Projekt Nuntii

Operating systems and multicore programming (1DT089)
Project proposal for group 3: Hampus Adamsson(900921-1492), Erik Andersson (860119-2936), John Davey (930213-0738), Per Albin Mattsson (900321-1613), Maria Svensson(920821-8041)
Version 2, 2014-06-06

Innehållsförteckning:

1. Inledning

Att kommunicera via datorer har blivit allt vanligare i dagens samhälle. Det vanligaste sättet att kommunicera via datorer är genom chattprogram. Det finns ett flertal att välja på, dock är de flesta chattprogram grafiskt avancerade och kräver en hel del resurser.

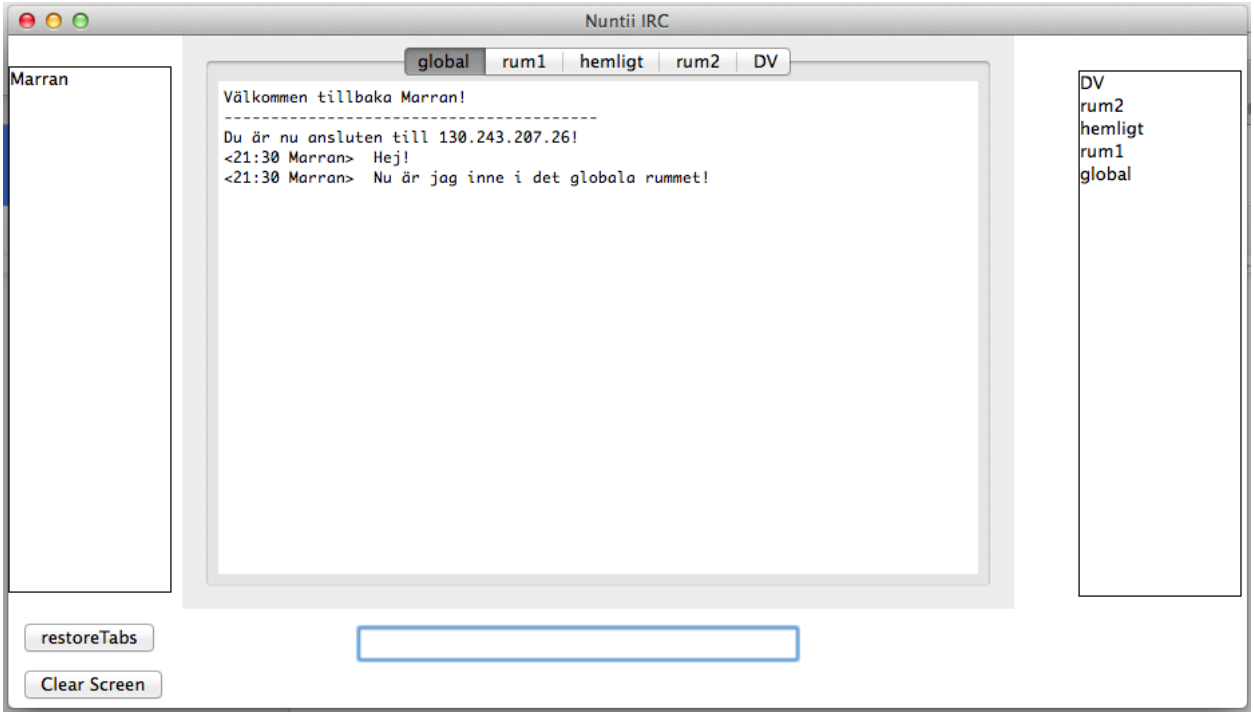
Det har visat sig vara svårt att använda äldre datorer för dessa moderniserade chattprogram därför har vi valt att ta fram ett chattporgram som alla kan använda sig av. Vårt chattprogram; Nuntii, är ett resurssnålt samt grafiskt förlåtande chattprogram.

2. Översikt över systemet

När användaren startar systemet så läser programmet in all information från config-filen som finns i programmappen. Om någon sådan fil ej existerar så skapas en standard config-fil. Beroende på inställningen så ansluter GUI't automatiskt till servern och etablerar en socket, alternativt att man får göra det manuellt.

Användaren är nu ansluten till det globala rummet, det rum som man automatiskt går in i vid anslutning och inte kan gå ur. Nu när användaren är ansluten till servern så kommer meddelande denne skickar konverteras till binär form och skickas via TCP socketen.

Användaren har möjlighet att skicka meddelanden till alla klienter som är anslutna till servern. Till vänster finner användaren en kolumn där klients användarnamn tydligt syns beroende på vilket rum man befinner sig i. Till höger finner användaren en lista på alla rum som existerar i servern.



