UNIVERSITETET I OSLO
Institutt for informatikk

Peer-to-peer-Telefoni over Internett

Masteroppgave

(60 studiepoeng)

Fredrik Oterholt

3. august 2011

Peer-to-peer-Telefoni over Internett

Masteroppgave

(60 studiepoeng)

Fredrik Oterholt

3. august 2011

Takk til

Jeg vil først og fremst takke min veileder Knut Omang som har lest og kommet med nyttige kommentarer gjennom hele arbeidsprosessen med denne oppgaven. Din tilgjengelighet og tilbakemelding setter jeg stor pris på.

Jeg vil også takke spesielt Frank Oterholt, Jan G. Haug, Maja Haug og Jonas A. Arneberg for kommentarer og korrekturlesning underveis.

Ikke minst vil jeg takke min familie og mine venner for all den støtte og oppmuntring jeg har fått. Takk for at dere har vært så tålmodige med meg gjennom hele studietiden.

Fredrik Oterholt,
3. august 2011

Innhold

Innhold **v**
Figurer . . . viii
Tabeller . . . ix
Listings . . . ix

1 Introduksjon **1**
1.1 Bakgrunn . . . 1
1.2 Problemstilling . . . 2
1.3 Oppgavens oppbygning . . . 2

2 Telefoni over Internett **5**
2.1 Voice over IP (VoIP) . . . 6
2.1.1 Fordeler med VoIP . . . 7
2.1.2 Ulemper med VoIP . . . 9
Forsinkelser . . . 9
«Jitter» . . . 9
Pakketap . . . 10
2.2 Implementasjoner av VoIP . . . 10
2.2.1 Session Initiation Protocol (SIP) . . . 10
Enheters oppgaver i SIP . . . 12
Eksempel på samtaler . . . 15
SIP-adresser . . . 15
Session Description Protocol (SDP) . . . 17
Real-time Transport Protocol (RTP) . . . 18

2.2.2 H.323 19
2.2.3 Skype 19
2.3 Oppsummering 21

3 Peer-to-Peer (P2P) 23
3.1 Hvorfor Peer-to-Peer? 23
3.1.1 Fordeler med P2P-teknologi 25
Fordeling av arbeidslast 25
Delt ansvar 25
Skalerbarhet 26
3.1.2 Problemer med P2P-teknologi 27
Innholdet er spredt 27
Noder som blir utilgjengelige 27
Privat og sensitiv informasjon 28
Store mengder informasjonsflyt 28
NAT- og brannmur-problematikk 29
3.2 Arkitektur og topologi 31
3.2.1 Ustrukturerte nettverk 32
Desentraliserte/Rent distribuerte nettverk 32
Sentraliserte P2P-nettverk 35
Hybride nettverk 37
3.2.2 Strukturerte nettverk 39
Hvilke problemer ønsker vi å løse ved å bruke strukturerte nettverk? 39
Organisering av strukturerte P2P-nettverk 40
Distribuert Hash-tabell (DHT) 40
3.3 Filoverføring som et bruksområde 42
3.3.1 Napster 42
Popularitet 43
3.3.2 Gnutella 43
Søk 43
3.3.3 Freenet 44
Hvordan lages nettverket 45

Publisering og konsumering 46
3.3.4 FastTrack 47
Hvordan nettverket fungerte 48
Svakhet i hash-algoritmen 49
3.3.5 BitTorrent 50
Hvordan fungerer BitTorrent 50
Hvordan dannes nettverket 52
Svakheter 53
3.4 Implementasjon av strukturerte P2P-nettverk 54
3.4.1 Chord 54
3.5 Oppsummering 58

4 Tidligere arbeid **59**
4.1 REsource LOcation And Discovery (RELOAD) 59
4.1.1 P2PSIP-overlay 60
4.1.2 Navneoppslag 60
4.1.3 Opprette samtaler i RELOAD 61
4.1.4 Robusthet 62
4.1.5 Skalerbarhet 63
4.2 Oppsummering 63

5 Implementasjon **65**
5.1 Formålet med prototypen 66
5.2 Applikasjoner og rammeverk 67
5.2.1 VoIP-klienter 67
PhonerLite 67
Ekiga 68
5.2.2 39 Peers 69
Standarder i 39 Peers 69
Applikasjoner i 39 Peers 70
5.2.3 OpenDHT 71
DHT-Gateway (GW) 72
5.3 Prototypen 73

5.3.1 Location Service ved hjelp av en DHT 73
dhtls.py 74
5.3.2 Kommunikasjon med DHT 78
opendht.py 78
5.3.3 Finne den optimale DHT-Gateway 82
findgateway.py 82
5.4 Resultater 87
5.4.1 Hvordan fungerte dette i praksis? 87
Forberedelser 87
Samtalen 88
5.4.2 Evaluering 90
Er prototypen skalerbar? 91
Er prototypen robust? 92
Plattform og SIP-klienter 93
5.5 Oppsummering 94

6 Konklusjon **95**
6.1 Videre arbeid 97

Bibliografi **99**

A Kildekode **107**
A.1 Distribuert Hash-tabell Location Service 107
A.2 OpenDHT 108
A.3 Find Gateway 112

B Forkortelser **117**

Figurer

2.1 Samtale i VoIP mellom sender og mottaker(e) 6
2.2 Enheter som kan ringe hverandre ved hjelp av VoIP 8
2.3 `REGISTER` 13

2.4 Eksempel på samtaler 16

3.1 De to nettverksmodellene 24
3.2 En node er bak Network Address Translation (NAT)/brannmur 29
3.3 Begge noder bak en NAT/brannmur 30
3.4 Strukturerte og ustrukturerte P2P-nettverk 32
3.5 Ustrukturerte P2P-nettverk. 33
3.6 Søk i desentraliserte P2P-nettverk. 34
3.7 P2P-nettverk med sentral enhet 36
3.8 Topologien i FastTrack 49
3.9 Chord fingertabell 55
3.10 Chord lookup 57

5.1 Samtale mellom Alice og Bob med Peer-to-Peer Session Initiation Protocol (P2PSIP) 66
5.2 PhonerLite: Configuration 89
5.3 Alice og Bob ringer ved hjelp av P2PSIP 91

Tabeller

2.1 SIP-forespørsler 11
2.2 SIP-reponser 12

5.1 Spesifikasjoner på SIP-klientene «PhonerLite» og «Ekiga» . 68

Listings

2.1 Eksempel på `INVITE` med SDP 17
5.1 Alice sender `REGISTER` til SIP-server 75
5.2 Alice sin SIP-URI blir lagret i DHT-en sammen med nettverksadressen(e) 76
5.3 LS gjør navneoppslag, og får tilbake verdier fra DHT-en . . . 77

5.4 Eksempel på en DHT-test 81
5.5 FindGateway tester 10 DHT-GW-er 85
5.6 INVITE fra Alice til Bob 88
A.1 Kildekode: dhtls.py 107
A.2 Kildekode: opendht.py 108
A.3 Kildekode: findgateway.py 112

Kapittel 1

Introduksjon

1.1 Bakgrunn

Telefoni har eksistert i over hundre år, og har blitt brukt av folk som et viktig redskap for å kommunisere med andre over både korte og lange avstander. Teknologien som muliggjør telefoni har vært i en konstant utvikling siden starten, og i nyere tid har Voice over IP (VoIP), telefoni over internett, blitt et populært alternativ til tradisjonell telefoni grunnet pris og fleksibilitet.

Signaleringsprotokollen Session Initiation Protocol (SIP) [1] er en veletablert standard for VoIP, men det finnes også aktører i VoIP-industrien som ikke følger denne standarden. I folks dagligtale er telefoni over internett ofte synonymt med tjenesten «Skype» [2]. «Skype» har har klart å oppnå stor popularitet, og tilbyr en velfungerende VoIP-tjeneste som lar folk kommunisere med hverandre med både lyd, video og lynmeldinger. Mye av styrken i «Skype» ligger i at den er veldig lett å bruke, og at den er robust og skalerbar.

«Skype» har sitt opphav fra fildelingstjenesten «KaZaa» [3], og er i stor grad basert på Peer-to-Peer (P2P). Jeg vil derfor i denne oppgaven presentere hvordan vi kan kombinere signaleringsprotokollen SIP og P2P

for å oppnå en robusthet og skalerbarhet slik som «Skype». Dette vil jeg vise ved å først gi en innføring i hvordan telefoni over internett fungerer, og deretter en innføring i P2P. Jeg vil også vise til den pågående strandardiseringen av en ny signaleringsprotokoll som kombinerer P2P og SIP [4]. I oppgaven vil jeg også presentere en prototype som kombinerer SIP og P2P.

1.2 Problemstilling

I denne oppgaven vurderer jeg følgende problemstilling:

Hvordan kan SIP kombineres med P2P for å gjøre SIP mer robust og skalerbar?

I sammenheng med oppgavens problemstilling besvarer også følgende underspørsmål:

- Hvilke tanker og ideer kan man ta med seg fra de allerede eksisterende P2P-tjenester over til telefoni over internett?
- Hvordan P2P-nettverk kan egne seg til telefoni over internett?
- Hvor langt på vei er industrien med å innføre distribuerte VoIP-tjenester?

1.3 Oppgavens oppbygning

Oppgaven er strukturert på følgende måte:

Kapittel 2 introduserer og forklarer telefoni over internett med vekt på signaleringsprotokollen Session Initiation Protocol. Jeg vil også belyse fordeler ved å gå over fra tradisjonell fasttelefoni til telefoni over internett.

Kapittel 3 I dette kapittelet introduseres begrepet Peer-to-Peer (P2P), og en oversikt over hvilke fordeler og ulemper P2P har. Videre presenterer jeg hvilke arkitekturer og topologier som finnes innenfor P2P, og kategorisering av disse. Jeg forklarer også hvordan P2P fungerer, og hvilke bruksområder det har hatt. Fildeling er et viktig bruksområde for P2P, og mange ideer kan overføres fra dette bruksområdet og over til telefoni over internett.

Kapittel 4 I dette kapittelet presenterer jeg teori som er relatert til kombinasjonen telefoni over internett og P2P ved å vise til arbeid som er gjort i forbindelse med en standardidseringsprosess av en ny signaliseringsprotokoll som kombinerer P2P og SIP.

Kapittel 5 I dette kapittelet presenterer jeg en prototype som et «proof of concept», av en implementasjon som tar i bruk SIP, og kombinerer dette med P2P. Kapittelet går detaljert igjennom prototypens formål og funksjoner, og avsluttes med en evaluering av prototypen.

Kapittel 6 Avslutningsvis i oppgaven presenterer jeg mine funn og erfaringer. Med en kritisk blikk oppsummerer jeg om har besvart spørsmålene i problemstillingen (se 1.2 på forrige side). Til slutt foreslår jeg ideer for videre arbeid med prototypen.

Kapittel 2

Telefoni over Internett

Telefoni har vært et viktig redskap for kommunikasjon mellom folk i over hundre år. Helt siden år 1878 har telefonsamtaler foregått over tradisjonelle telefoni-nettverk, kalt Public Switched Telephone Network (PSTN). Telefoni ble oppfunnet før den tid, men det var i 1878 [5, s. 89] de første samtalene uten en direkte linje mellom telefonene ble foretatt. Forbindelsen mellom telefonene ble manuelt «switchet» av telefoni-operatører.

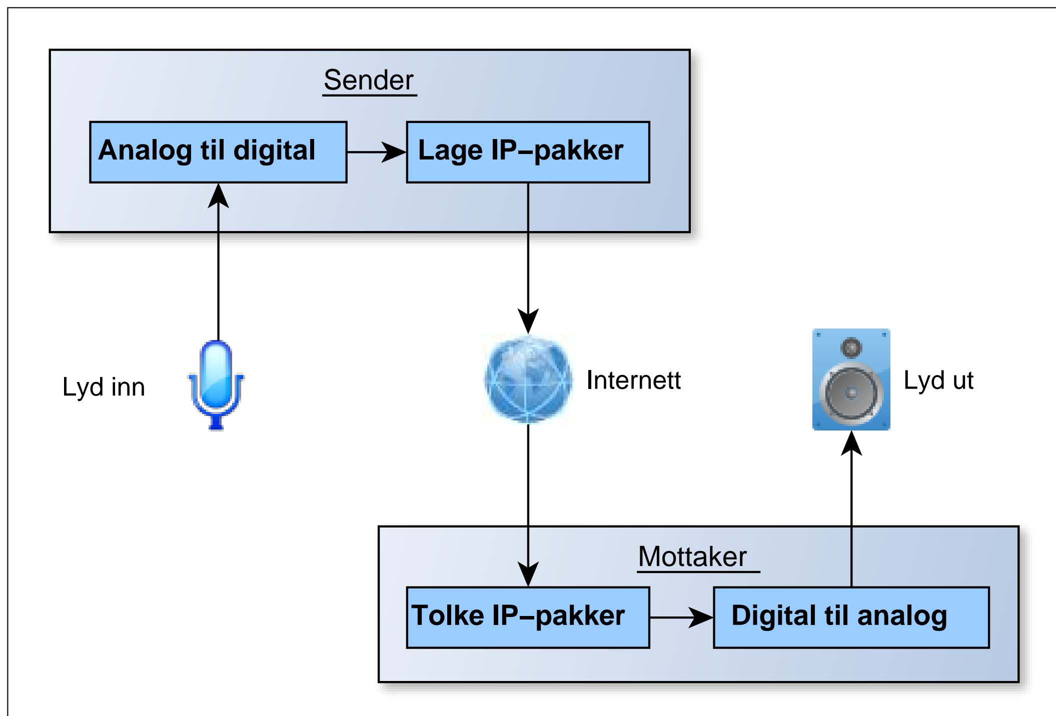
Siden den tid har teknologien forandret seg en god del, og oppgaver som måtte gjøres manuelt av telefoni-operatører gjøres nå av automatiske Private Branch Exchange (PBX)-er (telefonisentraler). Telefoni gjøres ikke lenger bare over PSTN. Telefoni gjøres nå også over internett ved hjelp av VoIP.

Videre i kapittelet presenterer jeg hva VoIP er, hvordan det brukes, og hvordan det fungerer. Vi skal også se på hvilke protokoller som benyttes for å gjennomføre samtaler ved bruk av VoIP.

2.1 Voice over IP (VoIP)

Som navnet Voice over IP (VoIP) tilsier, så er VoIP en metode for å sende telefonsamtaler over Internet Protocol (IP) fremfor å benytte seg av tradisjonell telefoni over PSTN. VoIP er også kjent under navn som «IP-telefoni», «internett-telefoni» og «telefoni over internett».

Enkelt fortalt er VoIP en måte å konvertere lyd og bilde over til et digitalt format, sende dette til en eller flere mottakere over internett, og spille av innholdet, som ble sendt, hos mottakeren(e). I figur 2.1 er dette illustrert. Her produserer senderen lyd ved å snakke inn i en mikrofon. Den analoge lyden blir så digitalisert, delt opp, og sendt som IP-pakker over internett. På mottakeren(e)s side blir IP-pakkene mottatt, lyden går fra digital til analog form, og blir spilt av på mottakerens høytalere. Stegene som er illustrert i figuren blir gjentatt igjen og igjen, frem og tilbake, så lenge samtalen pågår mellom sender og mottaker(e).



Figur 2.1: Samtale i VoIP mellom sender og mottaker(e)

Det er ikke bare samtaler med lyd og bilde som er mulig ved bruk av VoIP, men også konferansesamtaler med flere deltagere, lynmeldinger (Instant Messages (IM)), og andre typer multimedia-overføring.

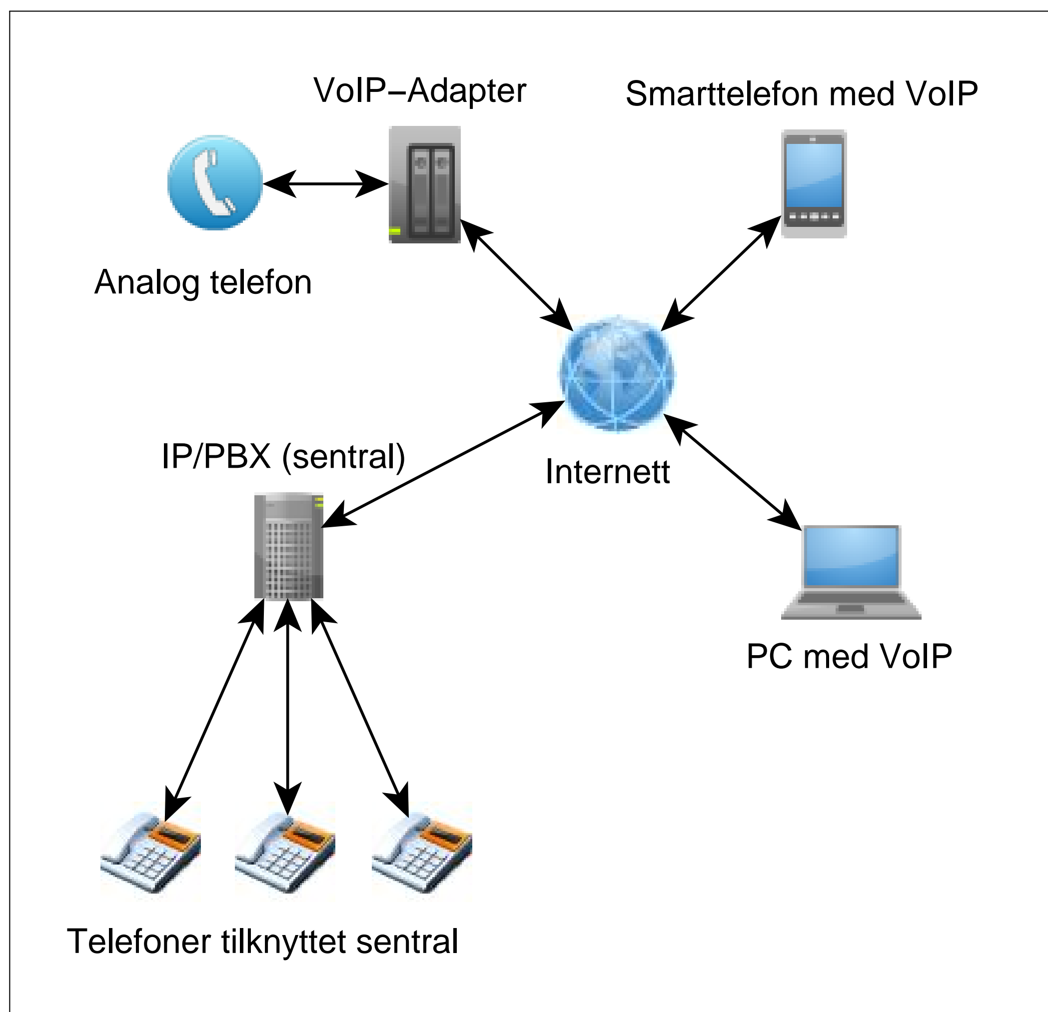
Samtaler med VoIP kan gjøres på flere måter. Disse punktene er illustrert i figur 2.2 på neste side:

- Ved bruk av adaptere som er direkte tilkoblet internett, med ett eller flere uttak hvor man kan koble til vanlige telefoner som tidligere var tilkoblet tradisjonelle telefoni-nettverk (PSTN).
- Ved bruk av applikasjoner på datamaskiner, smarttelefoner, Personal Digital Assistant (PDA)-er, eller andre enheter med tilgang til internett. En av de mest brukte applikasjonene for VoIP på datamaskiner er Skype [2] (se seksjon 2.2.3 på side 19).
- Det finnes også systemer som kombinerer VoIP og PSTN ved hjelp av en IP/PBX. Slike systemer gjør det mulig for VoIP-brukere å ringe til tradisjonelle telefoni-brukere, og at brukere av tradisjonell telefoni kan ringe til VoIP-brukere.

Det finnes forskjellige standarder/protokoller for hvordan denne kommunikasjonen mellom VoIP-klientene skal foregå, og disse skal vi se på i seksjon 2.2 på side 10.

2.1.1 Fordeler med VoIP

En av de største fordelene med VoIP er de økonomiske besparingene man kan gjøre med tanke på infrastruktur og bruk av tjenesten. Internet Service Provider (ISP)-er kan gjøre besparinger på infrastrukturen siden de ikke trenger å ha dedikerte linjer for telefoni, slik de trenger ved bruk av PSTN. Tilgang til et IP-nettverk, slik som internett, er selvfølgelig nødvendig for å benytte VoIP, men siden telefoni kan gå over dette nettverket er det ikke lenger noe behov for en linje dedikert til telefoni.



Figur 2.2: Enheter som kan ringe hverandre ved hjelp av VoIP

Kostnader for bruk av tjenestene i VoIP er også redusert i forhold til tradisjonell telefoni. Så lenge VoIP-samtalen kun går over internett, og ikke over PSTN-nettverk, vil det spesielt over lengre distanser være lønnsomt å benytte seg av VoIP fremfor tradisjonell telefoni. Tjenester slik som «hvem ringer», videresending og overføring av samtaler og konferanse-samtaler, er ofte betalingstjenester når man benytter tradisjonell telefoni. Disse tjenestene er ofte gratis i VoIP.

2.1.2 Ulemper med VoIP

Det finnes enkelte tekniske utfordringer ved bruk av VoIP. Disse utfordringene er ofte relatert til problemer med overføringen av data mellom sender og mottaker, og kan påvirke samtalekvaliteten. Vi skal i de neste avsnittene se på de tekniske utfordringene med forsinkelser, «jitter» og pakketap.

Forsinkelser

Forsinkelse av datapakker, som igjen vil føre til forsinkelser i lyden, er en av utfordringene. Undersøkelser viser at forsinkelser på lyden ikke bør overgå 200 millisekunder (ms) [5, s. 91], og at forsinkelser på over 270 ms er uakseptabelt. Forsinkelser kan gjøre at brukere ikke vet om samtalepartneren snakker, noe som lett kan skape avbrytelser.

Forsinkelser oppstår gjerne på grunn av overbelastede nettverk, annen type trafikk som går over det samme nettverket og distansen datapakkene må ta for å nå destinasjonen. Det er ikke bare nettverket som kan ha skyld i forsinkelse, men også prosessen hvor lyden konverteres fra analog til digital, eller fra digital til analog, slik det ble illustrert i figur 2.1 på side 6. Noe som ikke er illustrert i figuren, men som også kan forårsake forsinkelse, er kryptering og dekryptering.

«Jitter»

«Jitter» er et begrep som beskriver tidsvariasjoner i et signal, eller variasjon i forsinkelsen som forklart i forrige avsnitt. I VoIP kan dette oppstå siden datapakker ikke nødvendigvis bruker den samme tiden på å nå sitt mål hver gang. Dette fører igjen til at mottakeren kan oppleve hakking i lyden, eller perioder med stillhet [6, s. 13].

For å kompensere for «jitter» kan man på mottaker-siden spare opp et «buffer» som legger inn en liten forsinkelse på datapakker som kommer for tidlig, og kaster datapakker som kommer for sent.

Pakketap

Tap av datapakker kan også oppstå, og kan føre til at mottakeren vil oppleve stillhet eller brudd i samtalen. Lyden kan også oppfattes som om den er forvrengt.

Forskning viser at pakketap på over 5 % regnes som uakseptabelt for brukere av VoIP, og for å kunne oppnå den samme kvaliteten som i tradisjonell telefoni kan ikke pakketapet være på mer enn 1 % [6, s. 13].

2.2 Implementasjoner av VoIP

I likhet med tradisjonell telefoni opprettes det også en forbindelse mellom brukerne ved bruk av VoIP. For å opprette forbindelser mellom brukere benyttes det i VoIP egne protokoller som tar seg av disse oppgavene. Disse protokollene kalles signaleringsprotokoller.

De to signaleringsprotokollene SIP og H.323 [7] har vært de mest brukte og dominerende innen VoIP. I tillegg til disse er også applikasjonen Skype [2] mye brukt. Under denne seksjonen kommer jeg til å gi en forklaring på hva SIP er, og hvordan det fungerer. Jeg kommer også til å gi en kort presentasjon av H.323, og en presentasjon av Skype. Signaleringsprotokollen SIP er hovedfokuset for denne oppgaven, så presentasjonen av H.323 vil kun være en kort presentasjon av historisk karakter.

2.2.1 Session Initiation Protocol (SIP)

SIP er en signaleringsprotokoll som tilbyr avansert signalerings- og kontroll-funksjonalitet for en stor del multimedia-tjenester [8]. SIP er i stand til å opprette, modifisere og terminere sesjoner uavhengig av underliggende transportprotokoller [1, s. 8].

Forespørsel	Beskrivelse
INVITE	Invitasjon til sesjon. Sendes av User Agent Client (UAC) og mottas av User Agent Server (UAS). Kan også videresendes av en proxy.
REGISTER	Registering hos registrar (se seksjon 2.2.1 på side 14)
ACK	Bekrefter mottatt respons. UAC sender INVITE, UAS responderer med '200 OK', UAC sender da ACK for å vise at den har mottat '200 OK'.
BYE	Avslutter sesjon. Sendes når en av partene «legger på».
CANCEL	Avbryter oppringning. Brukes f.eks. hvis en INVITE tar for lang tid.
OPTIONS	En måte for User Agent (UA)-er å sjekke hvilke metoder, innholdstyper og forespørsler en annen UA eller en proxy støtter.

Tabell 2.1: SIP-forespørsler

SIP [1] ble først standardisert av Internet Engineering Task Force (IETF) i 1999 som Request for Comments (RFC) 2543. Standarden ble i 2002 erstattet av RFC 3261 [1], og det er denne jeg kommer til å fokusere på i oppgaven.

SIP befinner seg på applikasjonslaget, og overlater derfor transport av data til de underliggende nettverkslagene. Dataen som transporteres (lyd, video, IM) har gjene en «real-time»-karakteristikk, siden kommunikasjon og interaksjon foregår i sanntid. Overføringsprotokollen Real-time Transport Protocol (RTP) er derfor mye brukt i forbindelse med SIP (se seksjon 2.2.1 på side 18).

I likhet med Hypertext Transfer Protocol (HTTP) [9] og Simple Mail Transfer Protocol (SMTP) [10] er SIP en tekstbasert protokoll. SIP deler også mye av det semantiske fra HTTP, og syntaksen på forespørsler og responser har også mange likheter. I tabell 2.1 er det en beskrivelse av de vanligste SIP-forespørslene som er beskrevet i RFC 3261 [1, s. 253]. Responskodene [1, s. 181] er forklart i tabell 2.2 på neste side.

Responskode	Beskrivelse, og eksempler på responser.
1xx (100, 101, ..., 199)	Informasjon om progresjon '100 Trying', '180 Ringing'
2xx (200, 201, ..., 299)	Suksess '200 OK'
3xx (300, 301, ..., 399)	Omdirigering (redirect) og videresending '301 Moved Permanently', '302 Moved Temporarily'
4xx (400, 401, ..., 499)	Feil gjort av klient '403 Forbidden', '404 Not Found'
5xx (500, 501, ..., 599)	Feil hos server '502 Bad Gateway', '504 Server Time-out'
6xx (600, 601, ..., 699)	Global feil '600 Busy Everywhere', '603 Decline'

Tabell 2.2: SIP-reponser

Enheters oppgaver i SIP

Det er fire forskjellige enheter, eller entiteter, i et SIP-nettverk. Hver av enhetene har en spesiell oppgave de utfører. Enhetene er enten klienter som utfører forespørsler, servere som mottar forespørsler og sender respons tilbake, eller en enhet som kan opptre som både klient og server.

