Handelshögskolan
VID GÖTEBORGS UNIVERSITET
Institutionen för informatik

Skalbar och flexibel systemarkitektur med Web-services och J2EE

- En undersökning inom det starkt föränderliga och dynamiska domännamnsregistreringsområdet

En systemarkitektur påverkas av dels inre och yttre aspekter och dels av miljön. Internetvärlden är ett exempel på en dynamisk och föränderlig miljö. Systemarkitekturen hos en organisation inom denna miljö måste vara flexibel och möjliggöra snabba förändringar. Den måste även vara skalbar, enkel att bygga ut och vara plattformsoberoende. Detta förespråkar även en viss form av centralisering. Vi har studerat en möjlig sådan systemarkitektur för ett domännamnsregistrerings-företag eftersom affärsområdet ligger inom ramen för den dynamiska miljö vi är intresserade av. Vår forskningsfråga blir därför: *"Hur kan användandet av plattformsoberoende tekniker möjliggöra en centraliserad, flexibel och skalbar systemarkitektur för en starkt föränderlig miljö?"* För att utreda om detta är möjligt har vi genomfört en *feasibility study*, som omfattar en iterativ process av litteraturstudier, intervjuer och funktionstester samt en fallstudie i form av en prototyp av en idealarkitektur. Våra resultat visar att plattformsoberoende tekniker som JBoss, J2EE och *Web-services* tillsammans möjliggör den önskade systemarkitekturen för en föränderlig och dynamisk miljö. Man kan ha klienter skrivna i olika programmeringsspråk som alla via *Web-services* kan kommunicera med en JBoss applikationsserver och erhålla centraliserad data eller centraliserade funktioner. Dessutom medger applikationsservrar byggda kring J2EE att nästan vilken funktionalitet som helst kan kapslas in i komponenter och refereras till enligt treskiktsprincipen vilket ger en mycket hög funktionell flexibilitet. JBoss ger i sig själv arkitekturen en inbyggd skalbarhet sett till prestanda och kapacitet och genom att de program som driftsätts i applikationsservern betraktas som självständiga komponenter ges även en skalbarhet i form av utbyggbarhet.

Nyckelord: *Web services, SOAP, J2EE, JBoss, arkitektur, domännamnsregistrering*

Författare: Martin Arnoldsson
Erik Lupander
Peter Jönsson

Handledare: Alan B. Carlson
Magisteruppsats, 20 poäng

Innehållsförteckning

1 INTRODUKTION 3

1.1 PROBLEMOMRÅDE 1
1.1.1 Domännamnsregistrering 1
1.1.2 House of ports befintliga systemarkitektur 2
1.2 FRÅGESTÄLLNING 3
1.3 AVGRÄNSNINGAR 4
1.4 DISPOSITION 4
1.5 ANVÄNDANDET AV REFERENSER 5

2 TEORI 6

2.1 ARKITEKTUR 6
2.1.1 Arkitektur - övergripande 6
2.1.2 Tvåskiktad arkitektur 7
2.1.3 Treskiktad arkitektur 8
2.1.4 Model View Controller (MVC) 8
2.2 DOMÄNNAMNSREGISTRERINGSPROCESSEN 8
2.3 PROTOKOLL 9
2.3.1 Transportprotokoll över Internet 9
2.3.2 Web-services 11
2.3.3 XML 12
2.3.4 SOAP 13
2.3.5 Axis 13
2.3.6 UDDI 14
2.3.7 WSDL 14
2.3.8 Extensible Provisioning Protocol, EPP 14
2.4 J2EE 15
2.4.1 J2EE – en övergripande beskrivning 15
2.4.2 Java-böna 16
2.4.3 Enterprise Java Bean 16
2.4.4 EJB Container 16
2.4.5 Remote och Home interface 17
2.4.6 Entitetsböna 17
2.4.7 Sessionsböna 18
2.4.8 Message-driven bean 18
2.4.9 Java Management Extensions, JMX 19
2.4.10 MBean 20
2.5 APPLIKATIONSSERVER 20
2.5.1 JBOSS 21

3 METOD 23

3.1 FEASABILITY STUDY 23
3.2 LITTERATURSTUDIER 24
3.3 INTERVJUER 25
3.4 PROTOTYPING 25

4 EMPIRI 27

4.1 BESKRIVNING AV FALLFÖRETAGET 27
4.2 NUBILD 28
4.3 IDEALBILD 29
4.3.1 Egenskaper och kriterier 32
4.4 KRAV OCH KRITERIER 32
4.5 FALLSTUDIE, PROTOTYPBESKRIVNING 35
4.5.1 Komponentdesign 38
4.5.2 Konfigurationshantering i prototypen 42

4.5.3 Backup av konfigureringsdata med MBeans ... 43
4.5.4 Skriptbarhet direkt mot RegistryHandler-implementationer ... 44
4.5.5 Design av dataklasser ... 45
4.6 PROCEDUR ... 46
4.6.1 Testlista ... 47
4.7 RESULTAT: FUNKTIONSTESTER, TEKNIK- OCH PROTOTYPUTVÄRDERING ... 48
4.7.1 Överväganden kring val av teknisk arkitektur. ... 48
4.7.2 Test: JBoss och konfigurering ... 50
4.7.3 Test: Web-services ... 50
4.7.4 Test: JMX, Mbeans och konfigurering ... 54
4.7.5 Test: Dynamisk klassladdning ... 55
4.7.6 Test: Övrigt ... 60
4.8 RESULTATANALYS: UTVÄRDERING AV PROTOTYPEN GENTEMOT KRITERIERNA ... 61

5 SLUTSATS ... 67

5.1 KONKLUSIONER ... 67
5.1.1 Plattformsoberoende tekniker ... 67
5.1.2 En starkt föränderlig miljö ... 67
5.1.3 Centraliserad systemarkitektur ... 68
5.1.4 Flexibel systemarkitektur ... 69
5.1.5 Skalbar systemarkitektur ... 69
5.1.6 Sammanfattning ... 69
5.2 DISKUSSION ... 69
5.2.1 Applikationsservrar, J2EE och JBoss - en brant inlärningskurva? ... 70
5.2.2 Databashanteringen i JBoss. ... 70
5.2.3 Sessionshantering: Apache Axis vs. Microsoft .NET ... 71
5.2.4 Web-services och säkerhet ... 71
5.3 METODUTVÄRDERING ... 71
5.4 VIDARE FORSKNING ... 72
5.5 PERSONLIGA VÄRDERINGAR ... 73

6 REFERENSER ... 75

BILAGA 1: UTTRYCK OCH FÖRKORTNINGAR ... 1

BILAGA 2: INTERVJUFRÅGOR ... 1

BILAGA 3: ÖVRIGA TEKNISKA ERFARENHETER ... 1

Figurförteckning

Figur 1 - OSI-modellen ... 10
Figur 2 - Vår *feasibility study* ... 23
Figur 3 - Nubild av registreringssystemet för EPP ... 29
Figur 4 - Basic system architecture, flow for EPP-based registration ... 36
Figur 5 - Architecture for direct access to EPP-registries using SOAP ... 37
Figur 6 - Switch architecture – Yttre gränssnitt, Switch.jar och configurator.sar ... 39
Figur 7 - RegistryHandler architecture ... 40
Figur 8 - EPP-implementationerna ... 41
Figur 9 - JMX-konsolens webbgränssnitt ... 43
Figur 10 - Klassdiagram över dataklasserna ... 45
Figur 11 - Detaljerat klassdiagram för *Host*-klasserna ... 46
Figur 12 – Vår testfamilj: Paret Olle och Hannis med sina två barn Per och Stina. ... 52
Figur 13 – Testresultat av MBean-prestanda ... 55
Figur 14 – Exempel på Enterprise-arkiv för olika toppdomäner, ”info” och ”biz” ... 56

Figur 15 – FOO_Info.ear med gemensamt gränssnitt..60

1 Introduktion

Valet av systemarkitektur – som inom informationsteknologi är en term som kan appliceras på processen och resultatet av att tänka ut och specificera den övergripande strukturen [1] – påverkas av både yttre och inre aspekter. De yttre aspekterna är sådant en organisation inte själv kan påverka men som sätter krav och begränsningar på ett system. Det kan vara allt från priset på en utvecklingsprodukt till vilket Internetprotokoll organisationen måste använda för kommunikation med extern part. Inre aspekter är sådana organisationen själv påverkar genom olika systemutvecklingsregler och policyn, t ex utvecklingsspråk, vilken arkitektur eller vilken plattform som skall användas.

Förutom de yttre och inre aspekterna så är det ur systemutvecklingssynpunkt viktigt att veta hur statisk eller dynamisk miljön är som man arbetar i eftersom miljön i stor utsträckning påverkar utvecklingen och valet av systemarkitektur [2].

En organisation som arbetar i en dynamisk och föränderlig miljö måste vara flexibel och kunna genomföra snabba förändringar utan att behöva stänga ner affärsverksamheten under tiden förändringarna genomförs för att inte förlora affärsintäkter.

Internetvärlden är ett exempel på en dynamisk och föränderlig miljö på så sätt att det hela tiden utvecklas nya och bättre versioner av de utvecklingsspråk, protokoll mm som en organisation använder sig av. Ofta skall man dessutom sömlöst kunna växla mellan olika system, plattformar och språk vilket ställer höga krav på den systemarkitektur en organisation har.

Vad som intresserar oss är alltså att utreda hur man i en dynamisk miljö utvecklar en flexibel, säker och skalbar arkitektur där man kan genomföra förändringar under körning. Det finns två olika vägar som man kan utreda hur man i en dynamisk miljö designar en flexibel, säker och skalbar arkitektur. Antingen studerar man det utan att ta hänsyn till tidigare systemutvecklingar och då utvecklar som om man börjar från noll eller så tar man hänsyn till redan tidigare systemutvecklingar och skapar en ny arkitektur med hjälp av eller utifrån dem.

Vi har valt att använda oss av en organisations befintliga systemarkitektur och studera hur man på bästa sätt kan göra om en mer rigid systemarkitektur till en som blir mer flexibel, säker och skalbar.

En typ av organisation som lämpar sig mycket väl att studera då det gäller detta är en organisation vars affärsområde omfattar domännamnsregistrering eftersom organisationen i hög grad påverkas av yttre aspekter samtidigt som man verkar i Internetvärldens dynamiska miljö (detta förklaras längre fram i texten).

Ett företag som arbetar med domännamnsregistrering är House of Ports. Företaget startade i mindre skala och har sedan starten expanderat och utökat sin verksamhet. I dagsläget är företaget i behov av att förändra den existerande arkitekturen på grund av inre och yttre påtryckningar vilket gör att det passar bra in på vår studie. Det inre trycket bottnar i att man i dagsläget besitter en arkitektur som inte är tillräckligt flexibel och dynamisk för att optimalt möta en vidare utvidgning av företagets verksamhet. Det yttre trycket kommer utifrån den miljö som man verkar i.

För oss handlar det alltså om två olika problemområden som tillsammans skall behandlas. Det gäller nämligen att kunna förena den befintliga (inre aspekter) systemarkitekturen med en ny som i alla avseenden klarar av de yttre påverkningar som finns i den dynamiska och föränderliga miljö organisationen verkar inom.

De yttre påverkningar och den dynamiska miljön behandlas i problemområdet *domännamnsregistrering* och de inre påverkningarna behandlas i problemområdet *House of Ports befintliga systemarkitektur*.

Det är väl värt att nämna att det i uppsatsen används flera engelska uttryck som ofta tappar en del av sin betydelse när de översätts till svenska. Därför har vi sammanställt en lista på de ord, förkortningar och uttryck som används i uppsatsen. Denna lista finns bifogad som Bilaga 1.

1.1 Problemområde

1.1.1 Domännamnsregistrering

För att en kund skall kunna registrera ett domännamn så måste man göra detta via ett ombud (*registrar*) godkänt av den organisation som styr domänen (*registry*). Ett av de stora problemen för en *registrar* att registrera ett domännamn hos ett *registry* är kommunikationen dem emellan. Kommunikationen sköts nämligen på olika sätt hos olika *registries* och det är allt från telefonsamtal eller vanligt brev till e-post eller vissa specificerade Internetprotokoll. Om ett registreringsombud vill kunna registrera flera olika domäner måste denna alltså hålla reda på en mängd olika förfaranden vilket komplicerar den systemarkitektur de arbetar med.

De flesta toppdomäner håller dock på att övergå till att endast använda sig av det specifika Internetprotokollet Extensible Provisioning Protocol (EPP) som kommunikationsmedel [3] men detta skapar i sin tur andra svårigheter. Ett problem (samtidigt som det är en fördel ur andra synvinklar) är att EPP är ett *open-source*-protokoll. På grund av detta kan en mängd specifika toppdomänslösningar uppstå i en enskild version av protokollet eftersom det inte finns någon standardiserad version av EPP som alla toppdomäner använder sig av. Detta innebär att en *registrar* måste designa sitt system efter alla de olika versioner som de olika *registries* använder sig av. Ett annat problem är att systemet måste vara flexibelt nog för att man enkelt skall kunna konfigurera ett nytt förfarande för den toppdomän som byter kommunikationssätt utan att man nödvändigtvis måste stänga ner systemet och ändra i koden. Detta för att man inte skall förlora intäkter eller kunder under den tid som systemet är nere.

Förutom svårigheterna med kommunikationsförfarande och EPP så måste systemet kunna hantera införandet av nya toppdomäner och även här så skall detta ske på ett smärtfritt och enkelt sätt.

Systemarkitekturen måste alltså vara tillräckligt skalbar och ha flexibilitet nog för förändringar vad gäller olika versioner av EPP, nya kommunikationsförfaranden för en befintlig toppdomän och införandet av nya toppdomäner.

Dynamiken och föränderligheten i miljön med olika kommunikationsförfaranden, införandet av nya toppdomäner och olika EPP-implementationer är en sida av problemet

vid utvecklingen av en systemarkitektur inom området eftersom den dynamiska verkligheten har en mycket stor inverkan på arkitekturens utseende.

Den andra sidan av problemet uppstår då en *registrar* utökar sin verksamhet i denna dynamiska miljö. En utökning av verksamheten innebär i detta fall att man kan registrera fler toppdomäner vilket betyder att man måste kunna integrera flera sätt att registrera en domän i sitt system. För att integreringen skall fungera måste det existerande systemet ha en arkitektur som möjliggör detta.

I nästa avsnitt tittar vi närmare på en systemarkitektur som i vissa avseenden inte är tillräckligt skalbar eller flexibel för att optimalt verka i den dynamiska och föränderliga domänregistreringsmiljön och som därför behöver bearbetas.

1.1.2 House of ports befintliga systemarkitektur

House of Ports (Ports) är ett företag som arbetar med att registrera domännamn. Företaget började sitt arbete i liten skala och har på några år växt till att bli ledande inom domännamnsregistrering.

Under sin levnadstid har Ports haft flera olika systemutvecklare som varit experter inom olika delar av systemutveckling. Dessa har använt sig av olika programmeringsspråk vilket har lett till att det i takt med tiden utvecklats en mängd olika klientapplikationer i en mängd olika programmeringsspråk. Allt för att kunna hantera det dagliga arbetet och de olika sätt som toppdomänerna vill kommunicera på.

Eftersom de arbetar i en dynamisk miljö och är styrda av toppdomänernas sätt att kommunicera så innebär detta att när dessa ändrar kommunikationssätt så måste även företagets klientapplikationer ändras. I många fall måste detta ske snabbt vilket kan leda till diverse nödlösningar i befintlig kod, såkallade ”fulhack”.

Som arkitekturen ser ut idag så ligger det en Microsoft SQL Server (databas) i grunden mot vilken alla applikationer integreras. Att klientapplikationerna arbetar direkt mot databasen (2-skiktsarkitektur) är inte helt bra eftersom det är just dessa data – kundregister, domännamninformation för tunna domäner etc. – är affärskritiska. Just nu ligger dessa data publikt för alla delsystem, innehåller i allmänhet ingen datavalidering eller inkapsling vilket gör systemet sårbart sett ur 2-skiktsperspektivet. Därför är det önskvärt att få in ett lager mellan data och gränssnitt - att gå från en tvåskiktad arkitektur till en treskiktad.

Problemet inom företaget är alltså att man i dagsläget besitter en arkitektur som inte är optimal för den miljö som man opererar i. Affärskritisk data ligger oskyddad och affärslogiken i klienterna är oöverskådlig eftersom den är spridd över många olika applikationer. Affärslogiken är således svårare att felsöka och förbättra utan en genomgripande förändring av systemarkitekturen. Eftersom systemarkitekturen från början designades för dåtidens krav snarare än framtidens så innehåller arkitekturen ett antal mindre bra lösningar och det blir dyrare och svårare att underhålla och uppdatera systemet med tiden utan en genomgripande arkitekturförändring. I dagsläget är arkitekturen inte tillräckligt flexibel utan har en låg överskådlighet och låg skalbarhet.

Vi tror att detta problem är generellt för de flesta företag som börjar i mindre skala och sedan växer i takt med att arbetsbelastning och efterfrågan ökar. Med en tvåskiktad arkitektur går det åt en ansenlig mängd tid och pengar för att upprätthålla de resurser

som krävs för att underhålla mjukvara. Vid en förändring av ett förfarande så krävs förändringar i varje klientapplikation som berörs. Detta är i längden ohållbart och en lösning är då att utöka den tvåskiktade arkitekturen med ytterligare ett skikt och på så sätt centralisera hela funktionaliteten. En centraliserad logik och funktionalitet innebär att det blir enklare att underhålla och anpassa efter de behov som uppstår. Samtidigt som klienterna skiljs från logiken blir de mer flexibla och även enklare att underhålla.

För att möta framtidens behov behöver man därför utveckla en mer flexibel arkitektur som kan anpassas efter den dynamiska verklighet man verkar i. Målet är att införa en treskiktad arkitektur och centralisera mycket av den affärslogik som i dag finns utspridda på klientsidan och skapa ett system som är skalbart och flexibelt nog att ta hand om de problem som råder inom domännamnsregistrering.

1.2 Frågeställning

Inom informationsteknologi är en *legacy*-applikation eller *legacy*-data sådana som härstammar från språk, plattformar och tekniker tidigare än den nuvarande tekniken. De flesta organisationer som använder sig av datorer har *legacy*-applikationer och databaser som handhar kritiska affärsbehov. Vanligtvis är utmaningen att hålla *legacy*-applikationen uppe och körbar medan man konverterar till en ny mer effektiv kod som använder sig av den senaste tekniken och programmeringsförmågan [4].

De applikationer som Ports utvecklat klassas inte som *legacy*-applikationer eftersom de inte härstammar från språk, plattformar och tekniker tidigare än den nuvarande tekniken. Dock skulle man kunna säga att systemarkitekturen är en *legacy*-arkitektur eftersom den inte är tillräckligt flexibel eller skalbar, där mycket av logiken ligger på klienterna vilket nu skapar problem, och därför bör vidareutvecklas. Det är därför intressant att försöka minska risken att man skapar en systemarkitektur som i framtiden blir till ett trögt och resurskrävande *legacy*-system. För de allra flesta företag är det önskvärt att byta ut *legacy*-system och *legacy*-applikationer mot sådana som följer ett mer öppet eller standardiserat gränssnitt. Eftersom det då teoretiskt skall bli enklare att uppdatera applikationerna utan att helt behöva skriva om källkoden och som dessutom möjliggör för användande på olika plattformar och operativsystem [4].

Därför är det intressant att studera om det är möjligt att skapa en centraliserad, flexibel och skalbar systemarkitektur med endast plattformsoberoende tekniker. En systemarkitektur som är plattformsoberoende, både i sig självt och åtkomstmässigt.

Att domännamnsregistreringsområdet är intressant är på grund av det är så dynamiskt och föränderligt med alla problem vad gäller olika versioner av EPP, nya kommunikationsförfaranden för en befintlig toppdomän och införandet av nya toppdomäner.

Sammantaget dessa intressanta studieområden så kristalliseras då vår forskningsfråga till:

> *Hur kan användandet av plattformsoberoende tekniker möjliggöra en centraliserad, flexibel och skalbar systemarkitektur för en starkt föränderlig miljö?*

För att utreda om detta är möjligt kommer vi att genomföra en *feasibility study*. Via en iterativ process av litteraturstudier, intervjuer och funktionstester kommer att fastställa den verklighet som företaget verkar i, deras nubild, och den idealarkitektur de vill ha samt de krav och kriterier denna idealarkitektur måste uppfylla. Vi utför dels en mängd funktionstester och bygger dels en prototyp som kontrolleras gentemot de krav och kriterier vi fått fram, en fallstudie. På så sätt etableras ett teoretiskt och praktiskt fungerande ramverk för den planerade systemimplementationen.

Ur detta arbete kommer vi att dra slutsatser om hur en centraliserad, flexibel och skalbar systemarkitektur med hjälp av plattformsoberoende tekniker kan se ut och fungera inom det problemområde vi studerar.

1.3 Avgränsningar

Eftersom det rör sig om att studera endast plattformoberoende tekniker så har vi tillsammans med företaget kommit fram till att använda oss av ett Web-services-tänkande tillsammans med en JBoss applikationsserver som har stöd för Java 2 Enterprise Edition (J2EE). Eftersom vi inte hade någon budget att röra oss med så passar dessa tekniker oss mycket bra eftersom de är gratis. En avgränsning blev alltså att de tekniker vi använder oss av inte skall kosta något.

Tidsaspekten har också bidragit till att det inte har varit möjligt att utveckla en fullskallig prototyp av systemet. Vi har därför utvecklat mindre prototyper för att testa enskild funktionalitet som senare återanvänts vid utvecklandet av prototypen i fallstudien. Dock anser vi att detta tillvägagångssätt är fullgott för att kunna påvisa de resultat vi är ute efter.

1.4 Disposition

Vi har disponerat vår uppsats med att inleda varje kapitel med en förklaring om vad som skall tas upp samt en motivering om varför det tas med.

Kapitel 1 (Introduktion) innehåller en inledande bakgrunds- och problembeskrivning för att skapa förståelse om vad som kommer att behandlas. Beskrivningen mynnar ut i en övergripande forskningsfråga för att läsaren ytterligare skall förstå det övergripande problemet som skall studeras. Forskningsfrågan motiveras genom att forskningsområdet redovisas på vilket sätt det är viktigt och intressant, allt för att minimera tveksamheten om problemområdets eller forskningsfrågans berättigande.

Kapitel 2 (Teori) behandlar det teoretiska ramverk på vilken vi baserar våra fynd och resultat. Varje enskilt teoretisk avsnitt förklaras för att det skall vara enkelt att förstå dels vad teorin handlar om och dels varför vi har med det i uppsatsen.

Kapitel 3 (Metod) beskriver hur vi har gått tillväga då vi inhämtat kunskap för att kunna studera problemområdet. I kapitlet beskriver vi de olika former av kvalitativa forskningsmetoder vi använt oss av samt en modifierade version av en *feasibility study* som vi tagit fram. Vilken vi i samråd med vår handledare anser passar in på denna typ av arbete.

Kapitel 4 (Empiri) innehåller de fynd och resultat, kopplade till det teoretiska ramverket, som vi har erhållit utifrån de metoder vi använt oss av. Avsnittet innehåller både delresultat som vi använder oss i vår *feasibility study* och resultatet som kommer ur studien. Här får man en uppfattning om hur vi använt oss av de metoder vi specificerade i metodavsnittet, proceduren hur vi gått tillväga.

Kapitel 5 (Slutsatser) redovisar de slutsatser vi kommer fram till baserat på de erhållna resultaten. Till hjälp för att förstå resonemanget med slutsatserna förs en diskussion kring resultaten. Kapitlet innehåller också en utvärdering av de metoder som vi har använt oss av samt om det skulle ha varit möjligt att ha arbeta på ett annat sätt. Slutligen beskrivs den vidare forskning som kan bedrivas och våra personliga värderingar och reflektioner.

1.5 Användandet av referenser

Det sätt vi har valt att referera till källor tycker vi passar bäst eftersom de flesta av källorna är från Internet. Vi anser att det blir alldeles för plottrigt om man använder sig av källhänvisningar i den löpande texten eller av fotnoter.

2 Teori

Vår föränderliga miljö ligger i problematiken med domännamnsregistrering och för att förstå detta är generella kunskaper om olika transportprotokoll och hur dessa fungerar en förutsättning.

Då kraven från företagets sida var att vi skulle använda oss av ett *Web-services*-tänkande tillsammans med en JBoss applikationsserver som har stöd för Java 2 Enterprise Edition (J2EE) så är dessa teorier nödvändiga att studera.

Eftersom uppsatsen behandlar arkitektur och vad som kan klassas som en god arkitektur så behöver vi dessutom redogöra för olika typer av arkitekturer.

I detta avsnitt kommer vi därför att gå igenom följande begrepp:

- *Arkitektur* – där vi beskriver vad som menas med arkitektur samt teorier om olika typer av arkitekturer som tvåskiktad och treskiktad arkitektur samt ett exempel på en treskiktad arkitektur, *Model-View-Controller* (MVC).
- *Domännamnsregistreringsprocessen* – som ger en beskrivning om hur det går till när man registrerar en domän.
- *Protokoll* – som innehåller en allmän beskrivning om transportprotokoll över Internet men även en beskrivning om vad *Web-services* och Axis är och vilka protokoll som används till detta (t ex XML, SOAP, UDDI och WDSL). Dessutom förklaras det protokoll (EPP) som används vid registreringen av en domän.
- *J2EE* – som omfattar det vi använder oss av inom området: Java-böna, Enterprise Java Bean, EJB Container, *Remote* och *Home interface*, Entitetsböna, Sessionsböna, Message-driven Bean, Java Management eXtensions (JMX) och MBean.
- *Applikationsserver* – där vi förklarar generellt vad en applikationsserver är samt beskriver den applikationsserver (JBoss) som vi använder oss av i denna studie.

2.1 Arkitektur

För att förstå vad som menas med ordet *arkitektur* och dess övergripande betydelse inom vårt problemområde så ges en kort beskrivning av detta.

Företagets arkitektur, som den ser ut idag, följer något som kallas tvåskiktsarkitektur. Strävan är dock att införa ytterligare ett lager eller skikt till någon form av treskiktad arkitektur, t ex *Model-View-Controller* (MVC).

2.1.1 Arkitektur - övergripande

Ordet arkitektur betyder ”läran om samspelet mellan tekniska och konstnärliga faktorer vid byggande” [5]. Arkitekturen i ett program eller ett system beskriver strukturen i ett system på en övergripande nivå genom att visa hur systemet är organiserat.

Det första steget i designprocessen är designen av en arkitektur eftersom arkitekturen etablerar ett grundläggande ramverk för systemet. Arkitekturen fungerar ofta som en

utgångspunkt vid framtagningen av specifikationen för systemets olika delar. Eftersom arkitekturdesignen innehåller identifikation av systemets stora komponenter och identifieringen av kommunikationen mellan komponenterna [2].

Det finns en mängd olika modeller eller stilar som man kan använda när man utvecklar en arkitektur. Arkitekturen kan exempelvis vara datacentrerad, dataflödesbaserad, objektorienterad eller uppdelad i lager. Och eftersom arkitekturmodellerna är generella finns det ingen modell som helt täcker in ett problemområde utan det är designerns uppgift att finna den mest passande modellen och modifiera den så att den stämmer överens med kraven inom problemområdet [2].

En arkitekturmodell lägger ofta tyngden vid olika kritiska aspekter och påverkar snabbhet, säkerhet, distribution och underhåll i ett system. Och för att valet av arkitektur inte skall bli fel är det viktigt att systemutvecklaren är medveten om modellens styrkor och svagheter redan i design processen [2].

Arkitekturer i större system baseras sällan kring en modell utan olika delar i systemet kan använda sig av olika arkitekturmodeller och i vissa fall kan arkitekturen vara sammansatt av ett flertal olika arkitekturmodeller [2].

2.1.2 Tvåskiktad arkitektur

Den enklaste formen av klientserver-arkitektur kallas för tvålagers klientserver-arkitektur och i denna typ av arkitektur är en applikation organiserad som en server med en uppsättning av klienter. Den tvåskiktade arkitekturen kan anta två former. Den första formen kallas för tunna klient modellen vilket innebär att klienten enbart presenterar data medan servern ansvarar för datalagring osv. Den andra formen kallas för feta klient modellen. Här ansvarar servern enbart för data hanteringen medan mjukvaran i klienten implementerar applikationens logik och hanterar interaktionen med systemets användare [2].

I en tvåskiktad klientserver-arkitektur arbetar klienten direkt mot en resurs som exempelvis en databas [6]. Affärslogiken ligger inte som ett separat lager utan spänner över både server och klientdator och den huvudsakliga uppgiften för klienten är att upprätthålla ett gränssnitt så att användaren kan skicka förfrågningar till servern [7].

Tvålagersarkitekturen lämpar sig främst i mindre nätverk där miljön inte är för komplex och fördelarna med arkitekturen visar sig på ett flertal sätt. I arkitekturen finns det exempelvis ingen server till server kommunikation utan det är endast klienter som kan kommunicera. Arkitekturen innebär ofta ett litet nätverk vilket betyder att man inte behöver investera pengar i extra servrar och avancerade protokoll samt att tvålagersarkitekturen är lätt att administrera eftersom data oftast lagras centralt. Arkitekturen fungerar däremot inte lika bra i större och mer komplexa nätverk eftersom den blir svår att administrera och försvårar möjligheten att skala upp på ett smidigt sätt. Man kan exempelvis inte fördela arbetsbelastningen på flera olika servrar samtidigt som varje man måste konfigurera varje klient för varje existerande server [7]. En annan nackdel med den tvåskiktade arkitekturen är att väldigt mycket av logiken måst ligga i klienterna vilket i sin tur innebär att klienterna tenderar att bli väldigt stora [6].

2.1.3 Treskiktad arkitektur

Det finns ett antal problemet med den tvåskiktade arkitekturen. Med den tunna klientmodellen får man ofta problem med skalbarhet och prestanda eftersom den tunna klienten stjäl en hel del datorkraft från servern. Medan den feta klient modellen ofta leder administrationsproblem eftersom man exempelvis måste uppdatera och implementera ny mjukvara på samtliga klienter vilket kan bli mycket dyrt vad det gäller tid och pengar [2].

Genom att införa ytterligare ett skikt försöker man komma till rätta med problemen i den tvåskiktade arkitekturen. I den treskiktade arkitekturen gör man en uppdelning i tre olika funktionerna: presentation, affärsregler och data [2]. Uppdelningen innebär att man bryter ut logik som antingen ligger i klienten eller på servern och placerar den i ett eget lager. Logiken centreras därmed i ett eget lager vilket leder till att systemet blir lättare att underhålla och det blir enklare att byta ut enskilda komponenter [8].

2.1.4 Model View Controller (MVC)

MVC är en arkitektur som består av tre komponenter:

- En modellkomponent (*modell*) som innehåller en dynamisk modell av systemets problemområde
- En funktionskomponent (*controller*) som innehåller de faciliteter användarna behöver för att kunna uppdatera och använda modellkomponenten samt
- En gränssnittskomponent (*view*) som kopplar systemet till omgivningen på två sätt.

För det första inkluderar gränssnittet monitorer, utskrifter och andra faciliteter som låter användare aktivera systemfunktioner och för det andra är gränssnittet direkt förbundet med andra tekniska system [9].

MVC-tänkandet introducerades för första gången i Smalltalk-80's programmeringsmiljö och tanken var att modellkomponenten skulle inkapsla kärndata och funktionalitet och skiljas från inputbeteende och presentationen av output. Gränssnittskomponenten ska visa data från modellen för användaren och varje gränssnittskomponent skall ha en kontrollkomponent som styr förändringar av modellen i form av input från användaren. När man separerade modellen från gränssnitts- och funktionskomponenten tillåter man multipla vyer av en modell som kan förändra och uppvisa förändringar av modellen gjorda av andra vyer [10].

