GE.neric E.rlang (game) SE.rver

Operativsystem och multicoreprogrammering (1DT089) våren 2014.

Slutrapport för grupp 10

Johan Gille (930922-3452),
Niklas Hökenström (880615-6215),
Valle Magnusson (900215-5217),
Jonas Nilson (920927-2179),
Christian Törnqvist (920104-0616)

2014-07-13
version 2

Innehållsförteckning

1. Introduktion

Antalet spel med flerspelarläge har under de senaste åren ökat drastiskt. Generellt sett har spel med flerspelarmöjlighet en större möjlighet att bli väletablerade än spel utan då många lockas av möjligheten att spela tillsammans med andra. Kravet på robusthet och god responsivitet hos spelservrar växer med tiden och om man en gång har spelat mot servrar med bra svarstid och stabilitet kan det vara svårt att nöja sig med något sämre. Spel som är baserade på servrar med hög nertid och lång svarstid riskerar att således förlora spelare.

Med anledning av onlinespelens expandering ville vi utforska vad som krävs för att implementera en spelserver där egenskaperna robusthet och hög responsivitet är uppfyllda. Huvudfokuset var därmed riktad på att skapa en generell spelserver som skulle kunna hantera ett större antal anslutna klienter samtidigt med hjälp av *concurrency*. Med generell spelserver avses att det skulle vara möjligt att skriva ett godtyckligt spel till servern som enkelt skulle kunna bytas ut mot ett annat. En klient som anslutit sig till servern skulle sedan erhålla möjligheten att skapa ett spelrum med valfritt spel dit andra klienter skulle kunna ansluta sig och delta i spelet. Då tyngdpunkten i projektet låg på spelservern prioriterades inledningsvis inte spelimplementationen. Argumentet till detta var att spelet skulle vara simpelt och därav inte dra någon nytta av *concurrency*.

2. Översikt över systemet

När man som användare startar klientapplikationen möts man av ett *GUI* (Graphical User Interface), där man har möjlighet att skriva in *en IP-adress* och *port* till den spelserver man vill ansluta till. Om anslutningen lyckas uppdateras fönstret med fler knappar samt en lista som innehåller aktiva spelsessioner på servern, *se fig 2.1*. Som användare har man möjlighet att skapa en ny spelsession genom att trycka på knappen där det står *Add new table*. Övriga knappar som existerar är *Ping* som mäter svarstiden från servern, *Refresh* som uppdaterar listan med spelsessioner, samt *Change name* där användare har möjlighet att ändra sitt namn som kommer synas när denne spelar. När användaren har tryckt på *Add new table* möts denne av en ruta där man kan välja namn på spelsessionen, modifiera antal tillåtna spelare samt välja vilken speltyp som spelsessionen ska innehålla (för tillfället finns bara ett spel implementerat). Varje spelsession har även en *Join*-knapp på samma rad som den visas i listan. Om användaren trycker på *Join* kommer denne att försöka ansluta till den valda spelsessionen och om det finns minst en tillgänglig plats kvar möts användaren av en startbild där spelets instruktioner visas. Därefter startas spelet genom att användaren trycker på valfri knapp.

