Sikkert og pålideligt peer-to-peer filsystem

Jacob Nittegaard-Nielsen

Kgs. Lyngby 2004
IMM-THESIS-2004-56

Sikkert og pålideligt peer-to-peer filsystem

Jacob Nittegaard-Nielsen

Kgs. Lyngby 2004

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-THESIS: ISSN 1601-233X

FORORD

Denne rapport er udarbejdet i forbindelse med et eksamensprojekt ved institut for Informatik og Matematisk Modellering på Danmarks Tekniske Universitet. Projektet er udført under vejledning af Christian D. Jensen.

Jeg vil gerne benytte lejligheden til at takke min vejleder Christian D. Jensen for gode råd og konstruktive kommentarer under projektforløbet.

Jesper Kampfeldt og Søren Hjarlvig skal have tak for at dele deres erfaring om tilsvarende projekter.

Desuden en tak til venner og familie for opmuntring og kommentarer. Endelig en tak til Christina Burgos for hjælp til korrekturlæsning.

Kgs. Lyngby, 2004.

_____

Jacob Nittegaard-Nielsen

RESUMÉ

I dette eksamensprojekt undersøges mulighederne for at kombinere secret sharing kryptografi og peer-to-peer teknologi for at konstruere et sikkert og pålideligt backupsystem, hvor både fortrolighed, integritet og tilgængelighed af data er sikret.

Resultatet af undersøgelserne er en model for det ønskede system. Modellen analyseres for at identificere trusler og risici, som systemet skal kunne håndtere. På baggrund af modellen udvikles et design for det foreslåede system og en prototype af det designede system implementeres. Til sidst evalueres prototypen af systemet med henblik på funktionalitet, ydelse og sikkerhed.

Nøgleord: *sikkerhed, kryptografi, certifikater, secret sharing, verificerbart secret sharing, proaktivt secret sharing, peer-to-peer systemer, JXTA.*

ABSTRACT

This thesis explores the possibilities of combining peer-to-peer technology and secret sharing cryptography in order to create a safe and secure backup system in which confidentiality, integrity and availability of data is preserved.

The result of the work is a model of the desired system. The model is analyzed in order to identify threads and risks that the system should handle. A design of the proposed system is developed from the model and a prototype of the system design is implemented. Finally the prototype is evaluated with attention focused on functionality, performance and security.

Keywords: *security, cryptography, certificates, secret sharing, verifiable secret sharing, proactive secret sharing, peer-to-peer systems, JXTA.*

INDHOLDSFORTEGNELSE

1 INDLEDNING

En backup skal altid være tilgængelig, så den kan benyttes, når der er brug for den. Derudover er det vigtigt, at backup'ens integritet er sikret, sådan at de originale data kan gendannes ud fra backup'en. Da de fleste virksomheder tager backup af fortrolige data såsom kundedatabaser og forretningshemmeligheder, er det oftest essentielt, at backup'ens indhold holdes fortrolig, sådan at uvedkommende på intet tidspunkt kan få adgang til data.

Den traditionelle løsning til at sikre en backup's tilgængelighed og integritet er at replikere backup'en ud på flere servere på forskellige lokationer. Da der findes flere kopier, er det ikke kritisk, hvis en enkelt skulle gå tabt. For at sikre backup'ens fortrolighed vil man traditionelt gemme sin backup et enkelt sted, hvortil alt adgang er strengt kontrolleret. Herved kan man hele tiden styre, hvem der har adgang til de fortrolige data. Desværre er disse to traditionelle løsningsmetoder modstridende, da man mindsker tilgængeligheden væsentligt ved at øge fortroligheden og omvendt. I praksis er en ikke-optimal løsning på problemet, at have en række højtsikrede datacentre, hvorimellem data replikeres, og hvor der er strenge krav til adgangskontrollen. Denne løsning er imidlertid meget dyr og bliver således kun benyttet af store virksomheder, der prioriterer deres backup meget højt. I stedet negligerer mange vigtigheden af backup og får i stedet store problemer, hvis uheldet er ude.

Ved hjælp af *secret sharing*, der blev introduceret af Adi Shamir i 1979 [7], kan man dele data op i et antal shares, der hver for sig ikke giver noget information om det originale data, men hvoraf et forudbestemt antal shares kan bruges til at genskabe det originale data.

Formålet med dette projekt er derfor at undersøge, hvordan brugen af *secret sharing* kombineret med *peer-to-peer* teknologi kan benyttes til at lave et backup-system, der både er sikkert og pålideligt og som ikke kræver et alt for stort budget eller en meget kompliceret administration. Projektets fokus vil være på løsningens sikkerhed og robusthed.

Resultatet af disse undersøgelser er en analyse af systemets påkrævede sikkerhed. Analysen benyttes herefter til at udvikle et design for det foreslåede system. Til sidst vil en prototype blive implementeret og evalueret.

Denne rapport består af følgende seks kapitler:

State of the art
Dette kapitel vil give et indblik i de vigtigste eksisterende teknologier, der er benyttet i forbindelse med udarbejdelsen af dette projekt. Kapitlet er inddelt i tre dele. Første del beskriver den klassiske kryptografi. Anden del gennemgår secret sharing mens tredje del omhandler grundlæggende peer-to-peer netværksarkitektur samt en introduktion til JXTA.

Problemanalyse
En kravspecifikation blive udarbejdet på baggrund af en grundlæggende model, der opstilles og analyseres. Herefter beskrives hvordan systemet skal opnå de i kravspecifikationen opstillede sikkerhedsmål. Truslerne mod systemet vil blive beskrevet og analyseret og de forskellige risici analyseres.

Design
Dette kapitel beskriver det udarbejde design af det foreslåede system. Hertil benyttes bl.a. use case beskrivelser og sekvensdiagrammer fra UML-metodologien.

Implementering
Her beskrives implementeringen af prototypen på baggrund af de tidligere kapitler.

Evaluering
Dette kapitel evaluerer den implementerede prototype. Først afprøves prototypens funktionalitet. Derefter evalueres sikkerheden og effektiviteten.

Konklusion
Det samlede arbejde konkluderes og udvidelsesmuligheder diskuteres.

2 STATE OF THE ART

Dette kapitel vil give et indblik i de vigtigste eksisterende teknologier, der er benyttet i forbindelse med udarbejdelsen af dette projekt.

2.1 KLASSISK KRYPTOGRAFI

Kryptografi (græsk: kryptós = skjult, gráphein = at skrive) har været kendt i flere årtusinder. Det primære formål med kryptering har længe været, at to parter kan sende beskeder til hinanden, uden at uvedkommende kan få information om beskedens indhold, selvom de skulle komme i besiddelse af beskeden. Når data krypteres hos afsenderen, benyttes en krypteringsnøgle for at lave en såkaldt ciphertekst. Denne ciphertekst kan siden hen dekrypteres hos modtageren vha. en dekrypteringsnøgle for at genskabe det originale data.

Den klassiske kryptografi opfylder mange forskelligartede sikkerhedsmål. Dette afsnit vil beskrive de klassiske kryptografiske værktøjer, der er benyttet i dette projekt. Udover sikring af fortroligheden (*confidentiality*) vil det således bl.a. blive beskrevet, hvordan man med forskellige klassiske kryptografiske værktøjer kan sikre integritet af data (*integrity*), troværdig autenticitet (*authenticity*) og sikre, at afsenderen er den, han giver sig ud for at være, uden at afsenderen kan benægte at have sendt data (*non-repudiation*).

2.1.1 Symmetrisk kryptering

Ved anvendelsen af symmetrisk kryptering er det (som navnet antyder) den samme krypteringsnøgle, der benyttes til at kryptere data og til at dekryptere data.

Hvis Alice vil sende fortroligt data til Bob, skal følgende gøres:

- Alice og Bob bliver enige om et krypteringssystem.
- Alice og Bob bliver enige om en hemmelig krypteringsnøgle. (Dette er ikke et trivielt problem, da Alice og Bob skal være sikre på, at ingen andre kender nøglen.)
- Alice krypterer data med den hemmelige nøgle og sender det krypterede data til Bob.
- Bob dekrypterer data med den hemmelige nøgle og læser data.

Der findes to typer symmetriske krypteringsalgoritmer - stream-algoritmer og blok-algoritmer. Stream-algoritmer konverterer et symbol af beskedteksten til ciphertekst ad gangen, mens blok-algoritmer opererer på en større mængde af data fra beskedteksten (f.eks. 64 bit). Fordelene ved stream-algoritmerne er, at de er hurtigere og opståede fejl på enkelte bit, vil ikke nødvendigvis påvirke de resterende bits. Den største fordel ved blok-algoritmerne er, at sikkerheden øges ved den diffusion[1], der kan opnås ved at arbejde på en hel blok i stedet for et enkelt symbol.

[1] Diffusion betyder, at informationen fra beskedteksten spredes ud over cipherteksten, sådan at små ændringer i beskedteksten vil ændre store dele af cipherteksten.

Diffusionen opnås typisk ved brug af permutationer, hvor beskedtekstens symboler byttes rundt. Blok-algoritmerne er desuden immune overfor indsættelse af symboler i en blok, da blokken herved vil få en forkert størrelse.

Både stream-algoritmer og blok-algoritmer benytter forvirring (*confusion*), hvilket gør det sværere for en modstander at finde sammenhænge mellem ciphereksten og beskedteksten. Dette gøres ofte vha. substitutioner, hvor symboler i beskedteksten udskiftes med andre symboler.

De mest sikre symmetriske krypteringsalgoritmer er blok-algoritmer, som både benytter en høj grad af diffusion og confusion.

Advanced Encryption Standard
Advanced Encryption Standard (AES) er en symmentrisk blok-algoritme, der er udnævnt som en afløser for den ældre Data Encryption Standard (DES). Det største problem med DES er, at den effektive nøglestørrelse kun er 56 bit, hvilket er for let at bryde ved et brute-force-angreb[2] med nutidens computerkraft. I et forsøg på at udbedre denne svaghed blev 3DES udviklet. Denne betragtes som tilstrækkelig sikker, men har den ulempe, at den skal køre DES-algoritmen tre gange, hvilket er forholdsvist langsomt. Til gengæld opnås en effektiv nøglelængde på 112 bit.

AES har en standard nøglestørrelse på 128 bit som kan udvides til 192 og 256. Ved en nøglestørrelse på 128 bit bruger AES 10 runder (der bruges hhv. 12 og 14 runder ved 192 og 256 bit nøgler). Hver runde består af nogle operationer, der øger diffusion og confusion.

Der findes fire forskellige grundlæggende måder at afvikle algoritmen (modes of operation), som kan tilvælges. Disse blev oprindeligt udviklet til DES, men kan bruges på enhver blok-algoritme (såfremt blokstørrelsen tilpasses). Der findes fire grundlæggende modes of operation:

- Electronic codebook mode (ECB)
- Cipher block chaining mode (CBC)
- Cipher feedback mode (CFB)
- Output feedback mode (OFB)

Ved brug af ECB krypteres hver blok for sig med den samme nøgle. Dette er let og hurtigt, men øger risikoen for at en modstander kan begynde at gætte dele af en besked, ved at sammenligne blokkene, hvis tilstrækkeligt mange blokke er opsnappet. CBC øger sikkerheden ved at lave en XOR mellem den forhenværende ciphertekst-blok og den efterfølgende beskedtekst-blok, før denne krypteres med nøglen. Dette betyder at en ændring i en blok vil ændre samtlige resterende blokke. I CFB og OFB genereres en datastrøm fra krypteringsnøglen, som XOR'es med beskedteksten. Disse opfører sig som stream-algoritmer og fejltolerancen er større end i ECB og CBC. De fire modes of operation har forskellige anvendelsesmuligheder. Generelt betragtes CBC dog som den mest sikre til de fleste formål.

En mere detaljeret gennemgang af de forskellige *modes of operation* er bl.a. beskrevet af Schneier [1] og Stinson [2].

[2] Ved et brute-force-angreb afprøves alle mulige krypteringsnøgler, indtil den rigtige er fundet.

Der er p.t. ingen kendte effektive angreb mod AES og algoritmen betragtes som værende blandt de mest sikre. Det skal dog bemærkes, at den er forholdsvis ny og uprøvet (sammenlignet med f.eks. DES, der har eksisteret i næsten 30 år), og det kan ikke garanteres, at der ikke findes sikkerhedshuller, som kan udnyttes. En sådan garanti kan naturligvis aldrig stilles, men hvis man vurderer, at AES er for uprøvet, kan man f.eks. vælge at bruge 3DES i stedet, da denne har eksisteret længere.

2.1.2 Envejsfunktioner

Envejsfunktioner benyttes i mange dele af kryptografien, bl.a. til asymmetrisk kryptografi, som er beskrevet i afsnit 2.1.4. En envejsfunktion, f(x), er karakteriseret ved, at det er let at udregne f(x) ud fra x, men meget svært at udregne x ud fra f(x) [3]. To meget benyttede envejsfunktioner i kryptografien er det såkaldte faktoriseringsproblem og det diskrete logaritme problem.

Faktoriseringsproblemet baserer sig på, at det er tidskrævende at finde et stort tals primfaktorer. Derimod er det let at multiplicere primfaktorerne for derved at bestemme det pågældende. F.eks. kan det umiddelbart tage noget tid at dele tallet 252601 op i primfaktorer. Haves derimod primfaktorerne 41, 61 og 101, er det let at finde tallet. Problemet bliver naturligvis væsentligt sværere, hvis tallet er i en størrelsesorden af 1024 bit.

I det diskrete logaritmeproblem er grundantagelsen, at det ud fra a, x og q er let at udregne $y = a^x \pmod{q}$. Derimod er det svært at udregne x ud fra a, y og q. F.eks. er det let at udregne y, hvis $3^6 = y \pmod{17}$. Omvendt er det væsentligt sværere at finde x ud fra informationen at $3^x = 15 \pmod{17}$. Ligesom faktoriseringsproblemet bliver dette problem sværere, når størrelsen af tallet øges.

Faktoriseringsproblemet og logaritmeproblemet er nært beslægtede. Kompleksiteten ved at faktorisere primtal og løse det diskrete logaritmeproblem er nogenlunde den samme (med tal af samme størrelse) [1].

Da der findes effektive angreb mod begge problemtyper, er det essentielt at vælge en nøglestørrelse, der er tilstrækkelig stor til at gøre angreb meget svære. Den anbefalede nøglestørrelse afhænger naturligvis af det ønskede sikkerhedsniveau og vil skulle øges med tiden, da udviklingen i hardware (og matematik) vil gøre problemerne lettere at bryde som tiden går. En nøglestørrelse på 1024 bit betragtes som værende sikker for øjeblikket. Det er dog uvidst, hvor længe dette varer ved. Derfor benyttes ofte nøgler på 1576 og 2048 bit, hvis nøglerne skal være mere fremtidssikrede.

Der findes en særlig version af det diskrete logaritmeproblem, der er baseret på elliptiske kurver. Ellipseproblemet har den egenskab, at der ikke findes kendte effektive løsningsmetoder, og de kendte angreb mod det normale diskrete logaritmeproblem kan ikke bruges. For øjeblikket betyder dette, at der kræves væsentlig kortere nøglelængder når der bruges elliptisk kurve

[3] Det antages, at det med begrænset computerkraft er umuligt, at udregne x ud fra f(x). Hvis denne antagelse falder bort, som følge af ny matematisk viden om funktionen, vil alle systemer, der er baseret på den pågældende envejsfunktion, være ubrugelige.

kryptografi (*Elliptic Curve Cryptography* eller bare ECC) frem for de tidligere beskrevne problemer, for at opnå en tilsvarende sikkerhed. Ifølge Stinson [2] kan nøgler helt ned til 160 bit betragtes som sikre i den nærme fremtid.

2.1.3 Envejs hashfunktioner

En hashfunktion $H(B)$ tager et input B af variabel størrelse og returnerer en hashværdi med en fast størrelse. Hashfunktioner bruges til at lave et "fingeraftryk" (også kaldet en *hash*, *checksum* eller *message digest*) af noget data. Dermed kan de give sikkerhed for, at datas integritet er i orden. Hvis data er ændret, vil fingeraftrykket af data ligeledes være ændret (såfremt hashfunktionen fungerer korrekt). Integriteten kan altså undersøges ved at tage et nyt fingeraftryk af data og sammenligne med det gamle fingeraftryk. Her er det naturligvis påkrævet, at det gamle fingeraftryk har været gemt på en sikker måde, så man er sikker på, at det ikke er blevet ændret sammen med data. Typisk vil sammenligningen af fingeraftryk også være væsentlig hurtigere end at sammenligne hele data pga. størrelsen.

Hashfunktioner skal både være envejs, sådan at det oprindelige data ikke kan genskabes ud fra hashen og kollisionsresistente, sådan at det er svært at finde to tilfældige beskeder B og B', hvorved $H(B) = H(B')$.

Kravene skyldes primært det såkaldte fødselsdagsangreb (*birthday-attack*), hvor en modstander forsøger at finde to forskellige beskeder, der begge giver den samme hash. Schneier [1] beskriver dette yderligere. Selvom kravet om kollisionsresistens er overholdt, skal en modstander statistisk kun hashe $2^{(n/2)}$ beskeder, for at finde en kollision, hvor n er hashstørrelsen. Dermed er hashens størrelse vigtig for algoritmens sikkerhed.

To af de mest benyttede hashfunktioner er *Message Digest 5* (MD5) og *Secure Hash Algorithm* (SHA-1). MD5 er dog kun på 128 bit, hvorfor den effektive sikkerhed kun er 64 bit, hvilket ofte er utilstrækkeligt. SHA-1 benytter 160 bit hashlængder og har således en effektiv sikkerhed på 80 bit. SHA-1 er dog væsentligt langsommere end MD5 men betragtes som den mest sikre.

2.1.4 Asymmetrisk kryptering

Ved asymmetrisk kryptering bruges to forskellige nøgler til kryptering og dekryptering. I stedet for én nøgle, som de medvirkende parter er i besiddelse af, har hver part et nøglepar bestående af en privat og en offentlig nøgle. Den private nøgle er hemmelig mens den offentlige nøgle er frit tilgængelig for alle. Når den offentlige nøgle bruges til kryptering, er det kun den private nøgle, der kan dekryptere.

Hvis Alice vil sende fortroligt data til Bob, skal følgende gøres:

- Alice og Bob bliver enige om et krypteringssystem.
- Bob sender sin offentlige nøgle til Alice (eller Alice henter denne fra en database)
- Alice sikrer sig, at nøglen rent faktisk tilhører Bob (beskrevet i afsnit 2.1.5)
- Alice krypterer data med Bob's offentlige nøgle. Herved sikrer hun, at kun indehaveren af Bob's private nøgle kan læse data.

- Bob dekrypterer data med sin private nøgle og læser data.

Udover at sende fortroligt data kan asymmetrisk kryptering bruges til digitale signaturer. Hvis Alice vil sende signeret data til Bob gøres som følger:

- Alice krypterer data med sin private nøgle[4]. Herefter sendes det krypterede data til Bob.
- Bob dekrypterer med Alice's offentlige nøgle, som han allerede forinden har sikret er autentisk. Herved ved Bob, at kun Alice's private nøgle kan have foretaget krypteringen. Hvis Bob stoler på, at nøglerne er korrekte, kan han ligeledes stole på, at data rent faktisk kommer fra Alice. (Alice kan i øvrigt ikke benægte dette, da det krypterede data beviser, at hendes private nøgle er benyttet.)

Asymmetrisk kryptering kan således også sikre *non-repudiation* (under forudsætning af, at der er foretaget en korrekt autentificering af de involverede nøgler.) Integriteten af data bliver også kontrolleret. Hvis en modstander har ændret indholdet af det krypterede data undervejs, vil det være ulæseligt for modtageren, når der dekrypteres.

Asymmetrisk kryptering letter nøglehåndtering (key management) væsentligt, da det kun er de offentlige nøgler, der skal udveksles. Der skal således ikke sendes hemmelige nøgler frem og tilbage (eller laves hemmelige aftaler), før en sikker kommunikation kan oprettes. I stedet opretter hver part sit eget nøglepar og sender sin offentlige nøgle til den anden part. Desuden mindskes antallet af nødvendige nøgler dramatisk i en gruppe. Ved symmetrisk kryptering kræves en separat hemmelig nøgle for hver bruger man vil kommunikere med. Dette giver i alt $(n \cdot (n-1)/2)$ i et system med n brugere (hvis 100 brugere skal kommunikere med hinanden kræves således 4950 nøgler). Benyttes i stedet asymmetrisk kryptering er n nøglepar (et hos hver) tilstrækkeligt til at sikre kommunikationen.

Selvom asymmetrisk kryptering har mange fordele kan det ikke afløse symmetrisk kryptering. Dette skyldes først og fremmest, at asymmetrisk kryptering er meget (ofte mere end 1000 gange) langsommere [1]. I modsætning til symmetrisk kryptering er asymmetrisk kryptering desuden sårbar overfor "valgt tekst" – angreb (*choosen-plaintext*), hvor en modstander har opsnappet en krypteret besked og vil finde ud af, hvilken af n mulige beskedtekster, der er krypteret. Dette gør modstanderen ved selv at kryptere samtlige n muligheder med modtagerens offentlige nøgle (som er tilgængelig for alle) og herefter sammenligne med den opsnappede pakke. Modstanderen vil ikke kunne få fat i den private nøgle (dekrypteringsnøglen) herved, men han vil kunne gætte hvilken tekst, der er sendt.

I praksis benyttes ofte hybrid-krypteringssystemer, der kombinerer symmetrisk og asymmetrisk kryptering. I disse benyttes symmetrisk kryptering til at kryptere selve data (som kan være stort) mens asymmetrisk kryptering benyttes til at kryptere den væsentlig mindre symmetriske krypteringsnøgle og til generering af signaturer af data. Ved denne kombination opnås fordelene fra begge krypteringstyper.

[4] Ved signering i praksis, vil det oftest kun være en hash af data, der krypteres med den private nøgle for at skabe signaturen, da krypteringen ellers vil være meget langsom.

Asymmetriske krypteringssystemer baserer sig på envejs-funktioner. Det diskrete logaritmeproblem benyttes f.eks. i ElGamal krypteringssystemet. Dette er dog ikke lige så udbredt som Rivest-Shamir-Adelman (RSA) systemet som baseres på faktoriseringsproblemet. RSA blev introduceret i 1978 og der er til dato ikke ikke fundet alvorlige fejl, der kompromitterer sikkerheden. For detaljer om RSA henvises til Stinson [2].

2.1.5 Certifikater

Ved brug af asymmetrisk kryptografi er det som nævnt ikke nødvendigt at hemmeligholde den offentlige nøgle. Denne kan sendes frit over en usikker kanal, uden at sikkerheden kompromitteres. Modtageren har dog ingen garanti for, at den afsendte offentlige nøgle rent faktisk stammer fra den forventede afsender.

Før parterne hver især stoler på de modtagne offentlige nøglers autenticitet, er den asymmetriske kryptografi praktisk uanvendelig. Denne tillid kan f.eks. opbygges ved at parterne mødes og udveksler nøglerne vha. disketter eller andet digitalt medie. En anden løsning er at sende digitale fingeraftryk af nøglerne med et andet medie end det medie, der blev benyttet til nøgleudvekslingen (f.eks. telefon, normal post eller e-mail). Disse metoder er dog ganske besværlige (og dermed oftest praktisk uanvendelige) og udnytter ikke alle fordelene ved asymmetrisk kryptografi. Desuden kan selv disse metoder også forfalskes, hvis modstanderen er dygtig nok. Da man aldrig kan være 100% sikker på at autentificering er foregået korrekt, bør autentificeringsmetoden altid afhænge af sikkerhedsbehovet. Dette ses også i praksis, hvor en bank ofte vil kræve at se et pas eller et kørekort, før man kan oprette en konto eller låne penge, mens der blot spørges efter et telefonnummer, hvis man bestiller biografbilletter over telefonen.

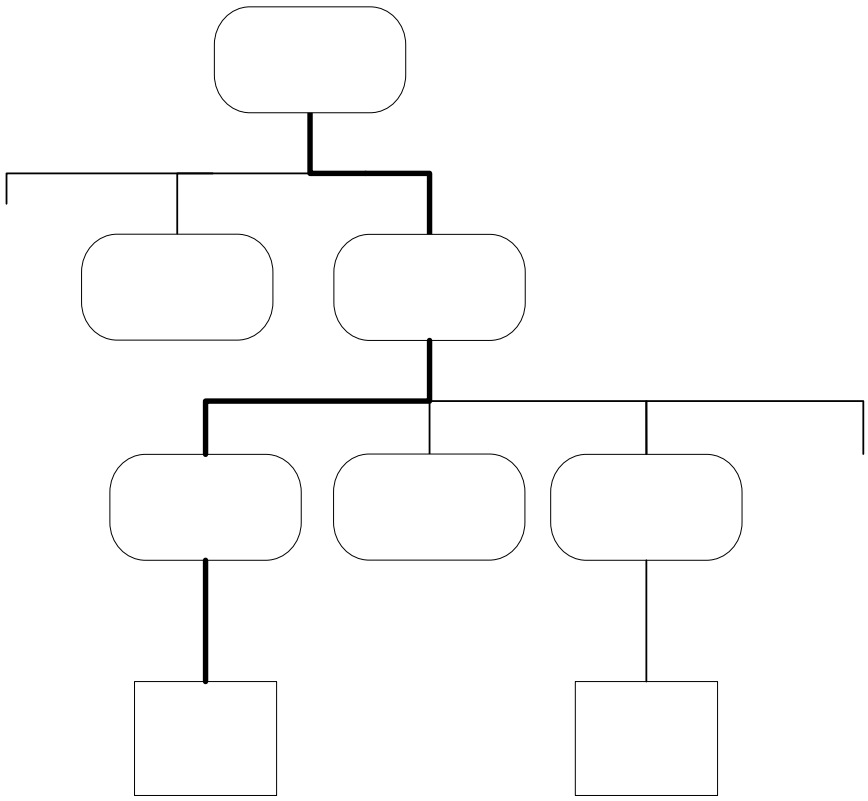
Selvom en personlig udveksling er nøgler oftest giver en bedre sikkerhed, findes i praksis nogle mere anvendelige metoder til at opbygge tillid til andres offentlige nøgler. Den mest udbredte er brugen af certifikater. Et certifikat er et digitalt dokument, der sammenkæder en offentlig nøgle med information om nøgleejerens identitet (ejeren kan f.eks. være et individ eller en organisation). Et certifikat udstedes af en certifikatautoritet (*Certificate Authority* eller bare *CA*). Ved udstedelsen signerer certifikatautoriteten certifikatet, for at bevidne, at nøglen hører sammen med den pågældende identitet. Certifikatautoriteten fungerer som betroet tredjepart, og man er således nødt til at stole på denne, før man kan stole på de certifikater, som den har udstedt.

Der findes flere certifikat-standarder, der bestemmer præcist, hvad et certifikat skal indeholde og hvordan de skal benyttes. Den mest udbredte er X.509 standarden. Heri er det bl.a. bestemt, at et X.509 certifikat skal indeholde et gyldighedsinterval. Man kan dog forestille sig situationer, hvor et certifikat er ugyldigt før udløbstidspunktet. Dette kunne f.eks. være tilfældet, hvis den private nøgle, hørende til den offentlige nøgle i certifikatet, er blevet kompromitteret. I dette tilfælde, bør certifikatet tilbagekaldes, sådan at det ikke kan bruges længere (*certificate revocation*). Dette er dog ingen let opgave, da grundideen bag certifikater bygger på, at det ikke er nødvendigt at være i kontakt med den betroede tredjepart (certifikatautoriteten), når man først kender dennes eksistens og offentlige nøgle. Hvis certifikatautoriteten og brugerne ikke er i kontakt med hinanden, har de ingen mulighed for at viderebringe information om tilbagekaldte certifikater. Oftest håndteres problemet ved at certifikatautoriteten gemmer en liste over kendte ugyldige

certifikater. Denne liste kan de forskellige brugere checke med jævne mellemrum, for at sikre sig, at de anvendte certifikater er gyldige.

Ved anvendelse af certifikater er systemets reelle sikkerhed bestemt af de involverede certifikatautoriteter, da disse bestemmer autentificeringsmetoden. Det er altså op til certifikatautoriteterne, hvor grundigt man skal identificere sig selv for at få udstedt et certifikat. Hvis en certifikatautoritet udsteder et certifikat til enhver, der sender en forespørgsel via e-mail, uden yderligere foranstaltninger, vil certifikatet i praksis være ubrugeligt.

X.509 modellen inkluderer beskrivelser af, hvordan certifikatautoriteter kan opbygge et større hierarki, hvor den øverste certifikatautoritet (rod-autoriteten) i hierarkiet uddelegerer noget ansvar til andre "lavere stående" certifikatautoriteter (under-autoriteter). Rod-autoritetens eget certifikat er signeret af rod-autoriteten selv. Det selvsignerede certifikat er herefter benyttet til at signere under-autoriteternes certifikater. I disse certifikater er det angivet, at brugeren (dvs. den pågældende under-autoritet) selv har ret til at udstede certifikater. En under-autoritet kan således både have almindelige brugere og lavere stående under-autoriteter under sig i hierarkiet. På Figur 1 herunder er et certifikatautoritetshierarki afbildet.



Figur 1 – Her ses et certifikatautoritets hierarki. Øverst er Rod-autoriteten og herunder findes under-autoriteter, som har ansvar for hhv. USA og Europa. Europa har signeret CA-certifikater til Danmark, Tyskland og England. Alice har fået udstedt et certifikat af Danmark CA'en og Bob's certifikat er udstedt af England CA'en. Alices certifikatautoritet-kæde er afbildet med fed og består således af hendes eget certifikat, Danmark CA'ens certifikat, Europa CA'ens certifikat og til sidst Rod CA'ens certifikat.