<u>Bild 2.1</u> En tidig version av klienten i anslutet läge. Notera att klientens användargränssnitt utgörs av en lista på användare, rummet i fråga samt alla tillgängliga rum på servern.

Användaren har möjlighet att ange dessa kommandon:
Alla dessa kommandon förutom “clear schreen” skickas som förfrågningar till servern.

- /join [rumnamn] - Om rummet inte existerar sedan tidigare så skapas det angivna rummet och sätts automatiskt till public. Om det existerar sen tidigare så skickas en förfrågan om att gå med
    - /join [rumsnamn] private - Flaggan private skickar till servern att rummet du vill skapa skall vara hemligt. Finns rummet sedan tidigare som ett private rum, kommer du bli nekad till att få delta i detta rum och måste bli invitad för att gå med. Finns detta rum som public så kommer du läggas till i det angivna rummet.
    - /join [rumsnamn] public - Se /join [rumsnamn]
- /invite [användarnamn] - Användaren kan välja att använda detta kommande för att lägga till en existerande klient till det rum som användaren befinner sig i.
- /exit [rumsnamn] - Går ur det angivna rummet
- /info - Användaren får information om rummet du står i antingen är public eller private.
- /connect [IP] - Försöker upprätta en anslutning till den angivna IP-adressen.
- /clear - Rensar chattfönstret från all text
- /rename [namn] - Byter användarnamn till namn
- /track [namn] - Ger info om vilka rum namn är med i
- /whois [namn] - Ger info om namns IP-address och port som den är ansluten med

Användaren kan avsluta systemet genom att klicka på krysset på rutan. Då kommer socketen och fönstret att stängas ner.

