Komponentteknologi for Distribuert Media Journalering

Roger Werner Olsen
Instituttet for informatikk
Universitetet i Oslo

Hovedfagsoppgave
1. mai 2001

FORORD

Denne hovedfagsoppgaven utgjør det praktiske arbeidet som fullfører min Cand. Scient. grad ved Instituttet for Informatikk (IFI), Universitetet i Oslo. Oppgaven inngår som en del av et prosjekt ved IFI som heter *Distribuert Media Journalering* (DMJ). Møtene i prosjektet hver uke har hjulpet meg til å forstå sammenhengen i prosjektet, og jeg har lært mye om digitale media, både i forhold til representasjon, innholdsanalyse og kommunikasjon.

Jeg vil gjerne benytte anledningen til å takke alle i DMJ-prosjektet for å ha lyttet og kommentert det jeg har bidratt med, og skapt et godt diskusjonsforum for min oppgave. Frank Eliassen har vært prosjektleder på prosjektet, og han har hjulpet meg med nyttige tilbakemeldinger. Min veileder i arbeidet med hovedfagsoppgaven, Ole-Cristoffer Granmo, har hele tiden kommet med verdifull tilbakemelding og rettledning. Jeg vil også takke Viktor Woll Eide for all hjelp i forbindelse med implementasjon og gjennomlesning av min oppgave. De to sistnevnte er doktorgrad stipendiater ved DMJ-prosjektet på IFI.

En stor takk går til min samboer Unni, som disse to årene har støttet meg i arbeidet med oppgaven min. Din hjelp har betydd mer enn du aner. Jeg vil også takke Jogeir Storås, Henrik Abler og Olav Brækhus for nyttig tilbakemelding og gjennomlesning mot slutten av arbeidet med denne oppgaven.

Blindern, 1. mai 2001

Roger Werner Olsen

SAMMENDRAG

Den økende tilgjengeligheten på datamaskiner med multimedia og gode kommunikasjonsmuligheter stiller hele tiden større krav til bedre og mer komplekse tjenester. Fjernundervisning, videokonferanser og distribuert legebehandling viser at samarbeid mellom mennesker også utføres på internett. Mange utnytter mulighetene som ligger i multimedia maskiner og internett til å utvikle nye distribuerte løsninger der analyse av media brukes. Et distribuert overvåkningssystem kan for eksempel finne ut antall personer i en bygning, noe som er mye brukt i banker og andre bygninger med krav til høy sikkerhet. Dette synliggjør et behov for et generelt rammeverk for innholdsanalyse av media.

Distribuert Media Journalering (DMJ) er et prosjekt ved Universitetet i Oslo som har definert en arkitektur for å støtte journalering av media i distribuerte systemer. Media journalering er et utvidet begrep for distribuert sanntids innholdsanalyse og annotering av media. Arkitekturen skal støtte kommunikasjon, distribusjon, migrering, modularitet, fleksibilitet og rekonfigurering i journalering av media.

I denne rapporten utvikles det en prototype som implementerer noen av komponentene i DMJ arkitekturen for å se om de nevnte kravene kan støttes. Det er spesielt behovet for en komponentmodell, kommunikasjon mellom komponentene og migrering som utdypes. Prototypen utvikles med teknologiene *Voyager Universal ORB* og *Message Bus* (MBus). Voyager støtter komponentmodell og mobilitet med mobile agenter. MBus støtter utveksling av meldinger mellom de løst koblede komponentene med *IP multicast*, og tilfredsstiller kommunikasjonsbehovet i en eventbroker. Video sendes som *Real-time Transfer Protocol* (RTP) data og analyseres med bruk av *Java Media Framework* (JMF) i analysekomponentene.

Resultatene viser at migrering av analysekomponentene i nettverket tar kortere tid enn å starte en ny komponent. Forskjellen er imidlertid liten, siden JMF ikke støtter serialisering av prosesseringsobjektet. Det viser seg også at det tar lang tid å laste klassene i den *Java Virtuelle Maskinen* (JVM). Selve migreringen fra en maskin tar forholdsvis kort tid, siden størrelsen på filen er relativt liten. Oppsummert viser implementasjonen av prototypen at alle kravene til DMJ arkitekturen kan støttes med den valgte komponent-teknologien.

ABSTRACT

The ever increasing access to computers with multimedia and good communication facilities calls for better and more complex services. Distance education, video conferences and distributed medical treatment shows that cooperation between people is also carried out on the Internet. Many people utilize the potentials of multimedia computers for developing new distributed solutions in which media analysis is applied. A distributed surveillance system can for example find the number of individuals in a building – a system much used in banks and other buildings in need of strong security measures. This displays a need of a general framework for content analysis of media.

Distributed Media Journaling (DMJ) is a project at the University of Oslo that has defined an architecture to support journaling of media in distributed systems. Media journaling is an extended concept for involving real time content analysis and media annotation. The architecture will support communication, distribution, migration, modularity, flexibility and reconfiguration in journaling of media.

In this report, a prototype which implements some of the components in the DMJ architecture is developed to see if the above-mentioned requirements can be supported. It is especially the demand for a component model, the communication between the components and the migration that will be elaborated on in this report. The prototype is developed with the technologies *Voyager Universal ORB* and *Message Bus* (MBus). Voyager supports the component model and mobility with mobile agents. MBus supports exchange of messages between the loosely connected components with IP multicast, and does meet the communication requirements in an eventbroker. Video is sent as *Real-time Transfer Protocol* (RTP) data and is analyzed by the use of *Java Media Framework* (JMF) in the analysis components.

The results show that the migration of the analysis components in the network is faster than starting a new component. Nevertheless, the difference is small as JMF doesn't support serializing of the processing object. It also turns out that it takes long time to load the classes in the *Java Virtual Machine* (JVM). The migration in itself does not take much time as the size of the file is relatively small. To sum up, the implementation of the prototype shows that all requirements for the DMJ framework can be supported by the chosen component technology.

INNHOLD

Kapittel 1

INNLEDNING

Denne hovedfagsoppgaven tar for seg hvordan komponentteknologi kan støtte distribuert journalering av media. Journalering av media er et utvidet begrep for opptak og analyse av media i distribuerte systemer. Komponenter benyttes i denne oppgaven om software komponenter. En applikasjon kan enkelt utvikles av komponenter med bruk av komponentteknologi. Utfordringen ligger her i å kunne kombinere komponentteknologi og journalering av media i distribuerte systemer.

I oktober 1998 startet prosjektet *Distribuert Media Journalering* (DMJ) ved Universitetet i Oslo, Instituttet for informatikk. Prosjektets mål er å utvikle et generelt rammeverk som skal støtte innholdsanalyse og annoteringer i sann tid av alle typer medier i distribuerte systemer. Denne rapporten skal bidra med et forslag til hvordan komponentteknologi kan benyttes i DMJ rammeverket.

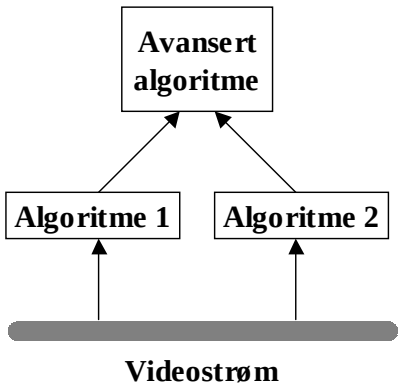
1.1 Bakgrunn og motivasjon

Kameraovervåkning er for tiden mye benyttet både på offentlige steder og i TV-programmer. Dette er et typisk eksempel på et system hvor distribuert journalering av media kan brukes. Kameraer er plassert på forskjellige steder og hvert kamera er ofte tilknyttet en datamaskin. Datamaskinene kan være koblet sammen i et nettverk slik at videostrømmen kan lagres i en sentral database for ivareta sikkerheten. Videostrømmen blir sendt gjennom nettverket og kan derfor analyseres enten sentralt eller ute på hver enkelt maskin. Målet med overvåkningen kan være å finne ut hvor mange personer som befinner seg i bildet og lagre videostrømmen som kommer fra kameraene. Dersom det ikke er noen personer i bildet kan denne lagringen opphøre.

Videre vil det bli sett på hvordan analyse av digitale media fungerer og hvordan komponentteknologi kan bidra til at dette utføres enklere. Overvåkningssystemet som ble nevnt benyttes til å enklere forstå betydningen av teknologiene.

1.1.1 Analyse av media

Mennesket tolker impulser fra de naturlige sensorer (øyne, ører, berøring) kontinuerlig, mange ganger uten å tenke på det. Tolkningen av disse impulsene gir et resultat som mennesket bruker for å ta avgjørelser. Et eksempel er om man skal kjøre hvis det er grønt lys. På samme måte som mennesket fanger opp sine impulser fra de naturlig sensorer, analyserer spesielle algoritmer digitale medier direkte. De spesielle algoritmene sender hendelser eller eventer til mer avanserte algoritmer. De avanserte algoritmene tolker hva hendelsene inneholder, og kan utføre handlinger på bakgrunn av denne analysen (Figur 1-1). Ved å sammenligne disse observasjonene kan det forenklet sies at analyse av media tilsvarer menneskets tolkning av nerveimpulser fra sine naturlige sanser. Se [14] for en oversikt over innholdsanalyse av media.



Figur 1-1 Enkel illustrasjon for analyse av video.

En enkel modell som overordnet beskriver tolkningsprosessen som mennesker bruker er den lingvistiske ”Uttrykk – Innhold modellen” [7]. ”Uttrykk” er kun en beskrivelse av det mennesket ser, mens ”innhold” er en tolkning av beskrivelsens mening i en kontekst eller sammenheng. Et eksempel fra den digitale verden er forholdet mellom OCR (Optical Character Recognition) og ICR (Intelligent Character Recognition). OCR beskriver bokstaver som skannes fra et ark mens ICR analyserer bilder og gjenkjenner strukturer og design i dokumentet [14]. Mennesker klassifiserer videre innholdet etter tidligere erfaringer og kunnskap i hukommelsen. Dette gjøres for å finne meningen med innholdet eller hva ordene som leses betyr. Forskning innen kunstig intelligens har vist at kunnskap bør representeres i en kunnskapsbase (KB) for å kunne brukes effektivt. Det finnes mange tilnærminger på hvordan kunnskapen er representert i KB, og det kreves at bruken av kunnskapsbasen kontrolleres slik at analysen blir optimal.

Overvåkningssystemet i det tidligere nevnte eksempelet må utføre en lignende analyse av videoen fra kameraet som i den lingvistiske modellen. Spesielle algoritmer må trekke ut informasjon om innholdet i videoen som brukes til å analysere antall personer som finnes i videoen. Det er ofte at flere kameraer overvåker samme sted, slik at informasjonen kan verifiseres. I slike tilfeller analyseres media på et høyere nivå og stiller større krav til kommunikasjon.

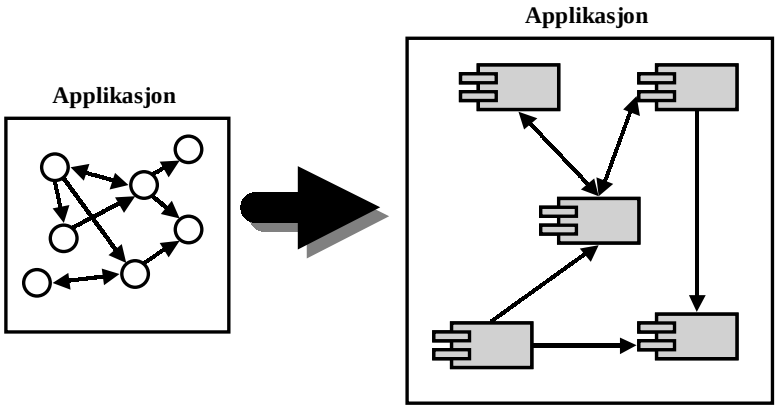
1.1.2 Kommunikasjon

Analyse av media er veldig komplekst og krever enorme ressurser. Bare tenk på hvor mye tid mennesker bruker for å tolke det vi ser og opplever. En måte å gjøre komplekse operasjoner mer effektive kan være å distribuere arbeidet på flere datamaskiner. Dette blir nesten på samme måte som gruppearbeid for å løse en vanskelig oppgave for oss mennesker. Når flere datamaskiner skal samarbeide er det også behov for kommunikasjon mellom disse datamaskinene, på samme måte som en diskusjon i gruppen. Det finnes mange muligheter for slik kommunikasjon mellom datamaskiner, og den metoden som er best egnet for analyse av media er ikke opplagt.

Økt tilgjengelighet av multimedia datamaskiner til privat og kommersielt bruk, samt mulighetene for multimedia både på internett, høyhastighetsnett og mobile nett, synliggjør behovet for distribuerte multimedia journalerings applikasjoner. I denne sammenhengen spiller mellomvare en sentral rolle, fordi utviklere av distribuerte systemer kan tilbys tjeneste-orienterte abstraksjoner. Mellomvare defineres gjerne som ”limet” mellom to eksisterende programmer. Common Object Request Broker Architecture (CORBA), Distributed Component Objekt Model (DCOM) og Java RMI (Remote Method Invocation) er standarder som er velkjente innen kommunikasjon. Blant annet disse vil vurderes som kommunikasjonsmodell i journalering av media i et distribuert miljø.

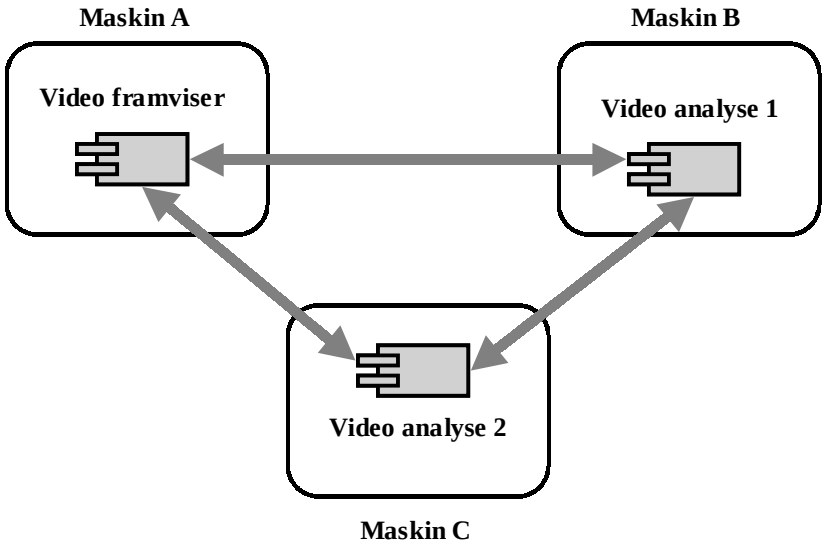
1.1.3 Komponentmodell

Komponentorientering er mer og mer brukt til utvikling av applikasjoner i dag. Komponentorientering betyr utvikling av applikasjoner med bruk av komponenter. Dette er først og fremst fordi komponentbaserte applikasjoner er lettere å vedlikeholde enn store tunge applikasjoner som er utviklet på tradisjonell måte (objektorientering eller andre utviklingsmetoder). Komponenter kan settes sammen til en applikasjon og kan gi samme resultat som ved å utvikle applikasjoner på tradisjonell måte (Figur 1-2). Den store fordelen ved en komponentbasert applikasjon er at man kan bytte ut eller legge til en komponent for å forbedre applikasjonen. For å gjøre dette med en tradisjonelt utviklet applikasjon må man ha kildekoden til programmet, legge til noen objekter der det er hensiktsmessig og kompilere på nytt. Objektorientering blir imidlertid ofte benyttet til å utvikle komponenter.



Figur 1-2 Tradisjonell til komponentbasert applikasjon

Komponentorientering har vunnet stor oppslutning fordi det er enklere å distribuere en applikasjon som er basert på komponenter. Komponentene kan plasseres ut i nettverket og allikevel kommunisere med hverandre (Figur 1-3). I tradisjonell utvikling er dette annerledes. Her må enten applikasjonens grunnleggende struktur endres og applikasjonen deles opp i flere applikasjoner (som komponentorientering), eller så må hele applikasjonen startes på alle maskinene, noe ofte som resulterer i mye unødvendig ressursbruk.



Figur 1-3 Eksempel på en distribuert applikasjon for videoanalyse

Med bruk av komponentorientering utvikles applikasjonen av komponenter, som har vært sitt ansvarsområde og har presist definerte grensesnitt mot de andre komponentene. Komponenter som er ferdiglaget blir brukt der det er mulighet til det for å slippe utviklingskostnadene og spare tid. Kravet til komponenten er kun at den skal tilfredsstille grensesnittene som brukes av de andre komponentene. Hva som skjer ”bak scenen” er ikke viktig for applikasjonen, bare den utfører det som kreves av den. En samling av komponenter i et komponentrammeverk forenkler utviklingen av applikasjoner.

En komponentmodell som beskriver hvordan komponentene skal utvikles og kommunisere med hverandre vil vurderes. Velkjente standarder som Microsoft Component Object Model (COM), Sun JavaBeans og Enterprise JavaBeans vil bli presentert og vurdert som komponentmodell for distribuert journalering av media.

Overvåkningssystemet som ble nevnt i innledningen til kapittelet kan nyttiggjøre seg av komponentteknologier for å distribuere arbeidet mellom maskinene som benyttes. For eksempel kan enkel analyse av videoen gjøres på maskinen som har kameraet tilknyttet seg, og sende data til en sentral komponent som tar seg av større beregninger. Hvis flere kameraer brukes i analysen ser dette også ut som en godløsning. Uten komponentorientering ville alle maskinene som ble benyttet utføre den samme applikasjonen, som skaper mye unødvendig bruk av ressurser.

1.1.4 Migrering

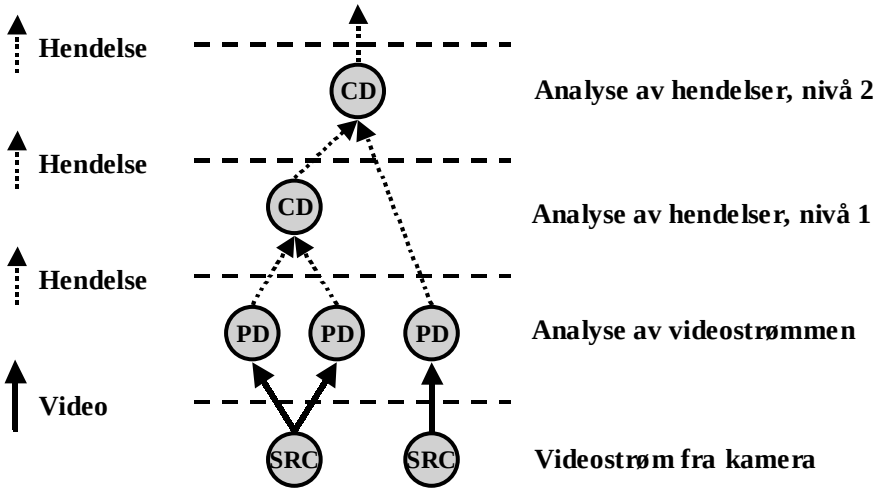
Maskiner i nettverket som benyttes til journalering av media kan variere mye med hensyn på ressurser. Komponentene som utfører journaleringen blir distribuert på maskinene i nettverket for å benytte ressursene bedre enn hvis hele journaleringen skulle utføres på en maskin. Men hva hvis en av maskinene som benyttes får mye å gjøre og journaleringen som gjøres på denne maskinen forsinkes? Komponentene skal i slike tilfeller kunne flyttes eller migreres til en annen maskin. På denne måten benyttes ressursene på alle maskinene best mulig, også under run-time av journaleringen.

1.2 Distribuert Media Journalering (DMJ)

Distribuert Media Journalering [47] som begrep omfatter prosessering av distribuert media i et distribuert miljø. Media journalering referer til en klasse av applikasjoner for å delta i reelle multimedia aktiviteter mens de foregår, samt å lagre et segmentert og synkronisert opptak av multimedia hendelser for avspilling på et senere tidspunkt. Viktige problemområder for multimedia i distribuerte systemer er blant annet støtte for synkroniserte mediastrømmer, QoS (Qualtiy of Service) håndtering, synkronisering av media og multideltaker kommunikasjon [9]. I DMJ legges det spesielt vekt på eventhåndtering / meldingsutveksling, synkronisert multimedia opptak og analyse i sanntid, avspilling og manipulering i disse applikasjonene. Et komponentrammeverk for DMJ må tilfredsstille mange krav til komponentmodell og kommunikasjon som blir beskrevet her.

Media blir journalert i DMJ ved å dele opp analysen i to steg, hvor algoritmene som brukes blir innkapslet som software komponenter [45]. De to typene komponenter er Primitive hendelses Detektorer (PD), og mer avansert analyse

som kan kalles sammensatte hendelses detektorer (CD)[1]. PD'er har ansvaret for å analysere media direkte og sende ut hendelser hvis definerte tilstander i media forekommer. Et eksempel på en hendelse fra PD'en er hvor stor forskjell det er mellom hvert bilde i videoen som indikerer bevegelse. CD'ene bruker hendelser fra PD'ene til å analysere hva mediet inneholder, og produserer også hendelser som andre CD'er bruker (Figur 1-4). Et eksempel på en hendelse som en CD produserer kan være en at det finnes en person i videoen. Høyere abstraksjonsnivå fra videostrømmen betyr som regel at man nærmer seg det brukeren spør om.



Figur 1-4 Analyse på forskjellige abstraksjonsnivå

1.2.1 Krav til DMJ arkitektur

Prosessering i DMJ prosjektet stiller en del krav til arkitekturen for at dette skal være mulig å gjennomføre. Her presenteres kravene som må stilles til arkitekturen for å kunne utføre prosessering i DMJ.

Kommunikasjon
Analysekomponentene (PD'er og CD'er) skal kommunisere med hverandre ved hjelp av hendelser. Derfor er det naturlig at systemet har en eventbroker til å formidle eventene mellom komponentene.

Distribusjon
PD'ene og CD'ene skal være komponenter med spesifikke oppgaver og funksjoner, som enkelt skal kunne distribueres blant flere maskiner i nettverket. Distribusjon gjør at analysen kan gjøres mer effektivt enn hvis alt kjøres på en maskin, siden ressursene på flere maskiner blir utnyttet. Applikasjoner som distribueres kan også skalere bedre enn applikasjoner som utføres på en maskin.

[1] Sammensatt hendelses detektor oversettes til engelsk og blir Composite event Detector (CD), siden implementasjonen er skrevet på engelsk.

Migrering
Analysekomponentene må også kunne flyttes i nettverket hvis en maskin blir overbelastet. Dette kalles migrering og er et nyttig hjelpemiddel for bruke tilgjengelige ressurser på maskinene i nettverket. Det stilles et krav til at komponenten som skal flyttes fortsetter journaleringen til flyttingen gjennomført, slik at ikke analyse av media skal gå tapt under flyttingen.

Modularitet
Komponentene i rammeverket skal kunne brukes til å enklere utvikle media journalering applikasjoner, enten ved å bruke rammeverket som det er eller å utvide funksjonaliteten selv. Komponentene skal kunne benyttes i et annet system som bruker grensesnittene til komponentene. Komponentrammeverket må også være fleksibelt slik at nye komponenter kan benyttes så lenge de tilfredsstiller kravene DMJ rammeverket setter til dem.

Fleksibilitet
Komponentene skal kunne styre journaleringen av media selv inntil en viss grad. Endres rammerate eller kvaliteten i media må komponenten takle dette slik at journaleringen ikke stopper. Komponentene skal også ha mulighet til å styre kvaliteten av media direkte fra kilden (kamera for eksempel). På denne måten vil komponenten ha kontroll over media slik at operasjonen blir utført innen avtalt tid for å støtte sanntid. Noen valg for komponentens fleksibilitet vil andre komponenter har lettere for å ta, noe som betyr at komponenten må ha mulighet til å motta informasjon som styrer komponentens adferd.

Rekonfigurering
Brukeren kan endre spørringen under utføring av journaleringen. Dette kan føre til at et nytt media skal journaleres eller at algoritmer ikke lenger kan brukes. Komponenten må derfor ha støtte for rekonfigurering slik at disse endringene kan tre i kraft i komponenten.

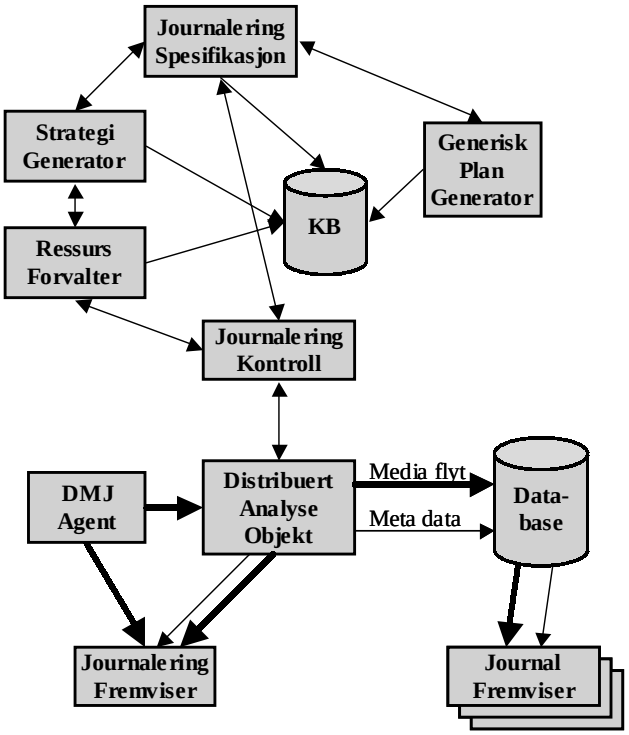
1.2.2 DMJ arkitekturen

DMJ prosjektet har kommet fram til et DMJ rammeverk som støtter kravene som ble satt opp ovenfor [46]. Arkitekturen tar for seg komponenter som alle har et avgrenset ansvarsområde i arkitekturen og tilbyr ny funksjonalitet til rammeverket. Komponentene blir her kommentert i forhold til hverandre og hvilken spesifikk funksjon de har.

Analyse av media gjøres i det *Distribuerte Analyse Objektet* (DAO) med mediastrømmer fra *DMJ Agenter* i nettverket (Figur 1-5). DAO består av analysekomponenter som enten er av typen *Primitiv hendelses Detektor* (PD), *Filter* eller *sammensatt hendelses detektor* (CD). Agentene henter mediastrømmer fra generatorer som for eksempel et kamera, eller fra sammensatte mediasesjoner. *Journalering Fremviseren* kan vise innholdet i media og metadata før, under og etter journaleringen. Det ferdige journalerte materialet blir lagret i en database til seinere bruk.

Journalering Spesifikasjon komponenten kontrollerer at brukeren har laget syntaktiske og semantiske riktige abstrakte spørringer som spesifiserer journalering oppgaver. *Generisk Plan Generator* oversetter spørringer fra

brukeren til en konfigurasjon av media prosessering funksjoner i henhold til en hierarkisk plan. For å velge en måte å løse brukerens spørringer, benyttes *Strategi Generatoren* til å velge den beste strategien. Alle disse tre komponentene bruker informasjon fra en *KunnskapsBase* om algoritmer (for eksempel kostnad og pålitelighet) og domeneprofiler for å kombinere lavnivå analyse og høynivå gjenkjenning.



Figur 1-5 Design av DMJ arkitekturen

Ressurs Forvalteren gjør det mulig å sette planen ut i live, og oppbevarer informasjon om den distribuerte prosessering omgivelsen (DPO). *Journalering Kontroller* (JK) tar en strategisk prosesserings plan som input og instansierer DAO som settes i arbeid i DPO. Den strategiske prosesserings planen inneholder navn på algoritmer som skal brukes i analysen. Analyse komponentene opprettes, kontrolleres og fjernes av JK komponenten, som får sine kommandoer fra *Journalering Spesifikasjon* og *Ressurs Forvalter* komponentene.

Presentasjonen av DMJ arkitekturens viser kompleksiteten i distribuert journalering av media. Presise grensesnitt mot komponentene og valg av teknologier som understøtter arkitekturen blir derfor en stor oppgave.

1.3 Problemstillinger og målet med oppgaven

Journalering av media er generelt en kompleks oppgave som krever store ressurser, noe som gjør at journaleringen bør foregå distribuert på flere maskiner. Komponentorientering gjør det enklere å distribuere en applikasjon, siden komponentene er deler av applikasjonen som kan plasseres ut i nettverket. Distribuert Media Journalering stiller en del krav til prosessering av media i et distribuert miljø som støttes i DMJ arkitekturen. Utfordringene videre vil være design, teknologivalg og implementasjon av komponentene, slik at de støtter kravene til DMJ arkitekturen. Hovedproblemstillingen blir derfor:

> *Hvordan kan state-of-the-art komponentteknologi understøtte kravene til DMJ arkitekturen?*

Det legges spesiell vekt på design og implementasjon av en prototype bestående av komponenter fra DMJ arkitekturen for å belyse dette spørsmålet. Prototypen vil bestå av analysekomponentene, Eventbrokeren og Journalering Kontroller komponenten. Målet med prototypen er at design og teknologier for de nevnte komponentene støtter kravene til DMJ arkitekturen.

Hovedproblemstillingen deles opp i tre underpunkter som tar for seg ulike deler av hovedproblemstillingen.

1.3.1 Hvilken komponentmodell skal benyttes i prototypen?

Prototypen skal ta for seg de viktigste komponentene i DMJ rammeverket for at kravene til DMJ arkitekturen skal støttes. Komponentene skal benytte en komponentmodell slik at applikasjonen kan distribueres i nettverket. Modellen må støtte sammenkobling av komponenter hvor fleksibilitet er viktig for at nye komponenter som skal kunne legges til. Komponentmodeller som JavaBeans, Enterprise JavaBeans og COM blir studert og Mobile Agenter blir diskutert som en mulig komponentmodell.

1.3.2 Hvilken kommunikasjonsmodell skal brukes mellom komponentene i prototypen?

Komponentene skal distribueres i nettverket, og det er derfor viktig å finne en kommunikasjonsmodell som passer for journalering av media i et distribuert miljø. Standarder som Java RMI (Remote Method Invocation), CORBA (Common Object Request Broker Architecture) og DCOM/COM+ (Distributed Component Object Model) blir evaluert. Analysekomponentene skal kommunisere med eventer og forskjellige teknologier som CORBA Event Notification, Java Messaging Service (JMS), Voyager Advanced Messaging og Message Bus (MBus) blir evaluert.

1.3.3 Hvordan skal migrering støttes i prototypen?

Analysekomponentene skal distribueres i nettverket og det stilles et krav om at de skal kunne flyttes. Migrering av analysekomponenter gjør at kapasiteten til maskinene i nettverket utnyttes bedre. Komponentmodellene JavaBeans, EJB og COM, samt Mobile Agentsystemer blir evaluert som en mulig løsning på migrering av komponenter.

1.4 Avgrensing

Journalering av media er i seg selv et stort tema, hvor media betyr alt fra tekst til video, og journalering er et samlebegrep for applikasjoner som brukes til håndtering av media i distribuerte systemer. Video beregnes for å være det mest krevende mediet å journalere, og analyse av video er en vesentlig del av journalering av media. Implementasjon av prototypen tar derfor kun for seg analyse av video.

Arkitekturen som defineres av DMJ prosjektet tar for seg mange komponenter som skal støtte for eksempel kunstig intelligens. I prototypen implementeres de komponentene som er nødvendig for å støtte de kravene som stilles til design av DMJ arkitekturen.

1.5 Metode

Metodene som ble benyttet i denne rapporten er observasjon, litteraturstudier og eksperimentell metode. I dette delkapittelet vil det beskrives hvordan disse tre metodene ble brukt for å finne løsninger problemstillingene.

Litteraturstudiene utgjorde av søk på internett og i biblioteker, for å finne relevant teori. Søk på litteratur som omhandlet komponentteknologier, kommunikasjon i distribuerte systemer og agentsystemer sto for en stor del av dette arbeidet. Det finnes mye informasjon om disse temaene, så det var viktig å finne arbeider som var akseptert i fagmiljøet.

Eksperimentet i denne oppgaven gikk ut på å implementere et forslag til komponentrammeverk. Implementasjonen tok tidlig sikte på å lage et fullt rammeverk for distribuert journalering av media. Dette viste seg å være for ambisiøst og noen utvalgte komponenter ble valgt i stede. *Unified Modeling Language* (UML) ble brukt for å definere grensesnitt til komponentene, og Java ble brukt som programmeringsspråk.

I perioden rapporten ble skrevet fulgte jeg et prosjekt ved Universitetet i Oslo som heter Distribuert Media Journalering (DMJ). Observasjonene jeg gjorde i DMJ prosjektet hadde stor betydning for arbeidet med rapporten, og mange ideer til implementasjonen har kommet fra diskusjoner i prosjektet.

1.6 Organisering av rapporten

Denne hovedoppgaven følger vanlig utviklings metode som for eksempel er beskrevet i [17], der beskrivelse av problemområdet er gjort i dette kapittelet. Kapittel 2 tar derfor for seg DMJ som begrep og gir en oversikt over DMJ prosjektet med krav til design og hvordan DMJ prosjektet løser dette i sin design av arkitektur. Design av komponentene i prototypen blir beskrevet i kapittel 3, og skal støtte kravene til design som settes i kapittel 2. Det er viktig at teknologier støtter designet som blir valgt. Dette blir diskutert i kapittel 4 og vurdert opp mot DMJ arkitekturen slik at implementasjonen løser de designkrav som settes i kapittel 2. I kapittel 5 blir implementasjonen beskrevet i detalj. Kapittel 6 tar for seg testing av implementasjonen og diskuterer andre muligheter som kunne gitt et bedre resultat. Konklusjonen i kapittel 7 er en sammenfatning av det viktigste i rapporten, og gir i avslutningen en kommentar til videre arbeid.

Det benyttes mange forkortelser i denne oppgaven. Navnene på komponentene i DMJ rammeverket repeteres ofte og mange teknologier har godt innarbeidede forkortelser. En ordliste over de mest brukte forkortelsene finnes i Appendix B.

Kapittel 2

DESIGN AV KOMPONENTENE I PROTOYPEN

Ved implementasjonen av prototypen blir man stilt overfor mange valg som ikke har relevans til teknologi. Dette kapittelet tar for seg valgene som er tatt for design av prototypen, og det blir kommentert alternative tilnærminger der det er mulig. Målet med prototypen er å tilfredsstille kravene som stilles til design av DMJ arkitekturen i kapittel 1.2.1. Kravene blir her diskutert opp mot designet av prototypen.