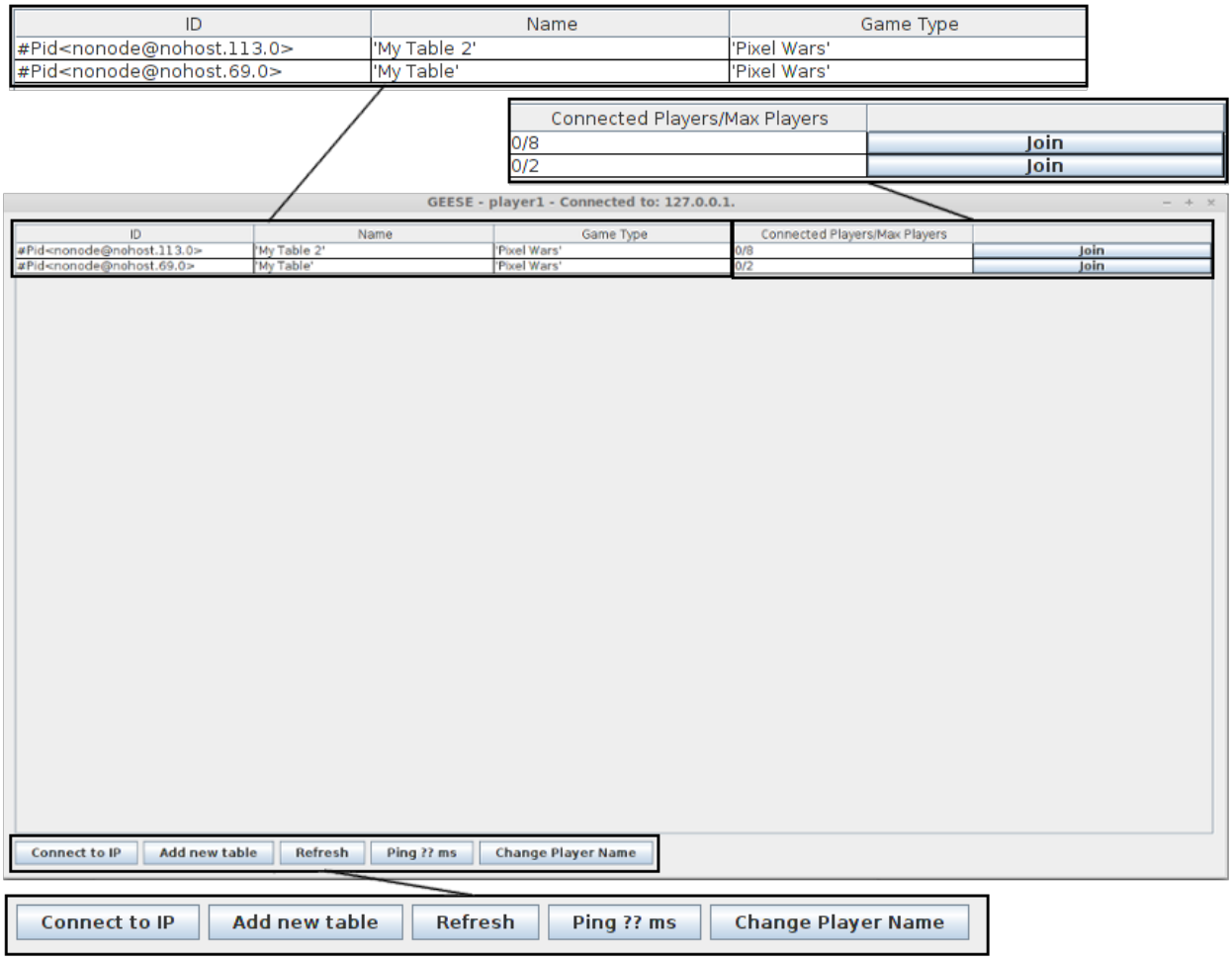

fig 2.1 Klientapplikationen listar serverns aktiva spelsessioner

2.1 Systemdesign

Systemet är uppbyggt på tre större delar: klient, server och spellogik *(se fig 2.2)* som alla i sig är uppdelade i flera delar och moduler.