2.1 Systemdesign - klient

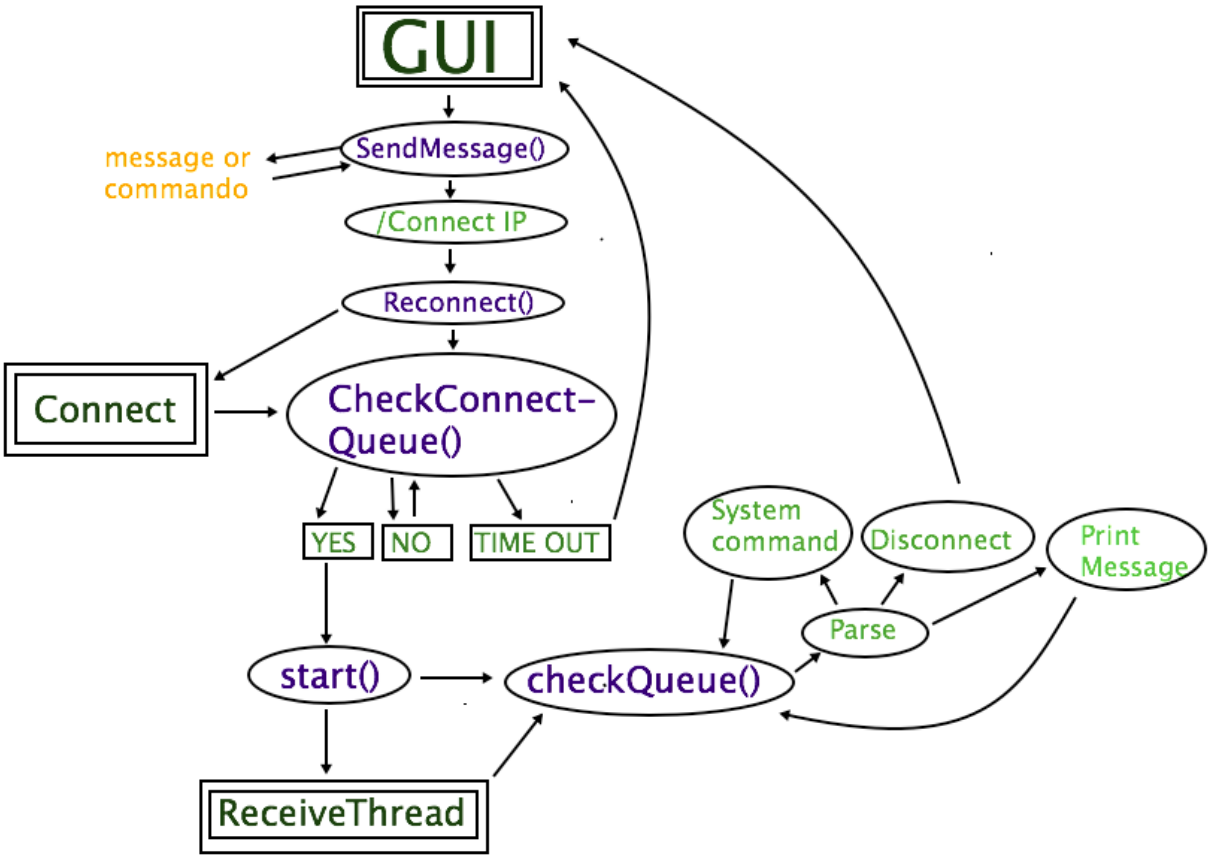


Bild 2.1.1 *Systemdesign för användargränssnittet .*

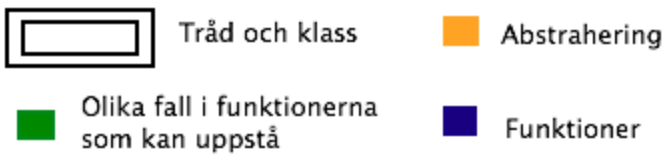


Bild 2.1.2 *Notation för systemarkitekturens uppbyggnad.*

(OBSERVERA att "message or commando" är en abstrahering (en förenkling av bild 1.2), se på bild 1.2 för tydligare beskrivning)

Vid systemets start kommer användaren ha möjlighet att ansluta till en server. Det görs via /connect [IP] som ligger i sendMessage.
Funktionen Reconnect startar en ny tråd och exekverar klassen Connect som försöker ansluta till IP adressen man angav med /connect [IP]. Dessutom så initierar Reconnect CheckConnectQueue som periodiskt hämtar information från Connect.

Det finns tre fall som kan uppstå. Det första fallet är när en connection lyckas. Då körs start() som skapar en ny tråd och exekverar klassen ReceiveThread samt initierar checkQueue().
Fall två uppstår när en anslutning misslyckas och ett återanslutningsförsök sker.
Det sista fallet uppstår om en återanslutning har misslyckats det antal gånger vi har angivit. Då måste man på nytt göra /connect [IP].

RecvThread har som uppgift att kontinuerligt ligga i receiveläge och ta emot alla meddelanden från servern. RecvThread innehåller en FIFO-kö och så fort vi har mottagit ett meddelande så läggs detta in i kön.
Var 50:e millisekunder så försöker checkQueue att poppa FIFO-kön i RecvThread. Om kön är tom returneras “empty”, annars returneras datan man mottog från servern och parsas.
Datan kan antingen vara ett systemkommando, en disconnectnotifiering eller ett meddelande.
Om det är ett systemkommando så utförs kommandot och är det ett meddelande så skrivs det ut i respektive chattfönster.
Om datan är en disconnectnotifiering så innebär det att man förlorat anslutningen till servern och man måste göra /connect [IP] på nytt

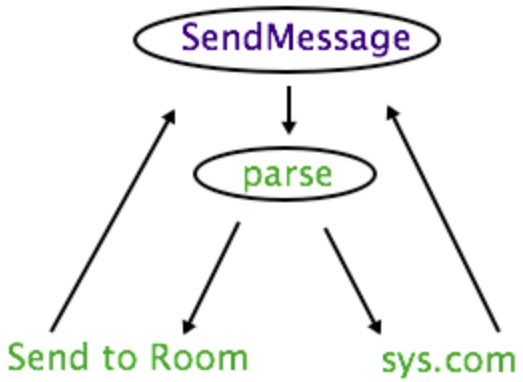


Bild 2.1.3 En utveckling av abstraheringen “message or commando” från bilden ovan.

Om anslutning finns och en användare har skrivit in ett kommando i textfältet så kommer funktionen SendMessage() att kontrollera om om det antingen är ett kommando (sys.com) eller ett vanligt meddelande som skall skickas till ett rum.

