UNIVERSITETET I OSLO
Institutt for informatikk

Integrasjon av datasystemer. Mellomvare-arkitektur og tjenesteorientert arkitektur med Web Services.

Mastergradsoppgave

Eirik Lied

2. mai 2006

Forord

Jeg ønsker å rette en stor takk til alle som har hjulpet meg i arbeidet med denne masteroppgaven. Først og fremst vil jeg takke IT-sjef Steinar Bakken og alle de andre jeg har arbeidet med i Scanpix for måten jeg har blitt tatt imot, og fordi de alltid har vært positive til arbeidet med casestudien i denne oppgaven. Jeg vil også takke amanuensis Ragnar Normann som har vært internveileder for hvordan han alltid har gjort meg mer motivert for å arbeide med oppgaven. Til slutt vil jeg takke Pappa for de givende diskusjonene vi har hatt, og de gode rådene han gav meg når jeg følte at jeg stod helt fast.

Bekkestua, 1. mai 2006
Eirik Lied

Innhold

Forord **iii**

1 Innledning **1**
1.1 Forskningstilnærming 2
1.2 Struktur på dokumentet 2

2 Teknologier **3**
2.1 Mellomvarearkitektur 3
2.1.1 Hva er mellomvare? 3
2.1.2 Min definisjon av mellomvare 5
2.2 Tjenesteorientert arkitektur 6
2.2.1 Hva er tjenester? 6
2.2.2 Hva er tjenesteorientert arkitektur? 6
2.3 Web Services 7
2.3.1 Samhandlingsmodell og grensesnitt 8
2.3.2 Funksjonalitet og tjenestekvalitet 12
2.3.3 Språk og protokoller 12

3 Realisering av en systemintegrasjon **17**
3.1 Bakgrunn 17
3.1.1 Om Scanpix 17
3.1.2 Eksisterende systemer 18
3.1.3 Selgere og push-salg 21
3.2 Realisert løsning med mellomvare 23
3.2.1 Arbeidsform og utviklingsprosess 23
3.2.2 Systembeskrivelse 23
3.2.3 Mellomvare eller multidatabasesystem? 28
3.3 Tenkt løsning med tjenesteorientert arkitektur 29
3.3.1 Overordnet arkitektur 29
3.3.2 Eksisterende tjenester 30
3.3.3 Nye tjenester 32

4 Metodikk **37**

4.1 Fremgangsmåte 37
4.2 Vurderingskriterier 38

5 Observasjoner/Funn **39**
5.1 Arbeidsmengde og kompleksitet 39
5.1.1 Mellomvarearkitektur 39
5.1.2 Tjenesteorientert arkitektur 40
5.1.3 Oppsummering 41
5.2 Gjenbrukbarhet 41
5.2.1 Mellomvarearkitektur 41
5.2.2 Tjenesteorientert arkitektur 42
5.2.3 Oppsummering 42
5.3 Vedlikeholdbarhet 43
5.3.1 Mellomvarearkitektur 43
5.3.2 Tjenesteorientert arkitektur 44
5.3.3 Oppsummering 44
5.4 Ytelse 45
5.4.1 Mellomvarearkitektur 45
5.4.2 Tjenesteorientert arkitektur 46
5.4.3 Oppsummering 47
5.5 Andre kriterier 47
5.5.1 Krav til kompetanse 47
5.5.2 Opplæringsbehov 48
5.5.3 Drift 48
5.5.4 Skalerbarhet 48
5.5.5 Stabilitet 48
5.5.6 Sikkerhet 49

6 Drøfting **51**
6.1 Arbeidsmengde og kompleksitet 51
6.1.1 Forutsetninger og antakelser 51
6.1.2 Ad hoc systemintegrasjon 52
6.1.3 Mellomvare 53
6.1.4 Tjenesteorientert arkitektur 54
6.1.5 Sammenlikning 55
6.1.6 Oppsummering og kritikk 57
6.2 Gjenbrukbarhet 57
6.2.1 Forutsetninger og antakelser 57
6.2.2 Mellomvarearkitektur 58
6.2.3 Tjenesteorientert arkitektur 59
6.2.4 Sammenlikning 59
6.3 Vedlikeholdbarhet 59
6.3.1 Mellomvarearkitektur 60
6.3.2 Tjenesteorientert arkitektur 61

6.3.3 Sammenlikning 61
6.4 Ytelse 61
6.4.1 Modell for interaksjon 62
6.4.2 Mellomvarearkitektur 62
6.4.3 Tjenesteorientert arkitektur 63
6.4.4 Sammenlikning 63
6.4.5 En merknad til ytelse 64

7 Oppsummering og konklusjoner **65**
7.1 Hva vi fant ut 65
7.2 Valg av arkitektur 66
7.3 Uløste problemer og videre arbeid 67

Kapittel 1

Innledning

Stadig nye krav fra organisasjoner sammen med fremveksten av Internett, har økt betydningen av teknologier som kan integrere flere heterogene datasystemer. Ifølge (Gong et al., 2005) viser estimater fra industrien at opptil 70% av IT-relaterte kostnader brukes på aktiviteter som har med integrasjon å gjøre.

Integrasjonsarkitektur som er basert på standarder gir større mulighet for gjenbruk av systemer, og dermed løsninger som er rimeligere og med mindre avhengighet til enkelte leverandører. Det er derfor sterke markedskrefter som arbeider for at integrasjon av systemer skal være mulig uavhengig av leverandør og plattform.

Stadig skiftende krav til løsninger gir behov for arkitekturer som gir fleksibilitet og forenkler systemintegrasjonen. Mange arkitekturer er fremstilt for å dekke disse behovene, og vi skal i dette dokumentet sammenlikne to arkitekturer beregnet på integrasjon av systemer: Mellomvarearkitektur og tjenesteorientert arkitektur.

I løsninger med *mellomvarearkitektur* finner vi en mellomvare, som leverer en mellomvare-tjeneste, med hensikt å forenkle tilgang til distribuerte og heterogene systemer. Mellomvare finnes i mange former og tilbys av mange leverandører. Eksempler på mellomvare er "Object Request Broker"-systemer basert på Common Object Request Broker Architecture (CORBA) (Bastide et al., 2000), web-mellomvare som webtjenere med flere.

I løsninger med *tjenesteorientert arkitektur*, kan systemer sees som tjenester med gjenbrukbar funksjonalitet. Alle tjenestene har veldefinerte grensesnitt, og de kan kommunisere med hverandre ved hjelp av disse grensesnittene. En tjeneste utfører oppgavene som defineres i grensesnittene uavhengig av underliggende plattform og programmeringsspråk. Den vanligste typen tjenesteorientert arkitektur er Web Services.

1.1 Forskningstilnærming

Hovedformålet med denne oppgaven er å gjøre en sammenlikning av de to arkitekturene, og identifisere fordeler og ulemper ved bruk av dem. For å kunne gjøre dette, har jeg gjennomført en casestudie.

Casen er bedriften Scanpix AS, der det var behov for integrasjon av flere systemer for å effektivisere en bestemt salgs- og faktureringsprosess. I løpet av høsten 2005 designet jeg og implementerte mellomvare som en løsning på dette. Systemet ble tatt i bruk rett før årsskiftet 2005/2006, og har siden vært i daglig drift (april 2006).

På grunnlag av systemintegrasjonen som ble gjort med mellomvarearkitektur, har jeg designet en mulig integrasjonsløsning med tjenesteorientert arkitektur. De to løsningene har gitt grunnlag for å gjøre en konkret sammenlikning av arkitekturene. På grunnlag av den konkrete sammenlikningen har jeg argumentert for måter å beskrive arkitekturene på generelt, slik at vi kan gjøre en generell sammenlikning av dem.

Et viktig formål med oppgaven har også vært å si noe om når hver arkitektur er egnet for integrasjon.

1.2 Struktur på dokumentet

Kapittel 2 gir en introduksjon med teori om henholdsvis mellomvarearkitektur og tjenesteorientert arkitektur. Her gir vi definisjoner og beskriver grunnleggende prinsipper.

Casestudien er beskrevet i kapittel 3, hvor vi får en detaljert beskrivelse av grunnlaget for systemintegrasjonen, den realiserte mellomvarearkitekturløsningen, og den tenkte løsningen med tjenesteorientert arkitektur.

I kapittel 4 finner vi et sett kriterier som hjelper oss å vurdere arkitekturene, og i kapittel 5 ser vi på funn jeg gjorde fra casen med hensyn på vurderingskriteriene. Kapittel 6 inneholder en drøfting, der vi identifiserer metoder for å vurdere arkitekturene generelt, og gjør en generell vurdering med hensyn på kriteriene vi kom frem til i kapittel 4.

Kapittel 7 inneholder oppsummering og konklusjoner for oppgaven.

Kapittel 2

Teknologier

I dette kapittelet skal vi se på de to teknologiene jeg skal vurdere med hensyn på integrasjon av systemer: mellomvarearkitektur og tjenesteorientert arkitektur med Web Services.

2.1 Mellomvarearkitektur

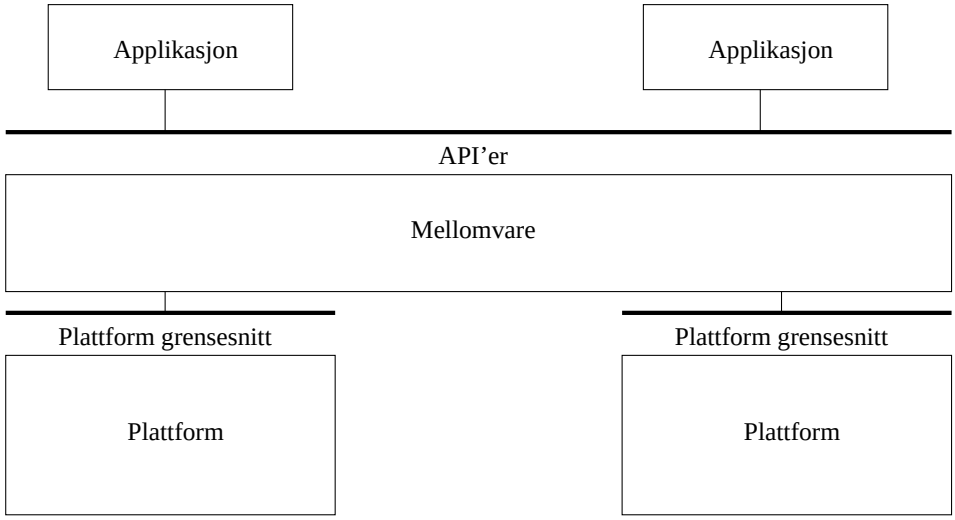
For å løse problemer rundt heterogenitet og distribuerte systemer er det ikke uvanlig å lage tjenester som tilbyr en standard måte å aksessere de forskjellige systemene på, også systemer som kan være fordelt utover et nettverk. Disse tjenestene kaller vi mellomvare-tjenester, eller bare mellomvare. Navnet "mellomvare" har vi fordi de typisk ligger over operativsystem og nettverk, men under annen programvare, vanligvis applikasjonsprogramvare.

Generelt kan vi si at mellomvare har til oppgave å erstatte spesifikke funksjoner som finnes på operativsystem-nivå med distribuerte funksjoner som kan brukes i et nettverk. Eksempler på vanlig mellomvare kan være distribuerte databaser, fjerntilgang på filer, eller Remote Procedure Call (RPC).

Eldre datasystemer kan ha blitt implementert uten hensyn til at data eller funksjonalitet skulle deles. Med mellomvare kan vi lettere dra nytte av eksisterende systemer ved å tilby et standard grensesnitt til disse.

2.1.1 Hva er mellomvare?

Vi kan begynne med å ta utgangpunkt i definisjonen av mellomvare, gitt i (Mahmoud, 2004):



Figur 2.1: Arkitektur for mellomvare, basert på (Bernstein, 1996)

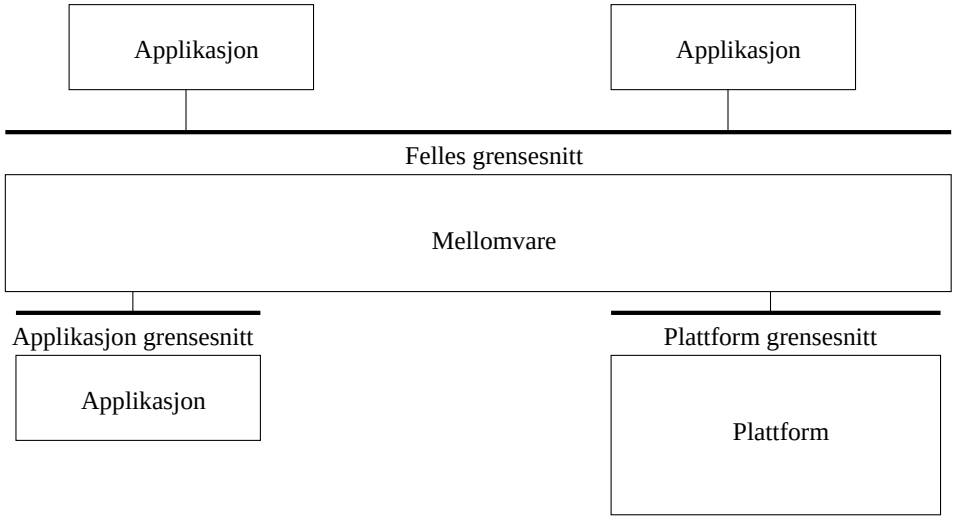
> "Middleware is a distributed software layer that sits above the network operating system and below the application layer and abstracts the heterogeneity of the underlying environment. It provides an integrated distributed environment whose objective is to simplify the task of programming and managing distributed applications."

Vi kan fra denne definisjonen merke oss at vi har et programvarelag som ligger over operativsystem og nettverk (plattform), men under applikasjonslaget. Det vil si at mellomvare typisk leverer tjenester til andre datasystemer. Mellomvaren har til oppgave å gi et samlet grensesnitt for de forskjellige underliggende tjenestene og dermed forenkle implementasjon og administrasjon av distribuerte systemer. Vi finner en noe løsere definisjon i (Bernstein, 1996):

> "A *middleware service* is a general purpose service that sits between platforms and applications."

Med "platform" i denne definisjonen menes det samme som vi definerte som "plattform" ovenfor.

Disse definisjonene likner hverandre da de begge definerer mellomvare som et programvarelag eller en tjeneste som ligger mellom en eller flere plattformer og et overliggende applikasjonslag. Figur 2.1 er inspirert av definisjonene i (Bernstein, 1996) og (Mahmoud, 2004).



Figur 2.2: Arkitektur for mellomvare med underliggende applikasjoner

2.1.2 Min definisjon av mellomvare

Beskrivelsen av mellomvare som ble gitt i henhold til (Mahmoud, 2004) gjør mellomvaren til et programvarelag som kun ligger mellom plattform og overliggende systemer. Men (Mahmoud, 2004) beskriver videre at noen mellomvareplattformer går lengre enn å ligge mellom plattform og systemer. De kan også skjule forskjeller i programmeringsspråk eller liknende. Dette tolker jeg til at mellomvare også kan være et programvarelag som ligger mellom både underliggende og overliggende applikasjoner. Denne mellomvaren har i så fall til oppgave å skjule forskjeller i de underliggende systemene, eller bare forenkle kommunikasjonen med alle disse. På dette grunnlaget gir jeg min definisjon av mellomvare som passer til bruk i denne oppgaven (se figur 2.2).

> "Mellomvare er et programvarelag som ligger mellom underliggende operativsystem, nettverk eller applikasjoner og overliggende applikasjoner. Mellomvaren har til hensikt å skjule distribusjon og heterogenitet i de underliggende systemer, og fremstille dette gjennom et samlet grensesnitt."

2.2 Tjenesteorientert arkitektur

Tjenestorientert arkitektur[1] har de siste årene sprunget frem som ett av de store moteordene i IT-bransjen. Vi skal her se hva tjenesteorientert arkitektur er, og hvorfor det satses på dette.

2.2.1 Hva er tjenester?

(Sim et al., 2005) beskriver tjenester som aktiva innen Informasjonsteknologi (IT) som samsvarer med ordentlige aktiviteter. Man får tilgang til disse tjenestene ved å følge deres regler (service policies). Tjenestens regler kan for eksempel definere hvem som har rett til å aksessere tjenesten eller ytelses- og sikkerhetsnivået til tjenesten.

Sett fra et teknisk perspektiv, er tjenester gjenbrukbare moduler som har veldefinerte grensesnitt og hvor det synlige eksterne grensesnittet er helt separert fra tjenestenes tekniske implementasjon. Denne separasjonen av grensesnitt og implementasjon gjør at tjenestesøkeren og tjenestetilbyderen er løst koblet, slik at de kan utvikle seg uavhengig av hverandre så lenge grensesnittene deres forblir uforandret.

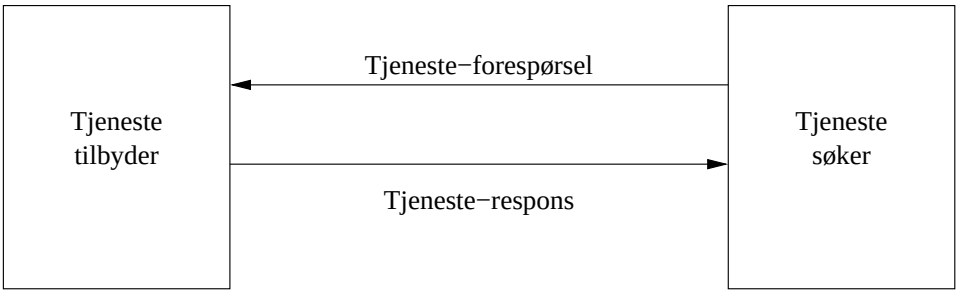
2.2.2 Hva er tjenesteorientert arkitektur?

(Barry, 2002) definerer tjenesteorientert arkitektur som en samling tjenester, der disse tjenestene kommuniserer med hverandre. Kommunikasjonen kan involvere enkel sending av data, eller det kan involvere to eller flere tjenester som koordinerer en eller annen aktivitet.

En fordel med tjenesteorientert arkitektur er at den fremmer samarbeid mellom og gjenbruk av systemer. Hvis noen har utviklet en velfungerende tjeneste, kan man lett gjenbruke denne istedet for å utvikle all funksjonalitet på nytt. Mye av verdien i å bruke tjenestorientert arkitektur vil derfor vise seg etter at det har vært i anvendelse en viss tid, når nye problemer/oppgaver kan løses ved å sette sammen eksisterende systemer istedet for å drive nyutvikling.

Se figur 2.3 på neste side for hvordan en enkel tjenesteorientert arkitektur fungerer.

[1] Engelsk: "Service Oriented Architecture", forkortet SOA.



Figur 2.3: Enkel illustrasjon av tjenesteorientert arkitektur

2.3 Web Services

Web Services er en realisering av tjenesteorientert arkitektur der vi bruker standarder som "eXtensible Markup Language" (XML), "Hypertext Transfer Protocol" (HTTP), "Simple Object Access Protocol" (SOAP), "Web Services Description Language" (WSDL) og "Universal Description, Discovery, and Integration" (UDDI). Vi skal se nærmere på disse standardene i kapittel 2.3.3 på side 12. I resten av oppgaven skal vi omtale "tjenesteorientert arkitektur med Web Services" bare som "tjenesteorientert arkitektur" med mindre noe annet er oppgitt.

