Datarace

Operating systems and multicore programming (1DT089)

Projektrapport grupp 17
Falk Nilsson, Max (910316-2518)
Jaksic, Marina (910411-0920)
Reeves, Max (860107-0033)
Sandberg, Joel (870326-1456)
Toghiani-Rizi, Babak (891109-6371)

Version 1, 4 juni 2014

Innehåll

1 Inledning **3**

2 Systemöversikt **4**
2.1 Systemdesign 5
2.1.1 User interface 6
2.1.2 Server 6
2.1.3 Concurrencymodeller 7

3 Implementation **8**
3.1 Algoritmer och datastrukturer 9

4 Slutsatser **11**

5 Appendix: Installation och utveckling **12**
5.1 Installation 12
5.2 Verktyg 12

1 Inledning

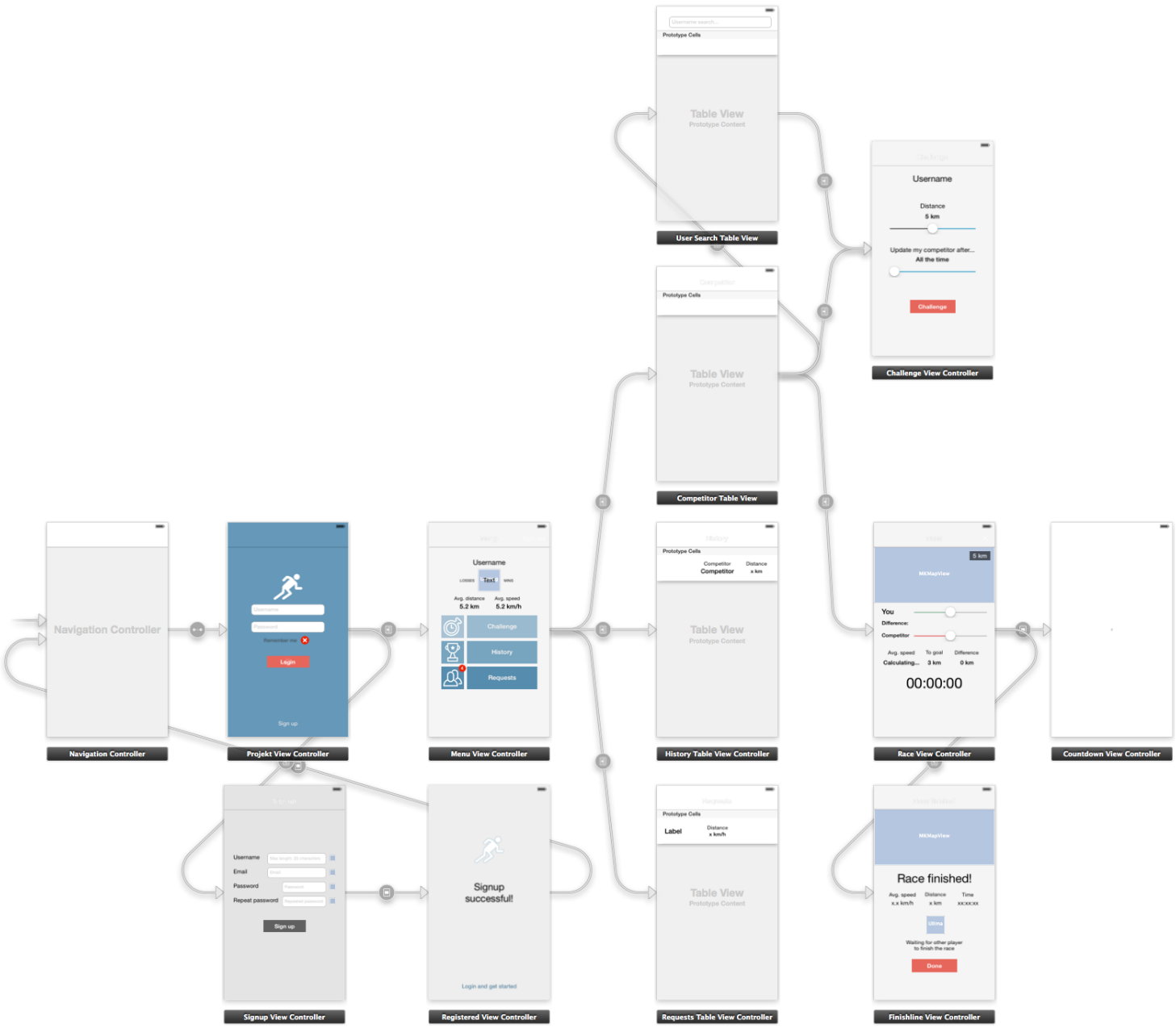
I ett samhälle där människor dagligen använder sig av smartphones och där träning blivit allt mer populärt har vi sett ett behov av att kombinera dessa två. Resultatet är en app där du kan utmana dina vänner i en löpartävling där man kan se vem som är snabbast över en bestämd distans.