Målet med MVC-strukturen är alltså att dela upp en applikation i tre olika delar eller skikt och göra dessa så oberoende som möjligt från varandra. Tanken är att man skall få en bra struktur på applikationen samtidigt som man vill göra det enklare att använda vissa delar av applikationen till andra applikationer [11].

2.2 Domännamnsregistreringsprocessen

Som vi tidigare har nämnt tidigare finns det många varianter på hur en domän registreras beroende på vilket toppdomän den skall ligga under. I huvudsak finns det fyra huvudgrupper av tillvägagångssätt.

1. RRP – Ett äldre registreringsprotokoll som är på väg ut, men som ändå kommer att användas av många mindre toppdomäner [12].
2. EPP – Ett modernt XML-baserat registreringsprotokoll som nu fasas in på fler och fler toppdomäner [13].
3. E-post – En registrering lagras som ett fördefinierat e-postmeddelande som skickas till den organisation som administrerar den aktuella toppdomänen. Observera att olika toppdomäner kräver olika format och information.
4. Telefon – Försäljaren får helt enkelt ringa upp och registrera domänen muntligt.

I vår studie kommer vi enbart att fokusera oss på EPP och anledningen till det är att den del av arkitekturen som vår fallstudie baserar sig på enbart behandlar just EPP.

De övriga kommunikationssätten skulle i ett senare skede kunna inkluderas men inte i denna studie.

Registreringsprocessen i sig är tämligen enkel [3]:

1. Kontrollera att domännamnet är ledigt
2. Ange informationsuppgifter – domännamn, olika kontaktuppgifter (ägare, administrativ, teknisk och debitering) och primär och sekundär server
3. Registrera domän

Svårigheten ligger alltså inte i själva registreringsprocessen utan i att det finns 4 olika tillvägagångssätt som i sin tur kan delas upp ytterligare. Exempelvis så används inte ett standardiserat e-postformulär för alla toppdomäner som använder sig av detta kommunikationssätt utan det finns olika varianter.

2.3 Protokoll

Detta avsnitt innehåller beskrivningar om olika protokoll som XML, SOAP, UDDI, WDSL och EPP samt en generell beskrivning om transportprotokoll över Internet. Vi kommer inte att använda oss av UDDI eftersom vi inte kommer att publicera några tjänster publikt över Internet men har ändå med det i teoriavsnittet eftersom det är en så väsentlig del av *Web-services*.

Vi förklarar även vad *Web-services* och Axis är, även om dessa i sig inte är protokoll utan istället använder sig av (bland annat) ovan nämnda protokoll.

2.3.1 Transportprotokoll över Internet

Ord som TCP/IP, HTTP, FTP, SMTP med flera förekommer ofta i den litteratur systemvetare studerar och är därför ganska självklara. Här följer endast en kort beskrivning av transportprotokoll över Internet för att försöka ge en enkel förklaring till hur tillämpningsprotokollen fungerar och hur de är relaterade till vår magisteruppsats.

Figur 1 - OSI-modellen

För att få en mer heltäckande bild av nätverkskommunikation rekommenderar vi att läsaren undersöker OSI-modellen och dess sju nätverkslager [14]. Kort går den ut på hur man går från materialet datasignalerna fortplantas i till de slutanvändarapplikationer som utnyttjar tekniken. Någonstans mittemellan här hittar vi TCP/IP som i mångt och mycket är grunden för Internet. Data som skall skickas över nätverket paketeras i paket om ett antal byte, förses med ett pakethuvud som talar om vart informationen ska och vart den kommer ifrån. Till sist beräknas en checksumma som lagras. När paketet till slut når sin destination kollar mottagarservern att den checksumma som paketet anger är densamma som den själv räknar fram. Om allting stämmer så skickar mottagarservern ett kvitto till avsändaren på att paketet mottagits korrekt. På så vis valideras all data. Om avsändaren inte får kvitto inom en viss tidsperiod så skickar den helt enkelt om paketet. Detta är grundreglerna, sedan finns det en del varianter på detta tema.

Ett lager ovanför TCP i OSI-modellen ligger de välkända transportprotokollen HTTP och FTP. Dessa protokoll fungerar genom att skicka textsträngar mellan TCP-sockets, antingen i klartext eller krypterad. Klienten initierar kommunikationen genom att skicka det definierade ”hello”-meddelandet. Servern svarar med en viss textsträng som även den är definierad i protokollet. Sedan fortsätter kommunikationen enligt detta mönster med textsträngar som protokollet ifråga definierar. Vill man botanisera i protokolldjungeln kan man studera såkallade *Request For Comments* (RFC)[1] som beskriver olika Internetstandarder.

Generellt kan man säga att ju högre upp i OSI-modellen man kommer, desto mer tekniskt relaterade frågor kan man abstrahera bort.

För att anknyta till den teknik vi utnyttjar i vår studie så kan man basera andra protokoll på t ex HTTP eller liknande. SOAP är ett ypperligt exempel på detta eftersom det kan använda sig av olika textbaserade protokoll såsom HTTP eller SMTP för sin kommunikation.

Ett annat exempel på ett protokoll som knyter an till vår forskning är *Extensible Provisioning Protocol* (EPP) som är ett protokoll som används vid kommunikation mot en del toppdomännamnsservrar. Detta protokoll förklaras i ett separat avsnitt men kort är att det inget finns något specifikt transportprotokoll eller någon explicit

[1] För ytterligare information om RFC, se t ex http://www.rfc.net

säkerhetsspecifikation inkluderad i protokollstandarden, utan EPP designades för att fungera i miljöer med olika protokollager [13].

2.3.2 Web-services

Även om *Web-services* är ett relativt nytt begrepp så bygger det på gamla, redan etablerade trender och förfaranden. Precis som så mycket inom systemtänkandet handlar det om att bryta ner stora komplexa system i mindre mer lätthanterliga delar och att kunna återanvända saker redan gjorda.

I en objektorienterad systemutveckling fokuserar man på objekt och komponenter som bland annat kapslar in funktioner i sig själva och som samverkar med andra objekt och komponenter på ett standardiserat sätt. I och med detta har man skaffat sig en mängd specialiserade och tydligt definierade komponenter, såsom till exempel en utskriftsfunktion, som då kan användas av en mängd olika system eller applikationer och som kan utvecklas ytterligare utan att andra behöver påverkas.

Vad *Web-services* handlar om i detta sammanhang är då att man skall kunna återanvända redan utvecklade rutiner och komponenter och samtidigt få tillgång till olika tjänster via Internet.

Genom *Web-services* kan företag kapsla in existerande företagsprocesser och via Internet publicera dem som tjänster (*services*), leta efter och prenumerera på andra tjänster och utbyta information genom och bortom företagets verksamhet [15]. Detta innebär alltså att man inte bara återanvänder sina egna gjorda investeringar utan även att man får tillgång till andras [16].

Enligt Castro-Leon [17] så krävs följande infrastruktur för att tillåta *Web-services* att kommunicera:

- *Internet* - ett fysiskt medium inom vilken kommunikation äger rum.
- *Universal Description, Discovery och Integration (UDDI)* – de olika beståndsdelarna i en Web-service innehåller information om sig själva och UDDI är ett tillvägagångssätt för olika tjänster att hitta varandra på Internet.
- *Extensible Markup Language (XML)* – Ett allmänt språk som kommunikationsprogram kan förstå.
- *Simple Object Access Protocol (SOAP)* – då Internet är det medium vari utbyte sker så behövs ett protokoll för att detta utbyte kan ske.

På grund av att all kommunikation sker med XML så är *Web-services* inte bundet till ett specifikt operativsystem eller programmeringsspråk - Java kan prata med Perl; Windows-applikationer kan prata med Unix-applikationer [18].

Nyckelfördelarna med *Web-services* är enligt WebServices.Org [15]:

- Mjukvara som en tjänst (*service*)
  Till skillnad mot paketerade produkter kan *Web-services* levereras and betalas för som en ström av tjänster and tillåter ständig tillträde från och till vilken plattform som helst. *Web-services* tillåter inkapsling och komponenter kan isoleras så att endast företagsdelens tjänster är exponerade. Detta resulterar i särkoppling mellan komponenter och mer stabila och flexibla system.

- Dynamisk företagsanvändbarhet
  Nya företagspartnerskap kan bildas dynamiskt och automatiskt eftersom Web services möjliggör ett komplett interagerande mellan olika system.
- ”Åtkomstbarhet” (Accessibility)
  Företagstjänster kan vara helt decentraliserade och distribuerade över Internet och kan kommas åt via en mängd olika kommunikationsenheter.
- Effektivitet
  Företagsverksamheten kan befrias från bördan av en komplex, långsam och dyr mjukvaruutveckling och istället fokusera på ”värdeberikning” och företagskritiska uppgifter. Web-services konstruerade från applikationer som designats för internt bruk kan enkelt exponeras för externt bruk utan att förändra koden.
- Universellt överenskomna specifikationer
  Web services är baserade på en universella överenskomna specifikationer för strukturerat datautbyte, meddelandehantering, upptäckten av tjänster och gränssnittsbeskrivning.
- Samverkan med ”utdaterade system” (*legacy systems*)
  Större rörlighet och flexibilitet från ökad samverkan mellan ”utdaterade system”.
- Nya marknadstillfällen
  Det kommer att finnas fler möjligheter för dynamiska affärsverksamheter att hitta nya marknader.

2.3.3 XML

I viss litteratur anges att XML står för *eXtended Markup Language* men vi följer definitionen från World Wide Web Consortium där XML står för *eXtensible Markup Language* [19].

XML kan beskrivas som ett generellt och standardiserat sätt att strukturera information och eftersom XML inte är bundet till någon viss typ av problem eller något visst applikationsområde så kan det användas i väldigt många sammanhang [20].

XML är ett enkelt och flexibelt textformat som kommer ur och är en begränsad form av *Standard Generalised Markup Language* (SGML), en ISO-standard (ISO 8879) från 1986 som även HTML bygger på [20].

XML är en upprensning av SGML som tillhandahåller en syntax för att representera innehållet i ett dokument. I XML tog man det bästa från SGML och kastade bort det folk inte använt i praktiken och där man samtidigt gör vissa förändringar för att göra det mer webbanpassat [20].

Ett XML-dokument består av ett eller flera element där varje element kan innehålla text eller andra element. Men då XML endast är en syntax behövs även en samling regler för hur olika typer av data skall beskrivas, vilka element som får förekomma i ett visst XML-dokument och vilka attribut som får associeras med dessa element. Denna samling regler kallas för ett XML-scheman [21].

Exempel på XML-syntax:

```
<NAMN>
   Detta element innehåller text och två element.
   <FÖRNAMN>Kalle</FÖRNAMN>
   <EFTERNAMN>Anka</EFTERNAMN>
</NAMN>
```

2.3.4 SOAP

Simple Object Access Protocol (SOAP) är ett XML-baserat protokoll för informationsutbyte i en decentraliserad och distribuerad miljö. SOAP definierar inte själv någon applikationssemantik, såsom programmeringsmodell eller implementationsspecifik semantik, utan definierar istället en enkel mekanism för att uttrycka applikationssemantik genom att tillhandahålla en modulär paketeringsmodell och kodningsmekanismer för kodning av data inom moduler. Detta tillåter SOAP att användas i en mängd olika system, allt från meddelande system till RPC [22].

SOAP består av tre olika delar [22]:

- Ett SOAP-kuvert (*envelope*) vars konstruktion definierar ett övergripande ramverk för att uttrycka vad som finns i ett meddelande, vem som skall handskas med det, och såtillvida det är frivilligt eller obligatoriskt.
- En uppsättning SOAP-kodningsregler som definierar en serialisationsmekanism som kan användas för utbyte av instanser av applikationsdefinierade datatyper.
- En SOAP RPC-representation som definierar en konvention som kan användas för att representera fjärranrop och svar.

Även om SOAP kan användas i en mängd olika typer av meddelandesystem och kan levereras via en mängd olika transporteringsprotokoll så är den största fokuseringen för SOAP *Remote Procedure Calls* (RPC) transporterade via HTTP. Precis som XML-RPC så är SOAP plattformsoberoende och möjliggör därför att en mängd olika applikationer kan kommunicera med varandra [18].

2.3.5 Axis

Axis står för *Apache eXtensible Interaction System* och kort sagt så är Axis en implementation av SOAP och i grund och botten en SOAP-motor – ett ramverk för att konstruera SOAP-processer såsom klienter, servrar, gateways, etc. [23]. Men Axis är inte bara en SOAP-motor utan inkluderar bland annat även en enkel fristående server, en server som kopplar in sig på servlet-motorer som till exempel Tomcat, omfattande stöd för WSDL och hjälpmedel för att övervaka TCP/IP-paket [23].

Axis är ett *open-source*-projekt och är för närvarande tredje generationen av Apache SOAP (som började på IBM som "SOAP4J") och levererar följande nyckelegenskaper jämfört med tidigare [23]:

- Snabbhet
- Flexibilitet
- Stabilitet
- Komponentorienterad *deployment*
- Transportramverk

- *Web Services Description Language* (WDSL)-stöd

2.3.6 UDDI

Universal Description, Discovery, och Integration (UDDI) representerar för närvarande lagret för att hitta och publicera tjänster (*services*) i en *Web-services*-protokollstack och bygger på olika två delar [18]:

- För det första så är UDDI en teknisk specifikation för att bygga ett distribuerat bibliotek av affärstjänster och *Web-services*. Data lagras i ett specifikt XML-format och UDDI-specifikationen inkluderar API-detaljer för att leta efter existerande data och publicera ny data.
- För det andra så är UDDI *Business Registry* en fullständig implementering av UDDI:s specifikationer. Detta register möjliggör för vem som helst att leta efter existerande UDDI-data samtidigt som företag kan registrera sig själva och deras tjänster (*services*).

2.3.7 WSDL

Web Services Description Language (WSDL) representerar för närvarande lagret för beskrivning av tjänster i en *Web-services*-protokollstack och i ett nötskal så är WSDL en XML-grammatik för specificerande av ett publikt gränssnitt för en Web-service, ett gränssnitt som kan innehålla följande [18]:

- Information om alla publika funktioner.
- Datatypsinformation för alla XML-meddelanden.
- Bindningsinformation om den specifika transporteringsprotokoll som skall användas.
- Adressinformation för att kunna lokalisera en specificerad tjänst (*service*).

2.3.8 Extensible Provisioning Protocol, EPP

Extensible Provisioning Protocol (EPP) är ett protokoll som används vid kommunikation mot en del toppdomännamnsservrar och är definierat i XML vilket innebär att det kan användas fungera på olika plattformar och i olika miljöer [13].

EPP uppfyller fullt ut kraven för domänregistreringar som beskrivs i dokumentet *Generic Registry-Registrar Protocol Requirements* [13]. Detta dokument beskriver och fokuserar i sin tur på de basala funktioner och gränssnitt som krävs av ett protokoll för att stödja multipla *registry*- och *registrar*-modeller [24].

Det finns inte något specifikt transportprotokoll eller någon explicit säkerhetsspecifikation inkluderad i protokollstandarden, utan EPP designades för att fungera i miljöer med olika protokollager [13].

EPP-tillämpningen i den kontext vi rör oss, dvs. domännamnsregistrering, är *socket*-baserad men använder sig av såkallade Secure Sockets Layer (SSL)-*sockets*[2] istället för

[2] Standardprotokoll för säker kommunikation över Internet, se t ex www.rsasecurity.com/standards/ssl/qa.html

vanliga TCP-*sockets* [3]. Skillnaden dessa transportprotokoll emellan är att SSL är krypterat så känslig kund- eller domäninformation inte skall hamna i orätta händer.

En grundläggande säkerhet finns alltså i de transportprotokoll som kommunikationen baseras på. Kommunikation går till så att parterna utbyter identifikation, autentisering och inställningsdata och därefter sker kommunikationen genom traditionella kommandosvar. Alla EPP-kommandon är atomära, dvs. att svaren alltid är helt framgångsrika eller helt misslyckade - några mellanting finns inte. Att exekvera ett kommando flera gånger i rad har samma effekt som att göra det enbart en gång [13].

EPP tillhandahåller fyra grundläggande tjänster: Publicering av tjänster, anrop, svar och ett ramverk som stödjer definition av objekt och relationen mellan protokollanrop och svar till dessa objekt [13].

På gott och ont så är EPP är ett *open-source*-projekt där olika aktörer använder olika, ofta icke-kompatibla versioner. Idag används fyra olika versioner i domännamnsregistreringsområdet [3].

2.4 J2EE

Detta avsnitt beskriver de teorier inom området kring J2EE som vi använder oss av i vår studie och omfattar: Java-böna, Enterprise Java Bean, EJB Container, *Remote* och *Home interface*, Entitetsböna, Sessionsböna, Message-driven Bean, Java Management eXtensions (JMX) och MBean.

2.4.1 J2EE – en övergripande beskrivning

Java 2 Enterprise Edition (J2EE) är en plattform som man kan utveckla komplexa applikationer på. J2EE är en Java-version för servrar och EJB (Enterprise Java Beans) är specifikation för hur tillämpningar skall skrivas [25]. EJB bygger på komponentstandarden *Java Beans* och syftet med EJB är främst att på ett säkert sätt kapsla in affärslogik på servrar [26]. Medan J2EE definierar en standard för att utveckla flerskikts affärsapplikationer. Tanken med J2EE är att förenkla företags applikationer genom att basera dem på standardiserade, modulära komponenter och erbjuda komponenterna en fullständig uppsättning av service. Samtidigt som man vill minska komplex programmering genom att hanterar många av applikationsdetaljer automatiskt [27].

En viktig del i J2EE är kompatibiliteten. Tillverkare och utvecklare som betalar licens för Java måste genomföra speciella tester av produkterna för att kunna vara kompatibla med J2EE [28].

I J2EE finns det fullt stöd för *Enterprise Java Bean* (EJB)-komponenter, *Java Servlet API*, *Java Server Pages* (JSP) och XML-teknologi. J2EE standarden inkluderar kompletta specifikations- och uppfyllelsekrav för att säkerställa portabilitet av applikationer över en mängd olika affärssystem som är kapabla att stödja J2EE [27].

2.4.2 Java-böna

Java-bönor (*JavaBeans*) är både ett paket i Java (java.beans) som erbjuder funktionalitet vid utvecklandet av bönor och ett dokument (*JavaBeans Specification*) som beskriver hur man använder klasser och gränssnitt i paketet java.beans [29].

En Java-böna är en återanvändningsbar mjukvarukomponent vilket innebär att arkitekturen inom *JavaBeans* är designad för att programmerare skall kunna bygga fristående mjukvarukomponenter som i slutändan skall kunna sättas ihop till större applikationskomponenter. Tanken med bönorna är att man skall utveckla applikationer som har enheter som är lätta att byta ut [30].

2.4.3 Enterprise Java Bean

Enterprise Java Bean (EJB) är ett objektorienterat ramverk för flerskiktade distribuerade system samt komponentintegration [31]. Dessutom är det en serverbaserad komponent som inkapslar en applikations affärslogik [32]. Tanken med EJB är att skapa återanvändbara mjukvarukomponenter i ett mellanskikt som förenar klienter och bakomvarande system (*backend*) som exempelvis plattformar och databaser [33].

EJB arkitekturen är uppbyggd av tre skikt [34]: presentationsskiktet, affärslogikskiktet och dataskiktet. Studerar man arkitekturen närmare så möter man ett flertal olika begrepp som t ex [35]:

- EJB-server – en applikationsserver som uppfyller J2EE specifikationen.
- EJB *container* – platsen där en *enterprise bean* exekveras.
- *Enterprise Java Bean* – som t ex entitetsböna, sessionsböna och m*essage-driven bean*.
- EJB Home – objekt för att hitta och skapa en EJB.
- EJB Objekt – klientens kontakt med affärsmetoderna.

Enligt Sun [32] kan man förenkla utvecklingen av stora, distribuerade applikationer på ett flertal sätt genom att använda *enterprise beans*. En *EJB container* erbjuder exempelvis service på systemnivå vilket innebär att en utvecklare kan koncentrera sig på att utvecklingen av affärsproblem medan *containern* ansvarar för service på lågnivå. Eftersom *enterprise beans*, och inte klienterna, innehåller en applikationens affärslogik behöver en klientutvecklare inte bry sig om applikationens bakomvarande logik utan kan fokusera på presentationen av klienten. Slutligen är *enterprise beans* portabla komponenter vilket innebär att applikationsutvecklaren kan konstruera nya applikationer från redan existerande bönor.

2.4.4 EJB Container

En EJB *container* är en vital del i EJB arkitekturen eftersom en *enterprise beans* inte fungerar utanför en *container*. En EJB *container* hanterar en *enterprise beans* på samma sätt som en *Java Web Server* hanterar en *servlet* eller som en webbläsare hanterar en *Java applet* [36]. En *container* är ansvarig för att skapa nya instanser av bönor och att se till så att dessa lagras korrekt av servern [31]. Vid exekveringen av en böna hanterar en *container* aspekter som fjäråtkomst av bönan, säkerhet, fortlevande, transaktioner, konkurrens, samt access till och samordning av resurser. Många av funktionerna sker

automatiskt vilket innebär att utvecklaren kan fokusera på inkapslingen av affärslogiken medan en *container* hanterar den övriga logiken [36].

Det som sker vid ett metodanrop från en klient är att en *container* isolerar *enterprise beans* från direktaccess av klientapplikationer och *container* kontrollerar sedan att anropet sker på ett korrekt sätt.

2.4.5 Remote och Home interface

En bönas *home interface* representerar en bönas livscykel medan ett *remote interface* representerar de affärsmetoder som bönan innehåller [36].

När en klient skall använda en böna fungerar ett *home interface* som en portal ute i världen. Det första som sker när klienten anropar bönan är att det skapas en instans av EJB-klassens *home interface*. Detta *home interface* innehåller metoder som gör det möjligt för klienten att få tillgång till en instans av EJB-klassens *remote interface*. *Remote interface*, som också kallas för EJB-objekt, innehåller bönans metoder och gör dessa synliga för klienten [34].

2.4.6 Entitetsböna

En entitetsböna (*entity bean*) representera ett affärsobjekt i ett system och det finns flera kännetecken för en entitetsböna [37]: En entitetsböna är beständig och lagras på någon typ av medium (ofta i en databas). En entitetsböna kan delas av multipla klienter och klienterna kan förändra samma data. Alla entitetsbönor har en primärnyckel. En entitetsböna kan ha ett förhållande till en annan entitetsböna.

Entitetsbönor erbjuder utvecklare en komponentmodell där han/hon kan fokusera på affärslogiken medan en *container* tar hand om fortlevandet, tillgångskontroll osv.

Det finns två typer av entitetsbönor: *Container Managed Persistance* (CMP) och *Bean Managed Persistance* (BMP). När man använder en CMP-entitetsböna hanteras bönans fortlevande av *containern* och man behöver inte skriva någon databas-*access*-kod i bönans klass för att bevara bönans entitetsfält i en databas. En BMP-böna innehåller däremot ingen databas-*access*-kod (JDBC) vilket innebär att bönan själv ansvarar för läsning och skrivning av bönans tillstånd till databasen. Detta innebär dock inte att utvecklaren av bönan är lämnad åt sitt eget öde eftersom utvecklaren får mycket hjälp av *containern* som talar om när det är dags att spara eller uppdatera bönan [36].

Det finns dock nackdelar i CMP:s transaktionshantering, och det är att den inte kan hantera åtkomsten på olika sätt beroende om det är en läs- eller skrivtransaktion [3].

Varför olika angreppssätt? I ett generellt fall så är de allra flesta databasanropen till för läsning av data som skall presenteras för en användare som inte skall ändra data. I ett sådant fall räcker det med att ta en ögonblicksbild av data ur den entitetsböna man vill visa. Om man vill skriva till entitetsbönan däremot, är det oftast viktigt att den är låst från det att den som vill skriva i den hämtat ut data, till att förändringarna är gjorda så det inte uppstår någon form av datainkonsistens.

Idag fungerar exempelvis JBoss 3.0.4 enligt det senare mönstret för alla typer av transaktioner [38]. Det mönstret kallas för *pessimistic record-locking* där databashanteraren utgår från att alla möjliga läsningar och skrivningar kommer att ha negativ inverkan på data [39]. Det andra angreppssättet är *optimistic record-locking*

[40]. Där försöker istället databashanteraren hantera olika läsningar och skrivningar i en modell som påminner om hur Concurrent Versions System (CVS)[3] hanterar olika versioner av källkod. CVS jämför ny data med gammal och kontrollerar om data som skall ändras har ändrats sen den aktuella transaktionen påbörjades. Först när det står definitivt klart att den önskade transaktionen inte går att utföra så kastar hanteraren ett felmeddelande.

Applikationsservern JBoss-3.0.4 hanterar transaktioner med entitetsbönor enligt *pessimistic record-locking* [38], vilket förvisso ger god säkerhet och datakvalitet, men samtidigt ger försämrad flexibilitet i dataåtkomstmodellen. Just nu (2003-03-11) diskuteras det som bäst i e-postlistan kring utvecklandet av JBoss 4.0 om JBoss transaktionsmodell helt enkelt skall skrivas om för att hantera denna problematik [83].

2.4.7 Sessionsböna

En sessionsböna (*session bean*) används för att hantera interaktionen mellan entitets- och andra sessionsbönor och för att få tillgång till resurser och utföra generella uppgifter från klienter. En sessionsböna är inte ett beständigt affärsobjekt som en entitetsböna. Vilket innebär att en sessionsböna inte representerar data på samma sätt i en databas. Det finns två typer av sessionsbönor: *stateless* och *stateful* [36].

En *stateless*-sessionsböna är uppbyggd av affärsmetoder som uppträder som procedurer vilket innebär att bönan bara arbetar med argumenten som skickas till den vid själva anropet. En *stateless*-sessionsböna är obestående och bevarar inte sitt affärstillstånd mellan metodanropen. Eftersom en *stateless*-sessionsböna är "tillståndslös" är den enklare att hantera för en EJB *container*. Bönan använder därmed mindre resurser och anropen av en *stateless*-sessionsböna går fortare [36].

En *stateful*-sessionsböna kapslar in en specifik klients tillstånd och affärslogik och bevarar affärstillståndet mellan metodanropen. En *stateful*-sessionsböna tillhör en specifik klient och bevarar information mellan metodanrop och till skillnad från en *stateless*-sessionsböna delar inte klienterna på *stateful*-sessionsbönor. När en klient skapar en *stateful*-sessionsböna utför bönans instans bara service till den klient som skapade den. Därmed blir det möjligt för bönan och klienten att upprätta ett konverserande tillstånd [36].

2.4.8 Message-driven bean

Message-driven bean (MDB) har utvecklats för att möjliggöra asynkrona applikationer [42]. Den stora skillnaden mellan en *message-driven bean* och sessions- och entitetsbönor är att en *message-driven bean* bara har en bönklass. Detta innebär att klienterna inte kan få tillgång till en *message-driven bean* genom ett interface som man får vid anropen av en entitets- eller sessionsböna. En *message-driven bean* påminner dock mycket om en *stateless*-sessionsböna:

- En instans av en *message-driven bean* bevarar inte data eller upprätthåller inte ett konversations stadium för en speciell klient.

[3] För ytterligare information om CVS se t ex http://www.cvshome.org/

- Alla instanser av en *message-driven bean* är likvärdiga och tillåter EJB containern att tilldela meddelanden till alla instanser av en *message-driven bean*.
- En enskild *message-driven bean* kan bearbeta meddelanden från multipla klienter.

2.4.9 Java Management Extensions, JMX

JMX är tidigare känt som *Java management API* (JMAPI) och är en Java-baserad arkitektur och relaterad service för applikations- och nätverkshantering [43].

Genom att använda JMX-specifikationen får utvecklare tillgång till en standard och kan bygga webbaserade, distribuerade, dynamiska och modulära lösningar för hantering av Java-baserade resurser. Dessutom gör det möjligt för en Java-utvecklare att integrera sina applikationer med existerande nätverkslösningar [44, 45].

JMX definierar en standard för att skriva JMX-objekt som kallas *Management Bean*, MBean. En MBean lever inuti en *container* som är definierad av standarden och *containern* gör det möjligt för en JMX-klient att anropa metoder och få åtkomst till attribut hos en MBean. Det är också möjligt för en klient att erhålla meddelanden från en MBean genom att registrera sig med en MBean [45].

JMX förenklar distribueringen av Java-applikationer eftersom en utvecklare kan konfigurera, hantera och övervaka Java-applikationerna under exekveringen och bryta ner applikationerna i komponenter som kan bytas ut. JMX erbjuder ett fönster där man kan se en applikations tillstånd och uppförande samt ett protokolloberoende sätt att ändra både tillstånd och beteende på en MBean [43].

JMX-arkitekturen är uppdelad i tre nivåer [46]:

1. Instrumentationsnivån (styr nivån) – Definierar hur man styr Java-resurser så att de kan administreras. Styrlagret exponerar applikationen som en eller flera administrationsbönor eller MBeans och varje MBean tillhandahåller tillgång till tillstånden genom publika metoder. En MBean kan vara vilket Java-objekt som helst som modifieras för att stödja gränssnitt och semantik specificerade i JMX-specifikationen för den typen av MBean som man skapar. Det finns fyra olika typer av MBean: *standard*, *dynamic*, *open* och *model* [47].
2. Agentnivån – Agenten i JMX-arkitekturen tillhandahåller fjärrdistribuerad applikationsåtkomst till alla registrerade MBeans. Man kan säga att agenten fungerar som en hub för JMX-ramverket. Agenten erbjuder samtidigt service, som dynamisk laddning och övervakning och kärnan i agentkomponenten kallas för MBean-server [47].
3. Distribuerade servicenivån (*distributed management services*) – tillhandahåller gränssnitt som implementerar JMX-distribuerare. Den här nivån definierar distribuerbara gränssnitt och komponenter som kan distribuera agenter eller hierarkier av agenter.

2.4.10 MBean

En *Management Bean* eller kort en MBean är ett Java-objekt som implementerar specifika gränssnitt, rättar sig efter vissa designmönster och med vars hjälp man påverkar hanteringen och styrningen av resurser [48, 49]. Dess gränssnitt tillhandahåller all den information som behövs för en applikation att kunna administrera, hantera och styra den [49].

Java Management eXtension (JMX) definierar fyra olika typer av MBeans där varje typ är definierad av, till exempel, att implementera vissa gränssnitt eller följa vissa namnkonventioner [49].

Generellt gäller för en MBean att den måste vara en konkret klass, måste ha en publik konstruktor, måste implementera dess egna MBean hanterings- och styrningsgränssnitt eller *DynamicBean*-gränssnittet, den kan generera händelsesignaler och dess attribut har *getter*- och *setter*-metoder [46].

MBeans kan tillhandahålla administrering, hantering och styrning för både Java- och icke-Java-applikationer [46].

Det fyra olika typerna av MBeans är: *Standard*, *Dynamic*, *Open* och *Model*. Varje typ stämmer överens med olika typer av administrerings-, hanterings- och styrningsbehov och är därför olika lämpade för olika situationer [49].

- *Standard MBean* är den enklaste att designa och implementera. Deras hanterings- och styrgränssnitt är beskrivet genom deras metodnamn och genom det faktum att de implementerar ett gränssnitt som slutar på “MBean”. Dessa är lämpliga där hanterings- och styrningsgränssnittet av en resurs är stabil.
- *Dynamic MBeans* måste implementera *DynamicMBean*-gränssnittet och exponera dess metoder och attribut vid körning genom att tillhandahålla de som anropar dem med metadata som beskriver dem. Dessa är användbara för att slå ihop icke-överensstämmande resurser eller där hanterings- och styrningsgränssnittet förändras.
- *Open MBeans* är dynamiska MBeans som bara använder de primitiva *wrapper*-typerna och ett fåtal andra i deras metoder och attribut. Dessa kan upptäckas och användas av vilken klient som helst vid körning och kan användas utan att behöva några extra JAR-filer (packad fil, ungefär som ZIP).
- *Model MBeans* är också dynamiska MBeans som är fullt konfigurerbara och självbeskrivande vid körning. De tillhandahåller en generisk MBean-klass med ett *default*-beteende för ett dynamiskt styrmedel av resurser.