Vi skal i denne seksjonen se nærmere på hva Web Services er, og forklare prinsipper rundt bruken av denne teknologien.

Jeg har valgt å ta utgangspunkt i W3C's[2] definisjon:

> *"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."*

Vi skal dele denne definisjonen inn i fire punkter og forklare hvert punkt. Denne beskrivelsen er basert på beskrivelsen i (Egeberg, 2004).

- *"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network."*
  Her skal vi merke oss to ting. Det ene er at vi har maskin-til-maskin kommunikasjon (ikke maskin-til-menneske). Det andre er at

[2]World Wide Web Consortium (W3C) er en organisasjon med hovedmål å utvikle standarder for World Wide Web.

kommunikasjonen kan foregår på et generelt nettverk, altså ikke bare Internett.

- *"It has an interface described in a machine-processable format (specifically WSDL)."*
  Beskrivelsen av grensesnittet til en Web Service, altså hvordan man kan kommunisere med tjenesten, er tilgjengelig i et format som kan leses og tolkes av andre datasystemer.

- *"Other systems interact with the Web service in a manner prescribed by its description using SOAP messages"*
  I tillegg til at meldingene som går mellom to Web Services skal gå i henhold til de beskrevede grensesnitt, skal meldingene formateres etter reglene i SOAP-protokollen.

- *"typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."*
  Vanligvis brukes transportprotokollen "HyperText Transfer Protocol" (HTTP) for utveksling av meldinger (dette er samme protokoll som brukes ved vanlig surfing på Internett). Data representeres med "Extensible Markup Language" (XML) i kombinasjon med andre internettstandarder.

2.3.1 Samhandlingsmodell og grensesnitt

En modell for kommunikasjon mellom to tjenester er vist i figur 2.4 på side 10.

Vi kan beskrive et typisk handlingsmønster ut fra tallene vist i figuren.

1. En tjenestetilbyder publiserer tjenestene den har tilgjengelig til en opplysningstjeneste som er allment kjent. Informasjonen om tjenestene publiseres med WSDL som er beskrevet nærmere i kapittel 2.3.3.3 på side 14.

2. Tjenestesøkere kan kontakte opplysningstjenesten og lete etter ønskede tjenester her.

3. Opplysningstjenesten leverer all nødvendig informasjon til tjenestesøkeren om tjenestetilbydere i WSDL formatet.

4. Ut fra informasjonen om tjenestetilbyderen i WSDL-dokumentet som ble sendt i punkt 3, kan tjenestesøkeren sende en forespørsel til tjenestetilbyderen. Forespørselen sendes med protokollen SOAP (forklares i kapittel 2.3.3.2 på side 13).

5. I et vanlig handlingsmønster vil tjenestetilbyderen returnere et svar til tjenestesøkeren. Også svaret følger SOAP-protokollen.

Som vi ser av dette handlingsmønsteret, brukes det en opplysningstjeneste for å holde orden på tjenester som tilbys, og gjøre disse mulig å oppdage for andre tjenester. Denne opplysningstjenesten behøver ikke være tilstede, men den beskriver noen sterke sider ved Web Services, da systemer kan kobles meget løst så lenge de kan oppdage andre tjenester hos en slik opplysningstjeneste.

Her kommer vi inn på "forsinket binding[3]" prinsippet beskrevet i (Kaye, 2003). Målet med "forsinket binding" er å unngå enhver form for fast eller hardkodet låsing til datatyper, strukturer, nettverksforbindelser eller liknende. Det argumenteres i (Kaye, 2003) for at dette er et viktig prinsipp for å oppnå løst koblede systemer, og det bør brukes dersom det er mulig.

En annen styrke ved Web Services er standardene som brukes til beskrivelser og kommunikasjon mellom tjenestene. Ved å bruke XML-baserte formater som WSDL og SOAP, kan systemene være implementert på en hvilken som helst plattform med et hvilket som helst programmeringsspråk. Som nevnt, sendes meldinger typisk på et nettverk med HTTP-protokollen, men SOAP-meldingene kan godt sendes over andre protokoller i transportlaget, som FTP[4] eller SMTP[5] (Newcomer and Lomow, 2004).

Web Services tillater tjenester å kommunisere på flere forskjellige måter. Vi skal se på de mest vanlige her.

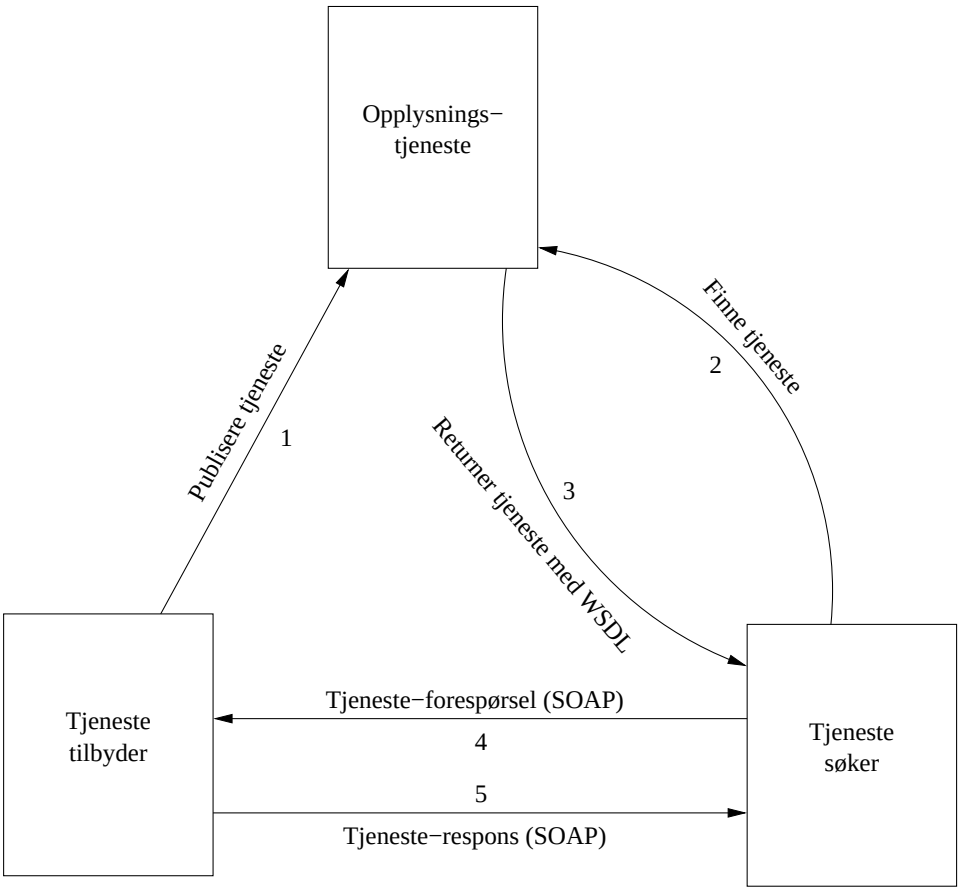
2.3.1.1 Request/response (synkrone meldinger)

Dette er et vanlig handlingsmønster, der tjenestesøkeren sender en forespørsel til en tjenestetilbyder. Tjenestesøkeren venter helt til den har fått en respons fra tjenestetilbyderen. Dette handlingsmønsteret kan være fordelaktig når man vil ha respons øyeblikkelig, men det kan fungere dårlig hvis en tjeneste ikke er tilgjengelig (tjenesten kan være nede, nettverket kan være nede eller liknende). Vi kan se dette handlingsmønsteret i figur 2.3 på side 7.

2.3.1.2 Persistente køer (asynkrone meldinger)

Dette handlingsmønsteret går ut på at meldinger kan sendes fra en tjeneste til en annen uten at senderen behøver å vente på noe svar

[3]Engelsk: delayed binding
[4]File Transfer Protocol
[5]Simple Mail Transfer Protocol



Figur 2.4: Samhandlingsmodell i Web Services

Figur 2.5: Handlingsmønster for persistente køer

fra mottakeren. En fordel med denne typen meldingsutveksling er at meldinger er persistente, altså at de lagres helt til mottakeren har fått dem. Dette gjør det mulig for senderen å jobbe videre selv om mottakeren ikke har mottatt meldingen. Vi kan merke oss at basisprotokollene i Web Services ikke kan garantere at meldinger kommer frem. Derfor er det lagt frem forslag til standarder for å garantere dette. Disse er

- WS-Reliability
- WS-ReliableMessaging

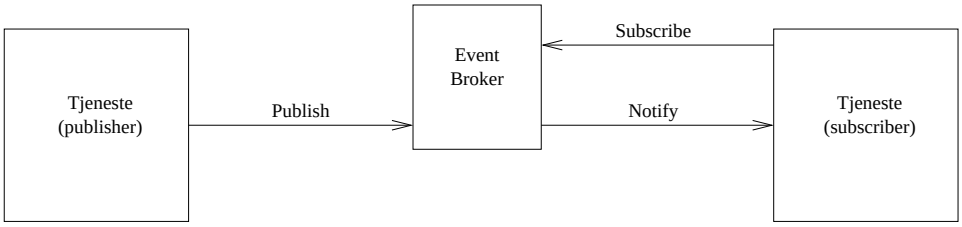
Det er ikke på dette tidspunktet (april 2006) bestemt hvilket av de to forslagene som skal brukes som standard.

En enkel modell for handlingsmønsteret sees i figur 2.5.

2.3.1.3 Publish/subscribe (publisere meldinger til mange tjenester)

Med publish/subscribe-interaksjoner kan abonnenter (subscribers) indikere hvilke hendelser de er interesserte i, og de vil motta et varsel når utgivere (publishers) sender ut relevante meldinger. Som vi kan se fra figur 2.6 på neste side, er abonnentene selv ansvarlige for å registrere hendelsene de er interesserte i. Vi ser også at publish/subscribe modellen bruker en "event-broker". Denne er ansvarlig for å ta imot varsel fra utgivere og så varsle alle abonnenter som har meldt sin interesse. Følgende standarder tilbyr publish/subscribe-interaksjon på toppen av basisprotokollene i Web Services (heller ikke her er det bestemt hvilken som skal brukes som standard):

- WS-Eventing
- WS-Notification



Figur 2.6: Handlingsmønster for publish/subscribe

2.3.2 Funksjonalitet og tjenestekvalitet

Vi kan kort oppsummere det ovenstående slik: Systemer som følger Web Services arkitektur, er systemer som tilbyr tjenester over et nettverk. Innenfor dette er det store variasjoner. Vi kan for eksempel ha store variasjoner i hva en tjeneste tilbyr, eller det kan være variasjoner i tjenestekvalitet ("Quality of Service"). Dette kan gå ut på hvor stor kapasitet en tjeneste kan ha, eller hvor pålitelig den er.

Med hensyn til utveksling av meldinger, kan de utvidede Web Services Standardene som WS-Reliability, WS-ReliableMessaging, WS-AtomicTransaction/Coordination, WS-Eventing og WS-Notification, tilby høyere tjenestekvalitet på forskjellige områder.

2.3.3 Språk og protokoller

I de foregående seksjonene om Web Services er det en del språk og protokoller som trenger en forklaring. Vi skal se på basisblokkene i Web Services (XML, SOAP, WSDL og UDDI), men vi kan merke oss at Web Services tilbyr standarder utover de vi skal se på her. Web Services er en relativt fersk teknologi, og flere standarder er ikke ferdig utarbeidet.

2.3.3.1 eXtensible Markup Language (XML)

XML er et enkelt, fleksibelt tekstformat avledet av Standard General Markup Language (SGML). XML ble i utgangspunktet designet for å møte utfordringer med elektronisk publisering i stor skala, men spiller en stor rolle innenfor utveksling av data på Internett og andre steder.

XML gir mulighet for å danne tre-baserte (hierarkiske) strukturer med informasjon, der tag-er brukes til å definere elementer og attributter. Når to systemer skal kommunisere ved hjelp av XML-dokumenter, må oppbyggingen følge definerte regler. Disse reglene ligger gjerne i

dokumenter som kalles "XML-schemas", og de kan leses og tolkes av datasystemer.

Et kort eksempel fra (Wikipedia.org, 2006b) følger nedenfor.

```
<?xml version="1.0" encoding="UTF-8"?>

<Recipe name="bread" prep_time="5 mins" cook_time="3 hours">
  <title>Basic bread</title>
  <ingredient amount="3" unit="cups">Flour</ingredient>
  <ingredient amount="0.25" unit="ounce">Yeast</ingredient>
  <ingredient amount="1.5" unit="cups" state="warm">Water</ingredient>
  <ingredient amount="1" unit="teaspoon">Salt</ingredient>
  <Instructions>
   <step>Mix all ingredients together, and knead thoroughly.</step>
   <step>Cover with a cloth, and leave for one hour in warm room.</step>
   <step>Knead again, place in a tin, and then bake in the oven.</step>
  </Instructions>
</Recipe>
```

Dette er et enkelt XML-dokument for en oppskrift på brød. Vi skal ikke gå nærmere inn på syntaksen; vi ser at den er relativt selvforklarende, med tag-er som har navn som beskriver innholdet av elementene.

Alle grunnleggende data i Web Services utveksles i XML.

2.3.3.2 Simple Object Access Protocol (SOAP)

SOAP er en XML-basert protokoll for å sende meldinger eller gjøre "Remote Procedure Calls" mellom datasystemer i et nettverk. Det finnes flere forskjellige meldingsmønstre i SOAP, men det mest utbredte er "Remote Procedure Call" (RPC-mønsteret), hvor en nettverksnode (klienten) sender en forespørsel (request) til en annen node (serveren), og serveren øyeblikkelig sender en melding tilbake til klienten.

SOAP kan sendes over mange transportprotokoller, men den vanligste er HTTP, da denne fungerer godt med infrastrukturen vi finner på Internett. Dette viser seg særlig i at HTTP fungerer bra med brannmurer[6] i motsetning til en del andre distribuerte protokoller.

Meldinger som sendes med SOAP legges i en SOAP-konvolutt med en forespørsel, og det returneres et svar som også er pakket inn i en SOAP-konvolutt. Nedenfor kan vi se et eksempel hentet fra (Wikipedia.org, 2006a)

[6]En brannmur er typisk et program som fungerer på datasystemer i et nettverk og har til oppgave å hindre uønsket trafikk mellom soner i nettverket.

på en SOAP-konvolutt med en forespørsel og tilhørende svar.

SOAP Request:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetails xmlns="http://warehouse.example.com/ws">
      <productID>827635</productID>
    </getProductDetails>
  </soap:Body>
</soap:Envelope>
```

SOAP Response:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetailsResponse xmlns="http://warehouse.example.com/ws">
      <getProductDetailsResult>
        <productName>Toptimate 3-Piece Set</productName>
        <productID>827635</productID>
        <description>3-Piece luggage set. Black Polyester.</description>
        <price>96.50</price>
        <inStock>true</inStock>
      </getProductDetailsResult>
    </getProductDetailsResponse>
  </soap:Body>
</soap:Envelope>
```

Her kan vi se hvordan en klient kan ha formet en SOAP-melding hvor det spørres om produktinformasjon fra en tenkt Web Service fra et varehus. I SOAP-responsen ser vi svaret fra varehuset med produktinformasjon i retur til klienten.

2.3.3.3 Web Services Description Language (WSDL)

WSDL er en XML-basert standard for å beskrive Web Services. En tjeneste som følger Web Services' standarder må alltid være beskrevet med dette språket. Denne beskrivelsen gjør det mulig for andre applikasjoner å få tilgang til en tjeneste. WSDL beskriver hvordan meldinger kan sendes til tjenesten, hvordan de må formateres, og hva som kan ventes å få i retur. En WSDL-beskrivelse definerer en eller flere porter som tjenesten tilbys over. En port er en kombinasjon av en tjeneste, en transportprotokoll og en URI[7].

[7] Uniform Resource Identifier

Det er mulig å definere flere porter slik at en tjeneste kan være tilgjengelig over flere forskjellig protokoller.

2.3.3.4 Universal Description, Discovery, and Integration (UDDI)

UDDI er en protokoll for å publisere informasjon om tjenester i en tjenesteorientert arkitektur, og den er i utgangspunktet en av Web Services' basisstandarder. Ifølge (Newcomer and Lomow, 2004) var den opprinnelige hensikten med UDDI å la det være en allmenn tjeneste hvor organisasjoner kunne registrere tjenestene sine for å la dem bli oppdaget av andre. (Newcomer and Lomow, 2004) skriver videre at selv om SOAP og WSDL har blitt svært utbredt, har ikke UDDI blitt like vellykket.

UDDI har også blitt brukt til opplysningstjeneste innenfor organisasjoner, hvor det har vist seg å bli mer populært, uten at det har blitt en standard. Selv om det klart at en opplysningstjeneste er en nødvendig del av Web Services-plattformen, er det ikke sikkert at UDDI blir den endelige standarden for dette.

Kapittel 3

Realisering av en systemintegrasjon

3.1 Bakgrunn

Før vi gjennomgår systemintegrasjonen som er tema for oppgaven, skal vi se på bakgrunnen for den integrasjonen som er gjort. Vi skal se på det aktuelle selskapet, systemene som skal integreres, og brukeres behov for å forenkle bruken av flere systemer.

3.1.1 Om Scanpix

Scanpix Norge AS (Scanpix) ble dannet i 1999 etter en fusjon av fotodelen til NTB Pluss og Scan-Foto. Scanpix er en del av Scanpix-Gruppen som består av Scanpix Sverige AB, Scanpix Norge A/S, Scanpix Danmark A/S og Scanpix Baltics. Scanpix-Gruppen eies av Schibsted, NTB og Berlingske Officin.

Scanpix leverer bilder til dags-, ukes- og fagpresse, samt reklamebyråer og forlag, og er Norges største totalleverandør av fototjenester. Selskapet selger bilder fra sitt eget arkiv som består av over 20 millioner bilder, og har samtidig salgsrettigheter fra VGs og Aftenpostens arkiver. Scanpix har 12 egne fotografer som sørger for nyhetsdekning i både inn- og utland. Videre tilbyr Scanpix oppdragsfotografering, studiofotografering og digital lagring av media for diverse kunder.

Scanpix har også to digitale bildearkiver, “Scanpix Digital Library” og “Scanpix Creative”. “Scanpix Creative” tilbyr skandinaviske og internasjonale illustrasjonsbilder til kommersielt og redaksjonelt bruk,

mens "Scanpix Digital Library" er en tjeneste som gir mulighet for å se på og laste ned et bredt spekter av sports-, arkiv- og nyhetsbilder. Vi skal komme tilbake til "Scanpix Digital Library".

De ansatte arbeider innenfor fotografi, redaksjonelt arbeid, salg, arkivarbeid og IT. Scanpix har lokaler i Akersgata i Oslo og har rundt 55 ansatte, hvorav 4 jobber i IT-avdelingen. Selskapet hadde i 2004 en omsetning på 83,4 millioner kr og et resultat på omtrent 7 millioner kr etter skatt.

3.1.2 Eksisterende systemer

Scanpix har flere eksisterende datasystemer i form av bildesystemer og økonomisystemer. Vi skal i dette kapittelet se hvilke systemer Scanpix har, og hvordan ansatte benytter de forskjellige systemene.