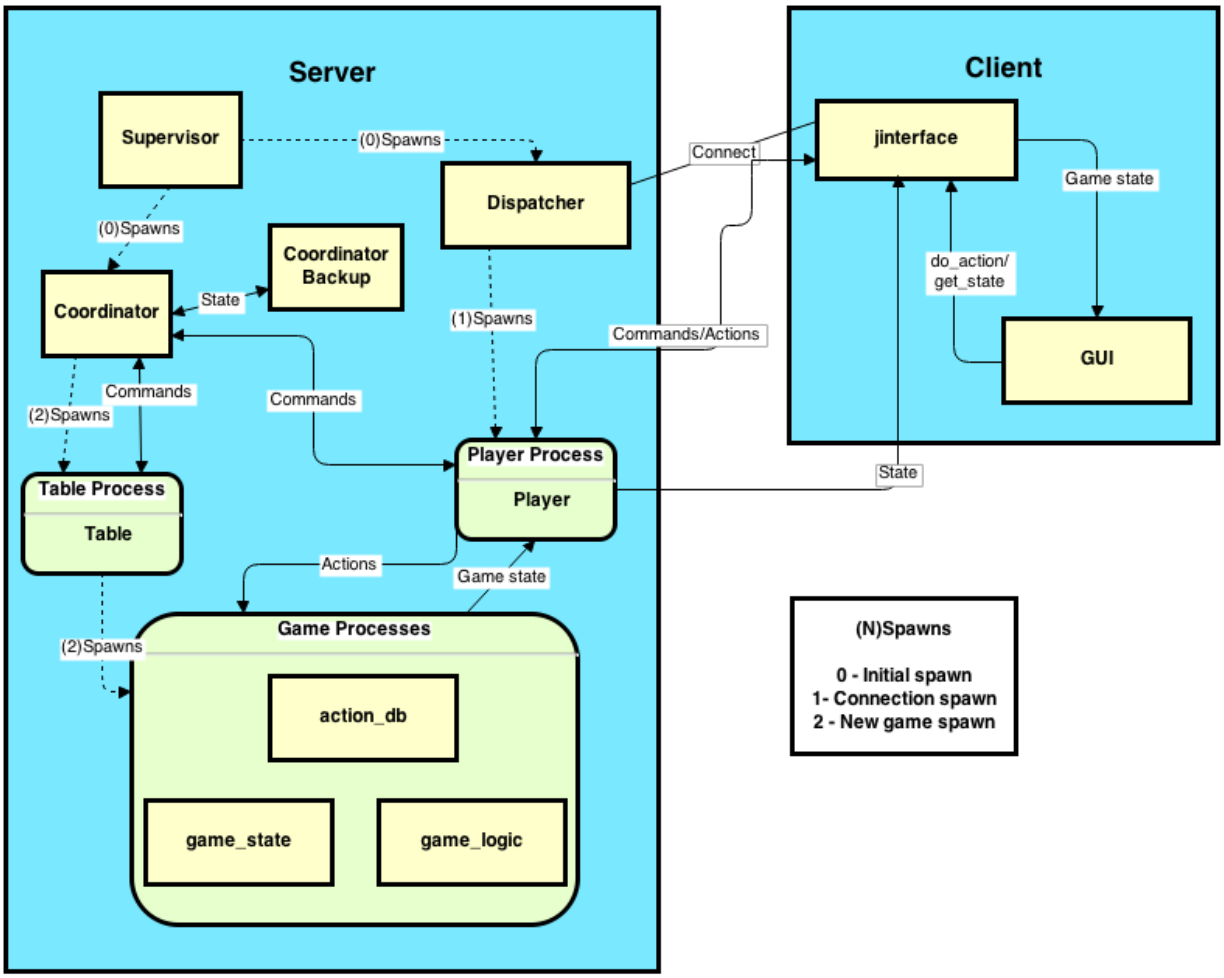

fig 2.2 Systemarkitektur för server och klient

2.1.1 Klient

För att klienten ska kunna kommunicera med servern finns en klass, *Jinterface (se fig. 2.2)*, som sköter kommunikationen med servern. Eftersom att servern är skriven i *Erlang* och klienten i *Java* representeras data på olika sätt och därför kräver det att klienten sköter de nödvändiga konverteringarna mellan *Erlangs* och *Javas* datatyper. Konverteringarna sker genom typomvandlingar med hjälp av *Java*-paketet *Jinterface* (inte att förvirras med klassen *Jinterface*) som tillåter representationer av *Erlang*-objekt i *Java.* När en klient har anslutit till en server och gått med i en spelsession börjar de första representationerna av spelet ritas ut. Allt som ritas ut i

det faktiska spelet är baserat på den data som tas emot från servern. När man rör på sig i spelet skickas den nödvändiga informationen till servern som då kan uppdatera spelarens position och inkludera det i informationen som därefter skickas ut till de anslutna klienterna.

2.2 Moduler

2.2.1 Coordinator-modulen

Coordinator-modulen (se *Coordinator* i fig 2.2) ansvarar för anslutna klienter, vilka spelinstanser som är initierade och hur många klienter som är anslutna till en specifik spelinstans. Exempelvis har varje spelinstans ett max antal spelare och om detta antal är uppnått hindrar *Coordinator-modulen* ytterligare spelare från att ansluta till den specifika spelinstansen. Denna modul har även en *backup-modul* (se *Coordinator Backup* i fig. 2.2) där tillståndet från *Coordinator-modulen* finns lagrat.

2.2.2 Dispatcher-modulen

Dispatcher-modulen (se *Dispatcher* i fig 2.2) ansvarar för att acceptera klienter som ansluter till servern. Efter att anslutningen har accepteras kommer modulen att starta en instans av *Player-modulen* (förklarad i 2.2.5) som kommer att kommunicera med klienten.

2.2.3 Supervisor-modulen

Supervisor-modulen (se *Supervisor* i fig 2.2) skapar och *länkar* en *Dispatcher-modul,* en *Coordinator-modul* samt även *Coordinatorns backup-modul.* Med *länkning* innebär det att om någon av de modulerna som *länkats* kraschar kommer *supervisorn* få reda på detta och kunna hantera krascher på ett flertal fördefinerade *omstartningsstrategier*. *“One-for-one”*, som är den omstartsstrategi (*restart strategy*) vi har valt, innebär att om en modul kraschar startas endast denna modul. För varje gång tillståndet alterneras i vår *Coordinator-modul* sparas detta tillstånd i *Coordinator-backup-modulen*. När *Coordinator-modulen* sedan startas om startar den med det tillstånd som fanns lagrat i *Coordinator-backup-modulen* om det fanns något lagrat där.

2.2.4 Table-modulen

Vid initiering av *Table-modulen* (se *Table Process* i fig 2.2) sätts spelinstansen namn samt max antal tillåtna spelare. Vid initieringen startas även en spelinstans. Därefter är *Table-modulens* ansvar att hålla koll på antalet anslutna spelare, max antal spelare samt vilket *process id* som spelinstansen har.

2.2.5 Player-modulen

För varje klient som ansluter skapas en instans av *Player-modulen* (se *Player Process* i fig 2.2). I *Player-modulen* finns det två tillstånd: speltillståndet och kommunikationstillståndet. I kommunikationstillståndet kan klienten lägga till ett nytt spelbord, få tillbaka en lista med tillgängliga spelbord, ta bort sig själv från ett spelbord, samt få tillbaka en lista med alla spelare. När en klient väljer att ansluta till ett spelbord och om denna anslutning lyckas kommer *Player-modulen* hamna i ett speltillstånd där den tar emot *spel-actions* som skickas från klienten.

