erlStream

Ett enkelt klient-serversystem för streaming av musik

Operativsystem och multicoreprogrammering (1DT089) våren 2014
Slutrapport för grupp 1

Jeanette Castillo (910929-4208)
Filip Hedman (931022-7476)
Robert Källgren (930405-0611)
Oscar Mangard (920820-4116)
Mikael Sernheim (920329-4179)

Version 2, 2014-06-04

Innehållsförteckning

1. Inledning

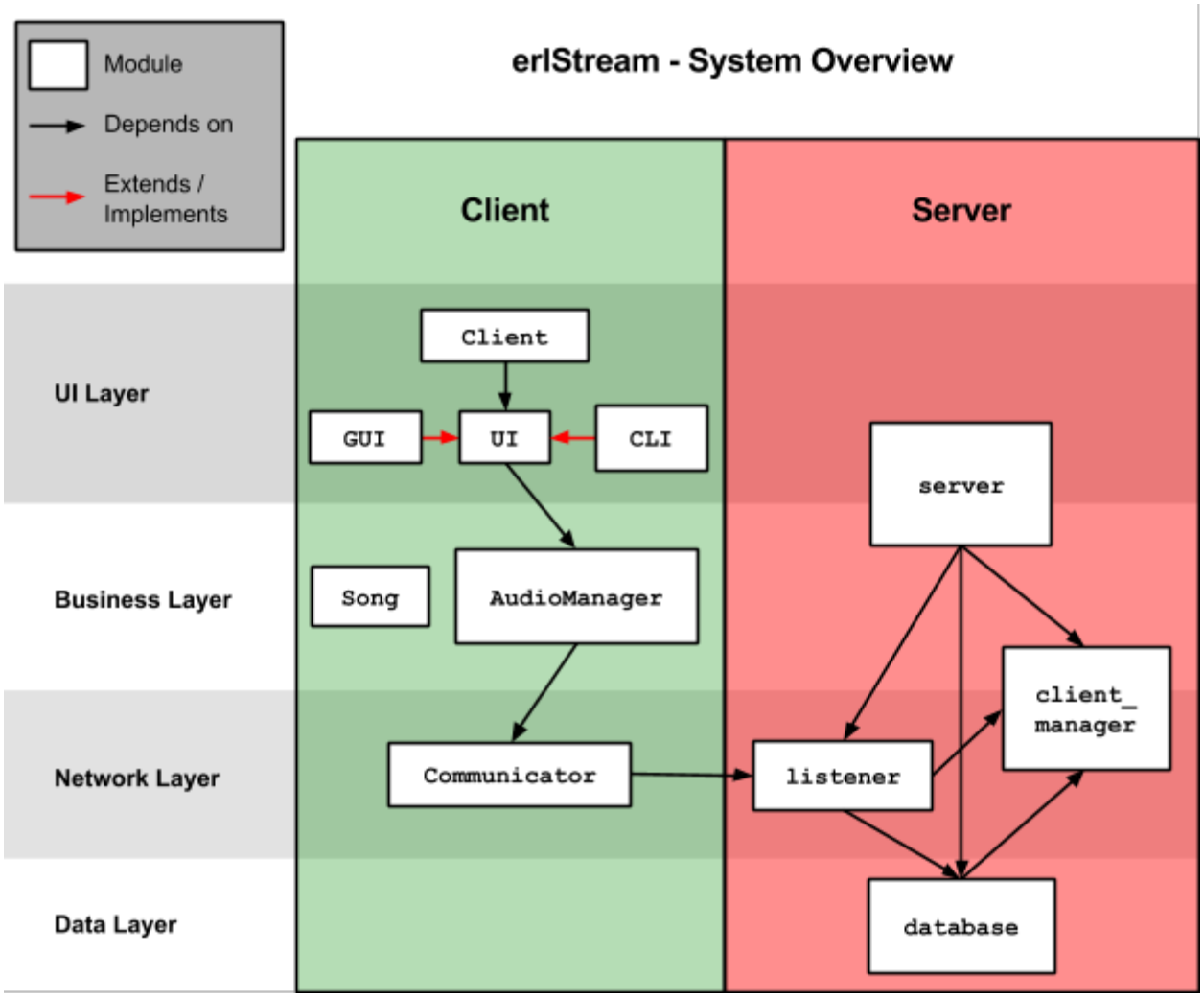
Nu när Internet växer som mest blir vi mer och mer vana vid de möjligheter det ger oss, och vi tar för givet att när som helst ha tillgång till oändligt med information. Streaming, att spela upp ljud- och videofiler samtidigt som de hämtas över Internet, är en metod som är en del av mångas vardag, men det är nog inte lika vanligt att veta hur det egentligen går till. Vi bestämde oss för att utforska detta och skapa ett system för att över Internet spela musikfiler i MP3-format från en server till klienter. Målet var att konstruera en server som kan hantera flera anslutna klienter samtidigt, och ett klientprogram med grafiskt gränssnitt och grundläggande funktionalitet.

2. Översikt över systemet

Klientprogrammet tillhandahåller två gränssnittsalternativ, ett textbaserat och ett grafiskt. För att använda det måste serverprogrammet vara igång. Detta kan köras lokalt på datorn, eller någon annanstans i världen där det finns tillgång till internet. I klienten kan användaren ange adress och port till servern för att upprätta en anslutning.

