CHALMERS | GÖTEBORGS UNIVERSITET

En jämförelse av Java och Erlang för nätverksbaserade verktyg

Kandidatarbete inom Data- och informationsteknik

Alexander Sjösten
Christoffer Persson
Håkan Ullström
Johan Olofsson
Johan Sjöblom
Oscar Utbult

Institutionen för Data- och informationsteknik

CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2012
Kandidatarbete/rapport nr 2012:23

Abstract

This report aims to investigate a number of technical solutions and programming languages that might be suitable to work with when developing administrative tools for Internet usage. The primary requirements for such a tool is security and reliability.

The main focus is on two programming languages, Java and Erlang. Their virtual machines, syntax, standard libraries and fault tolerance are some of the aspects that are presented. Different solutions for cryptography, memory management, network topologies and different ways of communicating in parallel systems are also described. Based on this study, two systems have been implemented, one in Java and one in Erlang. Both of these have been named Confector, Latin for "executor".

The two implementations have been developed from a common specification so that the comparison of the systems can be of interest. However, the systems have not been constrained concerning structures or how specific problems are to be solved. The comparison between the systems are on the overall development process, performance and scalability.

Based on the study no general solutions were achieved. Both Java and Erlang proved to be well suited for use in such a system.

In the future, it might be interesting to study and compare two more complex implementations of the systems. This could include the use of different network topologies and study how they work in different situations.

Sammanfattning

Denna rapport har för avsikt att utreda tekniska lösningar och programmeringsspråk som kan vara lämpliga att använda vid utveckling av ett administrationsverktyg som ska användas över internet. De huvudsakliga kraven för verktyget är säkerhet och tillförlitlighet.

Huvudsakligt fokus ligger på två programmeringsspråk, Java och Erlang. Deras virtuella maskiner, syntax, standardbibliotek samt feltolerans är några av aspekter som presenteras. Därutöver har olika lösningar för kryptering, minneshantering, nätverkstopologier samt olika sätt att kommunicera i parallella system granskats. Baserat på denna studie har två system implementerats, ett system i Java och ett i Erlang. Dessa har båda givits namnet Confector vilket är latin för "uträttare".

Dessa två system har utvecklats utifrån en gemensam kravspecifikation för att jämförelsen av de båda systemen ska vara av intresse. Systemen har däremot inte haft något krav på sig kring struktur eller hur vissa specifika problem lösts. Jämförelsen av systemen behandlar den övergripande utvecklingsprocessen, prestanda och skalbarhet i de olika systemen.

Utifrån studien kunde inga generella lösningar för problemet utses. Både Java och Erlang visade sig vara väl lämpade för att användas i ett sådant system.

I framtiden kan det vara intressant att studera och jämföra två mer komplexa implementationer av systemen. Att använda sig av olika nätverkstopologier och studera hur dessa fungerar i olika situationer är också av intresse.

Förord

Denna rapport är ett kandidatarbete som utfördes våren 2012 av studenter från Chalmers Tekniska Högskola och Göteborgs Universitet på Chalmers institution för Data- och informationsteknik. Vi vill tacka vår handledare Sven-Arne Andreasson för sitt stöd, vägledning och entusiasm under kandidatprojektet.

Innehåll

Definitioner

Embedded database: En databas som exekveras i samma virtuella maskin som det program som använder databasen. Att jämföras mot databaser som utgör egna processer eller befinner sig på andra datorer och endast kan nås via nätverk.

GTK: GTK står för GIMP ToolKit. GTK är verktyg för att skapa grafiska användargränssnitt. GTK har ett stort plattformsstöd.

GUI: GUI står för Graphical User Interface, vilket på svenska innebär ett grafiskt användargränssnitt.

IDE: IDE står för Integrated Development Environment. En IDE används ofta av utvecklare för att redigera källkod. Ofta består en IDE av verktyg för att debugga och kompilera kod.

Java SE: Java SE står för Java Standard Edition. Detta är ett mycket omfattande standardbibliotek för Java.

JSSE: JSSE står för Java Secure Socket Extension. JSSE är en samling Javapaket som möjliggör säker nätverkskommunikation.

Klient: En dator som används lokalt av en användare i taget, och stängs av relativt ofta. Det kan vara allt från en arbetsstation till en mobiltelefon.

NoSQL: NoSQL står för Not Only SQL. En NoSQL databas är en icke relationsbaserad databas. Detta innebär att man inte använder sig av SQL vid kommunikation till en databas.

ODBC: ODBC står för Open Database Connectivity. Detta är standardiserat gränssnitt för åtkomst till databashanterare.

Parallellism: Möjlighet att exekvera delar av ett program parallellt med varandra. Används även för att beskriva situationer då delar av ett program delas in i mindre delar och exekveras om lott.

RPC: RPC står för Remote Procedure Call. RPC är en teknik för att utföra subrutiner i en annan adressrymd, detta är vanligtvis på andra datorer inom ett nätverk.

Server: En dator som främst är till för att acceptera inkommande anslutningar och sällan stängs av. En viss del av ett program kan 'agera server' genom att vänta på inkommande anslutningar, utan att nödvändigtvis köras på en server-dator.

SQL: SQL står för Structured Query Language. SQL är ett språk som används vid hantering av relationsbaserade databaser

TCP: TCP står för Transmission Control Protocol. Det är ett protokoll för att skicka data över nätverk. TCP garanterar att mottagaren får datan i rätt ordning och skickar om paket som försvunnit på vägen.

UDP: UDP står för User Datagram Protocol. Liksom TCP så används det för att skicka data över nätverk, men då TCP garanterar att data kommer fram till mottagaren i rätt ordning så ger UDP inga sådana garantier. Det är upp till användarens program att hålla reda på det och be sändaren om eventuellt tappade paket.

1 Inledning

Att administrera och övervaka ett flertal datorer kan vara en tidskrävande uppgift både för företag och privatpersoner. Problemet blir särskilt svårt när datorerna befinner sig på flera geografiskt skilda platser. Fjärradministration av ett flertal datorer skulle kunna utgöra en besparing av tid och pengar samt förbättra arbetsmiljön för administratörer. Detta kan inbegripa användar- och inställningshantering, installation av mjukvara samt exekvering av kommandon.

Det krävs en hel del förarbete att ta fram ett verktyg som är lämpligt för denna uppgift. Att bestämma vilka tekniker som bör användas är ett tidskrävande projekt, likaså att välja programmeringsspråk bland det stora antal språk som finns tillgängliga. Användandet av nya språk kan innebära tidsbesparingar och mer effektiv utveckling, med det finns även en risk med att införa nya språk som inte är vältestade. Många företag har inte heller resurser till att utföra noggranna språkspecifika studier.

1.1 Syfte

Denna rapport har som mål att ge en bättre bild av hur programmeringsspråken Java och Erlang kan användas för att utveckla internetbaserade administrationsverktyg. Det är dock inte tänkt att vara en uttömmande studie i ämnet. Istället analyseras vissa aspekter av språken tillsammans med protokoll och teoretiska modeller som är nödvändiga för att bygga sådana verktyg. Detta ska förhoppningsvis även ge en bild av fördelar och nackdelar med olika alternativ inte bara för administrationsverktyg utan även för andra verktyg med liknande krav på kommunikation, säkerhet och tillgänglighet.

1.2 Avgränsningar

Antalet programmeringsspråk som skulle kunna användas för ett verktyg ämnat för kommunikation över lokala nätverk och internet är för stort för att göra en jämförelse av dem alla. Valet föll på en jämförelse mellan Java och Erlang som båda har gott stöd för nätverksprogrammering[1][2][3][4], körmiljöer för ett flertal operativsystem[5][6], automatisk minneshantering och har använts för flera framgångsrika mjukvarusystem[7][8].

Det finns också alltför många ramverk, protokoll, lagringssystem och gränssnitt som skulle kunna vara lämpliga för verktyg av denna sort för att alla skulle kunna granskas. Ett urval har därför gjorts baserat på egna erfarenheter, beskrivningar i vetenskapliga publikationer och en genomgång av välkända utvecklare och projekt.

Säkerhetshot mot nätverksbaserade verktyg rör både mänskliga aspekter, fysisk säkerhet och tekniska frågor. Bland dessa har de tekniska frågorna lyfts fram som de mest relevanta för jämförelsen.

Även om administrationsverktyg kan vara tämligen praktiska har arbetet utgått ifrån antagandet att en analys av de språk och tekniska lösningar som kan användas för det syftet är av större allmänintresse än utvecklandet av ett enskilt administrationsverktyg. Därför har utveckling av en färdig produkt inte varit en del av detta projekt.

1.3 Metod

Arbetet inleddes med en genomgång av teoretiska modeller och praktiska lösningar för olika aspekter av ett nätverksbaserat administrationsverktyg. Det var inte möjligt att fastställa på förhand exakt vilka

modeller, lösningar och problem som är relevanta för sådana verktyg. Istället gjordes bedömningar och utökningar av studien när ett ämne ledde vidare till relaterade ämnen.

Utifrån den information som samlats i den inledande studien valde vi att utveckla två prototyper av ett nätverksbaserat administrationsverktyg; ett skrivet i Java och ett i Erlang. Dessa testades sedan med avseende på prestanda och korrekthet. Tillsammans med dessa tekniska tester har även en kvalitativ analys genomförts av de två språkens lämplighet för vårt projekt. Detta inbegriper eventuella säkerhetsproblem, hur enkelt eller svårt det skulle vara att underhålla alternativt bygga ut de två prototyperna samt den arbetsinsats som krävts för att få prototyperna färdiga.

För att göra jämförelsen av de två programmeringsspråken mer informativ utvecklades ett antal *use cases*:

- Inom en organisation med ett kontor och ett internt nätverk ska en administratör på en server kunna utfärda kommandon som ska exekveras på de datorer i nätverket som ska administreras. Kommandona kan inbegripa att lägga till eller ta bort användare, ändra inställningar och utföra uppdateringar.

- En organisation med flera kontor utspridda över flera länder har flera separata interna nätverk med ett flertal datorer i varje nätverk. Varje nätverk har en lokal server (exempelvis Windows Server) som hanterar alla andra datorer på det nätverket. En administratör ska på en central server på ett av dessa nätverk kunna lägga in kommandon som sedan utförs på alla lokala servrar. Endast de servrar som uttryckligen konfigurerats för att fjärrstyras på detta sätt ska kunna ta emot kommandon.

- På ett lokalt nätverk ska stationära och bärbara arbetsdatorer kunna administreras när de kopplar upp sig mot en server. De ska då ta emot och utföra alla kommandon som utfärdats sen de senast var anslutna till server och även i den ordning som kommandona lades in på servern.

En kravspecifikation för funktionalitet togs sedan fram baserat på *use cases*, se Appendix A. Utformningen av kravspecifikationen var en avvägning mellan att vara öppen nog för att varje prototyp skulle få utrymme att utvecklas på ett optimalt sätt utan att tillåta så stora skillnader mellan dem att det inverkade menligt på jämförelsen. Vid utvecklingen av prototyperna så fördelades gruppen så att två personer arbetade med Java implementationen och fyra personer med Erlang implementationen.

2 Förarbete

Få böcker och avhandlingar har skrivits om hur administrationsverktyg bör struktureras eller vilka tekniska lösningar som kan lämpa sig för uppgiften, där undantagen inte är särskilt relevanta[9]. Följaktligen har de två språkens olika egenskaper granskats tillsammans med generella frågor som säkerhet och nätverkstopologi. Detta kapitel avslutas med de slutsatser som låg till grund för utvecklingen av de två prototyperna.

2.1 Tidigare språkstudier

I artikeln *Performance Measurements of Threads in Java and Processes in Erlang*[10] jämförs Javatrådar och Erlang-processer med avseende på prestanda genom att beräkna den tid det tar att skicka meddelanden mellan olika noder. Processoranvändning och beräkningshastighet ingick däremot inte i jämförelsen. Sedan undersökningen genomfördes av Ericsson 1998 har det skett stora förändringar i både Erlangs och Javas virtuella maskiner. Dessutom är det av begränsat intresse att enbart behandla prestanda.

Resultatet av studien visar att Erlang skickar meddelanden mellan processer betydligt snabbare än Java. Den visar att Erlang-processer kan skapas i ett betydligt större antal än Javatrådar. Trådar i Java nämns i 2.6.2. Erlang-processer beskrivs närmre i 2.7.2.

2.2 Säkerhet och autentisering

Tekniska hot mot ett administrationsverktyg som används över internet kan delas in i tre kategorier:

1. Läcka av information.
2. Obehörig exekvering av kommandon.
3. Försämrad tillgänglighet.

Eftersom själva syftet med ett administrationsverktyg för flera datorer är att ge en eller flera administratörer stort inflytande blir konsekvenserna av ett intrång mycket allvarliga. Ett säkerhetsproblem med en webbserver är jämförelsevis mindre allvarligt eftersom kontroll över en webbserver inte nödvändigtvis ger kontroll över andra servrar i samma nätverk.

Läckor av information kan ha mer eller mindre allvarliga konsekvenser. Om en dator som fjärrstyrs skulle skicka tillbaka information om hur mycket lagringsyta som finns ledigt på olika hårddiskar skulle risken vara mycket liten att det kunde användas för att skada systemet. Om ett kommando där en viss användares lösenord ändras till ett nytt värde vore det dock allvarligt om informationen föll i orätta händer då det kan ge en angripare tillgång till den mottagande datorn. För att skydda data över öppna nätverk finns det olika protokoll att använda sig av. De säkerhetsprotokoll som studerats närmre i detta arbete är SSL och SSH.

Som svar på ett intrång kan ett system behöva stängas av vilket kan försämra tillgängligheten inte bara av själva tjänsten utan även andra datorer som påverkats av intrånget. Överbelastningsattacker påverkar främst en utsatt tjänst men kräver inte nödvändigtvis att ett fel finns i koden utan kan uppnås genom att skicka ett stort antal förfrågningar.

2.2.1 Kryptering

Kryptografiska verktyg kan användas för att både dölja information från obehöriga och även bekräfta identiteten på en viss användare i ett system. Ett relaterat område inom kryptografin är kontroll av meddelanden för att garantera att de är intakta och inte har blivit modifierade av någon annan än avsändaren[11].

Det finns olika krypteringsskiffer, men gemensamt är att de består av en krypteringsalgoritm och krypteringsnycklar[12]. Nycklar är hemlig indata till krypteringsalgoritmen. Utan dessa kan krypteringssystemet lätt knäckas, då enbart kunskap om algoritmen i så fall hade krävts[11]. Inom kryptografin nämns Kerckhoffs princip, som går ut på att enbart ett system endast ska förlita sig på att nycklarna är hemliga. Krypteringssystemet och dess algoritmer ska antas vara känt av alla[13]. Utgångspunkten i texten här kommer alltså vara att Kerckhoffs princip gäller. En viktig uppdelning av krypteringssystem är mellan symmetriska och asymmetriska system[12].

I symmetriska system används samma nyckel både för kryptering och för dekryptering. Den största fördelen med sådana system är att de är snabba att använda i jämförelse med asymmetriska system, och därför används gärna symmetriska system för att kryptera större mängder data[14]. Ett mycket stort problem med symmetriska system är att båda parter måste känna till nyckeln innan kommunikationen kan ske. Nyckeln kan inte skickas öppet, eftersom en tredje part då kan avlyssna kommunikationen och därmed senare kan dekryptera vad som sägs. Alltså måste det finnas en befintlig säker kanal där parterna kan komma överens om nycklarna, exempelvis genom att ses innan kommunikationen sker och då utbyta nycklar[12].

Asymmetriska system använder olika nycklar för kryptering och dekryptering. Nycklarna är matematiskt relaterade till varandra, men det är i praktiken omöjligt att erhålla den ena nyckeln utifrån den andra[12]. I jämförelse med symmetriska system är de asymmetriska betydligt långsammare, men fördelen är att det inte krävs att deltagarna har kommit överens om några nycklar i förväg. Vanligt förekommande krypteringsprogram som GnuPG[15] och OpenPGP använder ett asymmetriskt system för att erhålla en krypterad anslutning mellan två datorer, för att sedan byta över till ett symmetriskt system som medger snabbare och mindre beräkningsintensiva krypteringsmöjligheter[16].

En användare av ett asymmetriskt krypteringssytem utför en matematisk beräkning som ger den de två nycklar som behövs för att kunna använda systemet. Den ena nyckeln används för att kryptera meddelanden ämnade för mottagaren och kallas för den publika nyckeln eftersom denna kan spridas fritt. För att kunna dekryptera meddelandet används den andra nyckeln, den privata nyckeln. Denna bör inte spridas till någon, eftersom alla som har tillgång till den privata nyckeln också kommer kunna dekryptera meddelanden krypterade med den publika nyckeln.[12]

2.2.2 SSL

Secure Sockets Layer (SSL) och dess ersättare Transport Layer Security (TLS) är kryptografiska protokoll som tillhandahåller säkerhet över internet. TLS och SSL krypterar nätverkslagren ovanför transportlagret. Protokollen används av bland annat webbläsare, epostklienter, chattprogram och voice-over-ip-klienter för att de ska kunna kommunicera ostört och säkert över olika nätverk. De flesta protokoll fungerar både med och utan SSL/TLS, vilket gör att servern måste få veta om det är en krypterad anslutning som efterfrågas. Detta kan ske antingen genom att ett annat portnummer används (HTTPS använder port 443 istället för standard-HTTP-porten som är port 80), eller genom att en mekanism används som är specifikt för protokollet som byter över till TLS.

Både SSL och TLS inleder en anslutning genom att använda asymmetrisk kryptering för konfidentialitet och för att bekräfta meddelandeintegritet. När en anslutning väl är upprättad övergår protokollet till symmetrisk kryptering.

2.2.3 SSH

SSH står för Secure Shell och är ett nätverksprotokoll med vilket man kan skicka data, exekvera kommandon, etc. I protokollet krypteras trafiken, vilket tillåter säker kommunikation över ett osäkert

nätverk, såsom internet. SSH utvecklades som en ersättning för Telnet och liknande protokoll som skickar information okrypterat över nätverket och därmed anses osäkra för konfidentiell information.

