Kid-MUD

Processorienterad programmering (1DT049) våren 2012.

Slutrapport för grupp 14

Erik Arnerlöv
900406-0092

Michael Bergroth
910123-4194

Magnus Lång
910409-2672

Mikael Wiberg
840526-0194

1 juni 2012

Innehåll

```
Name?> Gustav
Welcome to Kid-MUD!
You are on the bridge that connects the town to the outside
world. Beneath the bridge runs a mighty river and to the south
you see a large waterfall pouring down huge amounts of water
from the tall mountain. The sound of the waterfall is almost
deafening. To the north you can see the river flowing towards
the great sea. To the west you have the town and to the east you
see the road continues to follow along the foot of the mountain.
Here stands a Frog
There are exits to east, west
?>
```

Figur 1: När du loggar in

1 Inledning

Ett *MUD* (Multi User Dungeon) är ett textbaserat flerspelarspel där spelare utför handlingar genom att skriva dem (såsom ”go north” eller ”attack Goblin”). Världen är uppdelad i *zoner*, och varje spelare befinner sig i en av världens zoner åt gången, och kan förflytta sig till intilliggande zoner.

1.1 Mål och Syfte

Syftet med projektet är att konstruera ett *MUD* med fokus på samtidighet för att kunna skala på multiprocessorsystem. Målet är att bygga ett fullt fungerande *MUD* med så många funktioner som vi hinner med.

1.2 Avgränsningar

Vi valde att inte implementera ett Telnet-interface trots att detta är en kännetecknande komponent hos ett *MUD* eftersom vi ville lägga fokus på samtidigheten och grundläggande funktionalitet. Telnet är ett gammalt protokoll för att ansluta till en terminal över en nätverksanslutning.

2 Kid-MUD

När man loggar in i systemet anger man namnet på den karaktär man vill spela. Om karaktären inte finns skapas den. Därefter anländer man till en zon. När man anländer till en zon får man en beskrivning av hur zonen ser ut, vilka andra spelare och icke-spelare, NPC:er, som står där samt vart man kan gå (se Figur 1). Denna information kan man få upprepad för sig genom att skriva ”`look`”. Man kan också få endast raden om vart man kan gå genom att skriva ”`exits`”.

När man väl är i en zon finns det en del saker man kan göra. Man kan till exempel skriva "`say Meddelande`" för att säga saker till andra spelare i samma zon. Man kan även prata privat med en godtycklig spelare genom att skriva "`tell Vem Meddelande`". Man kan också attackera spelare och icke-spelare genom att skriva "`attack Vem`", och för att sluta slåss skriver man "`stop`", eller väntar på att offret dör. Om man vill uppskatta sina chanser mot en icke-spelare kan man använda kommandot "`consider Vem`". När man inser att man aldrig kan vinna striden är det lämpligt att fly. För att gå till en annan zon skriver man "`go Väderstreck`". Man kan bara gå i de riktningar som antyddes av "`exits`".

3 Programmeringsspråk

Uppgiften gick ut på att göra ett program som använde sig av samtidighet. Eftersom vi hade fått lära oss om Erlang tidigare i kursen kändes Erlang som ett bra val av språk att skriva programmet i. Ingen i gruppen hade någon erfarenhet av Erlang sedan tidigare.

Fördelarna med Erlang är att det är väldigt lätt att skriva kod som utnyttjar samtidighet och den enkla meddelandepassningen. De lätta processerna i Erlang fungerar väldigt bra till sättet vi valde att lösa problemet på.

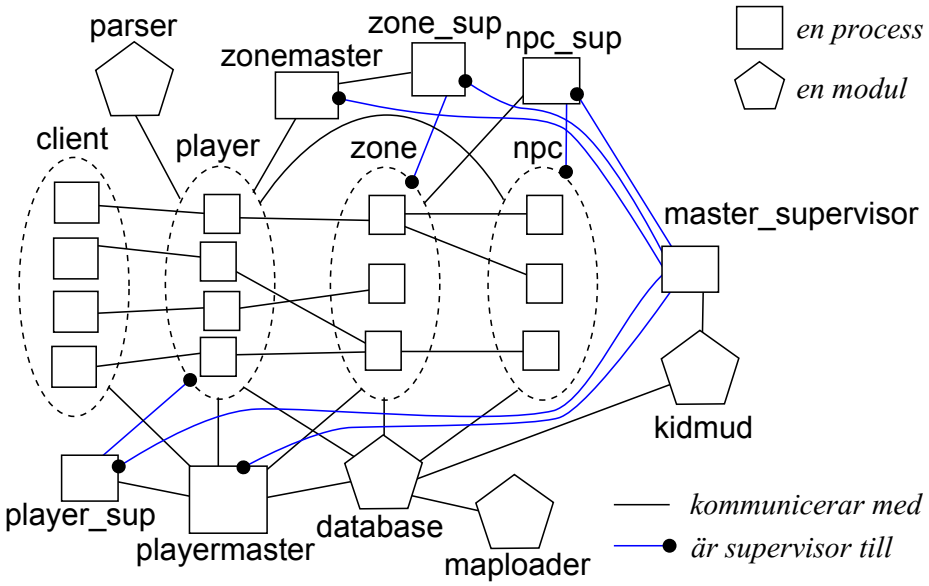
Vi har även använt oss av Erlangs Open Telecom Platform (OTP). Det är en samling av bibliotek och designprinciper för Erlang som i vårat fall har underlättat arbetet. De delar av OTP:n vi har använt är *Mnesia*, *supervisor* och *gen_server*. *Mnesia* är en databas som har varit mycket behändig för våra datalagringsbehov. *Supervisor* och *gen_server* är *beteende-moduler* som kan betraktas som basklasser i objektorienterad programmering.