De viktigste komponentene i prototypen er analyse komponentene, siden det er dem som gjør selve analysen. Ressursene bør fordeles mellom komponentene slik at maskinene i nettverket blir utnyttet maksimalt. Analysekomponentene kan derfor flyttes til den maskinen det finnes mest ledige ressurser, og derfor er distribusjon og migrering viktig for disse komponentene. Journalering Kontroller og Eventbroker komponenten skal kun legge mulighetene til rette for at analysekomponentene skal oppfylle kravene til designet.

Spørsmål som blir besvart i dette kapittelet er:

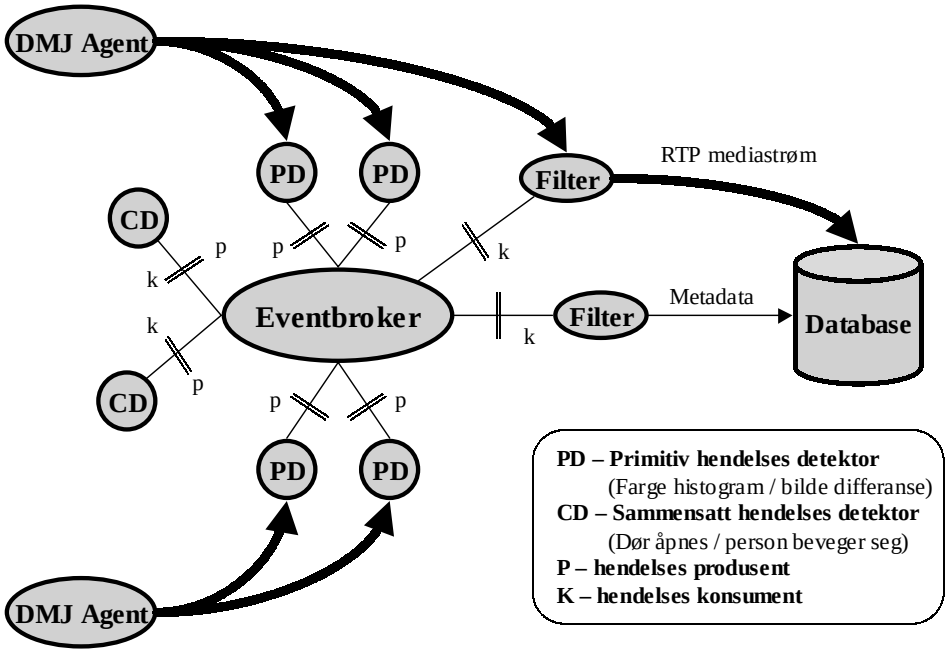
- Hvordan skal grensesnittet i analysekomponenten støtte distribuering, migrering, kommunikasjon, modularitet og rekonfigurering?
- Skal Eventbrokeren og Journalering Kontroller komponenten være sentrale eller distribuerte komponenter?
- Hvordan skal Eventbrokeren støtte bruk av flere kommunikasjons-modeller?
- Hvordan skal sending og mottak av eventer støttes i grensesnittet til Eventbrokeren?
- Hvordan skal grensesnittet mot journalering Kontroller komponenten se ut?

2.1 Design

Her beskrives komponentene overordnet og grensesnittene mot komponentene blir spesifisert. Valg som gjøres blir kommentert underveis.

2.1.1 Analysekomponentene

Distribuert Analyse Objekt (se Figur 1-5) er ikke en virkelig komponent i DMJ rammeverket, men er kun et abstrakt navn på samlingen av de underliggende analysekomponentene *Primitiv hendelses Detektor* (PD), *sammensatt hendelses detektor* (CD), og *Filter*. PD'er analyserer mediastrømmer direkte og genererer hendelser (Figur 2-1). CD'er analyserer eventene fra PD'ene for å finne hva mediene inneholder og produserer hendelser som andre CD'er bruker. Filter analyserer mediastrømmene direkte slik som PD'ene, men bruker hendelser fra andre PD'er og CD'er for å produsere en endret mediastrøm, men de kan også produsere kun metadata som hentes fra hendelsene.

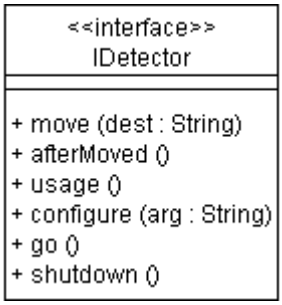


Figur 2-1 Eventbroker og Analyse komponentene

PD'er trekker ut lavnivå egenskaper fra media, som for eksempel fra video kan være farge-histogram, mønster, bevegelse, eller objekter. CD'ene bruker disse lavnivå egenskapene fra media til å generere hendelser med høyere abstraksjonsnivå, som ofte går ut på å kombinere flere typer hendelser. Eksempler på dette kan være start eller slutt av en sekvens i media, eller hendelser som "person i bilde", "person befinner seg mellom to definerte steder" eller "Personen befinner seg i rommet". I [21] brukes *Dynamic Object-Oriented Bayesian Network* (DOOBN) og i [37] bruker *Hidden Markov Models* (HMM) for gjenkjenning av høynivå hendelser. Innholdet i CD'ene styres av komponenter som ikke er vesentlig for design av komponentrammeverket, og tas derfor ikke med i denne rapporten.

Analysekomponentene er selve drivkraften i journalering av media. Det er derfor viktig at disse komponentene er distribuert i nettverket med mulighet for migrering. Grensesnittet som er skissert i UML diagrammet i Figur 2-2 definerer *move* til å flytte komponenten til en ny destinasjon. Metoden *aftermoved* blir kalt etter komponenten er flyttet og har ansvaret for å sette

komponenten i gang igjen. Komponenten skal selv beskrive hvordan den skal brukes, og til dette brukes metoden *usage.* For å konfigurere komponenten til nye oppgaver brukes *configure* som kan brukes både før og etter komponenten har startet. Det er legges vekt på konfigurering av hvilke algoritmer som brukes, lokalisering av media som skal analyseres og eventuelle parametere som terskelverdier i analysen. Metoden *go* brukes for å sette komponenten i gang og den kalles fra *aftermoved* når komponenten er flyttet. *Shutdown* avslutter oppgaven som komponenten er satt til å utføre, og fjerner komponenten fra maskinen slik at den ikke lenger bruker ressurser.



Figur 2-2 Analysekomponenten

2.1.2 Eventbroker

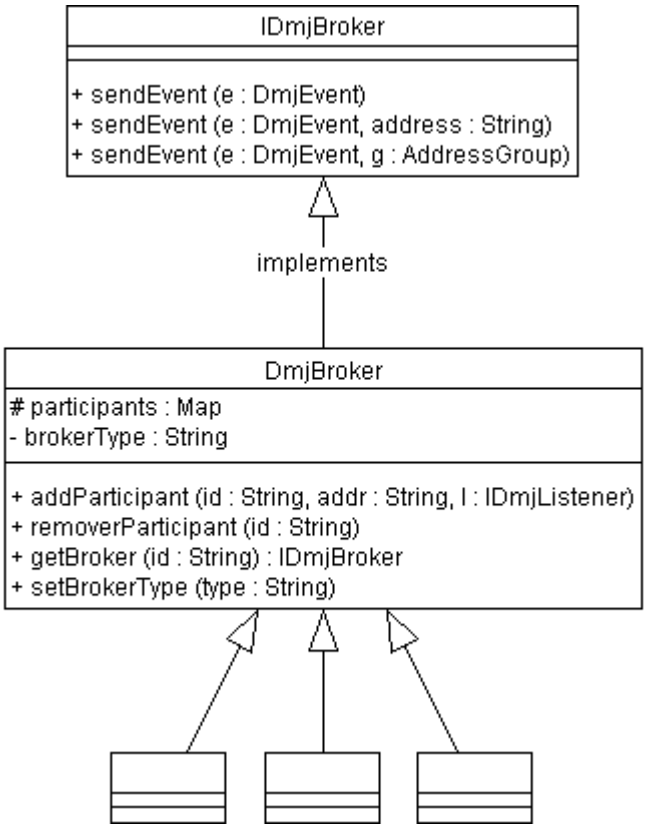
Eventbrokeren er selve limet mellom analysekomponentene slik Figur 2-1 viser. All kommunikasjon mellom PD'er, CD'er og Filtre gjøres ved hjelp av eventer (hendelser). En primitiv detektor kan for eksempel finne fargehistogrammet til bilderammer i en mediastrøm og distribuere disse til andre analysekomponenter ved å benytte eventbrokeren. Eventbrokeren kan sies å være en komponent som forvalter hendelser.

Eventbrokeren kan enten være distribuert på alle maskinene som benyttes eller lokaliseres på en bestemt maskin, siden denne komponenten ikke skal flyttes blant maskiner i nettverket. Hvis eventbrokeren skal være sentral må alle komponentene som bruker den vite om hvor den finnes. Med en sentral eventbroker må alle eventer gå gjennom en maskin i nettverket, som lett kan bli en flaskehals. Når eventbrokeren er distribuert unngås en slik flaskehals i nettverket, og komponentene kan sende eventer uten å vite hvor mottakeren befinner seg. En distribuert eventbroker er derfor brukt i dette designet.

Grensesnittet mot eventbrokeren støtter både sending og mottak av hendelser. Grensesnittet *IDmjBroker* (Figur 2-3) støtter forskjellige metoder for sending av hendelser, hvor både unicast og multicast er støttet ved at hendelsen sendes til en adresse. Adressen er beskrevet i en tekst eller et *AddressGroup* objekt hvor utvikling av implementasjonen kan velge hva adressen skal bety. Broadcast er støttet ved sending av hendelsen uten at adressen er spesifisert.

For å motta hendelser fra eventbrokeren må det registreres deltakere i *DmjBroker* klassen. Dette gjøres med *addParticipant* hvor et navn eller identitet på deltakeren blir gitt i første argument, adressen som deltakeren har i andre, og det siste argumentet er komponenten som lytter på hendelser. Deltakeren som lytter på eventer blir kalt hvis et event fanges opp av eventbrokeren. For å fjerne en deltaker som lytter etter hendelser, brukes *removeParticipant.*

Designet av eventbrokeren er også meget fleksibel for hvilken teknologi som brukes, fordi klassen som definerer kommunikasjonen arver fra *DmjBroker* klassen, som det vises i UML diagrammet nedenfor. Valget mellom hvilken metode man vil bruke for kommunikasjonen blir valgt så sent som mulig, og kan settes med *setBrokerType.* En komponent som bruker eventbrokeren må også kunne sende eventer, og for å gjøre det hentes en referanse til klassen som definerer kommunikasjonen med *getBroker*. Teksten i argumentet definerer identiteten til den deltakeren (komponenten) som bruker eventbrokeren. På denne måten kan flere deltakere melde seg på samme eventbroker og bruke forskjellige kommunikasjonsmetoder.



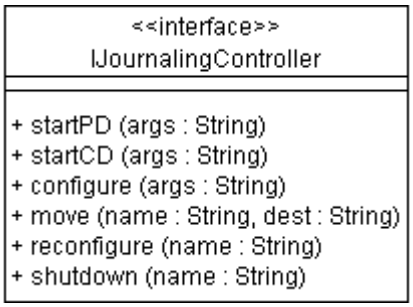
Figur 2-3 Eventbroker

2.1.3 Journalering Kontroller (JK)

Journalering Kontroller komponenten skal ta seg av opprettelsen, kontroll og destruering av analysekomponenter. JK får en plan for eksekvering fra Journalering Spesifikasjon (JS) komponenten, og oppretter og distribuerer de nødvendige komponentene for å tilfredsstille planen. Den kan også flytte komponenter blant maskinene i nettverket dersom ressursforvalteren eller JS ber om dette. Endring av analysekomponentenes utføring av analysen kan også JK komponenten gjøre.

Journalering Kontroller bør være en sentral komponent, fordi den skal opprette komponenter som skal migreres blant maskinene i nettverket. Det er ikke behov for at denne komponenten skal være distribuert siden det meste av kommunikasjon skal gå mellom eventbrokeren som skal være distribuert. JK komponenten skal opprette og styre analysekomponentene, som i utgangspunktet kun fører til flaskehals hvis Ressursforvalteren og JS komponenten vil ha noe utført samtidig. Maskinnavnet og portnummeret JK er registrert på kan eventuelt settes når komponenten startes opp, slik at komponenten kan flyttes før journaleringen begynner (Statisk plassering).

Komponenten skal starte, kontrollere og avslutte analysekomponenter. Den må derfor ha et grensesnitt som tilbyr disse tjenestene (Figur 2-4). Metodene som er definert i grensesnittet for å starte PD'er og CD'er tar parametere til analysekomponenten som argument. Parameterne inneholder for eksempel algoritmer og terskelverdier som brukes i analysen. Det er også mulig å konfigurere JK komponenten med *configure* som vist i figuren. Her kan det for eksempel settes om komponenten skal skrive ut debug-informasjon eller ikke. Det er mulig å flytte en analysekomponent mellom maskiner i nettverket som er støttet med metoden *move*. Parameterne beskriver hvilken komponent som skal flyttes og til hvilken adresse. Analysekomponenter skal kunne bytte algoritmer som brukes eller stille på parametere i algoritmene de allerede bruker. Til dette brukes metoden *reconfigure*. De nye innstillingene sendes med som parameter. *Shutdown* brukes for å avslutte en komponent og fjerne den fra maskinen den har blitt utført på. Teksten i parameteren i *reconfigure* og *shutdown* brukes til å identifisere analysekomponenten.



Figur 2-4 Journalering Kontrollerer

2.2 Evaluering av design

Komponentene er designet for å støtte de kravene som ble listet opp i kapittel 1.2.1. Her blir disse kravene vurdert opp mot designet som er presentert.

2.2.1 Kommunikasjon

Kommunikasjon støttes av alle komponentene gjennom eventbrokeren, som er en komponent som kan konfigureres til å bruke egne kommunikasjons-modeller. Analysekomponentene bruker eventbrokeren til å sende og motta hendelser, og er distribuert på de maskinene som kommuniserer over eventbrokeren.

2.2.2 Distribusjon og migrering

Distribusjon og migrering er viktig for at analysekomponentene skal kunne bruke ressurser på alle maskiner som er tilgjengelig. Dette støttes i metoden *move* i analysekomponentene. Analysekomponentene opprettes i Journalering Kontroller komponenten, og blir utplassert ved *move*. Når komponenten skal flyttes ved en seinere anledning brukes samme metode, noe som vil si at funksjonaliteten for å flytte komponenten må ligge i analysekomponenten. Analysekomponenten skal flytte seg selv, men skal ikke bestemme selv når og hvor den skal flytte.

2.2.3 Modularitet

Med modularitet menes det at komponentene skal kunne benyttes i andre systemer som oppfyller kravene komponenten stiller til run-time miljøet. Dette er også noe av grunnlaget for å velge komponentorientering, som stiller krav om at modularitet støttes. Det viktigste for å støtte modularitet er at komponentene har presist definerte grensesnitt, og at komponenten er inkluderende for nye komponenter i rammeverket. Eventbrokeren kan brukes av alle komponenter som bruker grensesnittet i Figur 2-3. Journalering Kontroller komponenten og analysekomponentene kan benytte seg av algoritmekomponenter som kan legges til seinere, og støtter dermed modularitet.

2.2.4 Fleksibilitet og rekonfigurering

Rammeverket er fleksibelt fordi det er mulig å styre analysekomponentene ved å konfigurere dem før de starter, og rekonfigurere dem etter de har startet. De har mulighet til å tilpasse seg ressurser som er tilgjengelige for komponenten. Design av prototypen legger ingen stopper for at komponentene skal ha mulighet til å styre bruken av media, noe som er en del av DMJ Agenten. Dette er også avhengig av teknologien som brukes, noe som blir diskutert videre seinere i rapporten.

2.2.5 Eksempel-case for prototypen

En typisk konfigurasjon for media journalering applikasjoner implementerer algoritmer som analyserer media og innholdsanalyse i samme kjørbare program, uten at det er muligheter for dynamiske utvidelser. Applikasjonen i [37] foretar automatisk indeksering av nyhetssendinger. Hendelser som begynnelse og avslutning på nyhetssendingen, nyhetsinnslag og værmelding blir detektert med bruk av *Hidden Markov Models* (HMM). Det lages også noen algoritmer som trekker ut egenskaper fra videoen som brukes i analysen av innholdet. Hele applikasjonen kjøres på en enkelt maskin og lærer hva som er korrekt deteksjon ved å kjøre forskjellige nyhetssendinger.

Målet med DMJ komponentrammeverk er at denne typen applikasjoner enkelt kan implementeres med komponenter fra rammeverket. I stedet for å lage hele applikasjonen som en kjørbar fil, kan den deles opp i komponenter. Algoritmene hentes fra algoritmebiblioteket hvor de mest brukte algoritmene er ferdig implementert, men med muligheter for å legge til egne algoritmer selv. Applikasjonen i eksempelet implementerer mange algoritmer som er standard, og vil finnes i algoritmebiblioteket. Algoritme komponentene kommuniserer med en sammensatt hendelses detektor (CD) komponent, ved hjelp av Eventbroker komponenten. CD komponenten representerer analysen av innholdet i videoen, og bruker HMM, DOOBN eller tilsvarende. På denne måten blir hele applikasjonen bygget opp av komponenter som allerede er definert i komponentrammeverket, noe som er mye enklere enn å lage alt selv. Det eneste man skal behøve å gjøre, er å konfigurere komponentene (grenseverdier og hvor videoen befinner seg osv.) og kjøre applikasjonen. Flere fordeler med å bruke komponentrammeverket er muligheten for å distribuere komponentene slik at ressursene på flere maskiner kan brukes samtidig. Videre kan flere og mer komplekse analyser av media utføres, samt at vedlikehold og endringer i applikasjonen forenkles.

2.3 Oppsummering

I dette kapittelet har de viktigste komponentene blitt designet med grensesnitt mot andre komponenter. Det er viktig at disse grensesnittene er definert presist, slik at implementasjonen som støtter disse grensesnittene har den

funksjonaliteten som de skal. Grensesnittene er også uavhengig av teknologi som brukes, slik at den neste fasen blir å diskutere hvilken teknologi som støtter funksjonaliteten til rammeverket best mulig.

Designet av komponentene som er diskutert i dette kapittelet støtter alle krav som settes til komponentene. Kommunikasjon støttes ved eventbroker komponenten. Migrering støttes ved at analysekomponentene skal kunne flyttes, noe som også brukes til utplassering av komponentene (distribusjon). Komponentene skal kunne brukes som de er, eller de kan utvides. Nye algoritmer kan legges til og benyttes uten å endre noe, og det samme gjelder for eventbrokeren som kan utvides med nye kommunikasjonsmodeller. Komponentene er også fleksible med mulighet for blant annet konfigurering før de startes og rekonfigurering mens de kjøres.

Designet av komponentene med brukergrensesnitt er utført i dette kapittelet. Det gjenstår å finne teknologier som støtter disse brukergrensesnittene, slik at funksjonaliteten som er beskrevet i dette kapittelet opprettholdes. Kriteriene for DMJ arkitekturen vil bli diskutert slik at teknologiene som velges er mest mulig optimale for prototypen.

Kapittel 3

VURDERING AV TEKNOLOGI

Dette kapittelet tar for seg forskjellige teknologier som kan støtte designet av komponentene som er beskrevet i kapittel 2. Kravene som ble satt til DMJ arkitekturen i kapittel 1.2.1 vil være sentrale for valg av teknologi. Disse blir evaluert for hver teknologi som blir presentert.

Komponentene i prototypen er avhengig av en komponentmodell som støtter behovet for modularitet. Dette vil si at komponentene blir utviklet slik at de kan gjenbrukes i andre systemer som bruker samme komponentmodell og som tilfredsstiller grensesnittene mot komponentene. Modellen skal også være åpen for nye komponenter, og innlemme disse i arkitekturen uten endringer. Kjente kommersielle komponentmodeller som JavaBeans, Enterprise JavaBeans og COM vil bli vurdert. Komponentene skal kunne kommunisere med hverandre over maskingrenser, og flyttes fra maskin til maskin under run-time. I forbindelse med migrering blir mobile agenter vurdert å kunne brukes i stedet for andre komponentmodeller.

Behovet for kommunikasjon mellom komponentene er presentert med en eventbroker. Denne komponenten skal være distribuert slik designet krever, og derfor kan det hende at noen kommunikasjonsmodeller passer bedre enn andre. Ulike måter å håndtere eventer diskuteres i større grad enn andre kommunikasjonsmodeller på grunn av behovet for en eventbroker. Velkjente modeller som RPC/Java RMI, CORBA og DCOM blir vurdert. Andre muligheter som for eksempel støtter multicast presenteres også.

Spørsmål som blir besvart i dette kapittelet er:

- Hvilken komponentmodell egner seg best i DMJ arkitekturen?
- Hvordan støttes kravene til DMJ arkitekturen i de forskjellige modellene?
- Hva kjennetegner eventbasert kommunikasjon, og hvordan kan dette støttes i eventbrokeren?
- Hvordan støtter forskjellige kommunikasjonsmodeller multicast?
- Hva kjennetegner agentsystemer, og hvordan støtter Voyager mobilitet?
- Egner Voyager seg som komponentmodell?
- Hvilken kommunikasjonsmodell egner seg best i eventbrokeren?

3.1 Komponentmodell

Modularitet er et av kravene til DMJ arkitekturen og dette stiller krav til komponentmodellen. Komponentorientering har mange fordeler som har stor betydning for utvikling av prototypen. Her blir det beskrevet litt generelt om komponenter og hva som er forskjellen mellom et objekt og en komponent. Deretter vil de viktigste komponentmodellene som er tilgjengelig bli beskrevet og vurdert opp mot kravene til DMJ arkitekturen. Det vil også bli vurdert om de støtter behovet for den funksjonaliteten som designet beskriver.

3.1.1 Generelt om komponenter

> *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*
>
> - Clemens Szyperski [5] (side 34)

> *"A component is a piece of software small enough to create and maintain, big enough to deploy and support, and with standard interfaces for interoperability."*
>
> - Jed Harris [32] (side 831)

> *A software component is reusable pieces of code and data in binary form that can be plugged into other software components from other vendors with relatively little effort. Software components must adhere to an external binary standard, but their internal implementation is completely unconstrained.*
>
> - Kraig Brockschmidt [16]

Szyperski [5] fokuserer på det at en komponent skal være gjenbrukbar. Den skal kunne utføre en tjeneste for systemet som benytter seg av den, og den skal kunne brukes av denne omgivelsen uten at komponenten endres. Det er viktig at grensesnittet til en komponent er godt definert og at metoder utfører nøyaktig det de skal. I tillegg skiller Szyperski mellom en programkomponent (software) og fysiske komponenter (hardware). I denne rapporten brukes komponent i betydning av programkomponent.

I boken til Orfali og Harkey [32] omtales komponenter som lages i programmeringsspråket Java. Likevel omfatter definisjonen til Jed Harris mange elementer. Han sier at en komponent må være liten nok til å være stabil og enkel å vedlikeholde, men samtidig stor nok til å støtte det den skal

gjøre. En komponent blir laget for å kunne brukes i flere systemer, og det er derfor viktig at den ikke feiler hvis den blir benyttet etter beskrivelsen. Den bør ha en måte å håndtere feilsituasjoner på, og ikke bruke unødvendige ressurser. Grensesnittene mot komponenten skal være presist definert slik at den kan benyttes av andre komponenter.

Brockschmidt skiller seg noe fra de andre definisjonene. Definisjonen er muligens noe farget av Microsoft, hvor komponenter kun kan være kjørbare programmer og tjenester (*services*) i Windows NT/2000. Videre skal en komponent være laget fra en binær standard, men innholdet kan være skjult.

Den viktigste grunnen for å velge komponentorientering er gjenbruk av komponentene som lages. Altfor mange ganger blir applikasjoner laget uten tanke på at deler av applikasjonen faktisk kan brukes seinere. Kunnskapen om applikasjonen forsvinner gjerne med dem som lager den, og arbeidet som legges ned i applikasjonen fungerer kun en gang. Hvis applikasjoner utvikles med komponentorientering kan deler av systemet enkelt brukes flere ganger. Komponenten har en klart definert oppgave som den har ansvaret for, og kan styres gjennom grensesnittet mot komponenten.

Et åpent komponentrammeverk må også vurderes, der åpenhet betyr hvor mye den som bruker komponenten skal vite om innholdet og hvordan komponenten løser sin oppgave. En åpen komponent forteller omverdenen hva som skjer i komponenten, ved å tilby grensesnitt som gir informasjon om komponenten. Et eksempel er JavaBeans grensesnittet *BeanInfo*, som tilbyr beskrivelse av komponenten.

Eksempler på komponenter

Etter definisjonene på en komponent, kan mye kalles en komponent. Her er noen eksempler på hva en komponent kan være [5]:

- Typedeklarasjoner
- C makroer
- C++ templates
- Smalltalk block
- Prosedyrer
- Klasser
- Moduler
- Hele applikasjoner

Visual Basic (VB) er et programmeringsmiljø som er tilpasset Microsoft Windows operativsystemer. I VB lages programmer ved å bruke og manipulere eksisterende visuelle komponenter skrevet i programmeringsspråket BASIC (Beginner's All-purpose Symbolic Instruction Code). VB er et eksempel på hvordan visuelle komponenter brukes til å utvikle applikasjoner. Det finnes mange andre verktøy også der komponenter blir satt sammen til et nytt program (Microsoft Visual J++ og Visual Cafe).

Netscape og Quicktime bruker Plug-ins for å få systemet til å støtte nye tjenester. Disse Plug-ins er komponenter som tilfører systemet noe. En slik komponent er laget spesielt for dette systemet, selvom Plug-ins også kan brukes av andre systemer så lenge de bruker grensesnittet til komponenten riktig.

En komponent kan også bestå av komponenter, slik at det blir en slags rekursiv konstruksjon. For eksempel består Quicktime av noen komponenter for at den skal spille av video. Et annet system kan da trenge denne funksjonaliteten og bruker Quicktime som en komponent for å spille av video.

Forskjellen mellom objekter og komponenter

Det er ofte at objekter og komponenter betraktes som det samme. Det kan være riktig i visse tilfeller selvom definisjonen av en komponent er mye bredere enn for et objekt. Objektorientering er ofte kommentert som en godt valg for å utvikle komponenter, og derfor utfyller komponentorientering. Her skisseres de viktigste forskjellen på et objekt og en komponent, slik at forskjellen blir tydeligere.

Egenskaper til objekter:

- Har unik identitet i applikasjonen, som ikke kan endres under objektets levetid.
- Har en tilstand som kan være persistent.
- Innkapsler objektets tilstand og atferd (variabler og metoder).

Egenskaper til komponenter:

- Gjør et arbeid uavhengig av andre komponenter.
- Brukes av en tredjepart, som setter komponenten i arbeid.
- Har en tilstand som kan være persistent.
- Har en atferd som gjerne er beskrevet med regler, mål eller krav, og kan endres under komponentens levetid.
- Innkapsler tilstand og atferd, slik at disse kun er tilgjengelig fra komponentens grensesnitt.
- En komponent kan gjerne bestå av flere objekter.

Forskjellig syn på komponenter

Szyperski ser på en komponent som noe som eksisterer bare en gang i et system ([5], side 30), fordi den skal kunne brukes av andre komponenter til å utføre en bestemt oppgave, uten at den trenger å vite hvordan dette gjøres i

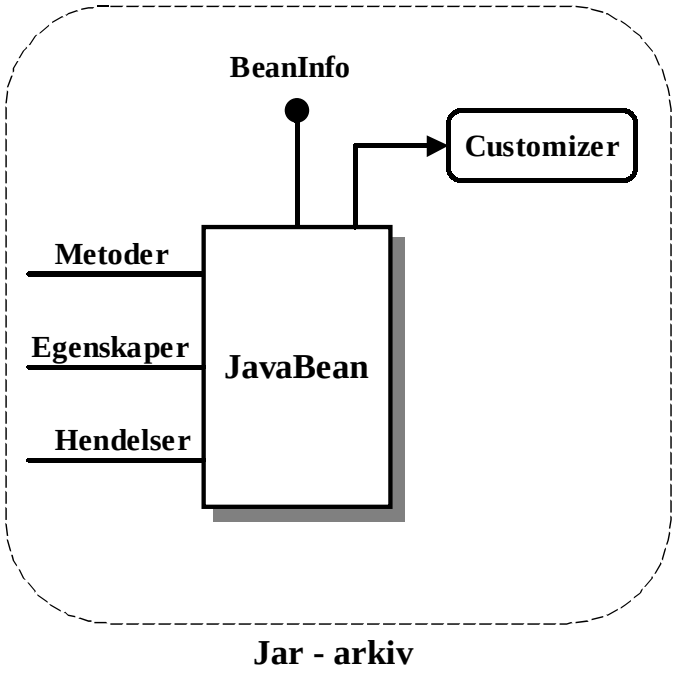
detalj (black box). Hvis det er flere instanser av samme komponent i et system blir dette overflødig siden en komponent ikke kan være i en tilstand.

Komponentmodellen som Szyperski beskriver er i strid med komponentmodellene JavaBeans som Harris baserer sin definisjon på. JavaBeans bruker objektmodellen i Java som basis for å lage komponenter. I klassen java.beans.Beans finnes en metode som lager en instans av en JavaBean (Beans.instantiate()) og setter denne i arbeid i en omgivelse (for eksempel et vindu hvis komponenten er visuell). Dette kan selvsagt gjøres et antall ganger slik at man får flere komponenter av samme type i et system. Dette er i strid med det Szyperski mener er en komponent.

3.1.2 JavaBeans

> *"A JavaBean is a reusable software component that can be manipulated visually in a builder tool."*
>
> - Sun Microsystems, 1997 [13]

Komponentmodellen JavaBeans kapsler inn egenskaper, metoder og hendelser i komponenter (Figur 3-1). JavaBeans er designet for bruk i grafiske utviklingsverktøy, der de brukes i en omgivelse. Det er også muligheter til å lage ikke-visuelle JavaBeans, og til å styre JavaBeans ved skripting. JavaBeans er ”black box” komponenter noe som betyr at komponenten ikke viser hvordan den er implementert. Dette gjør at det ikke er mulig å arve direkte fra en JavaBean.

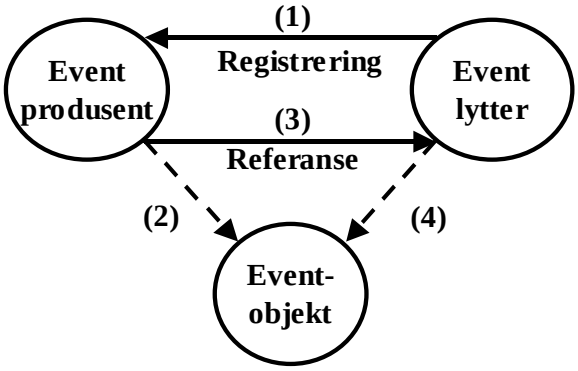


Figur 3-1 Modell av JavaBeans-arkitekuren

Programmet *BeanBox* følger med BDK versjon 1.1(*Bean Development Kit*), og brukes til utvikling med visuelle JavaBeans (knapper, tekstfelt, menyer, osv.). I dette programmet kan komponenter importeres (JavaBeans som er pakket som en jar-fil) og benyttes i en visuell ramme eller omgivelse. BeanBox er et eksempel på en omgivelse som JavaBeans trenger for fungere.

For en utvikler er JavaBeans vanlige klasser som er laget på en spesiell måte (se Figur 3-1). En JavaBean må være persistent (implementere grensesnittet *Serializable*), slik at det er mulig å lagre dens tilstand. Ved å bruke *BeanInfo* grensesnittet får brukeren av komponenten tilgang til informasjon om hva den gjør, ikon og beskrivelse. En JavaBean som arver fra *Customizer* klassen gir informasjon om hvordan komponenten kan brukes under design-time (da komponentene settes sammen) tilgjengelig for brukeren.

JavaBeans har ikke noe IDL (Interface Definition Language) slik som CORBA men det er mulig å lage egne grensesnitt mot komponenten med bruk av *interface* i Java. I tillegg kan man bruke en slags navne konvensjon, som går ut på å ha standard navn på metoder som henter eller setter verdier i feltene (variabler) komponenten disponerer (set<navn>( ) og get<navn>( )). For å teste på en tilstand brukes is<navn>( ) og for event-lyttere brukes add<navn>listener( ) og remove<navn>listener( ). Disse navne konvensjonene brukes i BeanBox for å lettere finne fram til riktige metoder eller variabler (BeanBox bruker Java Reflection API til dette).

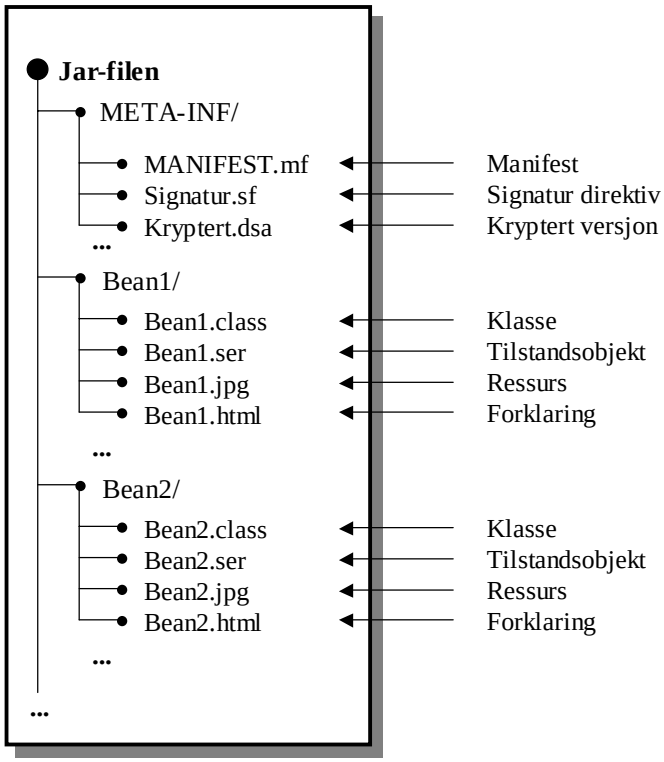


Figur 3-2 Eventhåndtering i Java

Kommunikasjon mellom JavaBeans foregår ved hjelp av eventer (Eventmodellen er skissert i Figur 3-2). JavaBeans som ønsker å motta eventer fra en spesiell komponent, må registrere seg i den aktuelle komponenten (1). Når hendelsen oppstår lages et event objekt (2) og komponenten som var interessert i slike hendelser får en referanse til event objektet (3) ved at en metode blir kalt i komponenten. Eventlytteren kan dermed få tilgang til eventobjektet.