Asymmetrisk kryptering används för att autentisera en anslutande klient och för att denna ska kunna autentisera användaren. Protokollet använder sig av klient-servermodellen, där den ena parten har en SSH-server och den andra en SSH-klient.

SSH används mestadels för att logga in på en dator över nätverket och där köra kommandon. Andra användningsområden är *tunneling* där andra applikationer skickar och tar emot data via en SSH-uppkoppling.

2.3 Minneshantering

Minneshantering handlar om att allokera och avallokera minne som används av ett program under dess körning. Denna hanteringen kan utföras på olika sätt. I språk såsom C/C++ ligger detta ansvar hos programmeraren. Programmeraren måste allokera minne åt sin applikation och även se till att detta utrymme sedan frigörs när det inte behövs. Om ett program allokerar minne som sedan inte frigörs kallas detta för en minnesläcka[17]. En annan fara är att minne frigörs som fortfarande används av programmet. Detta kan leda till svårförutsägbara konsekvenser för programmet. Felaktig minneshantering kan med andra ord påverka tillförlitligheten hos ett program. Genom att använda automatisk minneshantering minskar risken för minnesläckor och minnesrelaterade säkerhetsproblem. Sådan programvara kallas för *Garbage collector* vilket används av både Java och Erlang.

2.4 Nätverkstopologi

Hur ett system som kommunicerar över nätverk ska struktureras beror på olika faktorer men det finns i grova drag två sorters strukturer som används för distribuerade system: klient/server och peer-to-peer (P2P).

Klient/server är en enkel topologi med en central server som tar emot anslutningar från klienter. Vanligtvis finns servern att hitta under ett visst domännamn eller på en viss IP-address[18]. Däremot finns det nackdelar som att en server utgör en *single-point-of-failure* som hela nätverket förlitar sig på. Servern kan även utgöra en flaskhals när det kommer till prestanda[19]. Genom att använda kluster av datorer som agerar som en logisk server kan dock både tillförlitlighet och prestanda ökas.

P2P-nätverk undviker några av problemen med klient/server-nätverk då det inte finns någon enskild central punkt som allting beror på. Vissa P2P-nätverk lyckas också överkomma problemen med flaskhalsar genom att låta deltagare bidraga med resurser samtidigt som de förbrukar resurser. Kurose erbjuder en förenklad modell för att visa hur ett BitTorrent-liknande P2P-nätverk kan hantera nedladdning av filer avsevärt bättre än ett enkelt klient/server-nätverk[20].

Karrels et al. visar att skalbarhet inte kommer automatiskt i P2P-nätverk utan att det beror på hur de är strukturerade[21]. Framförallt skalar välstrukturerade nätverk bättre än ostrukturerade. Ostrukturerade nätverk är sådana där det inte sätts några tvingande regler för hur noder och data ska ordnas, vilket är bra för nätverk med stor omsättning av deltagare. Ett strukturerat nätverk däremot ordnar noder enligt vissa regler. Detta kan exempelvis innebära att noder delas in i en hierarki där mindre samlingar av noder kommunicerar med andra samlingar genom särskilda noder som kallas super-peers.

P2P-nätverk är dock komplicerade att skapa eftersom det är svårt eller rent av omöjligt att försäkra sig om att en given nod går att lita på. Noder måste även kunna hitta varandra när de först ansluter till nätverket och kunna handskas med problemen som kan uppstå när noder plötsligt lämnar nätverket[22].

2.5 Kommunikation

Kommunikation över nätverk kan skötas på en mängd olika vis. En sorts kommunikation baseras på konceptet *message passing* som är en enkelriktad och otypad kommunikation[23]. Det är inte robust eftersom en avsändare inte får veta om ett meddelande kommit fram eller försvunnit på vägen till sin destination. En robust hantering kan implementeras på processnivå och förutsätter att den mottagande parten automatiskt skickar ett svar på meddelandet det mottagit[24]. Därmed kan sändaren antingen observera att meddelandet kommit fram eller att så lång tid förflutit utan svar att den senaste sändningen misslyckats och borde upprepas.

Upprepade sändningar av meddelanden gör ett protokoll mer robust men innebär att andra problem uppstår. Om ett meddelande tas emot av mottagaren och ett svar skickas utan att nå den ursprungliga avsändaren, kommer den ursprunglige avsändaren att anta att meddelandet aldrig nått sin mottagare. Den kommer då skicka ut en andra kopia av meddelandet till mottagaren. Det kan därför bli nödvändigt att bestämma hur ett upprepat meddelande ska tolkas[25].

Antingen tolkas varje instans av meddelandet som separata meddelanden vars instruktioner ska upprepas en gång för varje instans, eller så tolkas flera instanser av samma meddelande som ett enda meddelande vars instruktioner ska utföras endast en gång. Av dessa två är det den sistnämnda som oftast är önskvärd eftersom den förstnämnda lätt kan leda till förstörelse av data. Hanteringen kallas då idempotent[25].

En annan sorts kommunikation baseras på *remote procedure call* som introducerades som ett alternativ till *message passing*[26]. I grunden skickas fortfarande meddelanden mellan de inblandade parterna men kommunikationen är dubbelriktad och programmeraren behöver inte sköta den manuellt. Ett proceduranrop på en dator omvandlas till ett motsvarande anrop på en annan dator. RPC bibehåller även typsäkerhet[23].

2.6 Java

Java är ett allmänt, parallellt, plattformsoberoende och objektorienterat programmeringsspråk med automatisk minneshantering. Det utvecklades i början av 90-talet på Sun Microsystems och var från början tänkt för att användas i hemelektronik. Java är idag ett av världens mest populära programmeringspråk enligt vissa källor[27][28][29].

2.6.1 Java VM

Java-program är plattformsoberoende, de kan alltså köras på olika operativsystem utan att ändras eller kompileras om. Denna egenskap kallades *"write once, run anywhere"*. Detta uppnås genom att källkod i Java inte kompileras till maskinkod (såsom är fallet med C eller Ada) utan istället genererar något som kallas Java byte-kod. Det är ett speciellt instruktionsset som exekveras på en abstrakt (eller virtuell) maskin kallad Java Virtual Machine (JVM). Den är definierad av Java Virtual Machine Specification[30].

För att till exempel kunna använda ett C-program på olika operativsystem krävs två saker: att biblioteket fungerar på de olika operativsystemen och att programmet kompileras för varje operativsystem som det ska användas på. Det går alltså att koda C på ett plattformsoberoende sätt också men det kräver visst merarbete och inför begränsningar i val av bibliotek.

Det är värt att tillägga att byte-koden som exekveras inte nödvändigtvis måste ha kompilerats från Javakällkod. Det är både teoretiskt och praktiskt möjligt att kompilera andra språk till Java byte-kod, till exempel Scala, JavaFX, Python eller Ruby. Möjligheter finns också för att kompilera Java-program till maskinkod i förväg. Den producerade koden är inte längre oberoende av plattform men kräver å andra sidan inte en JVM för exekvering och kan även ge högre prestanda[31].

HotSpot JVM

Det finns många JVM-implementationer, bland annat har IBM en egen och Oracle har två: HotSpot och JRockit. Detta arbete utgår ifrån HotSpot[32][33] som är välutvecklad och något av en standard för JVM:er.

HotSpot kan köras i två olika konfigurationer: *client* och *server*. Lite förenklat kan skillnaden förklaras med att server-versionen är optimerad för prestanda under körning och har därmed längre starttid. Klient-versionen startar snabbare och använder mindre minne men presterar i gengäld sämre. Att växla mellan server- och klient-versionen av HotSpot är trivialt. Det räcker att använda argumentet `-server` vid uppstart[34]. Ett undantag är 64-bitarsversionen som i skrivande stund bara levereras med server-konfiguration[35].

Just-in-time kompilering

Medan Javas byte-kod bidrar till att Java blir plattformsoberoende bidrar detta också till att exekveringen av Java blir långsammare än för språk som kompileras direkt till maskinkod. För att förbättra prestandan i Java tillkom en Just-in-time-kompilator (JIT) till Java 1.1[36].

En JIT-kompilator omvandlar ett mellanspråk (till exempel Java byte-kod) till maskininstruktioner och ökar därmed prestandan. Medan språk som C++ kompileras innan det kan köras, startas en JIT efter att programmet startat. I Javas fall analyseras den exekverade koden och frekvent använda delar av programkoden kommer att optimeras[37].

Hur mycket av programmet som kompileras skiljer sig mellan klient- och server-versionen av JVM. Klient-versionen av JVM utför mindre kodanalys och kompilering än server-versionen och startar därmed snabbare. Server-versionen försöker däremot göra fler optimeringar av koden vid uppstart för att därmed uppnå bättre prestanda när programmet väl exekveras[33].

Minneshantering

Java sköter den grundläggande minneshantering i linje med de principer som nämndes i 2.3. Dess skräphanterare frigör alla objekt som inte kan refereras till från någon aktuell tråd. Dessa objekt anses då vara *dead* medan objekt som har referenser anses vara *live*. Även om objekt refererar till varandra kan dessa bli frigjorda om denna kedja av objekt inte refereras till från någon tråd då hela kedjan är *dead*. Detta innebär att en skräphanterare hindrar minnesläckor och avallokering av minne som används av andra delar av programmet[38].

2.6.2 Trådar

Java har ett inbyggt stöd för trådar och behöver inte använda externa bibliotek för trådar, vilket gör det enklare att utveckla program med parallellt exekverande kod. Java tar dock inte bort de svårigheter som alltid finns med parallell programmering, såsom *deadlock* och *starvation*[39]. Eftersom Java exekveras på JVM hanteras trådarna i första hand av JVM och i andra hand av operativsystemet.

Exempel på att skapa trådar i Java

I Java är det mycket enkelt att skapa nya trådar, det görs genom anrop till klassen `java.lang.Thread`. Här följer ett enkelt exempel på hur en tråd skapas. `Main`-metod:

```
public static void main(String[] args){
   Thread thread = new OurThread();
   thread.start();
   //fortsättning av mainmetod
   ...

}
```

Definition av klassen `OurThread`:

```
public class OurThread extends Thread {
   public void run(){
      // exekvering av ny tråd följer
      ...
   }
}
```

I exekvering av denna exempelkod händer följande: Första raden i `main`-metoden initierar klassen `OurThread` samt sparar objektreferensen i variabeln `thread`. På rad två startas den nyligen initierade klassen och `run`-metoden i klassen kommer exekveras i en separat tråd samtidigt som exekveringen av `main`-metoden fortsätter.

2.6.3 Java Remote Method Invocation

Java Remote Method Invocation (RMI) är en teknik för att utföra objektorienterade RPC från en JVM till en annan[40]. Metod-invokeringar sker på samma sätt för RMI som för vanliga metoder. Den andra JVMen kan befinna sig antingen på samma maskin eller en annan.

Vid användning av RMI har programmen två olika roller: klient och server. En server skapar objekt och gör dessa tillgängliga medan klienten invokerar metoderna som detta objekt gör tillgängliga. När en metod invokeras på klienten skickar RMI en förfrågan till serverns JVM där metoden invokeras. Därefter returneras eventuellt returvärde till klienten[41].

En central princip med RMI är att skilja på beteende och implementation. Detta innebär att ett gränssnitt definierar vilka metoder som kan invokeras och med klasser implementerars metoderna[41] och att en implementation kan ändras utan att klienten påverkas.

RMI kan syfta på både RMI-JRMP och RMI-IIOP. Skillnaden mellan dessa är följande. RMI kan antingen ske från en JVM till en annan JVM (RMI-JRMP - Java Remote Method Protocol) eller plattformsoberoende via CORBA (RMI-IIOP - Internet inter-orb protocol).

Medan traditionell RPC är språkneutral är RMI byggt för att användas tillsammans med andra Java-komponenter. Oracle beskriver ett antal fördelar med RMI, bland annat[42].

- Objektorientering: Java RMI kan serialisera och deserialisera Javaobjekt. Detta innebär i praktiken att objekt kan ges som argument eller som returtyp utan att det kräver någon extrainsats från programmeraren. Egna objekt som man vill kunna skicka behöver implementera Serializeable vilket är ett gränssnitt utan några abstrakta metoder.
- RMI kan överföra implementationer från klient till server och tvärt om. Detta betyder att ett objekt som implementerar ett gränssnitt kan hämtas vid behov från server till klienten. Skulle något ändras i detta objektet kommer helt enkelt ett nytt objekt att skickas till klienten vid nästa anrop.
- RMI har stöd för trådar vilket gör att server- och klient-delarna kan använda trådar för att bli mer parallella.
- Distribuerad Garbage Collection: Objekt som inte längre används av server eller klienten kommer att tas bort.
- Write once, Run Anywhere gäller även för RMI. Precis som ett vanligt Javaprogram är plattformsoberoende är även ett Javaprogram som använder RMI det.

Säkerhet

Eftersom RMI kan ske över internet där trafiken är oskyddad och anrop kan manipuleras eller förfalskas finns det behov av ordentliga säkerhetsmekanismer för att skydda klient och server[43]. För att kryptera trafiken mellan klient och server kan bland annat JSSE användas för att få tillgång till bland annat SSL eller TLS. Det finns även andra skydd tillgängliga, till exempel Javaklassen `SecurityManager` som begränsar vad ett Javaprogram får och inte får göra. Är `SecurityManager` korrekt konfigurerad kan det hindra eller begränsa effekten av skadlig kod.

2.6.4 Serialisering

Serialisering (*serialization*) är processen att omvandla ett objekt till ett format som möjliggör överföring av objektet för att kunna återskapas senare. Att återskapa ett objekt kallas deserialisering. Ibland används även termerna *marshalling*/*unmarshalling*. I Java skrivs serialiserade objekt till `Streams` vilket ger programmeraren stor flexibilitet. Till exempel kan objekt skrivas till en fil för att återskapas senare av samma program, alternativt kan objektet serialiseras och överföras via nätverk till en annan JVM. Detta kan exempelvis ske i samband med RMI eller göras manuellt med hjälp av sockets.

För att ett objekt ska kunna serialiseras måste det implementera `Serializable`[44]. `Serializable` är ett gränssnitt utan abstrakta metoder eller fält och fungerar enbart som ett så kallat *marker interface* som markerar att ett objekt kan serialiseras[45]. När ett objekt serialiseras sparas värdet av samtliga primitiva datatyper. Har objektet referenser till andra objekt måste de referenserna antingen vara markerade som `transient` eller hänvisa till objekt som också implementerar `Serializable`. Om en referens är markerad som `transient` innebär det att objektet i fråga inte kommer att serialiseras utan istället måste återskapas vid tidpunkten för deserialisering. Statiska variabler kan per definition inte överföras via serialisering, eftersom dessa tillhör klassen medan det är objekt som serialiseras. Detta beskrivs mer ingående av bland annat Sierra[45].

Exempel på serialisering finns i Appendix B.

2.7 Erlang

Erlang är ett mindre språk än Java både i termer om biblioteksstöd och användning för mjukvaruutveckling. Det har dock ett antal egenskaper som gör det mycket väl lämpat för att skriva mjukvara med hög tillgänglighet[46] som kan spridas ut på flera samverkande datorer[47][4].

Det är ett funktionellt och dynamiskt typat språk som skapades 1986 av Joe Armstrong för Ericssons räkning. Målet var att använda Erlang som ett verktyg för att utveckla Ericssons egna system, såsom telefonväxlar. Fokus låg på tillgänglighet, robusthet och att göra det enkelt att distribuera ett program över flera noder[8]. 1998 valde Ericsson att släppa programmeringsspråket som öppen källkod då man ville uppmuntra användandet av Erlang utanför Ericsson[48].

2.7.1 Erlang VM

Erlang-kod exekveras i en virtuell maskin kallad Erlang VM som sköter minneshantering i likhet med JVM. Det finns däremot skillnader både i funktionalitet och optimering, där JVM har optimerats i långt större utsträckning.

När en Erlang-modul kompileras skapas en .beam-fil. I denna fil hamnar byte-koden från Erlang-modulen som kompilerades så att alla dess funktioner går att anropa[46]. Beteckningen BEAM står för Bogdan's Erlang Abstract Machine och Erlang VM är en implementation av denna abstrakta maskin. I likhet med Java är Erlang-kod portabel[49] mellan olika arkitekturer. Officiellt saknar Erlang VM en JIT-kompilator[50] men HiPE är en optimerande kompilator som omvandlar utvald byte-kod till maskinkod[51] och ingår sedan 2001 i Erlang[52].

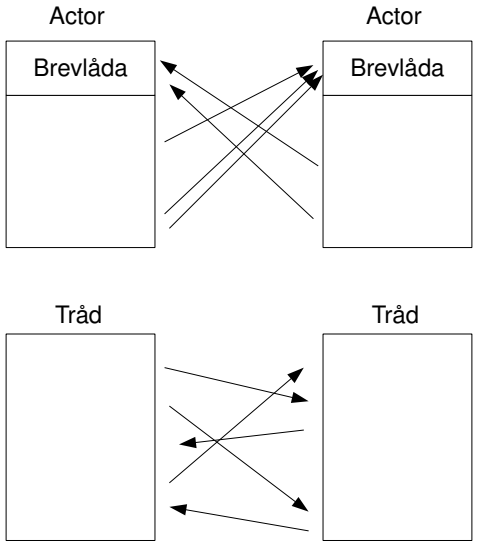
I Erlang VM har varje process en egen heap som skapas när processen startar och försvinner när processen är färdig. En process allokerar och avallokerar endast minne när den är schemalagd. På

grund av den specialdesignade schemaläggaren vet alltid Erlang VM när en process har körts. Detta innebär att Erlang VM endast behöver skräphantera heapen för processen som har körts sedan senaste skräphanteringen kördes.

2.7.2 Erlang-processer

Till skillnad från Java använder Erlang inte trådar för att exekvera kod parallellt utan en egen form av processer. Den underliggande modellen kallas för actor-modellen och är en av många modeller som härstammar från *message passing*. Andra modeller baserade på *message passing* kan vara anpassade till telefonnätverk där meddelanden skickas direkt, om linjen skulle vara upptagen kommer inte meddelandet att nå sin destination. Actor-modellen däremot kan liknas vid en posttjänst där meddelanden skickas asynkront till mottagaren [53]. Detta meddelande hamnar i mottagarens brevlåda och läses när mottagaren är tillgänglig. I Erlang är kommunikationen mellan olika processer asynkron med denna typ av kommunikation går det att implementera synkron kommunikation.