Hvis Alice og Bob vil kommunikere og tilhører samme certifikatautoritet, er det let for Alice at verificere certifikatautoritetens signatur på Bob's certifikat for at afgøre, om det er gyldigt (Hun kan evt. også kontakte certifikatautoriteten for at undersøge, om Bob's certifikat er blevet kompromitteret eller lign.). Hvis Alice og Bob ikke kender samme certifikatautoritet (som på figuren herover), følges certifikatautoritet-kæden op mod roden, indtil en fælles kendt certifikatautoritet mødes (i dette tilfælde Europa CA'en). Dermed kan autentificeringen forgå, hvis blot Alice og Bob er en del af det samme certifikatautoritetshierarki.

Selvom certifikater er en af de mest udbredte metoder til autentificering af brugere i et distribueret system, er det ikke den eneste. En anden foreslået løsning er et såkaldt *Web of Trust.* I et *Web of Trust* er der ikke brug for en central autoritet (eller et centralt hierarki). I stedet signerer man hinandens nøgler, når man stoler på dem. Når man vil kommunikere med en ny bruger, undersøger man, om dennes offentlige nøgle er signeret af nogen, som man kender og stoler på. Hvis dette er tilfældet, kan man vælge selv at stole på nøglen. På den måde spredes tilliden ud gennem systemet. Schneier [1] beskriver *Web of Trust* yderligere.

Fordelene ved et *Web of Trust* er, at systemet ikke afhænger af en central autoritet. Man har dog ingen garanti for, at man kan autentificere en nøgle, da dette kræver direkte eller indirekte kendskab til nogen, som allerede kender nøglen. Ulempen ved at benytte certifikater og certifikatautoriteter er den centrale styring. De involverede parter er således nødt til at kende certifikatautoriteten (eller som minimum en autoritet fra det samme hierarki). Hverken certifikatløsningen eller *Web of Trust* har problemer med *Single point of failure*. En certifikatautoritet kan ganske vist bryde sammen, men dette anses ikke for et reelt problem, da disse ikke behøver at fungere korrekt efter et certifikatet er udstedt, idet certifikatet benyttes uden kontakt med certifikatautoriteten (dette gælder dog ikke ved tilbagekaldelse af certifikater).

2.1.6 Nøglelængder

I moderne applikationer benyttes ofte flere slags kryptografi. Som udgangspunkt er det et mål for designerne af et system, at de forskellige slags kryptografi er omtrent lige svære at bryde, da den samlede kryptografiske styrke ikke er stærkere end det svageste led. Hvis nøglerne bliver for korte kan de brydes og hvis de bliver for lange koster det unødvendigt mange beregningsressourcer.

Hvis den eneste mulighed for et angreb på kryptografien er et brute-force-angreb, hvor alle nøglekombinationer afprøves, bør en 128 bit nøgle være tilstrækkelig til stort set hvad som helst. Med verdens samlede computerkraft vil det tage mere end 1000 gange universets alder at gennemføre et sådant angreb [1]. Mod AES findes ingen kendte angreb, hvorfor en nøglestørrelse på 128 bit betragtes som tilpas de fleste steder.

De fleste asymmetriske krypteringssystemer kræver væsentlig længere nøgler, da de baseres på envejsfunktioner, jf. afsnit 2.1.2. Systemer som RSA (baseret på faktoriseringsproblemet) og ElGamal (baseret på det diskrete logaritmeproblem) skal således bruge nøglestørrelser på mindst 2048 bit for at opnå nogenlunde den samme kryptografiske sikkerhed som en 128 bit symmetrisk

AES kryptering [1]. Systemer baseret på elliptisk kurve kryptografi er dog en undtagelse, og disse kan klare sig med væsentlig kortere nøgler.

Brugen af hashalgoritmer bør også overvejes, da disse let kan blive det svageste led. Dette skyldes de mulige fødselsdagsangreb, hvor modstanderen blot skal finde to forskellige beskeder, der giver den samme hash, hvilket er væsentligt hurtigere end at finde et match til en bestemt besked.

2.2 SECRET SHARING

Secret sharing er et vigtigt redskab indenfor kryptografi og distribuerede systemer. Det blev introduceret i to forskellige og uafhængige versioner i 1979 af henholdsvis Adi Shamir [7] og George Blakley [8]. Siden hen er mange andre forslag dukket op, hvoraf de fleste bygger på de samme grundlæggende tanker. Ideen bag secret sharing er at dele en hemmelighed mellem n deltagere, sådan at k (hvor $2k\text{-}1 \leq n$) af disse deltagere kan genskabe hemmeligheden. Mindre end k deltagere må imidlertid ikke kunne få noget som helt at vide om hemmeligheden. Dette kaldes et (n,k)-*threshold cryptography scheme* pga. denne grænse, som skal nås, før rekonstruktion kan foregå[5]. Den grundlæggende antagelse bag secret sharing er altså, at deltagere (servere) kan blive kompromitterede eller bryde sammen, men at maksimalt k-1 servere er kompromitteret eller på anden vis utilgængelige på samme tid. Derved vil der altid være mindst k korrekte servere tilbage. På den måde vil både hemmelighedens tilgængelighed (availability) og fortrolighed (confidentiality) være sikret. Secret sharing har mange forskelligartede anvendelsesformål. Oftest bliver det dog sammenkædet med nøglehåndtering (*key management*) eller andre operationer, der kræver en kombination af robusthed og sikkerhed. I dette kapitel vil forskellige aspekter af secret sharing blive præsenteret og diskuteret.

2.2.1 Grundlæggende secret sharing

Shamir's ide er baseret på polynomiers interpolation. En vigtig egenskab ved polynomier er, at k forskellige punkter entydigt bestemmer et polynomium af grad k-1. F.eks. bestemmer 2 punkter entydigt en linie mens 5 punkter entydigt bestemmer et 4. gradspolynomium. En anden vigtig egenskab, som udnyttes til at lave secret sharing er, at k-1 punkter ikke giver nogen brugbar information om et polynomium af grad k-1. Dette skyldes, at man kan lave uendelig mange polynomier af grad k-1 ud fra k-1 punkter. Det sidste punkt mangler altså, for at kunne bestemme hvilket af de mulige polynomier, der er tale om.

Hemmeligheden distribueres således ved, at den oprindelige indehaver (ofte kaldet *dealeren*) bestemmer de n deltagere, som skal være med til at dele hemmeligheden. Desuden bestemmes et

[5] I den anvendte litteratur er der uenighed om, hvordan et *threshold scheme* skal repræsenteres. Nogle steder er k tolerancen, sådan at k+1 shares er nødvendige for at rekonstruere (værdien k beskrives også med symbolet t i nogle dele af litteraturen). Det er ligeledes forskelligt, om n eller k nævnes først. I dette projekt benyttes et (n,k) *threshold scheme* til at repræsenterer modellen, hvor k er det antal shares, der er nødvendige for at gennemføre en rekonstruktion. Antallet af deltagere er n og tolerancen vil således være k-1.

primtal q, hvor $q >$ hemmeligheden[6] (det kan med rimelighed antages, at hemmeligheden er et heltal eller kan laves til et heltal). Dernæst laves et polynomium af (k-1)'te grad:

$$s(x) = a_0 + a_1 * x + a_2 * x^2 + ... + a_{k-1} * x^{(k-1)} \pmod{q}$$

I dette polynomium er a_0 (og dermed $s(0)$) hemmeligheden som ønskes distribueret. De øvrige koefficienter er valgt tilfældigt mellem 0 og q. For hver deltager x_i, i:1..n, udregnes $s(x_i)$ og værdien sendes til den pågældende deltager[7] (som kender sit eget index i samt primtallet q). Den modtagne værdi angiver et punkt i det hemmelige polynomium og er således denne deltagers del (share) af hemmeligheden. Når alle deltagere har modtaget et share sletter dealeren den oprindelige hemmelighed og det hemmelige polynomium, der afslører hemmeligheden.

Herved skal der minimum k shares til at genskabe det oprindelige polynomium vha. interpolation. Efter en interpolation kan $s(0) \pmod{q}$ evalueres for at finde hemmeligheden [7]. Rekonstruktionen kan enten foregå offentligt, hvorved alle lærer hemmeligheden på samme tid eller hos en betroet enhed (ofte kaldet en *combiner*), hvis det ikke er ønskværdigt, at alle skal kende hemmeligheden.

Der er mange fordele ved Shamirs model, hvorfor den grundlæggende ide stadig bruges i de fleste secret sharing sammenhænge. Først og fremmest er Shamirs secret sharing simpel og effektiv (deling og rekonstruktion foregår i polynomisk tid). Desuden giver den informationsteoretisk sikkerhed uden antagelser. Informationsteoretisk sikkerhed er en modsætning til såkaldte beregningssikre systemer (*computationally secure systems*), der antager, at en modstanders computerkraft er begrænset. Et eksempel herpå kunne være RSA, der som nævnt bygger på antagelsen om, at modstanderen ikke har tilstrækkelig computerkraft til at faktorisere store primtal. Med informationsteoretisk sikkerhed kan systemet ikke brydes selvom en modstander skulle have ubegrænset computerkraft til rådighed.

George Blakleys secret sharing [8] er ikke blevet ligeså udbredt som Shamir's. Dette skyldes primært, at den er mindre effektiv og mere kompliceret uden at yde en bedre sikkerhed end Shamirs version. I Blakleys secret sharing benyttes et punkt i et k-dimensionalt rum til at opbevare hemmeligheden. De hemmelige shares består af en (k-1)-dimensionalt "plan", således at skæringen mellem disse giver hemmeligheden.

Selvom grundideen i Shamirs secret sharing er god og har mange fordele, er den ikke uden problemer. Et af de første problemer man støder på, er muligheden for at snyde systemet med falske share-værdier. Når dealeren sender shares til medlemmerne har disse ingen mulighed for at verificere, at deres share er korrekt. Ligeledes kan det ikke lade sig gøre, at verificere korrektheden af shares, når hemmeligheden skal rekonstrueres. En deltager, der ikke ønsker hemmeligheden afsløret, kan således sende et falskt share, hvilket vil medføre, at den forkerte hemmelighed bliver rekonstrueret. Hvis rekonstruktionen foregår offentligt, vil den falske

[6] Shamir [7] bruger selv p i stedet for q til at beskrive primtallet. I denne rapport benyttes q for at være i overensstemmelse med Feldmans verificerbare secret sharing, jf. afsnit 2.2.2.

[7] Hvilket som helst punkt kan bruges. Det eneste krav er, at de forskellige deltagere får forskellige punkter, dvs. forskellige evalueringer af polynomiet $s(x)$. Evalueringerne må naturligvis heller ikke være fra $s(0)$, da deltagerens share derved vil være den delte hemmelighed.

deltager desuden få adgang til alle de andres shares, hvorefter denne selv kan rekonstruere hemmeligheden på et ønsket tidspunkt. For at imødekomme disse problemer er der kommet forskellige forslag til udvidelser af den grundlæggende secret sharing. Nogle af de vigtigste vil blive præsenteret i det følgende.

2.2.2 Verificerbart secret sharing

Verificerbart secret sharing (VSS) har til formål at udvide den grundlæggende secret sharing med en mulighed for at verificere korrektheden af shares, uden at rekonstruere hemmeligheden og uden at afsløre information om de involverede shares. Dermed vil systemet kunne modstå aktive angreb, hvor en modstander forsøger at snyde protokollen. De umiddelbare problemer med Shamirs secret sharing kan således undgås. Dette sker dog på bekostning af nogle af fordelene ved den grundlæggende løsning.

To af de mest kendte VSS er foreslået af P. Feldman [9] og af T.P. Pedersen [10]. Begge bygger på at det diskrete logaritmeproblem er svært at løse. Desuden udnyttes de homomorfiske egenskaber ved potensfunktioner, der betyder, at $x^a * x^b = x^{a+b}$. Både Feldmans og Pedersens VSS er ikke-interaktive (*non-interactive*), hvilket vil sige, at ingen ekstra interaktivitet er påkrævet mellem dealer og deltagere eller deltagerne imellem, når shares skal verificeres.

I Feldmans VSS [9] vælges først to store primtal, p og q, således at $p = mq + 1$, hvor m er et lille heltal. Z_p er verificeringsdomænet mens Z_q er det egentlige secret sharing domæne svarende til q i Shamirs secret sharing. Hemmeligheden, der skal deles, skal således stadig være mindre end q. Desuden vælges g som en generator[8] mod q. Herefter foregår distribueringen af hemmeligheden som i Shamirs secret sharing. Udover selve det hemmelige share til hver at de deltagende servere, broadcaster dealeren verificeringsinformation. Denne information består af:

$$g^{a_0}, g^{a1}, .., g^{a_{k-1}} \pmod{p}$$

Selve hemmeligheden (a_0) er således en del af verificeringsinformationen, men da det diskrete logaritmeproblem (antageligt) ikke kan løses, kan de enkelte servere hverken få information om det hemmelige polynomium eller om hemmeligheden.

Hver server kan nu benytte verificeringsinformationen til at checke, at det modtagne share er korrekt. Dette gøres ved at checke, at:

$$g^{s(x_i)} \stackrel{?}{=} \prod_{j=0}^{k-1} (g^{a_j})^{x_i^j} \pmod{p}$$

Hvis verificeringen fejler sendes beskyldninger mod afsenderen. Hvis k fejl opstår, er protokollen fejlet, da dette er i modstrid med modellens grundantagelse.

[8] En generator kaldes også for en primitiv rod. g er generator mod q, hvis der eksisterer et a, hvor $g^a = b \pmod{q}$ for alle b mellem 1 og q-1. Den ønskede generator kan her findes ved at teste, om $g^q = 1 \pmod{p}$. Se evt. Schneiers beskrivelse [1] for en uddybende forklaring.

Ved distribueringen af shares er denne kontrol dog ofte unødvendig, da man i de fleste modeller har god grund til at stole på, at dealeren ikke forsøger at snyde (det er trods alt dealeren, som kender hemmeligheden i forvejen).

Da verificeringsinformation er den samme hos alle servere kan denne dog også benyttes til verificering af shares ved genskabelsen af hemmeligheden. Dette er meget anvendeligt, da kompromitterede deltagere dermed ikke kan ødelægge rekonstruktionen ved at sende falske shares. Disse vil blive opdaget og bortsorteret.

For at illustrere dette gives her et lille eksempel på, hvordan distribueringen og rekonstruktionen foregår i Feldmans model:

Eksempel: Distribuering
Først vælges konstanterne $p = 283$, $q = 47$ og $g = 54$, som alle deltagere kender.
Konfigurationen vælges som et (5,3)-threshold scheme, sådan at der er fem deltagere, $n_1..n_5$, hvoraf tre kan gendanne hemmeligheden.

Hemmeligheden er tallet 6 og denne er kun kendt af ejeren af hemmeligheden (dealeren). Ejeren laver nu et polynomium s(x) af grad k-1, hvor koefficienterne er valgt tilfældigt mod q, bortset fra a_0, som er hemmeligheden. I dette eksempel er $s(x) = 6 + 5x + 2x^2 \pmod q$.

Ud fra s(x) kan ejeren nu udregne de forskellige deltageres shares.
Deltageren n_1 får sharet $x_1 = s(1) = 13 \pmod q$
Deltageren n_2 får sharet $x_2 = s(2) = 24 \pmod q$
Deltageren n_3 får sharet $x_3 = s(3) = 39 \pmod q$
Deltageren n_4 får sharet $x_4 = s(4) = 11 \pmod q$
Deltageren n_5 får sharet $x_5 = s(5) = 34 \pmod q$

Udover disse shares laver ejeren verificeringsinformationen bestående af værdierne:
$[g^6, g^5, g^2] \pmod p = [54^6, 54^5, 54^2] \pmod p = [155, 71, 86]$.
Denne verificeringsinformation broadcastes nu til de fem deltagere, som herved kan kontrollere rigtigheden af deres share.

Et eksempel på denne kontrol er deltageren n_4, som kontrollerer, at:

$$54^{11} \stackrel{?}{=} 155^{(4^0)} * 71^{(4^1)} * 86^{(4^2)} = 155 * 71^4 * 86^{16} \pmod p$$

Hvis dette passer, er sharet x_4 korrekt.
Når alle har kontrolleret deres share er distribueringen færdig.

Eksempel: Rekonstruktion
Deltageren n_2 vil gerne genskabe hemmeligheden. Derfor beder den de andre deltagere om at sende deres share. Deltageren n_5 svarer imidlertid ikke og n_3 synes ikke hemmeligheden skal genskabes, hvorfor denne sender værdien 43 i stedet for sit korrekte share. Deltagerne n_1 og n_4 sender dog deres rigtige share, hvorfor der i alt er tre korrekte shares tilstede (x_1=13, x_2=24 og x_4=11) samt et falskt share (x_3=43).

Deltageren n_2 kontrollerer nu de modtagne shares (eget share undtaget i dette eksempel).

x_1: $54^{13} \stackrel{?}{=} 155^{(1^0)} * 71^{(1^1)} * 86^{(1^2)} = 155 * 71 * 86 \pmod{p}$

x_3: $54^{43} \stackrel{?}{=} 155^{(3^0)} * 71^{(3^1)} * 86^{(3^2)} = 155 * 71^3 * 86^9 \pmod{p}$

x_4: $54^{11} \stackrel{?}{=} 155^{(4^0)} * 71^{(4^1)} * 86^{(4^2)} = 155 * 71^4 * 86^{16} \pmod{p}$

Herved opdater n_2, at x_3 er falsk, da de to sider af lighedstegnet giver hhv. 216 og 244 (de er altså ikke ens). Derfor ignoreres dette share. I stedet benyttes de tre korrekte shares (x_1=13, x_2=24 og x_4=11) til at genskabe hemmeligheden vha. Lagrange interpolation. Når punkterne (1,13), (2,24) og (4,11) interpoleres for at finde s(0) fås værdien -29/3 = -29∗16 = 6 (mod q)[9].

Dermed er rekonstruktionen fuldendt, selvom et falsk share var tilstede, og en anden deltager slet ikke svarede.

Fordelene ved Feldmans VSS er, at det opnår verificerbarhed vha. simple og overskuelige midler. Feldmans VSS reducerer dog den informationsteoretiske sikkerhed fra Shamirs secret sharing, da fortroligheden nu afhænger af en modstanders mulighed for at løse det diskrete logaritmeproblem. Ifølge Stinson [2] er det muligt at bestemme de lavest betydende bits i det diskrete logaritmeproblem, hvorved en del af hemmeligheden afsløres. Dette problem kan imødegås ved at lave en envelope, hvor den egentlige hemmelighed kodes ind i de mest betydende bits af den hemmelighed, der bliver delt. Alternativt kan Pedersens VSS benyttes. Dette har mange af de samme egenskaber som Feldmans VSS men bevarer den informationsteoretiske sikkerhed, ligesom Shamirs oprindelige secret sharing. Til gengæld er det også en smule mere kompliceret.

Pedersens VSS [10] benytter ligeledes konstanterne p, q fra Feldmans VSS. Desuden benyttes elementerne g og h (mod q) til at lave et såkaldt *commitment scheme*, hvor dealeren binder sig til en hemmelighed s. Først vælges et tilfældigt tal t (mod q) og derefter udregnes commitment-værdien $E_0 = E(s,t) = g^s * h^t$.

Nu laves to tilfældige polynomier af grad k-1, hvor koefficienterne er mellem 0 og q. Det første er magen til polynomiet fra Shamirs secret sharing (og Feldmans VSS). Det andet benytter tallet t. Polynomierne ser således ud:

$s(x) = a_0 + a_1 * x + a_2 * x^2 + ... + a_{k-1} * x^{(k-1)}$, hvor $a_0 = s \pmod{q}$
$t(x) = b_0 + b_1 * x + b_2 * x^2 + ... + b_{k-1} * x^{(k-1)}$, hvor $b_0 = t \pmod{p}$

For i = 1..k-1 laves nu commitmentværdierne $Ei = E(a_i,b_i)$. Alle commitmentværdierne broadcastes og til hver deltager n_i sendes $(s(i),t(i))$, hvor $s(i)$ er selve sharet.

[9] At dividere med 3 og multiplicere med 16 er det samme (mod 47), idet 3 ∗ 16 = 1 mod 47

Hver deltager kan nu kontrollere sit share (ligesom i Feldmans VSS) ved at verificere, at:

$$E(s(i),t(i)) \stackrel{?}{=} \prod_{j=0}^{k-1} E_j^{i^j}$$

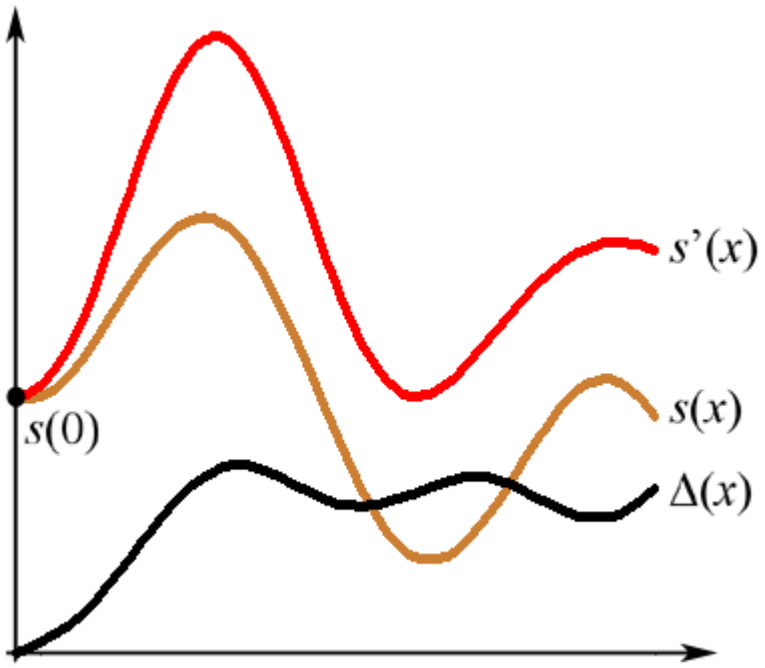
For dybdegående information om Pedersens VSS henvises til andre kilder [10].

2.2.3 Proaktivt secret sharing

Tages tiden med i betragtningen, er det ofte realistisk at antage, at en modstander ikke kan få fat i mere end *k* shares i løbet af et begrænset tidsrum og dermed ikke kan kompromittere systemet. Det er dog ikke altid realistisk at bevare denne antagelse, hvis tidsrummet gøres stort (eller uendeligt). De ovennævnte modeller giver ingen brugbare løsninger på dette problem. I slutningen af et sikkert tidsinterval kan man naturligvis vælge at rekonstruere hemmeligheden og siden hen dele den igen med en række andre shares, men dette vil give en uønsket sikkerhedsrisiko, da hemmelighedens fortrolighed er sårbar, mens den er rekonstrueret. Desuden vil det kræve unødvendigt meget administration af systemet. Derfor har man udviklet en særlig type secret sharing, kaldet *proaktive secret sharing*, som har til formål at vedligeholde systemets sikkerhed over flere tidsperioder. Der findes flere modeller for proaktivt secret sharing. Denne gennemgang vil dog primært afspejle en model udviklet af IBM [11], som bygger videre på Shamirs secret sharing [7] og Feldmans VSS [9].

Proaktivt secret sharing forbindes ofte med en opdateringsrutine, der sørger for at alle shares opdateres i slutningen af en tidsperiode, sådan at de nye shares stadig repræsenterer den samme hemmelighed og sådan at gamle shares bliver ubrugelige. Denne opdatering skal kunne foregå, selvom op til *k*-1 deltagende servere er kompromitterede. De kompromitterede servere kan undlade at følge protokollen, men de kan også lave aktive angreb mod systemet, for at få fat i (eller destruere) hemmeligheden.

Opdateringen forgår ved at et tilfældigt opdateringspolynomium $\Delta(x)$ genereres (flere opdateringspolynomier kan med fordel bruges på samme tid). $\Delta(x)$ har det samme antal koefficienter som det oprindelige hemmelige polynomium $s(x)$ og bliver desuden konstrueret sådan, at $\Delta(0) = 0$. $\Delta(x)$ lægges til $s(x)$ for at skabe det nye hemmelige polynomium $s'(x)$. Dette er illustreret på Figur 2 herunder.



Figur 2 – Figuren viser, hvordan systemets shares bliver opdateret. Det tilfældigt genererede opdateringspolynomium $\Delta(x)$, hvor $\Delta(0) = 0$, lægges til det oprindelige polynomium $s(x)$, hvorved det nye polynomium $s'(x)$ fås. Hemmeligheden er uændret, da $s(0) = s'(0)$, men gamle shares fra $s(x)$ er ubrugelige sammen med de nye shares fra $s'(x)$.

Da $s(x)$ ikke findes mere på opdateringstidspunktet[10], foregår sammenlægningen ved, at den server, der genererede $\Delta(x)$, bestemmer en række opdateringsshares på samme måde som den oprindelige uddeling af shares fra $s(x)$ foregik: For hver deltager i:1..n udregnes $\Delta(x_i)$ og værdien sendes til den pågældende deltager, som opdaterer sit share ved at lægge sin nye og gamle værdi sammen. Det nye polynomium $s'(x)$ har nu følgende egenskaber:

- $s'(0) = s(0) + \Delta(0) = s(0) + 0 = s(0)$
  Herved er den originale hemmelighed bevaret.

- Hvis $x \neq 0$ (og $\Delta(x) \neq 0$):
  $s'(x) = s(x) + \Delta(x) \neq s(x)$
  Dermed er det grundlæggende polynomium ændret. Hvis hemmeligheden skal rekonstrueres er det således det nye polynomium, der skal interpoleres, hvorfor gamle shares vil være forkerte og ubrugelige i denne sammenhæng.

Opdateringen kan foregå ved at dealeren alene opdaterer systemet. Dette er dog et problem, hvis denne bliver kompromitteret, da hemmeligheden derved kan afsløres, selvom mindre end k-1 servere er kompromitteret i hver tidsperiode. Alternativt kan alle være med til at opdatere. Dette foregår ved, at alle laver hver deres opdateringspolynomium og sender værdierne til hinanden.

[10] $s(x)$ blev slettet efter den oprindelige deling for at undgå, at en enkelt deltager alene havde tilstrækkelig information til at genskabe hemmeligheden.

Opdateringsfasen kan kombineres med VSS for at sikre verificerbarhed gennem hele protokollen og derved gøre den resistent overfor aktive angreb. Verificeringen af opdateringsshares kan foregå på samme måde, som verificering af normalt udelte shares. Forskellige verificeringsmetoder kan benyttes, men i modellen fra IBM [11] benyttes Feldmans VSS.

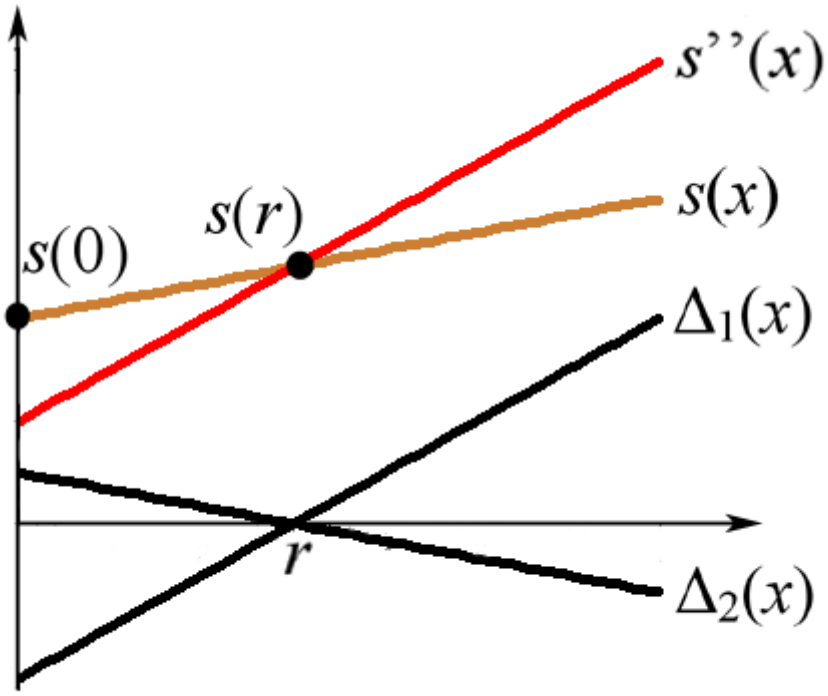
Selvom den beskrevne share-opdatering er grundstammen i et proaktiv secret sharing system, kræves yderligere funktionalitet, for at et proaktivt secret sharing system i praksis kan overleve i (uendelig) mange tidsperioder. Dette er særligt tilfældet, når det antages, at en kompromitteret server ikke kun kan give hemmeligt share-data til modstanderen, men er helt under dennes kontrol, hvilket ikke er urealistisk. Proaktive secret sharing modeller bygger typisk på en antagelse om, at kompromitterede servere kan blive "renset" (vha. en slags reboot-rutine) for at slippe af med modstanderen og derefter at deltage igen på normal vis.

Systemet skal således tage højde for, at servere, der ikke længere er kompromitterede, skal have korrekt information, dvs. et korrekt share. Shares kan være blevet slettet eller modificeret, mens serveren var kompromitteret. Et share kan også gå tabt, hvis en server fejler lokalt, genstartes eller udskiftes med en ny server. Derfor skal systemet kunne checke integriteten af de enkelte serveres shares for herved at sikre, at en modstander ikke gradvist (over flere tidsperioder) destruerer n-k+1 shares (eller at shares på anden vis går tabt), hvorved hemmeligheden ikke længere kan rekonstrueres af de k-1 tilbageværende shares. Når en fejl findes skal systemet ligeledes kunne genskabe det mistede share, uden at deltagere får information om andet end deres eget share.