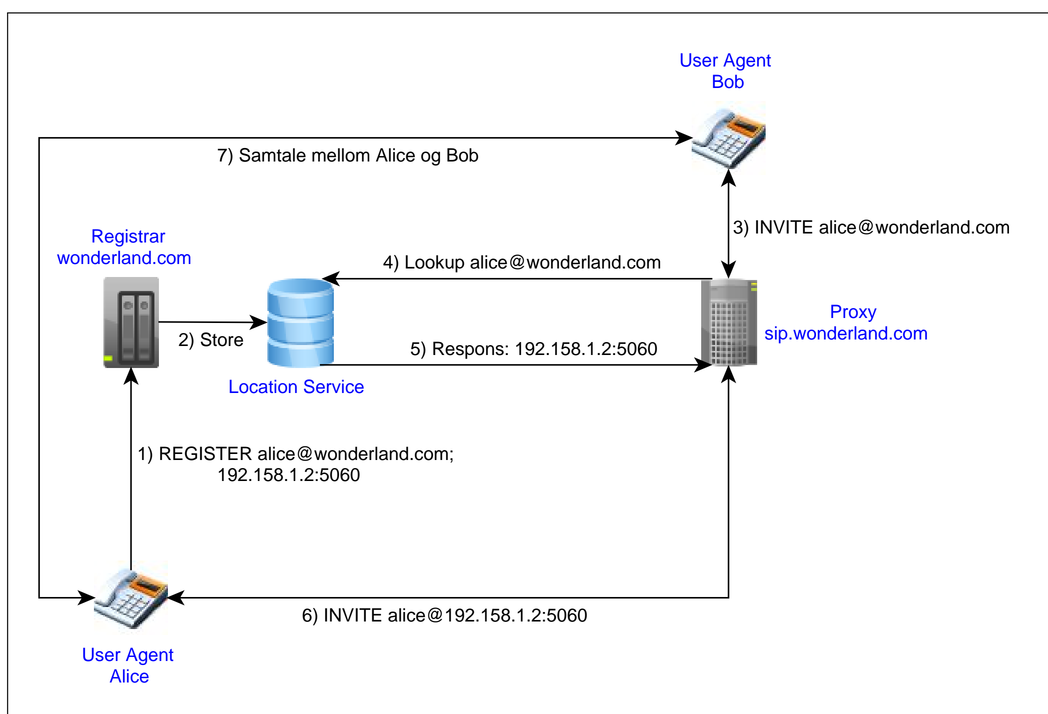
User Agent

User Agent (UA)-er, eller brukeragenter, er applikasjoner som er installert på endepunkter. Disse endepunktene kan f.eks. være datamaskiner med en SIP-applikasjon installert, en trådløs enhet, eller en PDA. Disse enhetene har hovedsaklig to funksjoner. De skal lytte etter SIP-forespørsler, og de skal kunne sende og respondere på SIP-forespørsler. Disse UA-ene kan innta to roller. Når en UA sender en SIP-forespørsel inntar den rollen som en User Agent Client (UAC), når en UA mottar en SIP-forespørsel opptrer den som en User Agent Server (UAS) [1, s. 33].

SIP-servere

SIP-servere er enhetene i nettverket som mottar forespørsler og responderer på disse. I listen nedenfor er de tre server-enhetene registrar, proxy og redirect forklart. Listen inneholder også en beskrivelse av hva en Location Service er.

En enhet kan være en kombinasjon av flere av enhetstypene, slik at en registrar også samtidig kan opptre som en proxy [1, s. 25].



Figur 2.3: REGISTER

Registrar er den delen av en SIP-server som mottar REGISTER-forespørsler [1, s. 23]. REGISTER-forespørsler blir sendt ut av UA-er med jevne mellomrom, og ved oppstart, for å gjøre SIP-serveren oppmerksom på hvilke nettverksadresse endepunktet befinner seg på. Forespørselen inneholder informasjon om IP-adressen, porten den lytter

på og SIP-adressen (se seksjon 2.2.1 på neste side) til endepunktet. Denne informasjonen blir brukt til å knytte SIP-adressen til hvor endepunktet faktisk befinner seg, og lagres i en Location Service (LS) (se neste punkt). Punkt 1) i figur 2.3 viser UA-Alice som sender en `REGISTER`-forespørsel til registraren.

Location Service (LS) er ikke en egen enhet, men gjerne en database vedlikeholdt av registraren som inneholder informasjonen (IP-adresse og port) tilknyttet en SIP-adresse. Denne databasen blir benyttet av Proxy-servere for å vite hvor den skal videresende `INVITE`-forespørsler og andre forespørsler. LS blir også benyttet av Redirect-servere. Punkt 2) i figur 2.3 viser hvordan registraren lagrer informasjon i LS-en, og punkt 4) og 5) viser en forespørsel fra en proxy.

Proxyer er enheter som oppfører seg både som servere og klienter for å sende SIP-forespørsler på vegne av andre klienter. Hovedrollen til en proxy er å hjelpe til med routing, og sørger for at SIP-forespørsler blir videresendt til andre enheter i nettverket som er nærmere destinasjonen klienten prøver å oppnå kontakt med [1, s. 22]. Proxyer blir derfor mellomledd mellom enheter. Den er kun et mellomledd frem til kontakten mellom enhetene er opprettet, deretter går kommunikasjonen direkte mellom enheten. Punkt 3) til 6) i figur 2.3 viser hvordan en proxy hjelper UA-Bob for å oppnå kontakt med UA-Alice.

Redirect-servere er enheter som tar i mot SIP-forespørsler på lik linje med proxyer, men som ikke sørger for å videresende informasjon på vegne av andre enheter. Redirect-servere «peker» heller enheten i riktig retning ved å gjøre oppslag i en LS, og respondere med informasjon korresponderende med forespørselen [1, s. 50]. Redirect-servere blir gjerne brukt for å redusere prosesseringskostnadene på proxyer ved å ikke være et mellomledd mellom enheter.

Eksempel på samtaler

Vi så i figur 2.3 på side 13 et eksempel på hvordan en `REGISTER`-forespørsel ble gjort av UA-Alice til registraren, og at UA-Bob ønsket å sende en `INVITE`-forespørsel for å oppnå kontakt med UA-Alice. Et eksempel på en slik `INVITE`-forespørsel sendt fra Bob til Alice finnes i Listing 2.1 på side 17.

Videre skal vi se på to figurer som illustrerer hendelsesforløpet i en SIP-sesjon med to UA-er og en proxy som mellomledd (figur 2.4a), og en SIP-sesjon med to UA-er og en redirect-server (figur 2.4b).

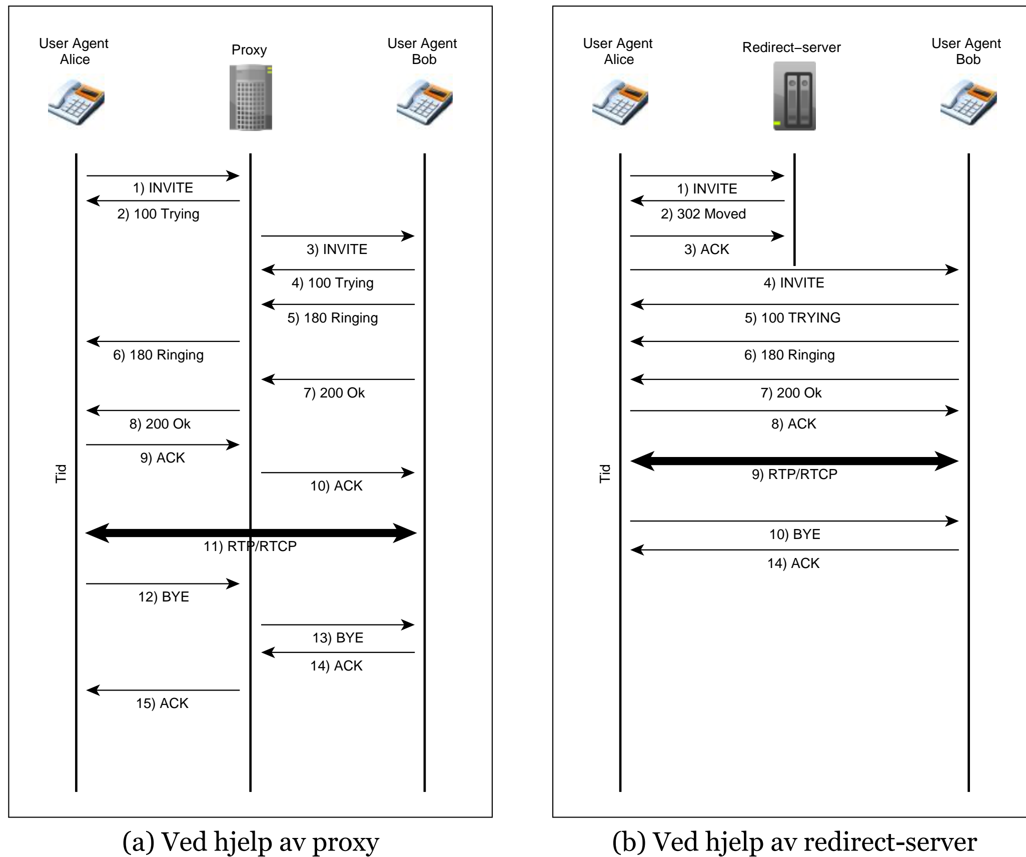
De horisontale linjene er SIP-forespørsler og responser som sendes mellom partene, og de vertikale linjene i figuren representerer tids-aksen. Hva de forskjellige forespørslene gjør er beskrevet i tabell 2.1 på side 11.

SIP-adresser

SIP-adresser er veldig like e-post-adresser. Både SIP-adresser og e-post-adresser følger formatet Uniform Resource Identifiers (URI) [11]. En SIP-adresse kan for eksempel se ut som dette: `sip:alice@wonderland.com`. 'sip:' antyder at dette er en SIP-adresse, og disse adressene skiller seg da fra e-post-adresser. 'wonderland.com' er domenet brukeren tilhører. Dette domenet er gjerne det samme som registrarens domene. 'alice' er brukernavnet, eller identifikatoren innenfor domenet 'wonderland.com'. SIP-adressen er en global identifikator, og kalles gjerne bare for SIP URI [1, s. 147].

Navneoppslag - Address-of-Record (AoR)

I SIP holder det ikke å kun vite SIP-URI-en når man ønsker å opprette en sesjon med en annen SIP-UA. SIP-URI sier ingenting om den fysiske nettverksadressen og porten til SIP-endepunktet som er nødvendig for å



(a) Ved hjelp av proxy

(b) Ved hjelp av redirect-server

Figur 2.4: Eksempel på samtaler

opprette kontakt. For å finne den fysiske nettverksadressen må det derfor gjøres et navneoppslag.

Som vi tidligere har sett benytter SIP-proxyer og redirect-servere en LS. LS-en sin oppgave er å gjøre disse navneoppslagene. Navneoppslag gjøres ved å finne AoR-en til SIP-URI-en. Med dette menes det at vi ønsker å finne den fysiske adressen til en SIP-URI, altså «mappingen», eller relasjonen mellom de to adressene.

SIP-registraren har i forkant lagret i LS-en at adressen «sip:alice@wonderland.com» befinner seg på den fysiske adressen «192.158.1.2:5060». LS-en lagrer da denne «mappingen» mellom den fysiske adressen og SIP-URI-en.

Et eksempel på et navneoppslag finnes i figur 2.3 på side 13 i punkt 4) og 5),

hvor Bob ønsker å hente ut den fysiske adressen til Alice. Bob vet allerede SIP-URI-en til Alice og ikke den fysiske. SIP-URI-en blir sendt fra Bob til proxyen, og proxyen gjør et navneoppslag ved hjelp av LS-en.

Session Description Protocol (SDP)

SDP [12, 13] er en protokoll som benyttes av SIP for at sendere og mottakere skal kunne vite hva som støttes av multimedia-protokoller, kodeker, og overføringsprotokoller. SDP blir gjerne sendt sammen med `INVITE`-forespørsler i forkant av en SIP-sesjon.

SDP består av en rekke parametere som gjør det mulig for SIP-klientene å bli enige om for eksempel hvilken kodek de skal benytte til lyd- eller video-overføring, hvilken protokoll som skal benyttes til overføringen av media, om det skal gjøres noen form for kryptering, og hastighet.

I Listing 2.1 ser vi et eksempel på en `INVITE`-forespørsel sendt fra Bob, hvor han ønsker å invitere Alice til en samtale. I linje 8 blir det angitt `application/sdp` som innholdstype, og i linje 14 blir det angitt hvor lang SDP-meldingen er. Linjene etter denne er selve SDP-innholdet. Fra linje 21 og til slutten ser vi hvilke protokoller og kodekser Bob støtter.

Listing 2.1: Eksempel på `INVITE` med SDP

```
1  INVITE sip:alice@wonderland.com SIP/2.0
2  Via: SIP/2.0/UDP 192.168.1.20:5060;branch=↩
       z9hG4bK80310267dcb7e0118067000ae486eded;rport
3  From: "Bob Byggmesterson" <sip:bob@wonderland.com>;tag=1405310408
4  To: <sip:alice@wonderland.com>
5  Call-ID: 80310267-DCB7-E011-8066-000AE486EDED@192.168.1.20
6  CSeq: 17 INVITE
7  Contact: <sip:bob@192.168.1.20:5060>
8  Content-Type: application/sdp
9  Allow: INVITE, OPTIONS, ACK, BYE, CANCEL, INFO, NOTIFY, MESSAGE, UPDATE
10 Max-Forwards: 70
11 Supported: 100rel, replaces
12 User-Agent: SIPPER for PhonerLitePortable
13 P-Preferred-Identity: <sip:bob@wonderland.com>
14 Content-Length:   416
15
```

```
16 v=0
17 o=- 1617915598 0 IN IP4 192.168.1.20
18 s=SIPPER for PhonerLitePortable
19 c=IN IP4 192.168.1.20
20 t=0 0
21 m=audio 2739 RTP/AVP 8 0 2 3 97 110 111 9 101
22 a=rtpmap:8 PCMA/8000
23 a=rtpmap:0 PCMU/8000
24 a=rtpmap:2 G726-32/8000
25 a=rtpmap:3 GSM/8000
26 a=rtpmap:97 iLBC/8000
27 a=rtpmap:110 speex/8000
28 a=rtpmap:111 speex/16000
29 a=rtpmap:9 G722/8000
30 a=rtpmap:101 telephone-event/8000
31 a=fmtp:101 0-16
32 a=rtcp:2741
33 a=sendrecv
```

Real-time Transport Protocol (RTP)

RTP [14] er, som navnet tilsier en «real-time» transportprotokoll. SIP benytter gjerne denne protokollen til å overføre multimedia. RTP er som SIP, også standardisert av IETF.

Hensikten med RTP er å tilby en ende-til-ende-overføring av data med «real-time»-karakteristikk, slik som overføring av lyd og video i sanntid.

RTP oppretter to tilkoblinger ende-til-ende. Den ene brukes til å overføre «real-time»-dataen, og den andre tilkoblingen benyttes av kontroll-protokollen til RTP, Real Time Control Protocol (RTCP) [15]. RTCP benyttes for å monitorere og kontrollere overføringen, og sørger for Quality of service (QoS) og at multimedia-overføringene er synkronisert.

RTP og RTCP benyttes også av andre protokoller enn SIP. For mer informasjon om RTP og RTCP, se [14]. Og RTCP i kombinasjon med SDP, se [15].

2.2.2 H.323

H.323 er en protokoll og en standard for VoIP laget av International Telecommunication Union - Telecommunication Standardization Sector (ITU-T) [7]. Utviklingen av standarden ble startet allerede i 1995, og hadde som mål å hjelpe VoIP-industrien bort fra proprietære VoIP-løsninger slik at løsningene fra forskjellige leverandører kunne fungere sammen [16]. Allerede i slutten av 1996 hadde standarden fått god oppslutning i VoIP-industrien, og de fleste aktørene laget nå VoIP-løsninger som fulgte standarden.

Standarden forble industri-standard en rekke år, frem til SIP kom på banen og tok opp konkurransen. SIP ble svært godt tatt i mot på grunn av sin enkelhet i forhold til H.323 [16].

2.2.3 Skype

Skype [2] er en P2P VoIP-klient som ble utviklet av de samme utviklerene som laget fildelingsklienten KaZaa [3].

Skype har oppnådd stor popularitet blant brukere på grunn av sin robusthet, og den enkle måten tjenesten fungerer på. Tjenesten gir brukere muligheter til å ringe gratis til andre brukere som også benytter seg av Skype, og det er også muligheter for å kjøpe ringetid som kan benyttes ut på det vanlige telefon-nettverket verden over for en liten sum. I tillegg til å ringe er det også mulig å bruke Skype til å sende lynmeldinger til andre brukere, og til å opprette konferanse-samtaler.

Skype som et program, og den underliggende protokollen, er proprietær. Det er likevel interessant hvordan Skype fungerer. For å bedre forståelsen av hvordan Skype fungerer ble det i 2004 gjort en analyse [17] som tok for seg kommunikasjonen mellom noder i Skype-nettverket.

Ved hjelp av analysen har de funnet ut at enkelte noder i nettverket har et større ansvar. Det er altså to forskjellige node-typer i nettverket.

Den ene node-typen i nettverket er en ordinær node, som ikke har noen spesielle egenskaper. Den andre node-typen kalles en super-node. Disse nodene utgjør kjernen i nettverket, og danner et «overlay»-nettverk (se seksjon 3.2.2 på side 39). Super-nodene har hver for seg ansvar for et utvalg av ordinære noder, og sørger for at disse kan kommunisere med resten av nettverket.

Hvilke ordinære noder som skal bli omgjort til super-noder velges automatisk ut etter enkelte kriterier. Noden må først og fremst oppfylle kravene om oppetid. Dette for å sørge for at super-noder ikke skiftes ut for ofte. Det er også et kriterie at nodene har rikelig med prosesseringskraft, minne, båndbredde og lagringsplass. I tillegg til disse kriteriene er det også viktig at noden står på et åpent nett, og at den ikke er bak noen form for Network Address Translation (NAT) eller brannmur. Dette er spesielt viktig fordi en av super-nodens viktigste oppgaver er i enkelte tilfeller å opptre som et «relay» (relé) for ordinære noder på mindre åpne nettverk.

Skype benytter seg av en variant av Session Traversal Utilities for NAT (STUN) [18] for å finne ut om en node befinner seg på et lukket nettverk. Noder som befinner seg på lukkede nettverk (se seksjon 3.1.2 på side 29) kan oppleve at de ikke oppnår kontakt med andre noder på utsiden av nettverket. Analysen har vist at Skype håndterer NAT- og brannmur-traversering på en svært effektiv måte, ved at de ordinære nodene benytter seg av super-nodene for å videreformidle informasjon til andre noder de ønsker å oppnå kontakt med [19, s. 2]. Skype-tjenesten i sin helhet blir svært robust ved å tilby disse tjenestene. Måten super-nodene i dette tilfellet opptrer på har flere likheter med Traversal Using Relay NAT (TURN) [20]. I tilfeller der det er mulig for de ordinære nodene å opprette direkte kontakt, gjøres dette.

Det som gjør at Skype ikke er et rent P2P-system er at tjenesten benytter seg av en sentral enhet for autentisering av brukere. Denne delen av tjenesten er altså ikke distribuert, men styres av Skype ved hjelp av sentraliserte servere.

Det kan være flere grunner til at Skype har valgt å sentralisere enkelte deler av tjenesten. En av grunnene kan være at de ønsker å ha kontroll på autentiseringsmekanismen, og at de derfor ikke ønsker å distribuere denne delen. Det kan også være på grunn av sikkerhet, i og med at brukernavn og passord er lagret her. En sentral enhet kan også være nyttig i forbindelse med «bootstrapping» av nettverket, slik at noder kan kontakte den sentrale enheten, og så bli tildelt en super-node. Dette blir da i stede for å finne en super-node på egenhånd.

Skype-tjenesten kategoriseres som et ustrukturert hybrid P2P-nettverk. Dette er fordi den både har super-noder og sentraliserer deler av tjenesten (autentisering av brukere). Som vi senere skal se i seksjon 3.2.1 på side 37 er dette et svært skalerbart P2P-nettverk, noe som igjen gjør at Skype kan håndtere store brukermasser.

2.3 Oppsummering

Vi har i dette kapittelet sett hva VoIP er, og hvilke fordeler VoIP har fremfor tradisjonell telefoni over PSTN. Det har blitt forklart hvordan signaleringsprotokollen SIP fungerer, og hvordan denne benyttes i forbindelse med VoIP. Vi har også fått en kort historisk presentasjon av H.323, og en gjennomgang av VoIP-tjenesten Skype.

Kapittel 3

Peer-to-Peer (P2P)

I den første delen av dette kapittelet introduserer jeg først begrepet Peer-to-Peer (P2P). Jeg presenterer også hvorfor P2P er et godt alternativ til den vanlige klient/server-arkitekturen, samt fordeler og ulemper med P2P.

I den andre delen av dette kapittelet forklarer jeg hvordan arkitekturen og topologien i P2P-nettverk kan deles i to kategorier; strukturerte P2P-nettverk, og ustrukturerte P2P-nettverk.

Videre skal vi se på hvilke bruksområder P2P tradisjonelt har hatt med presentasjon av enkelte applikasjoner og protokoller. Disse protokollene er spesielt rettet mot de ustrukturerte P2P-nettverkene. Til slutt vil det være en presentasjon av strukturerte P2P-nettverk.

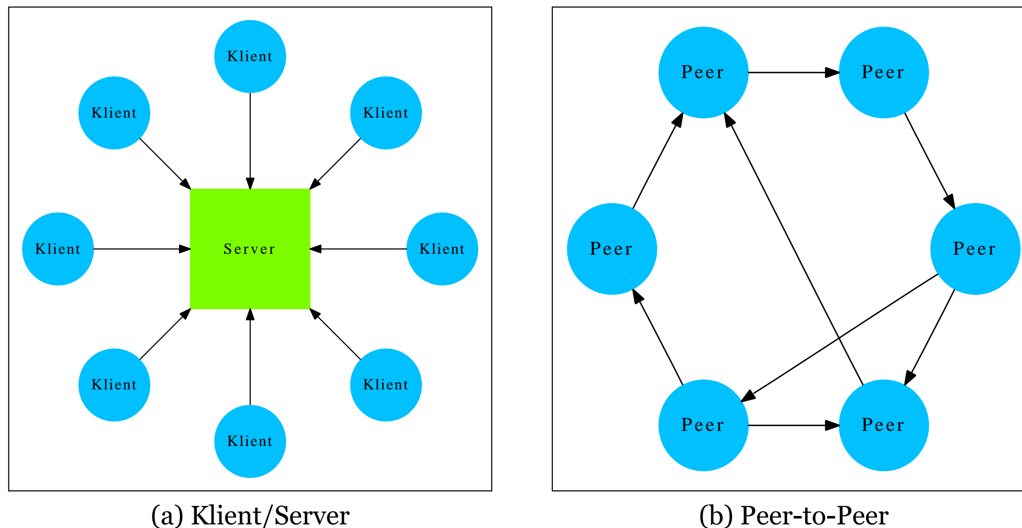
3.1 Hvorfor Peer-to-Peer?

I de tradisjonelle nettverksmodellene som bruker sentrale servere med en klient/server-arkitektur, har kostnader til drift, vedlikehold, overføring av data og prosessering ofte blitt høye. Skalerbarheten til denne typen nettverksmodeller har også vært dårlig, da man hele tiden vil ha behov for flere og kraftigere servere for å dekke behovene som oppstår når flere

noder/klienter benytter seg av tjenesten til den sentrale serveren.

Sentrale servere er også et «single point of failure»[1], da alle noder/klienter som er tilkoblet ofte vil være avhengig av denne sentrale serveren for at tjenesten skal fungere. Vi skal videre se på hvorfor P2P kan være et godt alternativ til den tradisjonelle nettverksmodellen, og da spesielt med tanke på «real-time»-protokoller som benyttes til forskjellige typer kommunikasjon og multimedia. Hovedvekten vil bli lagt på protokollen SIP.

Figur 3.1a illustrerer hvordan tradisjonelle klient/server-nettverk er knyttet sammen, med en sentral enhet i midten. Figur 3.1b viser hvordan P2P-nettverk er knyttet sammen av «peers», og på denne måten danner ett nettverk.



(a) Klient/Server (b) Peer-to-Peer

Figur 3.1: De to nettverksmodellene

[1]Begrepet «single point of failure» brukes på enheter i et system som er vitale, i den forstand at systemet ikke vil kunne fungere uten enheten.

3.1.1 Fordeler med P2P-teknologi

Systemer som baserer seg på P2P-teknologi har av natur en større skalerbarhet enn man vil kunne oppnå ved bruk av den tradisjonelle nettverksstrukturen med en server og flere klienter. Disse systemene har også en høyere feiltoleranse og er ofte mer robuste. Dette kommer av at P2P-systemene ikke har sentrale enheter i nettverket, og fordi de ofte er selvorganiserte. Fordelene med P2P-teknologi er fordeling av arbeidslast, ett delt ansvar mellom nodene og skalerbarheten til systemet.

Fordeling av arbeidslast

P2P-teknologi tilbyr en god måte å fordele arbeidslast på. CPU- og minnebruk ved prosessering av data, båndbredde ved overføring av data, og lagringsplass ved oppbevaring av data er noen av punktene som kan fordeles på en distribuert måte i et P2P-nettverk.

Delt ansvar

I et P2P-nettverk er det ikke nødvendig med en sentral server for koordinering av nettverket. Nettverket som en enhet, er selv ansvarlig for vedlikehold og lagring av informasjon og data. Ved hjelp av utveksling av informasjon om tilstanden til nettverket vil nettverket kunne håndtere endringer og forandringer når dette oppstår.

Nettverket består av enheter som kalles noder. Disse nodene er likestilt med hverandre. Med likestilt menes det at hver node i utganspunktet har en like stor betydning og viktighet som de andre nodene i nettverket.

Siden nodene i nettverket samarbeider om koordinering av nettverket ved hjelp av informasjon som sendes i mellom nodene medfører dette at ansvaret for at nettverket fungerer fordeles på alle de involverte nodene i nettverket.

Dette medfører også at det ikke vil ha en alt for stor betydning hvis en node faller ut eller forlater nettverket. Resten av nettverket vil da kunne håndtere tapet av noden, og vil kunne gjennopprette en normal tilstand relativt raskt. Dette er selvfølgelig avhengig av algoritmen som benyttes, og hvor god redundans-håndtering algoritmen har.

Den administrative rollen en server har i et tradisjonelt nettverk vil også fordeles/distribueres mellom nodene i nettverket ved å likestille nodene. Fordelingen av ansvaret ved å likestille nodene vil da gjøre at nettverket ikke lenger har et «single point of failure».

Uten et behov for en sentral server, på grunn av det distribuerte ansvaret i nettverket, vil P2P-modellen i mange tilfeller fungere som en god erstatning for den tradisjonelle klient/server-arkitekturen. Hvis behovet for en sentral koordinering av nettverket ikke lenger er tilstedeværende vil man derfor kunne kutte disse kostnadene, og erstatte det tradisjonelle nettverket med et P2P-nettverk. Dette forutsetter at P2P-teknologien eksisterer fra før, og ikke vil føre til en kostnad.