2.2 Systemdesign - Server

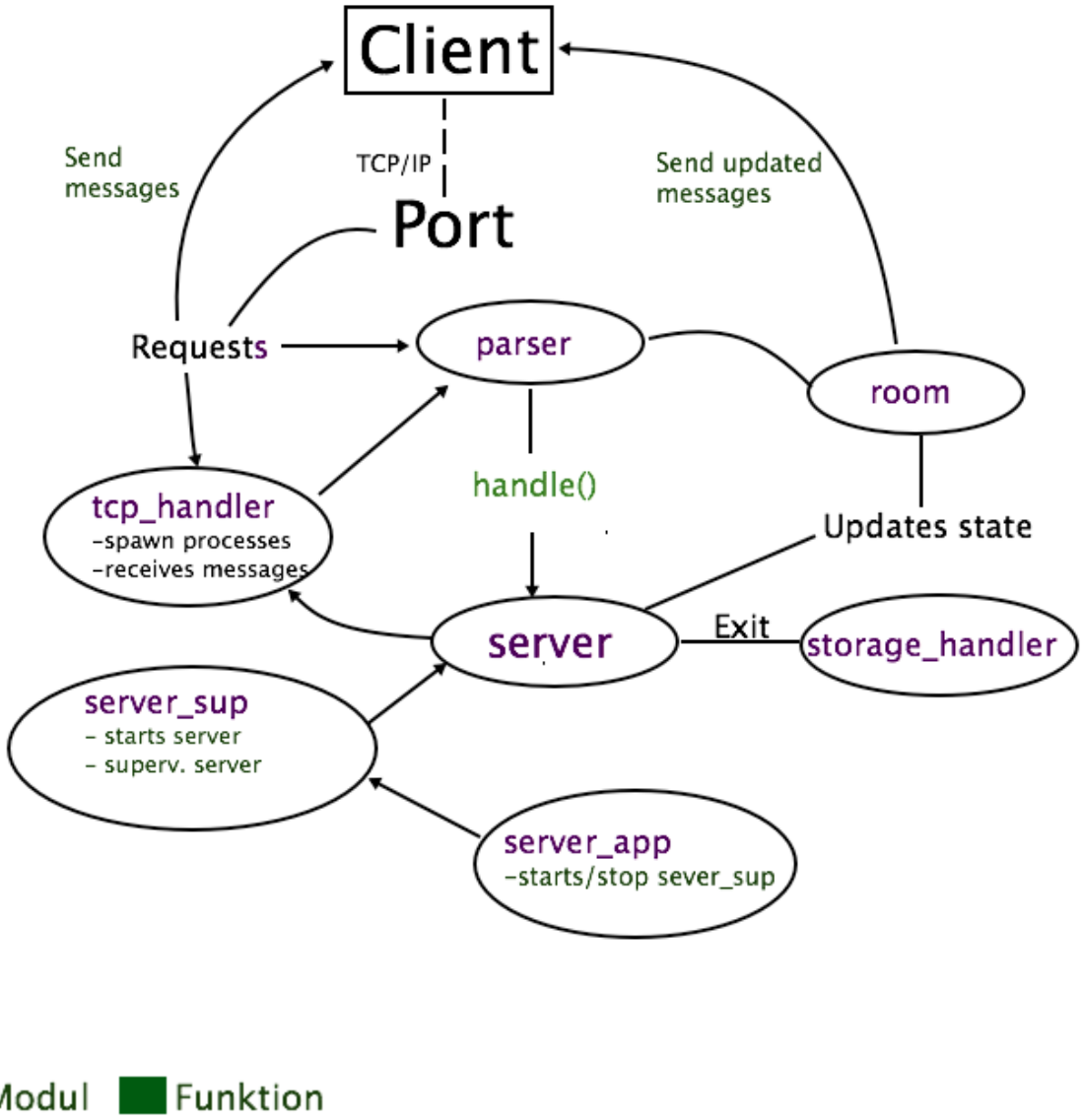


Bild 2.2.1 *Systemarkitektur för servern.*

I **tcp_handler** lyssnar vi på användare som försöker ansluta och skapar en process för varje anslutning som “accepteras”. När vi får ett meddelande från användaren i fråga skickas det till modulen **parser** som i sin tur skickar förfrågningar om uppdateringar eller meddelanden till modulen **server**. Från server skickas sedan antingen ett meddelande ut till varje socket som ska ha det, eller så uppdateras datastrukturen med hjälp av modulen **room**. Modulerna **Server_sup**, **server_app** och **storage_handler** är alla passiva och används vid

start/stop/krasch för att underlätta och säkerställa driftsäkerhet, mer om detta i underrubriken 3.4.

3. Implementation

3.1 Server - bakgrund

Servern - eller 'backend' - är implementerad i Erlang. Programspråkets robusta och paralleliseringsinriktade natur gjorde valet tämligen enkelt då projektets slutprodukt är en stabil och parallell chatserver. Erlangs OTP-bibliotek möjliggör dynamisk uppdatering och tillåter funktionsanrop att operera på ett s.k. "tillstånd". Det måhända kontraintuitiva beteendet som inte är karaktäristiskt för ett funktionellt programspråk är emellertid fundamentet till dess styrka. Då andra språk förlitar sig till sidoeffekter blir Erlangs returvärden mycket säkrare, framförallt tackvare testdriven utveckling. Dessutom utgår synkroniseringsproblem då OTP-standarden dikterar att endast en process har tillgång att modifiera tillståndet samtidigt. Med allt detta i åtanke, kombinerat med möjligheten att skapa processer med ett enkelt funktionsanrop, blir Erlang ett ypperligt val som programspråk till en chatserver.

