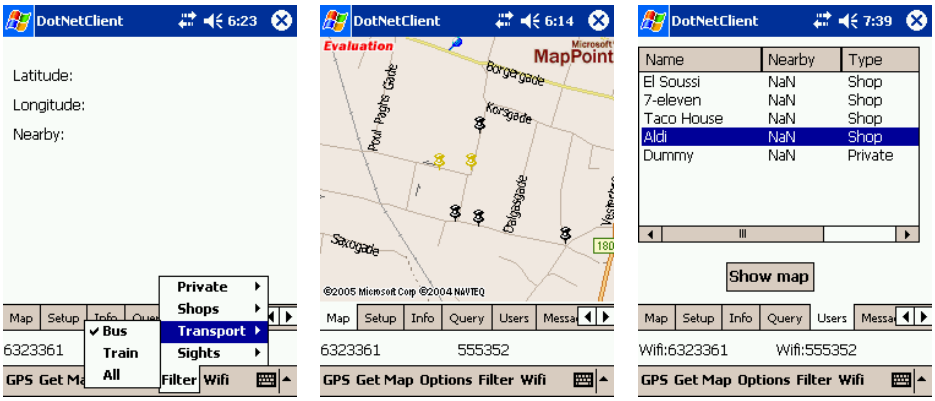
SPRQ-træ i distribueret DOPRI system

Main-memory baseret datastruktur i LBS system

Dat6 Speciale Projekt



Gruppe D629b	Vejleder
Kim Algreen	Bent Thomsen

Det naturvidenskabelige fakultet
Aalborg Universitet

Institut for Datalogi

TITEL:
SPRQ-træ i distribueret DOPRI system

UNDERTITEL:
Main-memory baseret datastruktur i LBS system

SEMESTERPERIODE:
DAT6 Speciale,
1. Februar 05 - 15. Juni 05

PROJEKTGRUPPE:
D629b
Kim Algreen, algreen@cs.aau.dk

VEJLEDER:
Bent Thomsen, bt@cs.aau.dk

ANTAL KOPIER: 5

ANTAL SIDER: 112

SYNOPSIS:

Denne rapport omhandler udviklingen af en type af distribuerede lokationsbaserede service systemer, kaldet DOPRI systemer, samt en main-memory baseret datastruktur, baseret på PR-Quadtræet, kaldet SPRQ-træ, til indeksering af to dimensionelle objekter. Systemet udvikles på baggrund af en undersøgelse af forskellige positionsbestemmelses teknikker, samt hvilken programmeringsteknologi der kan anvendes til at opbygge et distribueret DOPRI system.

Systemet skal kunne håndterer mange brugere, hvis position ofte skal opdateres, kunne dække et stort område, og kunne bruges både indendørs og udendørs. Den udviklede datastruktur benyttes i systemet, og skal understøtte kravene til systemet. Dvs. først og fremmest kunne foretage positionsopdateringer med stor hastighed. Derfor er datastrukturen main-memory baseret, da main-memory operationer er væsentlig hurtigere end disk operationer. Der undersøges om det er muligt og hensigtsmæssigt, at have både datastrukturen, samt en del af systemets information i main-memory, frem for på disk.

The Faculty of Engineering and Science
University of Aalborg

Department of Computer Science

TITLE:
SPRQ-tree in distributed DOPRI system

SUBTITLE:
Main-memory based data structure in LBS system

SEMESTER PERIOD:
DAT6 Master Thesis,
1st. February 05 - 15th June 05

PROJECT GROUP:
D629b
Kim Algreen, algreen@cs.aau.dk

SUPERVISOR:
Bent Thomsen, bt@cs.aau.dk

NO. COPIES: 5

NO. PAGES: 112

SYNOPSIS:

This report is about the development of a kind of distributed location based service system, named a DOPRI system, and a main-memory based datastrukture, based on a the PR-Quadtree, named SPRQ-tree, for indexing two dimensional objects. The system is developed on the basis of an examination of different positioning techniques, and what programming technology can be used to build a distributed DOPRI system.

The system should be able to handle many users, whose positions must often be updated, cover a wide area, and be used both indoors and outdoors. The developed data structure is used in the system, and must support the system requirements. This means first and foremost be able to make position updates fast. This is why the data structure is main-memory based, because main-memory operations are significantly faster than disk operations. It is examined whether it is possible and appropriate to have both the data structure, and a part of the systems information in main-memory, instead of on disk.

Forord

Denne rapport er skrevet som et DAT6 speciale projekt ved Institut for datalogi ved Aalborg Universitet. Rapporten er udarbejdet af gruppe D629b, inden for Database og Programmerings Teknologi afdelingen.

Aalborg, d. 15 Juni 2005

Kim Algreen

Cd

Den vedlagte cd indeholder følgende

Systemet:

.NET del
DotNetClient
Datastructure
Server
RemoteObject
RemoteServer
DotNetWS

Java del
WifiTracker

Databasen
Datdb_data.MDF
Datdb_Log.LDF

Hjælpeprogrammer:

.NET
DotNetStumbler (bruges til at lokationsbestemme steder via GPS og wifi)
Java
Stumbler (giver oplysninger om wifi APs til DotNetStumbler)
MapLoaderKA (loader information om APs ind i Place Lab database)

Rapporten i pdf format

Indhold

Kapitel 1

Introduktion

Indholdsfortegnelse

Der er indenfor de seneste år sket store fremskridt inden for udviklingen og udbredelsen af trådløs dataoverførsel, mobile apparater og positioneringsteknologi. Udviklingen tyder på at positioneringsteknologi, i form af GPS og GSM positionering vil blive standard på mobiltelefoner i løbet af de kommende år. Der findes på nuværende tidspunkt mobiltelefoner der kan tilsluttes en GPS modtager via USB eller bluetooth, og de første mobiltelefoner med indbyggede Assisted GPS (AGPS) chips er kommet på markedet [27, 28]. Desuden er AGPS chips der kan give indendørs positionsbestemmelse udviklet, men endnu ikke integreret i kommercielt tilgængelige mobile apparater. GSM positionering indbygges i mobile netværk i både Europa og USA i forbindelse med E112 [40] og E911 [11], med det formål at kunne lokationsbestemme nødopkald. Derudover kan den store udbredelse af wifi access points (AP) benyttes til positionering. Denne integration af positionsteknologi i mobiltelefoner og PDAer gør, at der opstår et stort marked for lokationsbaserede services (LBS).

Der er mange anvendelsesmuligheder for LBS. F.eks. navigering, information om en brugers omgivelser udfra brugerens position, også kaldet lokationsbaseret information, lokationsbaserede spil, lokationsbaseret overvågning af børn og kæledyr,

lokalisering af personer og lokationsbaserede reklamer. Disse og andre LBS kan gå hen at blive en integreret del af folks hverdag. Desuden vil mobilselskaberne kunne bruge LBS, som salgsargument på lige fod med eksisterende features som SMS, internet, kameraer osv.

Trådløs dataoverførsel som GPRS og UMTS bruges i forbindelse med AGPS, og kan også udnyttes af en række LBS. F.eks. kan en mobiltelefons position sendes til en server, hvorfra andre mobiltelefoner kan modtage positionen. Trådløs dataoverførsel gør det muligt, at have mobile apparater til at tilgå store mængder lokationsbaseret information, placeret på en server. Information der løbende kan tilføjes og opdateres fra en række forskellige informationsudbydere. Informationsudbydere kan være offentlige instanser som vejdirektoratet, der giver oplysninger om vejarbejde og trafikpropper, butikker der giver oplysninger om varer, tilbud og åbningstider eller busselskaber der giver information om afgangs -og ankomsttider på busser. Infomationen som udbydes af informationsudbyderne kan være knyttet til en række lokationsbestemte områder. F.eks. busstoppesteder, busser, butikker, seværdigheder, personer, veje, pladser, taxaer osv. Information knyttet til en vej vil kunne bruges til at informere bilister og cyklister om et nuværende eller kommende vejarbejde. Information om butikker vil give mulighed for at finde butikker og varer i det område en bruger befinder sig i. Det vigtigste ved informationen er, at den er knyttet til en bestemt lokation og er interessant for en bruger.

En LBS der tilbyder lokationsbaseret information skal understøttes af en række informationsudbydere, for at være interessant. Jo flere informationsudbydere, jo mere interessant bliver systemet for en bruger. Antallet af informationsudbydere, afhænger i høj grad af udbredelsen af systemet. Derfor er det vigtig, at de mobile apparater der understøtter en sådan LBS, kan købes til en pris der gør det muligt for et bredt udsnit af befolkningen at anskaffe sig en. Derudover skal de have en dataoverførselsforbindelse med en hastighed der gør, at kommunikationen med systemets server foregår med en acceptabel hastighed, og de skal benytte positioneringsteknologi der har en acceptabel præcision og anvendelighed.

Et system der tilbyder ovenstående LBS type vil efterfølgende blive beskrevet som Dynamisk Opdateret Positions Relateret Information (DOPRI) system. Kort skitseret har DOPRI systemer følgende egenskaber.

- Positioner bliver løbende opdateret.
- Information er relateret til positioner.
- Information ændres ofte.

En ting der kendetegner DOPRI systemer er behovet for effektivt at kunne foretage positionsopdateringer, samt effektivt at kunne udføre funktioner, hvor der er behov

for at have kendskab til den indbyrdes afstand på de objekter der repræsenterer positions relateret information. Et eksempel på en sådan funktion er område forespørgsler, hvor der laves forespørgsler på information der vedrører det område en bruger befinder sig i. F.eks. følgende forespørgsel "Find alle butikker inden for et område på 100*100 meter". For at dette kan ske effektivt skal objekterne arrangeres på en måde der afspejler deres indbyrdes afstand. Dette er nødvendig for at undgå at undersøge afstanden fra et objekt til samtlige objekter i systemet. Dette gøres ved at indeksere objekterne i henhold til x og y koordinater, også kaldet spatial indeksering i to dimensioner. Hvilket medfører, at indekseringen skal opdateres, når objekternes positioner ændres. Traditionelle relationelle databaser kan kun indeksere i én dimension, og er desuden ikke velegnet til hyppige ændringer i indekseringen. Derfor benyttes en datastruktur til at indeksere objekterne. Denne datastruktur er typisk en R-træ [56] eller Quad-træ [63] variant, placeret i main-memory, eller på disk. Der findes databaser der har en datastruktur til indeksering af spatial data i multiple dimensioner indbygget, f.eks. Oracle9i [35] og MySQL [29]. Hvor MySQL benytter en disk baserede variant af R-træet, og Oracle9i benytter disk baserede varianter af både Quad-træet og R-træet [64], til indekseringen. Der findes desuden en udvidelse af sql standarden, der gør det muligt at foretage spatiale sql kald [60].

Dette projekt omhandler et DOPRI system, hvor en række informationsudbydere har mulighed for at tilføje og ændre positionsbestemt information. Denne information kan så tilgås af brugere via klienter på mobile apparater. Brugerne selv bidrager også med information, herunder deres position. Til indeksering af positionsbestemte objekter foreslås en løsning der omfatter en datastruktur i main-memory, der effektivt kan indeksere data objekter i to dimensioner, kombineret med en traditionel relationel database. Datastrukturen skal udover indekses også indeholde en del af et objekts data, hvorved der kan foretages en række forskellige område forespørgsler uden at databasen benyttes. I litteraturen om datastrukturer til indeksering af multidimensionel spatial data, er de indekserede objekters data placeret i en database og datastrukturen indeholder kun indekses til databasen. Men der er store fordele forbundet med at benytte main-memory til at håndtere både positionsopdateringer og område forespørgsler, da tiden for database operationer måles i millisekunder, mens main-memory operationer måles i mikrosekunder.

Datastrukturen baseres på en variant af Quadtræet, ved navn PR-Quadtræ [63] (P står for point og R står for region), og udnytter den antagelse, at der i en datastruktur, der indekserer objekter der repræsenterer geo-refererede enheder som personer og bygninger, aldrig vil forekomme mere end en acceptabel mængde objekter inden for et enkelt område, tilknyttet en blad-knude, hvis hver blad-knude dækker et tilpas lille område. For at understøtte denne antagelse kan nævnes følgende tal. Hvis Danmark inddeles i områder på 50*50 meter, så vil der være behov for cirka 22.9 millioner knuder i træet, hvis hver knude har fire child-knuder, og træet vil have en højde på 12. Det store antal blad-knuder betyder, at det område én blad-

knude dækker er så lille, at der ikke vil forsamle sig en så stor mængde objekter på et sådant område, at det vil gå ud over hastigheden på område forespørgsler.

En knude består af referencer til child-knuder og til parent-knuden, samt af fire heltal der angiver start og slut værdi på x og y akserne i den kvadrat der angiver det område en knude dækker. Det betyder at der bruges få bytes per knude. Hvis vi antager, at der bruges 200 bytes per knude, vil træet i ovenstående eksempel bruge 4.580 GB hukommelse. For at få en ide om hvor meget RAM der er tilgængelige i main-memory i øjeblikket kan nævnes følgende tal. En 32-bit processor kan allokere 4 GB RAM ialt, og 2 GB til en enkelt applikation. 64-bit processorer kan allokere ialt 136 GB, og 4 GB til en enkelt applikation [1]. Windows Server 2003 Enterprise Edition kan allokerer 32 GB i 32-bit versionen og 64 GB i 64-bit versionen [52], mens Windows XP Professional kan allokerer 4 GB i 32-bit versionen og 128 GB i 64-bit versionen [53,54]. Motherbords kan allokerer op til 32 GB [5]. Det betyder, at for at kunne have en datastruktur i main-memory, hvor opdelingen af et område er så lille, at kun få brugere vil være forsamlet der på et givet tidspunkt, er det nødvendig at have flere instanser af datastrukturen. Enten placeret på samme maskine eller distribueret ud på flere maskiner. Det betyder at det DOPRI system som datastrukturen indgår i skal gøres distribueret for at kunne håndterer mange knuder og et stort antal brugere i main-memory. For at tilpasse systemet til det område det skal dække kan hver instans af datastrukturen tilpasses den brugertæthed der er i det område instansen dækker. Det vil sige små områder ved byer og større områder ved tyndt befolkede landområder.

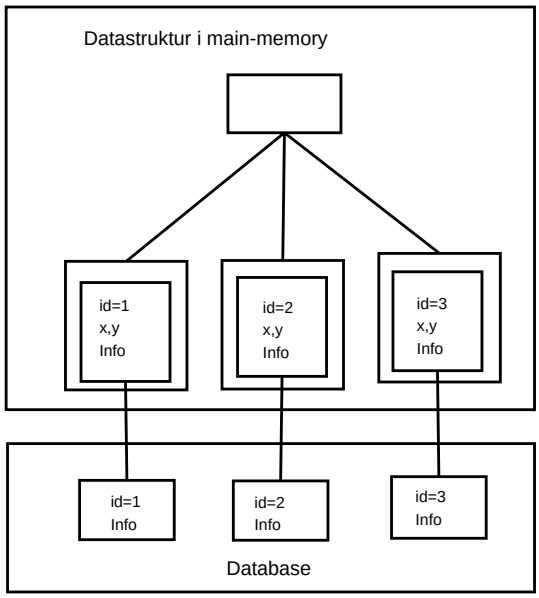
På grund af opdelingen i små områder og de indekserede objekters bevægelsesmønster, vil datastrukturen ikke behøves at tilpasse sig når objekterne ændrer positioner, eller der tilføjes objekter. F.eks. ved at ændre størrelse på de områder der er tilknyttet hver knude, eller ved at splitte, fjerne eller tilføje knuder. Datastrukturen vil med andre ord kunne gøres statisk, hvilket giver en høj hastighed på positionsopdateringer, da de ikke resulterer i ændringer i træet. Hastigheden på område forespørgsler vil også være høj da inddelingen i små områder gør, at en top-down traversering af træet hurtig udelukker del-træer der ligger uden for søge området. Da det at datastrukturen er statisk, er det der kendetegner den, og da den er en variant at PR-quadtræet, er den blevet døbt Statisk PR-Quadtræ (SPRQ-træ).

I dette projekt undersøges den teknologi der i øjeblikket er tilgængelig til at udvikle et distribueret DOPRI system, samt hvilken teknologi der i løbet af de kommende år vil være tilgængelig. Det drejer sig både om positioneringsteknologi, trådløs dataoverførsel og programmeringsteknologi. På baggrund af undersøgelsen udvikles et distribueret DOPRI system der anvender SPRQ-træet, og som kan dække et stort område, klare mange brugere og være anvendelig for en bruger både udendørs og indendørs. Dette DOPRI system benyttes til at vise hvordan et distribueret DOPRI system kan laves, samt til at vise at SPRQ-træet er anvendelig i et sådant system. Desuden bruges det til at undersøge og underbygge følgende påstande, angåen-

de SPRQ-træet, og opdeling af bruger information i en main-memory del og en database del.

> I et distribueret DOPRI system, der indekserer geo-refereret information om objekter, der repræsenterer fysiske enheder, som f.eks. butikker og personer, giver et main-memory baseret SPRQ-træ der indekserer spatiale data objekter i to dimensioner, det bedste forhold mellem hastigheden på område forespørgsler og positionsopdateringer. Desuden vil det forbedre hastigheden på område forespørgsler, at opdele en brugers information i en main-memory del og en database del frem for kun at have indekses i main-memory og al information i en traditionel relationel database, idet den main-memory baserede del af informationen gør det muligt, at udføre en række område forespørgsler uden at involvere databasen.

De to påstande skal ses i forhold til at benytte en traditionel relationel database, kombineret med en anden main-memory baseret datastruktur end SPRQ-træet, som kun indeholder information om indekses, eller en database der benytter en disk baseret datastruktur til indeksering af multidimensionel data. Figur 1.1 illustrerer opdelingen af et objekt i en main-memory del og en database del.



Figur 1.1: Opdeling af et objekts data

Denne rapport består overordnet set af beskrivelse og evaluering af SPRQ-træet, samt beskrivelse og evaluering af et distribueret DOPRI system, samt den teknologi der ligger til grund for dette system. Denne introducerende del af rapporten

indeholder, foruden ovenstående indledning, en kravsspecifikationen som systemet baseres på, et afsnit om LBS, et afsnit om relateret litteratur, samt et afsnit om de benyttede apparater. Den resterende del af rapporten er opdelt i fire hoveddele.

Del 1 er en gennemgang af relevant baggrundsviden, hvilket indbefatter et kapitel om positionsbestemmelse, samt et kapitel om den anvendte programmeringsteknologi.

Del 2 er en beskrivelse af udviklingen af systemet, herunder arkitektur, analyse, design og implementation. Dette indbefatter også design og implementation af SPRQ-træet.

Del 3 består af test og evaluering af systemet og SPRQ-træet.

Del 4 er afrunding, og består af konklusion og fremtidsperspektivering.

Desuden giver appendiks A en illustreret gennemgang af klienten.

1.1 Kravsspecifikation

Hovedformålet med systemet er, at understøtte de krav der blev opstillet i indledningen, dvs. være et eksempel på et distribueret DOPRI system der benytter SPRQ-træet til indeksering af objekter, og som har en opdeling af information i en main-memory del og en database del. Desuden skal systemet kunne benyttes både udendørs og indendørs, kunne håndtere mange brugerer og dække et stort område. Udover kravene fra indledningen skal det også kunne placerer brugere på et kort, i henhold til deres position. Systemet skal kunne udføre funktioner der er typiske for et DOPRI system. Det indbefatter positionsopdateringer, og en række område forespørgsler. Desuden skal brugere og informationsudbydere kunne ændre i den information systemet indeholder om dem. Formålet med denne kravsspecifikation er derfor at få afgrænset systemet, og få fastlagt hvilken information der skal indgå i systemet.

Der er blevet valgt fire typer information til at indgå i systemet, det er information om transportinfrastruktur, repræsenteret ved stoppesteder, butikker, seværdigheder og privatpersoner. Information om stoppesteder gør, at en bruger hurtig kan få oplysninger om hvornår der går et bus eller et tog til det sted hvor vedkommende skal hen, samt placeringen af det stoppested det foregår fra. Information om butikker indbefatter placering af butikkerne, hvilke produkter de sælger og til hvilke priser, samt åbningstider. Dette er nyttigt i en række situationer, f.eks. hvis en bruger ønsker at finde den butik i nærområdet, der er billigst med et bestemt produkt, eller at brugeren skal finde en vare som vedkommende ikke ved hvor sælges, f.eks. i forbindelse med gavekøb. Information om seværdigheder er interessant for turister

der ikke er bekendt med det område de befinder sig i, og ønsker at finde de seværdigheder der er i området. Endelig er der information om privat personer, der er medtaget da størstedelen af brugerne vil være privat personer, og det vil være interessant for en bruger, at kunne få information om andre brugere.

De fire valgte informationstyper skal opfattes som forskellige brugertyper, dvs. al information er knyttet til brugere, der er opdelt i fire forskellige brugertyper. Det drejer sig om følgende fire brugertyper, privatbrugere, butiksbrugere, seværdighedsbrugere og transportbrugere, der repræsentere information om stoppesteder. Derved skelnes der ikke mellem brugere og informationsudbydere, idet alle fire brugertyper kan benytte den samme grundfunktionalitet. Grundfunktionaliteten består i område forespørgsler, dvs. at kunne finde brugere inden for et bestemt område givet et kriterie, få brugere vist på et kort, få vist information om en bruger, samt positionsopdatering. Derudover har en bruger adgang til at ændre den information der er tilknyttet den pågældende bruger. Butiksbrugere skal have mulighed for at kunne tilføje produkter og ændre priser på produkter. Både butiksbrugere og seværdighedsbrugere skal have mulighed for at ændre oplysninger om åbningstider. Transportbrugere skal kunne ændre oplysninger om ruter, f.eks. hvornår en rute afgår fra et stoppested. Det at der ikke i systemet skelnes mellem privat personer og de andre brugertyper, gør det muligt for alle typer brugere at opdatere deres position. Dette vil for de tre andre brugertyper ske yderst sjældent, men der findes tilfælde hvor det vil være nyttigt, f.eks en butiksbruger der repræsenterer en isbil, eller en seværdighedsbruger der repræsenterer et karnevalsoptog.

Den valgte information og funktionalitet gør at systemet kan bruges til at demonstrere de centrale funktioner i DOPRI systemer. Noget af informationen skal placeres i main-memory og noget i databasen. Den del der placeres i main memory skal indekseres i datastrukturen, i form af objekter der indeholder nok brugerinformation til at databasen ikke skal benyttes i forbindelse med positionsopdatering og en række område forespørgsler.

Det er som nævnt også mål for systemet at det skal kunne vise et kort med brugere på, samt benyttes både udendørs og indendørs. Det første betyder at der skal bruges et Geografisk Informations System (GIS). Det andet betyder at der skal bruges en positioneringsteknologi der kan bruges både udendørs og indendørs, eller benyttes flere positioneringsteknologier der tilsammen kan bruges både udendørs og indendørs.

1.2 Lokationsbaserede services

Lokationsbaserede services (LBS) er et relativt nyt fænomen, der spås en stor fremtid, specielt i kombination med mobile apparater. Brug af IP addresser på internettet

til at afgøre hvilket land besøgende på et site kommer fra, og tilpasse sitet til dette land, er i øjeblikket den mest udbredte form for LBS. Men der hvor væksten for alvor kommer til at ske vil være indenfor LBS til mobile apparater, der kan tilsluttes, eller er integreret med, positioneringsteknologi, og har en trådløs netværksforbindelse. Sådanne apparater giver mulighed for at lave en række forskellige services der benytter bl.a. position, bevægelsesmønstre, brugerinformation og lokationsbaseret information.

De i øjeblikke mest udbredte LBS systemer til mobile apparater er navigeringssystemer, der får positionsinformation fra en GPS modtager og relaterer dette til et geokodet kort. Hvilket vil sige et kort hvor punkter på kortet er relateret til længde og breddegrads koordinater. Et navigeringssystem kan vise brugerens position på et kort, løbende opdatere positionen, og finde vej fra punkt A til punkt B. De nyeste navigeringssystemer kan desuden modtage traffikinformation og nogle kan vise Points Of Interests (POI), hvilket f.eks. kan være parkeringshuse, tankstationer eller hoteller. En anden udbredt type af LBS systemer er nødsystemer. Nødsystemer bruges til at finde ud af hvor et nødopkald kommer fra, i tilfælde af, at den nødstedte ikke ved eller ikke er i stand til, at fortælle hvor vedkommende befinder sig. I USA er det vedtaget ved lov, at der skal implementeres lokationsbestemmelse i mobile netværk, i form af E911 [11]. Mens det i EU, i form af E112 [40], kun er opsat som et mål medlemslandende arbejder henimod.

Der findes også spil der benytter positioneringsteknologi, bla. geo-caching [16], hvor en skat (cache) er gemt på en position der er kendt af deltagerne, spillet går så ud på at være den første der finder skatten. En anden form for lokations baserede spil tilbydes af Gizmondo [17]. Gizmondo er en mobil spille konsol med indbygget GPS modtager. Den tilbyder en række LBS, bl.a. forespørgsler som "Hvor er jeg"og "Find nærmeste", hvor bl.a. den nærmeste pengeautomat eller McDonalds kan findes. Den kan også bruges som "Buddy finder", dvs. finde en anden person, hvis vedkommende også har en Gizmondo. Desuden kan den bruges til lokations baserede spil, hvor spilleren bevæger sig rundt på gader og stræder. Der er i øjeblikket kun et sådant spil, nemlig spillet "Colors", der handler om bandekrig. Gizmondo er endnu ikke tilgængelig i Danmark. Der findes også lokations baserede spil til mobiltelefoner, f.eks. Mogi, der går ud på at gå rundt og samle virtuelle genstande op. Mogi er indtil videre i betatest i Tokyo. Tilsidst kan nævnes et spil som ikke kræver GPS udstyr, idet det benytter sig af celle-id GSM positionering. Spillet er Gunslingers, og det går ud på, at gå rundt og lokalisere andre spillere og "skyde"dem med en mobiltelefon. Gunslingers kan indtil videre kun spilles i Singapore, hvor celle-id kan give en præcision på 50 meter. Overordnet set er lokations baserede spil i opstartsfasen, og udbredelsen afhænger af hvor udbredt positioneringsteknologi bliver.

Det er ikke kun spil der benytter sig af brugerinformation, og løbende sender en brugers position til omgivelserne over et trådløst netværk. Det i øjeblikket mest

udbredte af denne type LBS er flådestyringssystemer, der benyttes af fragtfirmaer til at holde styr på hvor deres vogne eller skibe er henne. Sådanne LBS kan også bruges til, at omgivelserne er opmærksomme på, og kan reagere på en brugers tilstedeværelse. Et eksempel herpå er trådløse reklamer, dvs. reklamer fra butikker i det område en bruger befinder sig i, der er skræddersyede ud fra brugeroplysninger. DOPRI systemer, som dette projekt beskæftiger sig med, minder om ovenstående type, idet de også kan gøre brug af brugeroplysninger. Hvilket er tilfældet med det system der udvikles i forbindelse med projektet.

Sammenkoblingen af position og omgivelsernes kendskab til en bruger vil kunne danne basis for utallige LBS, idet det gør det muligt at give en bruger individuel tilpasset information om omgivelserne. Muligheden for at få løbende opdaterede oplysninger om ens omgivelser, samt at omgivelserne har kendskab til en, vil kunne ændre vores adfærdsmønstre på samme måde som mobiltelefoner har gjort det.

1.3 Relateret litteratur

Det har ikke været muligt at finde litteratur der omhandler opdelingen af et objekts data i en main-memory del og en database del, udover en artikel der nævner at det er en mulighed. Derimod er der adskillige artikler om datastrukturer til indeksering af objekter i to dimensioner, og tilpasninger af disse datastrukturer der gør dem mere egnet til at håndterer hyppige opdateringer af indeksene. Hvilket er nyttigt når positioner benyttes som indekses, samtidig med at et objekt løbende opdaterer sin position, hvilket er tilfældet i DOPRI systemer. Desuden antager størstedelen af litteraturen at den beskrevne datastruktur skal placeres på disk, og antal disk blokke medtages derfor som enhed i kompleksitets angivelser. Hvilket gør det svært at sammenligne disse angivelser med main-memory baserede datastrukturer som SPRQ-træet.