Med detta projekt ville vi tillverka en app där användare kan utmana sina vänner i distanslöpning. Syftet var att det ofta kan tyckas vara tråkigt att motionera (här: springa) och vi började fundera över hur vi ska göra det till något roligt. Det var då vi kom fram till att man kan utveckla en app, i vilken man kan utmana sina vänner, och på så sätt göra motioneringen till en rolig tävling.

Vi valde att begränsa oss till en iOS-app, eftersom detta var nytt för de flesta i gruppen och vi ville fokusera på att göra denna app komplett först. I gruppen har vi även lite erfarenhet av att programmera just iOS-appar så därför föll valet på just detta. Vi har även begränsat oss till att endast mäta sträcka utan att ta hänsyn till topografi.

Frågan som återstår är: Är du redo för en utmaning?

2 Systemöversikt



Figur 1: *Översikt över appens olika delar*

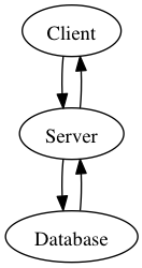
För att börja använda appen, bör man först registrera ett användarkonto och detta görs genom att klicka på signup-knappen. Denna knapp finns på den första bilden i Figur 1. Sedan loggar man in med det nyskapade användarnamnet

och kan nu börja tävla. För att kunna utmana någon skickas en request till en vän. Detta görs efter att utmanaren tryckt på Challenge-knappen. Bild nummer 2 i Figur 1. Vid en utmaning kan man själv välja distans och hur ofta man vill få en påminnelse om hur motståndaren ligger till i matchen. Sedan skickar utmanaren en förfrågan till sin motståndare. När motståndaren har accepterat utmanarens förfrågan kan racet startas när man vill. De tävlande behöver inte vara på samma plats eller starta matchen samtidigt då detta väljs individuellt. När loppet startas sker en tre sekunders nedräkning och sedan är det bara att börja springa. Under loppets gång kommer det visas två förloppsindikatorer den övre visar hur mycket av sträckan du sprungit och den undre visar hur din motståndare befinner sig i förhållande till dig, på den valda distansen. De tävlande kommer även se sin egen sträcka ritas upp på en karta på skärmen och man kommer även kunna se differensen mellan de två tävlande i antal meter. När de båda tävlande har sprungit klart distansen kommer en vinnare utses.

I Historiken kan man hitta statistik på sina genomförda matcher. Här finns antal vinster och förluster och vilka motståndare dessa har varit mot.

Under Request-knappen lagras alla förfrågningar som vänner skickat användaren, här kan man välja att acceptera eller neka en utmaning. Du kan även se dina skickade förfrågningar som ännu inte har accepterats. När användaren väljer att acceptera en motståndares utmaning startar en nedräkning och loppet är igång.