Programmet kan visa de tillgängliga låtarna på servern och erbjuder vanliga musikspelarfunktioner som bland annat play, pause, next, previous och sorteringsmöjligheter. Servern kan administreras genom ett textbaserat gränssnitt för att visa de befintliga låtarna och information om anslutna klienter. Om låtar tas bort eller läggs till i filsystemet uppdateras servern automatiskt, och de anslutna klienterna likaså. Om anslutningen bryts försöker klienten återskapa den i bakgrunden.

2.1 Systemdesign



Figur 1: Grov översikt över systemets arkitektur med de viktigaste modulerna. Lagren som syns till vänster är endast där för att demonstrera vilka områden modulerna arbetar med, men de har ingen påverkan i praktiken.

2.1.1 Server

Som *Figur 1* visar består servern av fyra huvuddelar. *server* är ingångspunkten till serverprogrammet och den modul som erbjuder användargränssnittet samt håller koll på de tre nyckelkomponenterna *listener*, *client_manager* och *database*. Dessa tre moduler samarbetar för att bilda en fungerande server. *listener* är den del som lyssnar på klienters anrop och hanterar dessa. *client_manager* innehåller information om de klienter som är anslutna, så att man enkelt kan skicka meddelanden till dessa. Slutligen finns *database*, som innehåller information om de MP3-filer som finns tillgängliga för klienter att spela upp.

På serversidan är concurrency betydande. *listener*, *client_manager* och *database* är separata processer och för att kunna hantera flera klienter samtidigt får varje klient dynamiskt en egen process tilldelad sig när den ansluter. På det här viset har alla klienter samma prioritet av servern. *database* har även en till process som med ett bestämt intervall kollar efter ändringar i katalogen som innehåller MP3, och sedan uppdaterar serverns information om dessa.

2.1.2 Klient

Klienten består av 7 huvudmoduler som enligt *Figur 1* kan delas in i tre lager. *Client* är ingångspunkten till hela klientprogrammet, den startar ett *UI* (User Interface) som användaren kan interagera med. Detta kan vara antingen grafiskt, *GUI* (Graphical User Interface), eller textbaserat, *CLI* (Command Line Interface). Funktionaliteten som dessa erbjuder är identisk, det är upp till användaren att välja. *UI* kommunicerar sedan med *AudioManager*, som är själva musikspelaren, det som sköter uppspelningslogiken. *Song*-modulen representerar en låt, och innehåller information som låtnamn, artist, album, och speltid. *AudioManager* använder in sin tur *Communicator* för att hämta data från servern. Denna uppdelning medför bra modularitet.

Det förekommer ett antal uppgifter som behöver skötas samtidigt och därmed kräver concurrency. Klienten måste kunna lyssna på meddelanden från servern medan den interagerar med användaren. Samtidigt behövs även en separat tråd för själva musikuppspelningen, för att förhindra blockering. För det grafiska gränssnittet behöver man också kunna uppdatera flera grafiska beståndsdelar under tiden, såsom förfluten uppspelningstid och vilken låt som spelas. Synkronisering är nödvändig för att kunna uppdatera låtinformation i *AudioManager* samtidigt som användaren interagerar med programmet. Vid uppspelning krävs även synkronisering av några variabler som används för att dela information mellan huvudtråden och tråden som spelar upp.

3. Implementation

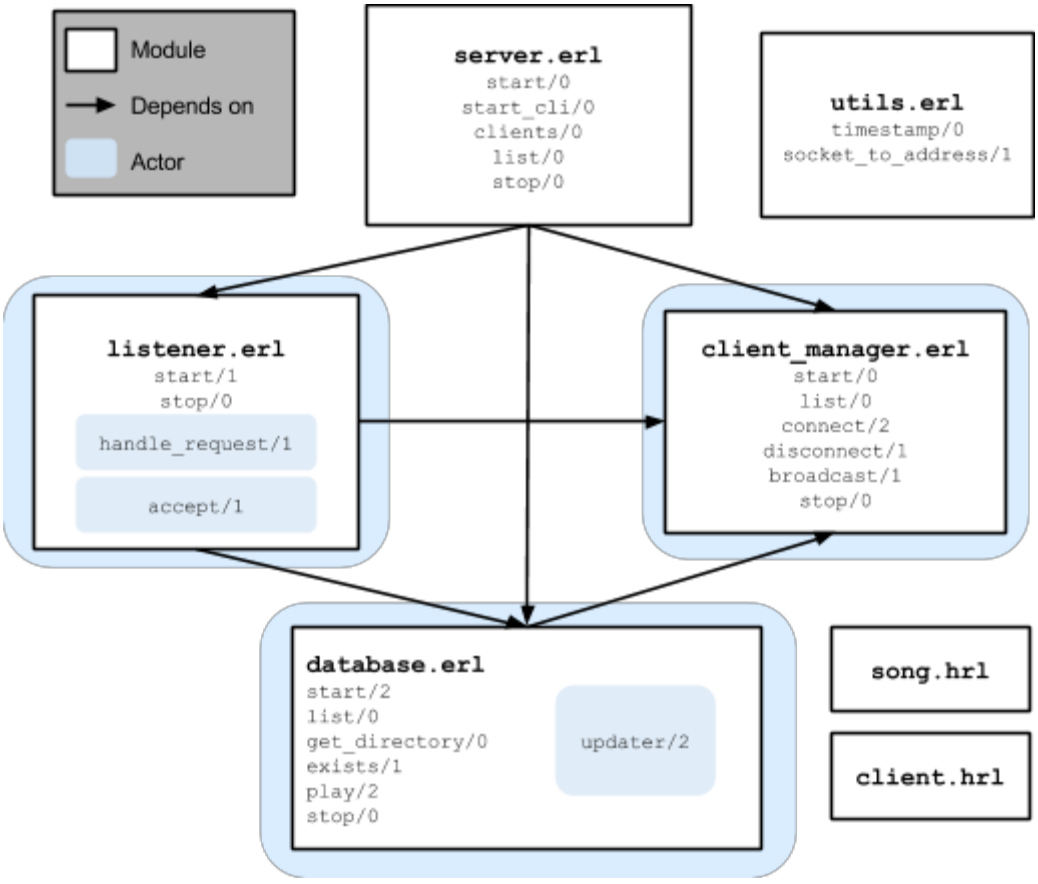
Det språk som valts för servern är Erlang, eftersom det erbjuder väldigt bra stöd för concurrency och nätverkskommunikation. Klienten är däremot skriven i Java, då det är enkelt att skapa grafiska gränssnitt med standardbiblioteket, och dessutom tillåter det att programmet kan köras på alla plattformer som stödjer Java. Kommunikationen görs via nätverksprotokollet TCP.