De oftest benyttede datastrukturer til at indeksere multidimensionel spatial data er R-træet (R står for region) og Quadtræet, samt varianter af disse. F.eks. R+ [38] og R* [39] varianterne af R-træet, samt PR-Quadtræ og P-Quadtræ varianterne af Quadtræet. Der vil i dette afsnit blive givet en beskrivelse af disse to typer træer, samt de af deres varianter der er mest velegnet til dette projekts problemdomæne.

Quadtræet og R-træet danne basis for en række datastrukturer, ofte tilpasset et bestemt problemdomæne. F.eks datastrukturer der medtager tid som en ekstra dimension [55, 57, 65], og bl.a. benyttes i forbindelse med spatio-temporale databaser, datastrukturer der benytter kendskab til et transportnetværk [61], samt datastrukturer der benytter en lineær ligning til at udregne fremtidige positioner [66]. Lineær ligninger og kendskab til transportnetværk er ikke brugbar i dette projekts problemdomæne, dels fordi brugere ikke nødvendigvis bevæger sig via et transportnetværk,

og dels fordi de ofte skifter retning og hastighed. Tid som ekstra dimension kan bl.a. bruges til at lave forespørgsler som "Hvornår er taxa x ved adresse y". Men det vil have en begrænset funktionalitet, da der kun er kendskab til afstande i fugleflugtslinie, og ikke til transportnetværk.

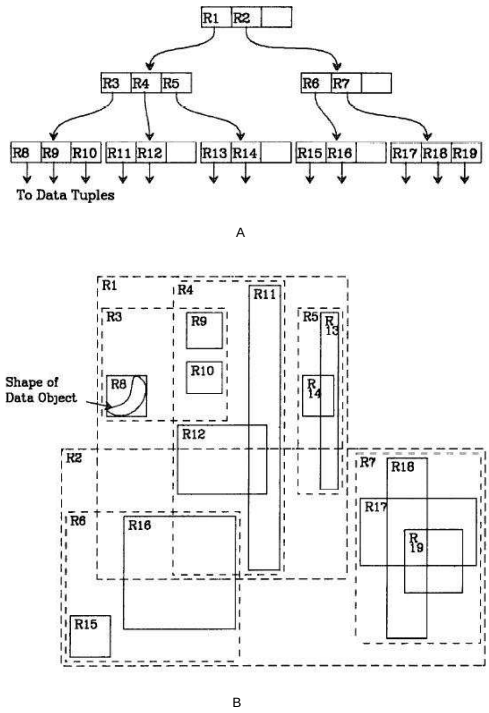
Der vil ikke blive sammenlignet med databaser der kan indeksere multidimensionelle objekter, idet de bygger på disk baserede R-træer og Quadtræer. Hvis SPRQ-træet er mere effektivt end main-memory baserede R-træer og Quadtræer vil det også være mere effektivt end disk baserede. Databaserne har dog den fordel at der kan foretages spatiale sql kald på dem, mens forespørgsler i systemet i dette projekt sker ved almindelige sql kald, i forbindelse med de algoritmer der håndterer de forskellige forespørgsler, eller kun med algoritmer, ved de forespørgsler der kun benytter main-memory data. Spatiale sql kald giver en bruger eller administrator mulighed for helt præcist at angive hvilken data der skal findes, uden at der skal laves en algoritme specielt til det. Disk baserede datastrukturerer har desuden den fordel at de har væsentlig mere RAM til rådighed, end main-memory baserede. Men det er ikke et mål for projektet at kunne foretage meget specifikke forespørgsler, derimod har projektet som mål at opnå højest mulig hastighed på område forespørgsler og positionsopdateringer. Derfor vil en en ren disk baseret løsning ikke kunne bruges, da disk operationer er væsentlig langsommere end main-memory operationer.

Både R-træet og Quadtræet vil blive beskrevet, herunder en variant af hver af dem, nemlig LUR-træet [21] og PR-quadtræet [63] (P står for point og R for region). Disse varianter er tilpasset et problemdomæne, der ligner projektets problemdomæne. SPRQ-træet er en variant af PR-quadtræet og sammenligningen skal vise at SPRQ-træet er den mest hensigtsmæssige variant. LUR-træet er den R-træ variant der bedst passer til problemdomænet, idet den er designet til at kunne håndterer hyppige positionsopdateringer, samt er main-memory baseret. PR-Quadtræet og LUR-træet indgår i en sammenligning med SPRQ-træet i afsnit 7.1.

1.3.1 R-træet

R-træet opdeler et område i delområder, hvor hver knude dækker et delområde. Objekter, i form af punkter eller regioner, tilknyttes blad-knuder. Et centralt begreb ved R-træet er Minimum Bounding Rectangle (MBR). En MBR er den mindst mulige rektangel der udspænder en mængde af objekter, hvor objekterne kan være punkter eller områder. MBR for et R-træs rod udspænder alle objekterne i træet, og dens child-knuders MBRs udspænder hver en del af rodens MBR. Child-knudernes child-knuder udspænder ligeledes en del af deres parent knudens MBR. Denne opsplitning af rod knudens MBR fortsætter i takt med at der tilføjes objekter til træet. MBRs kan være overlappende, men en knudes MBR må aldrig overskride parentknudens MBR. Det er tilladt at blad-knudernes MBRs tilsammen kun giver en par-

tiel dækning af rodens MBR. Desuden fastsættes en minimum og en maksimum værdi for antallet af objekter en blad-knude må have, og hvor mange child-knuder en ikke-blad knude må have. Dette gøres for at holde træet balanceret. Figur 1.2 viser hvordan et R-træ er opbygget, figur 1.2 A viser træstrukturen, mens 1.2 B viser hvilke områder træets MBRs udspænder.



Figur 1.2: Eksempel på R-træ [56]

R-træet håndterer opdateringer ved at fjerne det opdaterede objekt og derefter indsætte det igen. Hvis objektets nye position ligger udenfor dets nuværende knudes MBR kan det resultere i en række omrokeringer i træet, f.eks. at knuder bliver splittet, MBRs justeres eller der tilføjes en ny knude. Dette gør at R-træer ikke er velegnede til at håndtere hyppige opdateringer, da hver opdatering potentiel resulterer i omfattende ændringer i træet. Da R-træer tillader overlappende MBRs er det muligt at følge forskellige stier for at komme til det samme objekt, dette er specielt uhensigtmæssigt i forbindelse med søgning af objekter indenfor et bestemt område.

Der er blevet foreslået adskillige tilføjelser og ændringer til R-træet, for at gøre det mere velegnet til at håndtere hyppige opdateringer. F.eks. STAR-træet [62], der formindsker overlapningen af søskende knuder, og LUR-træet, der håndterer opdateringer ved at tjekke om et objekts nye position er inden for dets nuværende knudes MBR, hvis det er tilfældet flyttes objektet blot. Hvis objektets nye position derimod er uden for dets nuværende knudes MBR, foreslås følgende to metoder.

Forstørre MBR på objektets nuværende knude For at undgå at objektet skal flyttes over i en ny knude, så udvides MBR på objektets nuværende knude. Dette kan kun lade sig gøre hvis det ikke medfører at MBR på objektets nuværende knude overskrider MBR på den nuværende knudes parent. Metoden medfører et større overlap, hvilket går ud over søge-hastigheden.

Genindsættelse i parent-knude Da en ny position ofte er i umiddelbar nærhed, vil det være nyttigt først at undersøge om objektets nye position er i parent knuden til objektets nuværende knude. Denne process foregår rekursivt indtil den rigtige knude findes.

Foruden ovenstående opdateringsteknikker, benytter LUR-træet en direkte link til objekterne, i form af en hashtabel, hvilket gør det hurtigt at finde et bestemt objekt udfra en unik hashværdi.

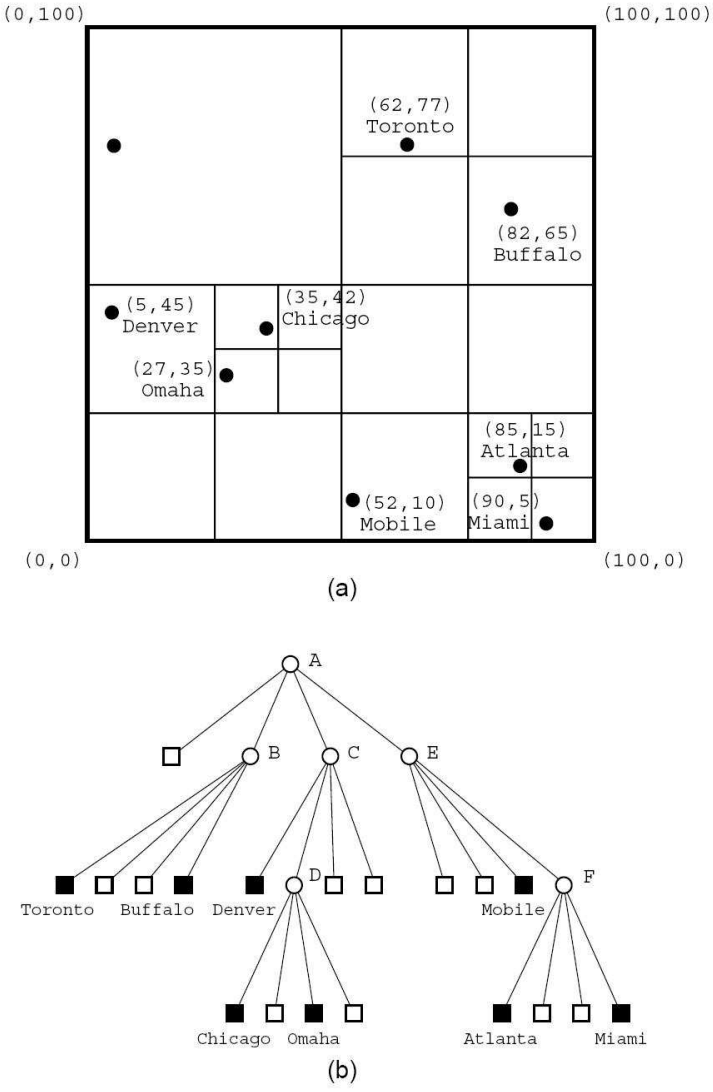
1.3.2 Quadtræet

Quadtræet er en datastruktur der rekursivt dekomponere et område i fjerdedele, dvs. hver ikke-blad knude har fire child knuder. Objekter er ligesom ved R-træet tilknyttet blad-knuder, og kan have multiple dimensioner. Quadtræets oprindelige formål var at indeholde to dimensionelt billeddata, og der findes en udgave af det til tre dimensioner, nemlig Octettræet. Der findes adskillige varianter af Quadtræet og de adskiller sig fra hinanden ved følgende.

1. Typen af data de skal repræsentere, f.eks. linier, regioner eller punkter.
2. De principper der styrer dekomponeringen, f.eks. hvor mange punkter der må være i et område.
3. Opløsningen, dvs. størrelsen på delområderne. Det bliver f.eks. påvirket af, om det er afgjort på forhånd hvor mange gange det samlede område opdeles, eller om det afgøres af de objekter der tilføjes.

Den variant der er mest relevant i dette projekt er PR-Quadtræet, der bruges til at indeksere punkter i to dimensioner. I et PR-Quadtræ er hver knude tilknyttet en kvadrat, og hver blad-knude kan indeholde nul til mange punkter, alt efter hvad der passer bedst til det den skal bruges til. Opdelingen er desuden uafhængig af i hvilken rækkefølge punkter tilføjes. Figur 1.3 giver et eksempel på et PR-Quadtræ der indekserer punkter der repræsenterer byer, og som kun tillader et punkt i hver blad-knude. Da SPRQ-træet er en variant af PR-Quadtræet, vil det være en ikke-statisk version med samme egenskaber som SPRQ-træet der sammenlignes med. Grunden til dette er at, det ikke har været muligt at finde artikler der beskriver en

bestemt version af PR-Quadtræet, samt tilhørende algoritmer til positionsopdatering og område forespørgsler.



Figur 1.3: Overblik over PR-quadtræ [63]

1.3.3 Teknikker

Udover valg af en bestemt træstruktur, findes der en række teknikker der i visse sammenhænge er brugbare, bl.a. til at nedbringe antallet af opdateringer. F.eks. ved at tillade en vis upræcision, af den gemte position. Det vil især være nyttig

i forbindelse med brugere der bevæger sig langsomt, f.eks. fodgængere, herunder specielt fodgængere der bevæger sig rundt i det samme område. I [58] beskrives en række teknikker, som kort vil blive gennemgået i følgende liste.

Repræsentation af positioner som funktioner Ved at bruge en linear funktion til at udregne et objekts position på baggrund af oplysninger om hastighed og retning, er det ikke nødvendig at opdatere objektets gemte position. Det er specielt velegnet hvis objektet bevæger sig via et transportnetværk, f.eks. et vejnet, som man har kendskab til.

Brug af udløbstid Ved at fastsætte en udløbstid, kan brugere, der ikke har opdateret deres position inden udløbstiden udløber, slettes. Jo længere tid der er gået siden en bruger sidst opdaterede sin position, jo større er chancen for at den gemte position ikke længere stemmer overens med brugeres aktuelle position. Dette vil i nogle sammenhænge gøre brugerens gemte position nytteløs, og det kan derfor ligeså godt fjernes fra datastrukturen.

Brug af buffering Det er mere effektivt at foretage en samlet overførsel af data fra main-memory til disk, end det er at foretage mange små overførelser. Dette kan udnyttes ved at benytte en buffer i main-memory, hvor en mængde opdateringer ophobes før opdateringerne overføres til disk.

Brug af tilgængelig main-memory Lagring i main-memory er væsentlig hurtigere end lagring på disk. Derfor vil det kunne forøge et systems hastighed, hvis dele af en brugers data var placeret i main-memory, og andre dele placeret på disk.

Tage højde for bevægelsesbegrænsninger Der er ofte en række begrænsninger på, hvor en bruger kan bevæge sig. F.eks. er biler begrænset til vejnettet og skibe er begrænset til sejlbart farvand. Det kan udnyttes til at forudse fremtidige positioner, og derved nedbringe antallet af opdateringer.

Ovenstående liste er general for datastrukturer til indekseringen af multidimensionelle objekter der bevæger sig. I dette projekt benyttes punktet "Brug af tilgængelig main-memory", idet main-memory benyttes til både datastrukturen og en del af en brugers data. Repræsentation af positioner som funktioner, samt udnyttelse af bevægelsesbegrænsninger er ikke brugbar, dels fordi brugere ikke nødvendigvis bevæger sig via et transportnetværk, og dels fordi de ofte skifter retning og hastighed. Brug af buffering er ikke aktuelt, da der ikke skal overføres information fra main-memory til disk, på nær når serveren lukkes ned. En teknik som evt. kunne indgå i systemet er, først at opdatere et objekts position i datastrukturen, når det har flyttet sig en vis afstand. Dette kunne eventuelt håndteres på klienten på det mobile apparat.

1.4 Benyttede apparater

For at kunne få systemet til at virke skal der benyttes et apparat der kan modtage både GPS og wifi signaler og som har en GPRS forbindelse. Desuden skal det kunne afvikle både java og .NET CF applikationer. I forbindelse med projektet er der blevet stillet to mobile apparater og en GPS modtager til rådighed, nemlig en Nokia 6600 mobiltelefon, en iPAQ h5550 Pocket PC og en TomTom GPS modtager. Tilsammen opfylder de kravene som systemet stiller, idet GPS modtageren kan modtage GPS signaler, iPAQen kan modtage wifi signaler, samt køre java og .NET CF, og Nokiaen har en GPRS forbindelse. Løsningen er så at skabe en bluetooth forbindelse mellem GPS modtageren og iPAQen, hvor systemets klient er placeret, og mellem iPAQen og Nokiaen, så iPAQen kan benytte Nokiaens GPRS forbindelse.

For at iPAQen kan køre java applikationer skal den have installeret en JVM, og til dette formål benyttes IBMs J9 JVM, der indgår i WebSphere Everyplace Micro Environment [45].

Del I

Baggrundsviden

Kapitel 2

Positionsbestemmelse

Indholdsfortegnelse

Positionsbestemmelse foregår ved at afgøre, hvor et punkt befinder sig i forhold til et eller flere punkter. I århundreder benyttedes himmellegemerne ved navigation på åbent hav, og efter opdagelsen af radiobølger blev disse benyttet til at bestemme position i forhold til der hvor radiobølgerne blev sendt fra. I slutningen af det 20 århundrede flyttede radiobølgesenderne ud i rummet, i form af satellit navigeringssystemer.

Til at identificere et bestemt punkt på jordoverfladen benyttes det geografiske koordinatsystem som definer to vinkler målt fra jordens centrum. De målte vinkler kaldes henholdsvis længde og breddegrader. Breddegrader har 0 grader ved ækvator og 90 grader ved hver af polerne. Længdegrader har 0 ved et nord-syd plan der skærer gennem Greenwich i England. Da jorden tilnærmelsesvis er en kugle kan den inddeles i 360 grader, og ved at kombinere længde og breddegrads vinklerne er det muligt, at tildele et unikt sæt koordinater til ethvert punkt på jorden. En anden måde at opdele jordoverfladen er Universal Transverse Mercator (UTM), der er en metode til at opdele jordens overflade i flade firkanter. Fordelen ved dette er, at det er lettere at udregne afstanden i meter mellem to punkter på et fladt kort, frem for med sfæriske koordinater.

Det geografiske koordinatsystem er ikke nødvendig at benytte i alle LBSer, det kan

i nogle situationer være nok at kunne angive at en bruger befinder sig i nærheden af et bestemt punkt. Det benyttes f.eks. i forbindelse med lokalisering via et GSM netværk, der i den simpleste udgave angiver en brugers positions udfra hvilken antenne brugeren er tættest på.

Enhver LBS har behov for en teknologi til positionsbestemmelse, derudover vil de fleste LBSer have behov for at kunne relatere fundne positioner til punkter eller områder på et kort. Den resterende del af dette kapitel beskæftiger sig med disse to områder.

2.1 Positioneringsteknologi

Dette afsnit omhandler de teknologier der kan benyttes til positionsbestemmelse.

2.1.1 Satellit Navigeringssystemer

Figur 2.1: Satellit navigeringssystem [44]

Satellit navigeringssystemer benytter satellitter i kredsløb om jorden til at fastlægge positioner, se 2.1. De fungerer ved at en modtager opfanger signaler fra et antal satellitter, hvorved modtageren kan udregne sin position. I øjeblikket findes der to satellit navigeringssystemer i drift, nemlig det amerikanske Globale Navigation System (GPS) og det russiske GLObal NAvigation Satellite System (GLONASS). Men på grund af Ruslands økonomiske problemer efter Sovjets sammenbrud, virkede kun 8 ud af de 24 satellitter i 2002. Men Rusland og Indien har indgået et samarbejde om at modernisere og vedligeholde GLONASS, og det skal efter planen være fuldt funktionsdygtig i 2007 [48]. EU har også planlagt et satellit navigeringssystem, som har fået navnet Galileo. Det skal efter planen være klar til kom-

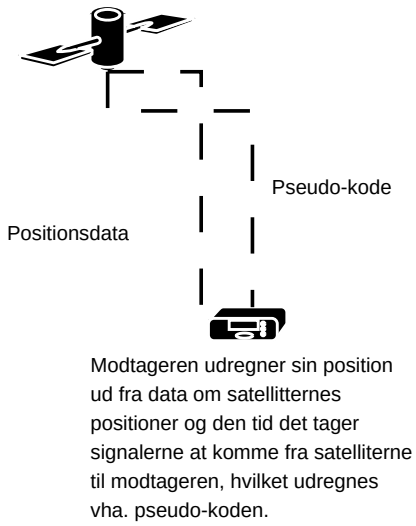
merciel brug i 2008 [47]. De nævnte systemer fungere efter de samme principper, og da GPS er det eneste fuldt fungerede system i øjeblikket vil det blive beskrevet i det følgende afsnit.

GPS

GPS er et satellit baseret navigationssystem udviklet og kontrolleret af USAs forsvarsministerium. Det har eksisteret siden 1978 og består af 24 satellitter i kredsløb omkring jorden. GPS giver en præcision på mellem 1 cm. og 20 meter. Hvor præcis målingen er afhænger af hvilke teknikker der benyttes og hvor GPS modtageren befinder sig, idet f.eks. bygninger kan forstyrre satellitsignalerne. En almindelig GPS modtager der modtager fire samtidige signaler vil have en præcision på cirka 5-10 meter. Selvom GPS fra starten var tilgængelig for alle, var det kun militæret der kunne benytte det med maksimal præcision. For civile var der indbygget en begrænsning der gjorde at præcisionen kun var omkring 80 meter. Denne begrænsning blev fjernet 1 maj 2000, så alle nu kan benytte GPS med maksimal præcision [49]. Måden hvorpå GPS udregner en position kan kort beskrives med følgende fire punkter.

- Modtageren modtager et pseudo-kode signal samt data om hvor satellitten befinder sig, se 2.2
- Afstanden til satellitterne findes ved at måle den tid det tager for et pseudo-kode signal at bevæge sig fra en satellit til modtageren.
- Modtageren skal have lagret oplysninger der giver mulighed for, ved hjælp af data modtaget fra satellitterne, at udregne hvor satellitterne befinder sig på et givet tidspunkt.
- Modtageren korrigerer for forsinkelser der opstår når signalet bevæger sig gennem atmosfæren.
- Brug af trilateration til at udregne positionen.

For at bestemme en position i en tre dimensional verden kræves det at modtageren benytter sig af samtidige målinger fra tre satellitter. Men det er en fordel også at bruge signalet fra en fjerde satellit. Det fjerde signal kan bruges til at modtageren kan korrigere sit ur, og da uret benyttes til at udregne afstanden til satellitterne kan urets øgede præcision øge præcisionen af positionsbestemmelsen. Grunden til at modtagerens ur ikke er så præcis som satellitternes er at den, i modsætning til satellitterne, ikke har et atomur, og dens ur vil derfor miste præcision langt hurtigere end satellitternes ure. Så fire signaler giver en bedre præcision for den aktuelle måling samt sørger for at modtagerens ur på længere sigt ikke mister præcision.



Figur 2.2: Dataudveksling mellem satellit og modtager

Derfor kræves det af GPS modtagere, at de har mindst fire kanaler og derved er i stand til at foretage fire samtidige målinger af signaler. Det er muligt at udregne en position ud fra kun to eller tre signaler, men det går ud over præcisionen [44].

Der findes forskellige teknikker til at forbedre præcisionen, f.eks. Differential GPS (DGPS), der har en præcision på 2-10 meter. DGPS benytter sig af stationære modtage stationer. Den stationære modtage station placeres et sted hvis position er blevet fastlagt med stor præcision. Fordi den stationære modtage station ved præcis hvor den selv er, samt ved præcis hvor hver satellit er kan den korrigere for fejl i målingerne af hvor lang tid signalerne er om at nå den. De korrigerede tal kan så benyttes af en modtager der bevæger sig inden for en kort afstand, relativ til afstanden til satellitten, til at udregne sin position med en præcision ned til et par meter. Grunden til at den stationære modtage stations korrigeringer også gælder for en modtager der bevæger sig inden for en relativ kort afstand er, at signalerne fra satellitterne må formodes at have passeret gennem en ens atmosfære for at nå modtage stationen og modtageren. DGPS systemer blev hovedsagelig udviklet for at få bedre præcision end de 80 meter civile kunne få før begrænsningen blev fjernet i 2000.

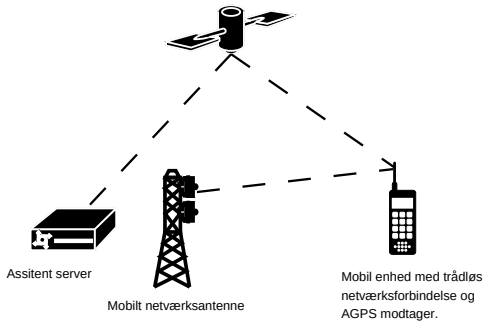
Der findes andre teknikker der kan forbedre præcisionen yderligere, herunder det amerikanske Wide-Area Augmentation System (WAAS), der er rettet mod luftfarten. WAAS benytter, udover stationære modtage stationer, specielle WAAS satellitter. WAAS er endnu under udvikling og kan for øjeblikket kun benyttes i øst og vest kyst regionerne i USA. Der bliver også udviklet lignende systemer i Europa og Japan, nemlig henholdsvis EGNOS og MSAS systemerne, hvor EGNOS fungerer som en forløber for Galileo, og siden hen skal bruges til at forbedre præcisionen af

Galileo.

Den teknik der benyttes af landmålere og andre der kræver ekstrem høj præcision kaldes Real-Time Kinematic positioning (RTK), og kan give en præcision på 1 cm. RTK er en videreudvikling af DGPS, og kræver udstyr af en sådan størrelse og pris at det for nuværende kun bruges til professionelle formål [41].

For samtlige af de ovenstående GPS teknikker gælder at de kun kan benyttes hvis satellitternes signaler kan nå uhindret frem til modtageren. Hvis signalerne skal passerer gennem bygninger vil det signal der når frem til modtageren være så svagt at det ikke er muligt for modtageren at opfange signalet. Dette betyder at de ikke kan benyttes indendørs eller mellem høje bygninger. Derved er de ikke tilgængelige der hvor størstedelen af befolkningen tilbringer størstedelen af tiden. Problemet med de svage signaler er, at for at opfange signaler skal modtageren foretage søgninger af en række frekvenser og ved hver frekvens "dvæle"et kort øjeblik for at modtage et evt. signal på frekvensen. For at opfange svage signaler skal modtageren "dvæle"længere ved hver frekvens, hvorved det ikke er muligt at gennemsøge nok frekvenser. Hvis modtageren viste hvilke frekvenser den skulle søge på ville det være muligt at opfange meget svage signaler. Kendskab til hvilke frekvenser der kan findes signaler på afhænger af satellitternes positioner.

Løsningen på problemet hedder Assisted GPS (AGPS). AGPS benytter en forbindelse til et mobilt netværk, f.eks GPRS, til at udveksle data med en assistent server der modtager positionsdata fra satellitterne og udregner deres position. Modtageren behøver derfor ikke gøre dette. Assistent serveren kan også benytte viden om hvilken celle modtageren befinder sig i, dvs. hvilken antenne der benyttes, samt hvilke satellitter der på et given tidspunkt er synlige fra den celle modtageren befinder sig i. Denne assistance gør at modtageren kan "dvæle"længe nok til at kunne opfange svage signaler og derved kan den angive en position selvom den befinder sig indendørs, se 2.3.



Figur 2.3: Oversigt over AGPS

Udover at kunne fange svage signaler giver AGPS også en væsentlig hurtigere Time-To-First-Fix (TTFF), dvs. den tid det tager fra modtageren tændes til den

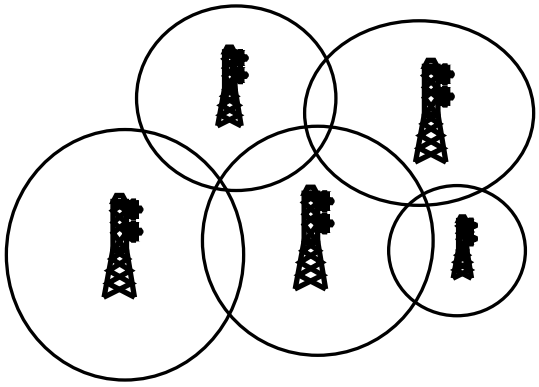
kan angive en position. TTFF kan med normal GPS tage adskillige minutter, mens det med AGPS kan gøres på under et minut og helt ned til få sekunder, alt efter hvor modtageren befinder sig. Præcisionen af AGPS, samt hvorvidt det kan bruges indendørs afhænger af modtagerens evne til at opfange signaler. Nye AGPS chipset har en præcision på ca. 5 meter [31], men ikke alle kan benyttes indendørs. Faktisk var det først i 2004 at de første AGPS chipset, der kan bruges indendørs, blev introduceret [14, 18, 30]. De AGPS løsninger der er på markedet i øjeblikket, som f.eks. Motorolas serie af mobiltelefoner med AGPS, kan ikke bruges indendørs og ifølge reviews er der problemer med præcisionen [27, 28]. Desuden koster det i Danmark 3 kroner pr. lokalisering. Siemens og Nokia har først AGPS løsninger klar i tredje kvartal af 2005 [34, 42].