En jar-fil innkapsler JavaBeans som naturlig hører sammen (Figur 3-3). Metainformasjon (META-INF) lagres i manifest-filer og inneholder informasjon om komponentene. Den forteller også hva slags beans som ligger i jar-filen (design eller run-time beans), sikkerhetsmekanismer brukt på komponentene når de ble lagt i jar-filen og navn på andre jar-filer som brukes av denne jar-

filen. Jar-filen kan signeres av den som opprettet filen, slik at brukeren av filen er sikker på hvem som laget den (Trusted signer). Signeringen av jar-filen gjøres ved bruk av *keytool* (JDK1.2 - *javakey* i JDK1.1). Hvis jar-filen er signert finnes et signatur direktiv og en kryptert verson av direktiv filen (Signatur.sf og Kryptert.dsa i figuren). Brukeren vet om en troverdig person har signert filen ved hjelp av sertifikater.



Figur 3-3 Jarfilens oppbyggning

For å lage en ferdig applikasjon, blir ofte jar-filer brukt. En jar-fil inneholder mange klasser/komponenter som naturlig hører sammen og danner en applikasjon. Dette gjør at applikasjonen lettere kan fraktes mellom maskiner i et distribuert nettverk, og dette forenkler gjenbruk.

En komponent som inneholder andre komponenter kalles *Container* i Java. Dette er en vanlig klasse som er laget slik at den tar vare på, og bruker andre komponenter. På denne måten settes komponenter sammen under design-time. Et eksempel kan være et programvindu som inneholder to knapper og et tekstfelt, der vinduet er Containeren som inneholder komponentene.

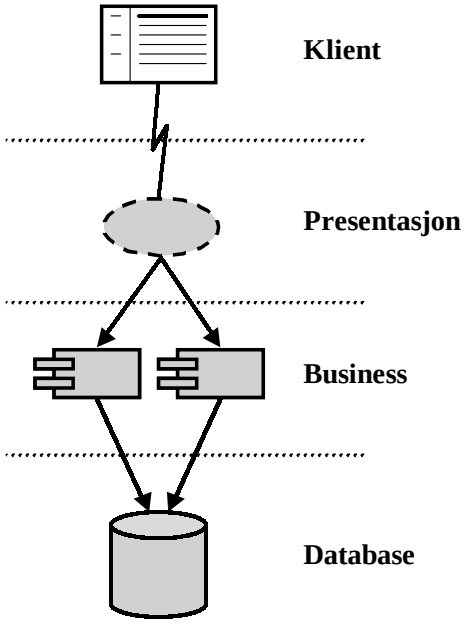
Vurdering av JavaBeans

JavaBeans bruker eventmodellen i Java som er positivt for DMJ arkitekturen, fordi komponentene skal sende og motta eventer. Eventene er derimot ikke distribuerte slik som designet krever. JavaBeans er opprinnelig visuelle

komponenter, og er derfor ikke egnet til DMJ komponentrammeverket. Ikke-visuelle JavaBeans som styres med skripting er en modell som kan benyttes, men er ikke godt egnet for distribusjon og migrering.

3.1.3 Enterprise JavaBeans

Enterprise JavaBeans (EJB) er en komponentmodell beregnet for business logikk på tjenermaskinen laget spesielt for flerlags applikasjoner [44] (Figur 3-4). EJB er en spesifikasjon som følger *Java 2 Enterprise Edition* (J2EE), som gir muligheter til sikkerhet, transaksjoner, mail, serverpages (web), servlets (web), xml, og meldingstjeneste. EJB komponenter tilbyr tjenester til klienter som kan være tunge beregninger som ofte er koblet til en database. En applikasjonsserver utgjør et nødvendig miljø for at EJB komponentene skal fungere. Man sier derfor at EJB komponenter utplasseres eller settes i arbeid (*Deployable*) [8].

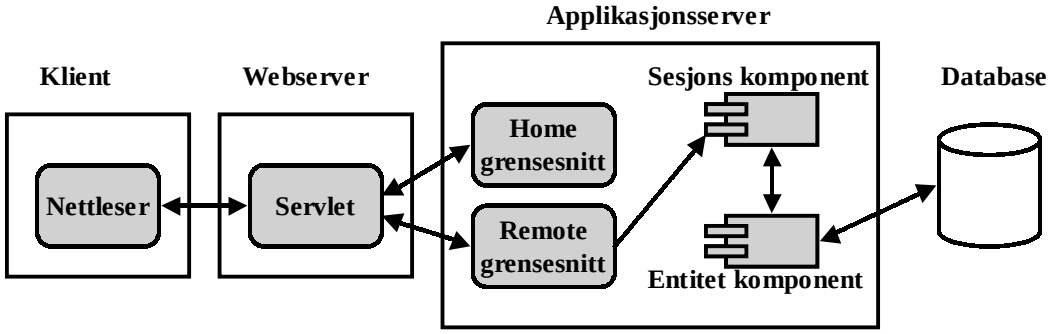


Figur 3-4 Fler-lags applikasjon.

EJB komponenter er ikke laget for å støtte grafisk brukergrensesnitt slik som JavaBeans. Klienten (som gjerne har grafisk brukergrensesnitt) har ikke noe direkte med EJB komponentene å gjøre og kan være laget i andre språk enn Java. Kommunikasjon mellom klienten og serveren er ikke avhengig av Java, men må støttes av applikasjonsserveren.

Det finnes to typer komponenter i EJB-arkitekturen (Figur 3-5). Sesjons-komponenten utfører operasjoner for klienten og lever til operasjonen er ferdig. Sesjons-komponenter som lever over flere transaksjoner eller forespørsler har tilstand, og kalles *"stateful"*. Andre sesjonskomponenter kalles *"stateless"* og dør etter en transaksjon eller forespørsel. Entitetkomponenten oppbevarer persistente data gjerne for et spesielt domene, og har ofte direkte kontakt med en database. *Home* grensesnittet definerer et *Remote* grensesnitt ved første

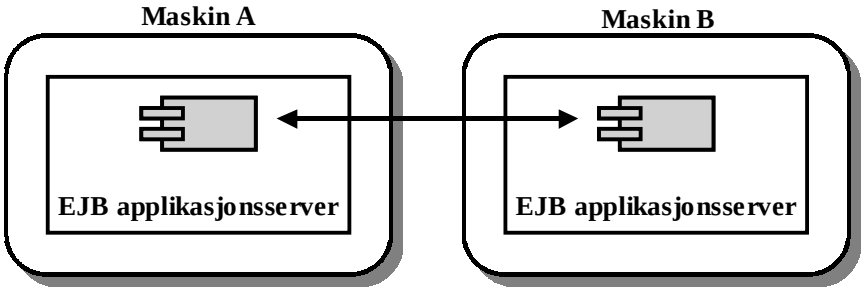
kontakt med serveren, slik at klienten kan gjøre transaksjoner mot sesjonskomponenten [19].



Figur 3-5 Eksempel på bruk av EJB komponenter.

EJB tilbyr en rekke tjenester som gjør utvikling av applikasjoner på tjeneren enklere. Grensesnittet mellom applikasjonsserveren og EJB komponentene er utformet slik at portabilitet mellom forskjellige applikasjonsservere er mulig. I tillegg slipper utvikleren å tenke på sikkerhet, ressurs- og transaksjonshåndtering, fordi dette allerede er implementert i de fleste applikasjonsservere som støtter EJB spesifikasjonen.

For å distribuere EJB komponenter på flere maskiner, må en applikasjons-server startes på hver maskin (Figur 3-6). En applikasjonsserver er en applikasjon som gjør at EJB komponentene fungerer som de skal (komponentomgivelse). Det finnes ingen funksjonalitet i EJB som støtter kommunikasjon mellom komponenter på forskjellige applikasjonsservere direkte.



Figur 3-6 EBJ komponenter som kommuniserer over maskingrenser.

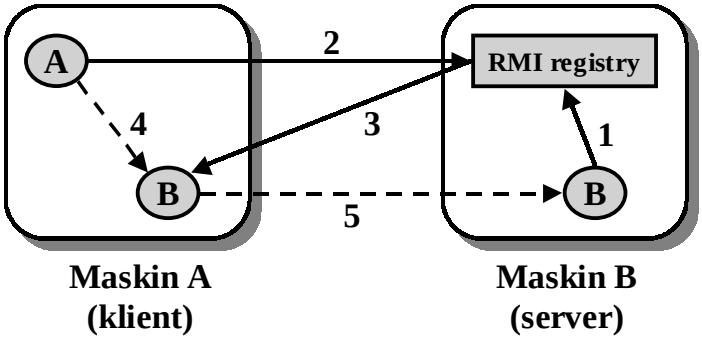
Java Remote Method Invocation (RMI)

Java RMI bygger på RPC (Remote Procedure Call) [1] som benyttes for å kalle prosedyrer i andre prosesser (interprosess kommunikasjon), som ofte kjører på andre maskiner. Java RMI tar RPC et skritt videre ved å tillate objekter å kalle metoder i andre objekter som kan befinne seg på andre maskiner (interobjekt kommunikasjon). Modellen bruker serialisering av objekter som

gjør at objekter kan sendes som parameter til metodekallet. EJB bruker Java RMI til distribusjon av komponentene og kommunikasjon mellom dem. Arkitekturen til Java RMI blir her vurdert og noen av de viktigste egenskapene med kommunikasjonsmodellen blir beskrevet.

Java RMI har ingen IDL (Interface Definition Language) slik som CORBA og DCOM, men har i steden en egen kompilator (*rmic*) som benyttes på de klassene som skal være distribuerte (*remote objects*). Dette gjøres etter at klassene er kompilert med Java kompilatoren slik at *stubs* og *skeletons* dannes. Disse *proxy*-klassene danner grunnlaget for kommunikasjonen til metoder i distribuerte objekter under run-time.

I Java RMI må de distribuert objektene registreres i RMI registry (1) (Se Figur 3-7 for nummerert beskrivelse). Dette er en egen applikasjon som må startes opp på den maskinen objektene skal være tilgjengelig fra (RMI registry lytter på en valgfri eller standard port på maskinen). Etter det kan andre objekter henvende seg til RMI registry på den aktuelle maskinen og hente en referanse til objektet (2) ved å oppgi et navn med URL-aktig notasjon. Når dette gjøres dannes det også en stub for det distribuerte objektet på klientmaskinen (3). Stub objektet er en kopi av det originale objektet, og vil gjøre kall for å få metoder utført (4 og 5).



Figur 3-7 Java RMI modellen

Det er viktig at objektene som skal være tilgjengelige ved Java RMI er serialiserbare, eller at det er mulig å lagre objektets tilstand. Dette er viktig i Java RMI fordi objekter skal kunne sendes over nettverket og representeres i en annen prosess. Serialisering i Java gjøres ved å merke objektet eller grensesnittet med *"serializable"*.

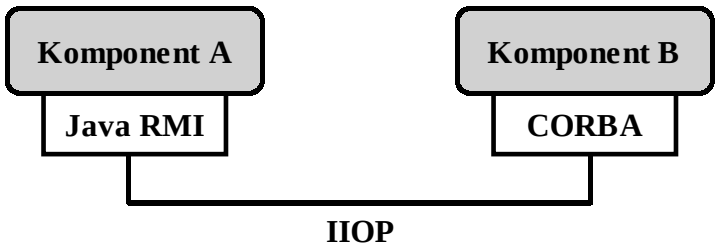
```
interface IName implements Serializable
```

Java RMI over IIOP

SUN og OMG har gått sammen om å lage en mapping mellom Java RMI og CORBA, og har kommet fram til en løsning der begge modellene er fullt kompatible med hverandre [24]. Mappingen beskriver en ny stubkompilator

(tilsvarende *rmic* for Java RMI) som lager RMI stubs og ties (representasjon av den implementerte klassen) som kommuniserer over IIOP.

RMI over IIOP gjør at utviklere ikke lenger trenger å velge mellom RMI og CORBA, men kan utnytte begge modellenes fordeler. Det gjør også at allerede Java RMI implementerte komponenter kan benyttes i et CORBA integrert miljø. Det finnes Java applikasjonsservere som har god støtte for infrastruktur, mens CORBA er sterk innen distribusjon. En sammenkobling mellom de to standardene er en god kombinasjon for utviklerne [29].



Figur 3-8 RMI over IIOP

Vurdering av EJB

Enterprise JavaBeans har mange interessante egenskaper ved seg som det er behov for i DMJ rammeverket. Kommunikasjon mellom komponentene er her støttet med Java RMI og RMI over IIOP. EJB støtter distribusjon ved at komponentene kan plasseres ut på flere maskiner og kommunisere med hverandre. Dette krever imidlertid at en applikasjonsserver kjøres på alle maskinene, som ofte er en tung og ressurskrevende applikasjon. Det er ikke direkte støtte for kommunikasjon på tvers av applikasjonsservere fra forskjellige leverandører (for eksempel BEA WebLogic[1] og IBM WebSphere[2]). Støtte for migrering av komponenter mellom maskiner, er avhengig av applikasjonsserveren. Ingen av applikasjonsserverne som har blitt vurdert har støtte for migrering av EJB komponenter over maskingrenser.

3.1.4 Microsoft COM med etterfølgere

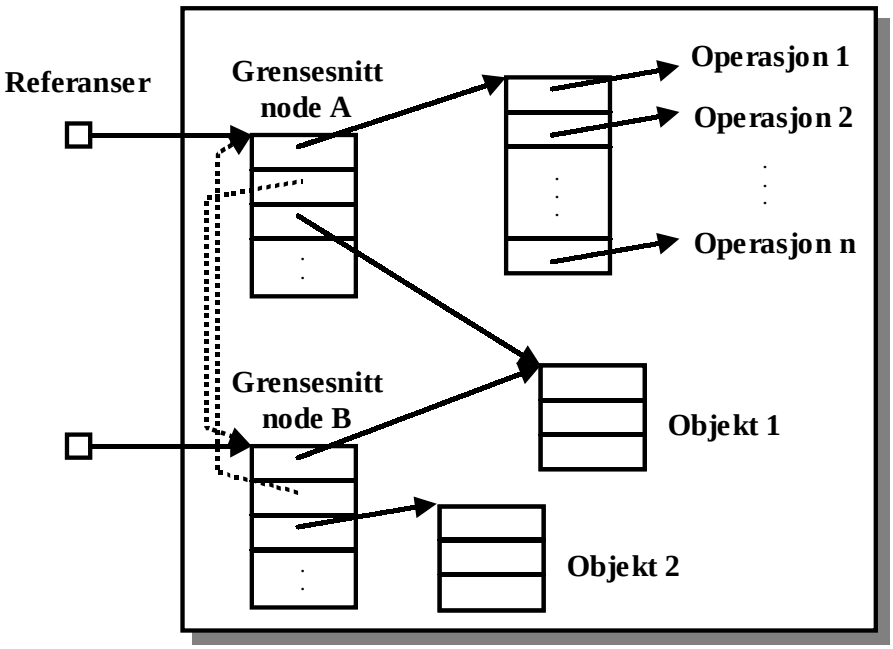
Microsoft COM (Component Object Model) er selve ryggraden i Microsoft Windows, og er grunnlaget for alle komponenter som er laget for den velkjente plattformen. COM består av både en spesifikasjon og en implementasjon, som tilbyr et rammeverk for integrering av komponenter. Modellen støtter at komponenter kommuniserer som COM objekter både innen og mellom datamaskinenes grenser, uavhengig av programmeringsspråk og

[1] http://www.bea.com/products/weblogic/server/index.shtml (mai 2001)
[2] http://www-4.ibm.com/software/webservers/appserv/ (mai 2001)

operativsystem [20]. COM objekter defineres i denne beskrivelsen som funksjoner og deres tilstand, og må lages i henhold til COM standarden. Grensesnittene som COM objektene støtter er gjerne laget på forhånd, men det går også an å lage egne grensesnitt.

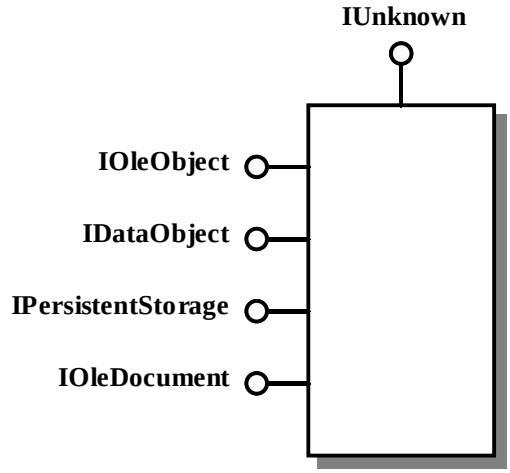
Arkitektur

COM er en binær standard, som vil si at alle komponenter laget i COM må ha en bestemt struktur i eksekverbar tilstand (applikasjon eller program). Modellen spesifiserer ikke hva en komponent er, og det er ikke nødvendig å bruke et objektorientert programmeringsspråk for å lage en COM komponent. Det som kreves er at visse grensesnitt er implementert av komponenten. Grensesnitt på binærnivå er representert med en peker/referanse (utenfor komponenten) til komponentens grensesnitt-node. En komponent kan ha flere slike grensesnitt-noder. Det eneste som befinner seg i grensesnitt-noden, er pekere videre til tabeller av prosedyre variabler (funksjons pekere), objekter og til andre grensesnitt-noder. Figur 3-9 viser en skisse av et COM objekt på binærnivå.



Figur 3-9 Et eksempel på et COM objekt

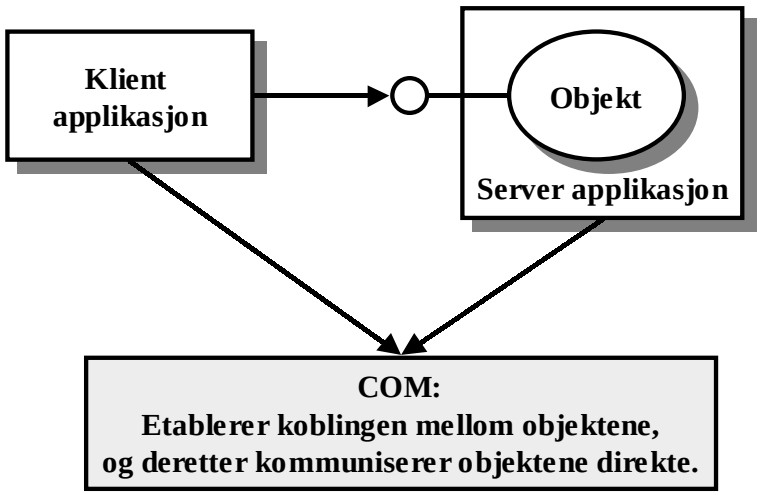
Alle COM objekter må ha grensesnittet *IUnknown* implementert, som er inngangsporten til komponenten. Operasjonen *QueryInterface* som også skal implementeres i alle COM objekter, gjør at komponentens grensesnitt kan brukes. Figur 3-10 viser hvordan et COM objekt vanligvis blir skissert (Eksempelet er et ActiveX dokument objekt). Det er mulig å lage grensesnitt selv, ved å arve fra *IUnknown* grensesnittet. COM objekter og grensesnitt blir definert ved Microsoft Interface Definition Language (MIDL).



Figur 3-10 Eksempel på skisse av et COM objekt

Linking mellom COM objekter

COM skiller klart mellom objekter som instanser av objekt klasser, og grensesnitt som en samling av abstrakte metoder [50]. Objektene og grensesnittene kjennetegnes med en global referanse (GUID) som er unik. GUID referansen til COM objekter kalles OID, for grensesnitt IID, og for objekt klasser CLSID. En *objekt server* er en instans av et dynamisk link bibliotek (DLL), som kan opprette og oppbevare objekter av en eller flere klasser. En *klient* er i COM en prosess som bruker metodene i et objekt. Siden referansen til objektklassen eller grensesnittet skal være global, kan ikke implementasjonen av klassen eller grensesnittet endres under samme GUID.



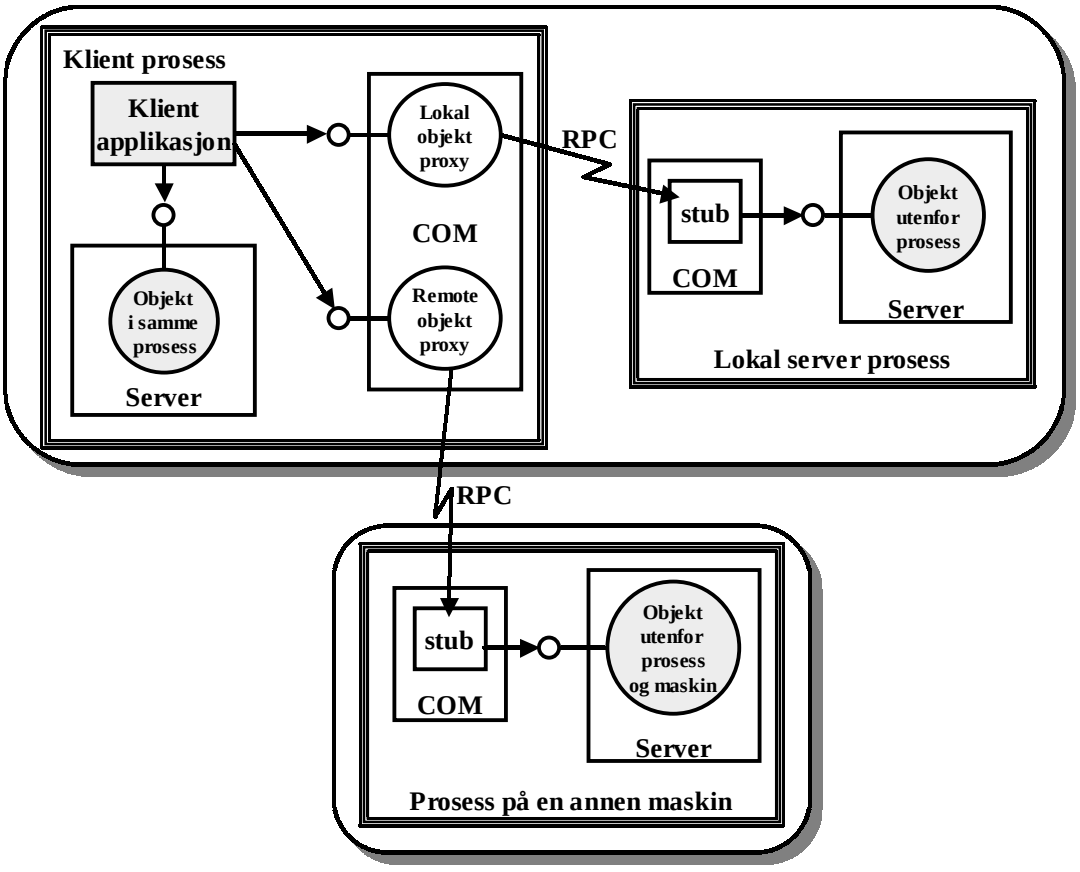
Figur 3-11 Linking mellom klient og objekt

COM lar operativsystemet være en slags sentral katalog for objekter, som styrer oppretting og sletting av objekter. Operativsystemet styrer også all kommunikasjon mellom objektene, uavhengig om de er i samme prosess eller ikke.

DCOM

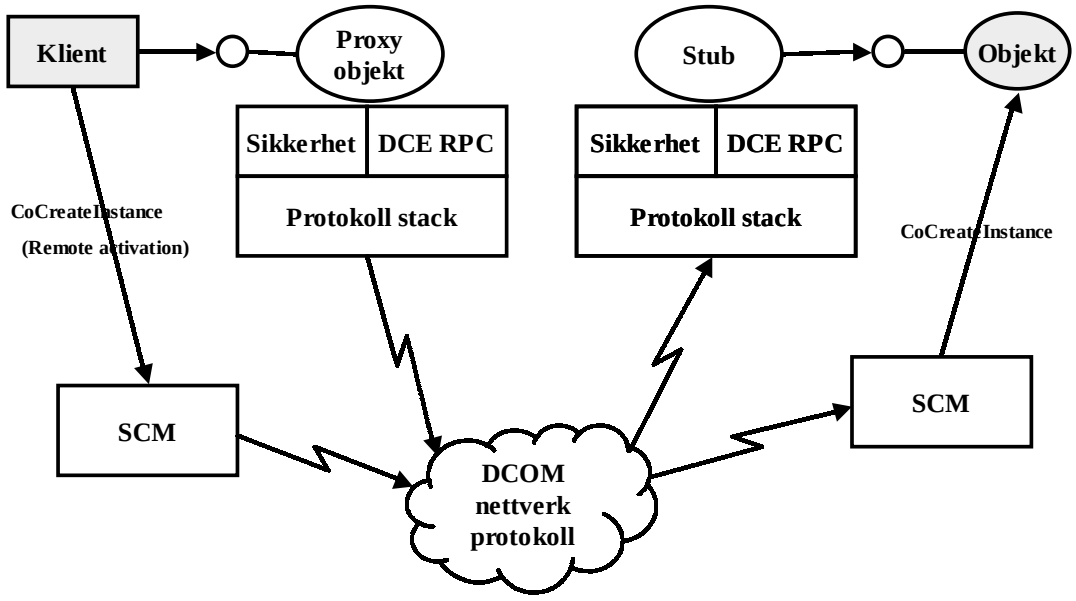
Microsoft DCOM utvider COM og bruker DCE RPC modell [27] for kommunikasjon mellom komponentene. Noe av det som omtales her gjelder også for COM, men hører naturlig med i en presentasjon av DCOM siden det innebærer kommunikasjon mellom distribuerte komponenter.

Det distribuerte perspektivet som COM tilbyr, dekker behovet for kommunikasjon mellom COM objekter. COM støtter distribuerte objekter, som vil si at man kan fordele en applikasjon på flere maskiner ved utføre komponentobjektene på forskjellige maskiner [20]. Dette gjøres utenfor selve objektene men innenfor ansvarsområdet til COM, ved at klienten kun refererer til en grensesnitt peker. Objektet refererer til et grensesnitt på samme måte for objekter som er i samme prosess, i en annen prosess eller på en annen maskin.



Figur 3-12 Lokasjons transparens i COM

DCOM tillater klienter å opprette COM objekter på andre maskiner. Klienten oppretter et objekt på vanlig måte (kaller på CoCreateInstace), og DCOM sørger for at objektet blir opprettet på den rette maskinen, basert på CLSID og servernavnet. Klienten kontakter den lokale tjenestekontroll enheten (SCM – Service Control Manager), som er en del av COM biblioteket. SCM på klientsiden tar kontakt med SCM på den aktuelle maskinen, og COM objektet opprettes slik at klienten kan bruke objektet på den andre maskinen via proxy objektet på klient maskinen.



Figur 3-13 DCOM arkitektur [18]

Etterfølgere

Den første versjonen av operativsystemet som brukte COM, var Windows 95. Etter det har Windows NT kommet med en ny modell som har utvidet COM, som heter DCOM (Distributed COM). Den nyeste versjonen av Windows er Windows 2000, og her heter komponent modellen COM+. Denne modellen inneholder langt mer, og erstatter MTS (Microsoft Transaction Server), COM og DCOM.

Microsoft har satt sammen mange applikasjoner og modeller, og kaller det Windows DNA (Distributed interNet application Architecture). Hovedingrediensene her er Windows 2000 og COM+, samt Microsoft Message Queue (MSMQ), Internet Information Server (IIS) og mange flere applikasjoner. Windows DNA tilbyr en arkitektur for distribuerte, ”enterprise-ready” web applikasjoner. Den siste arkitekturen Microsoft har presentert innen komponentutvikling, er .NET, som tar over mye av funksjonaliteten som Windows DNA tilbyr. Microsoft har laget .NET Enterprise Server som gjør det enklere å utvikle, sette i arbeid og styre skalerbare, integrerte web-baserte applikasjoner. Noen av teknologiene som inkluderes i .NET arkitekturen er

Microsoft SQL Server, Application Senter, Exchange Server og Mobile Information Server.

COM på andre plattformer

Microsoft COM er laget for Windows operativsystemer. Det har kommet konkurrerende komponentmodeller som tilbyr plattform uavhengighet og derfor vil også Microsoft tilby dette. Software AG er det eneste firmaet som tilbyr en plattform for COM/DCOM objekter på andre plattformer enn Windows. Applikasjonen heter EntireX [39], og den fungerer under mange UNIX plattformer, inkludert Linux, Sun Solaris og IBM OS/390.

Vurdering av COM

Det er kjent for de fleste som utvikler programvare at dersom Microsoft vurderes i arkitekturen, så fungerer ikke applikasjonen optimalt før alt er Microsoft produkter inklusiv operativsystemet. Beskrivelsen av COM på UNIX [4] bruker en kommersiell applikasjon som tilbyr et run-time miljø for DCOM applikasjoner på UNIX systemer. Run-time miljøet i EntireX [39] støtter ikke utvikling av grafisk brukergrensesnitt med COM objekter, men har full støtte for sammenkopling av COM komponenter. Det finnes lite forskning innen COM på andre operativsystemer enn Windows. Derfor hadde det vært interessant å se på en løsning i COM, der både Windows og andre plattformer ble brukt. Det er ingen stor etterspørsel etter å bruke COM komponenter på UNIX og det er usikkert hvordan COM på UNIX fungerer for innholdsanalyse av mediastrømmer siden grafisk brukergrensesnitt ikke er støttet. Med dette som utgangspunkt er ikke COM aktuelt for DMJ komponentrammeverket på grunn av at operativsystemet er veldig kritisk for ressurskrevende DMJ applikasjoner. Operativsystemet bør kunne velges noe mer fritt enn det COM tilbyr.

COM tilfredsstiller kravene for designet av komponentene på de fleste punktene. Med den nye arkitekturen .NET støttes mange av kravene til DMJ arkitekturen med mange interessante tilleggsapplikasjoner. Distribusjon og kommunikasjon blir støttet gjennom den mye brukte DCE RPC modellen. Migrering og lastbalansering blir støttet i Microsoft sin applikasjonsserver Application Center og transaksjonstjenester i Transaction Server, som begge er tett tilknyttet .NET arkitekturen.

3.1.5 Oppsummering av komponentmodeller

Komponenter har en bred definisjon og kan være mye forskjellig. I denne oppgaven betraktes komponenter som programenheter som har en avgrenset oppgave og innkapsler egenskaper slik at funksjonaliteten til komponenten kun er tilgjengelig via veldefinerte grensesnitt. Fordelen med komponent-

orientering er at komponentene er kan benyttes i andre sammenhenger og deler opp systemet i deler om er enklere å vedlikeholde og endre. En applikasjon som er utviklet med komponentorientering består av adskilte enheter og ikke en eksekverbar fil. I mange tilfeller kan et rammeverk samle komponenter som hører sammen i et komponentrammeverk, slik at utviklere lettere kan benytte seg av disse.

De mest brukte komponentstandarder som er akseptert av utviklingsmiljøene, er Microsoft Component Object Model (COM) med alle dens etterfølgere, JavaBeans og Enterprise JavaBeans (EJB). Slik situasjonen står i dag omtales gjerne COM og EJB side om side når business løsninger skal utvikles. JavaBeans er en komponentmodell som er tilegnet for visuelle komponenter, som benyttes mest til å lage grafiske brukergrensesnitt. JavaBeans kan også styres ved skripting. I tillegg til disse tre standardene, har OMG kommet med en komponentstandard som blir kalt CORBA Components [22]. OMG er en standardiserings organisasjon, som ikke direkte utvikler applikasjoner eller rammeverk. Spesifikasjonen av CORBA Components er ikke komplett og derfor blir ikke standarden støttet av programvare produsentene. Det er sagt at CORBA Components skal være en del av den kommende CORBA 3.0.

Et viktig tema i beskrivelsen av en komponent er hvordan den kommuniserer med andre komponenter. Derfor blir det også viktig å beskrive komponentene i prototypen etter en kommunikasjonsmodell er valgt. Neste kapittel tar for seg modeller for kommunikasjon mellom analysekomponentene i prototypen.

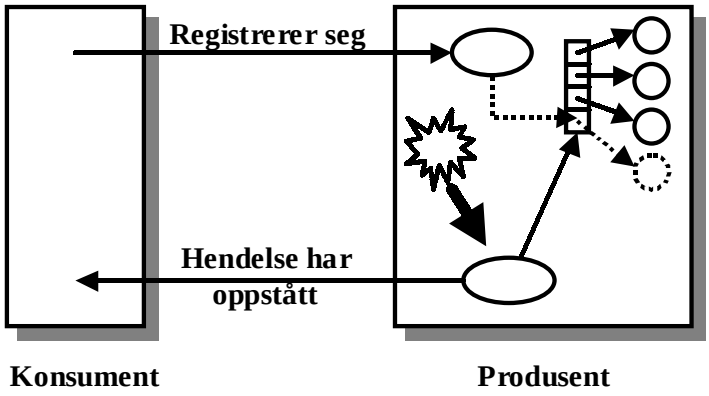
3.2 Kommunikasjonsmodell for eventbrokeren

Komponentmodellene EJB og COM definerer selv modeller for kommunikasjon mellom distribuerte komponenter. Her beskrives hvordan analyse-komponentene skal støtte event-basert kommunikasjon i eventbroker komponenten. Først diskuteres noen generelle modeller for eventhåndtering. Deretter presenteres den velkjente kommunikasjonsmodellen CORBA hvor *Notification Service* blir fremhevet som en modell for eventbrokeren. Java Messaging Service (JMS) er kjent fra Java 2 Enterprise Edition (J2EE) og støtter event-basert kommunikasjon mellom EJB komponenter. Til slutt presenteres Message Bus (MBus) som støtter meldingsutveksling med bruk av IP multicast. Alle modellene blir vurdert opp mot kravene som stilles til DMJ arkitekturen som skal støttes i prototypen.

3.2.1 Generelt om eventhåndtering

Eventhåndtering blir ofte brukt i sammenhenger hvor et objekt eller komponent vil ha beskjed når en spesiell hendelsen inntreffer. Dette kan være alt fra at brukeren trykker på en knapp i det grafiske brukergrensesnittet til at en børskurs på Wall Street synker under en viss grense. Hendelsen kan både være lokal og distribuert. Det kan være ingen, en eller flere som er interessert i hendelsen. Måten man som regel løser dette problemet på er å bruke en eventmodell. En typisk eventmodell tar utgangspunkt i en produsent og en

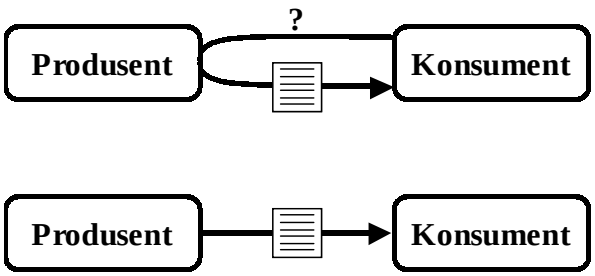
konsument og bygger funksjonalitet rundt disse for å støtte denne typen kommunikasjon. Konsumenten vil typisk ha beskjed hvis en hendelse oppstår hos produsenten, og registrerer seg derfor som interessert hos produsenten. Når en hendelse oppstår, sender produsenten hendelsen til alle som er registrert hos produsenten (Figur 3-14).