Endast genom att skicka ett meddelande till en actor och dess brevlåda kan utomstående actors påverka dess tillstånd. En actors tillstånd kan alltså bara direkt påverkas av den själv, alla andra måste gå genom brevlådan. Detta skiljer sig från hur trådar fungerar, där data kan manipuleras samtidigt från olika håll. En illustration av denna skillnad visas i figur 2.1.



Figur 2.1: Actor och tråd

För att skapa en process i Erlang (som motsvarar en actor) kan olika varianter av `spawn`-kommandot användas:

```
Pid = spawn(Modul, Funktion, Argument)
```

Där `Pid` är en variabel som innehåller en referens till den skapade processen. För att skicka ett meddelande till processen används !-operatorn:

```
Pid ! meddelande
```

Detta kan läsas som "skicka `meddelande` till `Pid`".

Erlang sägs vara ett processorienterat programmeringsspråk. I Erlang är olika processer en stor del av utvecklingen. En process kan liknas vid ett objekt i ett objektorienterat språk, där en process skapas för att lösa en specifik uppgift. Processer i Erlang är billiga och kan därför skapas mycket fort, vilket är användbart för att skapa system med hög parallellism.

2.7.3 Distributed Erlang

Distributed Erlang är ett system i Erlang som låter olika noder skapa processer på och skicka meddelanden till varandra på samma sätt som processer och meddelanden hanteras inom enskilda Erlang-noder. Detta kräver dock att noderna givits namn på samma form[54] och att de har samma cookie-värde.

Det är lätt att blanda ihop Distributed Erlang med vanlig nätverksfunktionalitet. Två Erlang-noder som antingen inte har samma cookie eller namn på samma form kan fortfarande kommunicera med hjälp av sockets. Fördelen med Distributed Erlang är att det gör det enklare för programmerare att skriva kod som exekveras på flera olika noder, något som visas mer utförligt i Appendix D. Dels behöver inte sockets och addresser hanteras manuellt och dels kan meddelandehantering skötas asynkront precis som inom en och samma nod.

En stor nackdel med Distributed Erlang är bristande säkerhet. Utvecklingen av Erlang fokuserade på tillgänglighet och feltolerans. Säkerhet var inte lika högt prioriterat eftersom de nätverk där Erlang var tänkt att användas antogs vara skyddade av brandväggar[8]. Kommunikation mellan noder som använder Distributed Erlang skickas exempelvis i klartext[55].

Distributed Erlang använder också en enkel autentiseringmetod där en enda sträng, som på något sätt finns att läsa i klartext på alla noder, avgör vilka Erlang-noder som är tillåtna. Även om ett *challenge/response*-förfarande används[56] när en anslutning skapas mellan två noder för att undvika att skicka strängen i klartext över nätverket är det en svag lösning säkerhetsmässigt.

Risken är inte bara att information kan hamna i fel händer eftersom Distributed Erlang inte bara gör det enkelt för noder att skicka meddelanden mellan sig utan även gör det enkelt för noder att starta processer hos varandra. Detta gör att en motståndare som fått kontroll över en nod i princip kan exekvera godtyckliga kommandon på alla andra noder med samma cookie[57].

2.7.4 Erlang/OTP

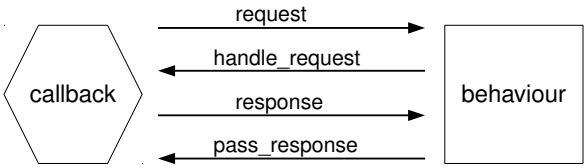
OTP (Open Telecom Platform) är ett bibliotek och en uppsättning designprinciper för Erlang och används för de flesta Erlang-projekt[58]. Det som ingår i OTP är bland annat en Erlang-tolk, kompilator samt Distributed Erlang för att förenkla kommunikation mellan noder[59]. Med hjälp av OTP:s moduler kan högkvalitativa system produceras på ett enkelt sätt. Det är inte heller nödvändigt att implementera typiska strukturer såsom server eller tillståndsmaskiner då det redan finns vältestade och modulära stommar implementerade i OTP-ramverket. I detta ramverk finns redan färdiga moduler för att övervaka processer samt ett systematiskt tillvägagångssätt för att uppgradera kodbasen under drift.

Behaviour

I OTP eftersträvas åtskillnad mellan de generiska och de specifika delarna av koden[60]. Den generiska modulen, eller behaviour-modulen, är ett vanligt mönster som formaliserats. Den specifika delen, även kallad callback-modulen använder sig av det generiska gränssnittet vid implementeringen. I OTP finns ett par olika behaviours tillgängliga för utvecklare. Det vanligaste är en generisk server där en central modul används för att hantera anrop.

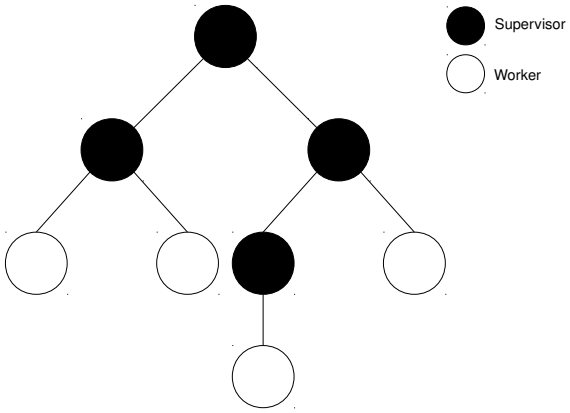
Figur 2.2 illustrerar ett exempel på hur kommunikationen mellan en callback-modul och en behaviour-modul skulle kunna se ut i en generisk server. Callback-modulen skickar en asynkron förfrågan till behaviour-modulen som i sin tur noterar till vilken process ett svar ska skickas. Denna förfrågan skickas sedan tillbaka till callback-modulen för att behandlas enligt programmerarens önskemål. Det sista som händer är att ett svar skickas tillbaka till behaviour-modulen som i sin tur vidarebefordrar det till den process som från början gjorde en förfrågan. Programmeraren behöver alltså inte tänka på hur meddelanden kommer fram, detta sköts helt av behaviour-modulen. I Appendix E studeras och implementeras en naiv version av den generiska server OTP bidrar med.

Supervisor är ett annat behaviour som ingår i OTP. Dess uppgift är att övervaka subprocesser enligt ett antal regler som utförs när en process terminerar[61]. Ett supervisor-träd består av supervisors och



Figur 2.2: Callback-modul och generisk modul

workers, där en arbetare omfattar olika OTP-behaviours. Ett exempel på ett supervisor-träd visas i figur 2.3.



Figur 2.3: Supervisor-träd

Att övervaka processer är mycket användbart, om processer skulle startas utan att vara övervakade blir det svårt att avgöra hur många processer som körs. Detta kommer medföra att Erlang VM kommer bli långsammare och långsammare och till slut krascha eftersom minnet tar slut[62]. En annan fördel med supervisors är att systemet kan avslutas på ett korrekt sätt.

En OTP-applikation bör följa en standard för hur applikationens struktur ska se ut. Det finns även ett specifikt OTP-behaviour för detta. Detta behaviour används för att specificera vilka resurser och olika beroenden de inblandade modulerna har[63]. På detta vis binder strukturen för applikationen samman samt grupperar relaterad kod.

Det finns ytterligare två behaviours som kan vara värda att känna till. Ett av dem är `gen_fsm` som är ett behaviour som implementerar en ändlig tillståndsmaskin. Callback-modulen för en `gen_fsm` består av olika definitioner av de tillstånd en process kan befinna sig i. Det andra är `gen_event` som används för att implementera en eventhanterare. `gen_event` används ofta vid loggning.

2.8 Slutsatser

Baserat på genomgången av tillgängliga tekniska lösningar och abstrakta koncept kring topologi och kommunikation valdes de som bedömdes vara mest lämpliga. Merparten av designvalen återfinns i prototyperna som utvecklades men ett säkert transportprotokoll för nätverkskommunikation valdes bort av tidsskäl.

Nätverksprotokoll och topologi

Då båda implementationerna använder sockets för kommunikation mellan server och klient valdes TCP-protokollet för transportlagret. UDP-protokollet var det andra alternativet men det valdes snabbt bort eftersom det inte tar hand om fel och förlorade paket skickas inte om automatiskt, vilket TCP gör.[64].

TCP-protokollet varken krypterar information eller autentiserar deltagarna som kommunicerar vilket gör att SSL eller TLS är givna kandidater som transportprotokoll i senare implementationer. Att använda SSH var ett alternativ som även det hade erbjudit en krypterad tunnel mellan servern och dess klienter. SSL och SSH innebär samma säkerhet om de implementeras korrekt, men SSL medger mer flexibilitet då det tillåter överföring av mer än bara shell-kommandon. Med SSL kan i princip vilken typ av meddelanden som helst skickas. Detta ger även större möjligheter att expandera till fler plattformar vid en eventuell vidareutveckling.

På grund av komplexiteten i P2P-nätverk kommer systemet att baseras på klient/server-strukturen. Det finns även svårigheter för noder i ett P2P-nätverk att passera brandväggar[65] vilket vore problematiskt för de fall där klientdatorer ska styras över internet eftersom lokala nätverk ofta skyddas av brandväggar.

Java

Java-gruppen valde att använda ett eget kommunikationsprotokoll mellan server och klient som använder sockets för att skicka serialiserade objekt mellan varandra. Lösningen är lik RMI i vissa avseenden men erbjuder bättre effektivitet prestandamässigt[66] och gav gruppen större kontroll över kommunikationen.

Detta val ledde dock till att flera fördelar med RMI mistes, bland annat enkelheten och relativt robusta kommunikationsegenskaper. Fördelen med avseende på prestanda och möjligheten att själva utforma protokollet vägde dock över.

Erlang

För Erlang-implementationen valdes OTP-ramverket. Genom att bryta ut gemensam kod för liknande funktioner i en modul blir det betydligt lättare att felsöka eventuella buggar samt att uppdatera systemet. Att uppdatera den generiska koden skulle även innebära att alla system baserade på denna kod också tar del av potentiella optimeringar. Koden är lätt att återanvända om man vill implementera ett nytt system som bygger på samma struktur. Prototypen i Erlang består därför huvudsakligen av OTP-behaviours.

3 Implementation

Utvecklingen av de två prototyperna var praktiskt uppdelad i två delprojekt. Detta kapitel beskriver strukturen för de två prototyperna.

3.1 Java

Java-gruppen har strävat efter att bygga implementationen enligt principerna *high cohesion* och *loose coupling*. *High cohesion* syftar på att varje klass ska göra en sak och göra det väl. *Loose coupling* syftar på i vilken utsträckning klasser har kännedom om andra klasser. Det är eftersträvansvärt att klasser har så lite kännedom som möjligt om varandra. Detta eftersom det tillåter en klass att skrivas om utan att andra klasser måste ändras till följd av detta.

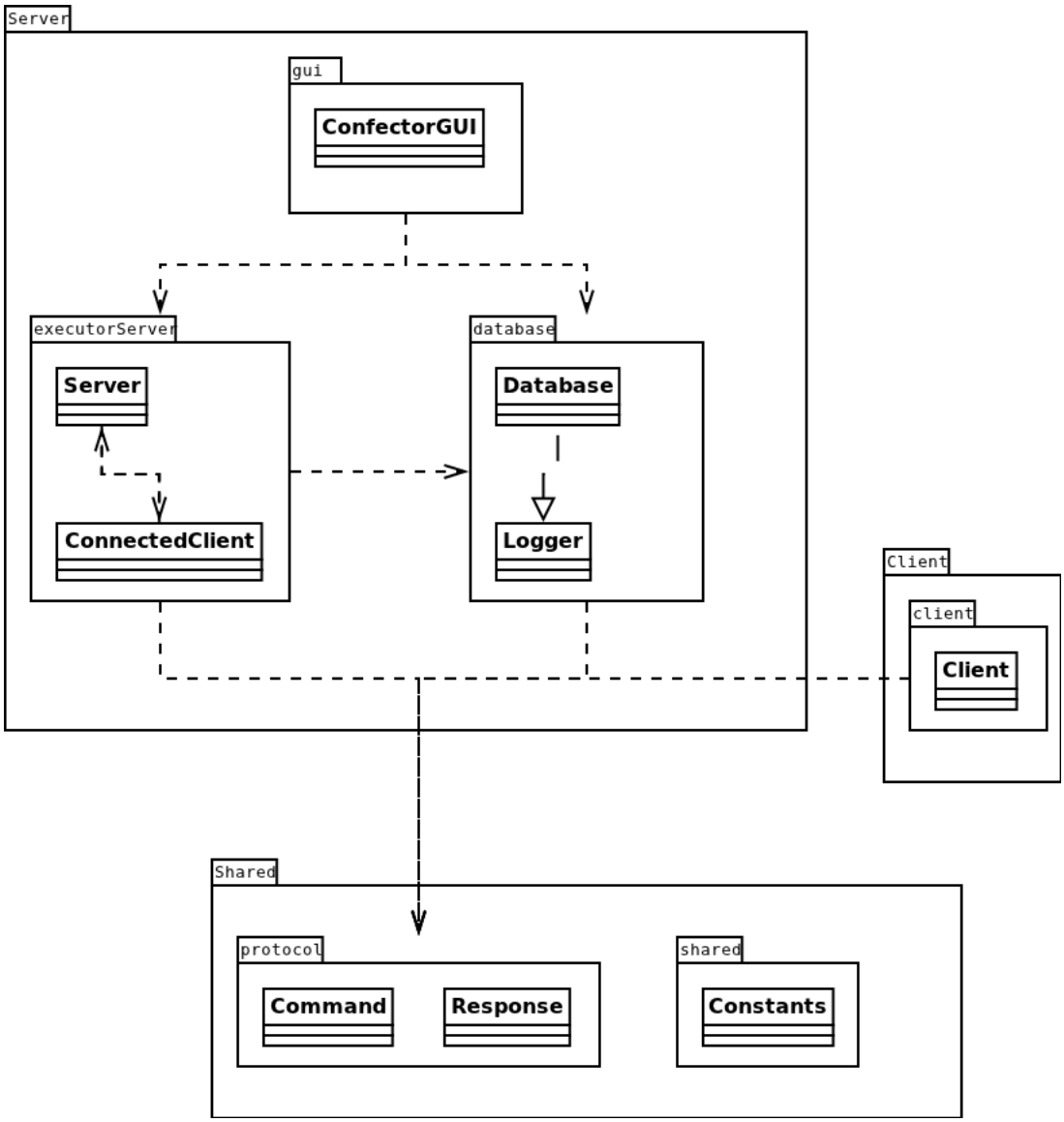
För att få bättre översikt över data och funktionalitet har Java-implementationen också utvecklat ett Swing GUI. Detta har lett till att implementationen använder sig av Model-Delegate-strategin. Denna strategi är snarlik MVC men skiljer sig på det sättet att View- respektive Controller-delarna i MVC är ersatta med en Delegate-del. Skillnaden är relevant eftersom det är problematiskt att skilja på View och Controller i Swing[67].

Allt har även delats in i olika paket för att tydligt skilja på olika komponenters roller. Detta är ett led i att sträva efter principerna som nämndes i inledningen. Några speciella bibliotek har inte behövts användas för Java-implementationen utan enbart Java SE.

Om projektet hade pågått under längre tid skulle kommunikationsdelarna för server och klienter implementerats som separata klasser. Dessa klasser hade fått implementera ett eller flera kommunikationsgränssnitt vilket hade gjort det möjligt att byta ut den faktiska implementationen utan att göra några ändringar i övriga programvaran.

3.1.1 Struktur

Följande är en övergripande beskriving av hur klasserna fungerar och deras uppgifter. Figur 3.1 visar hur javaimplementationen organiserats. Den övergripande strukturen är uppdelad i tre delar: Server, Client och Shared. Serverns uppgift är att tillhandahålla kienterna med arbete, Shareddelen innehåller den kod som är nödvändig för både klient och server. Det som delas mellan server och klient är ett protokoll för informationsutbyte och även olika konstanter.



Figur 3.1: Struktur på Java-implementationen.

De olika klasser och paket som utgör Java-implementationen presenteras nedan.

protocol: Detta paketet innehåller klasserna `Command` och `Response`. `Command` skickas av servern till en klient och innehåller ett kommando som ska utföras av klienten. `Response` skickas av klienten till servern efter det att ett kommando utförts.

shared: Innehåller `enum Constants` där diverse konstanta värden finns definerade. Detta för att andra klasser ska kunna referera till samma konstanter.

database: Här finns dels ett gränssnitt `Logger` som definierar vad databasen erbjuder för tjänster. Genom att referera till den faktiska implementationen genom ett gränssnitt kunde test-databasen bytas ut mot Oracle med en enkel ändring. `OracleDatabase` är en icke abstrakt klass som implementerar `Logger`.

client: `Client` är klient-implementationen av Confector. Den ansluter sig till servern och skickar sedan sitt användar-id till servern. Därefter lyssnar den efter `Command`-objekt och startar en process för varje `Command`-objekt som mottas. När en process exekverat klart skickas ett `Response`-objekt till servern med resultatet av körningen.

executorServer: `Server` sköter ett antal uppgifter. Dels skapar den, loggar och skickar alla `Command`-objekt. Den tar även emot nya klienters anslutningar och skapar ett `ConnectedClient`-objekt för varje klient som ansluter. `ConnectedClient` implementerar `Runnable` och startas av en `Excecutor` som `Server` har. När `ConnectedClient` startar tar den emot ett klient-id och kontrollerar sedan tre saker; dels att det inte redan finns en klient ansluten med detta id, dels invokerar den databasen för att se om klienten finns (annars skapas den) och sist om det finns några gamla kommandon till klienten som den ska utföra.

gui: I detta paketet finns den grafiska användargränssnittet för Confector. Klassen `ConfectorGUI` har ett antal uppgifter.