En modul kan använda sig av ett beteende genom att exportera förbestämda funktioner, och sedan skicka sitt namn till beteende-modulen, och kallas då för en *callback-modul*. *Supervisor* abstraherar processhantering genom att starta, starta om och stoppa processer enligt deklarationerna för sina *callback-moduler*. *Gen_server* abstraherar en process som skickar och tar emot meddelanden och tillåter programmeraren att fokusera på det som är unikt för sin implementation.

Nackdelarna med Erlang är främst att meddelandepassande kod är svår att enhetstesta och felsöka. Mer om detta i avsnitt 8 på sidan 8.

4 Systemarkitektur

I Figur 2 på följande sida kan man se hur modulerna kommunicerar (svart) och är uppdelade i supervisor-träd (blått). Supervisor-träd är en del av OTP-designprinciperna och är processer, organiserade i ett träd, som ansvarar för att hålla sina barn igång. För att starta ett program startar man dess rot-supervisor, eller *master supervisor*. Utöver rot-supervisorn innehåller vårt projekt tre dynamiska supervisors, som inte har några barn vid uppstart, men som lägger till och tar bort dem dynamiskt, *npc_sup*, *player_sup* och *zone_sup*. Nedan beskrivs alla modulers uppgifter.



Figur 2: Moduler som kommunicerar och supervisorträd

4.1 Moduler

client

Denna modul startas av en spelare som vill spela och skickar vidare kommandon skrivna av spelaren samt tar emot meddelanden om vad som händer och skriver ut dem på skärmen.

player

player är den modul som ligger emellan klienten och själva systemet. Alla kommandon som spelaren skriver skickas från client till player som sedan tolkar kommandot med parser och utför kommandot genom att skicka vidare meddelanden till andra moduler. player håller även reda på spelarens status, såsom om man är i en strid och spelarens liv.

playermaster

När en spelare loggar in ber client playermaster om att få logga in. Om det accepteras startas en player-process med hjälp av player_sup som ansluts till client. Har även en lista över alla inloggade spelare och har funktioner för att skicka meddelanden till alla vid speciella händelser samt att vidarebefordra privata meddelanden.

player_sup

En dynamisk supervisor för alla player-processer.

zone

zone tar hand om det som rör en specifik zon. Den har data såsom närliggande zoners ID och listor över spelare och NPC:er. Det är zonen som skickar ut meddelanden till andra spelare som säger vad som händer. När en zon startas upp startar den alla NPC:er i zonen via npc_sup.

zonemaster

Modulen har koll på vilka zoner som är aktiva, dvs zoner som körs i en egen process. Om en spelare vill gå till en inaktiv zon startar zonemaster den zonen via zone_sup.

zone_sup

En dynamisk supervisor för alla zone-processer

npc

npc har hand om information om och simulerar specifika NPC:er.

npc_sup

En dynamisk supervisor för alla npc-processer.

parser

Omvandlar ett kommando från spelaren i text till en form (atomer och tupler) som lättare förstås av player.

database

Gränssnitt mot databasen för systemet som lagrar information om zoner, spelare och NPC:er. Erbjuder lättanvända metoder för att läsa, skriva och söka i databasen. Sköter även att uppdatera databasen med karta (via maploader) och NPC-typer.

kidmud

Sköter uppstart av servern genom att initiera databasen samt starta master_supervisor.

maploader

Modul som kan läsa en karta från en fil och ladda in den i databasen.

master_supervisor

Rot-supervisor för playermaster, zonemaster, player_sup, zone_sup och npc_sup.