3.2 Server - exempel

Vi använder oss av gen_server; ett hjälpande så kallat "behaviour". Det gör det enkelt för oss att behålla "state" i vår datastruktur, alltså spara de ändringar vi gör. Som datastruktur har vi en lista. Där sparas som sagt all information, dvs. alla rum och användare. Om vi t.ex. har ett öppet rum som heter global och en användare som heter Tom så skulle listan se ut såhär:

[{"global", [{Socket1, "Tom"}], false}]

Om vi skulle lägga till en användare som heter Tova som också är med i det privata rummet Room1 så skulle listan se ut såhär:

```
[
{"Room1", [{Socket2, "Tova"}], true},
{"global", [{Socket2, "Tova"}, {Socket1, "Tom"}], false}
]
```

Socket1 och Socket2 är två olika anslutningar från klienten representerade som användare.
Att ha en lista rullandes på servern är dåligt med avseende på optimering då funktioner som hanterar en lista tar lång tid. Vår server är alltså inte lämplig för hundratals anslutningar. Vi insåg detta i ett sent skede och valde att fortsätta som vi hade börjat för att hinna göra ett färdigt program istället för att riskera att inte hinna klart.

3.3 Server - concurrency

Huvudprocessen - servern - initieras föga oväntat direkt vid uppstart. Serverns första agerande är att skapa processen som accepterar inkommande anslutningar varpå servern återgår till att invänta funktionsanrop. *Låt oss kalla huvudprocessen/servern **<1>** och den sekundära processen/accept **<2>***. När en klient ansluter så är det **<2>** uppgift att förse klienten med en uppkoppling till servern och möjligheten att kommunicera vidare med servern. Detta sker genom att **<2>** placerar anslutningen i en separat process **<3>** vars enda uppgift är att betjäna den nya uppkopplingen. Händelseförloppet återupprepas för varje ny anslutning och antalet processer kommer aldrig att vara mindre än antalet anslutna klienter. Parallelliseringen i servern yppar sig på så sätt att alla klienter har direktkontakt med sin indviduella process på servern.

Eftersom rum - *eller listor med användare* - är kommunikationmediumet och storleken på dessa växer och sjunker dynamiskt med antalet klienter finns ytterligare en implementation av parallellisering utarbetad enbart för att hantera meddelanden. Huvudprocessen används i minsta möjliga utsträckning för att undvika den redan uppenbara “flaskhalls” som följer med att nämnd process har ensamrätt på tillståndet - *listan med användare och rum*. Således skapas en ny process vilken, givet meddelande och rum, skickar samtliga meddelanden och tillåter servern att fortskrida innan detta har genomförts. Eftersom alla användare i ett godtyckligt rum - *låt oss kalla mängden användare i rummet för **x*** - förväntar sig att se alla meddelanden i rummet så krävs **x** antal gen_tcp-operationer för att delge samtliga klienter med meddelandet. Servern skapar alltså en process vars arbete växer linjärt mot **x** samtidigt som arbetsbelastningen på servern aldrig överskrider skapandet av processen och “kostnaden” att skicka argumenten.

Serverns tillstånd växer som bekant med antalet anslutna klienter och antalet rum varje användare befinner sig inuti. Således kommer alla operationer på tillståndet att bli mer resurs-/tidskrävande då fler användare ansluter och entrar rum. Eftersom en godtycklig serveroperation - *exempelvis då en användare entrar ett rum* - kommer att låsa hela tillståndet även för rum som denna operation inte påverkar, blir resultatet en innefektiv tillståndshantering av servern. Alternativet hade varit att dela upp servern i ett godtyckligt antal servrar vars tillstånd utgörs av enskilda rum. Detta skulle resultera i att ett godtyckligt rum inte påverkas av andra rum i lika stor utsträckning som tidigare (*se bild 2.5)*. Ett ytterligare alternativ är att arbeta mot andra tillstånd såsom databasen Mnesia[1]. *Mer om datastrukturen senare*.