2.2.6 Jinterface-modulen (På klientsidan)

All kommunikation mellan klienten och servern sker genom *Jinterface-modulen* (se *Jinterface* i fig 2.2). Modulen innehåller ett antal metoder som tillåter klienten att skicka och ta emot meddelanden via *TCP* från serversidans *Player-modul*. Kommunikationen mellan klienten och servern sker alltid genom ett ursprungligt meddelande från klientsidan och, ibland men inte alltid, skickas ett svarsmeddelande från serverns *Player-modul* till klienten. *Jinterface-modulen* vet i sådant fall om att ett svar kommer att komma och väntar in detta. Tanken från början med denna modul var att göra den så generell som möjligt att den skulle kunna fungera som en kommunikationslänk mellan klient och server för ett godtyckligt antal olika speltyper - men metoden *getState(),* som hämtar speltillståndet, är anpassad specifikt efter den speltyp som klienten använder.

2.2.7 GUI

Det grafiska användargränssnittet för programmet består av ett grafiskt gränssnitt för listan över spelinstanser och menysystemet, samt en grafisk representation av spelet *Pixel wars*. Dessa två delar skiljer sig åt både när det gäller utseende och implementation och kan sägas utgöra två separata moduler. Det grafiska gränssnittet för menysystemet är generellt då det fungerar likadant oberoende av vilken server du är ansluten till eller vilket spel man ska spela. Representationen av spelet bygger dock helt och hållet på spelspecifik data som klienten förväntas få av servern.

3. Implementation

3.1 Internetprotokoll - TCP

Kommunikation mellan server och klient sker med hjälp av en anslutning via *TCP* (*Transmission Control Protocol*). Vid sändning av ett *TCP-paket* garanteras paketets ankomst. Det finns även garanti för att det (i stort sett alltid) är samma paket som sändaren skickar som mottagaren tar emot. Vid uppstarten av projektet efterforskade vi lite vilka protokoll som skulle vara lämpliga att använda. Vi kom då fram till att kommunikation via internetprotokollet *UDP* (*User Datagram Protocol*) skulle vara passande. Fördelen med *UDP* till skillnad från *TCP* är att paketen kan skickas väldigt snabbt då informationen som skickas och tas emot inte felkontrolleras automatiskt. Nackdelen med *UDP* är att det inte finns någon garanti för att paketen har kommit fram. *UDP* kan inte heller lova att paketen har anlänt i korrekt ordning. Våra paket skulle även vara så små att tidskillnaden i överföring mellan *TCP* och *UDP* skulle vara försumbar. Dessa nackdelar ledde till att vi valde *TCP* över *UDP.*

3.2 Spelserver i Erlang

Målet med detta projekt var att skapa en generell spelserver som skulle kunna hantera många spelare och spelinstanser. För att uppnå en god robusthet föll valet gällande programspråk på *Erlang*. *Erlang* är ett *funktionellt* språk med bra stöd för *concurrency* som körs i ett *runtime-system*. Språket har ett ramverk som heter *Open Telecom Platform* (eller *OTP*) som ursprunglingen var skapat för telekomindustrin där kravet på låg *downtime* är av högsta prioritet. För att uppnå låg *downtime* har *OTP* något som kallas för *supervising. Supervising* innebär att

om en process skulle dö kan Erlang enkelt starta om denna med en mängd olika *restartstrategier* vilket gör systemet robust och säkert. I *OTP* finns det utöver *supervising-processer* även processer som utför allt arbete och beräkningar. Dessa processer kallas för *workers*. Enligt *OTP's designprinciper* kan man bygga upp stora träd av *workers* och *supervisors.* Kravet är dock att alla processer måste ha någon *supervisor* förutom den som är allra högst upp i trädet. Det vill säga, alla *workers* ska ligga längst ner i trädet. Vi har utnyttjat *supervising* för modulerna *Coordinator* (se kap 2.2.1) och *Dispatcher* (se kap 2.2.2) samt *Coordinatorns backup-modul*. Däremot har vi inte *supervising* av av processer som startar efter initieringen av servern (läs mer i 4.2.2).

Ytterligare en fördel med *Erlang* när det gäller serverapplikationer är dess lättviktsprocesser som bland annat är dynamiskt växande i *Erlangs* virtuella maskin. Tack vare denna egenskap kan servern skapa en ny process för varje klient som ansluter utan att det resulterar i någon större belastning.

3.3 Spellogik i Erlang

För att simplifiera komplexiteten i vårt projekt har vi valt att ha all spellogik på servern. Det finns både för- och nackdelar med detta designval. Några fördelar är följande:

1. Klienter kan inte fuska (Sanning med modifikation: om spelet skulle vara ett spel där det användaren exempelvis behöver sikta med muspekaren går det att skapa fusk som siktar åt spelaren).
2. Implementationen blir mycket enklare. Främst på grund av att man slipper problem som synkronisering mellan klient och server.
3. Det är enklare att utöka logiken om ändringar endast behöver göras på serversidan.

Däremot kan spelet upplevas som mindre responsivt när spellogiken ligger på servern eftersom att klienten måste vänta på svar från servern innan den kan se att något händer. Detta blir extra tydligt om svarstiden mellan servern och klienten är hög.

