  alle enten 5 eller 3 eller 1,5, men vil dette kunne garanteres ved en opskalering? Netop de resultater må jo ikke kunne forekomme hvor forespørgslen IKKE matcher.

En langt enklere og skalersikker sammenligning kan foretages med en binær operator AND (skrevet &) som netop sammenligner på bitværdierne. Formuleringen vil se sådan ud:

Når  $y \& x == x$

AND fungerer, som om den sammenligner bitværdierne og danner en tredje bitværdi. Se eksempler i Figur 19 og Figur 20. Kun hvis der forekommer 2 1-taller i sammenligningen vil AND give 1, ellers 0 (se Figur 18). Dermed kan  $y$  have flere 1'ere end  $x$  og stadig ende med at ligne  $x$ , men SKAL have 1'ere hvor  $x$  har, for at give et brugbart udfald.

& udfalds matrix	0	1
0	0	0
1	0	1

Figur 18 - Udfaldsmatrix for en standard AND gate.

x forespørgsel	1	0	1
y lokale	1	1	1
	1	0	1

Figur 19 - Eksempel hvor AND sammenligner x(101) og y(111) og resultatet bliver lig med x, altså et match.

x forespørgsel	1	0	1
y lokale	1	1	0
	1	0	0

Figur 20 - Eksempel hvor AND sammenligner x(101) og y(110) og resultatet IKKE bliver lig med x, altså ikke et match.

### Algoritmen i C#

I C# koden bruger vi AND sammenligningen som kan ses på Figur 21 nedenfor.

"stillNotFulfilled" er listen af forespørgsler på ventelisten. Forespørgslerne er som sagt sorteret efter propertien med koden, kaldet "requestClassroom", højeste først.

"lessThanThree" er listen af lokaler (som endnu ikke har tilknyttet 3 grupper). Lokalerne er sorteret efter koden "requestMatch", laveste først.

```
while (lessThanThree.Count > 0 && stillNotFulfilled.Count > 0)
{
    if ((lessThanThree[i].RequestMatch & stillNotFulfilled[0].RequestCode) == stillNotFulfilled[i].RequestCode)
    {
        RequestMatch requestMatch = new RequestMatch(stillNotFulfilled[0].GroupId, lessThanThree[i].Id);
        matchedRequests.Add(requestMatch);
        lessThanThree[i].RequestMatch -= stillNotFulfilled[0].RequestCode;
        lessThanThree[i].Size -= stillNotFulfilled[0].GroupSize;
        stillNotFulfilled.Remove(stillNotFulfilled[0]);
        lessThanThree[i].GroupCount++;
        if (lessThanThree[i].GroupCount >= 3)
        {
            lessThanThree.Remove(lessThanThree[i]);
        }
        i = 0;
    }
}
```

Figur 21 - Udsnit af algoritmens funktioner.

Det defineres, at så længe der stadig er objekter på enten forespørgselslisten eller lokalelisten, så må whileloopet fortsætte med at sammenligne over if-statementet.

If-statementet bruger AND sammenligningen, og bliver der fundet et match, så sker der følgende:

- Et nyt objekt med gruppeld og lokaleId bliver oprettet (requestMatch) og lægges på en liste med disse (matchedRequests).
- RequestMatch minus RequestCode giver ny RequestMatch for det pågældende lokale (som tidligere beskrevet)
- Lokalets størrelse minus gruppens størrelse giver lokalets resterende størrelse.
- Forespørgslen på det pågældende index, som nu er opfyldt, tages af listen stillNotFulfilled.
- Proprieten om grupper tilknyttet lokalet (GroupCount), tælles op med én.
- Slutteligt (i denne del af scopet) undersøges om lokalet også skal tages af listen med ledige lokaler, hvis GroupCount er over eller lig med 3.