3.1 Server

Servern är skriven med Erlangs OTP-designprinciper i tankarna. Som beskrevs under avsnitt 2.1.1 består den av tre nyckeldelar. Dessa delar; *listener*, *client_manager* och *database*, använder sig av *gen_server*, som är en del av OTP och agerar som en allmän server. Denna metod är lämplig att använda när man har en resurs som ska delas mellan flera processer (aktörer i Erlang). *gen_server* abstraherar bort all message passing och tillåter användaren att fokusera uteslutande på den faktiska logiken.

database har en lista med information om de befintliga låtarna. Detta är en resurs som behöver vara tillgänglig för alla klienter och därför är det lämpligt att låta *database* använda *gen_server*.

client_manager har en lista med information om de anslutna klienterna och det är en resurs som behövs för alla tillfällen då klienterna behöver meddelas om något. *gen_server* gör det också lätt att placera in modulerna i ett så kallat “Supervision tree”, som är en möjlig framtida förbättring.



Figur 2: Komplett översikt av serverns moduler och deras API:n.

Figur 2 visar servern i sin helhet. Modulen *utils* innehåller små funktioner som används i flera moduler. *song.hrl* och *client.hrl* innehåller records *song* och *client* som används för att representera en låt respektive en klient. De blåa rutorna indikerar separata aktörer.

Servern kan startas med eller utan det textbaserade gränssnittet via *start_cli/0* respektive *start/0*. När den kör igång startas de andra komponenterna automatiskt.

3.1.1 Klienthantering

När Listener-modulen startas öppnas en lyssnings-socket på den port som anges av servern. En ny process skapas med funktionen *accept/1* som väntar på att klienter vill ansluta. Så fort någon gör det skapar den processen ytterligare en *accept/1*, och går efter det in i funktionen *handle_request/1* och tolkar meddelanden från klienten. Se *Figur 2* ovan för en visualisering av detta tillstånd. När fler klienter ansluts, skapas fler *handle_request/1*. Om det är en ny klient anropas *client_manager:connect/2* för att lägga till den i listan över klienter, och därmed göra den

tillgänglig för de andra modulerna. Om en anslutning stängs anropas *client_manager:disconnect/1*. En detaljerad förklaring till de kommandon som klienter kan skicka för att tolkas i *handle_request/1* kan läsas i avsnitt 3.3.

3.1.2 Filhantering

När *database* startas laddar den information om de existerande låtarna. Den förväntar sig att alla låtar som ska distribueras ligger i en katalog kallad “files” och är av MP3-format med en bitrate på 192 kbit per sekund. För att den ska veta vilka filer som ska läsas in används Erlang-funktionen *os:cmd/1* med argumentet *ls ../files | egrep '.*.mp3'*. Detta gör att servern endast kan köras på unixbaserade operativsystem.

Varje låt i databasen representeras som nämnt av ett record kallat *song* och innehåller fälten *title*, *artist*, *album* och *duration*. *title* fås av namnet på filen, medan *artist* och *album* hämtas från filens ID3-taggar[1] genom att manuellt läsa in rätt bytes i Erlang. *duration* räknas ut genom att läsa av filens storlek. För att den ska blir korrekt bestämdes det som nämnt att alla filer antas ha en bitrate på 192 kbit per sekund. Funktionen *database:play/2* tar in filnamnet på en låt och returnerar en tupel med *{ok, rå binär data}* om låten hittades. Det är denna data som sedan skickas till klienten, utan någon modifikation.