[1] http://www.erlang.org/doc/apps/mnesia/

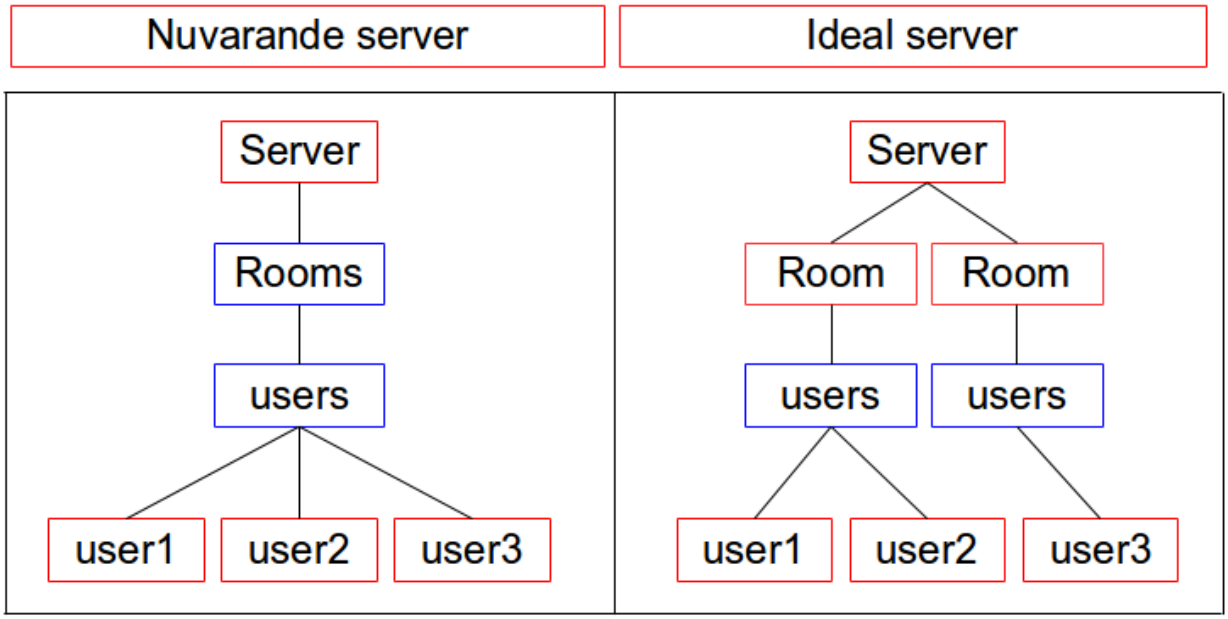
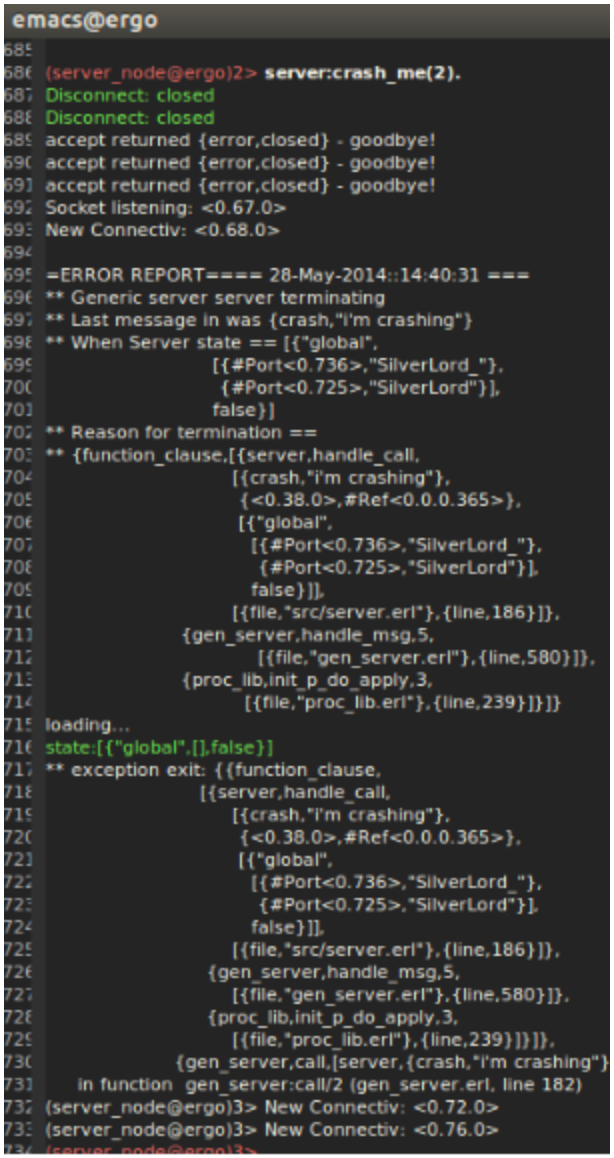


Bild 2.5 *Notera att rummen till höger utgör egna processer vars implementation är snarlik huvudprocessen. Den vänstra servern kommer att låsa alla tillgängliga rum vid serveroperationer medan den högra versionen inte gör det.*