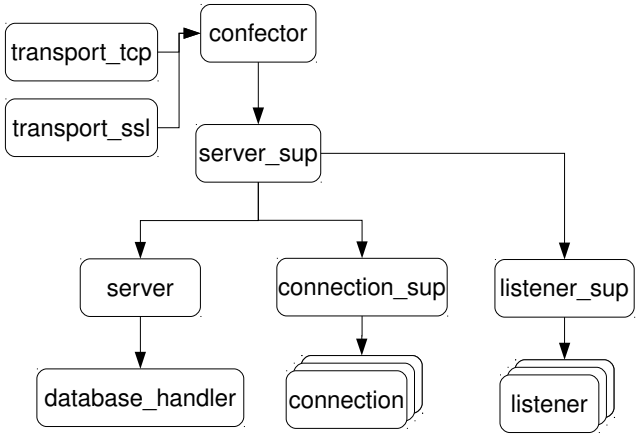
1. Det skapar det grafiska gränssnitt som användaren ser och interagerar med.
2. Det lyssnar efter `PropertyChangeEvents` från `Server`.
3. Det presenterar mottagen data från klienterna som finns i databasen.
4. Det tar emot indata från användaren och vidarebefodrar detta till `Server`.

3.2 Erlang

Att designa ett program i ett processbaserat språk skiljer sig lite från att designa ett objektbaserat. Istället för att kombinera olika objekt gäller det istället att bestämma vilka processer som ska kommunicera med varandra och på vilket sätt de ska kommunicera. I detta avsnitt presenteras strukturen för Erlangprototypen.

3.2.1 Struktur

Strukturen för implementationen i Erlang består av ett flertal olika moduler som bygger på OTP-ramverket. Figur 3.2 avser att visa de olika modulerna samt hur dessa beror på varandra.



Figur 3.2: Struktur på Erlang-implementationen.

Här följer en beskrivning av de olika modulerna som systemet är uppbyggt av.

`confector` - Detta är callback-modulen för det applikations-behaviour som sköter uppstart av alla supervisors. Modulen fungerar även som ett gränssnitt mot applikationen.

`server_sup` - Denna supervisor övervakar de olika delarna som är nödvändiga för att servern ska fungera ordentligt.

`connection_sup` - Denna supervisor vakar över alla anslutningar utåt.

`connection` - Detta är den interna representationen av en klient som är ansluten till servern. Det är via denna modulen servern kommer att kommunicera med utomstående klienter via dess specifika socket.

`listener_sup` - Denna modul ansvarar för att servern alltid finns tillgänglig genom att starta en eller flera `listener`.

`listener` - Uppgiften för denna modul är att öppna en socket och lyssna efter klienter som vill ansluta. När en klient ansluter skapas en ny `connection` som tar över en socket från denna `listener`.

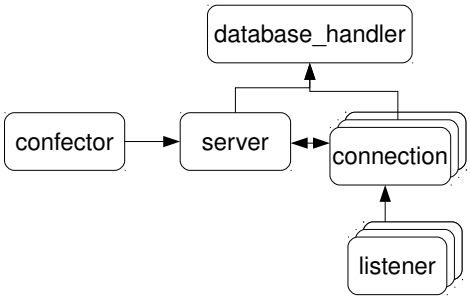
`server` - Uppgiften för denna modul är att distribuera ett meddelande till samtliga anslutna klienter. Modulen ser även till att de övriga modulerna startar med rätt information.

`transport_tcp` - En modul som implementerar ett gränssnitt som hanterar nödvändiga funktioner vid kommunikation via TCP.

`transport_ssl` - Implementerar motsvarande funktioner som i `transport_tcp` fast för SSL-kommunikation. Dessa moduler finns för att enkelt skifta mellan SSL och TCP vid behov.

`database_handler` - Kommunikationen till databasen sker genom denna modul.

Figur 3.3 visar hur kommunikationen mellan de olika modulerna ser ut. För slutanvändaren är endast `confector` synlig, det är via denna modul som servern sägs till att skicka meddelanden till de anslutna klienterna.



Figur 3.3: Beroenden inom Erlang-implementationen.

För att minimera risken för problem som kan uppstå i parallella system utförs endast asynkrona anrop mellan modulerna, dock med vissa undantag. Kommunikationen mellan olika processer är alltid asynkron medan vissa interna anrop sker synkront. På detta sätt kommer en process aldrig blockeras på grund av väntan på ett svar från en annan process. Ett svar på ett asynkront anrop sker istället när en den mottagande processen hanterat anropet som även detta skickas asynkront. Ett problem med detta kan vara att det blir nödvändigt att manuellt logga vilka anrop som inte har besvarats.

3.3 Skillnader i implementationerna

Den största skillnaden mellan implementationerna är serialisering och databaser. Implementationen i Erlang konverterar uttryckligen datastrukturer till och från binär form vid nätverkskommunikation medan Java-implementationen förlitar sig på de underliggande biblioteken för att sköta det automatiskt. Den andra stora skillnaden är de två implementationernas databaser, där Java-implementationen använder en extern SQL-databas medan Erlang-implementationen använder en inbyggd nyckel/värde-databas (se Appendix D). Det finns även en viss skillnad kring hur nya anslutningar accepteras där Erlang- implementationen använder sig av en fristående modul och Java-implementationen har denna funktionallitet implementerad i server-klassen. Förutom dessa skillnader var lösningarna mycket lika.

De båda prototyperna som implementerades uppfyllde kravspecifikationen till fullo. Java-implementationen överskred kraven då det gavs ett grafiskt gränssnitt för att förenkla testning samt en klientdel som kunde exekvera riktiga kommandon. Till Erlang-implementationen skapades en SSL-modul för att undersöka funktionaliteten. Testerna utfördes dock endast på de delar som ingick i kravspecifikationen.

4 Resultat och Diskussion

Resultat av genomförda tester och en utvärdering av de två implementationerna presenteras i detta kapitel. Utvecklingen av de båda systemen diskuteras och även de olika plattformarna som språken använder.

4.1 Utveckling

Något som ofta är viktigare än prestanda vid val av programmeringsspråk är utvecklingsprocessen. Detta inbegriper utvecklingsverktyg, hur välanvänt språket är samt vilka bibliotek och ramverk som finns tillgängliga.

4.1.1 Kodförståelse

Läsbarheten i ett språk kan vara svår att bedöma för någon som är välbekant med språket, vilket var fallet med Java. I största möjliga mån har vi försökt ha i åtanke även de läsare av koden som inte kan språket. Java är till utseendet likt andra vanliga språk som C, med klamrar som visar vilka delar som utgör logiska block. Information om vilken sorts objekt som en variabel hänvisar till är inte bara praktiskt för automatisk typ-checkning utan ger även läsaren en bättre förståelse av vad koden betyder. Att objekt utför namngivna operationer (metoder) är också beskrivande.

Java har också några nackdelar vad gäller läsbarhet. Exempelvis rekommenderas programmerare att dölja fält i klasser och endast göra dem tillgängliga via metoder[68]. Detta leder till två extra metoder för varje variabel i en klass som ska kunna både läsas och ändras av andra klasser, vilket i sin tur gör koden längre och svårare att överblicka. Genom att förlägga alla dessa metoder längst ner i en klass stör de inte läsning av klassens egna logik och om metoderna döps enligt `getVar()`/`setVar(T var)` blir det också enklare att sortera bort dem.

I Java går det att binda variabler mer än en gång, till skillnad från Erlang där en variabel endast kan tilldelas ett och endast ett värde. Möjligheten att binda variabler flera gånger kan ses som en nackdel. Det är dock både möjligt och rekommenderat att i största möjliga mån skapa Java-klasser där skapade objekt inte kan byta värde[69].

Erlang-kod är mycket kompakt och koncis jämfört med Java. Denna egenskap kan dock göra det svårt för nybörjare inom Erlang att förstå segment som utför många saker utan förklarande variabelnamn och funktionsanrop. Likaledes hjälper OTP till att strukturera kod och förhindra upprepningar som är gemensamma för många liknande moduler men flyttar också mycket aktivitet utanför den egna koden. Förståelse av kodens innebörd beror då på hur väl läsaren kan de bakomliggande modulerna.

En språkkonstruktion i Erlang som är mycket användbar är *records* men kombinerat med mönstermatchning blir uttrycken inte alltid intuitiva. Följande defintion av en funktion `func` kan läsas som "Två argument, det första kallad `Var`, den andra ett *record* S, vars fält `conn` ska läsas in och kallas `ConnNumber`".

```
func(Var,S=#state{conn=ConnNumber}) ->
  ...
```

I andra sammanhang betyder uttrycket `S=#state{conn=ConnNumber}` "Skapa en ny konstant `S` baserat på ett *record* av typen `state` där alla värden sätts till standardvärden, förutom `conn` som sätts till värdet i konstanten `ConnNumber`".

Något som försvårar både manuell och automatisk refaktorering är de olika separeringsuttrycken som används, nämligen `,` (komma) `;` (semikolon) `.` (punkt). Alla tre har olika betydelse och används ofta i Erlang-kod. Ett exempel där det kan ställa till problem:

```
func(true)  ->
  do(),
  now();
func(false) ->
  done().
```

För att ändra ordningen i funktionsklausulen så att `func(false)` kommer först behöver koden ändras på följande sätt.

```
func(false) ->
  done();
func(true)  ->
  do(),
  now().
```

Lägg märke till hur `done()` och `now()` även tvingades byta separeringsuttryck. Att byta ordning på `do()` och `now()` skulle även detta innebära uttrycksbyte. I längden kräver detta mycket arbete och ofta glöms det bort att ändra uttrycket. Utan en utvecklingsmiljö som markerar sådana fel resulterar dessa misstag i syntax-fel vid kompilering.

4.1.2 Bibliotek och ramverk

För implementationen i Java användes endast Java SE. Det finns också ett stort antal externa bibliotek och även Java Enterprise Edition som bland annat erbjuder översättning mellan objekt och databaser (*Object Relational Mapping*). För detta projekt räckte dock Java SE med stor marginal.

Även Erlang-implementationen använde huvudsakligen standardbiblioteket tillsammans med OTP vilket gjorde det enklare att med hjälp av behaviours undvika återkommande mönster. För denna prototyp var OTP framförallt till nytta vid design av applikationen och gjorde det tämligen enkelt att ordna en struktur med de behaviours som finns tillgängliga. Det var även någon som gjorde det enklare att skilja mellan synkrona och asynkrona anrop genom callback-gränssnittet och sparade mycket tid att jämfört med att själv skriva robust *message passing* mellan de olika modulerna.

4.1.3 Inlärning

Som påpekades i tidigare avsnitt var alla i gruppen välbekanta med Java sedan tidigare vilket gör att endast inlärningen av Erlang under projektets gång kan granskas. Erlang med dess standardbibliotek var relativt lätt att använda men OTP utgjorde en större utmaning. Dokumentationen av OTP är tekniskt sett bra men förklarar inte särskilt väl vad syftet är med vissa behaviours och designprinciper. Syftet med `supervisor` är tydligt men exempelvis `gen_server` framstår som trivialt vilket gjorde det svårt att förstå hur det skulle användas på rätt sätt. I takt med att Erlang-gruppen arbetat mer med Erlang, med och utan OTP, desto tydligare har fördelarna och syftet med OTP blivit.

4.1.4 Community

Java har ett mycket stort community och det finns många forum där man kan ställa frågor och få hjälp. I Sverige finns till exempel Javaforum[70] och engelskspråkiga forum innefattar Oracle Technology Networks Java-del[71]. Erlang har ett betydligt mindre community vilket gör att det kan vara svårare eller ta längre tid att få svar på avancerade frågor.

Det finns även mycket litteratur inom Java. Det kan däremot vara svårt att veta vilka böcker som är bäst lämpade för att lära sig vissa saker, men tack vare den stora användarbasen är det lätt att få rekommendationer av mer erfarna programmerare. Litteraturen som finns för Erlang täcker i stor utsträckning samma område och innehållet är ofta grundläggande. För närvarande finns endast en bok, "Erlang and OTP in Action"[72] som täcker mer avancerade ämnesområden och även går in djupare i hur OTP fungerar. För att hitta information om hur den virtuella maskinen fungerar är mailinglistor den bästa källan eftersom den inte har någon officiell dokumentation.

4.1.5 Verktyg

Java-applikationen utvecklades i Linux med textredigeraren VIM och med hjälp av make, vilket har fungerat väl. Det finns kompletta IDE:er som till exempel Eclipse och Netbeans som båda är mycket kraftfulla. Till Eclipse finns flera användbara plugins men eftersom Java-gruppen var bekant med VIM sedan tidigare valdes den. Valet av utvecklingsverktyg för Java har troligtvis haft liten praktisk påverkan för projekt.

Erlang har främst utvecklats med hjälp av textredigeraren Emacs som är det officiella verktyget för utveckling i Erlang[73]. En plugin för Eclipse kallad ErlIDE har dock använts i viss utsträckning vilket ger felinformation när koden skrivs istället för vid kompilering samt automatisk kompilering när ändringar sparas. ErlIDE ger också förslag på vilka funktioner som kan användas för en viss modul när namnet på en modul skrivits in[74].

4.1.6 Design

I Javautvecklingen användes en iterativ utvecklingsprocess, vilket ledde till att viss refaktorering av koden behövdes ibland. Till exempel behövdes vissa ändringar göras när ett gränssnitt visade sig vara otillräckligt. Processens mål var att alltid ha en fungerande applikation och att göra så små ändringar som möjligt. Det finns mängder av litteratur om utvecklingsprocesser och Java-applikations design men för detta arbete har främst Effective Java[69] använts i utvecklingsprocessen.

Att designa och strukturera upp en applikation i Erlang var enklare än vad gruppen trott. De olika modulernas beroende var naturligt och att utveckla denna struktur i framtiden är troligtvis inga problem. Med supervisor-behaviour är det mycket enkelt att lägga till nya arbetarmoduler och bygga ut systemet. Trots lite tidigare erfarenhet av detta så lyckades en robust och modulär server att skapas på kort tid. Även om det inte varit ett hinder för detta projekt bör det nämnas att Erlang inte erbjuder samma möjligheter till hierarkisk uppdelning av moduler som Java har för klasser.

4.2 Vidareutveckling

Möjligheterna att vidareutveckla de två implementationerna är goda. Några exempel är SSL, kluster av servrar och bättre administrationsmöjligheter. Det kan även finnas intresse för att ändra delar av systemet, såsom kommunikationsprotokollet mellan klient och server eller databasimplementation.

Sett till kodstruktur är båda implementationer väl lämpade för att ändra och utöka. Implementationen i Java använder sig av designmönstret Model-Delegate för server-delen vilket skapar en tydlig uppdelning av ansvarsområden. Detta torde hjälpa framtida utvecklare av systemet. Inte heller har några uppenbara svårigheter med att vidareutveckla Erlang-prototypen upptäckts. Användandet av OTP i Erlang-implementationen skapade vissa svårigheter under arbetet med det gör vidareutveckling enklare då alla delar följer bestämda mönster. Om olika moduler hade haft olika variationer av funktionsanrop och direkt meddelandehantering hade framtida utökningar av funktionaliteten krävt större kunskap om alla existerande delar av koden.

En svårighet med de två implementationerna är att båda har sina respektive kommunikationsprotokoll strikt definerade i koden. Det gör det svårare men inte omöjligt att byta protokollet för senare versioner. För kommunikation mellan server och klient vore ett XML-baserat protokoll önskvärt.

Valet att i Java-implementationen använda en extern databas och gränssnittet `Logger` för att kommunicera med databasklassen gör att det är enkelt att byta databas. Nuvarande databas är Oracle vars relationsdatabas är relativt lik andra databaser på marknaden. Utbyte av databasen är ett större hinder för Erlang-implementationen eftersom Mnesia är mycket tätt knuten till språket Erlang. Det är dock fullt möjligt att skriva en ny `database_handler` med SQL-anrop istället för Mnesia-transaktioner med hjälp Erlangs ODBC-funktionalitet[75].

Java stödjer flera plattformar och klientdelen skulle kunna användas på Windows redan i nuvarande skick. Detta med reservationen för att Java-implementationen inte kan separera klienter i olika grupper av plattformar. Erlang-implementationens klient kan endast utföra ett kommando där det skickar tillbaka

mottagna meddelanden men med hjälp av biblioteket OS skulle anrop till kommandoraden enkelt kunna utföras. Exempelvis går det att visa filerna i en mapp genom följande anrop på en Unix-liknande dator (på en Windows-dator behövs `dir` istället för `ls`):

```
os:cmd("ls").
```

Det är även enkelt att hämta miljön för operativsystemet, hitta exekverbara filer och ändra vilken miljö som ska användas med hjälp av detta bibliotek[76]. Java har dock bättre stöd för input/output i standardbiblioteket vilket förbättrar dess förmåga att både exekvera kommandon och läsa in resultat, jämfört med Erlang.

Klustring av servrar på ett lokalt nätverk skulle vara lätt för Erlang-implementationen tack vare Distributed Erlang. Ett kluster som saknar ett skyddat nätverk för intern kommunikation skulle försvåra läget men även det skulle gå att lösa genom att använda SSL som transportprotokoll[77]. Problemet med att Distributed Erlang ger en användare med kontroll över en nod kontroll över andra noder kvarstår dock. Klustring för Java kan antingen uppnås genom att antingen skriva om koden för att låta flera datorer dela på arbetsbördan eller genom att gå över till Java Enterprise Edition och välja en applikationsserver med stöd för lastbalansering[78].

För att underlätta för administratörer vore grafiska gränssnitt lämpliga. Att skapa ett nytt grafiskt gränssnitt i Java är relativt enkelt. Det kan antingen göras genom att skriva egen kod eller med hjälp av ett verktyg som NetBeans Swing GUI Builder[79]. Det enda officiella Erlang-biblioteket för att skapa grafik är Graphic System[80] som hör till OTP. Andra bibliotek finns dock, till exempel gtkNode som är ett bibliotek som mappar GTK till Erlang[81] eller wxErlang[82]. Ett annat alternativ är att skapa ett GUI i Java och sedan kommunicera mellan Erlang-modulen och GUI för att binda samman dem.