Som *Figur 2* visar startar databasen även en extra process som kör funktionen *updater/2*. Den kollar om någon fil har lagts till eller tagits bort i katalogen med ett bestämt intervall på tio sekunder. Det görs genom att köra en sträng-jämförelse av resultatet av *os:cmd*-kommandot som beskrevs tidigare, med resultatet från föregående intervall. Om något har ändrats laddar databasen in all låtinformation på nytt, och skickar ut den till alla klienter via *client_manager:broadcast/1*.

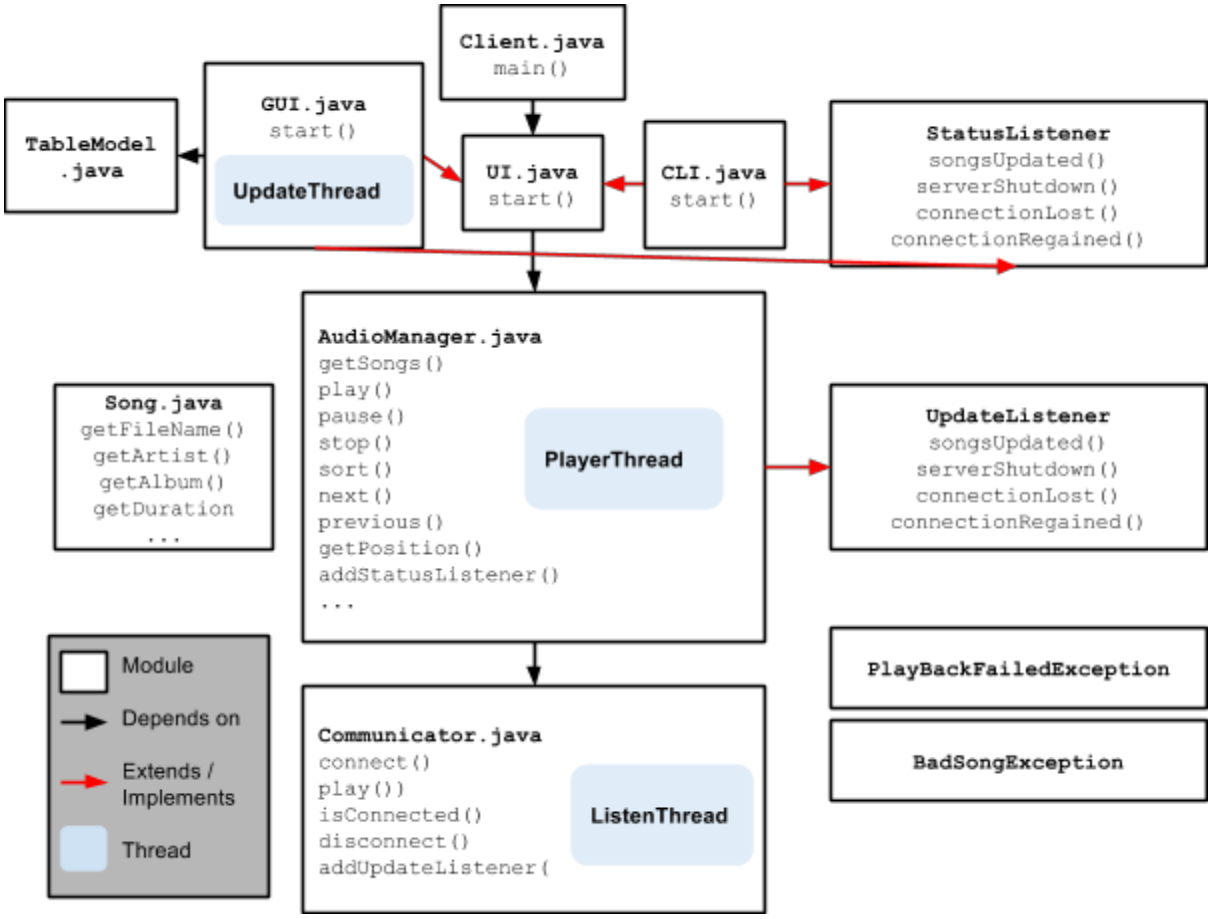
3.1.3 Concurrency

För servrar som hanterar stora mängder av klienter kan det vara för lite att bara ha en *accept/1*-aktör igång i taget. Det kan leda till att väntetiden blir onödigt lång och man skulle instället kunna ha några fler som lyssnar samtidigt.

Deadlocks kan inte uppstå tack vare Erlangs message passing.

[1] Metadata som finns i MP3-filer.

3.2 Klient



Figur 3: Komplett översikt av klientens klasser och de viktigaste delarna av deras API:n. För enkelhets skull visas inga argument.