3.1.2.1 Scanpix Digital Library

"Scanpix Digital Library" (SDL) er hovedsystemet Scanpix har for digitale bilder. SDL ble opprettet i 1996 og inneholder i dag syv millioner bilder. All programvare i SDL forvaltes av Scanpix' IT-avdeling, mens drift av maskinpark med all underliggende programvare er satt bort til andre.

Den viktigste funksjonaliteten til SDL er muligheten for å søke etter bilder. For å gjøre søking mulig brukes fritekstdatabasen Trip, som blir utviklet og eid av Tieto Enator Sverige.

SDL opplever stor last i form av mange brukere og strøm av mange og store filer. For å tåle slike påkjenninger og samtidig yte bra, er størstedelen av SDL implementert i språket C. C bruker lite maskinressurser og er meget raskt, men det krever ofte lengre utviklingstid, på grunn av lite kompakt kode og treg feilfinning.

Bilder inn i SDL I gjennomsnitt kommer det inn rundt 10000 nye bilder til SDL hver dag. Disse kommer fra nyhetsbyråer via satellitt, fra eksterne kunder via FTP[1], eller legges inn fra Aftenposten og VG internt i systemet. Alle bilder som kommer inn, er filer i formatet JPEG[2] JFIF[3]. Bildefilene inneholder også tekstlig informasjon om selve bildet

[1] File Transfer Protocol er en protokoll for overføring av filer mellom to datamaskiner, vanligvis på Internett.

[2] Joint Photographic Experts Group (JPEG) spesifiserer hvordan et bilde kan transformeres til en strøm av bytes, men ikke hvordan disse bytene blir pakket inn på noe spesielt lagringsmedium.

[3] JPEG File Interchange Format (JFIF) spesifiserer en måte å produsere en fil som er egnet for lagring og overføring mellom datamaskiner.

(bildeinformasjon). Standarden for beskrivelse og innpakking av bilder er definert av International Press Telecommunications Council (IPTC), mens metoden for hvordan dette legges inn i de binære filene er definert av Adobe.

Når et bilde legges inn i SDL, vil all bildeinformasjon trekkes ut og indekseres i Trip. Samtidig blir det laget to mindre versjoner av bildet som brukes for enkelt å vise bildene til de som benytter SDL. Deretter lagres den store og de to små bildefilene som en sammenhengende fil i SDL.

Bruk av SDL SDL brukes i stor grad av kunder som vil kjøpe bilder, men også av ansatte i Scanpix for å finne bilder. SDL tilbyr søking etter bilder med visning av søkeresultatene som mange småbilder (thumbnails). Videre tilbyr SDL forhåndsvisning (preview) av bildene hvor man får se en større versjon og mer informasjon om bildet. Deretter kan man laste ned høyoppløselig versjon av av bildene (kjøpe dem).

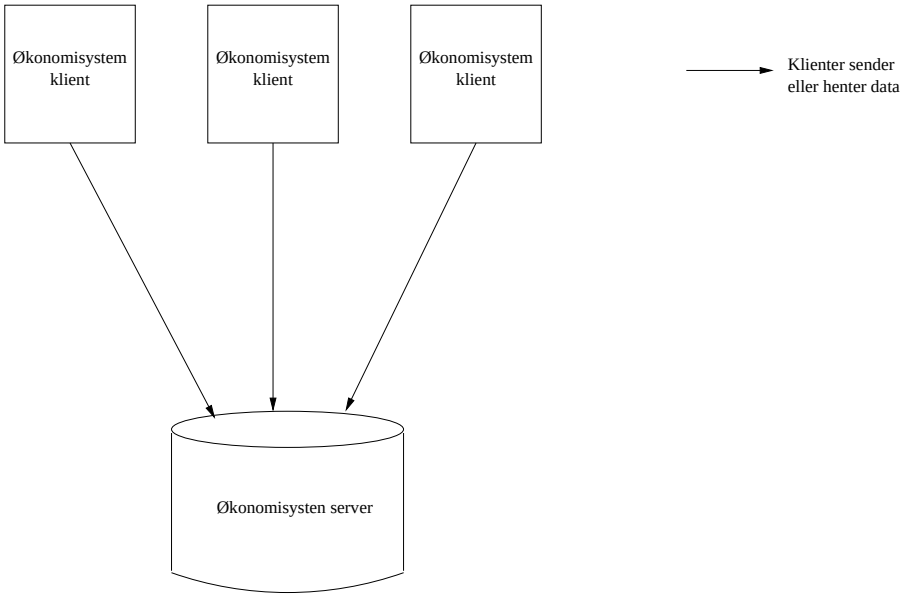
SDL brukes også av selgere i Scanpix. Vi skal komme tilbake til hvordan disse arbeider.

3.1.2.2 Økonomisystem

Scanpix har i flere år benyttet økonomisystemet Maconomy til flere formål i selskapet. Vi skal referere til dette systemet som "økonomisystemet". Fakturering av bilder har vært et viktig bruksområdet for økonomisystemet. Selgere har da opprettet ordre i økonomisystemet og skrevet inn all informasjon om ordren før den lagres. Ordren blir senere brukt til å lage fakturaer. Siden økonomisystemet har mange formål, er det viktig at alle ordre kommer inn i dette systemet.

Økonomisystemet er et ferdig system som leveres til mange bedrifter. Brukere arbeider med et klientprogram med grafisk brukergrensesnitt (se figur 3.1 på neste side). Klientprogrammene kommuniserer med en server hvor alle data lagres. Systemet leveres med et eget skriptespråk kalt Mscript. Mscript kan brukes til å utvide funksjonaliteten på serversiden i økonomisystemet.

Vi kan merke oss at dette ferdige økonomisystemet har implementert støtte for Web Services, og det burde derfor ikke være vanskelig å integrere økonomisystemet med andre systemer ved hjelp av denne teknologien.



Figur 3.1: Arkitektur for økonomisystemet Maconomy

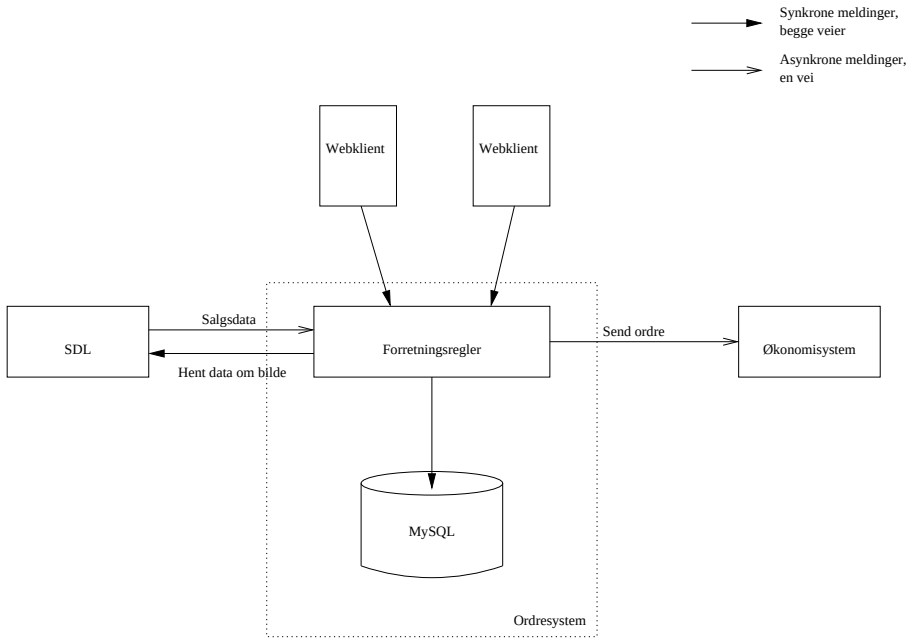
3.1.2.3 Ordresystem

De fleste bildene som selges av Scanpix idag, omsettes direkte fra SDL. Alt direktesalg fra SDL blir loggført, og tidligere måtte ansatte fra Scanpix gjennomgå disse loggene og skrive dem inn som ordre i økonomisystemet. For å korte ned denne arbeidsprosessen har IT-avdelingen i Scanpix utviklet et system for å håndtere disse ordrene. Vi skal referere til dette systemet som "ordresystemet". Ordresystemet hjelper de ansatte å fylle ut informasjon som skal inn i ordrene til økonomisystemet. Scanpix har også til hensikt å bruke ordresystemet til statistiske formål.

Ordresystemet er implementert i en lagdelt struktur. Nederst finner vi en MySQL-database hvor alle data om ordre blir lagret og håndtert. Over MySQL-databasen er det en hel del programmer som tar seg av å hente ut data om nedlastinger samt bildedata fra SDL. Videre finnes det en del funksjonalitet for å trekke ut bildedata og generere meningsfylte ordre ut fra dette.

Det trekkes jevnlig ut salgsdata fra SDL, mens bildedata hentes ut av SDL når en ordre skal genereres.

Ordrene blir lagret i MySQL-databasen og er tilgjengelige for brukere i et oversiktlig grensesnitt. Der kan brukerne forandre på ordrene og godkjenne dem for sending til økonomisystemet. Godkjente ordre sendes så til økonomisystemets server (se figur 3.1). For at økonomisystemet skal



Figur 3.2: Arkitektur for Scanpix' ordresystem

kunne ta imot ordre fra ordresystemet, er økonomisystemets funksjonalitet utvidet med Mscript.

Figur 3.2 viser arkitekturen til ordresystemet. Pilene viser dataflyten mellom systemene.

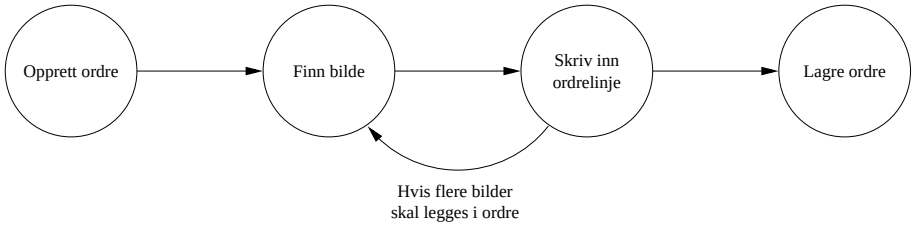
3.1.3 Selgere og push-salg

Det jobber flere selgere i Scanpix. Disse jobber aktivt med å sende bilder ut til kunder. Dette blir kalt push-salg, eller pushing av bilder.

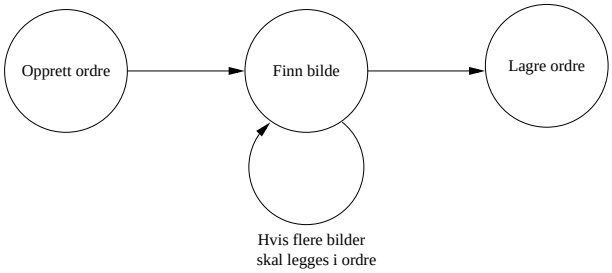
Bildene som pushes, kommer fra bildebyråer verden rundt. Selgerne bruker et program kalt Fotostation for å pushe bilder til kunder, der bildene sendes med FTP. Etter at bilder er pushet, legges de inn i SDL slik at de enkelt kan gjenfinnes. Det er kun en liten del av de pushede bildene som blir brukt, men det er bare disse Scanpix skal fakturere.

3.1.3.1 Om faktureringsprosessen

Siden de pushede bildene ikke selges direkte fra SDL, blir ikke disse salgene fanget opp av ordresystemet (og dermed heller ikke sendt videre til økonomisystemet). Derfor må selgerne på egen hånd fylle ordre inn i



Figur 3.3: Gammel arbeidsflyt ved fakturering



Figur 3.4: Ny arbeidsflyt ved fakturering

økonomisystemet. For hver ordre som skal lages, må selgerne søke i SDL etter de bildene som skal inngå i ordren. For hvert bilde som er funnet i SDL må det så skrives inn informasjon om bildet som linjer i ordren (ordrelinjer). Se figur 3.3 for arbeidsflyt.

3.1.3.2 Grunnlaget for et datasystem

I figur 3.3 er det særlig steget å skrive inn en ordrelinje som tar lang tid. Men all nødvendig informasjon for å fylle ut en ordrelinje ligger allerede i SDL. Hvis et datasystem kunne fylt ut linjene i en ordre, kunne selgerne produsere ordrene på langt kortere tid (se figur 3.4).

Vi skal kalle det nye datasystemet "push-systemet". Dette var kravene vi definerte for push-systemet:

> "Hovedmålet med push-systemet er å hjelpe selgere å fylle ut ordre på kortere tid. Systemet bør også kunne brukes til å trekke ut informasjon om pushing og salg av bilder. Videre må ordre fylt ut av push-systemet komme inn i økonomisystemet for å lage fakturaer, og helst også ordresystemet for statistiske hensyn."

3.2 Realisert løsning med mellomvare

I denne seksjonen skal vi gjennomgå det realiserte push-systemets design med de løsninger som er laget. Vi skal også se på hvordan arbeidet med systemet ble lagt opp.

3.2.1 Arbeidsform og utviklingsprosess

Selv om noen enkle krav til push-systemet var funnet, var det usikkert hvordan systemet burde bli bygget opp. Det var mange valgmuligheter gjennom alle deler av utviklingen, og dette gjorde at arbeidsformen var viktig.

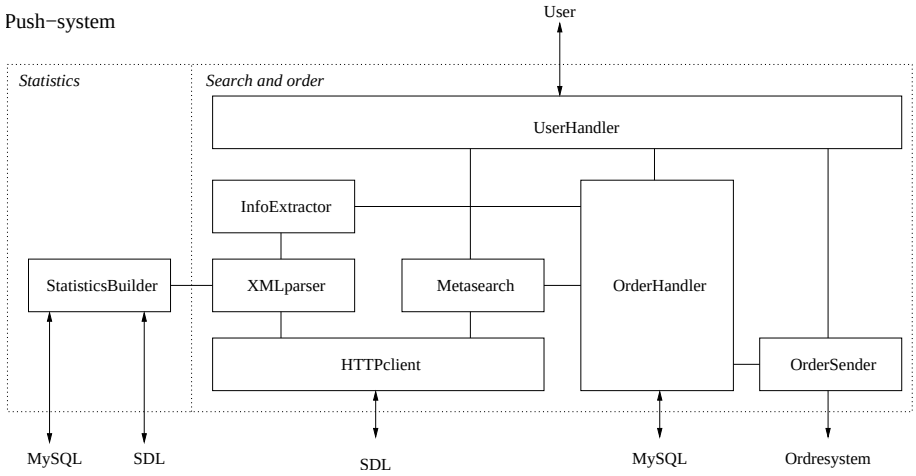
Siden det kun var selgere som hadde behov for å bruke push-systemet, var det viktig at de fikk være med å påvirke utformingen av systemet. Utviklingen skjedde derfor i nært samarbeid med selgerne for å skjønne hvordan de arbeidet og hvordan vi kunne få en velfungerende løsning. Gjennom hele utviklingen ble det vurdert hvordan hver del av systemet best kunne utformes, deretter ble det implementert en prototyp som raskt ble testet av selgerne.

Denne arbeidsformen viste seg å fungere bra tatt i betraktning at vi ikke hadde noen klart definerte spesifikasjoner for hvordan systemet skulle bygges opp.

3.2.2 Systembeskrivelse

3.2.2.1 Valg av tekniske løsninger

Det var et poeng å bruke minst mulig tid på implementasjon av push-systemet. Videre var det et behov for å benytte seg av SQL databaser for å bygge opp statistikk rundt push-salgene og for å arbeide med ordre som ikke var sendt videre til ordresystemet og økonomisystemet. Scanpix har en webserver som gir kunder tilgang til SDL og ansatte tilgang til ordresystemet. Det var et naturlig valg å utvikle push-systemet på denne serveren. Serveren har SQL-databasen MySQL installert, og blant annet skriptespråkene Perl og PHP. Jeg valgte å bruke skriptespråket PHP til utviklingen av selve integrasjonslaget, mens MySQL ble den naturlige løsningen for SQL-database. Siden PHP er et web-basert skriptespråk, tilbyr push-systemet et web-basert brukergrensesnitt som fungerer i en vanlig nettleser.



Figur 3.5: Logisk oppbygging av moduler i Push-systemet.

3.2.2.2 Moduler

Det realiserte push-systemet har to hoveddeler. Disse har jeg kalt "Statistics" og "Search and order" (se figur 3.5). "Statistics" inneholder kun én modul, StatisticsBuilder, mens "Search and order" inneholder mange moduler med forskjellige ansvarsområder. "Statistics" har ingen brukere. Det eneste ansvaret denne delen har, er å bygge opp en database over push-salg slik at vi senere kan foreta spørringer mot denne databasen.

Legg merke til at push-systemet ikke har noen spesiell modul for kommunikasjon med MySQL-databasen. Dette kommer av at PHP allerede er tett integrert med MySQL, og noen ekstra modul for dette blir overflødig.

Vi skal nå se på de forskjellige modulene.

StatisticsBuilder. Dette er en modul som hele tiden trekker ut informasjon om nylig pushede bilder fra SDL. I SDL ligger informasjon om push-salg svært ustrukturert, og det er derfor vanskelig å trekke ut noen statistikk på dette området. StatisticsBuilder trekker derfor ut den informasjonen som er relevant fra SDL og legger den mer strukturert i en MySQL-database. Det er poeng å ha data om push-salg i en SQL-database, da SQL tilbyr et langt rikere spørrespråk enn det man har tilgang til i SDL.

StatisticsBuilder benytter modulene XMLparser og HTTPclient for å trekke ut data fra SDL.

HTTPclient. SDL er i hovedsak designet for å kommunisere med nettlesere over HTTP. For enkelt å kommunisere med SDL, bruker vi modulen HTTPclient som fungerer som en HTTP-klient på lik linje med en nettleser. HTTPClient tar imot en URL[4], og returnerer det tekstlige innholdet som returneres av SDLs webserver. Kommunikasjonen mellom HTTPclient og SDL foregår via Internettet, og for å hindre at andre skal kunne misbruke SDL, benytter modulen vanlig HTTP-autentisering mot SDL.

XMLparser. Data hentet med HTTPclient er alltid tekst, og ofte er denne teksten XML. For å gjøre om dataene i XML til en datatruktur som lett kan brukes i PHP, bruker vi en XML-parser.

Denne parseren har vist seg å være relativt treg, men det er særlig StatisticsBuilder-modulen som bruker XML-parseren, og her er ikke hastighet viktig.

InfoExtractor. Som nevnt, var målet for push-systemet at ordrelinjer skulle fylles ut automatisk og dermed forenkle arbeidet til selgerne (som vist i figur 3.4 på side 22). Relevant informasjon som må legges inn i en ordrelinje, er SDL's idnummer på bildet, bildetekst (caption), fotograf, med mer.

Problemet med å la et datasystem hente inn nødvendig bildeinformasjon, er at bildeinformasjonen ligger svært ustrukturert i SDL. Bildene i SDL kommer som nevnt fra mange forskjellige byråer, og disse følger hver sin standard for å lagre bildeinformasjon. Løsningen var å implementere et lite bibliotek som har til oppgave å finne all relevant informasjon fra et bilde. Informasjonen hentes ut i to steg:

1. InfoExtractor finner ut hvilket byrå bildet kommer fra.
2. Avhengig av byrået bildet kommer fra, henter InfoExtractor ut informasjon fra de forskjellige feltene i SDL.