2.1 Systemdesign



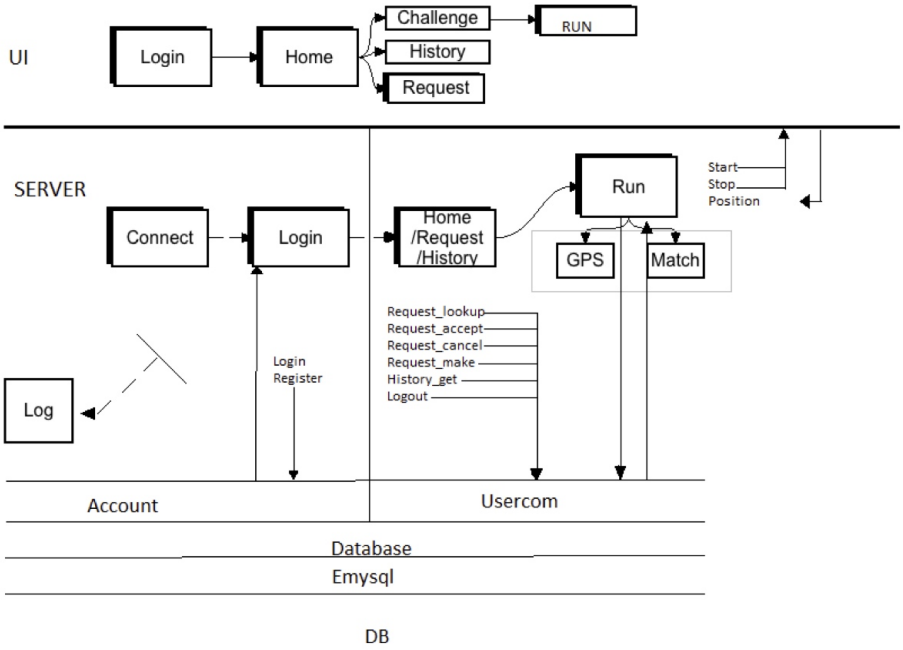
Figur 2: *Högnivåbild över hur systemet är uppbyggt*

Systemet är uppbyggt med en klient och en server, servern har även en databas i enlighet med Figur 2.

Klienten är det användargränssnitt som appen utgör. Servern tar hand om all logik för de användare som använder appen. Servern har i sin tur en databas där all information om användarna lagras och där uträkningar sker. Den gör även alla beräkningar och lagrar sedan dessa i databasen. Resultatet skickas sedan till appen.

Figuren nedan (Figur 3) visar hur de olika delarna i systemet interagerar med varandra. De förfrågningar som görs i UI:t skickas till servern där beräkningarna görs som sedan skickas tillbaka till appen.

När en ny klient ansluter till servern går den in på en ny nod där den får en egen process av typen ”client manager”. Denna kommer att ta hand om all kommunikation för klienten. Den mest relevanta datan kommer att lagras i client managers minne. Sedan använder client manager subprocesser för att hantera och slutföra olika uppdrag.



Figur 3: *Systemarkitekturen där de delar som använder sig av concurrency är markerade med svart.*

2.1.1 User interface

Eftersom appen endast är en dum klient händer inte mycket i de olika moduler som UI:t har. När man loggar in på appen kommer man till Home, hemskärmen, där man kan välja att starta ett race, se vilka förfrågningar som väntar eller se sin historik.

2.1.2 Server

När appen öppnas hamnar användaren i Login. Användaren loggar in med sitt användarnamn och denna information om användarkontot hämtas från databasen. Om det inte existerar ett sådant användarnamn så läggs det till när användaren lägger in sina uppgifter i sign up i UI.

När en person loggar in sparas detta och även eventuella felmeddelanden i Log. Detta underlättar felsökning av eventuella fel som kan uppstå.

Database är ett interface till Emysql, som är det interface som kommunicerar med databasen (DB). Database är alltså vårt eget interface som abstraherar bort det som inte behövs i kommunikationen till databasen. I Usercom sköter man förfrågningar som skickas av användaren. UI skickar koordinater till servern som beräknar positioner och sedan sparar ner denna information.

Home är startskärmen i appen där man kan välja att starta ett race, se vilka requests som väntar och se sin historik, alla dessa kommandon har sina anrop till servern. När användaren utmanar en vän skickas denna utmaning via servern till den klient som utmanats. När ett race startas hamnar man i Run där beräkningar sker i servern.

Dessa beräkningar hanteras av GPS och Match. GPS hanterar diverse beräkningar som GPS-koordinater som klienten skickar, sparar distans, hastighet och fart. I Match utses vinnaren av det avslutade racet och resultatet skickas till klienten och dess opponent.

2.1.3 Concurrencymodeller

I detta projekt har vi använt oss av Erlang som språk till servern som använder sig av Actor-modellen som concurrency-modell.