Som Figur 3 demonstrerar består klienten av totalt 12 publika klasser, med ytterligare några privata för multithreading i *GUI*, *AudioManager* och *Communicator*. *Song* används av *AudioManager, Communicator, CLI och GUI.*

Klienten använder ett tredjeparts-bibliotek kallat *JavaLayer* från *Zoom* för att spela upp MP3-filer direkt genom ett *InputStream*-objekt. Som nämndes i avsnitt 3.1.2 skickar servern rå binär data, och denna data läses alltså in direkt via strömmen till spelaren. I övrigt tillhandahåller *JavaLayer* ganska magert med funktionalitet. Klassen *AudioManager* bygger ett större API kring detta som bidrar med alla uppspelningsmöjligheter som klienten förses med. Det är egenskaper som att pausa uppspelning, spela nästa/föregående låt, få information om låten som spelas för tillfället och hur länge den spelats. *JavaLayer* innehåller även en klass *PlaybackListener* som AudioManager ärver för att kunna veta när en låt börjar eller slutar, och därmed ha möjlighet att byta till nästa låt eller liknande.

Information om låtar sparas i objekt av klassen *Song* och lagras i en *ArrayList* i *AudioManager*. Varje *Song*-instans är en representation av en fil som finns i serverns databas. En sådan instans innehåller information om låtens längd, titel, artist och album.

Communicator använder sig av strängen *address* för att veta vilken server den ska kommunicera med, *port* är porten för att hämta från och skriva data till servern. Modulen läser av från servern vilka låtar som är tillgängliga och sammanställer en lista med *Song*-objekt som används i *AudioManager*. *Communicator* har också en klass *ListenThread* som bygger på *Thread*, en instans av denna körs i bakgrunden och läser meddelanden från servern. Om anslutningen skulle brytas är det också denna tråds uppgift att med ett bestämt intervall på fem sekunder försöka återskapa den.

De två användargränsnitten finns för att användaren ska kunna nyttja *AudioManager* på önskat sätt. *GUI* (modulen) är utvecklat så att det ska vara tilltalande och enkelt att använda. För att designa applikationen har biblioteken *javax.swing, javax.swing.event, java.awt, java.awt.event* och *java.awt.Color* använts. När GUI:t skapas startas en separat tråd som ständigt uppdaterar grafiken.

CLI implementerades före *GUI* för att testa och använda klienten i ett tidigt stadie. Detta gränssnitt är en implementation av hantering av strängar som användaren skriver in under körning. Biblioteken som används här är *java.util.Scanner, java.io* och *java.net*.

Client är den modul som sätter igång applikationen och startar antingen gränssnittet *GUI* eller *CLI*.

3.2.1 Observerare

Två kniviheter som dyker upp med designen är att *Communicator*-klassen borde kunna meddela *AudioManager* om uppdateringar från servern, utan att vara medveten om klassen *AudioManager*. Samma sak gäller då användargränssnittet ska kunna få uppdateringar från *AudioManager* utan att *AudioManager* ska behöva känna till klasserna *GUI* eller *CLI*. För just dessa problem kommer designmönstret med observerare in i bilden. Genom att skapa två Java interfaces som definierar en metod för varje typ av uppdatering kan då *UI* respektive *AudioManager* implementera dessa, och lägga till sig själva som observerare till *AudioManager* respektive *Communicator*. Här nedan följer närmare beskrivningar av hur dessa interface ser ut.

UpdateListener

Detta interface implementeras av *AudioManager* och ges till *Communicator* genom metoden *addUpdateListener().* Det används för att hantera när serverns databas har uppdaterats eller om anslutningen har brytits eller återskapats. De metoder som definieras är följande:

- `songsUpdated(List<Song> newSongs)`
- `serverShutdown()`
- `connectionLost()`

- `connectionRegained(List<Song> songs)`

När *Communicator* får en uppdatering från servern anropas då motsvarande metod på varje observerare.

StatusListener
Detta interface implementeras av *GUI* och *CLI* och ges till *AudioManager* genom metoden *addStatusListener()*. För tillfället ser det ut exakt som *UpdateListener*, men för framtidssäkerhet hålls dessa separerade.

- `songsUpdated(List<Song> newSongs)`
- `serverShutdown()`
- `connectionLost()`
- `connectionRegained(List<Song> songs)`

3.2.2 Undantag
Klienten innehåller klasser för två specialundantag som kastas om fel uppstår.

- *BadSongException* - Kastas från *AudioManager* om en metod anropas med ett *Song*-objekt som inte finns i låtlistan.
- *PlaybackFailedException* - Kastas från *AudioManager* om något gick fel i *JavaLayer*-biblioteket.

Utöver dessa kastas även två undantag från standardbiblioteket vid socketfel kallade *IOException* och *UnknownHostException*. Med hjälp av undantagen kan gränssnittet visa felmeddelanden till användaren.

3.2.3 Synkronisering
Synkronisering behövs på många ställen i *AudioManager* på grund av att uppspelningen körs i en egen tråd, samtidigt som *ListenThread* i *Communicator* kan anropa de metoder som implementerats från *UpdateListener*. *AudioManager* innehåller en mängd variabler för att spara tidpunkt för paus-anrop, start av uppspelning och liknande. Dessa variabler använder nyckelordet *volatile* för att försäkra att läsning och skrivning inte sker samtidigt från olika trådar. Det är en väldigt enkel och effektiv lösning för just de här fallen av två anledningar; skrivningar till variablerna beror inte på dess nuvarande värde och de agerar inte invarianter mot några andra variabler. Hade något av dessa kriterier inte varit uppfyllt hade *volatile* inte varit tillräckligt, utan då skulle det varit lämpligare med en synkroniserad metod för att ändra dess värde. Eftersom det handlar om enkla read och writes utan några lås finns det heller ingen risk för deadlocks, så det är ett väldigt enkelt fall av synkronisering.