2.1.2 Globale System for Mobile Communications

Den mest benyttede form for trådløs kommunikation til mobiltelefoner er Globale System for Mobile Communications (GSM), som i 2005 benyttes af over en milliard mennesker [50]. GSM sender både data og tale digitalt, og benytter sig af Circuit Switched Data. GSM betegnes som et 2g mobiltelefon system, idet analoge systemer betegnes 1g. I 1997 blev der tilføjet Packet Switched Data, i form af GPRS standarden, der har en teoretisk hastighed på 170 kbit/s, og betegnes 2.5g. Men hastigheden afhænger af hvor benyttet netværket er, og en realistisk hastighed er på 20-70 kbit/s. I 1999 blev EDGE og UMTS standarderne tilføjet, og de betegnes som henholdsvis 2.75g og 3g. De giver begge mulighed for større hastighed på datatransmission. EDGE bygger ovenpå GPRS, og har en teoretisk hastighed på 384 kbit/s, og en reel hastighed på cirka 140 kbit/s. I Europa er EDGE ikke ret udbredt da de fleste mobile operatører har valgt at gå fra GPRS til UMTS. Men i USA, hvor CDMA standarden er udbredt, bruges EDGE som en konkurrent til CDMA2000. UMTS har en teoretisk hastighed på 1920 kbit/s, og benyttes i nuværende UMTS netværk med en hastighed på cirka 384 kbit/s. UMTS er implementeret i en række europæiske lande, og i en række storbyer i USA. Efterfølgeren til UMTS er HSDPA med en forventet hastighed på 8-10 Mbit/s.

Et GSM netværk er opdelt i celler, se figur 2.4, med en antenne i hver celle. Cellerne kan være overlappende, og af varierende størrelse. Det er med udgangspunkt i denne celleopdeling at GSM kan benyttes til positionsbestemmelse. Der findes forskellige metoder til lokationsbestemmelse i et GSM netværk. Den enkleste metode benytter celle id til at afgøre hvilken celle et opkald kommer fra. Andre metoder kan benyttes hvis en mobiltelefon er inden for rækkevidde af flere antenner. Det at cellerne varierer i størrelse betyder at præcisionen ligeledes varierer. I bymæssig bebyggelse, hvor cellerne er mindst kan der, hvis der er tre antenner inden for rækkevidde, opnås en præcision på 50 meter. I landområder, med store celler er præcisionen omkring 2-4 kilometer. GSM kan benyttes til positionsbestemmelse

i forbindelse med nødopkald. I USA er det i henhold til E911 direktivet lovpligtig at mobile operatører kan oplyse inden for hvilken celle et kald kommer fra. Fase 2 af E911, der gælder fra 31 december 2005, gør det lovpligtig at kunne bestemme et opkalds position med en præcision på 50-200 meter [11]. Et lignende system, kaldet E112, er ved at blive indført i EU og ifølge en tilstandsrapport om implementeringen af E112 [40] var der pr. 31 januar 2005 11 EU lande som havde implementeret positionsbestemmelse i deres GSM netværk. Danmark menes at have fuldført implementeringen i løbet af 2005. For at udvikle systemer baseret på GSM positionering skal man have tilladelse af en eller flere mobile operatører til at benytte deres netværks positioneringsteknologi. I England er det f.eks. mulig for virksomheder, at ansøge om tilladelse til at benytte Oranges Location API i deres applikationer [36].



Figur 2.4: GSM netværk opdelt i celler

2.1.3 Wireless Fidelity

En anden form for trådløs forbindelse med stor udbredelse er Wireless fidelity (wifi), der er en betegnelse for IEEE 802.11 standarden for trådløse lokal netværk. Wifi gør det muligt at få netværksforbindelse via et Access Point (AP), ved hjælp af et wifi kort. Det benyttes til at lave trådløse lokal netværk hos både firmaer og private, hvilket har den fordel, at man slipper for ledninger, idet der blot tilføjes yderligere hotspots, frem for at sætte switches og hubs op. Det benyttes også på offentlige steder, som lufthavne og restauranter, til at give internet adgang til besøgende. Den nyeste version af 802.11, nemlig 802.11g har en maksimal hastighed på 54 Mbps [2]. Signalet fra standard APer kan opfanges indenfor en radius af ca. 100 meter [51]. Afstanden er dog stærkt afhængig af om der står noget i vejen, f.eks. mure.

Wifi kan bruges til positionsbestemmelse, idet en modtager for at modtage et signal fra et bestemt AP, må være inden for en vis afstand til det pågældende AP.

Derudover kan den styrke signalet har når det bliver modtaget benyttes til yderligere at præcisere positionen. Ved at benytte signalerne fra flere APer, og knytte sæt af signalstyrke målinger til en bestemt position, angivet f.eks. ved længde og breddegrader eller ved punkter på et kort, kan der opnås en meget høj præcision. F.eks tilbyder Ekahau et produkt der knytter punkter på kort til et sådant sæt, som kan give en præcision på 1 meter [9]. Det kræver dog en omfattende kalibrering af systemet, idet området som systemet skal dække, skal kortlægges før systemet kan bruges. Derudover går det udover præcisionen hvis der bliver flyttet rundt på større ting, som f.eks. møbler og skillevægge. Det betyder at systemet er velegnet til enkelte bygninger men ikke til hele byer eller større landområder. Desuden findes der et stykke open-source software ved navn Place Lab som kræver mindre infrastruktur og mindre konfigurering på bekostning af præcision. Place Lab benytter sig af forskellige typer radio beacons, bl.a. wifi APs, til at udregne positioner. Place Lab vil blive beskrevet nærmere i afsnit 3.1.

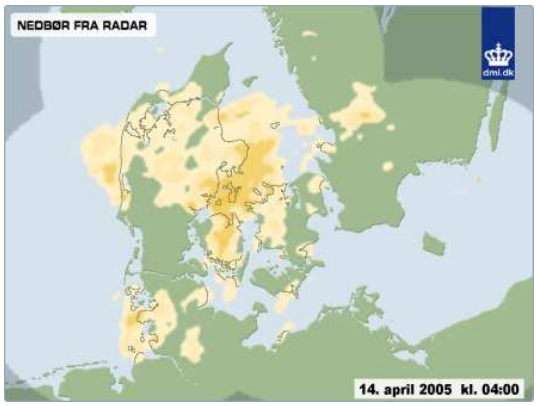
2.1.4 Konklusion

Alt hvad der har en kendt position og udsender eller reflekterer et signal kan bruges til positionsbestemmelse. Af teknologier ikke beskrevet i dette afsnit kan f.eks. nævnes Bluetooth, der minder om wifi, men med kortere rækkevidde, som bliver benyttet i bl.a. Blip Systems BlipNet [6], infrarøde signaler, der bl.a. bliver benyttet i Active Badge systemet [3], samt real-time analyse af videobilleder [20]. Hvilken teknik der er mest velegnet afhænger af hvilket formål positionsbestemmelsen har. Til vejnavigation benyttes i øjeblikket GPS, mens der i bygninger benyttes wifi, bluetooth eller infrarød baserede løsninger. GSM benyttes som nævnt i forbindelse med nødopkald og i kombination med GPS i forbindelse med AGPS. Netop AGPS vil højst sandsynlig få en stor udbredelse i fremtiden, efterhånden som AGPS chipset bliver standard i mobiltelefoner, på samme måde som kameraer, og AGPS løsninger bliver implementeret af mobile operatører. Det at AGPS muliggør hurtig TTFF, samt brug både indendørs og udendørs, gør at LBSer til apparater, der både har adgang til GSM samt har AGPS chipset, kan få stor kommerciel succes. I afsnit 3.1 om Place Lab, vil der blive gjort yderligere overvejelser om, hvorvidt wifi positionering har en fremtid, herunder specielt hvilken rolle Place Lab kan komme til at spille i den forbindelse.

2.2 Geografisk Informationssystem

Et geografisk informationssystem (GIS) er et system til at håndtere spatial data med associerede attributter. Et GIS computer system består typisk af en grafik fil tilknyttet en attribut database. Hovedideen med et GIS system er at kunne knytte

attributter til positioner, der angives vha. en geokode. En geokode er en geografisk kode til at identificere et punkt eller område på jorden. Et eksempel på geokoder er postadresser, der identificer et bestemt geografisk område. Til mere præcis positionsbestemmelse benyttes det geografiske koordinatsystem med længde og breddegrader. Processen med at knytte geokoder til spatial data kaldes geokodning. Et meget velkendt eksempel på GIS er et vejrkort, hvor en mængde målinger af bl.a. vind og regn associeres med punkter på et kort. Se figur 2.5.



Figur 2.5: Eksempel på GIS [8]

De fleste LBS har behov for et GIS, til at kunne relaterer fundne positioner til punkter eller områder på et kort. GISer tilbydes af en række virksomheder og organisationer, f.eks af Kort og Matrikel Styrelsen (KMS) i danmark. KMS har en tjenste ved navn "Kortforsyningen" [22], som tilbyder kort der benytter adresser, matrikelnumre og stednavne som geokoder. Det er ikke umiddelbart muligt at få adgang til "Kortforsyningen", istedet har KMS lavet en aftale med en række virksomheder om at udvikle løsninger baseret på "Kortforsyningen". ESRI [10] tilbyder en række web services ved navn ArcWeb services [4], som udviklere kan integrere i egne systemer. ArcWeb services gør det muligt at benytte en række forskellige kort, bla. kort med real-time updateret trafikinformation, samt mulighed for at få længde og breddegrad udfra en adresse, plus en række andre features. Et lignende produkt tilbydes af Microsoft i form af MapPoint Web services [25], der ligeledes har en række features der er nyttige i forbindelse med et LBS.

Kapitel 3

Programmeringsteknologi

Indholdsfortegnelse

Dette kapitel beskriver den programmeringsteknologi der benyttes i forbindelse med udviklingen af systemet.

3.1 Place Lab

Place Lab er et stykke open-source software udviklet af Intel Research i Seattle med det formål at understøtte udviklingen af LBSer der benytter radio beacons til positionsbestemmelse. De typer af radio beacons der understøttes er GSM antenner, Bluetooth APs og wifi APs. Place Lab er blevet udviklet med fokus på følgende.

A: Maksimer dækning målt i procent af den tid det er muligt at få en positionsbestemmelse i folks daglige liv.

B: Tilbyde en lav barrier for brugere og udviklere.

Udover de to ovenstående punkter lægges der også vægt på at privatlivets fred bevares, dvs. at der ikke er nogen myndighed eller virksomhed der får oplysninger

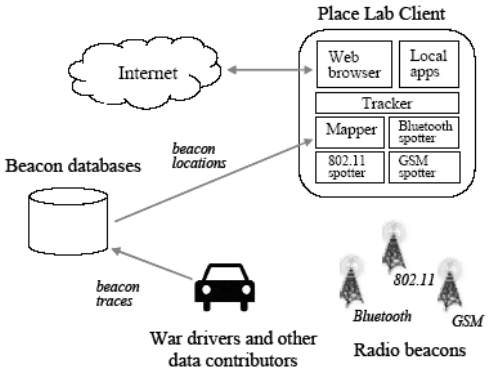
MAC	signal styrke	længdegrad	breddegrad	tid	type
00:80:c8:19:5a:1d	50	57.125	9.456	2000	Wifi
00:80:c8:19:5a:1d	51	57.126	9.456	2001	Wifi

Tabel 3.1: Eksempel på traces indsamlet ved war driving

om positioner. Foruden at gøre det muligt for udviklere at udvikle LBSer, er det overordnede mål med Place Lab, at opbygge et globalt lokaliseringssystem, i stil med GPS, men med radio beacons i stedet for satellitter. Det kræver at der bliver indsamlet positionsdata af en masse radio beacons. Det drejer sig især om wifi APs, idet adgangen til GSM antenner er begrænset og der er langt mellem dem, og Bluetooth APs har en meget begrænset rækkevidde og udbreddelse. Det er op til brugerne af Place Lab at opbygge en database af positionsbestemte radio beacons. Dette gøres ved at køre rundt og indsamle oplysninger om radio beacons, også kaldet war driving, og sende de indsamlede oplysninger til Intel Research i Seattle, hvor den mest præcise position for hver indsamlet radio beacon udregnes. Denne position, sammen med oplysninger om radio beaconen, indsættes i en database, hvis indhold kan downloades af alle. De oplysninger der indsamles i forbindelse med war driving er MAC adresse, signal styrke, længde og breddegrad fundet via GPS, samt tid og type af de fundne radio beacons. Tabel 3.1 viser et par linier af resultatet af war driving. Hver linie kaldes også et "trace", og disse traces kan bruges til at finde en tilnærmelsesvis korrekt position for en beacon.

Den del af Place Lab som udviklere kan downloade består af et Java API og native spotters for hver af de understøttede platforme. De platforme der understøttes er Windows, MAC OS, Linux, Windows CE/Pocket PC og Nokia serie 60 mobiltelefoner. En spotter er en driver til at modtage data fra enten wifi kort, Bluetooth eller GSM antenner. Udover spotters er de centrale dele af Java APIet mapper og tracker. Mapper bruges til at tilgå en lokal database af positionsbestemte beacons. Den lokale database kan fyldes op med beacons downloaded fra Intel Reasearch, eller man kan selv tilføje beacons til den. Den information der gemmes i databasen for hver beacon består som minimum af MAC adresse og længde og breddegrader. En tracker bruges til at koordinerer den information, som spotters indsamler, med de radio beacons der er gemt i en lokal database, og som tilgås via en mapper, for at afgøre brugerens position. Java APIet indeholder et par eksempler på trackers, f.eks. CentroidTracker der udregner en midterposition udfra de beacons der spottes. Figur 3.1 giver et samlet overblik over de forskellige komponenter der indgår i Place Lab.

Fordelen ved Place Lab, i forhold til GPS er, at brugere ikke behøves at investere i GPS modtagere og at det kan bruges indendørs. Hvis vi ser et par år frem i tiden, hvor AGPS har gjort det muligt at benytte GPS indendørs, så vil Place Lab stadig have den fordel for udviklere, at det er open-source. Hvorvidt mobile operatører



Figur 3.1: Oversigt over Place Lab [37]

vil gøre AGPS APIer offentlig tilgængelige er endnu uvis. Derudover er det gratis for en Place Lab bruger at få en positionsbestemmelse, sammenholdt med at det i dag koster penge pr. gang en mobiltelefons AGPS software benyttes. Men indendørs AGPS er endnu ikke kommercielt tilgængelig, og indtil det sker er Place Lab et billigt alternativ til f.eks. Ekahau og BlipNet, i forbindelse med indendørs positionsbestemmelse. Ulempen ved Place Lab ved brug i mere ukontrollerede miljøer er, at der kun kan fastlægges en position hvis der er kortlagde beacons i nærheden, og præcisionen afhænger af hvor mange det drejer sig om. I centrum af større byer er der masser af wifi beacons, og i en test udført af Intel Research i downtown Seattle opnåedes en præcision på 20.5 meter [37]. Dog var der gået to timers war driving forud for denne test. Men i områder hvor hverken war driving eller kendskabet til Place Lab er udbredt, vil et Place Lab baseret system være ubrugelig. Så det er yderst tvivlsomt om en LBS baseret på Place Lab vil kunne dække hele lande.

Der hvor Place Lab vil være nyttig, er for systemer der kun skal dække et begrænset område, f.eks centrum af en by, et universitets område eller en virksomhed. I et by centrum vil det være muligt at gennemføre en grundig war driving, evt. en gang om året, idet mængden af beacons hele tiden ændrer sig. På et universitetsområde og i virksomheder vil der ofte i forvejen være opsat APs, hvilket gør at der ikke skal investeres i ny infrastruktur for at understøtte et Place Lab baseret LBS. Et LBS til universitetsområder og virksomheder vil som oftest ikke have behov for høj præcision, idet det vil være nok at kunne positionsbestemme en person til et bestemt lokale, og ikke hvor i lokalet personen befinder sig. En sådan LBS vil kunne bruges til at undgå at man går forgæves hen til en persons kontor, eller til at finde den pågældende person, selv om vedkommende ikke befinder sig på sit kontor. En LBS til et by centrum vil kunne benyttes af turister til at få vist et kort over den del af byen de befinder sig i, samt f.eks. hvilke restauranter og seværdigheder der er i nærheden.

Men her opstår det problem, at det er de færreste turister der går rundt med et apparat med et wifi kort. Det apparat som langt de fleste går rundt med, nemlig en mobiltelefon, har ikke wifi kort som standard udstyr. Et lignende problem er der i forbindelse med brug af Place Lab baserede LBSer til både universitetsområder og virksomheder. Dog vil det her være muligt at forsyne de ansatte med en PDA med wifi kort. Derudover har mange en laptop, der enten har, eller kan forsynes med et wifi kort. Dette kan dog ændre sig i løbet af nogle år, idet der allerede nu findes high-end mobiltelefoner/PDAer med både wifi kort og GSM [33]. Det er dog yderst tvivlsomt om wifi kort i mobiltelefoner nogensinde vil blive benyttet i forbindelse med positionering. Umiddelbart ser det ud til at AGPS bliver fremtidens positioneringsteknologi, både til udendørs og indendørs brug. Da det vil være tilgængelig overalt på jorden, ikke kræver at der opbygges særskilte infrastrukturer i form af radio beacons, og giver en præcision der er høj nok til langt de fleste LBSer.

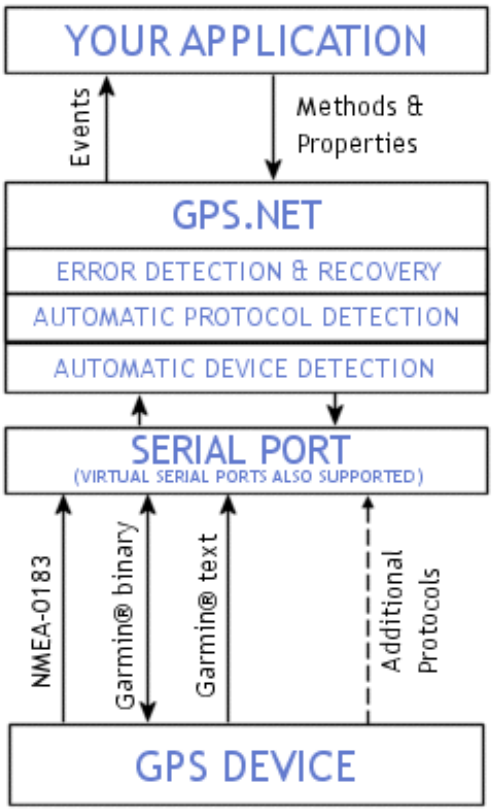
3.2 GPS.NET

GPS.NET er et API til .NET frameworket til at udvikle systemer der benytter en GPS modtager. GPS.NET gør, at en udvikler ikke behøves, at have kendskab til GPS protokoller, som f.eks. NMEA-0183, og indeholder desuden en række nyttige funktioner til at håndtere GPS data. GPS.NET sørger for at modtage real-time information fra en GPS modtager, omdanne og indsætte denne information i sin objekt model og gøre den tilgængelig, i form af events, til en applikation. Informationen fra GPS modtageren består, udover længde og breddegrad, bl.a. af hastighed og retning, samt satellit information, f.eks. hvor mange og hvilke satellitter der er et fix på. Figur 3.2 giver et samlet overblik over GPS.NET.

Det er nemt at benytte GPS.NET, først oprettes et objekt der repræsenterer GPS modtageren. Derefter "hookes"der op på de ønskede events, og tilsidst køres start metoden på GPS objektet. Derudover kan der vælges mellem automatisk "opdagelse"af GPS modtageren eller at udvikleren selv angiver serial port og baud rate. Udover at modtage information fra GPS modtageren, kan GPS.NET også omregne mellem længde og breddegrads koordinater og UTM koordinater. I GPS.NET giver UTM koordinater mulighed for at angive en position med metriske værdier, samt udregne afstand mellem to positioner.

GPS.NET kan bruges til .NET og .NET CF, herunder både SmartPhone og Pocket PC. Desuden understøttes alle GPS modtagere der benytter NMEA-0183 protokollen. Det er ikke gratis at benytte GPS.NET, og prisen afhænger af antallet af licenser man ønsker at købe. Én licens koster 153$ og licens til en hel organisation koster 2000$ [19]. Det er derudover muligt at downloade en gratis trial version.

For at få GPS.NET til at virke på et apparat med et GPS system installeret, som



Figur 3.2: Oversigt over GPS.NET [46]

f.eks. TomTom Navigator, er det nødvendig at oprette en virtuel port, der modtager den samme GPS information som den port det pågældende system benytter. Til det formål har Franson lavet et produkt ved navn GpsGate [12], der kan gøre lige præcis dette. GpsGate koster 15$ for en licens der kan benyttes på op til fire apparater [13].

3.3 Web services

Web services er en object-orienteret XML-baseret middlewareteknologi til kommunikation mellem heterogene systemer. Det betyder at web services muliggør kommunikation mellem applikationer skrevet i forskellige sprog og til forskellige platforme. Web services udgøres af tre XML baserede standardiserede teknologier, Web Service Description Language (WSDL), Simple Object Access Protocol (SOAP) og Universal Discovery Description Language (UDDI).

WSDL er et XML format der bruges til at beskrive en web service. Et WSDL dokument beskriver hvor en web service er tilgængelig, hvilke interfaces web servicen stiller til rådighed, og indeholder derudover information om, hvordan man bruger web servicen, herunder hvilke kommunikations-protokoller og besked-formater web servicen benytter. Et WSDL dokument leverer, via dets XML format, en entydig, struktureret information, der kan bruges som opskrift til at automatisere detaljerne omkring kommunikation mellem en web service og en web service klient. Der findes en række værktøjer der automatisk kan generere et WSDL dokument udfra applikations-interfaces. Ligeledes findes der værktøjer der automatisk kan generere den nødvendige kode til at kommunikere med en web service udfra et WSDL dokument.

SOAP er en XML baseret kommunikationsprotokol designet til at lade applikationer i forskellige sprog og på forskellige platforme kommunikere. SOAP kan bruges med enhver transportprotokol (HTTP, TCP, SMTP) og kan beskrive enhver datastruktur. Til at generere og parse SOAP beskeder findes der en række værktøjer til forskellige platforme og sprog.

UDDI er en specifikation der definerer hvordan man offentliggør og opdager information om web services. UDDI kan betragtes som en annonce for en web service, der hjælper potentielle brugere til at finde og læse detaljer om web servicen. En UDDI beskrivelse af en web service kan placeres et centralt sted, f.eks. Microsoft UDDI Business Registry Node [26].

3.4 MapPoint Web services

MapPoint Web services (MPWS) er en række web services udbudt af Microsoft, der gør det muligt for udviklere at integrere et GIS i deres systemer. MPWS er en del af Microsoft MapPoint systemet [23], der foruden MPWS består af MapPoint Location Server og en stand-alone applikation, hvis nyeste version hedder MapPoint 2004.

MapPoint Location Server er beregnet til virksomheder der ønsker at kombinere positionsdata, leveret af mobile operatører, med MPWS. Det kan f.eks. være fragtfirmaer der ønsker at have et real-time opdateret geografisk overblik over hvor deres lastbiler befinder sig.

MapPoint 2004 er en applikation der indeholder et geokodet landkort over den del af jorden som den pågældende version dækker. Der er bl.a. mulighed for at scrolle rundt på kortet, finde en rute mellem to destinationer, samt søge efter f.eks. byer og adresser, samt positioner defineret ved længde og breddegradskoordinater.

MPWS udgøres af følgende web services.

Common Service Indeholder hjælpe metoder, f.eks. en metode der returnerer hvilken version af MPWS der benyttes og en metode der returnerer et lande navn udfra et lande id input.

Find Service Indeholder metoder til at finde forskellige ting. f.eks. en metode til at finde en adresse position udfra en adresse, eller til at finde en liste af nærliggende adresser udfra en position.

Render Service Indeholder metoder der kan rendere kort udfra en beskrivelse af hvilket område kortet skal dække og hvilke POIs der skal sættes på kortet. POIs kan være brugerdefinerede eller komme fra den database med over 15 millioner POIs der er en del af MPWS.

Route Service Indeholder metoder til at udregne ruter mellem to eller flere punkter.

Som det ses af ovenstående liste så indeholder MPWS features der gør, at det vil være velegnet som GIS for en række forskellige LBSer. Den centrale web service i forbindelse med systemet er *Render service*, der gør det muligt at få et kort over det område en bruger befinder sig i, og få påsat systemets egne POIs på det pågældende kort.

3.5 .NET Remoting

.NET remoting er en middelware teknologi der muliggør kommunikation mellem applikationsprocesser og mellem applikationer på forskellige maskiner over et netværk. Det understøtter, ligesom web services, applikationsdistribution og integration. Måden det gøres på er ved at have et remote objekt, der tilbydes af en remote server, og som kan tilgås af en eller flere klient applikationer.

Remote objektet bruges til at indeholde fælles metoder. Den klasse som remote objektet er en instans af skal arve fra klassen **MarshalByRefObject**, og hvis returværdier til remote metoder selv er klasser, så skal disse klasser have attributten **[Serializable]** ovenover klassehovedet. Listing 3.1 illustrerer hvordan et remote objekt ser ud, og hvordan en klasse der benyttes som returværdi i et remote kald skal se ud.

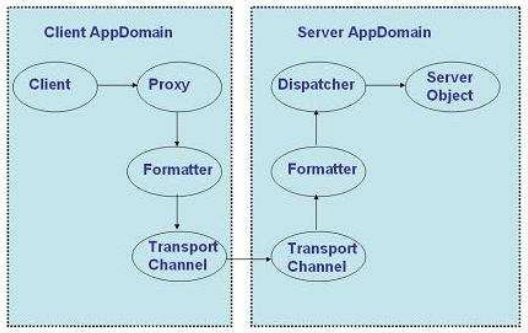
```
public class RemoteObject : MarshalByRefObject
{
    public RemoteObject(){}
    public User getUser(int id)
    {
        //code to get user
        return user;
    }
```

```
}
[Serializable]
public class User
{
    //code to class user
}
```

Listing 3.1: Eksempel på remote klasse

Remote serveren angiver hvilken port der kommunikeres på, hvilken transport protokol der kommunikeres over, og hvilken metode der bruges til at formatere kaldene og de tilhørende returværdier. Der kan vælges mellem to sæt bestående af en transportprotokol og en formateringsmetode. Enten HTTP protokollen og SOAP formatering eller TCP protokollen og binær formatering. TCP/binær giver den hurtigste kommunikation mens HTTP/SOAP gør det muligt at placere en remote server på en Internet Information server (IIS), med mulighed for at udnytte IISs forskellige features, f.eks. authentication, som man som udvikler ellers selv skal implementere.

Klient applikationer kalder ikke metoder direkte på remote objektet, men kalder istedet metoder på et proxy objekt i det lokale applikationsdomæne. Proxy objektet sørger så for at formatere kommunikationen med remote objektet, i henhold til den valgte formateringsmetode. Se figur 3.3 for et overblik over .NET remotings arkitektur.



Figur 3.3: .NET remoting arkitektur [32]

Som nævnt tjener web services og .NET remoting de samme formål. Hvilken af de to der vælges afhænger af i hvilken sammenhæng de skal bruges. De vigtigste egenskaber til at afgøre hvilken der vælges er følgende. Web services kan bruges på heterogene platforme, mens .NET remoting kun kan bruges på .NET platformen. Hverken .NET remoting eller web services er, med undtagelse af web service klienter, understøttet af .NET CF. Derudover er .NET remoting hurtigere end web services, hvis der benyttes TCP/binær.

I systemet benyttes .NET remoting til kommunikation mellem web servicen og ser-

veren. Der benyttes TCP/binær for at gøre kommunikationen hurtig, hvilket også medfører at remote serveren ikke kan placeres på IIS, som en del af web servicen, men derimod udgør en særskilt process. Systemet benytter remote objektet til, via delegates, at kalde metoder på serveren på vegne af web servicen. Delegates kan betragtes som metode pointerer, der gør det muligt at sende en reference til en metode som parameter til et metode kald. Derved kan remote objektet via delegates kalde metoder på serveren.

Adskillelsen af web servicen og remote objektet benyttes i forbindelse med distribuering af systemet, idet flere instanser af web servicen via remote objektet, kan tilgå den samme instans af serveren, og én instans af web servicen kan ligeledes tilgå flere server instanser. Dette vil blive beskrevet nærmere i afsnit 5.4.

