Universitetet i Oslo
Institutt for informatikk

Automatisk prosessering av informasjon i chat-systemer

Kristian F. Hansen

Hovedoppgave

10. mai 2004

Sammendrag

Sanntidsbaserte chat-systemer som opererer over internett har vist seg å være spennende medium for utveksling av informasjon. Et slikt system med eksponensiell vekst er IRC - Internet Relay Chat. Siden det i liten grad finnes søke- eller filterfunksjonalitet tilknyttet dette chat-systemet er en reell utfordring å kunne tilby tjenester med dette formålet.

Gjennom denne oppgaven har det blitt designet og implementert en prototype på et system kalt ACP(Automatisk Chat Prosessor) som forsøker å tilfredsstille ønsket om søk- og filtermekanismer gjennom avansert indeksering. Det overordnede målet har dog ikke vært å implementere et komplett system beregnet for bruk i en produksjonssammenheng, men et system som undersøker former for automatisk prosessering av informasjon forbundet med irc-systemet.

Fagfeltet information retrieval(IR) benyttes som teoretisk bakgrunn for oppgaven og produktet Fast Data Search er basis for implementasjonen. Et kriterie som ansees for å være nødvendig i en IR-kontekst er at man tilbys funksjonelle og intuitive grensesnitt mot indeksert data. Nevnte system implementerer alternative grensesnitt mot både irc-nettverket og indeksert data hvor forbedring av den totale brukeropplevelsen er en del av det overordnede målet.

Forord

Denne oppgaven er skrevet som en del av Candidatus Scientiarum-graden innenfor studieretningen kommunikasjonssystemer ved Institutt for Informatikk, Universitetet i Oslo.

Veileder for oppgaven har vært Dr. Knut Omang som har bidratt med konstruktive tilbakemeldinger på de mange utkastene av rapporten. Knut har i tillegg vært behjelpelig med å få på plass de maskinressurser som har vært nødvendige for å gjennomføre utvikling og testing av det eksperimentelle systemet som er en del av denne oppgaven.

En takk må gå til Bror Gundersen og Nils Petter Lyngstad for kommentarer og nyttige innspill som har vært inspirerende for oppgavens endelige utforming. I tillegg all honnør til Jens 'bastarden' Gjerdbakken for omfattende korrekturlesing.

Innhold

Figurer

Kapittel 1

Innledning

Tidlig på nittitallet begynte bruken av internett for alvor å bre seg ut og konturene av en ny type kommunikasjon tok form. Som et resultat av denne utviklingen ble nye verktøy basert på internett-teknologien introdusert. Et eksempel på et slikt verktøy er chat-systemer. Et chat-system er et medium hvor to eller flere deltakere kan kommunisere i sanntid(”realtime”), og hvor det som skrives normalt reflekteres til alle deltakerene. Et konkret eksempel på en slik tjeneste ,med relativt stor utbredelse, er IRC(Internet Relay Chat), opprinnelig skrevet av Jarkko Oikarinen og definert gjennom internettstandarden rfc1459 [OR93].

Gjennom denne oppgaven vil jeg undersøke former for automatisk prosessering av innhold i tilknytning til chat-kanaler(pratekanler), dette innebærer overvåkning og filtrering av meldinger i chat-systemer, automatiske agenter som opptrer i chat-systemet(”bot’er”), lagring/søk i chat-historien og forbedret grafisk presentasjon av chat-dialogene.

Jeg vil se på to formål for automatisk prosessering av informasjon i chat-systemer, henholdsvis *brukerhjelp* og *overvåkning*. En interessant fremgangsmåte er å forsøke og kategorisere brukerhjelp og overvåkning. Siden IRC er en relativt uformell tjeneste ligger mye av utfordringen i å finne ut hva som aksepteres. Det vil være interessant å undersøke om systemet baseres på nedskrevne regler eller kun tillit blant brukerne. De etiske aspekter vil generelt sett være interessant for oppgaven, krenking av privatliv vil eksempelvis være totalt uakseptabelt sett fra en irc-brukers synsvinkel. Overvåkningsenheter vil antageligvis vurdere dette annerledes. En utfordring ligger i det rent tekniske, hvordan skal man gå frem for å få ut chat-dialogene, med andre ord, hvordan kan man indeksere opp chat-kanaler og gjøre disse søk- og filtrerbare. Siden chat-kanaler ikke nødvendigvis er fritt tilgjengelige vil det også være interessant å avdekke eventuelle nettverk av skjulte/hemmelige kanaler hvor

disse gjennomgår samme analyse.

Fast Search & Transfer [FAS] utvikler programvare for søk og filtrering av informasjon. Denne programvaren tilbyr avansert funksjonalitet gjennom ulike grensesnitt for integrasjon mot andre systemer. Basis for implementasjonen i oppgaven er produktet Fast Data Search.

1.1 Problemstilling

Vi lever i dag i et samfunn som bombarderes med informasjon fra mange forskjellige kanaler. Datasystemer og spesielt internett har den senere tiden vist seg å være en stor bidragsyter på dette informasjonsmarkedet. Et produkt av internettet er det vi kaller chat-systemer. Ved å utveksle informasjon i sanntid har man i langt større grad gjort det mulig å distribuere informasjon hurtig på tvers av landegrenser, kultur og nasjonalitet. En reell utfordring som har oppstått er det å kunne skille ønsket eller relevant informasjon fra støy eller mindre relevant informasjon. Denne oppgaven tar for seg ovennevnte chat-former og systemer hvor spesielt en type er av primærinteresse - IRC. Hvordan skal man gå frem for å indeksere, søke og filtrere ut ønsket informasjon, er informasjonen i utgangspunktet fritt tilgjengelig og er 'tilgjengelighet' et begrep av tvetydig art.

Jeg har videre valgt å segmentere denne overordnede problemstillingen i følgende delproblemer:

- **Hvordan kan produktet Fast Data Search brukes som teknologisk platform til å indeksere, søke og filtrere i chat-system IRC og hvordan kan man presentere indeksert data med formålet om å lette tilgjengelighet og lesbarhet?**

  Det vil være interessant å se hvordan avansert teknologi, i form av Fast Data Search, kan brukes til å"demystifisere" chat-mediumet gjennom indeksering, søk og filterfunksjonalitet. Det vil også være vesentlig at det finnes verktøy for å søke og navigere i tilgjengelig data hvor disse fokuseres rundt forbedring av brukeropplevelsen.

- **Hvordan skal man gå frem for å indeksere, søke og filtrere skjulte/hemmelige chat-kanaler og er disse kanalene mer interessante/nyttige enn de som finnes fritt tilgjengelig?**

  Siden det finnes et nettverk av skjulte kanaler vil det være interessant å gjøre denne informasjonen tilgjengelig for indeksering, søk og filtrering. Hvordan kan man finne frem til slike kanaler og er informasjonen som utveksles på disse kanalene av mer interesse.

- **Er det realistisk å se for seg at klienter mot irc-nettverket i fremtiden integrerer støtte for informasjonsgjenfinning(søk/filter) eller vil det være for ressurskrevende å implementere slik funksjonalitet?**

  Er det mulig å delegere søk- og filterfunksjonalitet til irc-klienter eller fordrer dette at man bruker langt mer komplekse systemer som underbyggende plattform?

- **Er det mulig, sett fra et teknisk perspektiv, å skille informasjonssøk og overvåkning fra hverandre og hvordan reagerer ircmiljøet hvis informasjonssøkingen blir oppfattet som overvåkning?**

  Er det slik at man kan skille informasjonssøk(brukerhjelp) fra overvåkning eller er dette i realiteten to sider av samme sak.

1.2 Arbeidet med oppgaven

Arbeidet med denne oppgaven har vært en tidkrevende og utfordrende prosess hvor mye av forskningen har blitt viet til å finne ut hva som kan være interessant å undersøke nærmere. Siden jeg hadde liten kjennskap til disse områdene på forhånd, har også mye tid gått med til rene litteraturstudier. I tillegg har forsøk med Fast Data Search også tildels vært utført på tidlige utviklingsversjoner der dokumentasjonen fortsatt var mangelfull. På den annen side, ved å jobbe mot løst definerte mål tvinges man til å grave i interessant litteratur samtidig som mulige modeller for design- og implementasjonsløsninger kontinuerlig vurderes.

Av diverse grunner ble den endelige implementasjonen en realitet relativt sent i arbeidet med oppgaven, men allikevel viste denne seg å være robust og funksjonell nok til at ønsket testing kunne gjennomføres. Resultatene burde kunne gi flere mulige innfallsvinkler for videre forskning.

1.3 Oppgavens utforming

Oppgaven er strukturert slik at hvert kapittel står på egne ben, men allikevel følges en logisk struktur som fordrer at leseren opparbeider en forståelse av de forskjellige forskningsfeltene.

Kapittel 2 introduserer diverse chat-systemer slik at leseren får en generell forståelse av fagfeltet. Presentasjonen av systemene fokuseres

rundt de respektive protokollene og funksjonalitet i klientene.

Kapittel 3 beskriver chat-systemet irc som har hovedfokus gjennom oppgaven. Systemet belyses fra forskjellige perspektiv hvor relevant informasjons trekkes inn.

Kapittel 4 introduserer information retrieval som teoretisk grunnlag for oppgaven og knytter dette direkte til mulige implementasjonsløsninger.

Kapittel 5 knytter relevante systemer til denne oppgaven og leseren får et innblikk i ideer som har vært inspirerende og motiverende. Det gis også konkrete eksempler på hva som har blitt hentet inn i og hva som har blitt forkastet.

Kapittel 6 introduserer Fast Data Search som basis for implementasjonen i oppgaven. I tillegg beskrives verktøyet Fast Query Toolkit som leveres med Fast Data Search. Dette verktøyet brukes primært for å gjøre søk mot indeksert data.

Kapittel 7 introduserer en prototype på et system kalt ACP(Automatisk Chat Prosessor) som hovedsaklig har blitt implementert for å undersøke forskjellige former for automatisk prosessering tilknyttet chat-systemet irc.

Kapittel 2

Chat-systemer

Grupper av mennesker kan i dag kommunisere med hverandre interaktivt uavhengig av hvor de måtte befinnes seg i verden, en pre-betingelse er dog at de snakker samme språk. Populære medium som tar i bruk internett-teknologien finner vi blant annet gjennom e-post, nyhetsgrupper(news) og sanntidsbaserte chat-systemer, hvor sistnevnte er av primærinteresse for denne oppgavens omfang.

Et chat-system *defineres* som et sannstidsbasert medium som lar flere personer kommunisere interaktivt. 'Sanntid' eller *real-time* er forøvrig en fellesbetegnelse på kommunikasjon som foregår i nuet, det vil si informasjonen som utveksles reflekteres direkte til de involverte parter i motsetning til systemer som e-post. Således er det sanntidsaspektet som gjør dette til et mer spennende medium fordi man får en flytende dialog hvor brukerne har en oppfatning av hverandre med hensyn på tid og tilgjengelighet.

2.1 Noen utbredte chat-systemer

Under følger en beskrivelse av de vanligste og mest populære chat-systemene som finnes i dag. Beskrivelsen av disse styres i stor grad ut fra den underliggende protokollen og funksjonalitet som finnes i klientene.

Noen av de mest kjente/brukte chat-systemene er:

- *AIM* - AOL Instant Messenger. Dominerende IM-system som stadig øker i popularitet
- *Jabber* - En protokoll som underbygger et åpent og voksende IM-system

- *MSN* - Microsoft sitt IM-system. Skreddersydd for bruk sammen med MSN Messenger klienten
- *ICQ* - Tilgjengelig og populært chat-system basert på protokollen med samme navn
- *SMS-chat* - Alternativt chat-system hvor økonomisk profitt er fremtredende
- *Web-chat* - Alternativt chat-system med egne nettverk, men også mulig å bruke som ren klient mot andre nettverk(f.eks IRC)
- *IRC* - Internet Relay Chat. Fleksibelt og omfattende chat-system. Dette beskrives mer omfattende gjennom neste kapittel

Av disse er de tre første i listen(AIM,Jabber, MSN) naturlig å kategorisere som IM(Instant Messenging)-systemer. IM-systemer er distribuerte programmer som lar brukerne kommunisere med hverandre gjennom en klient-tjener løsning. Av de fire siste i listen er Web-chat og IRC delvis sammenkoblet mens ICQ og SMS-chat er selvstendige løsninger uten noen form for tilknytning til de andre.

2.1.1 AIM

2.1.1.1 Overordnet beskrivelse

AOL(America Online) tilbyr tjenesten de har valgt å kalle AIM [AIM], AOL Instant Messenger. Dette er en sanntidsbasert tjeneste som muliggjør kommunikasjon mellom brukere av AIM klienter. Offisielle AIM klienter benytter seg av en protokoll som kalles OSCAR(Open System for Communication in Realtime). Dette er en binær protokoll som i motsetning til hva navnet tilsier faktisk er lukket. Det eksisterer en AIM klient som er innebygget i AOL-programmet, men brukere av systemet må ikke nødvendigvis benytte seg av AOL for å bruke AIM. AOL tilbyr gratis klienter for alle plattformer(Windows, MAC og Unix), men historisk sett var det først i 1998 man kunne tilby en reell klient for Unix brukere. Frem til da kunne man kun bruke en javaklient som var særdeles ustabil og treg. AOL tok derfor utfordringen og oppgraderte systemet ved å lansere en ny protokoll for IM-systemer samtidig som de slapp en ny klient for Linux brukere. Denne protokollen ble kalt TOC, var strengbasert og ble sluppet under GNU General Public License. TOC er en relativt enkel protokoll sammenlignet med OSCAR, men ved å innføre denne åpnet man også for seriøs bruk av Linux/Unix klienter. TOC er dog ingen erstatning for OSCAR, men tilbyr derimot en proxy funksjon.

2.1.1.2 Funksjonell beskrivelse

OSCAR er den offisielle protokollen for AIM nettverket utviklet av AOL. Den er lukket og baseres kun på tcp-trafikk. Når en bruker kobler seg til nettverket opprettes minst tre forskjellige forbindelser. Først kobles man til en autorisasjonstjeneste som validerer brukernavn og passord. Neste forbindelse settes opp via BOS(Basic Oscar Service) som er knutepunktet for alle meldinger og avslutningsvis kobles man til en ChatNav(Chat Navigation) som gjør det mulig å logge seg på/forlate praterom. I tillegg realiseres alle forbindelser til praterom som selvstendige tcp-sesjoner. Alle filoperasjoner og direkte forbindelser instansiert av klientene er også egne tcp-forbindelser, men disse går aldri via AOL.

TOC-protokollen ble utviklet av AOL fordi man ønsket å la uoffisielle klienter koble seg til AIM-nettverket. Den første klienten som tok i bruk TOC var TiK, en AIM klient skrevet i tcl/TK, , men i dag finnes det flere gode klienter å velge mellom som f.eks Gaim. TOC er som "forgjengeren" en tcp- og ASCII-basert protokoll, men skiller seg ut ved å bruke en 6-bits binær header. Vanligvis består hver sesjon av bare en forbindelse, unntaket er for nevnte filoperasjoner som er direkte forbindelser mellom klientene. Brukernavn og passord som sendes over nettverket beskyttes forøvrig bare av en svak krypteringalgoritme.

TOC lider av den ikke er den offisielle løsningen for AIM nettverket. Utviklingen har stoppet fullstendig opp og man har ikke implementert noe ny funksjonalitet siden den offisielt ble sluppet.

For de klienter som offisielt støttet av AOL tilbys filoverføring, blokkering av brukere, mulighet til å sette status(borte, tilgjengelig), opprette kontaktliste(venner og bekjente, også kalt buddylist) osv. Til tross for at miljøet ønsker utbedret støtte og tilnærmet lik funksjonalitet for uoffisielle klienter begrenses man foreløpig av protokollen hvor kun kjernefunksjonalitet eksisterer.

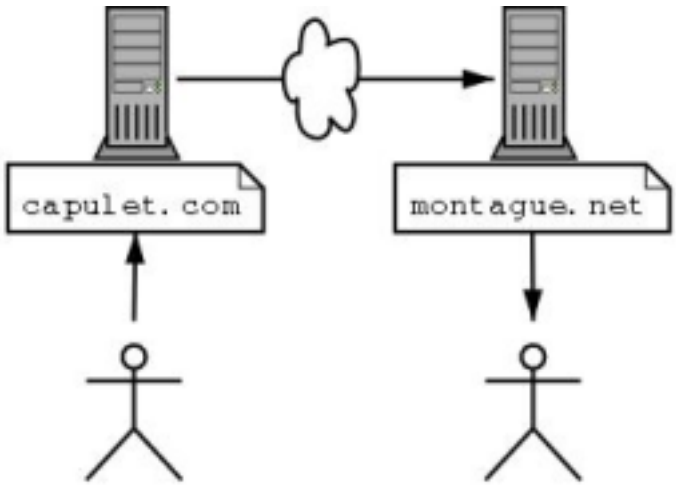
2.1.2 Jabber

2.1.2.1 Overordnet beskrivelse

I 1998 utviklet Jeremie Miller den første varianten av IM-systemet vi kjenner som Jabber [JAB]. Jabber har hatt en enorm vekst og er per i dag det største og mest fremgangsrike systemet basert på åpen kildekode. Jabber-tjenere og kildekode er tilgjengelige for nedlasting fra internettet.

2.1.2.2 Funksjonell beskrivelse

Jabber-protokollen er er en åpen XML-protokoll basert på sanntidskommunikasjon mellom to noder som bruker internett som transportlag. Jabber bygger på en klient-tjener arkitektur, dvs alle data som sendes fra en klient til en annen må gjennom minst en jabber-tjener. Et eksempel på hvordan meldingsutvekslingen foregår vises i figuren 2.1.



Figur 2.1: Meldingsflyt fra [JAB]

1. Bruker A(capulet.com) sender en melding til bruker B(montague.net)
2. Capulet.com tjeneren åpner en forbindelse mot tjeneren på montague.net hvis det som pre-betingelse ikke eksisterer en forbindelse allerede
3. Meldingen med opphav fra capulet.com rutes gjennom tjeneren i montague.net
4. Montague.net tjeneren ekstraherer brukernavnet som finnes innkapslet i meldingen og sørger for at den rutes til rett Jabber-klient
5. Meldingen fremvises i mottagerens klient

XML er en helt sentral del av jabber, når en klient åpner en forbindelse til en tjener åpner den samtidig en enveis XML-strøm med data fra seg selv til tjeneren, og tjeneren svarer med en enveis XML-strøm tilbake til

klienten. Hver sesjon består således av to XML-strømmer og all kommunikasjon mellom tjener og klient rutes gjennom disse strømmene.

Jabber tilbyr også det man har valgt å kalle "transport". Gjennom en transport kan man kommunisere med andre IM-systemer som AIM og MSN. For å kunne bruke en transport må man "abonnere" på tjenesten. I et gitt scenario sørger jabber-systemet for at man kobles mot den ønskede tjenesten og gjør klienter på den "andre siden" synlige.

Jabber tilbyr også ekstra fleksibilitet i forhold til andre chat-systemer. Det er ingenting i veien for å sette opp en egen jabber-tjener hvor denne globaliseres for nettverket. Det finnes også mulighet for å kryptere dataene man sender ved å implementere SSL(Secure Socket Layer) på tjenersiden eller PGP/GPG(Pretty Good Privacy/Gnu Privacy Guard) på klientsiden.

Gabber og Gossip er eksempler på populære jabber-klienter til Linux, men dette er et fritt og åpent system som er tilgjengelig for alle de store plattformene. Funksjonaliteten er i hovedsak lik den vi finner i klienter for andre IM-systemer.

2.1.3 MSN

2.1.3.1 Overordnet beskrivelse

I 1999 lanserte Microsoft MSN Messenger [MSN] tjenesten. En gratis tjeneste som hadde som mål å integrere seg mot eksisterende produkter, hovedsaklig Hotmail-systemet. Ideen var at hotmail-brukerne ikke skulle begrenses til e-post, men også ha mulighet til å kommunisere i sanntid. Slik hadde man allerede skaffet seg en solid skare av potensielle brukere. Tjenesten ble en umiddelbar suksess og er fortsatt dominerende i dag.

2.1.3.2 Funksjonell beskrivelse

All kommunikasjon som realiseres over MSN-protokollen må gå via en tjener, unntaket er klient-klient forbindelser som filoverføring og tale(voice chat). Dette gjennomføres ved at MSN-klientene kobler seg til flere forskjellige tjenere samtidig hvor hver tjener dupliseres et gitt antall ganger. Slik vil systemet til enhver tid støttet et ubegrenset antall klienter. En skisse av protokollen og aktuelle forbindelser vises i figuren 2.2

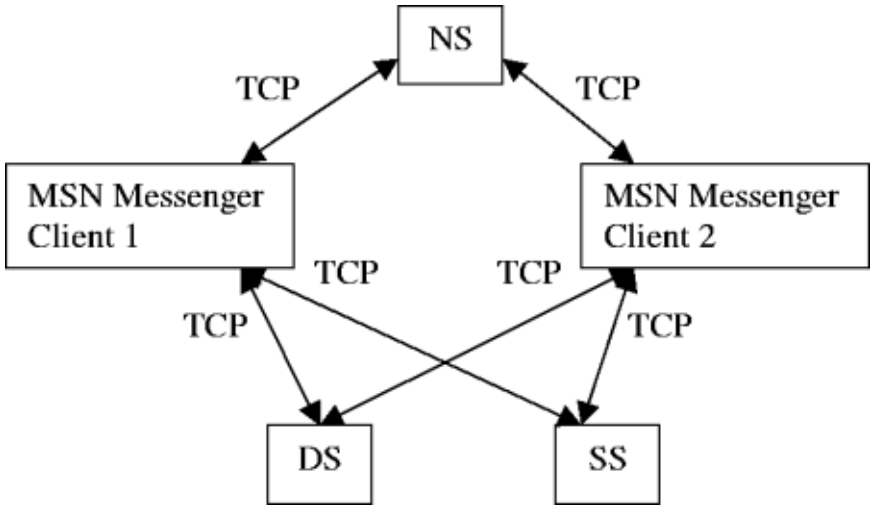
Dispatch Server(DS): Dispatch-tjeneren utgjør starpunktet for enhver klient-tjener forbindelse. I tillegg sørger den for å koble klienter

til rett notifikasjonstjener(se under).

Notification Server(NS): Notifikasjonstjeneren er hovedkomponenten på tjenersiden. Meldinger som utveksles mellom klienter og notifikasjonstjeneren vil typisk være autentisering, synkronisering av brukerattributter og asynkrone notifikasjonshandlinger som f.eks tilstandsendring. En reell handling vil være at en klient endrer status fra online til offline.

Switchboard Server(SS): Switchboard-tjeneren opptrer som en nærmest "usynlig" link mellom klient-klient forbindelser. Hvis en klient ønsker å kommunisere med en annen klient sender den først en melding til notifikasjonstjeneren som videreformidler denne til switchboard-tjeneren. Når forbindelsen er opprettet mottar destinasjonsklienten en melding fra sin lokale notifikasjonstjener hvor den oppfordres til å kontakte samme switchboard-tjener.

Kommandoer som ønskes utført av klientene sendes til tjeneren som forespørsler. Meldingsflyten mellom klientene og tjenerne er fullstendig asynkron slik at klienter kan sende mange samtidige forespørsler i sekvens uten at de behøver å vente på svar for hver enkelt kommando. Svaret som returneres av tjeneren er enten positivt(kommandoen er vellykket) eller negativt(det har oppstått en feil, kommandoen avvises). Det er heller ikke nødvendig at svarene returneres i samme rekkefølge som de ble forespurt ettersom hver klient forholder seg til en unik transaksjons-id som skiller forbindelsene fra hverandre.



Figur 2.2: MSN-protokollen fra [Uni, side 6]

Transaksjons-id'en representeres ved et tall mellom 0 og $2^{32} - 1$ og identifiserer alle kommandoer som sendes fra klientene til tjeneren. Vanligvis er det slik at klienten selv genererer denne id'en og i så måte er ansvarlig for å koble svar til rett forespørsel. Tjenerens primæroppgave er kun å returnere svar som samsvarer med id'en. I tilfeller hvor tjeneren sender kommandoer som avhenger av transaksjons-id og respons fra klienten er overflødig bruker den 0(null) som transaksjons-id. I tilfeller hvor tjeneren returnerer flere samtidige svar til en enkelt forespørsel benytter den seg av samme id hele veien.

MSN Messenger, Microsoft sin offisielle klient, er den mest utbredte klienten som brukes mot MSN-systemet. Denne er skreddersydd for dette formålet og integrerer således all funksjonalitet som tilbys av den underliggende protokollen. Dette inkluderer:

- Tale- og videokonferanser
- Sende og motta filer
- Utbredt språkstøtte(26 språk)
- Nettverksbaserte spill

MSN Messenger klienten finnes kun på Mac og Windows-plattformen, men ved bruk av plugins har man gjort MSN-nettverket tilgjengelig for andre operativsystemer selv om dette offisielt ikke støttes.

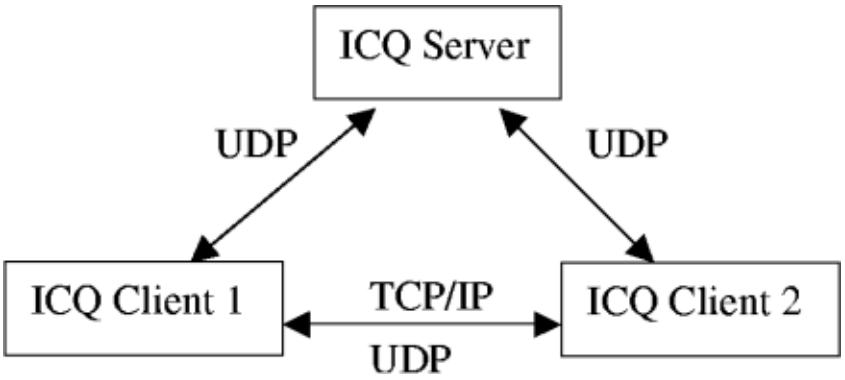
2.1.4 ICQ

2.1.4.1 Overordnet beskrivelse

I 1996 dannet tre israelere Mirablis, et nytt internettselskap med store ambisjoner. Deres hjertebarn var ICQ [ICQ] eller "I Seek You" som det populære akronymet er en forkortelse av. Målet var at flest mulig internettbrukere skulle kunne kommunisere med hverandre på en enkel og lettfattelig måte. Tjenesten har vokst seg stor og regnes i dag for å være en av de største bidragsyterne innenfor sanntidsrelatert kommunikasjon.

2.1.4.2 Funksjonell beskrivelse

ICQ-protokollen implementerer 2 "lag" av kommunikasjon, henholdsvis



Figur 2.3: ICQ-protokollen fra [Uni, side 5]

klient-tjener og klient-klient forbindelser. Se figuren 2.3 for skisse.

Klient-tjener Klient-tjener forbindelsene realiseres ved synkron udp trafikk. Pakker som sendes fra klientene til tjeneren krypteres i motsetning til pakker som går andre veien(tjener-klient). Hver pakke assosieres med en unik sesjons-id for hver klientforbindelse. Uten en gyldig id vil pakkene avvises av tjeneren. Id'en returneres i hver pakke som sendes tilbake fra tjeneren slik at klienten vet at denne garanterer ekthet(hindrer "spoofing"). Når klienten sender en pakke til tjeneren mottar den en bekreftelse i form av srv_ack. I tilfeller hvor klienten ikke mottar bekreftelse sender den pakken på nytt inntil svar mottas. Tjeneren forventer også å motta en bekreftelse for hver pakke den sender klienten, med unntak av selve bekreftelses-meldingene(srv_ack).

Klient-klient Kommunikasjonen mellom klienter foregår hovedsaklig via direkte tcp-forbindelser. I instansieringsfasen er dog forbindelsene udp-baserte, men etter utveksling av ipadresser og bruker-id(ICQ nummer) mappes linken til en tcp-basert forbindelse. Ut fra spesifikasjonene gitt av tcp-protokollen vet en at hver forbindelse instansieres separat. Dette betyr i praksis at det finnes en socket for hver operasjon klientene ønsker utført(filoverføring, meldinger osv). Straks en klient registreres som online opprettes det en meldings-socket mens for chat og annen funksjonalitet opprettes sockets først ved behov. En nødvendig del av kommunikasjonsprosessen over tcp innebærer at essensiell pakkeinformasjon sendes før selve pakken. Dette betyr at pakkestørrelsen sendes før dataene fordi pakkene er "størrelsesløse". En pakke som mottas kan for eksempel bare være et mindre segment av den fullstendige pakken. Slik vil man ved å sende størrelsen først hjelpe mottageren til

å bestemme om en pakke er fullstendig eller ikke.

Meldingsutvekslingen instansieres etter at klienten har bekreftet gitte tcp-pakke. Når destinasjonsenheten mottar og bekrefter pakkens ekthet returneres en ack-pakke til avsenderen.

Det finnes ICQ-klienter til de fleste plattformer, men det er verdt å merke seg at disse kun offisielt støttes under Windows. Typisk funksjonalitet integrert i klienter er:

- Tale- og videokonferanser
- Filoperasjoner - Sende og motta
- Stavekontroll
- Nettverksspill

2.1.5 SMS-chat

2.1.5.1 Overordnet beskrivelse

SMS-chat baseres på SMS tjenesten man har tilgjengelig via mobiltelefonen. Utover å sende meldinger til andre mobiltelefoneiere har man den senere tiden sett en trend hvor kommersielle aktører har "utvidet" dette markedet ved å tilby spesialtilpassede chat-tjenester. Typisk er dette tv-stasjoner som i tiden de ikke har sendinger tilbyr en chat-tjeneste hvor sms-meldinger reflekteres på tv-skjermen til den aktuelle stasjonen.

2.1.5.2 Funksjonell beskrivelse

SMS-chat er den formen for chat som skiller seg mest fra de andre typene som har blitt nevnt. Dette er i realiteten ingen ny type chat, det er bare profittbasert utnyttelse av eksisterende teknologi. Mobiltelefonen fungerer som "klient" mot systemet. Ofte er det slik at man tilbys å registrere seg med brukernavn som assosieres med dine meldinger, eventuelt kan man forbli anonym, dvs de respektive dialogene prefikses med 'anonym'. Fellesnevneren er uansett at for hver melding som utveksles belastes klienten med en gitt sum, avhengig av hvilken takst tjenesten følger. Dette varierer i stor grad avhengig av hvilken tjeneste man bruker.

Sammenlignet med andre chat-systemer er denne SMS varianten også mindre effektiv fordi man forholder seg til de begrensninger den underliggende SMS-protokollen har. SMS er et akronym for Short Message

Service og sørger for transport av meldinger fra en mobiltelefon til en annen mobiltelefon, telefaks og/eller ip-adresse. Meldingene kan forøvrig ikke overskride 160 tegn og det finnes ingen mulighet for å inkludere bilder eller annen type grafikk. Når en melding har blitt sendt mottas denne først av et "Short Message Service Center"(SMSC) som sørger for den videre rutingen til den aktuelle enheten. Dette skjer ved at SMSC'en sender en forespørsel til et "Home Location Register"(HLR) som søker frem mottageren. Når denne er kartlagt returnerer Home Location registeret status, dvs destinasjonsenheten betegnes som aktiv eller inaktiv. Ved inaktiv respons bevarer SMSC'en meldingen en gitt periode(vanligvis inntil 24 timer) før den forkastes. Har derimot mottageren blitt tilgjengelig innenfor dette tidsrommet gir HL-registeret beskjed til SMSC'en som forsøker å sende meldingen på nytt. Ved 'aktivt' svar leter systemet opp destinasjonsenheten og meldingen leveres direkte. Avslutningsvis mottar SMSC'en en bekreftelse på at meldingen har blitt sendt og mottatt og den merkes som 'sendt'. Felles for begge formene er at SMSC-meldinger innkapsles i et "Short Message Delivery Point to Point" format.

For en SMS-chat har man vanligvis en datamaskin som tar i mot all innkommende trafikk og leverer denne videre til tv-skjermen. Utover felles fremvisningen finnes det også mulighet for å segmentere i private "rom" hvor mindre grupper av personer kommuniserer. Det er også vanlig at man har hvertfall en person som fungerer som moderator for chattjenesten slik at meldinger som ikke egner seg for fremvisning kan fjernes/modereres og brukere som ikke innfinner seg etter gjeldende reglement identifiseres og stenges ute av systemet.

2.1.6 Web-chat

2.1.6.1 Overordnet beskrivelse

En av de mest voksende og populære formene for chat er i dag det vi kaller web-chat. Web-chat er i realiteten bare et brukergrensesnitt mot underliggende protokoller som irc. Motivasjonen for å bruke denne chatformen har sitt utspring i ønsket om lettfattelighet og tilgjengelighet. De fleste kjenner til hvordan en nettleser fungerer og mange portaler tilbyr i dag diverse chat-tjenester.

2.1.6.2 Funksjonell beskrivelse

Web-chat kan deles i tre klasser:

1. HTML web-chat - Systemer som bruker nettleseren som brukergrensesnitt og http som underliggende applikasjonsprotokoll
2. Applet web-chat - Systemer som bruker en applet som brukergrensesnitt. Denne lastes inn i nettleseren via http og en underliggende applikasjonsprotokoll
3. Applet irc-chat - Systemer som bruker Java eller Javascript applets som grafiske brukergrensesnitt mot velkjente protokoller, typisk irc

Web-chat: Brukerne av systemet benytter nettleseren som klient og all informasjon som utveksles på de forskjellige praterommene eller kanalene reflekteres i den aktuelle html-siden, referert til som chatvindu. Så lenge brukeren beholder den samme siden i nettleseren vil den kontinuerlig oppdateres med de nyeste dialogene. Dette opphører først når brukeren forlater systemet ved enten å logge av eller går videre til en ny nettside. En chat instansieres typisk ved at brukeren fyller ut et skjema som vanligvis er tilgjengelig på den aktuelle siden. Et minimum for å registrere seg som klient er at man oppgir ønsket brukernavn og passord. For at hver bruker skal bli assosiert med riktig output av kanaler/praterom genererer systemet en sesjons-id. Denne id'en er vanligvis en tall- eller bokstavsekvens utledet av innloggingsprosessen og følger brukeren gjennom systemet via *cookies* eller http-adresse. Input til systemet skrives inn via nettsiden i såkalte "forms" assosiert med den unike id'en. Hver linje som tastes inn distribueres til alle brukerne som befinner seg i det samme rommet/kanalen. I tillegg til grunnleggende funksjonalitet finnes det ekstra funksjoner som lar brukerne finstille grensesnittet ved blant annet å endre farger, bytte praterom, sende private meldinger osv. Disse operasjonene følges også av klientenes unike id.