För listan med *Song*-objekt räcker inte *volatile* på grund metoden *sort()*. När låtlistan uppdateras i bakgrunden med hjälp av *songsUpdated()*-anrop från *Communicator* måste den sorteras på nytt, och då behöver det försäkras att ingen försöker hämta listan med *getSongs()* samtidigt. Sorteringen bryter mot kriteriet att skrivningar inte beror på variabelns nuvarande värde, och anroparen till *getSongs()* riskerar att få en inkorrekt lista. Detta problem löses genom att göra både *sort()* och *getSongs()* till synkroniserade metoder, medan listan är *volatile*. Deadlocks kan inte uppstå här heller.

I *Communicator* finns det endast en boolesk variabel *connecte*d som behöver synkroniseras och det görs med hälp av volatile precis som tidigare. Implementationen av metoderna är noga genomtänkt; Som nämnt tidigare försöker *ListenThread* återskapa en anslutning om den tappas. Om detta sker samtidigt som *disconnect()* anropas i huvudtråden, ser huvudtråden till att vänta på att *ListenThread* avslutar sitt försök och stänger sedan anslutningen manuellt, för att försäkra sig om att *ListenThread* inte precis lyckades skapa en ny anslutning som blir igång i bakgrunden. För detta krävs ingen synkronisering, men det är ett ganska speciellt fall som är lätt att missa.

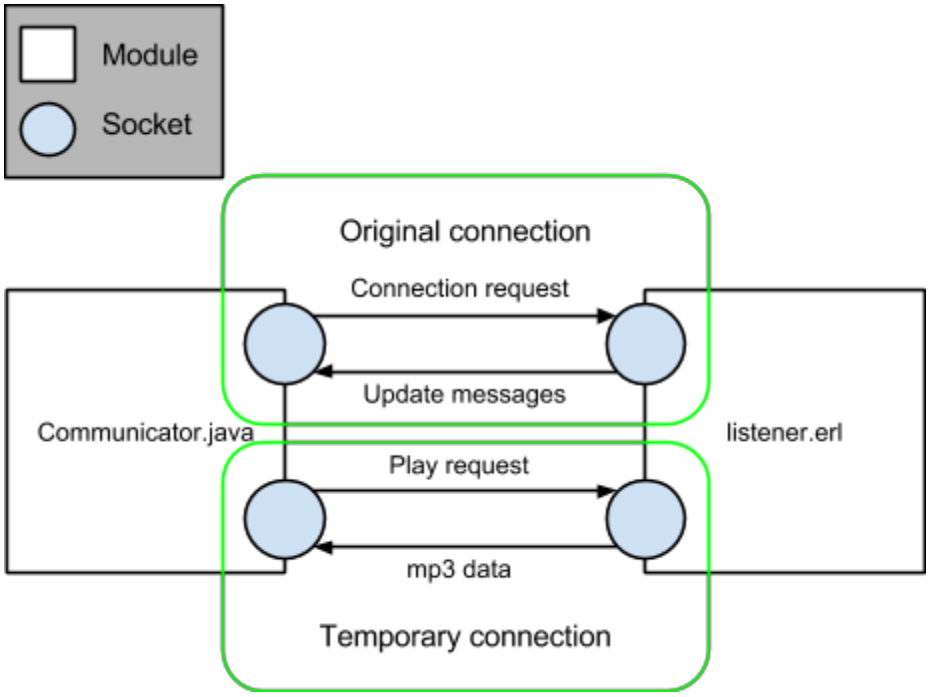
3.3 Kommunikation

Kommunikationen mellan servern och klienter görs helt och hållet genom nätverksprotokollet TCP.

3.3.1 Hur det fungerar i praktiken

En detaljerad beskrivning av alla tillgängliga meddelanden finns under avsnitt 3.3.3, men låt oss först gå igenom hur kommunikationen sker i praktiken.

En anslutning börjar med att klienten skapar en socket till servern och skickar ett meddelande med en förfrågan om anslutning. Servern tar emot meddelandet och returnerar information om de låtar som finns tillgängliga att spela upp. Klienten läser av dessa och bygger upp en lista med *Song*-objekt i *AudioManager*. Sedan startas *ListenThread* i *Communicator* (se *Figur 3*) som fortsätter läsa av denna socket för eventuella meddelanden från servern. När användaren vill spela upp en låt skapas en ny temporär socket till servern där en uppspelningsförfrågan skickas. Servern svarar med MP3-data för den låt som efterfrågades i samma socket. När uppspelningen är klar stängs anslutningen. Notera att allt sker i den nya socketen. Se Figur 4 för en visualisering.



Figur 4: En temporär anslutning skapas för uppspelningsförfrågningar.

Anledningen att uppspelningsförfrågningar inte görs i den första anslutningen är för att inte fördröja eventuella meddelanden från servern. Detta gör dessvärre att servern inte har någon koll över vilken klient som spelar upp vilka låtar, då inget kan visa att två sockets är kopplade till samma klient. Alltså kan vem som helst fråga efter MP3-data utan att ha bett om att bli ansluten först, så länge denna vet namn på någon låt som finns i servern.