Erlang var ett självklart val på grund av att man med lätthet kan utveckla concurrent mjukvara. Detta på grund av dess naturliga concurrency och att det finns bra verktyg för att tillverka och övervaka processer. Det är genom message passing som Erlangs concurrency sker. Vi ville även få en djupare föståelse och kunskap för Erlang OTP, dess designprinciper och ramverk, för att kunna förstå vilka verktyg som ska användas och hur Erlang applikationer utvecklas på ett professionellt sätt.

Appen har utvecklats i språket Objective-C, som är huvudspråket för iOS. Till skillnad från vad vi hade tänkt i projektförslaget så valde vi att göra appen concurrent. Eftersom vi använder oss av nätverkskommunikation och GPS-spårning blir detta ännu effektivare när man gör det concurrent. Det primära sättet som concurrency används är genom att ge kommunikationen en egen tråd där den exekverar sin egen kod och sedan återgår till huvudtråden. Objective-C har redan befintliga ramverk för tillämpande av concurrency, dessa är väldigt enkla att använda och det medför att man inte behöver oroa sig för synkronisering. De delar som inte använder sig av någon form av nätverkskommunikation har däremot inte någon concurrency, men dessa delar är endast det grafiska i appen.

3 Implementation

Systemet är implementerat på så sätt att vi använt Erlang OTP som serverspråk och Objective-C som språk till appen. Valet av just Erlang var för att vi alla blev intresserade av att lära oss det när vi började läsa om det i kursen. Fördelen är att det är väldigt enkelt att sätta sig in i och det har ett inbyggt stöd för concurrency. Vi ville ha en server som klarar av att många användare är inloggade samtidigt och då ville vi ha en server som var stabil och klarade av att hantera detta. När vi läste in oss på hur Erlang OTP fungerade insåg vi att detta var det språk vi behövde för att bygga vår server. OTP är ett redan färdigt ramverk och det är väldigt enkelt att använda sig av de redan existerande modellerna.

Vi hade arbetat lite med Erlang innan så vi behövde inte lägga ned mycket tid på att lära oss grunderna där. Erlang är inte statiskt typat och det gör det svårare att upptäcka fel då koden inte exekveras effektivt. Precis som Java använder sig Erlang utav en virtuell maskin. Detta gör att språket i sig inte är det snabbaste eftersom programmet först behandlas av den virtuella maskinen som sedan översätter instruktionerna till vår dator. Vi hade endast grundläggande kunskaper i Erlang så att lära sig mer var inte det lättaste. Språket är inte lika stort som Java/C så att utbudet av information på nätet var begränsat. Erlang använder sig också av annorlunda syntax än de språk vi tidigare arbetat mycket med så det var en omställning att växla mellan dessa.

När vi bestämde oss för att utveckla appen till iOS var det Objective-C som var det första vi tänkte på. Detta på grund av att det är huvudspråket för iOS och vi hade lite erfarenhet av att programmera i språket. Fördelen är att man har en app som är skriven i samma språk som operativet och därför undviks problem som kan uppstå med andra språk. Nackdelen med just Objective-C är att det är väldigt enkelt att använda vanlig C kod, vilket i och för sig är smidigt. Problemet blir ofta då att när man är van att skriva C kod så skriver man oftast det istället för att använda sig av Objective-C. Detta medför nog att man missar funktionalitet som annars hade funnits om man skrivit de som Objective-C är tänkt.

3.1 Algoritmer och datastrukturer

I servern används listtraversingar för att hitta den information som behövs för just den process som nu körs. Dessa traversingar går linjärt genom listan för att hitta den information som söks.

Nedanstående kod exempel är en hjälpfunktion till funktionen statistics compare som jämför data mellan två tävlande vid en specifik tidpunkt. Denna hjälpfunktion är till för att ta reda på den totala distansen för en tävlande vid en specifik tidpunkt.

```
calc_totaldistance(User_id1, Match_id) ->
    Gps1 = usercom:gps_get(User_id1, Match_id),
    calc_totaldistancehelp(Gps1, 0).


calc_totaldistancehelp([], Distance) ->
    Distance;

calc_totaldistancehelp([Last], Distance) ->
    Distance;

calc_totaldistancehelp([First,Sec | Tl], Distance) ->
    Sum_distance = distance(First#gps_table.longitude,
                            First#gps_table.latitude,
                            Sec#gps_table.longitude,
                            Sec#gps_table.latitude),

    calc_totaldistancehelp([Sec|Tl], Sum_distance + Distance).
```