Applet web-chat: Noen systemer implementerer denne løsningen fremfor å bruke en ren nettside til å fremvise dialogene. Typisk vil implementasjonen være basert på Java eller andre programmeringsspråk. Man ønsker også å tilby en grad av funksjonalitet gjennom ryddig fremvisning av dialogene, knapper for å logge seg av/på systemet og brukerlister relatert til de forskjellige praterommene/kanalene.

Applet irc-chat: Slike systemer endrer kun brukergrensesnittet mot den underliggende irc-protokollen. Funksjonaliteten er noenlunde lik den vi finner i de to ovennevnte variantene, men hovedforskjellen er at man konverterer alle meldinger som sendes til og fra appleten

slik at disse er kompatible med irc-protokollen. En effekt av dette er at man har et brukervennlig grensesnitt mot irc-nettverket og gjør irc mer tilgjengelig for brukere som ikke føler seg komfortable med de rene irc-klientene. Fra et nettverksperspektiv er slike systemer i realiteten irc-systemer mens fra et applikasjonsperspektiv er de applet-chats.

Oppsummering

Gjennom dette kapitlet har det blitt beskrevet noen utvalgte chat-systemer sett fra et bruker- og systemperspektiv. Kapitlet er primært bakgrunnsstoff for å få en forståelse av forskningsdomenet.

Kapittel 3

Internet Relay Chat

Gjennom dette kapitlet presenteres systemet Internet Relay Chat(IRC) grundigere. Fra tidligere har det blitt kartlagt at dette systemet vil opptre som forskningsdomene for oppgaven og derav følger en beskrivelse av systemet sett fra fire forskjellige perspektiv. Først presenteres en kort bakgrunnsbeskrivelse hvor også relevant historikk trekkes inn, vi plasserer irc i en modell og beskriver strukturen, irc-protokollen belyses nærmere ved å se på meldingsformat og meldingsflyt og avslutningsvis følger en vurdering av aktuelle grensesnitt mot systemet.

3.1 Bakgrunn

Internet Relay Chat baseres på en distribuert og utbredt protokoll som har eksistert i nærmere 16 år. Rent historisk har den sine forgjengerne i gamle Unix programmer som *write*, *finger* og *talk*, hvor sistnevnte er mest fremtredende. Disse eksisterer fortsatt og er innenfor noen idealistiske miljøer hyppig brukt, men utviklingsmessig er deres tid over, dagens chat-systemer er langt mer funksjonelle og estetisk appellerende.

Hvert IRC-nettverk består av mange uavhengige maskiner som kjører såkalte *daemons*. Disse programmene lytter vanligvis på en bestemt port, typisk 6667, hvor klienter kan koble seg til. Forøvrig utveksles også alle tjenermeldinger via denne porten.

Det er viktig å merke seg at det finnes flere uavhengige IRC-nettverk og brukerne kan kun kommunisere dersom de er koblet opp mot en tjener som representerer det samme nettverket. De største og mest populære nettverkene inkluderer Dalnet, EFNet, Undernet og Freenode. Mellom 6 og 7 juni i fjor(2003) hadde disse nettverkene totalt et gjennomsnitt på

457190 brukere [Uni].

Klientene skilles fra hverandre ved å bruke unike *kallenavn*("nickname") som de identifiseres ved så lenge de er innenfor nettverket. Det er tillatt å endre kallenavn så lenge det man ønsker å bytte til er ledig og gyldig i henhold til protokollen. Kallenavnet globaliseres og assosieres med meldingene klienten utveksler. En av de vanligste formene for meldinger er *PRIVMSG* som brukes for å kommunisere med andre brukere direkte eller via kanaler. Kanaler er selve kjernen i IRC-systemet og danner abstraksjonslaget mellom brukerne. Det er ikke nødvendigvis et krav at man som klient deltar i noen kanaler, men vanligvis er det slik at hver klient knytter seg opp mot minimum en kanal. Kanaler relateres forøvrig til et sett av attributter som bestemmes av kanaloperatørene, en dedikert gruppe klienter med primæroppgave å administrere og vedlikeholde disse.

Irc-nettverket kan også benyttes til å dele data mellom klientene. Dvs det finnes en nær tilknyttet protokoll som muliggjør fildeling. Ren datakommunikasjon foregår uten at man inkluderer irc-tjenerne. Dette er direkte tcp-forbindelser mellom klientene som realiseres ved en egen type meldinger.

3.2 IRC-modellen

3.2.1 Tjenermaskinene

Hver tjener innenfor IRC-nettverket innehar en liste over andre tjenere den kan koble seg til. Via disse listene identifiseres en *spanning tree* struktur mellom de involverte tjenerne fordi man ønsker å redusere kompleksitet forbundet med meldingsrutingen. Spanning tree er forøvrig den eneste nettverkskonfigurasjonen som tillates. Hver tjenermaskin opptrer således som en sentral node for resten av nettverket. Et eksempel på en slik struktur finner vi i figuren 3.1.

3.2.2 Klienter

Programmer som kobler seg til irc-nettverket(tjenerne), men per definisjon ikke er en tjener identifiseres som klienter. Klienter skilles fra hverandre ved bruk av unike kallenavn som kan ha en maksimal lengde på 9 tegn(bokstaver, tall eller tegn). Hvis man velger et kallenavn som allerede er i bruk vil man få et "tilfeldig" generert. Tilfeldig i en slik sammenheng innebærer at hvis en har valgt kallenavnet "smarten" vil man

få samme kallenavn postfikset med et vilkårlig tegn, vanligvis _(understrek), som i dette tilfellet vil gi "smarten_". De fleste klienter innehar funksjoner for å spesifisere alternative kallenavn hvis det skulle være ønskelig. I tillegg kreves det at maskin- og brukernavnet klienten styres fra gjøres tilgjengelig for den aktuelle tjeneren.

3.2.3 Tjeneroperatører

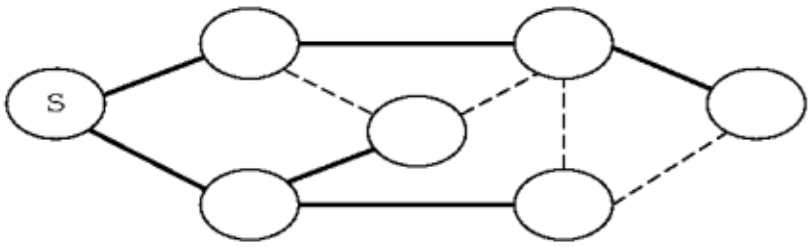
Irc-nettverket er helt avhengig av vedlikehold(regler for hva som er tillatt, at det fungerer tilfredsstillende) og derfor finnes det en egen type klienter, operatørene, som skal fylle denne rollen. Tjeneroperatørene innehar rotfunksjonen(jmf. Unix) og administrerer i høy grad nettverkene. En tjeneroperatør har myndighet til å overlappe lokale kanaloperatører og eksempelvis fjerne brukere fra nettverket. Dette bør dog kun gjøres i spesielle tilfeller hvor en eller flere klienter gjentatte ganger misbruker nettverket.

3.2.4 Kanaler

En kanal defineres som en gruppe med et utvalgt navn hvor en eller flere klienter kan utveksle meldinger [OR93]. En kanal kan opprettes med øyeblikkelig virkning og den vil ha levetid så lenge det er brukere der(kanalen dør når siste bruker forlater den).

Kanalnavnene er strenger prefixet med & eller # tegnene, hvor sistnevnte er mest utbredt. Bortsett fra kravet om & og # tegnene er de øvrige begrensningene kun at navnene ikke kan inneholde mellomrom, control G(Ĝ eller ASCII 7) eller komma(fordi komma brukes som listeseparator av protokollen).

Det finnes i så måte to typer kanaler som tillates av IRC-protokollen. Den



Figur 3.1: Spanning Tree

ene typen distribueres globalt til alle tjenerne som er en del av nettverket og har alltid en ledende # i kanalnavnet. Den andre typen er kun gjeldende på den spesifikke tjener kanalen ble opprettet og vil alltid ha en ledende & i kanalnavnet. I tillegg til dette har man også mulighet til å endre attributter på kanalene, f.eks hvor mange samtidige klienter som tillates, sette tema(”topic”) og evt kun la inviterte(”invite only”) brukere delta i kanalen. Disse attributtene styres av kanaloperatørene(begrepet utdypes nedenfor).

For at en potensiell bruker skal kunne være med i en kanal eller starte en ny må brukeren selv bli med i denne(”join”). Hvis kanalen ikke eksisterer fra tidligere vil den umiddelbart opprettes og brukeren blir tildelt kanaloperatør status. Hvis derimot kanalen eksisterer er man ikke garantert noe status når man tar del i den. Forøvrig kan det nevnes at den senere tiden har noen av de store nettverkene(Undernet og Freenode) implementert en løsning som gjør at man kan registrere kanaler(vanligvis etter en periode hvor kanalen etableres og får faste brukere), dette skjer gjennom utfylling av søknadsskjema og endelig avgjørelse fattes av nettverksadministratorene på det aktuelle nettverket. Går søknaden gjennom vil registraren få full kontroll over kanalen uavhengig om det er klienter der eller ikke(mulighet til å endre attributter dynamisk).

Irc-nettverk kan innta en ustabil tilstand pga *netsplit* mellom to tjenere, dette vil berøre kanaler og klienter på de respektive sidene av *”split’en”*. Klientene faller av kanalen(e) som berøres og mister sin daværende status, men når nettverket leges vil de aktuelle tjenerne annonsere seg i mellom hvilke brukere de tror var på kanalen(e) og hva slags attributter kanalen(e) hadde. Hvis kanalen(e) eksisterte på begge sider vil en sofistikert funksjon integrert i irc-protokollen føre til at tjenerne forhandler seg frem til hvilke klienter som tilhører hvilke kanaler og hva slags attributter disse assosieres med.

3.2.5 Kanaloperatører

Kanaloperatører eller ”op’s” som de populært kalles er egen gruppe administratorer som innehar ”eierskap” over den eller de kanalene de er en del av. Operatørene har makt, i den grad det kan kalles det, til blant annet å sette tema på kanalen(topic) og fjerne uønskede brukere. Deres rolle tjener til generelt vedlikehold av kanalen eller kanalene de er medlemmer av. Operatører trenger ikke nødvendigvis forklare sine handlinger selv om disse oppfattes som uriktige av opinionen. I helt spesielle tilfeller har det dog forekommet at personer har mistet sin operatørstatus pga destruktiv holdning i forhold andre brukere. Disse vedtakene fattes av irc-operatørene(se ovenfor) og er endelige. Man har selvfølgelig

muligheten til å forlate kanaler frivillig hvis operatørene er truende eller generelt sett har en negativ holdning mot brukerne. Noen kanaler har faktisk gått til grunne ved slik felles "boikott".

De vanligste kommandoene en operatør assosieres med er følgende:

KICK - Fjerne en bruker fra en kanal

MODE - Endre attributt på en kanal

INVITE - Invitere en klient til en begrenset kanal(modus +i)

TOPIC - Forandre tema på en kanal(modus +t)

3.3 IRC-protokollen

Irc-protokollen defineres ikke ved et gitt tegnsett, men baseres kun på en mengde koder satt sammen av 8 bits, tilsvarende en oktett, hvor hver melding består av et gitt antall oktetter. Det finnes dog unntak i form av skilletegn(delimiters), noen av oktettene er reserverte kontrollkoder.

Selv om protokollen begrenses til 8-bit er skilletegnene av en slik art at protokollen uten videre kan brukes via USASCII terminaler og over telnet forbindelser.

Irc-protokollen differensierer ikke på tegnene { } | [ ] og disse blir derfor sett på som ekvivalente. Typisk tilfelle er når man skal skille mellom to kallenavn, det er ikke mulig å ha [kallenavn] og {kallenavn} på samme nettverk.

3.3.1 Meldingsformat

Per definisjon er tjener-tjener og klient-tjener kommunikasjon asynkron. Hvis meldingene som utveksles inneholder gyldige kommandoer er det rimelig å forvente et svar, men dette garanteres ikke av protokollen.

En Irc-melding kan maksimalt segmenteres i 3 forskjellige deler, *prefiks*(valgfritt), *«kommando»* og *kommandoparametere*(maksimalt 15 parametre). Disse delene skilles ved et eller flere mellomrom(ASCII space).

Prefiks Hvis en melding inneholder prefix vil dette vises ved at kolon(:) er tilstedeværende og synlig. Kolontegnet må være første tegn i meldingen og mellomrom tillates ikke. Prefiks benyttes hovedsaklig for å knytte melding til rett klient, uten prefiks vil tjeneren anta

at meldingen hører til forbindelsen den ble mottatt via. Generelt sett skal ikke klienter benytte seg av prefiks , hvis dette allikevel innteffer vil det eneste gyldige prefikset være kallenavnet assosiert med klienten. Hvis så skulle skje at tjeneren, ved å traversere sin interne database, ikke finner klienten identifisert ved prefiks eller hvis klienten er registrert ved en annen forbindelse enn den meldingen stammer fra vil tjeneren forkaste meldingen.

Kommando Kommandoen må enten være en gyldig irc-kommando eller et tresifret tall i ASCII format.

Kommandoparametre Meldinger består av linjer med tegn som termineres ved et Carrage Return - Line Feed par(CR-LF). Meldingene kan ha en maksimal lengde på 512 tegn inkludert CR-LF. Med andre ord er 510 tegn den reelle lengden en kommando og tilhørende parametre kan bestå av.

```
<message> ::= [ ´:´ <prefix> <SPACE> ] <command> <params> <crlf>
<prefix> ::= <servername> | <nick> [ ´!´ <user> ] [ ´@´ <host> ]
<command> ::= <letter>  <letter>  | <number> <number> <number>
<SPACE> ::= ´ ´ { ´ ´ }
<params> ::= <SPACE> [ ´:´ <trailing> | <middle> <params> ]

<middle> ::= <Any *non-empty* sequence of octets not including SPACE
or NUL or CR or LF, the first of which may not be ´:´ >

<trailing> ::= <Any, possibly *empty*, sequence of octets not including
NUL or CR or LF>

<crlf> ::= CR LF
```

Figur 3.2: BNF-representasjon av meldinger fra [OR93, side 8]

For å skille ut meldinger fra oktettstrømmene benytter man seg per i dag av en metode som løser dette ved å innføre CR og LF som meldingsseparatorer. I tillegg ignoreres tomme meldinger slik at man uten videre kan bruke CR-LF sekvenser mellom meldinger. De ekstraherte meldingene parseres og klassifiseres i henhold til komponentene <prefix>, <command> og en parameterliste som matcher på <middle> eller <trailing> komponenter.

De forskjellige komponentene i figuren 3.2 har følgende egenskaper:

- SPACE består kun av ”tomme” tegn eller mellomrom(0x20)

- Parameterlisten betrakter alle parametre som likeverdige uavhengig om det matches på <middle> eller <trailing>. Sistnevnte brukes forøvrig som et syntaktisk triks slik at man kan bruke SPACE som parameterverdi
- Ut fra måten man betrakter meldinger kan ikke CR og LF benyttes som parameterverdier
- NUL tillates ikke brukt i meldinger pga av ekstra kompleksistet forbundet med dette
- Avsluttende parameter kan være en tom streng
- Utvidet prefiks ([´!´ <user>] [´@´ <host>]) er forbeholdt tjener-klient forbindelser slik at klientene til enhver tid oppdateres med informasjon relatert til meldingene de mottar

Meldinger som sendes til en tjener genererer et form for svar, vanligvis numerisk, som brukes til både feilmeldinger og vanlige meldinger. Et numerisk svar er en melding bestående av prefix(avsender), et tre sifret nummer(som nevnt tidligere) og destinasjon. Et slikt svar er forbeholdt tjenerne og eventuelle numeriske verdier sendt fra klienter forkastes av tjenerne. Foruten dette forbindes nummeriske svar ved samme egenskaper som vanlige meldinger, bortsett fra at nøkkelordet genereres ved bruk av tre numeriske siffer istedenfor en tekststreng.

3.3.2 Meldingsflyt

Den nåværende implementasjonen av IRC-protokollen segmenterer på forskjellige former for kommunikasjon:

3.3.2.1 En-til-en kommunikasjon

En-til-en basert kommunikasjon er vanligvis forbeholdt klienter fordi tjener-tjener trafikk ikke nødvendigvis er et resultat av at to tjenermaskiner snakker sammen. For at klienter skal kunne kommunisere på en sikker måte er det et krav at tjenerne sender meldinger i kun en retning i henhold til ”spanning tree”-strukturen, slik vil alle klienter nåes. Stien(”path”) en melding følger ved leveranse er den som gir kortest avstand mellom to noder i treet(”spanning tree”).

Kort følger noen eksempler som alle referer til figuren 3.3:

1. Meldinger mellom klientene 1 og 2 sees kun av tjener A som sender disse direkte fra klient 1 til klient 2.

2. Meldinger mellom klientene 1 og 3 sees av både tjenerne A og B. Ingen andre klienter eller tjenere vil kunne se disse meldingene.
3. Meldinger mellom klientene 2 og 4 sees av tjenerne A, B, C, D i tillegg til klient 4..

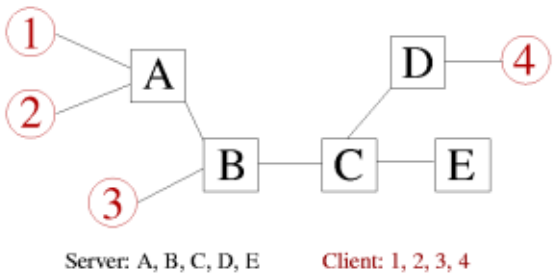
3.3.2.2 En-til-mange kommunikasjon

Til en liste Dette er den minst effektive formen for kommunikasjon innenfor en-til-mange kategorien. Kort fortalt gir klienten tjeneren en liste med destinasjonsadresser for en gitt melding, tjeneren bryter så ned listen og leverer separate kopier av meldingene til alle nevnte noder. Sammenlignet med masseutsendelse til en gruppe er denne løsningen mindre effektiv pga duplikat-faren(utsendelse fra listen blir ikke duplikasjons-sjekket).

Til en gruppe(kanal) Irc-kanaler er dynamiske av natur(klienter kommer til og andre faller fra, kanaler opprettes mens andre dør ut) og meldinger som utveksles på en kanal sendes kun til tjenere som som ser kanalen. Vi kan eksempelvis se for oss en kanal med mange klienter(alle klientene benytter seg av samme tjener). Hver melding som sendes vil kun mottas av tjeneren en gang og distribueres til alle klientene på kanalen. Dette gjentar seg for hver klient-tjener forbindelse helt til alle klientene på kanalen har mottatt meldingen.

Under følger noen relaterte eksempler som refererer til figuren 3.3.

1. *En gitt kanal som kun inneholder én klient.* Meldinger til kanalen sendes kun til tjeneren og ingen andre steder.
2. *En gitt kanal med to klienter.* Alle meldinger traverserer samme sti som om det skulle vært en privat dialog utenfor kanalen.



Figur 3.3: Et gitt IRC-nettverk

3. *Klientene 1, 2 og 3 befinner seg på samme kanal.* Alle meldinger som sendes til kanalen sendes også til alle klientene, men kun til tjenerne som traverseres av meldingen som om denne var en privat melding utenfor kanalen. Hvis klient 1 sender en melding vil denne sendes tilbake til klient 2 via tjener B til klient 3.

Til en maskin/tjener maske Denne funksjonen relateres primært til Irc-operatører. Det er mulig å kringkaste meldinger til et større antall klienter basert på maskinen/tjeneren de er tilkoblet. Meldingene sendes til maskiner eller tjenere som matcher nettmasken operatørene angir(f.eks *!@*test.no). Meldingene sendes kun dit hvor klientene er lokalisert, i likhet med kanaler.

3.3.2.3 En-til-alle kommunikasjon

En-til-alle meldinger beskrives best som kringkastings-meldinger som sendes til alle klienter eller tjenere, evt begge. Over store nettverk med mange klienter og tjenere kan en enkelt melding generere veldig mye trafikk i sitt forsøk på å nå alle ønskede destinasjoner.

Noen meldinger krever dog uten unntak kringkasting til alle tjenerne slik at tilstandsinformasjonen hver tjener forholder seg til er konsistent med hensyn på andre tjenerne.

Klient-klient Det finnes ingen klasse eller form for meldinger som, fra en enkel melding, resulterer i at meldingen blir sent til alle klientene.

Klient-tjener Meldinger som fører til en tilstandsendring(endring av kanalmodus, klientmodus etc) må i utgangspunktet sendes til alle tjenere og endringer i distribueringen er forbeholdt tjenerne.

Tjener-tjener I utgangspunktet vil all kommunikasjon mellom to tjenere kringkastes til ”alle” tjenere på nettverket. Opprinnelig er dette kun et krav hvis meldingene som sendes relateres til en klient, kanal eller tjener, men slik irc-protokollen fungerer vil nærmest alle meldinger fra en gitt tjener kringkastes til alle andre tilkoblede tjenere.

3.4 Brukergrensesnitt mot IRC

Irc-nettverket kan aksesseres på flere forskjellige måter. Tradisjonelt sett har dette blitt gjort ved å bruke teksbaserte klienter, men de senere årene har nye grafiske varianter økt i popularitet, dette inkluderer også web-chat løsninger som beskrevet i forrige kapittel. Siden irc ikke avhenger av plattform og operativsystem finnes det et utvalg klienter til de fleste operativsystemer. Utbredte klienter inkluderer *Irssi*(linux), *XChat*(linux/windows) og *mIRC*(windows). Fellesnevneren for disse er at de har et grafisk brukergrensesnitt mot IRC som gjør opplevelsen mer attraktiv, men ved bruk av Irssi har man i tillegg mulighet til å benytte klienten i ren tekstmodus. Hva som foretrekkes er smak og behag, uansett er rene tekstklienter ikke noe mindre funksjonelle enn de grafiske variantene.

Den senere tiden har vist at en ny trend er på vei. Som tidligere nevnt kan man snakke med andre chat-systemer ved plugins . Denne muligheten er også tilgjengelig mot irc-nettverket. Et godt eksempel er Bitlbee [BIT], en konsollbasert klient/gateway som lar brukerne kommunisere med andre chat-systemer(aim, msn og irc).

Oppsummering

Gjennom dette kapitlet har irc-systemet blitt introdusert som konkret forskningsdomene for oppgaven. Systemet har blitt beskrevet fra forskjellige perspektiv hvor relevant informasjon har blitt trukket inn.

Kapittel 4

Begreper og Teori

Teorigrunnlaget for denne oppgaven hentes i hovedsak fra fagfeltet Information Retrieval. Gjennom dette kapitlet kobles problemstillingen til nevnte fagfelt og det presenteres modeller og metoder som forsøker å løse relevante problemer.

4.1 Information Retrieval

Information retrieval(informasjonsgjenfinning) kan beskrives som et studie av systemer for indeksering, søk- og filtrering av data hvor spesielt tekst eller andre ustrukturerte former vektlegges. Ordene *information retrieval* utgjør et bredt og anvendelig begrep. Jeg vil gjennom denne oppgaven fokusere på automatiske IR-systemer. Automatisk som i motsetning til manuell og *information*(informasjon) som i motsetning til data.

Det har vist seg at ordet *information* i en IR-kontekst ikke nødvendigvis er entydig. Faktisk er det slik at i mange tilfeller vil en kunne beskrive typen av gjenfinning tilstrekkelig ved å bytte ut "informasjon" med "dokument". Likevel har "information retrieval" fått fotfeste som betegnende på det arbeidet Cleverdon, Salton, Sparck Jones, Lancaster og andre i sin tid publiserte. En mer nyansert og oppdatert problembeskrivelse av fagfeltet finner vi gjennom Bayeza-Yates [BYRN99] sin definisjon av IR:

> *Information Retrieval(IR) deal with the representation, storage, organisation of and access of information items. The representation and organization of the information items should provide the user with easy access to the information in which he is interested*

Vi kan oppsummere med at information retrieval er et fagfelt hvor man først og fremst ønsker å gjenfinne informasjon som er relevant i forhold til det informasjonsbehovet en bruker uttrykker. IR betraktes også ofte som ekvivalent med dokumentgjenfinning og tekstgjenfinning, men vi må ta høyde for at det eksisterer IR-systemer som leter frem bilder, lydfiler eller andre former for ikke-tekstlig informasjon.

Som beskrevet ovenfor vil denne oppgaven ha fokus på informasjon i motsetning til data. Det kan være nyttig å belyse disse formene for gjenfinning slik at vi får en forståelse av hva begrepet 'informasjon' innebærer. Det må også påpekes at dette er modeller, det er ikke slik at man enten gjør data- eller information retrieval.

Innenfor et IR-system vil data retrieval innebære å identifisere hvilke dokumenter som inneholder nøkkelord spesifisert via en spørring. En spørring er en brukerdefinert forespørsel oversatt til et spørreuttrykk mot det gjeldende systemet. Dokumenter som samsvarer i henhold til en spørring returneres etter å ha gjennomgått flere prosesseringssteg. Dette vil jeg komme nærmere tilbake til senere i oppgaven. Poenget er at data retrieval ikke tilstrekkelig tilfredsstiller informasjonsbehovet en bruker i utgangspunktet har. Benytter man seg av et IR-system er man trolig mer interessert i å innhente informasjon om et emne eller tema i motsetning til å innhente kun data som samsvarer med spørringen.

Data retrieval er best egnet når man ønsker eksakte treff i henhold til regulære eller algebraiske uttrykk. F.eks vil en liten feil i systemet føre til at prosessen kollapser, noe som ikke er tilfelle for information retrieval. Grunnen til dette er at information retrieval i hovedsak behandler tekst av naturlig språk og det er ikke nødvendigvis slik at denne er strukturert eller følger semantisk etikette. Hovedsaklig er det viktigst at det returneres data som er nyttig for brukeren og da tolereres en mindre feilrate fra tid til annen. Et tilsvarende scenario innenfor data retrieval, f.eks søk mot en relasjonsdatabase, betyr at man forholder seg til veldefinert og strukturert semantikk. Hvis en spørring returnerer et galt *record* blant flere tusen vil det føre til en alvorlig feil i spørresystemet.

Innenfor data retrieval kan man oppnå en grad av suksess bestemt ut fra hvor nøyaktig spørreuttrykkene formuleres. Mens innenfor en IR-kontekst står man ovenfor den intellektuelle utfordringen i å avgjøre hvilke dokumenter som er mest relevante i henhold til brukerdefinerte spørringer. Det er essensielt at systemet klarer å tolke dokumenter og rangere disse med relevans som ledende faktor. Tolkningen innebærer ekstrahering av syntaktisk og semantisk informasjon hvor denne brukes til å matche informasjonsbehovet brukeren uttrykte. Dette gir oss to sammenhengende utfordringer, hvordan skal informasjonen ekstraheres og hvordan kan denne brukes til å avgjøre relevans. Relevans er

uten tvil sentralt innenfor IR. Ideelt sett er det slik at man gjennom et IR-system innhenter alle relevante dokumenter, men samtidig utelater flest mulig irrelevante.

4.1.1 Bakgrunn

Det har vært arbeidet med IR siden 1940-tallet, men fortsatt eksisterer det et vedvarende problem rundt håndteringen av informasjonslagring og informasjonsgjenfinning. Kjernen i problemet har utspring i følgende faktum; vi har enorme mengder med informasjon tilgjengelig, men en presis og effektiv tilgang til denne informasjonen kompliseres. En effekt av dette er at relevant informasjon ignoreres og i noen tilfeller aldri blir avdekkes. Dette resulterer i duplikasjoner av tidligere arbeid. Som en følge av it-næringen og dens utvikling har man i dag raske og intelligente IR-systemer. Et konkret eksempel på dette finner vi rundt om i biblioteker. Her har man kraftige maskiner som benyttes i katalogiseringen.

I prinsippet er informasjonsgjenfinning og informasjonslagring relativt enkelt. Vi kan se for oss en større samling dokumenter hvor en gitt person har tilgang til dokumentsamlingen. Personen formulerer og uttrykker en spørring som resulterer i en mengde treff blant dokumentene. Brukeren må da manuelt lete gjennom hele dokumentsamlingen og plukke ut de mest relevante for å oppnå ønsket resultat. Det sier seg selv at dette er upraktisk og ressurssløsende.

Datamaskinteknologien har selvfølgelig vært banebrytende for IR fremgangen. Fortrinnsvis har man nå i større grad mulighet til å "lese" hele dokumentsamlinger samtidig som man velger ut de mest relevante dokumentene. Dog er det fortsatt en del kompleksistet forbundet med å analysere dokumenter. Input og lagringsproblemer har preget IR bildet opp gjennom årene, men selvfølgelig er det den intellektuelle utfordringen ved å karakterisere eller tolke dokumenter som har skapt størst hodebry. Det utvikles stadig nye og bedre teknikker for å ekstrahere informasjon av dokumenter, både syntaktisk og semantisk. Utfordringen ligger ikke bare i hvordan man skal ekstrahere denne informasjonen, men også hvordan den kan brukes til å avgjøre relevans.

Det er ikke uten grunn ordet *relevans* introduseres i en IR sammenheng. Som tidligere nevnt er ideen bak en automatisk gjenfinningsstrategi å innhente alle relevante dokumenter samtidig som alle irrelevante utelates. Ved å karakterisere dokumenter lager man samtidig en representasjon av disse, ofte referert til som tekstsurrogat. Er denne representasjonen relevant i forhold til en spørring skal følgelig det originale dokumentet returneres til brukeren. Tidligere, hvor man hadde mennesker som utførte indekseringen, karakteriserte man dokumenter ved å innfø-

	Indekstermer	Fulltekst	Fulltekst og struktur
Gjenfinning	Klassisk Mengdeteoretisk Algebraisk Probabilistisk	Klassisk Mengdeteoretisk Algebraisk Probabilistisk	Strukturert
Browsing	Flat	Flat hypertekst	Strukturstyrt hypertekst

Tabell 4.1: Hvordan gjenfinningsmodeller vanligvis assosieres med bestemte kombinasjoner av logiske perspektiv og brukeroppgave omarbeidet fra [BYRN99, side 21]

re såkalte indekseringsbetingelser. De som utførte indekseringen prøvde å forutse hva slags betingelser en bruker ville benytte seg av for å gjenfinne dokumenter. Implisitt konstruerte disse en spørring for hvor relevant hvert dokument var. Ved automatisk indeksering antas det følgelig at ved å mate dokumenttekst gjennom samme analyse vil det man får ut være en representasjon av innholdet. Om dokumentene er relevante eller ikke bestemmes således gjennom en maskinberegnet prosess.

4.1.2 Taksonomi av IR-modeller

Information retrieval er bredt og anvendelig fagfelt noe som uttrykkes gjennom de 15 forskjellige modellene som finnes. I praksis er det dog tre modeller som skiller seg ut, bedre kjent som de klassiske modellene. Henholdsvis er dette den bolske modellen, vektormodellen og sannsynlighetsmodellen. En klassifisering av samtlige modeller finner vi i figuren 4.1, som forøvrig er en speiling av klassifiseringen utarbeidet av Baeza-Yates og Ribeiro-Neto. Av figuren ser vi at de tar utgangspunkt i brukerens oppgaver og sorterer klassifiseringen deretter. Det er også fremtredende at dokumenter kan observeres på forskjellige måter ved å innføre det man kaller *logiske perspektiv* (logical views). Et eksempel på et slikt logisk perspektiv er indekstermer, gjerne i form av beskrivende ord(nøkkelord) som representerer det opprinnelige dokumentet og dets struktur. I tillegg til å representere tekst kan også avsnitt, overskrifter og deloverskrifter skilles ut. Det eksisterer selvsagt mange slike logiske perspektiv, men dette avhenger av hva man ønsker å gjøre med dokumentet. Det presenteres også en tabell som illustrerer hvordan modeller for gjenfinning kobles mot bestemte kombinasjoner av logiske perspektiv og brukeroppgave. Tabell 4.1 er en representasjon av Bayeza-Yates sin fremstilling.

- **Gjenfinning: Ad hoc eller filtrering**
  - **Klassiske modeller:** De tre klassiske modellene er henholdsvis den bolske modellen, vektormodellen og sannsynlighetsmodellen(probabilistiske). Disse kjennetegnes ved følgende:
    - **Mengdeteoretiske:** Fuzzy sett, utvidet bolsk metode
    - **Algebraiske:** Generaliserte vektorer, latent samantiske, nevrale nett
    - **Probabilistiske:** Inferens nett, *belief* nettverk
  - **Strukturerte modeller:** Foruten standardstrukturen med referanser til dokumenttekst-en(indekstermer) finnes det også varianter av dette
    - **Ikke-overlappende lister:** Hierarkisk dekomponering hvor hvert nivå av hierarkiet representerer en oppdeling uten overlapp av sin del av dokumentet
    - **Nærliggende noder:** Nærhet mellom termer(ord) eller andre komponenter brukes som et utgangspunkt for navigasjon
- **Browsing:**
  - **Flat:** Termene i dokumentet er en sekvens av symboler uten noen ytterligere struktur
  - **Strukturstyrt:** Hierarkisk dokumentstruktur brukes for navigering
  - **Hypertekst:** Det eksisterer lenker på tvers av hierarkisk struktur og dokumenter. Lenkene kan følges av en gitt bruker

Figur 4.1: Taksonomi av IR-modeller omarbeidet fra [BYRN99, side 21]

4.1.3 Gjenfinning: Ad hoc og Filtrering

Det finnes to typer gjenfinning som begge er kjennetegnende for tradisjonelle IR-systemer, henholdsvis:

1. *Ad hoc:* Brukeren har en bestemt oppgave og skal forsøke å finne dokumenter som er relevante i henhold til denne oppgaven. Dette har forøvrig utspring i at dokumenter er relativt statiske så lenge de behandles innenfor rammene av systemet

2. *Filtrering:* Oppgaven er konstant, men strømmen av tilgjengelige og relevante dokumenter er dynamisk. De endres over tid. Et konkret eksempel på bruk av filtrering finner vi gjennom løsninger tilknyttet online-aksjehandel(varslinger trigges når det skjer endringer).

   Det finnes også en variasjon av filtrering kalt *ruting*. I tillegg til å plukke ut dokumenter som kan være relevante kan disse rangeres ut fra kriterium som antas å beskrive hvor godt dokumentet passer til brukerens informasjonsbehov

4.1.3.1 Dokumentgjenfinning

Dokumentgjenfinning omfatter to relaterte aktiviteter, *indeksering* og *tekstgjenfinning*. Indekseringen referer til metoder som benyttes for å lagre dokumenter på en slik form at disse skal kunne søkes gjennom mer effektivt. Søkene er oversatt til søkeuttrykk spesifisert via selvstendige spørringer av brukerne. Målet er at søkeuttrykkene skal returnere dokumenter som er relevante i henhold til den opprinnelige spørringen. "Relevans" er som tidligere nevnt et viktig begrep i en IR-kontekst. Selv om definisjonen av begrepet ikke er entydig er fortsatt den generelle oppfatningen at relevante dokumenter er de som omfavnes av gitte spørring og dermed er nærmest det informasjonsbehovet som ble uttrykt.