Detekteringen af forsvundne eller modificerede shares kræver, at alle kan verificere alles shares. Dette kan eksempelvis opnås ved at sørge for, at alle hele tiden har en verificeringsværdi af alles shares. Ved brug af Feldmans VSS vil disse værdier være g^{x_i}, hvor x_i er den i'te servers share. Når detekteringsrutinen sættes i gang (ved overgangen fra en tidsperiode til en anden), starter alle servere med at udregne en ny verificeringsværdi (g^{x_i}) for deres eget share og sende denne til de andre servere, der sammenligner med deres egen værdi for den pågældende server. Hvis et share er blevet modificeret i løbet af tidsperioden, vil verificeringsværdien være forkert i forhold til resten af systemets viden. Verificeringsværdierne afslører ikke noget om det hemmelige share, så længe logaritmeproblemet ikke kan løses.

Når et forkert eller manglende share findes, skal dette genskabes, sådan at systemet bevarer sin (n,k)-konfiguration. Da det er vigtigt, at ingen får mere information om hemmeligheden end tiltænkt, kan man ikke blot sende resten af systemets shares til den genskabende server og lade denne interpolere det hemmelige polynomium for herved at genskabe sit share. I modellen fra IBM foreslås i stedet en procedure, der minder meget om opdateringsfasen. Alle i servere laver et tilfældigt opdateringspolynomium $\Delta_i(x)$. I stedet for at lade $\Delta_i(0) = 0$ sørges i denne fase for at $\Delta_i(r) = 0$, hvor r er den server, der skal have genskabt sit share. Disse opdateringspolynomier benyttes til at lave nye shares på samme måde som i opdateringsfasen, dog uden at servernes rigtige shares overskrives. I stedet sendes de nye shares til serveren r, der således kan interpolere sig frem til polynomiet $s''(x)$ for herved at finde sit rigtige share.

Rutinen er illustreret på Figur 3 herunder (for overskuelighedens skyld er det valgt, at k=2, hvorved de anvendte polynomier bliver af første grad):



Figur 3 – Figuren viser, hvordan noden *r* kan genskabe sit share. To korrekte noder laver de to opdateringspolynomier $\Delta_1(x)$ og $\Delta_2(x)$, hvor $\Delta_i(r) = 0$. Når disse lægges sammen med det oprindelige polynomium $s(x)$ fås polynomiet $s''(x)$, sådan at $s(r) = s''(r)$. Ved at interpolere $s''(x)$ kan *r* således genskabe sit share $s(r)$, uden at få yderligere information om $s(x)$ og $s(0)$ (hemmeligheden).

Polynomiet $s''(x)$, som *r* interpolerer sig frem til har følgende egenskaber:

- $s''(r) = s(r)$
  Herved kan *r* genskabe sit korrekte share, der passer til det rigtige hemmelige polynomium.

- <u>Hvis x ≠ r:</u>
  $s''(x)$ giver ingen information om $s(x)$.
  Information om hemmeligheden eller om andres shares bliver således ikke afsløret for *r*.

For at illustrere genskabelsen af et tabt share gives her et lille eksempel. Eksemplet vil ligeledes beskrive, hvordan informationen verificeres under operationen. For at minimere eksemplet bygges videre på det foregående eksempel af distribueringen og rekonstruktionen. Der gives ikke noget tilsvarende eksempel på opdateringsoperationen, da denne minder meget om gendannelsesoperationen.

Eksempel: Genskabelse af et tabt share
Fra det tidligere eksempel antages det nu, at n_4's share er gået tabt. Noden n_3 svarer ikke, så noderne n_1, n_2 og n_5 er de eneste fungerende noder på dette tidspunkt. For at verificeringen af operationen kan gennemføres er der brug for yderligere verificeringsinformation i systemet end ved den beskrevne distribuering og rekonstruktion. Den hidtidige verificeringsinformation har kun kunne verificere et polynomiums korrekthed. Til denne operation er det nødvendigt, at alle noder direkte kan verificere alle andres shares på alle tidspunkter. Dette gøres ved at udvide distribueringen af shares, sådan at selve share-værdien kan verificeres. Alle noder skal således kende værdierne g^{x_i}, hvor x_i er den i'te nodes share, hvorfor disse også distribueres fra starten og holdes opdaterede, når de forskellige shares ændrer værdi. I dette eksempel betyder det, at følgende værdier er kendte af alle deltagende noder:

Verificering af n_1's share: $54^{13} = 78 \pmod p$
Verificering af n_2's share: $54^{24} = 51 \pmod p$
Verificering af n_3's share: $54^{39} = 244 \pmod p$
Verificering af n_4's share: $54^{11} = 251 \pmod p$
Verificering af n_5's share: $54^{34} = 127 \pmod p$

For at starte genskabelsen af n_4's share laver hver korrekt node nu et tilfældigt gendannelsespolynomium, $\Delta_i(x)$, hvor $\Delta_i(4) = 0$. I dette eksempel ser de ud som følger:

$\Delta_1(x) = 25 + 10x + 40x^2 \pmod q$.
$\Delta_2(x) = 15 + 4x + x^2 \pmod q$.
$\Delta_5(x) = 33 + 22x + 13x^2 \pmod q$.

Heraf laves et share til hver node samt verificeringsinformation. Noden n_1 laver således følgende shares og verificeringsinformation:

Shares = $\Delta_1(1) = 28, \Delta_1(2) = 17, \Delta_1(5) = 41 \pmod q$
Verificering = $[54^{25}, 54^{10}, 54^{40}] \pmod p = [207, 230, 158]$ (verificering af polynomiet).
Verificering = $[54^{28}, 54^{17}, 54^{41}] \pmod p = [240,134,42]$ (verificering af shares).

Tilsvarende laver n_2 og n_5:
Shares = $\Delta_2(1) = 20, \Delta_2(2) = 27, \Delta_2(5) = 13 \pmod q$
Verificering = $[54^{15}, 54^{4}, 54^{1}] \pmod p = [199, 38, 54]$ (verificering af polynomiet).
Verificering = $[54^{20}, 54^{27}, 54^{13}] \pmod p = [262, 256, 78]$ (verificering af shares).

Shares = $\Delta_5(1) = 21, \Delta_5(2) = 35, \Delta_5(5) = 45 \pmod q$
Verificering = $[54^{33}, 54^{22}, 54^{13}] \pmod p = [60, 175, 78]$ (verificering af polynomiet).
Verificering = $[54^{21}, 54^{35}, 54^{45}] \pmod p = [281, 66, 181]$ (verificering af shares).

Verificeringsinformationen broadcastes og de forskellige shares fordeles, sådan at $\Delta_j(i)$ sendes sikkert fra den j'te til den i'te node.

F.eks. modtager n_2 værdien $\Delta_5(2) = 35$ fra n_5 sammen med n_5's verificeringsinformation.

Nu checker n_2 den modtagne værdi ved at kontrollere følgende vha. polynomiumverificeringsværdierne:

$$54^{35} \stackrel{?}{=} 60^{(2^0)} * 175^{(2^1)} * 78^{(2^2)} \pmod{p}$$

og

$$1 \stackrel{?}{=} 60^{(4^0)} * 175^{(4^1)} * 78^{(4^2)} \pmod{p}$$

Den første test verificerer, at det modtagne share passer til noden (n_2). Den anden test sikrer, at polynomiet, der ligger til grund for sharet er lavet sådan, at $\Delta_5(4) = 0$, da sharet ellers vil være ubrugeligt til gendannelsen.

Alle modtagne shares verificeres således på ovenstående måde af alle noder. Når alle værdierne er modtaget og verificeret, har n_2 følgende korrekt information:
$x_2 = 24$ (n_2's eget share fra distribueringen)
$\Delta_1(2) = 17$, $\Delta_2(2) = 27$, $\Delta_5(2) = 35$.

Noden n_2 lægger nu disse sammen for at lave et gendannelsesshare: 24+17+27+35 = 9 (mod q)
På samme måde finder n_1 frem til gendannelsessharet 35 og n_5's gendannelsesshare bliver 39.

De tre gendannelsesshares sendes nu til n_4, som skal gendanne sit share. Inden gendannelsen kan finde sted skal n_4 dog verificere den modtagne information. Et gendannelsesshare verificeres vha. shareverificeringsværdierne. Noden n_4 verificerer således gendannelsessharet fra n_2 som følger:

$$54^{9} \stackrel{?}{=} 51 * 134 * 256 * 66 \pmod{p}$$

Værdien 51 er verificering af n_2's eget share. De resterende værdier stammer fra shareverificeringsinformationen som blev broadcastet under opstarten af fasen.

Efter alle tre gendannelsesshares er verificeret af n_4, kan noden gendanne sit share ved at interpolere. Når punkterne (1,35), (2,9) og (5,39) interpoleres for at finde $s''(4)$ (= $s(4)$) fås værdien 11 som netop svarer til n_4's oprindelige share. Dermed er gendannelsen fuldført.

Disse faser, der detekterer forsvundne og modificerede shares og genskaber disse, medfører, at en modstander ikke kan ændre shares uden at systemet opdager og korrigerer dette ved slutningen af tidsperioden. Dermed tvinges modstanderen til at skulle kompromittere k servere i samme tidsperiode for at destruere hemmeligheden.

De beskrevne rutiner sikrer, at secret sharing systemet kan overleve adskillige tidsperioder, såfremt kommunikationen mellem ikke-kompromitterede servere kan betragtes som værende sikker. Det er dog stadig et krav, at antagelsen om, at modstanderen maksimalt kan kompromittere k-1 servere i løbet af en tidsperiode overholdes.

2.2.4 Vægtet secret sharing

Indtil nu har det været antaget, at alle fik et share hver. I den virkelige verden stoler vi ikke lige meget på alle og nogle har mere at skulle have sagt end andre. Et eksempel kunne være en virksomhed, hvor tre ud af fire under-direktører skal kunne mødes og underskrive vigtige dokumenter ved direktørens fravær, sådan at virksomheden ikke går i stå. Hver underdirektør kan dog afløses af to af de 7 afdelingschefer, hvis nogle af disse ligeledes skulle være væk. Seks afdelingschefer kan altså ligeledes mødes og underskrive vigtige dokumenter i virksomhedens navn.

Denne løsning klares ved at direktøren deler sin underskriftsnøgle i et (6,15)-*threshold scheme* (dvs. 15 shares, hvoraf 6 shares skal være i stand til at rekonstruere nøglen). Hver afdelingschef får tildelt et share (i alt 7 shares), mens under-direktørerne får to hver (i alt $2 * 4 = 8$ shares).

På den måde kan secret sharing vægtes efter, hvor meget man stoler på folk. Vægtning af secret sharing vil uden problemer kunne indgå i en proaktiv og verificerbar secret sharing model.

2.2.5 Secret sharing med skiftende konfiguration

Hvis en under-direktør fra scenariet i forrige afsnit vælger at forlade sit job, skal han ikke længere være en del af systemet. En måde at klare dette er ved at lave en helt ny secret sharing opsætning og dele nye nøgler til de tilbageværende deltagere. Dette er imidlertid besværligt. I stedet kan konfigurationen i et (n,k)-*threshold scheme* skiftes i forbindelse med en opdateringsfase, jf. afsnit 0. Hvis n ønskes mindsket pga. en kompromitteret server kan man blot lade være med at opdatere dennes share, der herved vil blive ubrugeligt. Andre konfigurationsskift foregår ligeledes ved at de medvirkende serveres opdateringspolynomier (og dermed opdateringsshares) tilpasses den nye opsætning, hvorved konfigurationen ændres. Der findes flere forskellige forslag til, hvordan dette gøres i praksis. Disse vil ikke blive beskrevet yderligere her, men for en uddybning anbefales andre kilder [12].

2.2.6 Generelle problemer med secret sharing

Ved anvendelse af Shamirs secret sharing skal alle shares være lige så store som selve hemmeligheden. Dette ses intuitivt ved at k-1 shares ikke giver noget information om hemmeligheden. Dermed må det sidste share indeholde lige så meget information som hemmeligheden selv.

Dette er en ulempe, hvis den delte hemmelighed er meget stor. Først og fremmest bliver regnearbejdet på systemets shares stort. Desuden bliver båndbreddeforbruget stort, når der sendes shares rundt i systemet (f.eks. som følge af de proaktive vedligeholdelsesrutiner eller gendannelsen af hemmeligheden). Når Shamirs secret sharing benyttes er systemets samlede pladsforbrug a størrelsesordenen n gange størrelsen på hemmeligheden, hvor n er antallet af medvirkende servere (som hver har et share). Det samlede pladsforbruget vil således også vokse kraftigt, når hemmeligheden bliver større.

Problemet med det voksende regnearbejde har en simpel og effektiv løsning, som ligeledes løser problematikken omkring båndbreddeforbrug ved systemets proaktive vedligeholdelsesrutiner og ved gendannelsen af hemmeligheden: Krypter data med en symmetrisk krypteringsalgoritme og lad den symmetriske krypteringsnøgle være hemmeligheden, der deles vha. secret sharing. Dermed begrænses hemmelighedens størrelse til den symmetriske krypteringsnøgles størrelse. (Ved brug af Feldmans VSS er det dog vigtig, at hemmeligheden heller ikke er så lille, at det er muligt at bryde den vha. angreb på det diskrete logaritmeproblem.)

Udover selve secret sharing delen af den symmetriske krypteringsnøgle vil man dog være nødt til at distribuere det (store) krypterede data, for at bevare tilgængeligheden. Ligeledes vil de proaktive modeller skulle udvides med et integritetscheck af det krypterede data, da det ikke længere er tilstrækkeligt at sikre sig, at systemets shares er i orden. Ellers vil man risikere, at genskabe den rigtige hemmelighed, men denne krypteringsnøgle vil være ubrugelig, hvis det krypterede data er modificeret. Et sådant integritetscheck kan dog forholdsvist let implementeres vha. en envejs-hashfunktion som f.eks. SHA-1.

Den beskrevne løsning vil dog stadig bruge det samme pladsforbrug, idet selve det store krypterede data som udgangspunkt skal distribueres ud til alle servere. Det følgende afsnit beskriver en informationsspredningsalgoritme til at mindske systemets samlede pladsforbrug.

2.2.7 Information dispersal algorithm (IDA)

Informationsspredningsalgoritmen (IDA) blev introduceret i 1989 af Rabin [6]. Ligesom Shamirs secret sharing er IDA et (n,k)-*threshold scheme* som således deler data op i n dele, sådan at k shares er nødvendige for at gendanne det oprindelige data. Den store forskel ligger i, at IDA spreder data ud, sådan at hvert share ikke er på størrelse med hemmeligheden, men i stedet har størrelsen $|D|/k$, hvor $|D|$ er størrelsen af data. Den samlede mængde data i systemet, er således $(n/k) * |D|$. Den samlede mængde af data, der skal til at genskabe det oprindelige data D, er $|D|$, hvorimod Shamirs secret sharing benytter $k * |D|$ (da alle shares her har størrelsen $|D|$). Den plads IDA benytter, som overskrider $|D|$, benyttes således til redundans i systemet, sådan at et antal shares kan undværes.

Kort fortalt splitter IDA først data D op i sekvenser af størrelsen k. For at dele data op i n shares, benyttes n kodningsvektorer af længden k. Tilsammen danner disse kodningsmatricen A. Hver sekvens af størrelse k transformeres nu til n shares vha. kodningsmatricen. Disse kodningsvektorer benytte igen, når data skal genskabes. Rabin foreslår derfor, at kodningsvektorer i praksis gemmes sammen med de enkelte shares, sådan at vektorerne ligeledes er beskyttet af spredningen. En detaljeret beskrivelse af informationsspredningsalgoritmen er givet af Rabin [6].

Fordelene ved IDA frem for Shamirs secret sharing er således åbenlyse, da den både kræver mindre båndbredde ved distribution og gendannelse af data og mindre diskplads.

Ulempen ved IDA er, at det umiddelbart er vanskeligt at opretholde den samme proaktivitet, som i det proaktive secret sharing, hvor shares opdateres og vedligeholdes. Opdatering af IDA-shares kan dog overflødiggøres ved at kryptere data, før det spredes. Dermed skal krypteringsnøglen

dog også gemmes, men dette kan den proaktive secret sharing håndtere, da nøglens størrelse ikke er af betydning. Det er dog stadig nødvendigt, at kunne kontrollere integriteten af IDA-shares og kunne genskabe disse, hvis de er blevet modificeret eller slettet. Hvis ikke dette kan lade sig gøre, kan systemet ikke overleve i uendelig mange tidsperioder, ligesom proaktivt secret sharing, da modstanderen over flere tidsperioder kan overtage mere end *n-k* servere og slette deres share, hvorved hemmeligheden ikke kan gendannes. Umiddelbart er det ikke let at gendanne forsvundne IDA-shares, ud fra de resterende shares i systemet.

2.3 PEER-TO-PEER

Dette afsnit vil beskrive først peer-to-peer arkitekturen overfor klient/server-strukturen. Derefter gennemgå JXTAs hovedbegreber og grundlæggende virkemåde.

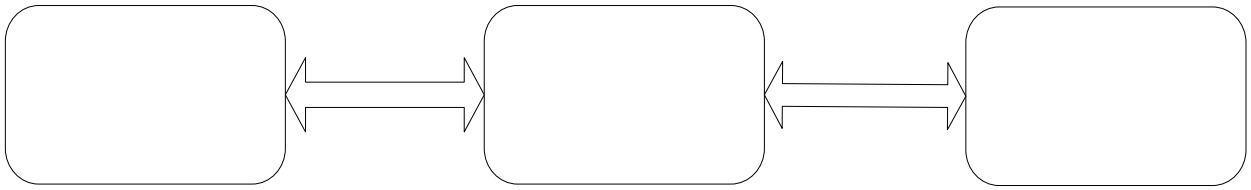
2.3.1 Peer-to-peer kontra klient/server

De fleste distribuerede systemer er baseret på en klient/server tankegang, hvor en server tilbyder services til en række klienter. Denne struktur har mange fordele, hvilket da også er årsagen til, at det stadig er langt den mest anvendte. Serverens centrale rolle letter bl.a. adgangskontrol og administrationen af systemet.

En ulempe ved denne type arkitektur er dog, at man har et enkelt kritisk sted (eller få steder) i netværket, der kan sætte hele det distribuerede system ud af spil, hvis der skulle opstå fejl eller sikkerhedsbrud (*single point of failure*). Et andet problem ved klient/server-løsninger er, at der kan opstå flaskehalsproblemer, når mange klienter vil tilgå serveren på samme tid.

Problemerne imødegås typisk ved at replikkere serveren, så flere servere kan tage del i arbejdet og tage over for hinanden, hvis en server skulle fejle. Denne løsning er ikke altid optimal, da den samtidig mindsker nogle af de oprindelige fordele ved løsningen og skaber nogle kritiske sektioner, som de replikkerede servere skal enes om.

Igennem de sidste årtier har man forsøgt sig med mere eller mindre dominerende servere. I den ene ende af skalaen findes en mainframe-løsning, hvor alt arbejde udføres af serveren (mainframen). Klienterne er såkaldte ”tynde” klienter, der ikke kan udrette noget på egen hånd udover at spørge serveren om at udføre et stykke arbejde. Bevæger man sig lidt væk fra de helt tynde klienter finder man en workstation-baseret løsning, hvor klienterne er ”tykkere” workstations med programmer (såsom tekstbehandling og andre kontorprogrammer), der afvikles lokalt. Typisk vil serveren (eller serverne) i disse løsninger stå for specialiserede opgaver såsom filhåndtering, styring af adgangskontrol, printstyring, mail og meget andet. Med denne middel-løsning tages en stor del af arbejdsbyrden væk fra serverne, men systemet er stadig meget afhængigt af, at serverne fungerer korrekt.



Figur 4 – Overgang fra altdominerende mainframeløsninger til peer-to-peer.

I den anden ende af skalaen (som er illustreret på Figur 4 herover) findes peer-to-peer systemer. Disse karakteriseres bl.a. ved, at hver node (peer) både kan agere klient og server overfor andre noder. Hver node er bekendt med eksistensen af andre noder og behøver således ikke nødvendigvis spørge en server hver gang kommunikation skal etableres. Kommunikationen foregår direkte mellem noderne og ikke via en server hver gang. Dette er med til at mindske både flaskehalsproblemer og problemer opstået pga. *single point of failure*. Dog kan administration af noder og data hurtigt blive besværligt, da alle noder skal have en slags serverfunktionalitet og være mere bevidste om resten af systemet end klienter i en klient/server-løsning.

Det første rigtig kendte peer-to-peer system, der kom frem, var fildelingssystemet Napster. I Napster sender noderne filer direkte mellem hinanden, uden at afhænge af servere. Selve søgningen efter filer er dog centraliseret i Napster på samme måde som almindelige klient/server-systemer. En central server har til opgave at udføre søgninger og give de søgende noder information om, på hvilke noder de kan finde det søgte. Napster har således stadig et *single point of failure*, da serveren er nødvendig for at systemet kan virke [4]. Dog er størstedelen af arbejdsbyrden (selve filoverførslen) flyttet fra serveren til de enkelte noder, hvorved de værste flaskehalsproblemer hos serveren er undgået.

Et peer-to-peer system, der blev lavet i kølvandet på Napster var Gnutella. Gnutella er i modsætning til Napster komplet decentraliseret [5]. Dette betyder, at *single point of failure* problematikken er løst. Til gengæld kræves meget båndbredde og flere af systemets samlede ressourcer til at gennemføre søgninger, da disse skal gå fra node til node.

Siden begyndelsen med Napster og Gnutella er utallige peer-to-peer systemer udviklet. De fleste har dog mange grundlæggende fællestræk, uafhængigt af systemernes individuelle funktioner. Blandt disse fællestræk kan nævnes peer-discovery[11] og kommunikation gennem firewall og Network Address Translation (NAT). Mange af disse fælles-funktioner er der ingen grund til at udvikle igen og igen, hver gang et nyt peer-to-peer system skal se dagens lys. Dette er årsagen til at projekt JXTA blev startet.

2.3.2

[11] peer-discovery finder sted når en node tilslutter sig et peer-to-peer netværk og skal i kontakt med andre noder. Problemet er ikke trivielt og der findes endnu ingen optimal løsning. For at finde andre noder, kan man f.eks. spørge en server, der kender til andres eksistens, eller man kan forsøge at finde noder vha. ip-multicast.

JXTA

JXTA er et open source peer-to-peer framework, der er lavet med henblik på at lette udarbejdelsen af peer-to-peer applikationer ved at fungere som et sæt byggeklodser for disses grundlæggende funktionalitet. JXTA blev startet af Sun Microsystems i 2001, men udviklingen forgår nu i et åbent forum, hvor alle kan deltage.

JXTA består af en række protokoller, som tilsammen udgør en infrastruktur, der lader enheder på et netværk kommunikere med hinanden peer-to-peer. Disse enheder er ikke begrænset til PC'ere, men kan ligeledes være PDA'ere, mobiltelefoner og meget andet. Udover at understøtte peer-to-peer kommunikation mellem forskellige enheder på et netværk, er det et overordnet formål med JXTA, at det skal kunne bruges uafhængigt af programmeringssprog, operativsystemer og netværkstyper.

De vigtigste grundlæggende komponenter i JXTA er:

- Peer (node)
- Peer group (Gruppe af noder)
- Advertisement (annoncering)
- JXTA ID (unik id)
- Pipe

Noder og grupper

En node er et virtuelt kommunikationspunkt, og adskiller sig fra en bruger, idet hver bruger kan have flere forskellige noder på samme tid. En node kan være medlem af en eller flere grupper. I en gruppe kan forskellige services være tilgængelige for gruppemedlemmerne og gruppemedlemskab kan begrænses, således at kun autentificerede medlemmer kan tilslutte sig gruppen.

Annonceringer

Alle ressourcer i JXTA (noder, grupper, pipes osv.) har sit eget unikke 128 bit JXTA ID. Ressourcerne er repræsenteret vha. en advertisement (annoncering), hvori bl.a. JXTA ID'et indgår. En JXTA advertisement er et XML dokument, der kan publiceres lokalt eller hos andre noder for at lade andre noder opdage, at ressourcen eksistere. Udover de grundlæggende advertisement typer, kan man selv lave sine egne advertisements og derved ligeledes annoncere for egne ressourcer, der ikke tilhører JXTA standarden.

Pipes

En JXTA-pipe er en virtuel forbindelse mellem noder i et JXTA -netværk. En pipe er ikke nødvendigvis kun en enkelt forbindelse, da mange noder ikke kan forbindes direkte pga. firewalls og NATs. Der findes en række forskellige pipes i JXTA, hvoraf kun enkelte er implementeret i den nuværende version. En standard pipe er som udgangspunkt usikker, envejs og asynkron[12]. De nuværende implementerede pipes involvere desuden en tovejs-pipe, der ligeledes er upålidelig og usikker samt en envejs, pålidelig og sikker pipe, der sikrer at kommunikationen kommer frem og krypterer data. Krypteringens styrke er dog begrænset til 512

[12] I nyere versioner af JXTA findes abstraktionslag (som f.eks. JXTA socket), der kan lette brugen af JXTA-pipes og bl.a. sikre at kommunikationen når frem.

bit asymmetriske nøgler i sin nuværende implementering (fra januar 2004), hvilket gør denne type forbindelse uanvendelig i mange henseender. En pipe er ikke bundet til en fysisk adresse. I stedet har hver pipe et unikt ID, sådan at en node kan beholde sine pipes, selvom denne flyttes mellem forskellige fysiske netværk (forskellige lokationer).

Netværksstruktur
JXTA netværket er et ad-hoc netværk bestående af tilsluttede noder. Noder kan tilslutte sig og forlade netværket når som helst, hvorved netværkstopologien skifter. Der findes som udgangspunk følgende 4 typer noder i et JXTA netværk:

- Minimal node
- Simpel node (også kaldet *edge peer*)
- Rendezvous node
- Relay node

En minimal node kan sende og modtage beskeder men gemmer ikke advertisements og router ikke kommunikation for andre noder. Denne type node er typisk tiltænkt PDA'ere eller telefoner, da den kun kræver få ressourcer.

Simple noder er standard noder i JXTA netværket, og de fleste noder vil typisk være af denne type. En simpel node har typisk et lager med advertisements. Lagrede advertisements sendes som svar på forespørgsler fra andre noder. Forespørgsler videresendes dog ikke til andre noder.

En rendezvous-node fungerer basalt set som en simpel node. Dog videresender den ressource-forespørgsler til andre noder (både til simple og minimale noder, som bruger rendezvous-noden som rendezvous, og til andre rendezvous-noder), og er derfor et væsentligt element ved søgningen efter ressourcer. En søgning fortsætter indtil et resultat er fundet eller forespørgslens *Time To Live* (TTL) er nået uden resultat.

En relay-node gemmer information om router til andre noder. Når en node vil i kontakt med en anden, kigger den først i sit lokale lager, men hvis den ikke finder routeinformation her, spørges kendte relay-noder. Relay-noder kan også videresende pakker fra noder, som ikke kan kommunikere direkte pga. firewalls og/eller NATs. En node bag en firewall kan typisk godt starte kommunikation med noder, der ikke er bag en firewall, men det omvendte er ofte umuligt. Når først forbindelsen er oprettet, kan pakker dog sendes gennem en firewall i begge retninger. Derfor klares problematikken med firewalls ved, at noder bag en firewall holder kommunikationen åben med en relay-node. Da kommunikationen er startet af noden bag en firewall, kan relay-noden videresende pakker til denne fra andre noder i systemet. Da JXTA også kan benytte HTTP-protokollen i stedet for TCP, kan kommunikation stadig lade sig gøre, selvom en firewall kun vil tillade udgående HTTP trafik.

Peer discovery

Når en node tilslutter sig JXTA netværket skal den forsøge at finde andre noder (*peer discovery*). Dette gøres på forskellige måder, afhængigt af nodens situation. Hvis en node allerede har været på JXTA netværket, vil den starte med at kontakte de rendezvous-noder, som den allerede kender. Hvis dette ikke er tilfældet, eller hvis rendezvous-noderne ikke kan kontaktes, vil noden forsøge at finde andre noder vha. ip-multicast på det lokale netværk[13]. Hvis dette fejler kontaktes nogle faste JXTA rendezvous-noder, der har til formål at hjælpe nystartede noder med at få kontakt til netværket.

Fordele og ulemper

En af fordelene ved JXTA er, at man vha. simple grundelementer kan bygge forskelligartede distribuerede applikationer. JXTA tilbyder stor fleksibilitet og kan bruges på forskelligartede platforme og med forskellige programmeringssprog. Udvikleren kan selv vælge, hvor centraliseret en løsning skal være. Det er således en erkendelse i JXTA-miljøet, at det ikke nødvendigvis er en fordel, at alle funktioner varetages af alle noder. I stedet kan det ofte være en fordel, at lade enkelte funktioner varetage af en mindre gruppe noder (et eksempel herpå er rendezvous-funktionaliteten).

JXTA er standardiseret til at kunne understøtte stort set alle slags peer-to-peer systemer. Dette er naturligvis en fordel i mange henseender, men kan dog også være en ulempe, da JXTA naturligvis ikke kan være optimeret til alle systemer på samme tid. Derfor vil systemer bygget på JXTA muligvis have en ringere ydelse end tilsvarende systemer bygget fra grunden. En anden ulempe ved JXTA er, at det stadig er meget nyt og under konstant videreudvikling og da dokumentationen er mangelfuld kan det være besværligt at arbejde med. Desuden er der ingen garantier for, at JXTA rent faktisk virker som forventet, hvilket besværliggør fejlfinding.

Udviklingen af JXTA er stadig i fuld gang og ikke alle mål er inden for rækkevidde endnu. Et lille eksempel herpå er konfigurationen af JXTA hos den enkelte bruger. Her er det en målsætning, at dette skal kunne foregå uden interaktion fra brugerens side. Dette er endnu ikke muligt, men forskellige delløsninger er blevet fremvist. Disse har dog alle deres ulemper, da de ikke besidder den tilstrækkelige dynamik og de fungerer således stadig kun som add-ons til JXTA.