Whileloopet har brug for at blive stoppet igen for ikke at køre uendeligt. Det kan håndteres på flere måder, men en gennemskuelig måde det håndteres på her, er ved at undersøge på index 0 hver gang, behandle det objekt og fjerne det fra listen, uanset om der er fundet et match eller ej. Da vi hele tiden fjerner den forespørgsel vi behandler fra forespørgslerne, vil den næste forespørgsel vi skal kigge på også ligge på index 0. Når listen af forespørgsler er tom vil while-loopet standse. Hvis der er så mange forespørgsler at der ikke er lokaler nok, vil listen af lokaler løbe tør og standse loopen.

```
else
{
    i++;
    if(lessThanThree.Count <= i)
    {
        notFulfilled.Add(stillNotFulfilled[0].GroupId);
        stillNotFulfilled.Remove(stillNotFulfilled[0]);
        i = 0;
    }
}
```

Figur 22 - "else" del i algoritme.

Som det ses af Figur 22 ovenfor vil forespørgslen altid blive placeret på en af 2 lister. Hvis der er match vil id'et blive overført til listen matchedRequests som tidligere beskrevet. Hvis der ikke findes et match på hele listen af lokaler, bliver "else" scopet aktivt og fjerner forespørgslen fra listen, men tilføjer den til en liste af "notFulfilled".

I forhold til skalerbarheden er det allerede belyst lidt tidligere. Det kan f.eks være en udbygning af inventarlisten. Det er ikke et problem, når der arbejdes binært med koden for inventaret og ligeledes binært i sammenligningen af forespørgsel og inventar.

En anden opskalering man kan se for sig, er meget store klasselokaler, hvor der vil kunne være mere end 3 grupper ad gangen, uden det vil gå ud over evnen til at koncentrere sig. Tidligt i processen arbejdede vi med 2 krav, udover at inventarkoderne skulle passe, nemlig at der max måtte være 3 grupper i et lokale eller max 20 personer. Af den grund er der en property på klasselokalerne som hedder "Size" og som var meningen der skulle tælles på, for hver tilføjet gruppens størrelse. Men når grupperne max må være 7 personer store, vil der ved 3 grupper i lokalalet alligevel maksimum kunne blive 21 personer. Der vil kunne udvikles på denne del, den dag det skulle blive relevant i forhold til ekstremt store lokaler.

## Test

Når man udvikler et distribueret system er det vigtigt at lave tests undervejs på forskellige dele af koden, for at sikre at det virker, og for at fange eventuelle fejl så tidligt som muligt.

En god metode til at teste på, er 'Test First' hvor man skriver en række unit tests før man koder systemet, for at vide præcis hvilke krav skal opfyldes. På den måde kan man holde sig til at løse de problemstillinger der er relevante lige nu, frem for at lave en masse kode som ikke er relevant før senere i projektet.

I dette projekt er der løbende blevet lavet tests til en del af funktionaliteten i systemet, for at teste de enkelte dele der er blevet lavet. Der er også oprettet testdata, som testene kan bruge i systemet. Testdataene er lavet, for det første fordi man så ikke nødvendigvis skal kontakte databasen hver gang man køre en test, og for det andet fordi man ikke altid kan være sikker på at de data der ligger i databasen er ens. Der kan testdata være godt, da man så ved præcis hvilket input der er til systemet, og derved let kan overskue det forventede udfald.

## Samtidighed

Når man ser på transaktioner i et distribueret system, kan man overveje behovet for ACID egenskaberne (Atomicity, Consistency, Isolation, Durability).

### ACID

**Atomicity** skal sikre at transaktionen er enten-eller. Hvis en transaktion tager flere steps, er det essentielt at alle dele af transaktionen lykkes, ellers vil alt blive ført tilbage til det oprindelige stade (rollback).

**Consistency** beskriver at transaktionen skal gennemføres inden for det regelset der er blevet sat, ellers er den mislykkedes.