Denne løsningen fungerer bra, men den har noen ulemper. Blant disse er at hver gang Scanpix tar inn bilder fra et nytt byrå, må biblioteket i push-systemet oppdateres. Det samme gjelder om et byrå forandrer måten de legger bildeinformasjon til bilder.

Metasearch. Som vi kan se fra arbeidsflyten i figur 3.3 på side 22 og 3.4 på side 22, må selgerne kunne søke etter bilder når de lager ordre. Derfor

[4]Uniform Resource Locator

lagde jeg en modul som gjør at brukere kan søke etter bilder rett fra SDL. For å få til dette, bruker Metasearch modulen HTTPclient. Fra brukerne tar Metasearch imot søkestrenger som brukes til å bygge opp en URL for å gjøre søk i SDL. SDL returnerer søkeresultatene på et fast tekstformat som Metasearch tolker og bruker til å bygge opp et oversiktlig resultat med småbilder og tekst. Når disse søkene gjøres i SDL, returneres ikke resultatene i XML format. En grunn til dette er at søkene helst skal gå så fort som mulig, og både SDL og Metasearch ville brukt lengre tid hvis XML først skulle genereres og så parses før søkeresultatene kunne vises. Det har vist seg at Metasearch bruker svært lite tid i forhold til de andre oppgavene (søk i SDL, nedlasting av thumbnails) som utføres ved søk.

OrderHandler. Vi så at det var behov for å arbeide med ordre (opprette, legge til ordrelinjer, oppdatere informasjon) før de skulle sendes videre til ordresystemet og økonomisystemet. Løsningen ble derfor å lage en database i MySQL hvor alle ordre lå lagret. OrderHandler er en samling med skript som brukes til alle deler av arbeidet med ordre og ordrelinjer før de sendes.

For å legge til nye bilder i en ordre kommuniserer OrderHandler med InfoExtractor for å hente ut all informasjon til en ordrelinje.

Push-systemet må sørge for at ordre er korrekt utfylt før de sendes. Grunnen til dette skal vi se på i neste avsnitt. En del av OrderHandler har fått ansvaret for å kontrollere ordrene. Her sjekkes det at de er korrekt utfylt i forhold til regler som finnes i både ordresystemet og økonomisystemet.

Denne løsningen fungerer godt, men kan være uheldig hvis man ikke har kontroll på regler som gjelder i de andre systemene, særlig ordresystemet hvor ordrene i første omgang skal sendes.

OrderSender. Denne modulen har ansvar for å legge ordre fra push-systemets database over til ordresystemet og økonomisystemet. Å legge ordre inn i økonomisystemet ville være relativt vanskelig fordi økonomisystemets servere måtte bli utvidet med Mscript for å ta imot ordre fra push-systemet. Disse serverne ligger i Danmark (driftet og forvaltet av Scanpix Danmark), og det ville være vanskelig å begynne med utviklingsarbeid på disse. Det vi så, var at ordresystemet allerede hadde funksjonalitet for å legge ordre inn i økonomisystemet på samme måten som push-systemet skulle gjøre. Derfor ble den naturlige løsningen å legge ordre fra push-systemet inn i ordresystemet, og la ordresystemet ta seg av kommunikasjon med økonomisystemet.

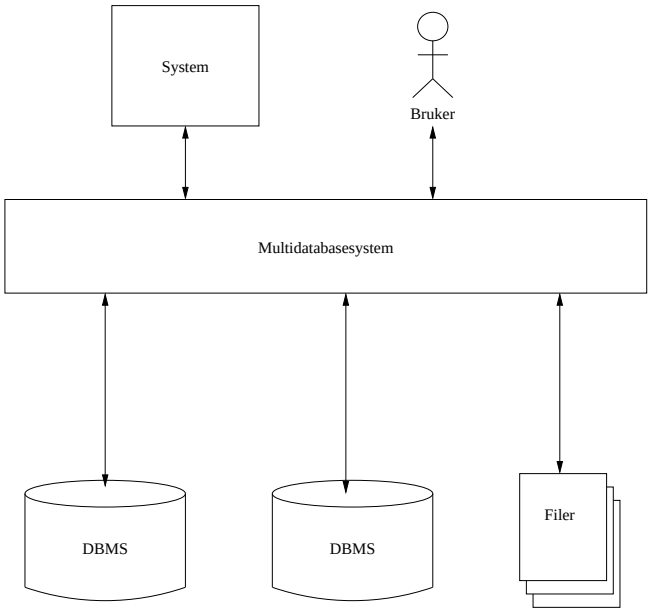
Som vi har sett, har ordresystemet ordrene sine lagret i en MySQL database.

Figur 3.6: Utdrag fra brukergrensesnittet dannet av UserHandler

For å slippe å gjøre om noe i ordresystemet, valgte vi å legge ordredata fra push-systemet rett inn i ordresystemets database. Det er lite regler implementert i selve databasen, og derfor fikk push-systemet ansvaret for kun å legge inn ordre som var korrekt utfylt. Regler for dette ble implementert i OrderHandler, så OrderSender bruker denne modulen for å sjekke at ordren er korrekt utfylt før den legges inn i ordresystemets database.

Den MySQL versjonen som brukes i Scanpix' systemer, tilbyr ikke transaksjoner. Derfor logges alle overføringer av ordre fra push-systemet til ordresystemet slik at det det skal være lett å gå tilbake dersom det skulle skje noe under under en overføring.

UserHandler UserHandler har det overordnede ansvaret for å gi brukerne et godt grensesnitt å arbeide i. Når dynamiske elementer (for eksempel søkeresultater fra SDL) skal vises, gir UserHandler midlertidig bort ansvaret til underliggende moduler (for eksempel Metasearch som har ansvaret for søking i SDL).



Figur 3.7: Modell for et multidatabasesystem

3.2.3 Mellomvare eller multidatabasesystem?

Systemet jeg har beskrevet i denne seksjonen kan på mange måter likne et multidatabasesystem. Med multidatabasesystem mener vi et system der vi har flere underliggende databaser, gjerne av forskjellig type, og disse er autonome (de har ingen kjennskap til hverandre). Typisk har man et programvarelag som har ansvar for all kommunikasjon mot de underliggende databasene, og som må ta seg av regler for integritet, transaksjonshåndtering og så videre. En typisk modell for et multidatabasesystem er vist på figur 3.7.

Denne beskrivelsen stemmer godt med det implementerte push-systemet som tar seg av kommunikasjon med SDL, flere MySQL databaser og filsystemer der alle disse systemene er uavhengige av hverandre. Så hvorfor kaller jeg push-systemet for mellomvare, og ikke et multidatabasesystem?

Som forklart tidligere i dette kapittelet, søker push-systemet i bildedata fra SDL, samt at bildedata hentes ut fra SDL. Vi ser her at det kun hentes data fra SDL; push-systemet legger aldri inn eller oppdaterer noe i SDL. Videre kan vi se at ordresystemet kun mottar data fra push-systemet, og push-systemet kan ikke lese eller oppdatere data når de er lagt inn.

Vi ser dermed at det kun flyter data én vei, nemlig fra SDL, innom push-systemets databaser, og videre til ordresystemet. I multidatabaser må databasesystemet ta ansvar for å bestemme hvor data skal hentes ut (da

det kan hentes fra flere databaser) og hvor de eventuelt skal lagres og oppdateres. I dette tilfellet er det aldri nødvendig med noen logikk for å avgjøre hvor data skal hentes fra eller skrives til. Derfor har jeg valgt å ikke kalle push-systemet et multidatabasesystem.

3.3 Tenkt løsning med tjenesteorientert arkitektur

I denne delen skal vi se på hvordan en løsning med tjenesteorientert arkitektur kunne vært brukt for å løse systemkravene beskrevet i kapittel 3.1 på side 17.

3.3.1 Overordnet arkitektur

Det finnes flere måter å tilnærme seg en tenkt løsning for en tjenesteorientert arkitektur. To av dem er:

1. Vi kan tenke oss at de eksisterende systemene ikke er utviklet med tanke på tjenesteorientert arkitektur, men at vi kan "pakke inn" systemene slik at de kan fremstå som tjenester. Vi vil dermed få en tjenesteorientert arkitektur. Denne måten å integrere systemer på kalles "Service Oriented Integration" (Newcomer and Lomow, 2004).
2. Alle de eksisterende systemene i Scanpix kan være utviklet som tjenester for en tjenesteorientert arkitektur.

Ifølge (Newcomer and Lomow, 2004) er en av de store fordelene med tjenesteorientert arkitektur at vi kan danne nye tjenester ved å sette sammen eksisterende tjenester. Derfor ser jeg det som hensiktsmessig at vi tar utgangspunkt i alternativ 2, og antar at alle de eksisterende systemene støtter Web Services og er satt sammen i en tjenesteorientert arkitektur.

For at en tjenesteorientert arkitektur skal være lettest mulig å håndtere og utvide, mener jeg det er hensiktsmessig å samle alle beskrivelser av tjenester i en opplysningstjeneste (se kapittel 2.3.1 på side 8 og figur 2.4 på side 10). Vi skal ikke definere spesielt hva slags opplysningstjeneste dette er, utenom at den fungerer som en felles database for alle tjenestebeskrivelser.

Hensikten med en opplysningstjeneste er å holde orden på tjenester samt gjøre det mulig for tjenestesøkere å oppdage tjenestetilbydere. Dette er ikke så relevant i vårt tilfelle, da systemene i Scanpix har et begrenset omfang. Et viktigere poeng er at det blir lettere å bruke et globalt skjema for alle tjenestene i hele arkitekturen. Selv om vi skal gå ut i fra at vi bruker en opplysningstjeneste i den tjenesteorienterte arkitekturen, kommer vi ikke

til å vise den i de figurene som følger senere i kapittelet. Dette er for å holde figurene så enkle som mulig.

En viktig årsak til at vi skal bruke et globalt skjema, er at vi vil ha unike navn for alle elementer som kan utveksles mellom systemer. Et eksempel på et problem som kan oppstå uten et globalt skjema, er at en tjeneste bruker navnet "date" for når bildet ble lagret i systemet, imens et annet system kan bruke "date" for når bildet ble tatt, eller dato for når en ordre ble opprettet i ordresystemet.

Siden vi går ut i fra at alle tjenestene i Scanpix er utviklet i henhold til en samlet tjenesteorientert arkitektur, kan vi regne med at navnbruk og mening (semantikk) er konsistent for alle de eksisterende systemene.

Et mål med tjenesteorientert arkitektur er å integrere systemer på en løs måte, slik at de

1. lettere kan byttes ut, eller
2. lett kan integreres med andre systemer

I vårt tilfelle skal vi se hvordan systemer lett kan integreres med andre ved hjelp av tjenesteorientert arkitektur. For å få frem dette vil jeg i størst mulig grad gjenbruke funksjonalitet som finnes i de ekisterende tjenestene. Før vi skal se hvordan push-systemet kunne blitt realisert i tjenesteorientert arkitektur, skal vi se hvordan de eksisterende tjenestene må fungere sammen.

3.3.2 Eksisterende tjenester

Vi skal her beskrive de tjenester som tilsvarer systemer som allerede eksisterer i Scanpix, og hvordan disse må fungere for at de skal utføre det samme som i virkeligheten.

3.3.2.1 SDL

Hvis SDL er laget med støtte for tjenesteorientert arkitektur, kan vi tenke oss at tjenesten må tilby disse operasjonene for at ordreystemet skal fungere:

Generer informasjon om salg av bilder. For at ordresystemet (og eventuelle andre systemer) skal kunne generere ordre eller liknende, må SDL levere informasjon om alle salg som blir gjort. Denne informasjonen må jevnlig sendes til ordresystemet, men det er ikke nødvendig at

dette skjer veldig raskt eller på noe spesielt tidspunkt. Hvis informasjon om bildesalg skal brukes av mange systemer, kan vi vurdere om publish/subcribe er den beste måten å fordele denne informasjonen på. Nå har ikke Scanpix så mange forskjellige tjenester, og antakelig bare én tjeneste (ordresystemet) som kan dra nytte av salgsinformasjon. Derfor mener jeg persistente køer med salgsinformasjon er det beste handlingsmønsteret for denne operasjonen.

Hent bildeinformasjon. Når selgere leter etter bilder, har de ofte behov for å se all informasjon som er lagret om et bilde. Dette er ikke tilgjengelig direkte fra søkeresultatet, og det er derfor nødvendig at SDL kan ta imot et id-nummer for et bilde og returnere all bildeinformasjon. I tillegg er det nødvendig for ordresystemet å hente ut bildeinformasjon for å generere meningsfulle ordre. Som for bildesøk, mener jeg det er mest hensiktsmessig med request/response handlingsmønster her.

3.3.2.2 Ordresystem

Ordresystemet må ta imot en del meldinger og sende en del meldinger. Disse omfatter:

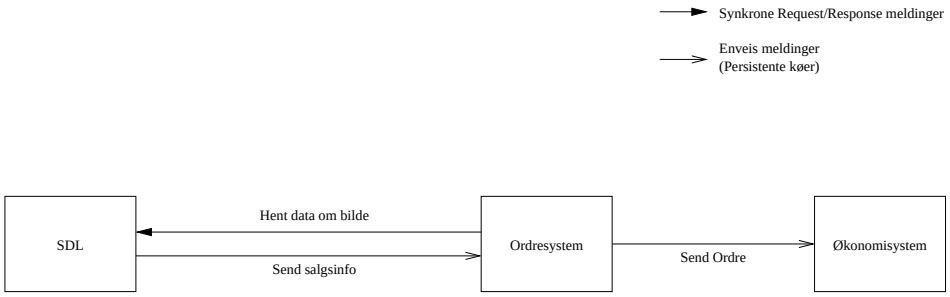
1. Å ta imot enveismeldinger fra SDL om bildesalg
2. Å sende request til SDL om bildedata og motta disse
3. Å sende enveismeldinger til økonomisystemet med ferdige ordre som skal faktureres

All denne dataflyten foregår også i den implementerte løsningen av ordresystemet, men vi tenker oss nå at det går via Web Services (se figur 3.8 på neste side).

Ellers må ordresystemet ha sin vanlige funksjonalitet som beskrevet i kapittel 3.1.2.3 på side 20. Dette innebærer at systemet har en database over ordre, at systemet tilbyr et brukergrensesnitt slik at brukere kan forandre og sende ordre til økonomisystemet, og at ordresystemet automatisk fyller ut ordrene med informasjon trukket ut fra bildedata. Vi ser her at denne funksjonaliteten i stor grad likner på funksjonaliteten som ble implementert i push-systemet beskrevet i kapittel 3.2 på side 23.

3.3.2.3 Økonomisystem

Som nevnt tidligere, fungerer allerede ordresystemet og økønomisystemet med hverandre, men vi skal se raskt på hva slags tjeneste økonomisystemet



Figur 3.8: Tjenesteorientert arkitektur med dataflyt for ordresystem

må være for å fungere på tilsvarende måte med Web Services.

Økonomisystemet har mange oppgaver, blant annet å kommunisere med klienter, lage fakturaer osv, men i vårt tilfelle er det bare én tjeneste som er relevant. Det er å ta imot ordre, enten fra ordresystemet eller et annet system om man måtte ønske det. Jeg ser det som mest hensiktsmessig at økønomisystemet tar imot ordrene som asynkrone meldinger. På denne måten behøver ikke andre systemer som skal levere ordre å vente på noe signal eller liknende fra økonomisystemet. For å være sikker på at alle meldinger kommer frem, er det hensiktsmessig å bruke Web Services-utvidelsene WS-ReliableMessaging eller WS-Reliability (se kapittel 2.3.1.2 på side 9).

Økonomisystemet er som nevnt av typen Maconomy som skal ha implementert støtte for Web Services allerede, så å bruke økonomisystemet i en tjenesteorientert arkitektur bør ikke være særlig krevende.

3.3.3 Nye tjenester

Vi har nå beskrevet hvordan systemene kunne fungere sammen i en tjenesteorientert arkitektur før vi begynte å implementere et push-system. For å kunne implementere et push-system som bruker de tjenestene som ble beskrevet ovenfor, må noen av tjenestene tilby flere operasjoner enn de vi har gjennomgått. Disse operasjonene gjelder tjenestene SDL og ordresystem. Vi skal nå gjennomgå alle operasjonene disse må tilby, og så se hvordan push-systemet kan realiseres.

3.3.3.1 Oppdatering av eksisterende tjenester

SDL. Ut i fra spesifikasjonene for hva push-systemet skal gjøre, må SDL tilby to nye operasjoner:

Bildesøk i SDL. Bildesøk i SDL tilsvarer et vanlig fritekstsøk som SDLs brukere vanligvis kan gjøre. Dette må push-systemet ha mulighet til fordi brukerne av systemet må ha mulighet til å lokalisere bilder i SDL på egenhånd. Her kan det poengteres at andre systemer fra før av kan søke i SDL, og vi kan dermed anta at systemet er klargjort til å brukes i en tjenesteorientert arkitektur med unntak av støtte for Web Services. Bildesøk vil også få en Request/Response meldingsutveksling der det sendes en forespørsel fra et et annet system med en søkestreng, og hvor søkeresultatet er responsen.

Hente ut data om pushing av bilder. Vi har tidligere sett at Scanpix ønsket å bygge opp statistikk over push-salg av bilder. For å få til dette må det jevnlig hentes ut data om push-salg fra SDL. Dette kan i likhet med data om salg av bilder (se kapittel 3.3.2.1 på side 30) gjøres på to måter, enten ved Publish/Subscribe modellen, eller ved hjelp av asynkrone enveismeldinger. Igjen har jeg vurdert hvorvidt data om push-salg av bilder vil være hensiktsmessig å sende til mange systemer, noe jeg ikke kom frem til at det var. Derfor har jeg valgt å la SDL sende enveismeldinger med data om push-salg til StatisticsBuilder.

Ordresystem. Som vi så i forrige seksjon om hvilken funksjonalitet som finnes i ordresystemet, mener jeg det må være mulig å legge alt arbeidet angående håndtering av ordre til ordresystemet. Dermed må tjenesten ordresystem tilby følgende operasjoner for ordrebehandling:

1. Opprett ny ordre
2. Legg bilde til ordre
3. Fyll ut ordreinformasjon automatisk
4. Forandre ordre
5. Sjekk at ordre er korrekt utfylt
6. Send ordre til økonomisystem

Dette er operasjoner som ble implementert i modulene InfoExtractor, OrderHandler og OrderSender i den fungerende løsningen. Hvis vi heller lar ordresystemet få ansvaret for all behandling av ordre, blir push-systemet mye enklere å implementere. Push-systemet må i så fall kunne kalle alle disse operasjonene med request/response meldinger. Her mener jeg det er hensiktsmessig å bruke request/response interaksjon fordi push-systemet hele tiden skal gi tilbakemeldinger til brukere når noe gjøres, og en respons fra ordresystemet er derfor naturlig.

Vi kan merke oss at for å legge et bilde til en ordre, kreves det at ordresystemet henter ut relevant bildeinformasjon om det aktuelle bildet fra SDL. Dette fremgår av request/response meldingen "Hent data om bilde" i figur 3.9 på side 36.