3.6 Cω

Cω [7] (C omega) er en eksperimentel udvidelse af C#, med det formål at slå bro mellem semi-struktureret hierarkisk data (XML), relationel data (SQL) og .NET Common Type System (CTS), samt gøre det lettere at håndtere samtidighed. I dette afsnit vil der blive fokuseret på Cωs måde at håndtere relationel data på, da det er denne funktionalitet der benyttes i systemet.

Cω tilføjer fire nye typer til C#, nemlig *streams*, *anonyme structs*, *choice types* og *content classes*. I forbindelse med håndteringen af relationel data benyttes streams og anonyme structs. Streams er homogene samlinger af ens typer, og minder om arrays, med den forskel at der ikke på forhånd kan angives en størrelse. Det er desuden muligt at søge i en stream ved at bruge XPath syntaks, hvilket benyttes til at slå bro mellem XML og CTS. Anonyme structs er structs hvor det er muligt at bruge unavngivne attributter og bruge det samme attributnavn flere gange. Desuden er det ikke nødvendig at angive type af hverken anonyme structs eller streams variabler. Hvis en variable bliver assignet til en anonym struct eller en stream, får variablen automatisk den assignede type. Listing 3.2 viser hvordan streams og anonyme structs bruges.

```
int* x = new{1,2,3};
public static struct{int x;int;int x}* getStructs(){
    for(int i=0;i<4;i++){
        yield return new{x=4,6,i};
    }
}
y = getStructs();
w = y.x;
struct{int x;int;int x;int;} c = new{x=4,5,6};
int d = c[1];
e = c.x;
```

```
12  f = c.int::*;
```

Listing 3.2: Eksempel på streams og ananyme structs

Linie 1 Variablen *x* er et eksempel på en int stream, indeholdende værdierne 1,2,3.

Line 2-6 Metoden *getStructs* returnerer en stream af anonyme structs.

Linie 7 Variablen *y* tildeles returværdien af metoden *getStructs*. Dette gøres uden at angive typen på *y*.

Linie 8 Variablen *w* tildeles værdien af *c.x*. Det betyder at den bliver en int stream indeholdene følgende talrække 4,0,4,1,4,2,4,3.

Linie 9 Det er muligt at erklære typen på en variabel til at være en anonym struct. Variablen *c* er et eksempel herpå. Erklæringen viser også at det er muligt at bruge det samme attribut navn flere gange, idet x benyttes som attributnavn to gange.

Linie 10-12 De resterende linier viser de tre forskellige måder man kan tilgå indholdet af en anonym struct på. Det kan gøres ved at angive placeringen på den attribut man ønsker at tilgå, man kan angive navnet på attributten, og i tilfælde af at der er flere attributter med det angivne navn så er resultatet en stream, og endelig er der mulighed for at tilgå alle attributter af en bestemt type, i dette tilfælde alle attributter af typen int, hvilket vil sige samtlige attributter i den anonyme struct.

Måden anonyme structs og streams bliver brugt i forbindelse med database kald er, at resultatet af et kald er en stream af anonyme structs. Database kald i Cω har samme syntaks som SQL kald, den væsentligste forskel er at syntaksen i Cω tjekkes ved compile-time, hvor det med C#s database komponenter først tjekkes ved runtime. For at compile-time tjek er muligt er det nødvendig at Cω har kendskab til den benyttede databasens opbygning. Det gøres ved at Cω, i Visual Studio 2003, har tilføjet et menupunkt under *references* i solution explorer, der hedder "add database schema". Dette menupunkt åbner en wizard hvor man kan vælge den Microsoft SQL Server database man ønsker, og derefter bygge et database objekt udfra den valgte database. Derudover sørger wizarden selv for at opbygge connect-strengen til den valgte database. Efter at wizarden er færdig, har man et objekt, repræsenterende databasen. Dette objekt kan benyttes som alle andre objekter i C#, hvilket betyder at man kan bruge prik-notation til at tilgå objektets attributter, der i forbindelse med database objektet repræsenterer tabeller.

Resultatet af et database kald udført på database objektet er som nævnt en stream af anonyme structs, hvor hver anonym struct repræsenterer en række, og hvor navnene på de anonyme structs attributter svarer til de kolonne navne der blev angivet,

i database kaldet, til at skulle indgå i resultatet. Efter kaldet skal variablen, der repræsenterer resultatet, bruges i en *foreach* løkke, hvor de anonyme structs bliver gennemløbet. Listing 3.3 giver et eksempel på hvordan Cω bruges til database kald.

```
res = select * from datdb.users;
User user;
 foreach(row in res){
    user = new User((int)row.x,(int)row.y,(int)row.uid,(int)row.utype,
    (string)row.navn,(string)row.password,(string)row.nearby));
 }
```

Listing 3.3: Eksempel på database kald i Cω

Linie 1 *datdb* er det objekt der repræsenterer databasen, og som det ses benyttes prik-notation til at tilgå dets attributter, der her repræsenterer tabeller. Det viste kald returnerer alle kolonner i tabellen *users* og assigner resultatet til variablen *res*.

Linie 2-5 *foreach* løkken gennemløber alle rækker i resultatet, hvor hver række er repræsenteret ved en anonym struct. Som det ses i linie 3 og 4 så benyttes også prik-notation til at tilgå attributterne i de anonyme structs, der her repræsenterer kolonner. Da værdierne af kolonnerne er **SqlTypes** skal de castes til C# typer. I dette eksempel oprettes et **User** objekt udfra data hentet i databasen.

Den største fordel ved at bruge Cω fremfor C#s database komponenter er, at syntaks fejl bliver fundet ved compile-time, og i visual studio 2003 bliver de fundet allerede mens man skriver, da Visual Studio 2003 har løbende syntaks tjek. En anden fordel er, at det ikke er nødvendig, at kunne huske præcist hvordan tabeller og kolonne navne staves, da de automatisk bliver vist ved brug af prik-notationen. Desuden slipper man for at holde styr på, om man har husket at køre *open* eller *close* metoderne på de objekter, der benyttes i forbindelse med database håndtering. Samlet betyder det at Cω gør, at man kan kode database tilgang hurtigere, og samtidig skrive mere fejlfri kode.

Del II

Udvikling af system

Kapitel 4

Analyse og arkitektur

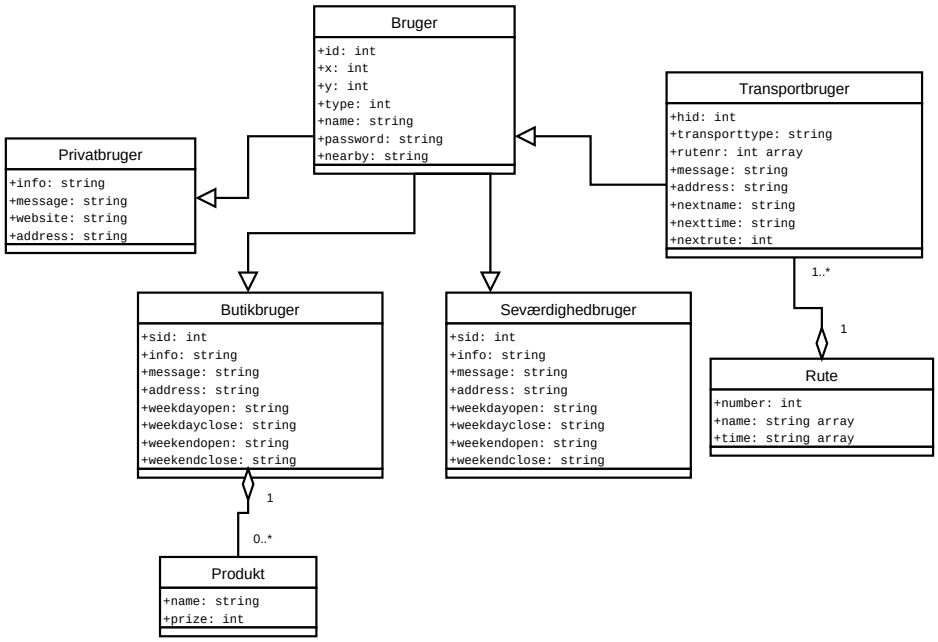
Indholdsfortegnelse

4.1 Analyse

Formålet med analysefasen er, med udgangspunkt i kravsspecifikationen, at fastlægge systemets ekstern observerbare adfærd. Den observerbare adfærd beskrives ved hjælp af en række klasser og hændelser. En klasse er defineret som "En beskrivelse af en til flere objekter med samme struktur, adfærdsmønster og attributter"og en hændelse er defineret som "Et øjebliks begivenhed som involver et eller flere objekter".

Udfra kravsspecifikationen er der blevet identificeret syv klasser, nemlig **Bruger**, **Butikbruger**, **Seværdighedbruger**, **Transportbruger**, **Privatbruger**, **Produkt** og **Rute**. De fire klasser der repræsenterer brugertyper er alle specialiseringer af klassen **Bruger**. Transportbrugere er knyttet til en eller flere ruter, og der kan være nul til mange produkter tilknyttet en butiksbruger. Bruger klassen er som nævnt en generalisering af de fire brugertyper, og indeholder den data der er placeret i main-memory. Denne data indgår i alle brugertyper, og benyttes i forbindelse med positionsopdatering, samt en række område forespørgsler. Det er instanser af klassen **Bruger** der indekseres i datastrukturen, mens der kun oprettes instanser af de andre klasser når data skal overføres fra databasen til klienten. Generaliseringen medfører desuden, at der kan benyttes polymorfi til at lave metoder der er generelle for alle brugertyper. F.eks. en metode *hentBruger(int id)* der med et bruger

id som input kan returnere alle fire brugertyper, der så efterfølgende kan castes til den konkrete type. Sammenhængen mellem klasserne kan ses i klassediagrammet i figur 4.1.



Figur 4.1: Klassediagram

Der er ligeledes, med udgangspunkt i kravsspecifikationen, blevet identificeret en række hændelser, der understøtter systemets ønskede funktionalitet. Nedenstående liste beskriver disse hændelser.

Positionsopdatering Klienten skal løbende sende oplysninger om sin position til serveren.

Vis kort Der er to måder, hvorpå brugeren kan vælge at få vist et kort. Den første måde er tilgængelig efter, at brugeren har fået en positionsbestemmelse, enten via GPS eller wifi. Den finder alle brugere inden for en brugerbestemt radius og repræsenterer dem med små ikoner på et kort, samt i en liste. Den anden måde er i forbindelse med forespørgsler, her er det muligt at få vist et kort med brugeren og den bruger der er resultatet af forespørgslen. Ved forespørgslen "Find produkt", hvor resultatet er en liste af butikker der sælger det angivne produkt, skal der vælges en af butikkerne i listen før der kan vises et kort med brugeren og den valgte butik.

Indstille filter Filtret lader brugeren bestemme hvilken type brugere der vises på kortet over det område brugeren befinder sig i. De fire typer af brugere kan vælges til og fra og derudover kan der vælges om det kun er åbne butikker

og seværdigheder der skal vises, samt hvilke type transportbrugere der skal vises.

Starte og stoppe GPS Der kan forsøges at etablere forbindelse til GPS modtageren, og forbindelsen kan afbrydes.

Starte og stoppe wifi Der kan vælges at modtage wifi information fra Place Lab klienten, og forbindelsen til Place Lab klienten kan afbrydes.

Find privatbruger Forespørgsel, hvor brugeren angiver navnet på en privatbruger samt en radius. Hvis brugeren med det angivne navn findes inden for søge radiussen, bliver denne brugeres information vist.

Find butiksbruger Forespørgsel, hvor brugeren angiver navnet på en butiksbruger samt en radius. Hvis brugeren med det angivne navn findes inden for søge radiussen, bliver denne brugeres information vist.

Find seværdighedsbruger Forespørgsel, hvor brugeren angiver navnet på en seværdighedsbruger samt en radius. Hvis brugeren med det angivne navn findes inden for søge radiussen, bliver denne brugeres information vist.

Find transportbruger Forespørgsel, hvor brugeren angiver navnet på en transportbruger samt en radius. Hvis brugeren med det angivne navn findes inden for søge radiussen, bliver denne brugeres information vist.

Find transportbruger udfra destination Forespørgsel, hvor brugeren angiver et navn på en transportbruger samt en radius. Hvis der findes en eller flere transportbrugere, som indgår i samme rute som den angivne bruger, så vises info om den, af de fundne transportbrugere, hvor der er kortest tid til en afgang til den angivne transportbruger.

Find produkt Forespørgsel, hvor brugeren angiver navn på et produkt samt en radius. Hvis der findes en eller flere butiksbrugere inden for den angivne radius, der sælger det angivne produkt, så vises disse butiksbrugere i en liste. Ved at vælge en af butiksbrugerene i listen vises information om denne butik. Via instilling af bruger filtret kan der vælges om der skal vises produkter fra alle butikker eller kun fra åbne butikker.

Gemme setup Der gemmes en række brugerdefinerede oplysninger i en fil på klienten. Det drejer sig om bruger id, password, radius ved visning af kort og hvilken server instans brugeren er tilknyttet. (Se mere om server instanser i afsnit 5.4).

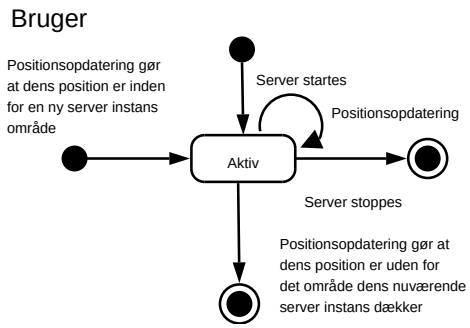
Vis produkter Der kan vises en liste af produkter som sælges i en bestemt butik. Dette forudsætter, at der er hentet information om en butik, enten i forbindelse med en forespørgsel eller ved hentning af kort og efterfølgende valg af en butik enten via et ikon på kortet eller via listen over brugere på kortet.

Vis rute Når der bliver vist information om transportbrugere, så vises der bl.a. en liste af ruter som stopper ved den viste transportbruger. Ved at vælge en af ruterne kan brugeren få vist den valgte rute. Det indbefatter en liste over alle de transportbrugere (stoppesteder) der indgår i ruten, og hvornår ruten afgår fra hver af dem.

Gem bruger objekt i database En del af en brugers information befinder sig i main-memory og den skal gemmes i databasen når serveren lukkes ned.

Ændre bruger information i database Dette indbefatter ændring af den bruger information der er placeret i database. F.eks. butiksbrugere der tilføjer nye produkter eller ændre priser på produkter, og privatbrugere der ændre den tekst der beskriver dem.

Sammenhængen mellem klasser og hændelser kan beskrives i et tilstandsdiagram, der beskriver livsforløbet af objekter af en klasse. Et tilstandsdiagram består af de tilstande objekterne kan være i, samt hvilke hændelser der skaber en transition fra en tilstand til en anden. Det er kun **Bruger** objekter der indgår i hændelser der gør at de skifter tilstand. De resterende klasser benyttes kun til at overføre information fra server til klient. Dvs. de bliver oprettet udfra data i databasen, hvorefter de overføres til klienten, hvor deres information vises på den grafiske brugergrænseflade. Figur 4.2 viser tilstandsdiagrammet for **Bruger**.



Figur 4.2: Tilstandsdiagram for bruger

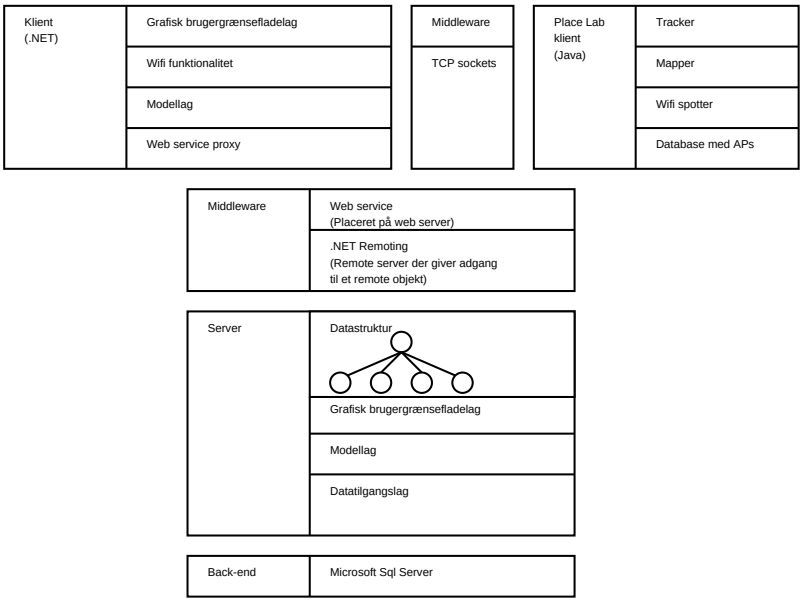
I kravsspecifikationen fremgår det at der skal benyttes et GIS, og til dette formål vil MapPoint web services blive brugt. Desuden skal det være muligt at opnå postionsbestemmelse både indendørs og udendørs. Indendørs positionsbestemmelse betyder i øjeblikket at der skal benyttes radio beacons, jvf. kapitel 3. Place Lab er en billig og let tilgængelig måde at benytte radio beacons og vil derfor indgå i systemet. Da Place Lab positionering kun er muligt i områder hvor der er indsamlet information om radio beacons, vil det ikke kunne stå alene, derfor skal der også benyttes GPS. Denne løsning er langt fra optimal, men indtil indendørs AGPS bliver kommercielt tilgængelig er det en acceptabel løsning.

4.2 Arkitektur

En arkitektur er en overordnet strukturering af systemets dele, og den udtrykker den helhed som de enkelte dele skal indpasses i. Arkitekturen for systemet er opbygget som N-niveau arkitektur for et klient-server system. Systemet består derfor af følgende fire niveauer, klient, middleware, server og back-end.

Klienten er opdelt i to dele, hvoraf den ene del er den egentlige klient som brugeren interagerer med. Den anden del indsamler information om wifi APs, udregner en position udfra de fundne APs og en database over positionsbestemte APs, og sender positionsinformationen til den anden klient del via TCP sockets. Grunden til at klienten er delt i to er at Place Lab APIet er skrevet i java, og derfor ikke kan indgå i den anden klient del, der er skrevet i C#, og dermed .NET frameworket. Der er to grunde til at .NET frameworket benyttes, for det første var det i forbindelse med den indledende eksperimenteren med forskellige GPS APIer ikke muligt, at få et java GPS API til at fungere på iPAQen. For det andet er det væsentlig hurtigere at lave grafiske brugergrænseflader til mobile apparater i .NET i forhold til java.

Udover niveau opdelingen er systemet opdelt i lag i henhold til OOAD [59]. .NET klient delen består af to web service proxyer, et modellag, der indeholder de syv klasser der blev identificeret i analysen, et wifi funktionalitetslag der sørger for kommunikation med Place Lab klient delen, samt et grafisk brugergrænsefladelag. Place Lab klient delen består af et wifi funktionalitetslag, bestående af listener, tracker, mapper og spotter, samt en database med positionsbestemte APs. Middleware delen består af en .NET web service, placeret på en IIS, samt en .NET remoting server der giver adgang til et remote objekt. Serveren består af modellaget, et datatilgangslag, datastrukturen der indekserer bruger objekter udfra deres position, og et grafisk brugergrænsefladelag. Det grafiske brugergrænsefladelag bruges i forbindelse med monitoring og vedligeholdelse af serveren. Back-end delen består af en Microsoft Sql Server database indeholdende bruger information. En oversigt over arkitekturen kan ses på figur 4.3.



Figur 4.3: Systemets arkitekur

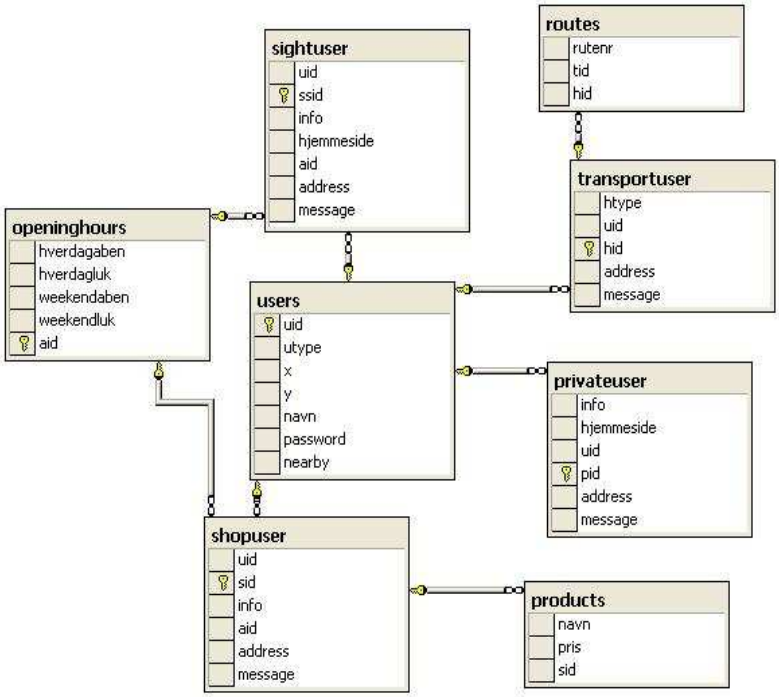
Kapitel 5

Design

Indholdsfortegnelse

5.1 Database

Databasen er en Microsoft Sql Server database og indeholder bruger information. Den består af otte tabeller, hvoraf de syv repræsenterer de syv klasser i modellaget. Den ottende indeholder åbningstidspunkter for butiksbrugere og seværdighedsbrugere. Grunden til at åbningstider har fået en tabel for sig er, at de samme åbningstider benyttes af mange brugere. Ved at butiksbrugere og seværdighedsbrugere ikke selv indeholder åbningstider, men i stedet har en reference til en kolonne i åbningstidstabellen undgås redundant data. I forbindelse med opstart af serveren oprettes bruger objekter udfra data i databasen. Desuden benyttes databasen ved forespørgsler der involverer bruger data der ikke er i main-memory, samt i forbindelse med visning af al information om en bruger. Databasen tilgås via Cω kald, placeret i klassen **DbAccess**. Figur 5.1 viser et diagram over databasen.



Figur 5.1: Diagram over databasen

5.2 Datastruktur

Et system med mange brugere, der løbende opdaterer deres positioner, og foretager søgninger, stiller en række krav til den datastruktur der skal indeksere brugernes data. For at kunne designe en passende datastruktur er det vigtig at fastlægge hvilke krav der er til datastrukturen, samt tilpasse datastrukturen til systemets problemdomæne. Følgende liste beskriver kravene til datastrukturen.

Hyppige opdateringer En del af brugerne bevæger sig, hvormed deres positionsdata løbende ændrer sig. Det skal reflekteres i de data objekter der repræsenterer brugerne i datastrukturen, derfor skal datastrukturen kunne klare hyppige opdateringer.

Område søgning Det skal være muligt at finde alle brugere inden for et bestemt område, defineret af en rektangel. Derfor skal datastrukturen understøtte at dette kan gøres hurtigt.

Indeksering af spatiale data objekter i to dimensionelt rum En datastruktur skal effektivt kunne indeksere data objekter udfra x,y koordinater, idet data objekterne repræsenterer personer og steder, hvis positioner angives med netop

x,y koordinater.

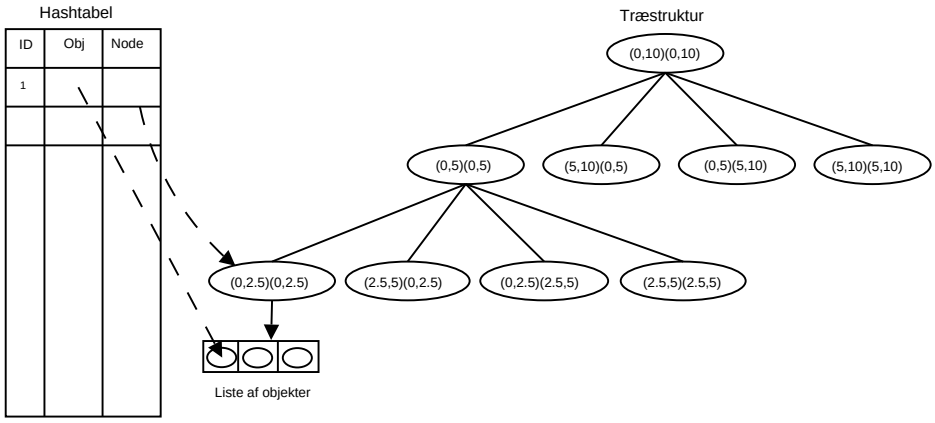
Hurtigt at finde bestemt bruger Da brugernes positioner hyppigt bliver opdateret er det nødvendigt hurtigt at kunne få fat på en brugers data i datastrukturen, så opdateringen kan udføres.

Designet af datastrukturen er baseret på PR-Quadtræet, samt teknikker fra LUR-træet. Fra LUR-træet benyttes følgende teknikker. Til opdateringer benyttes den opdateringemetode der rekursivt undersøger parent-knuder, med udgangspunkt i det opdaterede objekts nuværende knudes parent. Der gøres også brug af direkte link til objekterne via en hashtabel, der benytter brugerens id som hashværdi. Der benyttes de grundlæggende egenskaber ved PR-Quadtræet, nemlig en rekursiv opdeling af et område i fire kvadrater.

De operationer der hyppigst foretages håndteres i main-memory, uden at involvere databasen. Den brugerdata der er placeret i main-memory er position, navn, password, nearby, type og id. Denne information er nok til, at en bruger kan lokaliseres i forbindelse med område forespørgsler der kombinerer navn, lokation og type. F.eks. "Find alle butikker inden for et område på 200*200 meter som hedder Aldi"Desuden giver navn og type nok information til, at en bruger kan identificeres af andre brugere, når brugeren repræsenteres på et kort. Ved visse område forespørgsler er det dog nødvendig at benytte databasen, nemlig når åbningstider på butikker og seværdigheder, produkter i butikker, eller ruter tilknyttet stoppesteder er involveret. F.eks. "Find alle åbne butikker der sælger tun på dåse". Main-memory dataen bruges desuden til positionsopdateringer, der sammenholdt med den direkte link til objekter via en hashtabel hjælper med til, at systemet kan håndtere hyppige positionsopdateringer. Databasen involveres også når en bruger ønsker at se en liste af produkter i en butik, ønsker at se en rute, ønsker at ændre på data i databasen, eller ønsker at se uddybende oplysninger om en anden bruger. Enten ved at trykke på et ikon på et kort, eller ved at vælge en bruger fra en liste.

Figur 5.2 viser hvordan datastrukturen er opbygget. Den består af et træ samt en hashtabel. I træet er der angivet størrelsen af hver knudes rektangel, og ved den yderste blad-knude er der vist en liste af objekter. Rod knuden har en rektangel der dækker hele området, i dette tilfælde et areal på 100 enheder, idet værdierne på både x og y akserne går fra 0 til 10. Hashtabellen benytter bruger id som hashværdi, og indeholder, for hvert objekt, en reference til objektet, samt en reference til den blad-knude objektet er placeret i.