Alla typer av MBeans exponerar sina attribut och metoder för att tillåta resurser representerade av en MBean att kontrolleras av hanterings- och styrningsapplikationer (administrationsapplikationer) [49].

2.5 Applikationsserver

En applikationsserver är ett flernivåsystem eller en variant av klientserver-system där man kör hela programvaran eller delar av programvaran på olika servrar och koppla

ihop den med diverse olika komponentobjektgränssnitt som exempelvis CORBA, .NET, J2EE [50].

När man implementerar en applikationsserver implementerar man samtidigt en arkitektur som är uppdelad i flera skikt. Principen är att man placerar applikationsservern i ett skikt mellan databasen och klienten samt överför logiken i klienten och databasen till objekt med väl definierade gränssnitt (komponenter). Applikationsserverns uppgift är sedan att exekverar logiken och sköta kommunikationen mot databasen [51].

Genom att använda applikationsservrar kan man alltså isolerar affärslogik i komponenter [35]. Fördelen med att samla logiken centralt är att man kan centralisera säkerhet och administration. Om applikationsservern erbjuder möjlighet att använda digitala certifikat kan man sätta upp en säker applikationsplattform utan att den som skriver applikationslogiken behöver bekymra sig om det. Administrationen förenklas också av att logiken samlas centralt eftersom man kan uppdaterar logiken utan att gränssnitten påverkas. Resultatet av en applikationsserver är att man utvecklar system som är modulära, har en god skalbarhet eftersom exempelvis databaskopplingar används effektivt samt att resurser delas och återanvänds mellan klienter, och man får ett robust och dynamiskt system [51].

En av avgränsningarna var att vi att skulle använda oss av JBoss applikationsserver och därför förklarar vi lite mer om vad det är och dess specifika egenskaper.

2.5.1 JBOSS

JBoss startades i mars 1999 och var då endast en EJB *container* – i denna kontext är en *container* en applikation eller ett subsystem i vilken ett programblock känt som komponent är exekverad [52]. Idag är JBoss uppe i version 3.0 och har utökats med att utföra hela J2EE stacken och mer än så och är således en mogen och fullfjädrad Java-applikationsserver [53].

JBoss 3.x serien är helt och hållet baserat på J2EE och innehåller följande [53, 54]:

- Fullt stöd för J2EE-baserade API:er
- HTTP 1.1 *web server*
- JCA, *Java Connector Architecture*
- JMX, *Java Management eXtensions*
- CMP2.0, *Container Managed Persistence*
- Automatisering och affärsberedskap såsom *hot-deploy*, modulär design, dynamiska proxy-adresser och ingen kompilering

JBoss är ett gratis *open-source*-projekt och är distribuerat under *Lesser General Public License* (LGPL) [53].

2.5.1.1 Hur man använder sig av JBoss och J2EE [55]

Man skapar en Enterprise JavaBean som består av minst av tre klasser:

- Ett *remote interface* som är en klass som beskriver bönans metoder för yttervärlden.
- En *bean*-klass som implementerar de metoder som finns beskrivna i sitt *remote interface*.
- Ett *home interface* som beskriver hur bönan skapas, hanteras och tas bort.

Man skapar sedan en mapp som heter META-INF som skall innehålla en *deployment descriptor* vilket är en fil som heter "ejb-jar.xml". *Deployment descriptor* är en konfigurationsfil som talar för EJB-servern vilka klasser som formar bönan, *home interface* och *remote interface*. I *deployment descriptor* finner man också information om bönans JNDI-namn (Java Naming och Directory Interface) som är en tjänst för att finna objekt med hjälp av objektnamnet eller andra attribut. Finns det fler än en böna i paketet talar den angivna *deployment descriptor* även om för servern hur bönorna interagerar med varandra.

Klasserna och META-INF-mappen som innehåller *deployment descriptor* paketeras i ett JAR-arkiv som skall ha en mappstruktur som reflekterar hierarkin inom paketstrukturen. JAR-filen kopieras till en *deployment*-mapp på JBoss-servern vilket kallas att den "deployas".

För att nå de tjänster som EJB:n erbjuder behöver man utveckla en klient som exempelvis kan bestå av annan EJB, en JSP sida eller en enskild applikation. Klienten kommer att nå bönans service genom att använda sig av JNDI-tjänsten och ropa på *home interface* med hjälp av RMI.

3 Metod

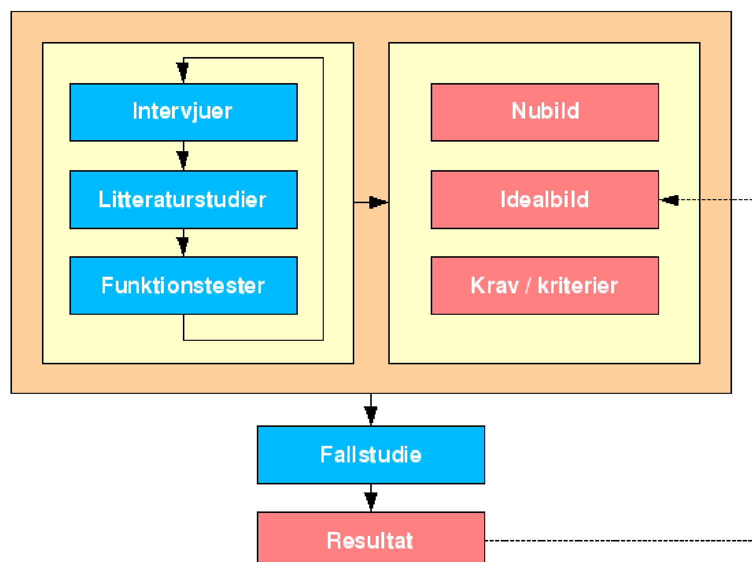
För att svara på frågeställningen i denna uppsats har vi genomfört en modifierad *feasability study*. Modifierad eftersom en *feasability study* vanligtvis används för utreda om ett informationssystem eller projekt kan genomföras och omfattar ett stort antal punkter och aspekter ur både ekonomisk, operationell, teknisk och tidssynvinkel [56].

Vår *feasability study* kommer att vara en mycket bantad men koncis och preciserad version av en omfattande och mer generell *feasability study* och bygger mer eller mindre på idéer framlagda av Castro et al. om innehållet i en sådan studie [56].

3.1 Feasability study

Inom ramen för vår *feasability study* ligger sådana steg som intervjuer, litteraturstudier och ett praktiskt programmeringsarbete som vi kommer att kalla en fallstudie. Denna fallstudie grundar sig på idéer inom *prototyping* då det gäller framtagning och testning (och det gäller då att skilja på *feasability study* och fallstudie).

En *feasability study* handlar i stort sett om att utreda om något är möjligt att genomföra [56]. Därför kan man kort säga att vår *feasability study* handlar om att via fallstudien ta reda på om det är möjligt att genomföra en systemförändring från den nuvarande arkitekturen till något som vi kallar **idealarkitekturen**. Denna idealarkitektur har framtagits i samband med de intervjuer som genomförts. Företaget har alltså en ganska klar bild om hur de vill att idealarkitekturen skall se ut.



Figur 2 - Vår *feasibility study*

Via en iterativ process med intervjuer, litteraturstudier och enklare funktionstester (för att utreda hur vissa applikationer, protokoll mm fungerar) skall tre delresultat tas fram.

- Nubild – en beskrivning av hur arkitekturen ser ut idag

- Idealbild – en beskrivning om hur arkitekturen skall se ut i framtiden och vilka egenskaper som denna arkitektur skall ha
- Krav och kriterier – för att idealarkitekturens önskade egenskaper skall uppfyllas.

Fallstudien är ett programmeringsprojekt där vi utreder om det överhuvudtaget är möjligt att ta sig från nubilden till idealbilden kopplat mot de krav och kriterier som tagits fram. Eftersom det är omöjligt för oss att hinna analysera och designa hela arkitekturen så utförs fallstudien på endast en del av hela idealbilden, men på en tillräckligt stor del för att vi skall kunna utläsa något adekvat resultat. Resultatet blir alltså en koppling mot idealbilden via krav och kriterier. Trots att vi bara tittar på detta specifika fall så kan vi utifrån detta konstatera om det är möjligt, vilka alternativa vägar man kan gå och vad som skall göras härnäst.

3.2 Litteraturstudier

Enligt Backman [57] måste all vetenskaplig forskning inledas med en litteraturstudie där forskaren läser in sig på det ämne som skall studeras och på vad som redan är gjort inom området.

För att överhuvudtaget kunna börja med detta var vi dock tvungna att snabbt skapa oss en uppfattning om exakt vad det var som vi skulle läsa in oss på genom att bilda oss en uppfattning om problemområdet i stort. Detta gjorde vi dels genom att diskutera med de systemarkitektursansvariga på företaget för att kunna förstå vad de ville åstadkomma och dels genom att skumma igenom avhandlingar som tidigare bearbetat delar av det som vi uppfattade som problemområdet. Vi läste dessutom in oss på olika tekniska lösningar som J2EE, SOAP och liknande tekniker som behandlar centralisering, skalbarhet och flexibilitet. Allt för att få en klarare bild av hur teknikerna fungerar och relaterar till varandra och om dessa kunde användas i vår studie.

Informationen baserar sig på både skriftliga och webbaserade källor. Dock är det så att eftersom vi har bearbetat ett område som är relativt nytt och använt oss av en mängd även dessa relativt nya tekniska lösningar så har mycket av den aktuella informationen funnits på Internet. Information har till stora delar bestått av dokumentation av de olika teknikerna, t.ex. hur de fungerar och med vilka andra tekniker de fungerar.

I samband med litteraturstudierna så har vi dessutom utfört diverse småtester på de olika tekniker vi studerat för att utröna om dessa skulle kunna vara applicerbara enligt de krav och kriterier som under studiens lopp började urskiljas. Allt för att vi inte skulle sikta in och fördjupa oss på ett spår som senare visade sig vara missriktat eller helt enkelt inte tillräckligt bra.

Då det gäller vilka tekniker som vi har studerat så har vi valt att endast behandla sådana tekniker där själva applikationen inte kostar pengar, utan där tillverkarna ofta istället tjänar pengar på dokumentationen. Dock finns det mycket information att hämta om de tekniker vi studerat på annat håll och att detta skulle påverka litteraturstudierna på något sätt har vi inte kunnat påvisa och dessutom har ju utvecklingskostnaden ur applikationssynpunkt varit helt gratis.

3.3 Intervjuer

Ändå från början har vi haft en öppen dialog med personer på företaget och det är tack vare detta som vi ha kunnat arbeta fram de grundfrågeställningar som präglar denna uppsats. Med detta vill vi säga att vi inte bara samlat information med hjälp av intervjuer utan även av ostrukturerade dialoger, förfrågningar och övriga diskussioner.

För den mesta övergripande förståelsen av problemområdet har vi dock genomfört två längre intervjuer med företagets VD och tre långa systemarkitekturmöten med en systemutvecklare på företaget. Dessa intervjuer/möten syftade till att ge en inblick i företagets nuvarande arkitektur samt den idealarkitektur som användare och utvecklare fastställt. Vi hade även ett möte med en domännamnsadministratör för att ytterligare öka förståelsen och kunskapen om problemområdet.

Intervjuerna som genomfördes stämmer väl överens med det som Holme och Solvang [58] kallar kvalitativa intervjuer vars syfte är att öka informationsvärdet och skapa en grund för djupare och mer fullständiga uppfattningar om de fenomen man studerar. I den kvalitativa intervjun använder forskaren sig inte av standardiserade frågeformulär utan ställer samman viktiga frågor i en manual eller handledning som slutligen ligger till grund för intervjun.

Styrkan med den kvalitativa intervjun är att undersökningssituationen har formen av ett samtal vilket innebär att forskaren utövar den minsta formen av styrning vad det gäller undersökningspersonerna. Nackdelen med tekniken är däremot att analysen av informationen ofta är både tidsödande och krävande [58].

Eftersom det vi är intresserade av gäller arkitektur så var syftet med intervjuerna inte så mycket att skriva ned exakt vad som sades under mötena eller att på förhand specificera en massa frågor. Istället lades fokus på och komma fram till hur systemarkitekturen ser ut idag och hur de vill att den skall se ut i framtiden. Vad som var av yttersta vikt att komma fram till var alltså de kriterier och krav som ställdes på den framtida ideala arkitekturen. Detta för att kunna utreda om det är möjligt att genomföra den systemförändring som krävs för att arkitekturen skall bli flexiblare och mer skalbar.

I bilaga 2 finns de intervjufrågor som vi använde oss av när vi genomförde en första intervju med en systemutvecklare på företaget. Dessa frågor användes endast som utgångspunkt och själva intervjuformen var mer som en dialog eller som ett möte.

Eftersom hela processen med denna studie är en iterativ process så har vi hela tiden när nya frågor eller funderingar dykt upp haft en öppen dialog med personer på företaget. Detta tillsammans med litteraturstudier och systemtester samt vår fallstudie som beskrivs i nästa avsnitt har lett fram till resultatet av studien.

3.4 Prototyping

Vi använder oss av *prototyping* eftersom detta anses vara ett sätt att försäkra sig om att informationssystemet motsvarar de krav och behov som användaren ställer på systemet. Dessutom ger det oss tillräckligt med underlag för att förkasta en teknik eller applikation.

Det primära användandet av *prototyping* är att tillförskaffa sig kunskap och information som behandlar och påverkar tidig produktutveckling och det är en metod för att öka

nyttan av användarkunskap med syftet att fortsätta utvecklingen till en slutlig produkt [59]. Erhållen information och kunskap är viktig för att kunna identifiera svårigheter och problem.

En prototyp är generellt en funktionellt omogen modell av en föreslagen produkt som konstrueras för att utforska kriterier och krav, undersöka alternativa synsätt eller för att demonstrera en modells möjliga utförande [59].

Vi har valt att använda oss av *prototyping* för att testa alla de olika tekniker vi kommit fram till utifrån de litteraturstudier och intervjuer vi genomfört och se om dessa kan fungera i den idealarkitektur som framtagits. Detta för att snabbt kunna eliminera tekniker som inte passar och för att kunna koncentrera krafter på de tekniker som faktiskt skulle kunna fungera.

I initialskedet (funktionstester) är vi enbart intresserade av att utreda om det överhuvudtaget är möjligt att använda oss av de tekniker vi hittat och om dessa går att använda sig av tillsammans. Det handlar alltså inte så mycket om att ta fram prototyper till vad som skulle kunna bli färdiga produkter utan istället att snabbt och effektivt kunna utvärdera de tekniker vi hittat gentemot de krav och kriterier som framtagits. Så för att exempelvis utreda om en ASP-klient skulle kunna kommunicera med en JBOSS applikationsserver så testades detta med ett SOAP-mellanlager. Vi programmerade alltså ett anrop från en ASP-klient via SOAP till en J2EE-böna som ligger i applikationsservern JBoss och kontrollerade att svaret tillbaka blev det som vi ville ha. Om detta var fallet så var det en fungerande prototyp som vi då kunde ha i åtanke senare.

När det gäller fallstudien sätter vi samman alla tekniker som fungerar med varandra för att på ett adekvat och resurseffektivt sätt se om det går att konstruera en prototyp enligt de krav och kriterier vi kommit fram.

Det går i stort sett ut på att ha en databaslösning i botten (precis som i ursprungsläget, nubilden) och sedan lägga på lager av olika tekniker ovanpå denna. Allt kontrolleras att det fungerar enligt de kriterier och krav på flexibilitet och skalbarhet som eftersträvas. I kort att gå från en tvåskiktslösning till minst en treskiktslösning, från plattformsberoende till plattformsoberoende och från decentraliserad affärslogik till centraliserad affärslogik.

Vi kommer som vi sagt tidigare att endast genomföra fallstudien på en del av idealarkitekturen. Dock anser vi att alla de funktionstester vi genomför tillsammans med fallstudien kommer ge oss tillräckligt med resultat för att kunna dra adekvata slutsatser.

4 Empiri

Beskrivningen av fallföretaget, nubild och idealbild är alla delresultat av de intervjuer och möten som vi haft med domännamnsadministratör, systemutvecklare och VD:n för företaget. Utifrån dessa resultat har vi sedan kommit fram till ytterligare ett delresultat, nämligen de egenskaper, krav och kriterier idealsystemet måste ha och kunna leva upp till. Kraven och kriterierna används för att kontrollera resultatet av fallstudien och hur användandet av plattformsoberoende tekniker kan möjliggöra en centraliserad, flexibel och skalbar systemarkitektur för en starkt föränderlig miljö.

Kapitlet inleder med en beskrivning av företaget (*beskrivning av fallföretaget*) och fortsätter med en redovisning av de tre delresultat som vi utfört vår fallstudie på för att studera vårt valda forskningsområde. *Nubild* beskriver hur dess systemarkitektur ser ut idag, *idealbild* med dess önskade egenskaper hur de vill den skall se ut i framtiden och *krav och kriterier* beskriver vad som måste uppfyllas för att nå fram till idealbilden.

Vi redovisar dessutom vår fallstudie, det vill säga den prototyp som vi byggt för att undersöka om det gick att realisera idealarkitekturen.

Till sist redovisas en procedurbeskrivning för att få en överblick av hur vi gick tillväga när vi utförde våra experiment, funktionstester och fallstudien (prototypen) samt resultaten av dessa. Resultaten sammanfattas dessutom i ett resultatanalysavsnitt kopplat mot kraven och kriterierna.

Det är inte av vikt att veta vem som sade vad eftersom vår intervjuteknik eller studie inte är i behov av detta. Det viktiga var istället att få fram sammanhängande resultat av våra möten och diskussioner med personer på företaget.

Alla olika teknikval och dess berättigande, bland annat när det gäller plattformsoberoende, finns dokumenterat i avgränsningar som ligger under introduktionsavsnittet och i teoriavsnittet. Kortfattat är att det fanns krav från företagets sida att vi skulle använda oss av en JBoss applikationsserver som har stöd för J2EE (med allt vad det innebär). Vi använder oss dessutom av en *Web-services*-lösning med hjälp av Axis, som är en SOAP-motor, när det gäller kommunikation mellan klientapplikationer och applikationsservern.

4.1 Beskrivning av fallföretaget

Koncernen House of Ports är baserad i natursköna Jonsered strax utanför Göteborg i de klassiska fabrikslokalerna. I denna kreativa miljö arbetar de ca 50 anställda med tjänsteutveckling och försäljning av domännamn. Internetadressering är grundstenen inom koncernen – de olika bolagen inom Ports tillhandahåller bland annat juridisk rådgivning kring domännamnsfrågor, *hosting*-verksamhet, telemarketingtjänster och mjukvaruutveckling.

Företagets verksamhet tog på allvar fart kring 1997 då Internet började användas på bred front i Sverige och grundarna såg affärsmöjligheten i just adresseringsproblematiken. Företaget såg tidigt till att bli en av få auktoriserade ombud för de stora domännamnen och blev snabbt en av Europas största aktörer inom området.

Idag, sex år senare, präglas företaget av hög omsättning och god lönsamhet. Dock så har företagets interna system inte riktigt hängt med i den snabba utvecklingen utan börjar visa ålderstecken och otidsenlighet.

Domännamnsregistreringen är den viktigaste inkomstkällan för företaget. Samtidigt lever House of Ports i stor utsträckning på förändringarna och snårigheten inom domännamnsvärlden, exempelvis försäljning av nya toppdomäner (”biz”, ”info”, ”us”, etc.) och de oerhört diversifierade reglerna kring registreringen. Att vara först med nya toppdomäner, erbjuda den bästa servicen och att erbjuda helhetslösningar för större kunder är uttalade målsättningar. Detta medför att företagets interna system ständigt måste uppdateras för att fungera i denna föränderliga kontext.

Det fanns alltså inte någon uttalad strategisk tanke kring systemutveckling eller arkitekturval på företaget under de första åren utan istället fick de affärsmässiga kraven styra. Det har alltså varit upp till systemutvecklarna att själva besluta och bestämma om hur sådana frågor skulle lösas på bästa sätt.

Dessa förändringar har oftast implementerats som tillägg i olika klientapplikationer, skrivna i det programmeringsspråk som utvecklaren ifråga har föredragit. Flera år av ostyrd kodning har gjort affärslogiken snårig och oöverskådlig, så företaget beslutat att satsa på en ny, centraliserad informationssystemsstruktur där affärslogiken finns samlad centralt och klienterna endast tillhandahåller gränssnittet för slutanvändarna.

Trots de nackdelar som finns med den nuvarande systemarkitekturen så är det stora investeringar som är gjorda och det gäller att kunna återanvända stora delar av den befintliga koden. Kraven inkluderar att man skall arbeta med diverse standarder och enligt det objektorienterade synsättet med bland annat inkapsling. Detta för att underlätta dels för ny personal eller konsulter som då lättare kan sätta sig in i koden och dels för att kunna återanvända kod lättare.

Ytterligare saker som ibland hamnat mellan stolarna under den tid då företagets systemarkitektur vuxit fram är dokumentationen. Därför finns det då krav om att all nyutveckling skall dokumenteras enligt gängse standarder, allt för att underlätta vid felsökning och fortsatt utveckling av både intern och extern personal.

Visionen då det gäller systemutveckling är alltså att ha en strategi som möjliggör för en flexibel, säker och skalbar arkitektur.

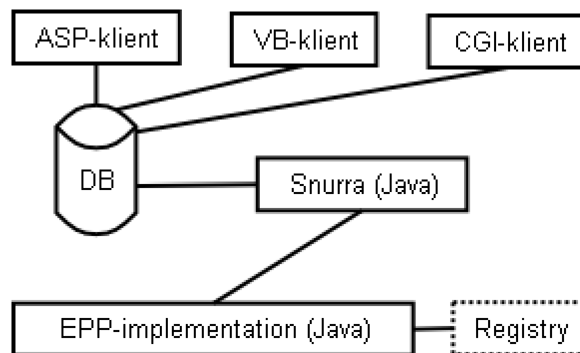
4.2 Nubild

Det ursprungliga behovet var att en användare via Internet skulle kunna registrera ett domännamn och för detta ändamål utvecklades en *Active Server Pages* (ASP) applikation som kopplades direkt mot en *Microsoft SQL Server* (databas). Modelleringen var att en användare skulle registrera en domän och det på exakt samma sätt hela tiden.

När man senare utökade verksamheten och möjliggjorde registrering mot fler domäner så gjordes detta antingen genom att lägga till kod i den befintliga ASP-applikationen eller så skapades nya applikationer. Eftersom det inte fanns någon uttalad systemutvecklingsstrategi så var det helt upp till utvecklaren att bestämma vilken lösning som passade bäst för henne eller honom. Detta betyder att komplexitet

inkrementellt har byggts in i systemet utan styrning med avseende på översikt, flexibilitet eller skalbarhet.

När affärsverksamheten startade så anpassades systemet till hur dessa *registries* (”com”, ”net” och ”org”) ville kommunicera vilket i stort sett endast var att via RRP skicka en textsträng med information om domänregistreringen. Vad man gjorde då var att implementera en ”jobbsnurra”, ett FIFO-kösystem som hela tiden var igång och kollade av om ny data hade lagts in i databasen.



Figur 3 - Nubild av registreringssystemet för EPP

Som arkitekturen ser ut idag så ligger det alltså en stor Microsoft SQL Server i grunden som i stort sett driver allt och med vilka alla applikationer integreras. Dessa applikationer består av allt från diverse shell-script, jobbköer från Unix och automatisk registervård till ett *Intranet* byggd med Active Server Pages (ASP). Förutom alla dessa olika applikationer så finns det dessutom en mängd olika specialprogram för t ex EPP. Jobbkön som skriver till växeln består av mängd kod bestående av en ”if … else”- satser och detta är i längden ohållbart vid bland annat felsökning.

Detta tillsammans med delvis dålig dokumentation gör affärslogiken oöverskådlig, svår att felsöka och svår att utveckla utan en genomgripande förändring av arkitekturen.

Förändringar sker snabbt när det gäller kommunikation med *registries*, vilka protokoll som används och vilken information de vill ha. Därför är det viktigt att kunna genomföra en systemarkitektur som klarar av att på ett relativt enkelt sätt genomföra dessa förändringar utan att tvingas gå in och nödvändigtvis koda nya metoder. Dessutom måste man ta höjd för nya sätt att registrera på och skapa förutsättningar för att detta kan implementeras utan att behöva genomföra ytterligare en ny systemarkitektur.

4.3 Idealbild

Övergripande handlar det om att gå från en 2-lagersmodell till en 3-lagersmodell, vilket innebär att data inte skall kunna påverkas direkt av applikationerna. Man skall alltså införa ett mellanlager, ett kontrollager mellan data och gränssnitt enligt MVC:s principer.

Man skall utforma arkitekturen enligt den komplexa och föränderliga miljön man verkar inom och skapa en arkitektur som är flexibel, säker, skalbar och enkel att konfigurera. Det skall inte vara nödvändigt att skriva om kod om en ny version av ett protokoll införs eller om en toppdomän byter protokoll.

Som det är nu är det mycket speciallösningar i klientapplikationerna samt att samma funktioner finns spridda i flera olika klientapplikationer. Ett sätt att förbättra situationen är bland annat att centralisera den affärslogik som nu finns lagrad i de olika klienterna. Istället för att alla de olika klientapplikationerna har ”samma kod” skrivna i varsitt språk så är det enklare att ha dessa funktioner på en central plats. Detta förenklar både vid validering och vid felhantering. Distribuerade komponenter ansvarar för validering. Att hantera fel är i sig ett massivt stycke logik och det är då fördelaktigt att investera i central felhantering eftersom man då i idealfallet kan styra, underhålla och uppdatera systemet utifrån en enda plats.

Utifrån de funktionstester och litteraturstudier vi genomfört samt de intervjuer och möten vi haft med personer inom företaget har vi tagit fram en konceptuell modell av ett mer flexibelt, säkert och skalbart system. Ett system som skall präglas av ett objektorienterat tänkande, en komponentarkitektur, olika designmönster och Model-View-Control-paradigmen (MVC). Dessutom skall systemet tillhandahålla plattformsoberoende kommunikation mellan View- och Control-komponenter eftersom de nuvarande klientapplikationerna är skrivna på olika plattformar.

När man utformar ett system så har beställaren (förhoppningsvis) en god bild av vad de vill ha ut av systemet, vilka egenskaper det bör ha och vilka kriterier som bör vara uppfyllda. I vår fallstudie så var utgångspunkten problematiken och den obefintliga skalbarheten i det dåvarande systemet. Visserligen var prestanda, stabilitet och säkerhet god men i övrigt så fanns det en del områden som behövde förbättras.

- Obefintlig skalbarhet – Nya protokollversioner krävde stora källkodsförändringar.
- Ingen skriptbarhet – Det fanns inga möjligheter att via ett externt kommersiellt skriptspråk arbeta mot EPP-*registries*. Operationer utöver standardregistreringen utfördes via ett textverktyg (”eppic”, ett av Ports egenutvecklat verktyg) där man manuellt kunde utföra operationer mot registries.
- Ej tillfredsställande konfigurerbarhet – Konfigureringen av SSL-*sockets*, certifikat med mera sker genom manipulering av XML- och *property*-filer.
- Underhållsproblem - Ett nytt *registry* togs om hand genom att ta en kopia på ”eppic” och sedan införa de nödvändiga förändringarna manuellt. Detta ger ett stort antal möjliga punkter för fel och underhåll att undersöka och åtgärda vid fel.
- Tvålagersproblematiken.
- Den allmänna dokumentationen av systemet är bitvis bristfällig och koden är således svårbegriplig för en utomstående systemutvecklare.
- Systemet använder sig ej fullt ut av gängse systemutvecklings- och programmeringsparadigmer såsom objektorientering och designmönster och är således behäftat med arkitekturella svagheter.

Utifrån dessa problem började vi med kravframtagningsprocessen för att i största möjliga utsträckning få fram mätbara kriterier som vi kan utföra vår *feasibility study* mot. Systemarkitekturen kommer då att bli ett resultat av problematiken, komplexiteten och miljön.

De problem vi listar ovan och den problematik den ger upphov till utgör basen för de *egenskaper* vi vill att systemet skall inneha. När vi väl definierat dessa egenskaper gäller det att konkretisera dessa till mätbara *kriterier*. Kriteriet i sin tur skall om möjligt kunna utvärderas mot prototypen med ett enkelt ja- eller nej-svar eller en kort beskrivning. Exempel på detta kan vara.

Egenskaper	**Kriterier**
• Erforderliga delar av systemet skall kunna konfigureras på ett enkelt vis och under körning.	• Erforderliga delar av systemet skall kunna konfigureras med hjälp av en webbläsare på ett användarvänligt sätt. • Ändringar i konfiguration skall direkt slå igenom och ej kräva någon omstart av systemet.
• Systemet skall vara dokumenterat så att en utomstående utan detaljkunskaper skall kunna programmera och underhålla det.	• Systemet skall vara kommenterat enligt ”Javadoc” och företagets dokumentationsstandard. • Systemets externa gränssnitt skall finnas väl definierade i dokumentationen med erforderliga användningsinformation i manualform.

Dessa kriterier kan tämligen enkelt besvaras. I de fall en mer flytande uppfyllnadsskala behövs har vi valt att använda oss av en poängmodell där 1 är ”inte alls” och 5 är ”helt tillfredsställande”. Poängen med att använda sig av en numerisk respresentation av en språkligt definierad uppfyllnadsgrad är att man senare kan utnyttja dessa värden i exempelvis ett Balanced Scorecard[4] i de fall man t.ex. utvärderar flera olika prototyper mot varandra. Exempel på detta är:

Krav och kriterier	**Utvärdering**	**Poäng**
• Systemet skall vara kommenterat enligt ”Javadoc” och företagets dokumentationsstandard.	• Väl uppfyllt. Egentligen inte något som är intressant ur arkitekturell synvinkel, men eftersom Java är väldigt enkelt att dokumentera genom Javadoc så får man väldigt mycket gratis. Företagets dokumentationsstandard är inte relevant ur uppsatsen synvinkel så det tar vi inte upp här.	5

[4] En metod för värdering av kriterier, se t ex http://www.balancedscorecard.org/basics/bsc1.html

4.3.1 Egenskaper och kriterier

Som ett delresultat i vår *feasibility study* kom vi fram till att idealbilden skall ha följande egenskaper:

- Systemet skall ha en hög skalbarhet på följande punkter:
  1. Framtida prestandaproblem skall kunna lösas utan förändringar i systemets grundarkitektur.
  2. Nya protokoll och protokollversioner för domännamnshantering skall utan övergripande arkitekturförändringar kunna införas.
  3. Nya toppdomäner som utnyttjar tidigare implementerade protokollversioner skall enkelt kunna hanteras utan programmerares inblandning.
- Systemet skall ha följande egenskaper gällande konfigurerbarhet:
  1. Konfigureringen av kommunikationen mot en befintlig toppdomän skall enkelt kunna utföras.
  2. Konfigureringen skall ske på ett användarvänligt vis.
  3. Konfigurationen av en ny toppdomän skall kunna ske utan en programmerares inblandning.
- Systemet skall vara lättunderhållet för någon även utan programmeringskunskaper.
- Systemet skall följa god programmeringsetik.
- Systemet skall utgöra ett lager mot *registries* som enkelt skall kunna utnyttjas genom externa skriptspråk.
- Systemet skall vara fullgott dokumenterat.
- Systemet skall skydda affärskritiska data från otillåten åtkomst och manipulering.
- Systemet skall ha en säker felhantering där det är fundamentalt att inga affärskritiska operationer ”försvinner”.
- Tillfredställande loggning skall ske.

Systemet skall vara plattformsoberoende, både i sig självt och åtkomstmässigt. Dessa egenskaper bröt vi sedan ner i konkreta kriterier enligt den modell vi beskrev ovan.