Det eksisterer også et nettverk av metoder for å evaluere søk(spørringer) og dermed også den underliggende algoritmen som benyttes. En delvis standardisert metode som brukes tar utgangspunkt i hvordan automatiske søk oppfører seg i forhold til manuelle søk "under best mulige forhold". Dette er en relativt krevende prosess ettersom hvert spørreuttrykk må evalueres av en person. Ideen er at en personen generer et spørreuttrykk og evaluerer alle dokumentene i den aktuelle dokumentsamlingen. De dokumenter som er relevante plukkes ut i henhold til de kriterier som er lagt til grunne ved spørreuttrykket. Den samme prosessen gjennomføres også med søkealgoritmen som drivkraft i systemet.

Den returner en liste over dokumenter den mener er relevante. Ved å sammenligne de manuelle resultatene med de som ble presentert av søkealgoritmen kan man beskrive algoritmens ytelse langs to sammenhengende dimensjoner:

Presisjon Andelen relevante dokumenter presentert av søkealgoritmen

Gjenfinning Andelen relevante dokumenter som fantes i dokumentmengden og som faktisk ble funnet

På grunn av høy grad av subjektivitet i evalueringen og fordi dokumentsamlingene kan variere størrelsesmessig har man forsøkt å samarbeide rundt evalueringen av de forskjellige søkemetodene. TREC(Text Retrieval Conference) [TRE] blir betegnet som et referanseforum for samordning av ressurser for evaluering av IR og effektivitet, nærmere bestemt søkealgoritmene for tekst. Dokumentsamlingene velges vanligvis ut fra kjente tidsskrifter. Spørringer blir utformet ved at man søker gjennom dokumentsamlingene etter bestemte emner. Spørringer og dokumentsamlinger distribueres til de forskjellige deltakerne som bruker bestemte algoritmer i jakten på dokumenter som samsvarer med en gitt spørring. Rangerte lister med dokumenter returneres til brukeren hvor disse evalueres med hensyn på relevans av den samme personen som skrev spørringen.

4.1.4 Gjenfinningsstrategi

Gjennom innledningen av oppgaven ble chat-systemet irc spesifisert som konkret forskningsdomene. Informasjonsgjenfinning brukt mot dette systemet innebærer to relaterte aktiviteter: Ad hoc og Filtrering. Det er gitt at søk mot irc-dialoger er interessant for denne oppgavens omfang, men hvor interessant vil det være å implementere en filterløsning.

Belkin og Croft [BC92] har funnet frem til flere kriterier som betegner filtersystemer. Under beskrives disse nærmere og for hvert punkt relateres de til denne oppgaven. Dvs, hvor godt passer disse i en chat-kontekst og hvor rimelig er det å se for seg en slik løsning.

- *Et filtersystem er designet med formålet om å behandle ustrukturert eller delvis ustrukturert data*

  Chat-dialoger er i høyeste grad ustrukturert data. Det finnes ingen form for organisering av dialogene, hvis vi ser bort i fra kanaltilhørighet.

- *Filtersystemer opererer vanligvis mot tekstlig informasjon. Faktisk er det slik at ustrukturert data brukes som synonym på tekstlig data*

  Chat-dialogene representeres som vanlig tekststrenger. Spesialtilfelle er når kryptering benyttes.

- *Filtersystemer håndterer vanligvis store mengder data*

  Chat-dialogene har ingen øvre grense størrelsesmessig. Kanaler og tilhørende dialog inngår et i ubredt og dynamisk hierarki hvor størrelse og begrensninger er abstrakte begreper

- *Filterapplikasjoner involverer typisk datastrømmer, gjerne kringkastet via eksterne kilder eller sendt direkte som f.eks e-post*

  Chat-dialogene kan sees på som datastrømmer hvor de uke ulike nettverkene/tjenerne man er koblet mot opptrer som eksterne kilder.

- *Filtrering baseres vanligvis på individuelle eller gruppebaserte profiler. Profilene skal i utgangspunktet speile brukernes ønsker og informasjonsbehov*

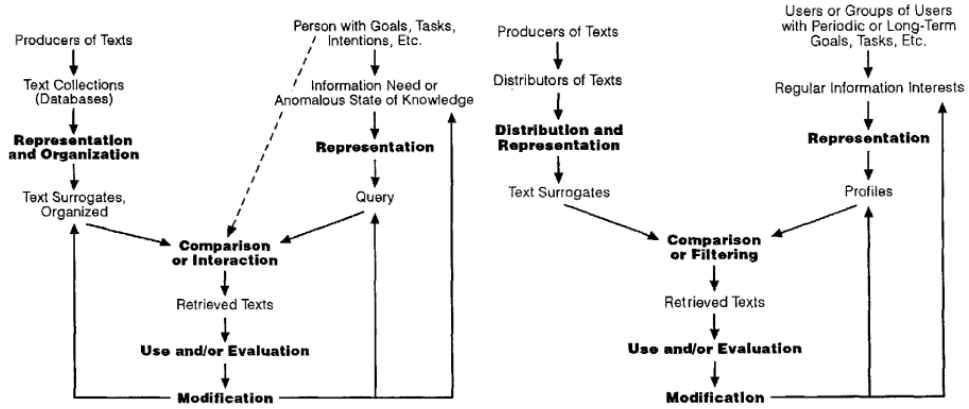
  Det vil være fullt mulig å opprette profiler som ivaretar brukernes informasjonsbehov. Dette reflekteres ved at man filtrerer ut informasjon fra chat-dialogene basert på brukerens preferanser

- *Filtrering innebærer ofte fjerning av data fra en innkommende datastrøm i motsetning til å finne data i samme strømmen. Et eksempel på førstnevnte har vi gjennom e-post filtrering(spam). Slik kan profilene også brukes til å uttrykke hva som ikke ønskes*

  I en chat-kontekst er det mest naturlig at man finner frem til ønsket informasjon.

Vi kan oppsummere med at det vil være interessant å implementere en løsning som inkluderer både søk- og filterfunksjonalitet. Begge disse formene for gjenfinning er knyttet tett opp mot hverandre og det er ikke nødvendigvis slik at man begrenses til en av dem. Belkin og Croft utformet også to abstrakte modeller som plasserer ad hoc og filtrering i IR-domenet. Under følger en nærmere beskrivelse av disse og hvordan de passer inn i denne oppgaven. Figuren 4.2 representerer modellene.

Figuren 4.2(a) representerer ad-hoc gjenfinning. En bruker søker informasjon og uttrykker informasjonsbehovet gjennom en spørring, som i denne oppgaven vil være mot chat-dialoger. På venstre side av figuren får vi et kort overblikk over de ressurser brukeren benytter seg av under IR-prosessen. Dette inkluderer tekst/dokument-forfatterne, dokumentkolleksjoner , dokumentrepresentasjoner og organisering av disse i



(a) Ad-hoc (b) Filtrering

Figur 4.2: Modeller for gjenfinning fra [BC92, side 31]

dokumentsurrogater. Dokumentsurrogater er som nevnt tidligere en fellesbetegnelse på dokumenter som er satt sammen av indekstermer eller nøkkelord.

Ved å sammenligne spørringer og surrogater evt se på direkte koblinger mellom brukerne og dokumentene eller surrogatene danner dette grunnlaget for valg av sannsynlig relevante dokumenter. Det er opp til brukeren å bestemme hvor nyttig gjenfunnede dokumenter faktisk er. Gjenfunnede dokumenter kan også evalueres gjennom en "feedback"-prosess med formålet om å forfine spørringen og derav informasjonsbehovet evt surrogatene.

Figuren 4.2(b) representerer filtermodellen. Filterprosessen instansieres ved en mengde brukere med statiske mål/ønsker som spenner over en lengre tidsperiode. Disse langsiktige målene organiseres i egne brukerprofiler og det skilles ikke på enkeltpersoner og grupper, begge kan karakteriseres på en slik måte. Dette leder til interesse. Man ønsker typisk å følge utviklingen av et gitt område som endres over tid ettersom forutsetninger, mål og kunnskap også forandres. For denne oppgavens formål er dette området chat-systemet IRC. "Interesse-modellen" fører til en passivisering av brukerne hvor relevant tekst presenteres fortløpende. Fra et IR-perspektiv er passiviseringen en positiv effekt siden automatiseringen er resultatet av uttrykk for informasjonsbehov.

På venstre side av figuren finnes tekstprodusentene. Dette f.eks i form av aviser, elektronisk børshandel eller irc-tjenere som for denne oppgaven, men også enkeltindivider. Målet er å distribuere tekst kontinuerlig og presentere denne for brukerne. Dette gjennomføres ved å sammenligne tekstsurrogatene mot personlige- og gruppebaserte profiler. Det er

opp til brukerne om teksten ansees for å være relevant eller om denne forkastes. Vanligvis evalueres også teksten slik at det taes høyde for eventuell forfining i profilene.

Ved å sammenligne figurene i 4.2 ser vi at ad hoc og filtrering har mange likhetstrekk, i noen tilfeller kan det virke som de nesten er identiske. Siden teorien gjennom denne oppgaven i hovedsak hentes fra IR og både ad hoc og filtrering er interessante implementasjonsløsninger vil det være nødvendig å skille disse formene for IR fra hverandre. Under følger en beskrivelse av de mest vesentlige forskjellene.

- *Ad hoc-modellen betrakter bruken av systemet som en engangsopplevelse hvor brukerne har én bestemt målsetning og spørring typisk stilles én gang. Filtermodellen er mer fokusert rundt gjenbruk av systemet hvor brukerne har langsiktige mål og interesser*
- *Innenfor en ad hoc-kontekst kan man stille seg spørsmål om hvor godt en spørring representerer informasjonsbehovet. Ved bruk av filtermodellen antar man at dette løses bedre ved bruk av profiler*
- *Hvor ad hoc fokuserer rundt kolleksjoner og organisering av tekst konsentreres filtrering rundt distribuering av tekst til grupper eller enkeltindivider*
- *Ad hoc baseres på utvalgt tekst fra statiske dokumentkilder mens filtrering fokuseres rundt utvelgelse eller eliminering av tekst fra dynamiske datastrømmer*
- *Innenfor rammene av en ad hoc-modell er man opptatt av brukerrespons innenfor et enkelt scenario mens innenfor filtrering er man mer opptatt av langsiktige endringer over flere scenarier*

I tillegg til ovennevte finnes det andre kontekstuelle forskjeller som er direkte relaterte til ad hoc og filtrering. Disse har utspring i sosiale og/eller praktiske situasjoner og kan kategoriseres i henhold til ulikhet mellom teksten(dokumentene), brukerne og miljøet. En fellesbetegnelse på dette er *sosial filtrering*.

1. *Tekstrelatert* - Innen filtrering er det fremtredende at ny tekst gjøres tilgjengelig for systemet til ”rett tid”(straks den er tilgjengelig). Eksempler på ny tekst er dynamiske dokumentstrømmer man allerede utfører filtrering mot. Innenfor ad hoc har dette ikke vært tema
2. *Brukerrelatert* - Innenfor ad hoc har man vanligvis studert veldefinerte brukergrupper og domener innen teknologi og vitenskap. I en

filtrerkontekst har man ofte å gjøre med udefinerte brukergrupper og domener. I tillegg kan man innenfor filtrering ikke alltid se for seg noen motivasjonsfaktor

3. *Miljørelatert* - Ved filtrering forsøker man så langt det lar seg gjøre og ikke forstyrre privatliv eller krysse moralske grenser. Innenfor en ad-hoc-strategi har dette ikke vært tema

Vi kan oppsummere med at begge disse formene for gjenfinning i realiteten er to sider av samme sak. Felles målsetning er å hjelpe brukerne til å finne frem relevant informasjon. Både ad hoc og filtrering er interessante for denne oppgavens omfang. Hvordan dette integreres i implementasjonen beskrives senere i oppgaven.

4.1.5 Definisjon av IR-modeller

For å definere modeller for relevans introduseres følgende struktur som er direkte sakset fra Modern Information Retrieval [BYRN99]:

> ***Definition*** *An information retrieval model is a quadruple* $[\mathbf{D}, \mathbf{Q}, \mathcal{F}, R(q_i, d_j)]$ *where*
>
> 1. $\mathbf{D}$ *is a set composed of logical views(or representations) for the documents in the collection*
> 2. $\mathbf{Q}$ *is a set composed of logical views (or representations) for the user information needs. Such representations are called queries*
> 3. $\mathcal{F}$ *is a framework for modelling document representations, queries and their relationships*
> 4. $R(q_i, d_j)$ *is a ranking function which associates a real number with a query* $q_i \in \mathbf{Q}$ *and a document representation* $d_j \in \mathbf{D}$*. Such ranking defines an ordering among the documents with regards to the query* q_i

Det vi kan merke oss er at innenfor rammene av denne oppgaven vil kun komponentene D og Q være direkte synlige siden Fast Data Search benyttes som underliggende plattform. Dette betyr at $\mathcal{F}$ og R komponentene representeres ved FDS-systemet og all kommunikasjon på laveste nivå skjermes bort, men benyttes likevel indirekte gjennom ferdigutviklede API'er. Dette vil jeg komme tilbake til senere i oppgaven. Det man uten videre bør merke seg er slik Bayeza-Yates påpeker at denne *DQFR-modellen* kan benyttes innenfor ad-hoc og filtrering og at det derfor er slik at modeller for IR i stor grad kan sees på som uavhengig av bruksområdet.

4.1.6 Modeller for relevans

Jeg vil nå presentere de klassiske modellene for analyse/sammenligning av dokumenter som eksisterer innenfor IR. Disse bringer oss videre fra nøkkelordtreff ideene som beskrevet tidligere. Følgende modeller er aktuelle:

- *Bolske modellen* - Dokumenter og spørringer representeres ved mengder av indekstermer, derfor blir modellen referert til som mengdeteoretisk
- *Vektormodellen* - Dokumenter og spørringer representeres som vektorer i et t-dimensjonalt rom, derav en algebraisk modell
- *Sannsynlighetsmodellen* - Baseres på sannsynlighetsteori (probabilitetsteori) og er derfor en sannsynlighetsmodell

De tre klassiske modellene beskriver dokumenter ved indekstermer, også referert til som 'nøkkelord'. Som nevnt tidligere er en indeksterm et beskrivende ord i dokumentteksten, med andre ord ønsker man å oppsummere dokumenter ved å velge passende indekstermer. Vanligvis er indekstermer substantiver på grunn av deres semantiske betydning, et substantiv gir mer mening når det skilles ut av dokumentteksten enn f.eks adjektiver og adverb. Dog kan det være interessant å betrakte alle ordene i et dokument som indekstermer slik noen søkemotorer gjør [BYRN99].

Etter å ha stadfestet hvilke ord som egner seg best som indekstermer evalueres disse, naturlig nok er ikke alle indekstermer like gode. Denne prosessen ved selektiv rangering av indekstermer er ingen enkel oppgave, men som et utgangspunkt har man "logiske regler" å forholde seg til. La oss eksempelvis si at vi har en dokumentkolleksjon med en stor mengde dokumenter. Hvis en indeksterm opptrer i samtlige av disse dokumentene er den totalt ubrukelig fordi den ikke klarer å skille dokumentene fra hverandre, mao en spørring vil ikke resultere i unike treff. Derimot vil et ord som bare opptrer i 4-5 dokumenter være langt mer interessant fordi dette gir en innsnevret treffprosent i henhold til en relatert spørring(ref. presisjon og gjenfinning). Ut fra dette er det tydelig at indekstermer ikke kan betraktes likt fordi grad av relevans vil variere. En metode som har blitt tatt i bruk for å bøte på dette er innføringen av *vekting*. Man vekter hver indeksterm i et dokument mhp på relevans.

Som en avrunding og bekreftelse på det som har blitt sagt kan det nevnes at Salton med flere tidlig utviklet et verktøy for å klassifisere en indeksterm som "god" eller "dårlig". De antok at en god term var en som skilte dokumentene betydelig fra hverandre, mens en dårlig term betraktet de som like.

4.1.6.1 Bolske modellen

Den bolske modellen baseres på mengdeteori og bolsk algebra. Fordelene er at mengdeteori er intuitivt og alle spørringer spesifiseres på bolsk form. Ulempen er at modellen styres av en rigid gjenfinningsstategi hvor dokumenter defineres som relevante eller irrelevante. Det eksisterer ingen form for nyansering noe som over tid fører til svakere gjenfinning. Et annet dilemma har utspring i spesifiseringen av spørringene. Til tross for at bolske uttrykk spesifiseres ved detaljert semantikk er det ugunstig å formulere informasjonsbehov gjennom bolske uttrykk. Selv om modellen bærer preg av ugunstighet blir den allikevel brukt av blant annet kommersielle leverandører av databasesystemer.

Modellen betrakter forøvrig indekstermer som tilstedeværende eller fraværende i et dokument, dette resulterer i binærvekting av indekstermer.

Vi kan oppsummere med at modellens fordeler er de formalismer som ligger til grunne og den enkle strukturen. Ulempene er mangel på nyanseringer under gjenfinningsfasen som kan medføre for mange eller for få dokumenter.

4.1.6.2 Vektormodellen

Vektormodellen betrakter binære vekter som utilstrekkelige. Istedet åpner man for nyanseringer ved å vekte indekstermene.

Gitt en samling dokumenter kolleksjon X og relaterte spørringer som et sett Q. I en IR-kontekst kan problemet brytes ned til følgende, hvilke dokumenter eksisterer i Q og hvilke gjør det ikke. Dette løses for det første ved å identifisere hvilke egenskaper som best beskriver spørringene i Q og for det andre hvilke egenskaper som best skiller disse fra de gjenværende objektene i kolleksjonen X. Vektormodellen betrakter både dokumenter og spørringer som samlinger av ord og for hvert ord lages en n-dimensjonal ”sparse” som representerer dokumentet. Dersom dokumentet har 30 forekomster av ordet ”bil” inneholder vektoren verdien tredve for indeksen ”bil”. Sammenligninger mellom dokumenter og spørringer betraktes da som sammenligninger av vektorer. Innen IR har det i praksis vist seg at kombinasjoner av vekter basert på termfrekvens(tf), invers dokumentfrekvens(idf) og sammenligninger basert på cosinus av vinklene mellom dokumentenes vektorrepresentasjoner har gitt gode resultater.

Vi kan oppsummere med at det som taler til vektormodellens fordel er måten den håndterer termvekting som resulterer i effektivisert gjenfinning. Den innehar også en nyansert gjenfinningsstrategi (dokumenter er

ikke bare relevante eller irrelevante) samt likhetsrangering av dokumenter. Teoretisk sett er ulempen at indekstermer er gjensidig uavhengig av hverandre, men i praksis er termavhengigheter ingen fordel. Faktisk kan dette ha innvirkning på ytelsen til systemet. Vektormodellen er i dag et populært valg og rangeringen i Fast Data Search er blant annet basert på tf/idf.

4.1.6.3 Sannsynlighetsmodellen

Sannsynlighetsmodellen ble først introdusert i 1976 av Robertson og Spark Jones [RJ76] og ble senere kjent som BIR-modellen(Binary Independence Retrieval).

Gitt en spørring og et sett av relevante dokumenter. Dette blir innenfor modellen betraktet som det ideelle dokumentsettet. Basert på det ideelle dokumentsettet forsøker man finne frem til relevante dokumenter. Problemet er bare at man ikke kjenner til disse egenskapene på forhånd, Men det man vet er at indekstermer kan brukes til å uttrykke disse egenskapene. Vanligvis er første trinn i denne prosessen å gjette. Ved å gjette former man også en løs beskrivelse av det ideelle dokumentsettet, mer spesifisert, en sannsynlig beskrivelse. På bakgrunn av denne gjettingen gjenfinnes også et første sett av dokumenter som er forhandlingsgrunnlag for den videre jakten på det ideelle dokumentsettet. Ut fra gjenfunnede dokumenter velger brukeren selektivt hvilke som er mest relevante og hvilke som ikke er det. IR-systemet redefinerer beskrivelsen av det ideelle dokumentsettet på bakgrunn av denne utvelgelsen. Ved å gjenta prosessen forventes det en gradvis utvikling og tilnærming mot beskrivelsen av det ideelle dokumentsettet. Man har også forsøkt å transformere beskrivelsen i form av sannsynlighetsteori. Modellen baseres på følgende antagelser [BYRN99]:

Man har en gitt spørring q og et dokument d. Via sannsynlighetsmodellen estimeres sannsynligheten for at brukeren vurderer dokumentet som relevant. Innenfor modellen antas det også at denne sannsynligheten kun avhenger av spørringen og dokumentet. Videre antas det at alle dokumenter består av et sett underliggende dokumenter som foretrekkes av brukeren, dvs disse tilfredsstiller spørringen q. Dette ideelle settet markeres som R(relevant), derav sees dokumenter i R på som *relevante* mens dokumenter som ikke finnes der betraktes som *irrelevante*.

Kort kan vi oppsummere med at fra et teoretisk perspektiv er fordelene ved denne modellen at dokumenter rangeres i synkende rekkefølge etter sannsynlighet av relevans. Ulempene er at den i utgangspunktet baserer relevans på gjetting i tillegg har den ikke noe forhold til termfrekvens.

4.1.6.4 Kort sammenligning

Man er av den oppfatning at den bolske modellen er den svakeste av de klassiske modellene. Dette fordi den ikke innehar noen form for nyanseringer eller delvis treff som det også kalles. Dokumenter sees kun på som relevante eller irrelevante. Angående de to siste modellene har det vært mye debatt rundt hvilken som foretrekkes. Blant annet har Croft, gjennom eksperimenter, kommet frem til at sannsynlighetsmodellen vil føre til effektivisert informasjonsgjenfinning. Men i ettertid har Salton og Buckley gjennomført eksperimenter som motstrider dette. De mente at vektormodellen *forventes* å være mer slagkraftig enn sannsynlighetsmodellen. Dette synet har tydeligvis mange sett sin enighet i derfor domineres IR-samfunnet av forskere og et web-miljø hvor vektormodellen har fått fotfeste som ledende.

4.1.7 Automatisk tekstanalyse

H.P Luhn, en av pionerene innenfor IR, antok at man kunne benytte seg av frekvensanalyse for å skille ut 'verdifulle' ord av dokumentteksten og derav generere fullverdige representasjoner av det opprinnelige dokumentet. Dette innebar blant annet at høyfrekvente ord kunne ekstraheres fra teksten fordi disse var av mindre semantisk betydning. Han kom også frem til at én eller to forekomster av et gitt ord i et stort dokument ikke kunne sees på som nyttig i forhold til å beskrive det originale dokumentet. Derfor foreslo han å bruke ordene i midtpartiet av frekvensspekteret. 'Vanlige' ord ble fjernet ved bruk av en spesiell ordliste som inneholdt såkalte "stopp"-ord. Men også ved å skille ut lavfrekvente ord, dvs ord som kun eksisterte én eller to evt flere steder i teksten, avhengig av tekststørrelsen.

Dette var bare starten på det vi i dag kjenner som automatisk tekstanalyse innenfor IR. Utviklingen innenfor nevnte domene har båret frukter og under følger en generell beskrivelse av hvordan en slik analyse gjennomføres i dag. I henhold til denne oppgavens omfang og begrensninger er denne prosessen noe jeg ikke vil ha direkte kontakt med, men den er fortsatt en del av Fast Data Search produktet og derfor relevant. Om ikke helt identiske, vil jeg nevne hva som gjøres annerledes i FDS-systemet.

4.1.8 Preprosessering av dokumenter

En typisk tekstanalyse deles normalt inn i fem selvstendige tekstoperasjoner [BYRN99]. Dette innebærer:

1. Leksikalsk analyse av teksten - Hvordan splitte opp i ord(tokenisering)
2. Eliminering av "stopp"-ord - Ord av liten gjenfinningsverdi identifiseres og fjernes
3. *Stemming* - Ord brytes ned til rotformen(f.eks kjørende -> kjøre)
4. Valg av indekstermer - Hvilke ord/ordstammer benyttes som indekselementer. Dette avhenger i stor grad av ordenes syntaktiske natur. Et substantiv er f.eks av mer semantisk verdi enn adjektiver og adverb
5. Konstruksjon av kategoriserende strukturer - Generering av *thesaurus*, en form for avansert ordliste

4.1.8.1 Leksikalsk analyse

Gjennom en leksikalsk analyse bryter man ned dokumentene i ord slik at det blir lettere å bestemme hvilke som egner seg best som indekstermer. En indeksterm er som tidligere nevnt et ord eller nøkkelord som er beskrivende for innholdet i et dokument. Vanligvis er indekstermer substantiv eller substantivgrupper.

Under en slik analyse er det spesielt fire tilfeller som betraktes som vesentlige. Dette inkluderer behandling av tall, bindestreker, punktum og små- og store bokstaver.

4.1.8.2 Fjerning av "stopp"-ord

Ord som opptar 80 prosent eller mer av dokumentteksten sees på som verdiløse i forhold til gjenfinningsprosessen. Disse ordene refereres til som "stopp"-ord og er vanligvis preposisjoner, konjunksjoner osv.

Luhn implementerte blant annet en 'øvre grense' i frekvensanalysen slik at man kunne fjerne "stopp"- eller høyfrekvente ord. Fordelen er uansett at man vil redusere størrelsen på dokumentene drastisk, et indeksert dokument kan ha en størrelsesreduksjon på inntil 40 prosent. Siden elimineringen av "stopp"-ord har hatt en slik omfattende innvirkning på dokumentstørrelsen har det også blitt vurdert å filtrere ut andre ord som "stopp"-ord, deriblant verb, adverb og adjektiver.

Til tross for positiv effekt har man funnet ut at eliminering av "stopp"-ord kan ha negativ innvirkning på gjenfinningen(recall). F.eks vil en setning som "å være eller ikke være" strippes fullstendig og man sitter kun igjen med "være" som gjør det nærmest umulig å identifisere hvilket dokument teksten opprinnelig tilhører.

4.1.8.3 Stemming

Et typisk scenario er at man via en spørring søker etter relevante ord i dokumenter. I noen tilfeller vil bare varianter av disse ordene eksistere. Flertallsform, fortid og nåtid er syntaktiske variasjoner av samme ord og for å få treff i henhold til en spørring må man bruke metoder som tar høyde for dette. En slik metode kalles stemming, dvs man bryter ned ordet til den opprinnelige ordstammen og matcher på denne isteden. Et eksempel på en ordstamme er ”løpe”. Varianter av ordstammen er ”løpende”, ”løper” og ”løpte”. Bruk av stemming fører til effektivisert gjenfinning fordi man ved ordstammer også representerer mulige varianter av ordene. En bieffekt er dog at man får færre indekstermer å forholde seg til som igjen resulterer i en størrelsesreduksjon på f.eks thesaurus(beskrives senere).

Selv om stemming i utgangspunktet vil føre til effektivisert gjenfinning finnes det også skeptikere. F.eks har Frakes [FBY92] gjennomført forskjellige typer studier som i utgangspunktet skal bygge opp under stemming, men resultatet av testene var ikke spesielt overbevisende. Faktisk har det seg slik at utviklere av søkemotorer har latt være å implementere noen form for stemming-algoritmer med bakgrunn i disse resultatene.

I Fast Data Search bruker man en nær tilknyttet metode som kalles *lemmatisering*. Man bryter først ned ord til rotformen for så å ekspandere disse til alle mulige former.

4.1.8.4 Valg av indekstermer

På grunnlag av noen grammatiske forhåndsregler plukker man ut de ord som best representerer dokumentet, manuelt eller automatisk, hvor sistnevnte er mest naturlig. Forøvrig har man kommet frem til at substantiver innehar mest semantisk verdi og det ønskes derfor at disse benyttes som indekstermer. Ut fra dette kan man velge en strategi som automatisk fjerner adjektiver, adverb og pronomen.

Med tanke på at det er vanlig å kombinere to eller tre substantiver i en dokumenttekst har man kommet frem til en løsning hvor man betrakter disse som *substantivgrupper*. Substantivgrupper defineres [BYRN99] som en samling av substantiver hvor deres syntaktiske avstand i dokumentteksten, som måles ved å se på antall ord mellom to substantiver, holdes innenfor rammene av et gitt *threshold*, f.eks 3.

4.1.8.5 Thesaurus

Thesaurus er en avansert variant av det vi kjenner som 'ordliste'(ord og synonymer). Thesaurus behandler alt vokabular gjennom en normaliseringsprosess og innehar derfor en langt mer kompleks struktur.

Motivasjonsfaktoren for generering av thesaurus har utspring i ønsket om et *kontrollert vokabular* for søk og indeksering. Med kontrollert menes et vokabular som tar for seg normalisering av indekskomponenter, støyreduksjon, identifisering av indekstermer og konseptbasert informasjonsgjenfinning fremfor ordbasert.

4.1.8.6 Clustering

I tillegg til ovennevnte finnes det en nær tilknyttet form for "tekstoperasjon" kalt *clustering*. Clustering er i realiteten ingen operasjon mot tekst, men heller en operasjon mot dokumentkolleksjoner. Man deler forøvrig clustering inn i to forskjellige klasser; *globale* og *lokale*. Innenfor en global clustering-strategi grupperes dokumentene i henhold til deres plassering i kolleksjonen. Innenfor en lokal strategi påvirkes grupperingen kun av konteksten definert ved gitte spørring og den *lokale* mengden gjenfunnede dokumenter.

Clustering brukes først og fremst for å mappe spørringer til et uttrykk for informasjonsbehov. Fra et slikt perspektiv er operasjonene mer rettet mot transformering av spørringer i motsetning til å transformere dokumenttekstene.

Oppsummering

Informasjonsgjenfinning har gjennom dette kapitlet blitt beskrevet som teoretisk grunnlag for oppgaven. Ad-hoc og filtrering, to former for gjenfinning, har blitt introdusert som interessante innfallsvinkler for implementasjonen. Vi har også sett på diverse IR-modeller og hvordan en typisk tekstanalyse knyttet til disse gjennomføres.

Kapittel 5

Relatert Arbeid

Gjennom dette kapitlet presenteres et knippe arbeider som er relevante for oppgavens problemstilling. Dette er i hovedsak ferdigutviklede eller prototyper på eksisterende systemer. Systemene blir først presentert ut fra struktur og funksjonalitet og deretter analysert i forhold til hvilke elementer eller ideer som brukes eller kunne vært brukt mot implementasjonen i denne oppgaven.

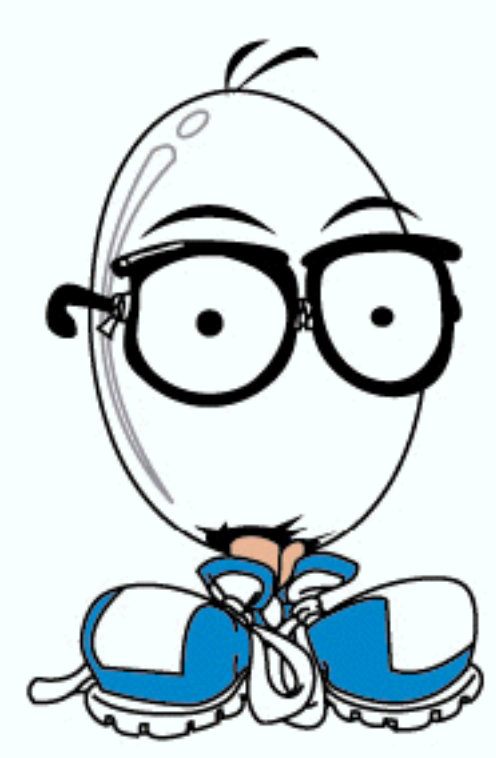
Arbeidene som presenteres er følgende:

1. *Eggdrop:* Et system som assisterer brukere av irc-nettverket. Systemet har flere bruksområder og kan annet brukes til å administrere kanaler og logge aktivitet
2. *Amalthea:* Et multi-agent system som finner frem til, overvåker og filtrerer informasjon
3. *Butterfly:* En avansert agent brukt på irc-nettverket. Agenten forsøker å søke frem til kanaler som kan være av interesse for brukeren basert på kriterier relatert til atferd
4. *Alternative Interfaces for Chat:* Et initiativ myntet på forbedret grafiske representasjoner av irc-dialog. Prototyper på aktuelle systemer beskrives

5.1 Eggdrop

Eggdrop[EGG] er en sofistikert og avansert IRC-robot eller "bot" som de ofte tiltales som. En IRC-bot er system som i hovedsak brukes for å kontrollere og administrere IRC-kanaler. En eggdrop kan ved første øyekast

minne mye om en normal bruker, så lenge den ikke blir direkte adressert av noen av brukerne evt satt til å utføre automatiske oppgaver innehar den en tilstand som refereres til som ”idle”. Slike bot’er ”styrer” kanaler ved å gi brukere operatørstatus, endre kanalattributter og fjerne uønskede brukere alt ettersom eierne har programmert den til å gjøre. En eggdrop kan i tillegg logge kanaldialogene(som manuelt kan søkes gjennom), tjene funksjonen som informasjonsbase evt modifiseres/tilpasses ved bruk av egne moduler. IRC-roboter er spesielt egnede på nettverk hvor man ikke kan registrere kanaler som f.eks EFNet og IRCnet. De kan også være nyttige på andre nettverk hvor registreringsprosessen av kanaler ikke er spesielt triviell, f.eks Undernet.



Figur 5.1: Eggman - Offisiell maskot for Eggdrop fra [EGG]

Det finnes også mulighet for å lenke mange Eggdrop bot’er sammen slik at de utgjør et botnett. Via botnettet kan de gi hverandre operatørstatus, kontrollere at kanaler ikke dynges ned med ”spam”(floodet), dele brukerlister og lignende.

I en veldig tidlig fase ble det faktisk vurdert å bruke Eggdrop direkte for å hente ut kanallogger. Dette ble senere forkastet da jeg var avhengig av et system med både klient- og robotfunksjonalitet. I utgangspunktet

begrenses en type bot ala Eggdrop fordi den ikke er ”intelligent” nok. Jeg ønsket et system med flytende interaksjon, man trenger i så måte tilleggsfunksjonalitet slik at Eggdrop nærmest ville emulert klientadferd, et semi-automatisk konsoll kunne vært en løsning. Evt kunne man forsøkt å delegere AI-funksjonalitet inn i roboten.

Den senere tiden har man sett en trend hvor automatiske roboter utestenges fra irc-nettverket. Slik sett er det vanskeligere å bruke disse til å fange opp kanallogger samtidig som man skjuler robot-identiteten. På kanaler hvor man ikke tilhører kjernen av brukergruppen vil man umiddelbart bli utestengt hvis slik aktivitet oppdages.

Det finnes dog flere interessante elementer som har blitt hentet fra Eggdrop inn i denne oppgaven. Hovedsaklig dreier dette seg om funksjonalitet knyttet til ”indeksering” av chat. Implementasjonen i denne oppgaven bruker de samme ideene til å hente ut dialog fra chat-systemet, men hvor Eggdrop ikke har noe søkefunksjonalitet integrert gjør jeg gjennom denne oppgaven chat-dialogene tilgjengelige for søk- og filtrering.