3.4 Server - supervisors

Eftersom OTP-biblioteket har stort fokus på stabilitet finns en färdigimplementerad mall för hur processövervakning och uppstart/avstängning hanteras. Lösningen heter supervisors[2]; en fristående process vars primära syfte är att kontrollera andra processer. Beroende på konfigurering kan supervisors starta upp en kraschad process eller starta om en hel applikation - *möjligheterna är många*. Ett mål med vårt projekt är driftsäkerhet vilket gör processövervakning en ovärderlig fördel vid eventuella driftstopp. En applikation utgörs av en hierarki av moduler som tillsammans bygger upp standarden för OTP-systemet. **Servern** monitoreras av **server_sup** vilken startar upp **servern** utifall denne kraschar. För att möjliggöra en permanent avstängning används **server_app** som stänger både **server** och **server_sup**.

[2] http://www.erlang.org/doc/man/supervisor.html



När en krasch uppstår i huvudprocessen - *vi simulerar detta (bilden t.v) med ett felaktigt funktionsanrop* - så fångas först “exit-begäran” i en specialbyggd funktion i **server.erl**. Här termineras först alla anslutningar systematiskt och kontrollerat. Därefter kommer serverns tillstånd att sparas till disk för att återupptas vid nästa omstart. Serverns sista steg innan processen avlivas är att skriva en log till disk med information rörande tillståndet, felmeddelandet vid kraschen och en lista med anslutna klienter. När servern avslutningsvis har terminerat och **server_sup** noterar att processen den ombetts övervaka inte längre existerar så kommer ett “återupplivningsförsök” att påbörjas. Vid det här skedet av krasch-proceduren så följer servern samma tillvägagångssätt som vid en vanlig start.

Bild 2.6 *Notera att alla tillgängliga rum sparas vid krasch-tillfället och skapas automatiskt vid återupplivningsförsöket strax därefter. Det är emellertid upp till klienterna att ansluta till servern individuellt, vilket sker automatiskt i tillhörande klienten.*

Det finns många strategier för att starta om vid eventuell krasch. Att huvudprocessen startar om är emellertid oundvikligt, således kommer modulen server att startas om ovillkorligt varpå underprocesser till server kan hanteras mer dynamiskt. Den största potentiella felkällan - *baserat på empiri under projektets gång* - är de individuella processerna för att lyssna på inkommande trafik från klienterna. Vår lösning till detta är att låta dessa krascha utan att hantera problemet. Resultatet blir föga oväntat att klienterna förlorar kontakten med servern och måste etablera en ny anslutning. Anledningen till att vi valde detta framför att låta supervisorn återskapa processen var för att undvika att problemet som renderade första kraschen utför samma misstag en andra gång. Ponera att servern kraschar för att vi har för många anslutna klienter vid ett givet tillfälle, att återskapa exakt samma tillstånd med supervisorn hade således försatt oss i samma situation som före kraschen.

3.5 Klient - Inledning

På klientsidan valde vi att programmera i Python. Valet stod mellan Java och Python men valde Python på grund av att vi fick intrycket att det skulle vara ett relativt enkelt språk att programmera i. Dessutom är Python smidigt när det kommer till nätverkstillämpningar.

En nackdel med Python är att det är något svårare att göra grafiskt snyggt och dokumentgenereringen (PyDoc) var inte helt självklar.

3.6 Klient - Kommunikation

Klienten kommunicerar med servern med hjälp av en TCP-socket. Meddelandena termineras med ett '\n' tecken.

3.7 Klient - Concurrency

Majoriteten av vår concurrency ligger inte helt oväntat på serversidan, men det existerar även till viss del i klienten. Utöver maintråden som man kör GUI:t på så används en tråd som hela tiden ligger och lyssnar på socketen. Denna tråd hanterar all mottagning av meddelanden från servern. En separat tråd används även när klienten försöker upprätta en anslutning till servern.

3.8 Klient - Datastrukturer och design

I klienten har vi valt att mestadels använda oss av datastrukturen dictionary (en slags hashtabell) för att lagra och hålla ordning på alla objekt vi skapar. Tack vare dessa kan vi på ett mycket smidigt sätt lagra alla objekt vi skapar och få åtkomst till dem i konstant tid vid behov.

Vidare, genom att använda sig av en egen tråd i klienten som hela tiden ligger och lyssnar på socketen så kommer huvudtråden som kör GUI:t aldrig att blockeras vid receive. Vi finner inga större nackdelar med att använda oss av en separat tråd, tvärtom, hade vi istället använt oss av polling eller dylikt i huvudtråden så hade det blivit mycket mer resurskrävande.