Datastrukturen har følgende egenskaber. For det første er rektanglerne som hver knude dækker kvadrater, og størrelsen på de kvadrater der er tilknyttet blad knuder er fastlagt ved en variabel i systemet. For det andet er antallet af knuder statisk, dvs. antallet af knuder når systemet startes vedbliver med at være det samme. Det betyder at det geografiske område som systemet dækker er fastlagt på forhånd, og



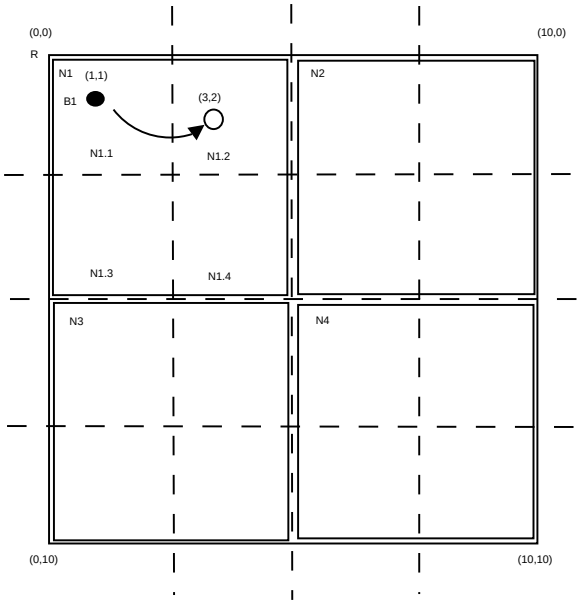
Figur 5.2: Datastruktur

brugere, hvis positioner ikke falder inden for dette område, bliver fjernet fra datastrukturen. De fjernes dog ikke fra databasen, og vil derfor igen blive tilføjet til datastrukturen når brugeren igen er inden for systemets område. For det tredje er der ingen begrænsning på antallet af objekter der kan være indeholdt af en blad-knude. Fordelene ved disse egenskaber er følgende. Ved at lade antallet af knuder være statiske, og ved ikke at begrænse antallet af objekter i blad-knuder skal der ikke foretages ændringer i træet når først det er oprettet. Der vil derfor ikke være behov for at indbygge mekanismer der gør at træet opretholde balancen, f.eks. at fastsætte grænser for hvor mange objekter der kan være i en knude, samt fjerne eller tilføje knuder. En sådan forudsigelig opførsel og fordeling af objekter i datastrukturen gør, at den kan gøres statisk, hvilket forbedrer hastigheden af opdateringer. Som nævnt i indledningen er datastrukturen døbt Statisk PR-Quadtræ (SPRQ-træ)

SPRQ-træets egenskaber er hensigtsmæssige på grund af systemets problemdomæne. Som nævnt i indledningen så repræsenterer de objekter der skal indekseres, personer, butikker, seværdigheder og transportinfrastruktur, som busstop og banegårde. Det vil derfor være forudsigeligt, hvilke områder en bruger befinder sig i størstedelen af tiden, og nye brugere vil være jævnt fordelt over det område systemet dækker. Der vil selvfølgelig være regionale forskelle i forbindelse med by og landområder, men dette løses ved at regulere opløsningen på træet ved områder med forskellige brugertætheder, i forbindelse med distribution af systemet. Opløsningsgraden, dvs. størrelsen af blad-knudernes område, er afhængig af antallet af knuder. Antallet af knuder afhænger af følgende. Som nævnt har hver ikke-blad knude fire child-knuder, hvilket medfører at antallet af knuder kan findes ved $4^0 + 4^1 + ... + 4^N = n$, hvor N er højden af datastrukturen og n er antallet af knuder. I systemet afgøres størrelsen af N (og n) af to variabler, nemlig størrelsen på den rektangel der gives som input til metoden der opbygger datastrukturen, og variablen der angiver minimum længde på siderne af blad-knudernes rektangler.

Størrelsen af det geografiske område som hver blad-knude skal dække over vil være afhængig af hvilket område systemet skal dække. Ideen med systemet er at det skal dække store områder, det betyder at hver blad-knude skal dække et passende område af det samlede område. Præcist hvor stort et passende område er, afhænger af hvor stor tætheden af brugere er. I byer vil tætheden være langt større end på landet, og område størrelsen skal derfor være mindre. I forbindelse med distribution af systemet kan datastruktur instanser med forskellige opløsningsgrader sættes til at dække over områder med ens brugertæthed. Dvs. instanser med høj opløsningsgrad dækker byer og instanser med lav opløsningsgrad dækker landområder. Det vil være nødvendig at udføre en række tests på områder med forskellige brugertætheder for at finde de optimale værdier på de to variabler ved forskellige brugertætheder.

Figur 5.3 illustrerer hvordan et område bliver opdelt. R er roden, der dækker hele området, og N1 til N4 er rodens child-knuder, derudover er vist opdelingen af N1 i knuderne N1.1 til N1.4. Der er vist en enkelt bruger B1, indsat på positionen (1,1), der er ved at opdatere sin position til positionen (3,2). Denne opdatering medfører at brugeren skal flyttes fra knude N1.1 til knude N1.2.



Figur 5.3: Opdeling af geografisk område

5.2.1 Algoritmer

Opdateringsalgoritmen finder en ny knude til objekter, ved rekursivt at undersøge parent-knuder, hvilket er muligt da hver knude indeholder en reference til dens parent. Den skal også sørge for at objekter, hvis position falder uden for det område

systemet dækker, fjernes fra både træet og hashtabellen, og ligeledes tilføje objekter hvis positioner, efter en opdatering, befinder sig på systemets område. Listing 5.1 viser i pseudokode, hvordan opdateringsalgoritmen fungerer .

```
Metode FlytBruger(x,y,id){
    Hvis(HashTabel indeholder bruger){
        objektArray = HashTabel[id]
        bruger = objektArray[0]
        node = objektArray[1]
        bruger.x = x
        bruger.y = y
        Hvis ikke(bruger indenfor knudes rektangel){
            TjekParent(knude.parent,bruger)
            fundetKnude = knudeTilNyPlacering (global variabel fundet ifm. TjekParent)
            Hvis(fundetknude == null){
                FjernBruger(bruger)
            }
            Ellers{
                FjernBruger(bruger)
                fundetknude.brugerliste.tilføj(bruger)
                HashTabel.tilføj(id,ny ObjektArray(bruger,fundetknude))
            }
        }
    }
    Ellers{
        bruger = database.hentBrugerFraDB(id)
        bruger.x = x
        bruger.y = y
        tilføjBruger(bruger)
    }
}
```

Listing 5.1: Pseudokode af opdateringsalgoritmen

I metoden *FlytBruger* bruges metoden *TjekParent* til at finde en ny knude, til den bruger der flyttes. Resultatet af *TjekParent* er, at metoden *FindknudeTilBruger*, assigner en global variabel til den knude som brugeren skal placeres i. I listing 5.2 ses pseudokoden til *TjekParent*.

```
Metode TjekParent(knude,bruger){
    Hvis ikke(bruger indenfor knudes rektangel){
        Hvis ikke(knude.parent == null){
            TjekParent(knude.parent,bruger)
        }
        Ellers{
            FindknudeTilBruger(bruger,knude)
        }
    }
}
```

Listing 5.2: Pseudokode af metoden *TjekParent*

Metoden *FindKnudeTilBruger* bruges i forbindelse med *TjekParent* og *TilføjBruger*. Den finder den knude en bruger skal placeres i, udfra brugerens position, og assigner som nævnt en global variabel til denne knude. Listing 5.3 gennemgår metoden i pseudokode.

```
Metode FindKnudeTilBruger(bruger,knude){
    Hvis(bruger indenfor knudes rektangel og knude er blad-knude){
        knudeTilNyPlacering = knude (knudeTilNyPlacering er en global variabel)
    }
    Ellers{
        Hvis(knude er ikke-blad knude){
            Hvis(bruger indenfor knude.child1 rektangel){
                FindKnudeTilBruger(bruger,knude.child1)
            }
            Hvis(bruger indenfor knude.child2 rektangel){
                FindKnudeTilBruger(bruger,knude.child2)
            }
            Hvis(bruger indenfor knude.child2 rektangel){
                FindKnudeTilBruger(bruger,knude.child2)
            }
            Hvis(bruger indenfor knude.child2 rektangel){
                FindKnudeTilBruger(bruger,knude.child2)
            }
        }
    }
}
```

Listing 5.3: Pseudokode af metoden *FindKnudeTilBruger*

En af de centrale funktioner i systemet er muligheden for, at finde brugere inden for en brugerdefineret rektangel. Hvilke brugere der vælges er afhængig af den område forespørgsel der foretages. Metoden *FindBrugereIRektangel* bruges til at finde alle brugere inden for et kvadratisk område, og kaldes af metoder der håndterer specifikke forespørgsler. Disse metoder udvælger så de af de fundne objekter der passer til forespørgslen. For at begrænse antallet af fundne objekter benyttes et filter, hvor der kan vælges hvilke typer brugere der skal findes. De fire brugertyper kan vælges til og fra, der kan vælges at kun de butikker eller seværdigheder der er åbne på søgningstidspunktet skal findes, og ved transportbrugere kan der vælges om alle, kun busstoppesteder, eller kun togstoppesteder skal findes. Pseudokode til *FindBrugereIRektangel* kan ses i listing 5.4.

```
Metode FindBrugereIRektangel(knude,rektangel,filter){
    Hvis(knude er ikke-blad knude){
        Hvis(knude.child1 overlapper rektangel){
            FindBrugereIRektangel(knude.child1,rektangel,filter)
        }
        Hvis(knude.child2 overlapper rektangel){
            FindBrugereIRektangel(knude.child2,rektangel,filter)
        }
        Hvis(knude.child3 overlapper rektangel){
            FindBrugereIRektangel(knude.child3,rektangel,filter)
```

```
        }
12      Hvis(knude.child4 overlapper rektangel){
            FindBrugereIRektangel(knude.child4,rektangel,filter)
14      }
    }
16  Ellers{
        For hver(bruger i knude.brugerList){
18          Hvis(bruger er indenfor rektangel){
                Hvis(ErBrugerIFilter(bruger,filter) == sand){
20                  iRektangel.tilføj(bruger) (iRektangel er en global variabel)
                }
22          }
        }
24  }
}
```

Listing 5.4: Pseudokode af metoden *FindBrugereIRektangel*

Sidst men ikke mindst er der metoden *BygStruktur*, der opbygger træet. Det gøres ved at give den en rod-knude som input. Rod knudes rektangel afgører hvor stort et område datastrukturen dækker, og den globale variabel *områdeStørrelse* fastsætter en minimumsgrænse på længden af siderne på blad-knudernes rektangler. Desuden gives en integer værdi som input, der bruges til at give hver knude et niveau nummer, hvor rod knuden har niveau 1. Psedokode til *BygStruktur* kan ses i listing 5.5.

```
Metode BygStruktur(knude,niveau){
2   halv = halvdelen af længden af knudens rektangel
    knudeNiveau = niveau+1
4   rek1 = ny rektangel(knude.rek.xMin,knude.rek.xMin+halv,
                        knude.rek.yMin,knude.rek.yMin+halv);
6   rek2 = ny rektangel(knude.rek.xMin+halv,knude.rek.xMax,
                        knude.rek.yMin,knude.rek.yMin+halv);
8   rek3 = ny rektangel(knude.rek.xMin,knude.rek.xMin+halv,
                        knude.rek.yMin+halv,knude.rek.yMax);
10  rek4 = ny rektangel(knude.rek.xMin+halv,knude.rek.xMax,
                        knude.rek.yMin+halv,knude.rek.yMax);
12  child1
    child2
14  child3
    child4
16  Hvis(halv > områdeStørrelse){
        child1 = ny ikke-blad knude(rek1,knude,null,null,null,null,knudeNiveau)
18      bygStruktur(child1,knudeNiveau)
        child2 = ny ikke-blad knude(rek2,knude,null,null,null,null,knudeNiveau)
20      bygStruktur(child2,knudeNiveau)
        child3 = ny ikke-blad knude(rek3,knude,null,null,null,null,knudeNiveau)
22      bygStruktur(child3,knudeNiveau)
        child4 = ny ikke-blad knude(rek4,knude,null,null,null,null,knudeNiveau)
24      bygStruktur(child4,knudeNiveau)
    }
26  Ellers{
        child1 = ny blad-knude(rek1,knude,knudeNiveau)
28      child2 = ny blad-knude(rek2,knude,knudeNiveau)
```

```
            child3 = ny blad-knude(rek3,knude,knudeNiveau)
30          child4 = ny blad-knude(rek4,knude,knudeNiveau)
        }
32      knude.child1 = child1
        knude.child2 = child2
34      knude.child3 = child3
        knude.child4 = child4
36  }
```

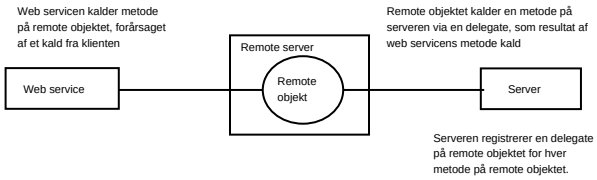
Listing 5.5: Pseudokode af metoden *BygStruktur*

5.3 Kommunikation mellem web service og server

Der er to problematiske punkter ved at have datastrukturen og web servicen samlet i en process. For det første er det ikke muligt at lave en brugergrænseflade, hverken grafisk eller konsol, i web service processen, når den placeres på IIS. Da en web service skal være kompileret til en ikke-eksekverbar DLL fil for at fungere på IIS. Derved er det ikke muligt at se oplysninger om datastrukturens tilstand, f.eks. hvor mange brugere der er indekseret i datastrukturen og hvilken kommunikation der er med klienterne. For det andet er det ikke muligt at have flere datastruktur instanser, hvilket går ud over skalerbarheden. Desuden skal der benyttes flere datastruktur instanser for at systemet kan benytte nok main-memory RAM til at have et højt SPRQ-træ, samt en del af brugernes data i main-memory. Derfor er det nødvendig at have yderligere en process, hvor datastrukturen er placeret. Det medfører, at der er behov for middleware til at kommunikere mellem de to processer. Den valgte middleware skal muliggøre kommunikation initieret af web servicen, samt kunne sende data om klient kommunikation, i real-time, fra web servicen til den anden process. Dette udelukker brugen af en web service proxy placeret i den anden process, da web servicen derved ikke kan initierer kommunikation og kommunikationen kan derfor ikke foregå i real-time.

Løsningen på problemet er at benytte remote events via .NET remoting. Derved kan både web servicen og den anden process være klienter, der kommuniker via et remote objekt placeret på en remote server. Den anden process indeholder udover datastrukturen også tilgangen til databasen (denne process vil herefter blive betegnet serveren). Metodehovederne på remote objektets metoder svarer til metodehovederne på web servicens metoder, samt til metoder på serveren. Web servicens metoder har ikke selv nogen funktionalitet men videresender blot klientens kald til serveren via remote objektet og returnerer en evt. returværdi til klienten. På remote objektet skal der registreres en delegate for hver af metoderne. Derved kan remote objektet kalde den metode på serveren der er tilknyttet delegaten. Grunden til at der benyttes delegates og ikke events har at gøre med distribution af systemet hvilket bliver forklaret i afsnit 5.4. Samlet set betyder det at kommunikationen fra klienten

til serveren fungerer på følgende måde. Når en metode på web servicen kaldes af klienten, så kalder web servicen en metode på remote objektet, der så via en delegate kalder en metode på serveren. Hvis der er en returværdi returneres denne først til remote objektet så til web serveren og endelig til klienten. Figur 5.4 illustrerer denne kommunikation.

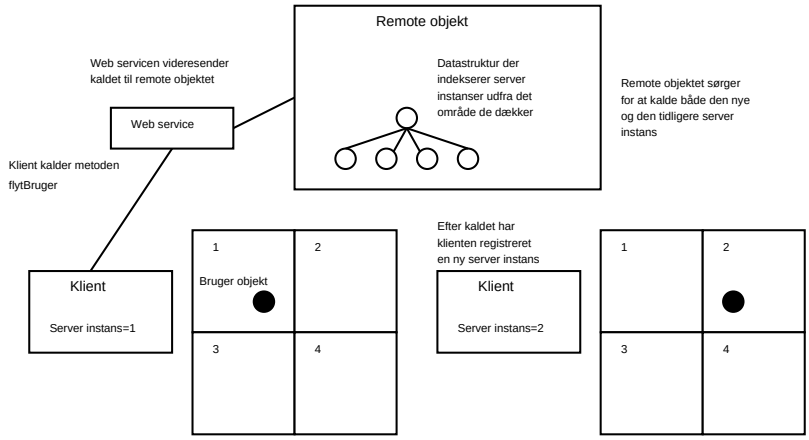


Figur 5.4: Remote events

5.4 Distribution

Opdelingen af web servicen og datastrukturen i to processer gør det muligt at lade flere web service instanser tilgå den samme remote objekt instans og lade en remote objekt instans tilgå flere datastruktur (server) instanser. Derved kan systemet distribueres ud på flere maskiner, hvilket øger skalerbarheden og gør det muligt at have mange knuder og bruger objekter i main-memory. Remote objektet er central i distribueringen, idet det skal sørge for at klient kaldene bliver videresendt til de rette server instanser. Det kræver at remote objektet har kendskab til hvilket område hver server instans dækker. Remote objektet skal derfor benytte en datastruktur magen til den datastruktur der bruges til at indeksere bruger objekter, blot med den forskel at det er server instanser der skal indekseres. Hver server instans er repræsenteret på remote objektet ved et unikt id, en rektangel, for det områder instansen dækker, samt en delegate for hver metode på serveren der skal kaldes som resultat af et kald til en metode på remote objektet. Desuden skal hver klient instans vide hvilken server instans det bruger objekt, der repræsenterer brugeren af den pågældende server instans, befinder sig på. Det betyder at en klient instans skal ændre hvilken server instans den befinder sig på, hvis en positionsopdatering gør at den kommer uden for det område dens nuværende server instans dækker. Figur 5.5 illustrerer dette.

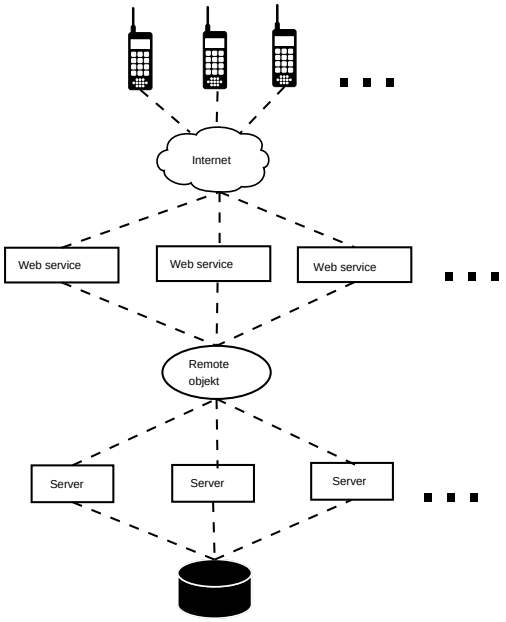
Remote objektet skal ved kald der involverer flere server instanser, f.eks. område forespørgsler, sørger for at sammensætte returværdierne af kaldene til hver server instans til en samlet returværdi der, via web servicen, returneres til klienten. I forbindelse med område søgning vil det betyde at remote objektet skal sammesætte de arrays, indeholdende bruger objekter, der kommer som returværdi fra kaldene til de involverede server instanser, til et samlet array. Ved positionsændring skal remote objektet finde ud af om ændringen kun involverer brugerens nuværende ser-



Figur 5.5: Eksempel på ændring af server instans

ver instans eller om ændringen gør at objektet skal flyttes over på en anden server instans. Hvis det sidste er tilfældet så kaldes først *flytBruger* metoden på den nuværende server instans, hvilket gør at bruger objektet fjernes fra datastrukturen ved denne server instans. Derefter kaldes *flytBruger* på den server instans som dækker det område som bruger objekter efter positionsændringen skal være på. Hvorved bruger objektet bliver tilføjet til datastrukturen ved den nye server instans. Tilsidst sender remote objektet den nye server instans id som returværdi til klienten. Formålet ved at klienten ved hvilken server instans den befinder sig på er, at remote objektet derved ikke behøver, at tjekke indenfor hvilken af de registrerede server instanser klienten befinder sig.

Der er fire dele som potentielt kunne have flere instanser distribueret ud på forskellige maskiner, nemlig web servicen, remote objektet, serveren og databasen. I systemet er det kun web servicen og serveren der har flere instanser. Databasen kunne også opdeles og distribueres, det ville kræve at klienten indeholdt information om hvilken server instans der tilgår den database del, hvor klientens bruger information er placeret. Desuden vil det kræve, at remote objektet står for koordineringen i de tilfælde, hvor bruger objektet er placeret på en anden server instans, end den server instans der tilgår databasen med den pågældende brugers information. Remote objektet selv kunne også distribueres. Dette kunne gøres ved, at hver remote objekt indeholder information om hvilke områder de andre remote objekter dækker, og hvilke server instanser der er registreret på dem. Det ville være muligt at indføre redundans blandt remote objekterne, dvs. at flere remote objekter dækker over det samme område. Men den vigtigste del at få distribueret er serveren, da den indeholder SPRQ-træet, og derfor skal bruge en stor mængde main-memory RAM. Figur 5.6 giver et samlet overblik over distribueringen af systemet.



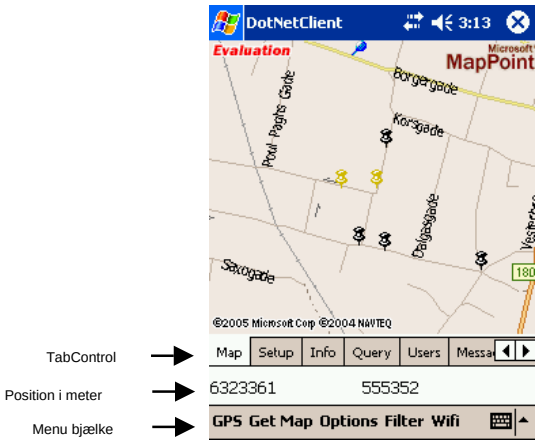
Figur 5.6: Oversigt over distribuering af systemet

5.5 .NET klient

.NET klienten er den del af systemet som brugere interagerer direkte med, hvilket sker via en grafisk brugergrænseflade. Den indhenter og sender information til systemets andre dele. Dette sker automatisk, i forbindelse med positionsinformation indhentet fra enten en GPS modtager eller fra Place Lab klienten, og ved positionsopdateringer. Desuden sker det på foranledning af brugeren, i forbindelse med forespørgsler, område søgninger, visning af kort og visning af information om en bestemt bruger. Overordnet set er den et værktøj der giver brugeren mulighed for, at udføre de funktioner der blev beskrevet i analyseafsnittet (4.1).

Følgende tre klasser udgør .NET klienten, **Form1** der nedarver fra **Form**, **WifiReceiver** og **WifiArgs**. Form klassen indeholder den grafiske brugergrænseflade, en instans af **WifiReceiver** og en tråd der kører metoden *run*, på klassen **WifiReceiver**. Metoden *run* kører en uendelig løkke hvori der modtages og videresendes wifidata. Klassen **WifiReceiver** fungerer som TCP klient og indeholder den førnævnte *run* metode. Desuden indeholder den en event som form klassen abonnerer på, for at kunne få overført wifi data, i form af en instans af **WifiArgs**, med det samme det modtages. **WifiArgs** opdeler den streng med wifidata der modtages fra Place Lab klienten, og tilføjer de enkelte dele til sine attributter. Foruden de tre klasser består klienten også af proxyer for både systemets web service, samt MapPoint web services.

MapPoint web services benyttes som GIS, og muliggør at der kan hentes kort med brugere placeret i henhold til deres koordinater. Dette gøres med metoden *GetMap* på web servicen *RenderService*. Den tager som input et centerpunkt med et tilknyttet koordinatsæt, og et array af objekter af klassen **PushPin**, hvor hvert objekt tilknyttes et koordinatsæt. Hvilket udsnit af verdenen et kort dækker afgøres af centerpunktet, samt en brugerdefineret radius. Koordinatsættene for **PushPin** objekterne afgør, hvor på kortet de vises. Desuden kan der vælges mellem hvilke ikoner objekterne skal vises som, og et bruger id knyttes til hvert objekt. Når kortet er hentet vises det på klienten, og brugeren kan så pege på en af ikonerne, hvilket resulterer i, at der via serveren hentes information om den bruger, hvis id er tilknyttet den **PushPin** instans, som ikonet repræsenterer. Figur 5.7 viser hvordan et MapPoint kort med ikoner ser ud på klienten. De gule ikoner repræsenterer privatbrugere, de sorte er butiksbrugere og den blå er en transportbruger. Ikonet i midten repræsenterer brugeren af klienten.



Figur 5.7: Eksempel på MapPoint kort med ikoner

Brugen af MapPoint web services som GIS er ikke optimalt i flere henseender. Det største problem er, at kortene ikke kan gemmes på klienten, men skal hentes over en netværksforbindelse hver gang de skal vises. Et andet problem er, at ikonerne bliver placeret på et kort i forbindelse med det web service kald der henter kortet. Derved er det ikke muligt løbende, at hente de brugere der er inden for det område et hentet kort dækker, og så opdatere deres placering på dette kort. Målinger af hvor lang tid det tager at hente 32 brugere og hvor lang tid det tager at hente et kort viser, at det tager 4-6 sekunder at hente brugerne og 15-24 sekunder at hente kortet (begge dele over en GPRS forbindelse). En bruger der går rundt vil befinde sig indenfor det samme korts område i et vist tidsrum, alt efter hvor retningsbestemt vedkommende bevæger sig. Derfor vil det være bedre kun at hente ét kort én gang og så opdatere bruger placeringer på det, istedet for at hente kortet hver gang brugeren ønsker at se de nyest opdaterede placeringer. En anden mulighed er at have kort over hele det område systemet dækker placeret på et hukommelseskort på de

apparater hvor klienten køres. Dette benyttes i forbindelse med navigationssystemer som TomTom [43] og Garmin [15]. Men dette er som nævnt ikke muligt når MapPoint web services benyttes som GIS. MapPoint web services har dog også visse fordele, bl.a. er det nem at benytte og det dækker hele Europa og Nord amerika, hvilket ikke ville være muligt på et hukommelseskort. Grunden til at MapPoint web services benyttes er, at geokodede kort ikke er umiddelbart tilgængelige, det kræver at der laves aftaler med de firmaer der ejer dem, for at få adgang til dem.

Den grafiske brugergrænseflade er designet med henblik på, at give brugeren hurtig adgang til samtlige funktioner. En given funktion er aldrig længere væk end et par tryk med stylusen, ligemeget hvor i grænsefladen brugeren befinder sig. Den er opdelt i tre dele, nederst en menubjælke, dernæst to labels der viser brugerens position i meter, og tilsidst en tabControl, hvor der kan skiftes mellem en række skærmbilleder. På figur 5.7 er der f.eks. valgt "Map"i tabControlen, hvorved der vises et kort, hvis brugeren forinden har trykket på menupunktet "Get Map", eller en "Show Map"knap i forbindelse med forespørgsler. I appendiks A er der en illustreret demonstration af klienten.

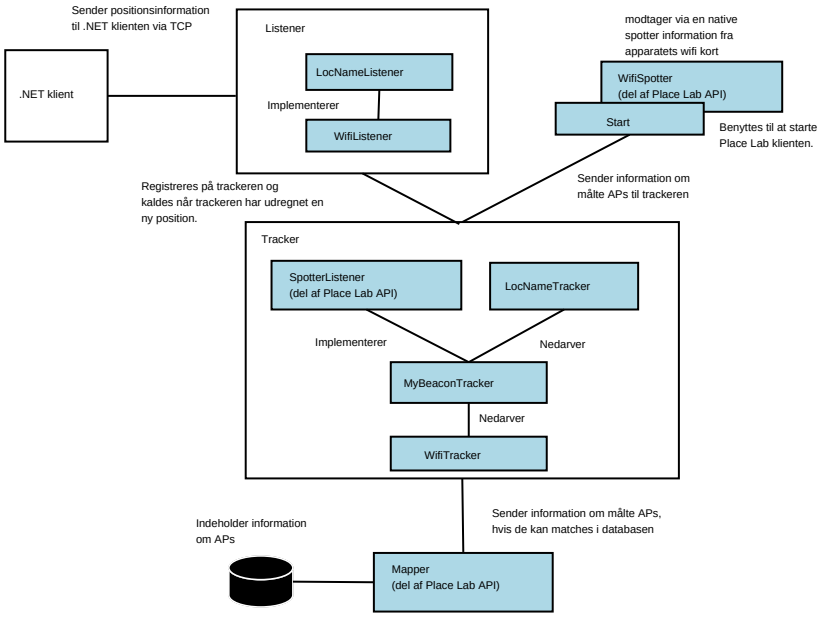
5.6 Place Lab klient

Place Lab klienten modtager wifi signaler, og udfra signalerne udregner den en position, som sendes til .NET klienten. Den består af de tre centrale komponenter fra Place Lab APIet, nemlig tracker, mapper og spotter, jvf. 3.1. Desuden fungerer den som en TCP server, for at kunne sende positionsinformation til .NET klienten.