**Isolation** sikrer at transaktionen låses, således andre brugere ikke kan tilgå dataen før transaktionen er afviklet.

**Durability** sikrer at hvis en transaktion er blevet gennemført vil den forblive gennemført og der kan ikke gennemføres rollback.

### Håndtering af samtidighed

Når der arbejdes med et distribueret system, dukker der ofte nogle udfordringer op i form af samtidighed, som man må forholde sig til. Samtidighed er en stor del af projektet, da det er muligt for flere brugere at se en liste af de ledige lokaler samtidigt. Således kan to forskellige brugere kigge på eksempelvis lokale "SD4.0.1" samtidig, hvor det fremgår som ledigt hos begge brugere. Den slags muligheder giver problemer, for hvad sker der hvis begge brugere lejer lokalet samtidigt? For at undgå at der sker noget utilsigtet må vi håndtere samtidighed.

Hvad angår samtidighed, kan det løses på forskellige måder. En metode hvorpå samtidighedsproblemet kan løses er, at "låse" på forskellige niveauer. Den liste af lokaler, som en bruger får præsenteret, kan f.eks låses i et tidsrum, så det kun er denne bruger der kan se listen som ledig. En anden bruger, ville således få vist disse lokaler som optaget, mens den første bruger beslutter sig til, hvilket lokale der ønskes. Dette svarer til "isolation"-egenskaben i ACID. Dette kaldes en pessimistisk løsning.

Problemet ved at bruge den ovenstående metode, hvor listen af lokaler bliver låst er, at der kun er én bruger der kan se listen ad gangen, hvilket ikke er hensigtsmæssigt, da der sandsynligvis vil være flere grupper der vil tilgå systemet samtidig, så en mere brugervenlig effektiv løsning er at foretrække. Dette problem kan løses ved at, der låses på et "lavere niveau".

Et system hvor niveauerne bedre kan illustreres er biografbooking; et højt niveau vil være at låse en hel sal imens én bruger finder en plads at booke. Tilsvarende kan man låse en række af sæder, eller man kan låse på et lavt niveau og kun låse sæderne individuelt i den tid det tager for brugeren at gennemføre bookingen. Jo længere nede i systemet der låses, jo større frihed til øvrige brugere. I vores program vil et lavere niveau svare til at låse et enkelt lokale, i stedet for at låse hele listen.

Ved denne metode, som kaldes en two-phaselocking, vil det kun være lokalet der er markeret, som vil blive låst, imens brugeren beslutter sig. Den tid der skal låses afhænger af hvilke aktioner en bruger skal igennem efter at have besluttet sig. Ved bestilling af biografbilletter, skal brugeren have tid til at lave en transaktion og indtaste oplysninger for at få bestilt billetten, og endeligt have krav på sædet. I vores projekt, tager vi udgangspunkt i, at brugeren allerede

er logget ind, og har oprettet en gruppe, før brugeren kan leje et grupperum. Det er derfor ikke nødvendigt, at indtaste data for brugeren efter brugeren har trykket "Book lokale" og dermed kan låsningen gøres relativt kort.

Det skal ligeledes håndteres hvad der rent faktisk sker, hvis en bruger går fra sin bookning af billetter eller er alt for langsom – Så er der behov for at lokalet efter gives fri for andre brugere. Den bruger som er for langsom til at foretage et valg, må så få en besked når der forsøges at booke, om at det lokale desværre ikke er ledigt længere.

Da brugeren, i dette projekt, blot skal trykke på "Book lokale" og straks får en bekræftelse, er der valgt at bruge en optimistisk løsning. Det vil sige at lokalet ikke låses før det er booket. Med denne metode, er det muligt for en bruger at forsøge at booke et lokale som allerede er booket. Dette løses på serveren, ved at kontrollere, om lokalet er blevet booket i mellemtiden, før brugeren får en bekræftelse. Der er altså implementeret et først til mølle princip, hvor samtidigheden håndteres, ved at give brugeren besked, hvis han eller hun ikke nåede at booke det ønskede lokale.