[13] I princippet kunne multicast med fordel benyttes til at finde noder på internettet (ikke kun på lokalnetværket). Årsagen til at dette ikke er praktisk muligt er, at multicast ikke er pålideligt nok på nutidens internet, da mange internet-routere stadig ikke understøtter multicast.

2.4 OPSUMMERING

I dette kapitel er vigtige dele af den klassiske kryptografi blevet beskrevet. Ideerne bag symmetrisk og asymmetrisk kryptografi er forklaret sammen med de grundlæggende sikkerhedsbetragtninger og anvendelsesmuligheder. Envejs-funktioner og kryptografiske hashfunktioner er ligeledes gennemgået. Nogle af de oftest anvendte algoritmer er nævnt og forklaret kort, dog uden at gå ned i de matematiske detaljer. Brugen af certifikater og certifikatautoriteter til autentificering er også beskrevet. De beskrevne værktøjer kan være med til at opnå sikkerhedsmål såsom fortrolighed, integritet af data, autenticitet og non-repudiation.

Forskellige secret sharing modeller er beskrevet og fordele og ulemper er skitseret. Shamir's grundlæggende ide er god, men har sine mangler ved praktiske anvendelser, hvorfor det er interessant at betragte verificerbare modeller, såsom Feldman's VSS, og ikke mindst proaktivt secret sharing.

Til sidst er peer-to-peer teknologien blevet beskrevet grundlæggende og JXTA-teknologien er skitseret. De vigtigste JXTA-koncepter er gennemgået sammen med JXTAs funktionalitet. Fordele og ulemper ved JXTA er beskrevet.

3 SECRET SHARING BACKUP

I dette kapitel vil en kravspecifikation blive udarbejdet på baggrund af en grundlæggende model, der opstilles og analyseres. Truslerne og risici vil blive beskrevet og analyseret for at identificere eventuelle svagheder.

3.1 BAGGRUND

I den grundlæggende problemstilling ønsker et individ eller en organisation at foretage en backup af data. Data skal til hver en tid kunne genskabes, såfremt det skulle ønskes. Da data er fortroligt, er det ligeledes et krav, at ingen uvedkommende må kunne få fat i data på noget tidspunkt.

Denne simple problemstilling har dog ingen simpel løsning. Ved normale backupløsninger vil man typisk vælge, hvilken grad af replikkering der er nødvendig for at sikre den rette grad af tilgængelighed. Dernæst vil man forsøge at mindske adgangen til de replikkerede servere, sådan at kun den rette ejer kan få fat i backup'en og genskabe data. Det kan aldrig lade sig gøre at sikre en server 100%. Derfor bliver denne opgave er naturligvis sværere, når antallet af replikkerede servere vokser. Det grundlæggende paradoks kan kort udtrykkes som følger:

- n+1 servere giver en bedre tilgængelighed end n servere.
- n servere giver bedre fortrolighed og integritet end n+1 servere.

Derfor ser det umiddelbart ud som om, det ikke er let at sikre både fortrolighed, integritet og tilgængelighed på samme tid.

Som en løsning på denne grundlæggende problemstilling foreslås et backupsystem baseret på secret sharing og peer-to-peer teknologi. Den grundlæggende model for systemet er beskrevet i afsnit 3.2. Hver af de medvirkende n noder vil således ikke være i stand til at genskabe data, men k af disse kan sammen foretage rekonstruktionen. På den måde kan en replikkering finde sted uden at mindske systemets fortrolighed væsentligt. Den decentraliserede peer-to-peer struktur er med til at øge systemets robusthed, da enkelte fejlende noder derved ikke er kritiske for systemet.

Til problemstillingen hører en aktiv mobil modstander[14] (*mobile adversary*), der ønsker at få fat i de hemmelige backup-data og/eller ødelægge dem, så systemet ikke senere kan genskabe data. Modstanderen vil med alle tænkelige midler forsøge at kompromittere de medvirkende noder. En node er kompromitteret, hvis modstanderen har fået adgang til hemmeligt information på denne,

[14] En mobil modstander karakteriserer en modstander, der midlertidigt kompromitterer en server for derfra at bevæge sig videre til det næste offer [12]. Det er dog en grundlæggende antagelse om mobile modstandere, at tiden påkrævet for at kompromittere en server er betydelig.

sådan at dette kan læses og/eller modificeres. Modstanderen vil desuden kunne kontrollere en kompromitteret nodes opførsel, hvorved denne f.eks. kan bruges til yderligere angreb mod resten af systemet. En node betragtes ligeledes som kompromitteret, hvis modstanderen forhindrer denne i at kommunikere med resten af systemet.

I forlængelsen af teorien om proaktivt secret sharing er det en grundlæggende antagelse i det foreslåede system, at der hele tiden eksisterer k korrekt fungerende noder, der følger systemets protokol. Den samlede mængde af fejlende noder og kompromitterede noder er altså maksimalt k-1 i løbet af samme tidsperiode.

3.2 GRUNDLÆGGENDE MODEL

I et peer-to-peer system inddeles noderne i backupgrupper, sådan at noder i forskellige grupper ikke behøver kommunikere med hinanden. I en backupgruppe med m noder deles en backup ud til n af disse ($n \leq m$) vha. verificerbart proaktivt secret sharing, sådan at k noder kan genskabe det oprindelige backupdata, mens mindre end k noder ikke kan få noget at vide om backupdata. Noderne kommunikerer sikkert vha. asymmetrisk kryptering, men inden dette kan ske, skal noderne autentificere hinanden. Dette gøres vha. certifikater og certifikatautoriteter.

Systemets grundlæggende operationer er at tage backup af en fil og gendanne filen. Når en bruger vil tage backup af en fil, krypteres denne med en symmetrisk krypteringsnøgle, som herefter deles blandt n noder vha. proaktivt secret sharing. Den krypterede backupfil kan herefter distribueres sikkert til noderne uden yderligere foranstaltninger. Det er således kun krypteringsnøglen, der er beskyttet af proaktivt secret sharing, og herved undgås store shares i systemet.

Ejeren af filen (og ingen andre) kan siden hen genskabe backup'en, ved at få de forskellige shares fra de deltagende noder. Af k shares kan den hemmelige krypteringsnøgle genskabes. Derefter kan denne bruges til at dekryptere selve backupfilen (som bliver tilsendt fra en af systemets noder).

Udover denne grundfunktionalitet findes proaktive operationer til at vedligeholde systemet. Først og fremmest haves en opdateringsoperation som sikrer, at tidligere kompromitterede shares bliver ubrugelige. Dermed skal en modstander kompromittere k noder inden en opdatering finder sted, for at kunne bryde systemets fortrolighed.

Systemets integritet sikres proaktivt vha. et integritetscheck, som kontrollerer, at de forskellige shares er umodificerede samt at den krypterede backupfil er umodificeret. Hertil benyttes hhv. verificerbart secret sharing og checksum af data. Hvis et falsk share opdages, bliver dette gendannet vha. en gendannelsesoperation, jf. afsnit 2.2.3. En krypteret backupfil, der er blevet modificeret er ubrugelig. Derfor hentes den korrekte fil fra en anden af systemets noder.

Disse vedligeholdelses operationer inddeler således systemet i sikre tidsintervaller, hvor modstanderen skal bryde sikkerheden (dvs. få fat i k shares eller ødelægge n-k+1 shares)

indenfor tidsintervallet, for at kunne ødelægge systemet eller bryde dets fortrolighed. Grundantagelsen bag modellen er derfor, at dette ikke kan lade sig gøre for modstanderen.

En vigtig styrke ved denne model er, at en bruger kan miste ALT data (både backupdata, shares, krypteringsnøgler og certifikat) og stadig være i stand til at genskabe dette. Brugerens certifikatautoritet kan udstede et nyt certifikat med samme identitet (men et andet nøglepar), sådan at brugeren kan autentificere sig overfor brugerne i backupsystemet og derved atter få rettigheder over sine filer. Den proaktive secret sharing betyder, at data stadig findes i systemet og kan genskabes, sådan at alt var som før.

3.3 MODELANALYSE

I dette afsnit videreudvikles den grundlæggende model for det foreslåede system og de vigtigste aspekter analyseres.

3.3.1 Kommunikation og autentificering

Systemet er afhængigt af sikker kommunikation, for at modstanderen ikke kan angribe secret sharing protokollerne. Sikker kommunikation mellem noderne kan opnås vha. asymmetrisk kryptografi, hvor hver bruger har sit eget nøglepar. Desuden benyttes symmetriske sessionsnøgler til at øge hastigheden på krypteringsoperationerne, hvilket er beskrevet nærmere i afsnit 4.4.1. Brugerens offentlige nøgle gives til de andre noder, mens den private holdes hemmelig. Herved sikres både fortrolighed og integritet af kommunikationen. Ved både at kryptere med modtagerens offentlige nøgle og lave en signatur med afsenderens private nøgle, sikres desuden non-repudiation. Dermed kan en kompromitteret node ikke sende falsk data og siden hen benægte afsendelsen. Hvis det ikke lykkes en bruger at holde sin private nøgle hemmelig, vil modstanderen både kunne læse brugerens kommunikation med andre noder og lave korrekte signaturer. Dermed vil modstanderen kunne generere sine egne kommunikationspakker og signere dem med brugeren private nøgle, før de sendes til andre noder, sådan at modtagernoderne vil tro, at pakkerne kommer fra den pågældende bruger selv. Det er således essentielt, at private nøgler holdes hemmelige i systemet.

Før kommunikationen er sikker, skal nøglerne dog kunne autentificeres. En fælles gruppeautentificering er utilstrækkelig til at gennemføre systemets autentificering, da det her er gruppens samlede holdning til det nye medlem, der afgør, om denne autentificeres. Da det ikke er gruppen i fællesskab, der skal tage en backup, men den enkelte bruger, er der behov for, at hver enkelt bruger (på hver enkelt node) kan autentificere hver af de andre og derved afgøre, om han tør stole på disse eller ej. Derfor benyttes i stedet certifikater og certifikatautoriteter til autentificering. Løsningen muliggør effektiv autentificering af brugere, og kræver ikke nogen tidligere kontakt eller fælles kontakter ligesom *Web of Trust*, hvor man helst skal kende nogen, som kender nogen osv.

Hver bruger skal således have et certifikat fra en certifikatautoritet, som bevidner, at dennes offentlige nøgle er autentisk og tilhører brugeren. En node gemmer desuden modtagne certifikater fra andre noder, der er autentificerede. Disse benyttes for at opnå sikker kommunikation.

3.3.2 Secret sharing

Ved anvendelsen af secret sharing mindskes paradokset omkring fortrolighed kontra pålidelighed. Det er dog ikke forsvundet, da de to egenskaber stadig er forbundne. Som eksempel herpå er en (25,2)-model meget pålidelig, idet 25 noder opbevarer data mens kun to kan genskabe data. Omvendt vil en (25,12)-model have en større fortrolighed, da 12 noder skal bruges for at genskabe hemmeligheden. Dette betyder til gengæld også, at robustheden og dermed pålideligheden mindskes, da hemmeligheden vil gå tabt, hvis 14 af de 25 noder mister deres data i samme tidsperiode.

Det samlede niveau af fortrolighed og pålidelighed kan dog med rimelighed antages at kunne øges vha. secret sharing. En (25,1)-model svarer til normal replikkering af data, da alle 25 noder hver for sig kan genskabe data. Ved f.eks. at øge *k* til 3, øges fortroligheden væsentligt, da det er markant sværere at skaffe tre shares end et enkelt. Derimod mindskes pålideligheden kun en lille smule, da der stadig skal gå 23 shares tabt, før data mistes.

Da systemet skal beskyttes mod aktive angreb, er det vigtigt, at en verificerbar secret sharing model vælges. Denne vil forhindre, at protokollen omgås ved at falske shares kommer i omløb. For at holde systemets protokol så simpel som muligt, vil det desuden være en fordel, hvis den verificerbare secret sharing er ikke-interaktiv. Feldmans VSS opfylder de krav, som systemet har til verificerbarhed, og da denne model desuden er den simpleste, vælges denne.

Backup-data kan ofte blive gemt i mange år, og selv når der ikke er brug for en rekonstruktionsfunktionalitet fordi data er forældet, er det ofte stadig et krav, at data holdes fortrolige. Dette kan bl.a. skyldes forretningshemmeligheder eller følsomme kundedata. Derfor er det vigtigt, at systemets sikkerhed ikke mindskes væsentligt som tiden går (udover den naturlige forringelse af krypteringsnøglernes styrke). Sikkerhedsforringelserne vil enten kunne ske i forbindelse med kompromitterede noder eller shares, der går tabt. Proaktivt secret sharing løser disse problemer med de omtalte share-opdaterings- og share-gendannelsesoperationer, jf. afsnit 2.2.3.

Til hver backup-fil hører en symmetrisk krypteringsnøgle, der krypterer filen, når en backup skal foretages. Den krypterede backup-fil sendes til alle deltagende servere. Disse er udvalgt blandt servere, som filejeren har autentificeret. Derefter benyttes secret sharing til at dele den symmetriske krypteringsnøgle mellem medlemmerne i et (*n,k*)-*threshold scheme*, som filejeren har valgt. Da Feldmans VSS afhænger af, at det diskrete logaritmeproblem ikke kan løses, vil en symmetrisk krypteringsnøgle alene være for lille til at sikre, at dette ikke kan ske. Derfor skal den symmetriske krypteringsnøgle pakkes ind i en større envelope, før den deles via secret sharing.

Denne løsning er valgt, da størrelsen af en backup-fil kan variere meget og da det ikke er hensigtsmæssigt at dele store filer direkte vha. secret sharing. En årsag hertil er, at shares skal sendes over netværket til filejeren i forbindelse med en rekonstruktion af det hemmelige data. Hvis en backup-fil fylder 100 Mb, vil direkte shares af filen ligeledes fylde 100 Mb. Disse vil således skulle sendes fra k noder til filejeren, for at denne kommer i besiddelse af k shares og kan rekonstruere data (antaget at filejerens eget share er gået tabt). Dette vil sandsynligvis gøre båndbredden til et flaskehalsproblem. Problemet undgås ved i stedet at dele en krypteringsnøgle med secret sharing. Filejeren behøver således kun at modtage k shares af en størrelse magen til krypteringsnøglen samt en enkelt version af den krypterede backupfil (100 Mb). Herved spares en båndbredde på $k * 100\text{Mb} - (100\text{ Mb} + k * \text{nøglestørrelse}) \approx (k\text{-}1) * 100\text{Mb}$. De samme flaskehalsproblemer vil opstå i forbindelse med opdatering og rekonstruktion af shares, hvis ikke denne løsning vælges.

Denne løsning betyder dog, at systemet også skal tage højde for at selve det krypterede backup-datas integritet er i orden. Systemets protokol skal altså designes til at kunne checke integritet af både filer og shares. Dette er beskrevet yderligere i afsnit 4.5.7.

De secret sharing protokoller, som systemet skal indeholde, er derfor som følger:

- Grundlæggende verificerbart secret sharing til at foretage backup og til at rekonstruere backup-data.
- Proaktiv verificerbar share-opdateringsfase, hvorved gamle shares bliver ubrugelige.
- Proaktivt integritetscheck af shares.
- Proaktiv verificerbar share-gendannelsesfase, hvor tabte og modificerede shares gendannes.
- Proaktivt integritetscheck af selve backup-data.

Desuden skal beskyldninger i forbindelse med protokolafvigelser håndteres (se afsnit 3.3.4).

3.3.3 Gruppestruktur

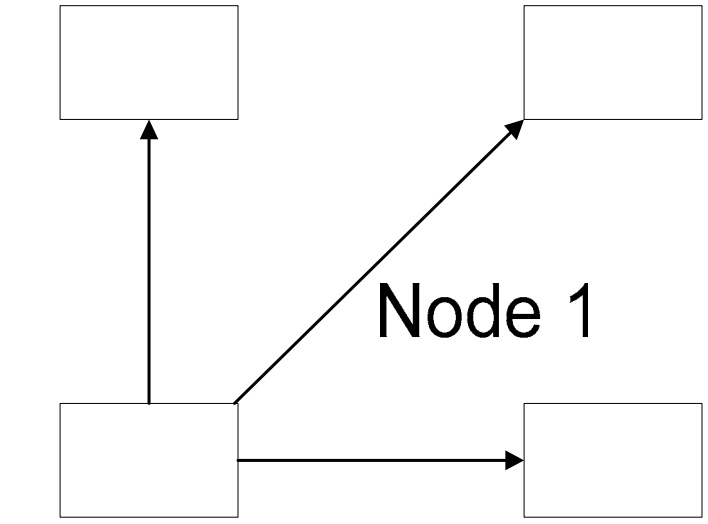
Hvis systemet skal fungere med mange brugere på samme tid, vil det være besværligt, at holde styr på alle uden en gruppeinddeling. Ved at brugere tilslutter sig en backup-gruppe, kan man vælge en delmængde af systemets brugere, som man ønsker at samarbejde med. En gruppeinddeling giver ligeledes mulighed for adgangsbegrænsede grupper, sådan at en virksomhed f.eks. kan lave sin egen password-beskyttede gruppe, som ingen andre kan blive medlem af. Det skal dog bemærkes i denne sammenhæng, at systemets sikkerhed ikke afhænger af en gruppeautentificering, da alle brugere skal autentificere hinanden enkeltvist. Gruppestrukturen laves således primært for at lette brugen og overskueligheden af systemet.

I en gruppe har alle mulighed for at foretage backup hos hinanden. For hver fil der tages backup af, vælger ejeren, hvilke gruppemedlemmer, der skal have et share, da alle medlemmer ikke nødvendigvis stoler på hinanden.

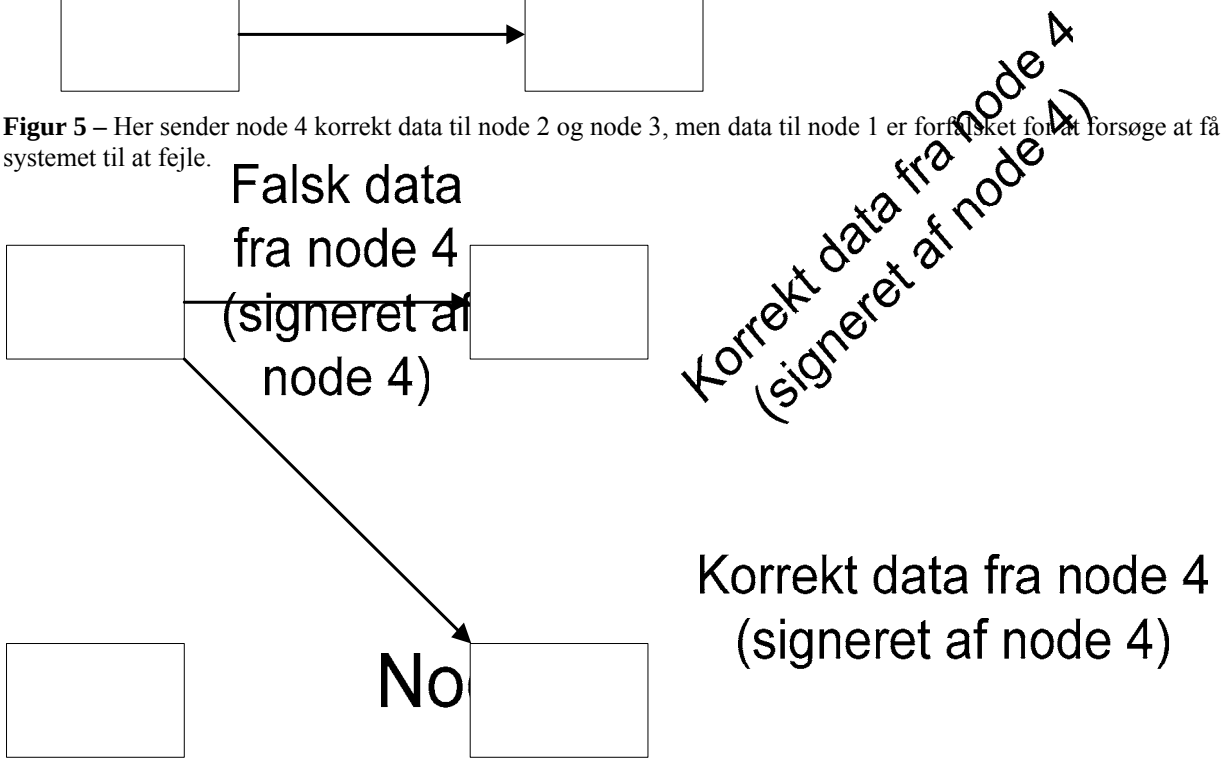
3.3.4 Kompromitterede noder

System skal være robust overfor kompromitterede og fejlende noder. En stor del af denne robusthed opnås vha. proaktivt secret sharing, hvori det antages, at kompromitterede noder ikke er kompromitterede for evigt men typisk kun i en eller to tidsperioder. Efter noder er kompromitterede kan den rigtige ejer af noden "rense" denne, sådan at noden ikke længere er styret af modstanderen. Denne rensning bliver beskrevet som en reboot rutine [11]. I praksis vil det ofte kræve en reinstallering af systemet, før man kan være sikker på, at modstanderen ikke længere befinder sig i systemet. Dette kræver dog, at systemet er klar over, at noden er kompromitteret, sådan at ejeren kan blive gjort opmærksom på situationen.

Systemet kan opdage kompromitterede noder ved kontinuerligt at sikre sig, at systemets protokol overholdes. I det øjeblik protokollen brydes, er den ansvarlige node kompromitteret. Ofte vil alle noder opdage dette på nogenlunde sammen tid (f.eks. hvis den kompromitterede node slet ikke svarer) men hvis modstanderen er mere snu, vil han forsøge at finde huller i protokollen ved at sende falsk data til en enkelt eller få noder, mens resten af systemet modtager korrekt information. Her er det væsentligt, at hver enkelt node kan vurdere, om den modtagne information er korrekt. Dette kan klares primært vha. verificerbarheden i secret sharing. Dernæst skal modtagernoden være i stand til at bevise, at den kompromitterede node afviger fra protokollen, sådan at resten af systemets noder ligeledes vurderer noden som kompromitteret. Derudover må det ikke kunne lade sig gøre for en kompromitteret node at overbevise resten af systemet om, at en korrekt node er kompromitteret, således at denne udelukkes fra systemet. Disse problemstillinger kan håndteres vha. den asymmetriske kryptografi, som benyttes til kommunikationen mellem noderne, da afsenderen ikke kan benægte at have sendt den pågældende pakke, når dennes signatur matcher. En node, der modtager falsk information, vil således sende en beskyldning (*accusation*) indeholdende informationen og signaturen videre til resten af systemets noder. Disse kan ved at betragte signaturen og informationen vurdere, om en node er kompromitteret. Når en node sender en beskyldning om en anden node til resten af gruppen, svarer det altså til, at den prøver at tilbagekalde den kompromitterede nodes certifikat. Tilbagekaldelsen i dette tilfælde foretages dog ikke af certifikatautoriteten men af systemets egen protokol. Figur 5 og Figur 6 herunder skitserer beskyldningerne.



Figur 5 – Her sender node 4 korrekt data til node 2 og node 3, men data til node 1 er forfalsket for at forsøge at få systemet til at fejle.



Figur 6 – Her benytter node 1 den falske data (og den tilhørende signatur) til at beskylde node 4 for at snyde. Node 2 og node 3 kontrollerer informationen fra node 1, og hvis de er enige i, at node 4 har snydt, registrerer de noden som værende kompromitteret.

Et særtilfælde til denne slags angreb er dog, hvis modstanderen vælger slet ikke at sende data til enkelte noder. Dermed vil de ikke kunne bevise, at modstanderens node afviger fra protokollen (han kan heller ikke selv være sikker, da årsagen kan skyldes simple fejl i stedet). Denne problematik er således nødt til at blive håndteret af protokollen, sådan at denne type angreb ikke kan finde sted. Det nærmere protokoldesign, der skal sikre, at disse angreb er umulige, er beskrevet i afsnit 4.5.

Hvis en node kompromitteres, lærer modstanderen grundet secret sharing ikke noget om backup data (med mindre *k* noder er kompromitteret). Dette er dog ikke gældende, hvis brugerens private nøgle ligeledes kompromitteres. Herved vil modstanderen få adgang til at genskabe dennes

backup, hvorved sikkerheden er brudt. Dog er det kun den kompromitterede nøgles ejers filer, der ikke længere vil være beskyttet. Det er derfor en nødvendighed for den enkelte brugers filers sikkerhed, at private nøgler ikke kompromitteres.

Når en node betragter en anden som kompromitteret ignorer den alt information fra nodens bruger. Dette gøres ved at ignorere pakker, der er signeret med brugerens private nøgle. Når den rette ejer har renset systemet for modstanderen og ønsker at deltage, vil han således stadig blive betragtet som kompromitteret af de andre noder. For at blive anerkendt igen, må han få et nyt nøglepar og et nyt certifikat fra certifikatautoriteten og autentificere sig på ny. Da brugerinformationen i det nye og gamle certifikat vil være ens (det er den samme bruger, der har fået udstedt begge certifikater), vil brugeren have samme rettigheder som tidligere overfor sine filer i systemet. Brugeren vil således atter kunne genskabe sine filer og virke på lige fod med de andre noder i systemet.

3.4 KRAV TIL SYSTEMET

Dette afsnit indeholder de mere udspecificerede krav til prototypen af backup-systemet. Disse er delt op i funktionelle og ikke-funktionelle krav. Kravspecifikationen er udarbejdet i forlængelse af problemanalysen og på baggrund af den opsamlede viden fra state of the art. Herefter beskrives fokusområderne for den videre udvikling.

3.4.1 Kravspecifikation

Funktionelle krav:

- Systemet skal kunne foretage en backup brugerne imellem.
- Når en backup-operation foretages, vælger filejeren, hvem der skal deltage (n) samt hvor mange nøgle-shares, der er nødvendige for genskabelsen af nøglen (k).
- Filejeren (og kun filejeren) skal automatisk kunne genskabe den originale krypteringsnøgle og derved genskabe data.
- Systemet skal kunne foretage periodisk opdatering af alle shares, således at kompromitterede shares i den forgangne tidsperiode bliver værdiløse for en modstander.
- Systemet skal kunne checke integriteten af shares.
- Shares, der er gået tabt eller er blevet ødelagt, skal kunne genskabes periodisk uden risiko for systemets sikkerhed, således at systemets konfiguration opretholdes.
- Systemet skal periodisk teste integriteten af det krypterede backup-data og foretage en ny replikkering, hvis dette skulle være nødvendigt.

Ikke-funktionelle krav:

- Systemet skal benytte peer-to-peer teknologi, således at dets sikkerhed og pålidelighed ikke afhænger af en enkelt server.
- Systemet skal virke og være sikkert på et LAN samt over internettet, også ved tilstedeværelsen af NAT og firewalls.
- Kompromitterede eller fejlende servere må ikke kunne få systemet til at bryde sammen eller afsløre hemmelige data.
- Systemet skal være platformuafhængigt.
- Systemet skal virke med mindst 10 brugere i en gruppe. (I en eventuel kommerciel version bør dette krav muligvis øges til ca. 20 brugere).
- Systemets sikkerhed må ikke forringes med tiden, udover den naturlige forringelse, der med tiden vil være på den grundlæggende krypteringsnøgle-længde.

3.4.2 Projektets fokusområder

Da dette projekt har til formål at udvikle et sikkert og pålideligt backup-system, vil det samlede systems fortrolighed og tilgængelighed være de primære designmål. Den videre udvikling vil således være med hovedvægt herpå. Enkelte noders pålidelighed er ikke essentielt, men disses fejl må ikke kunne få det samlede system til at fejle. Ligeledes er enkelte noders fortrolighed heller ikke kritisk, når blot en kompromitteret node ikke kan afsløre mere af systemets information, end noden indeholder.

Hvis backupsystemet skal bruges professionelt findes en række sekundære krav, som kun delvist vil blive opfyldt i dette projekt. Herunder kan nævnes stor valgfrihed af krypteringsalgoritmer og nøglelængder og stor funktionalitet til selve filhåndteringen, som det f.eks. findes i eksisterende fildelingssystemer (Napster, Gnutella osv.). Desuden vil et større fokus på brugervenlighed være nødvendigt. Der findes desuden masser af udvidelsesmuligheder, som kunne give et bedre slutprodukt. Disse vil bl.a. blive diskuteret i afsnit 7.1. De sekundære områder vil i et vist omfang blive taget med i de grundlæggende designovervejelser, sådan at de vil være let senere at foretage udvidelserne. De vil dog ikke blive designet eller implementeret til fulde.

Ved udvikling af software-systemer er der ofte fokus på systemernes svartider og ressourceforbrug. Der vil dog ikke blive fokuseret herpå i dette projekt. Dette skyldes, at backup-systemets brug stort set ikke vil være påvirket heraf. Ved at fokusere på systemets svartider kunne man sandsynligvis øge hastigheden på nogle operationer. Denne hastighedsforøgelse vil dog være minimal, da selve operationerne ikke vil være nær så tidskrævende som dataoverførsler mellem noderne. Det er altså båndbredden, der er systemets primære flaskehals. Det kunne derfor være interessant at undersøge udbredelsen af data i systemet. Man kunne f.eks. sørge for, at noder videresender data (vha. en slags træstruktur), sådan at filejeren ikke selv skal sende alt backupdata n-1 gange, hvilket vil fordele belastningen på flere noder. Endnu mere oplagt er det at overveje brugen af Rabins informationsspredningsalgoritme, da denne vil kunne mindske båndbredden (og systemets samlede forbrug af diskplads), som beskrevet i afsnit 2.2.7. Det er dog ikke oplagt, om de proaktive vedligeholdelsesoperationer kan overføres til Rabins model, hvorfor man muligvis er nødt til at opgive dele af vedligeholdelsen. Dette vil dog ikke blive beskrevet yderligere, da en undersøgelse heraf ligger uden for rammerne af dette projekt.