Figur 3-14 Standard eventmodell

Pull versus Push

Eventer som blir tilgjengelig hos produsenten er ikke alltid interessant for konsumenten med en gang. Pull modellen går i hovedsak ut på å be om informasjon når man trenger den. Push modellen benyttes når informasjon sendes til konsumenten uansett (Figur 3-15). Konsumenten må skille ut hva man er interessert i selv når push modellen benyttes. En analogi for å beskrive forskjellen mellom push og pull kan være å låne en bok i biblioteket og få post levert hjem. Postverket ville hatt problemer med å levere alle bøkene i biblioteket til alle menneskene i Norge for å spørre om man ville låne bøkene. I motsatt tilfelle ville det være en ulempe å stadig måtte spørre postverket om det har kommet noe post. Dette viser at push og pull bør benyttes i forskjellige sammenhenger.



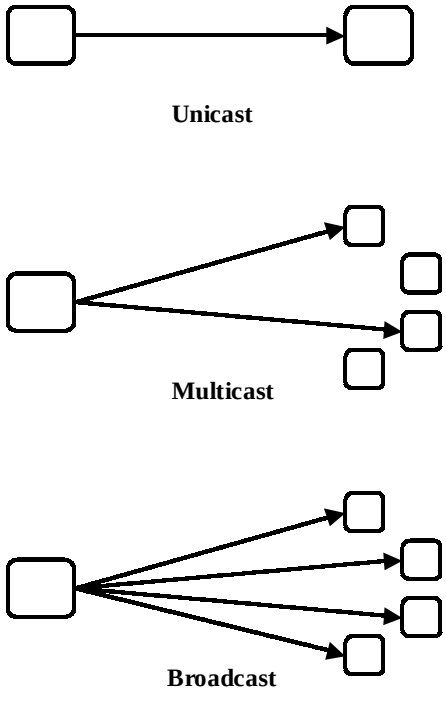
Figur 3-15 Pull (øverst) og push modellen

Ved utforming av distribuerte systemer er det viktig å begrense informasjonsflyten til et minimum. Dersom man tar sikte på at alle er interessert i hendelsene er det som regel best å benytte push modellen. På denne måten unngår man polling av produsenten, som er nødvendig i pull modellen. Dette gjør også at forsinkelsen blir liten før konsumenten får hendelsen.

Det kan også være et spørsmål om hvor komplekse sender og mottaker skal være. Push modellen krever nesten ingen logikk for at produsenten skal sende ut data til alle som vil ha. Det kreves adskillig mer logikk når konsumenten skal spørre produsenten hver gang den er interessert i en spesiell hendelse. I enkleste tilfelle må produsenten legge all informasjonen i et buffer som konsumenten kan hente informasjon fra. Utfordringene her ligger i å definere størrelse på bufferet (historie egenskaper), og det at produsenten må ha egenskapene til en server. Hvis det er mange som vil ha informasjonen som produsenten har, kan dette bety at produsenten må ha muligheten for å håndtere flere forespørsler på samme tid (multi-threaded server) for å øke ytelsen.

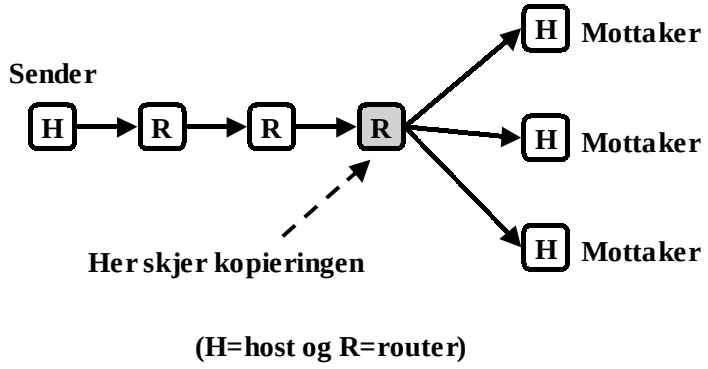
IP Multicast

Eventbasert kommunikasjon deles gjerne opp *unicast, multicast* og *broadcast* [10]. Modellene gjør forskjell på hvor mange som skal nås med budskapet man vil sende (Figur 3-16). Unicast betyr at man sender til en enkelt mottaker, som gjerne kalles ”klient-tjener” eller ”peer-to-peer” kommunikasjon. Multicast betyr at man sender samme melding til flere maskiner. Med broadcast sendes meldingene til alle som kan mottak meldingen.



Figur 3-16 Forskjellige måter å sende data i nettverket

IP Multicast [35] [15] brukes for å nå en gruppe av maskinene i nettverket. Dette gjøres ved å sende data til en spesiell adressegruppe, som er definert av IP (Internet Protocol). I IP versjon 4 er adresser mellom 224.0.0.0 til 224.255.255.255 reservert for multicast. For å motta data fra en multicast adresse, gjøres det samme som for en hvilken som helst IP adresse, hvor man lytter på en port på maskinen. Det som er det geniale med IP multicast er at data ikke flyter gjennom nettet hvis det ikke er noen som er interessert i å motta dataene. Dette gjør at belastningen på nettverket reduseres til et minimum. I tillegg kopieres pakkene i nettverket så sent som mulig dersom det er flere som abonnerer på den samme adressen (Figur 3-17).



Figur 3-17 IP multicast kopierer datapakkene så seint som mulig

3.2.2 CORBA

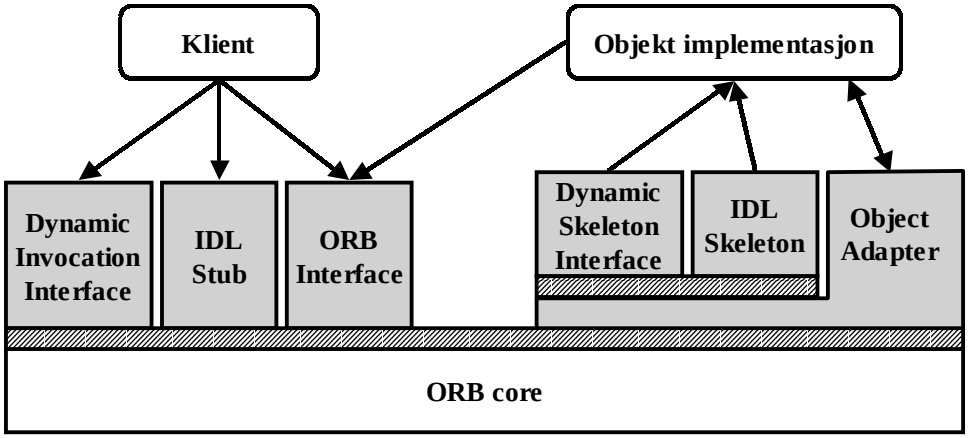
Common Object Request Broker Architecture (CORBA) er laget av Object Management Group (OMG), og er en standard for kommunikasjon mellom distribuert objekter. Grunnleggende for CORBA er Object Request Broker (ORB) som styrer kommunikasjonen som forespørsler (request) og responser mellom de distribuerte objektene. OMG definerer mange tjenester i CORBA som legges til en standard ORB som blir nærmere spesifisert.

COBRA Interface Definition Language (IDL)

Mye av arbeidet med å lage distribuerte systemer går gjerne med til å lage gode grensesnitt mellom systemets deler. OMG har utviklet et generelt språk for dette som heter CORBA IDL. Språket definerer kun hvordan metoder og attributter skal se ut. Dersom man ønsker å benytte CORBA ved utvikling av distribuerte applikasjoner, må man velge et programmeringsspråk som det finnes en IDL kompilator for (Java, C++, osv). En IDL kompilator oversetter IDL kode til programkode som senere kan kompileres.

Arkitektur

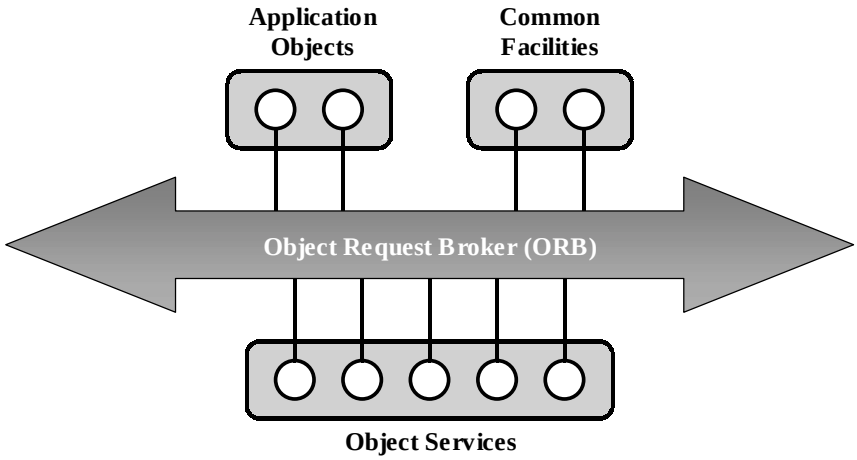
CORBA bygges over en ORB (Object Request Broker) som er vist i Figur 3-18. Her skjer kommunikasjonen mellom klienten og objekt implementasjonen skjult over maskin- og prosessgrenser. CORBA-objektene kommuniserer over en slik ORB ([42] side 2-3).



Figur 3-18 Grensesnittet mot ORB (Object Request Broker)

ORB arkitekturen har i bunnen en ORB core, som representerer den faktiske kommunikasjonen mellom objektene på nederste nivå (Figur 3-18). Laget mellom grensesnittene (Dynamic Invocation Interface, ORB Interface og Dynamic Skeleton Interface) og ORB core er ORB avhengig (skravert område) og vil si at det er skjult for utvikleren. Kommunikasjonen mellom klienten og serveren foregår i hovedsak mellom IDL stub og IDL skeleton. Disse blir laget etter kompilering av IDL-filer (omtaler IDL i neste avsnitt) som definerer hvordan kommunikasjonen mellom klient og server skal foregå. Disse må være laget før selve programmet kompileres, og blir derfor statiske. I klienten brukes så den genererte IDL-stub og kan kommunisere over ORB til serveren. Serveren implementerer den genererte IDL-skeleton og kan motta kall fra klienten over ORB. Oppgaven til objekt adapteren er å motta og dekode (unmarshall) kall fra klienten. Objekt adapteren vet ikke hvilke metoder som finnes i serveren, og må derfor sende kallet videre til IDL-skeleton som kaller på den riktige metoden i objektet.

CORBA utvider ORB arkitekturen ved å legge til tjenester (*services*) og hjelpemidler (facilities). Figur 3-19 viser dette. CORBA Services betyr tjenester som er direkte tilgjengelige for de distribuerte objektene, og Facilities er kun en beskrivelse av hvordan supplerende tjenester som ikke er direkte tilgjengelig før noen programvare produsenter har benyttet seg av dem.



Figur 3-19 CORBA (Common Object Request Broker Architecture)

Internet Inter-ORB Protocol (IIOP)

IIOP blir brukt for å kommunisere mellom ORB implementasjoner. Dette gjør at et CORBA-basert program kan samarbeid med et hvilket som helst annet CORBA-basert program. Programmene kan være på forskjellige maskiner i nettverket. CORBA støtter mange varianter av nettverk, maskintyper, operativ system og programmeringsspråk.

CORBA Services

CORBA Services er samlinger av tjenester på systemnivå som er pakket inn med IDL-spesifiserte grensesnitt. Disse tjenestene utvider direkte funksjonaliteten til ORB, og brukes til å lage, navngi og introdusere objekter i programmeringsomgivelsen. I CORBA 2.0 er det 15 slike tjenester som er ferdige til bruk. Life Cycle Service, Naming Service, Event Service, Transaction Service og Trader Service er noen av tjenestene som finnes i CORBA Services [23]. Her vil det bli lagt spesielt vekt på CORBA Notification Service, som er en utvidelse av Event Service, siden dette er det mest interessante for eventbrokeren.

CORBA Notification Service [25] beskriver en eventtjeneste basert på sending og mottak av eventer gjennom kanaler. Eventene kan være typet, utypet eller bestå av en veldefinert datastruktur, der Event Service kun støtter typede og utypede eventer. Det er også mulighet til filtrering av eventer, noe som gjøres i kanalen slik at ikke eventer sendes til mottakerne uten av det er nødvendig. Modellen støtter både push og pull slik som Event Service også gjør, men i Notification Service er funksjonaliteten flyttet ut i kanalen, slik at kommunikasjonen blir begrenset. En annen utvidelse i forhold til Event Service er at Notification Service støtter Quality of Service (QoS) i levering av eventer. Her implementeres standard grensesnitt for å sette QoS parametere

som pålitelighet, prioritet, tidsperspektiv på levering, belastningsgrenser på antall eventer pr konsument, megling av QoS parametere og konfliktløsning.

Figur 3-20 Eventmodellen i CORBA Notification Service

Vurdering av CORBA

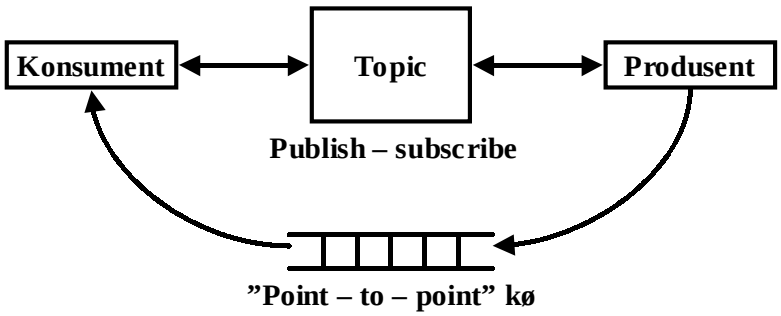
CORBA er beregnet for kommunikasjon i distribuerte systemer, hvor mange bedrifter er med på å utforme spesifikasjonen. CORBA er spesifisert slik at mange programmeringsspråk, operativsystemer og nettverk skal kunne brukes, og det gjør dette til en generell og sterk løsning. IIOP sørger for at kommunikasjon mellom forskjellige ORB implementasjoner er mulig, og gjør at EJB har en løsning som bruker RMI over IIOP.

CORBA har en rekke tjenester som ligger ferdig til bruk. Blant disse er det Event Service og Notification Service som er interessante for DMJ rammeverket. Notification Service har en meget velegnet løsning for eventbrokeren i DMJ arkitekturen som har støtte både for push og pull. Det er også støtte for filtrering av hendelser slik at bruk av nettet blir minimalisert. Hendelsene som skal sendes til et konsumentobjektet på en annen maskin, sendes via event kanalen med ORB'en som produsentobjektet implementerer. ORB'en i konsumenten mottar hendelsen over IIOP. Event kanalen tar vare på alle som er interesserte i en type event, og sender til hver mottaker uten å bruke fordelene som ligger i IP multicast.

3.2.3 Java Messaging Service (JMS)

JMS versjon 1.0.2 [41] er en eventtjeneste som kommer sammen med Java 2 Enterprise Edition (J2EE), og er i første rekke ment for bruk innen forretningsapplikasjoner. Modellen er relativt enkel i forhold til CORBA Notification Service, men definerer også en slags kanal som meldingene sendes via.

I JMS kan man enten bruke point-to-point (PTP) meldingskø eller publish-subscribe (Pub/Sub) meldingsutveksling (Figur 3-21). PTP går ut på at meldingsprodusentene sender meldingene til en kø fremfor for å sende de til en bestemt konsument. Pub/Sub modellen bruker en mer vanlig eventmodell, der meldinger kan sendes til de som registreres til å lytte på dette spesielle eventet. I modellen Pub/Sub heter kanalen mellom produsent og konsument *Topic*, og er en beskrivelse av et emne som en melding tilhører. Meldingene i JMS kan være av forskjellige typer som inkluderer objekter, XML og datastrømmer.



Figur 3-21 JMS eventmodell

QoS fasilitetene i JMS består av at meldingene enten kan leveres etter beste evne (*best-effort*) eller at garantert en og bare en melding kommer fram til mottakeren. Andre QoS parametere som tidsbruk, prioritet, megling og konfliktløsning er ikke implementert i JMS. JMS har heller ikke støtte for lastbalansering, feiltoleranse, administrasjon eller sikkerhet. Det finnes imidlertid programvare produsenter som utvider JMS spesifikasjonen med blant annet flere QoS parametere og høyere sikkerhet.

Vurdering av JMS

Java Message Service er en del av J2EE, og er tilpasset meldingsutveksling i business til business applikasjoner der EJB brukes. JMS har ikke støtte for noe slags filtrering av hendelser som sendes mellom produsent og konsument. Dette gjør at det brukes mye unødvendig båndbredde i en distribuert eventbroker som i DMJ. Push modellen er den eneste som er støttet, noe som vil si at eventer blir levert uansett om konsumenten vil ha den eller ikke. JMS bruker heller ikke multicast i sending av meldinger i nettverket, men Sun påstår at modellen skalerer bra.

3.2.4 Objectspace Voyager

Objectspace Voyager Universal ORB [12] er et rammeverk utviklet i Java som inkluderer en standard-nøytral ORB. Derfor er det mulig å lage distribuerte systemer som kommuniserer uavhengig av standarder (CORBA, RMI, DCOM). Voyager er et CORBA integrert miljø hvor det blant annet er mulig å lage mobile agenter. Voyager skjuler alt som har med kommunikasjonen mellom objekter å gjøre. Dette vil si at *stubs*, *skeletons* og *helpers* lages automatisk og brukes uten at utvikleren behøver å tenke på det. Professional utgaven av Voyager har blant annet muligheter for lastbalansering og avansert meldingsutveksling.

Voyager ORB er kompatibelt med CORBA, RMI og DCOM, som vil si at en CORBA/RMI/DCOM server og en Voyager klient kan snakke sammen, og

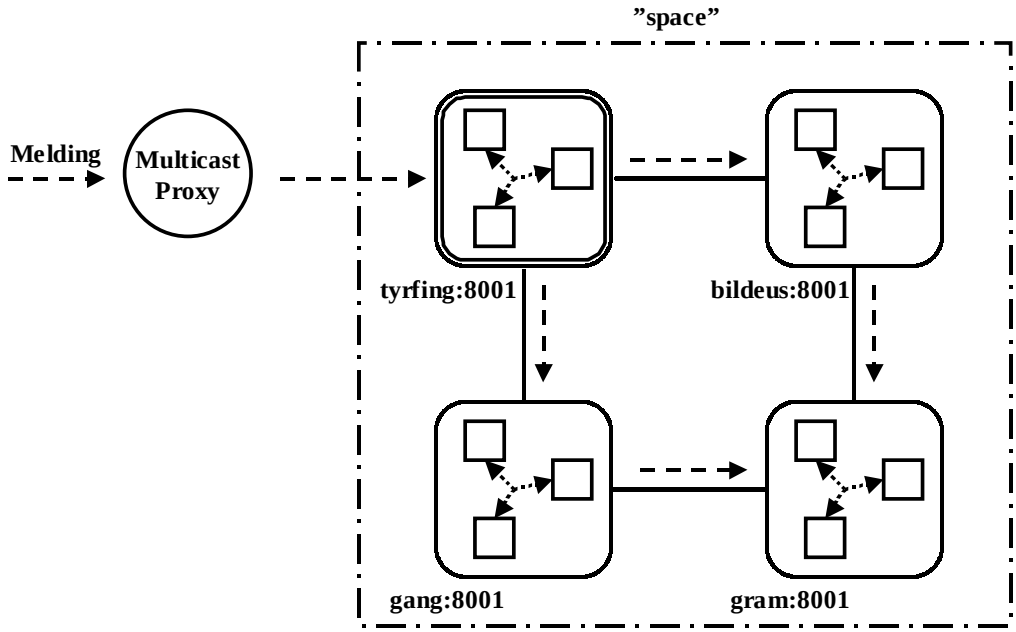
omvendt. På denne måten er Voyager et bindeledd mellom de nevnte teknologiene, som gjør at objekter som bruker en av disse tre standardene kan snakke sammen.

Objectspace har også utviklet en applikasjonsserver som støtter Enterprise JavaBeans. Voyager Application Server 4.0 bygger på Voyager Universal ORB, og har innebygget en egen administrasjons applikasjon for å administrere EJB komponentene. Det følger også med et eget utviklingsverktøy, Voyager EJB studio. Dette forenkler utvikling av EJB komponenter i Voyager. Applikasjonsserveren har støtte for sikkerhet (SSL, SOCKS og Firewall), transaksjoner (OTS og JDBC), XML, Universal Naming-, Directory-, og Messaging Service.

Voyager Advanced Messaging (VAM)

Voyager Advanced Messaging tar utgangspunkt i at hver tjener registrer et *serversubspace*, og at hver klient registrerer et subspace som begge er IP-adresse/port kombinasjoner. Dette kan sammenlignes med Java RMI eller CORBA, der tjenester registreres for at andre komponenter skal kunne bruke dem. Forskjellen er at her må også klientene registrere seg, og *subspace*s kan kobles sammen. Serversubspace er en type subspace som kan kobles til andre serversubspace. Meldingsutveksling foregår mellom de registrerte komponentene, ved at unicast, multicast eller broadcast blir brukt [26]. Det er viktig å ikke forveksle multicast her med IP multicast, som foregår på et lavere nivå enn Voyager. Hver sammenkobling av subspace bruker Voyager Universal ORB for å sende og motta meldinger mellom hverandre.

I Figur 3-22 vises et eksempel på hvordan subspace kan kobles sammen med serversubspace (merket med dobbel strek). Til sammen danner serversubspace og subspace noe de kaller for *space*. Dette er bare et navn på en abstrakt samlingen av subspace. Serversubspace må registreres på en maskinnavn og port (her: tyrfing:8001), og de andre må koble seg på dette ved en *lookup* i navnerommet. Multicast av meldinger foregår ved at det dannes en proxy for det aktuelle space som meldingen blir sendt til. Meldingen går deretter til serversubspacet og vandrer gjennom de registrerte subspace til alle har mottatt meldingen. Det er funksjonalitet for at meldinger ikke skal mottas flere ganger. Meldingene blir klonet for hvert subspace og levert til hvert objekt som er interessert i meldingen, slik at den ikke skal ta opp unødvendige ressurser.



Figur 3-22 Advaced Messaging i Voyager

VAM støtter også en Pub/Sub arkitektur, slik som det ble diskutert for JMS i kapittel 3.2.3. Topic er i Voyager et hierarkisk oppbygd felt adskilt med punktum (*"dmj.cd.motion-detect"* for eksempel), som brukes til adressering. Dette er imidlertid ikke støttet direkte, og utvikleren må registrere interesse i alle meldinger ved å implementere *PublishedEventListener* grensesnittet, og implementere kontrollen av topic feltet selv i metoden *publishedEvent*. Hvert subspace har selv ansvaret for å sende meldingen videre, noe som gir muligheter til å lage filtre selv.

Vurdering av Voyager Advanced Messaging

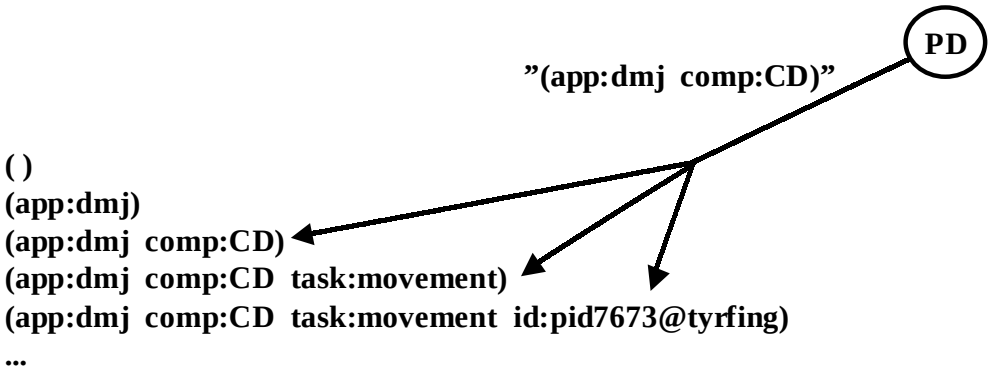
Voyager er en fleksibel arkitektur, som støtter kommunikasjon mellom DCOM, CORBA og RMI komponenter. Meldingsutveksling er også støttet i Voyager Universal ORB, der alle komponenter må registrere seg som *subspace*. Meldingen flyter mellom komponentene via ORB implementasjonen, og klones der det er mulig. Multicast blir imidlertid ikke reell så lenge alle meldingene kommer til en komponent uansett, hvis ikke et selvlaget filteret blir implementert i hvert subspace. Et interessant aspekt med Voyager for DMJ, er støtte for mobilitet. Dette vil bli beskrevet i detalj senere i rapporten.

3.2.5 Message Bus

The Message Bus (MBus) [43] er en enkel og kraftig teknologi for å sende og motta multicast meldinger mellom komponenter over IP multicast. MBus ble opprinnelig laget for å støtte interaksjon mellom komponenter som skulle inngå i komplekse multimedia konferanse systemer, men kan også benyttes i

andre sammenhenger. MBus meldinger som sendes mellom komponentene er designet for å styre og konfigurere multimedia komponenter og interaksjon med bruker. Det finnes implementasjoner av MBus både for Java og C++, og MBus kan brukes på alle plattformer som støtter IP multicast.

Adresseringen gjøres enkelt ved å bygge opp tekstpar som utgjør adressen. Alle komponentene som bruker MBus må registrere en adresse, og hvis meldingens adressen er et subsett av destinasjons adressen, så blir meldingen sendt til mottakeren. Figur 3-23 viser hvordan en melding går fra en PD til andre komponenter. Sendes en melding til adressen ”( )” kommer den til alle mottakere som bruker MBus. Komponentene holder også oversikt over alle andre komponenter, og hvilken adresse de har ved at de sender ut *hello* meldinger med jevne mellomrom for å fortelle at de er i live. Derfor blir ikke meldinger sendt ut på nettet hvis det ikke er noen som har registrert en adresse som meldingen skal til.



Figur 3-23 MBus adressering

Meldingene i MBus består av kommandoer som har en egen struktur. Disse ligner på en metode deklarasjon, *tekst (argumenter)*. Teksten først definerer kommandoen, og argumentene kan være av forskjellige MBus-typer. MBus har definert egne typer (*MDataType*), hvor de viktigste funksjoner er støttet. Eksempler på disse er *MData* (tabell av byte), *MFloat*, *MInteger*, *MList* (tabell med *MDataType*), MString (der ikke alle tegn er støttet) og MSymbol.

Vurdering av MBus

MBus støtter IP multicast som den eneste av de teknologiene som er vurdert. Dette er en stor fordel ved MBus som teller mye på grunn av egenskapene IP multicast har. MBus er også velegnet for en distribuert eventbroker, siden MBus må implementeres i alle komponentene som skal ha mulighet til å sende eller motta eventer. Det ligger en del restriksjoner i hvilke typer det er lov å sende i en MBus melding (Egendefinerte tekster, tall, tabeller, osv.). Dette kan være en faktor som er viktig for om denne teknologien blir valgt.

3.2.6 Oppsummering av eventbasert kommunikasjon

Kommunikasjon er et vidt begrep som må settes i en sammenheng for å kunne forstås. I denne oppgaven er kommunikasjon sett i forhold til komponenter og distribuerte systemer (DS). Mange begreper settes i sammenheng med DS som omhandler forskjellige sider ved kommunikasjon. Multicast av meldinger mellom komponenter er heldig i situasjoner med mange involverte maskiner over heterogene nettverk. The Message Bus (MBus) er en foreslått standard som bruker IP multicast teknologien i sin meldingstjeneste. Distribuerte filsystemer blir ofte brukt mellom maskiner i et lokalnett for å oppnå aksesstransparens. NFS (Network File System) [34] er et eksempel på dette. DMJ arkitekturen skal også håndtere mediastrømmer slik at kommunikasjon også omhandler hvordan mediastrømmer skal distribueres i nettverket.

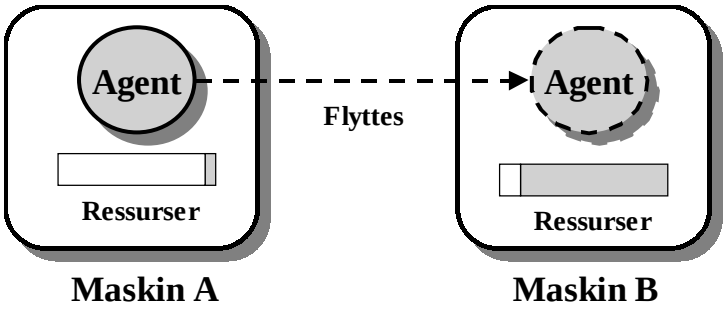
Som for komponentmodeller finnes det mange standarder som definerer kommunikasjon mellom komponenter. Noen av disse er DCOM, CORBA, Java RMI og ulike Object Request Broker (ORB) implementasjoner. DCOM krever et Windows miljø for å fungere godt, selvom det finnes løsninger for andre plattformer også. CORBA er utviklet av standardiserings organisasjonen Object Management Group (OMG), og den legger mange nyttige tjenester til en standard Object Request Broker (ORB). Java RMI er en del av Sun sin standard utviklingspakke, og er en enkel modell som bygger på Remote Procedure Call (RPC). Mobile Agenter brukes til migrering av komponenter mellom maskiner, og Voyager Universal ORB har mobilitet som en av sine sterke sider.

3.3 Mobile agenter for å støtte migrering

> *"A Mobile Agent is a computational entity, which acts on behalf of others, is autonomous, pro-active and reactive and exhibits the capability to learn, cooperate and move."*
>
> - Pagurek, Bieszczad og White [3]

Det er ikke lett å beskrive hva en agent er, fordi dette begrepet kan ha mange forskjellige betydninger. Definisjonen som er gjengitt over tar imidlertid for seg mange viktige egenskaper ved mobile agenter som er viktige å nevne. For det første blir agenter beskrevet som en egen entitet som kan utføre beregninger. Mobile agenter skal benyttes av en tredje-part, og de må være selvstendige/ autonome. Agenten skal også kunne ta valg selv som for eksempel det å migrere og tilpasse seg miljøet den brukes i. I tillegg beskriver definisjonen at mobile agenter innehar den egenskapen at de kan lære (som i kunstig intelligens), samarbeide og flytte (migrere). En noe mer enkel definisjon brukes i denne rapporten, hvor mobile agenter defineres som ”*programkomponenter som kan flyttes mellom maskiner i heterogene nettverk*”.

Mobile agenter er selvstendige utførende komponenter som kan migrere fra maskin til maskin i nettverket under run-time. Agentene bruker tilgjengelige ressurser på maskinen den befinner seg på, for å løse den oppgaven den er satt til (Figur 3-24). Mobile agenter krever en plattform å ”lande” på for å kunne flyttes til en maskin, noe som ofte kalles agentsystemet. Mobile agenter har sin egen tilstand og styringsmekanisme, og er uavhengige av andre komponenter for å fungere. Dette gjør at de er autonome eller selvstyrende.



Figur 3-24 Mobil agent

Det er flere grunner til å bruke mobile agenter i distribuerte systemer. I en vanlig klient – tjener arkitektur benyttes gjerne noe liknende RPC [1] for å kommunisere. En mobil agent har mulighet til å migrere fra klientmaskinen til servermaskinen for å bruke ressurser og kommunisere direkte med tjeneren. Dette fører ofte til at utnyttelsen av nettet øker, siden den mobile agenten kun flyttes en gang og klienten må sende gjentatte forespørsler mot tjeneren [31].

3.3.1 Agentsystemer

Det finnes mange Mobile agentsystemer, og for en oversikt over disse kan [2] og [31] anbefales. I denne oppgaven vil kun mobilitet i Voyager [12] bli beskrevet som et eksempel på et Java-basert agentsystem. Andre agentsystem fungerer på en lignende måte som Voyager og derfor betraktes Voyager som representativ i forhold til de andre agentsystemene.

Mobilitet i Voyager

Objectspace Voyager Universal ORB inkluderer et rammeverk for mobile agenter. Siden Voyager er implementert i Java, må også applikasjoner som bruker rammeverket implementeres i Java. Agenter er klasser i Java som er serialiserbare, og disse implementerer et grensesnitt *IAgent* eller utvider klassen *Agent*. I Voyager kan objekter flyttes til andre maskiner i nettverket med kall på *moveTo()* metoden. Objektet som skal flyttes må enten være en agent eller inneholde en *facet*. En *facet* er et objekt som "henges på” objektet som implementerer *IAgent* grensesnittet. Ved et kall på *Agent.of(this)* tilegnes en *facet* til objektet. Hele kallet på *moveTo()* blir *Agent.of(this).moveTo ("adresse", "metodenavn")*, der metodenavn er metoden som blir kalt når

objektet er flyttet til den nye JVM som befinner seg på adressen i første parameter.

Mobile agenter i Voyager er autonome hvis ikke annet er spesifisert. Et kall på *setAutonomous(false)*, gjør at agenten ikke lenger er autonom, og blir merket for bli deallokert av *garbagecollector* i JVM.

3.3.2 Mobile agenter som komponentmodell

En mobil agent i Voyager tilfredsstiller kravene som settes til en komponent. Grensesnittene mot en mobil agent kan defineres spesifikt med bruk av *interface* i Java. Den brukes av en tredjepart og må være serialiserbar. I tillegg brukes kommunikasjonsmodellen i Voyager til å kommunisere med den mobile agenten. En mobil agent i Voyager kapsler inn både tilstand og adferd, som kun er tilgjengelig fra agentens grensesnitt. Dette begrunner at mobile agenter i Voyager er komponenter, og derfor kan Voyager Universal ORB brukes som komponentmodell.

3.3.3 Vurdering av Mobile Agenter

Mobile agenter tilbyr mange interessante egenskaper som passer godt for DMJ komponentrammeverket. Kravet om migrering blir godt støttet med mobile agenter. Det å finne en komponentmodell som støtter migrering er ikke enkelt, selvom en mulighet er å formulere spørsmålet med en litt annen vinkling: Kan et agentsystem være en komponentmodell? For at mobile agenter skal kunne brukes som komponentmodell, må en mobil agent støtte kravene som stilles til en komponent. Mobilitet i Voyager Universal ORB er et agentsystem som enkel lar brukeren opprette agenter og plassere og migrere disse blant nettverkets maskiner.

Designet stiller krav til modularitet og migrering som begge er støttet av Voyager, noe som gjør at denne arkitekturen står sterkt i valget mellom teknologier. Mobile Agenter er også godt egnet til å være fleksible og tilpassningsdyktige til hvilke ressurser som er tilgjengelig. Agentsystemer er også velegnet for kunstig intelligens [38].