Kodexempel:

```
def addTab(self,name):
     tab = Text(self.master,state=DISABLED)
     self.windowList[name] = tab
     self.nb.add(tab, text=name)
```

Här skapas chattrutan tab och lägger in den i dictionaryt windowList med key:n name, där name är namnet på den tabb vi vill skapa. På detta sätt kommer vi få åtkomst till det chattfönster som

är förknippat med respektive tab i konstant tid. Vi kan på detta sätt hålla koll på alla fönster som för nuvarande är aktiva.

3.9 Klient - Felhantering och stabilitet

Klienten har vissa inbyggda verktyg för att hantera krasher. Om man förlorar anslutningen till servern så försöker klienten automatiskt återansluta X antal gånger med Y sekunders mellanrum. X och Y anges i configFilen. Misslyckas automatiska återanslutningen så uppmanas man ansluta manuellt till en server med hjälp av /connect [IP] kommandot.

När man återupprättat anslutningen till servern återställs även automatiskt samtliga konversationer man var med i och man kan fortsätta chatta. Har man inaktiverat denna funktion måste man manuellt gå med i rummen igen. Samtliga av dessa automatiska verktyg kan aktiveras/inaktiveras i configFile.

4. Slutsatser

4.1 Förbättringar som kan göras

Något som skulle kunna göras bättre i klienten är att optimera koden. Det är något som vi funderat mycket på men inte haft tid till. Att bryta ner koden i mindre moduler har också kunnat göras bättre.

Vi hade kunnat spara all information på servern i en databas istället för i en lista. Det skulle göra det möjligt att ha fler användare anslutna utan att göra programmet långsamt.

4.2 Lärdom av det genomförda projektet

Av detta projekt har vi lärt oss mer om: nätverksprogrammering, integrering av flera olika programmeringsspråk, förstått oss på hur klient och server hänger ihop, hur man skickar data över internet. Vi har fått en klarare bild av vad en server verkligen är och hur man kan bygga den.

Vi har också kommit längre i vår utveckling av samarbetsförmåga. Alla dagar är inte perfekta och man håller inte alltid med varandra men uppgiften kvarstår och det gäller att hitta ett sätt att få projektet att utvecklas ändå.

4.2 Svårigheter inom projektet

Några svårigheter som vi har stött på i Python är dokumentgenereringen. I början av projektet var även sockets en svårighet då det tog ett tag att förstå sig på det. De flesta problem som uppstod på klientsidan var inte speciellt svårta att förstå sig på, självklart har vi fastnat men oftast har det löst sig efter någon dags arbete.

Till en början var tanken med detta projekt att så fort som möjligt försöka få igång ett IRC liknande chattprogram där varje användare hade var en separat process. Skulle vi få det att fungera relativt tidigt skulle vi utveckla systemet så att användarna skulle kunna befinna sig i olika rum.

Vi lyckades uppnå våra mål, och hann utveckla systemet mer än det var tänkt från början. Självklart kan man utveckla systemet något ytterligare som; att kunna blockera en användare, adminrättigheter i rummen, kunna ha listor med sina vänner, samt implementera stöd för fildelning.

4.3 Vad vi hade gjort annorlunda om vi fick börja om

Vi hade utnyttjat GIT mer. Vi lärde oss att använda git branch på ett effektivt sätt först mot slutet vilket var synd. Hade vi gjort om klienten så hade vi istället valt Java framför Python då det hade varit mycket enklare att få till ett snyggare GUI. Vi hade möjligtvis gjort klienten lite mer “dum”; låtit servern ta hand om mer. Vi hade från början kunnat gå igenom mer i detalj vad servern respektive klienten skulle ha för uppgift.

Appendix: installation och utveckling

I Python har vi använt oss av versionen Python 3 och i Erlang har vi använt oss av Erlang/OTP17.

Koden finns tillgänglig på GitHub: https://github.com/hampusadamsson/osm/tree/master/projekt
Klienten ligger i mappen GUI och man startar programmet så här:

```
projekt/GUI$ python GUI.py
```

Alla moduler som utgör servern ligger i src. Vi kompilerar dessa med rebar. Rebar ska köras när man står i projekt för då läggs alla kompilerade, körbara binära filer i ebin. Man gör så här för att kompilera och starta servern:

```
projekt$ ./rebar clean compile
projekt$ erl -pa ebin
Eshell
1> application:start(server).
```

I projekt ligger även en mapp test där det finns testfiler för room och parser, som innehåller alla hjälpfunktioner som används av server. Testerna körs också med rebar och stöds av EUnit. Man skriver (flaggan -v ger mer information om testerna):

```
projekt$ ./rebar eunit -v
```

För dokumentation av servern använder vi Edoc, och de genererade html-filerna läggs i doc. För att generera dokumentationen kör:

```
projekt$ ./rebar doc
```