En tracker bruges til at udregne en position ved at have en reference til en mapper, og modtage signal information indkapslet i klassen **Measurement** fra en spotter. En eller flere listener klasser kan abonnerer på en tracker, dvs. trackeren kalder en metode på de registrerede listeners. Hvornår og med hvilke parametre trackeren kalder listener klasserne med er applikationsafhængig. I Place Lab klienten sker det når det har været muligt at matche en eller flere af de spottede APs med APs i den database der tilgås via mapperen. Det input der gives til metoden er et koordinatsæt indeholdende et gennemsnit af koordinaterne på de målte APs, samt det navn der er registreret i databasen for den AP, hvor der er målt den stærkeste signalstyrke. Navnet på APen med den stærkeste signalstyrke kan bruges til at afgøre, hvor i en bygning en bruger befinder sig. F.eks vil det i et stort indkøbscenter være svært at afgøre, hvor en bruger befinder sig udfra en placering på et MapPoint kort. Kortet viser ikke indkøbscentrets interne "veje"og hvis der er flere etager vil det ikke være muligt at afgøre hvilken etage brugeren befinder sig på. Ved at bruge signalstyrke samt registrerer indkøbscentrets APs med signede navne, f.eks. butiksnavn og etage, vil man kunne bruge navnet på den AP der er tættest på til at angive hvor brugeren er placeret. Dette vil også være brugbar i forbindelse med universiteter

og større virksomheder.

Place Lab klienten bygger på klasse hierarkiet i Place Lab APIet. Klasserne der vedrører den benyttede listener og tracker funktionalitet er blevet tilpasset til systemet. Klasserne der implementerer tracker funktionaliteten er baseret på **CentroidTrackerExample**, **BeaconTracker** og **Tracker**, der er omdannet til klasserne **WifiTracker**, **MyBeaconTracker** og **LocNameTracker**. Klasserne der vedrører listener funktionaliteten består af klassen **LocNameListener** der er baseret på interface klassen **SpotterListener**, og består derudover af en implementation af dette interface i klassen **WifiListener**. **WifiListener**, registreres på trackeren og kaldes derfor når trackeren har udregnet en position. Desuden indeholder den TCP server delen og sørger for at serialisere den information der sendes til .NET klienten. Place Lab klienten består desuden af klassen **Start**, der starter den, og som samtidig modtager information om målte APs, idet den indeholder en spotter i form af en instans af klassen **WifiSpotter**. Med fem sekunders mellemrum kaldes metoden *updateEstimate* på trackeren med spotterens målinger som input. Figur 5.8 giver en samlet oversigt over hvordan de forskellige dele og klasser hænger sammen.



Figur 5.8: Oversigt over Place Lab klienten

Kapitel 6

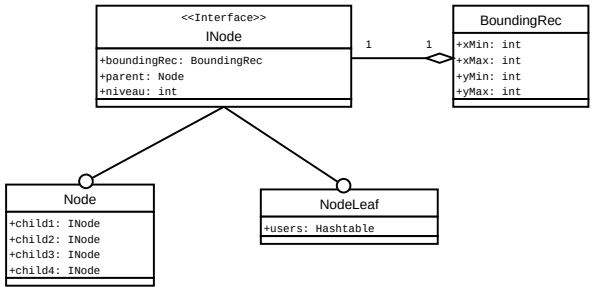
Implementation

Indholdsfortegnelse

6.1 Datastruktur

Implementationen af datastrukturen består af tre knude klasser, **INode**, **Node** og **NodeLeaf**, klassen **BoundingRec**, der repræsenterer en kvadrat, og klassen **Datastructure**, der indeholder de metoder der benyttes til at oprette, søge i og opdatere datastrukturen. **INode** er et interface der indeholder de attributter der er fælles for både blad-knuder og ikke-blad knuder. **Node** repræsenterer ikke-blad knuder, og implementerer **INode** interfacet. Desuden har den fire parametre der repræsenterer en ikke-blad knudes fire child-knuder. **NodeLeaf** klassen repræsenterer bladknuder, og implementerer ligeledes **INode** interfacet. Den har én attribut, nemlig en hashtabel der skal indeholde referencer til de bruger objekter hvis position er indenfor den **BoundingRec** der er tilknyttet en given **NodeLeaf** instans. Relationen mellem de tre knude klasser og **BoundingRec** kan ses i figur 6.1.

For at benytte datastrukturen skal der oprettes en instans af **Datastructure** med to input parametre. En **INode**, der kommer til at være roden i træet, og en int der angiver minimumsstørrelsen af siderne på de **BoundingRec** der tilknyttes **NodeLeaf** instanser. Størrelsen af den **BoundingRec** der er tilknyttet rod-knuden og den førnævnte int, afgører opløsningen og højden af træet. Listing 6.1 viser koden for konstruktoren i **Datastructure** klassen.



Figur 6.1: Relation mellem knude klasserne

```
    public Datastruc(INode rootNode, int areaSize)
2   {
        this.rootNode = rootNode;
4       this.areaSize = areaSize;
        userTable = new Hashtable();
6       inRec = new ArrayList();
        db = new DbAccess();
8       buildStructure();
        initUsers();
10  }
```

Listing 6.1: Konstruktoren i **Datastructure** klassen

Linie 3-4 Input parametrene assignes til attributter i klassen.

Linie 5 Der initialisers en **Hashtable** instans. Hver entry i Hashtabellen skal indeholde et objekt array der indeholder en reference til et bruger objekt og et **NodeLeaf** objekt. Formålet med hashtabellen er at den give direkte adgang til de bruger objekter der indekseres i datastrukturen, samt den knude de er tilknyttet. Dette gør, at det er hurtigt at opdatere en brugers position.

Linie 6 Der initialiseres en **ArrayList** instans. Denne instans fungerer som en global variabel der indeholder de bruger objekter der bliver valgt i forbindelse med område søgning.

Linie 7 Der initialiseres en instans af klassen **DbAccess**. Denne klasse indeholder de metoder der benyttes i forbindelse med tilgang til databasen.

Linie 8 Metoden *buidStructure* kaldes. Denne metode opretter træet med udgangspunkt i input parametrene til konstruktoren.

Linie 9 Metoden *initUsers* kaldes. Denne metode opretter bruger objekter udfra brugerne i databasen, og tilføjer dem til datastrukturen.

Den resterende del af dette afsnit vil bestå af en gennemgang af metoderne der benyttes til positionsopdatering og område søgning. Disse metoder er dem der bedst

demonstrerer hvordan datastrukturen benytter opdelingen af brugerinformation i en main-memory del og en database del, samt viser hvorfor datastrukturen er hurtig til at foretage positionsopdateringer.

Metoden *MoveUser* i **Datastructure** bruges i forbindelse med positionsopdatering. Den undersøger om der i datastrukturen findes et bruger objekt for den bruger der foretager positionsopdateringen, hvis dette ikke er tilfældet undersøges der om databasen indeholder information om brugeren, hvis det er tilfældet så oprettes et nyt objekt i datastrukturen. Hvis objektet i forvejen eksisterer i datastrukturen så undersøges der om den nye position, er indenfor **BoundingRec** på objektets nuværende knude. Hvis det ikke er tilfældet så undersøger den om der findes en knude med en **BoundingRec** der dækker objektets nye position. Hvis en sådan knude findes så flyttes objektet til denne knude, ellers fjernes objektet fra datastrukturen. Listing 6.2 viser koden til *moveUser*.

```
public int moveUser(int id,int x,int y,string nearby,string password)
{
    if(userTable.ContainsKey(id))
    {
        object[] res = (object[])userTable[id];
        NodeLeaf n = (NodeLeaf)res[1];
        User user = (User)res[0];
        if(user.password == password)
        {
            user.x = x;
            user.y = y;
            user.nearby = nearby;
            if(!(n.boundingRec.xMin < x && n.boundingRec.xMax > x &&
                n.boundingRec.yMin < y && n.boundingRec.yMax > y))
            {
                checkParent(n.parent,user);
                if(found == false)
                {
                    removeUser(user);
                }
                else
                {
                    found = false;
                    removeUser(user);
                    retNode.users.Add(user.id,user);
                    userTable.Add(user.id,new Object[]{user,retNode});
                }
                found = false;
            }
        }
        else{
            return -1;
        }
    }
    else{
        User user = db.getUserFromDB(id);
```

```
            if(user != null){
38              if(user.password == password)
                {
40                  user.x = x;
                    user.y = y;
42                  user.nearby = nearby;
                    addUser(user);
44              }
                else{
46                  return -1;
                }
48          }
            else{
50              return -1;
            }
52      }
        return 1;
54  }
```

Listing 6.2: Metoden *moveUser*

Linie 3 Der undersøges om hashtabellen *userTable* indeholder et objekt med det id der gives som input parameter.

Linie 5-7 Det objekt der er fundet i hashtabellen assignes den opdaterede position. Parametrene x og y angiver længde og breddegrad i meter mens nearby angiver hvilken AP brugeren er i nærheden af.

Linie 8-11 Der undersøges om den nye position er indenfor objektets nuværende **BoundingRec**. Hvis dette ikke er tilfældet kaldes metoden *checkParent*, der rekursivt undersøger om parent knuder dækker det område som objektet skal placeres i, startende med parent-knuden til objektets nuværende knude.

Linie 12-16 Hvis det ikke er muligt at finde en ny knude til objektet så fjernes objektet fra datastrukturen, hvilket indbefatter både træet og hashtabellen.

Linie 17-22 Hvis der blev fundet en ny knude til objektet så fjernes objektet, hvorefter det tilføjes til både hashtabellen, med en reference til den knude der blev fundet, og til den fundne knude.

30-45 Hvis der ikke findes et objekt med det id der gives som input parameter, så kaldes metoden *getUserFromDB* i **DbAccess**. Hvis det lykkedes at finde en bruger med det pågældende id i databasen, så oprettes et objekt med denne bruger, og dette objekt opdateres med den nye position. Hvorefter objektet tilføjes til datastrukturen via metoden *addUser*.

Generalt Metoden returnerer 1, hvis der fandtes et objekt i datastrukturen, eller der kunne oprettes et objekt udfra databasen, med det givne id, samtidig med at det givne password passer til objektets password. Hvis dette ikke er

tilfældet returneres -1. Derved kan den bruger der har foretaget positions-opdateringen gøres opmærksom på, at det id der er indtastet i klienten ikke eksisterer i systemet, eller det password der er indtastet i klienten ikke passer med det password objektet med det indtastede id har.

Metoderne *findUsersInRec* og *usersInRec* bruges til område søgning. Dvs. de finder alle de brugere der findes inden for et område repræsenteret ved en **BoundingRec**. Den bruger der har forårsaget søgningen har et filter på sin klient. Indstillingen af dette filter afgør sammen med det angivende område, hvilke bruger objekter der returneres til klienten. Grunden til at der er to metoder er, at det er letterer at opdele en metode i to metoder, når metoden skal kaldes rekursivt, samtidig med at der skal gives en returværdi. Listing 6.3 viser koden til de to metoder. (Parameter navne og typer er blevet forkortet i metode headerne pga. sidestørrelse)

```
public ArrayList findUsersInRec(int id,BndRec rec,int prv,int shp,int tp,int sght)
{
    inRec = new ArrayList();
    usersInRec(id,rootNode,rec,prv,shop,transport,sight);
    return inRec;
}
private void usersInRec(int id,INode node,BndRec rec,int prv,int shp,int tp,int sght)
{
    if(node is Node)
    {
        Node n = node as Node;
        if(overlap(n.c1,rec))
        {
            usersInRec(id,n.c1,rec,prv,shop,transport,sight);
        }
        if(overlap(n.c2,rec))
        {
            usersInRec(id,n.c2,rec,prv,shop,transport,sight);
        }
        if(overlap(n.c3,rec))
        {
            usersInRec(id,n.c3,rec,prv,shop,transport,sight);
        }
        if(overlap(n.c4,rec))
        {
            usersInRec(id,n.c4,rec,prv,shop,transport,sight);
        }
    }
    else
    {
        NodeLeaf leaf = node as NodeLeaf;
        IDictionaryEnumerator enumerator = leaf.users.GetEnumerator();
        while(enumerator.MoveNext())
        {
            User user = (User)enumerator.Value;
            if(user.x >= rec.xMin && user.x <= rec.xMax &&
                user.y >= rec.yMin && user.y <= rec.yMax && user.id != id)
```

```
38                  {
                        if(db.isUserInFilter(user,prv,shop,transport,sight))
40                          inRec.Add(user);
                    }
42              }
            }
44  }
```

Listing 6.3: Metoderne der bruges ifm. område søgning

Linie 1 Input parametrene i metoden *findUsersInRec* angiver følgende. *id* er id på den bruger der har forårsaget kaldet, det gives for at undgå at objektet for denne bruger tilføjes til resultatet af søgningen. *rec* angiver det område som søgningen skal dække. De resterende parametre repræsenterer indstillingen af brugerens filter på klienten. Værdierne af dem afgører hvorvidt et bruger objekt inden for det angivne område skal medtages.

Linie 3 Variablen *rec* initialiseres. *rec* er returværdien til *findUsersInRec*. I forbindelse med de rekursive kald af *usersInRec* tilføjes de objekter, hvis position er indenfor det angivne område, til *rec*.

Linie 4 Metoden *usersInRec* kaldes med alle parametrene fra *findUsersInRec*, plus rod-knuden.

Linie 9-28 Der checkes om parameteren *node* er en instans af **Node**, dvs. om det er en ikke-blad knude. Hvis det er tilfældet så checkes der for hver af child-knuderne, om de dækker et område der overlapper det område der søges i. For hver child-knude, hvor det er tilfældet, kaldes *usersInRec* med den pågældende child-knude som parameter.

Linie 29-43 Hvis input parameteren *node* er en instans af **NodeLeaf**, og altså en blad-knude, så gennemløbes de bruger objekter der er placeret i *users* attributten på den pågældende blad-knude. For hvert objekt checkes der om objektes position er inden for det område der søges i. Hvis det er tilfældet så kaldes metoden *isUserInFilter* i klassen **DbAccess**. *isUserInFilter* checker om objektet er er medtaget i filtret. Der checkes om objektets type er medtaget, og ved butiksbrugere og seværdighedsbrugere hentes data om åbningstider i databasen, hvis det i filtret er angivet, at kun dem der er åbne på søgningstidspunktet skal medtages. Den hentede data bruges til at afgøre om dette er tilfældet. Hvis *isUserInFilter* returnerer sandt, så tilføjes objektet til arraylisten *inRec*, der bruges som returværdi til *findUsersInRec*.

Som det ses så benyttes databasen sjældent i de viste metoder, da den nødvendige bruger information er placeret i main-memory. Hvilken bruger information der skal placeres hvor, er et spørgsmål om hvor mange brugere der er tilknyttet

datastrukturen og hvor meget main-memory der er til rådighed. F.eks. kunne data om åbningstider placeres i main-memory, hvorved område søgningen aldrig skulle benytte databasen.

6.2 Kommunikation mellem web service og server

Det centrale element i kommunikationen mellem web servicen og serveren er et remote objekt, der er registreret på en remote server. Web servicen kan, via remote objektet, kalde metoder på serveren. Dette afsnit gennemgår kode på serveren, remote serveren, remote objektet og web servicen, der demonstrerer hvordan denne kommunikation foregår.

Listing 6.4 viser hvordan remote serveren opretter en TCP kanal og hvordan remote objektet registreres. Desuden vises hvordan serveren får fat i remote objektet.

```
//Kode på remote serveren
BinaryClientFormatterSinkProvider clientProvider = null;
BinaryServerFormatterSinkProvider serverProvider =
    new BinaryServerFormatterSinkProvider();
serverProvider.TypeFilterLevel =
    System.Runtime.Serialization.Formatters.TypeFilterLevel.Full;
IDictionary props = new Hashtable();
props["port"] = 6123;
props["typeFilterLevel"] = TypeFilterLevel.Full;
TcpChannel chan = new TcpChannel(props, clientProvider, serverProvider);
ChannelServices.RegisterChannel(chan);
RemotingConfiguration.RegisterWellKnownServiceType(typeof(RemotingObject.RemoteObject),
    "RemotingObject",WellKnownObjectMode.Singleton);
...
//Kode på serveren
RemotingObject.RemoteObject remoteObject;
remoteObject = (RemotingObject.RemoteObject)Activator.GetObject
    (typeof(RemotingObject.RemoteObject),"tcp://localhost:6123/RemotingObject");
```

Listing 6.4: Udsnit af kode til remote server og server

Linie 2-4 Der oprettes instanser af **BinaryClientFormatterSinkProvider** og **BinaryServerFormatterSinkProvider**. Disse instanser bruges til at angive at den TCP kanal der benyttes skal overføre binært formateret data.

Linie 5-6 Attributten *TypeFilterLevel* sættes til værdien "Full", hvilket betyder at alle typer automatisk bliver deserialiseret. Den anden mulighed er at sætten den til "Low", hvorved kun de mest basale typer automatisk bliver deserialiseret. Formålet ved at sætte den til "Low"er som en sikkerhed for at uvedkommende ikke får deserialiseret al kommunikation automatisk.

Linie 7-9 Der oprettes en instans af **IDictionary**, til at indeholde to egenskaber ved den benyttede TCP kanal. Den første egenskab er port nummer, der sættes til 6123, og den anden er *TypeFilterLevel*, der sættes til "Full".

Linie 9-10 Der oprettes en TCP kanal, der efterfølgende registreres.

Linie 12-13 Der registreres et remote objekt, der er placeret i namespacet RemotingObjekt og består af en instans af klassen **RemoteObject**. I forbindelse med registreringen angives der at remote objektet skal være en "Singleton", hvilket betyder at der kun oprettes én instans af remote objektet, og at denne instans eksisterer så længe remote serveren kører.

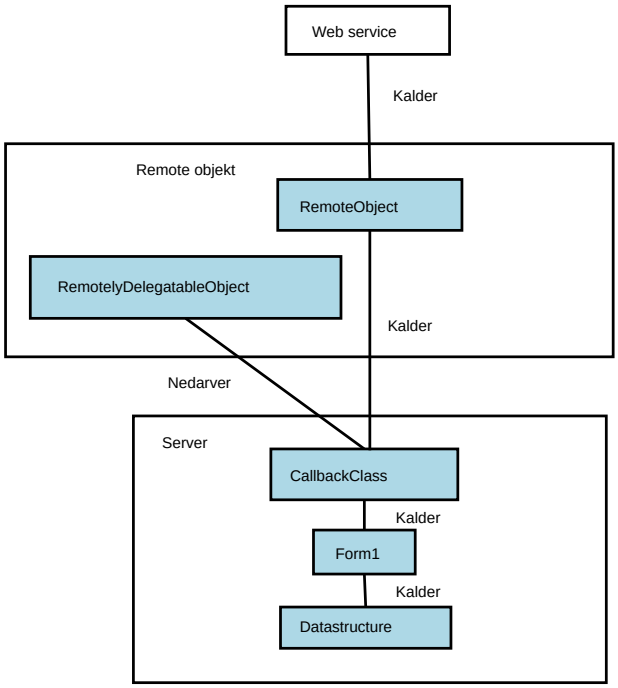
Linie 16-19 På serveren oprettes en instans af remote objekt klassen, der assignes til den remote objekt instans der er registreret på remote serveren.

Listing 6.5 viser en del af koden til klassen **CallbackClass** på serveren, der nedarver fra klassen **RemotelyDelegatableObject**, på remote objektet. Denne nedarvning er nødvendig for at remote objektet, via delegates, kan kalde metoder på serveren. Metoderne i **CallbackClass** er alle, via en event, tilknyttet en metode i klassen **Form1** på serveren. Det betyder at når web servicen kalder en metode i remote objektet, så kaldes en metode i **CallbackClass**, hvilket resulterer i at der kaldes en metode i klassen **Form1**. **Form1** indeholder en reference til datastrukturen, og kalder så den metode i datastrukturen der svarer til den metode på remote objektet der blev kaldt af web servicen. Figur 6.2 giver et overblik over denne kommunikation.

```
public class CallbackClass : RemotelyDelegatableObject
{
    public delegate void MoveUserDel(MoveUserArgs moveUserArgs);
    public event MoveUserDel MoveUserEvent;
    public delegate User[] GetUsersInRecDel(GetUsersInRecArgs getUsersInRecArgs);
    public event GetUsersInRecDel GetUsersInRecEvent;
    public delegate User GetUserDel(GetUserArgs getUserArgs);
    public event GetUserDel GetUserEvent;
    public delegate Product[] GetProductsDel(GetProductsArgs getProductsArgs);
    public event GetProductsDel GetProductsEvent;
    public delegate User QueryUserDel(QueryArgs queryArgs);
    public event QueryUserDel QueryUserEvent;
    public delegate ProdQuery[] QueryProdDel(QueryProdArgs queryProdArgs);
    public event QueryProdDel QueryProdEvent;
    public delegate void MessageDel(MessageArgs messageArgs);
    public event MessageDel MessageEvent;

    public CallbackClass () {}

    protected override void InternalMoveUserCallback (MoveUserArgs moveUserArgs)
    {
        if(MoveUserEvent != null)
            MoveUserEvent(moveUserArgs);
```



Figur 6.2: Kommunikation mellem web service og server

```
24      }
        protected override User[] InternalGetUsersInRecCallback
26                                      (GetUsersInRecArgs getUsersInRecArgs)
        {
28          if(GetUsersInRecEvent != null)
                return GetUsersInRecEvent(getUsersInRecArgs);
30          else
            {
32              return null;
            }
34      }
        (...Resterende metoder forkortet væk...)
36  }
```

Listing 6.5: Kode til **CallbackClass**

Linie 3-16 Der angives en delegate og en event for hver af de metoder der kaldes som resultat af kald til remote objektet. Disse events abonneres på i klassen **Form1**, hvorved en metode i **Form1** bliver kaldt når en metode i **CallbackClass** bliver kaldt.

Linie 20-24 Metoden *InternalMoveUserCallback* kaldes når metoden *MoveUserCallback* i klassen **RemotelyDelegatableObject** på remote objektet kaldes. Hvilket sker når metoden **MoveUser** i **RemoteObject** kaldes. *InternalMoveU-*

serCallback overskriver den tilsvarende abstrakte metode i **RemotelyDelegatableObject** og kalder de metoder der abonnerer på *MoveUserEvent*.

Linie 25-34 Metoden *InternalGetUsersInRecCallback* fungerer på samme måde som *InternalMoveUserCallback*. Den er medtaget her som et eksempel på at kald foretaget via events godt kan have returværdier. Det er dog kun hensigtsmæssigt hvis der kun er én metode der abonnerer på eventen.

I Listing 6.6 ses et eksempel på hvordan en delegate registreres på remote objektet, og hvordan en metode i **Form1** sættes til at abonnere på en metode i **CallbackClass**.

```
callback = new CallbackClass() ;
remoteObject.addMoveUserDel(new RemoteObject.moveUserDel
                            (callback.MoveUserCallback),1,rootRec);
callback.MoveUserEvent += new Server.CallbackClass.MoveUserDel
                            (callback_MoveUserEvent);
```

Listing 6.6: Kode på serveren

Linie 2-3 Ved at kalde metoden *addMoveUserDel* på remote object instansen på serveren, sættes metoden *MoveUserCallback* i **Form1**s instans af **CallbackClass** til at abonnere på en metode på remote objektet.

4-5 Metoden *callback_MoveUserEvent* i **Form1** sættes til at abonnere på metoden *InternalMoveUserCallback*, via eventen *MoveUserEvent*.

Når web servicen kaldes af klienten, så kaldes en metode på remote objektet. Listing 6.10 viser hvordan dette sker i metoden *getUser*.

```
[WebMethod]
[XmlInclude(typeof(PrivateUser)),XmlInclude(typeof(ShopUser)),
XmlInclude(typeof(TransportUser)),XmlInclude(typeof(SightUser))]
public User GetUser(int id) {
    User user = remoteObject.GetUser(id);
    if(user is PrivateUser)
    {
        return (PrivateUser)user;
    }
    if(user is ShopUser)
    {
        return (ShopUser)user;
    }
    if(user is TransportUser)
    {
        return (TransportUser)user;
    }
    if(user is SightUser)
```

```
        {
20          return (SightUser)user;
        }
22      return null;
    }
```

Listing 6.7: Kode til *getUser* på web servicen

Linie 2-3 Web service Attributtet *XmlInclude* benyttes til at angive at returværdien af metoden *getUser* kan være alle fire brugertyper.

Linie 5 Metoden *GetUser* kaldes på remote objektet, og returnerer en instans af **User**.

Linie 6-21 Der undersøges hvilke brugertype kaldet til *GetUser* returnerede, og returværdien af *GetUser* castes til den pågældende type. Grunden til at det er nødvendig at caste er, at web servicen skal vide hvilken type den skal serialisere.

6.3 Distribution

Systemet distribueres ved at klient instanser kan tilgå forskellige web service instanser. De forskellige web service instanser tilgår den samme remote objekt instans, der så kan tilgå forskellige server instanser. Det at klient instanser tilgår forskellige web service instanser kræver blot at der genereres en proxy for den web service instans som en klient er tilknyttet, og kræver derfor ikke nogen yderligere beskrivelse. Det centrale i implementationen af distribueringen er derfor hvordan remote objektet finder ud af, hvilken eller hvilke server instanser der skal kaldes.

Remote objektet får kendskab til de forskellige server instanser, ved at hver instans registreres på remote objektet. Dette gøres ved at hver server instans kalder en metode på remote objektet, for hver metode på remote objektet der skal tilknyttes en metode på serveren. Disse metoder tager en delegate knyttet til en metode, et server id og et **BoundingRec** objekt som input. Tilknytningen af metoder sker ved at en metode på serveren abonnerer på en delegate på remote objektet, server id'et benyttes til at identificerer hver server instans og **BoundingRec** objektet benyttes til at angive det område som hver server instans dækker. I Listing 6.8 ses et eksempel på hvordan en server instans registrerer metoden *MoveUserCallback* på remote objektet.

```
    //kode på serveren
2   callback = new CallbackClass() ;
    remoteObject.addMoveUserDel(new RemoteObject.moveUserDel
4                               (callback.MoveUserCallback),1,rootRec);
```

```
//kode på remote objektet
public void addMoveUserDel(moveUserDel del,int server,BoundingRec br)
{
    moveUserHash.Add(server,new object[]{del,br});
    moveUserEvent += del;
}
```

Listing 6.8: Eksempel på registrering af server på remote objekt

Linie 3-4 På serveren kaldes metoden *addMoveUserDel* på serverens remote objekt instans. Den tager en instans af delegaten *moveUserDel* tilknyttet metoden *MoveUserCallback*, et server id, i form af en int, og en instans af **BoundingRec** som input. Den **BoundingRec** instans der tages som input er den instans der er tilknyttet rod-knuden i serverens datastruktur.

Linie 6-10 I metoden *addMoveUserDel* tilføjes input parametrene til en hashtabel, med server id'et som nøgle. Desuden tilføjes delegate instansen til eventen *moveUserEvent*. Input parametrene skal efter planen tilføjes til en datastruktur, der ligner den datastruktur der benyttes til indeksering af bruger objekter, men hvor der indekseres på **BoundingRec** objekter i stedet for bruger objektet. Dette er ikke blevet implementeret på nuværende tidspunkt på grund af projektets deadline.

Når server instansen er registreret, kan den metoden der er tilknyttet delegaten kaldes, når et eller flere bruger objekter på den pågældende server instans er involveret i en positionsopdatering. I listing 6.9 ses koden til metoden *MoveUser*, der er den metode på remote objektet der bliver kaldt når der skal foretages en positionsopdatering af et bruger objekt. Som input parametre tages et bruger objekt id, de parametre der bruges til positionsopdateringen, samt et server id. Metoden kalder så den delegate der er tilknyttet server instanser med det givne input id, og i tilfælde af at bruger objektet skal flyttes over på en anden server instans findes og kaldes denne instans. Som returværdi gives en int, der angiver hvilken server instans bruger objektet er tilknyttet efter positionsopdaterigen.

```
public int MoveUser(int id,int x,int y,string nearby,string password,int server)
{
    if(moveUserEvent == null)
    {
        Console.WriteLine("Den er null");
        return -1;
    }
    else
    {
        object[] obj =(object[]) moveUserHash[server];
        moveUserDel moveDel = (moveUserDel)obj[0];
        BoundingRec rec = (BoundingRec)obj[1];
        MoveUserArgs arg = new MoveUserArgs(id,x,y,nearby,password);
```

```
14          if(rec.xMin <= x && rec.xMax >= x &&
                rec.yMin <= y && rec.yMax >= y)
16          {
                moveDel(arg);
18              return server;
            }
20          else
            {
22              moveDel(arg);
                int i = 0;
24              IDictionaryEnumerator enumerator = moveUserHash.GetEnumerator();
                IEnumerator enume = moveUserHash.Keys.GetEnumerator();
26              while(enumerator.MoveNext())
                {
28                  object[] ob = (object[])enumerator.Value;
                    enume.MoveNext();
30                  int key = (int)enume.Current;
                    moveUserDel mvDel = (moveUserDel)ob[0];
32                  BoundingRec rc = (BoundingRec)ob[1];
                    if(rc.xMin <= x && rc.xMax >= x &&
34                      rc.yMin <= y && rc.yMax >= y)
                    {
36                      mvDel(arg);
                        return key;
38                  }
                    i++;
40              }
            }
42      }
        return -1;
44  }
```

Listing 6.9: Kode til *MoveUser*

Linie 9-12 Den server instans der indeholder det bruger objekt der skal opdateres findes ved hjælp af input parameteren *id*. Der oprettes et **MoveUserArgs** objekt *arg*, med de input parametre der bruges til positionsopdateringen.