### Atomicitet

I et projekt som dette, er atomicitet utrolig vigtigt. Hvis en bruger ønsker et lokale mandag, tirsdag og onsdag, skal brugeren ikke foreslå et lokale der kun er tilgængeligt mandag og onsdag. Atomiciteten går altså ud på, i dette projekt, at sørge for at brugeren får grupperummet i alle de ønskede dage. Atomiciteten ligger også tæt op ad samtidigheden, forstået på den måde, at en bruger kan få præsenteret et lokale for mandag til onsdag, men imens brugeren er ved at booke lokalet, kan tirsdagen være blevet taget af en anden bruger.

På Figur 23 ses et udsnit af vores kode der håndterer udlejningen af grupperum.

```
public bool RentGroupRoom(int grouproomId, int groupId, DateTime dateStart,  
                           DateTime dateEnd)  
{  
    bool isRented = false;  
    if (dbCtrl1.TestGroupRoom(grouproomId, dateStart, dateEnd))  
    {  
        if (dbCtrl1.CanTheyRent(groupId, dateStart, dateEnd))  
        {  
            isRented = dbCtrl1.RentGroupRoom(grouproomId, groupId,  
                                              dateStart, dateEnd);  
        }  
    }  
    return isRented;  
}
```

Figur 23 - Udsnit af RentGroupRoom metoden der håndterer udlejning af grupperum.

Metoden "RentGroupRoom" står for at udleje et grupperum til en given gruppe, ved at kalde forskellige kald i vores database controller, der indeholder sql sætninger.

Som det kan ses på Figur 23 sættes værdien af "isRented" til false som det første i metoden. Denne værdi bliver returneret når metoden er slut. Værdien bliver brugt til, at fortælle slutbrugeren, om grupperummet blev lejet eller ej.

For at færdiggøre metoden og dermed leje grupperummet bliver der kaldt tre metoder i vores databasecontroller, nemlig "TestGroupRoom", "CanTheyRent" og "RentGroupRoom". Disse tre metoder returnerer hver en værdi af true/false.

Den første metode der bliver kaldt, er TestGroupRoom. Metoden kigger i databasen, om det givne grupperum er blevet lejet i mellemtíden, og om grupperummet er ledigt i alle dagene, som brugeren har angivet. Hvis det stadig er ledigt i alle dagene går metoden videre til næste kald.

Den næste metode der bliver kaldt er CanTheyRent. Denne metode kontrollerer om gruppen allerede har lejet et grupperum i perioden. Dette gøres, for at undgå, at en gruppe kan leje flere grupperum i samme periode. Også denne metode returnerer en true/false værdi, som definerer om metoden kører videre til næste og sidste skridt.

Den sidste og afgørende metode, der bliver kaldt i RentGroupRoom sørger for at oprette et "rent" i databasen, og returnere en true/false værdi. Værdien sendes så videre til slutbrugeren, som får en bekræftelse eller en fejmeddelelse.

### Sikkerhed

Når man snakker sikkerhed i et computer system, er der mange forskellige ting at tage højde for. Et distribueret system kan blive angrebet med trojans, virusser, malware osv. Det kan også være "exploits", eller "code injektion".

For at sikre at disse ting ikke sker bruger vi forskellige sikkerhedsredskaber, som fx firewalls, authentication, kryptering eller en VPN forbindelse. Disse sikkerhedsbegreber vil blive beskrevet nedenunder.

### Firewall

En firewall er et netværks adgangs kontrol værktøj, som bruges til at sørge for at et intranet kun får det trafik der er blevet givet tilladelse til. Der er generelt to typer af firewalls: En applikationslag firewall og en packet-filtering firewall.

Applikationslag firewall'en kan eksempelvis være den du kender fra windows. Den virker som en proxy firewall til alle dine forskellige applikationer som bruger netværk. Et set af regler definerer hvordan trafikken fra en applikation til netværket håndteres. Hvis disse regler ikke bliver fulgt, vil firewallen afbryde forbindelsen.