Økonomisystemet. Siden vi bruker Web Services for å utveksle data mellom systemene, bør det ikke være vanskelig å la push-systemet sende ordre til økonomisystemet. Allikevel mener jeg det er mer hensiktsmessig å la ordresystemet sende disse ordrene. Grunnen er at ordresystemet allerede er klargjort for dette, og det er mer naturlig at ordrene går rett fra ordresystemet til økonomisystemet, enn at de går fra ordresystemer til push-systemet før de går til økonomisystemet.

3.3.3.2 Push-systemet

Den foreslåtte løsningen for push-systemet skiller seg mest ut fra den realiserte løsningen ved at vi lagrer ordre i ordresystemet, og at vi bruker ordresystemets regler for ordrebehandling. Dette gjør at vi slipper å implementere mye funksjonalitet som ble laget i den realiserte løsningen. Hvis vi ser tilbake på figur 3.5 på side 24, kan vi se at ved å la ordresystemet ta seg av all ordrebehandling, kan vi fjerne hele den store OrderHandler modulen. Siden ordrene allerede vil befinne seg i ordresystemet, trenger vi heller ikke modulen OrderSender.

Siden Web Services tilbyr enkel kommunikasjon med alle tjenester, trenger vi ikke modulene HTTPClient og XMLParser for å kommunisere med SDL. Modulen InfoExtractor gjør samme jobb som utføres når ordre genereres i ordresystemet. Derfor kan vi også utelate denne.

Vi ser at de eneste modulene i push-systemet som ikke kan utelates, er StatisticsBuilder og UserHandler. Jeg ser det derfor som logisk å danne to tjenester, StatisticsBuilder for statistikk og UserHandler for alt som er i sammenheng med ordre. Vi skal nå se på disse to tjenestene.

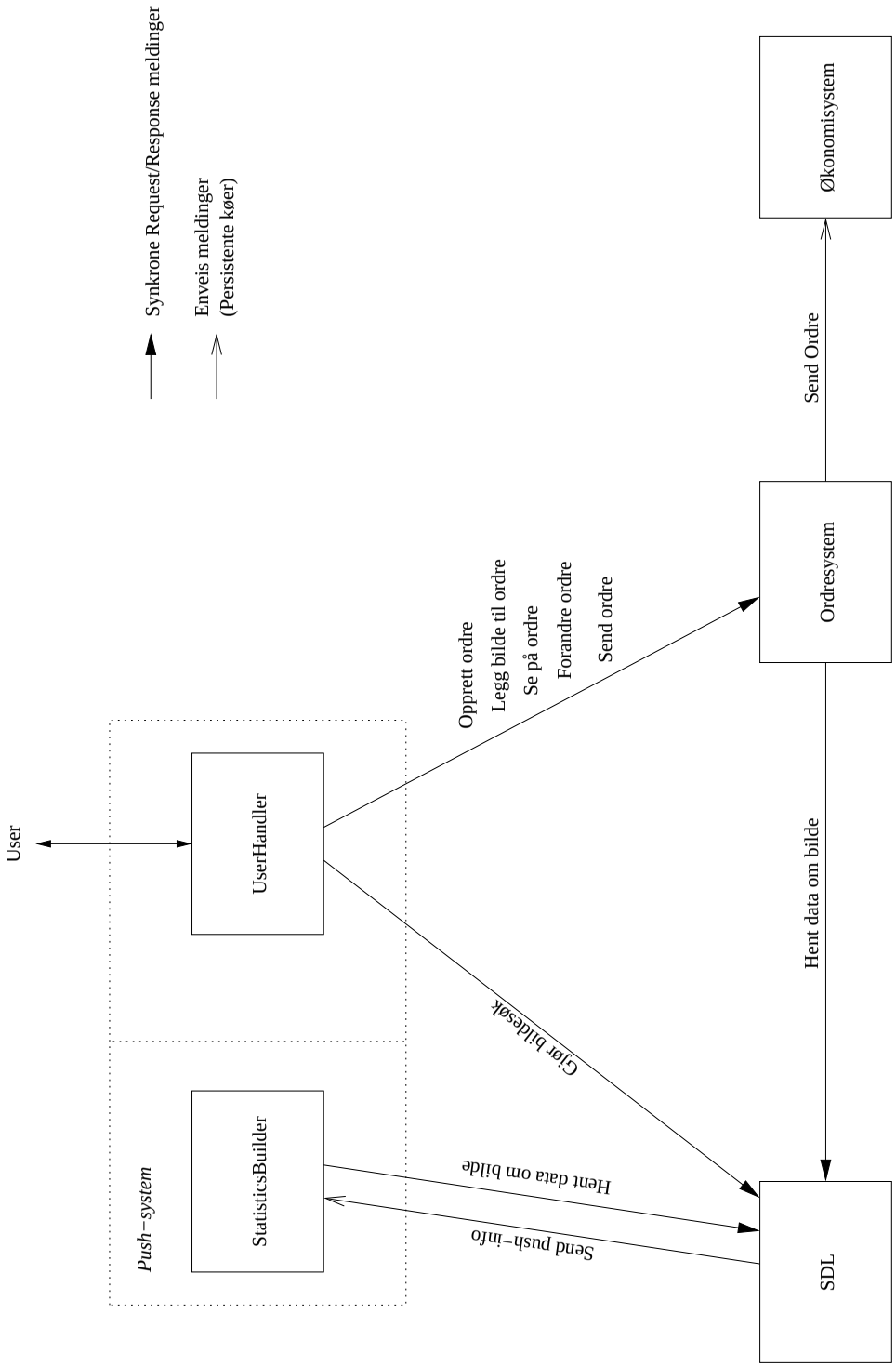
StatisticsBuilder. StatisticsBuilder skal utføre det samme som Statistics-Builder-modulen gjorde i det fungerende systemet. Her må det altså jevnlig tas imot informasjon om push-salg av bilder, det må hentes ut bildeinformasjon om hvert pushede bilde fra SDL, og dette må legges i en database hvor man kan hente ut statistikk og drive datamining. Den eneste forskjellen består i måten data flyter mellom SDL og StatisticsBuilder. I det realiserte designet skjedde det ved at filer ble lest inn, og SDL ble kalt via HTTP, her må det skje med asynkrone meldinger og request/response

meldinger. De asynkrone meldingene er beskrevet som en av de nye operasjonene tjenesten SDL måtte tilby. For å hente ut data om bilder, kan StatisticsBuilder bruke samme operasjon som ordresystemet bruker for å hente ut data.

UserHandler. Som vi så i kapittel 3.2 på side 23, ble "Search and Order" delen av push-systemet et relativt komplekst system med mange moduler.

Vi ser nå at all funksjonaliteten i modulene som fantes i "Search and Order" faktisk er implementert i ordresystemet allerede. Dermed behøver vi ikke implementere stort mer enn den funksjonaliteten som fantes i modulen UserHandler (se figur 3.5 på side 24). Dette burde gjøre imlementasjonen av "Search and Order" delen i push-systemet langt lettere enn det den viste seg å bli i den realiserte løsningen.

Figur 3.9 på neste side viser dataflyten mellom de forskjellige tjenestene.



Figur 3.9: Tjenesteorientert arkitektur med dataflyt for push-system

Kapittel 4

Metodikk

I dette kapittelet skal vi se på fremgangsmåten jeg har valgt for å sammenlikne mellomvarearkitektur med tjenesteorientert arkitektur for integrasjon av systemer.

4.1 Fremgangsmåte

Målet med undersøkelsen i denne oppgaven er å se om henholdsvis mellomvarearkitektur og tjenesteorientert arkitektur er egnet til integrasjon av datasystemer. Som case har vi bedriften Scanpix der det skulle gjennomføres en systemintegrasjon. For denne systemintegrasjonen ble det utviklet en mellomvarearkitekturløsning, og vi designet en tjenesteorientert arkitektur-løsning (TA-løsning).

Siden mellomvarearkitekturløsningen er en realisert løsning, har vi grunnlag til å trekke slutninger om hvor godt mellomvarearkitektur er egnet for integrasjon i casen. TA-løsningen ble ikke realisert, men arbeidet med mellomvarearkitekturløsningen har gitt grunnlag til å vurdere mange aspekter som er relevante for en systemintegrasjon med tjenesteorientert arkitektur.

For best å vurdere hvor egnede de to arkitekturene er i casen, skal vi sette opp en del kriterier vi kan vurdere dem etter. Ved å vurdere arkitekturene etter de samme kriteriene, er det lettere å sammenlikne dem og se hvor de skiller seg fra hverandre. Vi skal gå nærmere gjennom vurderingskriteriene i kapittel 4.2 på neste side.

Etter at vi har studert arkitekturene med hensyn på løsningene i casen, skal vi bruke de samme vurderingskriteriene for å identifisere generelle forskjeller mellom arkitekturene.

4.2 Vurderingskriterier

Som nevnt, er det hensiktsmessig å sette opp en del vurderingskriterier for å vurdere de to arkitekturene. Her følger noen:

Arbeidsmengde og kompleksitet Det er et mål at man legger ned minst mulig arbeid i å integrere datasystemer. Samtidig ønsker vi at integrasjonen fører til minst mulig teknisk komplekse løsninger.

Vedlikeholdbarhet. Vi ønsker at systemintegrasjonen skal være enkel å vedlikeholde for fagpersonell. Derfor er det viktig at arkitekturen er enkel å forstå, og samtidig enkel å forandre og utvide.

Krav til kompetanse. Er det nødvendig med noen spesiell kompetanse hos fagpersonell for at de skal kunne forandre arkitekturen eller forandre systemer som er en del av arkitekturen?

Gjenbrukbarhet. Å drive nyutvikling av programvare er regnet som dyrt, og vi ønsker derfor å kunne gjenbruke programvare.

Opplæringsbehov. Medfører arkitekturen noe spesielt behov for opplæring? Dette kan gjelde opplæring av brukere eller av fagpersonell som skal forvalte eller drifte systemene.

Drift. Krever arkitekturen spesiell behandling, som nedetid på noen tidspunkt, backup, ressursbehov eller liknende?

Ytelse. Gir arkitekturen tilfredsstillende ytelse for en løsning? Ytelse er et samlebegrep for flere problemer som har med systemers eller løsningers evne til å løse oppgaver. Eksempler på ytelse er mål på responstid, kapasitet til å takle last fra et antall samtidige brukere eller kapasitet til å overføre/behandle en mengde data innenfor et tidsrom.

Skalerbarhet. Sørger arkitekturen for at systemene kan takle uforutsett vekst i trafikk fra brukere eller systemer? Dette kan være viktig i mange tilfeller.

Stabilitet. Kan arkitekturen tilby stabilitet av noen spesiell grad?

Sikkerhet. Støtter arkitekturen former for sikkerhet som kryptering og autentisering?

Ikke alle kriteriene er like relevante, og noen overlapper hverandre. Vi skal se nærmere på kriteriene “arbeidsmengde og kompleksitet”, “gjenbrukbarhet”, “vedlikeholdbarhet” og “ytelse”. Jeg mener disse er egnet til å vurdere begge arkitekturene i casen, samtidig som de er viktige kriterier hvis vi skal vurdere arkitekturene mer generelt.

Kapittel 5

Observasjoner/Funn

Vi skal i dette kapittelet vurdere hvor godt hver arkitektur er egnet for systemintegrasjonen som ble gjort i casen. For å gjøre dette, skal vi studere arkitekturene med hensyn på vurderingskriteriene vi kom frem til i kapittel 4.2. Til slutt i kapittelet skal vi se kort på de vurderingskriteriene jeg har valgt bort, og hvorfor vi ikke skal gå nærmere inn på disse.

5.1 Arbeidsmengde og kompleksitet

Arbeidsmengde og kompleksitet er to faktorer som er sterkt relatert, og vi skal derfor vurdere de sammen i dette kapittelet. Når vi tenker på arbeidsmengde, sikter vi til at utviklingen av en løsning ikke skal medføre for mye arbeid. Dessuten ønsker vi at løsningene skal være minst mulig teknisk komplekse.

5.1.1 Mellomvarearkitektur

Da jeg skulle implementere push-systemet, la jeg særlig vekt på to ting: At vi ikke skulle forandre de eksisterende systemene mer enn nødvendig, og at systemet skulle tilby kun ett brukergrensesnitt som var enkelt og samtidig effektivt.

Mens push-systemet var under utvikling, så vi hele tiden etter hva selgerne hadde behov for, og hva andre systemer krevde (for eksempel hvordan ordre måtte være utfylt før de kunne legges inn i ordreystemet). Deretter implementerte jeg prototyper som raskt kunne testes ut. Prototypene ble hele tiden tilpasset de systemene de skulle kommunisere med, enten de skulle lese inn filer, kommunisere med HTTP, eller kommunisere med en

MySQL-database. Denne måten å arbeide på fungerte bra, og jeg mener løsningen ikke var spesielt vanskelig å implementere.

Riktignok opplevde jeg problemer med at andre systemer (her SDL) ikke alltid oppførte seg som ventet. Slike feil var ikke dokumentert noe spesielt sted, og var vanskelig å forstå i utviklingsprosessen. Under utviklingen var det problematisk nok å finne egne feil. Dette problemet var et resultat av designet jeg hadde valgt. Som sagt ønsket jeg å forandre de eksisterende systemene i minst mulig grad, og heller skreddersy mellomvaren til de underliggende systemene. Dette medførte at jeg måtte implementere feilhåndtering i mellomvaren, i tilfelle systemene oppførte seg på en annen måte enn det man regnet med. At mellomvaren måtte tilpasses hvert av de underliggende systemenes grensesnitt var også tungvint.

Jeg mener ikke at mellomvarearkitekturløsningen krevde mye arbeid, men arbeidet med å utvikle forskjellige typer grensesnitt, samt arbeid med å lage funksjonalitet for å behandle ordre (se kapittel 3.2.2.2 på side 26) tok en del tid. På den tekniske siden måtte jeg sette meg inn i flere protokoller for kommunikasjon mellom systemene, noe som igjen førte til lengre utviklingstid.

5.1.2 Tjenesteorientert arkitektur

Som det ble beskrevet i kapittel 3.3.1 på side 29, antar vi at de eksisterende systemene i Scanpix allerede har støtte for Web Services, og at de tilbyr operasjoner som beskrevet i kapittel 3.3.2 på side 30.

Å utvikle selve push-systemet (se figur 3.9 på side 36) ville ikke medføre særlig mye arbeid. På grunn av Web Services ville vi ikke ha behov for å implementere noen kommunikasjonsmoduler, og hvis vi kunne la all behandling av ordre skje i ordresystsemet istedet for i push-systemet, ville det ikke bli nødvendig å implementere noen modul for å håndtere ordre. På denne siden ser vi at vi kan spare mye arbeid i en TA-løsning.

På den annen side ser vi at flere tjenester må oppdateres for at push-systemet skal kunne bruke dem (se kapittel 3.3.3.1 på side 32). Det kan være vanskelig å si hvor mye arbeid dette ville medføre, men jeg mener det ville gi mindre arbeid å oppdatere for eksempel ordresystemet slik at det tilbød operasjoner for behandling av ordre enn arbeidet som måtte utføres for at push-systemet selv kunne behandle ordre. Et problem her kan være at de eksisterende systemene er implementert på forskjellige plattformer. Dette medfører at vi har behov for kompetanse innen de plattformer der det er tjenester som må oppdateres.

Når det gjelder tekniske løsninger, mener jeg vi ville fått en enklere løsning

med tjenesteorientert arkitektur. Web Services ville tilby en standard for kommunikasjon mellom alle tjenester, slik at vi ikke ville behøve å spesialtilpasse noen grensesnitt. Videre ville vi vært helt frie til å velge teknologi push-systemet skulle implementeres i, da Web Services tilbyr kommunikasjon helt uavhengig av plattform. Vi må ikke glemme at selv om vi står helt fritt til å velge teknologi, er det begrenset hvor mange som tilbyr god støtte for Web Services på dette tidspunktet (april 2006).

Feilhåndteringen beskrevet i kapittel 5.1.1 på side 39 mener jeg burde vært lagt i størst mulig grad til hver enkelt tjeneste, slik at disse ville tilby størst mulig tjenestekvalitet (Quality of Service). Web Services' basisstandarder ville nok ikke hjelpe mot grove feil som systemkrasj eller at nettverket ikke fungerer, men utvidelser som WS-ReliableMessaging (se kapittel 2.3.1.2 på side 9) eller liknende kan hjelpe i deler av TA-løsningen.

5.1.3 Oppsummering

Hverken mellomvarearkitekturløsningen eller TA-løsningen ville medført mye arbeid. Mellomvarearkitekturen medførte mer arbeid med å utvikle protokoller og grensesnitt for kommunikasjon mellom systemene, samtidig som en del kode måtte utvikles fra bunn (blant annet "behandling av ordre"). Den tjenesteorienterte arkitekturen ville gitt oss mer arbeid med å utvide eksisterende tjenester, men vi ville slippe å implementere en del kode på nytt. Dessuten ville kommunikasjon mellom systemene foregå med ferdig bestemte protokoller (SOAP).

5.2 Gjenbrukbarhet

Med gjenbrukbarhet mener vi at systemer som allerede er utviklet, kan tilby funksjonalitet i nye sammenhenger. Vi skal se hvordan hver av arkitekturene stimulerer til gjenbruk.

5.2.1 Mellomvarearkitektur

Mellomvarearkitektur stimulerer i utgangspunktet til en grad av gjenbruk ved å tilby et samlet grensesnitt mot flere, allerede implementerte systemer med sine forskjellige grensesnitt. I vårt tilfelle gjenbrukes funksjonalitet fra SDL i form av søk og uthenting av bildeinformasjon, da dette ikke kan hentes fra noe annet sted. Vi ser riktignok at ingen annen funksjonalitet gjenbrukes i mellomvarearkitekturen

Når det gjelder gjenbruk av funksjonalitet i for eksempel ordresystemet (slik jeg har designet TA-løsningen i casen), ville dette vært mulig, men det ville være vanskelig å realisere. Ikke bare måtte vi utvidet ordresystemet til å dele sin funksjonalitet, men vi måtte definert protokoller for utveksling av informasjon mellom de aktuelle systemene. Dette arbeidet vurderte jeg til å være så stort at det ikke ville være å foretrekke fremfor å implementere tilsvarende funksjonalitet på nytt i push-systemet.

Etter å ha vært i Scanpix en stund ble jeg klar over den store mengden forretningsregler det var nødvendig å ta hensyn til når systemer ble implementert eller utvidet. Å utvide ordresystemet som opprinnelig taklet vanlige ordre til at det skulle takle ordre fra push-salg, er ikke trivielt. En positiv side ved at vi ikke behøvde å gjenbruke funksjonalitet fra ordreystemet, var dermed at vi sto helt fritt til å implementere de nødvendige regler for behandling av push-salg.

At vi behandlet ordre direkte i push-systemet, og ikke gikk via ordresystemet, hadde også positive sider vi skal komme tilbake til i kapittel 5.4 på side 45.

5.2.2 Tjenesteorientert arkitektur

Ved å la alle tjenester tilby standardiserte grensesnitt med klare beskrivelser av hva de tilbyr, stimulerer tjenesteorientert arkitektur, og spesielt Web Services, til høy grad av gjenbruk. Designet jeg har valgt for push-systemet er laget med tanke på størst mulig grad av gjenbruk. Dette har som nevnt vist seg særlig ved å gjenbruke funksjonalitet som allerede eksisterte i ordresystemet.