Desuden er hastigheden ikke væsentligt, da systemet ikke skal bruges kontinuerligt i længere perioder men blot til enkelte backupoperationer en gang i mellem. Om det tager 5 eller 50 sekunder at foretage en backup er derfor ikke af stor betydning, når filoverførslerne er meget mere tidskrævende. Ligeledes gør det ikke noget at opdateringsrutiner er langsomme, så længe sikkerheden ikke kompromitteres undervejs. Hvis systemets shares skal opdateres en gang om ugen er det uden betydning, om selve operationen tager 10 sekunder eller 10 minutter. Systemet skal dog overholde rimelige krav til brug, sådan at det ikke er for langsomt til at være brugbart i praksis. Derfor evalueres ydelsen af systemets vigtigste funktioner i afsnit 6.3.

3.5 TRUSSELSMODEL

I dette afsnit beskrives identificerede trusler mod systemet. Det vil ligeledes blive beskrevet, hvordan truslerne håndteres. Afsnittet tager udgangspunkt i en aktiv mobil modstander, der forsøger at ødelægge systemets fortrolighed eller tilgængelighed.

3.5.1 Direkte angreb mod krypteringen

En modstander kan forsøge at gætte en af systemets krypteringsnøgler vha. kryptoanalyse. Afhængig af hvilke nøgler der gættes, vil et succesfuldt angreb betyde at systemets sikkerhed eller dele heraf er brudt. Dette må naturligvis ikke kunne forekomme.

Modstanderen kan vælge flere steder at foretage et sådant angreb. Først og fremmest kan han forsøge at bryde krypteringen af det hemmelige backupdata, som er krypteret med en symmetrisk nøgle. Ellers kan han forsøge at bryde den asymmetriske kryptering, der benyttes til kommunikationen mellem noderne eller kommunikationens symmetriske sessionsnøgle, sådan at shares kan opsnappes og bruges til at rekonstruere hemmeligheden. Sidst men ikke mindst kan han forsøge angreb på det diskrete logaritmeproblem i forbindelse med den verificerbare secret sharing, da denne vil afsløre den hemmelige krypteringsnøgle. Et sådant angreb vil dog kræve, at modstanderen først har skaffet sig adgang til kommunikationen, f.eks. ved at kompromittere en af systemets noder.

Disse angreb håndteres alle ved at vælge algoritmer og nøglelængder, der kraftigt sandsynliggør, at angrebene vil mislykkes. Det bør ligeledes være omtrent lige svært at bryde de forskellige krypteringssystemer. Dette er beskrevet yderligere i afsnit 4.4.3.

3.5.2 Angreb mod en node

Modstanderen kan forsøge at kompromittere en af systemets noder. Herved opnår han følgende:

- Mulighed for at benytte den kompromitterede node, som allerede er en del af systemet, til at udføre angreb på resten af systemet.
- Information om nodens data, såsom shares og krypterede backup-filer.

Angrebet kan være et direkte angreb på systemet, hvor svagheder i protokollen eller systemets forudsætninger og omgivelser udnyttes. Dette er beskrevet i de efterfølgende afsnit. Ellers vil angreb mod noder typisk foregå ved, at modstanderen skaffer sig adgang til hele nodens system, f.eks. vha. et hul i operativsystemet. Hvis modstanderen har kontrol med operativsystemet, vil backup-applikationen ligeledes være under dennes kontrol. Ligeledes må det antages, at modstanderen kan kompromittere en node, hvis han opnår fysisk kontrol med maskinen, som denne kører på (f.eks. ved at stjæle en bærbar pc).

Hvis en node kompromitteres, kan modstanderen muligvis også få fat i brugerens private nøgle. Med denne kan modstanderen siden hen udgive sig for at være den pågældende bruger. Ligeledes vil han kunne dekryptere data sendt specielt til brugeren. Den private nøgle vil dog eventuelt kunne beskyttes vha. smart card eller anden hardware, sådan at det bliver meget vanskeligt for modstanderen at få fat i denne. Ved en sådan konstruktion vil modstanderen dog sandsynligvis stadig kunne bede noden signere pakker til andre noder (hvis noden kompromitteres mens smart card'et er til stede), hvorved nodens brugers filer vil kunne genskabes af modstanderen.

Angreb mod de systemer, som backupsystemets noder befinder sig på, håndteres ikke af det foreslåede backup-system. Det er typisk brugerens (eller administratorens) opgave at sikre det enkelte system mod angreb. Backupsystemet vil dog tage højde for, at direkte angreb mod systemets protokol kan forekomme fra autentificerede brugere som er kompromittet, idet systemets protokol designes med henblik herpå. Ligeledes håndteres det, at modstanderen får fat i noget information om systemets shares og filer ved at kompromittere en node. Dette gøres vha. proaktivt secret sharing, sådan at informationen vil være ubrugeligt for modstanderen, medmindre k noder kompromitteres i samme tidsperiode, hvilket antages at være svært. Brugeren af en kompromitteret nodes filer vil dog muligvis ikke kunne beskyttes af secret sharing modellen. Derfor er det stadig vigtigt, at brugeren beskytter sit eget system. Dette er beskrevet yderligere i afsnit 3.6.1.

3.5.3 Angreb mod systemets kommunikationsprotokol

Der findes mange måder, hvorved svagheder i en protokol kan findes, som vil betyde, at systemets sikkerhed kan brydes. Et angreb på protokollen er succesfuldt, hvis det får systemet til at afsløre værdifuld information eller holde op med at virke efter hensigten. Følgende fire typer angreb vil blive beskrevet:

- Injektionsangreb
- Modificeringsangreb
- Man-in-the-middle angreb
- Replay angreb

Ved et injektionsangreb laver modstanderen selv en datapakke, som sendes til systemet (dvs. til en eller flere noder). Modstanderen vil forsøge at lave en pakke, som med den rigtige opbygning og på det rigtige tidspunkt kan give det ønskede resultat.

Ved et modificeringsangreb opsnapper modstanderen en pakke, hvor enkelte dele modificeres, før denne sendes tilbage til systemet på samme måde som ved et injektionsangreb. Det kan ofte være lettere for modstanderen at modificere en pakke en lille smule, sådan at systemet stadig genkender pakken, men reagerer utilsigtet i forhold til den rigtige protokol.

Ved et man-in-the-middle angreb opsamler modstanderen kommunikationen fra to parter, der prøver at kommunikere. Modstanderen sørger herefter for, at sende sine egne pakker tilbage til de to parter, sådan at de stadig tror, at de kommunikerer med hinanden. Man-in-the-middle angreb beskrives oftest i forbindelse med nøgleudvekslinger, hvor to parter vil bytte nøgler. Her er det modstanderen, der ender med at have begges nøgler, uden af parterne er klar over det, da modstanderen udgiver sig for at være begge parter. Hermed brydes krypteringens sikkerhed.

Både injektionsangreb, modificeringsangreb og man-in-the-middle angreb forhindres ved udelukkende at benytte krypteret kommunikation. Herved kan modstanderen ikke sende egne pakker til systemet, selvom han har modtagernodernes offentlige nøgler til rådighed, sådan at troværdige pakker kan laves til den enkelte modtager. Dette skyldes, at han mangler en autentificeret privat nøgle til at signere pakkerne, hvorfor noderne ikke vil godtage pakkerne. Modstanderen vil kunne modificere en pakke, men resultatet vil være det samme, da det dekrypterede data vil være ændret på en uforudsigelig måde og vil derfor med stor sandsynlighed være ganske ulæseligt for systemet. Man-in-the-middle angreb vil ligeledes være umuligt, når modstanderen ikke kender de anvendte krypteringsnøgler, da han ikke kan udgive sig for at være en anden, uden at kende deres offentlige nøgle.

Systemets indledende kommunikation er dog ikke krypteret, da der først skal udveksles nøgler. Her kunne de nævnte angreb komme på tale. Dette forhindres dog ved at benytte certifikater til autentificering. Medmindre modstanderen er i stand til at forfalske et certifikat, hvilket med rimelighed kan antages at være umuligt, kan et angreb ikke lade sig gøre her.

Et angreb der dog ikke kan håndteres vha. simpel kryptering er et replay angreb. Her opsamler modstanderen pakker fra systemet og sender disse uændret tilbage på udvalgte tidspunkter. Selvom pakken er krypteret og selvom modstanderen ikke ved præcist, hvad den indeholder, vil den således stadig have mening for systemet på et andet tidspunkt. Dermed kan modstanderen muligvis få systemet til at bryde sammen eller afgive fortrolig information. Den eneste måde at undgå replay angreb er således, at designe systemets protokol til at undgå dette. Dette problem er beskrevet yderligere i afsnit 4.4.2.

Da det er en antagelse at noder kan kompromitteres af modstanderen, vil denne kunne overtage kontrollen med en node og ændre nodens opførsel i forhold til systemets rigtige protokol. Dette muliggør pludselig injektionsangreb, da modstanderen nu kan sende forståelige pakker, der er signeret med en autentificeret nøgle. Den eneste måde at håndtere disse angreb, er derfor i designet af systemets protokol. Protokoldesignet er beskrevet i afsnit 4.5.

Modificeringsangreb og man-in-the-middle angreb vil stadig ikke kunne lade sig gøre, da andre brugeres kommunikation vil være krypteret med deres personlige nøgler, som stadig er uden for modstanderens rækkevidde.

3.5.4 Denial of service

Denial of service (DoS) angreb kan foregå på mange måder. Modstanderen kan f.eks. klippe et netværkskabel over, eller slukke for en node, som han har fået fysisk kontrol med. Derved vil denne node være sat midlertidigt ud af spil. Angrebet kan ligeledes foregå ved at modstanderen sender så mange pakker til en node, at den ikke er i stand til at håndtere arbejdsbyrden, hvorefter den fejler. En nodes båndbredde vil ligeledes kunne opbruges, sådan at protokollens trafik ikke vil kunne komme igennem.

De fysiske DoS angreb vil ikke blive gennemgået yderligere, da disse ikke kan forhindres vha. software. Overbelastning af en enkelt node, kan dog håndteres på forskellig vis, ved at begrænse nodernes arbejde. Først og fremmest kan man sørge for, at undlade unødvendige returpakker, når noden bliver kontaktet. Herved kan noden spare en del arbejde og blive mere resistent overfor denne type DoS angreb. For at forhindre at modtagne DoS-pakker tager for mange af nodens ressourcer, kan man implementere funktionalitet, der sørger for at noden ikke vil modtage en ubegrænset mængde pakker fra samme node. Ved et distribueret DoS angreb (DDoS) vil dette dog ikke virke, da pakkerne her kommer fra forskellige noder.

En node, der deltager i systemet, vil kunne opbruge en anden nodes båndbredde ved kontinuerligt at sende store backupfiler til denne. Dette vil virke som et DoS angreb og det kan være svært at afgøre, hvornår afsendernoden rent faktisk laver en lovlig backup, og hvornår der er tale om DoS. Umiddelbart er der ingen let løsning på problemet, men man kan evt. vælge at lave nogle indstillinger, som lader den enkelte bruger afgøre, hvor meget man ønsker at modtage fra andre noder. Dermed vil problemerne kunne mindskes, men dette vil kræve en del administration.

Hvis secret sharing backupsystemet havde haft en klient/server struktur, ville et DoS angreb på serveren kunne sætte hele systemet ud af drift. Idet systemet er lavet med en peer-to-peer struktur, vil det være nogenlunde robust overfor DoS-angreb. Hvis en modstander angriber en node vha. DoS svarer det til, at systemets n midlertidigt mindskes med én. Så længe der stadig er k korrekte noder tilbage i systemet, vil filer stadig kunne genskabes. Hvis flere end n-k noder angribes med DoS, vil systemet ikke længere kunne fungere. Dette er dog kun midlertidigt, indtil angrebet ophører. Systemets fortrolighed er således ikke kompromitteret og tilgængeligheden er kun midlertidigt kompromitteret. Dette kan naturligvis være kritisk nok, hvis angrebne indtræffer på de rigtige tidspunkter, men da data hverken er gået tabt eller afsløret for modstanderen, må det anses for mindre alvorligt.

Den maksimale længerevarende effekt modstanderen kan få af avanceret DoS angreb er, at der kun vil være k korrekte shares i systemet. Dette opnås ved at sætte n-k servere ud af spil under en opdateringsrutine, således at disses shares bliver ubrugelige. Da der stadig vil være k korrekte og fungerende noder tilbage i systemet vil disse opdatere deres shares efter planen. Hvis flere end n-k noder sættes ud af spil, vil opdateringen slet ikke kunne foregå, hvorfor den længerevarende tilgængelighed ikke er truet. Et sådant angreb vil kunne håndteres ved, at systemet selv kan skifte konfiguration, jf. afsnit 2.2.5. Herved vil de tilbageværende k noder kunne vælge en anden konfiguration, der ikke kræver, at alle tilbageværende noder er fejlfri.

Prototypen vil af tidsmæssige årsager ikke indeholde funktionalitet til at imødegå DoS angreb.

3.5.5 Social engineering

Ved et social engineering angreb forstås, at modstanderen vha. sociale relationer skaffer sig adgang til systemer. Det mest anvendte social engineering består i at ringe til folk og bede dem afsløre deres password[15], og undersøgelser har vist, at overraskende mange vælger at gøre det, til trods for, at de ikke kender personen i den anden ende. Social engineering er en af de mest benyttede og effektive angrebsmetoder og kræver ofte ingen teknisk viden, hvorfor alle og enhver kan gøre det.

Alle systemer er sårbare over for social engineering og dette backupsystem er ingen undtagelse, hvorfor systemet aldrig kan blive mere sikkert end de mennesker, der bruger det. Hvis modstanderen kan overbevise *k* deltagere om at afsløre deres share (eller blot overbevise dem om, at han lige skal kigge på deres computer i fem minutter, hvorved han selv kan finde frem til share-informationen), vil systemets sikkerhed være brudt. Det samme vil være gældende, hvis modstanderen kan franarre private krypteringsnøgler fra tilpas mange brugere, da han herved kan læse den hemmelige kommunikation og udgive sig for at være en af brugerne.

Dette backupsystem er dog mere robust overfor social engineering, end de fleste andre systemer. Dette skyldes, at modstanderen skal overbevise *k* brugere om at udlevere deres share, hvilket vil være væsentlig meget sværere, end at franarre en tilfældig godtroende bruger sit password. Man kan desuden forestille sig, at brugerne af backupsystemet ofte vil være spredt ud over mange dele af verden, hvilket besværliggør angrebet yderligere.

3.6 VURDERING AF RISICI

En risiko kan defineres som et potentielt problem, som et system eller dets brugere kan opleve. Dette afsnit vil i forlængelse af trusselmodellen beskrive og diskutere de tilbageværende risici efter håndteringen af de identificerede trusler. Der fokuseres på problemer, hvor fortroligheden eller tilgængeligheden brydes for enkelte brugere eller for hele systemet.

Følgende tre punkter kan bruges til at vurdere en given risiko:

- Tabet i forbindelse med en begivenhed.
- Sandsynligheden for at begivenheden opstår.
- Muligheder for at begrænse sandsynligheden og konsekvenserne.

Backupsystemet kan kompromitteres på følgende to måder:

- En brugers node kompromitteres. Modstanderen får vha. brugerens private nøgle rekonstrueret brugerens hemmelige filer.
- Systemets grundlæggende antagelser om fortrolighed og tilgængelighed brydes, dvs.:
    - Mindre end *k* korrekte noder tilbage i systemet. (tilgængelighed)
    - Mere end *k*-1 noder er kompromitteret af modstanderen. (fortrolighed)

[15] Dette kan evt. gøres under dække af, at man sidder i virksomhedens IT-administration el. lign.

3.6.1 Kompromittering af en enkelt node

Hvis en brugers node kompromitteres og dennes filer afsløres for modstanderen er tabet begrænset til denne brugers filer, samt eventuelle shares, som noden måtte holde for andre noder. Begivenheden er således kritisk for brugeren, men ikke for det samlede system. I en standardopsætning, hvor der ikke er gjort noget for at øge det lokale systems sikkerhed, er sandsynligheden ikke stor for at modstanderen gætter netop hvilken node, der skal angribes og at modstanderen er i stand til at gennemføre angrebet. Sandsynligheden er dog til stede og afhænger altså primært af brugerens eget system og opsætningen af dette. Brugeren kan sikre sit system ved at holde sin software opdateret og ved at installere firewall osv. (da dette er uden for projektets rammer, vil det ikke blive beskrevet yderligere). Ved at brugeren sikrer sit system, mindskes risikoen væsentligt.

Desuden vil brugeren med fordel kunne foretage backup'en med et nøglepar og et certifikat, der er gemt på hardware (f.eks. et smart card), sådan at det er sværere for modstanderen at få fat i brugeren private nøgle. For yderligere at optimere sikkerheden kan brugeren logge af, efter backup'en er fuldendt (han kan evt. logge på med en anden bruger, der ikke har de samme rettigheder overfor de hemmelige filer). Herved findes der ikke længere én kritisk node, der kan afsløre brugeren filer. Brugeren er dog nødt til at logge på med den oprindelige bruger, når filerne skal rekonstrueres. På denne måde indskrænkes risiko-perioden til tidspunktet, hvor backup'en foretages og tidspunktet, hvor filerne rekonstrueres. Hvis modstanderen skal gennemføre angrebet skal han således gætte hvilken node han skal angribe og på hvilket tidspunkt, hvilket må betragtes som en svær opgave.

3.6.2 Kompromittering af det samlede system

Hvis det samlede systems grundantagelser brydes, er systemets sikkerhed brudt. Herved er hverken fortrolighed eller tilgængelighed af data garanteret. Sandsynligheden for at dette kan forekomme, afhænger af flere faktorer. Først og fremmest er risikoen for at en enkelt node kompromitteres væsentlig. Dernæst er risikoen for normale systemfejl, hvor enkelte noder fejler, en vigtig faktor. Mulige angreb på systemets protokol udgør også en risiko, så længe protokollen ikke er bevist sikker, hvilket ikke er en let opgave. Sidst men ikke mindst er størrelsen af n og k i den valgte secret sharing model væsentlig for systemets sikkerhed.

Hvis flere noder kan kompromitteres på (næsten) samme tid, er det et væsentligt større problem for systemet, end en enkelt kompromitteret node (som beskrevet ovenfor), da modstanderen på den måde nærmer sig den magiske grænse for, hvornår det hemmelige data afsløres. I praksis vil en modstander dog ofte have lettere ved at kompromittere den anden og tredje node end den første. Dette skyldes, at systemerne ofte ligner hinanden og hvis modstanderen er heldig, har flere af nodernes bagvedliggende systemer de samme sikkerhedshuller[16]. Disse kan hurtigt udnyttes på samme måde, hvorved et stort antal noder kan kompromitteres på samme tid. Derfor er det ikke blot en feature, at systemet skal kunne virke på forskellige platforme (jf. kravspecifikationen i afsnit 3.4.1). Derimod kan det være med til at øge systemets samlede sikkerhed markant, hvis brugerne af systemet bevidst vælger forskellige platforme og forskellige

[16] Dette udnyttes bl.a. hyppigt af diverse orme og bagdøre, som spredes på internettet.

sikkerhedsopsætninger (f.eks. forskellige firewalls), da dette besværliggør modstanderens bestræbelser på at få fat på *k* shares.

Sandsynligheden for at noder eller deres bagvedliggende systemer til tider vil fejle, enten pga. software eller hardware, er stor. Faktisk kan det garanteres, at disse fejl vil opstå før eller siden. Hyppigheden afhænger af det bagvedliggende system og dettes hardware samt fejlraten i selve backupsystemet. Hvis disse fejl opstår, vil noden muligvis holde op med at virke. Desuden risikeres, at nodens lokale data mistes. Dette er dog ikke kritisk for hverken den enkelte bruger eller for det samlede backup-system, da noden kan genstartes og data kan genskabes vha. proaktivt secret sharing. Hvis hyppigheden af fejl er så stor, at systemet ikke altid har *k* fungerende noder vil problemet dog være større. Forebyggelse af disse problemer kan foregå ved at sørge for kontinuerligt at rette alle kendte fejl i backupsystemet. Det bagvedliggende system kan gøres mere stabilt ved at sørge for, holde styresystemet opdateret. Desuden kan robustheden øges ved, at backupnoden er den eneste applikation på maskinen, da der herved er færre programmer, der kan lave fejl.

Et succesfuldt angreb mod systemets protokol vil ofte være ensbetydende med, at systemets sikkerhed er brudt. De identificerede angrebstyper mod protokollen er beskrevet i afsnit 3.5.3. Sandsynligheden for at disse kan foretages med succes afhænger primært af protokollens design. Problemerne forebygges således ved at sørge for at systemets design er så gennemarbejdet og fejlfrit som muligt.

Der er således mange fejl og angreb, der kan lede til et brud på systemets grundantagelser. Der skal dog ske mange fejl på samme tid, før effekten er kritisk (under antagelse af, at systemets protokol ikke indeholder større fejl), hvorfor den samlede sandsynlighed må betragtes som værende lille. Tilsvarende er der ca. 25 % sandsynlighed for, at et fly vil opleve en fejl i forbindelse med en flyvning. Sandsynligheden for at flyet styrter, er derimod meget lavere (heldigvis), da langt de fleste fejl er ubetydelige.

Forebyggelse af samlede risiko kan gøres ved at mindske de enkelte risici som beskrevet ovenfor. Desuden kan valget af *n* og *k* spille en stor rolle for sikkerheden, hvorfor dette bør overvejes grundigt.

3.7 OPSUMMERING

Den grundlæggende model beskriver overordnet hvilke operationer backupsystemet skal kunne udføre. Desuden er det beskrevet hvilke slags data en node (og bruger) skal være i besiddelse af, når systemet er operationelt. De nødvendige data er:

- Et asymmetrisk nøglepar.
- Et certifikat.
- En liste af andre brugeres certifikater.
- Administrativt data om andre noder, sådan at disse kan kontaktes, når det er nødvendigt.
- Krypteret backup data, share-data og andre administrative data i forbindelse med secret sharing modellen.

Den benyttede secret sharing model skal både være verificerbar og proaktiv for at leve op til sikkerhedskravene for systemet. På baggrund af modellen og viden fra state of the art blev kravene til systemet udspecificeret i kravspecifikationen.

Konklusionen på analysen af trusler og risici er, at der findes mange muligheder for fejl og angreb mod systemet, der vil have betydelig effekt. Dog kan disse begrænses drastisk, ved at tage relevante forholdsregler, hvorved systemet kan opnå en høj sikkerhed. Sikkerheden vil dog altid være afhængig af, at systemets design er uden større fejl.

Der er umiddelbart to hovedformål for systemet. Det ene er at fungere som gensidig backup-applikation mellem forskellige private internetbrugere, der ønsker at gemme data hos hinanden, for at forhindre, at det går tabt. Den anden funktion er at lette dele af fil-backupfunktionen for virksomheder, sådan at dette bliver sikkert, pålideligt og nogenlunde billigt. Det vil uden tvivl være virksomhederne, der har de højeste krav til sikkerheden for et sådant system. Til gengæld vil disse også have størst mulighed for at vurdere systemets sikkerhed og styre opsætningen heraf, hvilket vil være væsentligt, før det tages i brug.

4 DESIGN

Dette kapitel vil på baggrund af modelanalysen og kravspecifikationen beskrive backupsystemets design med hovedvægt på systemets grundlæggende kommunikationsprotokol og funktionalitet. For at skabe et bedre overblik over systemets struktur, bliver en del af designfasen gennemført vha. UML. Et komplet programdesign i UML, hvor alle cases er inkluderet, vil tage for lang tid i forhold til projektets omfang og er således ikke hensigten. UML vil derfor kun blive benyttet til at beskrive systemets protokol og umiddelbare funktionalitet på en overordnet facon. Formålet med denne overordnede beskrivelse er at skabe overblik, som er tilstrækkeligt til at påbegynde en implementering. En del af systemets detaljer vil således ikke blive berørt. Af UML-værktøjer benyttes simple use case-beskrivelser samt en række sekvensdiagrammer.

4.1 USE CASE-BESKRIVELSER

Dette afsnit indeholder de vigtigste use case beskrivelser. Disse forklarer, hvordan applikationen skal opføre sig overfor brugeren ved brug af backupsystemets grundfunktionalitet. Systemets special-cases behandles ikke her.

4.1.1 Opstart af programmet

USE CASE: Brugeren starter programmet for første gang uden at have et gyldigt certifikat.

Brugerens handling	**Systemets svar**
1. Programmet startes	2. Et password vindue vises. Vinduet indeholder et password-felt samt knapperne ”Ok”, ”Register” og ”Quit”. Her kan brugeren enten indtaste et password for et eksisterende certifikat (dvs. for den private nøgle hørende til certifikatet) og trykke ”Ok” eller brugeren kan vælge at starte oprettelsen af et nyt certifikat ved at trykke ” Register”.
3. Brugeren trykker på ” Register”-knappen.	4. Et ”Opret-certifikat-vindue” vises. I dette vindue findes en række tekstfelter, hvori personlige data, påkrævet for at lave et certifikat, kan indtastes. Desuden findes to password-felter, hvor brugeren kan angive sit password til senere brug. Nederst er en ”Ok”-knap og en ”Cancel”-knap.
5. Brugeren indtaster personlige oplysninger. Derefter indtastes password to gange (for at undgå tastefejl). Til sidst trykker brugeren ”Ok”.	6. En fil kaldes peer.csr (csr = certificate signing request) dannes i programmets rod-bibliotek. Desuden dannes filen peer.keystore, som

	indeholder brugerens nøglepar. Denne er beskyttet af brugerens valgte password. Herefter lukkes programmet med en besked til brugeren om, hvordan et certifikat skal anskaffes.
7. Brugeren trykker ”Ok” til systemets besked. Derefter sender brugeren filen peer.csr til sin CA og identificerer sig overfor denne. Som svar herpå vil CA’en lave brugerens certifikat. Dette sendes sammen med CA’en eget certifikat tilbage til brugeren. Brugeren gør herefter begge certifikater tilgængelige for backup-systemet. Nu kan programmet startes igen (se næste use case)	8. Dette skridt er ikke en del af backup-systemet, der således ikke er involveret.

USE CASE: Brugeren starter programmet med et gyldigt certifikat til stede. Brugeren logger ind og opretter en ny peer-gruppe.
(Pkt. 1-2 er de samme, som i forrige use case. Disse er derfor udeladt.)

Brugerens handling	**Systemets svar**
3. Brugeren indtaster sit personlige password og trykker på ”Ok”-knappen.	4. Et ”Gruppetilslutnings-vindue” vises. Vinduet indeholder en liste over alle eksisterende peer-grupper på netværket. Denne opdateres løbende. Derudover rummer vinduet mulighed for at tilslutte sig - og oprette grupper vha. to knapper ”New” og ”Join”. Desuden findes en ”Quit”-knap, der afslutter programmet.
5. Brugeren trykker på ”New”-knappen.	6. Et ”Opret-gruppe-vindue” dannes. I dette vindue kan indtastes gruppenavn og gruppepassword. Desuden findes en ”Ok”-knap og en ”Cancel”-knap.
7. Brugeren indtaster det gruppenavn og et gruppepassword. Gruppepassword indtastes to gange for at forhindre tastefejl. Herefter trykkes ”Ok”.	8. Opret-gruppe-vinduet forsvinder og programmets hovedvindue dannes. Brugeren er nu første medlem af den oprettede backup-gruppe.

USE CASE: Brugeren logger ind og slutter sig til en eksisterende peer-gruppe. (Pkt. 1-4 er de samme, som i forrige use case. Disse er derfor udeladt.)

Brugerens handling	**Systemets svar**
5. En eksisterende peer-gruppe vælges fra listen. Herefter trykkes ”Join”.	6. Et ”Login-vindue” fremkommer. Her kan et gruppepassword indtastes. Desuden indeholder vinduet en ”Ok”-knap og en ”Cancel”-knap.
7. Det korrekte gruppepassword indtastes og der trykkes ”Ok”.	8. Login-vinduet forsvinder og programmets hovedvindue dannes. Brugeren er nu medlem af den valgte backup-gruppe.

4.1.2

Hovedvinduets operationer

Programmets hovedvindue indeholder en liste af gruppemedlemmer samt en række knapper til at bruge systemets funktionalitet og en menubar, hvorfra yderligere funktionalitet kan tilgås. Desuden indeholder vinduet en liste over filer, som brugeren har taget backup af hos andre noder, samt en liste over filer, som andre har taget backup af hos brugeren.

Disse use cases foregår efter brugeren har logget korrekt på systemet og er blevet medlem af en backupgruppe.

Use case: Lav backup: Et gruppemedlem laver en backup.

Brugerens handling	**Systemets svar**
1. Brugeren venter på at alle ønskede peers har meldt sig ind i dennes backupgruppe.	2. Hovedvinduets liste over gruppemedlemmer opdateres løbende, når medlemmer slutter sig til.
3. Brugeren trykker på ”Backup file”-knappen.	4. En fil-browser vises. Denne indeholder en standard filhåndtering samt knapperne ”Select” og ”Cancel”
5. Brugeren vælger en fil i fil-browseren og trykker på ” Select”-knappen.	6. Fil-browseren forsvinder. Et konfigurations-vindue vises. Dette indeholder en liste over backupgruppens medlemmer. Derudover findes et tekst-felt, hvori antallet af nødvendige nøgle-shares til en eventuel restore-operation kan defineres. Desuden indeholder vinduet en ”Ok”-knap og en ”Cancel”-knap.
7. Brugeren vælger de medlemmer, der skal deltage i backupoperationen fra medlemslisten. Derefter indtastes et heltal i tekstfeltet. Til sidst trykkes ”Ok”.	9. Systemet gennemfører operationen. Herefter vises en status-meddelelse til brugeren. Desuden tilføjes filnavnet til hovedvinduets liste over filer, der er taget backup af (”My shared files”).