For hver applikation man har, kan den konfigureres til forskellige indstillinger efter behov. Hvis du vil oprette en forbindelse til en FTP server, skal den følge FTP-protokollen, hvis den ikke gør det, stopper firewallen forbindelsen.

En packet-filterings firewall kigger først og fremmest på pakkens Header. En header indeholder information om pakken, som er ved at blive modtaget. Denne header har informationer der fortæller i hvilken sammenhæng denne skal bruges på computeren.

### Router

Nogle vil påstå at en router kan fungere som en firewall, men en routers opgave er at sende store mængder data hurtigt videre til den modtager som skal bruge dem, og derved skal den ikke bruge store mængder tid på at sortere data. Dog kan routeren bruges effektivt til at lukke veje til computeren, som en angriber muligvis vil forsøge at ramme computeren igennem. Da

routeren ved at computeren ikke vil modtage data fra fx port 80, vil den stoppe den angribende data, som angriberen sender imod port 80.

### Authentication

Authentication er en måde at validere noget information på. Ofte er dette set ved login systemer. En bruger kan validere sig selv ved at logge ind med brugernavn og password. Dette kaldes one-way authentication og det ville være denne løsning der skulle implementeres i dette projekt.

Two-way authentication bruger en sekundær part til loginet, et eksempel på dette er nemid. For at bruge systemet skal en bruger både have en kendskabsfactor (password og brugernavn) og en ejerskabsfaktor (nemid-kortet).

Authentication bliver ofte brugt til at begrænse information til en bestemt gruppe brugere, ofte følsom data. Den kan også blive brugt til at registrere hvilken bruger der har lavet hvilken transaktion.

### Kryptering

I et system med login, vil man altid skulle kryptere data. Dette gøres for at skjule brugerens password. En standard måde at gøre dette over nettet på, er med protokollen HTTPS, som er magen til HTTP, men bliver sikret med en SSL kryptering, mere om dette i næste afsnit. Dette gør at "Man-in-the-middle"(MITM) angreb bliver harmløse. MITM sker ved at en angriber sætter sig imellem brugeren og hjemmesiden, og kigger med på det data der bliver sendt. Hvis dette er krypteret med SSL, vil angriberen ikke kunne bruge det til noget.

### SSL (Secure socket layer)

Krypteringen foregår på den måde, at en bruger vil forsøge at oprette en sikker forbindelse med klienten. Når klienten gør dette, finder serveren den nyeste mulige SSL kryptering. Derefter kontrollerer klienten at den er verificeret hos en tredje part. Dette kunne være GeoTrust, her bliver det valideret at serveren som du vil i forbindelse med, rent faktisk er den, den udgiver sig for at være.

SSL bruger asymmetrisk kryptering. Dette gøres med to nøgler, en public key som bruges til at kryptere med, og en private key som bruges til dekryptering. Når jeg vil have følsom data overført over nettet, vil jeg som klient, sende min public key til serveren. Den ved nu hvordan den skal kryptere dataen, således at kun min private key kan åbne det igen.

### VPN

Virtual Private Network (VPN) laver en krypteret forbindelse over et ikke sikkert netværk, som fx internettet. Med en VPN-forbindelse kan en medarbejder koble sig på virksomhedens intranet, og arbejde som om klienten er tilsluttet virksomhedens LAN direkte.