Skalerbarhet

I tradisjonelle nettverks-modeller med klient/server-arkitektur begrenses skalerbarheten i stor grad av evnen serverne har til å prosessere, overføre og lagre data. Ønsker man da å øke ressursene, hvis et behov om dette melder seg, blir man nødt til å tilrettelegge for flere servere, eller man blir nødt til å gå til anskaffelse av kraftigere servere [21, s. 13].

Til forskjell fra den tradisjonelle nettverks-modellen fungerer P2P-modellen på en litt annen måte når det gjelder ressurser. I P2P-modellen er det nodene selv som formidler ressursene, så hvis en node ankommer et P2P-nettverk vil det ikke være slik at denne noden kun vil benytte seg av ressursene som er tilstede i nettverket, slik den ville gjort i et scenario med en klient/server-arkitektur, men den vil også kunne bidra med ressurser på lik linje som de andre nodene i nettverket. På tross av økt ressursbe-

hov totalt i nettverket når en ny node ankommer, vil det da også bli en økt mengde ressurser totalt tilgjengelig.

3.1.2 Problemer med P2P-teknologi

På tross av mange positive sider med P2P-modellen finnes det også negative sider, uløste problemer og også nye problemer som ikke har hatt så stor innvirkning i den tradisjonelle klient/server-arkitekturen.

Innholdet er spredt

I den tradisjonelle klient/server-arkitekturen er klienten ofte forhåndskonfigurert på en slik måte at den vet hvor den skal henvende seg for å få tilgang til ressursene serveren tilbyr. Med ressurser menes det i dette tilfellet for eksempel data og informasjon klienten ønsker å få tilgang til, eller for eksempel et oppslag som returnerer informasjon fra serveren til klienten.

Siden server-rollen i P2P-modellen er distribuert vil ikke klienten nødvendigvis vite hvor data og informasjon befinner seg i nettverket. Nodene i nettverket blir da nødt til å samarbeide om å kunne tilby en tjeneste som kan tilby denne funksjonaliteten på en effektiv måte.

Noder som blir utilgjengelige

Som tidligere nevnt kan serveren i den tradisjonelle klient/server-arkitekturen ofte bli et «single point of failure», noe som vil medføre at informasjonen som er lagret på serveren vil være utilgjengelig i perioder hvor serveren ikke fungerer som den skal. I P2P-modellen vil man også kunne møte på tilsvarende problemer hvis informasjon er lagret på en av nodene i nettverket, og denne noden faller ut av nettverket. Dette kan føre

til permanent tap av data, og dette kan også føre til at noder som har falt ut kommer tilbake i nettverket med data som er utdatert.

Privat og sensitiv informasjon

Data og informasjon i P2P-nettverket er av natur spredt ut blant alle nodene i nettverket. Denne informasjonen kan i mange tilfeller være privat eller sensitiv. Dette er da materiale man ikke ønsker at uvedkommende får tilgang til.

I den tradisjonelle klient/server-arkitekturen kontrolleres dette ofte med en form for tilgangskontroll hvor man for eksempel må kunne stadfeste hvem man er ved hjelp av brukernavn og passord eller andre typer autentisering, i tillegg til kryptering av data.

Siden data i P2P-modellen er spredt rundt på forskjellige noder må man anta at uvedkommende kan ha tilgang til disse nodene, da man ikke i alle tilfeller har kontroll på alle nodene i nettverket. Det vil derfor oppstå et ekstra behov for å sikre informasjonen mot innsyn, da gjerne ved hjelp av kryptering.

Store mengder informasjonsflyt

Ett av målene ved å innføre P2P-modellen er, som tidligere nevnt, for å unngå å ha sentral enhet alene som koordinerer nodene i nettverket. Denne sentrale enheten vil ikke ha like store skaleringsmuligheter som det et P2P-nettverk vil kunne tilby. I tillegg til at den er et «single point of failure».

For at P2P-modellen skal kunne dekke denne «server-rollen» er man da avhengig av at alle nodene i nettverket samarbeider om å oppdatere hverandre, slik at nodene i nettverket blir klar over forandringer som skjer i nettverket. P2P-modellen har ikke den samme hiearkiske oppbygningen

som et tradisjonelt klient/server-nettverk med serveren på toppen, men i stede en flat struktur som knytter nodene sammen.

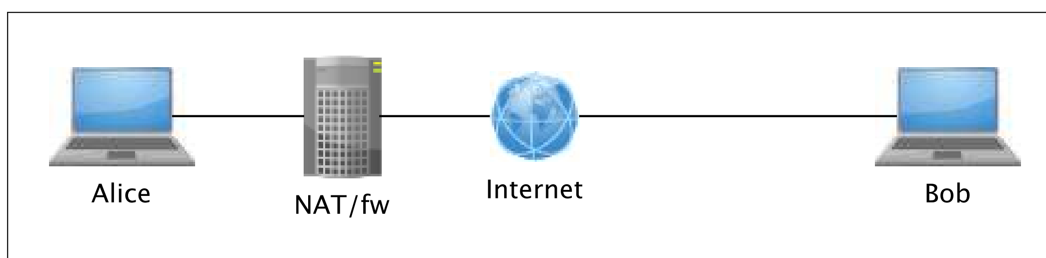
Alle disse opplysningene som utveksles mellom nodene kan utgjøre store mengder data som skal sendes rundt. Dette kan bli problematisk i enkelte tilfeller, og det kan bli en belastning for nettverket.

Enkelte typer applikasjoner er ikke like avhengig som andre av at tilstanden til nettverket er helt oppdatert. Slik som i vanlig nettverkstopologier hender det til tider at datapakker tar andre veier gjennom nettverket for å nå sitt mål enn andre datapakker. Det samme vil kunne skje i et P2P-nettverk. Dette kan i noen tilfeller føre til at enkelte oppgaver tar lengre tid enn vanlig. Dette kan da skyldes at nodene ikke er oppdatert på hvilke veier som vil gi den korteste og raskeste ruten til et mål.

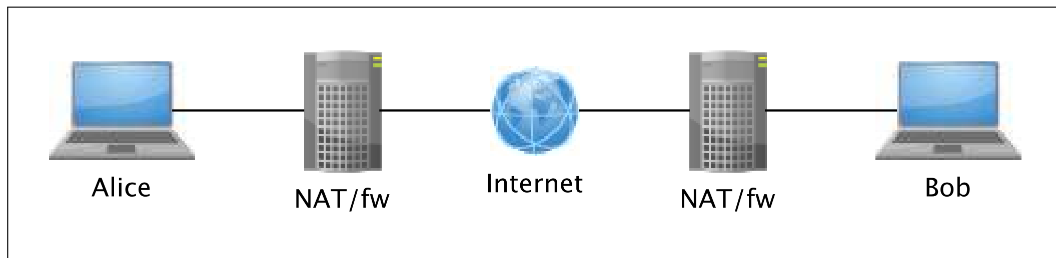
På grunn av ulike applikasjoners behov er det derfor viktig å sørge for at mengden informasjon som utveksles verken blir høyere enn nødvendig, eller for lav slik at dette går ut over kravene og kvaliteten til applikasjonen.

NAT- og brannmur-problematikk

En stor del av enhetene som i dag er koblet til internett er bak en NAT og/eller en brannmur [22, 23]. Hensikten med brannmurer er å unngå at skadelig trafikk i form av for eksempel virus og ormer skal slippe igjennom til enhetene bak brannmuren. Brannmurer kan også sørge for å sperre enkelte typer trafikk, og/eller kun slippe gjennom enkelte typer trafikk.



Figur 3.2: En node er bak NAT/brannmur



Figur 3.3: Begge noder bak en NAT/brannmur

I andre tilfeller kan enheter være bak en NAT. En NAT sørger for at flere enheter kan kobles opp mot internett med kun en IP-adresse utad. Denne teknikken gjør at enheter utenfor NAT-en ikke har mulighet til å se innsiden av nettverket. Denne metoden har blir brukt i tilfeller hvor man ikke har et tilstrekkelig antall IP-adresser tilgjengelig, men man samtidig trenger å ha flere enheter koblet opp til internett. Metoden brukes også i tilfeller hvor det er ønskelig å skjule hele nettverket bak en enkelt IP-adresse.

Innenfor et nettverk med NAT tildeles alle enhetene en egen IP-adresse. Denne IP-adressen gjelder kun innefor det samme nettverket, og kalles gjerne private eller lokale nettverksadresser.

Problemene med NAT oppstår når en enhet på innsiden kommuniserer med enheter på utsiden (se figur 3.2 på forrige side). Siden den private nettverksadressen kun gjelder på innsiden av NAT-nettverket vil det ikke være mulig for enheten på utsiden å se annet en adressen NAT-nettverket har utad. Det vil da i mange tilfeller bli umulig for enheten på utsiden å opprette kontakt med enheten på innsiden hvis ikke en tilkobling fra innsiden allerede er opprettet. Problemet oppstår også når begge enhetene er bak hver sin NAT, slik det er illustrert i figur 3.3.

De samme problemene oppstår også i forbindelse med bruk av brannmurer. Brannmurer velger ofte å kun slippe gjennom trafikk som er initialisert fra innsiden. Et eksempel på dette er trafikk mot nettsider gjennom HTTP-protokollen, hvor klienten da sender en forespørsel til web-serveren på port 80, og mottar data tilbake uten at brannmuren stopper dataen i

å nå klienten. Dette gjelder ikke bare trafikk over HTTP-protokollen, men også andre protokoller som benytter seg av den tradisjonelle klient/server-arkitekturen, hvor serveren gjerne har en globalt kjent IP-adresse, og klienten er på et privat nettverk bak en brannmur eller en NAT.

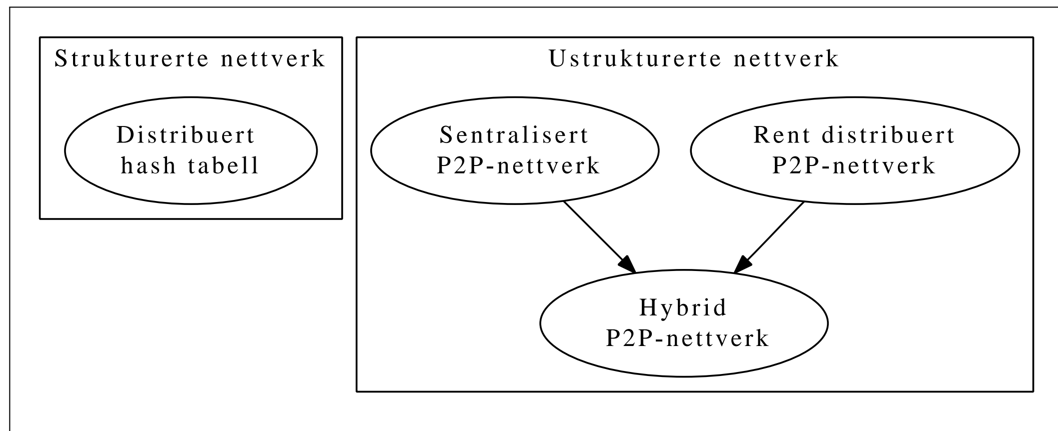
Det som derimot skaper problemer er at enkelte protokoller ikke nødvendigvis har det samme hendelsesforløpet som HTTP, og at portene som skal benyttes ikke er initialisert fra innsiden, og dermed ikke nødvendigvis åpnet i brannmuren.

Dette problemet kan for eksempel oppstå når en klient på utsiden av brannmuren prøver å kontakte en klient på innsiden ved hjelp av SIP-protokollen. Klienten på innsiden kan ikke nødvendigvis vite at noen prøver å kontakte den når trafikken på porten den lytter til blir blokkert av brannmuren. Problemet kan også oppstå i situasjoner hvor to forskjellige klienter på hvert sitt nettverk bak brannmurer eller NAT-er prøver å få kontakt med hverandre. Sistnevnte scenario er veldig typisk for P2P-modellen hvor klientene ønsker direkte kontakt med hverandre.

Eksempler på eksisterende løsninger som hjelper til mot denne typen problemer er STUN [18], TURN [20] og Interactive Connectivity Establishment (ICE) [24].

3.2 Arkitektur og topologi

Måten «peers» i et P2P-nettverk kobles sammen på er enten på en strukturert måte, eller en ustrukturert måte. De forskjellige typene P2P-nettverk kategoriseres som i figur 3.4 på neste side. I I denne seksjonen skal vi se hva forskjellene på disse to strukturene er, egenskapene, og hvilke fordeler og ulemper disse medfører [21].



Figur 3.4: Strukturerte og ustrukturerte P2P-nettverk

3.2.1 Ustrukturerte nettverk

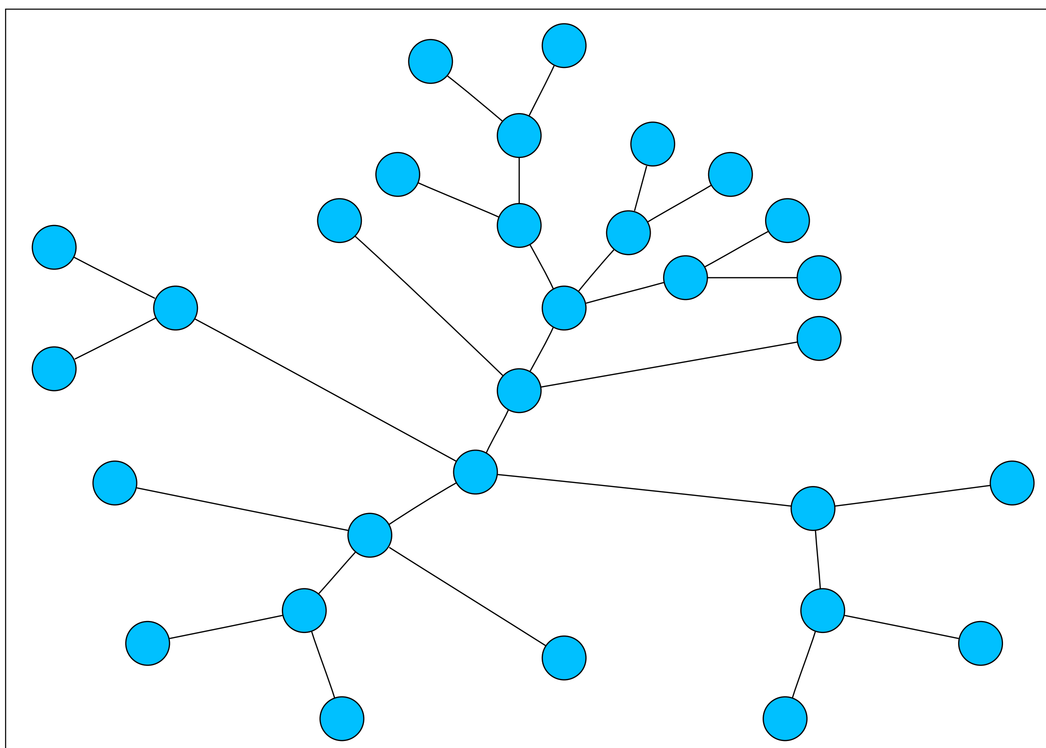
Grovt sett kan de ustrukturerte P2P-nettverkene deles inn i to forskjellige kategorier. I den ene kategorien har man arkitekturer uten sentrale enheter, og i den andre kategorien er det varianter av nettverk med sentraliserte enheter [25].

Det finnes også en P2P-kategori som bryter med begge disse kategoriene, så derfor introduseres enda en kategori som er en blanding av disse to. Denne kategorien blir derfor en hybrid av de to andre. Denne kategorien kalles Hybride P2P-nettverk (seksjon 3.2.1 på side 37).

Figur 3.5 på neste side illustrerer hvordan et ustrukturert nettverk kan se ut. Noder, eller «peers», er tilkoblet hverandre, og man kan se klare forgreninger eller klaser med noder.

Desentraliserte/Rent distribuerte nettverk

Av de tre typene ustrukturerte P2P-nettverk, er det den desentraliserte distribuerte nettverksarkitekturen som er «renest». Med renest menes det at alle noder i dette nettverket har samme rolle, og at det ikke finnes noen sentrale enheter i dette nettverket.

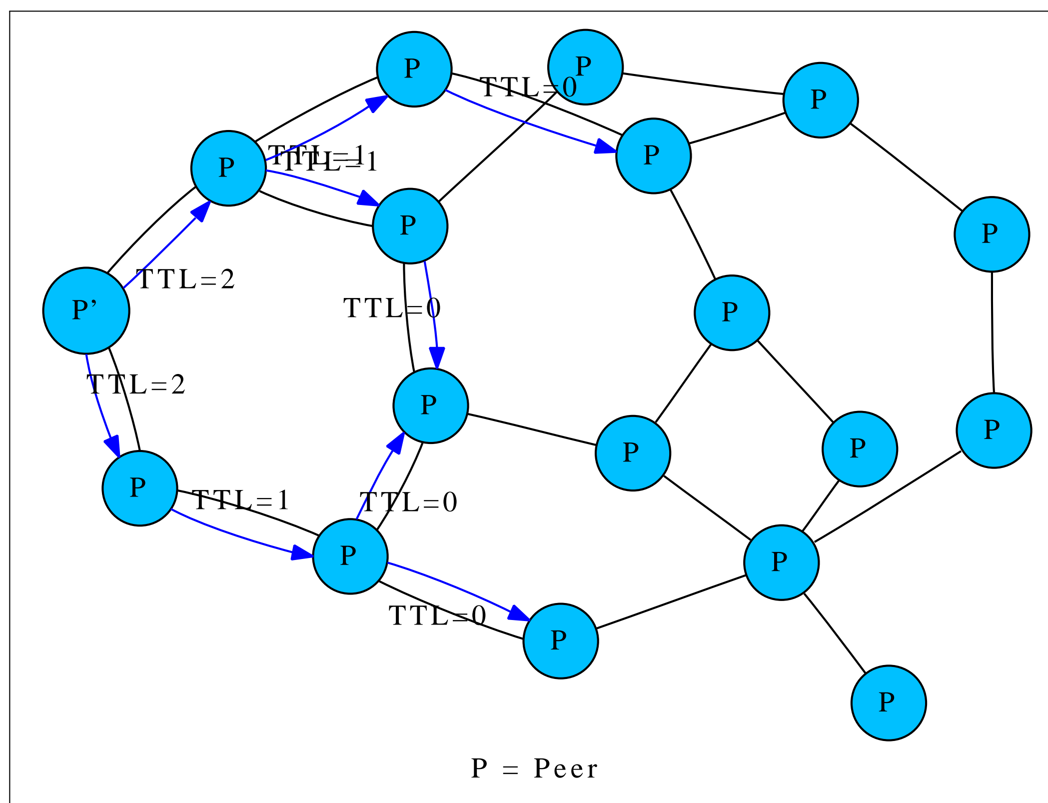


Figur 3.5: Ustrukturerte P2P-nettverk.

Noder i dette nettverket danner et «overlay»-nettverk ved å opprette forbindelser med hverandre. For å søke etter innhold i nettverket sender noder forespørselen til sine tilkoblede nabo-noder, som igjen sender søket videre til sine nabo-noder. Når søket gir resultater sendes dette gjerne tilbake til noden som utførte søket gjennom den samme ruten søket tok.

Disse forespørslene er ofte begrenset med en Time-to-live (TTL), som gjør at man kan begrense informasjonsmengden nettverket må håndtere, og at hopp fra node til node ikke går i løkker. Dette fører også til at man ikke vil kunne garantere at et søk vil returnere riktige resultater, siden innhold kan befinne seg i nettverket, men på grunn av TTL vil man ikke kunne gjøre nok hopp slik at man når noden som har innholdet.

Figur 3.6 på neste side viser hvordan et søk fra noden *P'* brer seg utover i det desentraliserte P2P-nettverket med en TTL satt til verdien 2.



Figur 3.6: Søk i desentraliserte P2P-nettverk.

En protokoll vi skal se nærmere på i seksjon 3.3.3 på side 44, Freenet [26], benytter seg av denne arkitekturen. Arkitekturen har vist seg å kunne tilby en grad av anonymitet, siden innholdet som sendes over nettverket går fra node til node, og at det derfor er vanskelig eller umulig å fastslå hvor opphavet til innholdet kommer fra. Protokollen Gnutella [27] (seksjon 3.3.2 på side 43) benytter også et rent distribuert nettverk.

Fordeler

Fordelen med denne arkitekturen er helt klart at det ikke eksisterer noen form for sentrale enheter i nettverket. Dette gjør at systemet ikke har noe «single point of failure», og at man derfor vil kunne ha et fungerende system selv om noder forlater nettverket.

Ulemper

Ved bruk av denne typen arkitektur kreves det mye informasjonsflyt mellom nodene i nettverket for å håndtere og koordinere hverandre. Også ved søk kan man som nevnt ikke garantere at søkeresultatene gjenspeiler et fullt bilde av hele nettverket, men kun de delene av nettverket man når før TTL-begrensningen slår til. TTL-begrensningene fører derfor til at mindre populært innhold ikke blir like lett tilgjengelig for hele nettverket som populært innhold som gjerne flere noder har tilgjengelig.

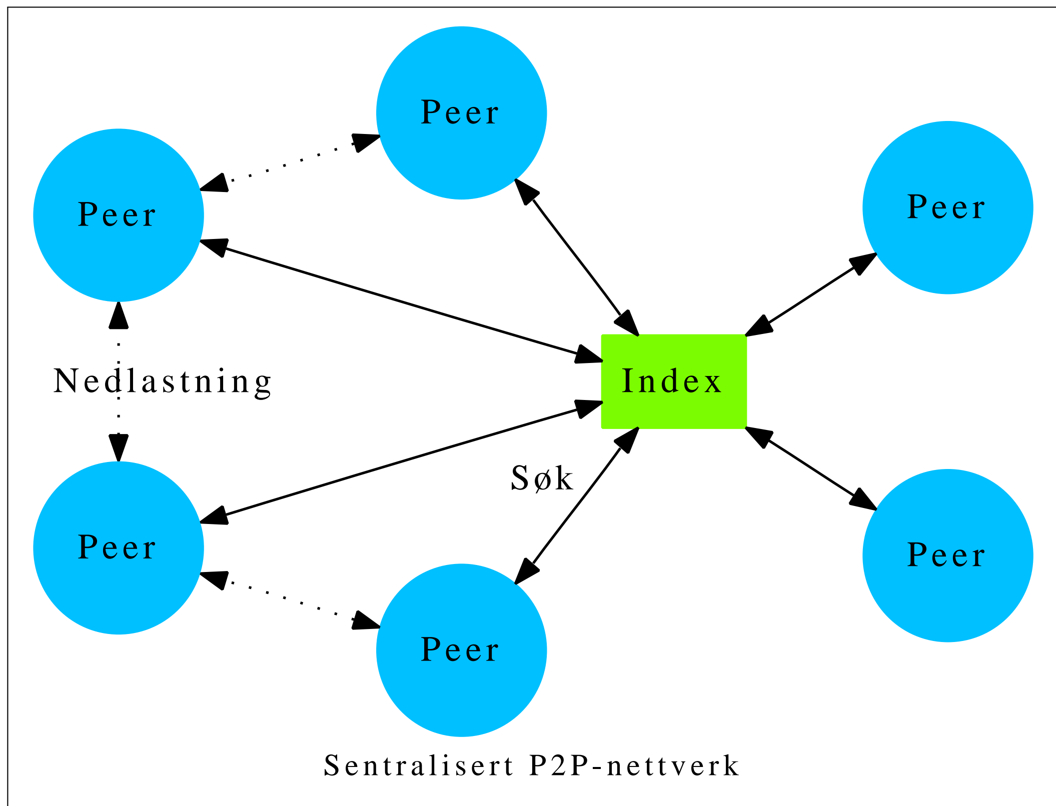
Skaleringsevnen til denne modellen begrenses av den store informasjonsflyten som foregår i nettverket [21, s. 36]. En løsning på dette kan i enkelte tilfeller være å heller følge den hybride nettverks-arkitekturen (3.2.1 på side 37).

Sentraliserte P2P-nettverk

Det vanlige for nettverksarkitekturen i et sentralisert P2P-nettverk er å ha en eller flere sentrale enheter som gjør en bestemt oppgave (se figur 3.7 på neste side). Denne oppgaven kan for eksempel være å samle inn indekseringer fra nodene i nettverket, og videreformidle dette ved forespørsler fra andre noder. En annen oppgave denne sentrale enheten kan ha, er å sørge for autentisering av nodene ved for eksempel å opptre som en Certification Authority (CA).

Denne typen arkitektur ligner mye på den tradisjonelle klient/server-arkitekturen. Det som skiller den fra dette, er at innholdet ikke blir sendt fra serveren til klienten, men fra klient til klient. Dette sparer serveren for båndbredde og lagringsplass.

Et eksempel på protokoller som benyttet seg av denne arkitekturen er Napster [28], som kan leses om i avsnitt 3.3.1 på side 42.



Figur 3.7: P2P-nettverk med sentral enhet

Fordeler

I denne arkitekturen legges mye av tjenestens funksjonalitet over på det sentrale enheter i nettverket. Dette åpner for at klient-applikasjonen som benyttes ikke trenger å være så avansert.

Hvis den sentrale enheten kan tilby tilstrekkelig med ressurser i forhold til antall klienter kan søketiden i denne arkitekturen vise seg å være bedre enn for eksempel en nettverksarkitektur uten en sentral enhet.

Ulemper

Sentrale enheter i nettverket har også sine ulemper. Mye av motivasjonen til å benytte seg av P2P er nettopp for å slippe disse sentrale enhetene

i nettverket. Dette er fordi de ofte kan bli flaskehalser, og fordi deler av tjenesten ikke vil fungere hvis den baserer seg på sentrale enheter («single point of failure»). Bruk av sentrale enheter fører også til at skalerbarheten til tjenesten blir svekket.

Det ble også nevnt i avsnittet om fordeler at klient-applikasjonen i mange tilfeller kan implementeres på en enklere måte ved bruk av denne typen arkitektur. I samme tilfelle vil også dette føre til en mer avansert rolle for det sentrale elementet i nettverket.

Sett i sammenheng med skaleringsproblemene som lett oppstår ved bruk av denne arkitekturen vil søketiden ved denne type arkitektur lett bli svekket om de sentralene enhetene får for mye å gjøre.

Hybride nettverk

Ved å kombinere de to andre ustrukturerte arkitekturene ender man opp med en blanding. Dette går under betegnelsen hybride P2P-nettverk, og tar de beste egenskapene fra begge. I stede for å ha sentrale enheter i nettverket er det ved bruk av denne arkitekturen noder i nettverket som får tildelt roller og ansvar. Noder som blir tildelt andre roller og ansvar kalles gjerne super-noder. Super-nodene er ofte valgt ut automatisk ved hjelp av algoritmer i systemet som avgjør hvilke noder som er best egnet til å opptre som en super-node.

Kriteriene det velges ut i fra er ofte krav om oppetid, prosseseringskraft, minne og båndbredde tilgjengelig. Disse super-nodene får gjerne tilleggsroller i nettverket, og blir gjerne ansvarlig for et utvalg av vanlige noder i nettverket. Super-noder samler gjerne inn meta-informasjon om hva noder deler, og viderefører dette til andre super-noder.