4.3 Enhetstester

Enhetstester är ett bra hjälpmedel att försöka upptäcka buggar genom att testa små delar av systemet separat. I detta fallet var enhetstester skrivna med avseende på kravspecifikationen.

Java

Javas enhetstester är skapade med JUnit-ramverket som erbjuder ett antal funktioner för att garantera att testfallen utfaller på specificerat sätt. Bland annat erbjuds funktionssamlingen `assert` som används flitigt för att säkerställa testfallens korrekthet. Följande testfall användes för att säkerställa korrektheten i Java-implementationen:

- Skapa en klient och en dummy-server som skickar ett meddelande. Svaret testas sedan för att bekräfta att det uppfyller de krav som specificerats.
- Skapa en server och en testimplementation av klienten skickar ett meddelande. Händelseförloppet granskas för att säkerställa att inga parametrar har fel värden.

Testernas resultat visade mot all förmodan inga fel samtidigt som det finns stort utrymme att utveckla mer tester visar testerna att grundläggande funktion fungerar felfritt. Eftersom testaren litar på att Oracles databas fungerar korrekt testades inte denna, även om detta skulle kunna göras enkelt med Javas JUnit-ramverk.

Erlang

Enhetstester i Erlang-implementationen använde EUnit. EUnit är ett ramverk för att underlätta testning av moduler skrivna i Erlang. Testerna skrivs som funktioner och värde utvärderas med `assert`-makron. Det finns många olika `assert`-makron, men den som användes mest för testerna i detta projektet var

`?assertEqual(Expected, Expr)`. Samtliga tester som ska köras i EUnit måste ha namn som slutar med `_test()`. Alternativt kan en serie tester anropas från en funktion som heter `test()`.

Testerna för detta projektet gjordes mot servern genom att koppla upp en klient och säkerställa att servern fungerade som den skulle och är uppräknade nedan.

- Koppla upp en klient och skicka meddelande. Samma meddelande ska skickas tillbaka till servern och tas bort ur databasen.
- Koppla upp en klient och skicka flera meddelanden. Samtliga meddelanden ska skickas tillbaka till servern och tas bort ur databasen.
- Koppla upp en klient, simulera att servern tappar kontakten med klienten, skicka ett meddelande, koppla upp klienten igen och säkerställa att meddelandet levereras till klienten när uppkopplingen är tillbaka. Samtliga meddelanden ska sedan tas bort ur databasen.
- Koppla upp en klient, simulera att servern tappar kontakten med klienten, skicka flera meddelanden, koppla upp klienten igen och säkerställa att meddelandena levereras till klienten när uppkopplingen är tillbaka. Samtliga meddelanden ska sedan tas bort ur databasen.
- Skicka ett meddelande utan att någon klient är uppkopplad, koppla sedan upp en klient och säkerställ att inga meddelanden skickas till klienten.

Resultatet av testerna var förvånande då inga buggar orsakade av fellogik kunde hittas i applikationen. Vissa tester stötte dock emot en standard-timeout på fem sekunder i `gen_server` vid vissa anrop[83]. Det var dock inte alltid uppenbart om denna timeout trädde i kraft först efter fem sekunder, eller om den kunde utlösas tidigare.

Ett litet problem med testandet var att applikationen saknade viss funktionalitet för att kunna testas smidigt. Funktion för att till exempel kunna få tag på databasen var tvungen att läggas till. Ett annat problem är att EUnit i vissa fall försvårar arbetet vid felsökning genom att ändra felinformationen som skrivs ut. Följande felmeddelande uppstod vid en körning av ett EUnit-test:

```
::badarg
    output: <<"Client socket: #Port<0.2041> Connection is accepted.">>}
```

Felmeddelandet i sig ger inte mycket information om det faktiska problemet. Den enda information som ges är att ett argumentet är av fel typ. Samma kod ger dock följande felutskrift när det anropas som en vanlig funktion skilt från EUnit:

```
** exception error: bad argument
in function  exit/2
called as exit({ok,<0.247.0>},test)
```

Den senare av de två meddelandena ger betydligt mer information. Då `exit` endast ska skicka med en `Pid` och i anledningen till varför `exit` skickas framgår det att felet beror på att en tupel på formen {*ok, Pid*} skickas in som `Pid`, vilket leder till att testet kraschar.

4.4 Prestanda

Båda implementationer har testats med avseende på prestanda (genomförandet förklaras närmre i Appendix F). Principen har varit att variera antalet anslutna klienter och mängden trafik till och från servern. Följande nomenklatur används:

Klient En process som ansluter till servern över nätverket. Under testningen har alla klienter exekverats på en JVM/Erlang VM.

Kommando En instruktion som ska skickas till alla anslutna klienter.

Meddelande Separata datapaket som skickas mellan servern och dess klienter.

Relationen mellan storheterna är följande:

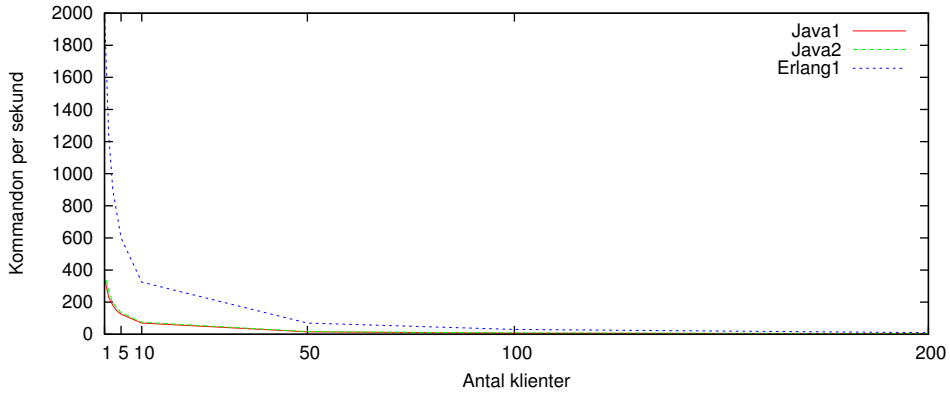
$$(\texttt{Meddelanden per sekund}) = (\texttt{Antal klienter}) * (\texttt{Kommandon per sekund})$$

När ett test har utförts med konstant antal meddelanden per sekund och varierande antal klienter mellan testerna kombinationer i tabell 4.1 använts.

Klienter	Kommandon per sekund	Meddelanden per sekund
1	100	100
5	20	100
10	10	100
20	5	100
50	2	100
100	1	100

Tabell 4.1: Inställningar vid tester med konstant genomströmning

Det högsta antalet exekverade kommandon per sekund mätas för respektive implementation givet ett visst antal anslutna klienter gav de tydligaste resultaten. Dessa värden illustreras i figur 4.1 och tabell 4.2. Serien Java1 visar resultaten då JVM:n och databasen befinner sig på två separata datorer och Java2 visar resultaten då JVM:n och databasen befinner sig på samma dator.

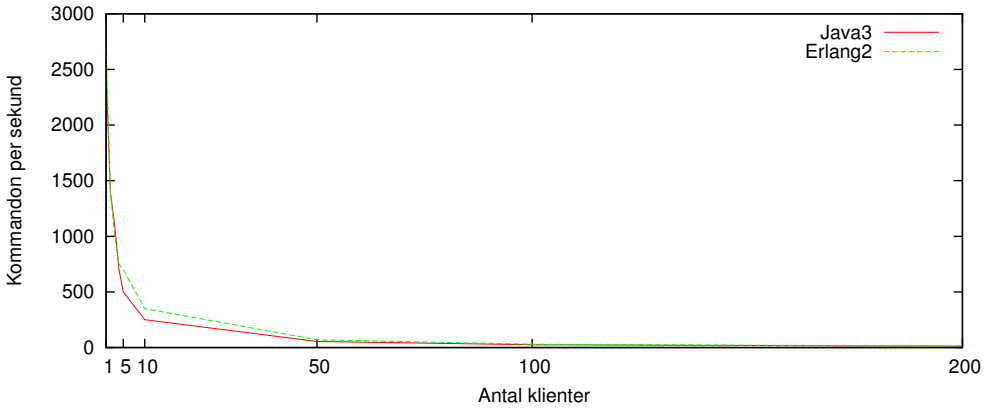


Figur 4.1: Maximalt antal exekverade kommandon per sekund.

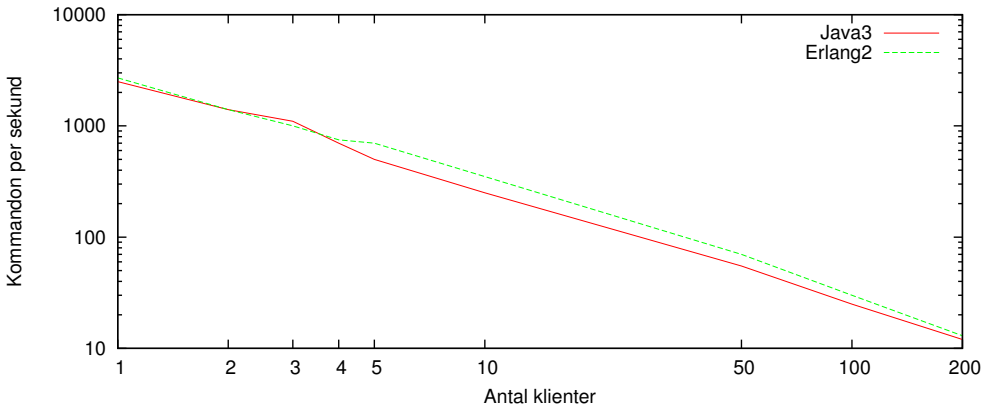
Klienter	Java1	Java2	Erlang1
1	350	385	2000
2	230	270	1250
3	180	195	900
4	145	165	750
5	125	135	600
10	70	75	325
50	15	17	70
100	7	8	30
200	3	4	10

Tabell 4.2: Maximalt antal exekverade kommandon per sekund

Skillnaden mellan de två implementationerna antogs bero på valet av databas och ytterligare tester utfördes där Java-implementationen utrustats med en *embedded* databas (som körs i samma JVM som Confector-servern). HyperSQL Database 1.8.0.10[84] hade så hög minneskonsumtion att tester avbröts på grund av att JVM fick slut på minne. Istället användes H2 Database Engine 1.3.166[85] som fungerade väl. Erlang-implementationen ändrades också för att liksom Java-implementationen hålla räknaren för nya meddelanden i minnet istället för databasen. Dessa två mätserier hänvisas till som Java3 och Erlang2. Likheterna mellan dem gör att en graf med logaritmisk skala (figur 4.2b) kan vara tydligare än en linjär graf (figur 4.2a). Observera att de optimerade implementationerna endast berörs i de två graferna nedan och tabell 4.3 . Alla övriga test av prestanda och korrekthet har utgått ifrån de ursprungliga implementationerna.



(a) Optimerade versioner, vanlig skala.



(b) Optimerade versioner, logaritmisk skala.

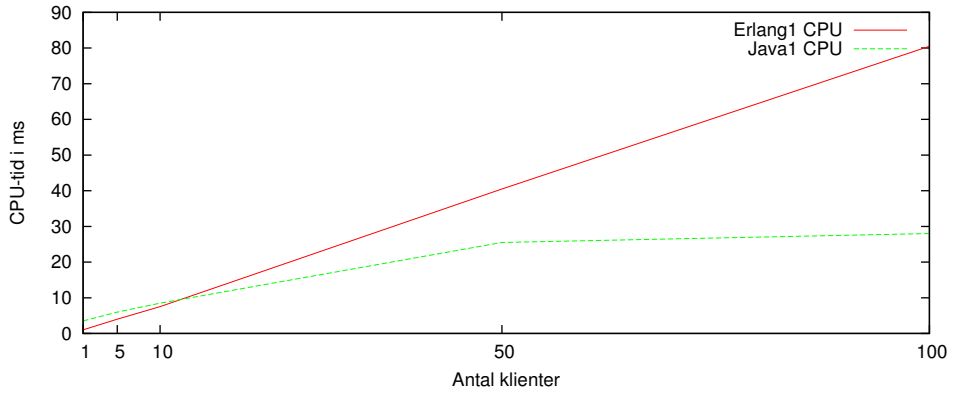
Figur 4.2: Kapacitet med databaser i samma JVM/Erlang VM som Confector.

4.4.1 Resursanvändning

Mätningar av resursanvändning såsom arbetsminne och processoranvändning gav mer tvetydig information. Processoranvändningen kunde endast mätas utifrån operativsystemets begrepp *CPU time*. Denna siffra som motsvarar tiden då processen körts på processorn[86] räknas bara till närmaste millisekund och för många belastningar blev det några enstaka millisekunders *CPU time*, vilket alltså ger dålig upplösning i mätningarna.

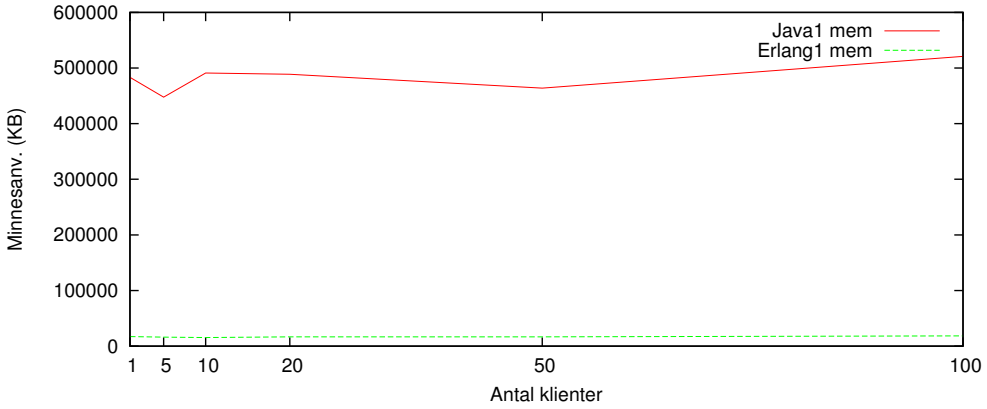
Klienter	Java3	Erlang2
1	2500	2700
2	1400	1400
3	1100	1000
4	700	750
5	500	700
10	250	350
50	55	70
100	25	30
200	12	13

Tabell 4.3: Maximalt antal exekverade kommandon per sekund för optimerade versioner



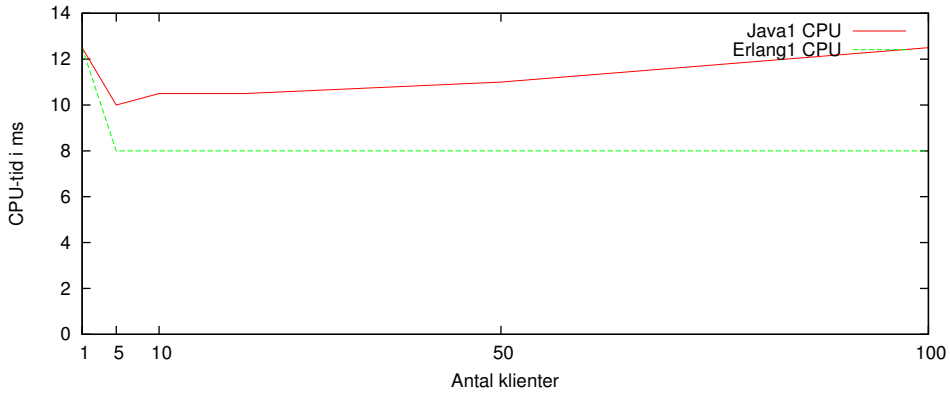
Figur 4.3: Processoranvändning vid varierande genomströmning.

Några mätserier kring processoranvändning erhölls som visar på trender som överstiger variationer mellan enskilda mätningar. Att låta de två implementationerna ta emot anslutningar från tio klienter och sedan variera antalet kommandon som ska utfärdas per sekund över 1, 5, 10, 50 och 100 visade tydligt Erlangs processoranvändning stiger i snabbare takt än den för Java (figur 4.3). Som nämndes tidigare klarar inte den ursprungliga Java-implementationen av 100 kommandon per sekund för tio klienter vilket innebär att mätvärdet för 100 kommandon per sekund snarare motsvarar 70 kommandon per sekund i faktisk genomströmning.



Figur 4.4: Minnesanvändning vid varierande antal klienter och konstant genomströmning.

Tyvärr reserverade JVM stort utrymme i arbetsminnet vid start även med ett minimalt program som endast väntar. Detta gör att en jämförelse av minnesanvändning i absoluta tal(figur 4.4) inte säger särskilt mycket om de två implementationerna. En jämförelse av hur minnesanvändningen *förändras* i takt med att antalet klienter ökar antyder dock att Erlang inte behöver använda särskilt mycket minne för hantera flera klienter.



Figur 4.5: Processoranvändning vid varierande antal klienter och konstant genomströmning.

En jämförelse av använd *CPU time* för de två implementationerna under samma omständigheter, 100 meddelanden per sekund uppdelat på 1, 5, 10, 20, 50 och 100 klienter, visar att processoranvändning minskar när antalet kommandon per sekund minskar och ersätts av fler klienter (figur 4.5). På grund av den dålig upplösningen i mätvärdena samt Java-implementationens något svårtolkade utveckling är det en mer konservativ observation av processoranvändningen inte ökar under dessa omständigheter.

4.4.2 Avvägningar

Valet av databas visar sig i detta fallet ha en större inverkan på den totala genomströmningskapaciteten än valet av programmeringsspråk. Med inbyggda databaser uppnådde de två implementationerna näst intill identiska resultat. Sådana databaser kan dock stöta på andra problem som totalt lagringsutrymme. Databasen Mnesia som använts i Erlang-implementationen hanterar exempelvis inte tabeller större än två GB[57] utan särskilda inställningar och håller oftast hela databasen i arbetsminnet[87]. Även den inbyggda databasen som användes för Java3-testet hade vissa svårigheter, främst att tömma ut data till hårddisk. Sökningar på internet om detta problem ger få resultat vilket antyder att problemet berodde på den lokala konfigurationen snarare än databasen i sig.