Use case: Modtag backup: Et gruppemedlem modtager en backup af en anden brugers fil.

Brugerens handling	**Systemets svar**
1. Brugeren gør ingenting.	2. Filnavnet på den modtagne fil tilføjes til hovedvinduets liste over andre brugeres filer, der gemmes på denne node (”Received files”).

Use case: Restore: En bruger ønsker at genskabe en fil, der tidligere er delt i systemet.

Brugerens handling	**Systemets svar**
1. Brugeren vælger filen i ”My shared files”-listen og trykker på ”Restore file”-knappen.	2. En dialog vises, hvor brugeren spørges, om denne er sikker på, at operationen ønskes foretaget. Dialogen indeholder både en ”Ok”-knap og en ”Cancel”-knap.
3. Brugeren trykker ”Ok”.	4. Systemet foretager nøgle-rekonstruktionen og dekrypterer filen. Derefter får brugeren vist en meddelelse, der angiver, om operationen har været succesfuld.

Use case: En bruger ønsker at slette en backup af en fil, der tidligere er delt i systemet.

Brugerens handling	**Systemets svar**
1. Brugeren vælger filen i ”My shared files”-listen og trykker på ”Delete sharing”-knappen.	2. En dialog vises, hvor brugeren spørges, om denne er sikker på at operationen ønskes foretaget. Dialogen indeholder både en ”Ok”-knap og en ”Cancel”-knap.
3. Brugeren trykker ”Ok”.	4. Systemet sletter den krypterede fil samt tilhørende nøgle-shares hos alle, der har deltaget i backup-operationen.

Use case: Refresh shares: En bruger ønsker at opdatere alle shares hørende til en bestemt fil.
(Denne use case kan evt. foregå automatisk med en timer (uden brugerens indblanding) for at sikre systemets fortrolighed over længere tid)

Brugerens handling	**Systemets svar**
1. Brugeren vælger filen i ”My shared files”-listen eller ”Received files”-listen og trykker på ”Refresh shares”-knappen.	2. Systemet opdaterer shares tilhørende den valgte fil.

Use case: Integrity check: En bruger ønsker at checke en fils integritet i systemet.
(Denne use case kan evt. foregå automatisk med en timer (uden brugerens indblanding) for at opretholde backup’ens integritet over længere tid)

Brugerens handling	**Systemets svar**
1. Brugeren vælger filen i ”My shared files”-listen eller ”Received files”-listen og trykker på ”Integrity check”-knappen.	2. Systemet checker shares og krypterede filer. Eventuelle fejl eller mangler rettes.

Use case: Find lost shares: En bruger har mistet nogle shares og ønsker disse genskabt.
(Denne use case kan evt. foregå automatisk (uden brugerens indblanding) for at opretholde systemets integritet over længere tid)

Brugerens handling	**Systemets svar**
1. Brugeren trykker på ”Request lost shares” i menu-baren.	2. Information fra de andre gruppemedlemmer afslører, om brugeren har mistet et eller flere shares. Hvis dette er tilfældet hentes den tilstrækkelige information til at starte et integritetscheck på hvert mistet share.

4.2 APPLIKATIONENS HOVEDSTRUKTUR

Når flere operationer skal kunne foregå parallelt, er det nødvendigt at starte flere tråde (*threads*). I en backupapplikation skal en del ting kunne foregå på samme tid. Dette afsnit beskriver backupapplikationens vigtigste tråde, deres funktioner og deres sammenspil.

Først og fremmest skal programmets hovedtråd, kaldet *Control*, stå for applikationens primære funktionalitet, såsom håndtering af data og styring af protokoller. *Control* modtager alt data og bestemmer, hvornår data skal sendes til andre noder. Det er således *Control*, der styrer programmets øvrige tråde. Ved at Control "bestemmer", er det lettere at sikre, at kritiske variable ikke tilgås fra flere tråde på samme tid.

Samtidig skal applikationen kunne modtage kommunikationspakker fra andre noder, hvilket håndteres af en separat tråd kaldet *packetReciever*. Denne tråd kan modtage pakker fra alle noder, idet forbindelserne afbrydes, når en pakke er modtaget. Tråden sørger således for at modtage en pakke, sende denne videre til *Control* uden ophold og derefter være klar til at modtage endnu en pakke (fra den samme eller en anden node).

Applikationen skal ligeledes kunne sende kommunikationspakker til andre noder. Hvis modtagernoden er svær at få fat på, risikeres det, at operationen tager (forholdsvis) lang tid. Derfor er det ikke nok med én tråd, der sender kommunikationspakker. I stedet haves en tråd kaldet *packetSender* for hver node, som applikationen skal kunne kommunikere med. Control sender således data til en *packetSender*, når der skal sendes data til en anden node.

Filer skal ligeledes kunne modtages og sendes, uden at resten af applikationen blokerer. En fil kan dog tage lang tid at modtage eller sende, så hvis flere filer skal kunne modtages eller sendes på samme tid, kræves en tråd til hver fil. Derfor oprettes en række tråde kaldet *fileReceiver* og *fileSender*. Disse oprettes og stoppes dynamisk, når en applikation skal modtage eller sende filer. En *fileSender* henter en fil direkte fra harddisken og en *fileReceiver* gemmer filen direkte på harddisken. Derudover sender en *fileReceiver* en besked til Control, når en fil er modtaget.

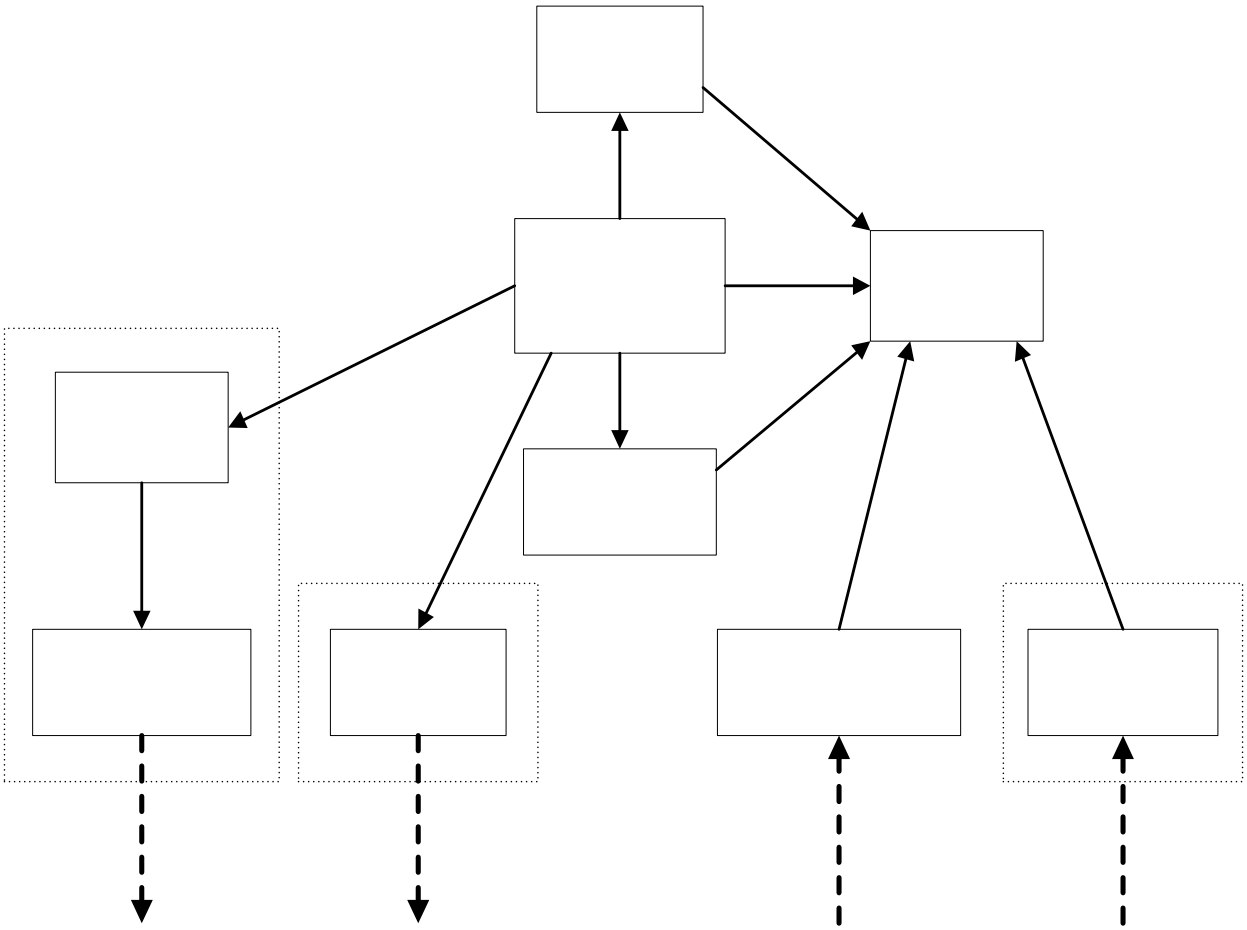
Den grundlæggende netværksarkitektur baseres på JXTA, hvilket beskrives i afsnit 4.3. JXTA-systemet skal kunne sende og modtage data (JXTA discovery events) uafhængigt af systemets øvrige tråde. Dette benyttes til at finde backup-grupper og gruppemedlemmer vha. JXTA-advertisements. Derfor oprette en separat tråd kaldet *jxtaManager*, som styrer JXTA-systemet og sender information til *Control*.

Den grafiske brugergrænseflade (GUI) skal ligeledes kunne fungere parallelt med resten af systemet. Ellers kunne man risikere, at denne ville være ubrugelig, mens systemet behandler en pakke fra en anden node, hvilket ikke er hensigtsmæssigt. Derfor laves denne i sin egen tråd, som udelukkende kommunikerer med *Control*.

Da *Control* kan få jobs (kommandoer eller beskeder) fra mange forskellige tråde, og da disse kan kræve noget tid at udføre, er der brug for en jobkø-tråd (jobMonitor), hvis eneste formål er at tage imod jobs fra andre tråde og aflevere dem til *Control*, når denne har tid. Ellers ville man

f.eks. kunne risikere, at *packetReceiver*-tråden ikke ville kunne aflevere en modtaget pakke til *Control* hurtigt nok (hvis *Control* er optaget med noget andet), hvorved den ikke bliver klar til at modtage nye pakker med det samme. Den samme type jobkø er anvendelig ved *packetSender*-trådene. Når *Control* vil sende en pakke til en node, skal denne afleveres til den rette *packetSender*. Som nævnt kan denne være optaget, hvis den tidligere sendeoperation endnu ikke er gennemført. For derfor at undgå, at Control skal vente på, at *packetSender*-tråden bliver ledig til at modtage endnu en pakke, benyttes en jobkø-tråd til hver *packetSender*.

Applikationens beskrevne hovedstruktur er illustreret herunder. Tråde markeret med en ”*”, kan der eksistere flere af på samme tid.



Figur 7 - På figuren ses programmets tråde og disses indbyrdes kommunikation. Hver solid firkant symboliserer en tråd og pilene symboliserer funktionskald. Stiplede firkanter med en ”*” angiver, at der kan eksistere flere af disse tråde. Øvrige tråde findes kun en gang i hver applikation. Stiplede pile symboliserer netværkskommunikation til/fra systemets andre noder.

4.3 GRUNDLÆGGENDE NETVÆRKSARKITEKTUR

Kravet til den grundlægende struktur i backupsystemet er, at alle noder skal kunne kommunikere med hinanden vha. peer-to-peer, selvom disse er placeret på forskellige systemer og platforme. Det er desuden ønskværdigt, hvis dette kan foregå på tværs af så mange forhindringer (firewalls, netværkstypologi osv.) som muligt. Strukturen skal indeholde en gruppeinddeling af noder, sådan at de beskrevne backupgrupper kan dannes. Hvis disse krav er opfyldt, vil backupsystemets egne protokoller selv håndtere resten.

JXTA indeholder de nødvendig byggesten til at skabe en sådan peer-to-peer struktur. Da det desuden er open source, virker det naturligt, at benytte JXTA til den grundlæggende netværksarkitektur i stedet for at opfinde den dybe tallerken igen. Det høje abstraktionslag i JXTA, som kan benyttes til at skabe mange forskelligartede peer-to-peer systemer, kan dog medføre en forringelse af effektiviteten. I dette projekt er effektiviteten dog ikke betragtet som kritisk, jf. 3.4.2.

For at forhindre backupsystemets gruppestruktur i at blive blandet med andre JXTA-baserede systemers JXTA-grupper haves én fælles JXTA-gruppe for backupsystemet. Denne standardgruppe melder alle backup-noder sig ind i, når de starter op. Derefter benyttes standardgruppen kun til at publicere information om egentlige backupgrupper. En node kan således lede efter gruppe-advertisements i standardgruppen og vil på den måde finde frem til de eksisterende backupgrupper. Noden kan herefter melde sig ind i disse eller at oprette en ny backupgruppe. Før systemets egentlige funktionalitet kan benyttes, skal noden finde de andre noder i backupgruppen, som noden er medlem af. Dette gøres ved at lede efter node-advertisements i denne backupgruppe. Herved finder noden de eksisterende medlemmer af gruppen, hvorefter kommunikationen kan oprettes.

4.4 NETVÆRKSSIKKERHED

Dette afsnit beskriver de vigtigste tiltag til at beskytte kommunikationen mellem systemets noder mod angreb.

4.4.1 Grundlæggende pakkestruktur

I backupsystemet skelnes mellem at sende kommunikationspakker og filer. Filer der sendes i systemet er altid backup-filer og disse eksisterer kun i krypteret form i systemet. Da filerne kun skal sikres for fortrolighed og integritet (og ikke for non-repudiation), skal der ikke tages yderligere højde for kryptering ved filoverførsler. Ved kommunikationspakker forstås alle systemets øvrige pakker, der har til formål at udføre systemets protokol. Disse indeholder ofte fortrolig information, som skal beskyttes. Bortset fra den indledende certifikatudveksling er alle pakker i systemet derfor krypteret. Certifikatudvekslingen beskrives i 4.5.2.

Til at sende en pakke med noget data benyttes en tilfældig genereret symmetrisk sessionsnøgle samt brugernes asymmetriske nøgler. En netværkspakke består derfor af fire dele:

Den første del er selve pakkens data, der er krypteret med en tilfældig sessionsnøgle. Den anden del er sessionsnøglen, der er krypteret med modtagerens offentlige nøgle. Herved er det kun den rette modtager, der med sin private nøgle kan dekryptere og få fat i den tilfældige symmetriske nøgle, der beskytter data. Den tredje del af pakken er en signatur, hvor afsenderens private nøgle er brugt til at lave en signatur af data (inden dette krypteres med den symmetriske nøgle). Signaturen laves ved først at lave en hash af data og siden hen udføre krypteringen. Herved undgås det, at den asymmetriske nøgle skal kryptere store datamængder, hvilket ville være meget tidskrævende. Signaturen kan benyttes til at sikre, at afsenderen ikke udgiver sig for at være en anden, og det kan således bevises, hvem der sendte pakken. Pakkens sidste del er afsenderens offentlige nøgle. Denne er frit tilgængelig og behøves ikke blive holdt fortrolig. Formålet med denne del er blot, at modtageren hurtigt kan identificere afsenderen. Dermed ved modtageren med det samme, hvis nøgle, der skal bruges til verificeringen af signaturen, sådan at dette ikke tager unødvendigt lang tid. Kortere forklaret ser en netværkspakke fra A til B altså ud som følger:

Krypteret netværkspakke[17]:

{ $(data)_{sessionKey}$;
$(sessionKey)_{BpublicKey}$;
Signatur = $(hash\ af\ data)_{AprivateKey}$;
$A_{publicKey}$;
}

Denne pakkestruktur sikrer fortroligheden af data. Desuden er integriteten af data beskyttet, sådan at modtageren vil opdage eventuelle ændringer i data. Sidst men ikke mindst giver signaturen non-repudiation, som kan bruges til at afgøre, hvem der forsøger at ødelægge protokollen, sådan at disse kan ekskluderes. Strukturen håndterer altså man-in-the-middle angreb og modificeringsangreb fra både eksterne og kompromitterede interne noder. Desuden beskyttes mod injektionsangreb fra eksterne noder. En kompromitteret node, der er en del af systemet, vil ikke kunne forhindres i at foretage injektionsangreb vha. kryptografi, hvorfor designet af protokollen skal tage hensyn til, at disse kan forekomme. Dette gøres ved at sikre, at en node ikke kan ødelægge resten af systemet eller få mere information ud af andre noder, end oprindelig tiltænkt, uanset hvilken pakke der sendes.

Kommunikationen kan ikke beskyttes mod DoS angreb vha. den klassiske kryptografi, og dette vil ikke blive beskrevet yderligere.

4.4.2 Håndtering af replay angreb

Ved at hver pakke indeholder noget information, som adskiller pakken fra andre lignende pakker, kan replay angreb undgås. Dette kunne f.eks. være en tid (*timestamp*) eller et unikt identifikationsnummer. Hvis en pakke indeholder et timestamp kan modtageren se, om pakken er

[17] Notationen $(X)_{key}$ betyder, at X er krypteret med nøglen "key".

sendt for nylig (antaget at modtagerens og afsenderens tid er synkroniseret, hvilket ikke nødvendigvis er en let opgave). Hvis ikke dette er tilfældet, er den sendt som et forsøg på et replay angreb og kan således ignoreres.

I dette backupsystem er det dog ikke engang væsentligt, om pakken er sendt for nylig. I stedet er det kun interessant, at den samme pakke ikke har været sent før. Dette kan let klares vha. et timestamp. Når Alice gerne vil sende en pakke til Bob gøres derfor følgende:

- Alice tilføjer sin lokale tid til pakken, inden denne krypteres og sendes til Bob.
- Bob modtager pakken og dekrypterer denne.
- Bob kontrollerer, at den lokale tid i pakken fra Alice er større end tiden fra den forrige pakke.
- Hvis Bob accepterer pakken, gemmes tiden fra denne, sådan at Bob kan sammenligne med tiden fra den næste pakke fra Alice.

Løsningen kræver således, at hver node gemmer en tid for hver af de andre noder, som denne skal kommunikere med. Medmindre noder stiller tiden tilbage på deres ure, vil der ikke findes to pakker fra den samme node, der er ens. Da tiden hele tiden skrider fremad, vil replay angreb således være umulige, da en modstander skal kunne bryde krypteringen for at lave en pakke med en tid, der vil blive accepteret af modtagernoden.

Den beskrevne løsning har dog det problem, at tiden i mange systemer (inkl. JAVA) måles som en 32 bit værdi, som starter fra 1970. Denne vil ikke vare evigt, hvilket i praksis for dette system betyder, at alle ure vil starte forfra. Dermed vil alle pakker blive betragtet som replay angreb, indtil systemet genstartes, hvorefter tidligere pakker vil kunne bruges til lave replay angreb. Dette kan dog løses simpelt med en ekstra tæller (eller en anden udvidelse af det originale nonce). I denne prototype er den ovenstående løsning dog tilstrækkelig, og denne vil blive brugt i implementeringen.

4.4.3 Valg af krypteringsalgoritmer og nøglelængder

Som nævnt i state of the art er det vigtigt, at de forskellige krypteringsformer er nogenlunde lige kraftige. Ellers vil modstanderen blot angribe det svageste sted, hvorved den kraftigere kryptering bliver overflødiggjort. I backup-systemet benyttes kryptering til flere dele, hvorfor dette skal overvejes nøje. Følgende kryptografiske værktøjer benyttes i applikationen:

- Symmetrisk kryptering af backup-data (filer), og af data i kommunikationspakkerne.
- Asymmetrisk kryptering af dele af kommunikationspakkerne.
- Hashalgoritme til udarbejdelsen af signaturer i kommunikationspakkerne samt til sikring af backup-datas integritet (fil-integritet).
- Det diskrete logaritmeproblem til verificering af shares.

Selvom det diskrete logaritmeproblem ikke kan angribes direkte, da kommunikationen er krypteret med den nævnte struktur, er det en antagelse, at noder kan kompromitteres. Derved kan modstanderen få fat i verificeringsinformationen, hvorfra hemmelige data kan udledes, hvis det

diskrete logaritmeproblem kan løses. Derfor skal styrken på logaritmeproblemet være lige så stor, som de andre krypteringsmekanismer.

Som symmetrisk krypteringsalgoritme vælges AES med *cipher block chaining* (CBC). Valget af AES skyldes, at denne betragtes som en af de mest sikre eksisterende algoritmer. Desuden er den forholdsvis hurtig og har en standard nøglestørrelse på 128 bit. AES er en ofte benyttet standard, og der findes flere implementeringer af denne, der er lette at anvende. 128 bit AES nøgler betragtes som tilstrækkelig til stort set alle formål. Derfor benyttes denne nøglestørrelse her.

RSA vælges til den asymmetriske kryptering, da denne ligeledes er en af de mest udbredte algoritmer. Algoritmen er ikke lige så hurtig, som løsninger baseret på elliptisk kurve kryptografi, men til gengæld har den været afprøvet i flere år, og det antages på denne baggrund, at RSA ikke har kritiske svagheder. Hvis RSA's styrke skal matche en 128 bit AES-nøgle, skal RSA nøglerne være væsentligt større. Ifølge Schneier [1] svarer brute-force styrken på en symmetrisk 128-bit nøgle til en asymmetrisk nøgle på 2304 bit. På denne baggrund vælges en RSA-nøglestørrelse på 2048 bit som værende nogenlunde tilsvarende.

Som hashalgoritme vælges SHA-1, da denne betragtes som værende sikker med sine 160 bit hashværdier. Svagere hashalgoritmer vil gøre denne væsentlig svagere end systemets øvrige krypteringsmekanismer.

Som beskrevet i state of the art er det nogenlunde lige så svært at angribe det diskrete logaritmeproblem, som det er at faktorisere store primtal. Derfor vælges nøglerne til det diskrete logaritmeproblem (dvs. systemets hemmelighed og de tilhørende shares) til at være 2048 bit ligesom RSA-nøglerne, da styrken herved vil matche. Nøglestørrelsen til det diskrete logaritmeproblem vil kunne mindskes væsentligt ved at bruge elliptisk kurve kryptografi. Dette er dog udeladt i denne prototype. Dette skyldes, at elliptisk kurve kryptografi er en del mere besværligt end det normale logaritmeproblem og den opnåede hastighedsforøgelse vil ikke være særlig stor, da effektiviteten ved anvendelsen af det normale diskrete logaritmeproblem til verificering af shares allerede er stor [9].

4.5 KOMMUNIKATIONSPROTOKOLLER

Dette afsnit beskriver systemets grundlæggende kommunikationsprotokoller og funktionaliteten internt i en applikation. Desuden illustreres opførslen vha. en række sekvensdiagrammer inspireret af UML- metodologien. Dette er ikke gjort på et objektniveau og skal således ses som en mere overordnet beskrivelse af funktionaliteten. Desuden indeholder sekvensdiagrammerne ikke alle detaljer og special-cases[18]. Dette er mindsket for at bevare overblikket og pga. de tidsmæssige rammer for projektet.

[18] Med special-cases menes cases, der kun opstår under specielle omstændigheder. Disse kan både være af funktionel og sikkerhedsmæssig karakter og i en eventuel kommerciel version, bør systemet tage højde for disse.

4.5.1 Generelt om sekvensdiagrammerne

Sekvensdiagrammerne beskriver både netværksprotokollen og funktionaliteten internt i en applikation. For at simplificere diagrammerne modelleres hver node som 3 tråde: En *main*-tråd, der laver det meste af applikationens arbejde, en *send*-tråd, der sørger for at sende pakker til de rette noder og en *receive*-tråd, der modtager pakker fra andre noder. GUI-tråden er f.eks. ikke medtaget, hvorfor kald fra brugeren vises som kald direkte til *main*-tråden. De fleste diagrammer indeholder en bruger og dennes applikation (node) samt en anden node, der enten kan symbolisere resten af backupgruppens noder eller enkelte individuelle noder. Simplificeringen er foretaget for at gøre modellerne mere overskuelige. Af samme årsag er kun den vigtigste del af protokollerne er medtaget. Diagrammerne læses oppefra og ned og den beskrevne funktionalitet foregår i den tråd, som den er placeret under.

Netværkspakker illustreret i sekvensdiagrammerne er altid krypteret på den beskrevne måde (jf. afsnit 4.4.1) medmindre andet er angivet. Pakkerne er typisk kun beskrevet som en type NetData (f.eks. sharingNetData). Her er det underforstået, at NetData er pakket ned i en krypteret netværkspakke som siden hen er sendt til modtageren. Alle NetData objekter indeholder desuden afsenderens tid, som bruges til håndtering af replay angreb.

Metode-argumenter er kun medtaget særlige steder, hvor de er vigtige og ikke giver sig selv. Hvis de er vigtige, beskrives de typisk i en tilhørende note. Dette er gjort for at undgå at diagrammernes størrelse skal eksplodere.

Stiplede linier og pile betyder, at denne handling er en special-case, der ikke nødvendigvis forekommer.

JXTA-systemets kommunikation (publiceringer og discovery-events) er vist uden modtager, da dette ikke er modelleret som en del af systemets protokoller idet JXTA-systemet selv håndterer dette.

Diagrammerne er lavet med henblik på at beskrive protokollerne og funktionaliteten, sådan at dette nogenlunde let kan implementeres. Der findes derfor en del noter på diagrammerne, der går mere i dybden, hvor det er væsentligt. Disse detaljer kan dog oftest udelades, hvis læseren blot ønsker at danne sig et overblik.

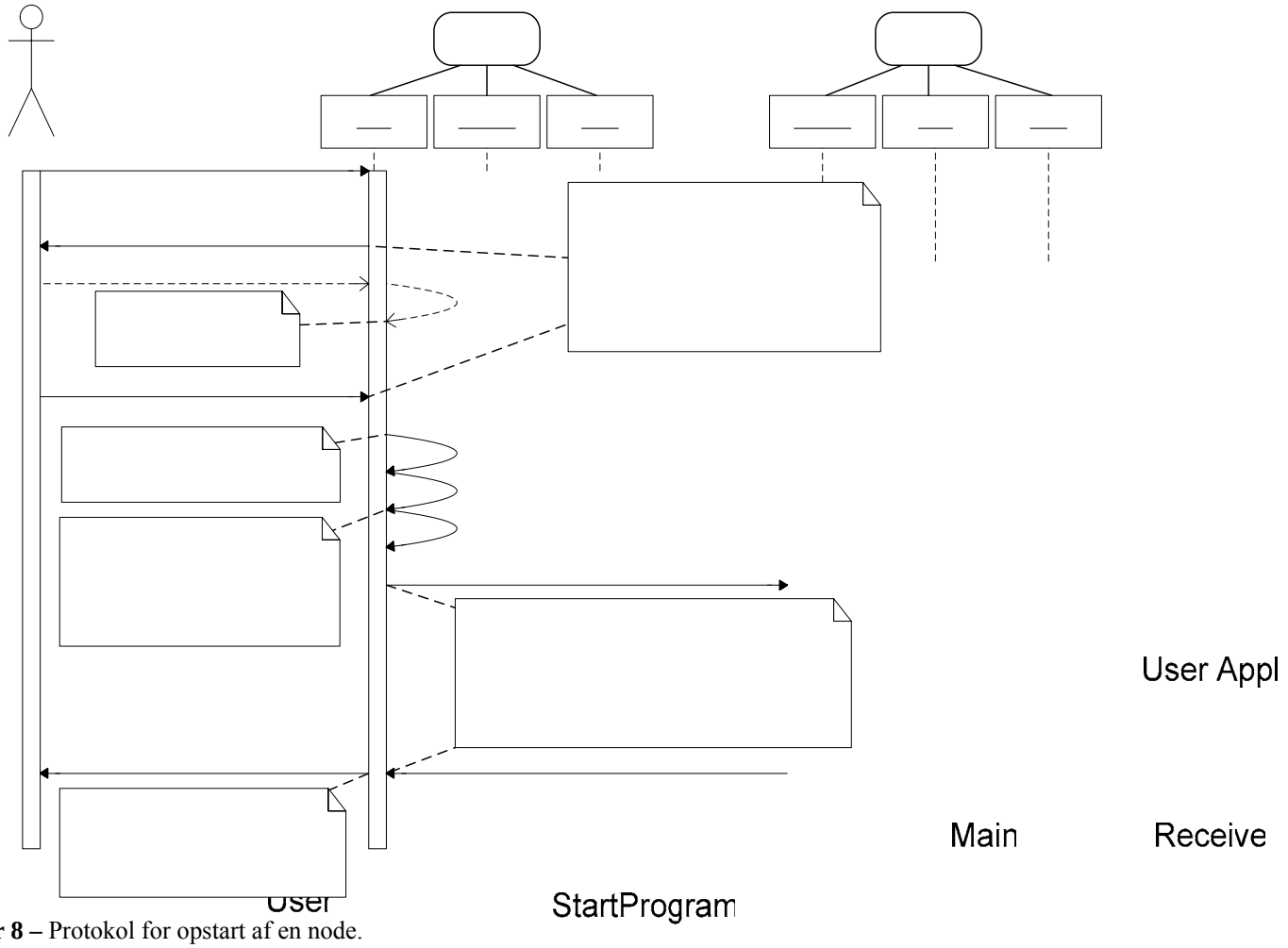
Sekvensdiagrammerne er lavet på engelsk for at lette overgangen til den senere implementering, som ligeledes er på engelsk.