I denne arkitekturen er det super-nodene som danner kjernen av nettverket ved å opprette et nettverk mellom seg. Super-nodene er ansvarlig for et utvalg andre vanlige noder, og super-nodene gjør da f.eks. søk på vegne av de vanlige nodene. Siden super-nodene besitter informasjon om hva sine

under-noder (nodene som super-noden er ansvarlig for) deler, vil super-nodene kunne sende forespørsler til hverandre om hva deres under-noder deler.

Super-nodene tar da rollen som en sentral enhet egentlig, men i stede for at dette er dedikerte servere slik som i klient/server-arkitekturen er det noder som har blitt tildelt denne rollen. Rollene til super-nodene kan også sammenlignes med de sentrale enhetene i sentraliserte P2P-nettverk. Et eksempel på en protokoll som benytter seg av denne arkitekturen er FastTrack (se seksjon 3.3.4 på side 47).

Fordeler

En av fordelene med denne arkitekturen er at vanlige noder har én enhet å forholde seg til når den skal foreta søk eller andre operasjoner. Siden det er super-nodene som videreformidler søk på vegne av sine under-noder vil det derfor føre til mindre informasjonsflyt rundt i nettverket, så lenge det kun er et fåtall super-noder som gjør disse oppgavene. Problemene med mye informasjonsflyt vi så i de rent distribuerte P2P-nettverkene er derfor løst i denne arkitekturen, noe som igjen vil føre til bedre skalerbarhet.

Ulemper

Siden vi i denne arkitekturen har noder som har et større ansvar enn andre noder vil vi igjen få «single point of failure». I denne arkitekturen vil en feil på en super-node ramme et utvalg under-noder, men det vil ikke ha like store konsekvenser som i P2P-nettverkene som var avhengig av sentrale enheter.

3.2.2 Strukturerte nettverk

Et av de største problemene med P2P er at noder ikke nødvendigvis vet hvor data de letere etter befinner seg. Før vi skal se på hvordan disse problemene er løst i strukturerte P2P-nettverk, også kalt strukturerte «overlay»-nettverk, skal jeg som en oppsummerering fra forrige seksjon nevne hvilke løsninger på problemet som ble foreslått der. Vi skal også se på hvilke andre egenskaper de strukturerte P2P-nettverkene har, i tillegg til fordeler og ulemper.

Hvilke problemer ønsker vi å løse ved å bruke strukturerte nettverk?

Blant de sentraliserte P2P-nettverkene (se 3.2.1 på side 35) finner vi eksempelvis tjenester som Napster [28]. Napster benytter seg av en sentralisert server som håndterer alle søk og spørringer. Denne sentraliserte serveren blir da først og fremst ett «single point of failure», men serveren vil også raskt kunne bli overbelastet hvis det blir stor pågang fra mange noder i nettverket.

I de desentraliserte/rent distribuerte P2P-nettverkene (se 3.2.1 på side 32) finner vi eksempelvis Gnutella [27] og Freenet [29]. For å søke etter innhold i disse nettverkene benyttes gjerne former for «flooding» ved at søket sendes til alle nabo-noder i nettverket, som igjen blir videresendt søket til sine nabo-noder inntil en TTL er nådd. Søk ved hjelp av «flooding» fører til at antallet meldinger som utveksles i nettverket stiger linjært i forhold til antall noder i nettverket. Søket blir som sagt også begrenset av TTL, noe som igjen fører til at søket i seg selv aldri når frem til noder som muligens «besitter» innholdet det søkes etter, men som befinner seg for mange hopp unna søker-noden.

Den siste varianten av ustrukturerte P2P-nettverk er de hybride P2P-nettverkene (se 3.2.1 på side 37). I denne varianten er det gjerne super-noders ansvar å vedlikeholde og utføre indeksering av innholdet i

nettverket, på lik linje med sentraliserte servere. Selv om faren for feil er mindre i de hybride P2P-nettverkene, siden ansvaret deles mellom supernodene, kan det fortsatt forekomme feil som rammer et stort antall noder.

Organisering av strukturerte P2P-nettverk

Måten noder i et strukturert P2P-nettverk organiseres er i form av et P2P-«overlay» nettverk. Dette nettverket består av noder som er koblet sammen ende-til-ende, og har som hensikt å kunne tilby sanntidskommunikasjon mellom alle nodene i nettverket. Nodene i nettverket kalles «peers». Disse «peerene» samarbeider kollektivt for å vedlikeholde en distribuert database ved hjelp av en algoritme. Ved hjelp av den distribuerte databasen er P2P-«overlayet» i stand til å lagre og utveksle data på en effektiv måte. Nettverket sørger også for å vedlikeholde kopier av dataen som er lagret i nettverket for å unngå tap av denne dataen hvis en peer blir koblet ut av nettverket.

Et nettverk med denne strukturen vil kunne tilby en distribuert metode for å kunne gjøre navneoppslag i en LS slik vi så i seksjon 2.2.1 på side 15.

Ved hjelp av en Distribuert Hash-tabell (DHT) med en godt tilpasset algoritme kan vi oppnå en effektiv plassering på lagret informasjon i P2P-nettverket.

Distribuert Hash-tabell (DHT)

En distribuert hash-tabell er en desentralisert variant av en vanlig hash-tabell. Forskjellen på en vanlig hash-tabell og en DHT er at den distribuerte er spredd over flere noder i et nettverk. Dette nettverket kalles et DHT-nettverk, og består av noder som ikke er koordinert fra en et sentralt hold. Uten koordinering fra et sentralt hold er nodene selv nødt til å samarbeide med hverandre for å oppnå den samme grad av koordinasjon som det vil være forventet av en vanlig hash-tabell.

DHT-nettverket vedlikeholdes ved å opplyse de andre nodene i DHT-nettverket om endringer, og de har et delt distribuert ansvar for at innhold i nettverket lagres riktig, og at dette innholdet er spredt jevnt utover DHT-nettverket.

DHT-nettverkets funksjonelle egenskaper

De funksjonelle egenskapene til DHT-en er av samme art som en vanlig hash-tabell med nøkkel- og verdi-par. Disse nøkkel- og verdi-parene distribueres, ideelt sett, jevnt blant alle nodene i DHT-nettverket på en effektiv måte [22].

Måten informasjon lagres i DHT-nettverket er ved hjelp av funksjonene `put(nøkkel,verdi)` og `get(nøkkel)`. Denne informasjonen kalles gjerne for et nøkkel-/verdi-par. Informasjonen (verdien) lagres i DHT-nettverket under en identifikator (nøkkelen) med kommandoen `put()`, og kan hentes ut fra DHT-nettverket ved bruk av funksjonen `get()`.

Det er tre karakteristiske egenskaper en DHT må ha:

- Desentralisert
  Det er nodene selv som styrer det distribuerte systemet, uten noen form for sentralisert koordinering.

- Skalerbart
  Systemet skal kunne opprettholde effektiviteten selv om antallet noder øker.

- Feil-toleranse
  Systemet skal kunne stoles på selv om noder forlater nettverket, ankommer eller på noe som helst vis feiler.

3.3 Filoverføring som et bruksområde

Det største bruksområdet for P2P er filoverføring. Ved å ta i bruk ressurser hos alle inkluderte noder har utvekslingen av filer både blitt lettere, og ofte mer effektivt. Vi skal i denne seksjonen se på enkelte applikasjoner og protokoller som har utmerket seg de siste årene, og forandret måten filer kan distribueres på.

Selv om ikke filoverføring er direkte tilknyttet telefoni over internett er det eksempler på hvordan mye av den samme tankegangen bak filoverføring ved hjelp av P2P kan overføres til telefoni over internett.

Det aller beste eksempelet på dette er hvordan applikasjonen «KaZaa» (se seksjon 3.3.4 på side 47) etter hvert ble videreutviklet til «Skype» (se seksjon 2.2.3 på side 19).

3.3.1 Napster

Napster [28] ble lansert i 1999, og var en tjenester for utveksling av MP3-filer. Tjenesten var basert på P2P, og hadde en applikasjon som lot brukere søke i andres musikk-samlinger, og laste ned materiale.

Selve arkitekturen i Napster bestod av servere som lagret indekseringer fra alle nodene tilkoblet nettverket, og klientene gjorde søk mot disse serverne for å få tilbake søkeresultater som fortalte hvor de kunne finne materialet de hadde søkt på [30]. Selve materialet kunne man så laste ned direkte fra andre klienter i nettverket [21, s. 13]. Lagringsplass og båndbredde var derfor distribuert blant klientene i nettverket, men på grunn av disse indekseringsserverne, som må kunne regnes som sentrale servere, var ikke Napster fullt og helt det man kan regne som et rent P2P-system, men et sentralisert P2P-nettverk (se seksjon 3.2.1 på side 35).

Popularitet

Napster ble raskt populært blant brukerne på grunn av den nye, og mye lettere måten de nå kunne laste ned musikk. Når Napsters popularitet var på topp var det 6 millioner brukere som benyttet tjenesten [21, s. 13]. Tjenestens levetid ble ikke langvarig, og i 2001 ble den stengt på grunn av anklager om at den bidro til spredning av opphavsbeskyttet materiale. Tjenesten hadde allikvel stor innflytelse på andre tjenester som fulgte etter Napster i måten filer kunne bli spredt via internett på en effektiv måte.

3.3.2 Gnutella

Gnutella [27] ble introdusert i 2001, og var et P2P-system som ikke hadde noen form for sentrale servere. Alle roller i nettverket var altså distribuerte. Gnutella ble i likhet med Napster brukt til å dele filer, men Gnutella var ikke begrenset til kun MP3-filer slik som Napster.

Noder som var tilkoblet hverandre ble ansett som naboer av hverandre. Ved hjelp av noders naboer, og naboers naboer ble det dannet et ustrukturert «overlay»-nettverk. Dette nettverket ble brukt til å søke etter innhold.

Søk

Søk etter innhold ble gjort ved å spørre sine nabo-noder på et propagerende vis. Nabo-noder spurte også sine naboer, og de spurte igjen sine naboer, og så videre. Når et søk ga resultater ble det i den opprinnelige protokollen sendt tilbake til søkeren gjennom den samme ruten søket hadde blitt sendt. Dette førte til mye ekstra trafikk over nettverket, og svekket skalerbarheten. I en nyere versjon av protokollen ble derfor søkeresultatene returnert tilbake til søkeren direkte, i stede for gjennom den samme ruten.

Etter at søkeren hadde mottatt resultater kunne den så starte å laste ned innholdet fra nodene som hadde gitt resultater. Denne filoverføringen gikk direkte mellom nodene, og ikke via nettverket.

Gnutella var et rent ustrukturert nettverk (se seksjon 3.2.1 på side 32), noe som medførte mye informasjonsflyt i nettverket. Dette gjorde hele systemet treigt, og skalerbarheten ble mindre god.

3.3.3 Freenet

Freenet [26] er en annen form for P2P enn systemene som tidligere er nevnt som fildelingssysterm.

Tjenestens hovedmål har hele tiden vært å kunne gi en mulighet til å publisere og konsumere informasjon på en distribuert og anonym måte. Dette målet har spesielt vært med tanke på borgere i land, eller under regimer med strenge sensureringslover.

I tillegg til anonymitet for både de som publiserer innhold og de som konsumerer innholdet har tjenesten også fire andre mål. 1) Ingen form for sentralisert kontroll eller administrasjon; 2) systemet må være robust mot feil i hardware og software; 3) systemet må kunne håndtere endringer på en effektiv måte, og 4) ytelsen må kunne måle seg med det eksisterende distribueringssystemet «World Wide Web».

Tjenesten har vært under utvikling siden 1999, og er helt og fullt et rent distribuert system (se seksjon 3.2.1 på side 32). Strukturen i nettverket som dannes har mange likheter med Gnutella sitt «overlay»-nettverk, men den største forskjellen er at man kan velge at klienten bare skal kobler seg til noder som allerede er kjent, og godkjent. Denne typen struktur, hvor man kun kobles opp til kjente noder, omtales som Friend-to-Friend (F2F), eller ett «Darknet». Kommunikasjonen i Freenet er kryptert, og blir rutet gjennom andre noder i nettverket. Dette gjør det vanskelig å finne ut hvem som ønsker å hente innhold, og også hva innholdet er.

Selve systemet kan lettest forklares som et distribuert system som fungerer på toppen av det eksisterende internett. Ved hjelp av en klient som kjører på datamaskinen åpnes det en port lokalt på datamaskinen. Hvis man åpner maskinens lokale adresse og porten Freenet lytter til i en nettleser blir man presentert enkelte tjenester som eksisterer på Freenet. Blandt disse tjenestene finner man muligheter for å publisere informasjon, og man finner også indekseringstjenester som hjelper til med å finne frem til informasjon som er lagret i nettverket. Informasjon publiseres gjerne i form av nettsider som i Freenet kalles for «freesites». Disse nettsidene er kun tilgjengelig gjennom Freenet, og ikke gjennom det tradisjonelle internett.

Hvordan lages nettverket

Når en node ankommer nettverket vil den velge seg en tilfeldig numerisk identifikator [29], og den vil koble seg opp til noder den allerede kjenner til. Noden som ankommer utveksler også sertifikater med de andre partene for å forsikre seg om at det ikke foregår noen form for avlyttning mellom dem («man in the middle»). Sertifikatet inneholder kontaktinformasjon til noden, og kryptografiske identifikatorer tilhørende nodene.

Den tilfeldige numeriske identifikatoren som noden velger når den ankommer nettverket plasserer noden i det som lettest kan visualiseres som et rutenett med to akser (X og Y). Plasseringen i dette rutenettet er ment å fortelle andre noder som skal rute trafikk gjennom eller til noden hvor den er plassert topologisk i nettverket. Denne informasjonen brukes så for å rute pakker så effektivt som mulig med tanke på at noder som er nære hverandre topologisk sett også blir plassert nære hverandre i rutenettet.

Siden dette er en tilfeldig verdi satt av noden, vil dette selvfølgelig ikke gi noen nytte når det kommer til ruting, og den sier ingenting om hvor noden er plassert i nettverkets topologi.

Det nodene gjør for å rette opp i dette er å bytte identifikatorer med andre noder, slik at nodene etter hvert plasseres nærmere relevant informasjon. Denne prosessen kalles for «switching algorithm» og optimaliserer plasseringene av nodene i rutenettet.

Denne prosessen pågår kontinuelig for å optimalisere så mye som mulig. Denne informasjonen kan nå brukes til rutinginformasjon. Ved hjelp av en «grådig»[2] ruting-algoritme, som alltid vil hoppe videre til noder som er nærmere målet sitt i rutenettet vil dette føre til mye kortere vei til destinasjonen enn de man ville fått med tilfeldige valgte verdiene.

Teorien om at dette vil fungere bygger på Stanley Milgrams [31] eksperimenter på 1960-tallet kjent som «small world»-eksperimentet, hvor han kom opp med teorien om at alle personer har gjennomsnitlig seks steg (gjennom relasjoner) til enhver annen, hvilken som helst person på jorden.

Publisering og konsumering

Informasjon som lagres i nettverket blir delt opp i segmenter, og hvert segment blir assosiert med en adresse-nøkkel.

Noden som ønsker å publisere denne informasjonen begynner å sende disse segmentene til sine nabo-noder, som igjen sprer disse videre til andre noder. Hvert segment blir håndtert individuelt, og blir spredt rundt til forskjellige noder i nettverket. En fil blir altså delt opp i segmenter og disse segmentene blir lagret rundt på forskjellige noder i nettverket.

Informasjon blir lagret i nettverket, og blir assosiert med en gitt adressenøkkel. Denne nøkkelen blir så bruk til å finne et sted i nettverket hvor den blir lagret. Når noen ønsker å få tak i denne informasjonen gjøres et søk i nettverket etter denne nøkkelen, og returneres deretter til søkeren gjennom den samme ruten søket tok.

[2] Algoritmen tar det første og beste

Innhold kan ikke slettes eller oppdateres direkte, men det går ann å publisere oppdatert eller nytt innhold med den samme nøkkelen. Dette er fordi systemet ønsker å tilby total anonymitet, og det er derfor ingen eierskap knyttet til innholdet som er lagret.

3.3.4 FastTrack

FastTrack er en P2P-protokoll som oppnådde stor popularitet rundt 2003, i etterkant av at Napster hadde blitt lagt ned. Tjeneste var til for å utveksle filer, og musikk-filer var spesielt populært. P2P-nettverket som blir dannet er et ustrukturert hybrid P2P-nettverk (se seksjon 3.2.1 på side 37).

Selve protokollen ble primært benyttet av appplikasjonen KaZaa, men det var også andre applikasjoner som etter hvert ble benyttet etter at enkelte greide å «reverse-engineere» den propritære protokollen.

På grunn av at protokollen er lukket, er det ikke så mye som er kjent når det gjelder hvordan den fungerer. Det er allikevel gjort en god del forskning fra flere parter [3, 32] for å finne ut om kommunikasjonen som foregår mellom nodene som benytter seg av tjenesten.

Grunnen til at det var interessant å vite hvordan tjenesten fungerte var at enkelte internett-operatører hadde en god del uidentifisert trafikk som de trodde kunne være fra tjenester slik som KaZaa, og andre lignende applikasjoner og protokoller. Ved å kartlegge funksjonaliteten til protokollen kunne også internett-opperatørene vurdere effekten av å «cache» noe av det mest populære innholdet i tjenesten.

Forskningen [3] viste at det var gode muligheter for at en løsning som «cachet» det mest populære innholdet ville komme internett-operatørene til gode. Ved å analysere logger fra en internett-operatør fant de ut at det foregikk trender på enkelte filer, og at 65% av trafikken ble brukt for å utveksle de 20% mest populære filene.

Hvordan nettverket fungerte

Som nevnt var protokollen lukket, men forskning og reverse-engineering har alikevel gitt et innblikk i kommunikasjonen mellom noder i nettverket.

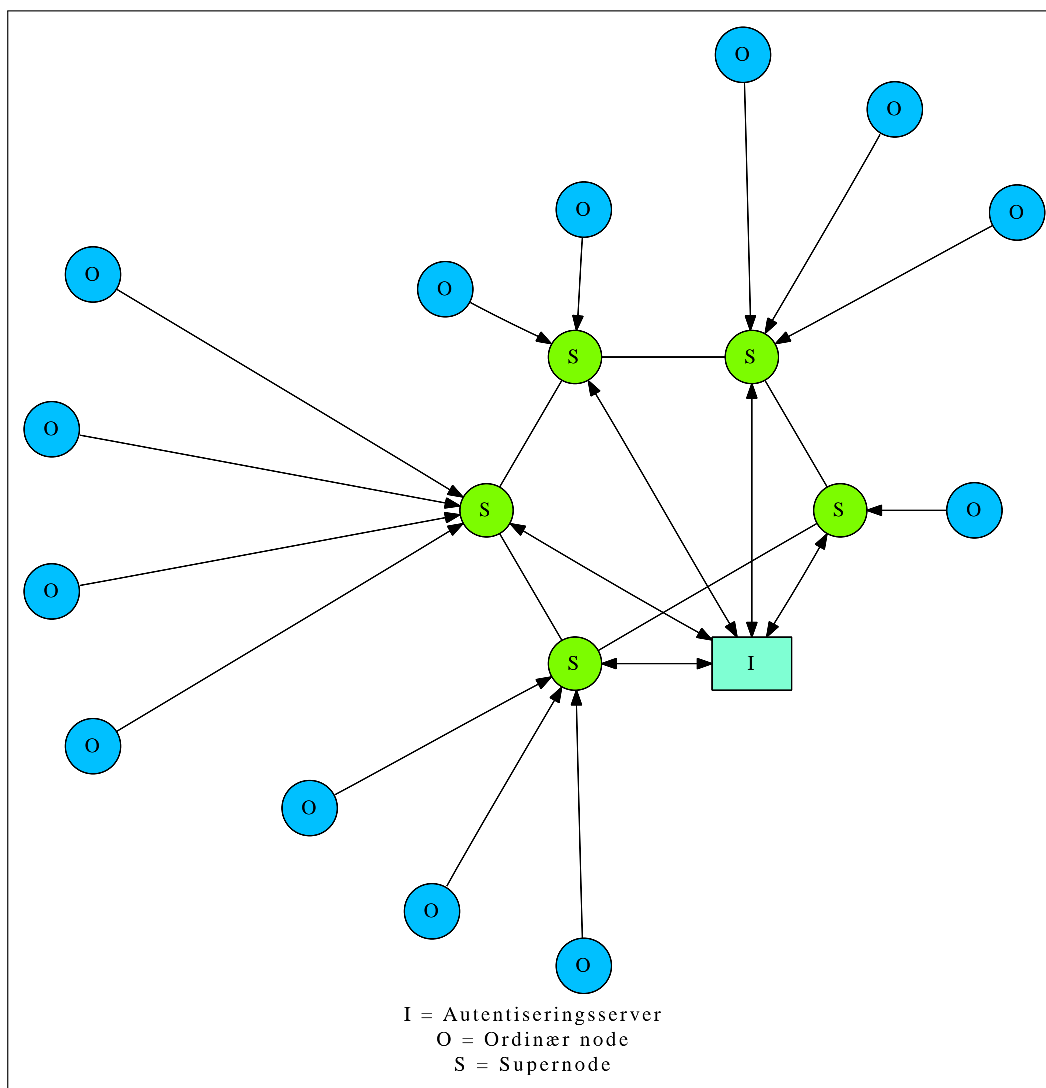
FastTrack er forløper, og også laget av de samme som utviklet Skype [2] (se seksjon 2.2.3 på side 19). Det er derfor grunn til å tro at mye av de samme teknikkene har blitt videreført til bruk i Skype. Det er derfor en del likheter mellom FastTrack og Skype.

Nettverket består av to typer noder. På samme måte som i Skype, er det også her en nodetype som man kan kalle ordinær, og en nodetype som har et større ansvar i nettverket som dannes, som kalles en super-node. Topologien i nettverket består av et nettverk med super-noder som er koblet sammen, og ut fra hver super-node er det ordinære noder som er tilknyttet i klaser (se figur 3.8 på neste side). Det er også i figuren illustrert en autentiserings-server (CA).

Kommunikasjonen som foregikk mellom ordinære noder og super-noder er det gjort en del forskning på. Måten super-noder kommuniserte med andre super-noder på er dessverre mindre kjent. Det som har blitt funnet ut er at super-nodene har ansvar for et utvalg ordinære noder. Super-nodene samler inn informasjon fra de ordinære nodene i form av indekseringer som forteller hvilke filer de ordinære nodene deler.

Denne indekseringen blir brukt for å gjøre søk i hele nettverket. Det er grunn til å tro at super-nodene gjorde søk på vegne av de ordinære nodene, ved å sende spørringer til andre super-noder, siden de ordinære nodene kun var koblet til nettverket via super-nodene, og kun hadde kontakt med andre noder når de lastet ned innhold.

Utvelgelse av hvilke noder som fikk et større ansvar i form av å bli super-noder ble gjort automatisk. For å være en kandidat til å bli super-node var det nødvendig at noden hadde rikelig med ressurser i form av båndbredde, prosesseringskraft, minne, og oppetid. Det var også en nødvendighet at noden ikke var på et lukket nett bak en NAT (3.1.2 på side 29).



Figur 3.8: Topologien i FastTrack

Svakhet i hash-algoritmen

Filer som ble utvekslet over FastTrack benyttet seg av en hash-algoritme for å kontrollere at filen var blitt nedlastet riktig. Denne hash-algoritmen viste seg å være for svak, slik at det lett var mulig å skape en hash-kollisjon. Etter hvert ble det derfor utført angrep mot tjenesten ved å legge ut filer som utnyttet svakheten i hash-algoritmen [33]. Dette korrupterte de nedlastede filene.

3.3.5 BitTorrent

BitTorrent [34] er en P2P fildelingsprotokoll som kan brukes til å distribuere store mengder data.

Distribusjon ved hjelp av BitTorrent gjør det mulig å distribuere store mengder data uten at den initielle deleren vil bli overbelastet. Hvis en fil, eksempelvis, legges ut på en tradisjonell webserver, og man vet at denne filen vil bli lastet ned av 100 klienter, vet vi også at denne filen må bli sendt 100 ganger fra webserveren og til hver av klientene som ønsker filen. Den totale mengden data denne webserveren da må sende ut tilsvarer 100 ganger størrelsen på filen. Hvis man derimot benytter seg av BitTorrent-teknologi vil denne filen i teorien kun trenge å bli lastet ned én gang, fordi resten av klientene vil sørge for distribusjon mellom seg til alle de andre klientene.

Hvordan fungerer BitTorrent

Det første som skjer når en mengde data skal deles over et BitTorrent-nettverk er at datamengden blir gjort tilgjengelig av en «seeder» for andre deltagere i nettverket. Disse deltagerene kalles «peers». BitTorrent baserer seg på en gi- og ta-tankegang, hvor de som gir blir belønnet, og de som kun tar blir straffet.

Publisering av torrent-filer

Selve datamengden som skal deles med andre «peers» blir først delt opp i segmenter. Størrelsen på alle disse segmentene er like. Hver av segmentene blir så tatt igjennom en prosess hvor de får hver sin hash-verdi. Denne verdien blir lagret i en «torrent»-fil. Torrent-filen blir gjerne lastet opp på en web-side, eller spredt via andre tjenester. Disse torrent-filene inneholder nok informasjon om den totale datamengden til at «peers» kan kontrollere om de har mottat riktig data (ved hjelp

av hash-verdiene). Torrent-filene inneholder også informasjon om hvilke «trackere» som kan benyttes for å få tak i disse posjonene. Hvordan «trackere» fungerer blir beskrevet mer detaljert i 3.3.5 på neste side.

Deling av segmenter

Deltagere («peers») sender så forespørsler til «seedere» om å få tilsendt små segmenter av datamengden. Den første «seederen» begynner så å sende ut de første segmentene av datamengden til resten av deltagerene.

Så snart deltagerene mottar segmenter av datamengden vil disse igjen starte å dele segmenter med andre deltagere i nettverket. Deltagere spør hverandre hvilke segmenter de har tilgjengelig. Deltagerne vil så sende forespørsler om å bli tilsendt segmentene fra deltagerne de nå vet har segmentene de er ute etter. Det er også muligheter for Peer Exchange (PEX) i enkelte BitTorrent-klienter. Dette gjøres for å utveksle informasjon, «peers» i mellom, om andre «peers» som også er i nettverket. Dette gjør klientene så de har flere andre deltagere å spørre om segmenter. Deltagerne får også denne typen informasjon av «trackeren».

Flere og flere deltagere i nettverket vil etter hvert få store mengder av den totale datamengden, og deltagerene vil da dele disse segmentene med resten av nettverket. Belastningen på deltagerene som har segmenter av datamengden vil da etter hvert jevne seg ut, siden flere og flere deler de samme segmentene.

For å få en jevn og rettferdig belastning på så mange «peers» som mulig har BitTorrent noen innebygde funksjoner som skal hjelpe til i denne prosessen. Den ene innebygde funksjonen i protokollen er å belønne de som deler. Protokollen sørger for at mengden data deltageren mottar er så proposjonal som mulig i forhold til mengden data deltageren deler. Dette utføres ved at de forskjellige deltagerne i nettverket bytter segmenter med hverandre, som en slags forhandling hvor man bytter en mot en.