4.4 Krav och kriterier

Egenskaper	**Kriterier**
• Framtida prestandaproblem skall kunna lösas utan förändringar i systemets grundarkitektur.	• Plattformen skall stödja sömlös klustring av flera servrar.
• Nya protokoll och protokollversioner för domännamnshantering skall utan övergripande arkitekturförändringar kunna införas.	• Nya versioner av EPP skall utan övergripande arkitekturförändringar kunna införas.

kunna införas.	• Arkitekturen skall medge att funktionalitet för andra registreringsmetoder och protokoll än EPP (ex. RRP, E-post) skall kunna införas.
• Nya toppdomäner som utnyttjar tidigare implementerade protokollversioner skall enkelt kunna hanteras utan programmerares inblandning.	• En vanlig användare skall utan specialistkunskap kunna lägga till ett nytt registry ifall detta registry använder sig av en redan implementerad EPP-version
• Konfigureringen av kommunikationen mot en befintlig toppdomän skall enkelt kunna utföras.	• Den data som konfigurerar en kommunikation skall presenteras begripligt och lättföränderligt via webbgränssnittet.
• Konfigureringen skall ske på ett användarvänligt vis.	• Konfigureringen skall ske genom ett webbgränssnitt i största möjliga mån
• Konfigurationen av en ny toppdomän skall kunna ske utan en programmerares inblandning.	• En vanlig användare skall genom webbgränssnittet kunna lägga till stöd för ett nytt registry.
• Systemet skall utgöra ett lager mot registries som enkelt skall kunna utnyttjas genom externa skriptspråk.	• Systemet skall offentliggöra utvalda publika metoder mot omvärlden så att dessa kan användas med hjälp av t ex Perl, PHP, VBScript, ASP etc.
• Systemet skall vara lättunderhållet för någon även utan programmeringskunskaper	• Underhållet skall ske via ett användarvänligt webbgränssnitt • Systemet skall eftersträva ”Single point of maintenance”
• Systemet skall följa god programmeringsetik	• Systemet skall vara implementerat på ett objektorienterat vis och använda sig av designmönster.
• Systemet skall vara fullgott dokumenterat	• Systemet skall vara kommenterat enligt ”Javadoc” och företagets dokumentationsstandard. • Systemets externa gränssnitt skall finnas väl definierade i dokumentationen med erforderlig användningsinformation i manualform.
• Systemet skall skydda affärskritiska data från otillåten åtkomst och manipulering	• Systemet skall använda sig av MVC-mönstret.
• Systemet skall ha en säker felhantering där det är fundamentalt att inga affärskritiska operationer ”försvinner”	• Systemet skall endast acceptera atomära kommandon – dvs. att alla operationer måste vara antingen helt framgångsrika eller helt misslyckade.
• Tillfredställande loggning skall ske.	• Loggning av samtliga operationer mot registries skall ske, samt kunna läsas via

- Systemet skall vara plattformsoberoende, både i sig självt och åtkomstmässigt.

en webbläsare.

- Åtkomst skall kunna ske från en godtycklig plattform med SOAP-stöd.
- Systemet skall kunna köras i godtycklig miljö med fullgott Javastöd.

4.5 Fallstudie, prototypbeskrivning

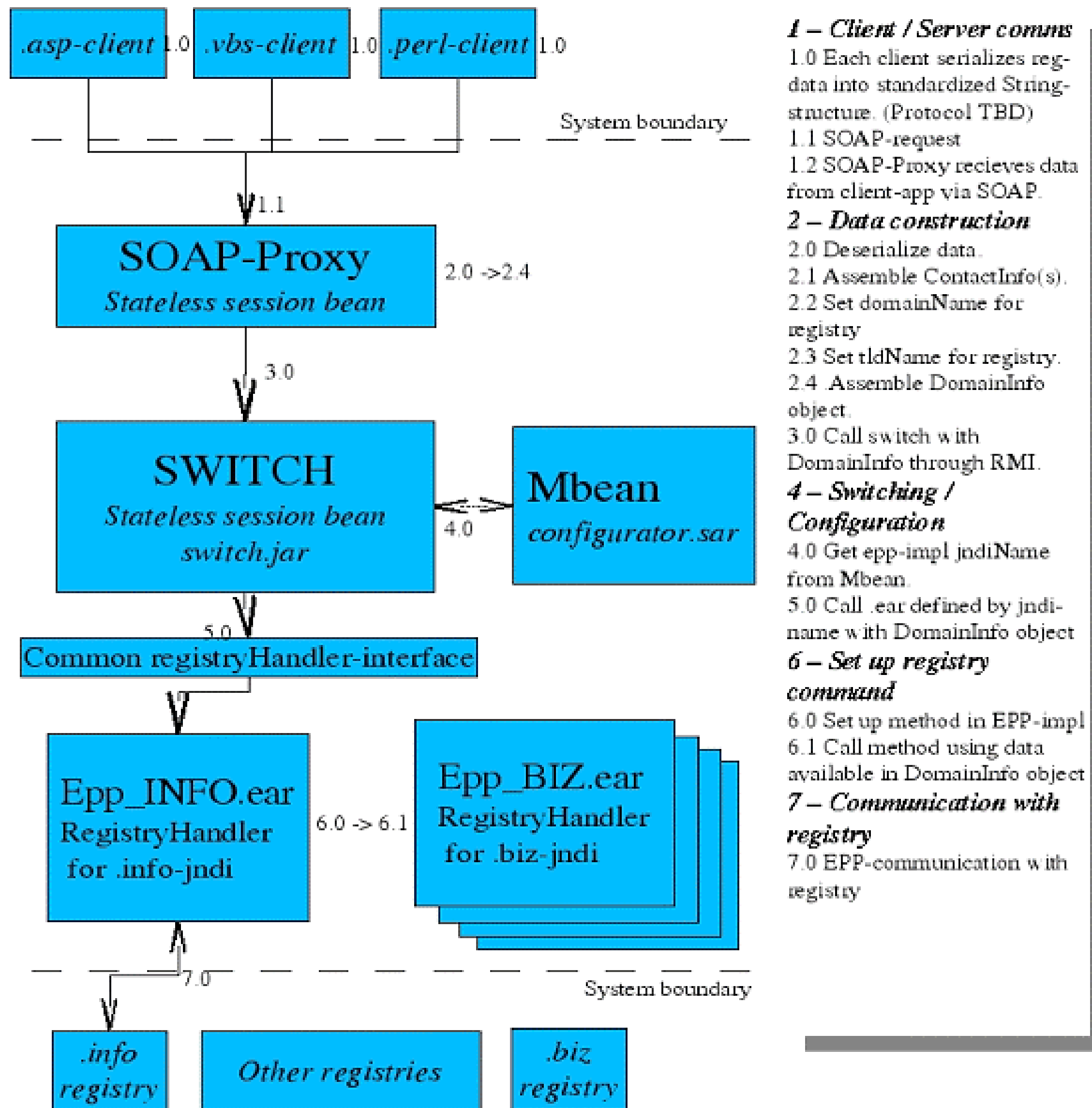
Vi vill påpeka att fallstudien inte är fullt applicerbar på samtliga av de egenskaper och svårigheter Ports system har, så som vi beskrev dem i inledningen. Att utforma och implementera en prototyp av en helt ny systemarkitektur för ett IT-företag av Ports storlek ligger klart utanför vad som hinns med under en magisteruppsats. Därför valde vi att koncentrera oss på vad som kan ses som kärnan i verksamheten – själva domännamnsregistreringen – och studera hur våra plattformsoberoende tekniker kan appliceras på detta område. För att sedan utvärdera om våra resultat är applicerbara för Ports framtida systemarkitektur och giltiga i en mer generell kontext.

Denna beskrivning av prototypen skall *inte* ses som någon komplett uppsättning analys- och designdokument enligt någon viss standard utan skall ses som en enkel beskrivning av prototypen, dess arkitektur och egenskaper. Vi använder oss dock av en UML-liknande notation i skisser och diagram.

Prototypen vi tagit fram är en J2EE-applikation för flexibel och skalbar domännamnsregistrering och hantering av domännamnsrelaterade användningsfall. Exempel på sådana användningsfall är att registrera namnservrar, lägga in kontaktinformation, förnya domäner eller överföring av domäner mellan *registrars*.

Som vi tidigare fastslagit så är applikationsservern JBoss 3.0.4 grunden i systemet där man sedan hakar på växeln, implementationer av olika EPP-versioner, MBeans som hanterar konfigureringen och gränssnitt som publicerar den funktionalitet som systemet skall tillhandahålla åt de olika aktörerna.

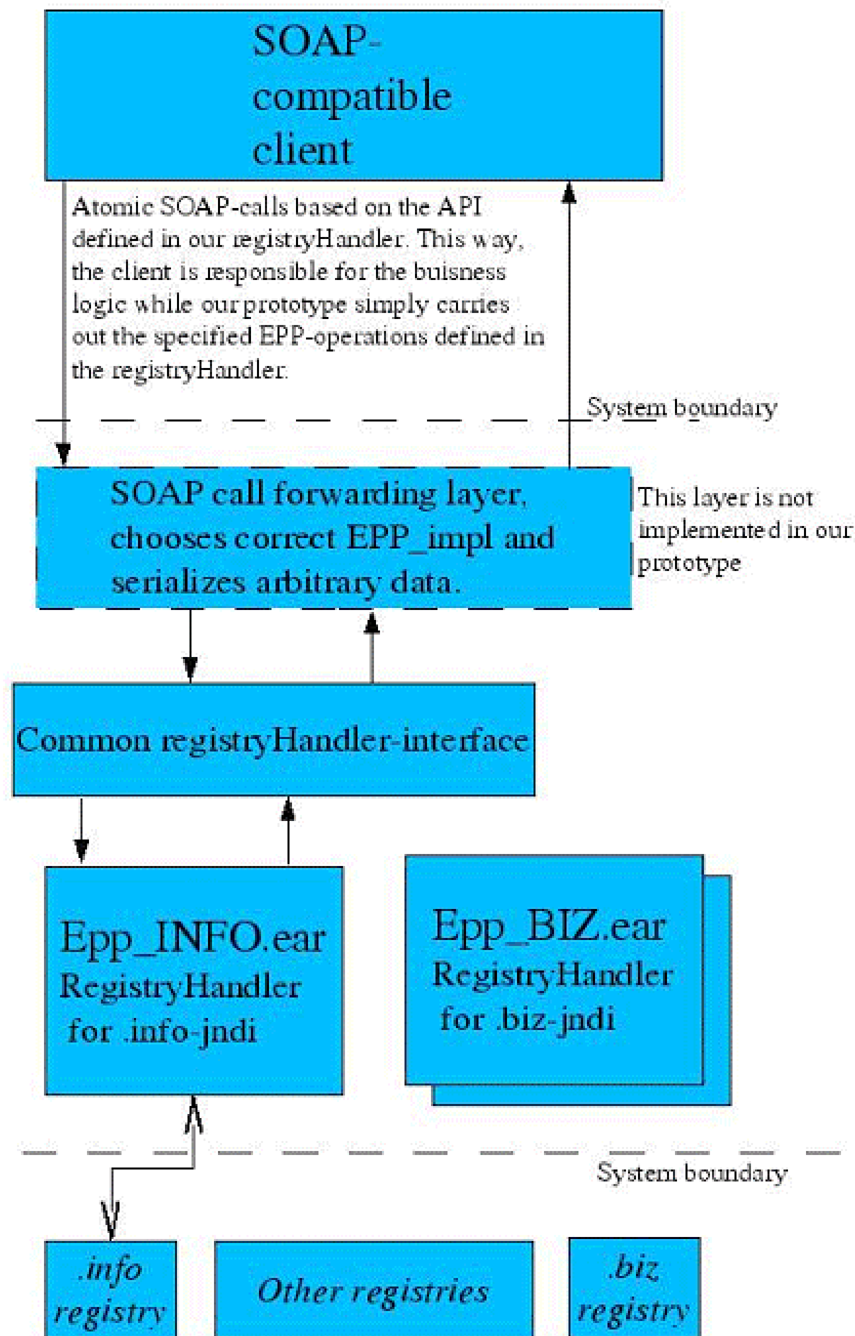
Basic system architecture, flow for EPP-based registration



Figur 4 - Basic system architecture, flow for EPP-based registration

I figur 4 ges en översiktsbild av systemet och dess omgivning tillsammans med en representation av programflödet där man kan följa standardfallet – en vanlig domännamnsregistrering för en godtyckligt EPP-domän – i detta fall en info-domän.

Observera att "TLD" är en förkortning för Top Level Domain i skisserna.



Figur 5 - Architecture for direct access to EPP-registries using SOAP

I figur 5 ser vi hur man istället arbetar direkt mot *registryHandler*-gränssnittet via ett tunt SOAP-skal med EPP-baserade kommandon snarare än *Switchens* användarfallsbaserade kommandon. Detta medför att ansvaret för affärslogiken hamnar hos klienten vilket precis är meningen eftersom det ger oss möjlighet att via SOAP-kompatibla skriptspråk utföra operationer mot *registries* utan att behöva lägga till dessa i *Switchens* källkod. Prestandan blir naturligtvis inte lika bra eftersom varje enskilt

kommando kommer gå fram och tillbaka mellan klient och server via SOAP, jämfört med *Switchen* där ett flertal kommandon kan exekveras internt inom servern.

Givetvis måste man ta höjd för felhantering även i detta fall. Eftersom objekt och parametrar måste instansieras på serversidan även här så medför vår design av dataklasserna att vi får datavalidering även i detta fall. Dock skall klienten givetvis försöka validera utdata.

Som vi senare kommer redovisa fungerar även felhanteringen via SOAP väl så dessa klienter kommer underrättas ifall något går snett inne i servern. Vi har inte implementerat det skikt som i skissen ligger mellan klient och *registryHandler*-gränssnitt.

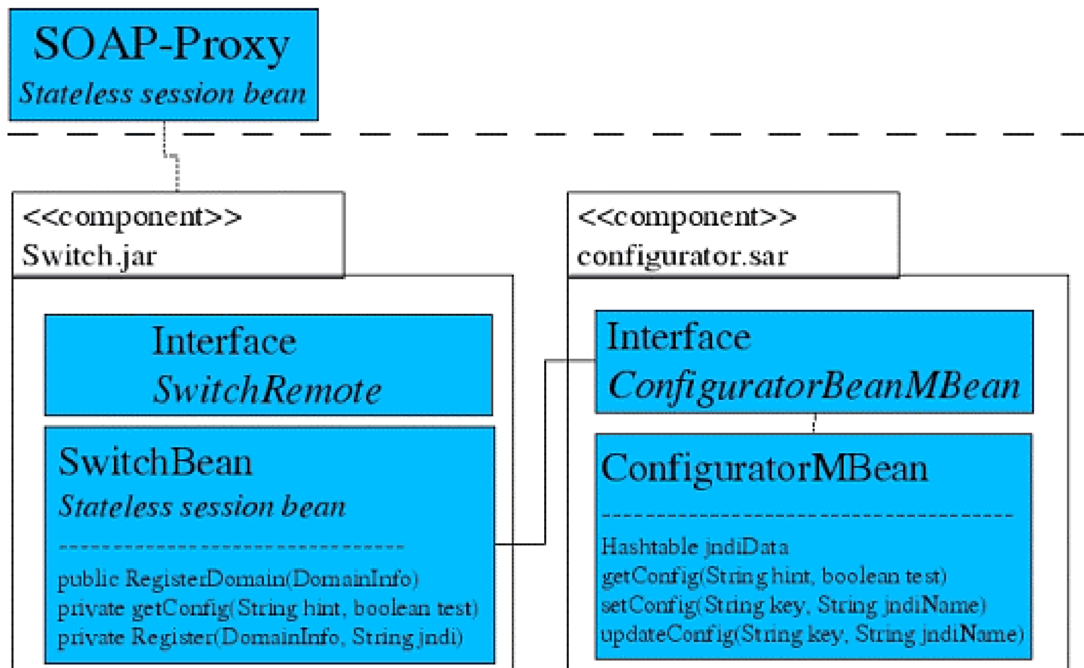
4.5.1 Komponentdesign

Vi kommer ytligt gå igenom systemprototypens fyra olika huvudkomponenter: SOAP-proxy, Switch, RegistryHandler, EPP-implementationer.

4.5.1.1 SOAP-proxy

Denna komponent skall ansvara för att serialisera och deserialisera de arbiträra objekt som kommer in till eller skall skickas från systemet via SOAP.
Vi har i prototypen inte implementerat denna komponent men designen av den är baserad på att den skall vara en stateless session bean vars metoder publiceras som *Web-services* genom Axis. Eftersom kompabiliteten mellan Axis och olika icke-Javabaserade plattformar inte är fullgod (mer utförligt beskrivet på annan plats i resultatdelen) så behöver växelkomponenten en yttre komponent som ansvarar för att bygga upp de objekt som ingår i en domännamnsregistrering. I vår nuvarande lösning är SOAP-proxyn helt sessionslös men den kan eventuellt i en skarp implementation skrivas som en stateful session bean istället som i så fall kan ge oss en viss sessionshantering.

4.5.1.2 Switch



Figur 6 - Switch architecture – Yttre gränssnitt, Switch.jar och configurator.sar

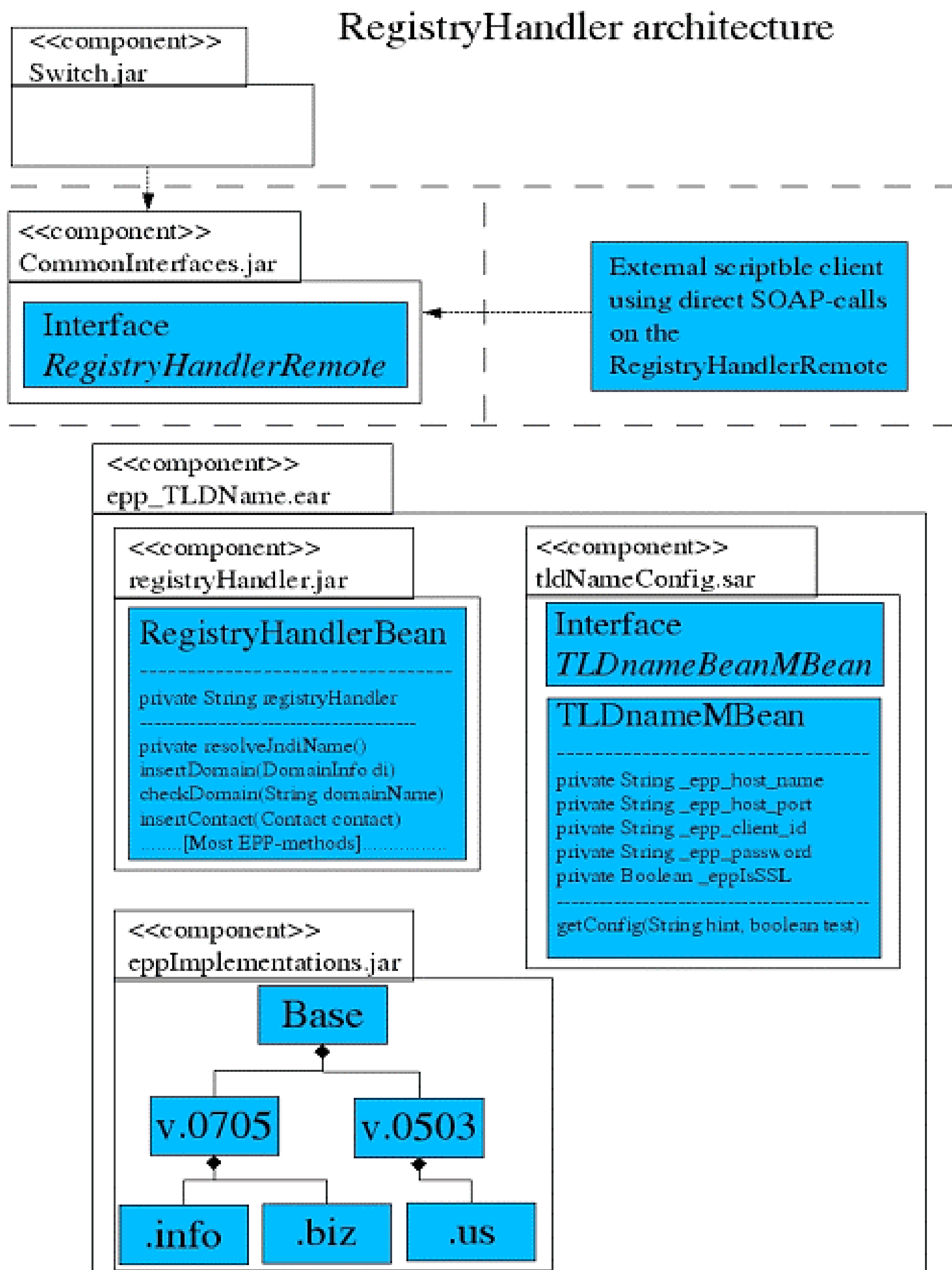
Systemprototypens nav är en *stateless*-sessionsböna ”Switch.jar” (se figur 6), som tar emot registreringsförfrågningar, hämtar rätt konfiguration för aktuell toppdomän och sedan anropar den komponent som ansvarar för registrering under den aktuella toppdomänen. Den publicerar en publik metod ”RegisterDomain(DomainInfo)”. Denna metod kan anropas från RMI-kompatibla klienter – exempelvis andra sessionsbönor, vanliga Javaapplikationer eller via en RMI-kompatibel SOAP-proxy som exempelvis Axis. Givetvis förutsatt att klienten kan tillhandahålla ett korrekt parameterobjekt *DomainInfo*, som innehåller all data som är nödvändig för en registrering.

Komponenten *configurator.sar* ansvarar alltså enbart för att plocka fram rätt referens till den komponent som skall sköta om kommunikationen med det aktuella *registriet*.

Slutligen används metoden *Register* för att anropa själva toppdomänen via dess *RegistryHandler*. Observera att funktioner för datavalidering och andra kontroller finns eller enkelt implementeras i växeln, även om dessa inte syns i vår skiss.

Switchens yttre gränssnitt är alltså användarfallsbaserade och kapslar således in de interna operationer som behövs för att fullborda en registrering – dvs. registrera kontakter, namnservrar och liknande utöver själva domännamnsregistreringen. På detta vis centraliserar vi affärslogiken. Vi har än så länge bara implementerat registreringsmetoden, men i framtiden kan man givetvis utöka *Switchens* funktionalitet med nya användarfallsbaserade metoder.

4.5.1.3 RegistryHandler



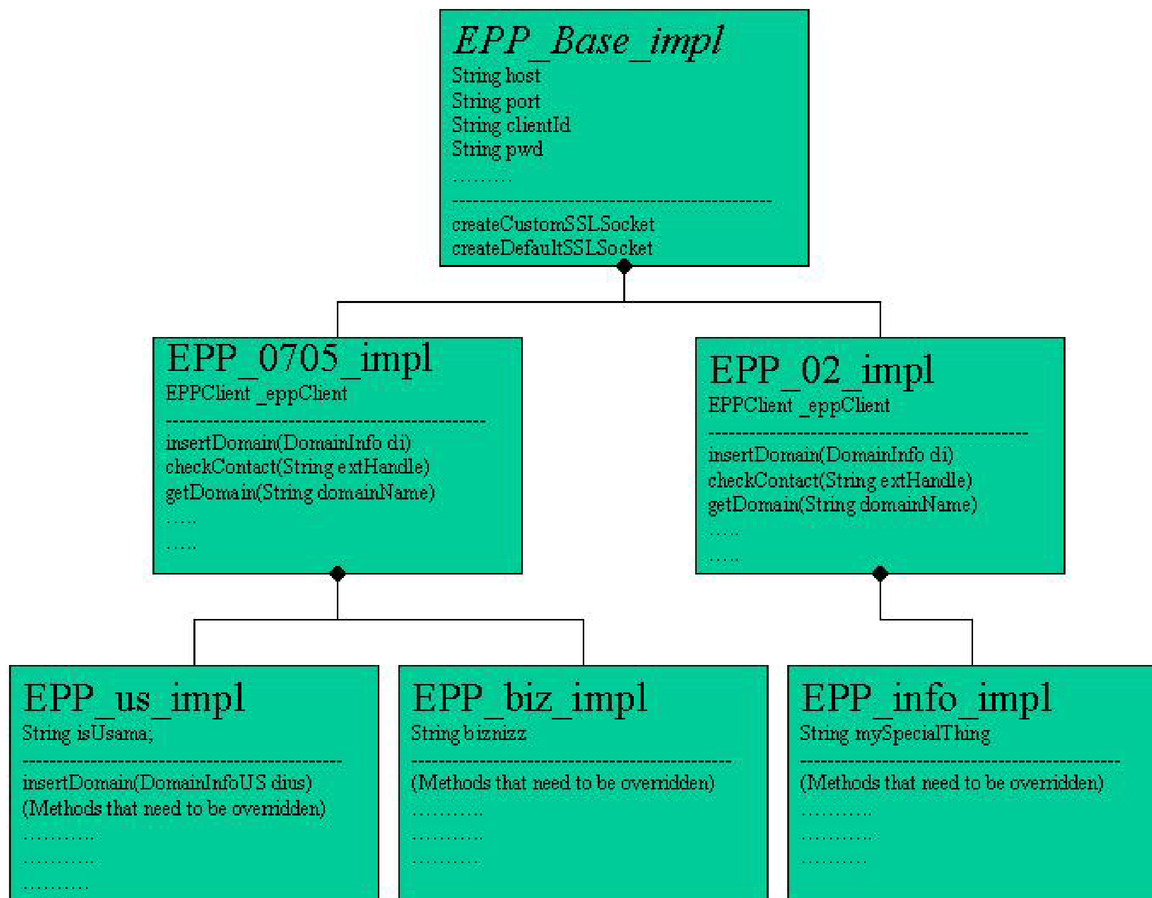
Figur 7 - RegistryHandler architecture

Figur 7 visar *registryHandler*-implementationen där varje ”epp_TLDName.ear” motsvarar den komponent som ansvarar för en viss toppdomän. Det finns alltså en komponent per toppdomän med sitt eget unika JNDI-namn. Dessa ligger driftsatta bredvid varandra i vår applikationsserver tillsammans med *Switchen*, gemensamma klasser, gemensamma gränssnitt etc. När växeln vill utföra en domännamnsregistrering

anropar den via JNDI den aktuella komponenten. *RegistryHandlerBean*, som är en *stateless*-sessionsböna, har i sin *deployment descriptor*-fil (”ejb-jar.xml”) JNDI-namnet till sin egen MBean som ansvarar för konfigureringen av komponenten. Eftersom varje *RegistryHandler*-komponent har exakt en MBean som ansvarar för dess konfigurering är det okej att ha en statiskt definierad sökväg (JNDI-namn) till den specifika MBean. Denna sökväg plockas fram genom ”resolveJndiName()”-metoden som anropas för varje kommando. Detta JNDI-namn sätts sedan som attribut i den specifika EPP-implementationen. För varje anrop mot ett registry kontrolleras om det finns någon aktiv, uppkopplad, instans av en *EPPClient*[5]. Aktiv instans jämförs med aktuell konfigureringsdata, om konfigurationerna inte överensstämmer instansieras en ny *EPPClient* med aktuell data. Om en inaktiv klient finns kopplas den upp efter att ha konfigurerats. Saknas klient instansieras givetvis en ny med aktuell konfigureringsdata.

4.5.1.4 EPP-implementationer

Som vi beskrev i teorin finns EPP-protokollet i ett antal olika versioner, varav flera är i skarp drift hos exempelvis ”biz”, ”us” och ”info”. För att få till en bra struktur på dessa olika versioner byggde vi en arvsstruktur för de olika implementationerna (se figur 8).



Figur 8 - EPP-implementationerna

[5] Den klass som hanterar den verkliga kommunikationen med ett EPP-Registry.

I toppen lade vi en abstrakt basklass som innehöll de attribut som beskriver ett *EPPClient*-objekt, samt metoder som skapar de SSL-uppkopplingar som används för kommunikationen mot registries. Dessa var naturligtvis gemensamma för samtliga möjliga EPP-versioner. I de fall en underversions *EPPClient*-objekt inte behöver all konfigureringsdata ignoreras helt enkelt de övertaliga attributen. Barnen till basklassen är konkreta klasser som var och en implementerar en viss protokollversion. Många metoder är visserligen exakt likadana oavsett version, men då själva *EPPClient*-objekten, som metoderna utförs på, inte är kompatibla över versionerna så implementeras ändå metoderna specifikt för varje version. Barn till denna nivå är domänspecifika implementationer som visserligen fullt ut använder en viss EPP-version, men som ändå valt att använda någon form av icke-standardiserad funktionalitet som måste tas hänsyn till. Oftast räcker det med att låta den specifika implementationen ärva med sig hela basklassens metoder för att sedan göra en *override* för specialfallen.

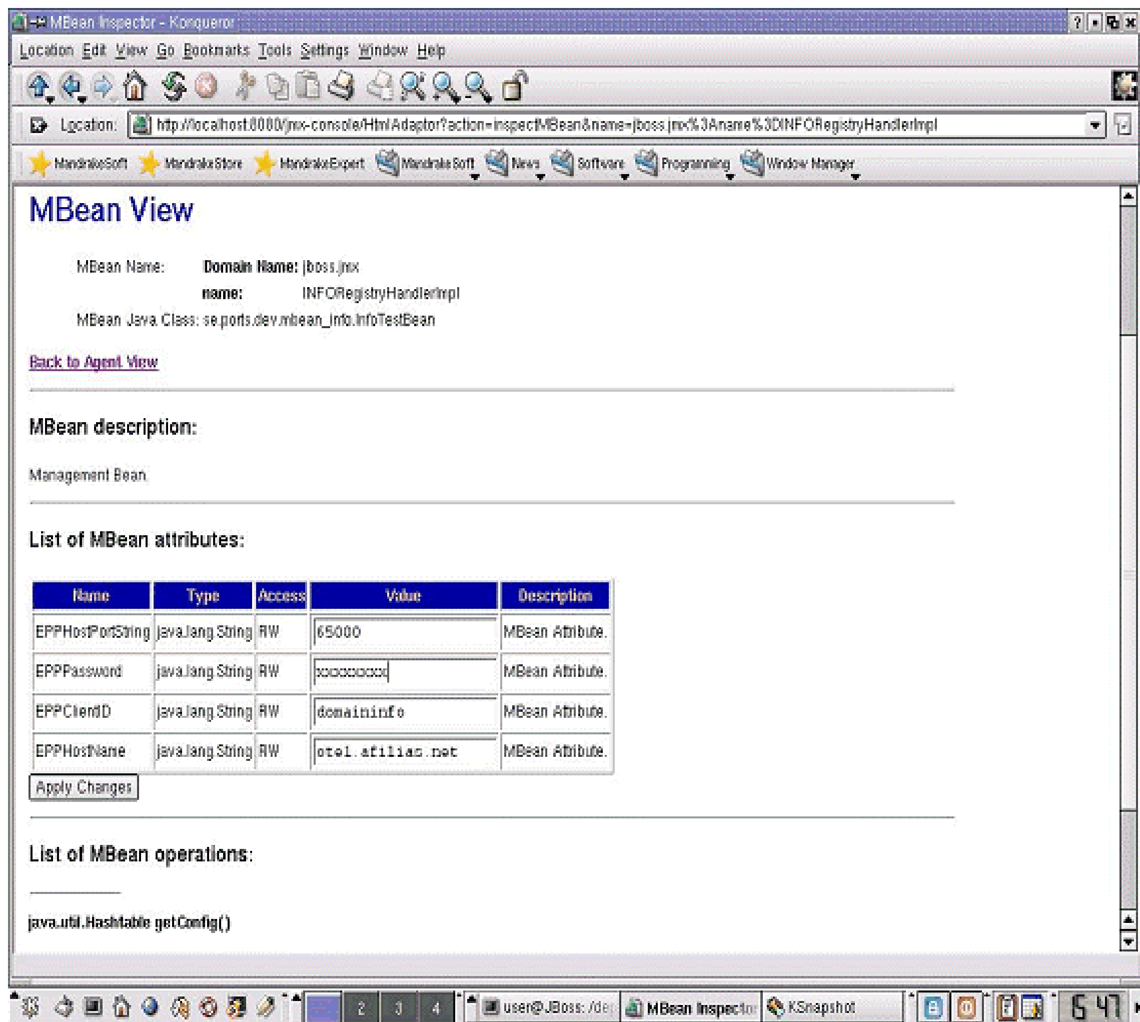
4.5.2 Konfigurationshantering i prototypen

Som vi tidigare redogjort för bygger flexibiliteten och skalbarheten till stor del på konfigureringen med hjälp av MBeans. Vi identifierade två konfigureringsfall.