Dette gjenbruket kommer imidlertid ikke gratis. Vi kan ikke regne med at denne funksjonaliteten blir tilbudt som operasjoner av tjenesten fra før av, og ordresystemet må altså klargjøres for dette.

5.2.3 Oppsummering

Selv om både mellomvarearkitektur og tjenesteorientert arkitektur stimulerer til gjenbruk av programvare, er det lettere å gjenbruke systemene i TA-løsningen. Til tross for at gjenbruk er mulig, kan det være vanskelig å realisere det, da tjenester eller systemer kanskje må modifiseres eller utvides for å kunne tilby nødvendige operasjoner.

5.3 Vedlikeholdbarhet

Vedlikeholdbarhet av en løsning kan måles på flere måter. Vi kan for eksempel se i hvilken grad det er mulig for fagpersoner uten forkunnskaper om systemene å forstå løsningens oppbygging. Videre må vi vurdere hvor lett det er for fagpersoner å oppdatere eller utvide systemene i løsningen. Muligheten for fagpersoner til å forstå og oppdatere eksisterende systemer avhenger blant annet av å holde seg til standarder. Her mener vi standarder for oppbygging av systemer, standard protokoller for overføring av informasjon, standarder for programmering og dokumentasjon, og ikke minst bruk av kjente teknologier. Når vi snakker om standarder skal vi se bort fra standarder for programmering og standarder for oppbygging internt i systemer, da dette går utenfor denne oppgavens omfang (disse standardene er viktige uavhengig av hvilken arkitektur vi velger). Vedlikeholdbarhet avhenger også av gjenbruk av programvare. Større andel av gjenbruk fører til færre steder hvor vi er nødt til å oppdatere kode hver gang en forandring skal gjennomføres.

5.3.1 Mellomvarearkitektur

Systemene i Scanpix byr på et meget heterogent miljø. SDL er et system som har utviklet seg over mange år, og som er forandret og utvidet mye. Scanpix' ordresystem er nesten helt nyutviklet, og push-systemet er også nyutviklet. Det er ikke lett for en fagperson å sette seg inn i hvordan hvert av disse systemene fungerer, og hvordan de fungerer med hverandre. Dokumentasjon av systemene er alltid viktig, og for mellomvarearkitekturløsningen er det spesielt viktig, da kommunikasjon med andre systemer ikke uten videre er forklart noe sted.

Det mest sannsynlige scenariet når systemer i Scanpix skal oppdateres, er at personell fra IT-avdelingen gjør oppdateringene. Det er sjelden arbeidskraft blir hentet utenfra. Derfor er det er poeng at teknologiene som brukes er kjent for de ansatte. Dette vil vanligvis ikke være noe problem, da Scanpix' systemer allerede er utviklet av de ansatte, og de er derfor kjent med de nødvendige teknologier.

Som beskrevet i kapittel 5.2.1 på side 41, har vi ikke stor grad av gjenbruk av programvare i mellomvarearkitekturløsningen. Særlig implementasjonen av "OrderHandler" og "OrderSender" modulene (se kapittel 3.2.2.2 på side 26) fører til at regler for behandling av ordre ligger både i ordresystemet og i push-systemet. Et resultat av dette er at vi må oppdatere både ordresystemet og push-systemet hvis regler for behandling av ordre forandrer seg.

5.3.2 Tjenesteorientert arkitektur

Hvis løsningen i Scanpix hadde hatt en tjenesteorientert arkitektur, må vi regne med at de hadde holdt seg strengt til bruk av Web Services for å definere hvilke tjenester som er tilgjengelige, og ikke minst, hva de tilbyr. Dette ville føre til at vi i høyere grad fikk en standard beskrivelse av alle tjenester. Dette ville være fordelaktig med hensyn på utenforstående som skulle forstå hvordan systemene kommuniserer. Tjenesteorientert arkitektur sier ingenting om hvordan hver tjeneste i en løsning kan være implementert internt, og vi skal derfor ikke si noe mer om hvordan koden eller oppbyggingen av hver tjeneste bør dokumenteres.

Som nevnt i kapittel 5.3.1 på forrige side, er det mest sannsynlig at Scanpix' egen IT-avdeling har ansvaret for å oppdatere tjenester i den tjenesteorienterte arkitekturen. Vi må gå ut i fra at hvis Scanpix har en tjenesteorientert arkitektur på løsningen sin, har de ansatte kompetanse innenfor dette området. Videre er denne arkitekturen i ferd med å bli meget utbredt, og vi kan derfor regne med at utenforstående fagpersonell enkelt kunne sette seg inn i Scanpix' løsning. Til tross for dette kan vi ikke regne det som lettere for utenforstående å sette seg inn i den tekniske implementasjonen av de forskjellige tjenestene.

Siden det tjenesteorienterte designet fører til høy grad av gjenbruk av programvaren, kan vi på én side anta at vedlikehold blir enklere med det tjenesteorienterte designet enn med mellomvaredesignet. På den annen side vil forandringer i krav til systemene kanskje medføre at funksjonaliteten til en tjeneste må forandres. Dette kan resultere i at grensesnittet mot andre tjenester må forandres, og de andre tjenestene må kanskje også forandres. Riktignok er dette avhengig av om tjenestene er implementert med hensyn på "forsinket binding" (se kapittel 2.3.1 på side 9).

5.3.3 Oppsummering

Vi ser at siden mellomvararkitekturløsningen er spesialtilpasset det allerede heterogene miljøet i Scanpix, er dokumentasjon av hvordan systemene fungerer sammen viktig for at fagpersonell skal kunne vedlikeholde systemene. Siden vi i TA-løsningen har standardiserte grensesnitt mellom tjenester som er godt beskrevet (med WSDL), vil vi ikke være like avhengig av ekstra dokumentasjon her. Gjenbruk av programvare fører til enklere oppdateringer da man kan forandre kode i færre systemer. I TA-løsningen kan forandringer i grensesnittet til en tjenestetilbyder føre til at tjenestesøkere også må oppdateres.

5.4 Ytelse

Å se på ytelse kan innebære å studere mange faktorer. Vi skal studere løsningene i casen, og identifisere de områder der ytelse som følge av arkitektur kan være av betydning. Vi skal se om arkitekturene fører til tilfredsstillende ytelse i løsningene, og om det er noen forskjell mellom dem.

5.4.1 Mellomvarearkitektur

Fra beskrivelsen av den realiserte mellomvarearkitekturløsningen i kapittel 3.2 på side 23, identifiserer vi disse punktene der ytelse kan være av betydning:

Generering av statistikk. Modulen StatisticsBuilder kommuniserer på flere måter med SDL for å hente ut informasjon om push-salg av bilder. Informasjon trekkes ut og legges i en MySQL-database. Dette er en relativt tidkrevende prosess. Riktignok er det ikke nødvendig at denne prosessen går spesielt fort. Noe jeg observerte fra å bygge denne modulen, var at parsing av XML var en treg prosess, ihvertfall i skriptespråket PHP, som jeg benyttet.

Bildesøk. Som en del av faktureringsprosessen søker brukerne av push-systemet etter bilder i SDL. Her er det et poeng at søkeresultater returneres raskest mulig. Derfor er modulen Metasearch, samt formatet for overføring av søkeresultater fra SDL til push-systemet, designet for å unngå unødvendig datatrafikk samt raskest mulig behandling av søkeresultatene i push-systemet før de vises til brukeren. Vi vurderte for eksempel å la SDL produsere søkeresultater som XML. Som nevnt ovenfor, ville bruk av XML ført til treg behandling av søkeresultatene før de kunne vises til brukere. Alt i alt viste det seg at løsningen vi brukte fungerte tilfredsstillende raskt.

Ordrebehandling. For brukerne er det viktig at ordrebehandling fungerer effektivt. Det vil si at bilder effektivt må kunne legges til en ordre, og informasjon må raskt kunne vises frem og oppdateres. I den realiserte mellomvaren bruker vi en egen MySQL-database kun med formål å ta vare på ordre for push-systemet. Denne databasen er tett integrert med OrderHandler-modulen i push-systemet og fungerer tilfredsstillende raskt.

Sending av ordre. Modulen OrderSender har til ansvar å sende ordre fra push-systemet til ordresystemet. Push-systemet kan ikke melde fra at en ordre er sendt til ordresystemet før ordren faktisk er lagt inn der.

Dette medfører at brukerne må vente til hele ordren er overført før de får tilbakemelding om at en ordre er sendt. Dette kan høres tregt ut, men fungerer raskt og tilfredsstillende.

5.4.2 Tjenesteorientert arkitektur

Både fra beskrivelsen av mellomvarearkitekturløsningenen i kapittel 3.2 på side 23 og TA-løsningen i kapittel 3.3 på side 29 ser vi at de stedene der ytelse er av betydning, er de samme som i kapittel 5.4.1 på forrige side.

Generering av statistikk. Som vi fant ut i delen for mellomvarearkitekturløsningen, er hastigheten for oppbyggingen av statistikk ikke viktig. Det eneste som kunne være et problem, er hvis SDL generer såpass store mengder at push-systemet ikke er i stand til å prosessere alle data som kommer inn. Etter å ha studert systemene vet jeg imidlertid at noe slikt ikke vil skje.

Bildesøk. Når det gjøres bildesøk, returneres det som regel mange søkeresultater (for eksempel 200 treff), fordi brukerne av systemet jobber mer effektivt når de kan arbeide med et stort resultatsett. Hvis resultater av et bildesøk skulle sendes med SOAP fra SDL til push-systemet, ville dette føre til en betraktelig større mengde data som måtte overføres mellom tjenestene, da SOAP med sin XML ikke er noe kompakt format. Videre måtte disse meldingen bli behandlet i push-systemet før de kunne vises til bruker. Dette setter store krav til både datakraft og båndbredde, og jeg mener at bildesøk ville gått mye tregere i TA-løsningen.

Ordrebehandling. En av hovedforskjellene mellom mellomvarearkitekturløsningen og TA-løsningen, er at vi i sistnevnte legger all ordrebehandling til ordresystemet. Når brukere arbeider med ordre, må det derfor sendes mange meldinger mellom push-systemet og ordresystemet. Hvis det arbeides med store ordre, kan meldingsstørrelsen, og dermed datamengden som må sendes mellom push-systemet og ordresystemet, bli stor. Dette fører antakelig til trege responstider for brukere som behandler ordre.

Sending av ordre. I mellomvarearkitekturløsningen overfører vi hele ordre fra push-systemet til ordresystemet før brukerne får tilbakemelding på at en ordre er sendt. I den tjenesteorienterte arkitekturen ser vi at ordre allerede ligger i ordresystemet. Dermed kan vi se bort fra denne problemstillingen.

Når det gjelder ytelse andre steder i arkitekturen, kunne vi sett på strømmer av meldinger, for eksempel fra ordresystemet til økonomisystemet. Jeg

har riktignok sett at ytelse i trafikk mellom disse og andre systemer ikke vil bli noe problem.

5.4.3 Oppsummering

De eneste stedene der ytelse kan bli noe problem i push-systemet, er der vi har brukerinteraksjon. Her er det viktig at push-systemet har tilfredsstillende responstid.

Mellomvarearkitekturløsningen er i drift idag, og den yter tilfredsstillende. Vi vet derfor at denne er vellykket med hensyn på ytelse. Særlig bildesøk i SDL og behandling av ordre er kritisk med hensyn på responstid. Mellomvaren bruker derfor spesialtilpassede formater for overføring av søkeresultater samt en egen database for ordrebehandling for å imøtekomme disse kravene.

Jeg mener TA-løsningen vil ha problemer med responstid, særlig når store datamengder skal overføres og prosesseres. Når det gjelder store søkeresultater, kan jeg ikke se noen måte å komme rundt dette problemet utenom stor overføringskapasitet og maskinkraft. Ordrebehandling kan også gi dårlig responstid hvis ordrene blir store.

5.5 Andre kriterier

I kapittel 4.2 på side 38 så vi en del vurderingskriterier som kan være relevante for å vurdere arkitekturene. Vi skal her se kort på de kriteriene jeg valgte å ikke se på i detalj, og hvorfor jeg valgte dem bort.

5.5.1 Krav til kompetanse

Er det nødvendig at fagpersonell som skal oppdatere eller utvide systemene i Scanpix har noen spesiell kompetanse? Det er det sannsynligvis. For eksempel er det viktig at fagpersonell kjenner systemenes arkitektur og plattformen de er implementert på. Hvis Scanpix skulle utviklet alle sine systemer i henhold til en TA-løsning, mener jeg det er høye krav til kunnskap om hvordan dette gjennomføres. Her har jeg riktignok argumentert hele veien for at hvis Scanpix' løsning hadde en tjenesteorientert arkitektur, ville også de ansatte hatt nødvendig kompetanse. Videre mener jeg at "krav til kompetanse" har noe sammenheng med vedlikeholdbarhet som vi allerede så på i kapittel 5.3 på side 43.

5.5.2 Opplæringsbehov

Jeg mener at opplæringsbehov har sammenheng med "krav til kompetanse" og dermed også "vedlikeholdbarhet" som ble beskrevet i kapittel 5.3 på side 43. Det er dermed ikke sagt at krav til kompetanse fører til større behov for opplæring av fagpersonell. Som det ble påpekt i kapittel 5.3, kan det være lettere for fapersonell med den rette kompetansen å sette seg inn i en tjenesteorientert arkitektur enn en mellomvare-arkitektur.

Når det gjelder opplæringsbehov av brukere av push-systemet, mener jeg disse ikke burde merke noen forskjell i brukergrensesnitt eller funksjonalitet avhengig av hvilken arkitektur vi har valgt. Derfor blir dette vurderingskriteriet uinteressant.

5.5.3 Drift

Fører noen av arkitekturene noen spesielle krav i forhold til vedlikehold, backup, ressurser eller liknende? Vi så i kapittel 3.1.2.1 på side 18 at drift av maskinpark med operativsystemer og annen underliggende programvare blir gjort utenfor Scanpix. Derfor har ikke jeg eller noen av de ansatte i Scanpix' IT-avdeling noe grunnlag til å vurdere spesielle driftsbehov. Jeg vil imidlertid ikke tro at de to arkitekturene krever store forskjeller innenfor vedlikehold eller backup. På grunn av all kommunikasjonen som foregår på et høyt nivå (tesktformater som overføres med HTTP) i TA-løsningen, vil jeg tro at denne er mer ressurskrevende enn i mellomvarearkitekturløsningen.

5.5.4 Skalerbarhet

At systemer eller tjenester i en løsning er i stand til å håndtere uforventet vekst i trafikken, kan i mange tilfeller være viktig. I vårt tilfelle vet vi at det kun er et visst antall brukere som vil benytte push-systemet. Vi har heller ikke noe grunnlag for å diskutere om noen av arkitekturene gir løsninger som skalerer godt. Jeg har derfor valgt å la være å se på om arkitekturene ville skille seg fra hverandre på dette området.

5.5.5 Stabilitet

Stabilitet eller oppetid er meget viktig i et driftsmiljø som finnes blant annet hos Scanpix. Riktignok mener jeg stabilitet avhenger mest av at systemene er implementert på en stabil plattform, riktig utvikling, samt ordentlig testing. Alt dette er noe som foregår internt i hvert system,

og mindre på arkitektur-nivå. Hvis vi skulle si noe om stabilitet for de forskjellige arkitekturene, mener jeg tjenesteorientert arkitektur ville ha en klar fordel med sin standardiserte protokoll for å utveksle meldinger. Siden tjenesteorientert arkitektur blir mye brukt, kan vi regne med at dette er en kommunikasjonsmetode som fungerer bra og er godt testet.

5.5.6 Sikkerhet

Sikkerhet i programvare er et meget aktuelt tema, også innenfor integrasjon av systemer. Til tross for dette, var det ikke spesielt stort fokus på dette under utviklingen av push-systemet i Scanpix. Derfor går dette området utenfor det jeg har studert i forbindelse med oppgaven. Allikevel vil jeg tro at Web Services med standarder som WSS (Web Services Security) vil gjøre det enkelt å implementere en sikker integrasjon mellom flere systemer. Dette kan jeg dessverre ikke si om mellomvarearkitektur.

Kapittel 6

Drøfting

Vi studerte i forrige kapittel hvordan de to arkitekturene (mellomvarearkitektur og tjenesteorientert arkitektur) er egnet for casen vår. I dette kapittelet skal vi trekke ut noen poeng vi fant fra casestudien med hensyn på vurderingskriteriene vi kom frem til i kapittel 4.2, og drøfte forskjeller for arkitekturene generelt.

Vurderingskriteriene vi definerte, er "arbeidsmengde og kompleksitet", "gjenbrukbarhet", "vedlikeholdbarhet" og "ytelse". Vi skal se på kriteriene hver for seg.

6.1 Arbeidsmengde og kompleksitet

Det er en nær sammenheng mellom arbeidsmengde og kompleksitet, det vil si at økt kompleksitet gir økt arbeidsmengde. Vi vil derfor begrense oss til å studere kompleksitet i dette kapittelet, siden alle vurderinger vi gjør av kompleksitet også bør gjelde for arbeidsmengde.

Vi vil forsøke å beskrive kompleksitet ved hjelp av en matematisk modell og vi vil benytte denne modellen til å sammenlikne arkitekturene.

6.1.1 Forutsetninger og antakelser

Før vi kan lage en matematisk modell for kompleksitet i integrasjonsløsninger, vil vi gjøre en del forutsetninger og antakelser.

Vi forutsetter at alle systemer i en løsning skal kunne kommunisere med hverandre. Dette er ikke nødvendigvis en realistisk antakelse, men en forutsetning for å kunne sammenlikne arkitekturene på likt grunnlag.

For å vurdere kompleksiteten i løsninger skal vi gå ut fra at den kan uttrykkes som en funksjon av antall systemer som integreres i løsningen (n). I tillegg skal vi ta hensyn til disse faktorene:

1. *Økning av løsningens kompleksitet som følge av et grensesnitt mellom to systemer som inngår i løsningen (X).*
2. *Antall grensesnitt mellom systemer i en løsning (g).*
   Dette antallet vil være et resultat av hvilken arkitektur som benyttes i løsningen, og hvor mange systemer som integreres (n).
3. *Arkitekturens initielle kompleksitet (c).*
   Med dette mener vi det nivået av kompleksitet som er nødvendig for å ta i bruk en arkitektur. Et eksempel er selve mellomvaren i en mellomvarearkitekturløsning.

Merk at vi i vurderingen av kompleksitet ikke tar med kompleksiteten i de systemene som integreres. Dette er en faktor som er uavhengig av arkitektur, og den er derfor ikke interessant når vi skal sammenlikne to arkitekturer.

6.1.2 Ad hoc systemintegrasjon

For å forstå hensikten og fordelene med mellomvarearkitektur og tjenesteorientert arkitektur på løsninger skal vi kort studere en løsning der vi integrerer et antall systemer (n), hvor alle systemene kommuniserer direkte med hverandre. Dette kaller vi "ad hoc systemintegrasjon".

Hvis vi skal få to systemer til å kommunisere, kan dette være enkelt, fordi vi bare behøver å utvikle ett grensesnitt mellom disse to systemene. Vi kaller kompleksiteten dette grensesnittet påfører løsningen for X_0. Videre ser vi at hver gang et system legges til løsningen, må vi i verste fall utvikle grensesnitt mot alle de eksisterende systemene. Denne metoden for å integrere systemer krever ingen initiell kompleksitet, og vi har dermed at $c_0 = 0$.