Denne funksjonaliteten kan skape problemer hvis veldig mange har de

samme posjonene, siden deltagerene i nettverket da ikke har segmenter de kan forhandle med for å få andre segmenter, og man får da rett og slett ikke utnyttet opplastningsegenskapene til alle deltagerne. Denne situasjonen kan kvele nettverket, siden det da ikke utveksles data mellom deltagerene.

Dette løses ved en annen innebygget funksjonalitet i BitTorrent-protokollen. Denne funksjonen er ansvarlig for å velge hvilke segmenter deltagerne i nettverket skal sende forespørsler om.

Valg av segmenter

Det aller viktigste for protokollen er å sørge for at påbegynte segmenter fullføres før den starter på nye segmenter. Dette gjør protokollen for å kunne dele disse segmentene videre med andre deltagere i nettverket, og for at nedlastningen for deltageren skal gå så fort som mulig.

En annen ting protokollen gjør er å sende forespørsler på de segmentene som er spredt minst i selve nettverket. Altså de segmentene som færrest av deltagerne i nettverket har. Effekten av dette er at deltageren nå mottar posjoner som andre i nettverket ønsker. I tillegg til at deltagerne får segmenter andre i nettverket ønsker, øker dette også sjansen for at hele den totale datamenden vil bli spredt ut til alle nodene. Dette er fordi ingen av deltagerne kan få tak i hele den totale datamengden uten at den orginale «seederen» har lastet opp alle segmenter i datamengden minst en gang. Det vil derfor være nyttig å få spredt de skjeldne segmentene først, slik at sansynligheten for at disse segmentene fortsatt vil være i nettverket selv om «seederen» forsvinner.

Hvordan dannes nettverket

BitTorrent-nettverket dannes etter at «peers» har lastet ned en «torrent»-fil som gir dem informasjon om hvor de skal koble seg opp. Stedet «peers» kobler seg opp til kalles en «tracker». Trackeren holder oversikt over

hvilke «seedere» som har hvilke segmenter, og deler denne informasjonen videre til «peers» som ønsker å laste ned disse. Etter hvert som «peers» selv har lastet ned disse posjonene melder de fra til trackeren om at de nå også deler disse posjonene. I tillegg til trackere gjøres det også, som tidligere nevnt, PEX mellom «peers» i nettverket.

Distribuerte Trackers

Som en forbedring av BitTorrent har det i senere tid blitt tilrettelagt for en distribuert måte å danne BitTorrent-nettverk på, uten bruk av trackere. Enkelte BitTorrent-klienter introduserte mot midten av 2005 implementasjoner av DHT-er som tilrettela for bruk av BitTorrent uten vanlige trackere. I disse implementasjonene er det selv «peers» i nettverket som opptrer som trackere for hverandre.

Svakheter

BitTorrent har vist seg å være et godt alternativ til hvordan man kan overføre filer på en effektiv måte som fordeler distribusjonskostnadene mellom alle deltagerne i nettverket, i stede for å legge alle disse kostnadene på den opprinnelige deleren. Det som derimot er et problem er at interessen for å dele videre med resten av nettverket ofte forsvinner så snart en «peer» har lastet ned hele den totale datamengden. Det som ofte skjer i slike tilfeller er at mange av segmentene som er nødvendig for å få lastet ned hele den totale datamengden ikke lengre finnes i BitTorrent-nettverket. Dette fordi alle «seedere» med disse segmentene har koblet seg ut av nettverket. På et vis blir da BitTorrent kun nyttig for populære filer, samtidig som det i mange tilfeller blir umulig å få tak i den totale datamengden fra eldre BitTorrent-filer.

Manglende anonymitet

En annen svakhet i BitTorrent er at brukerne ikke anonymiseres. Når en «peer» sender en forespørsel til en «tracker» vil den få tilsendt lister med IP-adresser til andre «peers» og «seedere» i nettverket. IP-adresser til «peers» spres også ved hjelp av PEX. Dette er selvfølgelig nødvendig for at protokollen skal fungere som den gjør, siden «peers» er nødt til å vite hvor de kan laste ned segmenter fra.

Det er en kjent sak at BitTorrent ikke bare brukes til lovlig distribuering av filer, men også ulovlig distribuering av opphavsbeskyttet materiale. Denne mangelen på anonymisering kan derfor føre til at rettighetshavere, eller representanter for rettighetshaverne kan sanke inn IP-adresser til deltagerne i slike ulovlige BitTorrent-nettverk.

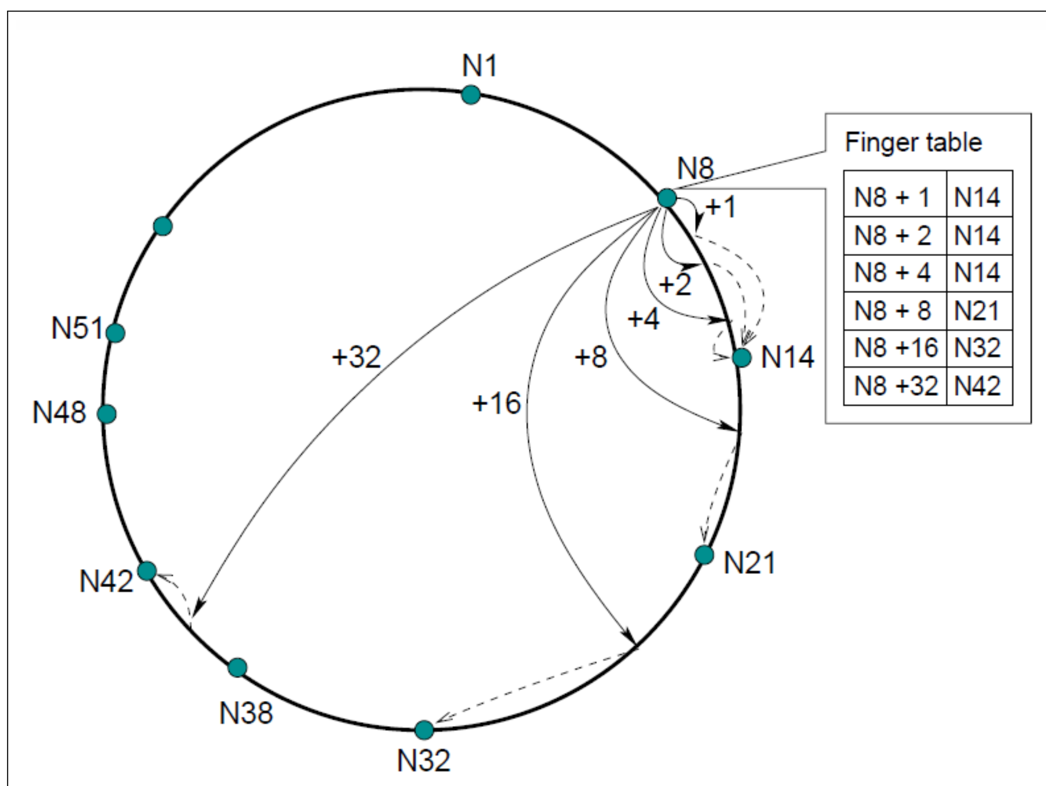
3.4 Implementasjon av strukturerte P2P-nettverk

Vi skal i denne seksjonen se på en implementasjon av et strukturet P2P-nettverk kalt Chord [35].

3.4.1 Chord

Chord [35] er en protokoll som danner et «overlay»-nettverk for de tilknyttede nodene i nettverket. Selve Chord-nettverket kan visualiseres som en sirkel som omfatter alle nodene i nettverket, med strenger (chords) som går på tvers av sirkelen. Dette er illustrert i figur 3.9 på neste side (figuren er lånt fra [35, s. 5]).

I Chord brukes US Secure Hash Algorithm 1 (SHA-1) [36] som en konsistent hash-funksjon for å tildele hver node i nettverket en unik identifikator. Denne identifikatoren vil være et tall fra 0 til $2^m - 1$, hvor



Figur 3.9: Chord fingertabell

m er antall bits identifikatoren representeres med. Nodens identifikator er «hashen» av nodens IP-adresse.

Tallet m angir også størrelsen på sirkelen, eller antall noder det maksimalt kan være. En Chord-ring med m bits identifikatorer vil kunne ha opp til 2^m noder i nettverket. Tallet m bør derfor være stort nok slik at ingen noder vil bli tildelt den samme identifikatoren. Figur 3.9 viser en sirkel med størrelse $m = 6$. Det er altså plass til 64 noder i dette nettverket.

Alle nodene i ringen har en «successor». «Successoren» til en node er den etterfølgende noden i sirkelen. I figur 3.9 ville node *N8* sin «successor» vært node *N14*. Alle nodene i nettverket vet om sin «successor» og har en adresse-peker til denne noden. Nodene i nettverket har også en «fingertabell». Denne tabellen inneholder m antall adresse-pekere til andre noder. Dette er en oppslagstabell som nodene bruker til rask routing

i sirkelen. Slik det er illustrert i figur 3.9 på forrige side har node *N8* en «fingertabell» med 6 rader. Adresse-pekerne noden har i sin «fingertabell» er relative i forhold til nodens identifikator ($nodeID$), og er følgende; $nodeID + 2^0, nodeID + 2^1, nodeID + 2^2, ..., nodeID + 2^{m-1}$.

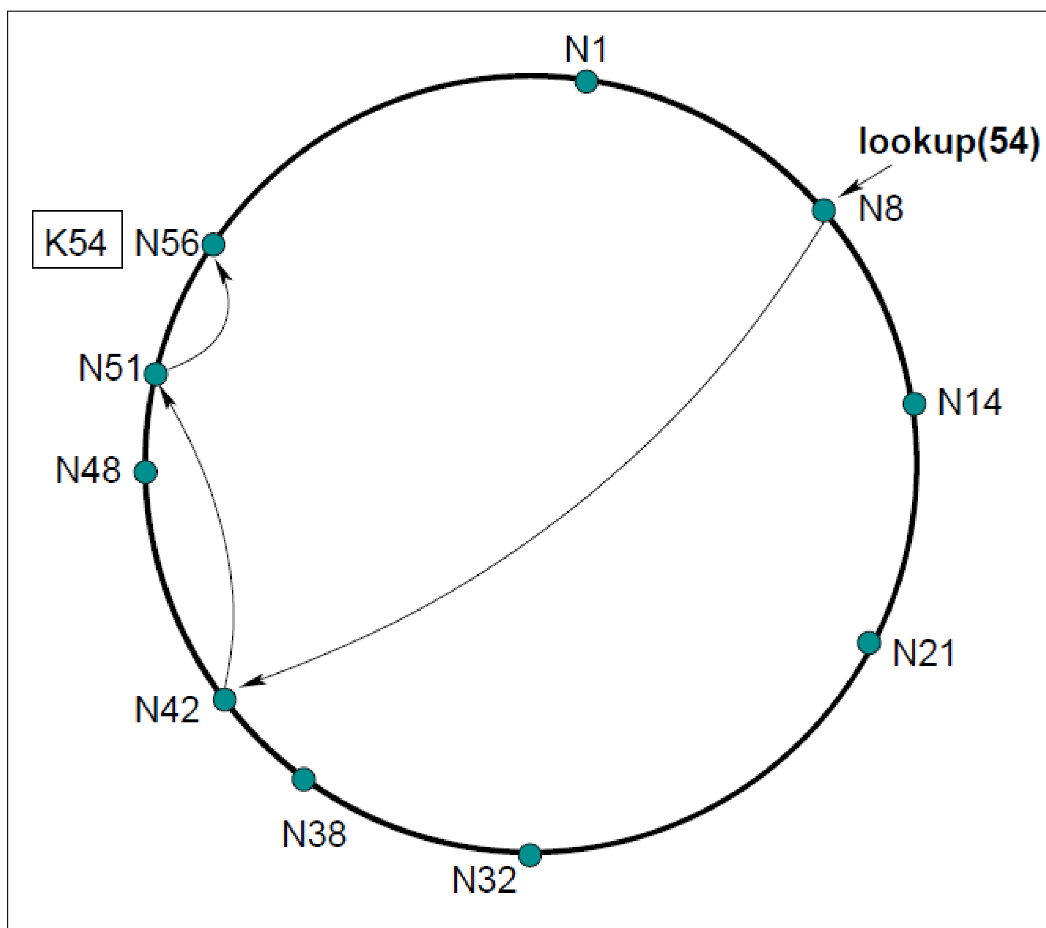
For node $N8$ i figur 3.9 på forrige side skulle dette da egentlig gitt følgende «fingertabell»; $N9$, $N10$, $N12$, $N16$, $N24$ og $N40$. Det Chord gjør for å rette opp i dette er å velge den første noden i sirkelen som har en høyere node-identifikator enn beregningen, slik at verdiene blir som i figuren.

Slik vi så i seksjon 3.2.2 på side 41 tilbyr en DHT gjerne funksjonene `put(nøkkel,verdi)` og `get(verdi)`. I Chord benyttes en funksjon `lookup(nøkkel)` for å finne ut hvilken node som er ansvarlig for denne nøkkelen, så det er da ikke Chord direkte som gir til verdiene, men Chord forteller hvilken node verdien ligger på. I [37] står det beskrevet hvordan Chord ble benyttet for å lage en DHT.

Noder i Chord er ansvarlig for tupler (tilsvarende nøkkel/verdi-par) av typen `(nøkkel,verdi)`. Disse tuplene lagres på noden med identifikator `hash(nøkkel)`, hvor `hash()` er SHA-1-funksjonen. Noder i nettverket er også ansvarlig for alle nøkler med en hash-verdi som er mindre enn sin egen node-identifikator, men større enn forrige noden i sirkelen sin identifikator. I figur 3.9 på forrige side ville da node $N42$ vært ansvarlig for nøklene; $K42$, $K41$, $K40$ og $K39$. Nodene i nettverket dekker altså nøkkel-området helt tilbake til forrige node i sirkelen.

Til funksjonen `lookup(nøkkel)` benyttes «fingertabellene» til nodene for å finne noden som er ansvarlig for nøkkelen. I figur 3.10 på neste side (figuren er lånt fra [35, s. 5]) er det et eksempel hvor node $N8$ ønsker å finne nøkkel $K54$. I listen under er en forklaring av figuren:

- $N8$ ser i sin fingertabell om den har $N54$. Det har den ikke.
- $N8$ sjekker fingertabellen igjen etter noden som har høyest mulig identifikator, men samtidig mindre eller lik 56. $N8$ finner $N42$ og sender forespørselen dit.



Figur 3.10: Chord lookup

- $N42$ har $N50$ i sin fingertabell, men det er $N51$ som er ansvarlig for denne verdien når $N50$ ikke er tilstede. $N42$ har også $N58$ i sin fingertabell, som i figuren kontrolleres av $N1$. $N51$ er derfor den nærmeste noden. $N42$ videresender forespørselen dit.
- Den nærmeste noden $N51$ har i sin fingertabell som ikke har høyere verdi enn 54 er $N56$.
- $N56$ er ansvarlig for alle nøkler mellom 52 og 56. Vi er derfor i mål.

En enklere, men mindre effektiv, måte for å finne den ansvarlige noden hadde vært om nodene hadde videresendt forespørselen til sin «successor», som igjen ville sendt til sin «successor», osv. Med det antallet

noder som det befinner seg i nettverket i figuren, ville denne måten krevd åtte videresendinger for å nå det samme målet. Ved å benytte seg av «fingertabellen» greide Chord den samme oppgaven på tre steg, slik vi så i figuren.

En av de største fordelene med Chord er bruken av «fingertabeller». Måten Chord benytter disse på fører til at rutingen sørger for å «hoppe» minst halvparten av den gjenstående distansen til destinasjonen for hvert «hopp». Dette gjør at søketiden kun øker logaritmisk i forhold til noder i nettverket, noe som fører til svært god skalerbarhet [35, s. 6]. Dette fører også til mye mindre informasjonsflyt rundt i nettverket slik jeg forklarte i seksjon 3.1.2 på side 28.

3.5 Oppsummering

Vi har i dette kapittelet sett på noen av bruksområdene hvor P2P har blitt benyttet. Vi har også sett at enkelte protokoller har tatt steg i nye retninger, og at de har blitt mindre avhengig av sentrale styringsenheter. Vi har også sett på protokollen FastTrack som har blitt videreutviklet av Skype og har gått fra å være en filoverførings-protokoll til å bli telefoni over internett.

Vi har også sett på hvilke arkitekturer og topologier i P2P-nettverk kan deles inn i kategoriene strukturerte og ustrukturerte P2P-nettverk. Vi har også sett fordeler og ulemper med disse.

Kapittel 4

Tidligere arbeid

Til nå har jeg presentert hvordan telefoni over internett (se kapittel 2 på side 5) tradisjonelt har fungert med VoIP, først og fremst ved hjelp av protokollen SIP.

Jeg har også presentert hva P2P er (se kapittel 3 på side 23), fordeler med P2P, den arkitekturiske og topologiske oppbygningen, og hvilke bruksområder P2P har hatt.

I dette kapittelet presenterer jeg hvordan disse to temaene kan kombineres ved å vise til arbeid som er gjort tidligere.

4.1 REsource LOcation And Discovery (RELOAD)

I 2005 etablerte IETF en arbeidsgruppe kalt P2PSIP Working Group (P2PSIP WG) [22]. Målet med arbeidsgruppen er å lage en standard for SIP-kommunikasjon over P2P-nettverk, altså Peer-to-Peer Session Initiation Protocol (P2PSIP). Det var spesielt to prosjekter som dannet grunnlaget for arbeidsgruppen, og dette var SIPpeer [38], og SOSIMPLE [39]. Begge disse prosjektene hadde en ganske lik tilnærming til problemet.

Arbeidsgruppen holder fortsatt på med standardiseringen, og de siste utkastene på en standard som heter REsource LOcation And Discovery (RELOAD) [4] er nå i siste fase av standardiseringsprosessen.

Det er definert flere bruksområder til denne nye protokollen, men jeg kommer i denne presentasjonen av RELOAD til å fokusere på bruksområdet SIP [40]. I dette bruksområdet er målet å kunne tilby en P2P telefonitjeneste som fungerer uten noen form for permanente proxyer eller registrarer.

4.1.1 P2PSIP-overlay

P2PSIP-overlayet i RELOAD består av noder, som omtales som «peers», som er organisert i form av et strukturert P2P-nettverk. Formålet med overlayet i RELOAD er å muliggjøre «real-time»-kommunikasjon ved hjelp av SIP.

Nodenes oppgave i P2PSIP-overlayet er å kollektivt kunne tilby en tjeneste i form av en distribuert database. Denne distribuerte databasen skal kunne brukes til navneoppslag for å kunne gjøre AoR-mapping. Denne tjenesten tilsvarer da LS-en sin oppgave i SIP (se seksjon 2.2.1 på side 14). Alle noder i overlay-nettverket blir tildelt en unik identifikator.

RELOAD er designet slik at det er mulig å bruke en hvilken som helst algoritme for overlay-nettverket, men i forslaget til den nye standarden er det spesifisert at Chord, som jeg omtalte i seksjon 3.4.1 på side 54, er obligatorisk å implementere.

4.1.2 Navneoppslag

I tradisjonell SIP er det SIP-registraren sin oppgave å ta imot `REGISTER`-forespørsler fra SIP-UA-er, og lagre AoR-en i en LS. I RELOAD blir denne oppgaven gjort av overlay-nettverket. Noder kan i RELOAD både lagre og hente ut «mappinger» av AoR ved hjelp av overlay-nettverket.

Forskjellen fra SIP er at det i RELOAD sin tjeneste vil bli returnert noders identifikatorer i stede for fulle nettverksadresser med IP-adresse og port. En «mappinger» kan for eksempel se slik ut:

- `'alice@wonderland.com'` → `'2011'`
- `'alice@wonderland.com'` → `'0801'`

Verdien '2011' kan for eksempel være identifikatoren til Alice sin node hjemme, og '0801' kan være Alice sin node på kontoret.

Det er også mulig å lagre en videresendings-adresse i overlayet. For å gjøre dette lagrer Alice følgende AoR:

- `'alice@wonderland.com'` → `'bob@wonderland.com'`

Denne «mappingen» returnerer ingen node-identifikator, men i stede en annen adresse. For å oppnå kontakt med Alice, må man derfor gjøre et nytt søk i overlay-nettverket etter den nye adressen, `'bob@wonderland.com'`.

4.1.3 Opprette samtaler i RELOAD

Hvis Bob ønsker å ringe Alice ved hjelp av RELOAD gjøres følgende:

- Alice registrerer seg P2PSIP-overlayet med adressen `'alice@wonderland.com'`. Alice har node-identifikator `'2011'`.
- Bob gjør et navneoppslag i P2PSIP-overlayet på `'alice@wonderland.com'`.
- Bob får tilbake fra overlayet at Alice er registrert med node-identifikator `'2011'`.
- Bob benytter overlayet til å sende en `AppAttach`-melding til Alice sin node.
- Meldingen går igjennom overlay-nettverket og når Alice. Alice svarer tilbake til Bob med `AppAttach`.

- `AppAttach` mottas av Bob.
- ICE [24] benyttes for å sørge for tilkobling mellom Alice og Bob selv om de er bak en NAT eller en brannmur.
- Det er nå opprettet en forbindelse mellom Alice og Bob.
- RELOAD har gjort sitt.
- SIP tar over, og Bob sender `INVITE` til Alice, som svarer tilbake.
- SIP-sesjonen er i gang.

Som vi ser i eksempelet, er `AppAttach` [4, s. 32] noe som ikke benyttes i vanlig SIP. Dette er en måte å opprette en forbindelse mellom to eller flere noder. Tilkoblingen som opprettes er en dedikert TCP- eller UDP-tilkobling som ikke går over overlay-nettverket. Vi ser også i eksempelet at ICE [24] benyttes. Dette gjøres fordi RELOAD antar at mange noder befinner seg bak NAT-er eller brannmurer [4, s. 7].

4.1.4 Robusthet

Den nye signaleringsprotokollen RELOAD sikter på å kunne tilby en tjeneste uten sentrale enheter slik som permanente proxyer og registrarer. Ved å overlate disse tjenestene til det strukturerte overlay-nettverket, da spesielt med tanke på navneoppslags-tjenesten og opprettelse av forbindelser, vil RELOAD kunne tilby en tjeneste som er robust.

Som jeg nevte i seksjon 4.1.3 på forrige side hjelper overlay-nettverket til med å opprette forbindelser mellom parter, og sørger for at forbindelsen opprettes selv om noden er bak en NAT eller brannmur. Til denne oppgaven benyttes ICE [24].

Overlay-nettverket i seg selv er også noe som bidrar til robusthet. Informasjon som lagres i overlay-nettverket vil være spredt ut blandt alle nodene i nettverket på grunn av hashing-algoritmen som benyttes av

Chord (se seksjon 3.4.1 på side 54), noe som vil utelukke «single points of failures».

4.1.5 Skalerbarhet

Skalerbarheten i RELOAD er i stor grad avhengig av algoritmen som kjøres i overlay-nettverket. RELOAD er, som jeg nevnte, designet slik at det skal være mulig å benytte en hvilken som helst algoritme for overlay-nettverket. Det ser også spesifisert at Chord er obligatorisk. Som vi så i seksjon 3.4.1 på side 54, har Chord en svært god skaleringsevne.

4.2 Oppsummering

Vi har i dette kapittelet sett en gjennomgang av den pågående standardiseringen av den nye signaleringsprotokollen RELOAD. Vi har med dette sett hvordan det er mulig å kombinere SIP med P2P for å oppnå en mer robust og skalerbar tjeneste. Dette viser at industrien er langt på vei med å innføre distribuerte VoIP-tjenester, ved å lage en felles standard for dette.

Kapittel 5

Implementasjon

Et av målene med denne oppgaven har vært å se på måter P2P har blitt brukt til å utøre forskjellige tjenester. Vi har allerede sett at tjenesten KaZaa/FastTrack (3.3.4 på side 47), som ble brukt til filoverføring, tok steget videre inn i et helt annet bruksområdet og ble til Skype (2.2.3 på side 19).

Målet med prototypen har hele tiden vært å kunne gjøre en SIP-klient mindre avhengig av en SIP-server for å gjøre den robust og skalerbar. Det har også vært viktig å kunne bevare kompatibiliteten med tradisjonell SIP-arkitektur, slik at den nye løsningen kan fungere side ved side med eksisterende løsninger.

I første del av dette kapittelet forklarer jeg formålet med prototypen. Videre gir jeg en detaljert beskrivelse av alle applikasjoner og rammeverk som har blitt benyttet for å oppnå et fungerende «proof of concept» for denne oppgaven.

I del tre av dette kapittelet skal vi se på hvilke endringer som ble gjort på de valgte applikasjonene og rammeverkene. Det vil bli vist kode-eksempler og hvordan hver enkelt del av prototypen fungerer.

I fjerde del skal vi se et eksempel på hvordan prototypen fungerte i praksis mellom to brukere. Videre i denne delen presenterer jeg en evaluering av

prototypen med erfaringer og funn.

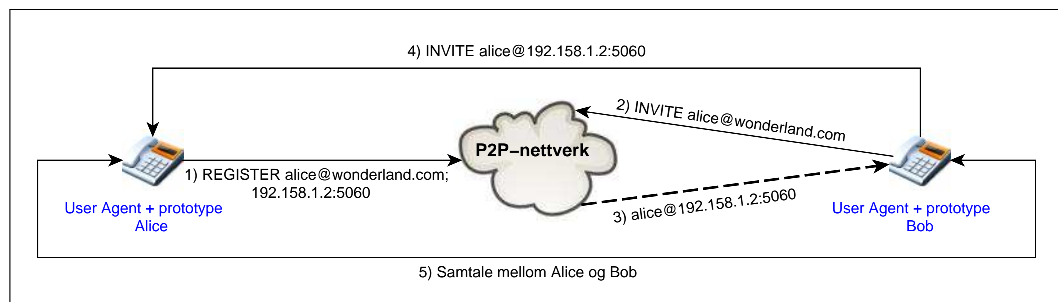
5.1 Formålet med prototypen

Som jeg nevnte innledningsvis i dette kapittelet ønsket jeg med min implementasjon å gjøre en SIP-klient mindre avhengig, eller helt uavhengig av en SIP-server for å gjøre dem robust og skalerbar.

I tillegg til dette ønsker jeg at prototypen skal kunne fungere på flere platformer, og at brukeren skal kunne velge SIP-klient fritt. Prototypen har derfor som mål i første omgang å fungere på Linux og Windows uavhengig av SIP-klient.

Med denne valgfriheten i valg av SIP-klient følger det også et krav om bakoverkompatiblitet. Det skal ikke være nødvendig med noen form for ekstra konfigurasjon av SIP-klienten utover det som er vanlig ved bruk av tradisjonell SIP.

I figur 5.1 er det illustrert hvordan Alice og Bob kan kommunisere med hverandre ved hjelp av prototypen. Som jeg også nevner i seksjon 5.3 på side 73 er dette kun et «proof of concept», og derfor skal SIP-UA-ene og prototypen anses som én enhet. SIP-UA-ene i figuren er derfor illustrert som én enhet, og ikke en SIP-UA som er tilkoblet prototypen, som igjen er tilkoblet P2P-nettverket.