5 Samtidighet

I Erlang används meddelandepassning för att kommunicera mellan processer. Det är väldigt lättanvänt och vi slipper att fundera på synkronisering med delat minne och liknande problem. Utifrån designprinciperna ur OTP:n delas meddelandepassningen mellan processer upp i två olika typer.

Den första typen är *calls*, de ska användas när en process vill ha ett svar på sitt meddelande. Detta garanterar att processen som vi kommunicerar med ska skicka tillbaka ett svar direkt efter den har hanterat meddelandet.

Den andra typen är *casts*, de ska användas när en process inte vill ha ett svar från processen som den skickar sitt meddelande till. Detta leder till att synkronisering kan hanteras på ett smidigt sätt när processer pratar med varandra. Vilket i sin tur leder till snabbare och mer robust kommunikation.

I våran nuvarande struktur har vi designat koden och modulerna på ett sätt som ska förhindra att dödlägen ska uppstå. Vi använder oss av enkelriktad kommunikation när processer kommunicerar med calls, detta gör att vi inte kan få några cirkulära vänteproblem.

6 Algoritmer och datastrukturer

De datastrukturer som främst använts är Erlangs tupler och records. Även listor har används av till exempel *playermaster* för att hålla reda på inloggade spelare, och *zone* för att hålla reda på vilka spelare och icke-spelare som är inloggade. I *zonemaster* används istället ett BBST (balanserat binärt sökträd) ur Erlang-modulen *gb_tree* för att kunna slå i aktiva zoner efter id, detta för att förbättra dess prestanda.

Records är en mycket vanlig notation i funktionella språk, som fungerar ungefär som en tupel, men där varje element, *fält*, ges ett namn, och kan plockas ut ur record:en genom att man uppger dess namn. Records är praktiska eftersom de går att spara i en Mnesia-tabell, och för att man kan lägga till fält utan att existerande kod slutar fungera. Exempelvis används syntaxen ”`State#state{players=UpdatedPlayers}`” flitigt, det betyder ”State men med fältet players satt till UpdatedPlayers”. Utan records skulle man istället skrivit en tupel med tiotals element där alla utom ett är detsamma som innan.

När det gäller algoritmer har vi inte implementerat något mer komplicerat än att traversera elementen i en lista, eller raderna i en fil, och utför en handling på varje, dessa främst i *zone* och *maploder*. Övriga algoritmer har uteslutande inbyggda funktioner använts till, främst ur modulerna *gb_tree* och *lists*.

När det gäller *parser* är den en wrapper runt en automatiskt genererad parsermodul *parser_grammar* som genereras av Neotoma[1], ett parsergeneratorbibliotek, ur filen ”parser_grammar.peg”. Detta innebär att algoritmen för att tolka spelarkommandon skrivs automatiskt. Grammatiken beskriver de kommandon som man kan skriva och hur de skall representeras i Erlang-data. Exempelvis är kommandot ”kill” definierat som följande:

[1] github.com/seancribbs/neotoma

```
attack_command <- ("attack " / "kill ") .* ‘
[_, Letters] = Node,
{attack, unicode:characters_to_list(Letters)}‘;
```

Första raden definierar vad som kan tolkas som ett ”attack_command” och det är antingen ”attack ” eller ”kill ” följt av vad som helst. De följande två raderna är Erlang, och säger hur det skall representeras i Erlang-data. Här binder vi andra delen av kommandot (vad som helst-biten) till Letters, och omvandlar det sedan till en sträng, och ger slutligen ett svar på formen `{attack, Namn}`.

7 Förslag på förbättringar

Autoförslag Istället för att behöva skriva all text fullständigt kan det vara behändigt att lägga till ett system som fyller i resten av ett t ex. kommando som är halvt skrivet. Detta är bara tillgängligt om det endast finns ett unikt ”slut” på den text man skriver.

Nedstängning av server När man inaktiverar servern så skall alla spelare som är inloggade få ett meddelande att server håller på att stängas ner och därför kommer de loggas ut om x antal sekunder. Sedan sparas världen och användaren kan ansluta till servern och komma till där dom var precis innan servern gick ner.

Grafiska förbättringar Olika ”typer” av text skall vara i olika färger så det blir lättare att läsa, även att man alltid ska kunna se hur mycket liv man har genom att det alltid står längst ner i textrutan för att underlätta strider mellan spelaren och andra.

Inloggningslösenord Varje spelare får välja ett lösenord som de skall använda för att logga in. Detta för att undvika att oönskade personer får tillträde till karaktärer de inte får använda.