5 Slutsatser

Denna studie har berört många olika aspekter av både Java och Erlang tillsammans med språkneutrala frågor som nätverkstopologi och säkerhet. Just arbetets bredd har begränsat djupet vilket gör att klara slutsatser om vilket språk som är lättast att lära sig eller mest flexibelt inte går att dra. Det har dock framkommit att både Java och Erlang är väl lämpade för att utveckla nätverksbaserade verktyg med var sitt sätt att hantera många parallella anslutningar. De kan båda på ett överkomligt sätt kommunicera med databaser och samverka med det underliggande operativsystemet.

Huruvida ett visst språk lämpar sig för ett projekt beror inte enbart på dess tekniska egenskaper utan även tillgången på programmerare, dokumentation och verktyg. Att rekrytera ny personal eller utbilda existerande personal innebär ofta extra kostnader[88] vilket utgör en tröskel att byta programmeringsspråk inom ett projekt eller på en arbetsplats. Java har påtagliga fördelar i detta avseende med många användare och ett stort urval av avancerade verktyg.

En återkommande fråga under arbetets gång har handlat just om hur enkelt eller svårt Erlang är att lära sig. Att skapa en Java-grupp med två personer och en Erlang-grupp med fyra personer berodde till stor del på antagandet att det skulle krävas mycket arbete för Erlang-gruppen att både lära sig språket och utveckla en prototyp. Det visade sig vara en missbedömning och Java-gruppen hamnade efter tidsplanen medan Erlang-gruppen hade goda marginaler. En jämn fördelning av medlemmar mellan de två språken hade alltså varit mycket bättre.

Ett mer subjektivt resultat kommer från en omröstning om vilket språk som borde ha använts om syftet med arbetet inte var att jämföra två programmeringsspråk utan bara utveckla en prototyp i Java eller Erlang. Då röstade fyra av sex medlemmar för Java medan rösterna var jämnt fördelade när frågan gällde vilken implementation som borde användas för vidare utveckling.

Jämförelsen av prestanda är inte särskilt praktiskt relevant då båda implementationer visade sig klara betydligt högre belastningar än vad som kan anses rimligt för ett administrationsverktyg. För andra sorters verktyg med ett stort antal klienter och många transaktioner per sekund kan det däremot vara av större betydelse. Den största skillnaden mellan språken i detta avseende är Javas låga processoranvändning vid hög belastning och de små ökningar i Erlangs minnesanvändning som uppstår när antalet samtidiga anslutningar ökar. Ett något överraskande resultat var att Java-implementationen utrustad med en intern databas baserad på SQL hade ungefär samma kapacitet som Erlang-implementationen vars interna databas använder en enkel nyckel/värde-uppslagning.

6 Fortsatt arbete

Innehållet i detta arbete är väldigt brett och en eventuell fortsättning skulle kunna fokusera på något av delmomenten. För ett nätverksbaserat verktyg finns det många ytterligare områden att granska. Som nämndes tidigare vore en mer omfattande utvärdering och diskussion kring olika säkerhetstekniker och dess tillämpningsområde varit ett givande arbete. Likaså finns det ett stor urval av databaser som kan studeras, exempelvis NoSQL-databaser.

En annan intressant fortsättning på arbetet vore att studera de två språken närmre och göra en teknisk analys för att identifiera potentiella brister i respektive språk. Att försöka lösa säkerhetsproblemen med Distributed Erlang skulle kunna utgöra en annan gren av arbetet. Det som ligger närmast till hands är att vidareutveckla de två prototyperna och utöka funktionaliteten samt utföra en mer genomgående jämförelse av utvecklingen av systemen.

Litteraturförteckning

[1] Java SE Core Technologies - CORBA / RMI-IIOP [Webbsida] Oracle [Hämtdatum 2012-05-06] Tillgänglig på: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138209.html

[2] Graba J. An Introduction to network programming with Java. London: Springer; 2007. ss. 12.

[3] Erlang - gen_tcp [Webbsida] Ericsson AB [Hämtdatum 2012-04-28] Tillgänglig på: http://www.erlang.org/doc/man/gen_tcp.html

[4] Erlang - Distributed Erlang [Webbsida] Ericsson AB [Hämtdatum 2012-04-28] Tillgänglig på: http://www.erlang.org/doc/reference_manual/distributed.html#id82772

[5] Oracle JDK 7 and JRE 7 Certified System Configurations [Webbsida] Oracle [Hämtdatum 2012-04-28] Tillgänglig på: http://www.oracle.com/technetwork/java/javase/config-417990.html

[6] Erlang - Building and installing Erlang/OTP [Webbsida] Ericsson AB [Hämtdatum 2012-04-28] Tillgänglig på: http://www.erlang.org/doc/installation_guide/INSTALL.html#Daily-Build-and-Test

[7] Oracle customer successes [Webbsida] Oracle [Hämtdatum 2012-04-28] Tillgänglig på: http://www.oracle.com/us/corporate/customers/index.html

[8] Armstrong J. Erlang - A survey of the language and its industrial applications; 1996

[9] Manabe A, Kawabata S. Administration tools for managing large-scale Linux cluster. Proceedings of the VIII International Workshop on Advanced Computing and Analysis Techniques in Physics Research. 2003; 2/3: 475-477.

[10] Halén J, Karlsson R, Nilsson M. Performance Measurements of Threads in Java and Processes in Erlang; 1998-11-02

[11] Kapoor B, Pandya P, Sherif JS. Cryptography: A security pillar of privacy, integrity and authenticity of data communication. Kybernetes. 2011; 40(9): 1422 - 1439

[12] Eskicioglu A, Litwin L. Cryptography. Potentials, IEEE. 2001 Feb/Mar; 20(1): 36-38

[13] Cayre F, Bas P. Kerckhoffs-Based Embedding Security Classes for WOA Data Hiding. Information Forensics and Security, IEEE Transactions on. 2008 March; 3(1): 1-15

[14] Secure Socket Layer Reference Manual [Webbsida] Ericsson AB [Hämtdatum 2012-04-06] Tillgänglig på: http://www.erlang.org/doc/apps/ssl/index.html

[15] The GNU Privacy Guard [Webbsida] Free Software Foundation [Hämtdatum 2012-04-06] Tillgänglig på: http://www.gnupg.org/

[16] RFC4880, s. 5-6 [Webbsida] The Internet Engineering Task Force [Hämtdatum 2012-04-06] Tillgänglig på: http://www.ietf.org/rfc/rfc4880.txt

[17] Frantisek, F. Kapitel 1 i Memory as a programming concept in C and C++. Cambridge University Press; 2004.

[18] Kurose. Computer Networking - A Top-Down Approach. Pearson Education International; 2008, s. 110

[19] Weija. Distributed Network Systems - From Concepts to Implementations. Springer Science + Business Media; 2005, s. 17

[20] Kurose. Computer Networking - A Top-Down Approach. Pearson Education International; 2008, s. 171-174

[21] Karrels, Peterson, Mullins. Structured P2P technologies for distributed command and control. Springer Science + Business Media; 2009.

[22] Weija. Distributed Network Systems - From Concepts to Implementations. Springer Science + Business Media; 2005, s. 461

[23] Weija. Distributed Network Systems - From Concepts to Implementations. Springer Science + Business Media; 2005, s. 34

[24] Weija. Distributed Network Systems - From Concepts to Implementations. Springer Science + Business Media; 2005, s. 41

[25] Weija. Distributed Network Systems - From Concepts to Implementations. Springer Science + Business Media; 2005, s. 43

[26] Weija. Distributed Network Systems - From Concepts to Implementations. Springer Science + Business Media; 2005, s. 33

[27] The history of Java technology [Webbsida] Oracle [Hämtdatum 2012-05-06] Tillgänglig på: http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html

[28] Java Technology: The Early Years [Webbsida] Sun Microsystems [Hämtdatum 2012-05-06] Tillgänglig på: http://web.archive.org/web/20080530073139/http://java.sun.com/features/1998/05/birthday.html

[29] TIOBE Programming Community Index for April 2012 [Webbsida] TIOBE Software [Hämtdatum 2012-05-06] Tillgänglig på: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[30] Java Language and Virtual Machine Specifications [Webbsida] Oracle [Hämtdatum 2012-05-06] Tillgänglig på: http://java.sun.com/docs/books/jvms/

[31] Excelsior JET Internals [Webbsida] Excelsior [Hämtdatum 2012-05-11] Tillgänglig på: http://www.excelsior-usa.com/jetinternals.html

[32] Java SE HotSpot at a glance [Webbsida] Oracle [Hämtdatum 2012-05-06] Tillgänglig på: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html

[33] The Java HotSpot Performance Engine Architecture [Webbsida] Oracle [Hämtdatum 2012-05-06] Tillgänglig på: http://java.sun.com/products/hotspot/whitepaper.html

[34] The Java HotSpot Virtual Machine. [Webbsida] Oracle [Hämtdatum 2012-05-06] Tillgänglig på: http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html

[35] Frequently Asked Questions About the Java HotSpot VM [Webbsida] Oracle [Hämtdatum 2012-05-06] Tillgänglig på: http://www.oracle.com/technetwork/java/hotspotfaq-138619.html#64bit_compilers

[36] Symantec's Just-In-Time Java Compiler To Be Integrated Into Sun JDK 1.1 [Webbsida] Symantec Corporation [Hämtdatum 2012-05-06] Tillgänglig på: http://www.symantec.com/about/news/release/article.jsp?prid=19970407_03

[37] The Java HotSpot Virtual Machine, v1.4.1 [Webbsida] Oracle [Hämtdatum 2012-05-06] Tillgänglig på: http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_4.html

[38] Memory Management in the Java HotSpot Virtual Machine, Sun Microsystems; April 2006

[39] Sanden, B. Computer. 2004; 37(4): 20-23.

[40] Java RMI Tutorial [Webbsida] Kenneth Baclawski [Hämtdatum 2012-05-06] Tillgänglig på: http://www.eg.bucknell.edu/ cs379/DistributedSystems/rmi_tut.html

[41] jGuru: Remote Method Invocation [Webbsida] Oracle [Hämtdatum 2012-05-06] Tillgänglig på: http://java.sun.com/developer/onlineTraining/rmi/RMI.html

[42] Java Remote Method Invocation - Distributed Computing for Java [Webbsida] Oracle [Hämtdatum 2012-05-08] Tillgänglig på: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html

[43] Java SE Security [Webbsida] Oracle [Hämtdatum 2012-05-06] Tillgänglig på: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html

[44] Interface Serializable [Webbsida] Oracle [Hämtdatum 2012-05-06] Tillgänglig på: http://docs.oracle.com/javase/6/docs/api/java/io/Serializable.html

[45] Sierra K, Bates B. Sun Certified Programmer for Java 6 Study Guide. McGraw-Hill Osborne Media; 2008. Kapitel 6, Serialization

[46] Cesarini F, Thompson S. Erlang Programming. Sebastopol: O'Reilly Media, Inc; 2009. s 41-42

[47] Erlang - Distributed Applications [Webbsida] Ericsson AB [Hämtdatum 2012-05-10] Tillgänglig på: http://www.erlang.org/doc/design_principles/distributed_applications.html

[48] Erlang Programming Language [Webbsida] [Hämtdatum 2012-05-06] Tillgänglig på: http://www.erlang.org/about.html

[49] Erlang - Implementation and Ports of Erlang [Webbsida] Ericsson AB [Hämtdatum 2012-05-06] Tillgänglig på: http://www.erlang.org/faq/implementations.html#id53691

[50] Erlang - Academic and Historical Questions [Webbsida] Ericsson AB [Hämtdatum 2012-05-06] Tillgänglig på: http://www.erlang.org/faq/academic.html#id55775

[51] Johansson E. HiPE Technical Reference; 2003. [Hämtdatum 2012-05-31] Tillgänglig på: http://www.it.uu.se/research/group/hipe/documents/hipe_manual.pdf

[52] High-Performance Erlang [Webbsida] Uppsala Universitet [Hämtdatum 2012-05-31] http://www.it.uu.se/research/group/hipe/

[53] Foundations of Actor Semantics av Clinger, William Douglas; 1981, s 14-17

[54] Distributed Erlang. [Webbsida] Ericsson AB [Hämtdatum 2012-05-06] Tillgänglig på: http://www.erlang.org/doc/reference_manual/distributed.html

[55] Rikitake K, Nakao K. Application Security of Erlang Concurrent System; 2008

[56] Intern dokumentation av OTP. (1999) https://github.com/erlang/otp/blob/master/lib/kernel/internal_doc/distribution_handshake.txt

[57] Cesarini F, Thompson S. Erlang Programming. Sebastopol: O'Reilly Media, Inc.; 2009

[58] Erlang Frequently Asked Questions [Webbsida] Erlang.org [Hämtdatum 2012-05-06] Tillgänglig på: http://www.erlang.org/faq/introduction.html#id49469

[59] Erlang/OTP R15B01 [Webbsida] Ericsson AB [Hämtdatum 2012-05-06] Tillgänglig på: http://www.erlang.org/doc/

[60] Cesarini F, Thompson S. Erlang Programming. Sebastopol: O'Reilly Media, Inc.; 2009. s 264-266

[61] Erlang - Supervisor Behaviour [Webbsida] Ericsson AB [Hämtdatum 2012-05-06] Tillgänglig på: http://www.erlang.org/doc/design_principles/sup_princ.html

[62] Who Supervises The Supervisors? [Webbsida] Learn You Some Erlang For Great Good [Hämtdatum 2012-05-06] Tillgänglig på: http://learnyousomeerlang.com/supervisors#supervisor-concepts

[63] Cesarini F, Thompson S. Erlang Programming. Sebastopol: O'Reilly Media, Inc.; 2009. s 281-282

[64] Kurose. Computer Networking - A Top-Down Approach. Pearson Education International; 2008, s. 235

[65] Kurose. Computer Networking - A Top-Down Approach. Pearson Education International; 2008, s. 184

[66] Advanced Socket Programming [Webbsida] Oracle [Hämtdatum 2012-04-27] Tillgänglig på: http://java.sun.com/developer/technicalArticles/ALT/sockets/

[67] Schildt H. Swing A Beginners Guide. McGraw-Hill; 2007. ss 5

[68] Jia, X. Object oriented software development with Java. Boston: Addison Wesley; 2003. ss. 210,211.

[69] Bloch J. Effective Java Programming Language Guide. Boston: Addison Wesley; 2007. ss. 63.

[70] Javaforum [Webbsida] [Hämtdatum 2012-05-01] Tillgänglig på: http://www.javaforum.se

[71] OTN Discussion Forums : Java [Webbsida] Oracle [Hämtdatum 2012-05-01] Tillgänglig på: https://forums.oracle.com/forums/category.jspa?categoryID=285

[72] Logan M, Merritt E, Carlsson R. Erlang and OTP in Action. Greenwich, Conn. : Manning; 2011.

[73] Erlang - Erlang Tools [Webbsida] Ericsson AB [Hämtdatum 2012-04-27] Tillgänglig på: http://www.erlang.org/faq/tools.html#id53125

[74] Erlide Features [Webbsida] [Hämtdatum 2012-04-27] Tillgänglig på: https://github.com/erlide/erlide/wiki/Features

[75] Erlang - odbc [Webbsida] Ericsson AB [Hämtdatum 2012-05-06] Tillgänglig på: http://www.erlang.org/doc/man/odbc.html

[76] Erlang - os [Webbsida] Ericsson AB [Hämtdatum 2012-05-06] Tillgänglig på: http://www.erlang.org/doc/man/os.html

[77] Erlang - Using SSL for Erlang Distribution [Webbsida] Ericsson AB [Hämtdatum 2012-02-20] Tillgänglig på: http://www.erlang.org/doc/apps/ssl/ssl_distribution.html

[78] Kuo, X. GlassFish Administration : Administer and Configure the GlassFish v2 application Server. Birmingham, GB: Packt; 2009. ss. 173.

[79] Swing GUI Builder (formerly Project Matisse) [Webbsida] Oracle [Hämtdatum 2012-05-05] Tillgänglig på: http://netbeans.org/features/java/swing.html

[80] Erlang - Graphics System (GS) Reference Manual [Webbsida] Ericsson AB [Hämtdatum 2012-05-06] Tillgänglig på: http://www.erlang.org/doc/apps/gs/index.html

[81] gtknode - GTK+2 binding for Erlang [Webbsida] [Hämtdatum 2012-05-06] Tillgänglig på: http://code.google.com/p/gtknode/

[82] wxerlang [Webbsida] [Hämtdatum 2012-05-06] Tillgänglig på: http://sourceforge.net/apps/mediawiki/wxerlang/index.php?title=Main_Page

[83] Erlang - gen_server [Webbsida] Ericsson AB [Hämtdatum 2012-05-06] Tillgänglig på: http://www.erlang.org/doc/man/gen_server.html#start_link-4

[84] HSQLDB [Webbsida] The hsql Development Group [Hämtdatum 2012-05-08] Tillgänglig på: http://hsqldb.org

[85] H2 Database Engine [Webbsida] [Hämtdatum 2012-05-08] Tillgänglig på: http://www.h2database.com/html/main.html

[86] Kerrisk M. The Linux Programming Interface: Linux and UNIX System Programming Handbook. San Fransisco, USA: No Starch Press; 2010. Kapitel 10, avsnitt 7.

[87] Erlang - Miscellaneous Mnesia Features [Webbsida] Ericsson AB [Hämtdatum 2012-05-09] Tillgänglig på: http://www.erlang.org/doc/apps/mnesia/Mnesia_chap5.html#id75194

[88] O'Connor, R. A study of computer programming language adoption for information systems development projects. International journal of technology, policy and management. 2003; 3(3/4): 343.