- Konfigurera växelkomponenten så den anropar rätt EPP-implementation
- Konfigurera en EPP-implementationskomponent

Det första fallet är i standardläget där växeln tar emot en registreringsförfrågan. Vår lösning är att låta växeln automatiskt konfigurera sig via en MBean. MBean tar emot hela namnet på den domän där den skall registrera sig och använder en enkel logik och en prioriteringslista som tar reda på under vilken toppdomän som registreringen skall ske. Det finns dock ett specialfall som kan krångla till det. Det är när toppdomänen består av mer än en ”del”. Med en del menas ”se” eller ”com”. Med flera delar menas ”co.uk” eller ”com.cn”.

Hur vet man när man skall registrera mot ”uk”:s servrar eller mot ”co.uk”:s servrar? Lösningen var att vi byggde en ganska avancerad *DomainName*-klass som utifrån vissa parametrar tar fram den korrekta toppdomänen. I en MBean har man sedan en prioriteringsordning på domännamnen då det rör sig om en toppdomän som exempelvis ”co.uk”. Först försöker man hitta en konfiguration för ”co.uk”, sedan låter man den försöka hitta en konfiguration för enbart ”uk”. Finns ingendera kastar förstås MBean ut ett felmeddelande som tas om hand av systemet. Ändringar i konfigurationen sker genom JMX-konsolen och dess webbgränssnitt.



Figur 9 - JMX-konsolens webbgränssnitt

Del två i konfigureringshanteringen är att se till att EPP-protokollimplementationerna har rätt användarnamn, lösenord, SSL-egenskaper, certifikat med mera för det *registry* den skall prata med. Lösningen blev att varje implementation fick varsin MBean där de aktuella attributen lagrades. Då något i konfigurationen behöver förändras görs detta enkelt via *JMX-konsolen* (se figur 9). Rent arkitekturmässigt baseras EPP-implementationerna på en arvsstruktur med en abstrakt superklass, en konkret implementation per protokollversion, samt i botten *registry*-specifika implementationer i de fall det behövs. På detta vis är det enkelt att lägga till nya implementationer då nuvarande och nya toppdomäner går över till EPP från RRP och e-post eftersom den allra mesta koden återanvänds med hjälp av arvet.

4.5.3 Backup av konfigureringsdata med MBeans

En punkt som vi ännu inte implementerat är backup av konfigureringsdata i MBeans. Vad händer om systemet kraschar och JBoss behöver startas om? I dagsläget försvinner all data som lagrats. Tre exempel på lösningar:

1. MBeans kan konfigureras av den information som finns specificerad i dess konstruktor. Så man kan hårdkoda en standardkonfiguration i en MBean i värsta fall, men detta är att betrakta som en mycket ”ful” lösning.
2. Lagra konfigurationerna i någon form av dokument, exempelvis i en text- eller XML-fil som bönan läser vid uppstart. Sedan låter man de specifika MBeans med jämna mellanrum skriva ner aktuell konfiguration till den aktuella filen så det då finns en fräsch backup att tillgå vid omstart. Dessutom bör man då centralt i systemet ha en funktion som man kan ropa på som tar en ögonblicksbild av samtliga konfigurationsdata som finns aktiv i systemet.
3. Man kan låta Entitetsbönor hålla konfigureringsinformationen och låta JBoss skriva ner data i den underliggande databasen med jämna mellanrum.

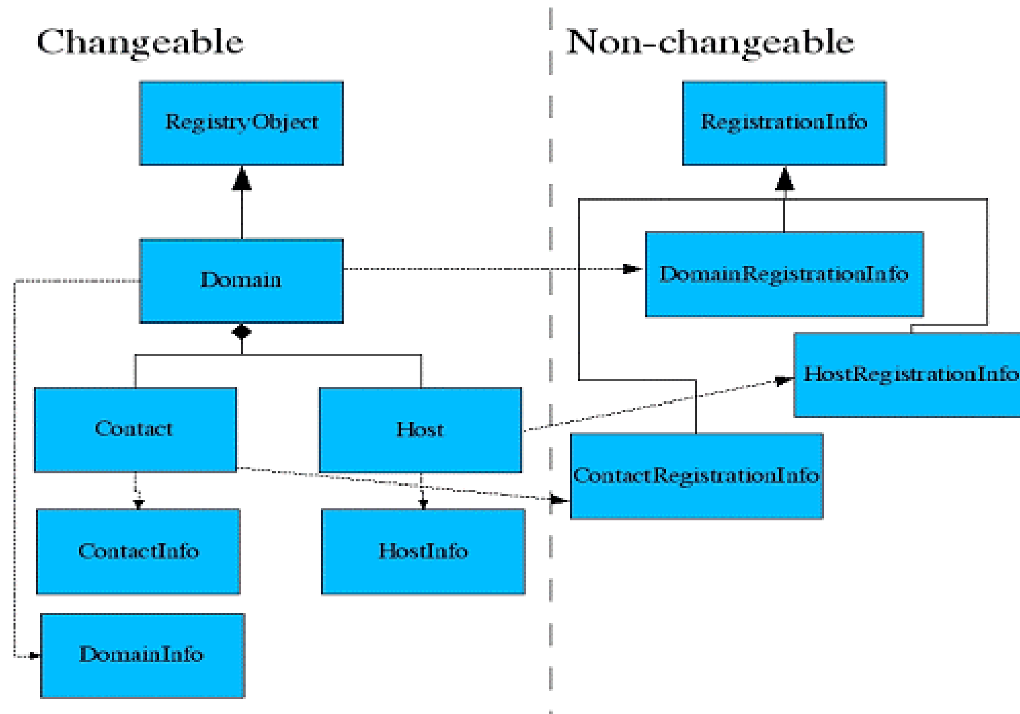
4.5.4 Skriptbarhet direkt mot RegistryHandler-implementationer

Som man ser i figur 7 (*RegistryHandler*) finns det en direkt väg in i systemet via *RegistryHandler*-gränssnittet. Eftersom alla EPP-metoder som är implementerade i *RegistryHandlerBean* är publika metoder finns alltså möjligheten att offentliggöra dessa som *Web-services*. Detta innebär i sin tur att en SOAP-kompatibel klient på så vis kan utnyttja denna väg in i systemet för att utföra specialiserade användningsfall som inte finns definierade i växeln. (Som i dagsläget bara stödjer rena registreringar). Orsaken till denna ”bakväg” in i systemet finns beskriven i Idealbild, men beror i kort på att specialkommandon mot EPP-*registries* idag måste ske via ett kommandogränssnitt (”eppic”) som helt saknar stöd för skriptning.

Denna del är i prototypen inte implementerad i den bemärkelsen att metoderna inte finns publicerade via Axis som *Web-services*, att någon form av proxy behöver skrivas även här för att ta hand om arbiträra objekt samt att se till att rätt registry refereras. Det finns ett par olika möjliga lösningar på det sista problemet.

- Den ena lösningen är att i varje kommando skicka med JNDI-namnet för det *registry* man vill jobba mot vilket är enkelt implementationsmässigt. Samtidigt kräver det dock att skriptprogrammeraren känner till JNDI-namnen vid kodning eller att klienten på något vis tar reda på dem vid körning.
- Den andra lösningen är att återanvända växelkomponenten och låta den avgöra vilken RegistryHandler-implementation den skall jobba mot.

4.5.5 Design av dataklasser



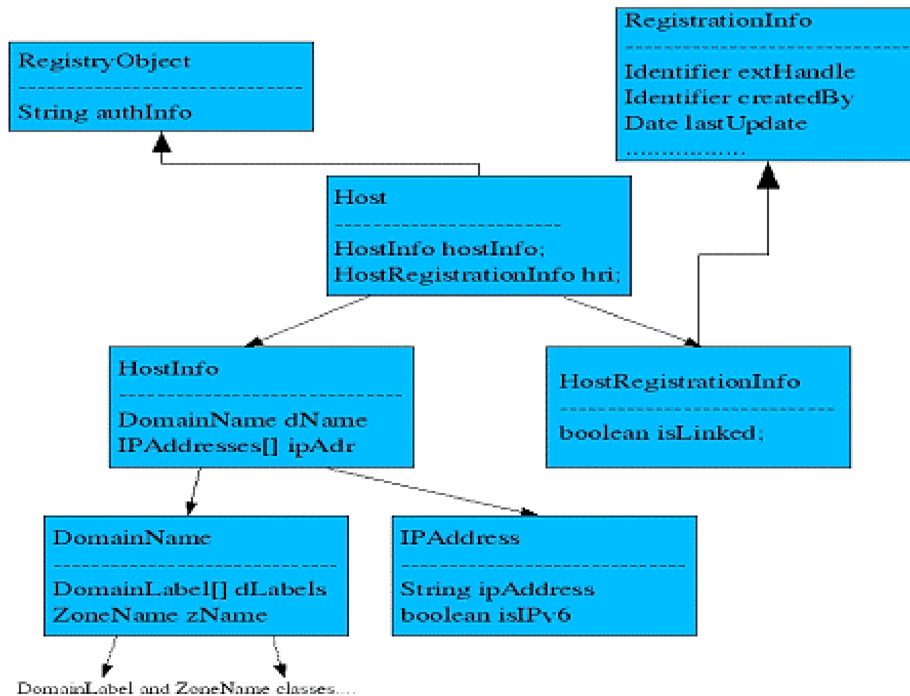
Figur 10 - Klassdiagram över dataklasserna

Vi valde att utforma våra dataklasser med hänsyn till vem som har rättighet att skapa dem.

- För de klasser vars data inte kan eller skall definieras av system under vår kontroll. Då data som returneras när man skapar en post i ett registry så använder vi exempelvis *Factory*- och *Builder*-mönstren där specifika funktioner har ensamrätt att skapa instanser av ”Non-changeable” klasser (se figur 10). Exempel på attribut som kapslas in i dessa klasser är nycklar (extHandles), datum och vissa typer av flaggor. Dessa klassers attribut kan vi endast läsa och de går inte att förändra efter att de väl skapats.
- Den andra typen är sådana objekt som vi själva bygger via den information vi tillhandahåller inifrån våra egna system (*Changeable*). För en typisk registrering finns det exempelvis kontaktinformation, domännamnet, namnservernamn etc. Dessa klasser kan vi instansiera på eget bevåg men givetvis måste vi använda oss av *get*- och *set*-metoder ifall vi behöver förändra något. Vissa klasser tillåter dessutom bara oss att läsa dem då de väl instansierats.

Detta förenklade men ändå något röriga klassdiagram visar hur de föränderliga respektive oföränderliga klasserna samtliga härstammar från varsin superklass som innehåller vissa gemensamma attribut. *Domain*, *Contact* och *Host*-klasserna är egentligen bara aggregathållare för sina respektive *Info* och *RegistrationInfo*-objekt.

I närmare detalj visar vi i figur 11 hur mönstret med sammansatta objekt ser ut för *Host*-klassen:



Figur 11 - Detaljerat klassdiagram för *Host*-klasserna

Här ser vi tydligare att *Host*-objektet egentligen bara är aggregathållare för *HostInfo*, som håller den användardefinierade data, medan *HostRegistrationInfo* håller *registry*-definierad data.

DomainName- och *IPAddress*-klasserna visar hur vår objekthierarki ser ut där vi ända ner till primitiver i största möjliga mån kapslar in vanliga datatyper i klasser eftersom vi då kan valideringsmekanismer i konstruktorerna så instansiering av felaktig data ej blir möjlig. Exempel på sådana klasser är:

- *Identifier* – Kapslar in en String, men validerar enligt de regler som gäller för lösenord, användarnamn och dylikt i ett EPP-*registry*.
- *RegistryPhoneNumber* – Kapslar in en String, men med validering för telefonnummersreglerna EPP-protokollet specificerar.
- *EmailAddress* – Kapslar in en String, som är validerad som en generellt syntaxiskt korrekt e-postadress.

4.6 Procedur

I resultatavsnittet redovisas experimenten och dess resultat samt hur vi utifrån dessa resultat gått vidare, en slags procedur sammanslagen med resultat. Detta gör vi för att vi känner att det blir enklare att följa arbetsgången och de beslut som togs i en löpande text snarare än med separat uppspaltade experimentbeskrivningar och resultatredovisningar. Sist i kapitlet kommer vi ha en resultatanalys där vi sammanfattar resultaten kopplade mot de krav och kriterier vi kommit fram till.

Vi har i möjligaste mån försökt använda oss av litteraturen när vi utvärderat teknik och vad som skulle kunna fungera eller inte. Men då stora delar av de plattformsoberoende tekniker vi valt att använda oss av är i sin linda eller har avgiftsbelagd dokumentation har vi blivit tvungna att utföra experiment av funktionalitet på en ibland lite väl enkel nivå.

Dessa experiment är sammanställda i en experimentlista för att man lättare skall kunna förstå och följa vilka tester av de olika teknikerna som har genomförts. Experimenten var nödvändiga för att förstå de olika teknikerna och hur de fungerade tillsammans och gav oss en grund att stå på när vi byggde vår prototyp.

4.6.1 Testlista

JBoss

- Driftsättning av JBoss – Versionsutprovning. Version 3.0.4 mot 4.0 Beta. Vilken fungerar och passar mot de behov vår prototyp har?

Web-services, Axis och SOAP

- Driftsättning av JBoss.Net under JBoss 3.0.4 och JBoss 4.0 Beta.
- Driftsättande och test av Axis under en fristående Apache Tomcat – Klienter: Java, PHP, VBScript och Perl.
- Driftsättande av Axis som en komponent i JBoss 3.0.4 – Klienter: Java, Perl, VBScript och PHP.
- Test av EJB-åtkomst via Axis - Klienter: Java, Perl, VBScript och PHP.
- Test av arbiträra datatyper via Axis – Klienter: Java, Perl och VBScript.
- Test av felhantering via Axis – Klienter: Java, Perl och VBScript

JMX, MBeans och konfigurering

- Test av grundläggande MBeans-funktionalitet för konfigureringsändamål.
- Test av MBean-prestanda vid upprepade anrop.

Dynamisk klassladdning

- Test med dynamisk klassladdning i en fristående Java-applikation.
- Test av dynamisk klassladdning, separerade EPP-implementationer i Enterprise-arkiv, driftsatt i JBoss.
- Test av dynamisk klassladdning inifrån i en sessionsböna, driftsatt i JBoss.

Övrigt

- Test av ”vanliga” connections och SSL-connections inifrån JBoss.

4.7 Resultat: funktionstester, teknik- och prototyputvärdering

Detta avsnitt beskriver de funktionstester samt de teknik- och prototyputvärderingar vi genomfört samtidigt som det beskriver proceduren hur vi gick tillväga. Det är alltså resultatet av dels funktionstesterna och fallstudien som beskrivs här.

4.7.1 Överväganden kring val av teknisk arkitektur.

I detta avsnitt beskriver vi de tankeprocesser som lett oss fram till vårt val av teknisk arkitektur.

4.7.1.1 Språk- och systemmiljö

Eftersom vi strävade efter att använda oss av gratis mjukvara bestämde vi oss för att köra Linux som operativsystem på serversidan, dels för att Linux är gratis och dels för dess väldokumenterade stabilitet och lämplighet i servermiljöer. Vi valde också att köra Linux på våra utvecklingsdatorer. Vi valde distributionen Mandrake 9.0.

För vår prototyp behövde vi en robust språklösning med bra kommunikationsmöjligheter både upp och ned i systemmodellen. En Java-lösning uppfyllde våra krav och önskemål väl. Att företagets befintliga lösningar gällande EPP-hantering redan var skrivna i Java, samt att EPP-protokollet är implementerat i Java, såg vi som positivt med tanke på återanvändning av komponenter.

Det finns ett otal olika varianter av applikationsservrar på marknaden men på grund av företagets önskan använde vi JBoss, som finns närmare beskriven i teoriavsnittet. Vi hade viss erfarenhet från arbete med JBoss sedan tidigare vilket gjorde att kunskapströskeln blev något lägre, läs mer om JBoss inlärningskurva i diskussionen.

4.7.1.2 Persistent datalagring

Trots att persistent datalagring i form av en underliggande relationsdatabas inte ingick i som krav eller önskemål i våra kriterier ville vi ändå se till att få in någon form av datalagring i systemet[6]. Det eftersom vi inte vill begränsa systemets skalbarhet och utvecklingsmöjligheter i framtiden.

Som vi redan beskrivit i teoridelen används entitetsbönor normalt i JBoss för persistent datalagring, men i botten har man fortfarande en normal relationsdatabas. JBoss levereras med en databas som standard - HyperSonicSQL[7]. Den verkar vara ändamålsenlig men vi undersökte även kort alternativa databaslösningar eftersom det är mycket enkelt att byta ut den underliggande databasen i JBoss. Med vår avgränsning att mjukvaran skulle vara kostnadsfri så sållade vi bort kommersiella Microsoft- och Oraclelösningar och tittade på gratisalternativen.

Ett potentiellt alternativ var att utforma någon form av egen databaslösning, men det kände vi låg helt utanför ramen för vår fallstudie. Vi konstaterade att någon av de vanligaste och bäst dokumenterade kostnadsfria SQL-databaserna, som exempelvis

[6] Loggning var dock ett krav, men skall inte förväxlas med databasskiktet.
[7] Läs mer om HyperSonicSQL på: http://hsql.sourceforge.net/

MySQL[8] eller PostgreSQL[9] förmodligen var vettiga alternativ. Dock så valde vi att tills vidare använda HyperSonicSQL.

4.7.1.3 Externt protokollval

Nästa steg var att bestämma oss för hur servern skall kommunicera med omvärlden. Dels skulle systemet kommunicera med domännamnsservrar med hjälp av EPP. Denna funktionalitet, inklusive SSL-*sockets* som används för EPP-kommunikation, fanns redan implementerad i Java så den biten var inget bekymmer. Utmaningen var att hitta ett protokoll som uppfyllde kraven på interoperabilitet, dvs. plattformsoberoende, mellan plattformar och maskininteraktion, samt till viss del prestanda och säkerhet.

Remote Method Invocation (RMI) [60] är ett protokoll som ger god distribution och god säkerhet men som dessvärre endast fungerar mellan Java-plattformar. En variant av RMI, RMI-IIOP (Inter Orb Protocol) [61] är CORBA-kompatibel (Common Object Request Broker Architecture) [62] och som med hjälp av ett gemensamt specificeringsspråk, Interface Definition Language (IDL)[10], fungerar mellan olika plattformar. Erfarenhetsmässigt visste vi dock sedan tidigare att CORBA är komplicerat att arbeta med och dessutom är det en något gammal teknik. Vi hade däremot hört talas om *Web-services* som är ett relativt ett nytt begrepp. *Web-services* är ett samlingsnamn för ett antal nya tekniker för kommunikation och interaktion mellan maskiner som verkade mycket lovande sett till våra behov. Därför valde vi att studera om det kunde användas inom vår forskning.

4.7.1.4 Val av SOAP-motor

Web-services med dess XML-baserade arkitektur och interoperabilitet såg mycket lovande ut. En av beståndsdelarna i *Web-services* är protokollet SOAP vilket verkade vara utmärkt för kommunikation med klienter skrivna på olika plattformar. Detta eftersom data i en SOAP-transaktion packas ihop i en XML-struktur som skickas via något av standardinternetprotokollen, exempelvis HTTP. Vanlig text som går via HTTP klarar i stort sett alla plattformar av. Problemet ligger i att på bägge sidor skapa/tolka SOAP-meddelandena med någon form av färdiga komponenter och XML-tolkar. Det var här SOAP-motorn Axis kom in.

Eftersom vi valt applikationsservern JBoss fanns det fyra huvudsakliga vägar att gå gällande SOAP-hanteringen.

- Den mest färdiga och integrerade lösningen var att använda modulen *JBoss.Net* som fullt ut skall integrera Axis med JBoss i en och samma *container*.
- Alternativ två var att använda Axis som en fristående komponent driftsatt i JBoss. Dokumentation över denna process fanns dock inte tillgänglig genom officiella källor eller Google-sökningar. Detta beror förmodligen på att Axis är utformat för att köras på en fristående Tomcat[11].

[8] Läs mer om MySQL på: http://www.mysql.com
[9] Läs mer om PostgreSQL på: http://www.postgresql.org/
[10] Se t ex http://java.sun.com/docs/books/tutorial/idl/intro/intro.html.
[11] Tomcat är en s.k. Servletmotor som hanterar Java-program gentemot webben

- Det tredje alternativet var att driftsätta Axis under en fristående Tomcat, så som det är tänkt från början, och sedan låta Axis referera till vår applikationsserver därifrån.
- Det fjärde alternativet var att själva implementera användandet av XML-tolkar, exempelvis Xerces[12] och skriva egna objektserialiserare, deserialiserare m.m. i en egen SOAP-hanterare. Det alternativet var aldrig realistiskt i fallstudiekontexten. Dock så är det aktuellt i en viss omfattning i de fall vi har problem med SOAP på klientplattformar som ännu inte stödjer den SOAP-funktionalitet vi behöver, exempelvis hantering av arbiträra objekt och grafstrukturer. Den problematiken beskrivs utförligare på annan plats i uppsatsen

4.7.2 Test: JBoss och konfigurering

4.7.2.1 Driftsättande av olika JBoss-versioner

Våra två testversioner av JBoss var 3.0.4 och 4.0 Beta. JBoss 3.0.4 var enligt källor på nätet (foruminlägg, dokumentationsmängd etc.) stabil och lämplig för skarp drift. Vi installerade den på vår Linux-baserade server med gott resultat. Vi byggde några enkla testfall kring sessions- och entitetsbönor vilket fungerade fint.

JBoss.4.0 beta var den senaste version av JBoss och ett ytligt driftstest fick den genomgå. Den startade som den skulle på vår tekniska plattform, men betedde sig underligt då vi försökte utföra lite mer avancerade test. Vi hade en uppsättning sessions- och enitetsbönor vi försökte testköra men vi fick stora problem att driftsätta dem. Det beror förmodligen på att våra testbönor och dess beskrivande xml-filer var baserade på hur vi utformat de testbönor vi använt för version 3.0.4. Enligt Ola Berg på Ports [3] så är dessutom JBoss 4.0 ännu inte moget för skarp drift i en affärsverksamhet, så vi valde att inte fortsätta våra tester kring JBoss 4.0 Beta.

Eftersom vi bara hade ett praktiskt val kvar så valde vi förstås version 3.0.4. Stabilitet och mogenhet var krav på vår prototyp vilket fick överväga de fina finesserna som finns i JBoss 4.0.[13]

4.7.3 Test: Web-services

4.7.3.1 Driftsättande av JBoss.Net

Vi installerade och konfigurerade JBoss.Net i vår JBoss-miljö men varken under JBoss 3.0.4 eller JBoss 4.0 beta lyckades vi få den att fungera korrekt. Efterforskningar visade dessutom att JBoss.Net vid den tidpunkten inte var mogen för skarp drift i en affärsmiljö [3]. Därför var JBoss.NET tyvärr inte ett alternativ för oss. I framtiden är det förstås intressant att få den närmare integration mellan Axis och JBoss som JBoss.Net erbjuder.

[12] XML-Xerces, en XML-tolk
[13] Läs mer om JBoss 4.0 på http://www.JBoss.org

4.7.3.2 Driftsättande av Axis i en fristående Tomcat

Nästa steg var då att försöka driftsätta Axis 1.0. enligt standardförfarandet på en fristående Tomcat som finns dokumenterat på Axis hemsida [63] och sedan utföra enkla tester med SOAP-klienter. Vi testade med följande klientplattformar:

(*Dessa fyra plattformar gäller för samtliga test vi utfört i detta avsnitt*)

- Java, med delar av Axis som SOAP-motor.
- VBScript, med MSSOAP Version 3.0 som SOAP-motor
- PHP, med PHP SOAP Toolkit[14] som SOAP-motor
- Perl, med modulen SOAP::Lite[15] som SOAP-motor

Vi skapade s.k. *SOAP-services* på serversidan som vi anropade från våra olika klienter. Detta fungerade fint från samtliga prövade klientplattformar. Observera att dessa tester endast använde primitiva typer som parametrar och returtyper.

Nästa steg i detta experiment var att knyta samman Axis med *EJB:er* som var driftsatta under JBoss på en annan maskin. Tyvärr lyckades vi inte få denna kommunikation att fungera, dvs. referera till en *Enterprise*-böna (EJB) från en annan maskin trots att vi noga följde specifikationerna för hur man skall göra med hjälp av Java Naming och Directory Interface (JNDI)[16]. Vi ägnade en hel del tid åt både felsökning och att leta information om problemet genom olika informationskällor på Internet, men fann tyvärr inga svar.

4.7.3.3 Driftsättande av Axis som en komponent i JBoss

Istället så gav vi oss på att få in Axis i samma Java Virtual Machine (JVM) som JBoss. Efter en del experimenterande med klassbibliotek och inställningar lyckades vi framgångsrikt paketera Axis i ett WAR (Web Archive)-arkiv tillsammans med Xerces-XML och driftsätta det i JBoss. Detta uppnås genom att lägga komponenten Axis.war i samma *deployment*-mapp som våra vanliga *Enterprise*-bönor. Observera alltså att man måste ladda hem en XML-tolk och lägga den i Axis ”lib”-katalog för att det skall fungera. Vi valde att använda Apache-Xerces 2.3.0[17].

4.7.3.4 EJB-åtkomst via Axis

Vi genomförde ett antal tester av EJB-åtkomst via Axis där vi gick från enklast möjliga ”Hello World” till mer avancerade tester där vi skickade standardtyper från Java, VBScript, PHP och Perl-klienter till en sessionsböna. Sessionsbönan ligger ju internt i JBoss, endast åtkomlig genom sina *remote* och *local interface*, så vi hade bevisat att man direkt från Axis kunde offentliggöra en sessionsbönas publika metoder som *Web-services*. Vårt nästa test var att från klienten nå till en sessionsböna som vidarebefordrade kommandon till en entitetsböna och sedan få entitetsbönans

[14] Läs mer om PHP SOAP toolkit på http://sourceforge.net/projects/phpsoaptoolkit
[15] Se t ex http://www.soaplite.com/
[16] http://java.sun.com/products/jndi/tutorial/
[17] XML-Xerces, en XML-tolk

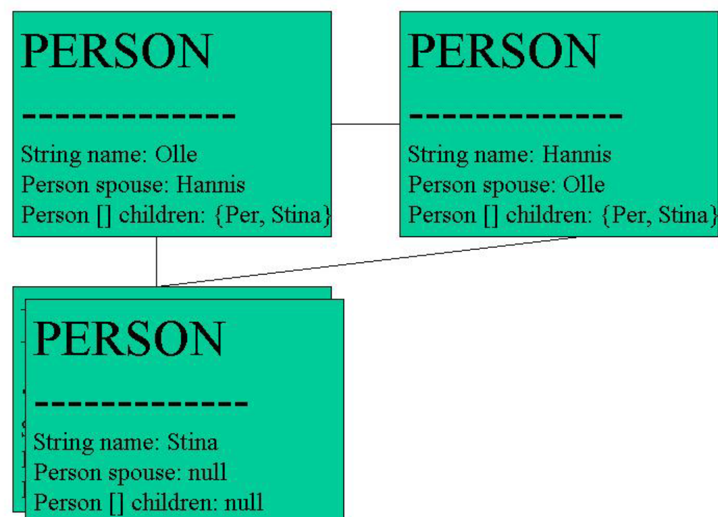
information hela vägen tillbaka till klienten. Samtliga dessa test fungerade väl. När man väl fått kommunikationen mellan sin EJB och klient att fungera med SOAP så kan man egentligen helt abstrahera bort vad som händer internt i Enterprisebönan eller klienten i SOAP-kontexten.

Första milstolpen var avklarad - vi hade lyckats med hela kommunikationskedjan från *View*-lagret till *Model*-lagret via *Control*-lagret i MVC. Vår ansats att använda en J2EE-applikationsserver tillsammans med Web Services fungerade rent tekniskt för att implementera en MVC-arkitektur.

4.7.3.5 Arbiträra datatyper och SOAP

Nästa viktiga del var att kunna skicka arbiträra objekt innehållandes diverse datastrukturer via SOAP. Tidigare hade vi bara använt de fördefinierade typerna (int, float, String etc.) som finns i SOAP's standardutbud. Nu ville vi få både klient och server att acceptera och korrekt serialisera och deserialisera arbiträra datastrukturer. Ett krav på dessa arbiträra objekt är att de måste följa JavaBean-mönstret med en tom konstruktor samt implementera gränssnittet *Serializable*.

Vårt testfall var ett huvudobjekt Person (se figur 12) vars attribut hade ett namn i strängformat, samt ett annat Person-objekt, "spouse" och en array av Person-objekt, "children".



Figur 12 – Vår testfamilj: Paret Olle och Hannis med sina två barn Per och Stina.

Vi lät vår sessionsböna bygga ihop objektgrafen och returnera dessa data via SOAP. Axis fungerar som så att den på serversidan tolkar de instruktioner som finns i konfigurationsfilen server-config.wsdd för varje enskild *Web-service* och skapar XML-kod för de funktioner och typer som skall skickas. Den klarar till och med av att, efter erhållit vissa instruktioner, skapa XML-kod för en s.k. "complexType" där den i tjänstens automatgenererade wsdl-dokument talar om för omvärlden hur vårt arbiträra Person-objekt ser ut. Om följande punkter uppfylls fungerar Person-exemplet ovan väl.

- Klienten skall vara Javabaserad – Det är endast med Java vi har lyckats deserialisera en komplex datatyp, se mer nedan.
- Peka ut wsdl-dokumentet istället för tjänstens namn i klienten – klienten måste gå via wsdl-dokumentet där den komplexa datatypen beskrivs vid anropet. Att referera till tjänstens namn och URL i klienten går inte i detta fall.
- Den kompilerade Javafilen – Person.class i vårt exempel – måste vara tillgänglig på klientsidan .
- Explicit ange för klienten vilken deserialiserare den skall använda för Person-objektet. Det fungerar så elegant att deserialiseraren endast behöver kunna se bytekoden för att förstå hur den skall bygga upp vårt arbiträra objekt igen.

Tyvärr fungerade det inte lika bra med Perl. Visserligen överfördes fortfarande data utan problem från vår sessionsböna till Perl via Axis, men Perl klarade inte av att *deserialisera* vårt komplexa objekt igen utan returnerade bara en meningslös referens. Enligt SOAP::Lite:s hemsida [64] så stöds ännu inte komplexa objekt vilket givetvis förklarar saken. Detta är ett problem då det finns Perl-skript internt på företaget som kan behöva integreras längre fram.

Med .ASP, VBScript och PHP har utvecklare på företaget genomfört några enklare försök där olika beteenden uppvisats. För .ASP och VBScript fungerar det men är besvärligt eftersom en *ActiveX-kontroll* måste skrivas och registreras på serversidan [3]. PHP klarar däremot av komplexa typer. ASP.NET samt C# har fint stöd för SOAP och komplexa typer [3], men dessa språk har vi ej undersökt inom ramen för vår fallstudie.

Man skall dock vara medveten om att det alltid finns möjligheten att själv definiera serialiserare och deserialiserare för varje enskilt klientspråk för att sedan på serversidan deserialisera och skapa objekten i en sessionsböna. Därefter kan man utan problem behandla dem som vanliga objekt internt i applikationsservern och mot utsidan så länge man håller sig till Java och RMI eller pratar SOAP mot andra applikationer som klarar av att hantera arbiträra objekt via SOAP.