4.5.2 Opstart og autentificering

Som nævnt i kravspecifikationen er oprettelse af certifikater ikke en del af systemets protokol. I stedet er det en procedure mellem de enkelte brugere og deres certifikatautoriteter, som skal finde sted, før systemet kan benyttes første gang. Hvis systemet ikke kan finde et korrekt certifikat og nøglepar fra brugeren vil applikationen kunne lave et nøglepar til denne, samt en *Certificate Signing Request* (CSR), som indeholder brugerens information i certifikatet. Ved at

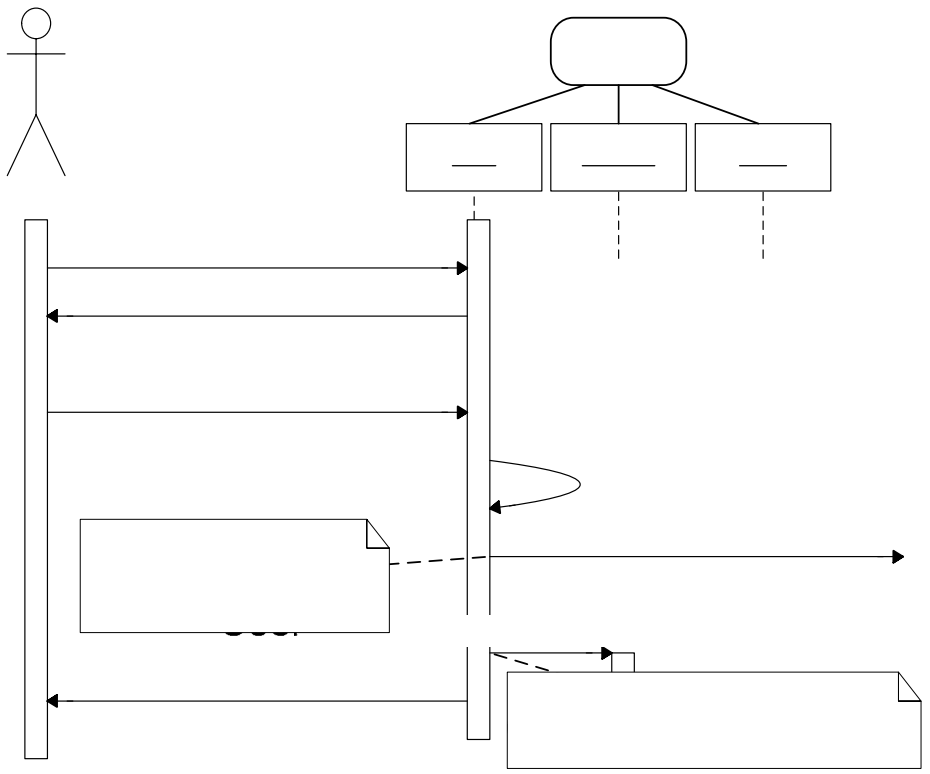
sende denne til sin certifikatautoritet og autentificere sig overfor denne, kan brugeren få udstedt et gyldigt certifikat.

Ved opstart checkes brugerens certifikat og private nøgle. Systemet skal have adgang til disse for at kunne oprette sikker kommunikation med andre noder. Herefter startes JXTA-systemet op. Dette leder efter backup-grupper, som brugeren kan slutte sig til. Protokollen for opstarten er vist herunder.



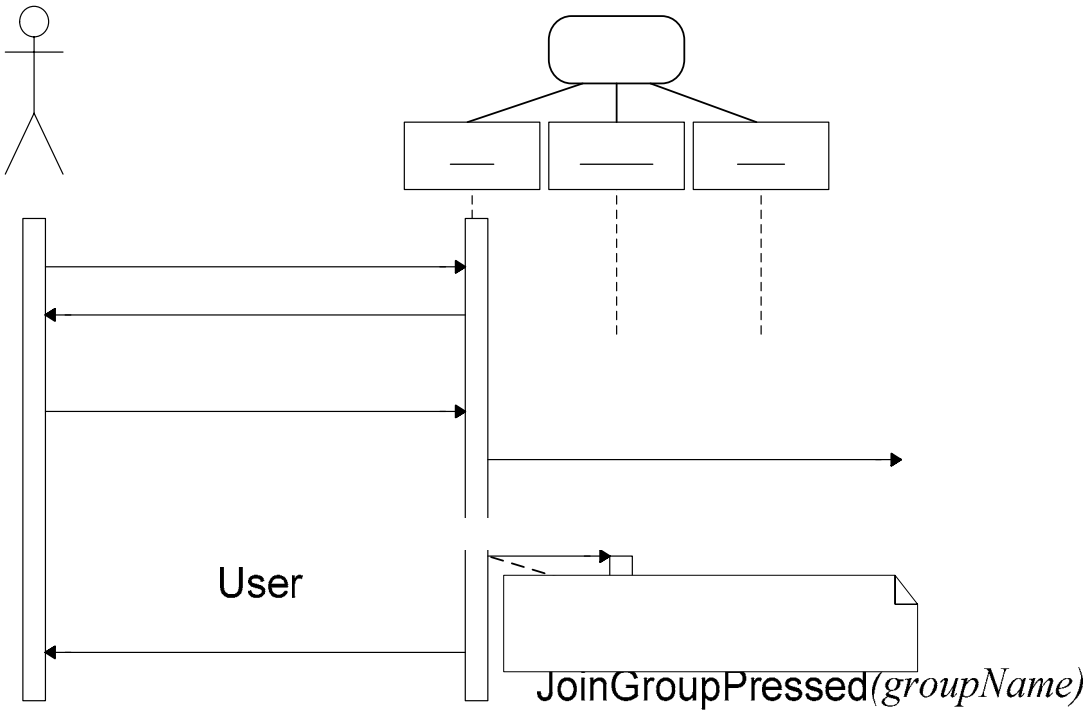
Figur 8 – Protokol for opstart af en node.

Efter denne opstart har brugeren to muligheder (jf. use case diagrammerne i afsnit 4.1.1): Enten kan brugeren tilslutte sig en eksisterende backup-gruppe eller også kan en ny dannes. Hvis brugeren vælger at oprette en ny gruppe skal gruppen publiceres i JXTA-systemet for at andre noder kan finde gruppen. Protokollen er vist på Figur 9 herunder.



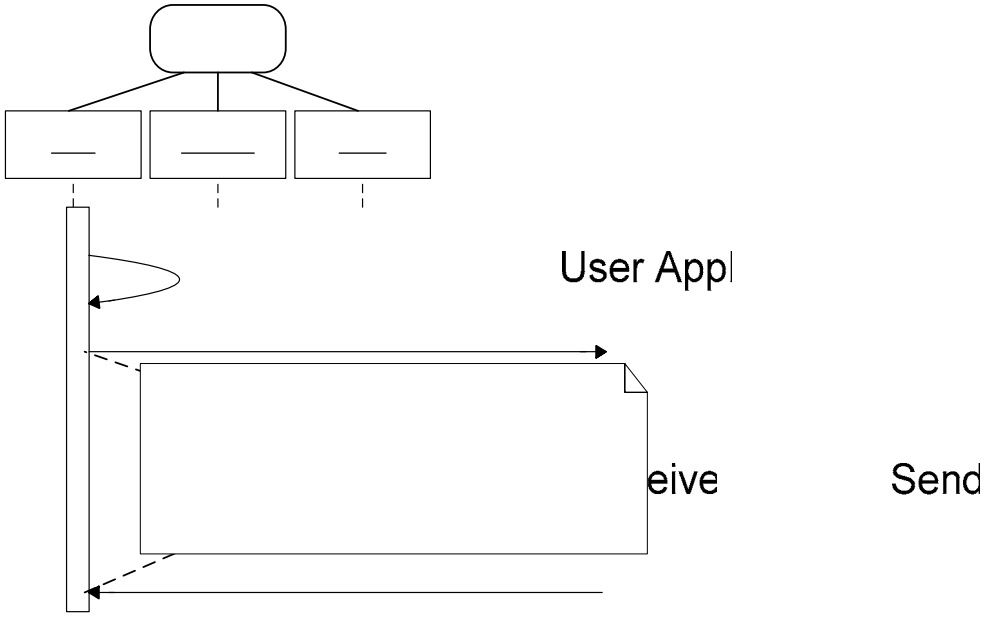
Figur 9 – Protokol for dannelse af ny backup-gruppe

Hvis brugeren i stedet ønsker at slutte sig til en eksisterende gruppe, ser protokollen ud som følger på Figur 10.



Figur 10 – Protokol for at tilslutte sig en eksisterende backup-gruppe.

Herefter er noden (og brugeren) medlem af en gruppe og programmets hovedvindue er tilgængeligt. Nu er det således ikke længere nødvendigt, at JXTA-systemet leder efter nye grupper, som noden kan tilslutte sig. I stedet skal JXTA-systemet finde andre noder (brugere), der er medlem af den samme backupgruppe, som noden selv. Dermed kan kommunikationen med gruppens øvrige noder etableres. Derfor udføres følgende, når noden (brugeren) er blevet medlem af en gruppe.



Figur 11 – Når systemet skal finde noder og ikke backup-grupper gennemføres dette.

Dermed vil JXTA-systemet lede efter andre noder med jævne mellemrum, så længe noden er en del af gruppen.

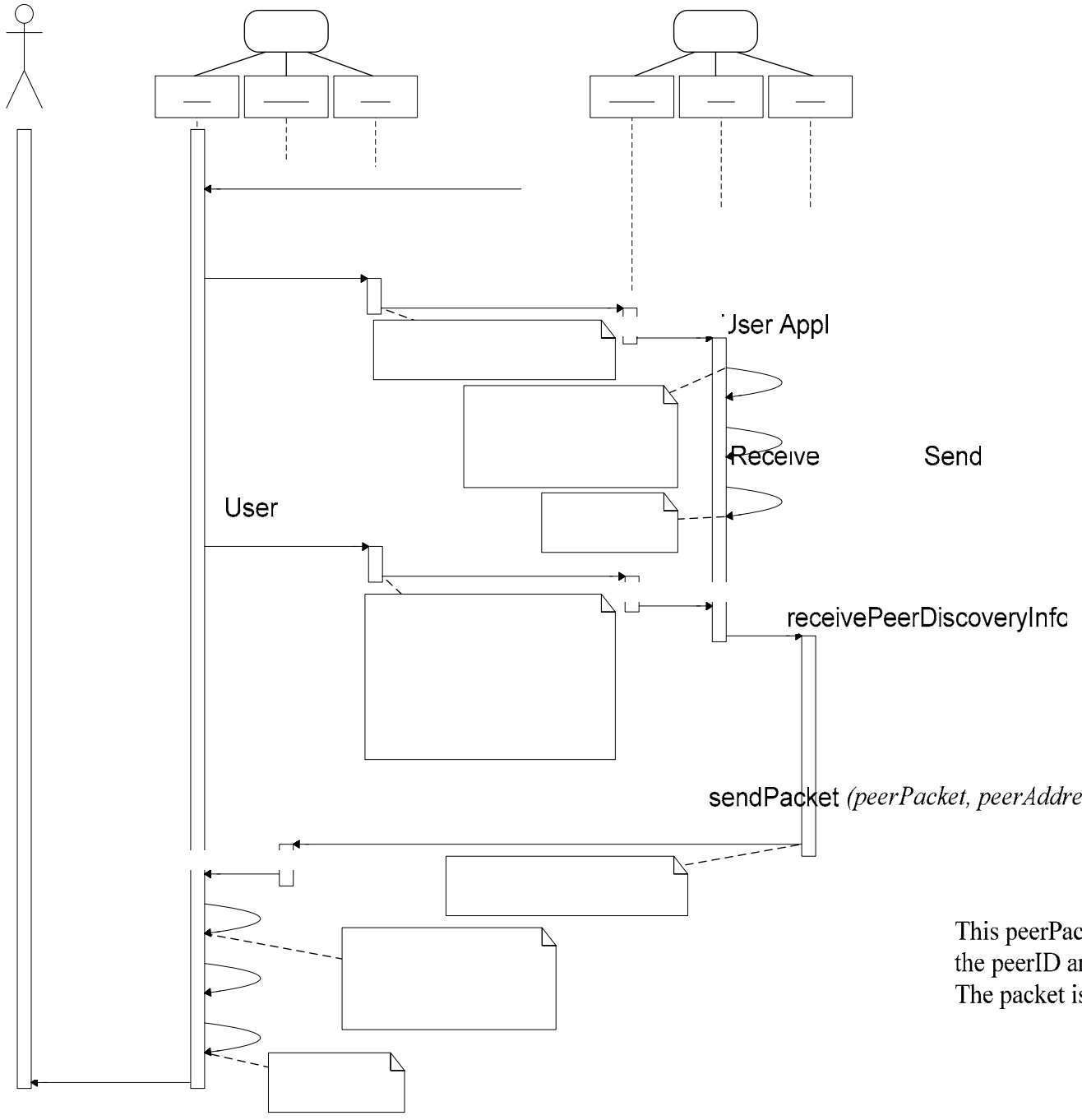
Når en node A finder en anden node B i backupgruppen vha. JXTA-systemet, vil den forsøge at autentificere denne. Derfor sender A en forespørgsel på information om B og B's certifikat. Desuden sender A sit eget certifikat, sådan at B kan autentificere A ved samme operation. Noden B vil herefter svare ved at sende den forespurgte information, som A så kan bruge til at gennemføre autentificeringen. Nodernes brugere afgør, hvilke certifikater man ønsker at stole på. Hvis en bruger allerede har valgt at stole på certifikatautoriteten, som har udstedt certifikatet, vil dette dog automatisk blive vurderet som troværdigt. Noderne gemmer desuden informationen om autentificerede noder, sådan at disse kan kontaktes igen uden (uden at systemet skal finde noden vha. JXTA discovery igen). Hver gang en ny node findes ser protokollen ud som følger.

Stop lookingForGroups

lookForPeers

This is handled by the JXTA system discovery protocol.
The send and received information will therefore not go through the send-thread and the receive-thread.
The communication of the JXTA system is not encrypted
Discovery requests are sent periodically in order to find peers and discovery responses are received in an asynchronous way.

receivePeerDiscoveryInfo



Figur 12 – Protokol for autentificering af noder.

Efter gennemførsel af denne protokol, vil de to noder have autentificeret hinanden, hvorved noderne kan kommunikere sikkert fremover vha. den i afsnit 4.4.1 beskrevne pakkestruktur.

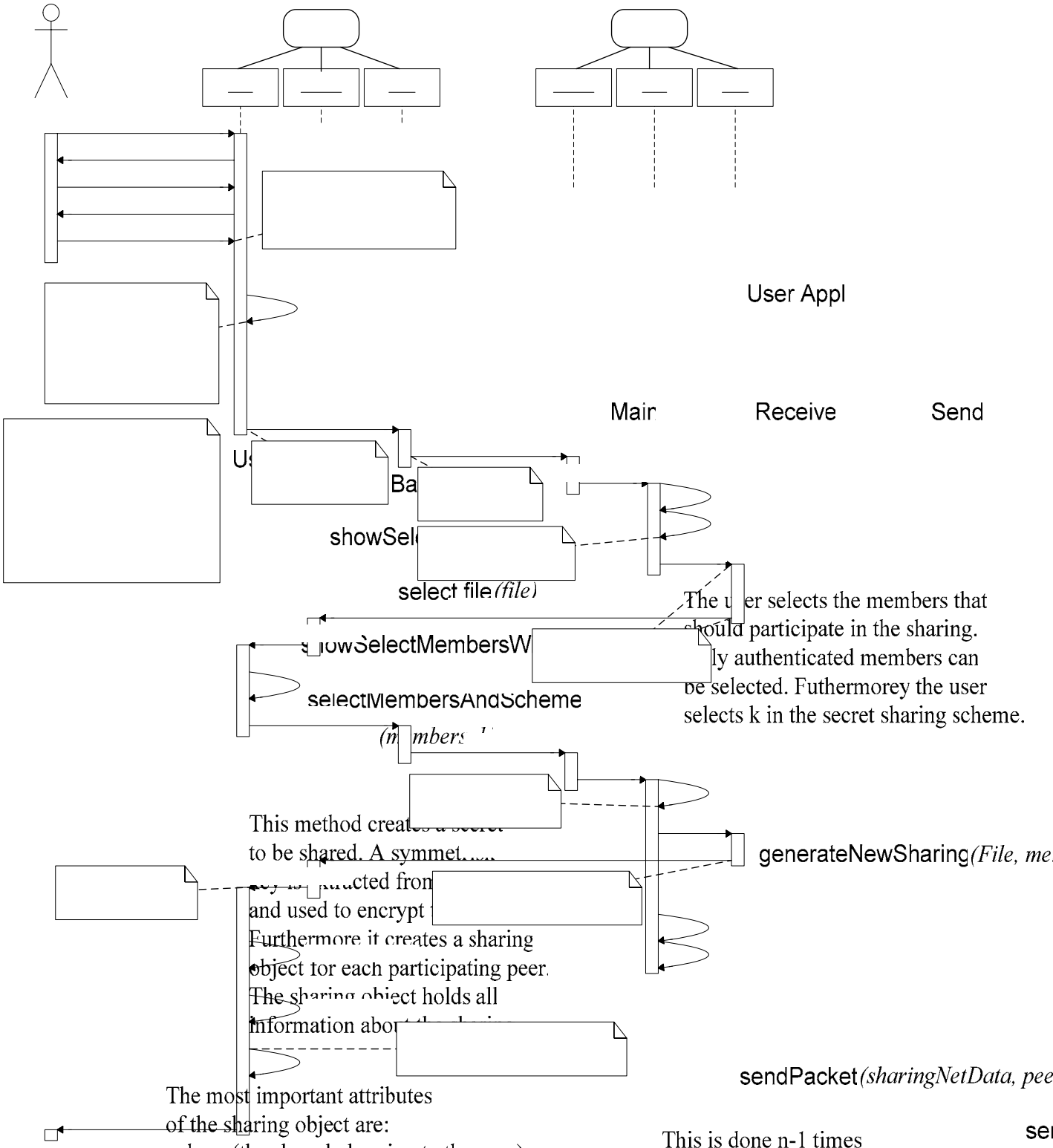
4.5.3

Tag backup af en fil

Når systemet er startet op og noderne har fundet og autentificeret hinanden, kan hver bruger vælge at lave en backup hos nogle af de andre. Når en bruger vil lave backup, starter han med at vælge den fil, som ønskes gemt på en sikker og pålidelig måde. Herefter vælges de autentificerede gruppemedlemmer, som skal deltage i den pågældende backup samt værdien *k*.

Metoden `generateNewSharing(File f, Vector members, int k)` står herefter for den centrale funktionalitet i genereringen af backup'en. Metoden sørger først for at danne en tilfældig hemmelighed (2048 bit), som skal deles mellem de deltagende noder. Dernæst udledes en AES-nøgle (128 bit) fra denne vha. en modulofunktion, og AES-nøglen benyttes til at kryptere filen f. Nu deles den tilfældige hemmelighed (værdien på 2048 bit). Dette gøres ved at noden opretter *n* sharing-objekter (et objekt til hver af de deltagende noder). Et sharing-objekt indeholder alt nødvendigt data om den pågældende backup (server-nummer, share-værdi, verificeringsinformation osv.), bortset fra filen selv. Selve share-værdierne laves vha. Shamirs secret sharing og verificeringsinformationen dannes vha. Feldmans VSS. Hver sharing indeholder desuden et unikt id, der adskiller denne fra andre sharings. Sharing-objektet er beskrevet mere detaljeret i afsnit 5.2.1.

Hver deltagende node modtager således et sharing-objekt fra filejeren. Herefter opretter noden en modtage-tråd, der kan tage imod selve den krypterede backup-fil og sender en besked til filejer-noden om, at sharing-objektet er modtaget. Filejer-noden sender nu selve backup-filen. Denne checkes af modtager-noden, der endnu engang svarer med en status. Hvis filejeren modtager tilstrækkelig mange succesfulde tilbagemeldinger (*k* eller derover), betragtes operationen som korrekt gennemført. Sekvensdiagrammet på Figur 13 herunder viser den beskrevne protokol.



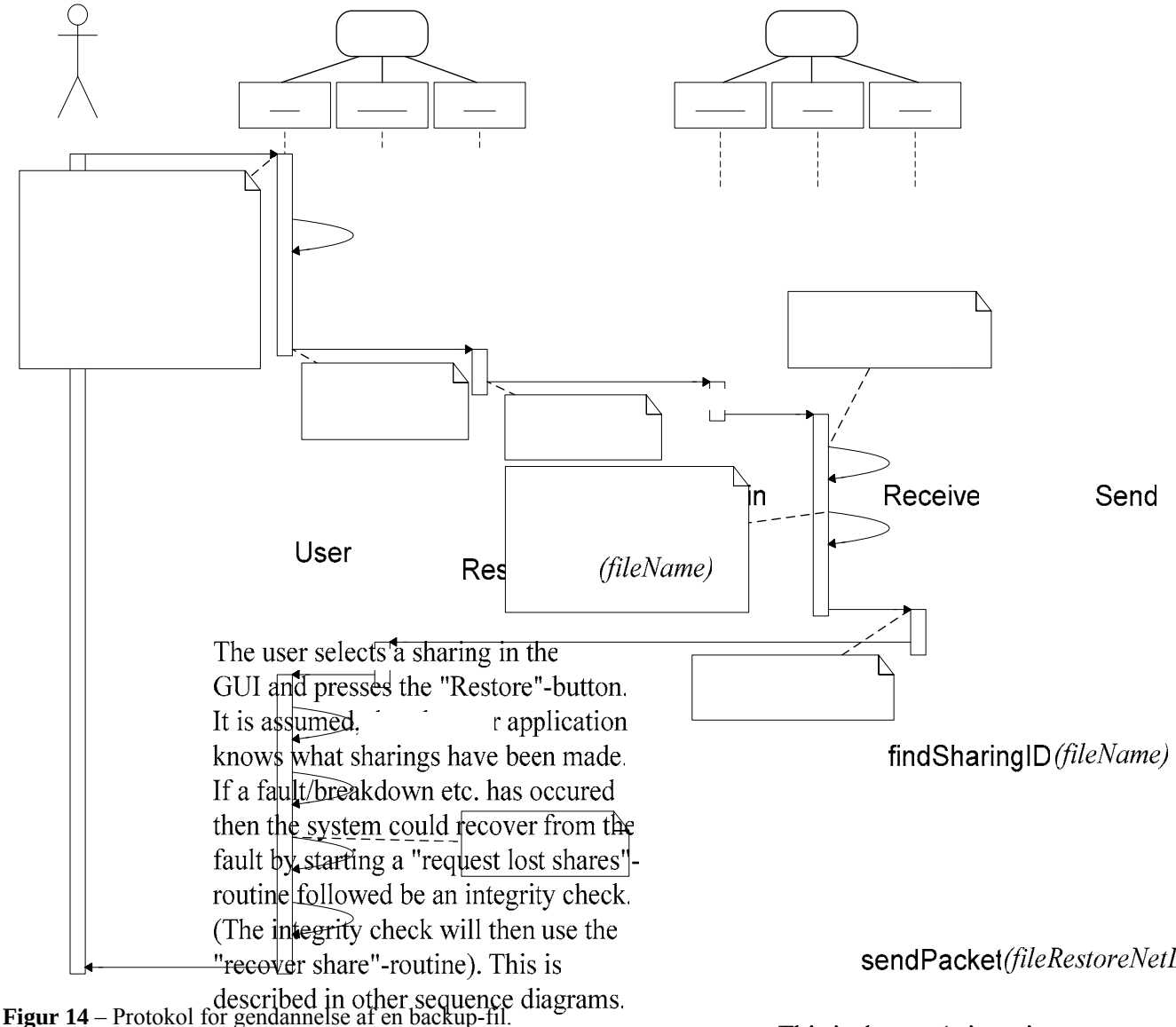
Figur 13 – Protokol for dannelse af en distribueret backup

4.5.4 Gendan en fil fra en backup

En backup kan kun gendannes af den oprindelige filejer. Denne vælger, hvilken fil der skal gendannes og sender en besked til alle noder, der deltager i backup'en af denne fil. Beskeden er en forespørgsel på den enkelte nodes share-værdi fra sharing-objektet, som skal bruges til at gennemføre gendannelsen.

En node, som modtager en forespørgsel på en sharing, skal først sikre sig, at forespørgslen kommer fra den rette ejer. Dette gøres ved først at finde det certifikat, der hører til den private nøgle, som blev brugt til at signere forespørgslen (hvis signaturen kan verificeres med den offentlige nøgle i certifikatet, er dette det rigtige certifikat). Derefter sammenlignes identifikationsinformationen i certifikatet med den identifikationsinformationen, der findes i sharing-objektet. Hvis disse matcher, kan noden regne med, at forespørgslen kom fra filejeren og sende sit sharing-objekt til denne.

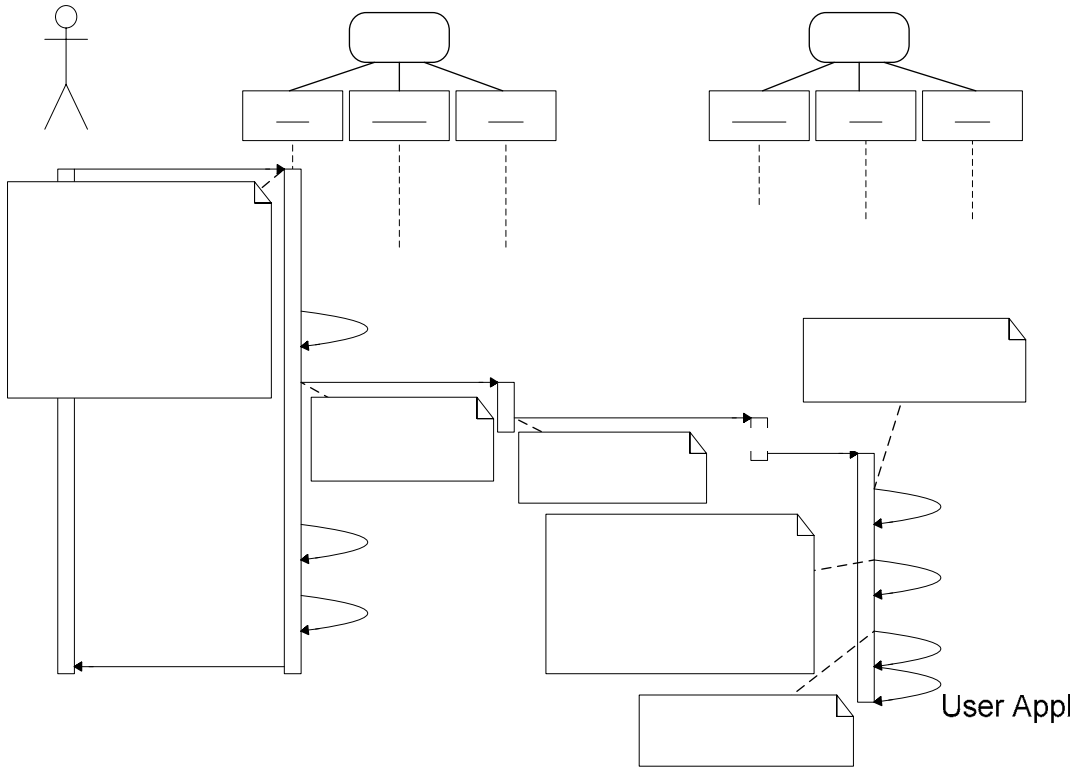
Filejeren checker nu de modtagne shares og rekonstruerer hemmeligheden, når *k* korrekte shares er modtaget. Protokollen kan ses på Figur 14.



Figur 14 – Protokol for gendannelse af en backup-fil.

4.5.5 Slet en backup

En backup skal kunne slettes af filejeren i hele systemet, hvis dette skulle være ønsket. Det skal bemærkes, at hvis n-k+1 shares slettes, kan de resterende shares ikke længere benyttes. Derfor er det ikke kritisk, at ALLE noder opfører sig korrekt ved udførslen af denne protokol. Operationen minder meget om gendannelsesprotokollen, idet noderne checker forespørgslen fra filejeren på samme måde. Protokollen kan ses herunder.



Figur 15 – Protokol for at slette en backup.