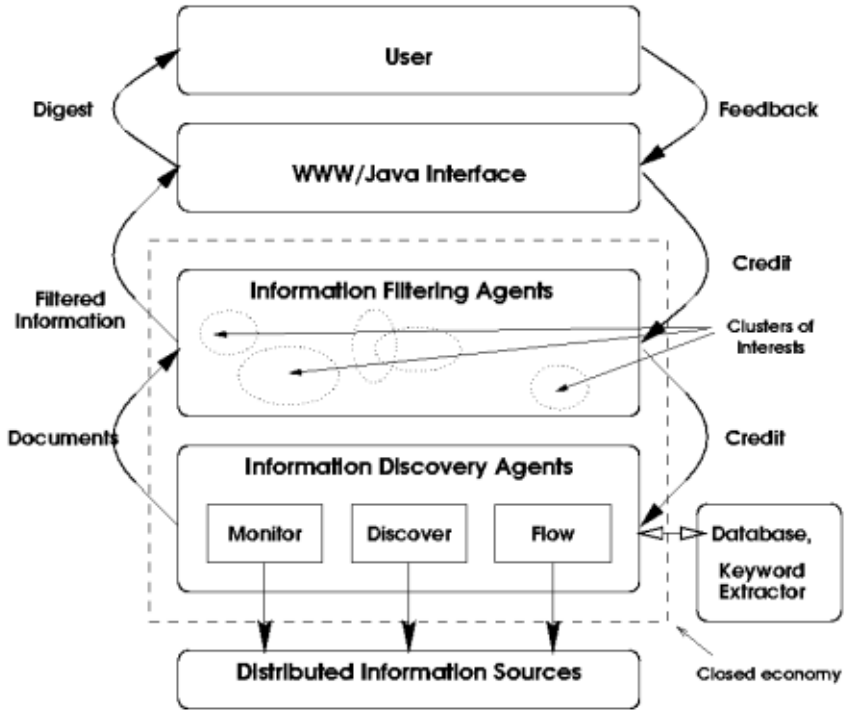
5.2 Amalthea

Amalthea er et system utviklet med MIT av Alexandros Moukas. Systemet søker etter interessant tekst gjennom søkemotorer og ved å lytte på nyhetsstrømmer som NNTP. Søkene realiseres gjennom to typer agenter, en for innsamling av data og en for utvelgelse av tekster. Sett fra brukerens perspektiv presenteres resultatene i nettleseren(java/HTML) hvor det også er mulig å gi tilbakemelding på hvor relevante resultatene var.

Gjenfinnings-agentene deles inn i tre klasser: Overvåkning(Monitor) som utføres mot aktuelle nettsider, oppdagelse(discovery) som finner frem til nye dokumenter via søkemotorer og ”flow” som lytter på strømmer av meldinger. Disse klassene gjenspeiles gjennom figuren 5.2.

Når man har innhentet dokumenter prosesseres disse av de forskjellige agentene hvor de indekseres, dvs først ekstraherer man nøkkelord av tekstene så forkorter man ord til ordstammer eller rotord gjennom stemming(se teori-kapitlet). Til slutt vektes hvert ord i henhold til forekomster i det originale dokumentet og evt metadata påføres.

Tilstanden til hver av filteragentene sammenlignes med en vektor og et tall som er betegnende for hvor ”dyktig” agenten har vært og derfor antas å være. For å avdekke hvor relevant et dokument er beregner den dette ved et sett av trigonometriske funksjoner. Når en bruker gir tilbakemelding på hvor relevante dokumentene er klassifiserer man samtidig agenten som ”flinkere” eller ”dårligere”, men dette er også avhengig av



Figur 5.2: Amalthea - Overordnet struktur fra [Mou96]

hvilken tiltro agenten i utgangspunktet hadde til sin egen anbefaling.

Filteragentene poster forespørsler mot helle populasjonen av gjenfinningsagenter og forsøker å hente ut relevante deler av de ressurser som er tilgjengelige. Gjennfinningsagentene blir løpende "evaluert" av filteragentene som produserer anbefalingene slik at resultater som fører til negative tilbakemeldinger også får konsekvens for gjenfinningsagentene. Et resultat av dette er at gjenfinningsagentene lagrer en intern liste over hvilke filteragenter den får flest positive tilbakemeldinger fra og prioriterer senere forespørsler fra disse.

Filteragentene er i Amalthea implementert som *genetiske algoritmer* der den representative vektoren er et "genom". Nye agenter produseres derfor ved å la de dyktigste agentene "parre" seg med hverandre. Dvs, de blander sammen sine respektive vektorer på bestemte måter. Agenter som ikke tilhører gruppen med god *fitness* får ikke love til å parre seg og dør derfor ut. Hvor mye variasjon man velger å påføre agentpopulasjonen styres ut fra et sett parametre som er betegnende for "total godhet". Dersom resultatene som leveres er dårlige vil det føre til både mye død og parring og i motsatt tilfelle vil de parre seg lite. Det utføres også ytterligere tilpasninger som sørger for at agenter som sjelden

eller aldri leverer gode resultater til slutt klassifiseres som udyktige.

Moukas utviklet også en teknikk for å måle hvor godt systemet faktisk fungerte. Ved å se på den totale godhet(fitness) ble det produsert diagrammer som viste at fitness hadde en jevn stigning mot grenser for konstante interesser hos brukeren. Dersom brukerens interesser endret seg sank også godheten ved endringspunktet, men etterhvert steg den igjen mot tidligere nivå.

Amalthea er ikke direkte tilknyttet til irc-nettverket, men har egenskaper som tilsier at det allikevel vil være mulig å indeksere og filtrere dette systemet. Det underbygges også ved at systemet allerede har støtte for å lytte på nyhetsstrømmer som NNTP. De elementer som har vært aktuelle å trekke inn i denne oppgaven er den overordnede strukturen i systemet. I implementasjonen brukes det en kombinasjon av nettleseren og et javagrensesnitt mot søk- og filtrerbar informasjon lik fremgangsmåten i Amalthea. Bytter man ut kjernen i Amalthea-systemet(filter og oppdagelsesagentene) med Fast Data Search tilsvarer det nærmest identiske systemer. Det som dog har blitt forkastet i forhold til denne oppgaven er tilbakemeldinger fra brukerne. Det finnes i så måte ingen mulighet til å betegne gjenfunnet informasjon som ”nyttig” eller ”unyttig”, dette ville i tillegg innebære å trekke inn elementer fra *recommender*-teori som går utover denne oppgavens omfang.

5.3 Butterfly

Butterfly[DLM99] er en agent som assisterer brukere på IRC-nettverket til å finne frem til kanaler som kan være av interesse. Søkene gjøres i bakgrunnen og baseres på kriterier hentet ut fra brukerens atferd. Konkrete forslag til aktuelle kanaler presenteres fortløpende av agenten.

Butterfly er resultatet av noen interessante funn gjort av tre forskere med MIT. Spesielt ble det bemerket at IRC som chat-system mangler et strukturert hierarki og tilhørende organiseringsmekanismer. Dette underbygges av det faktum at chat-kanaler kun identifiseres ved kort navn som gir lite eller ingen indikasjon på hva kanalen faktisk representerer. Det ble også slått fast at temastrengene for kanalene(topic) ikke gjenspeilet de aktuelle diskusjonene og derfor ikke var spesielt veiledende for evt nye brukere. Konklusjonen ble at det var både vanskelig og tidkrevende å søke seg frem til nye kanaler uten noen form for assistanse.

Butterfly-agenten tok sikte på å løse disse problemene ved å sample innholdet(chat-dialogene) på tusenvis av kanaler simultant og på bakgrunn

av dette komme med anbefalinger til brukeren basert på en modell som knytter nøkkelord og interesser sammen. To sammenhengende eksempler som illustrerer Butterfly i aksjon er vist under.

Bruker: "Jeg er interessert i objektorientert programmering og UML"

Butterfly: "OK. Du kan ha interesse av kanalene #java.no og #uml"

Butterfly genererer også en interessemodell på bakgrunn av de opplysninger den innhenter av brukerne. Denne modellen baseres på en enkel termvektor med positiv og negativ vekting.

Bruker: "Jeg er interessert i fotball"

Bruker: "Jeg er veldig interessert i linux, men ikke Redhat distribusjonen"

Bruker: "Hvem er jeg?"

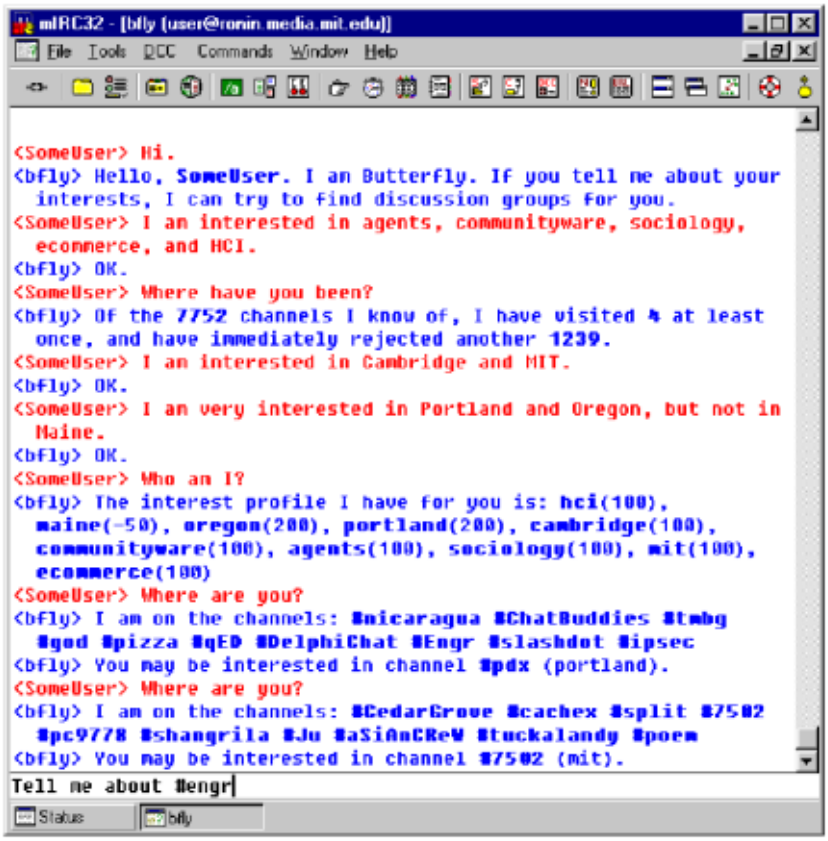
Butterfly: "Din interesseprofil er: linux(200), redhat(-50), fotball(100)"

Innholdet på kanaler blir også representert ved termvektorer. Vektingen refererer til frekvens av forekomster. Hvor interessant en kanal er avhenger av summen av vektene(produktet av vektorene). Hvis denne summen overskrider et gitt threshold anbefales kanalen.

Etiske aspekter ligger til grunne for at Butterfly ikke konstant overvåker all trafikk på en gitt tjener(folk forventer en viss grad av privatliv). Isteden benyttes en løsning hvor agenten kobler seg til en tjener og opptrer som en vanlig bruker på de kanalene den sampler. Samplingen foregår over en periode på 30 sekunder og opptil 10 kanaler samples simultant. Kanaler som aldri har blitt besøkt prioriteres fremfor de som allerede har blitt besøkt.

Butterfly-agenten har gjennom skalerbarhet tatt høyde for en av de vanligste krav/ønsker som stilles til slike typer internet roboter. Istedenfor at tusenvis av brukere har egne agenter som overbelaster IRC-nettverket, jobber en enkelt agent på vegne av mange brukere. Den skiller også mellom brukernes individuelle profiler slik at brukerne blir koblet mot "sine" data.

Det finnes dog begrensninger som gjør at Butterfly ikke vil kunne arbeide optimalt. Den senere tiden har en ny trend hvor kanaler "skjules" oppstått. Kanalstyrerne eller eierne av kanalen endrer attributt ved å sette +s-flagget på kanalen og den er da ikke lenger søkbar. Mange av kanalene som finnes på IRC nettverket er skjulte/hemmelige og Butterfly-agenten



Figur 5.3: Butterfly i aksjon fra [DLM99, side 40]

vil ikke oppdage disse. I mange tilfeller er det også disse kanalene som er blant de bedre og mer interessante(det finnes en kjerne av brukere, tema for kanalen følges konsekvent). En kan spørre seg hvorfor det skulle være nødvendig å skjule kanaler, grunnene kan være mange, men det har vist seg at det oftest skyldes brukernes ønske om mindre tilfeldig og forstyrrende trafikk på kanalen. På den annen side fører det til at behovet for nye brukere og friske innspill før eller siden vil melde seg.

På grunnlag av ovennevnte problematikk oppstod tanken om gruppeagenter(agenter som opererer på vegne av en gruppe/kanal), hvor målsetningen er å tiltrekke seg nye brukere med samme interesser. En løsning var en multiagentarkitektur hvor hver gruppe(kanal) og hvert individ har sin egen agent. De to typene agenter må selvfølgelig inneha egenskaper som gjør at de kan skilles fra hverandre. Gruppeagenter oppdateres på den aktuelle gruppens egenskaper/tema samtidig som den søker etter nye brukere som samsvarer med de ønsker gruppen måtte ha. De indi-

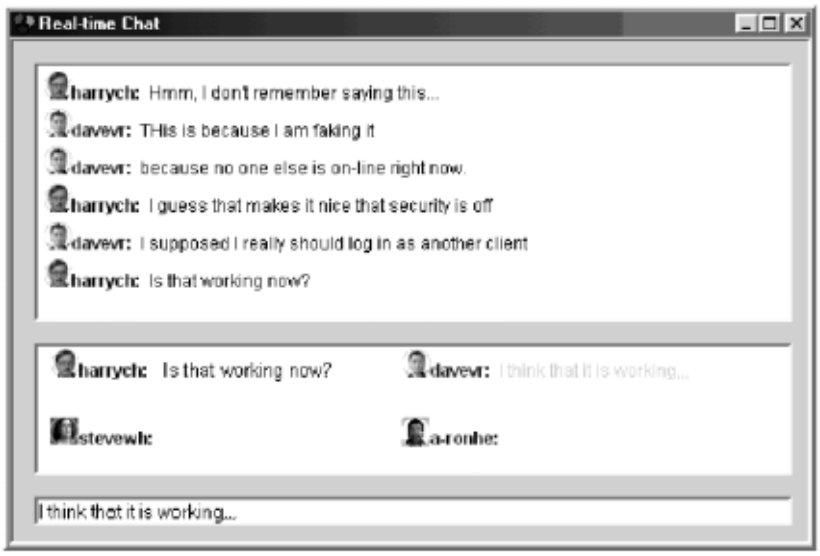
viduelt tilpassede agentene derimot fokuserer kun på de særegne interesser/ønsker hver bruker spesifiserer, og hva det søkes etter i en kanal. Gruppe inkluderer meninger/målsetninger, aktuelle temaer den vurderer, forskjellige formater (åpen diskusjon, spørsmål og svar osv.), type av interaksjon (for eksempel grad av høflighet), sosiale regler og gruppens dynamikk generelt.

Informasjonsflyten mellom grupper og gruppeagenter er dynamiske i den forstand at gruppene selv spesifiserer interesseområder, mens annen informasjon ekstraherer agenten selv fra gruppen/kanalen. La oss si at gruppen spesifiserer et tema og informerer agenten om at de gjerne vil tiltrekke seg andre brukere med samme interesser. Men agenten observerer at det til stadighet kun er ett begrenset område av gitte tema som diskuteres. Agentens oppgave blir da å tiltrekke seg brukere med detaljkunnskaper innenfor dette området.

Butterfly-agenten har fungert som inspirasjonskilde og motivasjonsfaktor for denne oppgaven. Agenten er interessant og innehar gode løsninger i forhold til de utfordringer den står ovenfor. Allikevel finnes det alternative fremgangsmåter som totalt sett kan være bedre. Å sample mange kanaler simultant var nok mer trivielt i 1999 enn det er i dag. Slik oppførsel blir raskt notert og vil på sikt føre til utestengelse fra større deler av nettverket. Slik systemet er implementert i denne oppgaven, i form av en klient mot irc-nettverket, tar den utgangspunkt i kanaler man allerede er en del av. Ut fra disse forsøker den å søke seg frem til nye og skjulte kanaler. Dette gjøres ved konstant filtrering av chat-dialog hvor aktuelle kanalnavn ekstraheres fra teksten og taes del i automatisk. Dette kan virke som en snever løsning, men har vist seg å være langt mer effektivt enn først antatt. I tillegg indekseres kanaler som sees på som interessante gjennom Fast Data Search. Det finnes også et eget grensesnitt for søk- og filtrering mot indeksert dialog som gjør opplevelsen mer behagelig for brukerne.

5.4 Alternative Interfaces for Chat

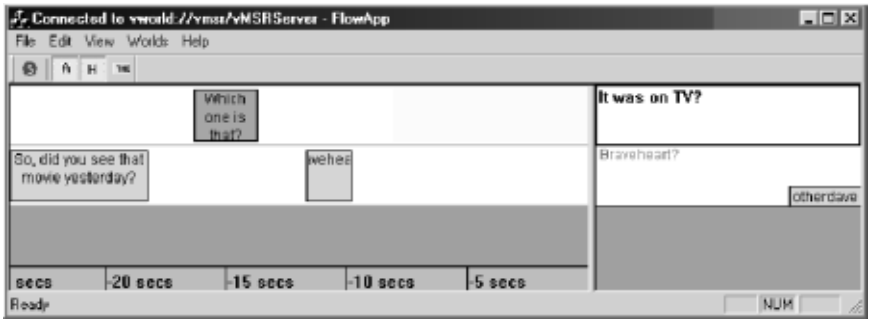
Alternative Interfaces for Chat er resultatet av forskjellige ideer myntet på grensesnitt mot IRC-nettverket. Man velger å betrakte chat som sanntids-mediastrømmer og prøver ut forskjellige løsninger forbundet med de problemer tradisjonelle tekstbaserte chat-klienter har stått ovenfor. Spesielt er det noen ledende faktorer forfatterne mener har hatt stor innflytelse på effektivitet forbundet med chat. Felles for disse er at de adresserer mangler i nåværende systemer og ved å introdusere en teknologi kalt *Virtual Worlds*, utviklet av en egen forskergruppe ved



Figur 5.4: Status-klienten fra [VSD99, side 21]

Microsoft, ønsker man å foreslå alternative grensesnitt med løsninger på tidligere problemer. Virtual worlds-plattformen er bygget på en COM-basert teknologi som lar programmerere enkelt utvikle nye applikasjoner. Teknologien underbygger et persistent, dynamisk og distribuert objektsystem hvor objekter kan legges til eller modifiseres ad hoc. Systemet skjermer også de underliggende forbindelse(klient-tjener) for brukerne. Applikasjoner kan forøvrig utvikles ved hvilket som helt COM-kompatibelt programmeringsspråk inkludert DHTML, Visual Basic og C++.

I startfasen av prosjektet kartla man hvilke grensensitt som var tilgjengelige mot chat slik at man hadde noe å sammenligne med. Det ble først designet en prototype på et system kalt ”status-klient”. Man samlet noen utvalgte testpersoner i en lab og lot disse delta i et eksperiment man fra tidligere hadde gode erfaringer med, dette ble referert til som ”konferanse”. Status-klienten hadde implementert funksjoner for å gjøre brukerne mer ”oppmerksomme” på hverandre. Blant annet kunne man via et eget vindu se når andre personer tastet inn tekst før denne ble publisert i det offisiell kanalvinduet. Siste linje brukere hadde sendt til kanalen ble også knyttet opp mot navnet i brukerlisten. Dommen fra brukerne var positiv, kun en av deltagerne foretrakk tradisjonelle grensesnitt. Spesielt var funksjonen som gjorde at man kunne ”se” de andre skrive appellerende. Noe som dog ble oppfattet som negativt var at rekkefølgen på brukernes inntasting kunne variere fra ”live”-vinduet til det offisielle



Figur 5.5: Flow-klienten fra [VSD99, side 24]

kanalvinduet, alt avhengig av hvem som trykket på enter-tasten først. Dette samt andre observasjoner og tilbakemeldinger fra brukerne første til et redesign av klienten. Det var også under denne fasen man anså chat for å være tilnærmet likt mediastrømmer og vurderte derfor grensesnitt som gjenspeilet denne oppdagelsen. I en mediakontekst(brukergrensesnitt) "flyter" tiden fra høyre mot venstre og skjermen brytes ned i horisontale kanaler som representerer de forskjellige kildene. Resultatet av dette ble "flow-klienten".

Først etter to testversjoner ble man enige om en mulig prototype av "flow-klienten". Strukturen på den endelige versjonen finner vi i figuren 5.5. Gjennom testfasten gav brukerne uttrykk for at de syntes testmiljøet var noe kunstig og ikke nok representativt for hvordan en reell chat foregår. Noe av dette skyldes at de ble satt til å diskutere rundt bestemte tema. Ut fra disse observasjonene besluttet forskerne å simulere et miljø hvor man tilnærmet seg aktive og reelle chat-kanaler. Til tross for delvis positive resultater ble konklusjonen at brukerne ikke følte seg komfortable med hvordan hovedvinduet(chat-historien) ble presentert. Man falt ikke for ideen å bruke horisontal refleksjon av dialogene og flere syntes også det ble vanskelig å følge med fordi teksten forlot vinduet tidligere enn med vanlige klienter. Til tross for problemer rundt designet av testklientene var det allikevel unison enighet om at det fortsatt vil være interessant å betrakte chat fra et mediastream perspektiv. Målet er da på sikt å dedikere en egen lab til dette formålet hvor man over en lengre tidsperiode skaper et "riktig" miljø for testingen. Aktuelle prosjekter ville være å se på mulige brukergrensesnitt som tar høyde for brukernes tilstand(skriver, leser, borte osv.) og metoder for å gjennomføre dette. I tillegg var det et ønske om at klientene skulle bli mer "informative" i fremstillingen av dialogene ved å innføre funksjoner som viser hvem som prater og hvor ofte.

Det som har vært interessant å hente inn i denne oppgaven er hvordan

man betrakter chat og hva slags grensesnitt brukerne føler seg komfortable med. I implementasjonen er det to typer grensesnitt som eksisterer, et mot irc-nettverket i form av en klient og ett mot indeksert data. Førstnevnte er noenlunde standard i utformingen, men har innebygd støtte for ekstra funksjonalitet ved filtrering av logger og en ryddig struktur i fremvisningen av kanaler den oppdager. Ideen med å vurdere chat fra et mediastream perspektiv implementeres ikke direkte, men gjenspeiles i layout hvor nye/skjulte kanaler automatisk presenteres fra høyre mot venstre på egne paneler.

Oppsummering

Gjennom dette kapitlet har det blitt beskrevet et utvalg systemer som er relevante ovenfor problemstillingen i denne oppgaven. Dette inkluderer hvilke ideer og funksjonalitet som har blitt hentet inn eller forkastet.

Kapittel 6

FAST Data Search

Fast Search and Transfer ASA [FAS] er et firma som utvikler kommersiell programvare for søk- og filtreringstjenester. De er sannsynligvis best kjent for søkemotoren *alltheweb* som har eksistert noen år. Deres kjerneprodukt er allikevel Fast Data Search(FDS), et avansert og modulært IR-system med integrert støtte for søk og filtrering i sanntid. Systemet er også designet med tanke på skalerbarhet og kan derfor behandle enorme mengder med data.

FDS-systemet har blitt brukt som plattform gjennom implementasjonen i denne oppgaven. Dette dels fordi man da i langt større grad har mulighet til å håndtere store datamengder uten å bekymre seg over om systemet vil kollapse og dels fordi man slipper å skrive et system fra bunn med alt det arbeid det ville medføre.

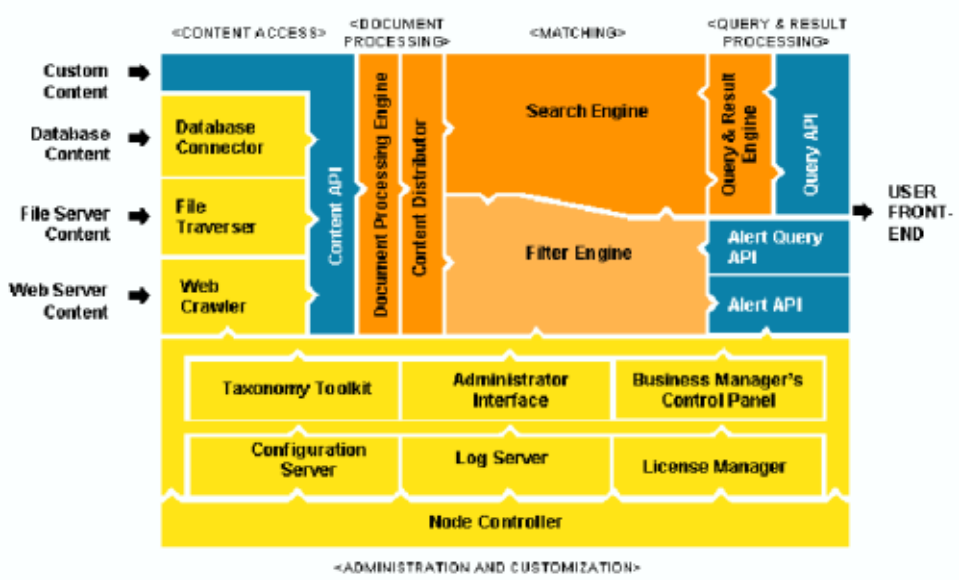
Systemet presenteres ut fra figuren 6.1 og beskrives nærmere gjennom dette kapitlet. Beskrivelsen deles inn i to hoveddeler styrt etter de ledende komponentene i systemet, *søkemotoren*(search engine) og *sanntidsfilteret*(filter engine). Hovedsaklig sentreres systembeskrivelsen rundt funksjonalitet knyttet til hovedkomponentene og hvordan disse underbygger de grafiske verktøyene som brukes mot systemet. Et eksempel på et slikt verktøy er Fast Query Tooltkit(FQT). Dette vertøyet leveres som en del av Fast Data Search og har blitt brukt mot indeksert data gjennom denne oppgaven. FQT-verktøyet beskrives nærmere i seksjon 6.5

6.1 Overordnet beskrivelse

Her følger en beskrivelse av funksjonaliteten som omkranser søkemotoren og sanntidsfilteret. Denne er uavhengig av hvilken komponent som prioriteres. Slik vi ser av figuren 6.1 kan data som er på vei inn til systemet ha opphav i flere forskjellige kilder, dette være seg fra en database, filtjener, *crawler* eller direkte via et *Content API* noe all data allikevel må gjennom. Dette API'et er i realiteten et sett av programmer som prosesserer innholdet fra et mangfold av formater(f.eks pdf, word, html). Totalt er hele 255 forskjellige formater offisielt støttet. Disse konverteres og lagres på et internt format med XML-struktur. Dokumentene går så gjennom flere former for dokumentprosessering hvor nødvendig informasjon ekstraheres og evt påføres. Forøvrig formidles og returneres spørringer via et av de tilgjengelige API'ene avhengig av hvilken komponent man ønsker å bruke.

6.2 Felles funksjonalitet

Dokumenter som ankommer FDS-systemet må gjennom en nødvendig livssyklus av normalisering, dokumentprosessering og indeksering. Figur 6.2 er en representasjon av denne prosessen.



Figur 6.1: Fast Data Search - Overordnet arkitektur fra [SYS, side 4]

I henhold til figuren representerer kilden, eller content som er fellesbetegnelsen, data som er på vei inn til systemet. Til sammenligning benyttes 'dokument' om data som allerede eksisterer innenfor rammene av systemet.

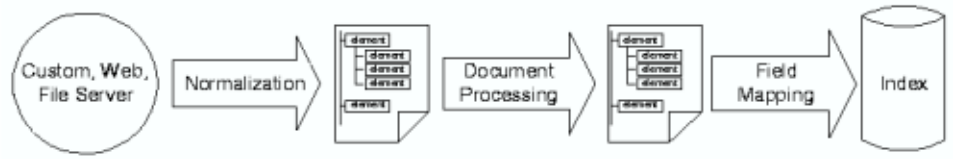
Dokumenter grupperes forøvrig i selvstendige kolleksjoner. Ideen baseres på at man grupperer dokumenter på grunnlag av kilden og prosesseringsmekanismen. Eksempelvis kan en kolleksjon kun bestå av dokumentkilder i form av domenenavn. Kolleksjoner kan også benyttes hvis man ønsker å snevre inn omfanget av søk. Man spesifiserer hvilke kolleksjoner som omfavnes av søket.

En av fordelene ved å benyttes seg av kolleksjoner er at man kan behandle forskjellige typer dokumenter helt uavhengig av hverandre. Vanligvis grupperes dokumentene basert på følgende karakteristikker:

- Kilde - Flere kilder støttes, databaser, filtjenere, web for å nevne noen
- Dokumentformat - Et mangfold av formater er støttet(pdf, word, html osv)
- Eierskap - Er dokumentet av intern eller ekstern opprinnelse
- Prosesseringsregler - Man forholder seg til et sett av prosedyrer for å håndtere metadata og lignende
- Tilgangsregulering - Er dokumentet adgangsbegrenset, f.eks til en spesiell gruppe eller bruker

Ved en slik oppdeling vil det være overflødig med ytterligere inndeling. FDS-systemet er skalert opp til å kunne håndtere veldig store kolleksjoner uten at dette har noen nevneverdig innflytelse på ytelsen. I tillegg finnes det muligheter for å prioritetsrangere kolleksjonene.

Som nevnt tidligere konverteres dokumenter til en Fast-spesifikk XML representasjon som forøvrig kalles FastXML. Dokumentene bygges opp



Figur 6.2: Livssyklusen fra [SYS, side 12]

ved bruk av dataelementer. Slike elementer inneholder vesentlig informasjon ekstrahert fra det opprinnelige dokumentet f.eks utledet av *tittel* eller *body*-delen av en HTML-side.

Under behandlingen av dokumentene forholder systemet seg til et sett av predefinerte tagger, dvs det er disse som bestemmer hva som skal ekstraheres fra det originale dokumentet. Noen eksempler på slike tagger er:

- Tittel
- Forfatter
- Body
- Unik ID
- Språk

Denne mappingen tar vare på den originale dokumentstrukturen og eventuelle metadata integrert i dokumentene. En positiv effekt av denne mappingen er at det opereres uavhengig av datatype. For eksempel vil det være mulig å representere en rekke i en databasetabell som selvstendige dokumenter. For søk og filtrering kan hvert dokument behandles som en søkbar enhet og dermed listes på denne formen.

Hvert dokument representeres i tillegg ved en unik URI(Universal Resource Identifier). Det eksisterer ingen begrensinger i hvordan man velger å formatere denne, men vanligvis er det slik at man ved crawling benytter seg av URL'en til det crawlede dokumentet. Alternativt kan man selv fôre systemet med dokumenter via egne applikasjoner ved å benytte seg av Content API'et. Da må man selv informere systemet om hvilken URI som skal brukes. For denne oppgaven er mest naturlig å bruke en modul kalt *filetraverser* til å sende inn dokumenter i FDS-systemet. URI'en som benyttes relateres til dokumentnavnet som i realiteten er navnet på chat-kanalen som indekseres.

Formateringen av elementnavnene er ikke spesielt restrektiv, det finnes f.eks ingen reserverte elementnavn. Hvert element er således en selvstendig entitet så lenge dokumentet er innenfor rammene av systemet.

Dokumentelementene mappes så videre til en spesiell type *felter*. Mer spesifikt er felter utvalgte dokumentelementer som gjøres søkbare. 'Utvalgt' er i denne sammenhengen et resultat av de valg brukeren av systemet har foretatt seg. Felter defineres forøvrig ved at brukeren oppretter en egen indeksprofil. Det er også mulighet for å gruppere felter på en

"logisk måte" i egne *composite*-felter som gjør det mulig å søke over flere felter samtidig.

Som beskrevet ovenfor er Content API'et en alternativ migreringsvei hvis ingen av de andre modulene er tilstrekkelige. Ved å mate dokumenter gjennom API'et vil man oppnå en tilsvarende effekt som om man benyttet seg av en offisiell modul. Applikasjoner som ikke per definisjon støttes av FDS-systemet gjøres allikevel tilgjengelige gjennom Contentgrensesnittet. Så lenge dokumentformatet er støttet av systemet frikobles alle forhold med tanke på hva slags applikasjon en velger å bruke.

Rent teknisk konverteres dokumenter til FastXML etter prosessering av Content API'et. XML-dokumenter som allerede tilfredsstiller FastXML kriteriene mappes direkte til dokumentindeksen og eventuelle andre XML-dialeketer konverteres gjennom en innebygget XSLT-transformasjon. Etter formateringen sendes dokumentene videre til *dokumentprosessoren*(se figur 6.1). Denne er satt sammen av flere uavhengige enheter referert til som prosessorer. En prosessor er en modul satt til å utføre spesifikke oppgaver. Dette kan for eksempel innebære å modifisere et dokument ved å fjerne eller legge til informasjon. Moduler kan også bearbeide flere dokumentelementer simultant hvor nye elementer produseres eller eksisterende modifiseres. Prosessorene har også den egenskap at de kan resirkuleres, de arbeider med andre ord mot bestemte oppgaver fullstendig uavhengig av hva som prosesseres. Det finnes også en type instansierte dokumentprosessorer eller dokumentprosesseringssteg som forøvrig er fellesbetegnelsen. Slike steg er strukturerte av art og organiseres i *pipelines*. En pipeline er funksjonelt sett sekvenser av dokumentprosesseringssteg. Den eneste reelle oppgaven disse har er å overvåke prosessorene. Dette inkluderer sekvensiell instansiering over hvilke prosessorer som trer i kraft til hvilket tidspunkt. F.eks er det mulig å gjøre endringer i dokumentprosesseringen uten direkte interaksjon mot dokumentkolleksjoner. Dette gjøres ved at man endrer innstillingene innenfor gitte pipeline konfigurert via det grafiske grensesnittet som tilbys gjennom Fast Data Search.

Et konkret eksempel på en pipeline er vist under. Denne tar utgangspunkt i hvordan et HTML-dokument behandles.