Figur 5.1: Samtale mellom Alice og Bob med P2PSIP

5.2 Applikasjoner og rammeverk

5.2.1 VoIP-klienter

Til implementasjonen var det nødvendig med en SIP-klient som var lett anvendelig, lett å konfigurerer, og som støttet de mest elementære protokollene. Jeg valgte til slutt å benytte meg av to forskjellige SIP-klienter for å kontrollere at implementasjonen fungerte uavhengig av Operativsystem (OS)-platform og SIP-klient.

Tabell 5.2.1 på neste side inneholder en liste med spesifikasjonene til de to valgte SIP-klientene «PhonerLite» [41] og «Ekiga» [42].

PhonerLite

«PhonerLite» [41] er en liten og lett VoIP-klient laget for Microsoft Windows [43] med støtte for SIP. PhoneLite støtter mediaoverføringsprotokollene RTP og Secure Real-time Transport Protocol (SRTP), STUN [18], i tillegg til IM.

«PhonerLite» er basert på den samme kode-basen som «storebroren» «Phoner». Begge versjonene er laget av samme produsent, men «Phoner» har et annet grafisk brukergrensesnitt (GUI), og har støtte for CAPI [44] og TAPI [45] som muliggjør tilkobling av PBX-er, slik at PC-en kan benyttes som en PSTN-telefon i tillegg til VoIP. I min implementasjon var det ikke nødvendig med de ekstra tjenestene Phoner tilbød, fremfor PhonerLite, så valget ble derfor PhonerLite.

I forbindelse med testing av implementasjonen ble versjon 1.86 av programvaren benyttet, og OS-et som ble benyttet var «Microsoft Windows 7 Professional version 6.1 (Service Pack 1)».

	PhonerLite	Ekiga
Protokoller		
Session Initiation Protocol (SIP) [1]	✓	✓
H.323 [16]	✗	✓
Session Traversal Utilities for NAT (STUN) [18]	✓	✓
Operativsystem (OS)		
Windows	✓	✓
Linux	✗	✓
Solaris	✗	✓
Kryptering		
TLS [46]	✓	✗
SRTP [47]	✓	✗
ZRTP [48]	✓	✗
Andre tjenester		
Lightweight Directory Access Protocol (LDAP)		✓
Konferanse-samtale	✓	
Opptak av samtale	✓	
Video-samtale	✗	✓
Videresending av samtale	✓	✓
Overføring av samtale		✓
Instant Messages (IM)	✓	✓

Tabell 5.1: Spesifikasjoner på SIP-klientene «PhonerLite» og «Ekiga»

Ekiga

«Ekiga» [42] er en VoIP-klient med en åpen kildekode. Den har støtte for SIP- og H.323-sesjoner, videokonferanse og IM. Klienten støtter ingen form for kryptering, slik som PhonerLite, men det har heller ikke vært aktuelt, og heller ingen nødvendighet, i forbindelse med implementasjonen.

I forbindelse med testing av implementasjonen ble versjon 2.0.2 av programvaren benyttet, og OS-et som ble benyttet var «Red Hat Enterprise Linux Client release 5.7 (Tikanga)».

5.2.2 39 Peers

I implementasjonen har jeg tatt utgangspunkt i et eksisterende rammeverk skrevet i Python[1] kalt «39 Peers» [49]. Dette rammeverket er laget av Kundan N. Singh, som også har skrevet en rekke publikasjoner [38, 50] om VoIP, P2P, og kombinasjonen av disse.

Rammeverket, og hele prosjektet rundt, er laget spesielt med tanke på at det skal være et verktøy for studenter og forskere for å eksperimenter med nye ideer. Målet for prosjetet er å lage en fungerende implementasjon av SIP i kombinasjon med P2P, altså P2PSIP.

Selve kildekoden til prosjektet er oppdelt i to kategorier. Jeg skal i de to neste avsnittene oppsummere kort de to kategoriene, standarder og applikasjoner.

Standarder i 39 Peers

Standarder som er inkludert i prosjektet er først og fremst RFC-er. De forskjellige standardene som er implementert i prosjektet er svært godt dokumentert, og følger RFC-ene nøye. Dette gjøres ved å legge ved sidetall og de aktuelle avsnittene fra RFC-ene inn i den tilhørende kildekoden som kommentarer.

Enkelte av RFC-ene det finnes kildekode av er ikke helt komplette, men de aller viktigste funksjonene er tilstede. De mest grunnleggende standardene det finnes kildekode på er RFC 3261 [1] som er standarden for SIP, RFC 3264 og 4566 som er standarder for SDP [12, 13], og RFC 3550 som beskriver standarden for RTP og RTCP. I kildekoden for RFC 3261 er det kun UA-delen av standarden som er inkludert.

Disse standardene, og noen flere som er utelatt, blir benyttet som moduler i de tilhørende applikasjonene (neste avsnitt) til prosjektet.

[1] Python versjon 2.6.5

Applikasjoner i 39 Peers

Prosjektet inneholder en rekke applikasjoner klare til bruk. Disse baserer seg på, og benytter, i stor grad standardene som ble nevnt i forrige avsnitt som moduler. Dette gjøres ved at standardene importeres som moduler i kildekoden, slik at funksjoner som er definert og implementert kan benyttes av applikasjonene.

VoIP-klient

Det er laget en implementasjon av en VoIP-klient, som f.eks. kan benyttes som et grunnlag for en mer avansert SIP-UA. Den er i stand til å registrer ser hos registrarer, motta `INVITE`-forespørsler, benytte seg av IM, og konferansesamtaler. Dette applikasjonen kunne fint blitt brukt som grunnlag til en fullt fungerende UA, men jeg har i min prototype valgt å benytte meg av en allerede eksisterende og velfungerende VoIP-klient til testingen (se seksjon 5.2.1 på side 67).

SIP-server

Ved hjelp av et Application programming interface (API) som er en del av prosjektet er det laget en SIP-server. Denne SIP-serveren har funksjonalitet for å ta i mot `REGISTER`-forespørsler slik at den kan opptre som en registrar. SIP-serveren kan også opptre som en proxy og en redirect-server.

SIP-serveren aksepterer enkelte parametere ved oppstart. Disse parameterne gjør at man kan velge hvilken nettverksport serveren skal lytte til, og om SIP-serveren kun skal benyttes lokalt på maskinen den kjøres fra. Det er også mulig å angi restriksjoner på hvilke domener registrar-serveren skal akseptere, og i tillegg til dette er det også muligheter for å kjøre denne serveren sammen med flere tilsvarende servere for å kunne lastbalansere serverene seg i mellom.

SIP-UA-er som registerer seg på SIP-serveren blir lagret i LS-en som er en del av registrar-funksjonaliteten. LS-en i SIP-serveren er ikke særlig avansert, og lagrer navneoppslagene i minnet. I prototypen er denne funksjonaliteten endret slik at navneoppslagene i stede blir sendt til en DHT. Detaljene rundt disse endringene blir gjennomgått i seksjon 5.3.1 på side 73.

P2P

Det er også laget en god del moduler som hjelper til med P2P-delen av prosjektet. Blant annet er det laget en variant av en DHT basert på Bamboo og Pastry [51]. Denne modulen har også en tilhørende GUI-del som muliggjør en visuell fremstilling av DHT-nettverket som ble dannet ved å starte flere noder som kjørte applikasjonen. Det er også påbegynt moduler i prosjektet som kombinerer P2P og SIP. Status på prosjektet for øyeblikket er dessverre slik at noen av disse modulene er langt i fra ferdig, noe som medførte en rekke uforutsigbare feil-meldinger.

Noe som derimot fungerte bedre var en modul som implementerer «OpenDHT» [52] (se seksjon 5.2.3) sine funksjoner `put()`, `get()` og `remove()` ved hjelp av XML Remote Procedure Call (XML-RPC) [53]. Denne modulen er i stor grad lånt fra «OpenDHT»-prosjektet sin kildekode og inkludert i «39 Peers». Det er gjort noen justeringer på kildekoden for å få den til å passe inn med resten av prosjektet. Modulen er også en spesielt viktig del av min implementasjon, og forandringene som er foretatt er beskrevet i seksjon 5.3.2 på side 78.

5.2.3 OpenDHT

På grunnlag av funn i kapittel 3 ønsker jeg å benytte et strukturert P2P-nettverk for lagring av informasjon. Det var de strukturerte P2P-nettverkene som viste seg å kunne tilby skalerbarhet og robusthet på best måte, for dette formålet.

«OpenDHT» [52] er et prosjekt og en tjeneste som tilbyr en DHT-løsning som forskere og andre interesserte kan benytte seg av. Selve DHT-en er hostet på et hundretalls datamaskiner i et globalt forskningsnettverk kalt «PlanetLab» [54].

«OpenDHT» tilbyr et lite sett med funksjoner som muliggjør lagring og henting av informasjon i DHT-en. Det er kommandoene `put(nøkkel, verdi)` og `get(nøkkel)` som danner grunnlaget for lagring og henting av informasjon.

DHT-Gateway (GW)

For å kunne lagre informasjon og hente ut lagret informasjon fra «OpenDHT» kan man enten kjøre sin egen node som kobler seg opp i mot resten av DHT-en og tar del i nettverket, eller man kan benytte ett API med et utvalg kommandoer som kobler seg opp til en av nodene som allerede er i DHT-nettverket. Disse nodene kalles DHT-GW-er.

I prototypen har jeg benyttet meg av det andre alternativet, altså ved å benytte meg av et API som gir funksjonene `put(nøkkel,verdi)` og `get(nøkkel)`. For å benytte seg av dette API-et trenger man en fungerende DHT-GW. Disse DHT-GW-ene er det mulig å kommunisere med ved hjelp av XML-RPC [53].

I API-et for å kommunisere med DHT-GW-er, er det spesielt de to kommandoene `put(nøkkel,verdi)` og `get(nøkkel)` som benyttes av prototypen. Det er veldig vanlig for DHT-API-er å tilby disse to funksjonene. Det hadde derfor vært fullt mulig å bytte ut «OpenDHT» med en annen DHT i prototypen. DHT-en kunne enten kjørt lokalt og tatt del i et DHT-nettverk, eller det kunne vært en ekstern DHT tilsvarende «OpenDHT».

5.3 Prototypen

Prototypen er en SIP-server som fungerer som en registrar, proxy og redirect-server. Hovedforskjellen på denne prototypen og andre SIP-servere er måten LS-en lagrer informasjonen. I stede for å lagre navne-oppslag (AoR) i minnet på LS-en, blir det i denne prototypen lagret i en DHT.

Prototypen består av tre deler jeg kommer til å presentere. I den første delen forklarer jeg hvilke endringer jeg gjorde på SIP-registraren og LS-en. Den andre delen er en forklaring på hvordan kommunikasjonen mellom LS-en og DHT-en foregikk. Og den tredje delen er en forklaring på hvordan jeg gikk frem for å finne en optimal DHT-GW.

Siden dette er en prototype og et «proof of concept» er det enkelte forbehold og arkitekturiske valg som er tatt. Disse finnes i listen under:

- Prototypen, i tillegg til SIP-klient anses som én enhet.
- DHT-en i prototypen kunne vært en hvilken som helst annen DHT-implementasjon så lenge kode-signaturen[2] til funksjonene `put()` og `get()` er like.
- Jeg kunne laget en egen implementasjon av en DHT, men for enkelhets skyld er «OpenDHT» valgt som DHT.
- Prototypen er laget med en antagelse at det ikke eksisterer ondsinnede noder. Det er derfor ikke laget sikkerhetstiltak mot dette.

5.3.1 Location Service ved hjelp av en DHT

Som nevnt i seksjon 2.2.1 på side 14 er LS-er noe som blir benyttet av proxyer og redirect-servere til å gjøre navneoppslag (AoR). Det er SIP-registraren som mottar opplysningene til navneoppslagene fra SIP-klienter.

[2] Lik rekkefølge og tilsvarende argumenter, samt likt format på returverdiene

Det er applikasjonen *sipd.py* i rammeverket «39 Peers» som starter i gang SIP-serveren. Applikasjonen benytter SIP-API-et (*sipapi.py*) i rammeverket, samt modulen til SIP-standarden (*rfc3261.py*). Når SIP-serveren startes oppretter den en LS som vanligvis benyttes til å lagre navneoppslag i. Den benyttes også når SIP-UA-er sender `INVITE`-forespørsler. I prototypen har jeg valgt å bytte ut denne LS-en.

Det blir i den uendrede *sipd.py*-kildekoden angitt på linje 55 at `sipapi.Location()` skal være LS. I prototypen er dette endret til `dhtls.LocationDHT()` hvor `LocationDHT()` er klassen (class), og `dhtls` er den nye LS-modulen som blir brukt i prototypen.

Denne klassen består av tre funksjoner, og har en lik kode-signatur som `sipapi.Location()`. Ved å ha lik kode-signatur vil det derfor ikke være nødvendig med flere endringer i *sipd.py* og *sipapi.py*.

dhtls.py

Hele den kommenterte kildekoden til filen *dhtls.py* finnes i Tillegg A.1 på side 107. Denne modulen inneholder klassen `LocationDHT()`. De tre funksjonene i klassen `LocationDHT()` er som følger; `init`, `save` og `locate`.

init

`init` er funksjonen som initialiserer klassen. Ved initialisering starters først og fremst `DHT()`-klassen. Denne klassen omtales i seksjon 5.3.2. Den starter også en `Debug()`-klasse som sørger for informativ utskrift underveis når applikasjonen kjører.

save

save er LS-funksjonen som lagrer innholdet som blir sendt fra SIP-registraren når den mottar INVITE- og PUBLISH-forespørsler. Funksjonen tar tre argumenter: msg, uri og defaultExpires.

msg-argumentet består av hele SIP-forespørselen registraren har mottatt. De interesante delene av SIP-forespørselen er Expires og Contact. Listing 5.1 er et eksempel på en REGISTER-forespørsel som er sendt fra Alice til registraren (wonderland.com).

Expires er et parameter, som gjerne konfigureres i SIP-klienten, av brukeren, og avgjør hvor lenge SIP-registraren og LS-en skal beholde informasjonen fra REGISTER-forespørselen. Hvis dette parameteret ikke er definert i SIP-forespørselen er det defaultExpires som blir benyttet. defaultExpires har en standard-verdi på 3600 sekunder. Denne verdien kan endres av SIP-serveren. I eksempelet er Expires angitt på linje 11.

Contact er det andre parameteret som er interessant i SIP-forespørselen funksjonen mottar. Denne inneholder den fysiske nettverksadressen og porten til SIP-endepunktet som har sendt REGISTER-forespørselen. REGISTER-forespørselen kan inneholde flere nettverksadresser og porter. Dette, sammen med det andre argumentet i funksjonen, danner grunnlaget for navneoppslag (se seksjon 2.2.1). Contact er angitt i linje 7 i eksempelet (sip:alice@129.240.65.77:5060).

Listing 5.1: Alice sender REGISTER til SIP-server

```
1 REGISTER sip:wonderland.com SIP/2.0
2 Via: SIP/2.0/UDP 129.240.65.77:5060;branch=↩
     z9hG4bK008523f7a2b7e011be1e001018839764;rport
3 From: "Alice in Wonderland" <sip:alice@wonderland.com>;tag=2458616260
4 To: "Alice in Wonderland" <sip:alice@wonderland.com>
5 Call-ID: 80EE8AF6-A2B7-E011-BE1D-001018839764@129.240.65.77
6 CSeq: 1 REGISTER
7 Contact: <sip:alice@129.240.65.77:5060>;+sip.instance="<urn:uuid:00AEF5B9-6942-↩
     E011-B2B4-001E8C4AFCF2>"
8 Allow: INVITE, OPTIONS, ACK, BYE, CANCEL, INFO, NOTIFY, MESSAGE, UPDATE
```

```
 9  Max-Forwards: 70
10  User-Agent: SIPPER for PhonerLitePortable
11  Expires: 3600
12  Content-Length: 0
```

Det andre argumentet i `save`-funksjonen er `uri`. Dette er SIP-URI-en til SIP-klienten som har sendt `REGISTER`-forespørselen. Dette argumentet utgjør nøkkelen som blir sendt til DHT-en.

Det siste argumentet i funksjonen er som jeg nevnte `defaultExpires`. Dette argumentet benyttes kun hvis `Expires`-parameteret ikke inneholder en verdi. Dette er et valgfritt argument, og blir satt til 3600 sekunder hvis annet ikke er spesifisert av *sipd.py*-applikasjonen.

Siden `Contact` i `msg`-argumentet kan inneholde en eller flere fysiske adresser er det en løkke som går igjennom alle adressene. For hver adresse sendes de videre til DHT-en. DHT-en mottar nøkkel-/verdi-par fra LS-en ved hjelp av funksjonen `put()`. Nøkkelen i dette tilfellet settes til verdien av `uri`-argumentet, og verdien er den fysiske adressen fra `Contact`. I tillegg sendes det også med `expires` som avgjør hvor lenge DHT-en skal beholde informasjonen lagret.

I Listing 5.2 ser vi hvilken informasjon som blir sendt fra LS-en til DHT-en. SIP-URI-en (`sip:alice@wonderland.com`) er nøkkelen, og `sip:alice@129.240.65.77:5060` er verdien som lagres. `Expire` er i dette eksempelet satt til 3600, siden dette var verdien LS-en fikk tilsendt fra registraren.

Listing 5.2: Alice sin SIP-URI blir lagret i DHT-en sammen med nettverksadressen(e)

```
1  DHT-LS Save Uri     : sip:alice@wonderland.com
2              Value   : <sip:alice@129.240.65.77:5060>
3
4  OpenDHT Put Key     : sip:alice@wonderland.com
5              Value   : <sip:alice@129.240.65.77:5060>
6              Secret  :
7              Expire  : 3600
```

Etter å ha lagret all informasjonen i DHT-en, om de fysiske nettverksadressene SIP-endenoden befinner seg på, returnerer funksjonen `True` for å signalisere tilbake til SIP-serveren at lagringen gikk i orden.

locate

Funksjonen `locate` er den delen av LS-en som henter informasjon om en SIP-URI. Som kjent gjør LS-en navneoppslag ved å oversette en SIP-URI over til den eller de fysiske nettverksadressen(e) (IP og port) SIP-endepunktet befinner seg på.

Funksjonen tar kun ett parameter; `uri`. Dette parameteret blir sendt fra en SIP-serveren når den mottar en `INVITE`-forespørsel. SIP-serveren (når den opptrer som en proxy eller en redirect-server) får kun tilsendt SIP-URI-en når den mottar `INVITE`-forespørsler, og det er LS-en sin oppgave å oversette denne SIP-URI-en om til en fysisk nettverksadresse.

Måten denne funksjonen gjør navneoppslag på, etter å ha mottatt en SIP-URI fra SIP-serveren, er å sende en forespørsel til DHT-en. I `save`-funksjonen forklarte jeg at det var SIP-URI som ble benyttet som nøkkel i DHT-en. LS-en sender derfor en `get(uri)` til DHT-en. DHT-en gjør et søk etter nøkkelen (`uri`), og returnerer verdiene som er lagret i DHT-en.

Listing 5.3: LS gjør navneoppslag, og får tilbake verdier fra DHT-en

```
1  DHT-LS Locate Uri    : sip:alice@wonderland.com
2
3  OpenDHT Get Key      : sip:alice@wonderland.com
4              Value    : <sip:alice@129.240.65.77:5060>
5              Expire   : 3340
```

Alle verdiene som nå har blitt hentet fra DHT-en blir gjort om til formatet rammeverket benytter seg av. Til slutt blir verdiene som nå har blitt gjort om til riktig format returnert tilbake til proxyen eller redirect-serveren som benyttet seg av DHT-en.

5.3.2 Kommunikasjon med DHT

I prototypen foregår kommunikasjonen mellom LS og DHT ved hjelp av to funksjoner. Disse to funksjonene sørger for lagring og henting av data i DHT-en. Funksjonene som benyttes av LS-en er `put(nøkkel, verdi)` og `get(nøkkel)`.

Modulen «OpenDHT», som allerede eksisterte i rammeverket «39 Peers», er modifisert i forbindelse med denne prototypen. Jeg kommer til å nevne de funksjonene som er i bruk i prototypen.

opendht.py

opendht.py er en modul som inneholder én klasse; `DHT()`. I forklaringen av denne modulen kommer jeg til forklare funksjonene som har blitt brukt i prototypen. Hele den kommenterte kildekoden finnes i Tillegg A.2 på side 108.

init

Denne funksjonen tar ett argument; `gateway`. `gateway` er et valgfritt argument, og kan defineres, av f.eks. LS-en, for å angi en DHT-GW som skal bli brukt.

Hvis det ikke er definert en GW vil DHT-modulen benytte seg av `FindGateway`-modulen for å finne en GW. Denne klassen forklares i seksjon 5.3.3 på side 82.

Som jeg forklarte i seksjon 5.2.3 på side 72 er det nødvendig med en DHT-GW for å bruke «OpenDHT» så lenge man ikke kjører en egen node i nettverket. Hvis `FindGateway`-modulen ikke finner en fungerende GW vil det bli gitt beskjed til brukeren om dette, og applikasjonen avsluttes. Ved neste oppstart av applikasjonen vil `FindGateway`-modulen prøve på nytt å finne en fungerende GW.

Etter å ha funnet en DHT-GW avsluttet `init` med å legge til '`http://`' og port på GW-en, og gjør brukeren av applikasjonen klar over hvilken DHT-GW som blir benyttet.

put

Det er funksjonen `put()` som sørger for å lagre informasjon i DHT-en. Funksjonen `put()` tar imot fire argumenter; `key`, `value`, `secret` og `ttl`. De to siste argumentene er valgfrie.

`key`-argumentet er nøkkelen som blir benyttet for å lagre data i DHT-en. Nøkkelen fungerer som en identifikator, og gjør det mulig å hente ned igjen informasjonen som er lagret på denne nøkkelen ved hjelp av `get()`-funksjonen.

`value`-argumentet er verdien man ønsker å lagre på den gitte nøkkelen i DHT-en. Til navneoppslagene LS-en lagrer, er verdien i disse tilfellene den fysiske nettverksadressen, og `key` er SIP-URI-en. I Listing 5.2 på side 76 så vi hvordan LS-en ble bedt om å lagre et navneoppslag, og sendte dette videre til `DHT()`-modulen. I DHT-en er det mulig å lagre flere verdier på den samme nøkkelen.

I `put()`-funksjonen er det også mulig å definere en `secret` (hemmelighet). Denne hemmeligheten blir brukt for å kunne slette den tilhørende verdien (`value`) under nøkkelen som er definert (`key`).

Det siste argumentet i `put()` er `ttl`. Jeg har tidligere forklart uttrykket TTL, men i dette tilfellet betyr det hvor mange sekunder meldingen, som har blitt lagret i DHT-en, skal bli liggende før den blir slettet.

Alle argumentene med unntak av `ttl` blir omgjort til binær form ved hjelp av funksjonen `Binary()` før de blir sendt til DHT-en. Både `key` og `secret` blir også kjørt igjennom en hashing-algoritme (SHA-1) før de blir omgjort til binær form.

Etter at denne prosessen er ferdig blir det brukt XML-RPC for å sende

informasjonen til DHT-en. DHT-GW-en vil deretter svare tilbake om `put()` gikk, eller ikke. Til slutt returneres dette resultatet tilbake til LS-en (`True` eller `False`).

I Listing 5.4 på neste side fra linje 3 til 6 er eksempel på hvordan en `get()` med nøkkel '`DHTtest-Fri Jul 29 19:40:12 2011`' blir lagret i DHT-en.

get

Som vi så i `put()`-funksjonen, ble den brukt til å lagre informasjon i DHT-en. For å hente ut informasjon fra DHT-en benyttes funksjonen `get()`. Funksjonen tar to argumenter; `key` og `maxvals`. `maxvals` er et valgfritt argument, med 10 som standardverdi.

Det første argumentet til funksjonen er `key`. Som kjent er all data som ligger lagret i DHT-en knyttet til en nøkkel, altså `key`. Funksjonens oppgave er å motta en nøkkel, og så returnere de verdiene som tilhører nøkkelen.

Før nøkkel-/verdi-par ble lagret i DHT-en av funksjonen `put()`, ble nøklene kjørt igjennom en hashing-algoritme (SHA-1). For å hente ut lagrede nøkkel-/verdi-par må vi derfor i `get()` også kjøre `key` igjennom den samme hashing-algoritmen.

Sett i sammenheng med bruksområdene til LS-en, så er det egentlig ikke SIP-URI-en som er nøkkelen, men hashen av SIP-URI-en.

For å hente ut informasjon av DHT-en bruker vi derfor denne hashen, og sender en beskjed til DHT-GW-en om at vi ønsker å hente ut verdien til nøkkelen. Siden det er mulig å lagre flere verdier under samme nøkkel, er det i `get()` mulighet for å begrense antall resultater vi får tilbake fra DHT-GW-en. Det er dette argumentet `maxvals` brukes til.

På samme måte som `put()` benytter XML-RPC til å lagre informasjon, benyttes det i `get()` også XML-RPC for å hente informasjon. For hver av

verdiene DHT-modulen mottar fra DHT-GW-en, lagres disse i et «array». Hvis brukeren har aktivert `Debug()`-modulen, vil vedkommende også se hvilke verdier som blir returnert fra DHT-en. Til slutt i denne funksjonen blir «arrayet» med alle verdiene returnert.

I Listing 5.4 fra linje 8 til 11 finnes det et eksempel på hvordan `put()` mottar verdier fra DHT-en med nøkkelen '`DHTtest-Fri Jul 29 19:40:12 2011`'.

test

Funksjonen `test()` er kun laget for at brukeren skal kunne vite om DHT-GW-en, som er valgt, fungerer. Funksjonen tar ingen argumenter, men benytter verdiene som allerede er lagret i klassen.

Når denne funksjonen blir kjørt gjør den to ting. Først kontrollerer den om den får kontakt med DHT-GW-en ved å gjøre en `put()`. Den benytter nøkkelen '`DHTtest-klokkeslett`', og verdien på nøkkel-/verdi-paret er host-navnet på maskinen, klokkeslettet og hvilken GW den er tilkoblet.

Etter å ha lagret informasjon i DHT-en med kommandoen `put()`, prøver den å hente tilbake den samme informasjonen ved å gjøre en `get()`. Funksjonen `get()` benytter da den samme nøkkelen som ble brukt i `put()`.

Til slutt sammenlignes den sendte verdien, med den mottate verdien, for å kontrollere at `put()` og `get()` gjør det de skal.

Hvis den sendte og det mottatte ikke er likt, vil brukeren få beskjed om dette i en utskrift i terminalen. Det hender også noen ganger at DHT-GW-ene slutter å respondere. Dette vil da gi en «timeout» og applikasjonen avsluttes.

I Listing 5.4 er et eksempel på en test hvor en test ble gjennomført med ett positivt resultat.