Om serverns databas uppdateras skickas ett meddelande till klienten genom den första socketen med ny information om de tillgängliga låtarna. När användaren har lyssnat färdigt och stänger klienten skickas ett meddelande som berättar detta för servern. Även om användaren skulle avbryta processen på ett oväntat sätt kommer servern meddelas om det då socketen stängs, men det kan anses vara bättre etikett att skicka ett meddelande. Om istället servern skulle stängas ner medan klienten är igång kommer ett meddelande skickas till klienten genom den första socketen, och klienten kan då meddela användaren.

3.3.2 Varför TCP?

TCP valdes bland annat för att man har en anslutning, till skillnad från UDP där man kan skicka paket till vilken address som helst. Detta tillåter servern att ha bra kontroll över de anslutna klienterna, och den kan därmed enkelt skicka meddelanden till dem. Man är även försäkrad att alla paket kommer fram, vilket är viktigt när det gäller att spela upp ljud.

3.3.3 Meddelandeprotokoll

TCP är i grunden ett protokoll avsett för streaming, dvs ett kontinuerligt flöde av data. Det innebär att man för att skicka meddelanden på något sätt måste veta hur många tecken man ska läsa innan ett helt meddelande har hämtats. Det finns flera olika lösningar till detta. Ett vanligt sätt är att reservera ett antal bytes i meddelandets början för ange längden (som ett heltal), och sedan läsa så många tecken. Erlang tillhandahåller sätt att göra detta automatiskt, medan man på Java-delen måste göra det själv. Ett annat sätt är att använda ett bestämt tecken för att markera slutet på ett meddelande. I detta fall bestämdes att den senare metoden passade bäst. Alla meddelanden avslutas med tecknet ‘\n’, då det förenklar avläsningen på Erlang-sidan med hjälp av tupeln *{packet, line}* i TCP-inställningarna, och tillåter oss att använda metoden *readLine()* på Java-sidan.

Utöver att alla meddelanden ska sluta med ‘\n’ så har ett speciellt protokoll definierats som används för kommunikationen. Nedan följer en detaljerad beskrivning om vilka meddelanden som är tillgängliga:

Klient till server

Förfrågan om anslutning (*Name* är förinställt i klienten, för att servern ska veta vilket klientprogram som används):

```
connect:Name\n
```

Förfrågan om MP3-data (*Offset* anger hur många millisekunder in i låten man vill börja):

```
play:Offset:Songname\n
```

Meddelande om avstängning av klient (Gäller bara om det är i samma socket som *connect* anropades):

```
disconnect\n
```

Server till klient

Svar på förfrågan om anslutning:

```
connect:ok\n
Title1:Artist:Album:Duration\n
Title2:Artist:Album:Duration\n
...
TitleN:Artist:Album:Duration\n
end\n
```

Svar på förfrågan om MP3-data (lyckad):

```
play:ok\n
Data
```

Svar på förfrågan om MP3-data (misslyckad):

```
play:error:Reason\n
```

Meddelande om uppdaterad databas:

```
update\n
Title1:Artist:Album:Duration\n
Title2:Artist:Album:Duration\n
...
TitleN:Artist:Album:Duration\n
end\n
```

Meddelande om avstängning av server:

```
exit:Shutdown\n
```

Om servern tar emot ett okänt meddelande genom en ny socket ignoreras det och socketen stängs på serversidan. Om ett okänt meddelande tas emot genom en socket som har anslutits med *connect:Name\n* ignoreras det också, men socketen hålls öppen.

4. Slutsatser

Vi har alltså skapat ett system med följande funktioner:

Server

- Textbaserat gränssnitt för administration
- Hantering av flera klienter samtidigt
- Inläsning av ID3-taggar
- Automatisk uppdatering

Klient

- Grafiskt och textbaserat gränssnitt
- Grundläggande musikfunktioner
- Automatisk uppdatering
- Automatisk återanslutning om anslutningen bryts
- Modulär struktur som gör det lätt att bygga nya UI:n

Det här har varit vårat största projekt inom programmering hittills, så det har gett oss många nya erfarenheter kring samarbete, planering, kodstrukturering och allt som hör till. Vi har även fått en god inblick i hur nätverkskommunikation kan gå till i praktiken, vilket är en kunskap som kan vara mycket värdefull att besitta. I efterhand kan vi dock konstatera att fokus har legat betydligt mer på programmens arkitektur än själva streamingen, vilket vi inte trodde från början. Vi hade förväntat oss att streamingen skulle ta lång tid att få fungerande, men med hjälp av tredjepartsbiblioteket i kombination med TCP löstes det väldigt fort. Detta ledde till att projektet flöt på väldigt problemfritt, utan att fastna mer än enstaka timmar på något moment. Därmed fick vi också en

chans att försöka designa en riktigt bra struktur i både klienten och servern som gör det enkelt att bygga ut systemet med ny funktionalitet i framtiden. Tyvärr fanns det trots det ett antal saker vi inte hann med, dessa tas upp under nästa rubrik.