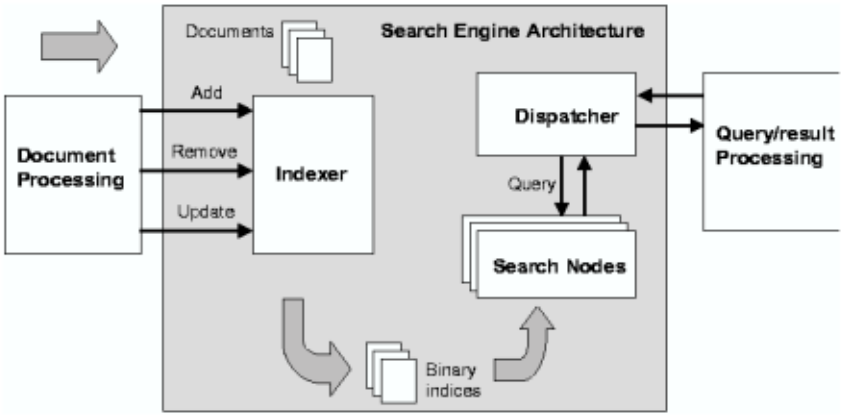
1. *Formatering* - Dokumenter formateres avhengig av MIME-type
2. *Konvertering* - Eksterne dokumenter konverteres til et internt håndterbart format
3. *Parsering* - Dokumenter parseres og strukturer ekstraheres(f.eks body)

4. *Språk- og kodedeteksjon* - Dokumenter identifiseres mhp på språk og koding
5. *Tokenisering* - Tekst splittes opp i ord, en egen variant benyttes for asiatiske språk
6. *Koding* - All koding representeres ved UTF-8
7. *Generering av dokumentrepresentasjoner* - Dokumenter sammenfattes i dokumentsurrogater
8. *Lemmatisering* - Ord brytes først ned til rotformen og ekspanderes så til alle mulige former

I henhold til denne oppgaven er ovennevnte pipeline nærmest identisk den som brukes mot chat-dokumentene. Chat-dokumentene er rene tekstfiler uten noen form for strukturer integrert slik at parseringen av disse er en ren formalitet.

En siste instans dokumenter må gjennom før de fordeles til en av hovedkomponentene i systemet, dvs søkemotoren eller sanntidsfilteret evt begge, er *dokumentfordeleren*. Via denne sendes dokumentene til rett komponent.

6.3 Søkemotoren



Figur 6.3: Overordnet arkitektur over søkemotoren fra [SYS, side 160]

I søkemotoren blir dokumenter gjort tilgjengelige for søk først etter at dokumentene er lagt inn i en intern database som jevnlig oppdateres.

Denne prosessen refereres til som indeksering. Innsetting av søkeuttrykk gjøres vanligvis via et grafisk grensesnitt integrert i nettleseren. Dette grensesnitttet kommuniserer direkte med et eget spørre API som utfører den faktiske innsettingen av søkeuttrykkene.

6.3.1 Funksjonell beskrivelse

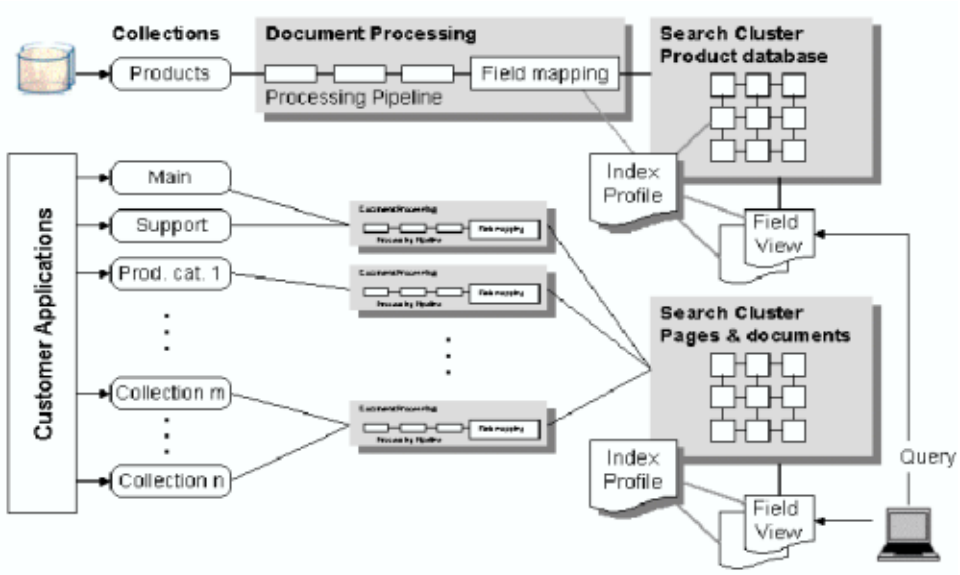
Som vi ser av figuren 6.4 grupperes dokumentkolleksjoner i søkbare *clustre*. Fellesnevneren er at alle kolleksjonene deler samme konfigurasjon og det eksisterer en en-til-en relasjon mellom indeksprofiler og clustere. Indeksprofiler er XML-baserte konfigurasjonsfiler som definerer hvordan dokumenter gjøres søkbare og hvordan felter behandles gjennom spørre- og resultatprosesseringen . Dette betyr at hvert cluster må assosieres med minimum en indeksprofil. Dvs, hvis prosesserte dokumenter omfavnes av kun en indeksprofil vil det være overflødig med mer enn et cluster. Tilsvarende vil man ved bruk av flere indeksprofiler benytte seg av flere clustere.

Indeksprofiler spesifiserer:

- Hvilke dokumentelementer mappes til søkbare felt i indeksen
- Definering av feltatributter som type(tekst, heltall), søkeattributter(lemmatisering mm), parametre for rangering og presentasjon av resultater
- Gruppering av felter og generering av resultat ”views”
- Definering av attributter tilknyttet resultatprosessering som f.eks resultat-clustering
- Hvilke dokumentelementer som mappes til felter og returneres som en del av resultatet

Indeksprofiler, dokumentprosessering og clustere er sterkt relatert til hverandre og figuren i 6.4 illustrerer dette forholdet.

I henhold til figuren 6.1 sendes spørringer inn via et eget API Søkemotorens oppgaver er da fortrinnsvis å undersøke om spørringene har treff i henhold til indekserte dokumenter. De som eventuelt måtte matche sorteres, ved bruk av parametre i spørreuttrykket, og returneres til det aktuelle API’et. Via det grafiske grensesnittet kan brukeren spesifisere hvor mange dokumenter som maksimalt skal returneres.



Figur 6.4: Dokumentprosessering, indeksprofiler og clustere fra [SYS, side 32]

Enhver spørring inneholder som nevnt ovenfor spesielle parametre, henholdsvis en *spørreparameter* og en *filtreringsparameter*. For eksempel ønsker man å finne ordet 'løpe' i en gitt dokumentttekst. Da vil 'løpe' representere spørreparameteren og 'language:no' være filtreringsparameteren.

Et kriterie som i tillegg må tilfredsstilles er at spørringene følger FASTs offisielle spørrespråk. Foruten dette er lingvistisk prosessering kun aktuelt for en enkel og en avansert variant av dette språket. Formatet på filtreringsparameteren er eksempelvis en underklasse av det enkle spørrespråkformatet. Filtreringen har forøvrig ikke noen direkte innvirkning på rangeringen.

Gjennom det grafiske grensesnittet som tilbys av FDS-systemet har man mulighet til å presentere returnerte dokumenter på egendefinert form gjennom "dokumentperspektiv". Et slikt perspektiv er en kortfattet oppsummering av et dokument og blir derav referert til som *teaser*. Det finnes et knippe funksjoner som gjør at disse perspektivene i høyeste grad forbedrer brukeropplevelsen ved økt anvendelighet og funksjonalitet. Under følger en kortfattet beskrivelse av disse:

- *Teasers* - Dynamiske eller statiske teaser-felt som uthever viktige deler av dokumentet definert gjennom spørringen
- *Sortering* - Man kan angi rekkefølgen på relevante dokumenter, sor-

teringen relateres kun til dokumentfelter(f.eks dato)

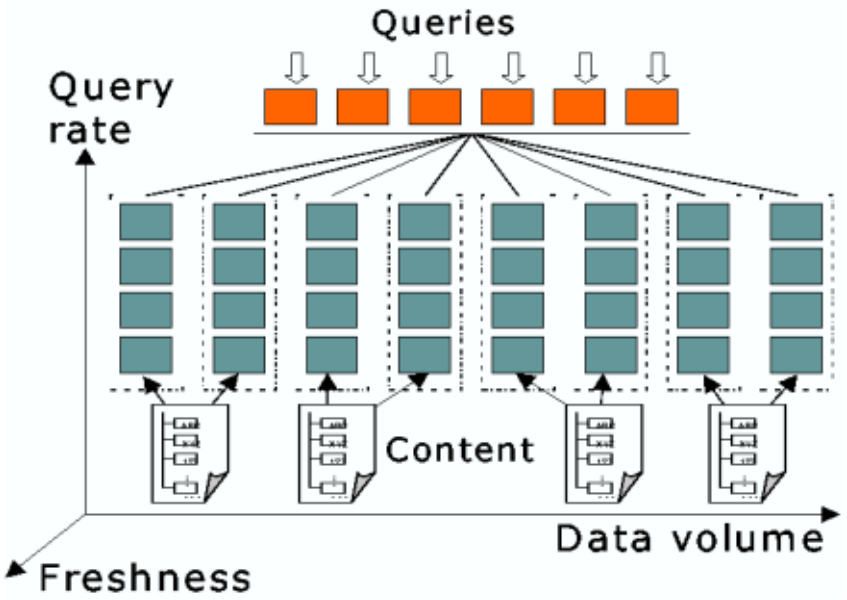
- *Rangering* - Ved å beregne en rangeringsverdi for hvert dokument kan man rangere returnerte dokumenter(høy verdi tilsvarer høy rangering, motsatt for lav verdi)
- *Boosting* - Det finnes flere former for boosting, men felles for disse er at returnerte dokumenter plasseres i resultatlisten ut fra predefinerte bestemmelser
- *Duplikatfjerning* - Dokumenter med høy relevans blir foretrukket fremfor eventuelle duplikater

6.3.2 Skaleringsegenskaper

Som beskrevet tidligere inneholder kolleksjoner søkbare clustere med den egenskap at de kan spres over flere søkbare kolonner eller rekker, primært for å fordele lasten på systemet og distribuere trafikken(spørringene). Ved å lagre indekserte dokumenter i kolonner tar man i stor grad høyde for skalering av volum, men skaleringen er også gjeldende for spørringer. Det betyr at når en forespørsel sendes til en søkenode(se figur 6.3) sendes den samtidig til alle clusterne og dermed alle dokumentrekker. Neste forespørsel sendes til neste tilgjengelige søkenode osv. Ettersom antallet forespørsler økes legges det på flere slike ”spørreprosessorer”. Slik fordeles søkene på en mengde søkenoder som foretar de faktiske søkene i dokumentene. Dersom man antar at søkenodene alltid er i stand til å returnere svar raskt vil økning i antall forespørsler håndteres med en proporsjonalt like stor økning antall spørreprosessorer. Figuren 6.5 forsøker å illustrere dette ved bruk av tre sammenhengende akser, en for datavolum(øker mot høyre), en for spørringsvolum(øker oppover) og en for friskhet(skrått ned til venstre). Friskhet er beskrivende for hvor ferskt et dokument er.

Søk skalerer slik at når datavolum øker segmenteres det på hvilke ord det søkes mot. Hver søkenode håndterer kun en begrenset mengde søkeord og derav en mindre del av den totale datamengden. Dette begrenser datamengden en enkelt søkenode håndterer og garanterer derfor en gitt prosesseringstid.

Ved at antall forespørsler øker kan dessuten segmentene dupliseres og forespørsler fordeles etter en gitt fordelingsnøkkel. På denne måten begrenses den totale mengden forespørsler en enkelt søkenode må besvare pr. tidsenhet. Antall søkenoder skalerer dermed proporsjonalt med produktet av mengde data som som indekseres og antall forespørsler som skal besvares pr. tidsenhet.

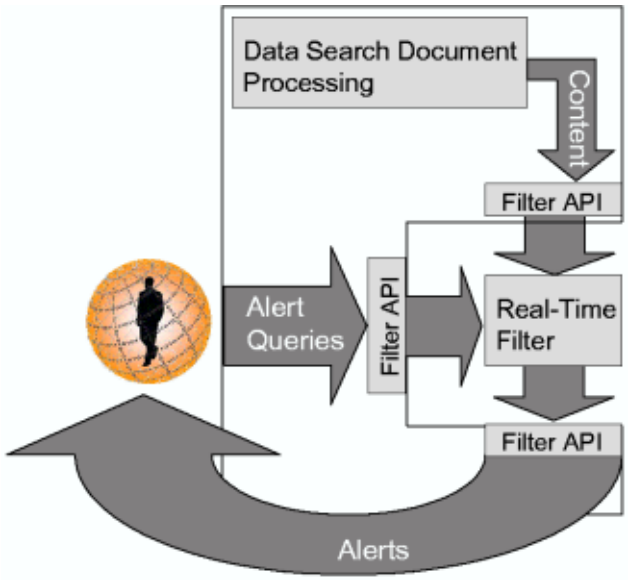


Figur 6.5: Fast Data Search sin arkitektur for skalering fra [SYS, side 106]

6.4 Sanntidsfilteret

I sanntidsfilteret behandles dokumenter ut fra en mengde søkeuttrykk som allerede finnes lagret. Dersom noen av disse får treff i henhold til innkommende dokumenter vil de programkomponentene som har registrert søkeuttrykket få beskjed om hvilke av de innkommende dokumentene som utløste treffet. Innsetting av filteruttrykk til filteret og returnerte dokumenter presenteres vanligvis via et eget grafisk brukergrensesnitt(web-side i nettleseren). Det grafiske grensesnittet skjuler således de underliggende komponentene som utfører de aktuelle operasjonene. I hovedsak dreier dette seg om to forskjellige API hvor det ene håndterer innkommende søkeuttrykk og det andre behandler returnert data.

Gjennom denne oppgaven behandles dynamisk data i form av chat-dialoger. I en slik sammenheng vil det være gunstig å få ut treff i tilnærmet sanntid. Dette gjøres mulig ved bruk av filteret fordi dette kan fange opp fremtidig informasjon i motsetning til allerede indekserte dokumenter.



Figur 6.6: Sanntidsfilteret - Overordnet arkitektur fra [SYS, side 165]

6.4.1 Funksjonell beskrivelse

Innenfor filterterminologien har man en egen type profiler(bestående av et kraftig språk) kalt "varselspørringer" også referert til som *triggere*. En trigger indentifiseres ved en unik id generert av systemet og et spørreuttrykk. Det er denne typen spørringer en bruker indirekte benytter seg av når han bestemmer hvilke ord som skal ha treff i dokumentteksten. Man kan også sende inn triggere sekvensielt eller mate inn flere samtidig da filteret tar høyde for dette. Dokumenttekst i en filtersammenheng er et fellesbegrep på data brutt ned av filteret. All data prosesseres fullstendig før det avgjøres hvilke triggere som hadde treff. Siden den samme dokumentstrukturen brukes for både søk og filtrering kan dokumenter indeksert av søkemotoren også brukes direkte av filteret.

Varselspørringer sendes inn til filteret via et eget Varselspørring API (Alert Query API), dvs man sender inn og mottar meldinger. Vanlig praksis innenfor en filterkontekt har dessuten blitt å omtale alle programmer som bruker dette API'et for en filterklient. Ved treff i henhold til de lagrede søkeuttrykkene vil sanntidsfilteret selv generere resultatet og returnere dette som en "varsling"(alert) via Varslings API'et(Alert API). Det returnerte dokumentet vil vanligvis være på XML-form og bestå av det originale "dokumentet" i tillegg til en liste med trigger id'er som hadde treff. Det finnes også mulighet for å generere flere delresultater hvis

forskjellige triggere slår til på samme dokument.

Varslinger og varselspørringer er dessuten bygd opp av et kraftig og fleksibelt språk beregner spesielt på dette formålet. Under følger en kort skisse av noen relevante eksempler.

En spørring kan ha som mål å omfavne flere forskjellige dokumenter. Et typisk scenario kan hentes fra en børskontekst hvor man er interessert i å motta varsling når aksjekursen endres for et gitt firma/selskap. Faller denne under et bestemt *threshold* trigges en varsling og dokumentet(ene) returneres. Foruten dette finnes det også mulighet for å bruke bolsk logikk direkte i spørringene. En egen *NEAR*-operator benyttes til å filtrere ut ord i en dokumenttekst basert på avstanden mellom ordene. Et NEAR-uttrykk defineres ved NEAR$(m, t_1, t_2, \ldots, t_n)$ hvor m er betegnende for bredden(målt i antall ord) og t_n er et gitt ord. Uttrykket er derfor gyldig hvis alle ordene $t_1 \ldots t_n$ befinner seg innenfor et tekstsegment av lengde n.

Varselspørringer utformes forøvrig ved å bruke AND, OR og NOT-operatorene. Vanligvis har man selvstendige uttrykk som opererer mot innkommende dokumenter, men det finnes også en alternativ fremgangsmåte hvor man ved å bruke en egen *threshold-operator* kan flette sammen flere uttrykk og innkapsle dette i en overliggende varselspørring. I så måte er spørringen et hierarki av deluttrykk med den egenskap at vektede summer for hvert deluttrykk akkumuleres og den totale verdien brukes i gjenfinningen av relevante dokumenter. F.eks vil en spørring trigges når n av m deluttrykk er sanne. Dvs, hvis en spørring inkluderer ordene 'HTML', 'markup', 'web', 'standard', 'XML' vil det være tilstrekkelig at tre av ordene identifiseres i et dokument.

Et eksempel knyttet direkte til denne oppgaven kan illustreres gjennom følgende: FDS-systemet indekserer chat-dialoger og brukerne søker etter relevant data. En gitt bruker søker etter kanaler som inneholder ordet "java", men ingen indekserte kanaler identifiseres ved denne teksten. Systemet tilbyr automatisk å generere en trigger basert på spørringen som inkluderer varsling per e-post og sms. Når kanaler som inneholder ordet "java" oppdages trigges varslingene og brukeren får beskjed i form av kanalnavn.

I en programmeringskontekst aksepterer sanntidsfilteret to typer input, henholdsvis triggere - en fellesbetegnelse på predefinerte spørringer og *events* - tilsvarende for dynamiske datastrømmer("dokumenter"). Triggerne sies å ha treff når 'true' returneres. Det er også vanligvis slik at triggene sender returnerte resultater til en "lytter" i den aktuelle filterklienten slik at resultatet kan vises frem til brukeren.

Programmeringen av filterklienter baseres på et fleksibelt og omfattende

Java API. Ved å kalle metoder direkte fra API'et integrerer man samtidig mye av den funksjonalitet som sanntidsfilteret innehar. Den overordnede arkitekturen av API'et er illustrert i figuren 6.7. Den videre beskrivelsen vil således styres fra denne figuren.

Kjennetegnende for filterapplikasjoner er at disse utvikles i et trådbasert miljø hvor all kjøring av metodekall er asynkrone. Sistnevnte oppnåes ved at triggere og events mates inn til filteret ved å benyttes interne API kall. Internt i sanntidsfilteret rutes både triggere og events i henhold til spesielle meldingsformat. Når resultater returneres kalles lytterobjekter i Filter API'et. Vanligvis er disse en del av filterapplikasjonen som opererer som grensesnitt mot filteret. I tillegg kan det nevnes at filteret har støtte for feiltoleranse ved resirkulering av komponenter.

Funksjonelt sett opptrer API'et som et transportlag mellom sanntidsfilteret og miljøet det omkranses av. En naturlig fremgangsmåte for denne oppgaven er å kategorisere chat-dialogene som events og søkeuttrykkene som triggere. Forutsetningene som må oppfylles for at denne fremgangsmåten skal være gyldig innebærer at:

- Klientene er noder innenfor filternettverket
- Kommunikasjonen mellom nodene er meldingsbasert. Klienter kan sende meldinger som instansierer både triggere og events
- Lytterobjekter sørger for at resultater fanges opp og returneres

Videre av figuren 6.7 ser vi at det finnes et abstrakt meldingslag som i så måte gjør det langt enklere å kode filterapplikasjoner. Under programmeringsfasen er ønsket å strukturerer koden ut fra tre sammenhengende operasjoner:

1. Opprette/terminere klientforbindelser
2. Sette triggere/lytte etter triggere(returnere resultater)
3. Sende inn events(dokumenter) ved å:
   (a) Bruke metoder i Content API'et
   (b) Bruke metoder i Content API'et med sanntidsfilteret som "backend"
   (c) Bruke metoder i Java API'et

Ved å benytte seg av metoder i Content API'et(3a) sikrer man samtidig størst grad av fleksibilitet funksjonelt sett, men dette er også den mest

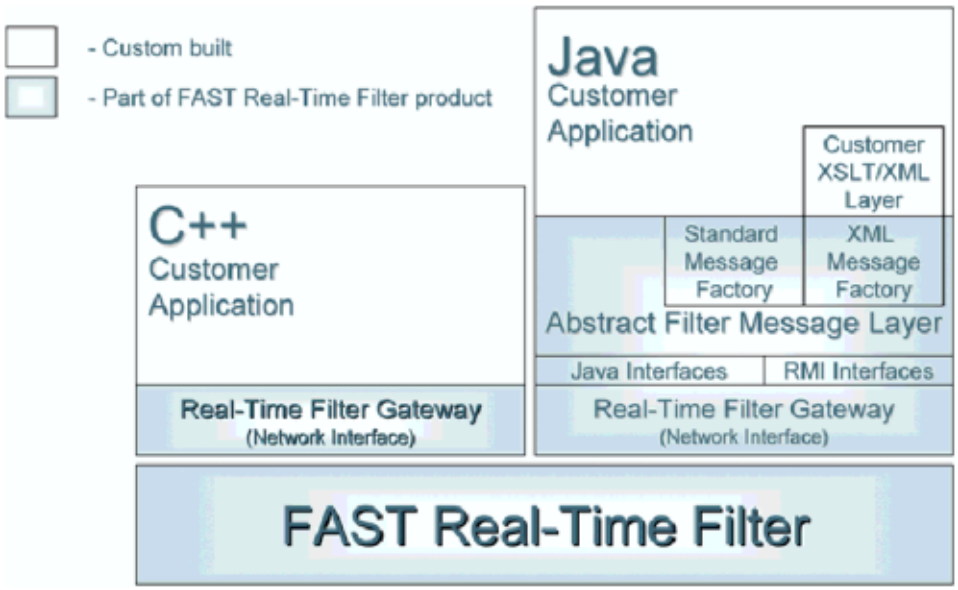
ressurskrevende metoden. Ved å mate inn events på denne måten sender man dokumenter direkte til dokumentfordeleren(Content Distributor) hvor objektene prosesseres. Det er også nødvendig at man konfigurerer den aktuelle pipelinen slik at prosesseringsstegene *EventML-Generator* og *RTF-Output* legges til i.

Man må også ta hensyn til forhold forbundet med ytelse av systemet. Hvis input-raten av dokumenter er veldig høy kan flaskehalser oppstå. I et slikt tilfelle har man mulighet til å gå utenom dokumentprosesseringen og sende dokumenter(eventobjekter) direkte til sanntidsfilteret. Dette kan gjøres:

Indirekte Ved å bruke metoder i Content API'et

Direkte Ved å bruke metoder tilgjengelig i Java API'et

Som det går frem av figuren 6.7 finnes det også støtte for å prosessere XML-dokumenter direkte via Java API'et. Ved å implementere XML-laget arver man samtidig alle underliggende egenskaper og kan i tillegg parsere XML-filer. I henhold til DTD'en(Document Type Definition) er både triggere og eventer gyldige meldinger. Resultat- og bekreftelsesmeldinger konverteres også til XML ved bruk av egne metoder. Forøvrig kan proprietære XML-formater konverteres til standard XML ved et sett av XSL-Transformasjoner(XSLT).



Figur 6.7: Overordnet beskrivelse av Java API'et fra [INT, side 104]

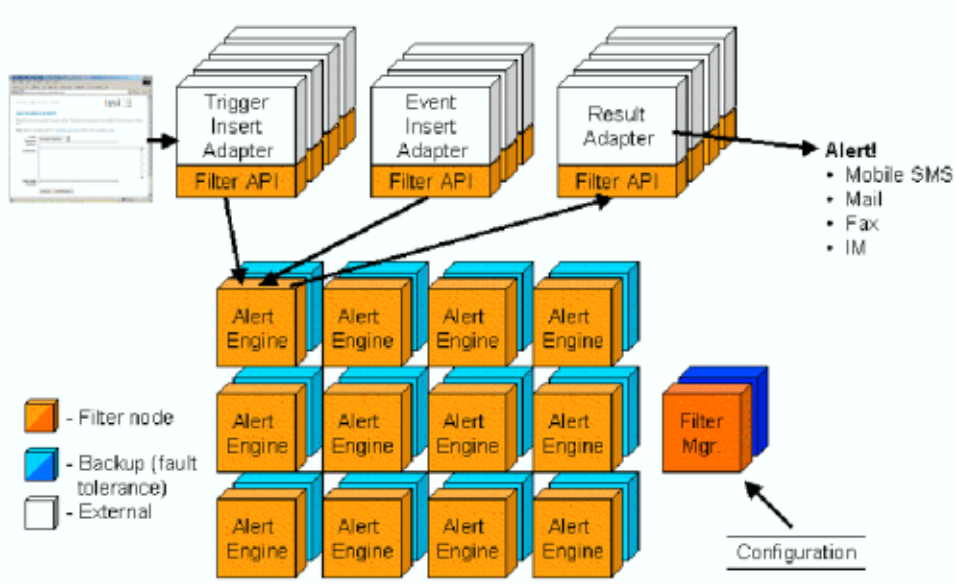
Av figuren 6.7 ser vi at det finnes et alternativ til det Java grensesnittet(Java Interface), RMI(Remote Method Invovation)-laget. Det er fult mulig å ha Fast Data Search installert på et "isolert" nettverkssegment hvor klientene kaller objekter over RMI-laget uavhengig av fysisk lokasjon. Kort følger noen ev de viktigste egenskaper som er kjennetegnende for RMI-laget:

- Fleksibilitet mhp bruk av tråder
- Støtte for *pooling* av klienter
- En klient kan tjene flere applikasjoner
- Meldingsobjekter serialiseres for å oppnå konsistent dataflyt mellom klienter og applikasjoner

6.4.2 Skaleringsegenskaper

Protokollmessig finnes det en underliggende infrastruktur på nettverkslaget skjermet av API'et hvor denne implementerer to lag av kommunikasjon. Infrastrukturen er bygget på toppen av TCP/IP og sørger for kommunikasjon mellom de respektive nodene i sanntidsfilteret. Figuren 6.8 er et eksempel på et slikt nettverk.

Av figuren ser vi at sanntidsfilteret er satt sammen av en mengde filternoder. Hver node kan igjen inneholde en mengde søkeuttrykk som hver igjen er koblet til en ekstern komponent gjennom nettverket(Real Time Filter Gateway i henhold til figur 6.7. Denne eksterne komponenten mottar meldinger om hvilke søkeuttrykk som hadde treff og hvilke data som utløste treffet. Filternodene er allikevel begrenset i hvor mange filteruttrykk de kan håndtere samtidig. Denne grensen bestemmes ut i fra eksterne maskinresssurser som er tilgjengelige(f.eks fysisk minne) eller hvor mye arbeid som ligger til grunne for å filtrere ut filteruttrykkene. Å prosessere et treff krever faktisk at det sendes melding til alle eksterne programkomponenter som har registrert interesse for filteruttrykket. Denne meldingsrutingen kan således føre til at enorme datamengder skal sendes ut og derfor representere en flaskehals. Innenfor rammene av denne oppgaven ser jeg det likevel ikke som trussel, de datamengder det her er snakk om er bare en liten andel i forhold til hva som kreves for å virkelig belaste systemet. Selve prosessen med å identifisere treff ut i fra søkeuttrykkene er ingen flaskehals i seg selv da denne foregår i lineær tid gitt av lengden av det innkommende dokumentet og kun i liten grad er avhengig av antall filteruttrykk som filternoden skal håndtere. En slik garanti kan gis siden typen av uttrykk det er mulig å søke etter begrenses. Det er f.eks ikke mulig å søke etter regulære uttrykk.



Figur 6.8: Eksempel på et filternettverk fra [INT, side 108]

Når antallet filteruttrykk øker håndterer man dette ved å opprette flere filternoder som hver inneholder en delmengde av den totale filteruttryksmengden. Alle innkommende dokumenter rutes så gjennom filternodene. Dersom mengden av innkommende dokumenter blir så stor at filternodene ikke lenger klarer å prosessere disse raskt nok grupperes nodene slik at de sammenlagt dekker alle filteruttrykkene. Hver delmengde av filteruttrykk er da dupliserte i hver filternode i en gruppe. De innkommende dokumentene fordeles så mellom grupper av noder slik at hver node i gruppen ikke blir belastet mer enn nødvendig. Med referanse til Fast Data Search sine skaleringsegenskaper vist i figuren 6.5 kan vi slå fast at antall filternoder skalerer på ovennevnte måte med produktet av antall innkommende dokumenter per tidsenhet og antall filteruttrykk. I tillegg er skaleringen betinget av hvor mye output som genereres av treffene i henhold til filteruttrykkene.

6.5 Fast Query Toolkit

FQT er et verktøy brukt mot FDS-systemet og leveres som en del av det totale produktet. FQT er et eksempel på hvordan man integrerer Spørre API'et mot en HTTP-basert søkeapplikasjon. Formålet med FQT er å utnytte funksjonalitet integrert i FDS-systemet, hovedsaklig tilknyttet clustering, kategorisering og "drill-down"-menyer. FQT er et relativt fleksi-

belt rammeverk som ikke begrenses til spesiell funksjonalitet, dette går også frem av designet siden det er implementert som et JSP-bibliotek.

FQT-verktøyet baseres i stor grad på J2EE-plattformen som rammeverk og dette betyr at er behov for å bruke verktøy som passer inn i slike omgivelser. Et naturlig fremgangsmåte for denne oppgaven var å finne frem til en gunstig applikasjonstjener med passende servlet-container. For bruk med FQT anbefales applikasjonstjeneren JBoss og servlet-containerne Tomcat eller Jetty. Valget falt på sistnevnte dels fordi denne er integrert i JBoss og foretrekkes av utviklerne som har skrevet applikasjonen, men også fordi begynnerterskelen er lavere funksjonelt sett. En introduksjon til J2EE, JBoss og Jetty følger under.

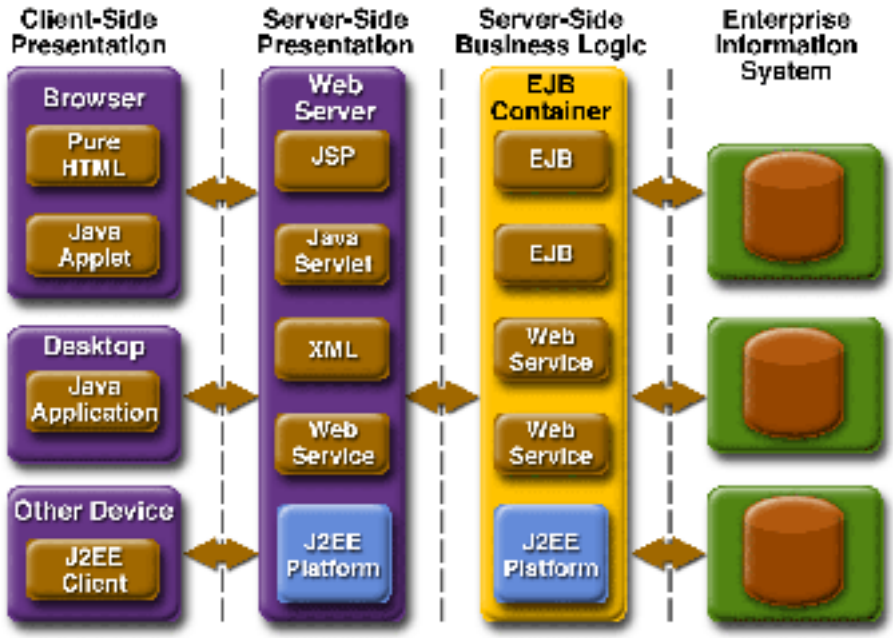
6.5.1 J2EE og JBoss med Jetty

Kort følger en presentasjonen av J2EE hovedsaklig basert på informasjonen tilgjengelig via den offisielle J2EE siden [Sunb].

J2EE eller Java 2 Enterprise Edition er et miljø for utvikling av flerlagsapplikasjoner. Ved bruk av J2EE forenkles utviklingen av enterprise applikasjoner fordi disse baseres på standardiserte og modulære komponenter med et bredt utvalg tjenester tilknyttet disse. J2EE-plattformen har også rutiner for automatisk håndtering av detaljerte oppgaver tilknyttet mulige applikasjoner. J2EE bruker Java 2 plattformen som utgangspunkt for utviklingen, men støtter i tillegg flere nye Enterprise komponenter som *JavaBeans*, *Java Servlets API* og *JavaServer Pages*(JSP).

Figuren 6.9 illustrerer J2EE arkitekturen i henhold til denne beskrevet i Sun BluePrints Design Guidelines for J2EE [Suna]. Arkitekturen underbygger funksjonalitet som transaksjonsstyring, livssyklus-kontroll og ressursfordeling. Dette forenkler utviklingsprosessen slik at utviklere i hovedsak kan fokusere på oppgaver relatert til eksempelvis businesslogikk og utforming av mulige grensesnitt mot applikasjonene. Figuren 6.10 illustrerer J2EE Standard Enterprise tjenestene.

Komponenter, Containere og Connectorer: Med applikasjonsmodellen som utgangspunkt skilles enterprise applikasjoner ut i tre ledende deler: komponenter, containere og connectorer. Ideen bak en slik struktur har forankring i formålet om å skjule kompleksistet og forbedre portabilitet. Dette betyr at utviklere hovedsaklig kan fokusere på komponenter siden containerne fungerer som et transisjonslag mellom komponenter og connectorer. Connectorene er den tekniske broen mellom J2EE-plattformen og andre applikasjoner uttrykt gjennom et portabelt API.



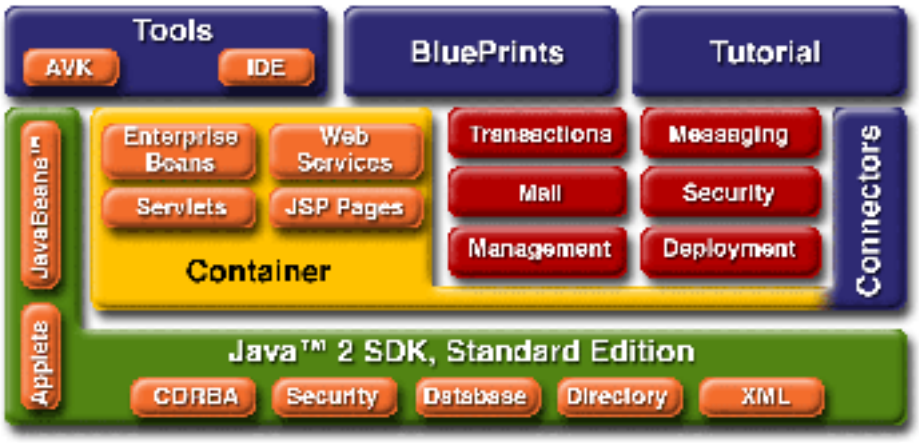
Figur 6.9: Applikasjonsmodell for J2EE fra [Sunb]

Fleksibilitet mhp på brukerinteraksjon: J2EE har utbredt støtte for forskjellige typer brukergrensesnitt. Alt fra Java applets og HTML via selvstendige applikasjoner til Java Servlets API og JavaServer Pages teknologi. Dette betyr at klienter nærmest kan eksekveres fra hvilken som helst digital enhet.