Linie 13-18 Der undersøges om bruger objektets nye position er inden for den server instans den på nuværende tidspunkt er tilknyttet. Hvis den er det så kaldes den metode på server instansen der er tilknyttet delegaten *moveDel*, med variablen *arg* som input parameter, hvorefter det nuværende server id returneres.

Linie 19-37 Hvis positionsopdateringen gør at bruger objektet skal flyttes til en ny server instans, så findes denne instans og den delegate der er tilknyttet denne instans benyttes til at kalde positionsopdaterings metoden på denne instans, hvorefter det nye server id returneres. Her benyttes den hashtabel som i listing 6.9 blev brugt til at registrerer server instanser til metoden *MoveUser*. Som tidligere nævnt skal denne hashtabel erstattes af en anden datastruktur.

GetUsersInRec er endnu et eksempel på en metode hvor remote objektet koordinerer server instanser. Den finder alle brugere inden for et søge område, som kan dække en eller flere server instanser. Dette gøres ved at kalde delegaten for de server instanser som overlapper søge området, og så opbygge en liste af brugere udfra de brugere der gives som returværdi til hvert delegate kald. I listing 6.10 ses koden til *GetUsersInRec*.

```
public User[] GetUsersInRec(int id,BoundingRec rec,int prv,int
shop,int transport,int sight) {
    ArrayList list = new ArrayList();
    if(getUsersInRecEvent == null)
    {
        return null;
    }
    else
    {
        GetUsersInRecArgs arg = new GetUsersInRecArgs(id,rec,prv,shop,transport,sight);
        IDictionaryEnumerator enumerator = getUsersInRecHash.GetEnumerator();
        while(enumerator.MoveNext())
        {
            ArrayList l = new ArrayList();
            object[] ob = (object[])enumerator.Value;
            getUsersInRecDel recDel = (getUsersInRecDel)ob[0];
            BoundingRec rc = (BoundingRec)ob[1];
            if(overlap(rc,rec))
            {
                list.AddRange(recDel(arg));
            }
        }
        User[] users = new User[list.Count];
        int i = 0;
        foreach(User user in list)
        {
            users[i] = user;
            i++;
        }
        return users;
    }
}
```

Listing 6.10: Kode til *GetUsersInRec*

Linie 11 Hashtabellen *getUsersInRecHash* indeholder oplysninger om de server instanser der abonnerer på *GetUsersInRec*. Der skabes en enumerator udfra denne hashtabel.

Linie 12 - 22 Indholdet af enumeratoren gennemløbes. For hver server instans tjekkes om instansen område overlapper søgeområdet. Hvis det er tilfældet kaldes delegaten for den pågældende instans og returværdien for kaldet tilføjes returværdien for *GetUsersInRec*.

Linie 23 - 29 Den arraylist der indeholder returværdien omdannes til et array af **User** objekter. Dette gøre fordi indholdet af en arraylist sendt via en web service ikke kan castes til andet end **object**.

Del III

Test og evaluering

Kapitel 7

Test og evaluering

Indholdsfortegnelse

I dette kapitel vil der blive foretaget test af datastrukturen, ved direkte at kalde metoder på datastrukturen, fra serveren. Der vil desuden blive foretaget test af de forskellige dele af systemet, dvs. remote objektet, web servicen og klienten. På baggrund af testene, angivelse af kompleksiteten, samt en sammenligning med LUR-træet og et ikke-statisk PR-quadtræ, vil der blive foretaget en evaluering af datastrukturen og det samlede system.

De forskellige tests vil blive udført på en maskine med følgende konfiguration.

CPU	AMD K6 800 MHZ
RAM	256 MB
OS	Windows XP Professionel SP2
Web Server	IIS 5.1
.NET Framework	v1.1
Internetforbindelse	512/128

Højde	Noder	MB	bytes pr knude
4	341	3	8797
5	1365	4	2930
6	5461	5	915
7	21845	8	366
8	87381	21	240
9	349525	71	203

Tabel 7.1: Hukommelsesforbrug for SPRQ-træet

7.1 Datastruktur

Hovedmålet med SPRQ-træet er, at det skal være hurtig til både positionsopdateringer og område forespørgsler. Hvilket opnås ved at det er gjort statisk, og at det anvendes med en høj opløsning. I dette afsnit vil der blive undersøgt hvor meget RAM SPRQ-træet bruger, samt hastigheden på positionsopdateringer og område forespørgsler. Desuden vil der blive foretaget en sammenligning med LUR-træet og et ikke-statisk PR-Quadtræ, på baggrund af en angivelse af SPRQ-træets kompleksitet. På baggrund af dette vil der blive foretaget en evaluering af dets anvendelighed i forbindelse med et DOPRI system.

7.1.1 Test

Der er to tal der afgør hvor meget RAM datastrukturen bruger, det er bytes per knude og bytes per bruger objekt. Da hver ikke-blad knude har fire child-knuder, så kræver det $4^{h+1} * b$ RAM (h = højde og b = bytes per bruger) at forøge højden af træet med 1. I tabel 7.1 ses en oversigt over hvor mange bytes per knude der bruges ved forskellige højder. Grunden til at der i denne og efterfølgende tests kun er brugt træer med en højde op til 9, er den tilgængelige mængde RAM på testmaskinen. Antal MB er fundet ved at trække det antal MB som server processen bruger, når der ikke er genereret et træ, fra det antal MB den bruger, når træet er genereret. Som det ses i tabellen så falder antal bytes per knude når højden stiger. Dette hænger sammen med at der bruges RAM til at oprette datastruktur klassen, og herunder også database tilgangs klassen. Knuderne består af en instans af **BoundingRec** klassen, referencer til parent knuden, og for ikke-blad knuder også referencer til de fire child-knuder, samt en 32-bit int, der angiver højde. Desuden har blad-knuder en instans af en hashtabel, der bruges til at indeholde bruger objekter. **BoundingRec** klassen består af fire 32-bit int der angiver start og slut værdi på begge akser, for det område en instans dækker.

Bruger objekter består af fire 32-bit int og tre strings, der er opbygget af 16-bits

Højde	1.1 mil	900k	700k	500k	300k
9	130 ms (430 100)	100 ms (380 90)	90 ms (150 50)	60 ms (50 200)	40 ms (40 50)
8	80 ms (120 50)	50 ms (60 40)	50 ms (60 50)	50 ms (60 50)	30 ms (40 30)
7	60 ms (140 40)	50 ms (110 40)	50 ms (110 40)	50 ms (110 40)	50 ms (80 40)

Tabel 7.2: Positionsopdatering

Område	900k	700k	500k	300k
200*200 m	50 ms	40 ms	30 ms	20 ms
500*500 m	120 ms	110 ms	100 ms	50 ms
1000*1000 m	610 ms	460 ms	340 ms	190 ms
2000*2000 m	2880 ms	2250 ms	2100 ms	1060 ms

Tabel 7.3: Område forespørgsel "Find alle brugere i område"i træ med højde 9

tegn. I en test hvor de tre strings bestod af ialt 24 tegn, brugte 1100000 bruger objekter 68 MB RAM, hvilket giver cirka 62 byte per bruger objekt. Et træ der inddeler hele Danmarks areal i områder på 50*50 meter, og har et bruger objekt for hver dansker, bruger cirka 5.3 GB RAM. Alle testene bliver foretaget ved at køre den testede funktion et stort antal gange. Derved viser resultatet et gennemsnit for den testede funktion. Desuden kan **DateTime** klassen i .NET kun måle ned til 10 millisekunder i .NET og 1 sekund i .NET CF, så der skal et vist antal kørsler af hurtige funktioner til, før det kan måles.

For at teste hastigheden på positionsopdateringer køres opdateringsmetoden *MoveUser* 10000 gange. I testen udvælges tilfældige brugere, som bliver flyttet et tilfældig antal meter mellem minus 10 og plus 10 på både x og y akserne. I forbindelse med opdateringerne vil nogle objekter skifte blad-knude og nogle vil blive fjernet fra datastrukturen, fordi deres position er uden for det område systemet dækker. Testen foretages med forskellige antal bruger objekter og med forskellige højder på træet. Bruger objekterne er jævnt fordelt over hele området, og der er foretaget 10 tests for hver måling. I parantes er angivet maksimum og minimum målingerne af de 10 tests. Resultatet kan ses i tabel 7.2.

Område forespørgsler er blevet testet ved at lave 1000 forespørgsler på forskellige område størrelser og med forskellig antal bruger objekter. I tabel 7.3 ses oversigten over område forespørgslen "Find alle brugere i område"i et træ med højde 9.

For at vise forskellen mellem at have data i main-memory eller i databasen er metoden *getUserFromDB*, der opretter et bruger objekt udfra databasen, blevet testet. Testen viste at 100 kald tager cirka 120 ms. Det betyder at de 1000 gange forespørgslen "Find alle brugere i område"blev kørt i tabel 7.3 vil tage 1,2 sekunder,

foruden den tid det tager at finde objekterne i træet, hvis databasen skulle bruges. Det viser at det er hensigtsmæssig, at kunne udføre de oftest udførte område forespørgsler uden at benytte databasen.

Område forespørgsel har en worst-case kompleksitet på O($4^0+4^1+...+4^N$), og en best-case kompleksitet på O(N), hvor N er højden af træet. Kompleksiteten på område forespørgsler afhænger af, hvor stort søge området er, i forhold til det samlede område som træet dækker. Positionsopdatering har en worst-case kompleksitet på 2N (O(N)), og en best-case kompleksitet på O(1). Worst-case kompleksiteten forekommer, når en bruger har bevæget sig over i en anden fjerdedel af træet, hvorved der rekursivt bliver tjekket parent knuder indtil rod-knuden nås. Fra rod-knuden skal træet så traverseres ned til en blad-knude. Langt de fleste opdateringer har best-case kompleksiteten, da denne forekommer så længe en bruger bevæger sig inden for samme blad-knudes område. Hvorved bruger objektet blot findes i hashtabellen, udfra et givent bruger id, og assignes nye x og y værdier.

LUR-træet har samme worst-case og best-case kompleksitet på område forespørgsler, som SPRQ-træet. Forudsat at det benyttes i et DOPRI system, og har passende minimums og maksimums grænser for hvor mange objekter/knuder der kan være tilknyttet en knude. Ved best-case positionsopdateringer foretages det samme som ved best-case for SPRQ-træet, plus en eventuel tilpasning af blad-knudens MBR. Ved positionsopdateringer, hvor det er nødvendig at finde en ny blad-knude til det opdaterede bruger objekt, skal MBR for den nuværende blad-knude tilpasses, hvorefter en af følgende tre situationer kan opstå. Situation 1, er det tilfælde hvor hverken minimums eller maksimums grænsen på antal objekter for hverken det flyttede objekts gamle eller nye blad-knude overskrides. Det medfører en eventuel justering af MBR i den gamle og den nye blad-knude. Situation 2 er de tilfælde, hvor flytningen resulterer i at minimums grænsen i objektets gamle blad-knude overskrides. Hvis dette er tilfældet så skal blad-knuden fjernes. Denne process skal propageres op gennem træet, så længe fjernelsen af en knude resulterer i at minimums grænsen overskrides (for ikke-child knuder er det grænsen for antal child-knuder og ikke bruger objekter). Tilsidst skal både de fjernede objekter og de fjernede ikke-blad knuder indsættes igen. Situation 3 er det tilfælde hvor flytningen resulterer i, at maksimums grænsen overskrides i den blad-knude objektet flyttes over i. Hvis dette sker så skal denne blad-knude splittes i to. De to blad-knuder der opstår pga. splitningen skal tilføjes til parent knuden på den splittede knude, hvilket igen kan resultere i en splitning af parent knuden. Denne process propageres op gennem træet indtil der findes en knude med en ledig plads, eller indtil rod knuden nås. Hvis der ikke er plads i rod-knuden så splittes rod knuden, og der tilføjes en ny rod knude, som får den splittede rod knude som child-knuder. Både i forbindelse med fjernelse og splitning af knuder skal MBR for de involverede knuder justeres. I forbindelse med fjerning af knuder, kan genindsættelsen af objekterne fra de fjernede knuder resulterer i en eller flere splitninger.

Opdelingen af en knudes objekter/child-knuder i forbindelse med splitning, har lineær kompleksitet [56] i antallet af objekter/child-knuder på $max+1$ (der er ikke benyttet O-notation da konstanter hjælper til med at vise hvordan kompleksiteten er fundet), og justering af MBRs har en worst-case kompleksitet på max, hvor max er maksimums grænsen for objekter/child-knuder i en knude. Splitningsprocessen har en worst-case kompleksitet på $N * (max + 1)$, hvor N er højden af træet. Fjernelse har en worst-case kompleksitet på $N + (N * (min - 1)) * (N * (max + 1))$, hvor min er minimums grænsen for objekter i en knude. Worst-case situationen opstår når der fjernes knuder hele vejen til roden, som derefter genindsættes med det resultat, at hver genindsættelse forårsager en splitning.

En variant af PR-quadtræet, hvor antallet af knuder ikke er statisk, hvor der skal være mindst ét objekt i en knude og hvor der er en maksimumsgrænse på antallet af knuder, men som benytter samme algoritmer som SPRQ-træet, vil have samme best-case kompleksitet på positionsopdateringer og område forespørgsler. Den eneste forskel er at der skal tilføjes og fjernes knuder hvis der pga. en positionsopdatering, er for få eller for mange objekter i en knude. Hvis de objekter der skal indekseres er jævnt spredt over områder der dækkes, så er der ingen grund til at fjerne og tilføje knuder. Fordelen ved at fjerne og tilføje knuder opstår først i den situation hvor der forekommer et højt antal objekter i en blad-knude, idet en yderligere opdeling af en sådan knude vil forøge hastigheden på område forespørgsler.

7.1.2 Evaluering

Et main-memory baseret SPRQ-træ, med de anvendte algoritmer, og med en direkte link til objekter i form af en hashtabel, giver et minimalt antal operationer ved positionsopdateringer. Ved område forespørgsler, benyttes en traditionel top-down traversering af træet til at udelukke de del-træer som ligger uden for søge området. At træet er statisk, begrundes med at brugere af systemet er jævnt fordelt over det område som systemet dækker. Denne antagelse kombineres med et stort antal knuder i træet, så hver blad-knude kun dækker et lille område. Derved er det ikke nødvendig at tjekke et stort antal objekter for at finde dem der ligger inden for søge området. Testene viser at datastrukturen ikke bliver flaskehalsen i systemet, idet den kan håndterer både positionsopdateringer og område forespørgsler med en hastighed der gør at én instans af datastrukturen vil kunne betjene et meget højt antal brugere med en acceptabel hastighed.

Sammenligningen med LUR-træet og en ikke-statisk variant af PR-quadtræet viser at SPRQ træet har en lavere kompleksitet end LUR-træet, og udfører færre operationer end det ikke-statiske PR-quadtræ, hvis begge træer indgår i et DOPRI system.

Opdelingen af et objekts data i en main-memory del og en database del er blevet

undersøgt ved at oprette objekter udfra databasen i forbindelse med den testede område forespørgsel. Testen viste at ved et søge område på 200*200 meter, med 900000 brugere og en træ højde på 9, vil det forøge tiden på en forespørgsel med en faktor 12, at benytte databasen. Hvor stor en del af en brugeres data der kan placeres i main-memory afhænger af mængden af data og mængden af RAM. I systemet kan en butiksbruger potentielt have en anseelig mængde data, idet navn og pris for de varer butikken sælger indgår i dataen. Som nævnt i indledningen kan 32-bit og 64-bit processorer allokerer henholdsvis 2 og 4 GB til én applikation. Mængden af RAM afhænger derfor af hvor mange instanser af datastrukturen der benyttes.

7.2 Systemet

Der vil blive undersøgt om systemet opfylder følgende to krav, det skal fungerer med en acceptabel hastighed og det skal være skalerbart. Måden systemet opfylder disse krav er ved at være distribueret og ved at have en hurtig datastruktur i main-memory. I dette afsnit vil der blive testet i hvor høj grad systemet opfylder kravene, og der vil blive givet en evaluering af systemet.

7.2.1 Test

Alle test er blevet udført på én maskine, hvilket betyder at netværksforbindelse ikke indgår i de tests hvor der indgår flere instanser af serveren eller web servicen. For at teste hastigheden på systemet bliver web servicen kaldt fra en testapplikation på samme maskine som web servicen, samt fra klienten. Kaldet fra testapplikationen giver en angivelse af systemets hastighed, uafhængig af en GPRS forbindelse, som kan varierer alt efter hvor belastet netværket er. Der vil ikke blive udført tests med forskellige antal brugere og træ størrelser, da dette er testet i 7.1.

En test med én server instans og kald fra testapplikation til positionsopdaterings metoden på én web service instans gav et gennemsnit på 15.5 ms per opdatering. Den samme test udført direkte på remote objektet, udenom web servicen, gav et gennemsnit på 5.5 ms per opdatering.

En test med to server instanser og kald fra en testapplikation til en web service instans, hvor det samme bruger objekt, grundet en positionsopdatering, blev flyttet frem og tilbage mellem de to server instanser, gav et gennemsnit på 47 ms per flytning. En sådan flytning involverer et kald fra remote objektet til den server instans, hvor objektet befinder sig før flytningen, og et kald til den server instans hvor objektet skal flyttes hen. På objektets nuværende server instans findes objektet via hashtabellen, der fungerer som direkte link til objekterne. Derefter fjernes den

Brugere	tid
10	4 sek
20	4-5 sek
40	5-6 sek
60	6-7 sek
100	7-8 sek
150	11 sek

Tabel 7.4: Hentning af brugere via GPRS

fra både hashtabellen og den knude den er tilknyttet. På den server instans den flyttes til skal objektet genskabes udfra databasen, og tilføjes til hashtabellen og den knude som dækker det område den flyttes til.

Hastigheden på område forespørgsler er testet ved at teste forespørgslen "Find alle brugere i område". En test hvor hele søge området var inden for den samme server instans, søge området var på 500*500 meter og træet havde højden 8, gav et gennemsnit på 20 ms per søgning. En test med et søge område på 1000*1000 meter og med en halvdel på hver af to server instanser, begge med træer med højden 7, gav et gennemsnit på 35 ms per søgning. En test med samme setup, men hvor to testapplikationer kører 100 forespørgsler på hver deres web service instans, gav et resultat på cirka 70 ms per forespørgsel. Det er ikke så overraskende, da det hele skal igennem den samme processor. Men det viser, at der ikke opstår problemer ved at have to web service instanser til at tilgå remote objektet.

De resterende tests er udført på kald på klienten. Det betyder at resultatet er afhængig af trafikken på GPRS forbindelsen. Testene er udført mellem kl. 22-24, hvor trafikken må formodes at være lav. Den første test er en test af hastigheden på positionsopdateringer, hvor der er benyttet et træ med højden 7, indeholdende 10 brugere. Resultatet viste at det tager 2-3 sekunder for omkring 80 procent af målingerne, den længste måling tog 7 sekunder.

Den næste måling på klienten er en måling af område forespørgslen "Find alle brugere i område". Den blev udført på et træ med højden 7, og med et varierende antal brugere. Resultatet af den kan ses i Listing 7.4.

Den sidste måling på klienten er en måling af tiden det tager at kalde metoden *RenderService* i MapPoint webServices. *RenderService* bruges til at hente et kort med brugere påsat. Resultatet kan ses i listing 7.5. Grunden til at der kun er foretaget målinger op til 100 brugere er, at MapPoint kun kan klare at påsætte 100 POI på et kort.

Brugere	tid
10	14-15 sek
20	18-20 sek
40	29-31 sek
60	34-39 sek
100	46-50 sek

Tabel 7.5: Hentning af kort med brugere påsat via MapPoint

7.2.2 Evaluering

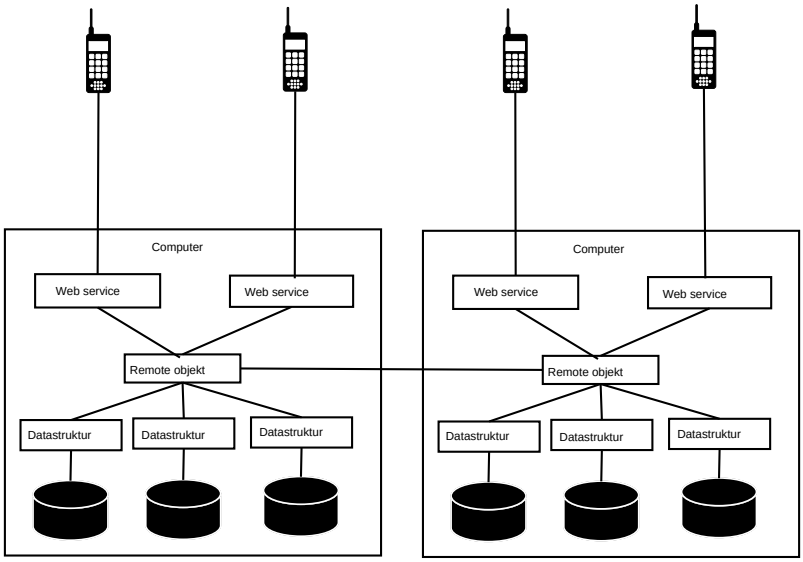
Testene viste at systemet kan udføre positionsopdateringer og område forespørgsler med en acceptabel hastighed. Samtidig viste de at det er muligt at benytte flere instanser af web servicen og serveren. Testene viste desuden, at dataoverførsel via GPRS, er den del af et kald fra klienten, der tager længst tid. Dernæst kommer kald til web servicen, og hurtigst er kald til remote objektet. Testene af klienten viste at det er en god ide at lade kaldene fra klienten ske i tråde, da hastigheden på kaldene måles i adskillige sekunder og klienten er ubrugelig mens et kald bliver udført. Desuden er det ikke hensigtsmæssigt at klienten sender en positionsopdatering hver gang der foretages en opdaterings event fra GPS.NET. Derfor er der blevet sat en minimums grænse på 5 meter, som positionen skal have ændret sig siden sidste opdateringskald til web servicen, før der foretages et nyt opdateringskald.

Det har desværre ikke været muligt at teste kald der involverer to eller flere maskiner. Men hvis de involverede maskiner kører i et lokal netværk, hvor kun systemet køres, så vil kommunikationen mellem maskinerne kunne måles i millisekunder. Det har heller ikke været muligt, at teste med et stort antal klienter.

Da det tager 15.5 ms per positionsopdatering, hvis opdateringen kun involverer én server instans, vil én web service instans kunne afvikle cirka 64 positionsopdateringer i sekundet. En person der går rundt med en klient skal have opdateret sin position cirka hver 3 sekund. Hvilket betyder at én instans af web servicen kan klare cirka 300 klienter der bevæger sig. Hvor mange server instanser der skal være på en computer afhænger af mængden af RAM. Der kan vælges mellem to strategier, få instanser af træer med høj opløsning, eller mange instanser med lav opløsning. Testene antyder at få instanser med høj opløsning er at foretrække da koordination mellem instanser tager væsentlig længere tid end metoder udført på kun én instans. Det kunne også være hensigtsmæssigt at benytte asynkrone kald fra web service og remote objektet. Dette kunne øge mængden af kald der kan afvikles med en acceptabel hastighed.

For at undgå at databasen eller remote objektet bliver en flaskehals i systemet kan disse eventuelt også distribueres. For at kunne distribuere databasen kræves det

at al data for objekter der kan skifte server instans er placeret i main-memory. Objekter kan så skifte server instans ved at remote objektet sender objektet til den nye instans, i stedet for at genskabe den udfra databasen, som er tilfældet med det nuværende system. Remote objektet kan distribueres ved at remote objekt instanser registrerer sig ved hinanden på samme måde som server instanser registrerer sig på remote objektet. Figur 7.1 viser det alternative forslag til distribution, hvor også remote objektet og databasen er distribueret.



Figur 7.1: Alternativ forslag til distribution

Del IV

Afrunding

Kapitel 8

Konklusion

I dette projekt er der blevet undersøgt om det er hensigtsmæssigt at benytte main-memory, frem for disk, til at indeholde en datastruktur, i form af SPRQ-træet, til indeksering af to dimensionelle spatiale data objekter, i forbindelse med et DOPRI system. Derudover er der blevet undersøgt hvorvidt, og i hvilken grad, det er muligt og hensigtsmæssigt at have en del af et objekts data placeret i main-memory. Desuden er der blevet undersøgt hvilken teknologi der er tilgængelig til at opbygge et distribueret DOPRI system, herunder positionerings -og programmeringsteknologi.

SPRQ-træet er en datastruktur, der opdeler et område i delområder, og udviklet med det mål at kunne udføre positionsopdateringer og område forespørgsler med en høj hastighed. SPRQ-træet er et statisk PR-Quadtræ, og den benyttes sammen med en hashtabel der fungerer som direkte link til de objekter der indekseres. SPRQ-træet er gjort statisk udfra den antagelse, at hvis det område der dækkes, inddeles i tilpas små delområder, så vil der ikke forsamle sig en så stor mængde objekter i et delområde, at det vil gå ud over hastigheden på område forespørgsler. Denne antagelse er selvfølgelig afhængig af hvilken type objekter der indekseres, men den holder hvis objekterne repræsenterer personer, bygninger, biler og lignende. Samtidig vil positionsopdateringer kunne udføres med en lav gennemsnitlig kompleksitet, da de ikke forårsager ændringer i træet. Denne inddelingen af et område i små delområder kræver at SPRQ-træet består af et højt antal knuder, så hver blad-knude kun dækker over et lille område. For at kunne have et SPRQ-træ med et stort antal knuder i main-memory skal der bruges en store mængde RAM. Da grænsen for hvor meget RAM der kan allokeres til én applikation er på henholdsvis 2 og 4 GB for 32 og 64-bit processorer, er det nødvendig at SPRQ-træet indgår i et distribueret system.

I forbindelse med projektet er der udviklet et distribueret DOPRI system, der be-

nytter SPRQ-træet. Distribueringen af systemet er baseret på .NET remoting og web services. .NET remoting bruges internt i systemet, til at have et .NET remoting objekt til at videresende og koordinerer kald fra web servicen til serveren. Web services bruges som middleware mellem en klient og systemet. Hvilket har den fordel at klienten ikke behøves at udvikles på, eller køre på en bestemt platform. Systemet er gjort distribueret således, at der kan benyttes flere web service og server instanser. Derved kan den samlede mængde main-memory som indgår i systemet have en størrelse som gør, at SPRQ-træet for hver server instans kan have en højde der gør, at hver blad-knude dækker et tilpas lille delområde. Samtidig kan størrelsen af de delområder hver server instans dækker tilpasses efter brugertætheden i området. Desuden er der i afsnit 7.2.2 givet et forslag til, hvilke modifikationer af systemet der skal til for at også databasen og remote objektet kan distribueres.

På baggrund af en undersøgelse af andre datastrukturer, blev der udvalgt to datastrukturer, som er udviklet med samme mål som SPRQ-træet. Disse to datastrukturer, LUR-træet og en ikke-statisk PR-Quadtræ, er indgået i en kompleksitets sammenligning med SPRQ-træet. Det viste sig at SPRQ-træet har en lavere gennemsnitlig kompleksitet end LUR-træet, på postionsopdateringer. Ved område forespørgsler afhænger kompleksiteten af LUR-træet, af maksimums og minimums grænsen for antal child-knuder/objekter. Men ved et stort antal jævnt fordelte objekter, er kompleksiteten for område forespørgsler ens for LUR-træet og SPRQ-træet. Det ikke-statiske PR-Quadtræ, har samme kompleksitet som SPRQ-træet ved både område forespørgsler og positionsopdateringer, igen forudsat et højt antal jævnt fordelte objekter, samt at det ikke-statiske PR-Quadtræ benyttes sammen med en hashtabel med direkte link til bruger objekter, samt benytter de samme algoritmer til positionsopdatering og område forespørgsler, som SPRQ-træet. Forskellen mellem de to er, at det ikke statiske PR-Quadtræ fjerner og tilføjer knuder i takt med at der fjernes og tilføjes objekter. Dette vil ved et stort antal jævnt fordelte objekter ske sjældent, og vil derfor ikke have nogen nævneværdig indflydelse på hastigheden af hverken positionsopdateringer eller område forespørgsler.