Listing 5.4: Eksempel på en DHT-test

```
OpenDHT: Using gateway http://planetlab2.wiwi.hu-berlin.de:5851
OpenDHT _test:
OpenDHT Put Key      : DHTtest-Fri Jul 29 19:40:12 2011
            Value    : Hostname: mirabella.ifi.uio.no Time: Fri Jul 29 19:40:12 ↩
                2011 GW: http://planetlab2.wiwi.hu-berlin.de:5851
            Secret   : secret
            Expire   : 30

OpenDHT Get Key      : DHTtest-Fri Jul 29 19:40:12 2011
            Value    : Hostname: mirabella.ifi.uio.no Time: Fri Jul 29 19:40:12 ↩
                2011 GW: http://planetlab2.wiwi.hu-berlin.de:5851
            Expire   : 29
OpenDHT _test: success!
```

5.3.3 Finne den optimale DHT-Gateway

For å kunne kommunisere med «OpenDHT» uten å være en del av nettverket selv trenger man en GW som kan gjøre kommunikasjonen med nettverket på vegne av applikasjonen. GW-ene i «OpenDHT»-nettverket har en noe varierende oppetid, og det derfor ingen garanti for at en GW som fungerer i dag vil fungere i morgen. Enkelte DHT-GW er mer belastet enn andre, og responderer derfor treigere [55].

Enkelte av GW-ene befinner seg også veldig langt borte med tanke på geografisk plassering og routing. For at responstiden til disse DHT-GW-ene skulle være akseptable laget jeg derfor en modul som sjekket responstiden på ett utvalg noder i «OpenDHT»-nettverket ut i fra en liste på over 100 noder.

findgateway.py

Modulen inneholder to klasser. Den første av disse er `SpeedTest()`-klassen. Denne klassen arver egenskapene til klassen `threading.Thread` som er en del av Python.

Den andre klassen i modulen er `FindGateway()`, og det er denne som benyttes direkte av `DHT-LS`-modulen. Hele den kommenterte kildekoden finnes i Tillegg A.3 på side 112.

Klassen `FindGateway()` inneholder en rekke funksjoner; `init`, `GetServers`, `GetList`, `NumServers`, `ShuffleServers`, `SaveWorking` og `TestServers`. Disse funksjonene blir forklart på neste side under overskriften FindGateway.

SpeedTest

Denne klassen mottar `threads`, eller tråder fra `FindGateway`-klassen. Argumentene denne klassen mottar er nettverksadressen til GW-en, IP-adressen, og en peker til `Debug`-klassen for utskrift til brukeren.

Siden denne klassen har arvet egenskapene til `Threads` må man følge kode-signaturen til denne klassen. Det er kun funksjonene `init()` og `run()` jeg har tatt med i denne klassen.

Funksjonen `init()` håndterer argumentene som klassen mottar. Dette er som sagt nettverksadresse (`servername`) og IP-adresse (`serverip`).

Funskjonen `run()` mottar ingen argumenter, men benytter den informasjonen som allerede er lagret i klassen. Det denne funksjonen gjør er å åpne en `socket`-tilkobling til IP-adressen den mottok i `init()` på port 5851. Dette er standard-porten for DHT-GW-er i «OpenDHT»-nettverket.

Det aktiveres en «timeout» på denne tilkoblingen på 5 sekunder, så hvis GW-en ikke responderer innen den tid, så settes variabelen `self.working` i klassen til verdien `False`. Dette indikerer da at serveren ikke responderer innen ønsket tid (maksimalt 5 sekunder).

Hvis funksjonen derimot greier å opprette kontakt med DHT-GW-en innen 5 sekunder settes `self.working`-variabelen i klassen til `True`. Dette indikerer at GW-en oppfyller kravet om en responstid på mindre enn 5 sekunder.

Hvis den globale variabelen `verbose` i modulen er satt til `True` vil applikasjonen skrive ut til brukeren resultatet fra alle testene (om de gikk gjennom, eller om de feilet).

I Listing 5.5 på neste side er det fra linje 7 til 16 eksempler på utskrift brukeren får tilbake fra `SpeedTest()`.

FindGateway

Denne klassen benyttes av DHT-modulen til å finne DHT-GW-er. Klassen har en rekke funksjoner som jeg allerede har nevnt tidligere, og flesteparten av disse er hjelpefunksjoner for andre funksjoner.

Ved initialisering (`init()`) av klassen, som gjerne blir foretatt av DHT-en ved oppstart, tar klassen i mot to argumenter. Det første argumentet (`numservers`) avgjør hvor mange GW-er som skal leses ut i fra GW-listen. GW listen blir angitt i det andre argumentet, `filename`.

Begge disse argumentene er valgfrie, og `numservers` har en standardverdi på 20. Det er også valgfritt å oppgi filnavn.

Etter å ha mottatt disse argumentene setter klassen i gang med å hente listen med GW-er. Dette skjer med funksjonen `GetList()`. Hvis det i `init` ble angitt et filnavn, er det denne filen som leses av funksjonen. Hvis det derimot ikke er oppgitt et filnavn vil funksjonen `GetList()` laste ned en GW-liste fra en nettadresse. Hvordan denne GW-listen er generert forklarer jeg på side 86.

Filen med listen blir gått igjennom linje for linje og lagt til i et «array». Dette «arrayet» danner grunnlag for GW-ene som skal kontrolleres senere.

De neste funksjonen som kjøres er `NumServers()` og `ShuffleServers()`. Dette er kun hjelpefunksjoner som hver gjør en liten oppgave. `NumServers` sjekker om variablen `numservers` (fra `init()`) er større enn antall GW-er i filen med listen, og `ShuffleServers` stokker om på rekkefølgen til GW-ene i listen, slik at ikke de samme (som ligger i starten av listen) skal

bli sjekket hver gang.

Etter at disse tre funksjonene nå er kjørt av `init()` har vi nå en liste med ’`numservers`’ antall GW-er i tilfeldig rekkefølge som vi skal teste. Testene blir satt i gang av `TestServers()`.

Funksjonen `TestServers()` genererer først like mange tråder (threads) som `numservers`, og lagrer disse i et «array». Disse trådene er av typen `SpeedTest` som jeg tidligere omtalte.

Etter å ha opprettet alle `SpeedTest`-trådene settes disse i gang å sjekke GW-ene. Som jeg forklarte tidligere er det en «timeout» på disse trådene. Tråder som ikke har et positivt resultat (`working`) blir forkastet. Etter at disse trådene da har feilet eller oppnådd kontakt med en GW sitter vi nå igjen med en liste fungerende DHT-GW-er.

Denne listen er lagret i klassen, og kan hentes ut at f.eks DHT-modulen. Det er med funksjonen `GetServers()` andre moduler henter ut GW-er som fungerer. Denne funksjonen er med vilje en blokkerende operasjon (blocking operation), slik at alle trådene skal kunne gjøre seg ferdig før funksjonen returnerer en liste med GW-er til andre moduler.

I Listing 5.5 er modulen kjørt fra kommandolinjen med 10 som parameter. Dette parameteret angir hvor mange servere ut i fra listen den skal kontrollere. Ingen filnavn er valgt som parameter, så `FindGateway()` laster derfor ned GW-listen fra en nettside. Denne listen inneholder 105 GW-er.

Videre ser vi i eksempelet at de 10 serverene kontrolleres av `SpeedTest`, og at `GetServers` venter på listen med de fungerende GW-ene. Til slutt ser vi at 2 fungerende DHT-GW-er returneres.

Listing 5.5: FindGateway tester 10 DHT-GW-er

```
1  $$ python fredrot/findgateway.py 10
2
3  FindGateway/ GetList: serverlist from URL: http://fredrot.at.ifi.uio.no/servers↩
       .txt
4  FindGateway/ GetList: Got 105 gateways
```

```
5
6  FindGateway/ TestServers: Testing 10 gateways
7  FindGateway/ SpeedTest failed : planetlab2.cesnet.cz / 195.113.161.83
8  FindGateway/ SpeedTest failed : planetlab1.eurecom.fr / 193.55.112.40
9  FindGateway/ SpeedTest failed : plab-2.sinp.msu.ru / 213.131.1.102
10 FindGateway/ SpeedTest failed : 146-179.surfsnel.dsl.internl.net / ↩
       145.99.179.146
11 FindGateway/ SpeedTest success: node1.planetlab.mathcs.emory.edu / ↩
       170.140.119.69
12 FindGateway/ SpeedTest failed : ricepl-2.cs.rice.edu / 128.42.142.42
13 FindGateway/ SpeedTest success: planetlab-2.cs.colostate.edu / 129.82.12.188
14 FindGateway/ SpeedTest failed : node2.lbnl.nodes.planet-lab.org / 198.128.56.12
15 FindGateway/ SpeedTest failed : planetlab1.informatik.uni-kl.de / ↩
       131.246.19.201
16 FindGateway/ SpeedTest failed : planet03.csc.ncsu.edu / 152.14.92.59
17
18 FindGateway/ GetServers: Waiting for server list
19
20 Found 2 working servers:
21 planetlab-2.cs.colostate.edu
22 node1.planetlab.mathcs.emory.edu
```

Generering av GW-listen ble gjort ved hjelp av applikasjonen «PyCrawler» [56]. Kun en liten del av koden ble redigert, så jeg har derfor valgt å ikke inkludere denne som et tillegg. Kildekoden til den modifiserte «PyCrawler» er vedlagt sammen med de andre kildekodefilene, og ligger i mappen tools.

For å generere listen trengte jeg først en fungerende «OpenDHT»-GW. Disse GW-ene svarer som kjent på XML-RPC, i tillegg til at de også responderer med å vise en nettside hvis man går inn på nettverksadressen til en fungerende server på standard-porten 5851. Denne nettsiden viser status for GW-en, med informasjon om oppetid, hvor mye den lagrer av informasjon og hvilke nabo-noder (andre GW-er) den har.

Disse nabo-nodene er vist som nettlenker på statussiden, og med disse lenkene kunne man da bruke en web-crawler for å samle sammen så mange noder som mulig til en liste. Ved hjelp av en modifisert «PyCrawler» fikk jeg den til å gå videre til neste node i nettverket. Alle disse nodene ble lagret i en «SQLite»-database [57], og deretter lagt i txt-

filen som `FindGateway` henter fra nettadressen.

5.4 Resultater

Vi har så langt i dette kapittelet sett hvordan prototypen er bygget, og hva de forskjellige delene av prototypen gjør. I denne seksjonen skal vi se et eksempel på en samtale mellom Alice og Bob. Avslutningsvis i denne seksjonen presenterer jeg mine funn og erfaringer med prototypen.

5.4.1 Hvordan fungerte dette i praksis?

I dette avsnittet skal vi se et eksempel på hvordan prototypen fungerer og hvilke forberedelser som kreves for å starte prototypen.

Forberedelser

Brukerene Alice og Bob benytter hver sin datamaskin. OS-et på Alice sin maskin er `Windows 7 SP1`, og OS-et på Bob sin maskin er `Windows Server 2003`. Begge brukerene benytter SIP-klienten «PhonerLite» som ble beskrevet i seksjon 5.2.1 på side 67, og har `Python 2.6.5 (32bit)` installert.

Oppstart av Prototypen

For å starte prototypen bruker man kommandoene i listen under. I dette eksempelet er port `5062` benyttet. Den samme porten må også benyttes i konfigurasjonen av SIP-klienten.

- Windows: `python app\sipd.py -d -l 0.0.0.0:5062`
- Linux: `python app/sipd.py -d -l localhost:5062`

`sipd.py` aktiverer `dhtls.py`, som igjen aktiverer `opendht.py`. Hvis det er hard-kodet en DHT-GW i `opendht.py` blir denne benyttet. Hvis ikke aktiveres `findgateway.py`, som finner en fungerende DHT-GW.

Konfigurasjon av SIP-klient

I konfigurasjonen på SIP-klienten fylles det inn `'<maskinnavn>:5062'` som Proxy/Registrar og `'<domene.no>'` som Domain/Realm (se figur 5.2a på neste side). I enkelte klienter kan det også spesifiseres «outbound proxy». Sett også denne til `'<maskinnavn>:5062'`. Under User-fanen i SIP-klienten fyller det inn navn, brukernavn, og passord (se figur 5.2b på neste side). Det er egentlig ikke nødvendig med passord, men enkelte SIP-klienter krever dette.

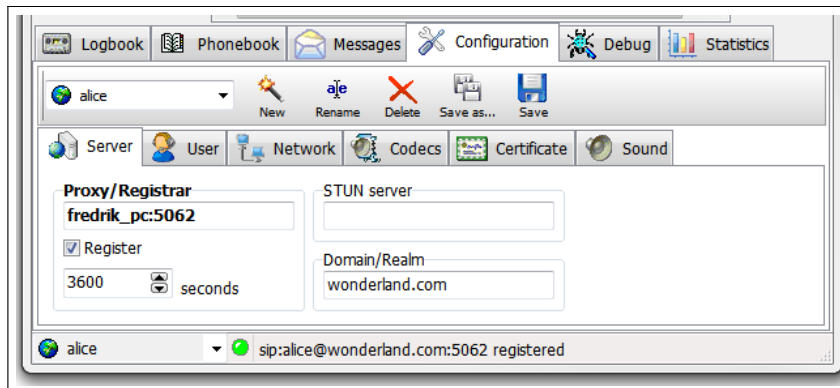
Samtalen

Alice og Bob har nå konfigurert sine SIP-klienter, og startet opp prototypen. Prototypen har tatt i mot `REGISTER`-forespørslene til begge, og lagret disse i DHT-en.

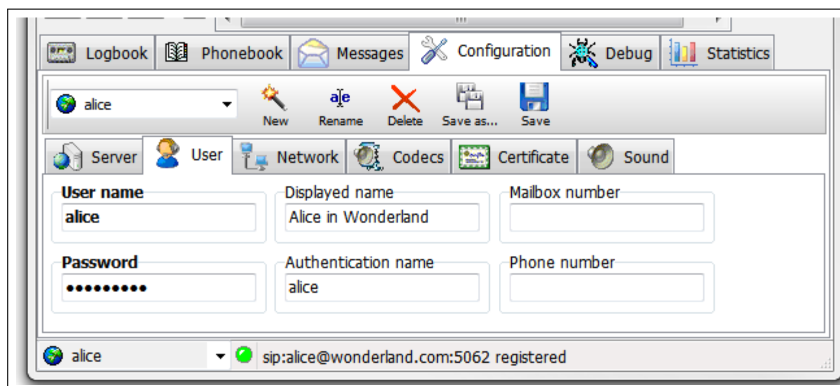
Alice ønsker nå å ringe til Bob. Bob har tidligere fortalt Alice at han kan ringes til på adressen `'sip:bob@wonderland.com'`. Alice prøver å gjøre dette i Listing 5.6. I dette eksempelet ser vi at Alice sender en `INVITE`-forespørsel til sin prototype-proxy. Når proxyen mottar denne forespørselen prøver den å finne den fysiske nettverksadressen til Bob (linje 20). Proxyen sender forespørselen videre til LS-en (linje 22), som igjen sender en forespørsel til DHT-en (linje 24). LS-en får svar fra DHT-en (linje 28), og har nå den fysiske nettverksadressen til Bob. Invitasjonen fra Alice blir nå sendt til Bob (linje 31). Etter en liten stund hører Bob at det ringer, og tar telefonen (se figur 5.3 på side 91).

Listing 5.6: `INVITE` fra Alice til Bob

```
1 # Alice skal invitere Bob til samtale, og sender INVITE til proxyen med Bob
```



(a) Server



(b) User

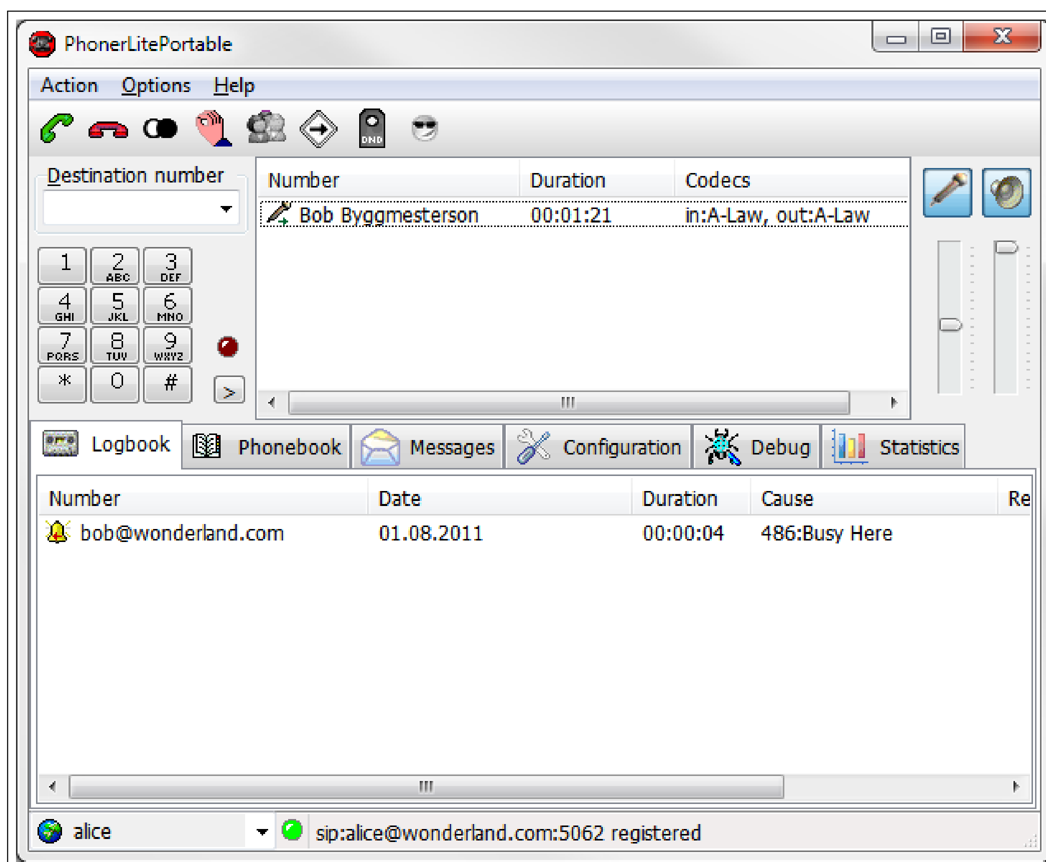
Figur 5.2: PhonerLite: Configuration

```
2  # sin SIP-URI
3  INVITE sip:bob@wonderland.com SIP/2.0
4  Via: SIP/2.0/UDP 192.168.1.33:5060;branch=↩
       z9hG4bK0096cd2255bae01183d9001e8c4afcf
5  2;rport
6  From: "Alice in Wonderland" <sip:alice@wonderland.com>;tag=435818064
7  To: <sip:bob@wonderland.com>
8  Call-ID: 0096CD22-55BA-E011-83D8-001E8C4AFCF2@192.168.1.33
9  CSeq: 3 INVITE
10 Contact: <sip:alice@192.168.1.33:5060>
11 Content-Type: application/sdp
12 Allow: INVITE, OPTIONS, ACK, BYE, CANCEL, INFO, NOTIFY, MESSAGE, UPDATE
13 Max-Forwards: 70
14 Supported: 100rel, replaces
15 User-Agent: SIPPER for PhonerLitePortable
16 P-Preferred-Identity: <sip:alice@wonderland.com>
17 Content-Length:    418
```

```
18 # ... SDP fjernet ...
19
20 received request from stack INVITE
21 # Alice sin proxy til LS:
22 DHT-LS Locate Uri    : sip:bob@wonderland.com
23 # LS til DHT:
24 OpenDHT Get Key      : sip:bob@wonderland.com
25             Value    : <sip:bob@192.168.1.20:5060>
26             Expire   : 3409
27 # DHT responderer tilbake til LS:
28 DHT-LS Locate return: [Contact: <sip:bob@192.168.1.20:5060>;expires=3409]
29
30 # Alice benytter den fysiske nettverksadressen til Bob:
31 INVITE sip:bob@192.168.1.20:5060 SIP/2.0
32 Call-ID: 0096CD22-55BA-E011-83D8-001E8C4AFCF2@192.168.1.33
33 Content-Length: 418
34 Via: SIP/2.0/UDP 192.168.1.33:5062;rport;branch=z9hG4bKvB-DK0zPyolPeJXp7VIUOA..
35 Via: SIP/2.0/UDP 192.168.1.33:5060;rport=5060;branch=↩
       z9hG4bK0096cd2255bae01183d9
36 001e8c4afcf2
37 From: "Alice in Wonderland" <sip:alice@wonderland.com>;tag=435818064
38 P-Preferred-Identity: <sip:alice@wonderland.com>
39 Supported: 100rel
40 Supported: replaces
41 To: <sip:bob@wonderland.com>
42 Contact: <sip:alice@192.168.1.33:5060>
43 CSeq: 3 INVITE
44 Allow: INVITE
45 Allow: OPTIONS
46 Allow: ACK
47 Allow: BYE
48 Allow: CANCEL
49 Allow: INFO
50 Allow: NOTIFY
51 Allow: MESSAGE
52 Allow: UPDATE
53 Max-Forwards: 69
54 Record-Route: <sip:192.168.1.33:5062;lr>
55 User-Agent: SIPPER for PhonerLitePortable
56 Content-Type: application/sdp
57 # ... SDP fjernet ...
```

5.4.2 Evaluering

I dette avsnittet gjør jeg en evaluering av de forskjellige delene i prototypen for å se om de ønskede formålene i seksjon 5.1 på side 66 ble nådd.



Figur 5.3: Alice og Bob ringer ved hjelp av P2PSIP

Er prototypen skalerbar?

For å oppnå skalerbarhet i denne prototypen benyttet vi en DHT. Som vi så i seksjon 3.4 på side 54 og 3.2.2 på side 39 var strukturerte overlay-nettverk en løsning som kunne tilby stor grad av skalerbarhet til en tjeneste av denne sorten. Siden dette var et «proof of concept» var det ikke nødvendig å implementere en egen DHT til dette formålet. DHT-en som ble valgt i denne implementasjonen var «OpenDHT». Prototypen var ikke selv en del av overlay-nettverket i «OpenDHT», men benyttet en DHT-GW for å lagre og hente ut informasjon.

Ved å benytte en DHT til dette vil jeg si vi oppnådde det ønskede formålet med god skalerbarhet i prototypen. Dette begrunner jeg med

at de strukturerte P2P-nettverkene har vist seg å kunne håndtere et stort antall noder, først og fremst i forhold til klient/server-arkitekturen tradisjonell SIP er bygget etter, og også i forhold til de ustrukturerte P2P-nettverkene. Prototypen da vil «arve» disse egenskapene fra de strukturerte P2P-nettverkene ved å benytte dette fremfor tradisjonell klient/server-arkitektur.

Er prototypen robust?

For å oppnå robusthet i en tjeneste mener jeg at tjenesten skal kunne fungere selv om det oppstår feil og/eller hindringer. I SIP, og VoIP generelt, vil feil si at sentrale enheter går ned, eller slutter og fungere. Dette vil ramme alle de tilkoblede klientene, og vil være et «single point of failure». Med hindringer mener jeg eksempelvis NAT-er og/eller brannmurer i nettverket som kan gjøre det vanskelig eller umulig for noder å opprette kontakt med hverandre uten ekstra hjelp.

I prototypen finnes det ikke et «single point of failure», siden LS-en sine oppgaver er tatt over av overlay-nettverket. I prototypen er ikke noden, som kjøres, selv en del av dette overlay-nettverket, men dette er kun et «proof of concept». I prototypen skal det likevel forestille at noden er en del av overlay-nettverket, og det hadde også vært mulig å byttet ut «OpenDHT» med en implementasjon av en annen DHT.

Hvis noden derimot hadde vært en del av dette overlay-nettverket (hvis dette ikke hadde vært et «proof of concept»), kunne vi begrunnet robustheten i tjenesten med at overlay-nettverk er robust. Overlay-nettverk kan oppnå denne robustheten siden informasjon som lagres i nettverket gjerne fordeles jevnt mellom nodene i nettverket, i tillegg til at informasjonen gjerne replikeres på flere noder. Dette gjør at overlay-nettverkene ikke blir påvirket i like stor grad, som for eksempel klient/server-arkitekturen blir, om sentrale servere feiler.

Når det gjelder hindringer er det ikke gjort noen tiltak i prototypen

for å omgå problemene disse kan medføre. Det som kunne blitt gjort i prototypen var å implimentere støtte for protokoller slik som STUN, TURN og ICE. Eksempelvis kunne noder som ikke var bak NAT-er og brannmurer tatt på seg rollen som en STUN-server, og lagret sin adresse i overlay-nettverket. Andre noder bak hindringer kunne hentet ned denne adressen, og benyttet seg av nodens STUN-tjeneste. Som vi så i seksjon 2.2.3 på side 19 benyttet «Skype» tjenester som lignet STUN og TURN.

I prototypen gjorde vi SIP, til en viss grad, mer robust, siden vi flyttet funksjonaliteten til LS-en over i overlay-nettverket. Vi tok på dette vises bort et «single point of failure» (registrarer og proxyer). Prototypen gjorde ingen tiltak for å hjelpe til med traversering gjennom NAT-er og brannmurer. Dette hadde vært ønskelig for å lage en enda mer robust tjeneste, og vi hadde med dette også kunne utnyttet resursene som er i nettverket på en bedre måte, ved å ikke ha dedikerte STUN-servere, men heller kjøre disse tjenestene på nodene i overlay-nettverket.

Plattform og SIP-klienter

Prototypen hadde som formål å kunne fungere på flere plattformer og uavhengig av SIP-klient. Rammeverkert «39 Peers» som jeg tok utgangspunkt i hadde allerede en stor kodebase som var kodet i Python. Python er i stor grad plattformuavhengig. Dette gjorde at jeg fikk testet prototypen på både Windows og Linux.

De testene jeg har gjort har vist at det er fullt mulig å kjøre prototypen på både Windows og Linux. Prototypen starter opp, og `findgateway.py` gjør jobben den er tiltenkt at den skal gjøre på begge OS-ene. Jeg vil derfor si at prototypen oppfylte målet om å fungere på flere platformer.

Det var også et mål at brukeren ikke skulle trenge å gjøre noen form for ekstra konfigurasjon utover det som er vanlig ved bruk av tradisjonell SIP. Prototypen oppfylte dette.

Til slutt hadde prototypen som mål at brukeren skulle kunne velge SIP-klient fritt, og jeg presenterte i seksjon 5.2.1 på side 67 to SIP-klienter. SIP-klienten «PhonerLite» fungerte bra med prototypen. Det var også mulig å aktivere et «Debug»-panel i denne klienten. Dette var til tider veldig nyttig. Den andre klienten, «Ekiga», fungerte ikke like bra. På grunn av dårlig responstid mellom prototypen og DHT-GW-en opplevde jeg svært ofte problemer med «Ekiga».

Etter at «Ekiga» hadde sendt «REGISTER» til prototypen, lagret prototypen dette i LS-en, som igjen sendte dette til DHT-en. Hvis denne prosessen tok særlig mer enn 5 sekunder fikk jeg i «Ekiga» meldingen «Registration failed: timeout». Det var dessverre ikke muligheter for å endre dette i «Ekiga», og responstiden til DHT-GW-en var det heller ikke mulig å gjøre noe med, annet enn å prøve en annen.

5.5 Oppsummering

Vi har i dette kapittelet sett en prototype som kombinerer SIP og P2P ved å bruke en DHT. Først så vi hvilke applikasjoner og rammeverk som ble benyttet i prosessen med å lage prototypen. Deretter var det en grundig gjennomgang av det som var blitt gjort i prototypen. Til slutt så vi ett eksempel på en samtale mellom Alice og Bob, samt en evaluering av prototypen.