Enterprise JavaBeans: Enterprise JavaBeans(EJB) er modulære og resirkulerbare komponenter som i oppførsel ligner normale Java-klasser. Dog omkranses de av at et rigid miljø hvor kravet er spesifisering av brukergrensesnitt mot containere og klienter. I tillegg må disse som prebetingelse styres fra en dedikert EJB-container som kontrollerer oppførsel og virkemåte.

Det finnes tre typer enterprise beans:

1. **Sesjons beans:** Er enten tilstandsorienterte eller tilstandsløse. Disse brukes primært for å innkapsle logikk, utføre oppgaver på vegne av en klient eller styre/kontrollere andre beans
2. **Entitets beans:** Er persistente objekter med vedvarende levetid, dvs de kan leve videre selv etter at en applikasjon har terminert. Disse lagres typisk i en relasjonsdatabase. Entitets



Figur 6.10: J2EE arkitekturen fra [Sunb]

beans utvikles ved "bean-styrt" persistens eller "container-styrt" persistens

3. **Meldingsbaserte beans:** Lytter asynkront etter JMS(Java Message Service)-meldinger fra gitte klienter eller komponenter og brukes typisk for batch-orientert prosessering.

6.5.1.1 JBoss

JBoss [JBO] er en applikasjonstjener implementert i Java basert på åpen kildekode. JBoss implementerer hele J2EE "stacken" og kan derfor integreres med servlet-containere som Tomcat og Jetty. En effekt av dette er at komponenter kan tilpasses den enkelte applikasjon. Det overordnende målet med JBoss-prosjektet er å tilby en fullverdig implementasjon av J2EE "stacken" i en åpen kildekode kontekst.

ACP-systemet bruker JBoss med Jetty som servlet-container.

6.5.1.2 Jetty

Jetty [Mor] er en av flere servlet-containere tilgjengelig, men har fått fotfeste som ledende av blant annet JBoss-prosjektet. Jetty er "default" servlet container i JBoss og foretrukket i ACP-systemet.

6.5.2 FQT i dybden

Kjernefunksjonalitet som underbygger Fast Query Toolkit er Java servlets, JavaServer Pages(JSP), JSP Standard Tag Library(JSTL) og Java Custom Tags. I tillegg følger FQT-rammeverket et Model-View-Controller(MVC) designmønster som gjør det mer hensiktsmessig å generere web-baserte grensesnitt mot FDS-systemet. FQT baseres på Java 1.3.1 eller senere versjoner og er sesjonsorientert.

Gjennom neste seksjon beskrives prinsipper og teknologi som underbygger FQT-rammeverket og hvordan grensesnittet integreres i ACP-systemet.

6.5.2.1 Servlets

Servlets er Java-teknologiens svar på Common Interface Programming(CGI). Servlets eksekveres fra en web-tjener og fungerer som et ekstra lag mellom HTTP eksempelvis databaser eller andre applikasjoner som kjører på web-tjeneren. Servlets brukes primært i en kontekst hvor kravet er dynamisk fremvising av data.

6.5.2.2 JavaServer Pages

JavaServer Pages(JSP er en teknologi som gjør det mulig å mikse statisk og dynamisk generert HTML. Mange web-sider skrevet med CGI-programmering er primært statiske, dvs kun mindre deler av designet implementerer dynamiske løsninger. De fleste CGI-variasjoner, inkludert servlets, tvinger deg til å generere hele siden via gitte program mens JSP lar brukeren separere disse delene siden de inneholder kall til metoder definert i Servlet API'et.

JSP-teknologien har ført til utvikling av nye verktøy for å trivialisere utviklingen av dynamiske web-sider:

Custom Tags Custom Tags er brukerdefinerte JSP-elementer som innkapsler dynamisk funksjonalitet. De er i tillegg resirkulerbare og distribueres via et eget Custom Tag bibliotek.

JavaServer Pages Standard Tag Library(JSTL) JSTL innkapsler kjernefunksjonalitet felles for mange web-applikasjoner ved enkle tagger. JSTL støtter iterering og avhengigheter, tagger for å manipulere XML-dokumenter og SQL-tagger. JSTL danner i tillegg et rammeverk for å integrere eksisterende custom-tagger med JSTL-tagger.

6.5.2.3 Generelle designprinsipper

Minimere andelen javakode i HTML FQT bruker JSTL-biblioteket som gjør det mulig å globalisere en rikere logikk i JSP-sider uten bruk av javakode. JSTL definerer flere predefinerte tagger som forenkler prosessen med å skrive JSP-sider.

Minimere andelen HTML i javakoden Man forsøker å minimalisere betydningen av javakode i HTML ved FQT spesifikke tagger. Taggene skriver ikke "markup" til siden fordi dette er et vedlikeholdsproblem relatert til endringer i layout. Taggene opererer mot navigasjonspunkter, modifiseringsmekanismer og navigering relatert til gitte spørring, itererer over disse og gjør de synlige for JSTL-taggene som aksesserer siden.

Et alternativ fremgangsmåte er å betrakte hoveddelen av tag'ene som en mal og bytte ut mønster i teksten med gjenfunnede verdier.

6.5.2.4 MVC - Model, View, Controller

Denne seksjonen beskriver ideen bak MVC-arkitekturen og hvordan FQT passer inn i denne strukturen.

MVC-modellen består av tre ledende komponenter - model, view og controller. Den videre beskrivelsen styres ut fra figuren 6.11 som er en avbildning av nevnte struktur.

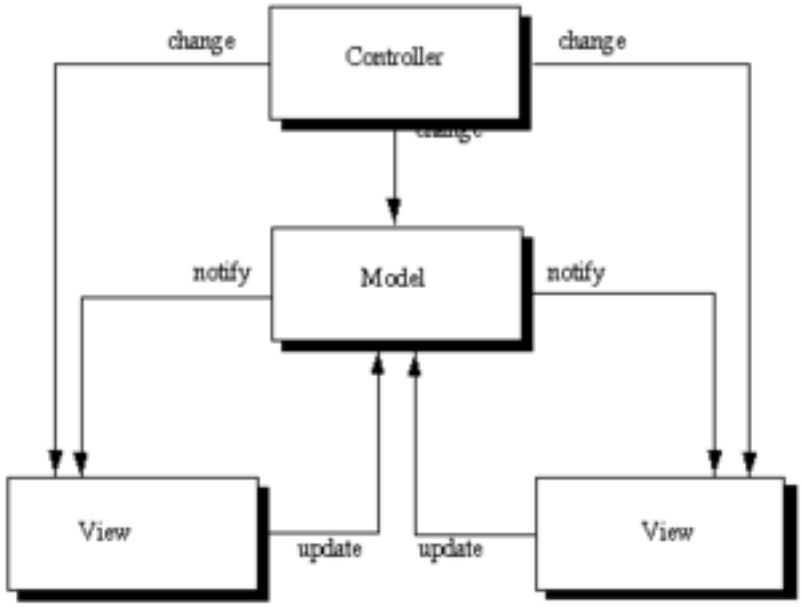
Model Modellen er kjernen i applikasjonen. Eventuelle endringer i tilstand eller tilgjengelig data styres av modellen. Når vesentlige endringer inntreffer oppdaterer modellen alle "views"

En brukersesjon modelleres eksplisitt med klassen SearchSession. Som definert i Servlet API'et assosieres hver HTTP-sesjon med kun en instans av denne klassen.

View View er grensesnittet mot modellen og presenterer informasjon om denne til brukeren. Eventuelle objekter som avhenger av informasjon om modellen må registreres som en del av modellens view.

Innenfor rammene av FQT representeres view av "markup" produsert av tjeneren som sendes til nettleseren for uttegning.

Controller Kontrolleren overvåker brukergrensesnittet og registrerer eventuelle handlinger utført av brukerne. Gjennom kontrolleren tillates i tillegg direkte manipulering av applikasjonen.



Figur 6.11: MVC-arkitekturen

FQT implementerer kontrolleren som en servlet, referert til som SearchServlet. Den oppdateres kontinuerlig om tilstandsendringer i sesjons-objektet basert på input via HTML-"forms" som produseres av relaterte view.

Oppsummering

Gjennom dette kapitlet har teorien fra fagfeltet information retrieval blitt knyttet direkte til Fast Data Search systemet som er basis for implementasjonen i oppgaven. Systemet har blitt beskrevet ut fra de ledende komponentene, søkemotoren og sanntidsfilteret, hvor relevant informasjon har blitt trukket inn. Verktøyet Fast Query Toolkit har blitt introdusert som søkeapplikasjon mot indeksert data og er en del av implementasjonen.

Kapittel 7

Design og Implementasjon

Gjennom dette kapitlet beskrives implementasjon og design av et eksperimentelt system kalt ACP(Automatisk Chat Prosessor). Systemet er utviklet som en del av denne oppgaven og er ikke beregnet for bruk i en produksjonssammenheng.

På nåværende tidspunkt er det nyttig å hente frem problemstillingen beskrevet i introduksjonen.

> *Vi lever i dag i et samfunn som bombarderes med informasjon fra mange forskjellige kanaler. Datasystemer og spesielt internett har den senere tiden vist seg å være en stor aktør på dette informasjonsmarkedet. Et produkt av internettet er det vi kaller chat-systemer. Ved å utveksle informasjon i sanntid har man i langt større grad gjort det mulig å distribuere informasjon hurtig på tvers av landegrenser, kultur og nasjonalitet. En reell utfordring som har oppstått er det å kunne skille ønsket eller relevant informasjon fra støy eller mindre relevant informasjon. Denne oppgaven tar for seg ovennevnte chat-former og systemer hvor spesielt en type er av primærinteresse - IRC. Hvordan skal man gå frem for å indeksere, søke og filtrere ut ønsket informasjon, er informasjonen i utgangspunktet fritt tilgjengelig og er 'tilgjengelighet' et begrep av tvetydig art*

ACP er en prototype på et system utviklet for å undersøke forskjellige former for automatisk prosessering av informasjon tilknyttet chatsystemet irc. Systemet kombinerer elementer fra fagfeltet Information Retrieval med eksisterende teknologi og implementerer alternative brukergrensesnitt mot irc-nettverket og indeksert data.

Ved å se på hvordan tidligere implementasjoner av information retrieval systemer har blitt brukt i en chat-kontekst, som for eksempel Butterfly[DLM99],

er det fremtredende at man forsøker å begrense seg til et system som har sosialt akseptert oppførsel. Hva om man fjerner denne begrensningen og utvider informasjonsfiltreringen til å inkludere elementer sett fra både en informasjonssøker og overvåker sitt perspektiv og i tillegg letter presentasjonen av indeksert data mhp på leselighet og fremstilling.

7.1 Overordnet om systemets arkitektur

Fremstillingen av ACP-systemet er hierarkisk styrt og presenteres ved å starte fra et høynivåperspektiv og gjennom gradvis forfining vise detaljer i designet og implementasjonen. De avgrensninger som gjøres presenteres underveis i teksten.

Overordnet er ACP et system som indekserer, søker, filtrerer og overvåker chat-kanaler. I en mer detaljrik beskrivelse kan vi si at dette foregår ved å knytte sammen programkomponenter fra kategoriene "agentteknologi"(irc klient), information- retrieval/filtrering(Fast Data Search) og "grafisk presentasjon"(FQT) hvor man utnytter de unike egenskapene disse innehar. Følgende komponenter utgjør systemet:

Silent En spesialtilpasset irc-klient skrevet i Java og utviklet for denne oppgavens formål. Den innehar et grafisk miljø og reflekterer kanaldialoger som også eksporteres til FDS-systemet.

Fast Data Search Søkemotoren i systemet lagrer og indekserer data mens sanntidsfilteret filtrerer ut informasjon ad hoc basert på predefinerte søkeuttrykk.

Fast Query Toolkit FDS-systemet integrerer det J2EE-baserte verktøyet Fast Query Toolkit(FQT). Dette brukes til å søke og navigere i tilgjengelig data.

7.2 Silent

7.2.1 Designkriterier

Silent er grensesnittet mot irc-nettverket. Den kan sees på fra to forskjellige perspektiv hvor den inntar rollene som enten informasjonssøker eller overvåker. Under design-fasen var det essensielt at generell funksjonalitet man finner i irc-klienter skulle bygges inn, men samtidig var det ønskelig å legge til nye funksjoner med tanke på indeksering, søk og filtrering. De valg som ble gjort gjenspeiles ved følgende kriterium:

- Klienten skal integreres i et grafisk miljø hvor det er lett å navigere
- Det må være mulig å koble seg til et irc-nettverk og registrere seg med brukernavn og kallenavn(nick)
- Som registrert klient skal det være enkelt å ta del i kanaler og om ønskelig forlate disse
- Kanaldialoger og brukerlister(nicklist) skal reflekteres i egne vinduer. Meldingene som utveksles skal også knyttes opp mot det aktuelle kallenavnet
- Tjenermeldinger som ikke er dialoger, dvs ping eller andre former for "støy"(administrative meldinger, status informasjon) skal sendes til et eget status-vindu
- Det skal finnes støtte for å sende private meldinger med andre brukere på nettverket hvor disse dialogene presenteres i egne vinduer
- Siden kanaldialogene skal behandles for indeksering, søk og filtrering er det nødvendig med en funksjon som skriver dialogene til filer og eksporterer disse til FDS-systemet
- Det skal konstrueres metoder som forsøker å finne frem til ny informasjon/nye kanaler basert på de man allerede er en del av og *crawle* seg frem til eventuelle skjulte kanaler. Ved crawling skal "kanalhierarkiet" traverseres rekursivt og kanaler som oppdages skal automatisk taes del i

Rekursjonsbegrepet som innføres under indekseringspunktene ovenfor er noe løst og må derfor utdypes. Kanaler er kun del av en flat liste og det finnes i realiteten inget hierarki. Derfor må dette taes i betraktning under den videre beskrivelsen.

I tillegg til nevnte kriterier finnes det flere begrensninger forbundet med irc-protokollen man må ta høyde for under en designfase. Spesielt er dette gjeldende for de to siste punktene i listen.

Taher H. Haveliwala [Hav02] har identifisert et knippe utfordringer tilknyttet indeksering av chat:

1. **Dynamiske kanaler:** Kanaler er dynamiske og oppstår/dør avhengig av brukerne. Generelt sett finnes det ikke eierskap forbundet med kanaler allikevel ser man konturene av en ny trend hvor kanaler registreres og en slags form for "eierskap" tildeles(ref. Undernet)

2. **Flat kanalorganisering:** Kanaler er ikke en del av et strukturert hierarki og søkefunksjonalitet er veldig begrenset

3. **Uformelt preg:** Sanntidsaspektet har gjort IRC til en relativt uformell tjeneste, det kan derfor være vanskelig å skille løssluppen prat fra meningsfulle samtaler

4. **Samtidige diskusjoner:** Det kan være vanskelig å skille samtaler fra hverandre ettersom det ikke finnes regler for hvem som prater og til hvilket tidspunkt

Dynamiske kanaler

IRC-kanaler kan betraktes som dynamiske i den forstand at de opprettes og dør ut ad-hoc avhengig av brukerskaren, unntatt når kanalen er registrert. I en indekseringskontekst vil det derfor være vesentlig at det finnes en kombinasjon av statisk/dynamisk indeksering. Grunnen til dette er at man grovt sett finner to typer kanaler, de som er etablerte og de som ikke er det. Etablerte kanaler dør ikke ut og kan derfor antas å være tilgjengelige hele tiden mens mindre "tilfeldige" kanaler på sikt kan dø ut evt flyttes til andre nettverk. Vi kan derfor slå fast at det vil være hensiktsmessig å ha en konstant indeksfunksjon mot etablerte kanaler mens for temporære kanaler bør man enten med automatisert scriptefunksjonalitet eller ved manuell sjekk i klienten fra tid til annen undersøke om en kanal faktisk er aktiv.

Flat kanalorganisering

IRC-protokollen er ikke skrevet med tanke på søkefunksjonalitet i "kanalhierarkiet". De søkeegenskaper som finnes er mest relatert til brukerinformasjon, hvilke kanaler en klient befinner seg på og hvor aktive/ikke aktive de er. I en indekseringssammenheng betyr dette at det nærmeste man kommer en struktur er kanalnavn og tilhørende dialog. En aktuell fremgangsmåte for å "kategorisere" chat blir derfor å skrive ut disse til filer hvor filnavnet er representativt for kanalen. Evt søk i dialogene kan derfor tidligst gjennomføres etter at man har en mengde tekst å forholde seg til. Dette fordrer at filene er dynamiske og at disse kontinuerlig oppdateres med relevant tekst. Slik vil "forsinkelse" i forhold til sanntidsaspektet ikke være spesielt fremtredende.

Uformelt preg

Det er ikke noe tvil om at det kan være vanskelig å skille løssluppen prat fra meningsfulle samtaler. Innenfor indeksering er det nok mer naturlig å bruke begrep som ”interessante samtaler” og ”støy”. Man er avhengig av å finne metoder som skiller disse formene for chat fra hverandre. Et utgangspunkt må derfor være å gjøre noe med teksten man har tilgjengelig, dvs tekstfilene. Innenfor IR har vi som beskrevet i Teori-kapitlet to sammenhengende former for gjenfinning som er aktuelle, ad-hoc og filtrering. Hvilken metode man bruker avhenger av hva man ønsker, det er heller ingenting i veien for å bruke begge som for denne oppgaven.

Samtidige diskusjoner

Det finnes ikke noen funksjoner i IRC-protokollen som gjør at man kan koble samtaler til hverandre. Dialogen er flytende og mange prater samtidig uten noen restriksjoner for når og om hva. Det man dog kan gjøre i en indekseringskontekst er å filtrere ut sammenhengende ord eller setninger i dialogene og skille disse ut i egne filer som organiseres. Evt kan man indeksere en mengde kanaler hvor dialogene representerer dokumenter som det er mulig å gjøre spørringer mot, slik det gjøres innenfor rammene av denne oppgaven.

En prebetingelse for å kunne gjennomføre noen form for indeksering, søk eller filtrering relatert til ovennevnte punkter er at man henter ut chat-dialoger for videre behandling. I henhold til Haveliwala [Hav02] kan dette gjøres ved å bruke en bot som logger all aktivitet til fil. Ut fra tidligere erfaringer og som nevnt i kapittel 5 ble denne fremgangsmåten i en tidlig fase vurdert. Allikevel ble dette forkastet ettersom det har blitt en ”strengere” kontroll av kanaler. Automatiske bot’er ikke uten videre akseptert og det ansees ikke som god politikk å sette opp en slik uten å avklare dette med kanaloperatørene. Dessuten kan slike bot’er lett avsløres siden de ikke er menneskelig styrt. Et semi-automatisk konsoll eller script med AI-funksjonalitet(kunstig intelligens) integrert kunne dog vært en løsning for å bøte på dette. Allikevel ville det ikke vært en tilstrekkelig løsning for denne oppgavens omfang siden ønskelig funksjonalitet beskrevet krever en aktiv klientforbindelse.

7.2.2 Implementert funksjonalitet

Implementert funksjonalitet deles inn i to sammenhengende deler: Systemets motor(engine) og det grafiske brukergrensesnittet(GUI). Hver del beskrives nærmere gjennom de neste avsnittene.

```
public class IncommingTraffic implements Runnable{
 ...
 try{
     File filename = new File(filedir, channel);
     filename.createNewFile();

     RandomAccessFile writeChanText =
                     new RandomAccessFile(filename, "rw");

     writeChanText.seek(writeChanText.length());
     writeChanText.writeBytes("<" + nick + "> " + message + "\n");
 }
 catch(IOException e){
       System.out.println(e);}
 ...
}
```

Figur 7.1: Kodeutdrag - Ekstrahering av kanaldialoger

7.2.2.1 Systemets motor

Det vil være for omfattende å gå ned i hver minste detalj i Silent derfor presenteres de viktigste delene som er relevante i henhold til denne oppgavens omfang. For konkret detaljering og kildekode se Tillegg A.

Silent har integrert de "vanligste" funksjoner det forventes av en ircklient. I bunnen finner vi klasser for tilkobling mot irc-tjenere hvor metoder for registrering av maskinnavn, brukernavn og kallenavn er tilgjengelige. Denne informasjonen har foreløpig blitt kodet inn statisk. Det er heller ikke lagt inn støtte for tilkoblinger mot flere samtidige irc-tjenere siden jeg har begrenset meg til et av de større nettverkene. Som et ledd i det å identifiseres på nettverket er det skrevet kode som gjør at klienten svarer på ping- og CTCP(Client-To-Client-Protocol) versjonsforespørsler hvor sistnevnte sørger for at klienten ikke forveksles med en automatisk bot som ville vært tilfelle med Eggdrop. I forbindelse med kanaler finnes det et utvalg metoder som gjør det enkelt å kommunisere med andre brukere, f.eks sorterte brukerlister, informasjon hvis/når brukere kommer til eller forlater kanalen, temaskifte, endring av kallenavn og handlinger utført av operatører(brukere sparkes ut og lignende).

Med tanke på indekseringsfunksjonalitet er det skrevet metoder for nettopp dette formålet. Kanaldialoger ekstraheres fra de kanaler man måtte befinne seg på og skrives ut til dynamiske filer. Loggføringen gjennomgår

```
public class IncommingTraffic implements Runnable{
  ...
  if(message.indexOf("#") >= 0){

  message = message.substring(message.indexOf("#"));
  commands.joinCrawledChannel(message);
  }
  ...
```

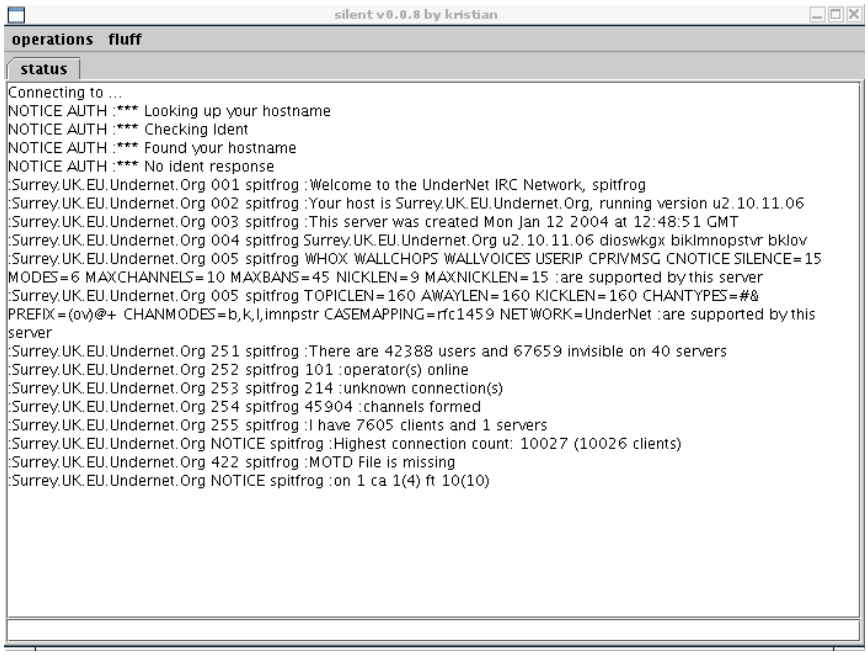
Figur 7.2: Kodeutdrag - Klienten melder seg på skjulte/hemmelige kanaler

en form for preprosessering slik at "støy" og annet fyll filtreres bort og destininasjonsfilene kun inneholder kallenavn og relatert dialog. Kanaler logges kontinuerlig og evt nye man tar del i går gjennom samme rutine, forøvrig brukes kanalnavnet som loggfil slik at noenlunde struktur bevares. Forlater man en kanal, men siden kommer tilbake igjen legges ny dialog til den opprinnelige filen. Innenfor rammene av denne oppgaven fokuseres det på kun et av de større nettverkene, Undernet, slik at ytterligere inndeling utelates. Et eksempel ville vært hvis man var på to kanaler med samme navn fordelt på to forskjellige nettverk. Kode relatert til loggingen illustreres i figuren 7.1.

I henhold til designet var det også ønskelig med metoder som forsøker å finne frem til ny informasjon(kanaler) og evt skjulte kanaler. Dette har blitt løst ved å ta utgangspunkt i kanaler og informasjon man allerede kjenner til. Silent inneholder dog ikke agentfunksjonalitet som vi f.eks finner i Butterfly [DLM99] hvor mange kanaler samples simultant. Isteden har det blitt skrevet en metode som søker etter tekststrenger prefikset med # tegnet myntet på å filtrere ut kanalnavn. Filtreringen er sanntidsbasert og skjer således før dialogene skrives til filer. Filtreringen tjener i praksis to sammenhengende formål:

1. Kanaler som filtreres ut er interessante fordi de kan være relevante mhp på informasjon/tema som diskuteres
2. Kanaler som filtreres ut kan være "skjulte" og dermed interessante å undersøke nærmere. Slike kanaler er i utgangspunktet ikke søkbare på irc-tjenerne(de vises ikke i kanallisten)

Den videre analysen av kanaler kan gjennomføres ved å kombinere automatisk og manuell(menneskelig) interaksjon eller ved å helautomatisere prosessen. Sistnevnte omfavnes av funksjonalitet delegert ut av irc-klienten og relateres til søk- og filterfunksjonalitet tilknyttet FDS-systemet. Dette er noe jeg vil komme tilbake til senere i oppgaven. Det



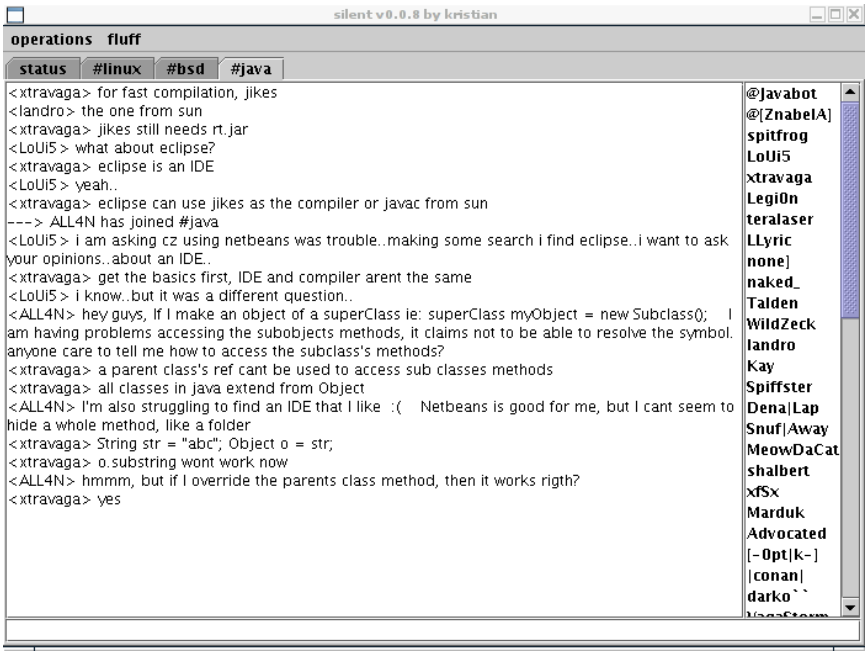
Figur 7.3: Silent i oppstartsmodus

er dog skrevet kode for en metode som gjør at klienten automatisk tar del i alle filtrerte kanaler. Funksjonen er også rekursiv i den forstand at nye kanaler som oppdages også gjennomgår samme tekstlige "analyse". Dette betyr at klienten kan crawle seg frem til en mengde nye kanaler uavhengig av om brukeren er interaktiv eller ikke. Brukerens oppgave blir primært å "undersøke" hvilke kanaler som er av mest interesse. Kode relatert til beskrevet funksjonalitet vises i figuren 7.2.

7.2.2.2 Grafisk brukergrensesnitt

Det grafiske grensesnittet er skrevet ved bruk av Swing-klassen i Java, et fleksibelt rammeverk for å skrive grafiske applikasjoner. Utgangspunktet fra et designperspektiv var at programmet skulle ta sikte på å integrere en noenlunde standardisert layout slik at navigering med hensyn på lettfattelighet og leselighet bevares. I tillegg skulle underliggende funksjonalitet kunne vises frem gjennom enkle og effektive løsninger i programmet. Resultatet av disse prebetingelsene illustreres i figurene 7.3 og 7.4. I førstnevnte finner vi en vanlig oppstartsprosedyre i klienten.

Som beskrevet i designet sendes alle tjenermeldinger som ikke er relaterte til kanaldialoger til et eget status-vindu. Ved å klikke seg inn på 'operations'-menyen finner man funksjoner for å koble seg til kanaler



Figur 7.4: Silent i interaktiv modus

Klienten i aktiv modus vises gjennom figuren 7.4. Status-vinduet og kanaler fordeles utover en ”container”-komponent i dedikerte ”tabs”. Denne løsningen ble foretrukket fordi tabs er enkle å forholde seg til og de er relativt dynamiske, dvs containeren holder styr på hvilke som er aktive og oppdaterer layout hvis det skjer endringer(fjernes/legges til). Funksjonelt sett reflekterer kanalvinduet i senter dialogene og ”følger” disse ved automatisk scrolling. Egne brukerlister som kontinuerlig oppdateres er plassert lengst til høyre for kanalvinduet. Forøvrig kan private samtaler instansieres ved å dobbeltklikke på et brukernavn(eget vindu åpnes). Under hovedvinduet finner vi input-feltet. Her kan brukeren selv taste inn tekst som sendes til den aktuelle kanalen. Indekseringsfunksjonene følger også strukturen introdusert ovenfor slik at når det oppdages nye eller skjulte kanaler fremvises disse på egne tabs og forblir der inntil brukeren lukker dem.

7.3 Fast Data Search

Fast Data Search er kjernen i ACP-systemet og danner plattform for den avanserte delen tilknyttet indeksering, søk og filtrering. Som et IR-system tjener FDS to sammenhengende formål, informasjonslagring og informasjonsgjenfinning. For nærmere detaljering rundt disse begrepe-

ne og teori forbundet med IR henviser jeg til kapittel 4.

7.3.1 Informasjonslagring

Fra et designperspektiv er informasjonen som lagres i FDS-systemet chat-dialogenene. Det som da må bestemmes er hvilken lagringsstruktur man velger. Som et utgangspunkt er chat-dialogene rene tekstfiler uten noen form for headere eller annen informasjon integrert(i motsetning til eksempelvis Usenet-meldinger). Dvs, man kan opprette en egen kolleksjon hvor alle "chat-dokumentene"(tekstfilene) samles uten at noen preprosessering eller andre former tekstlig analyse er nødvendig. Dokumentene samles i én kolleksjon hvor hvert dokument representeres ved én fil som inneholder dialogen ekstrahert fra en gitt kanal. Filnavnet indikerer hvilken kanal dialogen tilhører(f.eks #java.txt). I tillegg styres alle chat-dokumentene av en spesifisert indeksprofil. Dette innebærer at all mulig chat-dialog også omfavnes av samme cluster. Ettersom filene er strukturert på en veldig enkel måte vil filnavn være den mest naturlige parameteren å indeksere etter. For at kolleksjonen skal inneholde størst mulig grad av dokumentfriskhet er det også essensielt at dialoger gjeninnsendes straks ny tekst er tilgjengelig.

Et annet spørsmål det må taes stilling til er hvordan man skal forholde seg til tidsaspektet. Med andre ord, hvor langt skal man gå i kategoriseringen av chat uten at man beveger seg bort fra sanntidsaspektet. Det mest naturlige er å bruke "definisjonen" gitt av IR-systemet og crawlere generelt hvor man ser på tilgjengelig informasjon som "ferskest". I tillegg påføres alle chat-dokumentene informasjon tilknyttet dato og tidspunkt de ble indeksert. Denne strategien fører til nok et spørsmål som må besvares, hvor lenge skal man lagre indeksert informasjon? Innenfor rammene av denne oppgaven vil dokumentmengden være såpass begrenset at det neppe er merkbart for systemet, men i en sammenheng hvor store mengder data flyter gjennom systemet til enhver tid er det naturlig å vurdere mulige skaleringsstrategier.

7.3.2 Informasjonsgjenfinning

Som beskrevet i innledningen av kapitlet delegeres søk og navigasjon mot indeksert data i stor grad til FQT som er et grafisk brukergrensesnitt mot FDS.

FDS-systemet har innebygget støtte for gruppering og clustering av dokumenter, hvor sistnevnte er mest relevant for denne oppgaven. Under følger en kort beskrivelse av funksjonalitet som tas i bruk av ACP-

systemet:

Clustering: Clustering brukes først og fremst for å spre søk over og innenfor hele dokumentkolleksjoner. En effekt av dette er at informasjon man i utgangspunktet ikke søker etter også avdekkes.

Dynamiske "drill-down"-menyer: "Drill-down"-meyene er dynamiske og brukes primært for å navigere innenfor dokumenter. Hvilke elementer som gjøres tilgjengelige via menyen spesifiseres ved indeksprofilen. Innenfor ACP-systemet brukes "drill-down"-menyene til å navigere i returnerte chat-dialoger basert på felter som navn, kanaltilhørighet og dokumentstørrelse.

Resultat-perspektiv Resultatperspektiv er som beskrevet i kapittel 6 en løsning som brukes for å skreddersy brukerens ønsker, dvs hvordan man ønsker returnerte resultater presentert. Innenfor ACP-systemet brukes kun et perspektiv for chat-dialogene hvor (fil)navn og teaser presenteres.

Rangering og filtrering: Det finnes mulighet for manuell innsetting av egendefinerte verdier i rangeringsalgoritmen i FDS-systemet slik at chat-dialogene kan prioritetsrangeres. En interessant fremgangsmåte kan være å rangere dialoger med utvalgte ord fremfor andre evt omvendt. Resultat blir en form for "interessemodell" hvor man filtrerer ut chat-dialog den ene eller andre veien basert på predefinerte kriterier.

7.3.3 Implementert funksjonalitet

Gjennom denne seksjonen beskrives implementert funksjonalitet som er relevant i forhold til FDS og denne oppgavens omfang. Beskrivelsen er direkte knyttet til teori beskrevet i kapitlene 5 og 6 og disse sees derfor på som essensielt bakgrunnsstoff.