4.7.3.6 Felhantering med Axis

Något som är viktigt i systemarkitekturen är hur fel hanteras. Fel kan uppstå redan i klienten i form av inkorrekt indata. En servers publika metoder skall alltid validera sin egen indata och således måste den också kunna returnera vettiga felmeddelanden tillbaka till klienten. Vi designade alla dataklasser i vårt system så de kapslar in sina primitiva typer. På så vis kan vi, exempelvis för en e-postadress, validera att det är en syntaxmässigt korrekt e-postadress innan den instansieras i vårt e-post-objekt som en textsträng. Om det blir något fel här så har vi egendefinierade *exceptions* som kommer bubbla ut och kastas tillbaka hela vägen till klienten.

Eftersom det är ett krav i systemet att felhantering skall kunna ske inifrån servern och tillbaka till klienterna var det nödvändigt att pröva så denna funktionalitet fungerade korrekt mellan JBoss och tillbaka till olika plattformar via Axis. Alltså: Inget fusk med att låta metoderna returnera strängar och inget fusk med att ta till andra kommunikationsmedel än den aktuella SOAP-sessionen.

Vårt testskott var enklast möjliga. En ny metod ”throwException” definierades i sessionsbönan. I denna metod definierades ett java.lang.Exception med en meddelandesträng som sedan kastades. I detta fall prövade vi tre olika klientplattformar: Fristående Java-applikation, Perl-skript och Visual Basic-skript.

I samtliga fall kunde vi få ut det korrekta meddelandet genom att läsa från antingen SOAP-meddelandet eller respektive språks SOAP-felhantering. Vi kunde på så vis enkelt få fram dels typen av *Exception* – dvs. java.lang.Exception eller om vi så ville ett java.io.IOException – och dels kunde vi helt själva definiera felmeddelandet. I förlängningen innebar detta att vi på klientsidan kunde ta emot felmeddelanden som följde ett visst mönster och tala om för användaren vad som gått snett. Dessutom ger välspecificerade felmeddelanden möjlighet till viss automatisk felhantering på klientsidan vilket kan vara värdefullt ur human-computer interaction (HCI)-synpunkt[18].

Om detta inte kunnat lösas hade det endast varit möjligt att skicka tillbaka odefinierade fel till klienterna vilket inte hade varit ett fullgott alternativ.

4.7.4 Test: JMX, Mbeans och konfigurering

4.7.4.1 Test av grundläggande Mbeans-funktionalitet

För att kunna hantera olika konfigureringsmodeller för olika registreringsförfaranden på ett smidigt sätt valde vi att använda Java Management eXtensions (JMX) med sina Management Beans, Mbeans.

Vi utförde ett enkelt experiment där en standard Mbean med ett sträng-attribut anropades från en privat metod inne i en sessionsböna, dvs. att man fick anropa en publik metod i sessionsbönan först för att komma åt den privata metoden. Vi gjorde på detta vis eftersom vi i vår prototyp behövde pröva just användningsfallet att hämta data till en sessionsböna.

Vi skrev *get*- och *set*-metoder för själva attributet samt en publik ”getConfig()”-metod som skickade attributets värde till valfri anropande klient. Vi lade sedan till en Hashtabell som lagrade data och skrev metoder som arbetade mot Hashtabellens interna datastrukturer.

4.7.4.2 Test av MBean-prestanda

Något som var viktigt i detta sammanhang var att åtkomsten och uppdateringen inte fick ha för hög nätverksfördröjning på grund av två punkter:

- Osynkad konfigureringsdata. Sådana problem kan i normala fall undvikas med hjälp av synkroniseringsfunktionaliteten som finns inbyggd i Java men eftersom JBoss med sin CMP har hand om sådant själv så var det säkrast att tillse att det inte rörde sig om några höga svarstider.
- Vid fall där många atomära kommandon kan ske i serie, exempelvis vid en skriptad serie omregistreringar av domäner, så är det önskvärt att hålla nere svarstiderna.

[18] Läs mer om HCI, t ex Christine Faulkner: “The Essence of Human-Computer Interaction”.

Vi tog en Mbean från det förgående testet och lade till en enkel tidtagning baserad på Javas ”System.currentTimeMillis()” . Vi byggde samman en fräsch instans version av denna Mbean och driftsatte den för att utföra fyra olika test. Vår utformning av testfallen härstammar ur de möjliga situationer vår Mbean kan ställas inför då den skall konfigurera någon av de komponenter som ingår i vår prototyp.

Gemensamt för samtliga test var följande procedur där tiden som mättes var från precis före punkt 1 till precis efter punkt 3:

1. MbeanServer-lista plockades fram.
2. Vår Mbean hämtades med hjälp av dess JBoss:jmx-namn från MbeanServer-listan.
3. Vår Mbean höll ett sträng-attribut vars värde returnerades.

Test nr	Testbeskrivning	Resultat (i millisekunder)	Utvärdering
1	Enligt standarmallen. Snabb följd.	1 – 1 – 0 – 0 – 1 – 0 – 1 – 0	Tillfredsställande prestanda
2	Enligt standardmallen, men fyra stycken delresultat skrevs till skärment. Observera att test sex och åtta kom efter ca 10 minuters ”vila”.	42 – 5 – 6 – 5 – 5 – 11 – 5 – 10	Skärm IO drar ner prestandan kraftigt. Första anropet samt anrop med viss tid mellan försämrar prestandan.
3	Enligt standardmallen, men med nytt värde i teststrängen. Snabb följd.	0 – 1 – 0 – 1 –1 – 0 – 1 – 0	Attributförändringar påverkar ej prestandan.
4	Enligt standardmallen, 10 minuter ”vila” mellan iterationerna	1 – 1 – 0 – 1 – 0 – 1 – 0 – 0	Utan skärmutskrift så påverkades inte prestandan av tid som i test nr 2.

Figur 13 – Testresultat av MBean-prestanda

Som man ser i figur 13 så var nästintill all fördröjning orsakad av skärm-IO. Java verkar ha dålig skärm-IO-prestanda. Åtskilliga upprepade operationer mot MBeans gav negligerbar prestandaförlust. Vi drog slutsatsen att MBean-prestanda inte var ett område som skulle vålla oss några bekymmer då vi inte har någon som helst skärm-IO i den delen av systemet.

4.7.5 Test: Dynamisk klassladdning

Växeln i prototypen syftar till att låta det centraliserade systemet välja registreringsförfarande transparent oavsett vilken toppdomän registreringen sker under. Vi koncentrerade oss på EPP-protokollet som används i ett flertal versioner paketerade i varsitt ”EPP.jar”-arkiv[19]. Eftersom klasserna inne i dessa arkiv är namngivna likadant

[19] I den senaste binära releasen av EPP från dess upphovsmakare är samtliga EPP-versioner för första gånger paketerade i samma .jar.arkiv. Versionerna skiljs åt via olika paketnamn för respektive version.

oavsett version så går det inte att ladda in dem vid kompilering utan de måste laddas då de behövs under körning, så kallad *dynamisk klassladdning*..

För att få in växelarkitekturen i en EJB-lösning valde vi att studera två olika lösningar:

(O*bservera att nedanstående avsnitt ang. dynamisk klassladdning är ganska tekniskt till sin natur och inte alldeles lättläst.)*

4.7.5.1 Test med dynamisk klassladdning i en fristående Java-applikation.

Vi satte samman en testapplikation för att pröva dynamisk klassladdning i en kontext helt skild från J2EE och JBoss. Först gjorde vi testprogrammet helt skild från dessa för att verifiera att vår ansats fungerade.

Vi satte samman ett javapaket och lät sedan en Java-applikation ladda in olika CLASS-filer från en separat JAR-fil, vilket fungerade utmärkt. Dessa behöver inte finnas med i *import*-satser eller *classpath* vid varken kompilering eller run-time utan sådant löses under körning av JVM. Dock så måste givetvis gränssnitten på serversidan korrekt definiera de metoder i klasserna som skall vara åtkomliga.

4.7.5.2 Separerade EPP-implementationer i Enterprise-arkiv

Den första eventuellt möjliga lösningen på vårt problem i prototypen var att behandla varje domän i en egen kontext inom applikationsservern. I detta fall bakade man samman den centrala sessionsbönan med den aktuella protokollimplementationen i ett Enterprise-arkiv (.ear) för varje registreringsförfarande.

```
-Enterprise-arkiv FOO_info.ear
|--- Foo.jar (Sessionsböna m. interface)
|--- EPP.jar (version 4 t.ex.)
|--- MBean_info.jar (Konfigureringsböna)


-Enterprise-arkiv FOO_biz.ear
|--- Foo.jar (Sessionsböna m. interface)
|--- EPP.jar (version 5 t.ex.)
|--- MBean_biz.jar (Konfigureringsböna)
```

Figur 14 – Exempel på Enterprise-arkiv för olika toppdomäner, ”info” och ”biz”

Den enda skillnaden i dessa arkiv var att de olika EPP.jar -filerna hade olika protokollimplementationer, dock i likadant namngivna klasser (se figur 14). Dessa skulle särskiljas genom att metodanropen till EPP.jar var olika i sessionsbönan (Inne i Foo.jar). På så vis skulle man, exempelvis i *SOAP-proxyn*, kunna som en parameter i sessionsanropet eller via olika *JNDI*-namn, definiera vilket av EAR-arkiven som skulle anropas.

Problemet var dock om JBoss klarade av att hantera JAR-filer med samma namn i samma EJB-*container*. Givetvis krävdes det att de olika *Enterprise*-arkiven och dess

Detta innbär att de tester vi redovisar angående klassladdningen inte är relevanta för just problematiken med likadant namngivna klasser eftersom paketnamnen nu skiljer dem åt.

sessionsbönor hade olika *JNDI*-namn, men EPP.jar, som är en tredjepartsmodul måste kunna användas i olika versioner inom samma kontext. Vi byggde ihop en testapplikation för detta fallet.

Att *deploya* (driftsätta) de två ovan beskrivna *enterprise*-arkiven gick fint, utan felmeddelanden, så länge det inte fanns några *JNDI*-namns konflikter. Att anropa den först driftsatta ear-filen gick fint med korrekt och förväntat resultat, men när vi sedan anropade *Enterprise*-arkiv (EAR) nummer två så använde den sig tyvärr av det först laddade EAR-arkivets EPP.jar-fil. Detta innebär att exakt detta angreppssätt inte kunde realiseras utan modifieringar.

Problemet beror på att JBoss laddar klasserna i Foo.jar dynamiskt när de anropas vilket är vackert i sig, men sedan så låser sig JBoss vid just den nu inladdade Foo.jar och när nästa anrop kom till den andra Foo.jar, dvs. den i den andra .ear komponenten - då laddar JBoss inte om den nya JAR-filen ur det andra EAR-arkivet utan kör på det som redan finns laddat.

Detta höll på att omintetgöra detta angreppssätt till den dynamiska klassladdning vi behövde för vår växelarkitektur. Som tur var fanns det en lösning på problemet. Genom att definiera separata klassladdare (*Classloader repositories* för att vara exakt) för varje EAR-implementation, så skiljer vi våra olika EPP.jar och Foo.jar från varandras klassladdare då respektive protokollimplementation skall användas och sålunda är denna arkitekturmodell realiserbar.

På så vis får varje *Enterprise*-arkiv (.ear) sin egen klassladdare och den ovan beskrivna problematiken löstes.

4.7.5.3 Angreppssätt 2: Dynamisk klassladdning i JBoss

Angreppssätt nummer två var att testa dynamisk klassladdning inne i applikationsservern genom att explicit i källkoden tala om när en viss klass skall laddas in.

Vi satte samman en testapplikation för att pröva detta. Först gjorde vi testprogrammet helt skilt från J2EE och applikationsservern för att verifiera att vår ansats fungerade.

Vi satte samman ett javapaket och lät sedan en Java-applikation ladda in olika CLASS-filer från en separat JAR-fil, vilket fungerade utmärkt. Det fina med detta är att man då helt och hållet kan abstrahera bort protokollimplementationen i växeln. Den behöver inte finnas med i *import*-satser eller *classpath* vid varken kompilering eller run-time utan sådant löses under körning av JVM. Dock så måste givetvis gränssnitten på serversidan korrekt definiera de metoder i protokollimplementationen som skall vara åtkomliga. Men samtidigt så är det också en nackdel att *inte* hålla systemets samtliga delkomponenter inom applikationsservern eftersom en del av tänkandet

Nästa steg var att implementera samma lösning som när vi laddade klasserna manuellt (som i första testet) i applikationsservern. För att göra det hela så enkelt som möjligt paketerade vi testprogrammet i ett JAR-arkiv tillsammans med en sessionsböna som offentliggjorde vår testfunktionaliteten. Vi lade JAR-arkivet som innehöll de klasser som dynamiskt skulle laddas på en lokal URL medan själva sessionsbönan med tillhörande klasser *deployades* (driftsattes) i JBoss. Vårt testprogram startades men direkt såg vi ett problem:

Eftersom vi skiljt protokollimplementationen från själva EJB-applikationen behövde vi lägga till samtliga gränssnitt som låg på serversidan i protokollimplementationen. Det innebär en något minskad flexibilitet eftersom det då blir två olika punkter att underhålla – implementationen och servern. Dock så är det bara relevant ifall man behöver förändra gränssnitten – själva implementationerna är fortfarande förändringsbara så länge de följer de definierade gränssnitten.

Vi lade till de gemensamma gränssnitten i protokollimplementationen. Då laddades klasserna in dynamiskt men då infördes nya problem. *ClassCastExceptions*. JBoss har som bekant en egen klassladdare – UnifiedClassLoader3 – som laddar in systemklasser samt ”*import”*-klasser till ett gemensamt s.k. *Repository* för JBoss lokala *namespace*. En klass i Java identifieras av sitt namn och sin klassladdare. Även om det bytekodmässigt är samma klass kan man inte *casta* två klasser till varandra ifall de är laddade av olika klassladdare. Därför får man *ClassCastExceptions.*

Bekymret i vårt fall är att JBoss standardklassladdare UnifiedClassLoader3, som laddat våra *interface* via ”import” - hamnade i konflikt med den av oss specifikt angivna klassladdaren. Den naturliga lösningen skulle då vara att använda vår instans av UnifiedClassLoader3 för vår dynamiska klassladdning, men dessvärre så tillåter JBoss 3.0.0 till 3.0.4 endast en sökväg till sina klasser och denna sökväg (URL) är statisk så fort en klassladdarinstans har initieras. Eftersom vi i detta fall behövde separera våra implementationer från JBoss standard-*namespace* fungerade det inte.

Varför inte förändra sökvägen för UnifiedClassLoader3 till den plats där vår implementation ligger? Klassladdarens klassökväg (URL) ligger som ett *protected field* och det finns inte några *set*-metoder definierade som gör att man under körning kan ändra den sökväg varifrån den laddar klasser. Ganska självklart eftersom Java använder sig av dynamisk bindning vilket innebär att klasser laddas först när de behövs. Om sökvägen för standardklassladdaren förändrats under körning finns det en uppenbar risk att de klasser den tror den skall kunna ladda in inte finns på den sökväg som var giltig då komponenten driftsattes.

Ett angreppssätt är givetvis att subklassa standardklassladdaren och göra dessa förändringar manuellt, men det är ingen bra idé att gå in och pilla på JBoss standardklassladdare. Observera att JBoss 3.0.5 och JBoss 3.2 har en något ombyggd standardklassladdare som godtar flera URL:er i sin standardsökväg, men dessvärre var dokumentationen för denna funktionalitet avgiftsbelagd så vi valde att fortsätta på det inslagna spåret med version 3.0.4.

Vi försökte ta oss runt denna problematik, först genom att ladda gränssnittet med standardklassladdaren[20] och implementationen med olika klassladdare. Dessvärre fick vi då ofrånkomligen *ClassCastExceptions* när vi försökte *casta* implementationen till gränssnittet på grund av den nämnda svårigheten med klassidentifiering på grund av dess klassladdare. Det gick att ladda implementationen och arbeta med den på en specifik nivå - dvs. att gränssnittet förbigicks - genom att använda metoden *getMethod* som genom *reflection* hämtar metoden under körning och därefter kör kommandot *invoke(x,x)* som exekverar den hämtade metoden.

[20] Kallas även: Bootstrap class loader. I detta fall UnifiedClassLoader3

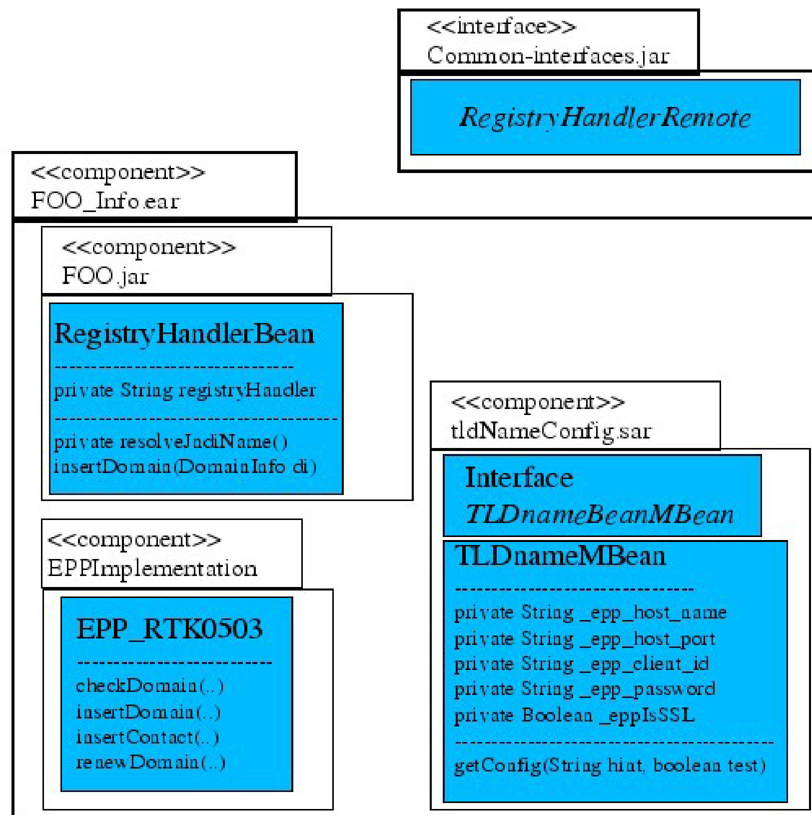
Det var dock inte en giltig lösning eftersom vi då inte använder oss av vårt gemensamma gränssnitt utan vi explicit måste hårdkoda in vilken variant av vår implementation som skall användas. En annan variant var att ladda in både gränssnittet och implementationen dynamiskt under körning, men då uppstod istället problemet att programmet inte kände till gränssnittets utseende vid bytekod-kompileringen. Därmed var det inte praktiskt möjligt att *casta* implementationen till ett visst gränssnitt eftersom JVM inte kände till gränssnittet vid kompileringen.

4.7.5.3.1 Vår lösning på klassladdningsproblematiken

Det finns enligt Ola Berg på Ports vägar att ta sig runt problemen i angreppssätt nummer 2 [3], men vi ville inte investera tid i det eftersom vi kunde använda oss av lösningen med EAR-scopade *namespaces* istället, dvs. angreppssätt 1.

Dock så fanns det förstås svårigheter med att använda sig av EAR (applikationsnivå) specifika (*scopade) namespaces* också. Eftersom vår ursprungliga idé var att paketera sessionsbönorna tillsammans med deras interface i EAR-arkivet som kapslar in våra RegistryHandlers så blev det nu problem när växeln (den ”centrala” sessionsbönan, även känd som *Switchen*) skulle använda sig av protokollimplementationernas *interface* som låg paketerade i EAR-arkivets separata *namespace*.

Klassladdningsproblematiken som uppstår då växeln i förväg inte kan känna till gränssnitten uppstod nu. Om man lät växeln känna till gränssnitten sedan tidigare genom att ha med dem vid kompileringen så blev det istället problem eftersom JBoss använt olika klassladdare för att ladda gränssnitten i växeln respektive implementationen med oundvikliga *ClassCastExceptions* som följd. Lösningen var enkel och faktiskt klart bättre än tidigare ansatser eftersom sessionsbönan Foo.jar faktiskt inte behöver känna till sitt eget gränssnitt:



Figur 15 – FOO_Info.ear med gemensamt gränssnitt

Eftersom alla Foo.jar kan och skall dela ett gemensamt gränssnitt var det enklast att helt enkelt paketera detta gränssnitt i ett eget JAR-arkiv och *deploya* detta i JBoss (se figur 15). På så vis laddar växeln gränssnitten från sitt eget *namespace*. Detta mönster har vi utnyttjat för samtliga klasser som behöver vara universellt åtkomliga i systemet, exempelvis alla dataklasser (*DomainInfo*, *ContactInfo*, *RegistryString* etc.)

4.7.6 Test: Övrigt

4.7.6.1 Åtkomst av externa resurser i JBoss

Eftersom våra EPP-implementationer kommer prata med toppdomännamnsservrarna var det förstås fundamentalt att en sådan kommunikation via vanliga protokoll gick att genomföra inifrån JBoss och att svaret *deserialiserades* korrekt. Ett enkelt test utfördes inifrån en metod i en sessionsböna där en helt fristående sida på en annan webbserver anropades med hjälp av en URLConnection. Testet fungerade felfritt.

De allra flesta aktiva EPP-registries använder sig av specialutformade SSL-kopplingar med egna certifikat och nycklar så vi utförde även ett enkelt test mot toppdomänen .info:s ote-miljö[21] som fungerade utan problem när vi väl fått certifikat och nycklar på plats.

[21] Open Test Enviroment – Flera registries tillhandahåller testmiljöer mot vilka registrarer kan testa sina system utan kostnad och risk för att ställa till det i det skarpa DNS-systemet.

4.8 Resultatanalys: Utvärdering av prototypen gentemot kriterierna

Detta avsnitt är en sammanfattning av våra resultat av funktionstester och fallstudien kopplat mot de krav och kriterier vi kommit fram till. Vi poängsätter hur väl vi uppnått kriterierna med en skala från 1 till 5 där 1 är ”inte alls” och 5 är ”helt tillfredsställande”.

Krav och kriterier	Utvärdering	Poäng
• Plattformen skall stödja sömlös klustring av flera servrar.	• Vår utvalda applikationsserver JBoss har inbyggt och lättanvänt stöd för klustring [65] så detta kravet anses helt uppfyllt.	5
• Nya versioner av EPP skall utan övergripande arkitekturförändringar kunna införas.	• Med hjälp av EPP-implementationernas arvsstruktur skall det vara tillräckligt att skapa en subklass av basklassen för en ny EPP-version. Föreslagen väg är att utgå från närmast föregående version och sedan genom testning (kompilering och körning av testprogram mot ote-miljö[22]) ta reda på vilka förändringar som behöver göras per metod. Det krävs alltså en del programmeringsjobb och testning för att införa en helt ny EPP-version.	4
• Arkitekturen skall medge att funktionalitet för andra registreringsmetoder och protokoll än EPP (ex. RRP, E-post) skall kunna införas.	• Växeln medger att andra protokoll än EPP kan implementeras eftersom växellogiken enbart ser till att ”rätt” JNDI-namn till en implementation refereras. Alltså kan man i ett nytt EAR-arkiv paketera RRP- eller e-postfunktionalitet som växeln kan referera till. Observera dock att det krävs en hel del programmeringsarbete för detta, men det är fullt möjligt. Likaså krävs det också en hel del arbete för att skapa och publicera den nya funktionaliteten som *Web-services*.	4
• En vanlig användare skall utan specialistkunskap kunna lägga till ett nytt registry ifall detta registry använder sig av en redan implementerad EPP-version	• Kravet ej uppfyllt i tillräcklig grad. En programmerare behövs för att göra en MBean som konfigurerar den nya implementationen. Dessutom krävs det viss kunskap för att paketera de erforderliga komponenterna i EAR-arkivet och driftsätta det. Dessutom kan	2

[22] Open Test Environment – Vissa registries tillhandahåller testservrar som utåt sett fungerar precis som de vanliga domännamnsservrarna men som inte påverkar det skarpa DNS-systemet.

	generering av Keystores, certifikathantering etc. vara en smula besvärlig.	
• Den data som konfigurerar en kommunikation skall presenteras begripligt och lättföränderligt via webbgränssnittet.	• Kriteriet väl uppfyllt. Eftersom majoriteten av konfigureringen sker i en MBean så får man webbgränssnittet på köpet ifall man utformat MBean korrekt. I MBean kan man presentera instruktioner till användaren. Kombinerat med en ”klassisk” användarmanual skall den data som behövs för en konfigurering vara lättbegriplig för någon med grundläggande kunskaper i hur ett *registry* fungerar.	4
• Konfigureringen skall ske genom ett webbgränssnitt i största möjliga mån	• Helt uppfyllt. Se ovan.	5
• En vanlig användare skall genom webbgränssnittet kunna lägga till stöd för ett nytt registry.	• Ej uppfyllt. Det krävs nykompilering av den MBean som skall hantera konfigureringen av ett nytt *registry*. Men då MBean är kompilerad och driftsatt så kan konfigureringen av ett nytt *registry* ske på enkel väg.	1
• Systemet skall offentliggöra utvalda publika metoder mot omvärlden så att dessa kan användas av t ex Perl, PHP, VBScript, ASP etc.	• Eftersom vår arkitektur använder sig av J2EE så kan vi publicera funktionalitet inkapslad i en *stateless*-sessionsböna som *Web-services* via Axis/SOAP. Observera alltså att man inte kan publicera funktionaliteten i en *stateful*-sessionsböna tillförlitligt som *Web-services* via den nuvarande versionen av Axis. Detta beror på problematiken med sessionshanteringen i SOAP som finns beskriven i teoridelen. • För en *stateless*-sessionsböna gäller att vi konceptuellt kan ropa på publika metoder i våra sessionsbönor från samtliga SOAP-kompatibla språk. Praktiskt sett finns det förstås en del bekymmer. Problemet i dagsläget är att få av språken (Java ett undantag) har stöd för komplexa datatyper. Detta finns beskrivet i närmare detalj på annan plats i uppsatsen.	4
• Underhållet skall ske via ett användarvänligt	• Ej helt uppfyllt. Underhållet av applikationsservern sker primärt genom	3

webbgränssnitt.	JBoss egna interna funktioner. Men om komponenter behöver bytas ut kan det ske under körning med hjälp av JBoss *hot-deployment*-funktionalitet. Komponenterna i sig är paketerade i arkiv som ansvarar för sina egna beroende i så hög utsträckning som möjligt, undantaget exempelvis applikationsgemensamma klasser och gränssnitt på grund av klassladdarproblematiken. I så stor utsträckning som möjligt är alla underhållskänsliga bitar implementerade i antingen MBeans eller i XML-filer vilket gör att omkompilering skall vara den sista utvägen vid underhåll.	
• Systemet skall eftersträva ”Single point of maintenance”	• Eftersom så stora delar av den föränderliga logiken i systemet bygger på generalisering med hjälp av arv så är uppfyllnaden av detta kriterium tämligen god. Likaså baseras systemet på en (1) växel, ett (1) externt gränssnitt mot växeln och ett (1) gränssnitt mot samtliga EPP-implementationer. Den punkt som är känslig är MBean som hanterar EPP-konfigureringen som måste finnas i ett unikt exemplar per implementation. Detta är ett arkitektoriskt problem där vi valt att ha så enkla och tydliga användargränssnitt som möjligt. Alternativt hade vi kunnat ha en gemensam MBean för samtliga EPP-implementationer och lagrat konfigureringsdatan i exempelvis objekt som fanns i en Hashtabell. Det hade medfört att det inte varit nödvändigt att skapa nya MBeans för varje ny EPP-implementation, men hade också inneburit att man inte kunnat använda användargränssnittet i vår MBean på ett lika smidigt vis eftersom konfigureringsinformationen inte funnits direkt tillgänglig som ett MBean-attribut. Detta är dock en punkt som eventuellt kan lösas längre fram med mer ingående studier av MBeans och JMX.	4

• Systemet skall vara implementerat på ett objektorienterat vis och använda sig av designmönster.	• Då Java, J2EE och JBoss i grund och botten är objektorienterade får man mycket gratis gällande objektorientering och designmönster med det angreppssätt vi valt. Designmönster är visserligen mer implementationsspecifika sett till varje enskilt fall, men vi har i de kontexter där vi sett nytta med dem använt oss av ett par av de designmönster som ”The Gang of Four”[23] förespråkar.	4
• Systemet skall vara kommenterat enligt ”javadoc” och företagets dokumentationsstandard.	• Väl uppfyllt. Egentligen inte något som är intressant ur arkitekturell synvinkel, men eftersom Java är väldigt enkelt att dokumentera genom javadoc så får man väldigt mycket gratis. Företagets dokumentationsstandard är inte relevant ur uppsatsen synvinkel så det tar vi inte upp här.	5
• Systemets externa gränssnitt skall finnas väl definierade i dokumentationen med erforderlig användningsinformation i manualform.	• Medelgod uppfyllnad. Samtliga EJB-*remote-interface* finns väl dokumenterade genom javadoc och de .html-dokument som javadoc genererar. Däremot är inte de metoder som *Web-services* tillhandahåller lika snyggt dokumenterade. Visserligen erbjuder Axis dels .wsdl-koden (xml-kod) för en tjänst, och dessutom finns det en ”listService”-tjänst i Axis som listar de tillgängliga tjänsterna, men dessa är inte speciellt pedagogiska då xml-kod inte tillhör kategorin lättläst dokumentation i våra ögon.	3
• Systemet skall använda sig av MVC-mönstret.	• Systemet är utformat enligt MVC-mönstret på så vis att J2EE och applikationsservrar i allmänhet bygger på detta mönster med en tydlig 3-skiktsarkitektur. Dock så är modellskiktet bortabstraherat i vårt fall eftersom vårt system inte svarar för någon datalagring utan skall matas med information utifrån och endast returnera data vars fortsatta levnad andra komponenter sedan svarar för. Den datalagring som förekommer i vårt system är konfigureringsinformationen	5

[23] Läs mer om ”The Gang of Four” på http://hillside.net/patterns/DPBook/GOF.html

	och en del SSL-relaterade data. Dessa data lagras primärt i MBönor och sekundärt i XML-dokument. Eftersom den data som lagras som attribut i MBönor är direkt nåbar genom JMX-konsolen är den inte att betrakta som helt skyddad enligt MVC, men å andra sidan så är det tanken i dessa fall – just att användaren på ett enkelt sätt kunna manipulera konfigureringsdatan i realtid. Den bästa lösningen är dock att låta Mbönan innehålla en data-*container* som i sin tur håller den aktuella datan, exempelvis en Hashtabell eller liknande. På så vis kan man skriva accessmetoder som kan innehålla logik för datavalidering och liknande.	
• Systemet skall endast acceptera atomära kommandon – dvs. att alla operationer måste vara antingen helt framgångsrika eller helt misslyckade.	• Eftersom JBoss innehåller *Rollback*-funktionalitet finns denna funktionalitet inbyggd från början [55]. Dessutom har vi designat systemet så att alla förutsedda fel alltid kastar ett av våra egendefinierade exceptions kommer exekveringen avbrytas och felet kommer bubbla tillbaka hela vägen till klienten. Just i vårt system sker ingen persistent datalagring förutom i MBönorna så vi använder oss inte av *Rollback*.	5
• Loggning av samtliga operationer mot registries skall ske, samt kunna läsas via en webbläsare.	• Ej implementerat, men JBoss har inbyggd loggning (Log4j) som kan användas.	1
• Åtkomst skall kunna ske från en godtycklig plattform med SOAP-stöd.	• Väl uppfyllt mha. Axis. Dock så är vi begränsade till primitiver som parametrar och returtyper än så länge. I nästa steg så blir det förmodligen JBoss4.0 med JBoss.Net som då ger oss en ännu närmare integration mellan *Web-services* och JBoss.	3
• Systemet skall kunna köras i godtycklig miljö med fullgott Javastöd.	• Vi har endast kört systemet på en Linuxbaserad server (Mandrake 9.0). Men eftersom både Axis och JBoss är Java-baserat skall det åtminstone i teorin inte vara några problem att köra systemet på exempelvis Windows, Unix eller MacOS X.	5

Vi genomför ingen viktning av kriterierna eller utvärderar snittpoängen eftersom vi ändå inte har någon annan arkitekturmodell att jämföra mot. Vår bedömning är dessutom subjektiv eftersom vi själva har utformat och implementerat prototypen. Poängen finns främst till för att underlätta tolkningen av den utvärderingstext vi bifogar till varje kriterie.