I en implementasjon av mobile agenter [30] ble Voyager brukt som underliggende kommunikasjonsmodell. I [30] tar Jahr for seg problematikken med å flytte agenter rundt i et distribuert system (DS), og hvordan de behandler kode og ressurser. Jahr har laget et system som kalles MORP (Multi-Organization Resource Planner), som er et system som støtter reservering av møterom i andre bedrifters lokaler. MORP benytter en mobil agent som utvider det eksisterende systemet med en agendafunksjon. Applikasjonen benyttet Voyager for å kommunisere mellom objektene, og for å migrere agenten i nettverket.

Multicast Mobile Measurement Architehture (M^3A) [11] bruker Voyager i en arkitektur for målesystemer. Eksemplene som brukes i dokumentasjonen er monitorering og detektering av frekvenser og strømmer fra generatorenheter som er koblet sammen med internett. Mobile agenter flytter seg til den enheten som produserer strømmer, og sender meldinger til en egen komponent som viser resultatene. Arkitekturen er fleksibel, åpen, portabel og dynamisk rekonfigurerbar. Den bruker Voyager Advanced Messaging sammen med IP multicast for å sikre god utnyttelse av nettet.

3.4 Oppsummering og valg av teknologi

I dette kapittelet ble ulike teknologier for komponentmodell og kommunikasjon presentert og ble vurdert om disse var egnet i forhold til å tilfredsstille design av implementasjonen. For hver teknologi ble det vurdert hvordan modellen støtter kravene som stilles til designet. Her begrunnes valget av teknologi nærmere.

Valget av en komponentmodell er vanskelig siden det henger nøye sammen med hvilken kommunikasjonsmodell som blir brukt. For eksempel ville et valg av COM som komponentmodell være veldig styrende for hvilken kommunikasjonsmodell som velges. Komponentmodellen må ha støtte for migrering, og det er det ikke alle modellene som støttet. Voyager tilbyr en interessant løsning som også støtter kravet om migrering. I forhold til mange av de andre modellene som COM, JavaBeans og EJB, er ikke omgivelsen her så ressurskrevende. I tillegg er Voyager implementert i Java, noe som gjør at plattformuavhengighet følger med.

Begrepet mobile agenter passer veldig godt til distribuert eventtjeneste, fordi agentene flyter rundt mellom maskiner og kan være hvor som helst i nettet. Eventbrokeren skal være distribuert, noe som vil si at hver agent må ha tilgang til eventbrokeren fra der den befinner seg. Teknologien som synes å passe best til DMJ eventbrokeren, er MBus. Dette er begrunnet i at teknologien er veldig enkel og kraftig, noe som passer veldig godt sammen med mobile agenter. MBus bruker også IP multicast og gir god utnyttelse av nettet og lokasjonstransparens.

Sammen støtter mobile agenter i Voyager og den lokasjonstransparente MBus kravene som er satt i designet av DMJ komponentene. Kommunikasjon mellom analysekomponentene støttes av MBus som benyttes til en distribuert eventtjeneste. Distribusjon og migrering støttes av mobile agenter i Voyager, og modularitet og fleksibilitet støttes gjennom komponentmodellen som agent-systemet i Voyager tilbyr. Vurderingen av teknologier er gjort ut fra et rimelig antall teknologier som kan være innen rammen for en hovedoppgave. Det påpekes derfor at vurderingene ikke er uttømmende. Neste kapittel tar for seg hvordan disse to teknologiene kan brukes til å implementere de utvalgte DMJ komponentene.

Kapittel 4

IMPLEMENTASJON AV PROTOTYPEN

Dette kapittelet tar for seg hvordan design av komponentene fra Kapittel 2 kan implementeres med bruk av teknologiene som ble valgt i Kapittel 3. Voyager og MBus danner grunnlaget for implementasjonen av prototypen som presenteres i dette kapittelet. Komponentene som i detalj blir beskrevet er analysekomponentene (PD og CD), Eventbrokeren og Journalering Kontroller komponenten. Kravene til design av DMJ arkitekturen vil også i dette kapittelet danne bakgrunnen for diskusjoner rundt den valgte løsningen.

Først i dette kapittelet vil det bli beskrevet hvilket språk som er best egnet til å bruke i implementasjonen. Dette er et viktig valg, fordi ytelse og andre egenskaper varierer mye mellom programmeringsspråk. Deretter vil implementasjonen av prototypen bli beskrevet, der komponentene som nevnt vil bli nøye beskrevet. Det nevnes også hvordan kommunikasjon med andre komponenter i rammeverket kan løses med Voyager. Til slutt beskrives det hvordan prototypen kan brukes til å utvikle distribuerte journalerings applikasjoner.

Spørsmål som blir besvart i dette kapittelet er:

- Hvilket programmeringsspråk egner seg best til denne prototypen?
- Hvordan kan design og teknologier benyttes i implementasjonen

4.1 Programmeringsspråk

Det finnes mange forskjellige programmeringsspråk å velge mellom, og valget har stor innflytelse på andre valg i rammeverket. Et viktig krav til programmeringsspråket er at det skal være mulig å utvikle komponenter eller applikasjoner. Dette er stort sett mulig i alle programmeringsspråk, men det er også mange som har definert sin egen komponentmodell. En vurdering av programmeringsspråk bør sees i sammenheng med valg av komponentmodell. Det stilles krav til programmeringsspråket om mulighet til distribusjon. Det må tas hensyn til hvilke plattformer som programmeringsspråket kan kjøres på, og om det må rekompileres for å skifte plattform. Ytelse er også viktig,

siden det stilles store krav til fornuftig bruk av ressurser i et DMJ system. Støtte av multippel arv, polymorfi og skille mellom grensesnitt og klasser, er elementer som kan forenkle utvikling av komponenter [28].

Interface Definition Language (IDL) kan brukes til modellering av grensesnitt for å utelate spørsmålet om implementasjonsspråk. Modelleres grensesnittene i IDL er imidlertid valget tatt for hvordan komponentene skal kommunisere. De to mest kjente kommunikasjon modellene som har en IDL er CORBA og COM. [28] er en teknisk rapport som tar for seg objektorienterte programmeringsspråk for komponentorientert programmering. Rapporten beskriver dette som en vanskelig situasjon for utviklere, som ofte må velge enten kommunikasjonsmodell eller programmeringsspråk.

Det er noen av programmeringsspråkene som tillater bruk av kode skrevet i andre språk, slik at det er mulig å bruke flere språk samtidig, mens andre språk er mer restriktive, noe som setter store begrensninger for områder i utviklingen av prototypen. Dette gjelder særlig Visual Basic (VB), som må ha et Windows miljø (eller tilsvarende) for å fungere godt. VB er utelukket fordi kravet om at språket skal fungere på flere forskjellige plattformer ikke blir støttet.

C++ og Java har hver sine særpreg og fortrinn. Java har laget en Java Virtuell Machine (JVM), som gjør at den koden som kompileres kan kjøres på alle plattformer det finnes en JVM til. Dette gjør at også at en JVM må starte opp før et program kan utføres, noe som gjør utførelsen tregere. Selve språket har mye til felles med C++, men er ikke så maskin-nært. C++ har sitt fortrinn nettopp av den grunn at det er maskin-nært. Derfor blir programmene raskere men plattformavhengige. Dette gjør at C++ ofte velges når det er et krav om rask eksekvering. Ytelsestester som tar for seg forskjellen mellom C++ og Java i eksekveringshastighet viser at C++ ennå er noe raskere, selvom forskjellen ikke er stor som det var da Java kom på banen. Dette har mest med Java Just-In-Time (JIT) kompilator som gjør nytte av at operasjoner utføres flere ganger. Nedenfor er det listet opp noe av de viktigste egenskapene til disse to språkene.

C++

- Maskinavhengig
- Rask eksekvering
- Optimalisert kode kan være meget uleselig (gjenbrukbarhet)
- Er i noen tilfeller ikke tilstrekkelig for å lage komponenter

Java

- Maskinuavhengig
- Java har en fritt tilgjengelig utviklingsomgivelse
- Er åpen mot andre teknologier
- Har egen komponentmodell
- Det er mulig å bruke kode skrevet i annet språk (Java Native Interfaces)
- Er det programmeringsspråket som blir mest brukt av utviklere i dag.

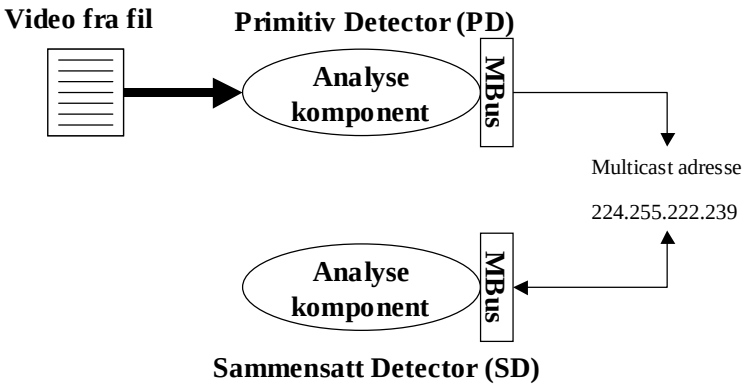
I "Component Software" [5] skriver Szyperski at C++ ikke i alle tilfeller strekker til for å lage komponenter. Dette begrunnes med at objektmodellen til C++ er for svak og ikke støtter gjenbruk av "black box" komponenter, som ble valgt bort til fordel for multippel arv. I Java er dette innebygget med grensesnitt (interface) som klassene implementerer. Dette vil si at klassene må implementere de metodene som blir definert i grensesnittet.

I en implementasjon av PREMO (PResentation Environment for Multimedia Objects) [6], et rammeverk for multimedia mellomvare, ble Java valgt som programmeringsspråk. Valget ble tatt på bakgrunn av om programmeringsspråket har multippel/single arv, skille mellom interface/objekt, primitive datatyper og opprettelse/frigjøring av objekter. Java har unntak (*exception*), eventer og tråer (*thread*) som standard i sitt språk, noe som også PREMO brukte. I tillegg har Java et pakke (*package*) begrep til å samle klasser som naturlig hører sammen, og som har støtte for mange plattformer uten at endringer i koden er nødvendig.

Valget av programmeringsspråk har i tillegg innvirkning på komponent-modellen som brukes og omvent. Voyager er valgt som teknologi for å støtte komponentmodell og mobile agenter, og den universale ORB'en er utviklet i Java. Java er også støttet i kommunikasjonsmodellen MBus, og dermed er det valgt å implementere DMJ komponentene i Java.

4.2 Prototypen

Komponentene som ble designet i Kapittel 2 skal implementeres med den valgte teknologien fra Kapittel 3. Her vil det bli gått nærmere inn på hvordan komponentene er implementert. En basis for prototypen ble laget først, som et utgangspunkt for videre testing (Figur 4-1). Basis prototypen skal lese en videofil og analysere den med algoritmer som kan implementeres i JMF (Java Media Framework [40]). Dette skal danne primitiv detektoren (PD), som bruker et sett med algoritmer (implementert som JMF Codec's). Slik JMF er bygget opp settes det inn en kjede av algoritmer (Codec-chain), før analyseprosessen startes. Algoritmene sender eventer hvis tilstanden i analysen tilsier det (Sceneskifte algoritmen skal levere et event hvis det oppdages sceneskifte i videoen for eksempel). Eventene distribueres med MBus til sammensatte detektorer (CD). MBus sender eventene til en multicast adresse, slik at senderen ikke trenger å vite lokaliseringen til mottakeren i nettverket. En PD skal også kunne sende nye RTP strømmer, som da kalles et Filter.



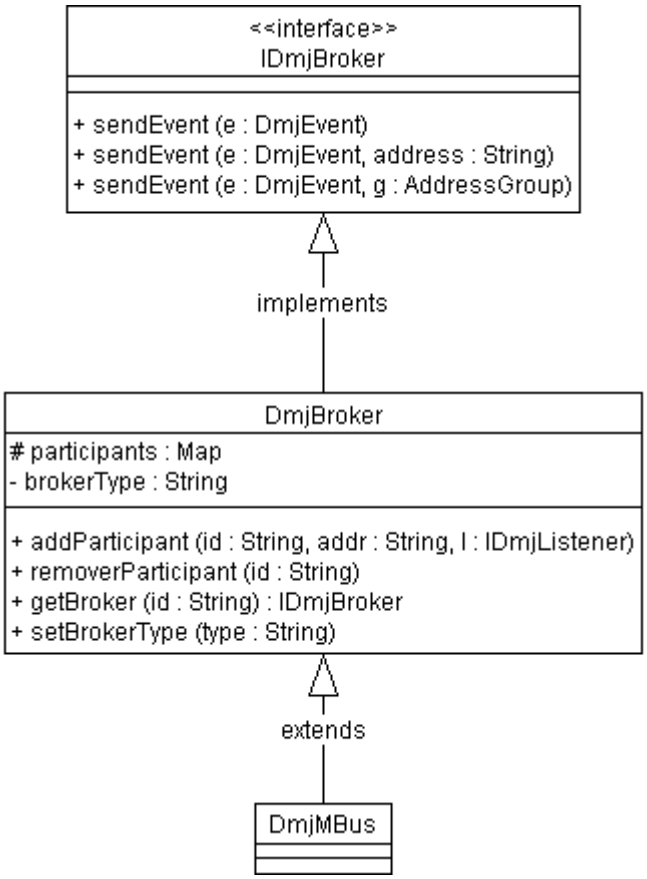
Figur 4-1 Basis for prototype i DMJ.

Det finnes mange rammeverk for utvikling av multimedia applikasjoner, som både er ment for å vise og prosessere media. JMF er valgt fordi det ligger nært til programmeringsspråket Java, og fordi rammeverket er under hyppig utvikling for å tilfredsstille nye krav. For en oversikt over rammeverk for utvikling av multimedia applikasjoner kan [33] anbefales.

Implementasjonen skissert i Figur 4-1 danner en basis for den videre utvikling. Koden til alle Java-klassene som er brukt i prototypen er inkludert i appendix A.

4.2.1 Eventbroker

Eventbrokeren er en sentral komponent i rammeverket. Den definerer hvordan analysekomponentene kommuniserer mellom hverandre ved hjelp av hendelser (eventer). I denne implementasjonen er det bare en klasse som står for kommunikasjonen. Man står likevel fritt til å arve fra *DmjBroker* klassen for å benytte andre kommunikasjonsmodeller som håndterer eventer. MBus er brukt her som vist i Figur 4-2, for å utnytte den gode egenskapen ved multicast. Hvis et stort antall datamaskiner blir brukt i DMJ applikasjonen kan multicast skalere bedre enn andre løsninger.



Figur 4-2 Eventbrokeren i DMJ

Eventbrokeren brukes ved å statisk (*static* i Java) sette hvilken type kommunikasjonsobjekt som skal brukes med metoden *setBrokerType*. *DmjBroker* klassen bruker dynamisk lasting av klasser som tilbys i Java (*Class.forName()*). Dette gjøres hver gang noen vil registrer en ny deltaker for å opprette et nytt kommunikasjonsobjekt fra klassen som *brokerType* beskriver. Dersom ikke Java *classloader* finner en klasse som har dette navnet, brukes *DmjMBus* klassen som er definert av rammeverket og er initielt satt i *brokerType*.

Enten man vil lytte etter eller sende eventer, må man registrere en deltaker (objekt av klassen Participant) i *DmjBroker* klassen. Dette gjøres for at flere komponenter kan registrere seg på samme maskin på forskjellige adresser. Adressene som settes i MBus tilsier hvilke eventer deltakeren vil motta (se beskrivelse av adressering i MBus i kapittel 3.2.5). Deltakeren registrer et navn (ID) slik at den samme MBus adressen kan benyttes flere ganger. Hvis det ikke er vesentlig hvor eventet kommer fra og man ikke forventer noen eventer tilbake, kan man bruke en deltaker som har ”dummy” som ID. Denne finnes i *DmjBroker* default. Koden under viser hvordan dette kan gjøres.

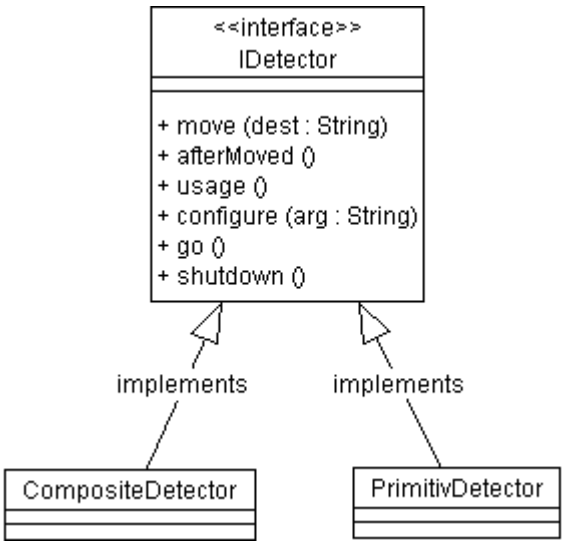
```
DmjBroker.getBroker(“dummy”).sendEvent(evt_obj);
```

I henhold til definisjonen av en komponent er det viktig å kunne gjenbruke komponenten og at grensesnittet er veldefinert. Eventbrokeren skal kunne plugges inn i andre systemer og brukes uten å endre komponenten. Dette støttes i implementasjonen av eventbrokeren, som gjør at kravet om modularitet er tilfredsstilt.

4.2.2 Analysekomponentene

Grensesnittet *IDetector* (Figur 4-3) definerer metoder som er felles for både Primitive hendelses Detektor (PD) og sammensatte hendelses detektor (CD). Rammeverket er implementert på engelsk, og derfor kalles CD for Composite Detector (CD) i implementasjonen. En analysekomponent skal kunne flyttes til en annen adresse, og dette gjøres ved kall på *move.* Metoden *afterMoved* blir kalt etter at komponenten er plassert ut eller flyttet. Migrering av komponenter er beskrevet i kapittel 4.2.5. Ved kall på *usage* blir en tekst som forteller hvordan komponenten kan brukes, skrevet ut. Komponenten kan konfigureres med kommandoer representert i en tekst ved kall på *configure.* PD'ene bruker i denne implementasjonen Java Media Framework (JMF), og konfigurering av komponenten vil gjøre at en ny codec-chain blir satt inn i prosesseringsobjektet. Metoden *go* setter komponenten i startet tilstand, og *shutdown* gjør at komponenten avslutter.

Filter er også en type analysekomponent, som analyserer media direkte. Den kan produsere media som er endret i henhold til konfigurasjonen og lytte på eventer fra andre PD'er. Dette kan gjøre at den endrer atferd. I implementasjonen blir filtre startet som PD'er, og algoritmen som brukes mottar hendelser fra eventbrokeren.



Figur 4-3 Analysekomponenten

Det er kun Journalering Kontrolleren (JK) som bruker metodene i analyse komponentene. Andre komponenter går gjennom JK komponenten for å nå analysekomponentene. PD'ene og CD'ene kommuniserer med hverandre via DMJ Eventbroker.

Rekonfigurering av analysekomponenten

I grensesnittet mot analysekomponentene (PD'er og CD'er) finnes det en metode som heter *configure*. Denne metoden tar en tekst som angir algoritmer (codec's i JMF) og andre innstillinger for analysekomponenten. Analysen må avbrytes og de nye innstillingene blir satt inn i prosessen, før komponenten startes opp igjen. For å ikke miste data må en kloning av analysekomponenten opprettes mens rekonfigureringen gjøres. Dette utføres av Journalering Kontroller (JK) komponenten. Dette forutsetter at de komponentene som lytter på eventer klarer å se bort fra kopier eller eventer som sier det samme. Dette er nødvendig fordi en klonet komponent kan sende eventer som inneholder samme informasjon som den originale. Modellen støtter også kun push modellen.

Rekonfigurering av komponenter vil si at algoritmer skal kunne byttes ut, og en algoritme (som kan være sammensatt av flere algoritmer) kan fordeles på forskjellige maskiner for å gjøre analysen mer effektiv. Derfor må dette gjøres der analysekomponentene opprettes og styres fra, nemlig JK komponenten. Her kommer *reconfigure* metoden i JK komponenten til nytte, som finner den spesifiserte komponenten og kaller *reconfigure* i denne. JK kan fordele komponenter til forskjellige steder i nettverket, starte opp nye eller avslutte analysekomponenter etter hva som er nødvendig for å tilfredsstille rekonfigureringen.

4.2.3 DMJ Agent

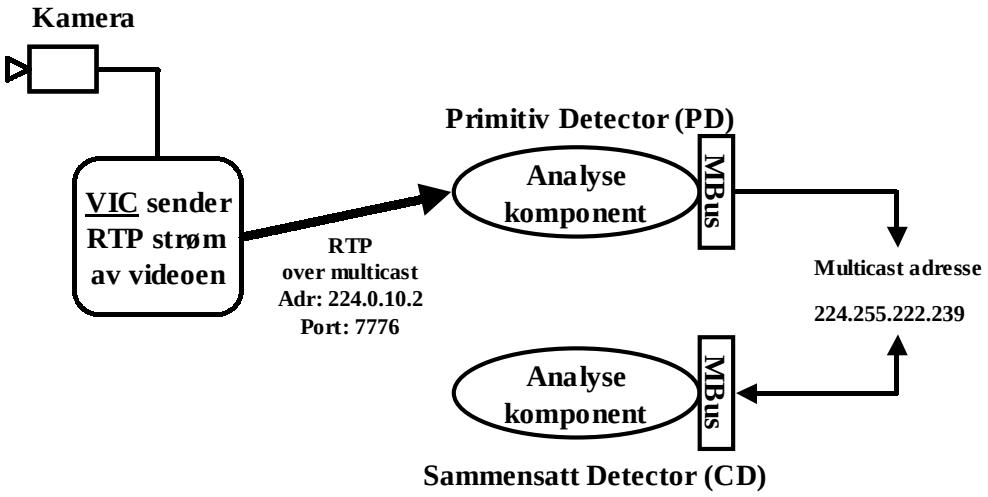
DMJ Agenten skal lese mediastrømmer fra multicast sesjoner eller fra opptaksenheter (Videokamera/Mikrofon), og sende dem videre til analyse komponenter. Den skal også kunne konvertere mellom mediatyper og stille på parametere som oppløsning og frekvens i media før det sendes videre.

Mediastrømmer må håndteres spesielt når de skal sendes over nettverket. De trenger en standard oppdeling for å kunne transporteres gjennom nettverket. En mediastrøm er for eksempel en konstant video- og lydstrøm fra et kamera som ikke har noen fast start eller slutt. Derfor må denne kommunikasjonen behandles på annen måte enn for eventer og migrering av komponenter.

Real-time Tranport Protocal (RTP) [36] er en standard for sending og mottak av datastrømmer som er støttet i JMF. Strømmene blir delt opp i datapakker og stemplet med en del metainformasjon for kontroll og synkronisering. I hver pakke som blir sendt finnes et tidsstempel og rammenummer, slik at videoen kan spilles av hos mottakeren i riktig rekkefølge. RTP blir ofte brukt sammen

med UDP (User Datagram Protocol) for å benytte seg av checksum og multipleksing, slik at dataene ikke er ødelagte, og for at det skal være mulig å sende flere strømmer samtidig. UDP pakkene kan ta forskjellige veier gjennom nettverket, slik at senderen ikke har kontroll over dataene som blir sendt. Hvis en pakke blir borte i nettet betyr dette at en ramme i videoen ikke kan vises. Dette har ikke vesentlig innvirkning på hvordan videoen blir presentert hos mottakeren. RTP tilbyr en enkel og rask måte å sende media i nettet, og den blir derfor ofte brukt til for eksempel video konferanser.

I prototypen transporteres video over RTP mellom DMJ Agenten og de primitive analysekomponentene (PD'ene). Implementasjonen bruker VIC (Video Conference Tool) [49] som DMJ agent, og den sender videostrømmen ut på en multicast adresse (Figur 4-4). VIC sender RTP strømmer som kan leses av de primitive analysekomponentene.



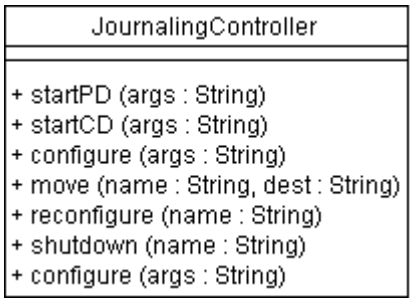
Figur 4-4 Utvidelse av basismodellen med video fra VIC

DMJ Agenten er ikke den viktigste komponenten i rammeverket, fordi ved testing av komponentrammeverket kan en fil like gjerne benyttes. I kravet om fleksibilitet bør DMJ Agenten kunne styres fra analysekomponentene, slik at for eksempel oppløsning og rammerate kan endres under run-time. Dette er et vesentlig punkt for fleksibilitet som ikke denne implementasjonen tar opp. JMF støtter ikke at oppløsning av video endres under run-time, som er en svakhet med JMF.

4.2.4 Journalering Kontroller

Journalering Kontroller (JK) styrer opprettelsen og destruering av analyse komponenter. Komponenten samarbeider nært med en ressursforvalter, fordi ved utplassering av analysekomponenter må den spørre etter adressen til maskinen som er best egnet på det gitte tidspunktet. JK styrer flytting og rekonfigurering av analysekomponenter. Dette gjøres ved at en klonet komponent opprettes før den kan stoppes og rekonfigureres (Se kapittel 4.2.2

for en detaljert beskrivelse). Figur 4-5 viser en oversikt over de implementerte metodene i komponenten.

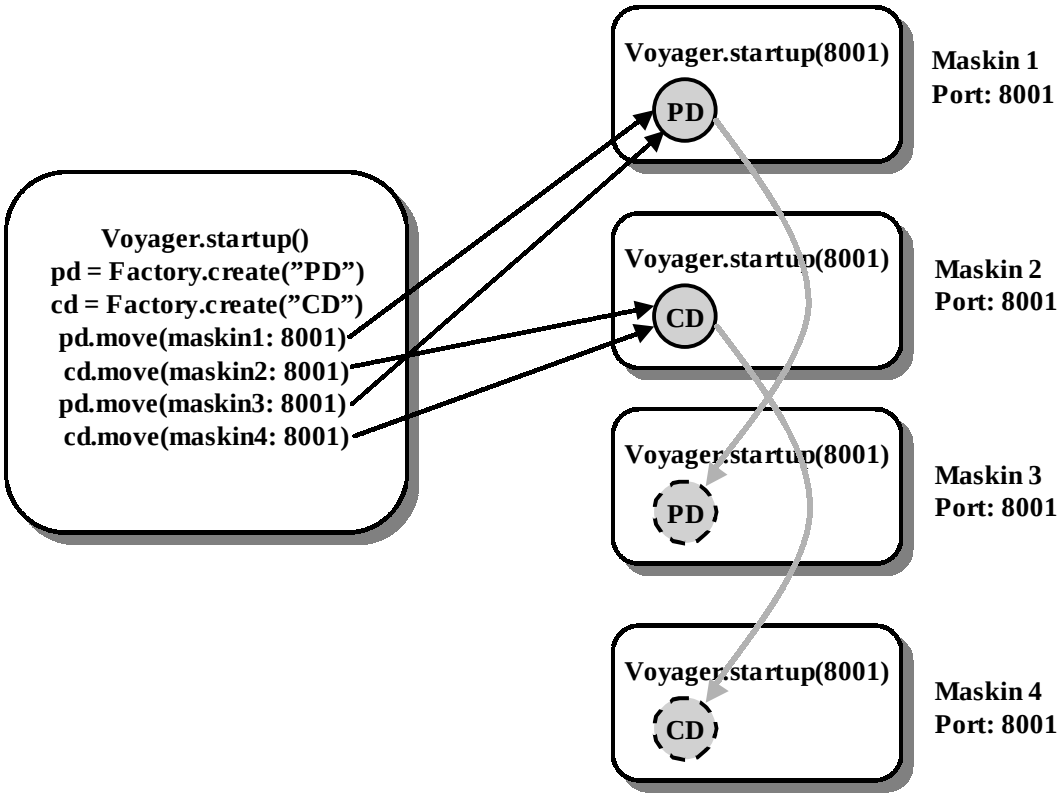


Figur 4-5 Journalering Kontrollerer komponenten

4.2.5 Migrering

En del av problemstillingen i denne oppgaven er hvordan analyse komponentene skal flyttes mellom maskingrenser. I kapittel 3.3 vurderes mulige måter å flytte objekter i nettverket, og her foreslås Voyager som støtter migrering av komponenter mellom maskiner i nettverket. Det er viktig at alle objektene er serialiserbare for at migrering av komponenter fra maskin til maskin skal fungere.

Journalering Kontroll komponenten genererer PD'er og CD'er som plasseres ut til maskiner i nettverket med metoden *move* (se Figur 4-6). Dette er samme metoden som brukes for å flytte komponenter mellom maskiner under run-time. Etter at komponenten er flyttet startes prosesseringen av media fra metoden *afterMoved*. Migrering av komponenten startes når analyse-komponenten får et kall på *move* under run-time. Referansene til objektene som ikke kan serialiseres blir flyttet til en klasse som heter *ObjectKeeper*. På denne måten avsluttes ikke prosessen som analyserer media på den gamle maskinen før den nye analysekomponenten har startet, slik at ikke media skal gå tapt i flyttingen.



Figur 4-6 Eksempel på hvordan Voyager agenter kan flyttes i nettverket.

4.2.6 Kommunikasjon og distribusjon av andre komponenter

Komponenter som ikke skal flyttes rundt i nettverket trenger ikke en hendelsesdrevet kommunikasjon. Her passer en modell som Java RMI, CORBA eller andre metoder, hvis komponentene skal lokaliseres på forskjellige maskiner. Et eksempel på komponenter som det er nødvendig å koble sammen er Ressursforvalteren og Journalering Spesifikasjon mot Journalering Kontroller (se skisse av arkitektur i Figur 1-5 på side 8). JK komponenten er implementert med Voyager som oppretter analysekomponentene som mobile agenter. Voyager støtter bruk av RMI, CORBA og DCOM, noe som betyr at utvikleren kan velge hvilken kommunikasjonsmodell de vil bruke for å koble sammen komponentene. Eksempel 1 og Eksempel 2 viser hvordan Voyager og Java RMI kan kombineres. Ved å bruke Voyager slipper man i tillegg å starte RMIregistry, og alle stubs blir generert automatisk av Voyager.

```
// Starte Voyager tjeneren på port 8001
Voyager.startup("8001");

// Lage et nytt objekt av JK komponenten
IJournalingController  control;
control = Factory.create(path+"JournalingController");

// Gjøre ressurs tjeneren tilgjengelig i RMI
ClassManager.enableResourceServer();

// Binde objektet til rmi adressen
Namespace.bind("rmi://gang.ifi.uio.no:8001/control",
control);
```

Eksempel 1 Slik settes Voyager opp i JK komponenten

```
// Installere RMI security manager
System.setSecurityManager( new RMISecurityManager() );

// Hente referansen til objektet som er registrert i JK
komponenten
IJournalingController control;
control =
Naming.lookup(”rmi://gang.ifi.uio.no:8001/control" );
```

Eksempel 2 Slik hentes referansen til JK komponenten

4.3 Bruk av prototypen

DMJ komponentrammeverk skal være enkelt å bruke, og komponentene skal kunne brukes hver for seg eller sammen. Kapittel 2.2.5 beskriver en media journalering applikasjon [37] som komponentrammeverket kan utvikle enklere. Her blir det sett på hvordan en tilsvarende applikasjon kan utvikles med prototypen.

For å bruke prototypen er det en del krav som stilles til miljøet (operativsystem og installerte programmer). Hele rammeverket er definert i en jar-fil, og derfor må en referanse til denne filen settes i omgivelses-variabelen CLASSPATH i operativsystemet. Java 2 (JDK versjon 1.2 eller nyere) må også være installert, samt Java Media Framework (JMF, versjon 2.1 eller nyere). For å bruke flere maskiner til applikasjonen, må disse kravene tilfredsstilles på alle maskinene. Det må finnes en *Java Virtuell Machine* (JVM) implementert for hvert operativsystem som benyttes.

Eksempel 3 viser Java kode for å starte en Primitiv Hendelses Detektor med noen spesifiserte argumenter. Argumentet *codec* står for hvilke(n) algoritme man vil bruke. Utvikleren står dermed fritt til å lage egne algoritmer og

benytte de i steden. På tilsvarende måte kan Sammensatte Hendelses Detektorer lages og settes i arbeid.

```
// Lage argumentene til primitiv hendelses detektorene
String args = ”debug noaudio
              url=file:news_file_01052001.mov
              codec=NameOfAlgorithm(threshold=10 ...)”;

// Lage en ny primitiv detektor
IDetector pd = new PrimitivDetector();

// Konfigurere detektoren med argumentene
pd.configure(args);

// Starte detectoren
pd.go();
```

Eksempel 3 Oppstart av en Primitiv hendelses detektor

Man kan eventuelt benytte de medfølgende hjelpeklassene som befinner seg i *no.uio.ifi.dmj.utils* pakken som støtter opprettelse, migrering, rekonfigurering og fjerning av analysekomponenter. Disse klassene bruker Journalering Kontroller komponenten til å styre opprettelsen av analysekomponenter. Eksempel 4 viser hvordan Journalering Kontrolleren startes opp, og hvordan analyse komponentene kan startes opp ved hjelp av JK.

```
// Starter Journalering Kontroller
java no.uio.ifi.dmj.utils.StartJC

// Starter en PD med argumentene som er gitt i args
java no.uio.ifi.dmj.utils.StartPD name=PD1
                                  dest=localhost
                                  args

// Starter en CD med argumentene
java no.uio.ifi.dmj.utils.StartCD name=CD1
                                  dest=localhost
                                  codec=HMM(args)
```

Eksempel 4 Oppstart av analysekomponentene med bruk av JK

I eksemplene ovenfor er *localhost* brukt som adresse. Dette betyr at alle komponentene vil bli opprettet på den lokale maskinen, og ikke flyttet til en annen lokasjon. Hvis det er ønske om å distribuere applikasjonen på et senere tidspunkt, kan det gjøres kun ved å endre adressefeltet. I stede for å bruke *localhost* som adresse (*dest=localhost*), kan man sette inn navnet på en annen maskin og portnummer (*dest=//maskin:port*). Det er viktig at Voyager startes opp på hver maskin som brukes av DMJ applikasjonen, for å kunne plugge inn komponentene på andre maskiner. Dette kan gjøres ved å kjøre kommandoen under i terminal-viduet.

```
java no.uio.ifi.dmj.utils.StartVoyager
```

En applikasjon for automatisk indeksering av nyhetssendinger som beskrives i [37], kan med bruk av prototypen implementere de nødvendige algoritmer og spesifisere disse når PD'ene og CD'ene startes opp. Analysekomponentene kan enkelt distribueres ved å spesifisere destinasjonsadressen når de startes, eller flytte dem senere. På denne måten brukes ressursene på flere maskiner i nettverket, som kan føre til bedre ytelse. Komponenter er i tillegg enklere å endre på i etterkant, og applikasjonen er fleksibel mot nye algoritmer.