I evalueringen så vi at SIP kunne gjøres mer skalerbar, og mer robust mot «single point of failures» ved hjelp ac P2P. SIP ble ikke mer robust mot hindringer i nettverket slik som NAT-er og brannmurer ved hjelp av prototypen.

Kapittel 6

Konklusjon

I denne oppgaven har vi ønsket å finne ut om vi kan gjøre SIP mer robust og skalerbar ved hjelp av P2P.

I oppgaven har vi gått grundig gjennom hvordan telefoni over internett fungerer, med spesielt fokus på signaleringsprotokollen SIP. Vi så også på VoIP-tjenesten «Skype» som har oppnådd en stor popularitet, og jeg beskrev «Skype» som en robust og skalerbar VoIP-tjeneste. Denne robustheten, som jeg beskriver, gjør at «Skype» fungerer godt selv om brukeren befinner seg bak NAT-er eller brannmurer. Traversering gjennom NAT-er og brannmurer så vi at ellers var et stort problem ved bruk av denne typen tjenester. Jeg beskrev også «Skype» som en skalerbar VoIP-tjeneste og begrunnet dette med evnen «Skype» har til å håndtere store brukermasser.

Vi så videre at «Skype» i stor grad var basert på P2P og fildelingstjenesten «KaZaa», og at dette var grunnen til robustheten og skalerbarheten i tjenesten. På grunnlag av dette ønsket jeg å finne ut om det var andre tanker og ideer som kunne videreføres fra eksisterende P2P-tjenester, og over til telefoni over internett.

For å finne ut av dette ble det presentert en grundig gjennomgang av begrepet P2P, og hvorfor dette var et godt alternativ til den tradisjonelle

klient/server-arkitekturen. Vi så også en presentasjon av fordeler og ulemper med P2P.

Videre ble det forklart hvordan arkitekturen og topologien i P2P-nettverk kunne deles inn i to kategorier, og vi så eksempler på P2P-tjenester innefor hver av disse kategoriene. I kategorien strukturerte P2P-nettverk ble vi presentert «lookup»-protokollen «Chord» som dannet et strukturert overlay-nettverk, og vi ble forklart begrepet DHT. Vi så at strukturerte P2P-nettverk er robuste og skalerbare, og at det kan egne seg til telefoni over internett.

I kategorien ustrukturerte P2P-nettverk ble filoverføringsprotokollene «Napster», «Gnutella», «Freenet», «FastTrack/KaZaa» og «BitTorrent» presentert. Disse tjenestene tilhørte underkategoriene sentraliserte, rent distribuerte og hybride ustrukturete P2P-nettverk.

Ved å se på den pågående standardiseringen av P2PSIP og signaleringsprotokollen RELOAD så vi at industrien er langt på vei med å innføre distribuerte VoIP-tjenester ved å lage en felles standard for dette formålet.

Til slutt i oppgaven ble det presentert en prototype som kombinerer både P2P og SIP. Denne prototypen er et «proof of concept», og hadde som formål å kunne gjøre SIP mindre avhengig, eller helt uavhengig av sentrale enheter slik som proxyer og registrarer for å gjøre den mer robust og skalerbar.

Arbeidet med prototypen gav viktig innsikt i hvordan SIP og P2P kan kombineres. Det gav også innsikt i hvilke fordeler dette hadde i forhold til tradisjonell SIP. Vi så at SIP kunne gjøres mer skalerbar ved å benytte strukturerte P2P-nettverk, og at SIP kunne gjøres mer robust mot «single point of failure» ved flytte oppgaver SIP-registrarer og proxyer vanligvis gjør, over i det strukturerte P2P-nettverket.

6.1 Videre arbeid

Arbeidet med prototypen som har blitt presentert i denne oppgaven er kun et «proof of concept» som viser hvordan SIP kan kombineres med P2P. Videreutvikling av denne prototypen er nødvendig for å lage en anvendbar VoIP-tjeneste. Det kunne vært interessant å gjøre noe tilsvarende som prototypen med en allerede eksisterende SIP-klient hvor det blir impelementert P2P-funksjonalitet slik at klienten kan ta del i et P2P-nettverk.

I forbindelse den pågående standardiseringen av P2PSIP og RELOAD kan det også være interessant å lage en implementasjon som følger denne standarden.

Bibliografi

[1] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002. Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630, 5922, 5954, 6026, 6141.

[2] Skype Technologies, S.A. http://www.skype.com/.

[3] Nathaniel Leibowitz, Matei Ripeanu, and Adam Wierzbicki. Deconstructing the Kazaa Network. In *Internet Applications. WIAPP 2003. Proceedings. The Third IEEE Workshop on*, pages 112–120. IEEE, June 2003.

[4] Cullen Jennings, Bruce B. Lowekamp, Eric Rescorla, Salman Abdul Baset, and Henning Schulzrinne. REsource LOcation And Discovery (RELOAD) Base Protocol. draft-ietf-p2psip-base-17, July 2011. Work in progress.

[5] Upkar Varshney and Andy Snow and Matt McGivern and Christi Howard. Voice over IP. *Communications of the ACM*, 45:89–96, January 2002.

[6] Pramode K. Verma and Ling Wang. Voice over Internet Protocol. In *Voice over IP Networks*, volume 71 of *Lecture Notes in Electrical Engineering*, pages 9–23. Springer Berlin Heidelberg, 2011.

[7] H.323 : Packet-based multimedia communications systems. http://www.itu.int/rec/T-REC-H.323/e.

[8] Henning Schulzrinne and Jonathan Rosenberg. The Session Initiation Protocol: Internet-Centric Signaling. *IEEE Communications Magazine*, 38(10):134–141, October 2000.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266.

[10] J. Klensin. Simple Mail Transfer Protocol. RFC 2821 (Proposed Standard), April 2001. Obsoleted by RFC 5321, updated by RFC 5336.

[11] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396 (Draft Standard), August 1998. Obsoleted by RFC 3986, updated by RFC 2732.

[12] J. Rosenberg and H. Schulzrinne. An Offer/Answer Model with Session Description Protocol (SDP). RFC 3264 (Proposed Standard), June 2002. Updated by RFC 6157.

[13] M. Handley, V. Jacobson, and C. Perkins. SDP: Session Description Protocol. RFC 4566 (Proposed Standard), July 2006.

[14] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), July 2003. Updated by RFCs 5506, 5761, 6051, 6222.

[15] C. Huitema. Real Time Control Protocol (RTCP) attribute in Session Description Protocol (SDP). RFC 3605 (Proposed Standard), October 2003.

[16] Hong Liu and Petros Mouchtaris. Voice over IP Signaling: H.323 and Beyond. *IEEE Communications Magazine*, 38(10):142–148, October 2000.

[17] Salman Abdul Baset and Henning Schulzrinne. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. Technical Report CUCS-039-04, Columbia University, New York, NY, USA, September 2004.

[18] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389 (Proposed Standard), October 2008.

[19] Wookyun Kho, Salman Abdul Baset, and Henning Schulzrinne. Skype Relay Calls: Measurements and Experiments. In *INFOCOM Workshops 2008*, pages 1–6. IEEE, April 2008.

[20] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766 (Proposed Standard), April 2010.

[21] Dejan S. Milojičić and Vana Kalogeraki and Rajan Lukose and Kiran Nagaraja and Jim Pruyne and Bruno Richard and Sami Rollins and Zhichen Xu. Peer-to-Peer Computing. Technical Report HPL-2002-57 (R.1), HP Laboratories, Palo Alto, July 2003.

[22] David A. Bryan, Marcia Zangrilli, and Bruce B. Lowekamp. Challenges of DHT Design for a Public Communications System. Technical Report WM-CS-2006-03, College of William & Mary, Guwahati, India, June 2006.

[23] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-Peer Communication Across Network Address Translators. In *2005 USENIX Annual Technical Conference*, pages 179–192, 2005.

[24] J. Rosenberg. Indicating Support for Interactive Connectivity Establishment (ICE) in the Session Initiation Protocol (SIP). RFC 5768 (Proposed Standard), April 2010.

[25] Rüdiger Schollmeie. A Definition of Peer-to-Peer Networking for the Classification of Peer-toPeer Architectures and Applications. In

Proceedings of the First International Conference on Peer-to-Peer Computing, pages 101–102. IEEE, 2001.

[26] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In Hannes Federrath, editor, *Designing Privacy Enhancing Technologies*, volume 2009 of *Lecture Notes in Computer Science*, pages 46–66. Springer Berlin / Heidelberg, 2001.

[27] Matei Ripeanu. Peer-to-Peer Architecture Case Study: Gnutella Network. Technical Report TR-2001-26, The University of Chicago, Department of Computer Science, Chicago, IL, USA, July 2001.

[28] Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble. Measuring and analyzing the characteristics of Napster and Gnutella hosts. *Multimedia Systems*, 9:170–184, 2003.

[29] Ian Clarke, Oskar Sandberg, Matthew Toseland, and Vilhelm Verendel. Private Communication Through a Network of Trusted Connections: The Dark Freenet, 2010.

[30] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Computing Surveys (CSUR)*, 36(4):335–371, 2004.

[31] Stanley Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.

[32] Stefan Saroiu, Krishna P. Gummadi, Richard J. Dunn, Steven D. Gribble, and Henry M. Levy. An analysis of Internet content delivery systems. *SIGOPS Oper. Syst. Rev.*, 36:315–327, December 2002.

[33] Thomas Mennecke. End of the road for overpeer. http://www.slyck.com/story1019.html, December 2005.

[34] Bram Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems*, May 2003.

[35] Robert Morris, M. Frans Kaashoek, David Karger, Hari Balakrishnan, Ion Stoica, David Liben-Nowell, and Frank Dabek. Chord: A Scalable Peer-to-Peer Lookup Service For Internet Applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, 2001.

[36] D. Eastlake 3rd and P. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174 (Informational), September 2001. Updated by RFCs 4634, 6234.

[37] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building Peer-to-Peer Systems with Chord, a Distributed Lookup Service. In *Workshop on: Hot Topics in Operating Systems*, pages 81–86, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[38] Kundan Narendra Singh and Henning Schulzrinne. Peer-to-Peer Internet Telephony using SIP. Technical Report CUCS-044-04, Columbia University, New York, NY, USA, October 2004.

[39] David A. Bryan, Bruce B. Lowekamp, and Cullen Jennings. SOSIMPLE: A serverless, standards-based, P2P SIP communication system. In *Proceedings of the 2005 International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications (AAA-IDEA 2005)*. Citeseer, 2005.

[40] Cullen Jennings, Bruce B. Lowekamp, Eric Rescorla, Salman Abdul Baset, and Henning Schulzrinne. A SIP Usage for RELOAD. draft-ietf-p2psip-sip-06, July 2011. Work in progress.

[41] Heiko Sommerfeldt. PhonerLite. http://www.phonerlite.de.

[42] Damien Sandras. Ekiga. http://ekiga.org.

[43] Microsoft Windows. http://www.microsoft.com.

[44] CAPI - Common ISDN API. http://www.capi.org.

[45] TAPI - Microsoft Telephony API. http://www.tapi.info.

[46] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[47] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. The Secure Real-time Transport Protocol (SRTP). RFC 3711 (Proposed Standard), March 2004. Updated by RFC 5506.

[48] P. Zimmermann, A. Johnston, and J. Callas. ZRTP: Media Path Key Agreement for Unicast Secure RTP. RFC 6189 (Informational), April 2011.

[49] Kundan Narendra Singh. Implementing SIP Telephony in Python. http://39peers.net/download/doc/report.pdf, 2008. Work in progress.

[50] Kundan Narendra Singh. *Reliable, Scalable and Interoperable Internet Telephony*. PhD thesis, Columbia University, New York, USA, June 2006.

[51] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[52] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: A Public DHT Service And Its Uses. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 73–84, New York, NY, USA, 2005. ACM.

[53] Dave Winer. XML-RPC Specification. http://www.xmlrpc.com/spec, 1999.

[54] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: An

Overlay Testbed for Broad-Coverage Services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.

[55] Sean Rhea, Byung-Gon Chun, John Kubiatowicz, and Scott Shenker. Fixing the Embarrassing Slowness of OpenDHT on PlanetLab. In *Proceedings of the Second Workshop on Real, Large Distributed Systems*, pages 25–30, 2005.

[56] PyCrawler. http://code.google.com/p/pycrawler/, 2011.

[57] SQLite. http://www.sqlite.org/, 2011.

Tillegg A

Kildekode

Kildekoden til hele prototypen ligger på følgende adresse:

http://fredrot.at.ifi.uio.no/master/kildekode.tar.gz

Les README for instruksjoner om installasjon.

A.1 Distribuert Hash-tabell Location Service

Listing A.1: Kildekode: dhtls.py

```
1  # -*- coding: utf-8 -*-
2  # A simple location service using OpenDHT gateways to store AoR-entrys.
3
4  import time, os, hashlib
5  from fredrot.opendht import DHT
6  from fredrot.debug import Debug
7  from std.rfc3261 import Header
8  _debug = True
9
10 class LocationDHT(dict):
11
12     def __init__(self):
13         dict.__init__(self)
14         self.dht = DHT()
15         self.dht._test()
```

```
16          self.d = Debug(_debug)
17
18      def save(self, msg, uri, defaultExpires=3600):
19          '''Save the contacts from REGISTER requests.'''
20          self.d._info("DHT-LS Save Uri      : %s" % uri)
21          expires = int(msg['Expires'].value
22                        if msg['Expires']
23                        else defaultExpires)
24          for c in msg.all('Contact'): # for all contacts in the msg
25              self.d._info("DHT-LS Save value    : %s" % c.value)
26              self.dht.put(uri, str(c.value), '', expires)
27          return True
28
29      def locate(self, uri):
30          '''Return all saved contacts for the given SIPURI.'''
31          self.d._info("DHT-LS Locate Uri    : %s" % uri)
32          existing = self.dht.get(uri)
33          values = []
34          for e in existing:
35              if int(e[1]) > 0:
36                  c = Header(e[0],'Contact')
37                  c['expires'] = str(int(e[1]))
38                  values.append(c)
39          self.d._info("DHT-LS Locate return: %s" % values)
40          return values
41
42  if __name__ == '__main__':
43      import doctest
44      doctest.testmod()
```

A.2 OpenDHT

Listing A.2: Kildekode: opendht.py

```
1   # -*- coding: utf-8 -*-
2   # Copyright (c) 2007, Kundan Singh. All rights reserved. See LICENSING
3   # for details.
4   """
5   OpenDHT API borrowed from http://www.opendht.org.
6   The put, get and remove functions are generators so that we don't
7   block the main multitask thread while doing XML-RPC.
8   """
9
10  # Modified by Fredrik Oterholt:
11  # - Made own class for DHT and its functions
```

```
12 # - Added functionality to find suitable gateway
13 # - Added test function to check
14 # - Removed multitask
15
16 import hashlib, os, time, socket, sys #,multitask
17 from xmlrpclib import ServerProxy, Binary
18 from fredrot.findgateway import FindGateway
19 from fredrot.debug import Debug
20
21 _debug  = True
22
23 class DHT:
24     """DHT() takes one argument (gateway). If this is set, it won't
25        use FindGateway to find a gateway"""
26     def __init__(self,gateway=None):
27         self.d = Debug(_debug) # Starts debug-module
28         self.gateway = gateway
29         if self.gateway == None:
30             try: # Using the first gateway found
31                 self.gateway = FindGateway().GetServers()[0]
32             except: # FG failed to find gateway
33                 self.d._warning("OpenDHT: FindGateway failed. Quiting!")
34                 sys.exit() # User needs to restart application,
35                            # causing FG to find a new gateway
36         # Adding 'http://' and port to the gateway-address
37         if not self.gateway.startswith("http://"):
38             self.gateway = "http://%s:5851" % self.gateway
39         self.d._info("OpenDHT: Using gateway %s" % self.gateway)
40
41     def _test(self):
42         """Running a few tests on the DHT to see if its working"""
43         # Enabling debuging while testing
44         dis = self.d.on
45         if not self.d.on:
46             self.d.enable() # Enabled debug output
47
48         self.d._info("OpenDHT _test:")
49         try: # Trying to put a message into the DHT, and then get the same back
50             msg = ('Hostname: %s Time: %s GW: %s' % \
51                   (socket.gethostname(), time.ctime(), self.gateway))
52             key = 'DHTtest-%s' % time.ctime()
53             self.put(key, msg, 'secret', 30)
54             get = self.get(key,1)
55             #print 'msg: %s' % msg
56             #print 'get: %s' % get[0][0]
57             if msg == get[0][0]:
58                 self.d._info("OpenDHT _test: success!")
59             else: self.d._warning("OpenDHT _test: Caution - put- and %s" % \
60                                   "get-values did not match")
61         except: # Failed to put, get or both put and get. Quiting the DHT.
```

```
62              self.d._warning("OpenDHT _test: Timeout on gateway: %s" %
63                              self.gateway)
64              self.d._warning("OpenDHT _test: Quiting!")
65              sys.exit()
66
67
68          # Disable debug after test ending
69          if not dis: self.d.disable()
70
71      def put(self, key, value, secret='default', ttl=180):
72          """
73          Invoke XML-RPC put(H(key), value, H(secret), ttl) on
74          the OpenDHT gateway.
75          Return:
76          True on success and False otherwise."""
77          pxy = ServerProxy(self.gateway)
78          self.d._info("OpenDHT Put Key      : %s" % key)
79          self.d._info("             Value   : %s" % value)
80          self.d._info("             Secret  : %s" % secret)
81          self.d._info("             Expire  : %d" % ttl)
82
83          # Hashes the key
84          key = Binary(hashlib.sha1(key).digest())
85          # Translates the value into binary
86          value = Binary(value)
87          # Hashes the secret
88          shash = Binary(hashlib.sha1(secret).digest())
89
90          if not secret:
91              result = pxy.put(key, value, ttl, 'put.py')
92          else:
93              result = pxy.put_removable(key, value, 'SHA',
94                                         shash, ttl, 'put.py')
95          # pxy.put returns 0 on success.
96          # returns True on success
97          return (result == 0)
98
99      def get(self, key, maxvals=10):
100         """
101         Invoke XML-RPC get_details(H(key), maxvals) on the
102         OpenDHT gateway.
103         Return:
104         a list of tuple(value, remaining-ttl, hash-algorithm, H(secret))
105         where remaining-ttl is int, hash-algorithm is string and
106         H(secret) is lower-case hex of hash of secret starting with 0x.
107         """
108         self.d._info("OpenDHT Get Key      : %s" % key)
109         pxy = ServerProxy(self.gateway)
110         pm = Binary('')
111         key = Binary(hashlib.sha1(key).digest())
```

```
112         result = []
113         i = 0
114         while True:
115             vals, pm = pxy.get_details(key, maxvals, pm, 'get.py')
116             i = i + 1
117             for v in vals:
118                 result.append([v[0].data, v[1], v[2], v[3].data])
119                 self.d._info("             Value    : %s" % v[0])
120                 self.d._info("             Expire   : %s" % v[1])
121             if not pm.data: break
122         return result
123 
124     def remove(self, key, value, secret='default', ttl=180):
125         """
126         Invoke XML-RPC rm(H(key), H(value), secret, ttl) on the
127         OpenDHT gateway.
128         Return:
129         True on success and False otherwise.
130         """
131         self.d._info("OpenDHT Remove %s: %s (%s)" % (key,value,secret))
132         pxy = ServerProxy(self.gateway)
133         key = Binary(hashlib.sha1(key).digest())
134         valueHash = Binary(hashlib.sha1(value).digest())
135         secret = Binary(secret)
136         return (pxy.rm(key, valueHash, 'SHA', secret, ttl, 'rm.py') != 0)
137 
138     def lookup(self, service, maxvals=10):
139         '''
140         TODO:
141         Need to implement the ReDiR interface, but for now use get.
142         '''
143         return get(service, maxvals)
144 
145     def advertise(self, key, value, ttl=180):
146         '''
147         TODO:
148         Need to implement the ReDiR interface, but for now use put.
149         '''
150         return put(key, value, secret='', ttl=ttl)
151 
152 if __name__ == '__main__':
153     Debug()._info("OpenDHT: Finding gateway")
154     dht = DHT()
155     dht._test()
```

A.3 Find Gateway

Listing A.3: Kildekode: findgateway.py

```
1  #!/usr/bin/python -u
2  #
3  # Copyright (c) 2005 Regents of the University of California.
4  # All rights reserved.
5  #
6  # See the file LICENSE included in this distribution for details.
7  #
8  # Simple script to output the fastest opendht servers
9  # Input: number of servers to ping (optional 1st cmd line arg)
10 #        file of opendht servers (optional 2nd cmd line arg)
11 # Output: fastest servers, in order, for up to 5 seconds.
12 #
13 # - Barath Raghavan, 07/06/2005
14 #
15 # $Id: find-gateway.py,v 1.2 2005/08/18 18:49:40 srhea Exp $
16 # Based on: www.opendht.org/find-gateway.py
17
18 ###
19 # Modified by Fredrik Oterholt:
20 # - Made own class for FindGateway and its functions
21 # - Added timeout on the sockets
22 # - Added debug-information
23 # - Added a test-function
24 # - Class stores list with working servers
25 # - Downloads server-list from alternative location
26 #   The old one does not exist anymore
27 ###
28
29 import threading
30 import sys
31 import os
32 import commands
33 import time
34 import socket
35 import random
36 import urllib
37 from fredrot.debug import Debug
38
39 _debug = True
40 _verbose = False
41
42 class SpeedTest(threading.Thread):
43     def __init__(self, servername, serverip, debug):
44         threading.Thread.__init__(self)
```

```
45         self.debug = debug
46         self.servername = servername
47         self.serverip = serverip
48         self.working = False
49 
50     def run(self):
51         try:
52             for x in range(0, 1):
53                 s = socket.socket(socket.AF_INET,
54                                   socket.SOCK_STREAM)
55                 s.settimeout(5.0)
56                 s.connect((self.serverip, 5851))
57                 s.close
58             self.working = True
59             if _verbose:
60                 self.debug._info("FindGateway/ SpeedTest success: %s / %s"
61                                  % (self.servername, self.serverip))
62         except:
63             self.working = False
64             if _verbose:
65                 self.debug._warning("FindGateway/ SpeedTest failed : %s / %s"
66                                     % (self.servername, self.serverip))
67             pass
68 
69 class FindGateway:
70     def __init__(self, numservers=20,filename=None):
71         self.filename = filename
72         self.numservers = int(numservers)
73         self.servers = []
74         self.working = []
75         self.d = Debug(_debug)
76 
77         self.GetList()
78         self.NumServers()
79         self.ShuffleServers()
80         self.TestServers()
81 
82     def GetServers(self):
83         """Returns a list of working servers"""
84         self.d._info("FindGateway/ GetServers: Waiting for server list")
85         # Loop until threads are done
86         while threading.activeCount() > 1:
87             pass
88         if len(self.working) == 0:
89             self.d._warning("FindGateway/ Failed to find gateway!")
90             pass
91         # Returning only the working
92         ret = filter(lambda x: x[1], self.working)
93         return map(lambda x: x[0],ret)
94 
```

```
95      def GetList(self,url='http://fredrot.at.ifi.uio.no/servers.txt'):
96          """Downloading / Reading server list"""
97          # populate server list
98          if not self.filename:
99              self.d._info("FindGateway/ GetList: serverlist from URL: %s" %
100                         url)
101 
102             # Download server-list
103             u = urllib.urlopen(url)
104             while 1:
105                 line = u.readline()
106                 if not line: break
107                 linet = line.split()
108                 self.servers.append(linet[0])
109         else:
110             self.d._info("FindGateway/ GetList: gateways from file: %s" %
111                         filename)
112 
113             # Read server list from file
114             serverfile = open(self.filename, 'r')
115             self.servers = serverfile.read().split()
116         self.d._info("FindGateway/ GetList: Got %d gateways" %
117                     len(self.servers))
118 
119     def NumServers(self):
120         """Numbers of servers to check"""
121         self.numservers = int(self.numservers)
122         if self.numservers > len(self.servers):
123             self.numservers = len(self.servers)
124 
125     def ShuffleServers(self):
126         """Randomize the server list"""
127         random.shuffle(self.servers)
128 
129     def SaveWorking(self,servername,working):
130         """Saving working servers"""
131         self.working.append((servername,working))
132 
133     def TestServers(self):
134         """Runs response test on every server"""
135         serverthreads = []
136         # Starting threads
137         self.d._info("FindGateway/ TestServers: Testing %d gateways" %
138                     self.numservers)
139         for x in range(0, self.numservers):
140             try:
141                 serverthreads.append(
142                     SpeedTest(self.servers[x],
143                               socket.gethostbyname(self.servers[x]),
144                               self.d))
```

```
145                except:
146                    pass
147
148            # Start all servers on the same time
149            for x in serverthreads:
150                x.start()
151
152            # Defining the callback function for the threads
153            def thread_callback(threads=[]):
154                for t in threads:
155                    # save working servers
156                    if t.working:
157                        self.SaveWorking(t.servername,t.working)
158                # kill thread
159                sys.exit()
160
161            # Sets time when callback function is launched
162            t = threading.Timer(2.0, thread_callback, [serverthreads])
163
164            # wait for the threads to finish at the same time
165            for x in serverthreads:
166                x.join()
167
168            t.start()
169
170    if __name__ == '__main__':
171        _debug = True
172        if len(sys.argv) == 1:
173            fg = FindGateway()
174        elif len(sys.argv) == 2: # with limit
175            fg = FindGateway(sys.argv[1])
176        else: # with limit, server list from file
177            fg = FindGateway(sys.argv[1],sys.argv[2])
178
179        servers = fg.GetServers()
180        fg.d._info("Found %d working servers:" % len(servers))
181        for s in servers:
182            fg.d._info(s)
```

Tillegg B

Forkortelser

AoR	Address-of-Record
API	Application programming interface
CA	Certification Authority
CPL	Call Processing Language
DHT	Distribuert Hash-tabell
DNS	Domain Name System
F2F	Friend-to-Friend
GUI	grafisk brukergrensesnitt
GW	Gateway
HTTP	Hypertext Transfer Protocol
ICE	Interactive Connectivity Establishment
IETF	Internet Engineering Task Force
IM	Instant Messages
IP	Internet Protocol
ISP	Internet Service Provider

ITU-T International Telecommunication Union - Telecommunication Standardization Sector

LDAP Lightweight Directory Access Protocol

LS Location Service

NAT Network Address Translation

OS Operativsystem

P2PSIP Peer-to-Peer Session Initiation Protocol

P2PSIP WG P2PSIP Working Group

P2P Peer-to-Peer

PBX Private Branch Exchange

PDA Personal Digital Assistant

PEX Peer Exchange

PSTN Public Switched Telephone Network

QoS Quality of service

RELOAD REsource LOcation And Discovery

RFC Request for Comments

RTCP Real Time Control Protocol

RTP Real-time Transport Protocol

SDP Session Description Protocol

SHA-1 US Secure Hash Algorithm 1

SIP Session Initiation Protocol

SMTP Simple Mail Transfer Protocol

SRTP Secure Real-time Transport Protocol

STUN Session Traversal Utilities for NAT

TTL	Time-to-live
TURN	Traversal Using Relay NAT
UA	User Agent
UAC	User Agent Client
UAS	User Agent Server
URI	Uniform Resource Identifiers
VoIP	Voice over IP
XML-RPC	XML Remote Procedure Call