5 Slutsats

För det första kommer vi nu utifrån våra resultat att dra vissa slutsatser (konklusioner). Därefter kommer vi att diskutera kring de resultat vi erhållit för att ytterligare förstärka slutsatserna och för att dela med oss av vad vi lärt oss av denna forskning. Vi kommer att diskutera kring vårt metodval och utvärdera den samt vilka möjliga andra vägar man skulle kunna gå. Vi kommer även att beskriva områden varpå vidare forskning kring detta fascinerande område skulle kunna bedrivas samt personliga värderingar som vi anser vara av vikt.

5.1 Konklusioner

I vår *feasibility study* har vi tagit fram en prototyp på en systemarkitektur som skall svara mot de kriterier och krav som fastställdes. Dessa krav och kriterier skall möjliggöra att systemarkitekturen (prototypen) uppfyller alla de egenskaper som idealmodellen skall ha. För att enklast kunna dra några slutsatser så kopplar vi alltså våra resultat till vår forskningsfråga som är:

> *Hur kan användandet av plattformsoberoende tekniker möjliggöra en centraliserad, flexibel och skalbar systemarkitektur för en starkt föränderlig miljö?*

För det första kan vi bryta ner forskningsfrågan i mindre delar för att bättre kunna svara på den. Den består egentligen av tre olika delar som skall fungera i symbios med varandra: plattformsoberoende tekniker, centraliserad, flexibel och skalbar systemarkitektur samt en starkt föränderlig miljö.

Vi kommer nu att konkludera varje del för sig och sedan sammanfatta det hela i en heltäckande slutsats av vår forskning. Vi delar även upp centraliserad, flexibel och skalbar för att enklare kunna dra slutsatser om dessa var för sig.

5.1.1 Plattformsoberoende tekniker

Alla de tekniker vi använt oss av är helt och hållet plattformsoberoende så länge plattformen har fullgott Javastöd. Kärnan i Web-services, med tekniker som Axis, SOAP och XML, är plattformsoberoende, det är det som är syftet med det hela. Applikationsservern bygger på en Enterprise Java Bean-lösning och är alltså även den plattformsoberoende.

5.1.2 En starkt föränderlig miljö

En starkt föränderlig miljö är en miljö där det finns mycket yttre påverkan på ett system och där det ofta sker förändringar och vi tycker oss ha studerat en mycket bra föränderlig miljö. Domännamnsregistreringsvärlden är en mycket dynamisk värld med bland annat alla olika kommunikationsförfaranden och versionshanteringen av Internetprotokoll. Därför är studien vi genomfört inom domännamnsregistreringsområdet en legitim miljö med tanke på forskningsfrågan.

5.1.3 Centraliserad systemarkitektur

En centraliserad systemarkitektur för en starkt föränderlig miljö möjliggörs med *Web-services* och en applikationsserver, såsom JBoss, med stöd för J2EE. JBoss applikationsserver bygger på ett MVC-liknande mönster vilket innebär att dess arkitektur är gjord för att centralisera data och funktioner på ett enkelt sätt. *Web-services* i sin tur medgör att klienter kan komma åt centraliserad data och centraliserade funktioner oavsett vilket språk de är skrivna i.

En centraliserad systemarkitektur har sina för- och nackdelar. Vi har i teoridelen beskrivit skillnaderna mellan 2-skikts och 3-skikts arkitektur. Enligt MVC-mönstret som går hand i hand med 3-skiktsarkitekturen flyttar man i många fall in logik från *view*-skiktet till det nya *control*-skiktet. Detta medför oftast en centralisering av logiken i ett system samtidigt som man får ett extra datavalideringslager. Vi anser att vår systemarkitektur definitivt kan anses som centraliserad enligt MVC och vi vill peka på flera fördelar med detta:

- ”Single point of failure/correction/maintenance ” – Delar många klienter på samma logik får man enbart en plats att underhålla, söka buggar på och felkorrigera. Visserligen kan man också dra slutsatsen att man får en försämrad redundans i systemet med centraliserad logik samt att en enskild bugg då drabbar alla klienter som använder sig av den utsatta funktionen. Men vi drar också slutsatsen att det är billigare felsöka *en* enskild server och åtgärda den än att uppdatera en till väldigt många klienter. Och redundansproblemet går att minska genom att tillse att man har en robust IT-miljö med reservström, redundant anslutning till Internet, klustrade servrar etc.
- Vi drar också slutsatsen att ett centraliserat system kan upplevas som trögare och mer oflexibel i vissa hänseenden. Eftersom funktionskomponenten implementerar logiken finns inga möjligheter för klienten att förändra sitt eget funktionella beteende, såvida funktionskomponenten inte implementerar ett maskingränssnitt. Men i så fall så är ju faktiskt logikansvaret delegerat ut till klienten och då kan man inte prata om centraliserad logik. Tröghet kan upplevas eftersom svarstiderna över en nätverksanslutning ofrånkomligt kommer att vara högre än svarstiderna internt inom en dator. Enligt Kajbrink och Lorentsson [8] så upplevs oftast 2-skiktsarkitekturer som snabbare än motsvarande tillämpning i en 3-skiktsmodell. Detta är en nackdel man får ta och väga mot de fördelar en centralisering av logik medelst MVC och 3-skikt ger. Man kan undvika detta genom att minimera datautbytet mellan klient och funktionskomponent.

Vi konstaterar att vi i vår prototyp enkelt kunde förändra beteendet hos våra komponenter genom att manipulera konfigureringsinformationen. Vi drar slutsatsen att centralisering av affärslogik är en intressant väg att gå i de fall problemområdet är likt vårt med en föränderlig miljö som främsta egenskap. I mer statiska miljöer är centraliseringen kanske inte lika viktig eftersom en stabil miljö rimligtvis ger upphov till färre buggar och fel och således inte är i ett lika stort behov av de fördelar centraliseringen ger.

5.1.4 Flexibel systemarkitektur

En flexibel systemarkitektur möjliggörs med hjälp av *Web-services* och en applikationsserver. Man kan ha en mängd olika klienter skrivna i en mängd olika programmeringsspråk som alla via *Web-services* kan kommunicera med en applikationsserver och erhålla centraliserad data eller centraliserade funktioner. Applikationsservrar byggda kring J2EE medger att nästan vilken funktionalitet som helst kan kapslas in i komponenter och refereras enligt treskiktsprincipen. Det ger en mycket hög funktionell flexibilitet.

Ett exempel på ett problemområde där vår prototyps systemarkitektur kan projiceras är betalsystem på Internet, där en och samma E-handelsplats skall kunna sälja en och samma vara på ett och samma vis men ändå erbjuda en många olika betalningsalternativ. Dessa betalningsalternativ skulle kunna vara t ex kreditkort, Paypal[24], postförskott eller direktbetalning via Internetbank.

Vi ser tydligt hur vår modell kan appliceras på betalningsexemplet där en växel anropar olika betalningsmoduler beroende på vilket betalningssätt kunden vill betala. Även om de olika betalningssätten skiljer sig betydligt mer åt än de subtila skillnaderna mellan EPP-protokollets versioner så handlar det fortfarande egentligen bara om att designa och definiera ett riktigt bra och universellt gränssnitt mot de olika betalningskomponenterna.

5.1.5 Skalbar systemarkitektur

En skalbar systemarkitektur för en starkt föränderlig miljö möjliggörs med hjälp av en applikationsserver med stöd för J2EE. Skalbarhet som begrepp kan brytas ner till flera egenskaper såsom prestanda, kapacitet och utbyggbarhet. Som vi konstaterade i vår prototyputvärdering ger applikationsservern JBoss vår arkitektur en inbyggd skalbarhet sett till prestanda och kapacitet på grund av dess inbyggda stöd för klustring av flera servrar som delar JNDI-träd. Vidare så ger plattformoberoendet i sig en bra prestandaskalbarhet eftersom vi får ett större spann av maskinvara som systemet kan köras under – allt ifrån små enprocessor-PC till klustrade Unixsystem med många processorer. Utbyggbarhet får vi genom att de program vi driftsätter i applikationsservern där betraktas som självständiga komponenter. Visserligen kan det förekomma vissa beroenden i komponenterna mot andra komponenter, t.ex. gemensamma klasser, men det är en konsekvens av att vi eftersträvar centraliserat underhåll av systemet. Att ta höjd för skalbarhet i systemarkitekturen är viktigt eftersom ett system som inte kan växa eller förändra sin funktionalitet förmodligen inte kommer att överleva någon längre tid i en föränderlig miljö som t ex domännamnsregistreringsområdet [8].

5.1.6 Sammanfattning

En centraliserad, flexibel och skalbar systemarkitektur för en starkt föränderlig miljö möjliggörs med hjälp av plattformsoberoende tekniker som *Web-services* och en JBoss applikationsserver.

---

[24] Paypal – ett smidigt system för online- betalningar - http://www.paypal.com/

Olika klienter skrivna i olika programmeringsspråk kan via *Web-services* kommunicera med en applikationsserver och erhålla centraliserad data och centraliserade funktioner vilket både möjligör för en centraliserad och flexibel systemarkitektur.

En plattformsoberoende teknik som JBoss applikationsserver bygger på ett MVC-liknande mönster vilket betyder att användandet av denna möjliggör en centraliserad, flexibel och skalbar systemarkitektur. Mycket av den logik som förut låg ute i klienterna ligger i vår arkitektur samlad centralt i ett eget skikt vilket innebär att systemet blir billigare och lättare att administrera. Applikationsservrar byggda kring J2EE medger samtidigt att nästan vilken funktionalitet som helst kan kapslas in och refereras enligt treskiktsprincipen vilket ger en mycket hög funktionell flexibilitet. I och användandet av applikationsservern JBoss får systemarkitekturen en inbyggd skalbarhet om man ser till prestanda och kapacitet tack vare dess inbyggda stöd för klustring av flera servrar som delar JNDI-träd. JBoss bidrar också till skalbarhet i form av utbyggbarhet eftersom programmen som driftsätts i applikationsservern betraktas som självständiga komponenter.

JBoss applikationsserver i kombination med *Web-services* möjligör alltså för en mycket bra systemarkitektur för en starkt föränderlig miljö.

5.2 Diskussion

5.2.1 Applikationsservrar, J2EE och JBoss - en brant inlärningskurva?

Inlärningskurva är förstås något mycket subjektivt som är starkt beroende på den tidigare kunskapsbasen hos den inlärande. Som tidigare nämnts hade vi viss erfarenhet av JBoss och J2EE sedan tidigare. Dock var denna erfarenhet inte särskilt djupgående och det tog tid innan vi fick riktig kunskap om hur saker och ting hörde ihop. Själva installationen och driftsättandet av JBoss under Linux var inga större konstigheter förutom småproblem med nätverkskonfigurering. Ytterligare problem var att förstå de XML-filer som beskriver komponenternas egenskaper för JBoss. Definitioner över hur dessa filer skall se ut var knapphändig och för den oinvigde var det inte alltid självklart var man skulle börja leta. Dokumentationen över JBoss som finns tillgänglig gratis går dock genom grunderna och genom att studera källkod och XML-filer för exemplen får man en skaplig bild av hur man skulle gå till väga för att få ordning på sin ”Helloworld”-böna.

Vi drar slutsatsen att det krävs en ansenlig mängd förkunskaper och en väl utvecklad förmåga att söka och finna kunskap för att på ett effektivt vis komma igång med J2EE och dess relaterade tekniker. I vår fallstudiekontext fanns det tid att grundligt sätta sig in i de tekniker vi använt oss av – i en affärsverksamhet är det inte alltid det finns möjlighet till detta. Enligt

5.2.2 Databashanteringen i JBoss.

Vi konstaterade tidigare att JBoss kommer med en integrerad SQL-databas, HypersonicSQL. Även om vi inte utnyttjat entitetsbönor i vår prototyp så utförde vi några tester med sådana bönor. Vi valde att använda oss av *container-managed persistence* så själva dataskrivningen var helt transparent för oss.

Vi har vid vår tidigare användning av JBoss konstaterat att JBoss speciella språk för att utföra frågor till databasen, EJBQL, är behäftat med en hel del brister. Många av de vanliga funktionerna saknades vilket medförde att vi istället fick hämta ut stora datamängder och utföra exempelvis MAX- eller MIN-kontroller manuellt.

Detta har egentligen bakomliggande orsaker som beror på att mappning från J2EEs objektorienterade synsätt mot den underliggande traditionella relationsdatabasen. En relationsdatabas vilar teoretiskt och logiskt på den väl utforskade mängdläran medan den teoretiska basen för objektbaserad lagring är mer diffus och svårgripbar [66]. Det måste rimligtvis finnas någon anledning till att de objektorienterade databaserna idag, snart 30 år sedan objektorienteringens födelse, fortfarande inte har konkurrerat ut relationsdatabasen. Enligt [66] så är användningskontexten den avgörande faktorn - för enkel datarepresentation passar relationsdatabasen bäst medan den objektorienterade hanterar komplexa datastrukturer bättre.

Nu var som sagt persistent datalagring inte någon kritisk punkt i vårt system så detta är att se som en erfarenhet dragen inför framtiden mer än något annat.

5.2.3 Sessionshantering: Apache Axis vs. Microsoft .NET

Man kan även ta en diskussion kring vilken sessionslösning som är bäst – Microsofts .NET eller Apache-Axis? Utan att gå in på djupet så konstaterar vi att i en webbkontext där klienten styrs från en webbläsare så är det ganska vanligt med användare som inte tillåter cookies på sina datorer. Om sessionsinformationen istället bakas in i SOAP-huvudet så kan detta problemet till viss del undvikas. Eftersom Axis än så länge förespråkar att sessionshanteringen skall lösas med http-cookies eller i transportlagret, anser vi att Microsoft .NET har en mer genomtänkt lösning för sessionerna då sessionsinformationen där bakas in i SOAP-huvudet.

5.2.4 Web-services och säkerhet

Eftersom SOAP uttryckligen inte definierar någon form av säkerhetsstandard är det upp till den enskilde applikationsutvecklaren att lösa det på egen hand. Däremot finns det ett fristående initiativ till säkerhetsmekanism för SOAP - SOAP Security Extensions där man bakar in en digital signatur i SOAP-headern. Enligt Clemens Vasters - Chief Technologist på newtelligence AG [67] så är annars det vanligaste sättet att låta transportlagret sköta om säkerheten genom SSL.

I vår prototyp har vi inte implementerat någon form av säkerhetshantering för de delar som använder SOAP. Det låter förstås underligt, men då prototypen i nuläget inte kommer användas annat än inom företagets intranät så är just säkerhetsaspekterna inte något större bekymmer.

5.3 Metodutvärdering

Vi har använt oss av en ganska löst styrd form av att utreda och forska kring ett problemområde vilket vi själva har uppfattat gett ett bra resultat. Det fanns inga direkta klara och konkreta lösningar eller vägar att gå då vi startade vårt arbete utan vi fick istället mer eller mindre treva oss fram och hitta vägen själva.

Metoden ställer dock ganska höga krav på de inblandade och de som medverkar i studien. Det är en hög intensitet redan från början då beslut skall tas om vilken typ av litteratur som skall studeras för att införskaffa sig bred kunskap. Sedan gäller att snabbt kunna utvärdera vilka vägval som skall tas, vilken väg som är bättre än en annan.

Dock är det som vi anser det än viktigare att de personer med vilka man bollar idéer och tankar även de är väl insatta i ämnet och vad det är man försöker påvisa. Därför att de då bättre kan erbjuda stöd och hjälp vid bland annat vägval, *prototyping* och arkitektur.

Eftersom metoden vi valt omfattar en mängd olika angreppssätt för att studera ett område så gäller det att de iblandade personerna är bekanta med dessa sätt. Metoden omfattar ju bland annat litteraturstudier, intervjuer och *prototyping*. De två första angreppssätten är så pass vanliga att dessa åligger personerna att faktiskt kunna utföra utan större kunskapsinsamling men *prototyping* är lite mer komplicerat. I vårt fall så upplevdes det i alla fall så eftersom vi oftast befann oss på ganska ny mark och var tvungna att nästan uteslutande testa teknikerna för första gången. Detta för att det dels inte fanns något dokumenterat om tillvägagångssätt eller vilka av de olika teknikerna som fungerade med vad. Om man då inte har tillräcklig kunskap om programmering sedan tidigare och är någorlunda van att leta information ur en mängd olika källor eller att sitta och försöka på egen hand så kan det vara svårt att komma vidare.

Om vi hade valt att inte utföra någon *prototyping* i denna studie hade vi helt säkert inte kommit fram till samma resultat eftersom informationen om alla de olika teknikerna inte fanns tillgängliga. Istället var vi tvungna att själva pröva oss fram.

Att genomföra en *feasibility study* såsom vi utformade den var alltså enligt oss den enda rätta vägen att gå i denna studie. Endast tack vare den iterativa metoden med litteraturstudier-intervjuer-*prototyping* som resulterade i nubild, idealbild samt krav och kriterier på vilka vi genomförde en fallstudie kunde vi erhålla ett adekvat resultat.

5.4 Vidare forskning

Under tiden av arbetet med uppsatsen påträffades flera olika intressanta frågor men som på grund av tidsbrist inte kunnat inkluderas i denna studie. Dessa skulle däremot vara intressanta i framtida studier.

Rent generellt gäller att de tekniker vi valt inte på något sätt är de enda tekniker som faktiskt, i alla fall rent teoretiskt, kan användas. Vi har valt dels utifrån att de är gratis och för att det företag som vi utförde studien på ville använda sig av dessa tekniker. Därför hade det varit intressant att studera om det är någon skillnad i att använda sig av dessa tekniker gentemot andra (t ex sådana som kostar pengar), bland annat SOAP-motorerna Axis mot .NET:s SOAP-implementation och applikationsservrarna JBoss mot WebLogic.

JMX och MBeans är något som vi använde oss av vid *prototyping* och som vi uppfattade som något mycket bra. Dock hann vi bara snudda vid vad vi tror är dess fulla potential och därför hade det varit intressant att studera hur den tekniken skulle kunna förbättra vår forskning ytterligare. Dessa tekniker verkar dessutom vara intressanta att studera utifrån sig själva.

Vi tror att det resultat vi erhållit och de slutsatser vi dragit utifrån dessa resultat är applicerbara inom andra områden och i andra miljöer som är starkt dynamiska och föränderliga, t ex en E-handelsplats med flera olika betalningssätt. Det kan därför vara intressant att studera detta.

Det finns vissa produkter som kan automatgenerera dokumentation för *Web-services* och Enterprise Java Beans. Detta är något som skulle vara intressant att utvärdera och eventuellt utveckla då komponentarkitekturen i J2EE gör att källkodsdokumentationen man gör via Javadoc inte är lika intressant dokumentationen av EJB:er och deras tjänster. JBoss inbyggda JMX-konsol beskriver förvisso de EJB:er som finns driftsatta, men kanske inte på ett fullgott vis, och bara för driftsatta bönor. En bra standard för att dokumentera J2EE-komponenter och deras relation till Web-services är ett potentiellt forskningsområde.

5.5 Personliga värderingar

Vi tycker att det är en spännande framtid vi går till mötes där framförallt Web-services kan komma att förändra IT-relaterade affärsprocesser världen över. Just möjligheten att publicera och sälja maskintjänster direkt via nätet är mycket intressant då det kan betraktas som en annorlunda form av *out-sourcing*. Vem vet, kanske kommer House of Ports konkurrenter låta bli att utveckla egna system för att hantera det ökande antalet EPP-registries för att istället köpa registreringstjänsterna av House of Ports. Detta är faktiskt fullt praktiskt möjligt redan idag med några enklare modifieringar av vårt system (olika login/lösenord etc.) eftersom just domänregistreringsfunktionen idag är publikt åtkomlig som en *Web-service*. På samma vis kan jag se att Ports i framtiden kanske väljer att köpa in tjänster som exempelvis kundupplysning och liknande genom Web-services från externa företag.

Microsofts *Web-services* initiativ .NET kommer bli intressant att följa i framtiden. Microsofts olika programmeringsplattformar, exempelvis ASP, Visual Basic, C# och C++ integreras mot .NET på ett för Microsoft uniformt sätt. Kan Microsoft integrera *Web-services* i sina operativsystem och *desktop*-applikationer har de nog en god chans att ta över även detta område. På andra sidan barrikaderna står *open-source*-rörelsen tillsammans med bl.a. Sun Microsystems och försöker stå emot Microsoft med Java, Apache, Axis, JBoss och otaliga andra projekt vars gemensamma nämnare är att de oftast har ett öppet förhållande mot sina användare. *Desktop*-marknaden har Microsoft fortfarande ett järngrepp kring med sitt Windows-operativsystem, MacOS X är det mest mediafolk som använder, medan Linux förekommer i den akademiska världen samt hos systemutvecklare och entusiaster.

Servermarknaden ser däremot intressantare ut. Unixvärlden inkl. Linux erbjuder en beprövat stabil och säker miljö till skillnad från Windows serveroperativsystem som har plågats av medialt uppmärksammade säkerhetshål och tekniska problem. För små eller nystartade företag innebär också Microsofts serverprodukter en avsevärd ekonomisk investering. 2003-05-16 kostar en serverkonfiguration med Windows 2000 Server och MS SQL 2000 Server från ca 15 000 SEK för enklaste versionen upp till en bra bit över 100 000 SEK[25] för de mest avancerade varianterna. En typisk Linuxrelease kostar oftast

[25]Priser från http://www.dustin.se 2003-05-16

några hundralappar i affären eller laddas hem gratis och då ingår allt från själva operativsystemet till webbservrar, databaser och office-paket.

Men man får samtidigt helt klart ta i beaktande att Microsofts produkter oftast upplevs som klart enklare att installera och ”komma igång” med än motsvarigheterna under Linux. För även om man kanske får lägga ut 60 000 kronor mer på en Microsoftserver än en Linuxdito så får man inte glömma att inköpskostnaden bara är en liten del av den totala kostnaden. Om man behöver anställa en extra systemadministratör eller anlita en konsult för att få igång eller drifta det där billiga Linuxsystemet upplever man nog snart att det hade varit ett bättre val att köra med Microsoft i varje fall.

Denna frågeställning får man helt enkelt försöka besvara i fall till ifall, mycket beroende på exakt vad man vill ha ut av systemet, samt vilken kompetens som finns internt inom ens organisation.

6 Referenser

[1]
TechTarget, "architecture".
Copyright [TechTarget, Inc.] 2001-07-29.
Tillgänglig på: http://whatis.techtarget.com/definition/0,,sid9_gci211589,00.html
Nerladdad 2003-05-05.

[2]
Ian Sommerville "Software Engineering",
6th edition, Addison-Wesley Publishers Limited 2001.

[3]
Intervjuer, möten och löpande samtal med systemutvecklaren på House of Ports.

[4]
TechTarget, "legacy application".
Copyright [TechTarget, Inc.] 2001-07-31.
Tillgänglig på: http://search390.techtarget.com/sDefinition/0,,sid10_gci212472,00.html
Nerladdad 2003-05-06.

[5]
Nordstedts Svenska Ordbok, Språkdata, Sture Allén och Nordstedts förlag 1990.

[6]
Ralf Krakowski & Johan Dahlgren, "Viktiga egenskaper hos en applikation vid migrering till intranet".
Maggisteruppsats IA7400 23 maj 1998.
Tillgänglig på:
http://www.handels.gu.se/epc/archive/00002156/01/Dahlgren.Krakowski.IA7400.pdf

[7]
Hans Jones, "n-tier-mts".
Copyright [Hans Jones] 2001-05-08.
Tillgänglig på: http://users.du.se/~hjo/vb/n-tier-mts.pdf
Nerladdad 2003-05-22.

[8]
Kajbrink Janni & Lorentsson Johan, ""Client/Server is dead!" En studie av skiktade arkitekturer fokuserad på säkerhet, tillgänglighet och dynamik".
Magisteruppsats i Informatik, Publicerad VT 1999.
Tillgänglig på:
http://www.handels.gu.se/epc/archive/00002017/01/kajbrink.lorentsson.pdf
Nerladdad: 2003-05-08.

[9]
Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielasen, Jan Stage, "Objektorienterad analys och design", Studentlitteratur 2001

[10]
Partick Larsson & Frans Nilsson, "Designmönsters påverkan på kommunikationen i systemutvecklingsprojekt".
Magisteruppsats Luleå Tekniska Universitet 2002:041 SHU, Publicerad 2002-02-27.

Tillgänglig på: http://epubl.luth.se/1404-5508/2002/041/
Nerladdad: 2003-04-29.

[11]
Johan Flodström, "Infrastruktur för utveckling av webbapplikationer på SDC".
Magisteruppsats Dataterknikprogrammet, Publicerad 2002-06-06.
Tillgänglig på: http://www.ite.mh.se/~love/magister/uppsatser/johan_flodstrom.pdf
Nerladdad: 2003-04-29.

[12]
RFC2832 - NSI Registry Registrar Protocol (RRP) Version 1.1.0
Copyright [The Internet Society] 2000.
Tillgänglig på: http://www.faqs.org/rfcs/rfc2832.html
Nerladdad: 2003-03-06.

[13]
Scott Hollenbeck, "Extensible Provisioning Protocol".
Copyright [Scott Hollenbeck, VeriSign] 2001-10-02.
Tillgänglig på: http://xml.coverpages.org/epp.html
Nerladdad: 2003-03-03.

[14]
Magnus Ewert, "Datakommunikation – Nu och i framtiden", Studentlitteratur, Lund 1998.

[15]
WebServices.Org, "Why Web Services?".
Copyright [WebServices.Org] 2002.
Tillgänglig på: http://www.webservices.org/index.php/article/articlestatic/75
Nerladdad 2003-02-11.

[16]
Höij Magnus, "Computer Swedens artiklar om Web Services".
Copyright [ComputerSweden] 2003.
Tillgänglig på: http://computersweden.idg.se/special/webservices
Nerladdad 2003-02-11.

[17]
Castro-Leon Enrique, "A perspective on Web Services".
Copyright [WebServices.Org] 2002-02-18.
Tillgänglig på: http://www.webservices.org/index.php/article/articleview/113/1/61
Nerladdad 2003-02-11.

[18]
Cerami Ethan, "Top Ten FAQs for Web Services".
Copyright [O'Reilly & Associates, Inc.] 2002-02-12.
Tillgänglig på:
http://www.oreillynet.com/pub/a/webservices/2002/02/12/webservicefaqs.html
Nerladdad 2003-02-12.

[19]
W3C, "http://www.w3c.org/XML/".
Copyright [W3C] 2002-02-26.

Tillgänglig på: http://www.w3c.org/XML
Nerladdad 2003-03-03.

[20]
Lars Celander, ”XML, en skonsam men effektiv introduktion”.
Copyright [Azelia AB] 2002-11-07.
Tillgänglig på: http://www.azelia.se/xml.pdf
Nerladdad 2003-03-03.

[21]
Pär Hammarberg, ”Så är Net uppbyggt - från XML till UDDI”.
Copyright [IDG, International Data Group AB] 2002-10-11.
Tillgänglig på:
http://www.idg.se/ArticlePages/200210/10/20021010174320_IDG.se103/20021010174320_IDG.se103.dbp.asp
Nerladdad 2003-02-12.

[22]
Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, Dave Winer, ”Simple Object Access Protocol (SOAP) 1.1, W3C Note 08 May 2000”.
Copyright [DevelopMentor, International Business Machines Corporation, Lotus Development Corporation, Microsoft, UserLand Software] 2000-05-08.
Tillgänglig på: http://www.w3.org/TR/2000/NOTE-SOAP-20000508
Nerladdad 2003-02-12.

[23]
Axis, “Axis User's Guide”.
Copyright [The Apache Software Foundation]1999-2001.
Tillgänglig på:http://cvs.apache.org/viewcvs.cgi/~checkout~/xml-axis/java/docs/user-guide.html
Nerladdad 2003-03-11.

[24]
Scott Hollenbeck, “RFC3375”.
Copyright [The Internet Society] 2002-09.
Tillgänglig på: http://www.faqs.org/rfcs/rfc3375.html
Nerladdad 2003-05-08.

[25]
Computersweden, “Webbtjänster hetast under Suns Javaone”.
Copyright [ComputerSweden] 2001-06-05.
Tillgänglig på: http://computersweden.idg.se/text/010605-CS9
Nerladdad: 2003-02-20.

[26]
Computersweden, “Tillämpningsservrar nästa webbtrend”.
Copyright [ComputerSweden] 1997-12-17.
Tillgänglig på: http://computersweden.idg.se/text/971217-CS5
Nerladdad: 2003-02-20.

[27]
JavaTM 2 Platform, Enterprise Edition (J2EETM)
Copyright [java.sun.com] 2002-08-13.
Tillgänglig på: http://java.sun.com/j2ee/overview.html
Nerladdad: 2003-02-22.

[28]
Computersweden, " Sun vill knyta ihop javas olika delar".
Copyright [ComputerSweden] 1999-04-02.
Tillgänglig på: http://computersweden.idg.se/text/990402-CS2
Nerladdad: 2003-02-20.

[29]
JavaWorld.com, "A walking tour of JavaBeans"
Copyright [JavaWorld.com] Augusti 1997
Tillgänglig på: http://www.javaworld.com/javaworld/jw-08-1997/jw-08-beans-p2.html
Nerladdad: 2003-05-20.

[30]
John Zukowski, "Introduction to the JavaBeans API"
Copyright [java.sun.com] 1995-2003.
Tillgänglig på:
http://developer.java.sun.com/developer/onlineTraining/Beans/JBeansAPI/shortcourse.html#BeansOverview
Nerladdad: 2003-05-20.

[31]
Jonas Bergquist & Anders Ericsson, "Enterprise JavaBeans ur ett systemutvecklingsperspektiv".
Magisteruppsats i informatik VT2001.
Tillgänglig på:
http://www.handels.gu.se/epc/archive/00001694/01/bergqvist.ericsson.pdf
Nerladdad: 2003-03-03.

[32]
"The J2EE Tutorial – What Is an Enterprise Bean"
Copyright [java.sun.com] 2002-04-24.
Tillgänglig på: http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBConcepts2.html
Nerladdad: 2003-03-03.

[33]
Hemrajani Anil, "The state of Java middleware, Part 2: Enterprise JavaBeans"
Copyright [JavaWorld.com] April 1999.
Tillgänglig på: http://www.javaworld.com/javaworld/jw-04-1999/jw-04-middleware-p1.html
Nerladdad: 2003-03-04.

[34]
Kent Sundberg, "Optimeringar för Enterprise JavaBeans arkitekturer".
Magisteruppsats, Instutionen för Datavetenskap, Umeå Universitet 2001.
Tillgänglig på: http://www.cs.umu.se/~c97ksg/optimeringar.pdf
Nerladdad: 2003-03-04.

[35]
Eva Ljunggren & Joakim Viker, “Prestanda och portabilitet för komponentbaserade system”
Magisteruppsats, Datavetenskap Göteborgs Universitet 2001.
Tillgänglig på: www.math.chalmers.se/~kentp/Xjobb/ejb.pdf
Nerladdad: 2003-02-20.

[36]
Enterprise JavaBeansTM Technology Fundamentals
Copyright[java.sun.com] Maj 2000.
Tillgälig på:
http://developer.java.sun.com/developer/onlineTraining/EJBIntro/EJBIntro.html
Nerladdad: 2003-03-04.