För att beräkna distansen mellan två koordinater använder vi oss av en känd algoritm vid namn Haversine formula. Denna algoritm beräknar cirkelavståndet mellan två punkter på en sfär från deras longitud och latitud. Namnet Haversine kommer från en matematisk formel:

$$\operatorname{haversin}(\theta) = \sin^2\left(\frac{\theta}{2}\right) = \frac{1-\cos(\theta)}{2}$$

Figur 4: *Matematiska versionen av Haversine formeln*

Med hjälp av denna formel beräknar vi koordinater i vår GPS-modul. Vi får två decimala tal från appen som sedan beräknar koordinaterna i servern med vår funktion distance:

```
distance(Long1, Lat1, Long2, Lat2) ->
    DegToRad = fun(Deg) -> math:pi()*Deg/180 end,
    [RLong1, RLat1, RLong2, RLat2] = [DegToRad(Deg) ||
                                Deg <- [Long1, Lat1, Long2, Lat2]],

    DLong = RLong2 - RLong1,
    DLat = RLat2 - RLat1,

    A = math:pow(math:sin(DLat/2), 2) + math:cos(RLat1)
        * math:cos(RLat2)
        * math:pow(math:sin(DLong/2), 2),

    C = 2 * math:asin(math:sqrt(A)),

    %% radius of Earth is 6372.8 km
    Km = 6372.8 * C,
    Km.
```

Vi använder två punkter (longitud och latitud) i decimal form som argument. Dessa omvandlar vi sedan från grader till radianer. Beräkningar görs enligt den matematiska algoritmen. Vårt slutgiltiga svar omvandlar vi till km genom att multiplicera det med jordens radie.

Vi har inte använt några speciella algoritmer till detta projekt förutom de ovan nämnda.

De datastrukturer vi använder oss av är listor, detta på grund av att vi aldrig har större mängder data som hanteras i minnet samtidigt. Den största datamängden hanteras i databasen och där har vi använt oss av MySQL. Som tidigare nämnts så används så pass lite data i minnet så vi valde att använda oss av listor där vi lagrar informationen. De paket vi skickar från appen till servern består av structar i vilka vi har all information som behövs för de olika funktionerna som tillhandahålls.

Vår implemetantion använder sig av concurrency på flera sätt. Både i själva appen och i servern. I appen har vi implementerat concurrency genom att nätverkskommunikation och GPS-spårningen görs i egna trådar. Servern är concurrent genom att varje användare får en egen process i servern och där med så sker allt i servern concurrent så fort det är flera som använder appen. Detta underlättar när vi vill kunna bevaka en viss användares process och det går väldigt lätt att se hur många som använder appen vid ett visst tillfälle.

Fördelen med denna implementation är att användare hanteras separat. När två är inloggade samtidigt och en användare kraschar, påverkas inte den andra. Appen drar inte onödig energi genom att ha nätverkskommunikation i separat tråd som inte är huvudtråden. Den blir på detta sätt snabbare för användaren. Vi använder även Erlang OTP som är färdiga ramverk, vilket minskar risken för fel eftersom dessa ramverk redan är testade.

En nackdel är att många processer är beroende av varandra och måste samverka för att det ska fungera. Yttligare en nackdel är att det är svårare att programmera så att det blir concurrent i appen. Det är då inte är helt uppenbart vilka metoder som finns tillgängliga för att just få till concurrencyn man vill åt. Då det finns många olika bibliotek för concurrency.

Inga deadlocks kommer uppstå i servern, detta på grund av att Erlang använder sig av message passing. Det kommer inte heller att uppstå deadlocks i appen eftersom vi där använder oss av NSOperations-metoden som abstraherat bort all synkronisering och denna metod ser till att dessa inte uppstår.

4 Slutsatser

Vi har i detta projekt gjort en app och till denna en server som tar hand om alla de beräkningar som behövs för att appen ska göra det som utlovat är. Vi anser att vi lyckats bra med det vi ville ha gjort med projektet. Vi har en produkt som har alla de funktionaliteter som vi hade bestämt att vi ville ha med och vi har även hunnit lägga till ytterligare funktionalitet. De vi lyckats utöka vår app med är till exempel att även appen gör allt det den ska göra concurrent. Vi har även lyckats hålla de deadlines vi satte till en början vilket har hjälpt oss i utvecklandet.