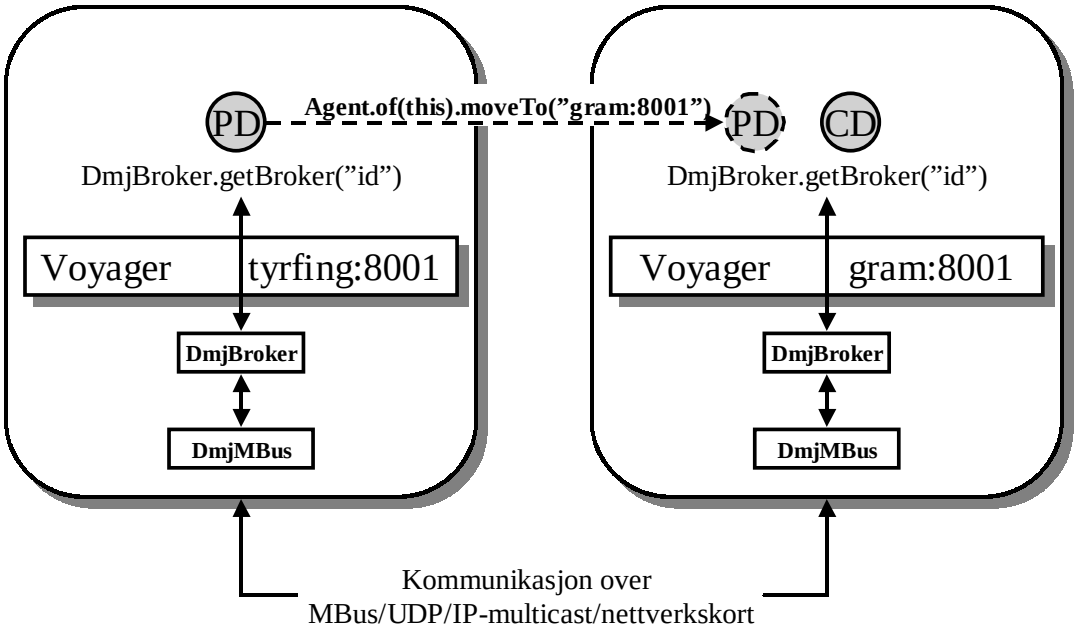
4.4 Oppsummering

Implementasjonen beskriver hvordan alle komponentene i rammeverket er implementert. I dette kapittelet er det i tillegg tatt med noen valg av teknologi, som for eksempel JMF til å bruke i primitiv detektorene. Hvilken teknologi som brukes for å prosessere media er ikke fokus i denne oppgaven. Derfor ble dette valgt her og ikke i teknologikapittelet.

MBus ble benyttet til å implementere eventbrokeren som alle analyse-komponentene bruker, slik at de kan sende og motta eventer. Hovedstrømmen av eventer går fra PD'ene til CD'ene, og mellom CD'ene. Når PD'en bruker en filter algoritme, skal PD'ene også kunne motta eventer.

MBus blir implementert som den eneste kommunikasjonsmodellen i eventbrokeren, men det står åpent for utvikleren å bruk andre metoder for å sende og motta eventer. Eventbrokeren er på denne måten en fleksibel komponent, som kan brukes av DMJ prosjektet for testing av forskjellige kommunikasjonsmodeller. Dersom en ny kommunikasjonsmodell vil testes, kan egenskapene til *DmjBroker* klassen utvides (*extends* i Java). Metodene som defineres i *IDmjBroker* grensesnittet må implementeres, slik at sending av eventer kan utføres. Kommunikasjonsmodellen kan byttes i *DmjBroker* klassen ved å spesifisere navnet på den nye klassen i metoden *setBroker*.

Voyager danner komponentmodellen for analysekomponentene i prototypen, og brukes også til migrering. Komponentene blir distribuert med *move* metoden, som flytter komponenten til en ny maskin. På denne måten støttes både distribusjon og migrering i Voyager. Komponentmodellen i Voyager er også fleksibel for nye komponenter som blir laget, fordi en analysekomponent kan bruke alle algoritmer som støtter JMF codec, bare ved å vite navnet på den (path + navn på klasse). Implementasjonen støtter også rekonfigurering av analysekomponenten ved at den først blir klonet for ikke å miste noe data, deretter blir den ene stoppet og rekonfigurert, og når den har startet igjen kan kloningen fjernes.



Figur 4-7 Kommunikasjon mellom analysekomponenter.

Figur 4-7 gir en oversikt over bruk av Voyager og MBus sammen. Den tar for seg bruk av eventbrokeren og flytting av en mobil agent. Voyager opptrer som en plattform som analysekomponentene ”lander” på, og bruker eventbrokeren (*DmjBroker*), til å sende og motta eventer. Eventbrokeren er tegnet inn som en protokoll-stack, som kommunikasjonen går gjennom. Nivåene under DmjMBus kommer UDP/IP og fysisk lag for å sende eventer til en annen maskin med Voyager.

Dette kapittelet har tatt for seg hvordan prototypen er implementert, og det er nå interessant å se hvordan prototypen virker i praksis. Neste kapittel tar for seg testing av prototypen for å undersøke om prototypen kan danne en plattform for videre utvikling i DMJ prosjektet. Resultatene fra testene blir vurdert opp mot kravene som stilles DMJ arkitekturen.

Kapittel 5

TESTING AV PROTOTYPEN

Dette kapittelet tar for seg testing av prototypen hvor det fokuseres på hvordan prototypen støtter sanntid og om implementasjonen skalerer til større systemer. Testene omfatter oppstart, migrering og rekonfigurering av en Primitiv hendelses Detektor (PD). Dette blir diskutert nærmere i testbeskrivelsen i forhold til kravene som stilles til DMJ arkitekturen. Utføringen av testene bør ha en godt beskrevet omgivelse som er rimelig stabil. Testomgivelsen tar for seg hvilke maskiner og kommunikasjon som blir benyttet, og beskriver hvordan komponentene i prototypen brukes på disse maskinene. Resultatene blir deretter satt opp med målinger fra testene som ble utført. Til slutt diskuteres resultatene opp mot kravene til DMJ arkitekturen.

5.1 Testbeskrivelse

Det blir her bli diskutert hvilke spørsmål testene kan besvare, noe som vil utforme hvordan testene skal utføres. Kravet om modularitet som stilles til DMJ arkitekturen er vanskelig å teste. Dette kravet har imidlertid blitt diskutert i forbindelse med og valg av design (kapittel 2.2.3) og teknologi (kapittel 3.1). De andre kravene vil her bli diskutert i hvilken grad de kan testes. Til slutt oppsummeres testene som skal utføres og beskrives i detalj.

5.1.1 Kommunikasjon

Analysekomponentene bruker eventbrokeren til å kommunisere med andre komponenter og det er interessant å finne ut hvordan eventbrokeren fungerer når antallet analysekomponenter øker (skalerbarhet). Det kan imidlertid regnes ut hvor lang tid et event bruker fra en maskin til en annen, ved å spesifisere kommunikasjonen mellom maskinene og kommunikasjonsmodellen som benyttes. Det er også viktig å fastslå størrelsen på dataene som skal sendes mellom maskinene. Et event kan inneholde forskjellige typer data og variere i størrelse etter hva som skal meddeles, så dette kan være vanskelig å finne ut av før DMJ rammeverket er fullstendig utviklet. Eventbrokeren bruker MBus for å sende meldinger mellom komponentene som benytter egenskapene ved IP multicast for å sende data. Dette kan gjøre at meldingene

bruker kortere tid enn ved unicast kommunikasjon, siden kopieringen av datapakkene skjer så sent som mulig i nettverket (se kapittel 3.2.1).

5.1.2 Distribusjon og migrering

DMJ rammeverket skal støtte distribuert journalering av media i sanntid, og det er derfor interessant å finne ut hvor lang tid prototypen bruker på diverse operasjoner. Sanntid støttes i den grad applikasjonen kan utføre operasjoner innen en avtalt tid. Implementasjonen av prototypen tar ikke hensyn til sanntid, men for videre utvikling av prototypen er dette interessant. Det er i denne sammenhengen viktig å vite hvor lang tid det tar å starte og migrere en analysekomponent på maskiner i nettverket.

Voyager og Java Virtuell Machine (JVM) bør avsluttes mellom hver gang testen utføres slik at det ligner mest mulig på en reell oppstart av systemet. Oppstart av en analysekomponent på en maskin der klassene er lastet i JVM bør også testes, for å kunne si noe om hvor lang tid det tar å laste klassene. Skalering av systemet bør også vurderes i den grad det er mulig, siden komplekse operasjoner bør kunne utføres av DMJ applikasjoner.

5.1.3 Fleksibilitet og rekonfigurering

Prototypen har ikke støtte for endring av media i DMJ Agenten siden denne komponenten ikke er utviklet, men støtter rekonfigurering av algoritmene som brukes. Fleksibilitet støttes derfor ikke fullt i prototypen, noe som er et krav til rammeverket. Dette er en svakhet med prototypen, og vil være et punkt for forbedring når flere av komponentene i rammeverket utvikles. JMF støtter derimot at media endrer rammerate og kvalitet (for eksempel *quality* i JPG eller H.261) under prosesseringen. Det er ikke egnet med en test på fleksibilitet utover rekonfigurering av komponentene, siden dette ikke er støttet i prototypen.

Kravet om rekonfigurering støttes av prototypen ved at parametere i komponenten kan endres under run-time. I forhold til videre utvikling av prototypen for å støtte sanntid er det viktig å kunne fastsette hvor lang tid det tar å rekonfigurere komponentene. Det er interessant å se på forskjellige måter å rekonfigurere analysekomponentene.

5.1.4 Oppsummering og utforming av testene

Vurderingen av kravene til DMJ arkitekturen viser at testene som kan utføres av prototypen tar utgangspunktet i at prototypen senere skal støtte sanntid. En fornuftig test av skalerbarhet kan være å la et stort antall PD'er og CD'er kommunisere med hverandre over MBus, slik at eventuelle kommunikasjons-problemer kan avdekkes. Denne testen tas ikke med her, men MBus er teoretisk vurdert kapittel 3.2.5. Det er også mulig å vurdere hvordan skalerbarhet kan støttes i prototypen ved å analysere resultatene for distribusjon og migrering nærmere. Testene som ble utført av prototypen er beskrevet under.

Test 1. PD'en ble startet opp på maskin B hver gang. Voyager ble startet og stoppet på alle maskinene mellom hver utføring.

Test 2. PD'en ble startet opp på maskin B hver gang. Hjelpeprogrammet *Shutdown* ble brukt for å fjerne PD'en mellom hver utføring.

Test 3. PD'en ble flyttet fra maskin C til maskin B. Voyager ble startet og stoppet på alle maskinene mellom hver utføring.

Test 4. PD'en ble flyttet mellom maskin C og maskin B.

Test 5. *ShotDetector* algoritmen som ble utflørt i PD'en ble rekonfigurert til å vise/ikke vise debug-informasjon.

Test 6. *ShotDetector* algoritmen som ble utflørt i PD'en ble rekonfigurert til en annen terskelverdi.

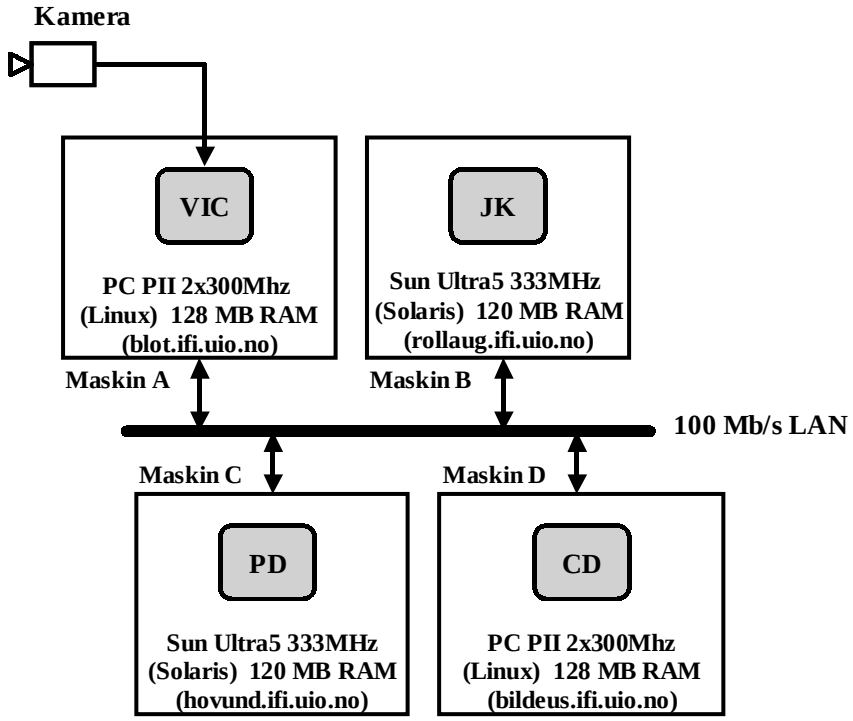
Test 7. PD'en ble rekonfigurert til å vise/ikke vise debug-informasjon.

Det anses som en rimelig antall å utføre testene 10 ganger for å kunne beregne gjennomsnittsverdier fra utføringene. Dette vil danne et bilde av tiden de forskjellige operasjonene tar. I test 1 og 2 ble tiden tatt fra PD'en ble laget (fra konstruktøren ble kalt) til den var i gang med analysen av videostrømmen. Tiden i test 3 og 4 ble tatt fra *move* ble kalt i PD'en, til den fikk et event tilbake om at komponenten hadde fullført migreringen. PD'en prøver å hente dette eventet hvert 10 millisekund (*polling*), så tiden kan i værste tilfelle være 10 millisekunder lavere. I test 5-7 tas tiden fra *reconfigure* blir kalt i PD'en, til denne metoden er ferdig utført. Alle tider er målt i millisekunder (ms).

Maskinene som ble brukt til testingen ble ikke benyttet til andre oppgaver, og nettverket var meget lite belastet når testene ble gjennomført. Det var maskin B og C som ble benyttet i testene og har installert Solaris operativsystem. Det ble kun benyttet telnet-innlogging (ikke vindussystem) mot disse maskinene, for å benytte mest mulig av prosessoren til testene.

5.2 Testomgivelse

Det er viktig å beskrive omgivelsen for testingen av prototypen, slik at testene settes i perspektiv. En testomgivelse forenkler også testingen, fordi man vet nøyaktig hvilke maskiner og nettverk som brukes, og kan plassere ut komponentene best mulig. Maskinene og nettverket som ble brukt er beskrevet i Figur 5-1.

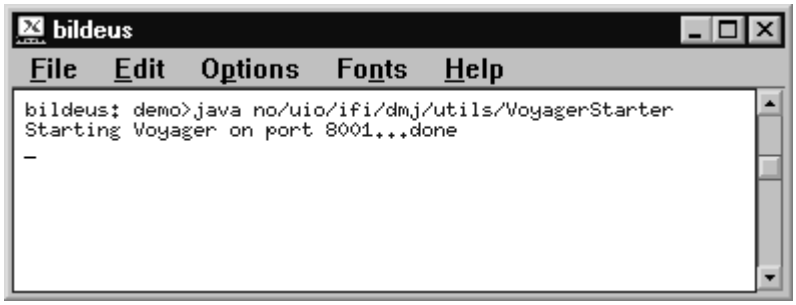


Figur 5-1 Testomgivelse for ytelsestest

Testen bruker komponentene Journalering Kontroller (JK), Primitiv og sammensatt hendelses detektor (henholdsvis CD og PD), Eventbroker og Ressursforvalter. Maskin A har et kamera tilkoblet som overvåker en korridor, og VIC startes på denne maskinen (Figur 5-2). VIC genererer RTP trafikk av video fra kameraet, og sender den til en IP multicast adresse. JK blir startet på maskin B, og via JK plasseres ut en PD på maskin C og en CD på maskin D. PD'en skal hente videoen fra multicast adressen, og analysere den med JMF. Hvis det er bevegelse i videoen, vil PD'en sende ut et event via eventbrokeren til alle CD'er. CD'en har registrert en deltaker i eventbrokeren, og får eventet tilsendt når det blir tilgjengelig. CD'en gjør ikke annet enn å skrive ut innholdet av eventet i denne implementasjonen.

Figur 5-2 Oppsett av VIC på maskin A

Testomgivelsen bruker fire maskiner, der maskin A er den eneste analysekomponentene ikke kan bruke (vil ikke bruke denne på grunn av kameraet). For å bruke de andre maskinene i nettverket må Voyager startes opp, som vist i Figur 5-3. Dette trenger man ikke gjøre på maskin B hvor JK skal kjøres, fordi JK starter Voyager selv (Figur 5-4). I *no.uio.ifi.dmj.utils* pakken finnes hjelpeprogrammer som blir brukt for å starte Voyager og starte, fjerne, rekonfigurere og flytte analysekomponenter (se Appendix A, side 108).



Figur 5-3 Voyager har startet

Figur 5-4 Journalering Kontroller har startet

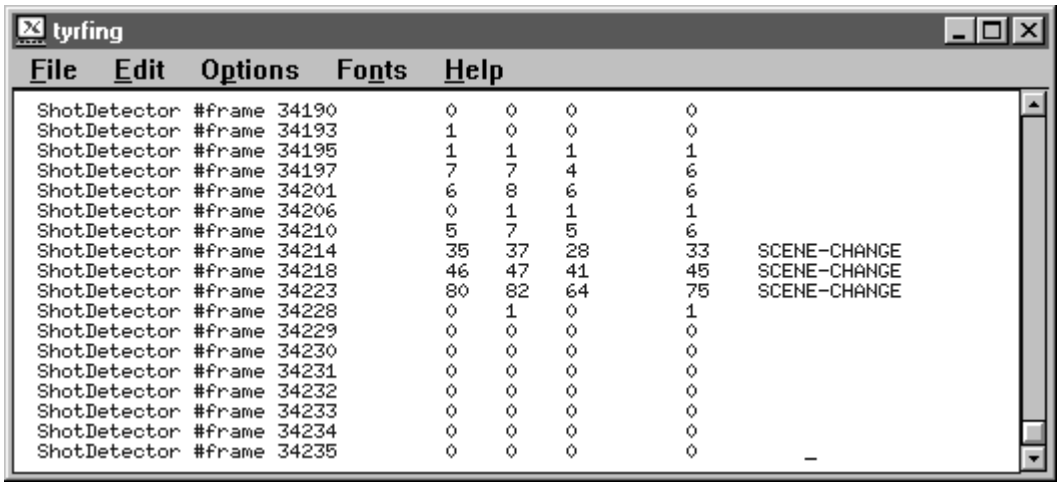
Etter å ha startet Voyager på alle maskinene og JK er i gang, kan analysekomponentene startes. Dette vises i Figur 5-5 hvor en PD startes opp

med noen parametere ved å bruke JK. Den første parameteren er en miljøvariabler (-D foran) som blir brukt til sikkerhet i Java RMI. Parameter nummer to brukes til å definere hvilken maskin JK komponenten kjøres på. På samme måte kan CD komponenten startes med hjelpeprogrammet *CDStarter*.

Figur 5-5 Primitiv Hendelses Detektor har startet

Journaleringen er i gang når analysekomponentene er utplassert med JK komponenten på de spesifiserte maskinene i nettverket. JMF analyserer video som kommer fra VIC, og blir det bevegelse i korridoren (Figur 5-6), så blir et event sendt via eventbrokeren til CD'en. Figur 5-7 viser det PD'en skriver ut av debug-informasjon når det er noen som går i gangen. CD'en vil kun skrive ut innholdet av eventet, som for eksempel er hvor stor forandring det er mellom forrige og det nåværende bildet i videoen.

Figur 5-6 Noen går i korridoren...



Figur 5-7 Slik ser det ut når PD'en oppdager hendelsen i gangen

Under run-time av prototypen kan analysekomponentene flyttes mellom de tre maskinene som starter Voyager (maskin B, C og D) med hjelpeprogrammet *Move*. Rekonfigurering av PD'en og CD'en utføres med *Reconfigure*. Begge disse programmene leter opp JK komponenten som gjør de nødvendige kallene i analysekomponenten.

5.3 Resultater

Tabell 5-1 viser resultatene fra testene som er beskrevet i kapittel 5.1.4. Tidene fra hver test er listet i hver sin kolonne, og gjennomsnittstiden står i bunnen. Testene er nummerert fra 1 til 10 i hver rad. Testene i hver rad er utført separert.

	Test 1	**Test 2**	**Test 3**	**Test 4**	**Test 5**	**Test 6**	**Test 7**
1	7524	497	7421	476	7	5	3
2	7418	607	7460	581	4	4	2
3	7445	646	7568	617	6	4	3
4	7450	552	7362	560	3	4	2
5	7447	701	7451	494	5	4	2
6	7444	525	7386	590	4	3	2
7	7571	585	7510	500	4	4	3
8	7852	485	7301	493	5	4	2
9	7334	597	7429	593	3	4	2
10	7403	455	7403	613	4	4	2
Gsn[1]	**7488.8**	**565**	**7429.1**	**551.7**	**4.5**	**4**	**2.3**

Tabell 5-1 Resultater fra testing av prototypen (i millisekunder)

PD'en ble lagret til fil før den ble flyttet slik at størrelsen på filen kunne leses av. Objektet ble lagret med vanlig serialisering i Java som Eksempel 5 viser. Størrelsen på objektet som ble flyttet var **1629 Byte**.

```
FileOutputStream f = new FileOutputStream ("obj.ser");
ObjectOutputStream s = new ObjectOutputStream (f);
s.writeObject (this);
```

Eksempel 5 Serialisering av objektet

5.4 Evaluering

Det utgjør stor forskjell om PD'en startes opp på en maskin som har lastet klassene i JVM, og en som ikke har kjørt en analysekomponent tidligere. Den store forskjellen i tid kan også begrunnes med *Just In Time* (JIT) kompilatoren, som gjør at operasjoner som er utført tidligere utføres mye raskere. Startes en PD på en maskin som ikke er benyttet til analysekomponenter tidligere, kan det ta opp i mot 7,5 sekunder før den er startet. Det er imidlertid ikke slik i de fleste tilfellene siden det er meningen at analysekomponentene skal flyttes rundt mellom maskinene. Resultatene viser at det vil ta lang tid for å starte opp systemet, men så snart alle maskinene har startet minst en analysekomponent tar resten av utplasseringen rimelig kort tid (under et halvt sekund pr komponent).

I forhold til skalerbarhet kan det være viktig å kunne starte mange analysekomponenter samtidig. I denne sammenhengen kan det være avgjørende om Voyager støtter synkrone eller asynkrone kall for å starte en analysekomponent. En enkel test ble gjennomført hvor fem PD'er ble startet tilnærmet samtidig der Journalering Kontroller (JK) opprettet alle komponentene parallelt. Dette viser at Voyager støtter asynkrone kall. Det er imidlertid et poeng at alle opprettelser gjøres fra JK komponenten, som kan bli en flaskehals hvis mange komponenter skal opprettes samtidig.

Flytting av en analysekomponent tar omtrent like lang tid som å starte opp en ny analysekomponent. Tiden det tar å bygge opp *codec-chain* i JMF, som man ikke trenger under en flytting, tar derfor omtrent like lang tid som å serialisere objektene som skal flyttes. Til sammen er det i beste tilfelle ca 20 ms raskere å flytte en PD mellom to maskiner (som har lastet klassene), enn å opprette en ny PD. Dette er med 10 ms feilmargin og utgjør ca 4 % av flyttingen eller opprettelsen. Besparingen kan virke liten, men når man jobber mot sanntid så er det viktig at tiden som brukes er minst mulig.

Rekonfigureringen av parametere i PD'en og i algoritmene som brukes tar som antatt kort tid. Det er utskifting av algoritmer som vil ta lang tid, og dette har jeg ikke funksjonalitet for i implementasjonen av prototypen. Operasjonen vil utgjøre å starte opp en PD som konfigureres med de nye parameterne, og

stoppe den gamle PD'en. Dette vil ta like lang tid som å starte en ny PD, og er dermed ikke støttet i implementasjonen.

Størrelsen på filen som flyttes er kun et eksempel på en filstørrelse på en PD med de gitte parameterne. Filen vil variere i størrelse etter hvor mange og størrelsen på algoritmene som blir satt inn i prosessen. Resultatene viser at det tar kortere tid å flytte PD'en enn å starte en ny, og det viser at serialiseringen ikke tok lang tid. Selve overføringen av 1629 Byte på et 100 Mbit nettverk, tar ubetydelig tid. Hadde filen blitt mye større og nettverket vesentlig dårligere, ville størrelsen hatt betydning. Det er ikke utenkelig at DMJ applikasjoner kan kjøres på 10 Mbit nettverk, men noe særlig lavere ville ikke egnet seg med tanke på overføring av videostrømmer.

5.5 Oppsummering

I dette kapittelet ble testingen av prototypen beskrevet og resultatene vist og diskutert. Prototypen ble testet i en omgivelse med fire PC'er, hvor to har installert Linux og to har Solaris operativsystem. Den ene maskinen har et kamera montert og VIC kjøres her for å sende ut RTP strømmen av videoen fra kameraet til en multicast adresse. På de tre andre maskinene ble analysekomponenter startet, konfigurert, migrert og rekonfigurert, ved å bruke hjelpeprogrammene i *no.uio.ifi.dmj.utils* pakken.

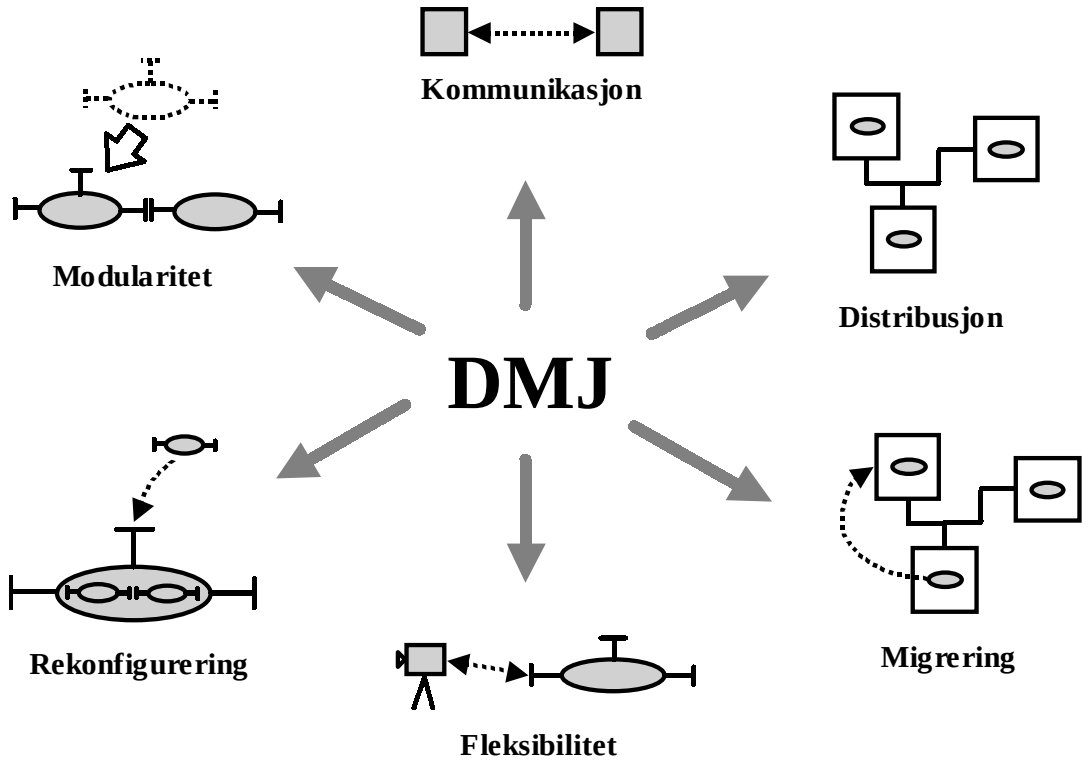
Resultatene viser at prototypen bruker forholdsvis lang tid på å starte opp analysekomponenter på maskiner som ikke er benyttet tidligere, fordi klassene ikke er lastet i JVM og siden JIT kompilatoren kun brukes når operasjoner utføres flere ganger. Opprettelse og flytting av PD'er til en maskin som har brukt en PD tidligere vil imidlertid ta under et halvt sekund. Tiden er kun 4 % lavere ved migrering av PD'en enn ved opprettelse av en ny PD, som ikke er en betryggende margin. Det er imidlertid store muligheter til forbedring hvis flere objekter kan serialiseres i analysekomponenten. Serialiseringen gjør at objektene ikke trenger å opprettes på nytt etter flyttingen, og dette har stor betydning for hurtigheten av migreringen. En mulighet for å unngå at klassene skal lastes idet en analysekomponent skal starte, kan være å laste alle klassene når Voyager startes opp på maskinen. Dette har ikke blitt testet i denne oppgaven.

Prototypen støtter også enkel rekonfigurering av PD komponenten, ved at parametere byttes ut mens prosesseringen pågår. Dette utføres på rimelig kort tid, siden det kun er parametere som endres i komponenten.

Kapittel 6

KONKLUSJON OG VIDERE ARBEID

I denne rapporten har en prototype blitt utviklet for å teste kravene som stilles til Distribuert Media Journalering (DMJ) arkitekturen. Kravene som DMJ prosjektet stiller til arkitekturen er kommunikasjon, distribusjon, migrering, modularitet, fleksibilitet og rekonfigurering av komponentene (Figur 6-1). Prototypen tar for seg noen vesentlige komponenter i DMJ arkitekturen, og vurderer hvordan design, teknologi og implementasjonen av prototypen støtter de nevnte kravene.



Figur 6-1 Krav til DMJ arkitekturen

Komponentene i prototypen utgjør analysekomponenter (PD, CD og Filter), Eventbroker og Journalering Kontroller fra DMJ arkitekturen. Designet av prototypen støtter kravet om kommunikasjon mellom analysekomponentene

med Eventbrokeren. Eventbrokeren er distribuert på alle maskinene i nettverket for å unngå at alle meldinger må innom en maskin i nettet, noe som lett blir en flaskehals i systemet. Analysekomponentene støtter kravet om distribusjon og migrering med metoden *move* i grensesnittet. De skal kommunisere med hverandre over eventbrokeren og er løst sammensatt. Dette vil si at de kan byttes ut under run-time og støtter med det kravet om modularitet. Eventbrokeren er fleksibel med hensyn på utvidelser siden nye klasser som definerer kommunikasjonsmodellen kan arve fra hovedklassen *DmjBroker*.

Komponentmodellen som støtter modularitet i DMJ arkitekturen er valgt til agentsystemet Voyager. Med denne modellen kan analysekomponentene byttes ut under run-time, noe som implementasjonen og testene av prototypen viser. Modellen gjør det også mulig for utviklere å benytte nye komponenter uten endringer i de eksisterende komponentene.

Analysekomponentene er løst sammenknyttet med eventbasert kommunikasjon. MBus benytter fordelene i IP multicast som er sentralt for god utnyttelse av nettet. Modellen inneholder funksjonalitet for å registrere alle deltakere som kommuniserer, slik at en hendelse kun blir sendt ut på nettet hvis det er noen som er interessert i den.

Migrering av komponentene mellom maskinene i nettverket gjøres med mobilitet egenskapene i Voyager Universal ORB. Implementasjonen bruker mobile agenter i distribusjonen av analyse komponenter i nettverket. Voyager krever at objektene i komponenten er serialiserbare for å kunne flytte dem. ORB implementasjonen er utviklet i Java, noe som gjør at modellen fungerer på flere operativsystemer. Modellen støtter i tillegg både kommunikasjons-modellene CORBA, DCOM og Java RMI. Implementasjonen av prototypen viser at Voyager egner seg godt til migrering av analysekomponenter. Resultatene fra testene som ble utført av prototypen viser imidlertid at migrering tar nesten like lang tid som å starte en ny komponent. Dette skyldes at JMF hindrer Voyager i å serialisere alle objektene.

Målet med denne rapporten var finne den best egnede komponentmodellen og kommunikasjonsmodellen til å støtte DMJ arkitekturen, samt å kunne migrere analysekomponenter mellom maskinene i nettverket. Som en oppsummering har DMJ komponentene i prototypen blitt designet og implementert med den valgte teknologien og viser gode resultater. Analysekomponentene har blitt implementert med komponentmodellen i Voyager som mobile agenter, noe som gjør at de kan flyttes mellom maskiner i nettverket. Eventbrokeren bruker multicast når meldinger skal sendes til flere mottakere og er fleksibel med hensyn på utvidelser.

6.1 Videre arbeid

Implementasjonen av prototypen har vært et lærerikt eksperiment for å teste forskjellige teknologier. Komponentteknologien Enterprise JavaBeans kunne imidlertid vært spennende å sett mer på. Det hadde vært interessant å sett hvordan EJB ville fungert i prototypen, og om de antagelsene som ble gjort om komponentmodellen stemmer. Objectspace Voyager kommer også som en

applikasjonsserver og støtter EJB. Voyager Application Server bygger på Voyager Universal ORB, og det hadde vært spennende å sett om de samme egenskapene med mobile agenter støttes i applikasjonsserveren. EJB er interessant fordi utvikling av komponenter støttes i diverse verktøyer som forenkler utviklingsprosessen.

Prototypen viser at migrering med Voyager fungerer bedre enn å starte en ny analysekomponent hver gang en komponent skal flyttes. Det er muligheter til å forbedre serialiseringen av objektene i analysekomponentene, slik at migrering lønner seg med en mye større margin. For eksempel serialiseres ikke prosesseringsobjektet i JMF som må opprettes på nytt etter flyttingen. Det finnes mange rammeverk som støtter prosessering av video, og det hadde vært interessant hvis et av dem kunne støtte serialisering av alle objektene som brukes. Dette ville utgjøre en formidabel forbedring av migreringen, siden det er starting av prosesseringen som tar mest tid (se kapittel 5.3).

Prototypen har ikke full støtte for fleksibilitet, siden det ikke er mulig å endre kvalitetsegenskaper til media som blir overført fra DMJ Agenten til analysekomponentene. Dette vil bli en utfordring i videre utvikling av prototypen og avgjørende for implementasjonen av DMJ Agenten. JMF støtter ikke at mediatypen eller oppløsningen endres under prosessering, derfor må alternative metoder vurderes. En mulighet er at nye PD'er startes opp hvis slike endringer inntreffer. Det ville vært nyttig å vurdere om andre rammeverk er mer fleksible i slike situasjoner enn JMF. Dette vil imidlertid si at analysen må opphøre en kort periode slik at objektet kan serialiseres. En vurdering av hvordan dette skal utføres må også tas stilling til.