[37]
”The J2EE Tutorial - What Is an Entity Bean?”
Copyright [java.sun.com] 2002-04-24.
Tillgänglig på: http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBConcepts4.html
Nerladdad: 2003-03-03.

[38]
”JBoss.org – Forum post”
Copyright [JBoss.org – Forum post] 2003.
Tillgänglit på:
http://www.JBoss.org/forums/thread.jsp?forum=121&thread=16948&message=3725490&q=record+locking#3725490
Nerladdad 2003-03-11

[39]
Terry Donoghue, “Pessimistic Locking”.
Copyright: [Apple Computer] 1999.
Tillgänglig på:
http://developer.apple.com/techpubs/webobjects/WebObjects_5/Topics/ProgrammingTopics.2d.html#14326
Nerladdad: 2003-03-11

[40]
Terry Donoghue, “Optimistic Locking”.
Copyright: [Apple Computer] 1999.
Tillgänglig på:
http://developer.apple.com/techpubs/webobjects/WebObjects_5/Topics/ProgrammingTopics.2c.html#18760
Nerladdad: 2003-03-11

[41]
”The Mail Archive – JBoss development”
Copyright [The Mail Archive] 2003
Tillgänglig på: http://www.mail-archive.com/JBoss-development@lists.sourceforge.net/maillist.html
Nerladdad 2003-03-11

[42]
”The J2EE Tutorial - What Is a Message-Driven Bean?”
Copyright [java.sun.com] 2002-04-24.
Tillgänglig på: http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBConcepts5.html#64640
Nerladdad: 2003-03-03.

[43]
Benjamin G. Sullins & Mark B. Whipple, “JMX in Action”.
Manning Publications Company, November 2002.
Tillgänglig på: http://developer.java.sun.com/developer/Books/javaprogramming/jmx
Nerladdad: 2003-04-05

[44]
Java Management Extensions (JMX) Specification.
Copyright [java.sun.com] 1995 – 2003.
Tillgänglig på: http://www.jcp.org/en/jsr/detail?id=3
Nerladdad: 2003-04-04.

[45]
Billy Newport, “JMX”.
Copyright [The Serverside.com] 2003.
Tillgänglig på: http://www.theserverside.com/resources/article.jsp?l=JMX
Nerladdad: 2003-04-04.

[46]
Orbism, “JMX”.
Copyright [Orbism Ltd.] 2001
Tillgänglig på: http://www.orbism.com/english/downloads/JMX_pres2.pdf
Nerladdad 2003-04-03

[47]
Todd Bowker, “Superior app management with JMX - Integrate JMX, a reusable configuration framework, with your JSPs”.
Copyright [JavaWorld.com] June 2001.
Tillgänglig på: http://www.javaworld.com/javaworld/jw-06-2001/jw-0608-jmx.html
Nerladdad: 2003-04-04.

[48]
AdventNet Inc., “JMX Agent Architecture.”
Copyright [AdventNet Inc.] 2002
Tillgänglig på:
http://www.adventnet.com/products/javaagent/help/quick_tour/j_jmx_agent_archi.html
Nerladdad 2003-04-02.

[49]
SpiritSoft, “SpiritWave Message Server, JMX Management Guide – Version 5.2”.
Copyright [SpiritSoft Inc.] 2001-2002
Tillgänglig på:
http://www.beyondjms.com/documentation/wave/5.2.0/JMX_Managment_Guide_5.2.pdf
Nerladdad 2003-04-02

[50]
Susning.nu, "Applikationsserver"
Copyright [Aronsson Datateknik] 2003-05-11
Tillgänglig på: http://susning.nu/Applikationsserver
Nerladdad: 2003-03-04.

[51]
Stefan Norlin, "Arkitekturer i flera skikt öppnar nätet".
Copyright [Nätverk & Kommunikation] 1999-04-08.
Tillgänglig på:
http://domino.idg.se/natkom/nokart.nsf/26fed193db0a590ac125637d0054e68d/c8e18658c17069f2c125674d0050cfb5?OpenDocument
Nerladdad: 2003-03-04.

[52]
TechTarget, "Container".
Copyright [TechTarget, Inc.] 2001-07-25.
Tillgänglig på:
http://searchdatabase.techtarget.com/sDefinition/0,,sid13_gci211833,00.html.
Nerladdad 2003-04-02

[53]
JBoss.org, "JBOSS OVERVIEW: J2EE AND BEYOND, THE SUPER-SERVER".
Copyright [JBoss.org] 1999-2002.
Tillgänglig på: http://www.JBoss.org/overview.jsp
Nerladdad 2003-03-14

[54]
JBoss.org, "What if the best application server out there were free?".
Copyright [JBoss.org] 1999-2002.
Tillgänglig på: http://www.JBoss.org/faq.pdf
Nerladdad 2003-03-14

[55]
JBoss, "JBoss 2.2+ Documentation".
Copyright [JBoss Organization] 2001-10-17.
Tillgänglig på:
http://www.dsg.cs.tcd.ie/~dowlingj/teaching/ds/tutorials/ejb/JBoss_v2_2_online-manual.pdf
Nerladdad: 2003-05-20.

[56]
Castro, Steve Easterbrook, John Mylopoulos, "Lecture 4: the Feasibility Study".
Copyright [Castro, Mylopoulos, Easterbrook] 2002.
Tillgänlig på: http://www.cs.toronto.edu/~sme/CSC340F/lecture_notes/05-06-info-acq-2up.pdf
Nerladdad 2002-03-03.

[57]
Jarl Backman, "Rapporter och uppsatser", Studentlitteratur Lund, 1998.

[58]
Idar Magne Holme & Bernt Krohn Solvan, "Forskningsmetodik om kvalitativa och kvantitativa metoder", Studentlitteratur 1991.

[59]
Helge Hüttenrauch, "Designing HRI – A Primer of Methods".
Copyright [Helge Hüttenrauch] 1998-09-04.
Tillgänglig på: http://www.nada.kth.se/~hehu/robo/report/webrep1.html.
Nerladdad 2003-04-24.

[60]
"Java Remote Method Invocation (RMI)"
Copyright [Sun Microsystems, Inc.] 2003
Tillgänglig på: http://java.sun.com/products/jdk/rmi/
Nerladdad: 2003-05-07

[61]
"Java RMI over IIOP"
Copyright [Sun Microsystems, Inc.] 2003
Tillgänglig på: http://java.sun.com/products/rmi-iiop/
Nerladdad: 2003-05-07

[62]
"CORBA® BASICS"
Copyright [Object Management Group, Inc.] 1997-2003.
Tillgänglig på: http://www.omg.org/gettingstarted/corbafaq.htm
Nerladdad: 2003-05-07

[63]
"Apache Axis homepage"
Copyright [Apache Software Foundation] 1999-2001.
Tillgänglig på: http://ws.apache.org/axis/
Nerladdad: 2003-05-05

[64]
Paul Kulchenko, "SOAP::Lite for Perl"
Copyright [Paul Kulchenko] 2000-2003.
Tillgänglig på: http://www.soaplite.com/
Nerladdad: 2003-05-05

[65]
Bill Burke & Sacha Labourey, "Clustering with JBoss 3.0"
Copyright [O'Reilly & Associates, Inc.] 2000-2003.
Tillgänglig på: http://www.onjava.com/pub/a/onjava/2002/07/10/JBoss.html?page=1
Nerladdad: 2003-05-06.

[66]
Frank Stajano, "A Gentle Introduction to Relational and Object Oriented Databases"
Copyright [Olivetti & Oracle Research Laboratory] 1998.
Tillgänglig på: http://www-lce.eng.cam.ac.uk/~fms27/db/tr-98-2.pdf
Nedladdad: 2003-05-17

[67]
Clemens Vasters, ”Why SOAP doesn't lack security while it does”.
Copyright [newtelligence AG] 2000.
Tillgänglig på: http://www.newtelligence.com/news/text01.aspx
Nerladdad: 2003-05-08

[68]
JBoss.org, “FAQ CONTENTS”.
Copyright [JBoss.org].
Tillgänglig på: http://www.JBoss.org/faq.jsp
Nerladdad 2003-04-02.

[69]
Richard Monson-Haefel, “What is a CMP bean?”.
Copyright [JGuru, Richard Monson-Haefel] 2000-04-11.
Tillgänglig på: http://www.jguru.com/faq/view.jsp?EID=1087.
Nerladdad 2003-04-02

[70]
TechTarget, “Concurrent Versions System”.
Copyright [TechTarget, Inc.] 2001-07-24.
Tillgänglig på: http://searchvb.techtarget.com/sDefinition/0,,sid8_gci211874,00.html
Nerladdad 2003-04-02

[71]
TechTarget, “domain name system”.
Copyright [TechTarget, Inc.] 2001-12-17.
Tillgänglig på:
http://searchwebservices.techtarget.com/sDefinition/0,,sid26_gci213908,00.html
Nerladdad 2003-04-02

[72]
Frontec, Ordbok.
Copyright [Frontec] 2003.
Tillgänglig på: http://www.frontec.se/Integration/ordboken/ordlista.htm
Nerladdad: 2003-04-03.

[73]
Afilias Limited, “General Registrar FAQ”.
Copyright [Afilias Limited] 2002-10-16.
Tillgänglig på: http://www.afilias.info/faqs/for_registrars/general_registrar
Nerladdad 2003-04-28.

[74]
Greg Voss, ”JavaBeans Update”.
Copyright [java.sun.com] December 1996.
Tillgänglig på:
http://developer.java.sun.com/developer/technicalArticles/Interviews/Update/
Nerladdad: 2003-03-03.

[75]
Mikael Printz, Torben Norling, Robert Burén, Ali Abida, “Teknisk beskrivning av fixa

fest.nu"
Copyright [Blufish AB] 2002-07-22
Tillgänglig på: http://www.bluefish.se/aquarium/fixafest3.html
Nerladdad: 2003-04-03.

[76]
Susning.nu, "JSP"
Copyright [Aronsson Datateknik] 2003-02-04.
Tillgänglig på: http://susning.nu/JSP
Nerladdad: 2003-04-03.

[77]
Susning, "LGPL"
Copyright [Aronsson Datateknik] 2002-06-11.
Tillgänglig på: http://susning.nu/LGPL
Nerladdad 2003-04-02

[78]
Susning, "RFC"
Copyright [Aronsson Datateknik] 2002-10-18.
Tillgänglig på: http://susning.nu/RFC
Nerladdad 2003-04-02

[79]
TechTarget, "Remote Procedure Call".
Copyright [TechTarget, Inc.] 2001-12-26.
Tillgänglig på:
http://searchwebservices.techtarget.com/sDefinition/0,,sid26_gci214272,00.html.
Nerladdad 2003-04-02

[80]
Joe Abley, Bill Manning, "DNS Registries - Shanghai, China, October 2002".
Copyright [Bill Manning, Joe Abley] 2002.
Tillgänglig på: http://www.wwtld.org/nameserver/Shanghai Registry CD 1x11.pdf
Nerladdad 2003-04-28.

[81]
TechTarget, "Secure Sockets Layer".
Copyright [TechTarget, Inc.] 2001-09-05.
Tillgänglig på:
http://searchSecurity.techtarget.com/sDefinition/0,,sid14_gci343029,00.html
Nerladdad 2003-04-02

[82]
TechTarget, "TCP/IP".
Copyright [TechTarget, Inc.] 2003-02-10.
Tillgänglig på:
http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci214173,00.html.
Nerladdad 2003-04-02.

[83]
Dennis M. Sosnoski, "Stateful session beans"

Copyright [Sosnoski Software Solutions Inc.] 2002
Tillgänglig på: http://www.sosnoski.com/presents/seajug/ejbws/siframes.htm
Nerladdad 2003-03-11.

Bilaga 1: Uttryck och förkortningar

- Axis, Apache eXtensible Interaction System

  Axis en implementation av SOAP och i grund och botten en SOAP-motor men inkluderar även bland annat en enkel fristående server, en server som kopplar in sig på servlet-motorer som till exempel Tomcat, omfattande stöd för WSDL och hjälpmedel för att övervaka TCP/IP-paket [23].

- BMP, Bean Managed Persistence

  Det finns två typer av *persistence* i EJB – BMP och CMP. Den stora skillnaden mellan dessa två typer är vem som faktiskt är ansvarig för att hålla EJB:n vid liv. I BMP så är den som utvecklat bönan som är ansvarig för att hålla reda på bönans tillstånd medans i CMP så är det *containern* som är ansvarig. I CMP så är det inte ens så att utvecklaren behöver veta vad objektrelationsdatabas-*persistence* är eller hur det fungerar eftersom detta sköts automatiskt [68].

- CMP, Container Managed Persistence

  En CMP-böna är en entitesböna vars tillstånd är automatiskt synkroniserad med databasen. Med andra ord så behöver inte utvecklaren av bönan skriva några explicita databaskopplingar i koden för bönan, *container*n kommer automatiskt att synkronisera de persistenta fälten med databasen så som dikterats av utvecklaren vid tiden för *deployment* [69].

- Container

  Inom JavaBeans komponentarkitektur så är en *container* en applikation eller ett subsystem i vilken ett programblock känt som komponent är exekverad [52].

- CVS, Concurrent Versions System

  Ett program som låter en programmerare spara och hämta olika utvecklingsversioner av källkod samt låter en grupp utvecklare dela kontrollen på flera olika versioner av filer på en gemensam förvaringsplats. Denna typ av program är ibland känd som *version control system* [70].

- Deploy

  Deployment är handlingen av att förbereda och skicka en eller flera bönor till applikationsservern för att göras tillgänglig som en applikationskomponent [68].

- DNS, Domain Name System

  Domännamnssystemet (DNS) är det sätt på vilken Internet-domännamn är lokaliserade och översatta till IP-adresser [71].

- EJB, Enterprise Java Bean

  EJB är en del av specifikationen i J2EE och är en arkitektur för att skapa komponenter i Java-miljö avsedda att exekveras på serversidan av en applikation.[72]

- Entitetsböna

  En entitetsböna representerar ett affärsobjekt i ett system och erbjuder bönutvecklare en komponentmodell där utvecklaren kan fokusera på affärslogiken medan en *container* tar hand om fortlevandet, access-kontroll osv. Det finns två typer av entitetsbönor, CMP och BMP [32].

- EPP, Extensible Provisioning Protocol
  EPP är ett XML-baserat protokoll och används som ett standardprotokoll för domännamnsregistrering hos en "tjock" *registry* [13, 73]. Att en *registry* är tjock betyder att all information om registrerade entiteter är lagrad i en central *registry* förvaringsplats [73].

- FTP, File Transfer Protocol
  Ett standardiserat protokoll för att föra över filer mellan datorer över Internet.

- Hot-deploy
  Hot deploy är en egenskap som tillåter dig att *deploy*, *redeploy* och *undeploy* en EJB medan servern körs.

- HTML, Hypertext Markup Language
  Är en uppsättning regler för hur en fil skall se ut om den är tänkt att visas upp som en web-sida.

- HTTP, Hypertext Transfer Protocol
  En uppsättning regler för utbyte och överföring av filer såsom text, bilder, ljud, video och andra mutlimediafiler.

- Java Bean
  En javaböna är en återanvändbar mjukvarukomponent skriven i programspråket java [74]

- J2EE, Java 2 Enterprise Edition
  J2EE är en specifikation över hur man bygger flerskiktade affärsapplikationer. J2EE förenklar den flerskiktade applikationsutvecklingen för utvecklare genom att erbjuda en komplett uppsättning komponenter för detta ändamål.[27]

- JBoss
  JBoss är en open-source-implementation av J2EE. JBoss är fokuserat kring EJB-delen av J2EE och lämnar webblagret till andra att implementera. JBoss 3 stödjer senaste EJB specifikationen (2.0) med allt vad det innebär samt har en intressant JMX baserad microkernel-arkitektur genom vilken klustring lär vara en enkel match att få snurr på.[75]

- JCA, Java Connector Architecture
  JCA är en specifikation över hur integration mot legacy-system och paketerade affärssystem går till. Det finns möjligheter för en affärssystemutvecklare att ansluta sig till specifikationen och därmed underlätta för andra system att integrera sig mot detta.[72]

- JMX, Java Management eXtensions
  JMX är tidigare känt som Java management API, JMAPI, och är en javabaserad arkitektur och rellaterad service för applikations och nätverks hantering.[43]

- JSP, JavaServer Pages
  JSP och är en Server Pages-lösning i Java. JSP-sidor kompilerar automatiskt till servlets som körs av en servletmotor.[76]

- Legacy
  Inom informationsteknologi är en legacy-applikation eller legacy-data sådana som

har härstammar från språk, plattformar och tekniker tidigare än den nuvarande tekniken.
De flesta organisationer som använder sig av datorer har legacy-applikationer och databaser som handhar kritiska affärsbehov. Vanligtvis är utmaningen att hålla legacy-applikationen uppe och körbar medan man konverterar till en ny mer effektiv kod som använder sig av den senaste tekniken och programmeringsförmågan.
Då man tidigare skrev applikationer för specifika operativsystem är det önskvärt att byta ut legacy-applikationer mot nya programmeringsspråk eller operativsystem som följer ett mer öppet eller standardiserat gränssnitt. Teoretisk skall det då bli enklare att uppdatera applikationerna utan att helt behöva skriva om källkoden och möjligör för användande på olika plattformar och operativsystem [4].

- LGPL, GNU Lesser General Public License
  LGPL är en avtalstext från Free Software Foundation (FSF) och GNU-projektet för subrutinbibliotek och klassbibliotek som är fri programvara och är en variant av GPL som båda är en copyleft-licens [77]. Kortfattat handlar det om fri mjukvara (har ej med pris att göra), frihet att distribuera kopior av fri mjukvara, att man har tillgång till källkoden, att man kan ändra källkoden och att man kan använda delar av källkoden i nya fria program [77].

- MBean, Management Bean
  En MBean är ett javaobjekt med vars hjälp man påverkar administreringen av resurser och dess gränssnitt tillhandahåller all den information som behövs för en applikation att kunna hantera och styra den [48, 49].

- MVC, Model View Controller
  MVC är en arkitektur som består av tre komponenter: en modellkomponent (model), en funktionskomponent (controller) samt en gränssnittskomponent (view) [9].

- .NET (Microsoft)
  Microsofts implementation av Web-services

- OSI, Open Systems Interconnection
  OSI är en standardiserad beskrivning eller "referensmodell" för hur meddelanden skall skickas mellan två olika noder i en telekommunikationsnätverk. OSI-referensmodellen delar upp datornätets funktioner i sju olika nivåer eller lager som ligger ovanpå varandra, där de högre nivåerna utnyttjar funktioner som tillhandahålls av de lägre nivåerna.

- Persistence
  Med hänsyn till EJB så är *persistence* en term som beskriver processen däri tillståndet av en aktiv (stateful?) EJB är lagrad (vanligtvis mot en databas) på ett sådant sätt att EJB:n kan återaktiveras senare. Det är *serialization* för EJB så att säga [68].

- Registrar
  Ett, av en specifik *registry*, godkänt ombud för att registrera en domän.

- Registry (registries)
  En organisation som äger rätten till och styr en specifik toppdomän.

- RFC, Request For Comments
  RFC är en dokumentserie som utges av Internet Engineering Task Force (IETF) och beskriver metoder och tekniska protokoll som används på Internet [78].
- RPC, Remote Procedure Call
  RPC är ett protokll som ett program kan använda sig av för att begära en tjänst från ett program lokaliserad i en annan dator inom ett nätverk utan att behöva veta några specifika detaljer om nätverket [79].
- RRP, NSI Registry-Registrar Protocol (RFC 2832)
  Används mellan Verisign Registry, som är en "tunn" *registry*, och godkända *registrars* [80]. I kontrast till EPP som används för tjocka registries så använder sig andra av en "tunn" modell där endast domän- och namnserverinformation lagras inom en *registry*. Kontaktinformationen lagras hos den som registrerade domänen och ett resultat av detta är att varje registrar måste ha egna WHOIS-servrar för att tillhandahålla denna information [73].
- Serialisera och deserialisera
  Att serialisera innebär att man tar en arbiträr datastruktur och representerar den som en struktur som kan lagras persistent över tid och tolkas av andra systeminstanser. Att deserialisera innebär att återskapa den arbiträra datastrukturen utifrån den serialiserade datan. Som exempel kan vi dra en parallell till C++ där man ofta bygger sina objekt kring pekare som pekar ut andra objekt och attribut. Dessa pekare är egentligen bara minnesreferenser som är helt meningslösa för andra system än det system som håller strukturen i minnet. För att kunna lagra ett sådant objekt som egentligen bara består av referenser, *serialiserar* man då ner innehållet på minnesplatserna på ett strukturerat sätt till exempelvis hårddisken eller en datastruktur som inte innefattar pekare. Ett vanligt sätt att serialisera på är att använda sig av kommaseparerade strängar eller xml-data. För att kunna deserialisera sådana objekt måste förstås deserialiseraren ha en mall som berättar för den hur den skall bygga upp objektet igen utifrån den serialiserade datan.
- SSL, Secure Sockets Layer
  SSL är en vanligt använt protokoll för att sköta säkerheten med meddelandeöverföring över Internet och använder ett programlager som ligger mellan HTTP och TCP. SLL använder sig av det publika-och-privata-nyckelkrypteringssystemet från RSA (www.rsasecurity.com) [81].
- SMTP, Simple Mail Transfer Protocol
  SMTP är ett TCP/IP-protokoll som används för att skicka och ta emot e-mail.
- SOAP, Simple Object Access Protocol
  SOAP är ett XML-baserat protokoll för informationsutbyte i en decentraliserad och distribuerad miljö [22].
- TCP/IP, Transmission Control Protocol/Internet Protocol
  TCP/IP är det grundläggande kommunikationsspråket eller protokollet över Internet. Det är ett två-lagersprogram där det högre lagret, TCP, hanterar själva meddelandet som kan delas upp i flera mindre paket medan det lägre lagret, IP, hanterar adressen för varje enskilt paket så att alla kommer till rätt destination [82].

- UDDI , Universal Description, Discovery and Integration
  UDDI representerar för närvarande lagret för att hitta och publicera tjänster (services) i en Web-services-protokollstack [18].
- Web services,
  Genom Web-services kan företag kapsla in existerande företagsprocesser och via Internet publicera dem som tjänster (services), leta efter och prenumenera på andra tjänster och utbyta information genom och bortom företagets verksamhet [15]. Detta innebär alltså att man inte bara återanvänder sina egna gjorda investeringar utan även att man får tillgång till andras [16].
- WSDL, Web Services Description Language
  WSDL representerar för närvarande lagret för beskrivning av tjänster i en Web-services-protokollstack och i ett nötskal så är WDSL en XML-grammatik för specificerande av ett publikt gränssnitt för en Web-service [18]:
- XML, eXstensible Markup Language
  XML kan beskrivas som ett generellt och standardiserat sätt att strukturera information och eftersom XML inte är bundet till någon viss typ av problem eller något visst applikationsområde så kan det användas i väldigt många sammanhang [20].

Bilaga 2: Intervjufrågor

Omgivning

- Vad heter du och vad är din arbetsuppgift / titel på företaget?
- Hur ser företagets informationssystemmiljö idag ut i stort?
- Vilka tekniska standarder försöker företaget basera sina lösningar kring?

Nubild

- Beskriv hur en typisk domännamnsregistrering går till, sett utifrån användaren – vilka praktiska steg behöver domännamnssäljaren utföra för en lyckad registrering?
- Beskriv hur en typisk domännamnsregistrering går till tekniskt – ifrån att säljaren startar sin säljapplikation, till att domännamnet är registrerat och transaktionen är lagrad.
- Beskriv hur dagens systemarkitektur uppkommit. Vilka behov har varit primära?
- Varför är dagens registreringsprocess inte fullgod?

Idealbild

- Beskriv den idealbild av registreringsarkiktekturen som Ni vill ha.
- I vilka aspekter är den bättre än dagens lösning?
- Vi pratar om att centralisera affärslogiken, finns det i Era ögon ökad risk för
    1. Driftstopp
    2. Konsekvenser av driftsstopp
    3. Konsekvenser av buggar i servertillämpningen?

Er systemmiljö går mer och mer mot en homogen plattforms- och språkstruktur. Är det, om vi bortser från plattformsoberoendeaspekten, motiverbart att utföra detta arbete när Era plattformar standardiseras?

Bilaga 3: Övriga tekniska erfarenheter

Erfarenheter av SOAP

SOAP såg ut att vara ett mycket bra val men har även en del svagheter. Framförallt är det en teknik som i en tillämpningskontext fortfarande är i sin linda. Det finns en del oklarheter i hur tillämpningarna fungerar i olika miljöer beträffande XML-tolkar och hantering av arbiträra objekt.

Vi har konstaterat att det finns svårigheter hanteringen arbiträra objekt. Företaget kommer därför med största sannolikhet att utveckla serialiserare för .ASP och Perl som kommer serialisera .COM-objekt och Perl-objekt till fördefinierade strängarrayer. Sedan kommer det i en ”SOAP-proxy” finnas en deserialiserare som gör om de strängbaserade meddelandena till arbiträra objektstrukturer, givetvis baserade på Java. Eftersom allting från SOAP-proxyn och inåt är Java i någon form så är det inga problem alls att använda komplexa objekt där oavsett om man använder sig av RMI (som JBoss använder) eller Axis/SOAP (som klarar arbiträra objekt då det är Java i bägge ändar).

Det finns två olika skolor när det gäller SOAP och sessioner. Den ena är Microsoftskolan med sin .NET-arkitektur medan den andra är Apache’s Axis.

Microsoft har valt att hantera sessioner genom att definiera dem i SOAP-kuvertets *header* (jämför *head*-taggen i HTML-kod) medan Axis vill hantera dem genom vanliga HTTP-*cookies* eller SSL-anslutningar [83]. Detta medför att ett totalt plattformsoberoende i sessionskontexten inte kan uppnås med dagens standardarkitekturer. Man kan fövisso välja att enbart använda sig av Microsoftlösningar i sitt system eller tvärtom, men då går plattformsoberoendet som är en av de stora poängerna med SOAP förlorad.

Förhoppningsvis kommer denna problematik lösas med applikationsservern JBoss 4.0 (som nu är under utveckling), där modulen JBoss.Net skall vara helt integrerad (se annat avsnitt angående JBoss.Net tillsammans med JBoss 3.0.4).

Sessionshantering är förstås bara ett problem i de fall där det är helt nödvändigt. Ett typexempel på när sessionshantering är viktigt är när en kund besöker en Internetbank. Då är det kritiskt ur säkerhetssynpunkt att servern hela tiden vet att den pratar med samma dator.

Sessionshanteringen var ett bekymmer eftersom Microsoft och Apache valt olika tillvägagångssätt i sina respektive SOAP-lösningar Axis och .NET. På grund av detta valde vi att designa bort SOAP-sessioner i vårt system eftersom det egentligen inte är speciellt aktuellt i ett system där en enda dator (Databasläsarsnurran) kommer vara klient i 99% av fallen. Men i ett av testfallen prövades en arkitektur där man kunde hantera persistens med hjälp av enitetsbönor som kommunicerade med omvärlden via *stateless*-sessionsbönor och *Web-services*. En annan möjlig lösning är att använda sig av en *stateful*-sessionsböna som sessionshållare så denna brist är inte kritisk i alla kontexter utan den går att bygga runt.

Det finns heller inget som hindrar oss från att implementera en egen sesssionshanterare. Vi har sett att vår kommunikationsmodell med EJB som *Web-services* endast hanterar *stateless*-sessionsbönor. Det finns därför inget som hindrar oss från att implementera en

egen sesssionshanterare. Ett exempel på en sådan implementation är att använda en s.k. *Wrapper*-klass på serversidan som verkar som en proxy mot omvärlden. I proxyn implementerar man en logik som kontrollerar sessionshanteringen innan sessionsbönan tar över anropet. För varje metodanrop från en klient så bifogar man aktuell session som en parameter. Parametern kan man definiera själv, ett löpnummer, en sträng, datum/nr etc. Även om metoden inte är speciellt ”snygg” så blir den åtminstone plattformsoberoende. Ett annat sätt är att titta på HTTP-klasserna i Java där det finns funktionalitet för nyckelgenerering och liknande som kan gå att utnyttja. För att uppnå persistens på serversidan kan man utnyttja entitetsbönor som är persistenta över tiden för att lagra sessioner.

Buggar, buggar och mer buggar..

Ett problem vi stött på är förmodligen en bugg i Axis. Om vi ändrar och kompilerar om en sessionsböna som pekas ut av en *Web-service* så går systemet ner fastän själva funktionaliteten och gränssnitten är helt oförändrade. Detta beror förmodligen på att Axis *cachar* bytekoden för sessionsbönorna och när den digitala signaturen inte längre stämmer överens mellan den tjänst som verkligen finns på servern och den tjänst Axis har *cachat* så får vi problem med ClassCastExceptions. Felet avhjälps genom att starta om applikationsservern.

Däremot går det utmärkt att skapa nya *Web-services* genom att definiera funktioner i redan existerande sessionsbönor genom att lägga till tjänsten ifråga i konfigurationsfilen server-config.wsdd och sedan *re-deploya* Axis.

Vi har undersökt bugglistan för Axis men har inte funnit detta problem beskrivet. Det är förmodligen inte helt relaterat till enbart Axis utan det är snarare kombinationen Axis/Jboss 3.x som spökar. Detta är förhoppningsvis ett av de problem som Jboss 4 / Jboss.Net kommer lösa. Eftersom vi konstaterade att man *inte* behöver starta om JBoss när man kör Java-baserade klientapplikationer mot nyförändrade bönor som inte *accessar* dem via SOAP utan via RMI. Därför vet vi att JBoss ensamt inte är ansvarig.

En annan bugg i Axis är när man i konfigurationsfilen server-config.wsdd deklarerar vilka metoder i en tjänst som är tillåtna att användas. Många exempel vi funnit på Internet definierar taggen;

```
<parameter name="allowedMethods" value="*"/>
```

där ”*” betyder att alla metoder tillåts. Av två anledningar är detta en dålig lösning:

- Främst innebär det att denna specifika *Web-service* inte kommer fungera utan kasta ur sig ett konstigt felmeddelande om *namespaces* när man försöker använda denna *Web- service* med den teknik vi använder. Det är förmodligen en bugg relaterad enbart till kombinationen Jboss / Axis.
- ”*” innebär att en sessionsbönas inbyggda metoder såsom ejbCreate, ejbDestroy m.fl. offentliggörs vilket absolut inte är meningen utan sådant skall man låta JBoss sköta om på egen hand.

Istället ska man explicit tala om för Axis vilka metoder som är tillåtna, kommaseparerat, exempelvis:

```
<parameter name="allowedMethods"
value="getPerson,setPerson,removePerson"/>
```

Driftsättande av Jboss

När det gäller JBoss så hade vi vissa problem med att få den att hantera entitetsbönor korrekt. Nyckeln ligger i att definiera sina ejb-xml.jar och JBoss.xml filer helt korrekt och att få en bra struktur på JNDI-benämningarna. För att uppnå en korrekt skiktad arkitektur valde vi att i vår experimentkod göra våra entitetsbönor till ”lokala” bönor. Det innebär att de inte har några ”remote”-gränssnitt utan bara ”local”-gränssnitt. Därigenom kan endast sessionsbönor inom samma Java Virtual Machine komma åt dataskiktet. I sin tur innebär detta att utomstående endast kan komma åt data genom att gå via sessionsbönorna som har funktionalitet för att arbeta mot datalagret, precis som MVC-mönstret är tänkt att fungera.

Ett annat JBoss-relaterat problem var att JNDI-benämningarna förlitade sig på maskinens ”/etc/hosts”-fil när den skulle returnera referensen till ett objekt. Ett tag var *localhost* inställd på 0.0.0.0 vilket av någon anledning ledde till att servern inte fick iväg några vettiga svar till den anropande klienten.