Som beskrevet i kapittel 6 identifiseres hver dokumentkolleksjon med minimum en indeksprofil og en pipeline. For at designkriteriene beskrevet skal oppfylles må bestemte deler av FDS-systemet konfigureres. Implementert funksjonalitet kan deles inn i tre sammenhengende deler:

7.3.3.1 Friskhet

Som beskrevet i kapittel 6 finnes det et utvalg eksterne moduler som kan brukes til å sende inn dokumenter til FDS-systemet. Det mest naturlig valget for denne oppgaven ble vertøyet *Filetraverser*. Denne traverserer

filhierarkiet rekursivt og sender inn dokumentbolker til Fast Data Search. Et kriterium som må oppfylles er at hver dokumentbolk sendes inn til en predefinert kolleksjon.

Filtraversereren eksekveres med jevne intervall via en cron-job slik at ny eller oppdatert chat-dialog sendes inn til systemet når denne er tilgjengelig. De originale filene blir værende i samme katalog slik at ny dialog blir lagt til hvis filen eksisterer eventuelt opprettet hvis den ikke finnes fra før. Dette betyr at all dialog som assosieres med en kanal tilhører samme dokument-enhet. En alternativ fremgangsmåte kunne vært å betrakte hver melding som et selvstendig dokument, men dette ville sannsynligvis ført til enorme dokumentmengder og har derfor blitt forkastet.

Som beskrevet i kapittel 6 indentifiseres hvert dokument med en unik id(URI). Det mest naturlige innenfor denne oppgaven har vært å benytte seg av kanalnavnet som unik id. Hvert chat-dokument blir også postfikset med en txt-endelse slik at systemet vet at det er rene tekstfiler.

7.3.3.2 Indeksprofil

En indeksprofil er en XML-basert konfigurasjonsfil som bestemmer hvordan dokumenter gjøres søkbare. Den definerer hvilke dokumentelementer som blir søkbare felter, hvilke felter som skal returneres som en del av resultatperspektivet og hvordan verdier for rangering og sortering beregnes. Chat-dialoger, i form av tekstdokumenter, er så enkle i strukturen at det har vært tilstrekkelig å bruke en helt standard indeksprofil lik denne levert med FDS-systemet. Unntaket er for felter av naturlig tekst som lemmatiseres. Under følger en beskrivelse av de mest aktuelle elementene i indeksprofilen.

Standard felter Standard felter inneholder informasjon ekstrahert fra dokumentene i tillegg til meta-informasjon påført gjennom dokumentprosesseringen. Data ekstrahert fra dokumentene organiseres i felter for Tittel, body, overskrifter, ankertekst og dato. Metainformasjons som påføres grupperes i felter for dokumenttype, språk, tegnsett, URI, modifiseringstidspunkt, størrelse, rangering og teaser.

Logiske felter Logiske felter brukes for å gruppere standard felter slik at man kan aksessere flere felter simultant. Hvert logisk felt kan også implementere rangering spesifikt for feltet. Indeksprofilen brukt i denne oppgaven spesifiserer ”dokument” som et logisk felt hvor body, overskrifter, tittel og ankertekst betraktes som ett felt. På

grunn av chat-dialogenes enkle struktur bør det nevnes at ikke alle disse feltene vil ha innhold. Overskrift- og tittelfeltene vil eksempelvis være tomme.

Resultatfelter Resultatfelter brukes for å bestemme hvordan dokumenter returnert fra FDS-systemet presenteres. Innenfor denne oppgaven presenteres dokumentene ut fra følgende kriterier:

- **Resultatfilter:** Basert på URL-feltet fjernes eventuelle duplikater. Dvs, indekserte chat-kanaler forekommer kun en gang i resultatlisten.
- **Navigasjonspunkter:** Flere navigasjonspunkter er definert i indeksprofilen. Noen av disse inkluderer størrelse, dokumenttype, tegnsett, språk og dato
- **Resultatperspektiv:** Chat-dialogene presenteres kun på en form hvor dato, kanalnavn(URL) og teaser presenteres

7.3.3.3 Dokumentprosessering

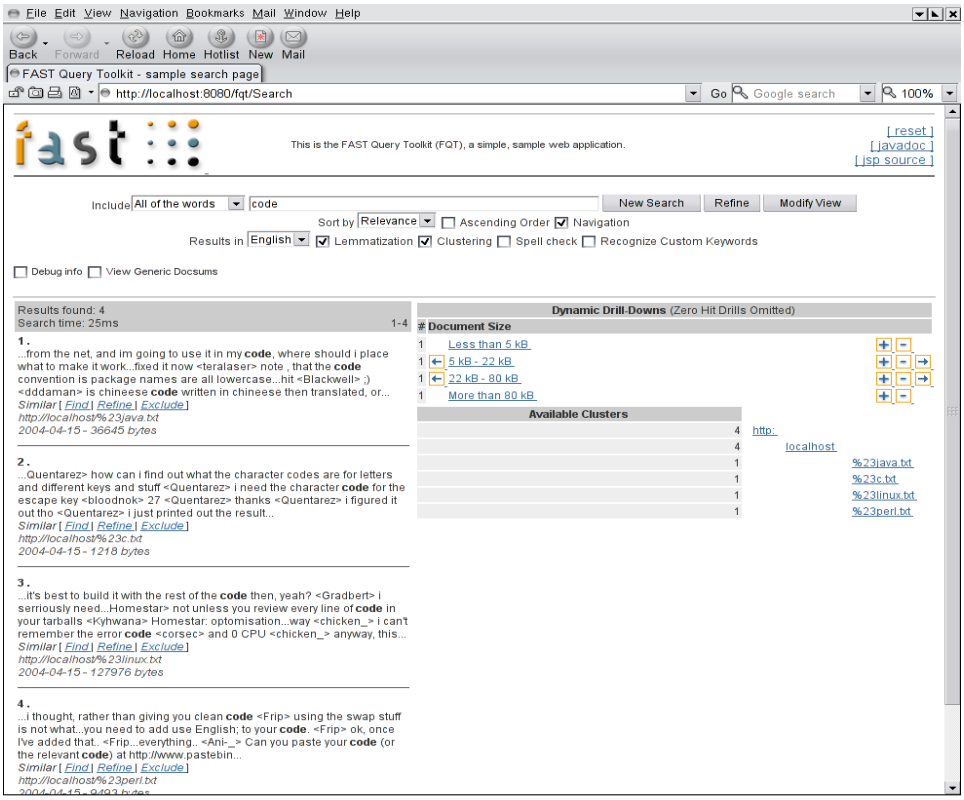
Fast Data Search har en mengde predefinerte pipelines integrert. For denne oppgaven var det mest naturlig å benytte seg av en *MultiConverter*(Multi Purpose Pipeline) som håndterer flere forskjellige dokumentformater. Rene tekstfiler er som nevnt tidligere veldig enkle i struktur og er derfor en relativt smal sak å prosessere for systemet.

7.4 Fast Query Toolkit

Gjennom denne seksjonen presenteres designmessige forhold rundt brukergrensesnittet mot FDS-systemet, dette innebærer eksisterende funksjonalitet og mulige løsninger i et fremtidig system.

Fokuset gjennom denne oppgaven har ikke vært å lage et fullverdig system som kan settes inn i en produksjonssammenheng, men et system som undersøker forskjellige former for automatisk prosessering av informasjon tilknyttet chat-systemet irc. Resultatet av dette er ACP-systemet. ACP er et todelt system med grensesnitt mot irc-nettverket(Silent) og dataene i FDS(FQT). Målet har vært å gjøre tilgjengelig data søk- og filtrerbart via navigasjonsverktøyet Fast Query Toolkit.

Beskrivelsen av design og layout tar utgangspunkt i figuren 7.5 som viser hvordan FQT benyttes mot FDS-systemet.



Figur 7.5: Fast Query Toolkit anvendt mot indekserte kanaler

7.4.1 Søkefunksjonalitet

Det er mange måter en kan presentere dataene på i et brukergrensesnitt, men for denne oppgavens del benyttes standard layout gitt av

FQT-verktøyet. Målet har ikke vært å presentere et fullstendig design, men allikevel er det naturlig å trekke inn mulige fremtidige løsninger. Foruten inntastingsfelt for aktuelle spørringer og tilhørende funksjonalitet deles brukergrensesnittet inn i to ledende komponenter:

Hovedpanel Hovedpanelet er den ledende komponenten i grensesnittet og utgjør startpunktet for browsing og søk. Returnerte dokumenter vises også frem i hovedpanelet. Følgende opsjoner er tilknyttet navigering i hovedpanelet:

- Finne "like" dokumenter basert på returnerte resultater
- Funksjon for å ekskludere og raffinere returnerte dokumenter

Navigasjonspanel Navigering innenfor ACP-systemet gjørs mulig gjennom dynamiske "drill-down"-menyer. Menyene defineres i indeksprofilen og returnerte resultater kategoriseres basert på hva som faktisk returneres og konfigurasjonen av menyene. Det finnes også mulighet for å kombinere flere "drill-down"-menyer slik at disse representeres som én meny.

Av figuren 7.5 ser vi at et mulig navigasjonspunkt er størrelse. Størrelsen på et dokument angir i stor grad hvor aktiv en kanal faktisk er. Andre aktuelle navigasjonspunkter er tegnsett og språk. Chat-dokumentene som er tilgjengelige illustreres ved en trestruktur ordnet etter grad av relevans.

Clustering brukes primært for å utbre søk, men kan også benyttes for å gruppere returnerte resultater i cluster avhengig av hvor like dokumentene antas å være. Konfigurasjon tilknyttet denne funksjonaliteten defineres også gjennom indeksprofilen.

7.4.2 Filterfunksjonalitet

Funksjonalitet beskrevet i denne seksjonen har ikke blitt implementert, men er allikevel av interesse for oppgaven.

Filtrering i en IR-kontekst baseres i utgangspunktet på brukerprofiler som beskrevet i teori-kapitlet. Profilene kan enten være dekkende for enkeltpersoner eller større grupper, dette betyr at det må finnes en form for skille på brukerne av systemet. For at dette skal gjennomføres i ACP-systemet må man i større grad delegere klientfunksjonalitet inn i FQT-verktøyet. I en produksjonssammenheng kunne dette være en interessant fremgangsmåte, evt utvide det nåværende systemet til en slik tilstand at det det skiller på forskjellige brukere. Fra et teknisk perspektiv kan slik funksjonalitet integreres ved direkte manipulering av sanntidsfilteret ved Java API'et beskrevet i kapittel 6. Gjennom dette rammeverket tilbys klasser og metoder som interakterer direkte med sanntidsfilteret og således er en alternativ migreringsvei.

Det finnes flere interessante innfallsvinkler til hvordan man kan generere profiler. Under følger noen ideer som relateres til dette.

- Generere varselspørringer basert på brukerspørringer: En interessant fremgangsmåte er at systemet automatisk tilbyr å generere varselspørringer på grunnlag av brukerspørringer som ikke får treff i indekserte dokumenter. Fordelen ved en filterstrategi er som beskrevet i kapittel 6 at fremtidig informasjon kan fanges opp

- Generere varselspørringer basert på dokumenter: Dvs, ved å implementere en metode basert på vektormodellen hvor en liste over de mest vesentlige ordene i et gitt dokument vedlikeholdes kan brukeren velge ut og evt vekte ord mhp på relevans. En editert liste sendes inn til systemet i form av en varselspørring

7.5 Om utviklingsmiljøet

Utviklingen har vært en todelt prosess. Silent ble kodet på en Dell Latitude C640 med operativsystemet Debian GNU/Linux, der Vim ble brukt som editor. Java(TM) 2 Runtime Environment, Standard Edition (build Blackdown-1.4.1-beta) ble brukt som utviklingsverktøy sammen med byggeverktøyet Ant [ANT].

Installasjon, konfigurering og testing av Fast Data Search ble gjort på en pc med operativsystemet Red Hat Linux 7.3, fysisk plassert ved Universitet i Oslo, Institutt for Informatikk sine lokaler. ACP-systemet ble en realitet ved å hente over alle nødvendige programmer og bibliotek til denne maskinen og testene ble eksekvert over en ssh-forbindelse med portforwarding aktivert. Tilgjengelig data har blitt søkt gjennom og navigert ved bruk av nettleseren Opera.

I tillegg kan det nevnes at ACP-systemet ikke er spesielt avhengig av et bestemt miljø, men kan tilpasses flere plattformer selv om utviklingen innenfor denne oppgaven hovedsaklig har blitt utført over Linux maskiner.

Oppsummering

Gjennom dette kapitlet har ACP-systemet blitt beskrevet, et eksperimentelt system som har blitt implementert i forbindelse med denne oppgaven. Formålet med systemet har ikke vært å produsere en komplett løsning, men heller et system som undersøker forskjellige former for automatisk prosessering av informasjon tilknyttet chat-systemet irc.

Kapittel 8

Konklusjon

Gjennom den overordnende problemstillingen introdusert i innledningen av oppgaven ble chat-systemet irc beskrevet som problemdomene. Grovt sett har målet vært å finne ut hvordan dette systemet kan gjøres søk- og filtrerbart gjennom avansert indekseringsfunksjonalitet. En mer presis og detaljert formulering av problemstillingen har blitt uttrykt gjennom flere relaterte delproblemer:

Hvordan kan produktet Fast Data Search brukes som teknologisk plattform til å indeksere, søke og filtrere i chat-system IRC og hvordan kan man presentere indeksert data med formålet om å lette tilgjengelighet og lesbarhet?

Det har blitt implementert en prototype på et system kalt ACP som forsøker å løse disse problemene. Indekseringsfunksjonalitet gjøres mulig gjennom en dedikert klient mot irc-nettverket som eksporterer chat-data til Fast Data Search-systemet. Et alternativt grensesnitt mot indeksert data finnes i form av et J2EE-verktøy kalt Fast Query Toolkit som integrerer både søk- og filterfunksjonalitet. Løsningen virker ut fra de testene som har blitt eksekvert til å være relativt robust og mulige utvidelser kan være interessante.

Hvordan skal man gå frem for å indeksere, søke og filtrere skjulte eller hemmelige chat-kanaler og er disse kanalene mer interessante/nyttige enn de som finnes fritt tilgjengelig?

Det finnes flere aktuelle fremgangsmåter, men for denne oppgaven har slik funksjonalitet blitt delegert til grensesnittet mot irc-nettverket, dvs Silent. Løsningen baseres på kanaler man allerede kjenner til/er en del av og det forsøkes å søke frem til nye eller skjulte kanaler ved å ekstrahere kanalnavn fra chat-dialogene. Kanaler som oppdages gjennomgår samme tekstlige analyse som ”vanlige” kanaler og dialogen eksporteres

til FDS-systemet.

Løsningen viste seg å være langt mer effektiv enn først antatt og kan derfor sees på som hensiktsmessig i forhold til problemet.

Om skjulte/hemmelige kanaler er mer interessante eller nyttige i forhold til vanlige kanaler finnes det intet entydig svar på. Motivasjonen for å skjule en kanal er ikke nødvendigvis at det som utveksles er av stor verdi, det kan like godt skyldes at brukerne ønsker mindre tilfeldig trafikk. Erfaringene gjennom denne oppgaven har både vært gode og mindre gode, det man allikevel kan slå fast er at slike kanaler i utgangspunktet er interessante å undersøke nærmere.

Er det realistisk å se for seg at klienter mot irc-nettverket i fremtiden integrerer støtte for informasjonsgjenfinning(søk/filter) eller vil det være for ressurskrevende å implementere slik funksjonalitet?

Sannsynligvis er det ikke realistisk å se for seg funksjonalitet vi finner i et kraftig produkt som Fast Data Search integrert i irc-klienter, men dette er ikke ensbetydende med at det vil være for ressurskrevende å implementere lignende funksjonalitet. Mulige lettvektsversjoner som skaleres ned til å passe inn i en irc-kontekst kan være en interessant løsning og således ikke usannsynelig å se for seg i fremtidige klienter.

Er det mulig, sett fra et teknisk perspektiv, å skille informasjonssøk og overvåkning fra hverandre og hvordan reagerer irc-miljøet hvis informasjonssøkingen blir oppfattet som overvåkning?

Om mulig, så er det hvertfall ikke spesielt trivielt. Informasjonssøk eller filtrering innebærer at man indekserer en mengde data i form av chatdialoger, dette betyr at all dialog i utgangspunktet gjøres tilgjengelig uavhengig om brukerne godkjenner dette eller ikke. En generell oppfatning i miljøet er at all form for indeksering enten ved agenter eller variasjoner av dette som brukt i denne oppgaven kategoriseres som overvåkning. En effekt av dette er at kanaler skjules og gjøres hemmelige, men også at toleransegrensen for tilfeldig trafikk på kanalene innsnevres.

Kapittel 9

Videre Arbeid

Gjennom denne delen presenteres noen ideer for hva som kan være hensiktsmessig å se på videre. Dette innebærer konkrete forslag til utvikling av ACP-systemet og generelle design- og forskningsideer forankret i systemet eller teorien.

Utvidelser av implementasjonen:

Et relativt utbredt fenomen i irc-miljøet har blitt å bruke egne forkortelser og omskrivninger av ord og uttrykk. Slik kan det være vanskelig å oppfatte sammenhengen i samtaler for en utenforstående. En interessant fremgangsmåte for å bøte på dette kunne være å legge inn en flat liste, eventuelt integrere en database i systemet som inneholder den faktiske betydningen av de mest kjente/brukte forkortelsene. Ved treff i henhold til lagrede ord og uttrykk skriver systemet ut de fullstendige ordene/setningene slik at dialogene ikke bærer preg av slang eller overdrevent muntlig språk.

Et interessant aspekt ved ACP-systemet ville i tillegg være å implementere en løsning som skalerer. Fast Data Search som underliggende plattform er allerede tilrettelagt for skalering, men ACP som helhet støtter ikke slik funksjonalitet. En mulig fremgangsmåte kunne være å parallellisere klient-prosessen mot irc-nettverket og fordele lasten på flere noder. Slik kunne man indeksert enormt mange kanaler simultant og og teoretisk sett lagd en søkemotor for irc. Dette betyr at det vil være nødvendig med et dedikert miljø hvor man kan prøve ut forskjellige konfigurasjoner av systemet. I en slik kontekst kunne det også være interessant å ”benchmarke” systemet og på dette viset finne frem til den konfigurasjon som gir best ytelse.

Teoretiske utvidelser:

Slik ACP-systemet er konstruert finnes det ingen mulighet for brukerne å gi eksplisitte tilbakemeldinger på hvor relevante chat-dialogene faktisk er. Den nåværende implementasjonen antar at returnerte resultater er endelige og dermed oppfyller de krav brukerne måtta ha. En mulig innfallsvinkel ville være å trekke inn elementer fra recommender-teori slik at brukerne tilbys intuitive funksjoner for å kvalitetstbedømme resultatene. Hvis systemet så "lærer" av disse tilbakemeldingene ville man over tid finjustere systemet og dermed eget informasjonsbehov gjennom en feedback prosess.

Tillegg A

Kildekode

Vedlagt følger programkoden skrevet ved utvikling av irc-klienten Silent, brukt som en del av ACP-systemet i denne oppgaven. Programkoden struktureres i henhold til tre ledende komponenter: Systemets motor(engine), lyttere(listeners) og det grafiske brukergrensesnittet(gui) mot irc-nettverket. Systemet startes ved at Main-metoden vist under eksekveres.

A.1 Main

```
import java.awt.*;

public class Main
{
  static GuiFrame window;

  public static void main (String[] args)
  {
   window = new GuiFrame("silent v0.0.8 by kristian");
   Toolkit toolkit = window.getToolkit();
   Dimension size = toolkit.getScreenSize();

   window.setBounds(size.width/4, size.height/4,
                    size.width/2, size.height/2);

   window.setVisible(true);
 }
}
```

A.2 Engine

A.2.1 ConnectToServer

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

```
public class ConnectToServer implements Runnable
{
  public static Socket ircSocket = null;

  public void run()
  {
    try
      {
        ircSocket = new Socket("surrey.uk.eu.undernet.org", 6667);

        OutgoingTraffic outtraffic = new OutgoingTraffic();

        Thread intraffic = new Thread(new IncommingTraffic());
        intraffic.start();

        outtraffic.sendDataToServer("NICK spitfrog");
        outtraffic.sendDataToServer("USER vingilot 8 * :kfh");
      }
    catch(UnknownHostException e)
      {System.out.println(e);}
    catch(IOException e)
      {System.out.println(e);}
    }

  public static void disconFromServer()
  {
    try
      {
        ircSocket.close();
      }
    catch(IOException e)
      {System.out.println("Disconnecting from server");}
   }
}
```

A.2.2 IncommingTraffic

```
import java.io.*;
import java.util.*;

public class IncommingTraffic implements Runnable
{
  private String supernick = "spitfrog";
  private BufferedReader instream = null;
  private Commands commands;
  private boolean done;

  public void run()
  {
    commands = new Commands();

    try
      {
        instream = new BufferedReader(
                   new InputStreamReader(
                   new DataInputStream(ConnectToServer.ircSocket.getInputStream())));

        String inText;

        while((inText = instream.readLine()) != null)
        {
          done = false;
          validateInStream(inText);
        }

        instream.close();

      }
    catch(IOException e)
      {System.out.println(e);}
   }

  /**validate the instream*/

  public void validateInStream(String valtext)
  {
    /**ping request*/
    if(valtext.startsWith("PING :"))
```

```
   {
     commands.answerPing(valtext);
     done = true;
   }

  /**error message*/
  else
  if(valtext.startsWith("ERROR"))
   {
      commands.errorMessage(valtext);
      done = true;
   }

 /**banned from channel*/
 else
 if (valtext.indexOf("Cannot join channel") > 0 && valtext
     .indexOf("PRIVMSG") < 0 && valtext.indexOf("474") >= 0)
  {

    String channel;

    /**some servers don't write ':' before error message*/
    if(valtext.indexOf(" :Cannot") > 0)
     {
       channel = valtext.substring(valtext.indexOf(" ",valtext
                 .indexOf(supernick)) + 1,
       valtext.indexOf(" :",valtext.indexOf(supernick)));
     }

     else
     {
      channel = valtext.substring(valtext.indexOf(" ",valtext
                .indexOf(supernick)) + 1, valtext
                .indexOf(" Cannot",valtext.indexOf(supernick)));
     }

      commands.bannedFromChannel(channel);
      done=true;
    }

  /**moderated channel*/
  else
  if(valtext.indexOf("Cannot send to channel") > 0 && valtext
     .indexOf("PRIVMSG") < 0 && valtext.indexOf("404") >= 0)
   {
     String channel;

     /**some servers don´t write ':' before error message*/
     if(valtext.indexOf(" :Cannot") > 0)
      {
        channel = valtext.substring(valtext.indexOf(" ",valtext
                  .indexOf(supernick)) + 1,
                  valtext.indexOf(" :", valtext.indexOf(supernick)));
      }
     else
     {
       channel = valtext.substring(valtext.indexOf(" ",valtext
                 .indexOf(supernick)) + 1,
                 valtext.indexOf(" Cannot", valtext.indexOf(supernick)));
      }

       commands.handleModeratedChannel(channel);
       done=true;
    }

  /**user in query disappears*/
  else
  if(valtext.indexOf("No such nick/channel") > 0 && valtext
     .indexOf("PRIVMSG") < 0 && valtext.indexOf("401") >= 0)
   {
     String nick;

     /**some servers don´t write ':' beforce error message*/
     if(valtext.indexOf(" :No") > 0)
      {
        nick = valtext.substring(valtext.indexOf(" ", valtext
               .indexOf(supernick)) + 1,
               valtext.indexOf(" :", valtext.indexOf(supernick)));
       }
      else
      {
        nick = valtext.substring(valtext.indexOf(" ", valtext
               .indexOf(supernick)) + 1,
               valtext.indexOf(" No", valtext.indexOf(supernick)));
```

```
      }
        commands.noNickNoChannel(nick);
        done=true;
    }

/**some nets don't use /NAMES*/
else
if(valtext.indexOf(":End of") >= 0 && valtext.indexOf("NAMES list") >= 0
  && valtext.substring(valtext.indexOf(":") + 1, valtext.indexOf(" "))
  .indexOf("!") < 0)
  {
    String channel = valtext.substring(valtext.indexOf(" ", valtext.indexOf
    (supernick)) + 1, valtext.indexOf(":End") - 1).toLowerCase();

    //Legg til tab, f.eks addTab(channel)

   done = true;
  }

/**if no error message, no ping etc..*/
if(!done)
{
  try
  {
    String info = valtext.substring(valtext.indexOf(":") + 1, valtext
    .indexOf(" "));

    if(valtext.startsWith(":"+ supernick +"!"))
     {
       String command = valtext.substring(valtext.indexOf(" ") + 1, valtext
       .indexOf(" ", valtext.indexOf(" ") + 1));
       String channel;

       if(command.equals("JOIN"))
        {
          if(valtext.indexOf("JOIN :") > 0)
           {
             channel = valtext.substring(valtext.indexOf(" :", 1) + 2);
           }

          else
          {
            channel = valtext.substring(valtext.indexOf("JOIN ") + 5);
          }

          commands.joinChannel(channel.toLowerCase());

        }

       else
       if(command.equals("NICK"))
        {
          String newuser;
          if(valtext.indexOf("NICK :") > 0)
           {
             newuser = valtext.substring(valtext.indexOf(" :", 1) + 2);
           }
          else
          {
            newuser = valtext.substring(valtext.indexOf("NICK ") + 5);
          }

          commands.changeNick(supernick, newuser);
        }

       else
       if(command.equals("TOPIC"))
        {
          channel = valtext.substring(valtext.indexOf("TOPIC") + 6, valtext
          .indexOf(" ", valtext.indexOf("TOPIC") + 6));

          commands.setTopic(channel.toLowerCase());
         }
       }

       else
       if(info.indexOf("!") >= 0)
        {
          String nick = info.substring(0, info.indexOf("!"));
          String hostmask = info.substring(info.indexOf("!") + 1);
          String command  = valtext.substring(valtext.indexOf(" ") + 1,
                            valtext.indexOf(" ", valtext.indexOf(" ") + 1));
          String channel;
          String message;
```

```
        String topic;
        String newnick;

        if(command.indexOf("PRIVMSG") >= 0)
         {
           channel = valtext.substring(valtext.indexOf("PRIVMSG") + 8,
                    valtext.indexOf(" ", valtext.indexOf("PRIVMSG") + 8));

           message = valtext.substring(valtext.indexOf(" :",1) + 2);


           /**ctcp and regular query*/
           if(channel.toLowerCase().equals(supernick.toLowerCase()))
            {
              if(message.equals("\001VERSION\001"))
               {
                 commands.userVersionRequest(nick);
               }

              else
              if(message.startsWith("\001PING"))
               {
                 //long rtime = Long.parseLong(message
                 //.substring(7, message.length() - 1));

                 //long ltime = new Date().getTime() / 1000;

                 //long time = new Long(ltime-rtime);

                 commands.userPingRequest(nick);
               }

              else
              {
                commands.userQueryMessage(nick, message);
              }

            }

           /**investigate channel messages*//
           if(message.indexOf("#") >= 0)
            {
                message = message.substring(message.indexOf("#"));

                commands.joinCrawledChannel(message);
            }

           /**random user sends hannel message*/
            else
            {
              commands.userChannelMessage(channel
              .toLowerCase(), message, nick);

              File filedir = new File("/home/kristian/silent/logs/");

              if(!filedir.exists())
               {
                 filedir.mkdir();
               }

              else
              if(!filedir.isDirectory())
               {
                 System.out.println(filedir + " is not at directory");
               }

              try
                {
                  File filename = new File(filedir, channel);
                  filename.createNewFile();

                   RandomAccessFile writeChanText =
                                  new RandomAccessFile(filename, "rw");

                  writeChanText.seek(writeChanText.length());
                  writeChanText
                  .writeBytes("<" + nick + "> " + message + "\n");
                }
              catch(IOException e)
              {System.out.println(e);}
            }
          }

      /**random user changes topic*/
```

```
else
 if(command.indexOf("TOPIC") >= 0)
  {
    channel = valtext.substring(valtext.indexOf("TOPIC") + 6, valtext
              .indexOf(" ", valtext.indexOf("TOPIC") + 6));
    topic = valtext.substring(valtext.indexOf(":", valtext.indexOf(
            "TOPIC " + channel + " :")) + 1);

    commands.userChangesTopic(channel.toLowerCase(), nick, topic);
   }

/**random user parts channel*/
else
if(command.indexOf("PART") >= 0)
 {
   if(valtext.indexOf(" ", valtext.indexOf("PART") + 5) >= 0)
    {
      channel = valtext.substring(valtext.indexOf("PART") + 5, valtext
                .indexOf(" ", valtext.indexOf("PART") + 5));

      message = valtext.substring(valtext.indexOf(":", valtext
                .indexOf("PART")) + 1);
     }
    else
    {
      channel = valtext.substring(valtext.indexOf("PART") + 5);
      if(channel.startsWith(":"))
       {
         channel = channel.substring(1);
       }

      message = "";
     }

     commands.userPartsChannel(nick, channel.toLowerCase(), message);
   }

/**random user joins channel*/
else
if(command.indexOf("JOIN") >= 0)
 {
   if(valtext.indexOf("JOIN :") > 0)
    {
      channel = valtext.substring(valtext.indexOf(" :", 1) + 2);
    }

   else
   {
     channel = valtext.substring(valtext.indexOf("JOIN ") + 5);
   }

    commands.userJoinsChannel(nick, channel.toLowerCase());
  }

/**random user quits irc*/
else
if(command.indexOf("QUIT") >= 0)
 {
   if(valtext.indexOf("QUIT :") >= 0)
    {
      message = valtext.substring(valtext.indexOf("QUIT :") + 6);
    }

   else
   {
     message = valtext.substring(valtext.indexOf("QUIT ") + 5);
   }

    commands.userQuits(nick, message);
  }

/**random user changes nick*/
else
if(command.indexOf("NICK") >=0)
 {
   if(valtext.indexOf("NICK :") >=0)
    {
      newnick = valtext.substring(valtext.indexOf("NICK :") + 6);
    }

   else
   {
     newnick = valtext.substring(valtext.indexOf("NICK ") + 5);
   }
```

```
            commands.userChangesNick(nick, newnick);
        }

        /**random op kicks someone*/
        else
        if(command.indexOf("KICK") >=0)
         {
           channel  = valtext.substring(valtext.indexOf("KICK") + 5, valtext
                      .indexOf(" ",valtext.indexOf("KICK") + 5) );

           //i can´t convert channel to lowercase at this point
           //because the next substrings need the correct spelling

           String op = valtext.substring(valtext.indexOf(" ",valtext
                       .indexOf(channel)) + 1, valtext.indexOf(" :",
                       valtext.indexOf(channel)));

           message  = valtext.substring(valtext.indexOf(":",valtext
                      .indexOf(channel)) + 1);


         commands.userKickFromChannel(channel.toLowerCase(), nick, op, message);
         }

        /**random op op's/de'ops/voice/devoices someone*/
        else
        if(command.indexOf("MODE") >= 0)
         {

           channel = valtext.substring(valtext.indexOf("MODE") + 5, valtext
                     .indexOf(" ",valtext.indexOf("MODE") + 5) );

           String mode = valtext.substring(valtext.indexOf(" ", valtext
                         .indexOf("MODE") + 5) +1, valtext.indexOf(" ",
                         valtext.indexOf("MODE") + 5) + 3);

           String op = valtext.substring(valtext.indexOf(" ", valtext
                          .indexOf(mode)));

           if(mode.equals("+v") || mode.equals("-v") || mode.equals("+o") ||
              mode.equals("-o") || mode.equals("+b") || mode.equals("-b"))
            {
             commands.userModes(nick, channel.toLowerCase(), mode,op.trim());
            }

           else
           {
             commands.appendToStatus(valtext);
           }
         }/**end PRIVMSG-if()*/
}

/**checking for topic and userinfo on given channel*/
else
{
  String info2 = valtext.substring(1, valtext.indexOf(":", 1));
  String message = valtext.substring(valtext.indexOf(":", 1) + 1);
  String channel = "";

  if(info2.indexOf("#") >= 0)
   {
     channel = info2.substring(info2.indexOf("#") , info2.indexOf(" ",
               info2.indexOf("#")));
    }

  /**if topic exists, put on tab?*/
  else
  if(info2.indexOf("+") >= 0)
  {
    channel = info2.substring(info2.indexOf("+") , info2
              .indexOf(" ", info2.indexOf("+")));
  }

   if(info2.indexOf(supernick + " #") >= 0 && !message
                       .startsWith("End") || info2.indexOf(
                       supernick + " +") >= 0 && !message
                       .startsWith("End"))
   {

     //commands.setTopicOnChannelFrame(channel.toLowerCase(), message);
   }

  /**adds user to nicklist*/
```

```
      else
      if(info2.indexOf(supernick + " = ") >= 0 ||
         info2.indexOf(supernick + " @ ") >= 0 ||
         info2.indexOf(supernick + " * ") >= 0)
        {
          commands.addUserToUserList(channel.toLowerCase(), message);
        }

        else
        {
          commands.appendToStatus(valtext);
         }

       }
     }
     catch(StringIndexOutOfBoundsException e)
     {commands.appendToStatus(valtext);}

     } //if(!done)

   } //public void valtext
}
```

A.2.3 OutgoingTraffic

```
import java.io.*;

public class OutgoingTraffic
{
  public static DataOutputStream outstream = null;

  public OutgoingTraffic()
  {
    try
      {
        outstream = new DataOutputStream(ConnectToServer.ircSocket
                        .getOutputStream());
      }
    catch(IOException e)
      {System.out.println(e);}
   }

  public static void sendDataToServer(String data)
  {
    try
      {
        outstream.writeBytes(data + "\n");
      }
    catch(IOException e)
      {System.out.println(e);}
   }
}
```