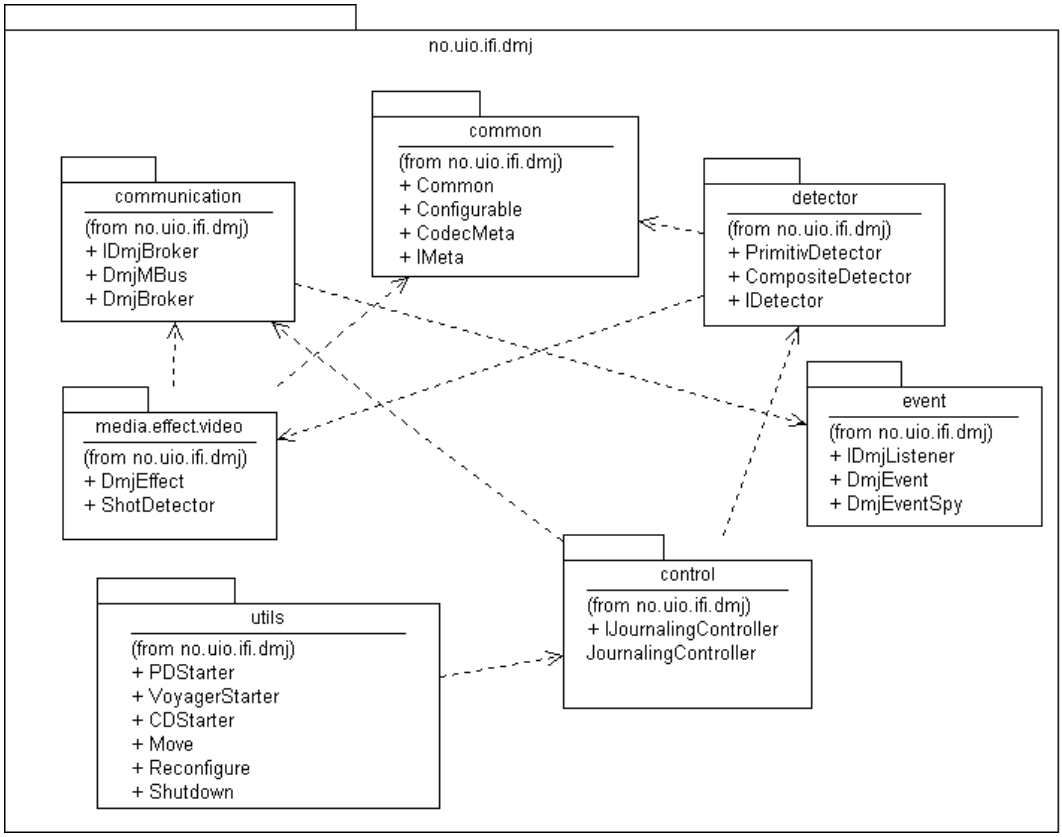
Den sammensatte analysekomponenten (CD) som ble implementert i prototypen er ikke ferdig utviklet. CD'ene skal implementere en modell for innholdsanalyse av media som baseres på *Bayes-nett* [21], *Hidden Markov Models* [37] eller tilsvarende modeller. Andre komponenter i DMJ rammeverket må også implementeres for å støtte kunstig intelligens og spesifisering av strategi. Dette innebærer Ressursforvalteren, Journalerings Spesifikasjon, Kunnskapsbasen, Generisk Plan Generator og Strategi Generator. Ressursforvalteren og Journalering Spesifikasjon komponentene er mest interessante for prototypen, siden disse direkte benytter Journalering Kontroller komponenten som prototypen har implementert et forslag til.

Prototypen støtter ikke sanntidsprosessering, siden ingen av operasjonene blir lovet ferdig innen en viss tid. Det er vanskelig å støtte sanntid så lenge operativsystemet ikke støtter reservering av ressurser. Derfor ville det ha vært interessant å benytte operativsystemer som støtter slik reservering. QLinux er et operativsystem hvor dette er mulig [48]. Operativsystemet støtter blant annet "fairly" allokering av CPU (Central Prosessing Unit) samt bedre kontroll med lagring av data (Chello disk scheduler) og bruk av nettverket (Lazy receiver).

DMJ rammeverket har behov for en måte å lagre media og metadata, slik at innholdet kan vises og analyseres på et senere tidspunkt. En multimedia server ville muligens løse dette hvor støtte for lagring av sanntids mediastrømmer finnes. Det hadde vært spennende å sett nærmere på hvilke løsninger som er tilgjengelige slik at kravene som DMJ stiller til en slik tjeneste blir støttet.

APPENDIX A - LISTING AV KODE

Her listes koden til implementasjonen av prototypen nummerert med navnet på pakkene. UML *package* diagrammet under viser en oversikt over alle pakkene i prototypen hvor relasjoner mellom klassene i pakkene vises med stiplede piler.



1 no.uio.ifi.dmj.detector

IDetector.java

```
1   package no.uio.ifi.dmj.detector;
2
3   public interface IDetector {
4
5     public String move(String dest);
6     public void moved();
7     public void usage();
8     public void configure(String arg);
```

```
9     public void reconfigure(String arg);
10    public void shutdown();
11    public void go();
12
13  } // IDetector.java ends here.
```

PrimitivDetector.java

```
1   package no.uio.ifi.dmj.detector;
2
3   import java.util.*;
4   import java.io.*;
5   import java.awt.Frame;
6   import java.awt.Component;
7   import javax.media.*;
8   import javax.media.control.TrackControl;
9   import javax.media.format.*;
10  import no.uio.ifi.dmj.common.*;
11  import no.uio.ifi.dmj.event.*;
12  import no.uio.ifi.dmj.communication.*;
13  import no.uio.ifi.dmj.media.effect.video.*;
14  import com.objectspace.voyager.agent.*;
15  import com.objectspace.voyager.mobility.*;
16  import com.objectspace.lib.timer.*;
17
18  public class PrimitivDetector
19         implements  IDetector,
20                     ControllerListener,
21                     Configurable,
22                     Serializable,
23                     IMobile {
24
25    private Processor p = null;
26    private boolean stateTransitionOK = true;
27    private Frame frame = null;
28    private String rate  = "1.0";
29    private String audio = "yes";
30    private String video = "yes";
31    private String visual = "no";
32    private String url = null;
33    private String contentType = "file";
34    private Vector codeclist = new Vector();
35    private Codec[] codecs= new Codec[100];
36    private int num_codec = 0;
37    private MediaLocator ml = null;
38    private String pathcodec =
39      "no.uio.ifi.dmj.media.effect.video.";
40    private Common c = new Common();
41    private boolean moved = false;
42    private boolean inAction = false;
43    private long timeInNano = 0;
44    private String currentLocation = null;
45    private String lastLocation = null;
46    private String newLocation = null;
47    private String msg = null;
48    private boolean eventReceived = false;
49    private Stopwatch stopwatch = new Stopwatch();
```

```
50    long start = 0;
51    long stop = 0;
52
53    public String move(String dest) {
54      if (moved && currentLocation.equals(dest))
55        return "You are already there!";
56      if (!moved) {
57        lastLocation = dest;
58        currentLocation = dest;
59      }
60      newLocation = dest;
61      c.debug_msg("\nPD is moving to "+dest+"\n");
62      if (inAction) {
63        ObjectKeeper.setObject(p, "processor "+hashCode());
64        if (frame!=null)
65          ObjectKeeper.setObject(frame, "frame "+hashCode());
66        p = null;
67        frame = null;
68      }
69      else
70        while (moved) {
71          try { Thread.sleep(20); } catch (Exception e) {}
72          if (inAction) {
73            p = null;
74            frame = null;
75            break;
76          }
77        }
78      try {
79        Agent.of(this).moveTo(dest, "moved");
80      } catch (Exception e) {
81        frame = (Frame)ObjectKeeper.getObject
82          ("frame "+hashCode());
83        ObjectKeeper.removeObject
84          ("frame "+hashCode());
85        p = (Processor)ObjectKeeper.getObject
86          ("processor "+hashCode());
87        ObjectKeeper.removeObject
88          ("processor "+hashCode());
89        return "Not OK!  \nERROR: "+e+"\n";
90      }
91      c.debug_msg("\nAfter moving...\n");
92
93      if (inAction) {
94        frame = (Frame)ObjectKeeper.getObject
95          ("frame "+hashCode());
96        p = (Processor)ObjectKeeper.getObject
97          ("processor "+hashCode());
98      }
99
100     if (moved && inAction) {
101       DmjEventSpy spy = new DmjEventSpy(false);
102       DmjBroker.addParticipant
103         ("PD_move","app:dmj component:PD type:moving",spy);
104       int ms = 0;
105       while ((spy.getEventText()==null) ||
106               (!(c.substringInText(spy.getEventText(),
107               "moving.pd (started)")))) {
108         ms += 10;
109         try { Thread.sleep(10); } catch (Exception e) {}
110         if (ms > 60000) break;
```

```
111       }
112       c.debug_msg("\nEvent (started) received\n");
113       p.stop();
114       inAction=false;
115       p=null;
116       if(frame!=null)frame.setVisible(false);
117       frame=null;
118       ObjectKeeper.removeObject("processor "+hashCode());
119       ObjectKeeper.removeObject("frame "+hashCode());
120       c.debug_msg("\nMoving PD completed!\n");
121       DmjBroker.removeParticipant("PD_move");
122     }
123     return "OK!";
124   }
125
126   public void setMoved(boolean b) {
127     c.debug_msg("Setting move to: "+b);
128     moved=b;
129   }
130
131   public void moved() {
132     c.debug_msg("\nPD has moved to "+newLocation+"\n");
133     moved = true;
134     lastLocation = currentLocation;
135     currentLocation = newLocation;
136     go();
137   }
138
139   public void shutdown() {
140     if (p!=null) p.stop();
141     p=null;
142     inAction=false;
143     if(frame!=null) {
144       frame.setVisible(false);
145       frame=null;
146     }
147     Agent.of(this).setAutonomous(false);
148     try { Thread.sleep(1000); } catch (Exception e) {}
149     c.debug_msg("\nThe PD has been shut down!\n");
150   }
151
152   public void usage(){
153     String myname = this.getClass().getName();
154     String options = "debug help novideo noaudio "+
155                      "rate= url= codec=";
156     String example = "java PrimitivDetector \"help "+
157                      "url=file:./bedtime_story.qt "+
158                      "rate=0.5 codec=ShotDetector(help)"+
159                      " codec=ColorFilter(help)\"";
160     c.usage(myname, options, example);
161   }
162
163   public boolean setKey(String key,String value,String arg)
164   {
165     if (key.equals("debug"))
166       c.setDebug(true);
167     else if (key.equals("help") ||
168              key.equals("--help") ||
169              key.equals("-h"))
170       usage();
171     else if (key.equals("noaudio"))
```

```
172       audio = "no";
173     else if (key.equals("visual"))
174       visual = "yes";
175     else if (key.equals("novideo"))
176       video = "no";
177     else if (key.equals("rate"))
178       rate = value;
179     else if (key.equals("url"))
180       url = value;
181     else if (key.equals("codec")) {
182       codeclist.add(value);
183       codeclist.add(arg);
184     }
185     else return false;
186     return true;
187   }
188
189   public void configure(String arg) {
190     if (!inAction)
191       c.parse((Configurable) this, arg);
192     else
193       reconfigure(arg);
194   }
195
196   public void reconfigure(String arg) {
197     if (inAction) {
198       start=stopwatch.getMilliseconds();
199       List l = c.parse(null, arg);
200       String d = c.getArg(l, "debug", false);
201       String na = c.getArg(l, "noaudio", false);
202       String nv = c.getArg(l, "novideo", false);
203       String v = c.getArg(l, "visual", false);
204       String r = c.getArg(l, "rate", false);
205       String co = c.getArg(l, "codec", false);
206       if (d!=null)
207         if (d.equals("yes")) c.setDebug(true);
208         else c.setDebug(false);
209       if (na!=null) {
210         TrackControl tc[] = p.getTrackControls();
211         TrackControl audioTrack = null;
212         for (int i = 0; i < tc.length; i++) {
213           if (tc[i].getFormat() instanceof AudioFormat) {
214               audioTrack = tc[i];
215               break;
216           }
217         }
218         if (audioTrack != null) {
219           audioTrack.setEnabled(false);
220         }
221       }
222       if (nv!=null) {
223         TrackControl tc[] = p.getTrackControls();
224         TrackControl videoTrack = null;
225         for (int i = 0; i < tc.length; i++) {
226           if (tc[i].getFormat() instanceof VideoFormat) {
227               videoTrack = tc[i];
228               break;
229           }
230         }
231         if (videoTrack != null) {
232           videoTrack.setEnabled(false);
```

```
233         }
234       }
235       if (v!=null) {
236         if (v.equals("yes")) {
237           Component vc;
238           frame = new Frame("DMJ-PD visual");
239           if ((vc = p.getVisualComponent()) != null) {
240               frame.add("Center", vc);
241           }
242           frame.setVisible(true);
243           frame.setSize(200,200);
244         }
245         else {
246           frame.setVisible(false);
247           frame=null;
248         }
249       }
250       if (r!=null) {
251         float timescale = p.setRate(Float.parseFloat(rate));
252         c.debug_msg("setRate() returned:" + timescale);
253       }
254       if (co!=null) {
255         String codecname = co.substring(0, co.indexOf("("));
256         String codecarg =
257           co.substring(co.indexOf("(")+1, co.indexOf(")"));
258         StringTokenizer tz = new StringTokenizer(codecarg);
259         IMeta meta = new CodecMeta();
260         while (tz.hasMoreTokens()) {
261           String token = tz.nextToken();
262           String x = token.substring(0,token.indexOf("="));
263           String y = token.substring(token.indexOf("=")+1);
264           meta.setValue(x,y);
265         }
266       }
267     stop=stopwatch.getMilliseconds();
268     System.out.println("moving PD: "+(stop-start));
269     }
270     stop=stopwatch.getMilliseconds();
271     System.out.println("moving PD: "+(stop-start));
272   }
273
274   public void go() {
275     c.debug_msg("Buiding the MediaLocator");
276     if (!inAction) {
277       if (url == null)
278         c.error_msg("Cannot build media locator!\n");
279       if ((ml = new MediaLocator(url)) == null)
280         c.error_msg("Cannot build media locator!\n");
281     }
282     c.debug_msg("Creating the processor");
283     try {
284       p = Manager.createProcessor(ml);
285     } catch (Exception e) {
286       c.error_msg("Failed to create a processor from "+
287                   "the given url: " + e);
288     }
289     p.addControllerListener(this);
290     c.debug_msg("Putting processor in configure-mode\n");
291     p.configure();
292     if (!waitForState(p.Configured))
293       c.error_msg("Failed to configure the processor!\n");
```

```
294     p.setContentDescriptor(null);
295     c.debug_msg("Obtain the track controls");
296     TrackControl tc[] = p.getTrackControls();
297     if (tc == null)
298       c.error_msg("Failed to obtain track controls "+
299                   "from the processor.");
300     TrackControl videoTrack = null;
301     c.debug_msg("Obtain the video track");
302     for (int i = 0; i < tc.length; i++) {
303       if (tc[i].getFormat() instanceof VideoFormat) {
304         videoTrack = tc[i];
305         break;
306       }
307     }
308     if (videoTrack == null) {
309       c.debug_msg("The input media does not contain a "+
310                   "video track.");
311     } else {
312       c.debug_msg("Video format: "+videoTrack.getFormat());
313       if (video.equals("no")) {
314         c.debug_msg("Disabling video track");
315         videoTrack.setEnabled(false);
316       }
317       else if (!inAction) {
318         c.debug_msg("Making the codecs");
319         Class cls = null;
320         Configurable configurable = null;
321         int i = 0;
322         while (i < ( codeclist.size() - 1) ) {
323           try {
324             cls = Class.forName(pathcodec+
325                         (String)codeclist.elementAt(i));
326           } catch (Exception e) {
327             c.error_msg(e.toString());
328           }
329           String arg = (String) codeclist.elementAt(i+1);
330           try {
331             configurable = (Configurable)cls.newInstance();
332             configurable.configure(arg);
333           } catch (Exception e) {c.error_msg(""+e);}
334           codecs[i / 2] = (Codec) configurable;
335           c.debug_msg("Inserting codec: "+configurable+
336                       " arg: "+arg);
337           i += 2;
338         }
339       }
340       c.debug_msg("Setting the codec-chain");
341       try {
342         videoTrack.setCodecChain(codecs);
343       } catch (Exception e) {c.error_msg(""+e);}
344     }
345     TrackControl audioTrack = null;
346     c.debug_msg("Obtain the audio track");
347     for (int i = 0; i < tc.length; i++) {
348       if (tc[i].getFormat() instanceof AudioFormat) {
349         audioTrack = tc[i];
350         break;
351       }
352     }
353     if (audioTrack == null) {
354       c.debug_msg("The input media does not contain "+
```

```
355                   "an audio track.");
356     } else {
357       c.debug_msg("Audio format: "+audioTrack.getFormat());
358       if (audio.equals("no")) {
359         c.debug_msg("Disabling audio track");
360         audioTrack.setEnabled(false);
361       }
362     }
363     c.debug_msg("Realizing the processor.");
364     p.realize();
365     if (!waitForState(p.Realized))
366       c.error_msg("Failed to realize the processor.");
367     float timescale;
368     timescale = p.setRate(Float.parseFloat(rate));
369     c.debug_msg("setRate() returned:" + timescale);
370     c.debug_msg("Prefetching");
371     p.prefetch();
372     if (!waitForState(p.Prefetched))
373       c.error_msg("Failed to prefetch.");
374     if ( visual.equals("yes")) {
375       c.debug_msg("Displaying the visual component\n");
376       Component vc;
377       frame = new Frame("DMJ-PD visual");
378       if ((vc = p.getVisualComponent()) != null) {
379         frame.add("Center", vc);
380       }
381       frame.setVisible(true);
382       frame.setSize(200,200);
383     }
384     c.debug_msg("Starting the processor");
385     p.start();
386     inAction = true;
387     if (moved && inAction) {
388       c.debug_msg("Sending started message to the old PD");
389       DmjEvent evt = new DmjEvent("moving.pd (started)");
390       String addr = "app:dmj component:PD type:moving";
391       (DmjBroker.getBroker("dummy")).sendEvent(evt, addr);
392     }
393   }
394
395   synchronized boolean waitForState(int state) {
396       try {
397         while (p.getState() != state && stateTransitionOK)
398           wait();
399       } catch (Exception e) {}
400
401     return stateTransitionOK;
402   }
403
404   public void controllerUpdate(ControllerEvent e) {
405     c.debug_msg("" + e);
406     if (e instanceof ConfigureCompleteEvent ||
407         e instanceof RealizeCompleteEvent ||
408         e instanceof PrefetchCompleteEvent) {
409       synchronized (this) {
410         stateTransitionOK = true;
411         notifyAll();
412       }
413     } else if (e instanceof ResourceUnavailableEvent) {
414       synchronized (this) {
415         stateTransitionOK = false;
```

```
416         notifyAll();
417       }
418     } else if (e instanceof EndOfMediaEvent) {
419       p.setMediaTime(new Time(0));
420       p.start();
421     } else if (e instanceof ControllerClosedEvent) {
422       c.debug_msg("Closed down\n");
423       System.exit(0);
424     } else {
425       c.debug_msg("Did not handle the event " + e);
426     }
427   }
428
429   public void preDeparture(String source,String destination)
430                                     throws MobilityException {
431     c.debug_msg("preDeparture: \nsource="+source+
432                 "\ndestination="+destination+"\n");
433   }
434
435   public void preArrival() throws MobilityException {
436     c.debug_msg("preArrival");
437   }
438
439   public void postArrival() {
440     c.debug_msg("postArrival");
441   }
442
443   public void postDeparture() {
444     c.debug_msg("postDeparture");
445   }
446
447 } // PrimitivDetector.java ends here.
448
```

CompositeDetector.java

```
1   package no.uio.ifi.dmj.detector;
2
3   import no.uio.ifi.dmj.event.*;
4   import no.uio.ifi.dmj.communication.*;
5   import no.uio.ifi.dmj.common.*;
6   import com.objectspace.voyager.agent.*;
7   import java.io.*;
8
9   public class CompositeDetector
10        implements IDetector, Serializable {
11    private String currentLocation = null;
12    private String lastLocation = null;
13    private String newLocation = null;
14    private DmjEventSpy spy = new DmjEventSpy();
15    private boolean moved = false;
16    private Common c = new Common();
17
18    public CompositeDetector () {}
19
20    public String move(String dest) {
21      if (moved && currentLocation.equals(dest))
22        return "You are already there!";
```

```
23      c.debug_msg("CD is moving to "+dest+"\n");
24      if (!moved) {
25        lastLocation = dest;
26        currentLocation = dest;
27      }
28      newLocation = dest;
29      try {
30        Agent.of(this).moveTo(dest, "moved");
31      } catch (Exception e)
32        {return "Not OK!  \nERROR: "+e+"\n";}
33      if (moved) {
34        DmjEventSpy event_rec = new DmjEventSpy(false);
35        DmjBroker.addParticipant
36          ("CD_moving", "app:dmj component:CD type:moving",
37          event_rec);
38        int ms = 0;
39        while ((spy.getEventText()==null) ||
40                (!(c.substringInText(spy.getEventText(),
41                   "dmj.cd.moving (started)")))) {
42          ms += 10;
43          try { Thread.sleep(10); } catch (Exception e) {}
44          if (ms > 60000) break;
45        }
46        c.debug_msg("\nEvent (started) received\n");
47        DmjBroker.removeParticipant("CD_moving");
48        DmjBroker.removeParticipant("CD");
49      }
50      return "OK!";
51    }
52
53    public void moved() {
54      c.debug_msg("CD has moved to "+newLocation+"\n");
55      moved = true;
56      lastLocation = currentLocation;
57      currentLocation = newLocation;
58      go();
59    }
60
61    public void usage() {
62      c.debug_msg("Usage: comming...");
63    }
64
65    public void configure(String args) {
66      if (c.substringInText(args, "debug")) {
67        c.setDebug(true);
68        c.debug_msg("Setting debug in CD: true");
69      }
70    }
71
72    public void reconfigure(String arg) {
73      c.debug_msg("Not implemented");
74    }
75
76    public void go(){
77      String addr = "app:dmj component:CD";
78      DmjBroker.addParticipant("CD", addr, spy);
79      DmjEvent evt = new DmjEvent("dmj.cd.moving(started)");
80      String addr_move = "app:dmj component:CD type:moving";
81      (DmjBroker.getBroker("CD")).sendEvent(evt, addr_move);
82      System.out.println("\nReady to receive events!\n");
83    }
```

```
84
85    public void shutdown() {
86      DmjBroker.removeParticipant("CD");
87      c.debug_msg("CD has shut down");
88    }
89
90  } // CompositeDetector.java ends here.
```

2 no.uio.ifi.dmj.communication

IDmjBroker.java

```
1   package no.uio.ifi.dmj.communication;
2
3   import no.uio.ifi.dmj.event.*;
4
5   public interface IDmjBroker {
6
7     public void sendEvent(DmjEvent e);
8     public void sendEvent(DmjEvent e, String addr);
9     public void sendEvent(DmjEvent e, AddressGroup g);
10    public void shutdown();
11
12  } // IDmjBroker.java ends here.
```

DmjBroker.java

```
13  package no.uio.ifi.dmj.communication;
14
15  import java.util.*;
16  import no.uio.ifi.dmj.event.*;
17
18  public abstract class DmjBroker implements IDmjBroker {
19    protected static List listeners = new ArrayList();
20    private static Map brokerList = new Hashtable();
21    private static String dummy = "app:dmj component:dummy";
22    private static String brokerType =
23      "no.uio.ifi.dmj.communication.DmjMBus";
24
25    public static void addParticipant(String id,
26                                      String addr,
27                                      IDmjListener newListener) {
28      Participant p = new Participant(id, newListener);
29      IDmjBroker broker = null;
30      Class cl = null;
31      try {
32        cl = Class.forName(brokerType);
33        broker = (IDmjBroker) cl.newInstance();
34      } catch(Exception e1) {
35        try {
36          cl = Class.forName
```

```
37            ("no.uio.ifi.dmj.communication.DmjMBus");
38          broker = (IDmjBroker) cl.newInstance();
39        } catch(Exception e2) {
40          e2.printStackTrace();
41          System.exit(0);
42        }
43      }
44      brokerList.put(id, broker);
45    }
46
47    public static void removeParticipant(String id) {
48      IDmjBroker broker = (IDmjBroker)brokerList.remove(id);
49      broker.shutdown();
50    }
51
52    public static void setBrokerType(String cl) {
53      brokerType = cl;
54    }
55
56    public static IDmjBroker getBroker(String id) {
57      Object out = brokerList.get(id);
58      if (out == null) {
59        addParticipant("dummy", dummy, null);
60        return (IDmjBroker)brokerList.get("dummy");
61      }
62      else
63        return (IDmjBroker)out;
64    }
65
66  } // DmjBroker.java ends here.
```

DmjMBus.java

```
1   package no.uio.ifi.dmj.communication;
2
3   import org.mbus.*;
4   import java.util.*;
5   import java.net.*;
6   import no.uio.ifi.dmj.event.*;
7
8   public class DmjMBus extends DmjBroker
9                        implements MBusListener {
10    Address addr = null;
11    private TransportLayer transport = null;
12    private String me = this.getClass().getName();
13    private String host = null;
14    private String id = null;
15    private Participant p = null;
16
17    public DmjMBus(Participant new_p, String new_addr) {
18      p = new_p;
19      try { host = InetAddress.getLocalHost().getHostName();}
20      catch(Exception e) {e.printStackTrace();}
21      id = TransportLayer.getID();
22      addr = new Address(new_addr);
23      try {
24        transport  = new SimpleTransportLayer(addr, this);
25      } catch (MBusException e) { e.printStackTrace(); }
```

```
26    }
27
28    public void sendEvent(DmjEvent e){
29      String msg = e.toString();
30      transport.broadcast(addr, makeCommand(msg));
31    }
32
33    public void sendEvent(DmjEvent e, String group){
34      String msg = e.toString();
35      Address sendTo = new Address(group);
36      transport.send(makeCommand(msg), addr, sendTo, false);
37    }
38
39    public void sendEvent(DmjEvent e, AddressGroup g){
40      String msg = e.toString();
41      String[] addrs = g.getAddresses();
42      for (int i=0; i<addrs.length; i++) {
43        if (addrs[i] != null) {
44          Address sendTo = new Address(addrs[i]);
45          transport.send(makeCommand(msg),addr,sendTo,false);
46        }
47      }
48    }
49
50    public void shutdown() {
51      transport.close();
52      System.out.println("transport closed");
53    }
54
55    // A little workaround...
56    private Command makeCommand(String s) {
57      StringBuffer sb = new StringBuffer(s);
58      int level = 0;
59      for (int i=0; i<sb.length(); i++) {
60        if (sb.charAt(i) == '=') {
61          sb.deleteCharAt(i); sb.insert(i, "ERLIK");      i += 4;
62        }
63        else if (sb.charAt(i) == ':') {
64          sb.deleteCharAt(i); sb.insert(i, "KOLON");      i += 4;
65        }
66        else if (sb.charAt(i) == '/') {
67          sb.deleteCharAt(i); sb.insert(i, "SLASH");      i += 4;
68        }
69        else if (sb.charAt(i) == '(') {
70          level++;
71          if (level>1) {
72            sb.deleteCharAt(i);
73            sb.insert(i, "PSTART");     i += 4;
74          }
75        }
76        else if (sb.charAt(i) == ')') {
77          if (level>1) {
78            sb.deleteCharAt(i);
79            sb.insert(i, "PSLUTT"); i += 4;
80          }
81          level--;
82        }
83      }
84      return new Command(sb.toString());
85    }
86
```

```
87    private String makeMessage(String s) {
88      StringBuffer sb = new StringBuffer(s);
89      for (int i=0; i<sb.length(); i++) {
90        if ((sb.length()>(i+5)) &&
91             sb.substring(i, i+5).equals("ERLIK")) {
92          sb.replace(i, i+5, "=");
93        }
94        else if ((sb.length()>(i+5)) &&
95                  sb.substring(i, i+5).equals("SLASH")) {
96          sb.replace(i, i+5, "/");
97        }
98        else if ((sb.length()>(i+6)) &&
99                  sb.substring(i, i+6).equals("PSTART")) {
100         sb.replace(i, i+6, "(");
101       }
102       else if ((sb.length()>(i+6)) &&
103                 sb.substring(i, i+6).equals("PSLUTT")) {
104         sb.replace(i, i+6, ")");
105       }
106       else if ((sb.length()>(i+5)) &&
107                 sb.substring(i, i+5).equals("KOLON")) {
108         sb.replace(i, i+5, ":");
109       }
110     }
111     return sb.toString();
112   }
113
114   public void incomingMessage(Message m)
115   {
116     Enumeration e = m.getCommands();
117     while(e.hasMoreElements()) {
118       Command com = (Command)e.nextElement();
119       String host = m.getHeader().getSourceAddress().
120                     getValue("component");
121       p.sendEvent(makeMessage(com.toString()));
122     }
123   }
124
125   public void deliveryFailed(int seqn, Message m) {}
126   public void deliverySuccessful(int seqn, Message m) {}
127   public void entityDied(Address a) {}
128   public void entityShutdown(Address a) {}
129   public void newEntity(Address a) {}
130
131 } // DmjMBus.java ends here.
132
```

Participant.java

```
1   package no.uio.ifi.dmj.communication;
2
3   import java.util.*;
4   import no.uio.ifi.dmj.event.*;
5
6   public class Participant {
7     private String id = null;
8     private IDmjListener listener = null;
9
```

```
10    public Participant(String i, IDmjListener l) {
11      id = i;
12      listener = l;
13    }
14
15    public void sendEvent(String e) {
16      if (listener != null )
17        listener.DmjEventReceived(new DmjEvent(e));
18    }
19
20    public String getId() {
21      return id;
22    }
23
24  } // Participant.java ends here.
```

3 no.uio.ifi.dmj.control

IJournalingController.java

```
1   package no.uio.ifi.dmj.control;
2
3   public interface IJournalingController {
4
5     public void configure(String args);
6     public void startCD(String arg);
7     public void startPD(String arg);
8     public void move(String arg);
9     public void reconfigure(String arg);
10    public void shutdown(String arg);
11
12  } // IJournalingController ends here.
13
```

JournalingController.java

```
1   package no.uio.ifi.dmj.control;
2
3   import no.uio.ifi.dmj.detector.*;
4   import no.uio.ifi.dmj.common.*;
5   import no.uio.ifi.dmj.communication.*;
6   import no.uio.ifi.dmj.event.*;
7   import com.objectspace.voyager.*;
8   import com.objectspace.voyager.agent.*;
9   import java.util.*;
10  import java.net.*;
11  import java.io.*;
12
13  public class JournalingController
14          implements IJournalingController,
15          Serializable {
```

```
16    private IDetector pd = null;
17    private IDetector cd = null;
18    private String path = "no.uio.ifi.dmj.detector.";
19    private Hashtable detectors = new Hashtable();
20    private Common c = new Common();
21
22    public void configure(String args) {
23      if (c.substringInText(args, "debug")) {
24        c.setDebug(true);
25      }
26    }
27
28    public void startCD(String arg) {
29      List arglist = c.parse(null, arg);
30      String tmp = c.getArg(arglist, "dest", true);
31      boolean local = c.substringInText(tmp,"localhost");
32      String dest = fixAddr(tmp);
33      String name = c.getArg(arglist, "name", true);
34      String conf = null;
35      if (detectors.containsKey(name)) {
36        System.out.println("\nThe name is already used!"+
37                           "Use annother one!");
38        return;
39      }
40      StringBuffer sb = new StringBuffer();
41      Iterator it = arglist.iterator();
42      while (it.hasNext())
43        sb.append(" "+(String)it.next());
44      conf = sb.toString();
45      try {
46        System.out.print("Creating CD ...");
47        cd = (IDetector)
48          Factory.create(path+"CompositeDetector");
49        detectors.put(name, cd);
50        System.out.println("ok!");
51        cd.configure(conf);
52        if (local) {
53          c.debug_msg("Localhost - not moving\n");
54          cd.go();
55        }
56        else cd.move(dest);
57      }
58      catch(Exception exception) {
59        System.err.println( "Error: Could not start CD." );
60        detectors.remove(name);
61      }
62    }
63
64    public void startPD(String arg) {
65      List arglist = c.parse(null, arg);
66      String tmp = c.getArg(arglist, "dest", true);
67      boolean local = c.substringInText(tmp,"localhost");
68      String dest = fixAddr(tmp);
69      String name = c.getArg(arglist, "name", true);
70      String conf = null;
71
72      if (detectors.containsKey(name)) {
73        System.out.println("\nThe name is already used!"+
74                           "Use annother one!");
75        return;
76      }
```

```
77      StringBuffer sb = new StringBuffer();
78      Iterator it = arglist.iterator();
79      while (it.hasNext())
80        sb.append(" "+(String)it.next());
81      conf = sb.toString();
82      try {
83        System.out.print("\n\nCreating PD ...");
84        pd = (IDetector)
85          Factory.create(path+"PrimitivDetector");
86        detectors.put(name, pd);
87        System.out.println("ok!");
88        pd.configure(conf);
89        if (local) {
90          c.debug_msg("Localhost - not moving\n");
91          pd.go();
92        }
93        else System.out.println(pd.move(dest));
94      }
95      catch(Exception exception) {
96        System.err.println( "Error: Could not start PD.");
97        exception.printStackTrace();
98        detectors.remove(name);
99      }
100   }
101
102   public void reconfigure(String arg) {
103     List arglist = c.parse(null, arg);
104     String name = c.getArg(arglist, "name", true);
105     String conf = null;
106     StringBuffer sb = new StringBuffer();
107     Iterator it = arglist.iterator();
108     while (it.hasNext())
109       sb.append(" "+(String)it.next());
110     conf = sb.toString();
111     IDetector detector = findObject(name);
112     if (detector != null) {
113       detector.reconfigure(conf);
114     }
115     else System.err.println("Could not find the "+
116                             "spesified detector!");
117   }
118
119   public void move(String arg) {
120     List arglist = c.parse(null, arg);
121     String dest = fixAddr(c.getArg(arglist, "dest", true));
122     String name = c.getArg(arglist, "name", true);
123     c.debug_msg("\nMoving "+name+" to "+dest+" ... ");
124     IDetector detector = findObject(name);
125     if (detector != null) {
126       String ans = detector.move(dest);
127       c.debug_msg(ans);
128     }
129     else System.err.println("Could not find the "+
130                             "spesified detector!");
131   }
132
133   public void shutdown(String arg) {
134     List arglist = c.parse(null, arg);
135     String name = c.getArg(arglist, "name", true);
136     c.debug_msg("\nShutting down "+name+" ... ");
137     IDetector detector = findObject(name);
```

```
138     if (detector != null) {
139       detector.shutdown();
140       detectors.remove(name);
141       c.debug_msg("done!\n");
142     } else
143       System.err.println("Could not find the "+
144                          "spesified detector!");
145   }
146
147   private IDetector findObject(String id) {
148     return (IDetector)detectors.get(id);
149   }
150
151   private String fixAddr(String addr) {
152     String tmp = addr;
153     if (addr!=null) {
154       if (!(tmp.startsWith("//"))) tmp = "//"+tmp;
155       if (!(c.substringInText(tmp, ":"))) tmp = tmp+":8001";
156     }
157     return tmp;
158   }
159
160 } // JournalingController.java ends here.
161
```