Udover at have en datastruktur i main-memory, har systemet også placeret en del af en brugers data i main-memory. Dette er gjort for at kunne udføre en række område forespørgsler uden at benytte databasen, og derved opnå en højere hastighed. En test af område forespørgslen "Find alle brugere i et område på 200*200 meter"i et SPRQ-træ med højden 9, hvor navn og type på de brugere der findes i søge området skal bruges, viste at det ville forøge tiden det tager at udføre forespørgslen med en faktor 12, hvis navn og type skal findes via database kald, hvis SPRQ-træet bliver testet direkte ved kald fra serveren. Sammenlignet med den tid det tager at kalde serveren fra remote objektet eller web servicen, så udgør kaldene fra serveren til datastrukturen kun en meget lille del. Hvor det mest tidskrævende er kald til web servicen. Hvis klienten medregnes, så bruges langt størstedelen af den tid en område forespørgsel tager, på at overføre data om brugerne i søge området til klienten via GPRS. Set fra en brugeres synspunkt er det uvæsentlig om

en område forespørgsel tager 5 sekunder og 10 millisekunder eller 5 sekunder og 100 millisekunder. Men for systemet betyder hastigheden på område forespørgsler og positionsopdateringer, at der kan afvikles flere med en acceptabel hastighed. Hvis et SPRQ-træ indekserer en million bruger objekter der alle bevæger sig, og skal have opdateret deres position en gang i sekundet, så har det stor betydning om en positionsopdatering tager 1 eller 100 millisekunder i den benyttede datastruktur.

Der er udviklet et eksempel på en klient i .NET CF, som kører på en Windows CE Pocket PC. Klienten benytter to former for positioneringsteknologi, nemlig GPS og wifi-positionering, via Place Lab. Det gør det muligt at få en positionsbestemmelse både indendørs og udendørs, idet klienten automatisk skifter til wifi-positionering, hvis det ikke er mulig at benytte GPS. Brugen af Place Lab sammen med klienten forudsætter, at det køres på et apparat der understøtter både Java CDC/Personal Profile og .NET CF. Klienten benytter en GIS, i form af MapPoint web services, til at vise brugere på et kort. MapPoint web services kan også bruges til at finde adresser og planlægge ruter. Disse features kan derfor let implementeres på klienten, hvilket vil forøge klientens anvendelse som et værktøj, der benyttes dagligt. Klienten giver mulighed for at udføre en række område forespørgsler, hvis resultat kan kombineres med et MapPoint web service kort med brugere påsat. Der kan også vises uddybende information om en bruger, og via et filter kan der vælges hvilke brugertyper der skal findes ved område forespørgsler.

Der er som nævnt benyttet GPS og wifi-positionering, for at kunne få en positionsbestemmelse både indendørs og udendørs. Denne løsning er valgt fordi den var mulig med tilgængelig teknologi. Wifi-positionering har den ulempe at wifi APs skal positionsbestemmes og registreres for at kunne benyttes til positionsbestemmelse. Desuden er der ikke wifi dækning i en lang række bygninger, dels fordi wifi signaler svækkes væsentlig når de passerer gennem mure og væge, dels fordi der udenfor centrum af større byer endnu er relativt langt mellem dem. Men løsningen ser ud til at være på vej i form af AGPS chips, der kan benyttes indendørs. De første af disse AGPS chips blev offentliggjort i 2004, og apparater der har dem indbygget er endnu ikke tilgængelige.

For at undersøge om systemet fungerer som planlagt og hvorvidt SPRQ-træet opfylder målene med hensyn til høj hastighed på område forespørgsler og positionsopdateringer, er der blevet foretaget en række tests af både systemet som helhed og SPRQ-træet. De viser at SPRQ-træet kan håndtere både positionsopdateringer og område forespørgsler med en høj hastighed, selv ved et stort antal brugere og et stort antal knuder. Desuden viste testene at både klienten og den resterende del af systemet fungerer, og kan tilbyde en acceptabel hastighed på positionsopdateringer og område forespørgsler, samt at distribueringen fungerer som planlagt.

Derved kan der konkluderes følgende. Systemet kan bruges som et DOPRI system, der dækker et stort område, kan håndtere mange brugere, og kan bruges både in-

dendørs og udendørs. Et main-memory baseret SPRQ-træ opfylder målene for den datastruktur der skal indgå i systemet, nemlig høj hastighed på område forespørgsler og positionsopdateringer. Desuden er der blevet vist at det er bedre at benytte SPRQ-træet i et distribueret DOPRI system, der indekserer geo-refereret information om objekter, som f.eks. butikker og personer, end at benytte et LUR-træ eller et ikke-statisk PR-Quadtræ. Da SPRQ-træet indgår i et distribueret system kan der benyttes så meget main-memory, at det område systemet skal dække kan inddeles i tilpas små delområder, samtidig med at der kan være både indekses og data om et stort antal brugere i main-memory. Derved er det ikke nødvendig at benytte en disk baseret datastruktur, hvilket er hensigtmæssigt da main-memory operationer er meget hurtigere end disk operationer. I den forbindelse er det vist at det forbedrer hastigheden på område forespørgsler væsentlig, hvis den data der benyttes i forbindelse med forespørgslen er placeret i main-memory, fremfor disk.

Kapitel 9

Fremtidsperspektivering

Systemet fungerer som det ser ud på nuværende tidspunkt, og det vil derfor være muligt at bruge det som udgangspunkt for et mere omfattende system. Det vil sige et system hvor der er flere informationsudbydere, i form af brugertyper. Samt flere funktioner på klienten, f.eks. ruteplanlægning og mulighed for at finde adresser. Systemets design og arkitektur gør, at det er relativt enkelt at tilføje yderligere brugertyper, idet hver brugertype er repræsenteret ved en klasse, der nedarver fra den generelle bruger klasse, samt af en tabel i databasen. Ruteplanlægning og lokalisering af adresser kan udføres ved kald til MapPoint web services, og vil derfor være let at implementere.

En yderligere undersøgelse af systems skalerbarhed, i form af tests kunne være interessant. Tests der involverer flere maskiner og web service og server instanser, vil kunne sige noget om hvordan hastigheden er på operationer der involverer flere web service og server instanser på samme maskine, sammenlignet med operationer der involverer web service og server instanser spredt ud på flere forskellige maskiner. Desuden kunne der undersøges hvor mange kald per sekund der skal til før systemet bliver overbelastet, eller giver en uacceptabel hastighed. Web servicen er placeret på IIS, og kan derfor benytte IISs kø funktion, der har en maksimum længde på 8000 http kald [24]. Hverken remote objektet eller serveren har implementeret en kø, hvilket kunne forårsage fejl eller sløve systemet. Derfor kunne der evt. implementeres en kø på serveren og remote objektet. Desuden kunne der undersøges om brug af asynkrone kald, både på web servicen og remote objektet vil kunne forøge hastigheden. Som foreslået i afsnit 7.2.2 kunne der også undersøges hvorvidt det vil være en fordel at distribuere databasen og remote objektet. For at sammenligne med en disk baseret version af SPRQ-træet, kunne en sådan implementeres. Hvorved der kunne foretages tests, der undersøger forskellen i hastighed mellem en main-memory -og en disk baseret udgave.

For at undgå at der bruges så lang tid på at hente kort, med brugere påsat, fra MapPoint web service, kunne der lagres kort på klienten. Derved er det kun brugerne der skal hentes over på klienten, hvorefter de kan påsættes et lagret kort. Dette ville kræve at der indgås en aftale med en kortudbyder, om adgang til geokodede kort. Ulempen ved denne fremgangsmåde er, at systemet så selv skal implementerer de funktioner som MapPoint web services tilbyder.

I systemet bruges Cω til database kald, men Cω streams indeholdende structs kunne også bruges til at indeholde en træ-struktur i main-memory. Hvilket giver mulighed for at foretage xpath forespørgsler på træ-strukturen. Derved er det ikke nødvendig at lave en algoritme for hver forespørgsel, idet f.eks. forespørgsler af typen "Find alle brugere der begynder med Al"eller "Find alle brugere på længdegrad 54"vil kunne klares med xpath forespørgsler.

Det kunne være interessant at undersøge mulighederne for at benytte AGPS i systemet. Specielt med henblik på apparater med AGPS chips der kan benyttes indendørs. Da AGPS kræver at der benyttes et mobilt netværk med AGPS implementeret, vil det højst sandsynlig være nødvendig at indgå en aftale med en mobil operatør, og sikkert også at benytte det API som den pågældende mobile operatør stiller til rådighed til AGPS.

Del V

Appendix

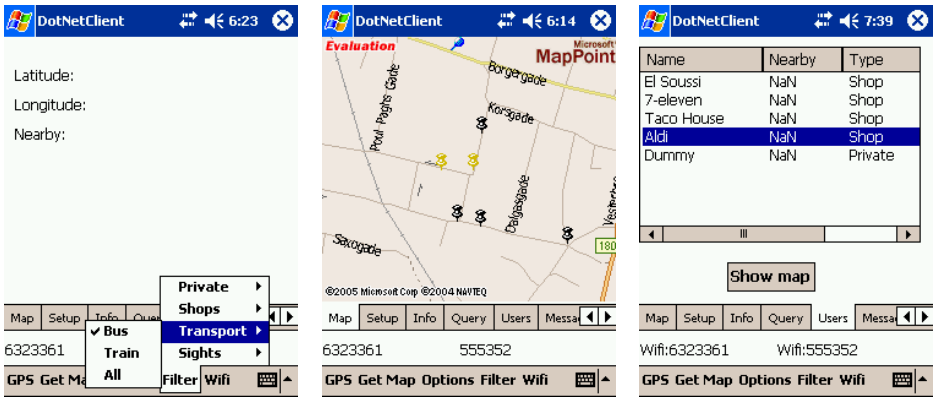
Bilag A

Demonstration af klient

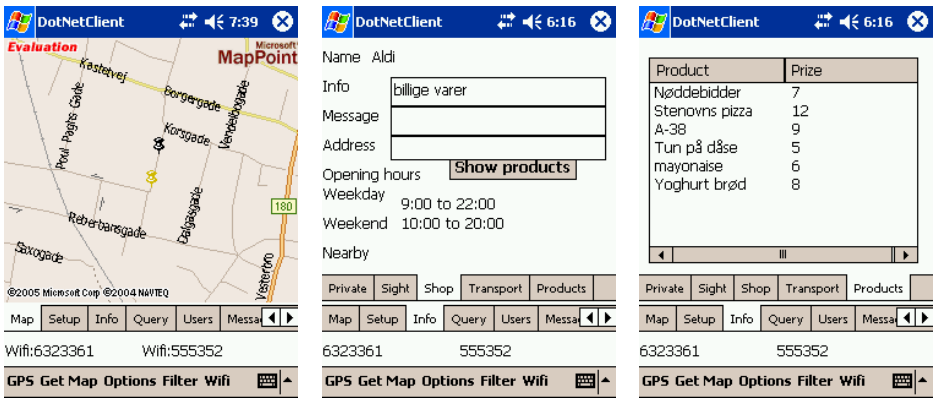
Dette afsnit er en illustreret gennemgang af hvilke funktioner der kan udføres på klienten. Klienten er lavet i .NET CF, på nær Place Lab delen, der er lavet i Java CDC/PP. Den køres på en PDA, hvilket betyder at der benyttes en stylus til at trykke på menuer og knapper, og til at markerer linier i en select box, samt punkter på et kort.

Ved at trykke på menupunktet "Get Map"køres forespørgslen "Find alle brugere i område", hvorefter de fundne brugere påsættes på et kort, der vises under menupunktet "Map". Menupunktet "Filter"bruges til at vælge hvilke brugertyper der skal medtages i forespørgslen. I figur A.1 ses på det første skærmbillede hvordan transportbrugere af typen "Bus"vælges i filtret. Det næste skærmbillede viser et kort med transportbrugere, butiksbrugere og privatbrugere påsat. Det sidste skærmbillede viser menupunktet "Users", som indeholder en liste over de brugere der blev valgt i forespørgslen. Fordelen ved listen fremfor kortet er, at det viser navn og type på brugerne. En bruger i listen kan markeres, hvorved der hentes uddybende information om den valgte bruger. Desuden kan der trykkes på knappen "Show Map", hvorved der hentes et kort med brugeren af klienten og den markerede bruger påsat. I det viste eksempel er en butiksbruger ved navn "Aldi"markeret. I figur A.2 ses i det første skærmbillede et kort med klientens bruger og den markerede butiksbruger påsat. På skærmbillede 2 ses den uddybende information der blev hentet da "Aldi"blev markeret. Ved en butiksbruger er der mulighed for at se hvilke produkter der sælges i butikken, hvilket gøres ved at trykke på knappen "Show Products". På skærmbillede 3 ses hvilke produkter der sælges i den valgte butik.

I den nuværende udgave af klienten er det kun muligt at ændre én form for bruger information, nemlig "Message". Dette gøres ved at vælge menupunktet "Message", indtaste en ny besked og trykke på knappen "Commit Message". I figur A.3, ses i skærmbillede 1 et eksempel på en besked. I skærmbillede 2 kan man se at beske-



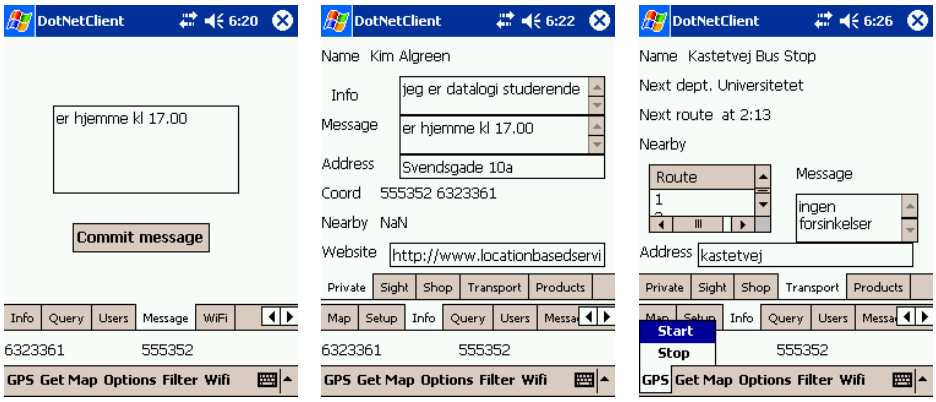
Figur A.1: Filter, vælg bruger og vis kort



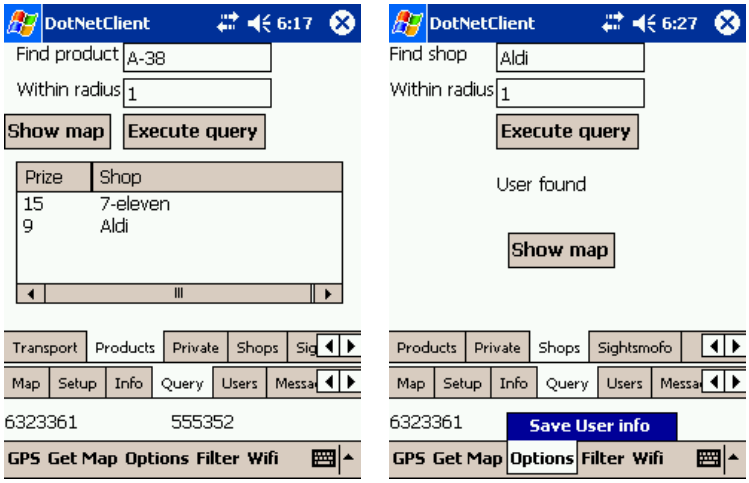
Figur A.2: Bruger på kort, butiksbruger info og produkter

den bliver vist i forbindelse med visningen af information om den privatbruger der gemte beskeden. På skærmbillede 3 ses hvilken information der vises om transportbrugere. Det drejer sig om navn på stoppestedet, hvor næste afgang kører hen, og hvornår den gør det. Desuden vises hvilke ruter der er tilknyttet stoppestedet, en besked, samt en adresse.

Figur A.4 viser to eksempler på forespørgsler. På skærmbillede 1 ses forespørgslen "Find alle butikker der sælger et produkt ved navn A-38 inden for et område på 1 km*1km". For at denne forespørgsel finder nogen butikker skal der i filtret vælges om alle butikker der opfylder forespørgslen skal findes, eller om det kun er butikker der er åbne på forespørgselstidspunktet. På skærmbillede 2 ses et eksempel på forespørgslen "Find alle butikker ved navn Aldi inden for et område på 1 km*1 km". Ved alle forespørgsler gælder, at der ved at trykke på knappen "Show map", kan vises et kort med klientens bruger og resultatet af forespørgslen. Ved produkt forespørgslen i skærmbillede 1 skal der først markeres en af de fundne butikker, før dette kan gøres.



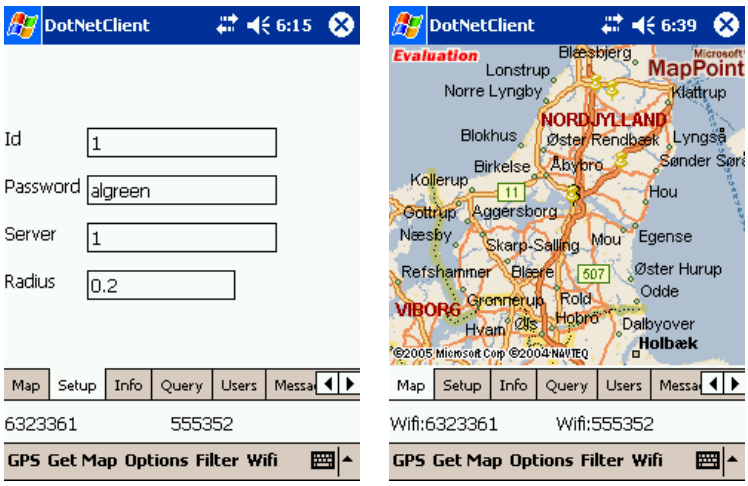
Figur A.3: Besked og bruger info



Figur A.4: Produkt og butik forespørgsel

I figur A.5 ses menupunktet "Setup", hvor id og password på klientens bruger skal indtastes. Desuden vises hvilken server instans klientens bruger objekt befinder sig på, samt hvor stort et område det kort, der hentes ved at trykke på menupunktet "Get Map", skal dække. På skærmbillede 2 ses et kort som dækker et område på 60 km*60 km.

Foruden de her viste funktioner så kan klienten modtage positionsoplysninger, både fra Place Lab klienten og fra GPS modtageren, via GPS.NET APIet. Positionsoplysningerne sendes til systemet, når de modtagende positionsoplysninger adskiller sig med mere end 5 meter, fra den position der sidst blev sendt til systemet.



Figur A.5: Setup og kort

Bilag B

English resume

This report is about the use of a main-memory data structure in connection with a distributed LBS system. The system is a Dynamic Updated Position Related Information (DOPRI) system. Which means that the system is containing position related information and that the positions are updated when the positions of the objects they are representing are changing. Furthermore the information is often updated, to reflect changes in the real-world objects it represents. The objects in the system are representing geo-referenced objects in the form of persons, shops, sights, and transport infrastructure in the form of bus and train stops. One important aspect of a DOPRI system is the ability to efficiently handle position updates and to efficiently relate the positions of objects to each other. Fx. in connection with area request like "Find all users within an area of 200*200 meter". This is done by using a data structure that is capable of indexing two dimensional spatial objects. Such a data structure in usually a tree structure, which can be placed either in main-memory or on disk. The two most used tree structures for indexing two dimensional objects are R-tree and quad-tree variants.

The datastructure developed in this project is based on a Quad-tree variant called PR-Quadtree (P stands for point and R stands for region). The datastructure is based on the assumption that in a datastructure which indexes geo-referenced objects like persons and buildings, there will never be gathered more than an acceptable number of objects within a single area covered by a leaf-node, if each leaf-node covers a sufficiently small area. Therefore the data structure can be made static, in the sense that no changes occurs in the data structure when objects are added, remove or changes positions. Because of its static nature, and because it is based on the PR-Quadtree, the data structure is called SPRQ-tree.

To split an area like Denmark into sufficiently small areas, requires a high number of nodes, which takes up a large amount of main-memory. 32-bit processors can

allocate 2 GB to each application and 64-bit processors can allocate 4 GB to each application. But this is not enough to contain the needed amount of nodes, plus a high number of objects. Therefore it is necessary that the SPRQ-tree is part of a distributed system.

This report examines the claims that in a distributed DOPRI system it is appropriate to use a main-memory based SPRQ-tree as data structure. And that the splitting of an objects data in a main-memory based and a disk based part will improve the speed of requests which only uses main-memory based data. Furthermore the report examines what technology is available at the moment and what technology will be available in the future. This involves technology to determine positions, wireless data transfer and programming technology. Based on this examination a distributed DOPRI system, using the SPRQ-tree, is developed. The system is designed to be able to handle the following requirements. It mus be able to handle a large number of users, cover a wide area, and be used both indoors and outdoors.

Del VI

Litteratur og lister

Tabeller

Figurer

Listings

Litteratur

[1] *64-bit Computing*. http://compreviews.about.com/cs/cpus/a/aapr64bit.htm.

[2] *802.11g hastighed*. http://www.tutorial-reports.com/wireless/wlanwifi/802.11g.php.

[3] *Active Badge*. http://www.uk.research.att.com/ab.html.

[4] *ArcWeb Services*. http://www.esri.com/software/arcwebservices/index.html.

[5] *Asus Supermicro Motherboard*. http://www.bizrate.com/buy/products__att19--1326-43992,cat_id--419.html.

[6] *BlipNet*. http://www.blipsystems.com/Default.asp?ID=238.

[7] *C Omega*. http://research.microsoft.com/Comega/.

[8] *Dansk Meteologiske Institut*. http://www.dmi.dk.

[9] *Ekahau*. http://www.ekahau.com/products/engine/features.html.

[10] *ESRI*. http://www.esri.com.

[11] *FCC about E911*. http://www.fcc.gov/911/enhanced/.

[12] *Franson GpsGate*. http://franson.com/gpsgate/.

[13] *Franson GpsGate pris*. http://franson.com/gpsgate/purchase.asp?platform=ppc&license=express.

[14] *Fujitsu indoor AGPS chipset*. http://rfdesign.com/products/GPS-AGPS-chipset/.

[15] *Garmin*. http://www.garmin.com.

[16] *Geo-caching*. http://www.geocaching.com/.

[17] *Gizmondo*. http://www.gizmondo.com/.

[18] *Global Locate indoor chipset*. http://www.globallocate.com/SEMICONDUCTORS/SEMI_2nd_Gen_Frameset.htm.

[19] *GPS.NET Pris*. http://www.gpsdotnet.com/Pricing.aspx.

[20] *Human Locator*. http://www.freeset.ca/locator/demo.html.

[21] *Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree*. http://db.snu.ac.kr/subby/publications/mdm2002.pdf.

[22] *Kort og Matrikel Styrelsen*. http://www.kms.dk/C1256AED004E87BA/(AllDocsByDocId)/EA08FB744ABCAB9FC1256D1A0030DB87.

[23] *The MapPoint System*. http://www.microsoft.com/mappoint/default.mspx.

[24] *Maximum queue length*. http://support.microsoft.com/default.aspx?scid=kb;EN-US;840875.

[25] *Microsoft MapPoint Web services*. http://www.microsoft.com/mappoint/products/webservice/default.mspx#5.

[26] *Microsoft UDDI Business Registry Node*. http://uddi.microsoft.com/default.aspx.

[27] *Motorola review 1*. http://www.eaid.dk/testcenter.php?action=vis&id=65.

[28] *Motorola review 2*. http://www.3g.co.uk/PR/Feb2004/6516.htm.

[29] *MySQL Spatial*. http://dev.mysql.com/tech-resources/articles/4.1/gis-with-mysql.html.

[30] *Nemerix indoor chipset*. http://www.nemerix.com/site/products/index.htm.

[31] *Nemierix chipset*. http://www.gpsworld.com/gpsworld/product/productDetail.jsp?id=145076.

[32] *.NET Remoting versus Web Services*. http://www.developer.com/net/net/article.php/11087_2201701_1.

[33] *Nokia 9500 Communicator*. http://www.nokiausa.com/phones/9500/0,7747,,00.html.

[34] *Nokia AGPS*. http://www.cursor-system.com/cps/news_detail.asp?ID=134.

[35] *Oracle9i Spatial indexing*. http://www.oracle.com/technology/products/spatial/htdocs/data_sheet_9i/9iR2_spatial_ds.html.

[36] *Orange Location API*. http://www.orangepartner.com/site/enuk/tools/orange_network_apis/orangelocationapi/p_orange_uk_location_api.jsp.

[37] *Place Lab: Device Positioning Using Radio Beacons In The wild*. http://www.placelab.org/publications/pubs/pervasive-placelab-2005-final.pdf.

[38] *R+ tree*. http://en.wikipedia.org/wiki/R_plus_tree.

[39] *R* tree*. http://en.wikipedia.org/wiki/R%2A_tree.

[40] *Rapport om E112*. http://www.gstforum.org/download/rescue/status%20e112%20and%20costs%20eCall%20v03.pdf.

[41] *Real-time Kinetics udstyr*. http://www.neigps.com/SPECSHEETS/VRS_Spec_Sheet.pdf.

[42] *Siemens AGPS*. http://communications.siemens.com/cds/frontdoor/0,2241,hq_en_1_85463_rArNrNrNrN,00.html.

[43] *TomTom*. http://www.tomtom.com.

[44] *Trimble All About GPS*. http://www.trimble.com/gps/.

[45] *WebSphere Everyplace Micro Environment*. http://www-306.ibm.com/software/wireless/weme/.

[46] *What Is GPS.NET*. http://www.gpsdotnet.com/Tour/WhatIsGPSNET.aspx.

[47] *Wikipedia about Galileo*. http://en.wikipedia.org/wiki/Galileo_positioning_system.

[48] *Wikipedia about GLONASS*. http://en.wikipedia.org/wiki/GLONASS.

[49] *Wikipedia about GPS*. http://en.wikipedia.org/wiki/Gps.

[50] *Wikipedia about GSM*. http://en.wikipedia.org/wiki/GSM.

[51] *Wikipedia about wifi*. http://en.wikipedia.org/wiki/Wifi.

[52] *Windows Server 2003*. http://www.microsoft.com/windowsserver2003/evaluation/sysreqs/default.mspx.

[53] *Windows XP Professional 32-bit*. http://www.microsoft.com/windowsxp/pro/evaluation/features.mspx.

[54] *Windows XP Professional 64-bit*. http://www.microsoft.com/windowsxp/64bit/evaluation/top5.mspx.

[55] P. K. Agarwal, L. Arge, J. Erickson, and H. Yu. *Efficient Tradeoff Schemes in Data Structures for Querying Moving Objects*. 2004.

[56] A. Guttman. R-trees: A dynamic index strucure for spatial searching.

[57] C. S. Jensen, D. Lin, and B. C. Ooi. *Query and Update Efficient B+ Tree Based Indexing of Moving Objects*. 2004.

[58] C. S. Jensen and S. Saltenis. Towards increasingly update efficient moving-object indexing. 2002.

[59] L. Mathiasen, A. Munk-Madsen, P. A. Nielsen, and J. Stage. *Objekt Orienteret Analyse og Design*. 1998.

[60] NaN. *OpenGIS Simple Features Specification For SQL*. 1999.

[61] T.-D. Nguyen. *IMORSx*. 2005.

[62] C. M. Procopiuc, P. K. Akerwal, and S. Har-Peled. Star-tree: An efficient self-adjusting index for moving objects.

[63] H. Samet. *Spatial Data Structures*.

[64] J. Sharma. *Oracle spatial White Paper*. 2001.

[65] Y. Tao, D. Papadias, and J. Sun. *The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries*.

[66] J. Tayeb, O. Ulusoy, and O. Wolfson. *A Quadtree-Based Dynamic Attribute Indexing Method*. 1998.