[89] Class String [Webbsida] Oracle [Hämtdatum 2012-05-06] Tillgänglig på: http://docs.oracle.com/javase/6/docs/api/java/lang/String.html

[90] Class ObjectOutputStream [Webbsida] Oracle [Hämtdatum 2012-05-06] Tillgänglig på: http://docs.oracle.com/javase/6/docs/api/java/io/ObjectOutputStream.html

[91] Class ObjectInputStream [Webbsida] Oracle [Hämtdatum 2012-05-06] Tillgänglig på: http://docs.oracle.com/javase/6/docs/api/java/io/ObjectInputStream.html

[92] Inside the Erlang VM with focus on SMP förberedd av Kenneth Lundin, Ericsson AB Presentation vid Erlang User Conference, Stockholm, November 13, 2008

[93] Logan M, Merritt E, Carlsson R. Erlang and OTP in Action by Martin Logan; 2010. s 143

[94] Erlang - qlc [Webbsida] Ericsson AB [Hämtdatum 2012-05-09] Tillgänglig på: http://www.erlang.org/doc/man/qlc.html

[95] Erlang - Introduction [Webbsida] Ericsson AB [Hämtdatum 2012-05-09] Tillgänglig på: http://www.erlang.org/doc/apps/mnesia/Mnesia_chap1.html

[96] Erlang - Transactions and Other Access Contexts [Webbsida] Ericsson AB [Hämtdatum 2012-05-09] Tillgänglig på: http://www.erlang.org/doc/apps/mnesia/Mnesia_chap4.html

[97] VMware vSphere 4.1 Networking Performance. WMware [Hämtdatum 2012-05-06] Tillgänglig på: http://www.vmware.com/files/pdf/techpaper/Performance-Networking-vSphere4-1-WP.pdf

[98] Bilting, U. Vägen till C. Lund: Studentlitteratur; 2000. ss. 2.

[99] Huss, E. The C library reference guide [Webbsida] [Hämtdatum 2012-04-30] Tillgänglig på: http://www.acm.uiuc.edu/webmonkeys/book/c_guide/index.html

[100] Amdani, S. Y. C Programming. New Dehli: Laxmi; 2009. Kapitel 2.

[101] Eaddy, M. Dr. Dobb's Journal, volym 26, nr. 2; 2001. ss. 74.

[102] Freeman, A. Introducing Visual C# 2010. New York: Apress; 2010, ss. 29.

[103] Mono [Webbsida] Mono Project [Hämtdatum 2012-04-30] Tillgänglig på: http://www.mono-project.com/Main_Page

[104] About Python [Webbsida] Python Software Foundation [Hämtdatum 2012-04-30] Tillgänglig på: http://www.python.org/about/

[105] Computer programming for everybody [Webbsida] [Hämtdatum 2012-04-30] Tillgänglig på: http://www.python.org/doc/essays/cp4e.html

A Kravspecifikation

Kraven för de två prototyperna utformades så att funktionallitet som berör kommunikationsprotokollet är gemensamt hos de två prototyperna. Även viss felhantering har specificerats. Eftersom inte allt har specificerats har prototyperna utvecklats med förhållandevis stor frihet för att gynna programmeringsspråkens paradigmer och således bidragit till att en så väl anpassad lösning som möjligt tagits fram. För prototyperna har kravet på säkerhet sänkts för att undvika komplikationer kring SSL och de certifikat som då krävs.

Klienten:

- Ansluta till server med ett id, detta id specificeras av klienten innan denna ansluter till servern.
- Ta emot godtyckligt meddelande från server och skicka tillbaka samma meddelande.
- Ska kunna utföra parallella anrop från servern.

Servern:

- När en klient ansluter kontrolleras dess id om den varit ansluten eller är ansluten.
- Varje anslutet id för klienter ska vara unikt.
- Sköter loggning av vilka klienter som är anslutna och vilka som har varit anslutna.
- Ta bort klient.
- Skicka meddelanden till samtliga klienter, för de som inte är anslutna kommer de missade meddelandena att skickas i efterhand när denna klient ansluter på nytt.
- När en ny klient ansluter tar denna endast del av kommande meddelanden.
- När servern skickar meddelanden till klienterna håller servern reda på vilka meddelanden som skickats och vilka av dessa som är obesvarade från klienterna.
- Meddelanden behöver inte skickas eller tas emot i någon speciell ordning.

B Java: Serialization med sockets

I detta exemplet så kommer en Server att skicka ett objekt till en klient med hjälp av sockets och serialisering. Klienten kommer i första steget att öppna en anslutning till Servern. Servern har i förväg skapat ett objekt av klassen Command som den kommer att serialisera och skriva till socketens output stream. Klienten kommer därefter att deserialisera detta objektet och skriva ut det till terminalen.

Klassen Command är det objekt som kommer att serialiseras. Det implementerar Serializable och innehåller en primitiv datatyp och ett Stringobjekt. Eftersom String implementerar Serializable[89] så kommer även det objektet att serialiseras hos Server och överföras[45] till klienten.

```
import java.io.*;

public class Command implements Serializable {
  String command;
  boolean wantAnswer;

  public Command(String c, boolean w) {
    command = c;
    wantAnswer = w;
  }

  public String toString() {
    return command + ", want answer: " + wantAnswer;
  }
}
```

Servern i detta exempelet behöver göra följande saker:

- Instansierar ett Command objekt som kommer att serialiseras och överföras via en socket.
- Starta en ServerSocket för att kunna invänta en anslutning. Detta sker med `servSockaccept()`.
- När en klient är ansluten skapas en `ObjectOutputStream` som med hjälp av metoden `writeObject(cmd)` skriver objektet till till ObjectOutputStream[90].

```
import java.io.*;
import java.net.*;

public class Server {
  public static void main(String... args) throws IOException {
    Command cmd = new Command("touch file$RANDOM", true);

    ServerSocket servSock = new ServerSocket(9001);
    Socket client = servSock.accept();

    ObjectOutputStream outObject =
          new ObjectOutputStream(client.getOutputStream());
    outObject.writeObject(cmd);
    outObject.flush();
    outObject.close();
  }
}
```

Klienten är det program som skapar anslutningen till Servern via en socket. Det skapar en `ObjectInputStream` och använder sedan metoden `readObject()`[91] för att deserialisera objektet. I detta exempelet så gör klienten ingen mer än att skriva ut det mottagna objektet.

```
import java.net.*;
import java.io.*;

public class Client {
  public static void main(String... args)
    throws IOException, ClassNotFoundException {

        Socket socket = new Socket("127.0.0.1", 9001);

    ObjectInputStream ois =
          new ObjectInputStream(socket.getInputStream());
    Command cmd = (Command)ois.readObject();
    System.out.println("Got command: " + cmd );
  }
}
```

C Java: RMI-exempel

Följande exempel är mycket primitivt men visar på det centrala i RMI. I exemplet finns en klient som tar emot tre argument. Det första argumentet är det IP-nummer på vilket servern befinner sig (till exempel 127.0.0.1) följt av två stycken heltal. Dessa skulle kunna ses som höjd och bred. Klienten gör sedan ett metodanrop till servern och får tillbaka ett resultat, i detta fallet en area.

För att klienten ska känna till vilka metoder som finns tillgängliga hos servern så behövs ett gränssnitt. Detta gränssnitt ärver från `Remote` och implementeras av servern. I detta exempel används följande gränssnitt:

```
import java.rmi.*;

public interface ConfectorServerInterf extends Remote {
  public int area(int width, int height) throws RemoteException;
}
```

Själva klienten ser ut på följande vis:

```
import java.net.*;
import java.rmi.*;

public class ConfectorClient {
  public static void main(String[] args) {
    try {
      String getServer = "rmi://" + args[0] + "/ConfectorServer";
      ConfectorServerInterf con =
                (ConfectorServerInterf)Naming.lookup(getServer);

      int i1 = Integer.valueOf(args[1]);
      int i2 = Integer.valueOf(args[2]);
      System.out.println("The area is: " + con.area(i1, i2));

    } catch (RemoteException rex) {
      rex.printStackTrace();
    } catch (MalformedURLException mex) {
        mex.printStackTrace();
      } catch (NotBoundException nex) {
        nex.printStackTrace();
    }
  }
}
```

För att klienten ska fungera saknas en stub-fil. Stub-filen skapas med hjälp av programmet **rmic** och själva implementationen av servern vilket förklaras närmre senare.

Servern implementeras på följande sätt:

```
import java.rmi.*;
import java.rmi.server.*;

public class ConfectorServer extends UnicastRemoteObject
  implements ConfectorServerInterf {

  public ConfectorServer() throws RemoteException {};

  @Override
  public int area(int width, int height) throws RemoteException {
    return width * height;
  }
}
```

För att starta servern används följande program:

```
import java.net.*;
import java.rmi.*;

public class StartServer {
  public static void main(String[] args) {
        try {
          ConfectorServer cs = new ConfectorServer();
          Naming.rebind("ConfectorServer", cs);

        } catch (RemoteException rex) {
          rex.printStackTrace();
        } catch (MalformedURLException mex) {
          mex.printStackTrace();
        }
  }
}
```

För att starta exemplet behövs först alla källkodsfiler kompileras som vanligt med **javac**. Verktyget **rmic** används sedan för att skapa en stub av server-implementationen. I detta exemplet blir det **rmic ConfectorServer**. På servern (klient kan köras på samma maskin) behöver rmi-registret startas vilket görs genom att skriva **rmiregistry**. Efter detta kan server-programmet startas med **java StartServer**. På klienten exekveras exemplet med **java ConfectorClient**.

D Erlang SMP och distribution

Sedan OTP R11B har Erlang stöd för SMP (Symmetric Multi Processing)[92]. SMP i Erlang är aktiverat om inget annat anges. Före SMP fanns det enbart en schemaläggare i huvudprocessen eftersom det endast fanns en tråd som kördes. Schemaläggaren har i uppgift att schemalägga Erlangprocesser och Input/Output-jobb som ligger i körningskön.

Sedan SMP infördes ser det däremot annorlunda ut. SMP har stöd för upp till 1024 schemaläggare [92], där varje schemaläggare använder sig av en egen tråd. Vanligtvis är det ingen fördel med att använda fler schemaläggare än processorer eller kärnor. Alla datastrukturer i Erlang SMP VM som delas mellan de olika schemaläggarna är låsskyddade. Ett exempel på en delad datastruktur är körningskön. Schemaläggarna körs i varsin operativsystem-tråd och dess schemaläggning bestäms av operativsystemet[93].

Distributed Erlang

Distributed Erlang är ett system i Erlang som låter olika noder skapa processer på och skicka meddelanden till varandra på samma sätt som processer och meddelanden hanteras inom enskilda Erlangnoder.

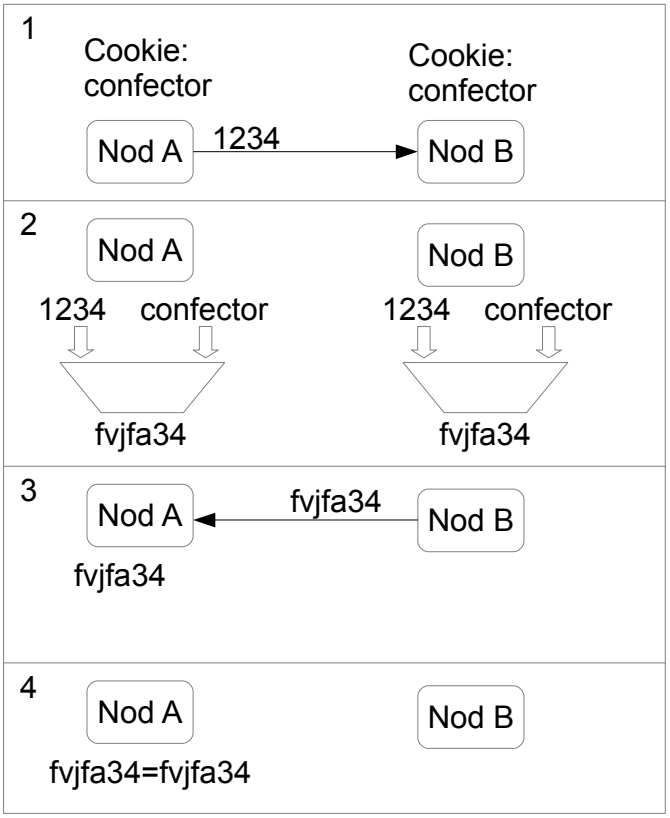
I Erlang räknas en instans av Erlang Runtime System (ERTS) som en nod. Det är alltså möjligt att ha flera noder på en och samma dator, men alla noder på en dator måste ha olika namn. Nodnamn kan ges antingen i kort eller lång form[54].

Kort form erl -sname server@erlnode1

Lång form erl -name server@erlnode1.chalmers.se

En nod med namn på kort form kan inte kommunicera via Distributed Erlang med en nod med ett namn på lång form[54].

Noder kan inte heller kommunicera genom Distributed Erlang om de inte har samma cookie. Detta är antingen en sträng som läses in från en fil eller som ges vid kommandoraden när en Erlang-nod startas. För att noder ska kunna avgöra om de har samma cookie-värde utan att behöva skicka informationen i klartext används ett challenge/response-förfarande när en anslutning skapas mellan två noder. Däremot kommer all annan trafik att skickas i klartext mellan noderna[55]. (Beskrivningen av processen är förenklad men representativ)



Figur D.1: Schematisk beskrivning av cookie-verifiering i Distributed Erlang.[56]

Nod A skickar en slumpmässigt vald 32-bitars siffra till nod B som skapar en kontrollkod (*message digest*) av siffran och sitt cookie-värde. Nod B skickar tillbaka sin kod till nod A som har utfört samma operation själv och kan avgöra om nod B har använt rätt cookie-värde när den skapade sin kontrollkod[56]. Samma procedur upprepas sedan för den andra noden.

Kodexempel

I följande kod startar en process en annan process och skickar den ett meddelande. Den andra processen svarar till den första som sedan skriver ut svaret på skärmen.

```
-module(dist).
-export([sender/1, adder/0]).

sender(Num) ->
    Adder = spawn(dist, adder, []),
    Adder ! {num,self(),Num},
    receive
    {num,NumPlusOne} ->
        io:format("Num plus one => ~w~n", [NumPlusOne]);
    Other ->
        io:format("Unexpected message: ~w~n", [Other])
    end.
```

```
adder() ->
    receive
    {num,Src,Num} ->
        Src ! {num,Num+1};
    Other ->
        Other
    end.
```

Exekvering i ett Erlang-skal ger följande resultat:

```
(gen@laptop)1> dist:sender(2).
Num plus one => 3
ok
```

Att använda Distributed Erlang för att låta `adder` exekveras på en annan nod behöver bara två rader i koden ändras, nämligen början på `sender`. Ett nodnamn ges som argument till funktionen och det värdet ges till `spawn` för att indikera att processen `dist:adder` ska skapas på den noden.

```
...
sender(Node,Num) ->
    Adder = spawn(Node,dist, adder, []),
...
```

Nu startas Erlang-noder på två datorer, laptop och erlnode1. Båda behöver startas i en mapp där filen `dist.beam` finns. I detta fallet är det en NFS-mapp som båda datorerna har tillgång till. För att de två ska koppla upp sig mot varandra automatiskt skapas en fil .hosts.erlang i denna mapp där de två värdnamnen finns inskrivna:

```
'laptop'
'erlnode1'
```

laptop erl -sname gen@laptop -setcookie dist_test

erlnode1 erl -sname gen@erlnode1 -setcookie dist_test

Följande exekvering utförs på laptop:

```
(gen@laptop)1> dist:sender(gen@erlnode1,2).
Num plus one => 3
ok
```

Det är också möjligt att komma åt processer på andra noder utan att ha en direkt referens som i exemplet ovan. För att åstadkomma detta måste processen som ska nås ges ett namn som finns tillgängligt för den anropande processen. I detta fallet är namnet inskrivet i själva koden.

```
-module(dist).
-export([sender/2, adder/0]).

sender(Node,Num) ->
    Adder = {adder,Node},
    Adder ! {num,self(),Num},
    receive
    {num,NumPlusOne} ->
        io:format("Num plus one => ~w~n", [NumPlusOne]);
    Other ->
```

```
        io:format("Unexpected message: ~w~n", [Other])
    end.

adder() ->
    register(adder,self()),
    receive
    {num,Src,Num} ->
        Src ! {num,Num+1};
    Other ->
        Other
    end.
```

Mnesia

Modulen `database_handler` i Erlang-implementationen använder databasen Mnesia som är *embedded*, alltså körs i samma virtuella maskin som processerna som kommunicerar med den. Mnesia har inte stöd för SQL men erbjuder ett mycket begränsat alternativ kallat QLC[94]. Data lagras som tupel-värden där någon del i tupeln defineras som nyckeln med vilken hela strukturen sedan kan hämtas[95]. Åtkomst till databasen sker främst genom att funktioner definieras utan att anropas direkt. Istället ges de som argument till en funktion `mnesia:transaction(f)` varpå funktionen exekveras som en atomär transaktion[96].

E Generisk server

För att demonstrera en generisk server har en naiv sådan implementerats (`naive_gen_server`). Följande exempel skulle troligtvis inte vara användbart i ett verkligt system utan är endast till för att visa hur en generisk server kan fungera. Vi kommer kort följa utvecklingen av en callback-modul och till sist presentera behaviour-modulen.

För att starta servern kommer i detta fall följande två funktioner att behövas.

```
start_link() ->
    naive_gen_server:start_link(?MODULE, undefined).

init(State) ->
    State.
```

Två funktioner har byggts för att kunna skicka meddelanden till server.

```
set_msg(Pid, Msg) ->
    naive_gen_server:call(Pid, {set, Msg}).

get_msg(Pid) ->
    naive_gen_server:call(Pid, {get}).
```

För att servern som bygger på det generiska gränsnittet ska kunna matcha på ovanstående mönster måste, i detta fall, motsvarande `handle_call/4` funktioner byggas.

```
handle_call({set, Msg}, From, Ref, _State) ->
    naive_gen_server:reply(From, Ref, {ok, Msg}),
    Msg;
handle_call({get}, From, Ref, State) ->
    naive_gen_server:reply(From, Ref, State),
    undefined.
```

Genom att anropa `set_msg/2` eller `get_msg/1` kommer ett meddelande skickas via `naive_gen_server`. Detta meddelande kommer i sin tur att behandlas och sedan vidarebefordras det till `handle_call/4`-klausul. Beroende på vilket mönster som matchas kommer det att behandlas på olika sätt.

Som synes är det enkelt att implementera den funktionalitet som behövs, givet att gränsnittet är väl dokumenterat. Detta kan göras trots att den underliggande strukturen är dold för utvecklaren.