4 no.uio.ifi.dmj.common

Common.java

```
1   package no.uio.ifi.dmj.common;
2
3   import gnu.regexp.*;
4   import java.io.*;
5   import java.util.*;
6
7   public class Common implements Serializable {
8     private boolean debug = false;
9
10    public String getArg(List args, String key,
11                         boolean remove) {
12      Object o = null;
13      for (int i=0; i<args.size(); i++) {
14        if (key.equals("debug") ||
15            key.equals("noaudio") ||
16            key.equals("novideo")) {
17          if (((String)args.get(i)).startsWith(key)) {
18            if (remove) o=args.remove(i);
19            return key;
20          }
21        }
22        else if (((String)args.get(i)).startsWith(key))
23          if (remove)
24            return ((String)
25              args.remove(i)).substring(key.length()+1);
```

```
26          else
27            return ((String)
28              args.get(i)).substring(key.length()+1);
29      }
30      return null;
31    }
32
33    public boolean substringInText(String text,
34                                   String sub) {
35      int i = 0;
36      boolean ret = false;
37      if (text==null || sub==null ||
38         (text.length()< sub.length()))
39        return false;
40      while (i < (text.length() - sub.length() + 1)) {
41        if (text.startsWith(sub, i)) {
42          ret = true;
43          break;
44        }
45        i++;
46      }
47      return ret;
48    }
49
50    public void setDebug(boolean b)     {
51      debug = b;
52    }
53
54    public boolean getDebug()     {
55      return debug;
56    }
57
58    public  void error_msg(String s)    {
59      System.err.print("\nError, " + s + "\n");
60      Thread.currentThread().dumpStack();
61      System.exit(1);
62    }
63
64    public void debug_msg(String s)     {
65      if (debug) System.out.print("\n" + s);
66    }
67
68    public void usage(String name,
69                      String options,
70                      String example)   {
71      System.out.print("\n\n");
72      System.out.println("Usage:");
73      System.out.println("Class:  " + name);
74      System.out.println("Options: " + options);
75      System.out.println("Example: " + example);
76      System.out.print("\n");
77    }
78
79    public List parse(Configurable caller, String s)      {
80      StringBuffer sb = balance(s);
81      RE re = null;
82      List l = new ArrayList();
83      try {
84        re = new RE
```

```
85
     ("((\\w+)\\s+)|((\\w+)=([\\w\\.:/_]+)\\s+)|((\\w+)=([\\w\\.:/
   _]+)\\{(.*?)\\}\\s+)");
86     } catch (REException e) { error_msg("\n"+e); }
87     REMatchEnumeration enum = re.getMatchEnumeration(sb);
88     while (enum.hasMoreElements()) {
89       REMatch r = enum.nextMatch();
90       String key = null, value = null, arg = null;
91       if (!r.toString(1).equals("")) {
92         key = r.toString(2);
93       } else if (!r.toString(3).equals("")) {
94         key = r.toString(4);
95         value = r.toString(5);
96       } else if (!r.toString(6).equals("")) {
97         key = r.toString(7);
98         value = r.toString(8);
99         arg = r.toString(9);
100      } else {
101        error_msg("\nShould not get here.");
102      }
103      boolean ret = true;
104      if (caller!=null) ret = caller.setKey(key,value,arg);
105      else {
106        if (value==null) l.add(key);
107        else if (arg==null) l.add(key+"="+value);
108        else l.add(key+"="+value+"("+arg+")");
109      }
110      debug_msg("key value arg: | "+key+" | "+
111                value+" | "+arg+" | ");
112      if (!ret)
113        error_msg("Unrecognized key: " + key);
114    }
115    return l;
116  }
117
118  private StringBuffer balance(String s){
119    int level = 0;
120    StringBuffer sb = new StringBuffer(s);
121    sb.insert(0, " ");
122    sb.insert(sb.length(), " ");
123    for (int i=0; i < sb.length(); i++) {
124      switch (sb.charAt(i)) {
125      case '{':
126      case '}':
127        error_msg("'{' and '}' used as metachars, "+
128                  "not allowed in input.");
129      case '(':
130        if (level == 0) sb.setCharAt(i, '{');
131        level++;
132        break;
133      case ')':
134        level--;
135        if (level == 0) sb.setCharAt(i, '}');
136        break;
137      default:
138        break;
139      }
140    }
141    if (level != 0) error_msg("Not balanced "+
142                        "parenthesis in input");
143    return sb;
```

```
144   }
145
146 } // Common.java ends here.
```

ObjectKeeper.java

```
1   package no.uio.ifi.dmj.common;
2
3   import java.util.*;
4
5   public class ObjectKeeper {
6     private static Hashtable array = new Hashtable();
7     public ObjectKeeper() {}
8
9     public static void setObject(Object ref,
10                                 String hashValue) {
11      array.put((hashValue), ref);
12    }
13
14    public static Object getObject(String hashValue) {
15      return array.get(hashValue);
16    }
17
18    public static void removeObject(String hashValue) {
19      if (array.containsKey(hashValue))
20        array.remove(hashValue);
21    }
22
23  } // ObjectKeeper.java ends here.
24
```

IMeta.java

```
1   package no.uio.ifi.dmj.common;
2
3   public interface IMeta{
4
5     public void setValue(String key, String value);
6
7   } // IMeta.java ends here.
```

CodecMeta.java

```
1   package no.uio.ifi.dmj.common;
2
3   import java.util.*;
4   import java.io.*;
5
6   public class CodecMeta implements IMeta, Serializable{
7
8     private static Hashtable objects = new Hashtable();
```

```
9     private static Common c = new Common();
10
11    public static void setV(String key, String value) {
12      objects.put(key, value);
13      c.debug_msg("Setting pair: "+key+" "+value);
14    }
15
16    public void setValue(String key, String value) {
17      setV(key,value);
18    }
19
20    public static String getString(String key) {
21      return (String)objects.get(key);
22    }
23
24    public static double getDouble(String key) {
25      String tmp = (String)objects.get(key);
26      double d = 0;
27      try {
28        d = Double.parseDouble(tmp);
29      } catch(Exception e) {e.printStackTrace(); System.exit(0);}
30      return d;
31    }
32
33    public static int getInt(String key) {
34      String tmp = (String)objects.get(key);
35      int i = 0;
36      try {
37        i = Integer.parseInt(tmp);
38      } catch(Exception e) {e.printStackTrace();}
39      return i;
40    }
41
42    public static boolean isEmpty() {
43      return objects.isEmpty();
44    }
45
46  }
47
```

Configurable.java

```
1   package no.uio.ifi.dmj.common;
2
3   public interface Configurable{
4
5     public void usage();
6     public void configure(String arg);
7     public boolean setKey(String key,String value,String arg);
8
9   } // Configurable.java ends here.
```

5 no.uio.ifi.dmj.event

DmjEvent.java

```
1   package no.uio.ifi.dmj.event;
2
3   import java.util.*;
4   import java.io.*;
5
6   public class DmjEvent extends EventObject
7                         implements Serializable {
8     String theString;
9     public DmjEvent(Object source) {
10      super(source);
11    }
12    public DmjEvent(String source) {
13      super(source);
14      theString = source;
15    }
16    public String toString() {
17      return theString;
18    }
19
20  } // DmjEvent.java ends here.
21
```

IDmjListener.java

```
1   package no.uio.ifi.dmj.event;
2
3   public interface IDmjListener {
4     public void DmjEventReceived(DmjEvent e);
5
6   } // IDmjEventListener.java ends here.
```

DmjEventSpy.java

```
1   package no.uio.ifi.dmj.event;
2
3   import no.uio.ifi.dmj.communication.*;
4   import java.io.*;
5
6   public class DmjEventSpy implements IDmjListener,
7                                       Serializable {
8     String str = null;
9     boolean printIt = true;
10
11    public DmjEventSpy() {}
12    public DmjEventSpy(boolean b) {printIt = b;}
13
14    public void DmjEventReceived(DmjEvent e) {
15      if (printIt) System.out.println(e);
16      str = e.toString();
17    }
18
```

```
19    public String getEventText() {
20      return str;
21    }
22
23  } // EmjEventSpy.java ends here.
24
```

6 no.uio.ifi.dmj.media.effekt.video

DmjEffekt.java

```
1   package no.uio.ifi.dmj.media.effect.video;
2
3   import javax.media.*;
4   import javax.media.format.*;
5   import javax.media.format.RGBFormat;
6   import no.uio.ifi.dmj.common.*;
7
8   public class DmjEffect implements Effect, Configurable {
9
10    protected Format input = null, output = null;
11    protected int rMask = 0x000000FF;
12    protected int gMask = 0x0000FF00;
13    protected int bMask = 0x00FF0000;
14    protected int width, height;
15    protected int depth;
16    protected Common c = new Common();
17    protected Format[] supportedOuts =
18      new Format [] { new RGBFormat() };
19
20    protected Format supportedIns[] = new Format [] {
21      new RGBFormat(null,             // size
22              Format.NOT_SPECIFIED,   // maxDataLength
23              int[].class,            // buffer type
24              Format.NOT_SPECIFIED,   // frame rate
25              32,                     // bitsPerPixel
26              rMask,                  // red component mask
27              gMask,                  // green component mask
28              bMask,                  // blue component mask
29              1,                      // pixel stride
30              Format.NOT_SPECIFIED,   // line stride
31              Format.FALSE,           // flipped
32              Format.NOT_SPECIFIED) };// endian
33
34    public void usage() {
35      String  myname = this.getName();
36      String  options = "debug help";
37      String  example = new String(myname + "(help)");
38      c.usage(myname, options, example);
39    }
40
41    public boolean setKey(String key, String value, String arg) {
42      if (key.equals("debug"))
43        c.setDebug(true);
```

```
44      else if (key.equals("help") || key.equals("--help") ||
45               key.equals("-h")) { usage(); }
46      else return false;
47      return true;
48    }
49
50    public void configure(String arg)   {
51      c.parse((Configurable) this, arg);
52    }
53
54    public void accessFrame(Buffer frame) {}
55    public String getName() {return "DmjEffect";}
56    public void open()  {}
57    public void close()  {}
58    public void reset()  {}
59
60    public Format [] getSupportedInputFormats() {
61      return supportedIns;
62    }
63
64    public Format [] getSupportedOutputFormats(Format in) {
65      if (in == null) return supportedOuts;
66      else {
67        Format outs[] = new Format[1];
68        outs[0] = in;
69        return outs;
70      }
71    }
72
73    public Format setInputFormat(Format format)     {
74      input = format;
75      RGBFormat fmt = (RGBFormat) format;
76      depth = fmt.getBitsPerPixel();
77      width = fmt.getSize().width;
78      height = fmt.getSize().height;
79      return input;
80    }
81
82    public Format setOutputFormat(Format format) {
83      output = format;
84      return output;
85    }
86
87    public int process(Buffer in, Buffer out) {
88      accessFrame(in);
89      Object data = in.getData();
90      in.setData(out.getData());
91      out.setData(data);
92      int inflag = in.getFlags();
93      int outflag= inflag;
94      out.setFlags(outflag);
95      out.setFormat(in.getFormat());
96      out.setLength(in.getLength());
97      out.setOffset(in.getOffset());
98      return BUFFER_PROCESSED_OK;
99    }
100
101   public Object[] getControls() {return new Object[0];}
102   public Object getControl(String type) {return null;}
103
104 } // DmjEffect.java ends here.
```

ShotDetector.java

```
1   package no.uio.ifi.dmj.media.effect.video;
2
3   import java.io.*;
4   import javax.media.*;
5   import javax.media.format.*;
6   import no.uio.ifi.dmj.common.*;
7   import no.uio.ifi.dmj.communication.*;
8   import no.uio.ifi.dmj.event.*;
9
10  public class ShotDetector
11          extends DmjEffect
12          implements Serializable {
13
14    private int nr_prev = 0, ng_prev = 0, nb_prev = 0;
15    private Common c = new Common();
16    private IMeta meta = new CodecMeta();
17    private double threshold = 50;
18    private double indication = 0.05;
19    private String debug = "no";
20
21    public void usage()  {
22      String  myname = this.getName();
23      String  options = "debug help indication= threshold=";
24      String  example = new String(myname +
25        "(debug threshold=50 indication=0.05)");
26      c.usage(myname, options, example);
27    }
28
29    public boolean setKey(String key,
30                          String value,
31                          String arg)  {
32      if (key.equals("debug"))
33        debug = "yes";
34      else if (key.equals("help") || key.equals("--help") ||
35                 key.equals("-h"))  usage();
36      else if (key.equals("threshold"))
37        threshold=Double.parseDouble(value);
38      else if (key.equals("indication"))
39        indication=Double.parseDouble(value);
40      else return false;
41      return true;
42    }
43
44    public String getName() {
45      return "ShotDetector";
46    }
47
48    public void accessFrame(Buffer frame)
49    {
50      if (CodecMeta.isEmpty()) {
51        meta.setValue("threshold", ""+threshold);
52        meta.setValue("indication", ""+indication);
53        meta.setValue("debug", debug);
54      }
55      threshold = CodecMeta.getDouble("threshold");
```

```
56      indication = CodecMeta.getDouble("indication");
57      debug = CodecMeta.getString("debug");
58      if (debug.equals("yes"))
59        c.setDebug(true);
60      else c.setDebug(false);
61
62
63      StringBuffer out = new StringBuffer
64        ("                                    ");
65      int[] buf;
66
67      out.insert(1, "" + this.getName() + " #frame " +
68        frame.getSequenceNumber());
69      java.lang.Object o = frame.getData();
70
71      if (o instanceof int[]) {
72        buf = (int[]) o;
73        int nr = 0, ng = 0, nb = 0;
74        for (int i=0; i < buf.length; i++)  {
75          nr += (buf[i] & 0X000000FF) ;
76          ng += (buf[i] & 0X0000FF00) >>  8;
77          nb += (buf[i] & 0X00FF0000) >> 16;
78        }
79        int nr_diff = 0, ng_diff = 0, nb_diff = 0;
80        double ch = 0.0, r_ch = 0.0, g_ch = 0.0, b_ch = 0.0;
81        nr_diff = Math.abs(nr_prev - nr);
82        if (nr_prev != 0)
83        r_ch = (nr_diff * 100.0) / nr_prev;
84        ng_diff = Math.abs(ng_prev - ng);
85        if (ng_prev != 0)
86        g_ch = (ng_diff * 100.0) / ng_prev;
87        nb_diff = Math.abs(nb_prev - nb);
88        if (nb_prev != 0)
89        b_ch = (nb_diff * 100.0) / nb_prev;
90        ch = (r_ch + g_ch + b_ch) / 3;
91        out.insert(35, (int) b_ch);
92        out.insert(40, (int) g_ch);
93        out.insert(45, (int) r_ch);
94        out.insert(55, (int) ch);
95        if (r_ch > threshold || g_ch > threshold ||
96            b_ch > threshold ) {
97          out.insert(60, " SCENE-CHANGE");
98          if (indication > 0.0 ) {
99            int x = (int) (indication * height - 1);
100           int y = (int) (indication *  width - 1);
101           for(int h = 1; h < x; h++)
102             for(int w = 1; w < y; w++)
103               buf[h*width+w] = ~buf[h*width+w];
104         }
105         String front = "dmj.detect.scene-change";
106         String event = front+"("+ (int)ch + " percent)";
107         DmjEvent evt = new DmjEvent(event);
108         String addr = "app:dmj component:CD";
109         (DmjBroker.getBroker("dummy")).sendEvent(evt, addr);
110       }
111       nr_prev = nr;
112       ng_prev = ng;
113       nb_prev = nb;
114       c.debug_msg(out.toString());
115     } else    c.error_msg("\nWrong format! ");
116   }
```

```
117
118 } // ShotDetector.java ends here.
119
```

7 no.uio.ifi.dmj.utils

VoyagerStarter.java

```
1   package no.uio.ifi.dmj.utils;
2
3   import com.objectspace.voyager.*;
4   import com.objectspace.voyager.space.*;
5   import no.uio.ifi.dmj.communication.*;
6
7   class VoyagerStarter {
8     public static void main (String args[]) {
9       int port = 8001;
10      if (args.length != 0) {
11        try {
12          port = Integer.parseInt(args[0]);
13        } catch (Exception e) {e.printStackTrace();}
14      }
15      try {
16        System.out.print("Starting Voyager on port "+
17          port+" ... ");
18        Voyager.startup(""+port);
19        if (Voyager.isStarted()) System.out.println("done");
20      } catch (Exception e) {e.printStackTrace();}
21
22    }
23
24  } // VoyagerStarter.java ends here.
25
```

CDStarter.java

```
1   package no.uio.ifi.dmj.utils;
2
3   import java.rmi.*;
4   import java.util.*;
5   import no.uio.ifi.dmj.control.*;
6
7   public class CDStarter {
8
9     public static void main(String[] args) {
10      String tmp =
11        System.getProperties().getProperty("JCserver");
12      String JCserver = "rmi://"+tmp+":8001/control";
13      IJournalingController control = null;
14      StringBuffer sb = new StringBuffer();
```

```
15      if (args.length < 2) {
16        System.out.print("Usage: java CDStarter [debug] ");
17        System.out.print("name=<PD/CD name> ");
18        System.out.println("dest=<machine>");
19        System.exit(0);
20      }
21      else {
22        for (int i=0; i<args.length; i++)
23          sb.append(" "+args[i]);
24      }
25      try {
26        System.setSecurityManager(new RMISecurityManager());
27        System.out.print( "Looking up controller ... " );
28        control = (IJournalingController)
29          Naming.lookup(JCserver);
30        System.out.println( "done!" );
31        System.out.print( "Starting CD ... " );
32        control.startCD(sb.toString());
33        System.out.println( "done!" );
34      }
35      catch( Exception e ) { e.printStackTrace(); }
36      System.exit(0);
37    }
38
39  } // CDStarter.java ends here.
40
```

PDStarter.java

```
1   package no.uio.ifi.dmj.utils;
2
3   import java.rmi.*;
4   import java.util.*;
5   import no.uio.ifi.dmj.control.*;
6
7   public class PDStarter {
8
9     public static void main(String[] args) {
10
11      String tmp =
12        System.getProperties().getProperty("JCserver");
13      String JCserver = "rmi://"+tmp+":8001/control";
14      IJournalingController control = null;
15      StringBuffer sb = new StringBuffer();
16      if (args.length < 2) {
17        System.out.print("Usage: java PDStarter [debug] ");
18        System.out.print("name=<PD/CD name> dest=<machine> ");
19        System.out.print("[visual] [noaudio] [novideo] ");
20        System.out.print(" [rate=<int>] url=<path> [help]");
21        System.out.println(" \"codec=algorithm(arg)\"");
22        System.exit(0);
23      }
24      else {
25        for (int i=0; i<args.length; i++)
26          sb.append(" "+args[i]);
27      }
28      try {
29        System.setSecurityManager(new RMISecurityManager());
```

```
30        System.out.print( "Looking up controller ... " );
31        control = (IJournalingController)
32          Naming.lookup (JCserver);
33        System.out.println( "done!" );
34        System.out.print( "Starting PD ... " );
35        control.startPD(sb.toString());
36        System.out.println( "done!" );
37      }
38      catch( Exception e )  { e.printStackTrace(); }
39      System.exit(0);
40    }
41
42  } // PDStarter.java ends here.
```

JCStarter.java

```
1   package no.uio.ifi.dmj.utils;
2
3   import no.uio.ifi.dmj.control.*;
4   import com.objectspace.voyager.*;
5   import com.objectspace.voyager.agent.*;
6   import com.objectspace.voyager.rmi.*;
7   import java.net.*;
8
9   public class JCStarter {
10
11    public static void main(String[] args) {
12
13      String path = "no.uio.ifi.dmj.control.";
14      IJournalingController control = null;
15      StringBuffer sb = new StringBuffer();
16      if (args.length > 0) {
17        int i = 0;
18        while (i<args.length) {
19          sb.append(args[i] + " ");
20          i++;
21        }
22      }
23      try {
24        String localhost_name =
25          InetAddress.getLocalHost().getHostName();
26        System.out.print("Starting Voyager on "+
27          localhost_name+", port 8001 ... ");
28        Voyager.startup("8001");
29        if (Voyager.isStarted())
30          System.out.println("done");
31        System.out.print("Creating JournalingController...");
32        control = (IJournalingController)
33          Factory.create(path+"JournalingController");
34        control.configure(sb.toString());
35        System.out.println("ok!");
36        System.out.print("Binding the adress to ");
37        System.out.print("the JournalingController...");
38        ClassManager.enableResourceServer();
39        Namespace.bind("rmi://"+localhost_name+
40          ":8001/control", control);
41        System.out.println("ok!");
42      }
```

```
43      catch(Exception exception) {
44        System.err.println( "\n"+exception );
45        System.exit(0);
46      }
47      System.out.print("JournalingController is ");
48      System.out.println("running !");
49    }
50
51  } // JournalingController.java ends here.
52
```

Move.java

```
1   package no.uio.ifi.dmj.utils;
2
3   import java.rmi.*;
4   import java.util.*;
5   import no.uio.ifi.dmj.control.*;
6
7   public class Move {
8
9     public static void main(String[] args) {
10      String tmp =
11        System.getProperties().getProperty("JCserver");
12      String JCserver = "rmi://"+tmp+":8001/control";
13      IJournalingController control = null;
14      StringBuffer sb = new StringBuffer();
15      if (args.length < 2) {
16        System.out.print("Usage: java Move [debug] ");
17        System.out.print("name=<PD/CD name> ");
18        System.out.println("dest=<//machine:port>");
19        System.exit(0);
20      }
21      else {
22        for (int i=0; i<args.length; i++)
23          sb.append(" "+args[i]);
24      }
25      try {
26        System.setSecurityManager(new RMISecurityManager());
27        System.out.print("Looking up controller ... ");
28        control = (IJournalingController)
29          Naming.lookup(JCserver);
30        System.out.println( "done!" );
31        System.out.print( "Moving detector ... " );
32        control.move(sb.toString());
33        System.out.println( "done!" );
34      }
35      catch( Exception e ) { e.printStackTrace(); }
36      System.exit(0);
37    }
38
39  } // Move.java ends here.
```

Shutdown.java

```
1   package no.uio.ifi.dmj.utils;
2
3   import java.rmi.*;
4   import java.util.*;
5   import no.uio.ifi.dmj.control.*;
6
7   public class Shutdown {
8
9     public static void main(String[] args) {
10      String tmp =
11        System.getProperties().getProperty("JCserver");
12      String JCserver = "rmi://"+tmp+":8001/control";
13      IJournalingController control = null;
14      StringBuffer sb = new StringBuffer();
15      if (args.length < 1) {
16        System.out.print("Usage: java Shutdown [debug] ");
17        System.out.println("name=<PD/CD name> ");
18        System.exit(0);
19      }
20      else {
21        for (int i=0; i<args.length; i++)
22          sb.append(" "+args[i]);
23      }
24      try {
25        System.setSecurityManager( new RMISecurityManager() );
26        System.out.print( "Looking up controller ... " );
27        control = (IJournalingController)
28          Naming.lookup(JCserver);
29        System.out.println( "done!" );
30        System.out.print( "Shutting down detector ... " );
31        control.shutdown(sb.toString());
32        System.out.println( "done!" );
33      }
34      catch( Exception e ) { e.printStackTrace(); }
35      System.exit(0);
36    }
37
38  } // Shutdown.java ends here.
39
```

APPENDIX B - ORDLISTE

CD – Composite event Detector / Sammensatt hendelses detektor, kapittel 1.2.2

COM – Component Object Model, kapittel 3.1.4

CORBA – Common Object Request Broker Architecture, kapittel 3.2.2

DAO – Distribuert Analyse Objekt, kapittel 1.2.2

DCOM – Distributed COM, kapittel 3.1.4

DMJ – Distribuert Media Journalering, kapittel 1.2

DOOBN – Distributed Object-Oriented Bayesian Network, kapittel 1.2.2

DPO – Distribuert Prosessering Omgivelse, kapittel 1.2.2

DS – Distribuert System, kapittel 3.3.3

EJB – Enterprise JavaBeans, kapittel 3.1.3

HMM – Hidden Markov Models, kapittel 2.2.5

IDL – Interface Definition Language, kapittel 3.2.2

IIOP – Internet Inter-ORB Protocol, kapittel 3.2.2

IP multicast – Multicast over Internet Protocol, kapittel 3.2.1

J2EE – Java 2 Enterprise Edition, kapittel 3.1.3

Java RMI – Java Remote Method Invocation, kapittel 3.1.3

JK – Journaling Kontroll komponent, kapittel 1.2.2

JMS – Java Message Service, kapittel 3.2.3

JVM – Java Virtuell Machine, kapittel 3.3.1

MBus – Message Bus, kapittel 3.2.5

ORB – Object Request Broker, kapittel 3.2.2

PD – Primitv Hendelses Detector, kapittel 1.2.2

QoS – Quality of Service, kapittel 1.2

RPC – Remote Procedure Call, kapittel 3.1.3

RTP – Real-time Transport Protocol, kapittel 4.2.3

UDP – User Data Packet, kapittel 4.2.3

UML – Unified Modelling Language, kapittel 1.5

VIC – Video Conference tool, kapittel 4.2.3

Windows DNA – Windows Distributed interNet application Architechture, kapittel 3.1.4

BIBLIOGRAFI

[1] A. D. Birell og B. J. Nelson. *Implementing Remote Procedure Calls*. ACM Transactions on Computer Systems, vol. 2, no. 1, feb. 1984.

[2] A. Lee. *Multimedia Services based on Mobile agents*. The Chinese University of Hong Kong, 2000.

[3] B. Pagurek, A. Bieszczad, og T. White. *Mobile Agents for Network Management*. IEEE Communication Suveys, sept. 1998.

[4] C. Gross, Microsoft Corp. *Building COM components on UNIX*. Teknisk rapport. Juli 1998. Tilgjengelig fra: http://msdn.microsoft.com/library/techart/msdn_unixcom.htm

[5] C. Szyperski. Component Software. Addison-Wesley, 1997.

[6] D. J. Duke, I. Hermann, og M. S. Marshall. *PREMO: A Framework for Multimedia Middleware*. Springer Verlag, 1999.

[7] E. Hartley, A. P. Parkes og D. Hutchison. *A conceptual framework to support content-based multimedia applications*. Lecture Notes in Computer Science, Volume 1629, side 297-315, 1999.

[8] E. Roman. *Mastering Enterprise JavaBeans*. John Whiley & Sons Inc, 1999.

[9] G. Blair og J. Stefani. *Open Distributed Processing and Multimedia*. Addison-Wesley, 1998.

[10] G. Coulouris, J. Dollimore og T. Kindberg. *Distributed Systems, Concepts and Design*. 2. utgave, Addison-Wesley. 1998.

[11] G. Fortino, D. Grimaldi og L. Nigro. *Multicast Control of Mobile Measurement Systems*. IEEE Transactions on Instrumentation and Measurement, vol. 47, no. 5, okt. 1998.

[12] G. Glass. *Voyager - The Universal ORB*. Teknisk rapport, ObjectSpace, Jan. 1999.

[13] G. Hamilton (editor). *JavaBeans API Spesification, v1.01*. Sun Microsystems, Juli 1997.

[14] H. Zhang og Q. Tian. *Digital Video Analysis and Recognition for Corltent-Based Access*. ACM Computing Surveys, vol. 27, no. 4, des. 1995.

[15] J. Crowcroft, M. Handley og I. Wakeman. *Internetworking Multimedia*. UCL Press, okt. 1999.

[16] K. Brockschmidt. *Inside Ole*. MSDN Online Library, apr. 1995.

[17] L. Mathiassen, A. Munk-Madsen, P. A. Nielsen og J. Stage. *Objekt Orientert Analyse og Design*. 2. utgave, Forlaget Marko ApS, 1998.

[18] M. Horstmann og M. Kirtland, Microsoft Corporation. *DCOM Architecture*. Tilgjengelig fra http://msdn.microsoft.com/library/

[19] M. Pawlan. Writing Enterprice Applications with Java 2 SDK, Enterprise Edition. Tutorial, Sun Microsystems, 1999.

[20] Microsoft Corporation. *The Component Object Model Specification*. Tilgjengelig fra: http://www.microsoft.com/com/

[21] O. Granmo, F. Eliassen og O. Lysne. Dynamic object-oriented Bayesian networks and particle filters for flexible resource-aware content-based indexing of media streams. Forsknings rapport, Universitetet i Oslo, okt. 2000. Tilgjenglig fra: http://www.ifi.uio.no/~dmj/publications.html

[22] Object Management Group. *CORBA components*. OMG TC Document orbos/99-02-05. Tilgjengelig fra: http://cgi.omg.org/cgi-bin/doc?orbos/99-02-05

[23] Object Management Group. *CORBA Services: Common Object Services Spesification*. OMG Document formal/98-12-09. Des. 1998. Tilgjengelig fra: http://cgi.omg.org/cgi-bin/doc?formal/98-12-09

[24] Object Management Group. *Java Language to IDL mapping*. OMG Document ptc/99-03-09. Tilgjengelig fra: ftp://ftp.omg.org/pub/docs/ptc/99-03-09.pdf

[25] Object Management Group. *Notification Service Spesification*, versjon 1.0. OMG Document formal/00-06-20. Juni 2000. Tilgjengelig fra: http://cgi.omg.org/cgi-bin/doc?formal/00-06-20

[26] Objectspace Voyager ORB Developer Guide, versjon 4.0. Tilgjengelig fra: http://support.objectspace.com/doc/Orb/

[27] Open Software Foudation. *Distributed Computing Environment, An overview*. 1994. Tilgjenglig fra http://www.osf.org/dce/

[28] P. H. Frölich og M. Franz. *Component-oriented Programming in Object-Oriented Languages*. Technical Report 99-49, Department of Information and Computer Science, University of California, October 1999.

[29] P. Moxon. *Integrating CORBA and J2EE*. Java Developer's Journal, vol. 6, no. 2, Feb. 2001.

[30] P. T. Jahr. *Bruk av mobile agenter til vedlikehold i distribuerte systemer*. Hovedfagsoppgave, Universitetet i Oslo, Instituttet for Informatikk, Nov. 1999.

[31] R. Gray, D. Kotz, G. Cybenko og D. Rus. *Mobile Agents: Motivations and state-of-the-art systems*. Handbook of agent technology, AAAI/MIT Press, Apr. 2000.

[32] R. Orfali og D. Harkey. *Client/Server programming with Java and CORBA*. John Wiley and Sons, 1998.

[33] R. S. Mitchell. *Dynamic Configuration of Distributed Multimedia Components*. Dr. avhandling, Universitetet i London. Aug. 2000.

[34] RFC1094. *NFS: Network File System Protocol Specification*. Mars 1989. Tilgjengelig fra: http://rfc.net/rfc1094.html

[35] RFC1112: *Host Extensions for IP Multicasting*. Aug. 1989. Tilgjengelig fra: http://rfc.net/rfc1112.html

[36] RFC1889. *RTP: A Transport Protocol for Real-Time Applications*. Jan. 1996. Tilgjengelig fra: http://rfc.net/rfc1889.html

[37] S. Eickeler og S. Müller. *Content-based Video Indexing of TV Broadcast News using Hidden Markov Models*. IEEE Int. Conference on Acoustic, Speech and Signal Processing. s. 2997-3000, Phoenix, Mar. 1999.

[38] S. Ossowski. *Distributed Artificial Intelligence*. Lecture Notes in Computer Science, Volume 1535, side 31- 63, 1999.

[39] Software AG Corp. *EnireX DCOM under UNIX*. Teknisk rapport. Juni 2000. Tilgjengelig fra: http://www.softwareag.com/enitirex/download/dcomunix531.pdf

[40] Sun Microsystems Inc. *Java Media Framework - API Guide*. Nov. 1999. Tilgjengelig fra: http://java.sun.com/products/java-media/jmf/

[41] Sun Microsystems. *Java Message Service*. Versjon 1.0.2. Nov 1999. Tilgjengelig fra: http://java.sun.com/products/jms/jms1_0_2-spec.pdf

[42] *The Common Object Request Broker: Architecture and Spesification. Revision 2.4*. OMG, okt. 2000. Tilgjengelig fra http://cgi.omg.org/cgi-bin/doclist.pl

[43] *The Message Bus: A communication & Integration Ifrastructure for Component-based systems*. Teknisk rapport, Jan. 2000. Tilgjengelig fra: http://www.mbus.org

[44] V. Matena og M. Hapner. *Enterprise JavaBeans Spesification, v1.1*. Sun Microsystems, Des. 1999.

[45] V. S. W. Eide, F. Eliassen og O. Lysne. *Supporting Distributed Processing of Time-based Media Streams*. Proceedings fra DOA'2001. Apr. 2001. Tilgjengelig fra: http://www.ifi.uio.no/~dmj/publications.html

[46] V. S. W. Eide, F. Eliassen, O. Granmo og O. Lysne. *Distributed Journaling of Distributed Media*. Proceedings fra NIK'2000, Okt. 2000. Tilgjengelig fra: http://www.ifi.uio.no/~dmj/publications.html

[47] V. S. W. Eide, F. Eliassen, O. Granmo og O. Lysne. *Distributed Media Journaling*. Posisjonsrapport for DMJ prosjektet. Apr. 1999. Tilgjengelig fra: http://www.ifi.uio.no/~dmj/publications.html

[48] V. Sundaram, A. Chandra, P. Goyal og P. Shenoy. *Application Performance in the QLinux Multimedia Operating System*. Proceedings from the 8. ACM Conference on Multimedia.

[49] *VIdeo Conference tool (VIC): Userguide for VIC 2.8*. Sept. 1998. Tilgjengelig fra: http://www-mice.cs.ucl.ac.uk/multimedia/software/

[50] Y. Wang og P. E. Chung. *Customization of distributed systems using COM*. IEEE Concurrency, vol. 6, no. 3, jul-sept. 1998.

(Referansene er alfabetisk listet. Alle linker er kontrollert i mai 2001).

FIGURLISTE