Om vi skulle haft mer tid så finns det några delar som vi tänkt skulle vara kul att implementera. Det vi skulle vilja göra är utöka appen med ännu fler funktioner. Exempelvis utmarkerade banor, utmana fler personer och att man kan springa fler än bara två per race samtidigt. Detta projekt har lärt oss väldigt mycket om både server och app delen då vi inte tidigare arbetat med att synkronisera dessa två. Speciellt intressant har det varit att vi har använt två olika programmeringsspråk och länkat ihop dessa. Det svåra har varit att arbeta med appen eftersom det är ett helt nytt språk och det tar längre tid att åtgärda problemen som uppstår.

Samarbetet i gruppen har fungerat väldigt bra och vi har arbetat tillsammans under hela projektets gång. Detta har underlättat för alla medlemmar då vi har kunnat jobba mer effektivt med att lösa problem som uppstår.

Det som vi tyckte var oväntat svårt var att lära sig hur Objective-C fungerade, eftersom det är så pass likt C så är det lätt att råka skriva kod som fungerar bra i C men som inte är lika effektiv i den slutgiltiga appen. Vissa gånger var vi tvungna att ta god tid på oss att leta upp information och läsa in för att lösa problem.

Hade vi fått göra projektet på nytt hade vi velat vara mer pålästa om Objective-C. Då skulle det vara lättare att arbeta med appen och det hade inte tagit lika mycket tid som vi har lagt ner nu. Det hade även kunnat lösa några av de problem som vi stött på på grund av att vi blandat C kod och Objective-C kod. En Android-app hade varit ett alternativ då alla i gruppen har programmerat i Java och har grundläggande kunskaper.

5 Appendix: Installation och utveckling

5.1 Installation

För att installera appen krävs att man är registrerad utvecklare hos Apple. Man måste ladda hem källkoden och genom Xcode läggs appen in i iPhonen, detta går endast att göra om om man är betalande utvecklare, annars kan man simulera appen i Xcode. Installationen kommer att se annorlunda ut om den läggs till i App Store, då kommer appen att installeras på samma sätt som alla andra appar installeras.

För att installera servern krävs det att man har en dator med ett Unix/Linux-system installerat. Servern är körd och testad på en Debian-version, men bör fungera på fler distributioner. Den är uppdelade i två delar för installationen, själva servern är skriven i Erlang och databasen skriven i Mysql. Erlang-delen av servern är skapad som en fristående utgåva och kan köras genom att bara packa upp arkivet och köra erl i /bin/erl när man står i mappen rel under server mappen. Alla delar för servern kommer följa med här så ingenting extra krävs.

Servern sparar information om användare och lopp i en MySql-databas. Denna databas måste också finnas installerad för att servern ska fungera. Du kommer behöva Mysql installerat och lite kunskap om hur MySql fungerar. Det kommer finnas en MySql dump med hur databasen är uppbyggd som du kan använda för att göra en tom kopia av databasen. Beroende på hur du gör detta kan du behöva ändra i konfig filen i servern för databasen, /configs/db_config. Om allt detta går bra ska du kunna starta servern med /bin/erl. Beroende på hur du tänkte använda servern kan du behöva ändra inställningar för vilken port som ska användas samt konfigurera för att denna port ska vara åtkomlig.

5.2 Verktyg

Verktyg som använts för utveckling är Xcode 5.1 till appen, detta är Apples egna verktyg för apputveckling och därför kändes det som ett självklart val. Till servern användes Emacs på grund av att vi alla har erfarenhet av att jobba med det samt är installationen av Erlang i Emacs mycket enkel.

Vi har använt oss av GitHub för att distribuera koden mellan oss i gruppen. Det är även via GitHub som det går att hämta hem repot med källkoden till detta projekt. Dokumentationen av koden har skett genom eDoc för Erlang och Doxygen för appen. Vi använder EUnit för testning av koden till servern. Det är svårt att testa koden till appen eftersom den bygger på funktioner och metoder som inte returnerar något, och eftersom det som koden genererar är endast ett UI ansåg vi att det inte behövdes.