A.2.4 Commands

```
import javax.swing.*;
import java.util.*;

public class Commands
{

  /**server ping request*/
  public void answerPing(String valtext)
  {
    OutgoingTraffic.sendDataToServer("PONG :" + valtext.substring(6) + "\n");     }

  /**error message*/
  public void errorMessage(String valtext)
  {
    InternalPanelManager.getTextArea("#status").append(valtext + "\n");
    ConnectToServer.disconFromServer();
  }
```

```
/**banned from channel*/
public void bannedFromChannel(String channel)
{
  InternalPanelManager.getTextArea("#status")
  .append(": Cannot join " + channel + "(banned)" + "\n");
}

/**moderated channel*/
public void handleModeratedChannel(String channel)
{
  if(InternalPanelManager.getTheInternalPanels().containsKey(channel))
   {
     InternalPanelManager.getTextArea(channel)
     .append(channel + " is moderated" + "\n");
   }
}

/**no such nick/channel*/
public void noNickNoChannel(String nick)
{
  if(nick.startsWith("#") || nick.startsWith("+"))
   {
     if(InternalPanelManager.containsHashtableTheKey("querypanel",nick))
      {
        InternalPanelManager
        .getTextArea(nick).append("No such nick/channel" + "\n");
      }

    }

  else
  {
    if(InternalPanelManager.containsHashtableTheKey("querypanel",nick))
     {
       InternalPanelManager.getTextArea(nick).append("No such nick/channel\n");
      }
   }

}

/**join crawled channel*/
public void joinCrawledChannel(String channel)
{
  OutgoingTraffic.sendDataToServer("JOIN " + channel);
  //CreateInternalChannelPanel.CreateChannelPanel(channel);
}

/**join channel*/
public void joinChannel(String channel)
{
  CreateInternalChannelPanel.CreateChannelPanel(channel);
}

/**change nick*/
public void changeNick(String nick, String newuser)
{
}

/**set topic*/
public void setTopic(String channel)
{
  OutgoingTraffic.sendDataToServer("TOPIC " + channel);
}

/**random user requests ctcp version*/
public void userVersionRequest(String nick)
{
  InternalPanelManager.getTextArea("#status")
  .append("Version request from " + nick + "\n");
  OutgoingTraffic.sendDataToServer("NOTICE "
  + nick + " :\001VERSION silent v0.0.8 by kristian");
}

/**random user requests ping*/
public void userPingRequest(String nick)
{
  InternalPanelManager.getTextArea("#status")
  .append("Ping request from " + nick + "\n");
  OutgoingTraffic
  .sendDataToServer("NOTICE " + nick + " :\001PING PONG");
}

/**random user sends query message to me*/
```

```
public void userQueryMessage(String nick, String message)
{
  if(InternalPanelManager.containsHashtableTheKey("querypanel", nick))
   {
     InternalPanelManager.getTextArea(nick)
    .append("<" + nick + "> " + message + "\n");
    }

 else
 {
   CreateInternalQueryPanel.createQueryPanel(nick);

     TabbedPane.addTabToPane(nick, InternalPanelManager.getQueryPanel(nick));

     InternalPanelManager.getTextArea(nick).append("<" + nick + "> " + message + "\n");

  }
 }

/**random user sends channel message*/
public void userChannelMessage(String channel, String message, String nick)
{
  if(InternalPanelManager.containsHashtableTheKey("textarea", channel))
   {
     InternalPanelManager.getTextArea(channel)
     .append("<"+ nick +"> "+ message +"\n");
   }
}

/**random user parts channel*/
public void userPartsChannel(String nick, String channel, String message)
{
  if(InternalPanelManager.containsHashtableTheKey("userjlist",channel))
   {
    if(InternalPanelManager.getUserList(channel).indexOf(nick) >= 0)
     {
       InternalPanelManager.getUserList(channel).removeElement(nick);
     }
    else
    if(InternalPanelManager.getUserList(channel).indexOf("@"+nick ) >= 0)
     {
       InternalPanelManager.getUserList(channel).removeElement("@"+nick );
     }
    else
    if(InternalPanelManager.getUserList(channel).indexOf("+" + nick ) >= 0)
     {
       InternalPanelManager.getUserList(channel).removeElement( "+" + nick);
     }

    InternalPanelManager.getTextArea(channel)
    .append("---> "+ nick +" has left " +channel +" (" + message +")" + "\n");
  }

}

/**random user joins channel*/
public void userJoinsChannel(String nick, String channel)
{
  InternalPanelManager.getUserList(channel).add(InternalPanelManager
  .getUserList(channel).getSize(), nick);

   InternalPanelManager.getTextArea(channel)
   .append("---> "+ nick +" has joined "+ channel + "\n");

}

/**random user quits*/
public void userQuits(String nick, String message)
{
  Enumeration channels = InternalPanelManager.getTheUserList().keys();
  String channel;
  String user;

  while(channels.hasMoreElements())
  {
    channel = channels.nextElement().toString();

    Enumeration UsersAtThatChannel = InternalPanelManager
    .getUserList(channel).elements();

     while(UsersAtThatChannel.hasMoreElements())
     {
       user = UsersAtThatChannel.nextElement().toString();
```

```
        if(user.indexOf(nick) >= 0)
         {
           if(user.startsWith("@"))
            {
              InternalPanelManager.getUserList(channel)
              .removeElement( "@" + nick );

              InternalPanelManager.getTextArea(channel)
              .append("---> "+ nick +" has quit IRC ("+ message +")" + "\n");

            }

           else
           if(user.startsWith("+"))
            {
              InternalPanelManager.getUserList(channel)
              .removeElement( "+"+ nick);

              InternalPanelManager.getTextArea(channel)
              .append("---> "+ nick +" has quit IRC ("+ message +")" + "\n");

            }

           else
           {
             InternalPanelManager.getUserList(channel).removeElement(nick);
             InternalPanelManager.getTextArea(channel)
             .append("---> "+ nick +" has quit IRC ("+ message +")" + "\n");

           }
           break;
         }
      }
    }
}

/**random user changes nick*/
public void userChangesNick(String oldNick, String newNick)
{
  Enumeration channels = InternalPanelManager.getTheUserList().keys();
  String channel;
  String user;

  while(channels.hasMoreElements())
  {
    channel = channels.nextElement().toString();

    Enumeration UsersAtThatChannel = InternalPanelManager
    .getUserList(channel).elements();

    int pos = 0;

     while(UsersAtThatChannel.hasMoreElements())
     {
       pos++;

       user = UsersAtThatChannel.nextElement().toString();

       if(user.indexOf(oldNick) >=0)
        {
          if(user.startsWith("@"))
           {
             InternalPanelManager.getUserList(channel)
             .set(pos-1,"@"+newNick);

             InternalPanelManager.getTextArea(channel)
             .append("---> "+ oldNick +" is now known as "+ newNick + "\n");

           }
          else
          if(user.startsWith("+"))
           {
             InternalPanelManager.getUserList(channel)
             .set(pos-1,"+"+ newNick);

             InternalPanelManager.getTextArea(channel)
             .append("---> "+ oldNick +" is now known as "+ newNick + "\n");

           }
          else
          {
            InternalPanelManager.getUserList(channel).set(pos-1,newNick);
            InternalPanelManager.getTextArea(channel)
            .append("---> "+ oldNick +" is now known as "+ newNick + "\n");
```

```
            }
           break;
          }
        }
     }
  }

  /**update status tab*/
  public void appendToStatus(String valtext)
  {
    InternalPanelManager.getTextArea("#status").append(valtext + "\n");
  }

  /**create sorted nicklist for given channel*/
  public void addUserToUserList(String channel, String nicks)
  {
    int start = 0;
    boolean foo = true;

    while(foo)
    {
      if(nicks.indexOf(" ") >= 0)
        {
          start = nicks.indexOf(" ");
          foo = true;
        }
      else
      {
        start = nicks.length();
        foo = false;
      }

      /**if op add to top of nicklist*/

      if(nicks.substring(0,start).startsWith("@"))
       {
         InternalPanelManager.getUserList(channel)
         .add(0, nicks.substring(0,start));
       }
      else
      {
      //String voicenick = nicks.substring(0,start);
      String nick = nicks.substring(0,start);
      if(nick.substring(0,start).startsWith("+") || !nick.equals(""))
       {
          //InternalPanelManager.getUserList(channel).addElement(voicenick);
          InternalPanelManager.getUserList(channel).addElement(nick);
        }
      }

       /**are there more users on the channel?*/
       if(nicks.indexOf(" ") >= 0)
         {

          nicks = nicks.substring(start + 1);
          //InternalPanelManager.getUserList(channel).addElement(nicks);
         }
      }
   }
}
```

A.2.5 StatusCommands

```
public class StatusCommands
{
   private static final String NICK = "/nick";

   public static void statusConsole(String command)
   {

     if(command.toLowerCase().startsWith(NICK))
      {
        String value = command.substring(5).trim();
        OutgoingTraffic.sendDataToServer("nick " + value);

       }
```

```
     else
     {
       InternalPanelManager.getTextArea("#status")
                           .append(command +" Unknown command\n");
      }
   }
}
```

A.3 Listeners

A.3.1 GuiFrameListener

```
import java.awt.event.*;
import javax.swing.*;

public class GuiFrameListener implements ActionListener
{
  public GuiFrame guiFrame;
  private Thread ircconnect;

  public void actionPerformed(ActionEvent e)
  {
   if(e.getActionCommand().equals("connect"))
    {
      InternalPanelManager.getTextArea("#status")
      .append("Connecting to ..." + "\n");

      ircconnect = new Thread(new ConnectToServer());
      ircconnect.start();
     }

   else
   if(e.getActionCommand().equals("disconnect"))
    {

      ircconnect.interrupt();

      OutgoingTraffic.sendDataToServer("QUIT :borf");
      ConnectToServer.disconFromServer();

      InternalPanelManager.getTextArea("#status").append("Disconnected\n");
     }

   else
   if(e.getActionCommand().equals("channel"))
    {
      new ChannelChoiceFrame(guiFrame);
     }


   else
   if(e.getActionCommand().equals("partchan"))
    {
      TabbedPane.removeTabFromPane();
    }

   else
   if(e.getActionCommand().equals("exit"))
    {
      System.exit(0);
     }
  }
}
```

A.3.2 InternalPanelComponentListener

```
import java.awt.event.*;
```

```
public class InternalPanelComponentListener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    if(e.getActionCommand().startsWith("channel"))
     {
       this.appendTextToChannelTextArea(e.getActionCommand().substring(8));
     }

    else
    if(e.getActionCommand().startsWith("query"))
    {
      this.appendTextToQueryTextArea(e.getActionCommand().substring(6));
     }
   }

  /**send text to channel textarea*/
  public void appendTextToChannelTextArea(String channel)
  {
    if(channel.equals("#status"))
     {
       StatusCommands.statusConsole(InternalPanelManager
                    .getTextField(channel).getText());
      }

    else
    if(!InternalPanelManager.getTextField(channel).getText().equals(""))
     {
       String message = InternalPanelManager.getTextField(channel).getText();
       if(message.startsWith("/"))
        {
          StatusCommands.statusConsole(message);
         }
      else
      {
        OutgoingTraffic.sendDataToServer("PRIVMSG " + channel + " :" + message);
        InternalPanelManager.getTextArea(channel)
                          .append("<" + "spitfrog" + "> " + message + "\n");
      }

      }

       InternalPanelManager.getTextField(channel).setText("");
    }

    public void appendTextToQueryTextArea(String nick)
    {
      if(!InternalPanelManager.getTextField(nick).getText().equals("") )
       {
         String message = InternalPanelManager.getTextField(nick).getText();
         if(message.startsWith("/"))
          {
            StatusCommands.statusConsole(message);
          }

         else
         {
           OutgoingTraffic.sendDataToServer("PRIVMSG "+ nick + " :" + message);
            InternalPanelManager.getTextArea(nick)
                .append("<" + "spitfrog" + "> " + message + "\n");
         }
      }
      else
      if(!InternalPanelManager.getTextField(nick).getText().equals(""))
       {
         InternalPanelManager.getTextArea("#status").append("Not connected \n");
       }

         InternalPanelManager.getTextField(nick).setText("");

    }
}
```

A.3.3 ScrollPaneChangeListener

```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
```

```
public class ScrollPaneChangeListener implements ChangeListener{

   private int lastHeight;

   public void stateChanged( ChangeEvent e ) {
      JViewport vp = ( JViewport )e.getSource();

      int h = vp.getViewSize().height;
        if ( h != lastHeight ) {
           lastHeight = h;
           int x = h - vp.getExtentSize().height;
           vp.setViewPosition( new Point( 0, x ) );
            }
     }
}
```

A.3.4 ChannelChoiceListener

```
import java.awt.event.*;

public class ChannelChoiceListener implements ActionListener
{
   ChannelChoiceFrame channelChoiceFrame;

   public ChannelChoiceListener(ChannelChoiceFrame channelChoiceFrame)
   {
      this.channelChoiceFrame = channelChoiceFrame;
   }

   public void actionPerformed(ActionEvent e) {

      if(e.getActionCommand().equals("join") ||
      e.getActionCommand().equals("txtChannel") )
       {
         if(!channelChoiceFrame.txtChannel.getText().equals("") &&
            channelChoiceFrame.txtChannel.getText().startsWith("#") ||
            !channelChoiceFrame.txtChannel.getText().equals("") &&
            channelChoiceFrame.txtChannel.getText().startsWith("+") )
          {

         OutgoingTraffic.sendDataToServer("JOIN "+channelChoiceFrame
                                              .txtChannel.getText());

          channelChoiceFrame.channelDialog.dispose();
          channelChoiceFrame.txtChannel.setText("");
        }
      else
      {
       InternalPanelManager.getTextArea("#status")
        .append(channelChoiceFrame.txtChannel.getText()+": bad channel \n");

         channelChoiceFrame.channelDialog.dispose();
         channelChoiceFrame.txtChannel.setText("");

       }
      }
   }
}
```

A.3.5 UserListMouseListener

```
import javax.swing.*;
import java.awt.event.*;

public class UserListMouseListener extends MouseAdapter
{
  private String supernick = "spitfrog";
  private JList userList;

  public UserListMouseListener(JList userList)
  {
```

```
    this.userList = userList;
   }

  public void mouseClicked(MouseEvent e)
  {
    if (e.getClickCount() == 2 && e.getButton() == MouseEvent.BUTTON1)
     {
       int index = userList.locationToIndex(e.getPoint());
       String user = userList.getModel().getElementAt(index).toString();

       if(user.startsWith("@") || user.startsWith("+"))
          user = user.substring(1);

       if(!InternalPanelManager.containsHashtableTheKey("querypanel",user)
          && !user.equals(supernick) && !user.equals(""))
        {
          CreateInternalQueryPanel.createQueryPanel(user);
        }
      }
   }
}
```

A.4 Gui

A.4.1 GuiFrame

```
import java.awt.*;
import javax.swing.*;

public class GuiFrame extends JFrame
{
  private JMenuBar menubar = new JMenuBar();
  private JMenu operationsmenu = new JMenu();
  private JMenu fluffmenu = new JMenu();
  private JMenuItem connect = new JMenuItem();
  private JMenuItem disconnect = new JMenuItem();
  private JMenuItem join = new JMenuItem();
  private JMenuItem part = new JMenuItem();
  private JMenuItem exit = new JMenuItem();
  private JMenuItem crawl = new JMenuItem();
  private JMenuItem nocrawl = new JMenuItem();
  private JMenuItem about = new JMenuItem();

  private GuiFrameListener guiFrameListener = new GuiFrameListener();

  public GuiFrame(String title)
  {
    setTitle(title);

    new CreateInternalStatusPanel();

    operationsmenu.setText("operations");
    fluffmenu.setText("fluff");

    connect.setText("connect");
    connect.setActionCommand("connect");
    disconnect.setText("disconnect");
    disconnect.setActionCommand("disconnect");
    join.setText("join channel");
    join.setActionCommand("channel");
    part.setText("part channel");
    part.setActionCommand("partchan");
    exit.setText("exit");
    exit.setActionCommand("exit");
    about.setText("about");
    about.setActionCommand("about");
    crawl.setText("start crawler");
    crawl.setActionCommand("crawl");
    nocrawl.setText("stop crawler");
    nocrawl.setActionCommand("nocrawl");

    operationsmenu.add(connect);
    operationsmenu.add(disconnect);
    operationsmenu.addSeparator();
```

```
    operationsmenu.add(join);
    operationsmenu.add(part);
    operationsmenu.addSeparator();
    operationsmenu.add(exit);

    fluffmenu.add(crawl);
    fluffmenu.add(nocrawl);
    fluffmenu.addSeparator();
    fluffmenu.add(about);

    menubar.add(operationsmenu);
    menubar.add(fluffmenu);

    /**operationsmenu actions*/
    connect.addActionListener(guiFrameListener);
    disconnect.addActionListener(guiFrameListener);
    join.addActionListener(guiFrameListener);
    part.addActionListener(guiFrameListener);
    exit.addActionListener(guiFrameListener);

    /**fluffmenu actions*/
    crawl.addActionListener(guiFrameListener);
    nocrawl.addActionListener(guiFrameListener);
    about.addActionListener(guiFrameListener);

    /**add menubar*/
    setJMenuBar(menubar);

    /**add tabbedpane to frame*/
    getContentPane().add(TabbedPane.tabpane, BorderLayout.CENTER);
  }

}
```

A.4.2 InternalPanelManager

```
import java.util.*;
import javax.swing.*;

public class InternalPanelManager
{
  private static Hashtable panels = new Hashtable();
  private static Hashtable textareas  = new Hashtable();
  private static Hashtable scrollpanes = new Hashtable();
  private static Hashtable textfields = new Hashtable();
  private static Hashtable defaultlistmodels = new Hashtable();
  private static Hashtable userjlists  = new Hashtable();
  private static Hashtable querypanels = new Hashtable();

  private static InternalPanelComponentListener
                internalPanelComponentListener =
                new InternalPanelComponentListener();

  public static JPanel getInternalPanel(String key)
  {
   return ((JPanel)panels.get(key));
  }

  public static JTextArea getTextArea(String key)
  {
     return ((JTextArea)textareas.get(key));
  }

  public static JScrollPane getScrollPane(String key)
  {
   return ((JScrollPane)scrollpanes.get(key));
  }

  public static JTextField getTextField(String key)
  {
   return ((JTextField)textfields.get(key));
  }

  public static DefaultListModel getUserList(String key)
  {
   return ((DefaultListModel)defaultlistmodels.get(key));
  }
```

```
public static JList getUserJList(String key)
{
 return ((JList)userjlists.get(key));
}

public static JPanel getQueryPanel(String key)
{
 return ((JPanel)querypanels.get(key));
}

public static void putInternalPanel(String key, JPanel iPanel)
{
   panels.put(key, iPanel);
}

public static void putTextArea(String key, JTextArea textArea)
{
   textareas.put(key, textArea);
}

public static void putScrollPane(String key, JScrollPane scrollPane)
{
   scrollpanes.put(key, scrollPane);
}

public static void putTextField(String key, JTextField textField)
{
   textfields.put(key, textField);
}

public static void putUserList(String key, DefaultListModel defaultListModel)
{
   defaultlistmodels.put(key, defaultListModel);
}

public static void putUserJList(String key, JList UserJList)
{
   userjlists.put(key, UserJList);
}

public static void putQueryPanel(String key, JPanel queryPanel)
{
   querypanels.put(key, queryPanel);
}


public static boolean containsHashtableTheKey(String hashTable, String key)
{
 if(hashTable.equals("panel"))
   {
    return panels.containsKey(key);
  }

  else
  if(hashTable.equals("textarea"))
   {
    return textareas.containsKey(key);
  }

  else
  if(hashTable.equals("scrollpane"))
   {
    return scrollpanes.containsKey(key);
  }

  else
  if(hashTable.equals("textfield"))
   {
    return textfields.containsKey(key);
   }

  else
  if(hashTable.equals("defaultlistmodel"))
   {
     return defaultlistmodels.containsKey(key);
  }

  else
  if(hashTable.equals("userjlist"))
   {
     return userjlists.containsKey(key);
  }

  else
```

```
    if(hashTable.equals("querypanel"))
     {
       return querypanels.containsKey(key);
     }

    return false;
  }

  public static Hashtable getTheInternalPanels()
  {
    return panels;
  }

  public static Hashtable getTheTextAreas()
  {
    return textareas;
  }

  public static Hashtable getTheScrollPanes()
  {
    return scrollpanes;
  }

  public static Hashtable getTheTextFields()
  {
    return textfields;
  }

  public static Hashtable getTheUserList()
  {
    return defaultlistmodels;
  }

  public static Hashtable getTheUserJLists()
  {
    return userjlists;
  }

  public static Hashtable getTheQueryPanels()
  {
    return querypanels;
  }

  public static InternalPanelComponentListener
                getInternalPanelComponentListener()
  {
    return internalPanelComponentListener;
  }
}
```

A.4.3 CreateInternalStatusPanel

```
import java.awt.*;
import javax.swing.*;

public class CreateInternalStatusPanel extends JPanel{

   public CreateInternalStatusPanel()
   {

    //Titel
    String tabname = "status";

    //iPanel initial
    InternalPanelManager.putInternalPanel("#status", new JPanel(new BorderLayout()));

    //TextArea
    InternalPanelManager.putTextArea("#status",new JTextArea());
    InternalPanelManager.getTextArea("#status").setLineWrap(true);
    InternalPanelManager.getTextArea("#status").setWrapStyleWord(true);
    InternalPanelManager.getTextArea("#status").setEditable(false);

    //JScrollPane scrollPaneTextArea
    InternalPanelManager.putScrollPane("#status", new JScrollPane());
    InternalPanelManager.getScrollPane("#status")
    .setHorizontalScrollBarPolicy(InternalPanelManager
    .getScrollPane("#status").HORIZONTAL_SCROLLBAR_NEVER);
```

```
    InternalPanelManager.getScrollPane("#status").getViewport()
    .add(InternalPanelManager.getTextArea("#status"), null);
    InternalPanelManager.getScrollPane("#status").getViewport()
    .addChangeListener(new ScrollPaneChangeListener());

    //TextField
    InternalPanelManager.putTextField("#status", new JTextField());
    InternalPanelManager.getTextField("#status")
    .setActionCommand("channel:#status");
    InternalPanelManager.getTextField("#status")
    .addActionListener(InternalPanelManager.getInternalPanelComponentListener());

    //addComponents to the iPanel
    InternalPanelManager.getInternalPanel("#status")
    .add(InternalPanelManager.getScrollPane("#status"), BorderLayout.CENTER);
    InternalPanelManager.getInternalPanel("#status")
    .add(InternalPanelManager.getTextField("#status"),  BorderLayout.SOUTH);

    //add the status panel to tabbedpane
    TabbedPane.tabpane.addTab(tabname, InternalPanelManager.getInternalPanel("#status"));

   }

}
```

A.4.4 CreateInternalChannelPanel

```
import java.awt.*;
import javax.swing.*;

public class CreateInternalChannelPanel
{
  public static void CreateChannelPanel(String channel)
  {
    String tabname = channel;

    JScrollPane scrollPaneUserList = new JScrollPane();

    InternalPanelManager.putInternalPanel(channel, new JPanel(new BorderLayout()));

    /**textarea*/
    InternalPanelManager.putTextArea(channel, new JTextArea());
    InternalPanelManager.getTextArea(channel).setLineWrap(true);
    InternalPanelManager.getTextArea(channel).setWrapStyleWord(true);
    InternalPanelManager.getTextArea(channel).setEditable(false);

    /**jscrollpane*/
    InternalPanelManager.putScrollPane(channel, new JScrollPane());
    InternalPanelManager.getScrollPane(channel)
    .setHorizontalScrollBarPolicy(InternalPanelManager.getScrollPane(channel)
    .HORIZONTAL_SCROLLBAR_NEVER);

    InternalPanelManager.getScrollPane(channel).getViewport()
    .add(InternalPanelManager.getTextArea(channel), null);
    InternalPanelManager.getScrollPane(channel).getViewport()
    .addChangeListener(new ScrollPaneChangeListener());

    /**textfield*/
    InternalPanelManager.putTextField(channel, new JTextField());
    InternalPanelManager.getTextField(channel)
    .setActionCommand("channel:"+ channel);
    InternalPanelManager.getTextField(channel)
    .addActionListener(InternalPanelManager.getInternalPanelComponentListener());

    /**userlist defaultlistmodel*/
    InternalPanelManager.putUserList(channel, new DefaultListModel());

    /**Userjlists*/
    InternalPanelManager.putUserJList(channel,
    new JList(InternalPanelManager.getUserList(channel)));
    InternalPanelManager.getUserJList(channel)
    .addMouseListener(new UserListMouseListener(InternalPanelManager
    .getUserJList(channel)));

    /**jscrollpane for the nicklist*/
    scrollPaneUserList.setHorizontalScrollBarPolicy(scrollPaneUserList
    .HORIZONTAL_SCROLLBAR_NEVER);
    scrollPaneUserList.getViewport().add(InternalPanelManager
```

```
    .getUserJList(channel), null);


    InternalPanelManager.getInternalPanel(channel)
    .add(InternalPanelManager.getScrollPane(channel), BorderLayout.CENTER);
    InternalPanelManager.getInternalPanel(channel)
    .add(scrollPaneUserList, BorderLayout.EAST);
    InternalPanelManager.getInternalPanel(channel)
    .add(InternalPanelManager.getTextField(channel), BorderLayout.SOUTH );

    /**add channel panel to tabbedpane*/
    TabbedPane.tabpane.addTab(tabname, InternalPanelManager.getInternalPanel(channel));
  }
}
```

A.4.5 CreateInternalQueryPanel

```
import java.awt.*;
import javax.swing.*;

public class CreateInternalQueryPanel
{

  public static void createQueryPanel(String nick)
  {

    String tabname = nick;

    InternalPanelManager.putQueryPanel(nick, new JPanel(new BorderLayout()));

    /**textarea*/
    InternalPanelManager.putTextArea(nick, new JTextArea());
    InternalPanelManager.getTextArea(nick).setLineWrap(true);
    InternalPanelManager.getTextArea(nick).setWrapStyleWord(true);
    InternalPanelManager.getTextArea(nick).setEditable(false);

    /**jscrollpane*/
    InternalPanelManager.putScrollPane(nick, new JScrollPane());
    InternalPanelManager.getScrollPane(nick)
    .setHorizontalScrollBarPolicy(InternalPanelManager.getScrollPane(nick)
    .HORIZONTAL_SCROLLBAR_NEVER);

    InternalPanelManager.getScrollPane(nick).getViewport()
    .add(InternalPanelManager.getTextArea(nick), null);
    InternalPanelManager.getScrollPane(nick).getViewport()
    .addChangeListener(new ScrollPaneChangeListener());

    /**textfield*/
    InternalPanelManager.putTextField(nick, new JTextField());
    InternalPanelManager.getTextField(nick).setActionCommand("query:"+ nick);
    InternalPanelManager.getTextField(nick)
    .addActionListener(InternalPanelManager.getInternalPanelComponentListener());

    InternalPanelManager.getQueryPanel(nick)
    .add(InternalPanelManager.getScrollPane(nick), BorderLayout.CENTER );
    InternalPanelManager.getQueryPanel(nick)
    .add(InternalPanelManager.getTextField(nick), BorderLayout.SOUTH );

    /**add query panel to tabbedpane*/
    TabbedPane.tabpane.addTab(tabname, InternalPanelManager.getQueryPanel(nick));

   }
}
```

A.4.6 TabbedPane

```
import javax.swing.*;

public class TabbedPane
{
  public static JTabbedPane tabpane = new JTabbedPane();
```

```
  public static void addTabToPane(String tabname, JPanel panel)
  {
    TabbedPane.tabpane.addTab(tabname, panel);
  }

  public static void removeTabFromPane()
  {
    String channel;
    String nick;

    if(tabpane.getModel().isSelected())
     {
       int index = tabpane.getSelectedIndex();

       channel = TabbedPane.tabpane.getTitleAt(index);

       if(channel.startsWith("#") || channel.startsWith("+"))
       {

       TabbedPane.tabpane.removeTabAt(index);
       TabbedPane.tabpane.revalidate();
       TabbedPane.tabpane.repaint();

       OutgoingTraffic.sendDataToServer("PART " + channel);

       InternalPanelManager.getTheInternalPanels().remove(channel);
       InternalPanelManager.getTheTextAreas().remove(channel);
       InternalPanelManager.getTheScrollPanes().remove(channel);
       InternalPanelManager.getTheTextFields().remove(channel);
       InternalPanelManager.getTheUserJLists().remove(channel);
       InternalPanelManager.getTheUserList().remove(channel);
       }

       else
       {
         nick = TabbedPane.tabpane.getTitleAt(index);

         TabbedPane.tabpane.removeTabAt(index);
         TabbedPane.tabpane.revalidate();
         TabbedPane.tabpane.repaint();

         InternalPanelManager.getTheQueryPanels().remove(nick);
         InternalPanelManager.getTheScrollPanes().remove(nick);
         InternalPanelManager.getTheTextAreas().remove(nick);
         InternalPanelManager.getTheTextFields().remove(nick);
       }
     }
   }
}
```

A.4.7 ChannelChoiceFrame

```
import java.awt.*;
import javax.swing.*;

public class ChannelChoiceFrame
{

   public JDialog channelDialog;

   public JTextField txtChannel;

   private JButton btnJoin;

   public ChannelChoiceFrame(GuiFrame guiFrame)
   {
     channelDialog = new JDialog(guiFrame,"Channel",true);
     ChannelChoiceListener channelChoiceListener =
     new ChannelChoiceListener(this);

     txtChannel = new JTextField(15);
     btnJoin = new JButton("Join");

     txtChannel.setActionCommand("txtChannel");
     txtChannel.addActionListener(channelChoiceListener);

     btnJoin.setActionCommand("join");
     btnJoin.addActionListener(channelChoiceListener);
```

```
        channelDialog.getContentPane().setLayout(new FlowLayout(FlowLayout.LEFT));
        channelDialog.getContentPane().add(txtChannel);
        channelDialog.getContentPane().add(btnJoin);

        channelDialog.pack();

        Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
        channelDialog.setLocation( (d.width - channelDialog
        .getSize().width) / 2,(d.height - channelDialog.getSize().height) / 2);

        channelDialog.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

        channelDialog.setResizable(false);
        channelDialog.setVisible( true );
    }
}
```

Bibliografi

[ACT] ActiveBuddy. Se http://www.activebuddy.com.

[AFS01] James Allen, George Fergusen og Amanda Stent. An Architecture For More Realistic Conversational Systems. I *Proceedings of international conference on Intelligent user interfaces*, side 1–8, januar 2001.

[AIM] America Online Instant Messenger. Se http://www.aim.com.

[ANT] Ant. Se http://ant.apache.org/.

[BC92] Nicholas J. Belkin og W. Bruce Croft. Information Filtering and Information Retrieval: Two Sides of the Same Coin? *Comunication of the ACM*, 35:29–38, 1992.

[BIT] BitlBee. Se http://www.bitlbee.org.

[BUD] BuddyScript. Se http://www.buddyscript.com.

[BYRN99] Ricardo Baeza-Yates og Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[CON] Fast Data Search Configuration Guide. Fast Search and Transfer ASA.

[DLM99] Neil W. Van Dyke, Henry Lieberman og Pattie Maes. Butterfly: A Conversation-Finding Agent for Internet Relay Chat. I *Proceedings of international conference on Intelligent user interfaces*, side 39–41, 1999.

[Don02] Judith Donath. A Semantic Approach To Visualizing Online Conversations. *Association for Computing Machinery*, 45(4):45–49, april 2002.

[EGG] Eggdrop. Se http://www.eggheads.org.

[FAS] Fast Search & Transfer. Se http://www.fast.no.

[FBY92] W.B Frakes og R. Baeza-Yates. *Information Retrieval: Data Structures & Algorithms*. Prentice Hall, 1992.

[FCM+00] Shelly Farnham, Harry R. Chesley, Debbie E. McGhee, Reena Kawal og Jennifer Landau. Structured Online Interactions: Improving the Descision-Making of Small Discussion Groups. I *Proceedings of conference on Computer Supported Cooperative Work*, side 299–308, desember 2000.

[Hav02] Taher H. Haveliwala. Search Facilities for Internet Relay Chat. I *Proceedings of Joint Conference on Digital Libraries*, side 395, juli 2002.

[ICQ] ICQ. Se http://web.icq.com.

[INT] Fast Data Search Integration Guide. Fast Search and Transfer ASA.

[IRC] IRChelp. Se http://www.irchelp.org.

[JAB] Jabber. Se http://www.jabber.org.

[JBO] JBoss Application Server. Se http://www.jboss.org/.

[Lie95] Henry Lierberman. Letizia: An Agent That Assists Web Browsing. I *Proceedings of International Joint Conference on Artificial Intelligence*, side 924–929, 1995.

[Mae94] Pattie Maes. Agents that Reduce Work and Information Overload. *Communications of the ACM*, 37(7), juli 1994.

[MED] MEDLINE. Se http://www.nlm.nih.gov/pubs/factsheets/medline.html.

[Mor] Mort Bay Consulting Pty. Ltd.(Australia). Jetty Web Server & Servlet Container. Se http://jetty.mortbay.org/.

[Mou96] Alexandros Moukas. Amalthea: Information Discovery and Filtering using a Multiagent Evolving Ecosystem. *Proceedings of the Conference on Practical Application of Intelligent Agents & Multi-Agent Technology, London*, 1996.

[MSN] MSN Messenger. Se http://messenger.msn.com.

[Nea97] Lisa Neal. Virtual Classrooms and Communities. I *Proceedings of International Conference on Supporting Group Work*, side 81–90, november 1997.

[OR93] Jarkko Oikarinen og Darren Reed. Internet Relay Chat Protocol, 1993.

[Rij79] C.J Van Rijsbergen. *Information Retrieval, 2nd edition*. Dept. of Computer Science, University of Glasgow, 1979.

[RJ76] S.E Robertson og K. Spark Jones. Relevance weigthing of search terms. *Journal of the American Society for Information Sciences*, 27(3):129–146, 1976.

[RJS02] Ammon Ribak, Michal Jacovi og Vladimir Soroka. "Ask Before You Search" Peer Support and Community Building with ReachOut. I *Proceedings of conference on Computer Supported Cooperative Work*, side 126–135, november 2002.

[SFD00] Marc A. Smith, Shelly D. Farnham og Steven M. Drucker. The Social Life of Small Graphical Chat Spaces. I *Proceedings of conference on Human factors in computing systems (CHI)*, april 2000.

[SMA] SmarterChild. Se http://www.smarterchild.com.

[Suna] Sun Microsystems Inc. Guidelines, patterns and code for end-to-end Java applications. Se http://java.sun.com/blueprints/guidelines/.

[Sunb] Sun Microsystems Inc. Java 2 Platform Enterprise Edition(J2EE). Se http://java.sun.com/j2ee/.

[Sunc] Sun Microsystems Inc. Java Servlet Technology. Se http://java.sun.com/products/servlet/.

[Sund] Sun Microsystems Inc. JavaServer Pages Standard Tag Library. Se http://java.sun.com/products/jsp/jstl/.

[SYS] Fast Data Search System Reference Guide. Fast Search and Transfer ASA.

[TRE] TREC(Text Retrieval Conference). Se http://trec.nist.gov/.

[Uni] University of Southampton. AOL ICQ vs MSN Messenger. Se http://citeseer.ist.psu.edu/555217.html.

[VD99] Fernanda B. Viegas og Judith S. Donath. Chat Circles. I *Proceedings of conference on Human factors in computing systems (CHI)*, side 9–16, 1999.

[VSD99] David Vronay, Marc Smith og Steven Drucker. Alternative Interfaces for Chat. I *Proceedings of annual ACM symposium on User interface software and technology*, side 19–26, november 1999.

Kolofon

Dokumentet har i hovedsak blitt skrevet på en Dell Latitude C640 portabel datamaskin med operativsystemet Debian GNU/Linux hvor Vim har blitt brukt som tekstbehandler. Dokumentet har blitt typesatt med dokumentspråket LaTeX og satt til 11pt Lucida Bright for A4 ark.