### Konklusion

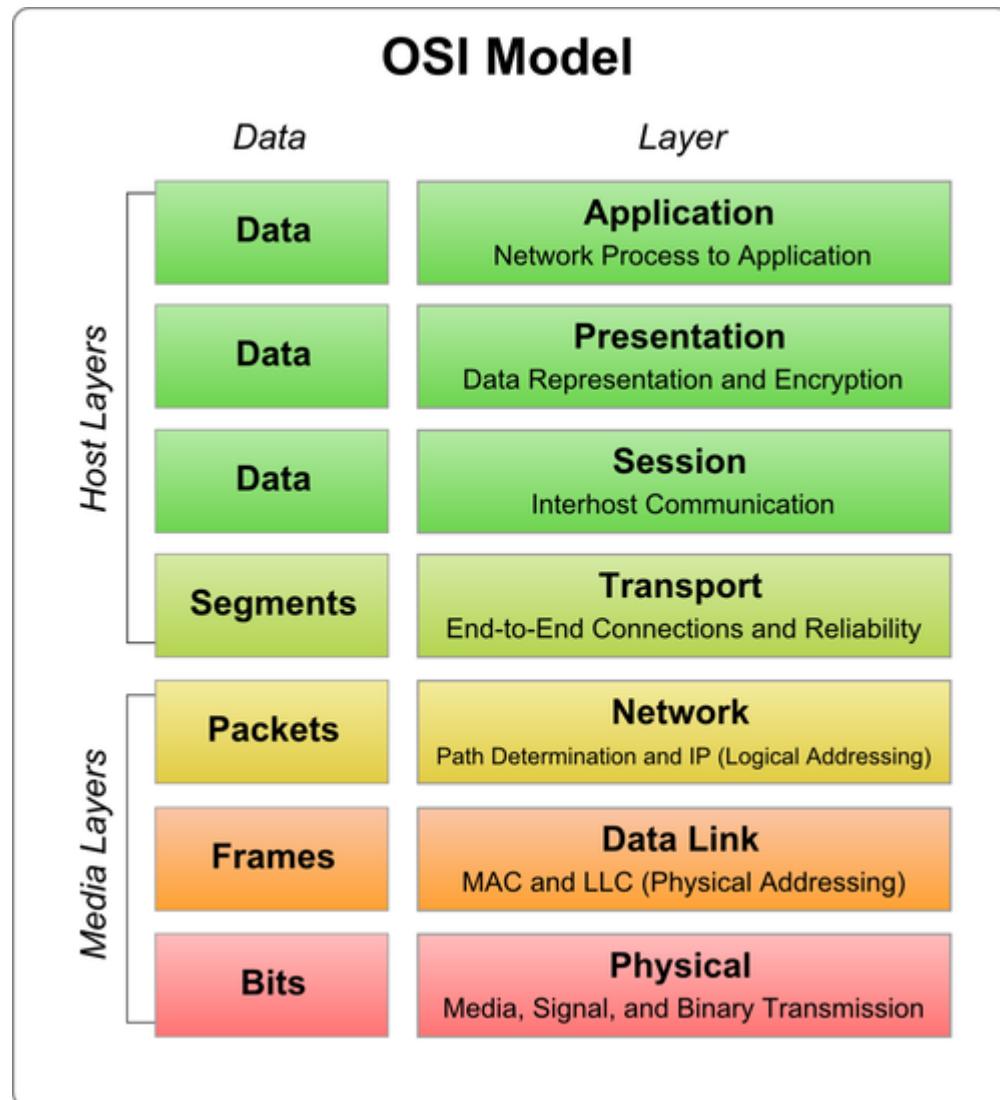
Ud fra opgaven stillet, at vi skulle kunne producere et distribueret system med indbygget håndtering af samtidighed og atomicitet, kan vi konkludere at dette er lykkedes. Det er ikke mange af delene der er implementeret, men de dele der kræves, for at demonstrere samtidighed og atomicitet i et distribueret system, fungerer med en brugerflade og er testet.

### Kildehenvisning

(2007). I Tanenbaum, & Van Steen, *Distributed Systems: Principles and Paradigms* (2 udg., s. 2). Prentice-Hall.

### Bilag

#### Bilag 1



<sup>8</sup> Kilde: <http://www.tech-faq.com/osi-model.html>