Om projektet hade varit mer omfattande och spelet varit mer avancerat skulle det troligen vara lättare att skriva spellogiken i ett objektorienterat språk. Det fanns planer på att implementera spellogiken på klientsidan - men redan i startfasen av projektet insåg vi att det skulle ta längre tid än väntat. Dessutom var personen som skrev logiken mera insatt i funktionell programmering och han ansåg att vi skulle tjäna tid på att skriva logiken i *Erlang*.

3.4 Klient i Java

Vid implementationen av klienten ville vi ha ett språk vi var någorlunda bekanta med och som dessutom hade bra kommunikationsmöjligheter med *Erlang*. *Java* uppfyllde båda dessa önskemål då det finns ett *Java-paket* som heter *Jinterface* (se rubrik 3.4.1), samt att vi alla tidigare har programmerat i *Java*. Klientens funktionalitet kan delas upp i två separata delar; den ena delen tar hand om allt som har med kommunikationen med servern att göra och den andra delen behandlar användarens knapptryckningar samt ritar ut spelets grafiska tillstånd som mottagits från servern.

3.4.1 Jinterface

Kommunikationen mellan *Java* och *Erlang* sker som tidigare nämnts via ett TCP-protokoll (se rubrik 3.1). För att hantera och manipulera data i *Java* som skickas till och tas emot från *Erlang-servern* använder vi oss av ett ramverk som heter *Jinterface*. Med hjälp av *Jinterface* blir det i Java möjligt att via TCP ta emot den binära koden som servern skickat och tolka den som en *Erlang-datatyp* och därefter göra de nödvändiga omvandlingarna för att kunna representera informationen i *Java*. När klienten ska skicka information till servern används *Jinterface* igen, men nu för att omvandla *Javas* datatyper till binärkod som *Erlang* kan tolka.

3.4.2 Swing och Java 2D

Den grafiska meny som klienten använder sig av i form av knappar, dialogrutor och tabeller har byggts upp med hjälp av *Java Swing* medan representationen av spelet ritas ut med hjälp av *Javas Graphics2D*. Prestandamässigt finns det många externa bibliotek som antagligen hade passat oss bättre, men då de båda fanns inkluderade i *Javas* standardbibliotek var det enkelt att komma igång med dem.

3.4.3 Spelet

Vi var från början intresserade av att skapa ett spel som liknade *Liero. Liero* härstammar från Finland och själva namnet betyder “mask” på finska. Spelet i sig är ett plattformsspel i flerspelarmiljö där spelarna styr en varsin mask vars uppdrag är att skjuta ihjäl motståndarna. Vårt spel som vi har valt att kalla för “Pixel wars” är mycket likt *Liero* i anseende av speltyp. En av de större skillnaderna (förutom att *Liero* är oerhört mycket mer välgjort) är att i vårt spel så “siktar” användaren med hjälp av musen medans man i *Liero* siktar med hjälp av tangentbordets piltangenter. Se *fig. 3.1* för att få en bättre översikt av hur vårt spel ser ut. Varje spelare har information om ping till servern och position på spelplanen tillgängliga i sitt övre vänstra hörn (se fig. 3.2). Informationen uppdateras varje gång som ett nytt tillstånd har mottagits från servern. I spelarens övre högra hörn finns information om den aktuella spelsessionen (se fig. 3.3) där alla spelare listas med antal vinster, hur många spelare som spelaren har eliminerat och hur många gånger som spelaren har dött.

Varje spelare startar med 100 HP (health points, se den röda rektangeln i fig. 3.4) som minskar varje gång man blir träffad av en annan spelares skott. Om en spelares HP hamnar på 0 eller blir negativt beräknas denne vara död. När en spelare är död tillåter inte servern att denna spelare varken rör på sig eller skjuter. Däremot kan spelaren med en knapptryckning välja att återuppstå (respawn) och fortsätta att spela som vanligt.

Varje gång som en spelare skjuter minskar dennes tillgängliga energi (se den vita triangeln i fig. 3.4) och om man inte har tillräckligt med energi kan man inte skjuta. Genom att röra på sig fylls energin stegvis på och man kan strax skjuta igen.