4.1 Möjliga förbättringar

- Testerna kan förbättras markant. Det var väldigt svårt att skriva tester med tanke på att det är ett nätverksprogram, som dessutom är starkt beroende av vilka filer som finns på servern.
- Synkroniseringen i AudioManager kan behöva tänkas igenom mer med avseende på den separata tråden som sköter uppspelning..
- Servern kan använda fler OTP principer såsom t.ex. Supervisors för att hålla bättre koll på processer.
- Vi hade kunnat använda ett annat bibliotek för uppspelning såsom *BasicPlayer* (http://www.javazoom.net/jlgui/api.html). Detta hade gett oss mer koll på buffring, vilket vi för tillfället inte har överhuvudtaget.
- Vi skulle kunnat skapa en riktig Erlang OTP Application av servern.
- Strukturen av klienten kan förbättras med paket för att hållas bättre till Javastandarden och göra katalogstrukturen mer organiserad.
- Prestandan i servern skulle kunna förbättras på vissa ställen. Funktionen *client_manager:broadcast/1* är inte optimal i den meningen att den blockerar anrop till t.ex. *connect/1* och *disconnect/1*, som används av *listener*, tills meddelandet har skickats ut till alla klienter. Om det är många klienter och ett långt meddelande kan det ta tid.
- Man hade kunnat lägga till fler metoder i *StatusListener* såsom *songChanged(Song)* för att notifiera UI:t om att en ny låt börjar spela och liknande.
- Vi borde ha definierat variabler istället för att använda “magiska värden” bland annat i *database* och *Communicator* för uppdaterings- och återanslutningsintervallen.
- Vi borde ha skapat en till Exception-klass som kastas från *play()* i *Communicator* om ett felmeddelande fås från servern. För tillfället returneras bara en tom *InputStream*.
- GUI:t kan finslipas på många ställen, t.ex. ändras inte playknappen till pausknapp om man trycker next eller previous när en låt är pausad.
- *database:start/2* hade kunnat tagit in en lista med inställningar för att bestämma bitrate på låtar, uppdateringsintervall mm.
- Vi hade möjligen kunnat använda Erlangs file-modul istället för *os:cmd* för att lista innehåll i mappar, och därmed göra servern körbar på andra system än bara unix.
- Dokumentationen av servern kan förbättras med exempel, mer ingående förklaringar mm.
- Metoder i klientens klasser borde kodas mer defensivt för god praxis (kolla efter null-argument och liknande).
- *database* kunde använda en *dict* för att binda filnamn till *song*-records för snabb åtkomst.

Appendix: Installation och utveckling

Utvecklingsverktyg:

- Erlang version R14B04
- Java version 7
- Emacs och Sublime Text för kodning
- EUnit för tester av servern
- JUnit för tester av klienten
- EDoc för dokumentation av servern
- JavaDoc för dokumentation av klienten
- Make för automatiserad kompilering, generering av dokumentation och testkörning
- Git via GitHub för versionshantering
- Trello för planering
- Google Drive för rapportskrivning

Information om JavaLayer-biblioteket finns här:
http://www.javazoom.net/javalayer/javalayer.html

Koden tillsammans med instruktioner för att köra systemet finns här:
https://github.com/Vertig0/erlStream

Katalogstruktur:

- `client/` - Innehåller alla filer som rör klienten
  - `client/bin/` - Innehåller all grafik för GUI:t samt kompilerade .class-filer
  - `client/doc/` - Innehåller genererad dokumentation för klienten
  - `client/libs/` - Innehåller JavaLayer-biblioteket samt två bibliotek för JUnit
  - `client/src/` - Innehåller all källkod för klienten
- `server/` - Innehåller alla filer som rör servern
  - `server/doc/` - Innehåller genererad dokumentation för servern
  - `server/ebin/` - Innehåller kompilerade .beam-filer
  - `server/files/` - Innehåller .mp3-filer som servern distribuerar
  - `server/include/` - Innehåller .hrl-filer som inkluderas av serverns moduler
  - `server/src/` - Innehåller all källkod för servern
- `doc/` - Innehåller projektförslaget, slides från slutpresentationen samt den här projektrapporten

De tillgängliga make-kommandona är föjande:

- `all` - Kompilerar, kör tester samt generar dokumentation för både servern och klienten
- `server` - Kompilerar servern
- `client` - Kompilerar klienten
- `start_server` - Kompilerar och startar servern

- `start_client` - Kompilerar och startar klienten med grafiskt gränssnitt
- `start_client_cli` - Kompilerar och startar klienten med textbaserat gränssnitt
- `jar` - Skapar ett körbart jar-arkiv av klienten för att enkelt flytta mellan olika system (Notera att mappen client/libs/ innhållande jl1.0.1.jar måste ligga i samma sökväg som jar-filen vid körning. För att skapa en enda jar-fil som innehåller alltihop kan ett program som t.ex. JarSplice (http://ninjacave.com/jarsplice) användas)
- `test` - Kör tester för både servern och klienten
- `test_server` - Kör tester för servern
- `test_client` - Kör tester för klienten
- `doc` - Genererar dokumentation för både servern och klienten
- `doc_server` - Genererar dokumentation för servern
- `doc_client` - Genererar dokumentation för klienten
- `clean` - Tar bort alla kompilerade filer och genererad dokumentation