Färdighetssystem För att göra allt mer intressant skulle ett system där spelaren kan utvecklas och lära sig nya färdigheter. Exempel på färdigheter skulle kunna vara att slå snabbare med ett svärd eller lära sig att använda en yxa på ett effektivt sätt. Detta kommer göra att strider blir mer intressanta och mindre enformiga.

Personifiera karaktärer För att undvika att alla spelare ser ungefär ”likadana” ut så skulle man kunna lägga till en ålder vilket t ex medför att man får högre *Intelligence* vilket kanske leder till att man har lättare för att använda magi.

8 Reflektion

Detta projekt har framförallt gett oss en fördjupning i Erlang men även hur det är att arbeta i grupp. Vi valde projekt genom att först utföra en brainstorming

och sedan bestämde vi oss för de tre idéerna som vi tyckte var mest intressanta. En stor fördel med att göra ett *MUD* är att det är väldigt lätt att skala, alltså lätt att relativt snabbt få en fungerande grund men även lätt att bygga vidare på.

Vi valde att sitta tillsammans när vi jobbade eftersom det är väldigt smidigt om man behöver hjälp med något eller kanske behöver diskutera ett förslag på t.ex. en förändring.

I början delade vi upp arbetet så att varje person fick var sin modul att fokusera sig på. Vi hade förbestämt vad varje modul skulle innehålla för funktionalitet. Problemet med detta var att vissa moduler krävde mindre arbete än andra, vilket ledde till att ibland så blev det en del väntande på att en viss modul skulle bli klar. En förbättring på detta skulle kunna vara att lägga lite mer tid på att uppskatta hur lång tid olika delar tar att implementera. När grundmodulerna var klara satte vi oss ner och brainstormade om vilka nya funktionaliteter vi ville implementera. Vi valde sedan ut de funktioner som vi tyckte var viktigaste för ”milestone 1”. När den var färdig gjorde vi samma sak inför ”milestone 2” osv.

Det som har vart svårt med projektet är att debugga och göra testfall. Detta eftersom att det mesta körs *samtidigt* och när man får ett fel så kan det vara svårt att lokalisera precis vart felet uppstår. Detta eftersom att det inte finns någon så kallad *stack trace*, vilket gör att man bara ser vilken modul som har krashat. Problemet där kan vara att den modul som krashade står inte själv för kraschen utan någon annan modul kan vara delaktig i det hela.

Testfallen blev lite av våra ”ruttna ägg” eftersom de flesta blev långa, komplicerade och beroende av flera andra funktioner/moduler. T.ex. ett av testfallen startade upp en hel server, lade in ett antal zoner bara för att testa en funktion.

De vi tyckte blev bra är bland annat användningen av *gen_server*, vilket gjorde att programmeringen blev mer abstrakt.

Slutresultatet av projektet är vi väldigt nöjda med. Vi fick in den funktionalitet vi ville och allt funkar som det ska. Projektet har flutit på väldigt bra under de veckor vi jobbade med det, vi har inte upplevt att vi fastnat på något som vi inte har kunnat komma vidare från.

9 Installation och fortsatt utveckling

De versioner av Erlang som använts är R15B01 och R15B. Koden finns att hämta på GitHub[2], och kan hämtas, om man har Git, med kommandot ”`git clone git://github.com/margnus1/kid-mud.git`”. Katalogstrukturen kan ses i Tabell 1 på nästa sida.

Systemet använder sig av Make så kompilering görs med kommandot ”`make`”. Enhetstester finns och använder sig av EUnit, dessa körs med kommandot ”`make test`”. Dokumentationsgenerering finns det också stöd av i form av samt EDoc-kommentarer. Den genereras med kommandot ”`make doc`”.

[2] github.com/margnus1/kid-mud

doc/src	Innehåller genererad dokumentation
doc/pdf	Innehåller rapporter i pdf-format
doc/src	Innehåller rapporter i källformat
ebin/	Innehåller kompilerad kod
include/	Innehåller headerfiler med typdeklarationer
neotoma/	Innehåller parser-generatorn Neotoma.
priv/	Innehåller datafiler som används vid körtid
src/	Innehåller källkoden

Tabell 1: Katalogstruktur

Klienten körs i en Erlang-prompt och startas med kommandot "`make start_client`". Därifrån ansluter man till en server genom att skriva "`client:connect(Node).`" där "`Node`" är namnet på den Erlang-nod som kör servern, vanligtvis en atom på formen kidserver@Server där "Server" är namnet på maskinen som kör servern.

För att kunna köra en server så måste man först konfigurera en databas. Detta behövs bara göras en gång och utförs genom att skriva "`make setup`". Därefter så startar man servern genom att skriva "`make start_server`". Man kan från serverprompten ansluta till sin egen server genom att skriva "`client:connect().`".