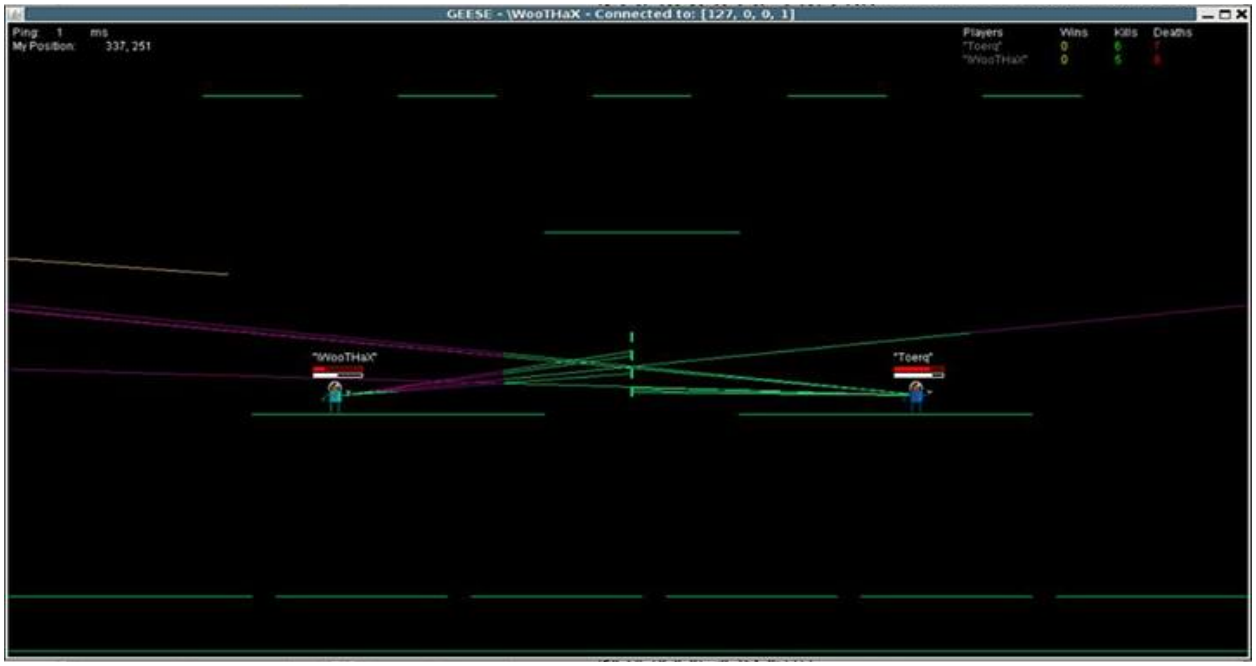

fig. 3.1 Bild av spelet då två spelare skjuter mot varandra

fig. 3.2 / 3.3 Information som finns uppe i vänstra respektive högra hörnen

fig. 3.4 En grafisk representation av en spelare

4. Slutsatser

4.1 Resultat

4.1.1 Server i Erlang

Vår server är skriven i Erlang. Den kan hantera en mängd olika anslutna klienter samtidigt. Servern har en *Coordinator-modul* (se rubrik 2.2.1) som håller reda på anslutna spelare samt aktiva spelsessioner. Det finns även en *Dispatcher-modul* (se rubrik 2.2.2) som accepterar anslutande klienter och sedan skapar en ny process som har i uppgift att kommunicera med klienten. Serverns viktigaste moduler är dessutom under *supervision* vilket ger en ökad

robusthet. Ett stort problem med servern är att om det finns fler än en aktiv spelsession förstörs spelens funktionalitet. Läs mer om detta i *4.2.5.*

4.1.2 Enkel grafiskt klient i Java

Klienten är skriven i *Java* och består av ett grafiskt användargränssnitt som innehåller ett menysystem och en grafisk representation av spelet *Pixel wars*. Menysystemet innehåller en lista över befintliga spelinstanser och ett antal knappar för menyval. Den grafiska representationen av *Pixel wars* bygger på data som hämtas från servern via *TCP*.

4.1.3 Enkel spellogik i Erlang

All spelhantering på serversidan är skriven i *Erlang*. Denna spelhantering består av tre moduler, en för att hantera alla actions som spelarna skickar, en som hanterar all logik i spelet och en som är själva spelet som itererar allt som händer.

4.1.4 Grupparbetet

Det har varit väldigt lärorikt att arbeta i projektform då man får möjlighet att utveckla sina programmeringskunskaper i kombination med att samarbeta mot ett gemensamt mål. Vi har upptäckt vikten av hur viktigt det är att distribuera arbete efter förmåga samt även hur snabbt tiden rinner iväg om man inte har någon bra struktur i arbetet. Mot slutet av projektet hade alla en egen roll när det gällde arbetsuppgifter. Någon kunde sitta med logik på servern medans någon annan kunde fokusera på klienten. Vi har även träffats varje dag i skolan vilket har lett till att om någon hade problem med något så fanns det alltid hjälp att få. Med anledning av begränsad modularisering i projektet samt ett ständigt förändrande *API* (*Application Programming Interface*) till både server och klient har det varit särskilt tacksamt att träffas fysikt i vårt grupparbete. Då *API:t* på både server- och klient-sidan har ändrats regelbundet har detta försvårat sammanvävningen av klient och server, men tack vare att vi ofta har suttit i samma rum har vi snabbt kunnat göra varandra medvetna om förändringarna.

4.2 Förbättringar

Vi haft många idéer, ett flertal som har implementeras men även en mängd idéer som har slopats. Den slopade mängden är dessvärre relativt stor och hade kunnat vara mindre om vi hade strukturerat arbetet på ett bättre sätt. På grund av en inledningsvis oklar systemarkitektur kombinerat med okunskap inom projektområdet var det svårt att få en effektiv uppdelning av problemen mellan projekgruppsmedlemmarna.