Behaviour-modulen

Det är även enkelt att följa kedjan för anropen som görs. När `set_get_server:start_link/2` anropas skickas ett meddelande till:

```
naive_gen_server:start_link(Module, InitState) ->
    spawn_link(fun() -> init(Module, InitState) end).
```

Här anropas i sin tur `naive_gen_server:init/2` som sätter starttillståndet för servern via callback-modulen. `naive_gen_server:init/2` startar även en `receive`-loop som väntar på inkommande meddelanden.

```
loop(Module, State) ->
    receive
        {call, Pid, Ref, Msg} ->
            loop(Module, Module:handle_call(Msg, Pid, Ref, State))
    end.
```

När sedan en klient gör ett anrop via `set_get_server` kommer `naive_gen_server:call/4` att anropas.

```
call(Pid,Msg) ->
    Ref = make_ref(),
    Pid ! {call, self(), Ref, Msg},
    receive
        {Ref, Reply} ->
            Reply
    after 2000 ->
            timeout
    end.
```

`naive_gen_server:call/4` har i uppgift att distribuera vidare meddelandet till rätt process. Detta meddelande kommer att hanteras i `naive_gen_server:loop/2` där `set_get_server:handle_call/4` kommer att behandla meddelandet om det matchas. Returvärdet från `set_get_server:handle_call/4` kommer att bli serverns nya tillstånd i detta fall.

`naive_gen_server` är en extremt förenklad variant av ett Erlang/OTPs standard-behaviour `gen_server` som är ett vältestat och robust behaviour som bygger på en klient/server-modell.

Exempelkörning

```
Eshell V5.8.5  (abort with ^G)
1> c(naive_gen_server).
{ok,naive_gen_server}
2> c(set_get_server).
{ok,set_get_server}
3> Pid = set_get_server:start_link().
<0.44.0>
4> set_get_server:set_msg(Pid, {new_state}).
{ok,{new_state}}
5> set_get_server:get_msg(Pid).
{new_state}
```

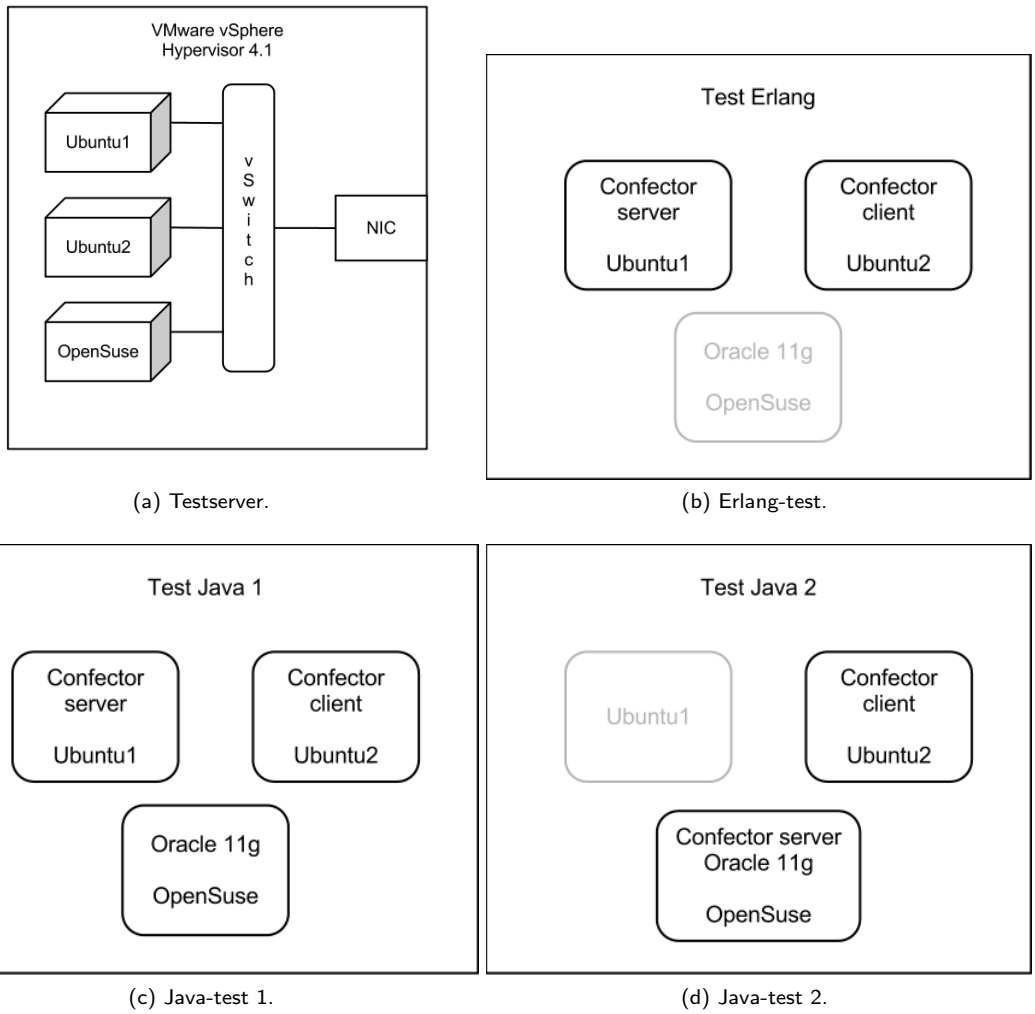
F Detaljer om mätning av prestanda

Infrastruktur

Alla tester har genomförts på en server med en Intel Quad Core processor och åtta GB RAM. Med hjälp av VMware vSphere Hypervisor 4.1 har sedan tre virtuella maskiner skapats, var och en med en GB RAM och en processorkärna. Två av de virtuella maskinerna använde Ubuntu Server 11.10 x86_64 medan den tredje använde OpenSuse 12.1 x86_64. Dessa virtuella maskiner kommer hädanefter kallas Ubuntu1, Ubuntu2 och OpenSuse (figur F.1a). Ingen av dem använde ett grafiskt gränssnitt. Nätverkskommunikation mellan de tre maskinerna sköts internt av vSphere Hypervisor's vSwitch[97].

Konfiguration

Erlang-testerna genomfördes med server-delen på Ubuntu1 och klientdelen på Ubuntu2 (figur F.1b). Eftersom Java-implementationen använder en separat databas genomfördes två tester, en med server-delen på Ubuntu1, klientdelen på Ubuntu2 och databasen på OpenSuse kallad Java1 (figur F.1c), en annan med server-delen på OpenSuse tillsammans med databasen kallad Java2 (figur F.1d).



(a) Testserver.

(b) Erlang-test.

(c) Java-test 1.

(d) Java-test 2.

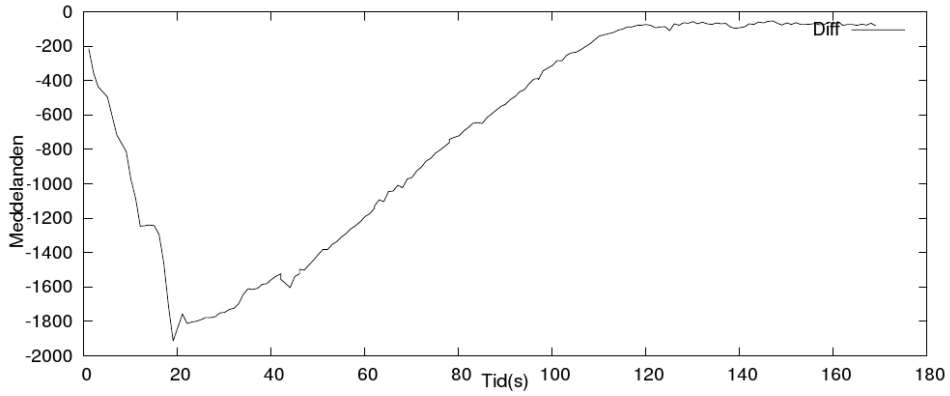
Figur F.1: Olika konfigurationer för tester.

För att förenkla testningen modifierades de två implementationerna. För låga belastningar fungerade en enkel algoritm där K kommandon lades in på servern följt av en sekunds väntat och upprepning. Vid högre belastningar däremot måste väntetiden mellan införandet av K kommandon anpassas eftersom servern behöver en viss tid för att registrera alla kommandon. Därför gavs algoritmen en enkel regulator som anpassade väntetiden T (i millisekunder) mellan införandet av kommandon till servern efter hur många svar M servern fått från C klienter efter S sekunder:

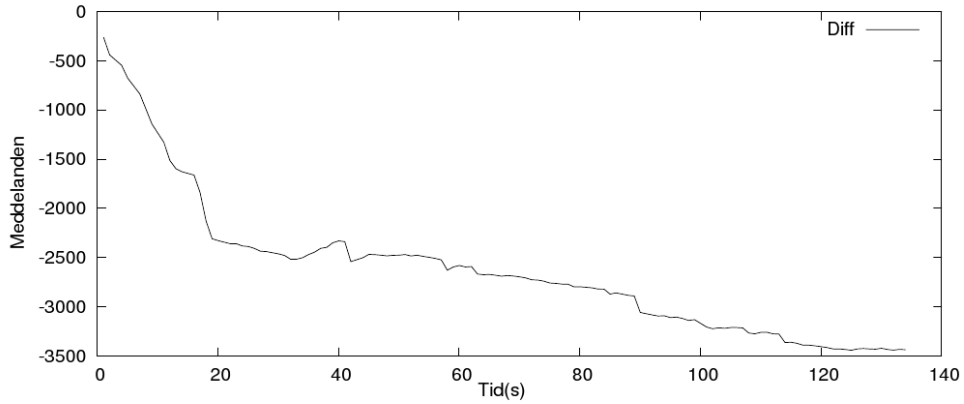
$$T = 500 + 0.5 \cdot (M - C \cdot K \cdot S)$$

Varje kombination av C klienter och K kommandon per sekund gav en dataserie bestående av antalet sekunder sedan testets början och skillnaden mellan det faktiska och det förväntade antalet mottagna meddelanden. I graferna nedan motsvarar alltså x-axeln tiden sedan testets början. Positiva värden på y-axeln betyder att för många meddelanden tagits emot (på grund av att kommandon har tillförts alltför fort) och negativa värden att för få meddelanden tagits emot. I idealfallet är grafen en rak linje utmed x-axeln men en viss variation är rimlig. Graferna används enbart för att visa hur de olika implementationerna fungerat under en belastning som de klarat av respektive inte klarat av och ger ingen ytterligare information om faktisk prestanda.

Java-implementationen kunde bedömas tämligen objektivt eftersom det antingen avvek mer och mer från det förväntade antalet mottagna meddelanden (figur F.3) eller närmade sig det förväntade antalet efter en inledande svacka (figur F.2). Figurerna F.2 till och med F.5 visar skillnaden mellan det förväntade antalet mottagna meddelanden och det faktiska antalet mottagna meddelandet under ett visst antal sekunder.

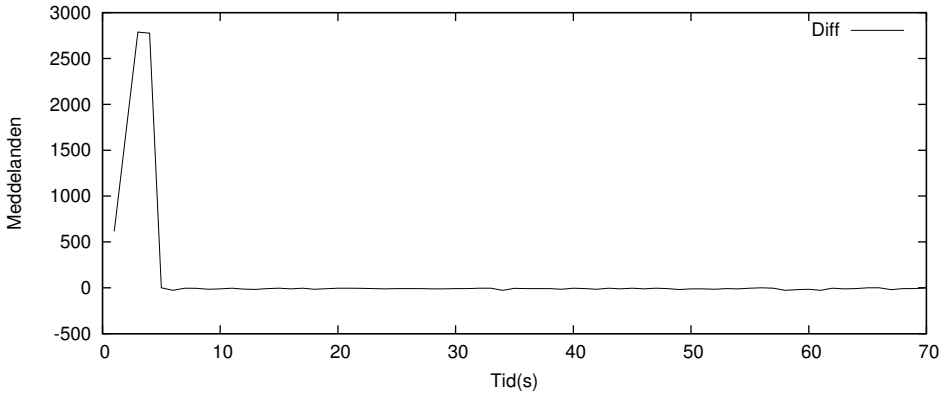


Figur F.2: Java1 med fungerande belastning.

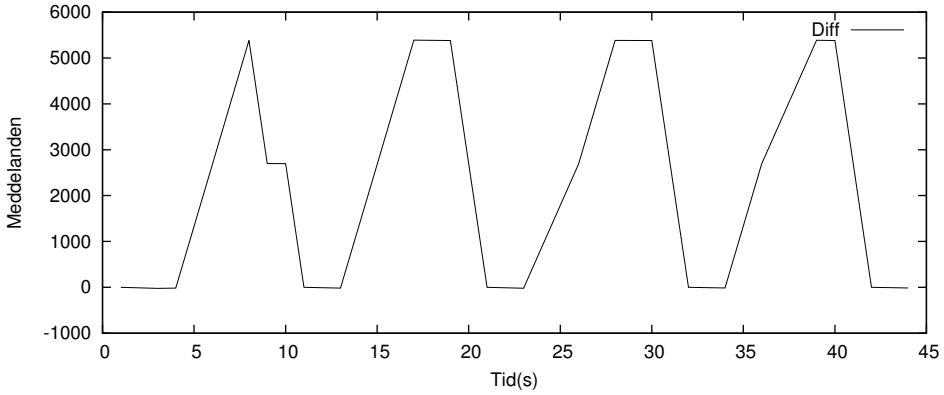


Figur F.3: Java1 med alltför stor belastning.

Erlang-implementationen däremot var inte lika enkel att bedöma eftersom den blev mer och mer oregelbunden ju högre belastningen blev. Därför ansågs den implementationen ha misslyckats att hantera en kombination C och K om den uppvisat "för stor" oregelbundenhet. Exempelvis kunde Erlang-implementationen efter ett inledande fel bete sig regelbundet för fyra klienter och 700 kommandon per sekund (figur F.4) men inte en klient och 2700 kommando per sekund (figur F.5)

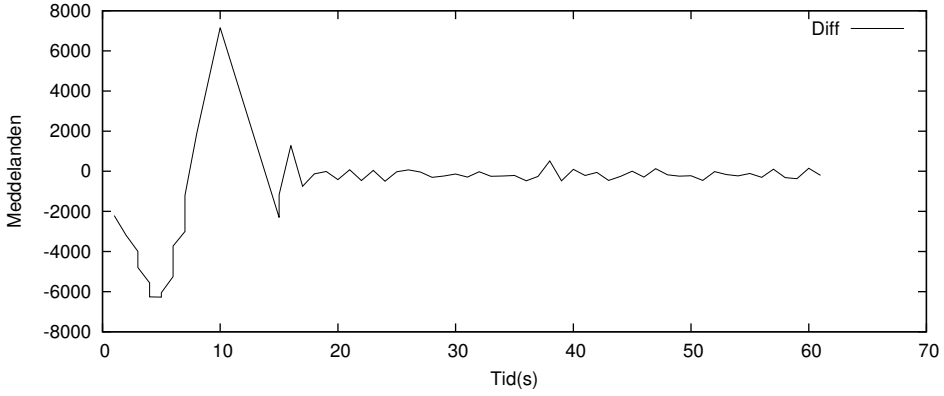


Figur F.4: Erlang1 med fungerande belastning.

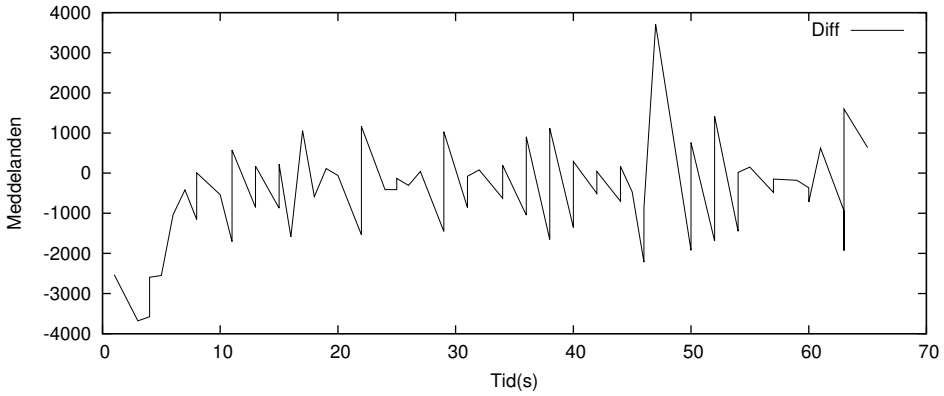


Figur F.5: Erlang1 med alltför stor belastning.

I det senare testet där både Java- och Erlang-implementationen utrustats med databaser som kördes i samma virtuella maskin som Confector-servern uppvisade Java-implementationen egenskaper liknande de som tidigare observerats i Erlang-implementationen. För en klient och 2500 kommandon per sekund som mål svänger Java-implementationen snabbt in mot 0 (figur F.6) medan den vid alltför stor belastning är instabil (figur F.7).



Figur F.6: Java3 med fungerande belastning.



Figur F.7: Java3 med alltför stor belastning.

G Uteslutna språk

C/C++ Även om C fortfarande är ett av de mest använda programmeringsspråken i världen[29] är det gammalt[98] med ett litet standardbibliotek[99] och inget inbyggt stöd för trådar. För mjukvara tätt knuten till hårdvara är C väl lämpat då det ger direkt tillgång till minne och andra resurser[100]. C++ är mer mångsidigt och har även stöd för objektorienterad programmering. Den direkta tillgången till arbetsminnet leder dock till problem med stabilitet och säkerhet som nämndes i 2.3.

C# Trots namnet är C# närmre besläktat med Java än C och C++ och har bland annat automatisk minneshantering och körs i en virtuell maskin[101]. Däremot skapades körmiljön för C# med Microsoft Windows i åtanke[102] och även om tredjepartsutvecklare erbjuder kompatibla körmiljöer för Linux, Solaris och Mac OS X[103] är det tydligt att Java har ett bättre stöd för flera plattformar.

Python Detta språk har automatisk minneshantering, ett stort standardbibliotek och stöd för flera plattformar[104]. Det erbjuder även flera programmeringsparadigm såsom objektorientering, funktionell och procedurell programmering[105]. Även om språket är mångsidigt är det enligt vår åsikt mer lämpat för skript och klientapplikationer än för server-bruk.