Vi ser at en løsning med n systemer vil få en kompleksitet ($K_0(n)$) avhengig av kompleksiteten per grensesnitt (X_0) og antall grensesnitt (g_0):

$$K_0(n) = X_0 g_0$$

Før vi kan identifisere antall grensesnitt (g_0) i ad hoc-løsninger, må vi finne sammenhengen mellom grensesnitt og systemer i løsningene. Vi har gått ut fra at alle systemer i en ad hoc-løsning må kunne kommunisere med alle de andre systemene i løsningen. Antall grensesnitt finner vi da som:

$$g_0 = \frac{n(n-1)}{2}$$

Dermed får vi at kompleksiteten for en ad hoc-løsning kan beskrives som:

$$K_0(n) = X_0 \frac{n(n-1)}{2} \tag{6.1}$$

Vi ser at dette er en polynomisk funksjon av annen grad (n^2), og at kompleksiteten i ad hoc-løsninger øker fort når n vokser. Videre påvirker også X_0 kompleksiteten i løsningene. Vi kan anta at X_0 er relativt stor, fordi hvert grensesnitt som utvikles i en ad hoc-løsning må skreddersys, og dermed øker løsningens kompleksitet merkbart.

6.1.3 Mellomvare

I kapittel 5.1.1 så vi at mellomvarearkitekturløsningen i casen var relativt enkel å utvikle. Den viktigste faktoren som økte kompleksiteten, var at alle grensesnitt mellom systemer måtte skreddersys. Vi kan ikke anta at mellomvarearkitekturer generelt har standardiserte metoder for kommunikasjon mellom systemer i en løsning. Derfor er det rimelig å forutsette at problemet fra casen gjelder generelt, nemlig at kommunikasjon mellom systemer i en mellomvarearkitekturløsning må skreddersys. Kompleksiteten et slikt grensesnitt påfører en mellomvarearkitekturløsning, kaller vi X_1.

Før vi finner en funksjon som kan beskrive kompleksiteten i mellomvarearkitekturløsninger skal vi gjøre en forutsetning til:

> *Alle systemer som integreres i en mellomvarearkitekturløsning må ha grensesnitt mot mellomvaren. Ingen systemer har grensesnitt direkte mot hverandre.*

Antall grensesnitt mellom systemer i en mellomvarearkitekturløsning (g_1) blir da likt antallet systemer i løsningen (n).

Siden en mellomvarearkitektur krever at vi utvikler selve mellomvaren, fører dette til en økning i kompleksitet. Den økte kompleksiteten som følger av mellomvaren vil følge mellomvarearkitekturløsningen uansett hvor stor løsningen er. Dette er mellomvarearkitekturløsningens initielle kompleksitet (c_1).

Dermed har vi at mellomvarearkitekturløsninger med n systemer, får en kompleksitet, $K_1(n)$, på formen:

$$K_1(n) = c_1 + X_1 g_1 = c_1 + X_1 n \tag{6.2}$$

I casen opplevde jeg ikke det totale arbeidet med å opprette mellomvarearkitekturløsningen som spesielt stort. Arbeidet for å utvikle den initielle mellomvaren (c i vår case) var ikke stort i forhold til merarbeidet for å utvikle grensesnitt mot de ulike systemene. Dette gir støtte til en antakelse

om at den initielle kompleksiteten i en generell mellomvarearkitekturløsning (c_1) ikke er spesielt stor.

6.1.4 Tjenesteorientert arkitektur

I casen så vi at push-systemet blir enklere å utvikle enn mellomvarearkitekturløsningen fordi utviklingen av grensesnitt mellom systemene er enkelt. Dette gjelder for TA-løsninger generelt, fordi disse alltid vil følge Web Services' standarder for å kommunisere. Når vi skal finne en funksjon for kompleksiteten i TA-løsninger ($K_2(n)$), skal vi kalle kompleksiteten som følger av et grensesnitt for X_2. Siden kompleksiteten i grensesnitt generelt sett er liten for TA-løsninger, vet vi at X_2 er liten.

Det er mulig å integrere systemer med Web Services uten at vi bruker en opplysningstjeneste, men som vi argumenterte for i kapittel 3.3.1, er det mest hensiktsmessig å la en TA-løsning ha en opplysningstjeneste. I denne vurderingen skal vi dermed forutsette at TA-løsninger alltid har en opplysningstjeneste.

Kompleksiteten som følger av å ta i bruk en TA-løsning med opplysningstjeneste og støtte for Web Services, er relativt stor. Som med mellomvarearkitektur, skal vi kalle dette TA-løsningens initielle kompleksitet, eller c_2. Vi la ikke vekt på denne kompleksiteten i casen, men dette er en viktig faktor når en TA-løsning skal vurderes helhetlig.

For at hver tjeneste skal kunne kommunisere med alle de andre tjenestene i en TA-løsning, tilbyr alle tjenestetilbydere en beskrivelse av hvordan tjenestesøkere kan kommunisere med dem. Ideelt sett kan derfor alle systemer i en TA-løsning kommunisere med hverandre utelukkende fra å lese disse beskrivelsene. Vi ser at dette kanskje ikke er reelt, og at hver tjeneste til en viss grad må tilpasses de tjenestene den skal kommunisere med. En slik tilpasning er riktignok nødvendig uansett hvilken arkitektur vi bruker, og vi skal derfor ikke regne dette som økt kompleksitet. Dermed får vi at kompleksiteten som følger av antall grensesnitt i en TA-løsning (g_2), er likt antall tjenester i løsningen:

$$g_2 = n$$

En funksjon for kompleksiteten i TA-løsninger ($K_2(n)$) er da:

$$K_2(n) = c_2 + X_2 g_2 = c_2 + X_2 n \tag{6.3}$$

6.1.5 Sammenlikning

Vi skal nå gjøre en sammenlikning av arkitekturene med hensyn på funksjonene vi nettopp har kommet frem til. Vi ser fra funksjonene for $K_1(n)$ og $K_2(n)$ at begge har arbeidsmengde og kompleksitet som vokser lineært. Dette er store fremskritt i forhold til løsningen for $K_0(n)$, der kompleksiteten øker som et polynom av andre grad.

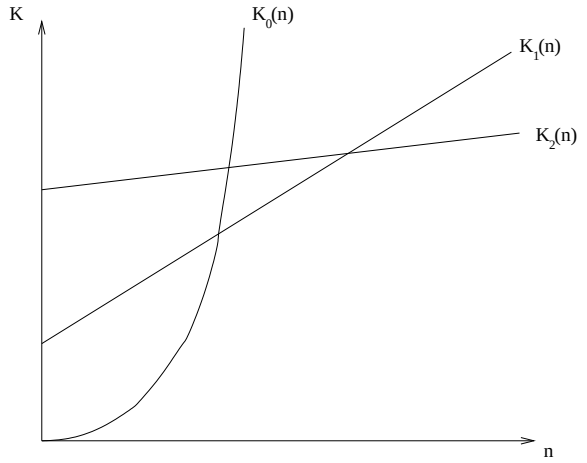
Som nevnt, vil kompleksiteten av ad hoc-løsninger $K_0(n)$ være lav hvis vi integrerer få systemer. Dette kommer først og fremst av at disse løsningene ikke har noen opprinnelig kompleksitet ($c_0 = 0$). Derimot stiger kompleksiteten raskt når n øker, både fordi kompleksiteten er proposjonal med n^2, og fordi X_0 er relativt stor.

I en mellomvarearkitekturløsning er også arbeidet med å utvikle hvert grensesnitt relativt stor, siden hvert grensesnitt må skreddersys også her. Vi kan derfor regne med at verdien til X_1 er i nærheten av X_0. Videre ser vi at kompleksiteten i mellomvarearkitekturløsninger vokser lineært. Dette medfører at disse løsningene blir langt mindre enn for ad hoc-løsninger når n blir stor. Siden mellomvarearkitekturløsninger har en opprinnelig kompleksitet, c_1, vil slike løsninger være mer komplekse enn ad hoc-løsninger for små verdier av n.

Ser vi på TA-løsninger, vokser kompleksiteten deres også lineært. Som vi var inne på i kapittel 6.1.4 er det enkelt å utvikle grensesnitt mellom systemene i en TA-løsning. Sidene kommunikasjonen foregår på et standardisert vis, øker ikke kompleksiteten så mye hver gang et nytt grensesnitt utvikles. Vi kan derfor regne med at X_2 er mindre enn X_0 og X_1.

Som vi så i kapittel 6.1.4, har TA-løsninger en opprinnelig kompleksitet (c_2). Denne kompleksiteten oppstår på grunn av opplysningstjenesten som må eksistere før systemer kan begynne å kommunisere, samtidig som det må defineres et globalt skjema for hele løsningen. Dette er tungvint hvis vi skal integrere noen få systemer, og vi ser derfor at c_2 er relativt stor. Ut fra arbeidet jeg gjorde i casen, men også fra drøftingen generelt (se kapittel 6.1.3), kan vi regne med at den opprinnelige kompleksiteten i mellomvarearkitekturløsninger (c_1) er mindre enn for TA-løsninger (c_2). Vi kan se sammenhengen mellom kompleksiteten de forskjellige arkitekturene medfører i grafene på figur 6.1 på neste side.

Fra grafene ser vi at mellomvarearkitekturløsninger blir mer komplekse enn TA-løsninger bare n blir stor nok, men hvor stort er dette? Hvis vi kjente denne verdien og kalte den n_{TA}, hadde det vært enkelt å velge hvilken arkitektur vi skulle bruke i en løsning: Hvis $n < n_{TA}$ velger vi mellomvarearkitektur, og hvis $n > n_{TA}$ velger vi tjenesteorientert



Figur 6.1: Sammenlikning av kompleksitet som følge av arkitektur.

arkitektur. Dessverre er det vanskelig å si noe om verdien til n_{TA}.

Fra figuren ser vi at grafen til $K_0(n)$ krysser de andre grafene. Det er vanskelig å si noe om størrelsen til n der dette skjer, men vi kan regne med at det skjer for relativt lave verdier. Siden denne oppgaven skal sammenlikne mellomvarearkitektur og tjenesteorientert arkitektur, er et nærmere studie av $K_0(n)$ ikke relevant.

I (Newcomer and Lomow, 2004) hevdes det at vi må integrere minst to til fire systemer for at en TA-løsning med opplysningstjeneste skal lønne seg i forhold til andre løsninger. Forfatterne legger i denne vurderingen vekt på arbeidsmengde og ikke kompleksitet. Som vi argumenterte for i starten av dette kapittelet, er det sammenheng mellom arbeidsmengde og kompleksitet, og jeg mener derfor at denne vurderingen er relevant her.

I casen vurderte vi push-systemet til å være enkelt å utvikle enten vi brukte mellomvarearkitektur eller tjenesteorientert arkitektur. En forskjell mellom løsningene her, er at en TA-løsning antakelig fører til økt kompleksitet i resten av løsningen i Scanpix, mens utvidelsen push-systemet innebærer, ikke øker kompleksiteten i særlig grad. Som nevnt, forespeiles det i (Newcomer and Lomow, 2004) at en løsning bør integrere minst to til fire systemer for at det skal være hensiktsmessig å bruke tjenesteorientert arkitektur. Fra erfaringene i casen mener jeg dette antallet er et absolutt minimum for TA-løsninger.

6.1.6 Oppsummering og kritikk

Vi har i dette kapittelet kommet frem til matematiske modeller for å beskrive kompleksiteten i løsninger med hensyn på antall systemer som integreres. Ut fra disse modellene har vi vurdert om mellomvarearkitektur og tjenesteorientert arkitektur skiller seg fra hverandre med hensyn på vurderingskriteriet "arbeidsmengde og kompleksitet". Vi har kommet frem til at de gjør det.

Selv om vi har identifisert en forskjell mellom tjenesteorientert arkitektur og mellomvarearkitektur, er dette gjort med en modell hvor vi har gjort en del forutsetninger og antakelser. Vi har for eksempel satt som forutsetning at mellomvarearkitekturløsninger kun har grensesnitt mellom mellomvare og systemene som skal integreres. Vi har altså ikke grensesnitt mellom de andre systemene i en løsning. Dette er lite realistisk, og vi ser også fra casen at forutsetningen ikke holder der, da for eksempel ordresystemet allerede har grensesnitt mot økonomisystemet og SDL.

Det kan også være mange faktorer vi ikke har identifisert, men som påvirker kompleksiteten i løsningene. Å finne flere faktorer og å bruke disse ville imidlertid gått utenfor rammene til denne oppgaven. Som videre arbeid kan det være interessant å identifisere slike faktorer. Hvis man vil gå videre og bruke funksjonene vi har funnet i dette kapittelet, kan det være interessant å identifisere størrelsesforholdet for faktorene c_1, c_2, X_1 og X_2. Dessuten kan det være interessant å vurdere en eventuell verdi for n_{TA}.

6.2 Gjenbrukbarhet

I kapittel 5.2 så vi i hvilken grad mellomvarearkitektur og tjenesteorientert arkitektur førte til gjenbruk i casen. I dette kapittelet skal vi vurdere hvor enkelt det er å gjenbruke programvare ved å bruke henholdsvis mellomvarearkitektur og tjenesteorientert arkitektur til systemintegrasjon generelt.

For å vurdere hvor enkelt det er å gjenbruke programvare, skal vi definere en metode for å vurdere gjenbrukbarheten en arkitektur tilbyr. Deretter skal vi bruke denne metoden for å sammenlikne arkitekturene.

6.2.1 Forutsetninger og antakelser

Vi skal nå sammenlikne to arkitekturer med hensyn på gjenbrukbarhet, og en slik sammenlikning forutsetter at vi har samme formål når vi benytter

arkitekturene i integrasjonsløsninger. Generelt kan vi ikke regne med at dette er tilfellet.

I en TA-løsning designes systemer som tjenester der de tilbyr sine data eller sin funksjonalitet. Vi ser at denne metoden for å designe systemer tar hensyn til gjenbruk helt fra starten av utviklingen.

Når vi utvikler mellomvare i en mellomvarearkitekturløsning er dette typisk for å få konkrete systemer til å kommunisere, eller vi vil gjenbruke data eller funksjonalitet fra eksisterende systemer. De eksisterende systemene er ikke nødvendigvis utviklet med hensyn på gjenbruk. Som vi så ovenfor, er systemer i en TA-løsning designet for enkelt å gjenbrukes. Dermed er det en forskjell i hensikt og bruk av de to arkitekturene, og en generell sammenlikning på dette grunnlaget er ikke særlig interessant.

For å muliggjøre en sammenlikning, kan gi gjøre følgende ikke helt realistiske forutsetning:

> *I både mellomvarearkitekturløsninger og TA-løsninger er systemene i løsningene designet for enkelt å kunne gjenbrukes.*

Nå kan vi anta at systemer kan gjenbrukes like enkelt, enten de er i en mellomvarearkitektur eller en tjenesteorientert arkitektur.

For å få brukt et system som er klargjort for gjenbruk, må andre systemer i arkitekturen kunne kommunisere med dette systemet. Kommunikasjonen må følge protokoller som defineres i grensesnitt mellom systemene. Det er viktig at disse grensesnittene kan utvikles enklest mulig, og grensesnittene gir oss derfor et grunnlag for å vurdere gjenbrukbarheten av systemer i forskjellige løsninger.

Siden vi antar at systemer kan gjenbrukes like enkelt uansett arkitektur, vil ikke mellomvarearkitektur og tjenestorientert arkitektur skille seg fra hverandre på dette punktet. Derfor kan vi vurdere gjenbrukbarheten en arkitektur tilbyr på grunnlag av hvor enkelt det er å utvikle grensesnitt mellom systemer i en løsning.

6.2.2 Mellomvarearkitektur

For å se hvor enkelt det er å utvikle grensesnitt mellom systemer i en mellomvarearkitekturløsning, kan vi ta utgangspunkt i vurderingen gjort i kapittel 6.1.3 på side 53. Her ser vi blant annet på kompleksiteten knyttet til grensesnittene i mellomvarearkitekturløsninger. Som jeg argumenterte for, er det en sammenheng mellom kompleksitet og arbeidsmengde. Det er derfor rimelig å anta at kompleksiteten til grensesnittene i en

mellomvarearkitekturløsning (X_1) også sier noe om arbeidsmengden for å utvikle dem.

Vi fant ut at X_1 var relativt stor, og vi kan derfor regne med at det er relativt vanskelig å utvikle grensesnittene i en mellomvarearkitekturløsning.

6.2.3 Tjenesteorientert arkitektur

For å se hvor enkelt det er å utvikle grensesnitt i en TA-løsning, skal vi gå frem på samme måte som vi gjorde for mellomvarearkitektur. Vi ser fra kapittel 6.1.4 at X_2 er liten, og vi kan derfor regne med at det er relativt enkelt å utvikle grensesnitt mellom systemer i en TA-løsning.

6.2.4 Sammenlikning

For å sammenlikne gjenbrukbarheten til henholdsvis mellomvarearkitekturløsninger og TA-løsninger, kan vi nå sammenlikne utviklingsarbeidet for grensesnitt mellom systemer i løsningene.

Vi har sett at det er relativt vanskelig å utvikle grensesnittene i mellomvarearkitekturløsninger, mens det er lettere å utvikle grensesnittene i TA-løsninger. Dermed ser vi at tjenesteorientert arkitektur gjør gjenbruk av programvare enklere enn mellomvarearkitektur.

Selv på grunnlag av forutsetningen vi gjorde for å likestille mellomvarearkitektur og tjenesteorientert arkitektur, er det liten tvil om at tjenesteorientert arkitektur tilbyr gjenbruk av programvare enklere enn mellomvarearkitektur. Dette stemmer godt overens med hva vi fant ut i casen, der vi så at mellomvarearkitekturen ikke gjenbrukte funksjonalitet som allerede fantes i andre systemer (se kapittel 5.2.1 på side 41).

Som vi påpekte i kapittel 6.2.1, er en sammenlikning av arkitekturene med hensyn på gjenbrukbarhet ikke uten videre realistisk. TA-løsninger er generelt bedre egnet for gjenbruk enn mellomvarearkitekturløsninger, men dette er ikke spesielt overraskende; en av de viktigste grunnene til at tjenesteorientert arkitektur har oppstått, er for å lett kunne gjenbruke programvare.

6.3 Vedlikeholdbarhet

Vi vurderte i kapittel 5.3 hvor enkelt det er å vedlikeholde mellomvarearkitekturløsningen og TA-løsningen fra casen. For å vurdere løsningene i

casen så vi på hvor enkelt det er for fagpersoner uten forkunnskaper om løsningen å sette seg inn i den, samt å oppdatere den. Generelt ser vi at muligheten for dette er avhengig av å følge standarder for utvikling og beskrivelse av løsningene.

Vedlikeholdbarhet er viktig både i store og små miljøer[1]. I store miljøer ser vi viktigheten fordi løsninger forvaltes av mange forskjellige personer. I små miljøer kan noen få personer ha all nødvendig oversikt, men hva skjer hvis disse personene slutter? Her kommer vi inn på sårbarhet i løsninger, og vi skal ikke gå videre inn på dette. Allikevel hjelper dette oss til å se viktigheten av vedlikeholdbarhet, fordi det må være mulig å bytte ut personell også i små miljøer.