4.2.1 Game prediction

Game prediction (spelförutsägelse) innebär att klientapplikationen “gissar” hur spelservern kommer att agera på användarens input. Förutsägelsen som beräknats lokalt kan snabbt visas för användaren vilket gör att spelet blir mer responsivt och man får en bättre spelupplevelse. Ibland händer det att gissningarna är felaktiga och tillståndet i klientapplikationen är felaktigt i förhållande till det tillstånd som beräknats på servern. Frekvensen på korrekta gissningar ökar om svarstiden från servern är hög. Är svarstiden däremot låg resulterar detta i att felaktiga gissningar oftare görs vilket resulterar i att klienten måste korrigera den felaktiga gissningen. En lösning på en sådan korrektion är genom så kallad *rubberbanding* (en slags gummisnoddseffekt

som uppstår när en spelare flyttas från en punkt till en annan, även kallat *warping*) som helt enkelt synkroniserar användarens tillstånd i klientapplikationen med serverns tillstånd och flyttar spelaren till sin korrekta position. Ett exempel på hur *rubberbanding* skulle kunna fungera är om det finns en *kortaste-vägen-algoritm* implementerad. Låt oss säga att användaren spelar ett strategispel och vill flytta sin karaktär från punkt A till punkt B. Mellan punkt A och punkt B står en sten i vägen. Klienten tror att karaktären står lite till vänster om punkt A och kommer därför att välja den vänstra vägen runt stenen. Servern vet att karaktären står till höger om punkt A och kommer att låta karaktären färdas den högra vägen runt stenen. Klienten gör sin gissning men när karaktären står i höjd med stenen får den veta att karaktärens egentliga position är till höger om stenen vilket resulterar i att karaktären "teleporteras" till den egentliga punkten vilket kan upplevas som en störande faktor för användaren. Denna "störande faktor" är priset användaren får betala med *game prediction*. Den stora anledningen till att vi valde att inte ha med *game prediction* i vårt projekt är för att det kan lätt bli väldigt komplext. Detta kombinerat med att logiken måste finnas på både klient och serversidan. Det finns många realtidsspel som inte har *game prediction* implementerat. Ett exempel på dessa är sportspel där det är oerhört viktigt att anslutna klienter alltid har samma tillstånd.

4.2.2 Supervising av processer som skapas efter systemets initiering

Supervising av de viktigaste modulerna, *dispatcher, coordinator* och *coordinator backup*, initieras i samband med start av serverapplikationen. Processer som startas efter initiering har vi dessvärre inte lyckats *supervis:a*. De processer som startas efter initiering är instanser av *player*- och *table-modulen* samt en *spelinstanser*. Vikten av att ha dessa processer under *supervision* är inte lika hög som *supervision* av dem tre vi startar vid initieringen, men det hade givit servern en ökad robusthet. Dessvärre har vi inte lyckats med *supervision* av dessa processer. Anledningen är okänd då det har fungerat på trivial kod.

4.2.3 Interpolation

När svarstiden från servern är hög kan spelet upplevas som långsamt och oresponsivt. En hög svarstid kan röra sig mellan 50-100 ms. Om svarstiden blir ännu högre skulle vårt spel vara i stort sett ospelbart, men det finns lösningar för att göra klienten mindre beroende av svarstiderna från servern. En sådan lösning skulle kunna vara att implementera *interpolation* som innebär att klientens tillstånd alltid är minst ett steg bakom servern. När klienten får ett nytt tillstånd kommer exempelvis alla karaktärer att röra sig mjukt mot sin nya koordinat. Med *interpolering* kommer spelet alltså att kännas som mer flytande.

4.2.4 Låta servern skicka tillstånd med en fördefinerad frekvens

När klienten har ritat ut det senaste tillståndet kommer den att fråga servern om ett nytt tillstånd. Om meddelandet tas emot av servern kommer den sedan att, om allt går väl, skicka tillbaka det senaste tillståndet som beräknats på servern. Fördelen med detta är att det är en simpel lösning som är enkel att implementera. Nackdelen är däremot att tiden det tar att få ett tillstånd blir längre. En bättre lösning hade varit att för varje klient som är ansluten skapa en egen *Erlang-process* som skickar det gällande tillståndet med en fördefinerad frekvens, ofta kallad *tick-rate*. Klienten kommer sedan att läsa sin *TCP-inkorg* om något nytt tillstånd har anlänt och

därefter rita ut det senaste. Detta designval hade med största sannolikhet ökat responsiviteten i spelet.

4.2.5 Mnesia

När en spelinstans startas på servern så skapas en *Mnesia*-databas för att hantera alla spelares *actions. Mnesia* är inte optimal för att spara temporär data som skrivs över i en hög frekvens hela tiden då mensia har en log för varje insättning av data. En spelare skickar runt 30 *actions* i sekunden, vilket leder till prestandaproblem vid ett högt antal spelare.
Det som skulle kunna ersätta *Mnesia*, ifall tid fanns, skulle vara en *ETS* (Erlang built-in term storage) databas som är konfigurerad för *concurrent*-writes. Mnesia bygger på *ETS* och *DETS* (Disk-*ETS*) men har dessutom ett flertal extra egenskaper tex. sina loggar som egentligen inte behövs för detta projekt.

Mnesia valdes eftersom det var enkelt att använda, ett problem som uppstod var dock att konfigureringen av *Mnesia* inte fungerade helt rätt. Vanligtvis ska alla *entries* i en *Mnesia*-databas ha samma namn som den databas-instansen. En *entry* är i vårt fall en *action*, men olika instanser av spel ska ha egna instanser av *Mnesia* fast fortfarande innehålla samma typ av *entries.* Detta kan man ställa in i *Mnesia* med en flagga om vad för *entries* som ska användas, men av okänd anledning fungerade detta inte i vårt fall så vi har bara en *Mnesia*-instans vilket leder till att vi inte kan ha mer än en aktiv spelinstans.

4.3 Sammanfattning

Syftet med vårt projekt var att skapa en spelserver som skulle kunna hantera en större mängd spelare samt ett flertal spelsessioner. Målsättningen var att implementera en generell server för att på så sätt möjliggöra utbytbara spelmoduler.

Vi ville även att servern skulle vara så generell som möjligt. Det vill säga att man skulle kunna byta ett spel mot ett annat.

Även om vi bara har skrivit ett spel skulle vi ändå enkelt kunna byta vårt spel mot ett nytt. Däremot skulle vi inte kunna hantera att ha två olika spel på servern samtidigt som klienten hade kunnat välja mellan. Hade vi haft någon månad till på oss hade vi ökat modulariteten och implementerat möjligheten att låta användaren kunna välja mellan olika spel när denne skapar en ny spelinstans. Det största problemet som kvarstod blev i slutändan databasproblemet (läs 4.2.5). Några ytterligare mindre problem var avsaknaden av *supervising* på de processer som initierades efter serverns uppstart (läs 4.2.2). Mot slutet av projektet låg fokusen mest på att få vårt spel att fungera någorlunda bra med poängräkning, omstarter och så vidare. Engagemanget från gruppmedlemmarna var stort under denna fas och många idéer föreslogs.

Appendix: Installation och utveckling

Versioner

- Java
    - Kompilering och exekvering: Java version 6 (1.6) och 7 (1.7)
    - Dokumentation: Javadoc version 1.7
    - IDE: Eclipse 3.8.1 och 4.3.2
    - Externt paket: Jinterface 1.5.6
- Erlang
    - Erlang version R15B01 (Eshell V5.9.1) och R16B (Eshell V5.10.1)
    - Redigering: Emacs 23 och 24
    - Automatiserad testning: EUnit
- Versionshantering av vår programkod har underlättats med hjälp av Git (Github)
- Dokument har enkelt delats via en gemensam mapp i Google Drive

Repository

Koden finns tillgänglig i vårt publika *repository:*
https://github.com/Toerq/OSM-Project

Dokumentation

Dokumentationsgeneratorerna vi har använt oss av i *Erlang* respektive *Java* är *EDoc* samt *JavaDoc.* För att generera dokumentationen behöver användaren stå i roten i projektmappen samt i ett terminalfönster skriva **"**`make doc`**"**. Därefter hamnar all dokumentation i *doc/html/erlang_server* samt *doc/html/java_client.*

Testning

Den testning som vi använder oss av på serversidan är *EUnit*. Vi har endast testat de servermoduler som vi ansåg ha varit i behov av testning. Detta gällde modulerna *coordinator* samt *table*. Vi ansåg att det inte fanns något behov av att testa de andra modulerna då de var triviala och genom att köra koden så blir de mer eller mindre tillräckligt "testade". På klientsiden genomför vi ingen testning alls. Anledningen till detta är att, även här, blir den största delen av koden "testad" under körning. Fel som upptäckts under körningar har vi enkelt kunnat åtgärda.

Kompilering och körning

Erlang

Erlang-koden kompileras genom att stå i roten av projektmappen samt skriva **"**`make geese_compile`**"** i en terminalruta. För att starta servern skriver man **"**`make server`**"** som även kompilerar *Erlang-filerna.* Efter att kommandot **"**`make server`**"** har körts startas ett *Erlang-skal* där användaren måste skriva **"**`geese_sup:start_link(<valfri port>)`**"** vilket också står som instruktion i terminalfönstret. När detta kommando har exekverats startar servern och klienter kan därefter ansluta.

Java

I undermappen *jar/* finns det en färdigkompilerad klientapplikation, *geese_client.jar*, som kan köras via kommandot **"**`make client`**"** då man står i roten av projektmappen. Om man själv vill kompilera klientapplikationen måste projektmappen *src/GeeseClient* importeras i Eclipse och kompileras därifrån.

Katalogstruktur

All *Erlangkod* finns i geese/src/ medans all *Javakod* finns i *src/GeeseClient*. Binärkod från Erlang lagras i ebin/. *Jar*-filen för klienten finns i *jar/.* Dokumentationen återfinns i *doc/html.*