Vedlikeholdbarhet er også avhengig av faktorer vi har studert allerede, for eksempel kompleksitet. Generelt kan vi anta at løsninger er lettere å vedlikeholde om de er mindre komplekse.

For å vurdere vedlikeholdbarhet for generelle løsninger, skal vi derfor se på bruk av standarder for utvikling og beskrivelse i løsningene, sammen med deres kompleksitet.

Merk at vi ikke skal studere standarder eller liknende for systemer internt, fordi arkitekturer ikke vil skille seg fra hverandre på dette punktet.

6.3.1 Mellomvarearkitektur

Som vi har sett fra casen, og som vi har argumentert for at gjelder generelt, er mellomvare skreddersydd til systemene som er integrert i en mellomvarearkitekturløsning. At mellomvaren tilpasses hvert system, medfører lav grad av standardisering for kommunikasjon mellom systemene. Som i casen ser vi dermed at det generelt er viktig med dokumentasjon og beskrivelser av grensesnitt i mellomvarearkitekturløsninger. At grensesnitt mellom systemer ikke er standardisert, gjør det vanskeligere å vedlikeholde en mellomvarearkitekturløsning enn hvis grensesnittene hadde vært standardisert.

Når det gjelder mellomvarearkitekturløsningers kompleksitet, vurderte vi dette i kapittel 6.1.3 på side 53, hvor vi kom frem til at en mellomvarearkitekturløsnings kompleksitet er minst i forhold til TA-løsning når løsningen består av få systemer.

[1]Med store og små miljøer mener vi størrelse på fagmiljø med hensyn på antall mennesker.

6.3.2 Tjenesteorientert arkitektur

Siden tjenester i en TA-løsning alltid har beskrivelser av seg selv og hva de kan gjøre (med WSDL), blir en TA-løsning til en viss grad selvforklarende. Derfor er ikke dokumentasjon av systemene og deres måte å kommunisere på like viktig som i mellomvarearkitekturløsninger. I tillegg til at systemene i TA-løsninger er beskrevet med WSDL, er all kommunikasjon mellom systemene standardisert med utveksling av SOAP-meldinger. Dermed ser vi at TA-løsninger har høy grad av standardisering for både beskrivelse av systemer og kommunikasjon mellom systemene.

Kompleksiteten i TA-løsninger vurderte vi i kapittel 6.1.4 på side 54. Vi fant ut at kompleksiteten i TA-løsninger er større enn for mellomvarearkitekturløsninger hvis løsningene er små (få systemer), men at kompleksiteten er mindre for TA-løsninger enn for mellomvarearkitekturløsninger når løsningene er store (mange systemer).

6.3.3 Sammenlikning

Vi har ikke funnet noen funksjoner for å beskrive eller måle vedlikeholdbarhet i forskjellige løsninger, da dette har vist seg å være vanskelig. Riktignok har vi identifisert noen faktorer som er viktige for at løsningene skal være lettest mulig å vedlikeholde: “kompleksitet”, “standarder for utvikling i løsninger” og “standarder for beskrivelse i løsninger”.

I henhold til de identifiserte faktorene ser vi at mellomvarearkitekturløsninger generelt virker vanskeligere å vedlikeholde enn TA-løsninger. Dette skyldes at TA-løsninger i høyere grad enn mellomvarearkitekturløsninger er standardisert med tanke på beskrivelse av systemer og kommunikasjon mellom systemene.

Til tross for denne forskjellen, kan vi fra sammenlikningen i kapittel 6.1.5 på side 55 anta at mellomvarearkitekturløsninger er mindre komplekse enn TA-løsninger når løsningene er små. På dette grunnlaget er det vanskelig å konkludere med at TA-løsninger alltid gir bedre vedlikeholdbarhet, men det er rimelig å si at TA-løsninger gir bedre vedlikeholdbarhet for store løsninger (løsninger med mange integrerte systemer).

6.4 Ytelse

Som vi har vært inne på, er ytelse et samlebegrep for flere faktorer som har med systemers evne til å utføre ønskede oppgaver. Vi så i kapittel 5.4 at

Figur 6.2: Modell for systemer som kan kommunisere i et nettverk.

responstid der brukere er involvert, er en av de kritiske faktorene i casen. Vi skal derfor studere responstid for generelle løsninger.

6.4.1 Modell for interaksjon

Vi skal ta utgangspunkt i modellen på figur 6.2, der vi har to systemer S_1 og S_2 som kan kommunisere på et nettverk. Vi kan se for oss at S_1 skal sende en forespørsel til S_2 og forventer en respons. Vi kan da definere responstiden i løsningen (T) som tiden det tar fra S_1 vil kontakte S_2 med en forespørsel til S_2 har gitt en respons, og denne responsen er klar til å brukes internt i S_1.

Når vi skal uttrykke T, kan vi skille mellom to typer tidsbruk:

Tidsbruk som følge av arkitektur (A). Denne tidsbruken består i å pakke data som skal sendes på formater avhengige av arkitekturen (for eksempel XML), sende pakkede data over et nettverk, pakke ut/parse sendte data før dataene kan brukes internt i et system. Vi ser at disse formene for tidsbruk alle har med kommunikasjon mellom systemer å gjøre, og at denne tidsbruken er avhengig av arkitektur.

Tidsbruk uavhengig av arkitektur (B). Dette er tidsbruken som følger av S_1 og S_2 sitt interne arbeid. Dette er arbeid som må utføres uansett hva slags løsning vi har, og tiden som blir brukt som følge av dette er derfor uavhengig av arkitektur.

Responstiden i en løsning, T, kan dermed uttrykkes som:

$$T = A + B \tag{6.4}$$

Vi skal nå vurdere størrelsen til A for hver av arkitekturene.

6.4.2 Mellomvarearkitektur

Mellomvarearkitektur stiller ingen krav til hvordan kommunikasjonen mellom systemer foregår, og vi får derfor mange spesialtilpassede grensesnitt. Spesialtilpassede grensesnitt gir relativt lite administrasjonsarbeid

for systemene som er integrert, samtidig som formater for overføring av data vanligvis gir korte meldinger. Dette medfører at tidsbruken som følge av mellomvarearkitektur (A_1) blir relativt liten. At mellomvarearkitekturløsninger har mange spesialtilpassede grensesnitt og formater, betyr at systemene i løsningene er tett intergrert.

6.4.3 Tjenesteorientert arkitektur

Tjenesteorientert arkitektur krever at alle data som sendes mellom systemer skal pakkes inn i XML, et format som gir relativt lange meldinger. I (Kaye, 2003) argumenteres det for at de lange XML-baserte meldingene fører til lengre overføringstid på nettverket. Riktignok kan det hende at en økning i størrelsen på meldinger ikke står i forhold til annen tidsbruk som øker responstiden i en løsning. Et lite eksempel kan illustrere nærmere:

> Hvis vi har to systemer som skal kommunisere over et nettverk, og dette nettverket er et lokalt nettverk, er det vanlig at nettverket har en overføringskapasitet på $100Mb/s$. Hvis en melding er $5KB$ istedet for $1KB$ som følge av en tjenesteorientert arkitektur på løsningen, ser vi at overføringen uansett vil ta minimalt med tid.

Som følge av eksempelet ovenfor, er det rimelig å anta at økt overføringstid som følge av tjenesteorientert arkitektur bare er relevant hvis meldingene er veldig store eller nettverksforbindelsen mellom systemene er dårlig.

I tillegg til at XML gir lange meldinger, er prosessering av XML relativt tungt. Dette gjør at transformering til og fra XML kan ta en del tid. Denne standardiserte måten å kommunisere på, gjør at systemer blir løsere integrert enn for mellomvarearkitektur. Dette betyr at systemene i høy grad er uavhengige av hverandre og lett kan byttes ut. Vi ser at tung transformering til og fra XML, samt overføring av større datamengder, fører til at tidsbruken ved bruk av tjenesteorientert arkitektur (A_2) er relativt stor.

6.4.4 Sammenlikning

Vi har nå kommet frem til at tidsbruk ved bruk av mellomvarearkitektur, A_1, ikke er spesielt stor, mens tidsbruk ved bruk av tjenesteorientert arkitektur, A_2, er større.

Videre ser vi at hvis A blir stor i forhold til B, kan dette øke den samlede responstiden til et uakseptabelt nivå. For løsninger der responstid er kritisk (for eksempel sanntidssystemer), vet vi at B er liten. I slike løsninger vil

A fort blir dominerende, og vi kan derfor regne med at tjenesteorientert arkitektur er dårligere egnet for disse enn mellomvarearkitektur.

For å oppsummere: Vi ser at responstid blir påvirket av hvilken arkitektur vi velger, og at mellomvarearkitekturløsninger, der systemer integreres tett, yter bedre enn TA-løsninger hvor systemer integreres løst. Dette stemmer godt overens med blant annet (Erl, 2005) hvor det påpekes at tett integrerte systemer typisk yter bedre enn løst integrerte.

6.4.5 En merknad til ytelse

Vi har i dette kapittelet avgrenset oss til å studere responstid fordi dette var den viktigste ytelsesfaktoren i casen. Denne vurderingen gir antakelig ikke et riktig bilde av hvordan løsninger kan yte med hensyn på andre faktorer.

Andre viktige ytelsesfaktorer er ulike former for kapasitet, slik som antall samtidige brukere, antall forespørsler en løsning kan takle innenfor en tidsramme og liknende. En metode for å øke kapasiteten er å sørge for best mulig utnyttelse av tilgjengelig maskinkapasitet ved hjelp av asynkron kommunikasjon.

Ved synkron kommunikasjon (som benyttes i denne oppgaven) må hvert system vente på respons fra andre systemer når de kommuniserer. Dette kan føre til at vi får dårlig utnyttelse av maskinkraften i systemene.

Hvis kommunikasjonen mellom systemene skjer asynkront, behøver ikke et system å vente på respons hver gang det skal kommunisere med andre systemer, og dette kan gi stor økning i kapasiteten til hvert system.

Asynkron kommunikasjon kan derfor gi god utnyttelse av kapasiteten i en TA-løsning, og ytelse med hensyn på kapasitet kan derfor skille seg fra ytelse med hensyn på responstid når vi sammenlikner arkitekturene.

Kapittel 7

Oppsummering og konklusjoner

Vi har i denne oppgaven studert de to arkitekturene mellomvarearkitektur og tjenesteorientert arkitektur med formål å vurdere og sammenlikne dem.

Som en del av vurderingen har vi gjort en casestudie hvor det ble gjennomført en systemintegrasjon i Scanpix AS. Den realiserte integrasjonen er en mellomvarearkitekturløsning. Tilsvarende har vi foreslått et design for en systemintegrasjon med tjenesteorientert arkitektur.

Med casen som grunnlag har vi vurdert fordeler og ulemper ved bruk av arkitekturene i Scanpix, og vi har generalisert resultatene til ikke bare å gjelde casen, men arkitekturene generelt.

7.1 Hva vi fant ut

For å vurdere arkitekturene identifiserte vi noen vurderingskriterier:

1. Arbeidsmengde og kompleksitet
2. Gjenbrukbarhet
3. Vedlikeholdbarhet
4. Ytelse (responstid)

Disse kriteriene har vi brukt for å finne relevante forskjeller som følge av de to arkitekturene i casen, og senere har vi drøftet forskjeller for arkitekturene generelt.

Vi har utviklet en enkel modell for å beskrive kompleksiteten i løsninger avhengig av arkitektur. Ved hjelp av modellen vurderte vi kompleksitet i løsninger ut fra antall systemer i en integrasjonsløsning, kompleksitet i grensesnittene mellom dem, samt kompleksiteten selve arkitekturen påfører løsningen. Vi kan fra denne vurderingen konkludere med følgende:

> *Mellomvarearkitektur fører typisk til mindre kompleksitet enn tjenesteorientert arkitektur for løsninger der antall integrerte systemer er lite. Hvis antall integrerte systemer blir stort nok, vil en tjenesteorientert arkitektur føre til minst samlet kompleksitet.*

Dessuten har vi argumentert for at det er sammenheng mellom kompleksitet og arbeidsmengde, slik at en løsning med høy grad av kompleksitet typisk fører til større arbeidsmengde.

Både mellomvarearkitektur og tjenesteorientert arkitektur stimulerer til gjenbruk, men vi har sett at tjenesteorientert arkitektur ville ha ført til bedre gjenbrukbarhet i casen. Fordi tjenesteorientert arkitektur fører til høyere grad av standardisering av grensesnitt mellom systemer konkluderer vi med:

> *Løsninger med tjenesteorientert arkitekturer kan forventes å gi bedre gjenbrukbarhet enn mellomvarearkitekturløsninger.*

Siden en tjenesteorientert arkitektur gir lavere kompleksitet for løsninger med mange systemer, og baserer seg på standarder for utvikling og beskrivelse av løsninger, kan vi konkludere med følgende:

> *Tjenesteorientert arkitektur vil normalt gi bedre gjenbrukbarhet enn mellomvarearkitektur, med mulig unntak av løsninger med få systemer.*

Videre har vi vurdert responstid, som var den viktigste ytelsesfaktoren i vår case. Vi så at mellomvarearkitekturløsninger vanligvis har tettere integrerte systemer enn løsninger med tjenesteorientert arkitektur. På grunnlag av dette kan vi konkludere:

> *Mellomvarearkitektur fører normalt til bedre responstid i løsninger enn tjenesteorientert arkitektur.*

7.2 Valg av arkitektur

Ut fra våre konklusjoner har vi nå et hjelpemiddel for å vurdere når de to arkitekturene er egnet.

Vi kan skille mellom store og små løsninger, avhengig av om de integrerer mange eller få systemer. For å vurdere hvor godt arkitekturene

Kriterie	Mellomvare-arkitektur	Tjenesteorientert arkitektur
Arbeidsmengde og kompleksitet	+	−
Gjenbrukbarhet	−	+
Vedlikeholdbarhet	+	−
Ytelse (responstid)	+	−

Tabell 7.1: Tabell for sammenlikning av arkitektur i små løsninger

Kriterie	Mellomvare-arkitektur	Tjenesteorientert arkitektur
Arbeidsmengde og kompleksitet	−	+
Gjenbrukbarhet	−	+
Vedlikeholdbarhet	−	+
Ytelse (responstid)	+	−

Tabell 7.2: Tabell for sammenlikning av arkitektur i store løsninger

er egnet i små løsninger, kan vi se på sammenlikningen av dem i tabell 7.1. Arkitekturen som er vurdert som best med hensyn på et vurderingskriterium representeres med + mens den andre representeres med −.

Likeledes kan vi gjøre en samlet vurdering av arkitekturene for store løsninger. Denne sammenlikningen er vist i tabell 7.2.

Vi ser at for løsninger med få systemer, eller der det er strenge krav til responstid (som i sanntidssystemer), vil mellomvarearkitektur være det beste valget. For løsninger med mange systemer der det ikke er spesielt strenge krav til responstid, vil tjenesteorientert arkitektur være fordelaktig.

7.3 Uløste problemer og videre arbeid

Det er en del problemer vi i løpet av denne oppgaven burde sett nærmere på, men ikke har studert i særlig grad. Fra arbeidet med casen og fra drøftingen vi har gjort, har vi dessuten funnet en del problemer det kan være interessant å se videre på, men som dessverre faller utenfor rammen for en masteroppgave.

I casestudien realiserte vi bare en mellomvarearkitekturløsning. Det ville være interessant å realisere den tenkte løsningen med tjenesteorientert arkitektur for å underbygge konklusjonene i oppgaven. For videre å underbygge eller avkrefte konklusjonene vi har kommet frem til, er det interessant å teste de ut i nye praktiske eksempler.

Vi ser at å realisere den tjenesteorienterte løsningen i casen ikke er gjennomførbart i samme grad som den realiserte mellomvarearkitekturløsningen. Dette skyldes at Scanpix ikke har utviklet systemene sine i henhold til en tjenesteorientert arkitektur. En tilnærming som hadde vært mer realistisk for casestudien kunne derfor vært å studere en integrasjon der vi hadde modifisert de eksisterende systemene på en slik måte at de passet inn i en tjenesteorientert arkitektur.

Selv om vi har vurdert arkitekturene med hensyn på noen kriterier, var disse valgt ut med hensyn på hva som var relevant i casen. Derfor kan andre kriterier som skalerbarhet, stabilitet og sikkerhet være relevante for å vurdere arkitekturene generelt.

Vi har funnet ut at tjenesteorientert arkitektur er best egnet i løsninger med mange systemer, mens mellomvarearkitektur kan være mest fordelaktig i løsninger med få systemer. Vi har riktignok ikke funnet noen konkrete verdier for hva som er få eller mange systemer. Å finne noen konkrete verdier eller intervaller for dette, kan derfor være interessant.

Et av vurderingkriteriene i denne oppgaven har vært "arbeidsmengde og kompleksitet". I den generelle vurderingen av dette kriteriet har vi fokusert på kompleksitet, og bare argumentert for at arbeidsmengde henger sammen med kompleksitet. I virkelige tilfeller kan vi regne med at arbeidsmengde er mer relevant enn kompleksitet, og å finne modeller for arbeidsmengde kan derfor være verdifullt.

Den matematiske funksjonen som beskriver kompleksiteten i en løsning er avhengig av noen flere faktorer. Disse er den initielle kompleksiteten i en løsning som følge av arkitektur (c), og kompleksiteten i et grensesnitt som følge av arkitektur (X). Vi har ikke funnet noen målbar størrelse for disse, og det er åpenbart interessant med en metode for å finne dem.

Videre er funksjonene for å beskrive kompleksitet sterkt forenklet. Derfor bør man identifisere flere faktorer som har betydning for kompleksiteten i en løsning.

Bibliografi

D. Barry. Service-oriented architecture (SOA) definition. 2002.

Rémi Bastide, Philippe Palanque, Ousmane Sy, and David Navarre. Formal Specification of CORBA Services: Experience and Lessons Learned. *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications OOPSLA '00*, 35(10):105 – 117, Oktober 2000.

Philip A. Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, 39(2):86 – 98, Februar 1996.

Bjørn Tore Egeberg. Bruk av Web Services i Altinn – på vei mot et enklere Norge? Master's thesis, Høgskolen i Agder, 2004.

Thomas Erl. *Service-Oriented Architecture – Concepts, Technology and Design*. Pearson Education, Inc, 2005.

Peng Gong, Ian Gorton, and David Dagan Feng. Dynamic Adapter Generation for Data Integration Middleware. *Proceedings of the 5th international workshop on Software engineering and middleware SEM '05*, pages 9 – 16, September 2005.

Doug Kaye. *Loosely Coupled - The missing Pieces of Web Services*. RDS Press, 2003.

Qusay H. Mahmoud, editor. *Middleware for Communications*. John Wiley & Sons, Ltd, 2004.

Eric Newcomer and Greg Lomow. *Understanding SOA with Web Services*. Addison-Wesley Professional, 2004.

Y. W. Sim, C. Wang, L. Gilbert, and G. B. Wills. An overview of service-oriented architecture. 2005.

Wikipedia.org. Soap. Internett: `http://en.wikipedia.org/wiki/SOAP`, 02 2006a.

Wikipedia.org. Xml. Internett: `http://en.wikipedia.org/wiki/Xml`, 02 2006b.