Main Receive Send

User

delete sharing *(fileName)*

The user selects a sharing in the GUI and presses the "Delete"-button. It is assumed, that the user application knows what sharings have been made. If a fault/breakdown etc. has occured then the system could recover from the fault by starting a "request lost shares"-routine followed be an integrity check. (The integrity check will then use the "recover share"-routine). This is described in other sequence diagrams.

findSharingID *(fileName)*

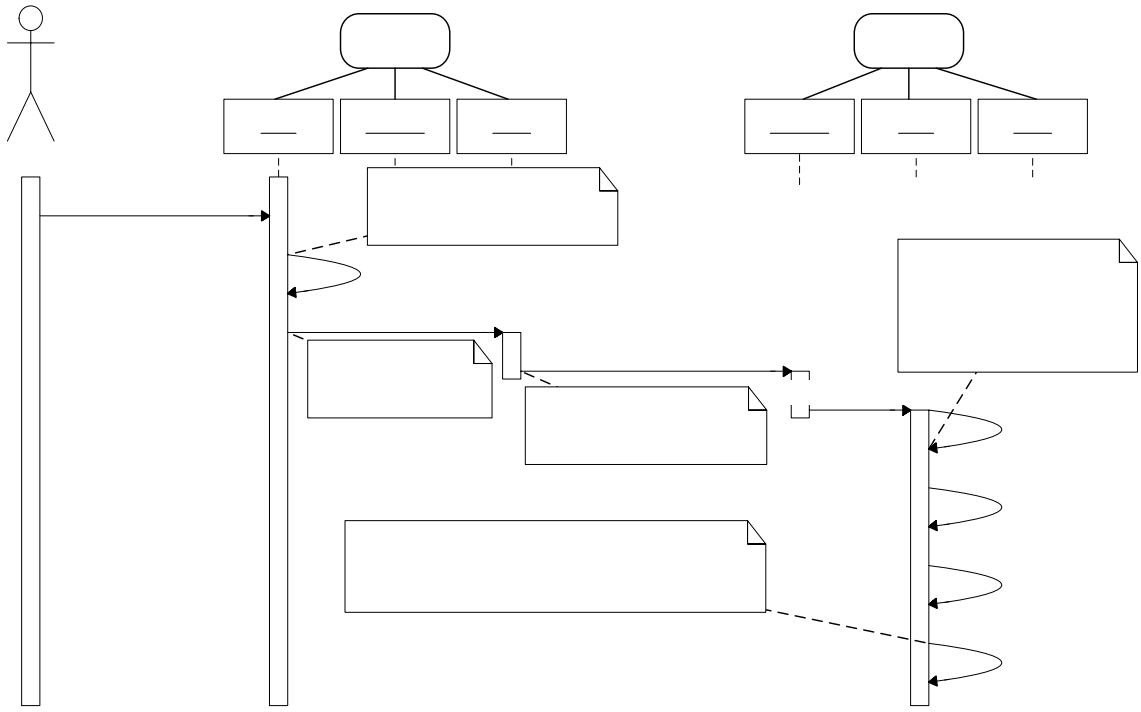
sendPacket *(deleteSharingNetData, peer*

This is done n-1 times in

send delet

4.5.6 Opdatering af shares

Når opdateringen skal sættes i gang, skal den enkelte node først sikre sig, at der ikke er andre operationer i gang, som forhindrer dette. Desuden skal der være gået et stykke tid siden sidste opdatering, sådan at denne ikke kan sættes i gang hele tiden. Hvis alt er i orden sendes en besked til de resterende noder, som herefter foretager det samme check. Derefter sender disse ligeledes en besked til alle i backupgruppen. Dette skyldes, at vi ikke nødvendigvis kan stole på den første node, som vil sætte operationen i gang. Denne kunne være kompromitteret og kunne forsøge, at få en del af systemet til at opdatere, uden at resten af noderne var med. Herved sikres således, at alle ved, at opdateringen er sat i gang. En node, som modtager en "start-opdatering"-pakke vil naturligvis ignorere denne, hvis en opdatering allerede er i gang. Protokollen for igangsættelsen af opdateringen er vist herunder.



Figur 16 – Protokol for start af opdateringsfase.

Herefter er alle noder enige om, at en opdatering er i gang. Disse fortsætter således til selve opdateringsoperationerne og det følgende foregår i alle deltagende noder.

Først laves et tilfældigt opdateringspolynomium $\Delta(x)$ af grad k-1 (ligesom den oprindelige konfiguration), hvor $\Delta(0) = 0$. Af dette laves et share til hver af de deltagende noder. Desuden laves et verificeringsobjekt, der indeholder verificeringsinformation om både opdateringspolynomiet og de dannede opdateringsshares. Verificeringsinformationen om opdateringspolynomiet består af g (den konstante værdi fra Feldmans VSS) opløftet i alle

polynomiets koefficienter. Verificeringsinformationen om de dannede opdateringsshares består af *g* opløftet i de forskellige shareværdier.

Verificeringsobjektet er offentligt tilgængeligt og sendes til alle noder sammen med selve nodens share. Herved kan alle noder verificere alle andres opdateringsshare. Dog kan en modstander stadig vælge at sende forskellige verificeringsobjekter til forskellige noder, da vi ikke er i besiddelse af en broadcast kanal. Derved snydes systemet, når de forskellige shares skal verificeres. Derfor er det vigtigt, at alle noder er enige om, hvem der har sendt hvilke verificeringsobjekter.

For at opnå dette signeres verificeringsobjektet separat, inden dette sendes til andre noder. Alle noder kan nu opsamle verificeringsobjekter og deres signaturer fra alle andre noder. Hver node har således en ide om, hvordan alle andre noders verificeringsobjekter ser ud. Denne information skal efterfølgende udveksles mellem noderne i systemet, for at sikre, at ingen har snydt og sendt forskellig information til forskellige noder. Derfor sender alle noder deres opsamlede information (bestående af en liste af verificeringsobjekter og tilhørende signaturer) til alle andre noder. Herefter vil hver node således være i besiddelse af en liste af verificeringsobjekter fra alle noder. Da optil *k*-1 noder kan være kompromitteret findes det korrekte verificeringsobjekt fra hver enkelt node ved at sammenligne de modtagne verificeringsobjekter fra denne node. Alle korrekte noder vil nå til samme konklusion, medmindre den pågældende node snyder, hvilket vil blive afsløret pga. de tilhørende signaturer. En node som opdager, at en anden node snyder, vil sende en beskyldning (*accusation*) mod noden til resten af systemet. Beskyldningen vil i dette tilfælde indeholde to forskellige verificeringsobjekter fra den samme node, samt de tilhørende signaturer[19]. Dermed kan de resterende noder se, at den beskyldte node snyder, antaget at begge signaturer er korrekte. Det skal dog bemærkes, at der kan eksistere lige så mange forskellige slags beskyldninger, som der er måder at afvige protokollen på. Pga. projektets tidsmæssige rammer vil beskyldninger ikke blive gennemarbejdet.

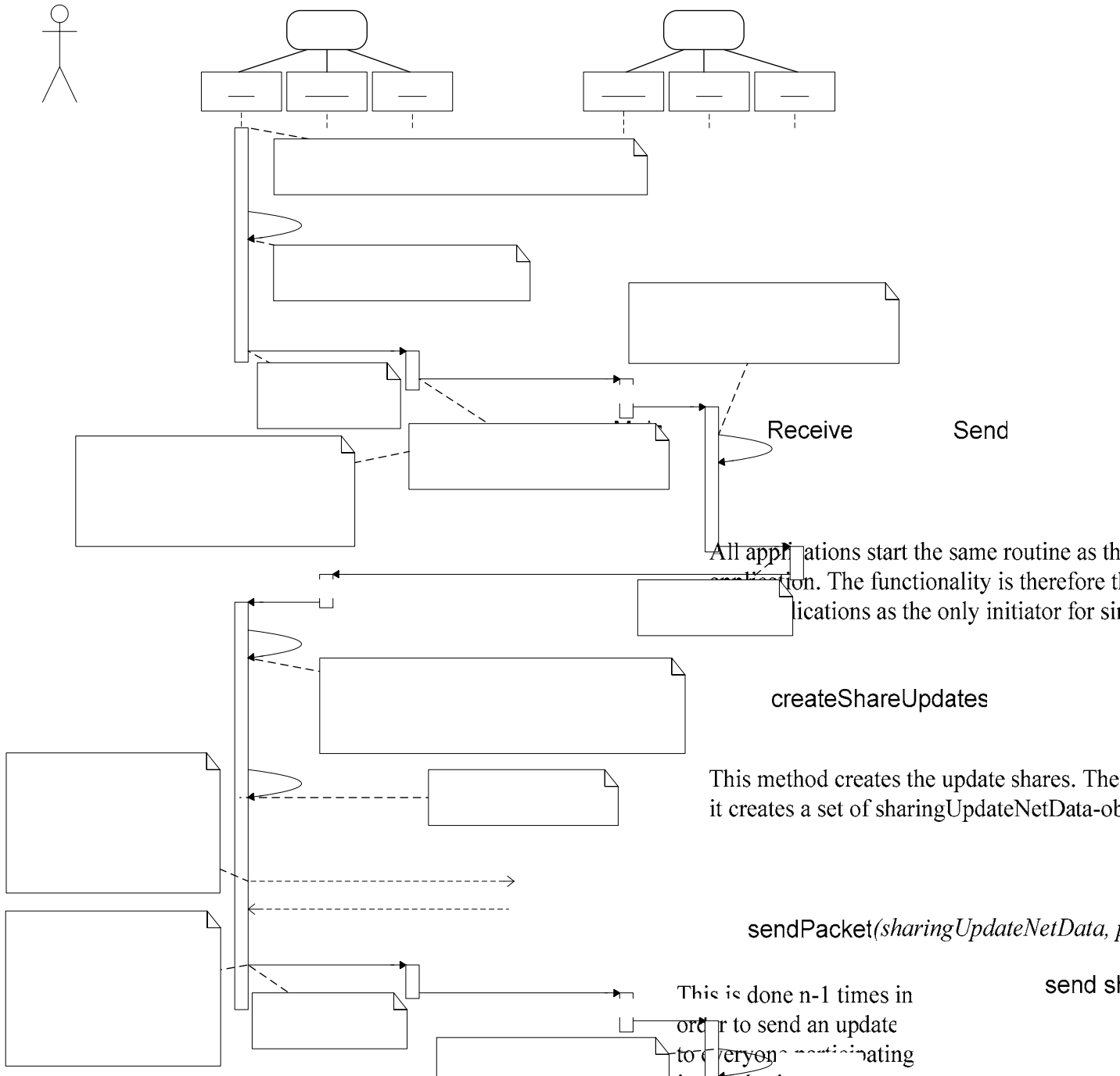
Nu har alle korrekte noder det samme verificeringsobjekt fra alle andre deltagende noder (medmindre en node slet ikke svarer). Verificeringsinformationen kan nu benyttes til at verificere de opdateringsshares, som blev modtaget sammen med de oprindelige verificeringsobjekter. Hvis mindst *k* shares verificeres korrekt, kan opdateringen fortsætte. Det er dog vigtigt, at alle andre korrekte noder ligeledes har tilstrækkelig information til at gennemføre opdateringen og derfor sendes et *updateValidationOkNetData*-objekt til ikke-kompromitterede noder, når en applikation er klar. Hvis en korrekt node ikke er klar til at starte opdateringen på dette tidspunkt, er det fordi denne først skal have afklaret en eller flere beskyldninger af andre noder. Efter dette er klaret vil den således sende *updateValidationOkNetData*-objekt til de andre noder. Når *updateValidationOkNetData*-objekter er modtaget fra alle korrekte noder (de noder som medvirker i opdateringen), gennemføres den lokale opdatering. Herved opdateres både selve share-værdien og den tilhørende verificeringsinformation.

Denne noget besværlige protokol er nødvendig, fordi der ikke er en broadcast kanal til rådighed, som det f.eks. er antaget i artiklen fra IBM [11]. Med en sådan (og med de antagelser, der gælder

[19] Verificeringsobjekterne skal i denne forbindelse indeholde et periodenummer, som alle noder er enige om, sådan at en modstander ikke blot kan bruge et gammelt verificeringsobjekt til at lave en falsk beskyldning mod en korrekt node.

om denne) kan man let sikre, at alle noder får den samme information, hvilket ville simplificere systemets protokol markant.

På Figur 17 herunder er den beskrevne protokol illustreret. Bemærk at alt beskrevet funktionalitet i diagrammet foregår i alle noder. F.eks. vil alle noder både eksekvere *createShareUpdate*, *collectVerifyValues* og *findCorrectVerifyValues*, selvom disse ikke er beskrevet under den samme node i diagrammet. Illustrationen er således simplificeret for at øge overskueligheden.



Figur 17 – Protokol for opdatering af shares

Herefter er alle korrekte noders share opdateret. De forskellige beskyldninger håndterer kompromitterede noder, som sender falske pakker. Det største problem for protokollen er dog, at modstanderen kan undlade at svare på udvalgte tidspunkter. Derved genereres nemlig ikke noget data, som kan benyttes som bevis for, at noden er kompromitteret. Desuden kan en simpel fejl ligeså godt være årsagen. Et manglende svar fra en node kan være ligeså destruktivt for protokollens funktionalitet som noget forkert data. Dette skyldes, at dele af systemet vil tro, at noden deltager korrekt i opdateringen, mens andre slet ikke har fået et opdateringsshare fra noden. Hvis opdateringen fortsætter, vil de to gruppers shares således ikke være kompatible.

Et muligt angreb ser derfor ud som følger: Hvis modstanderens node slet ikke sender *sharingUpdateNetdata*-objekter til korrekte noder i systemet, vil denne blot blive udeladt fra opdateringen. Hvis modstanderen derimod sender et *sharingUpdateNetdata*-objekt til blot en enkelt korrekt node, vil alle korrekte noder komme i besiddelse af det inkluderede verificeringsobjekt, men de vil stadig mangle selve opdateringssharet fra modstanderens node. De kan dog ikke bevise, at det er modstanderens node, der er kompromitteret, da de principielt ligeså godt selv kunne have ignoreret informationen for at forsøge at få gruppen til at vende sig mod en node, som ikke havde gjort noget galt.

Protokollen skal således håndtere, at de noder som mangler et share enten kan få dette, eller at afsendernoden erklæres for kompromitteret, sådan at denne ikke deltager i opdateringen. Dette kan gøres vha. en mere avanceret protokol for beskyldninger (*accusations*) end i de andre tilfælde, hvor der findes tydeligt bevis. Ved denne beskyldningsprotokol haves intet bevis, så hele systemet (de korrekte noder) skal således blive enige. Da dette er en speciel case, vil løsningen blot blive skitseret kort for at vise, at problemet kan løses på en rimelig måde.

En node, der mangler et share fra en anden node (afsendernoden), sender en forespørgsel til alle noder i gruppen. Alle er således opmærksomme på, at der muligvis er et problem. Forespørgslen videresendes til afsendernoden, som muligvis ikke har sent et share. Når denne har modtaget k forespørgsler fra k forskellige noder, sender denne det forespurgte share til alle noder i gruppen, hvis den følger protokollen. Disse vil herefter verificere sharet og videresende det til den rette node. Når den rette node modtager det manglende share, sendes *updateValidationOkNetData*-objektet og der vendes tilbage til den oprindelige opdateringsprotokol. Systemets har således blot videresendt data og ingen er erklæret for kompromitteret. Hvis afsendernoden imidlertid ikke følger protokollen og sender det forespurgte share, vil alle systemets noder få en timeout (sat i gang lokalt, da forespørgslen blev videresendt til afsendernoden). Dermed vil de sende en timeout-beskyldning til alle andre i systemet. Når k af disse er modtaget hos en node, markeres afsendernoden som kompromitteret. Herved sikres det, at der enten modtages et svar, eller at afsendernoden betragtes som kompromitteret.

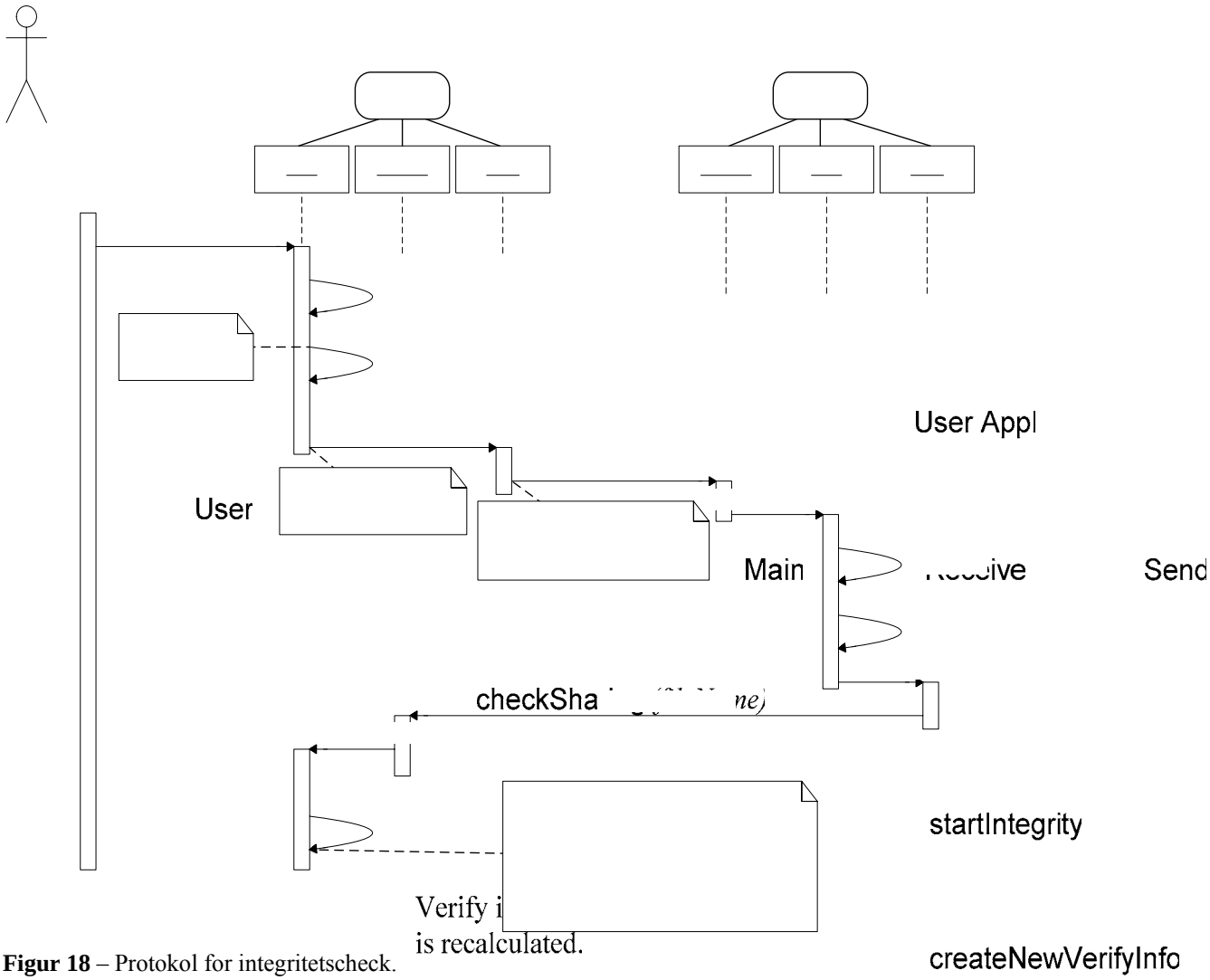
Problemet med denne løsning er, at alle lærer opdateringssharet til den node, som manglede sit share. Ud fra dette kan man dog ikke sige noget om nodens egentlige share, hvorfor informationen ikke er særlig brugbar medmindre samtlige opdateringsshares er til rådighed. Noden som fremprovokerede denne protokol (noden som ikke sendte sit share i første omgang) kendte informationen i forvejen og får på den måde ikke noget ekstrainformation. Derfor må løsningen betragtes som tilstrækkelig.

4.5.7 Integritetscheck

Som beskrevet skal systemet kunne checke integriteten af shares (dvs. sharing-objekter) samt af de krypterede backup-filer. Når et sharing-objekt skal checkes, sendes denne til de andre noder som et *integrityNetData*-objekt. Selve share-værdien i sharing-objektet er dog hemmelig og sendes naturligvis ikke med. Verificeringsinformationen om nodens eget share skal udregnes på ny. Herved vil det blive opdaget, hvis share-værdien er blevet ændret, idet verificeringsinformationen (g^{share}) ligeledes vil være ændret. Alle noder holder hinandens verificeringsinformation, sådan at alle kan se, om den modtagne nyudregnede information er korrekt eller ej. Da noder kan være kompromitterede kan vi dog ikke stole på en enkelt nodes udsagn. Derfor er hele systemet nødt til at teste integritet af en backup på samme tid. Noderne samler således *integrityNetData*-objekter fra alle andre noder. Ud fra disse kan noden afgøre, om dele af nodens sharing-objekt er modificeret. Dette sker ved at betragte majoriteten af pakkerne, da majoriteten altid vil have korrekt information i følge systemets grundantagelser. Hvis selve share-værdien viser sig at være modificeret, skal gendannelsesoperationen sættes i gang. Dette er beskrevet i det efterfølgende afsnit. Hvis blot mindre dele af sharing-objektet er ukorrekt, kan dette udskiftes med informationen fra majoriteten af *integrityNetData*-objekterne.

Sharing-objektet indeholder bl.a. en checksum af den krypterede backup-fil. Dermed kan kontrol af backup-filens integritet klares i samme omgang. Dette gøres ved at udregne en ny checksum for filen, når sharing-objektets integritet er sikret. Hvis den nye checksum ikke matcher checksummen i sharing-objektet, beder noden en af de andre noder om at sende filen.

Figur 18 herunder viser protokollen for integritetscheck. Protokollen for at genskabe et korrupt share er beskrevet i det efterfølgende afsnit. I tilfælde af at filen er korrupt, hentes denne fra en anden node. Dette er trivielt og er således ikke illustreret i diagrammet.



Figur 18 – Protokol for integritetscheck.

4.5.8 Gendannelse af forsvundet eller modificeret share

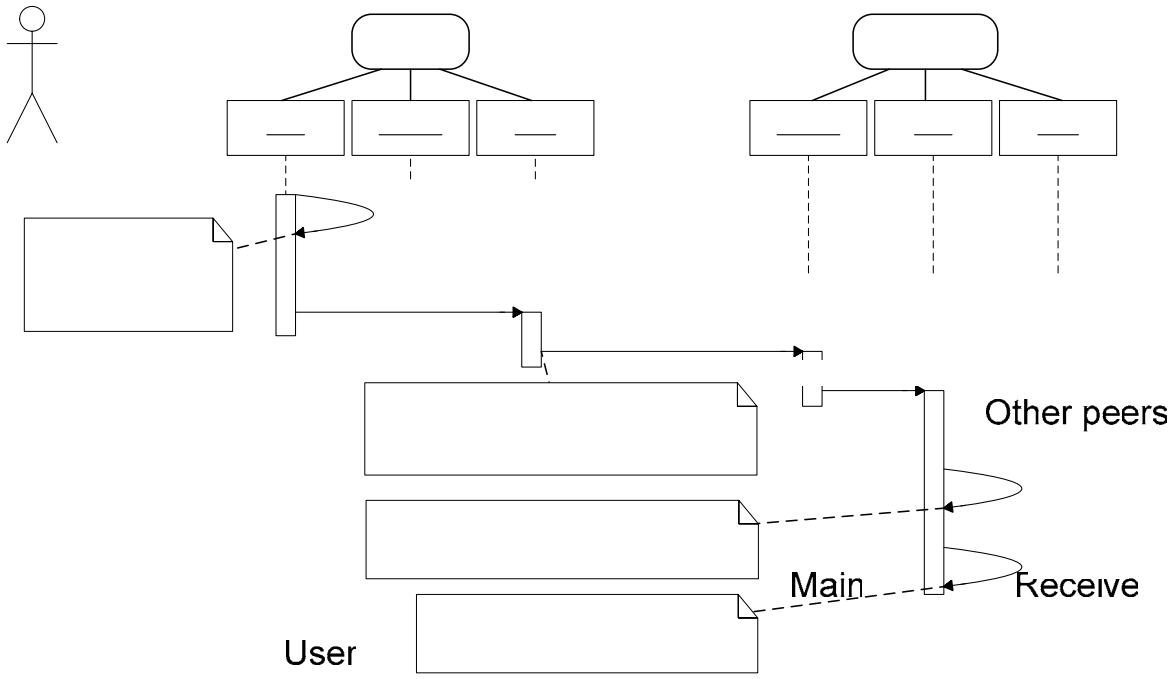
Protokollen for gendannelse af et forsvundet share minder om protokollen for share-opdatering. Den har til formål at genskabe share-værdien hos en node, som på den ene eller den anden måde har mistet denne. Den mest grundlæggende forskel er, at ved en share-opdatering er alle noder lige og udfører det samme arbejde. Ved en gendannelse af et forsvundet share, er noden, som skal have gendannet sit share, speciel. Denne er den eneste, der skal have ændret noget ved operationen, og den eneste, der vinder noget (nemlig sit korrekte share), ved at operationen gennemføres korrekt. Derfor kan man til en hvis grad stole på denne node under en gendannelsesoperation. Det er f.eks. usandsynligt, at noden vil forsøge at ødelægge operationen, kun for at denne skal fejle. Man kan dog godt forestille sig at en modstander vil forsøge at udnytte gendannelsesprotokollen til at bryde vigtigere dele af systemet, hvorfor vi naturligvis ikke kan stole fuldt på en node, der vil gendanne sit share.

Denne grundlæggende forskel viser sig allerede i opstarten af protokollen. I modsætning til share-opdatering behøver vi ikke sikre, at alle har fået besked om operationen ved at alle noder sender pakker til alle, da dette styres af noden med det manglende share. Hvis denne ikke sørger for at sætte operationen ordentligt i gang, er det kun den selv, det går ud over. Det er altså nok med en enkelt pakke fra den gendannende node til alle andre noder, som deltager i backup'en. Dette svarer til den første del af Figur 16 og er således udeladt.

Når først protokollen er sat i gang, er den første del af gendannelsesoperationen næsten lig opdateringsprotokollen (se Figur 17). Den eneste væsentlige forskel er, at opdateringspolynomiet $\Delta(x)$, hvor $\Delta(0) = 0$, som alle noderne laver, er erstattet af et gendannelsespolynomium, hvor $\Delta(r) = 0$ (r er nummeret på den node, der skal gendanne sit share). Ellers fortsætter protokollen med at lave shares og verificeringsobjekter og distribuere disse på samme måde. Indtil det tidspunkt, hvor alle noder er klar til at opdatere deres shares er protokollerne således ens, hvorfor denne del ikke illustreres igen. I stedet henvises til afsnit 4.5.6.

Når alle noder er klar til at opdatere deres shares, adskiller protokollerne sig imidlertid fra hinanden. I stedet for at opdatere sit eget share, laver hver node nu et midlertidigt gendannelsesshare[20], som sendes til den node, som vil gendanne sit share. Sharet sendes i et sharing-objekt, sammen med resten af informationen om sharing-objektet, sådan at noden kan gendanne alt relevant information. Når gendannelsesnoden modtager disse, verificeres share-informationen vha. de distribuerede verificeringsobjekter. Herefter udregnes share-værdien ved at interpolere de modtagne værdier. Den sidste del af gendannelsesprotokollen, som ikke overlapper med opdateringsprotokollen, er vist herunder.

[20] Et gendannelseshare svarer til et opdateringsshare, bortset fra at den beskyttede hemmelighed ikke er vedligeholdt. Det er derimod det share, som skal gendannes.



Figur 19 – Sidste del af gendannelsesprotokollen.

Hvis en node helt har mistet sin sharinginformation, kan denne ikke medvirke i vedligeholdelsesprotokollerne og kan således heller ikke gendanne sit share vha. den ovenstående protokol eller deltage i et integritetscheck. Dette skyldes, at den grundlæggende viden, som findes i sharing-objektet, er nødvendig for denne (heriblandt viden om medvirkende noder, sharingID osv.). Denne vigtige viden findes dog andre steder i systemet. Derfor kan noden vælge at sende et *requestSharingDataNetData*-objekt til alle noder i gruppen. Disse vil herefter undersøge, hvilke backup'er den pågældende node deltager i og sende den grundlæggende information tilbage. Denne information vil ikke indeholde shareværdier, men er tilstrækkelig til at gennemføre en integritetscheckprotokol og en gendannelsesprotokol, hvorved det korrekte share gendannes sammen med den krypterede backupfil.

4.6 HÅNDTERING AF PARALLELLE BEGIVENHEDER

Ved at Control-tråden hele tiden holder styr på applikationens data og dermed kritiske sektioner, vil fejl pga. parallelle begivenheder kunne undgås internt i applikationen. Control-tråden modtager en række jobs fra andre tråde, men disse klares et ad gangen og vil således ikke give hinanden problemer.

Mange af systemets operationer består imidlertid af flere jobs. Mens en enkelt operation eksekverer, kan Control-tråden således modtage jobs fra andre operationer, som vedrører de samme kritiske sektioner, som den igangværende operation. Applikationen er således nødt til at holde styr på igangværende operationer, sådan at andre operationer nægtes adgang.

Alle systemets operationer involverer et sharing-objekt. Så længe en ny operation involverer et andet sharing-objekt end den igangværende operation, er der således ingen fare. Hvis det samme sharing-objekt er involveret, må applikationen dog sørge for, at den nye operation ikke får lov til at eksekvere. De operationer, som er kritiske for et sharing-objekt, er vedligeholdelsesoperationerne:

- Opdatering af shares.
- Integritetscheck af shares.
- Gendannelse af forsvundet share.

Disse operationer må således ikke eksekvere på det samme sharing-objekt på samme tid. Dette skyldes, at operationerne kan ændre på sharing-objektet. Opdatering af shares kan dog startes flere steder uafhængigt af hinanden, da operationen er ens på alle noder. Det er således ikke kritisk, hvem der starter operationen eller om den startes af flere på en gang. Det samme gælder integritetscheck. Det er dog et særligt tilfælde, som skal håndteres, når en operation som allerede eksekverer, bliver startet af en node igen. Gendannelse af forsvundne shares er mere kritisk, da denne operation ikke er ens på alle noder (noden som skal gendanne sit share er speciel). Operationen kan således blive startet på forskellige noder på samme tid, uden at betyde det samme, da det er forskellige noder, der vil gendanne deres share. Herved risikerer kritiske sektioner at blive berørt, når den enkelte node skal lave flere forskellige gendannelses-shares til forskellige noder. Sharing-objektet skal derfor indeholde en række statusvariable, som sikrer, at ulovlige parallelle begivenheder ikke kan eksekvere.

Principielt vil man heller ikke kunne garantere, at backup'ens data kan gendannes, mens der opdateres shares, da filejeren kan risikere, at modtage nye shares fra den ene halvdel af noderne og gamle shares fra resten, hvorved operationen fejler. Det kritiske tidsrum er dog ganske lille og filejeren kan blot forsøge igen et øjeblik efter, hvor operationen vil lykkes. Derfor behandles dette ikke yderligere.

Grundfunktionaliteten (opret backup, gendan backup og slet backup) kan kun foretages af filejeren og ændrer ikke sharing-objektet, medmindre dette oprettes eller slettes helt, hvorved de kritiske vedligeholdelsesoperationer alligevel ikke spiller nogen rolle.

4.7 OPSUMMERING

Dette kapitel har beskrevet de vigtigste designbeslutninger, der er foretaget for at leve op til de krav, som er stillet til systemet. Dette inkluderer overvejelser af den grundlæggende kryptering, hvortil der bruges 128 bit AES-nøgler samt 2048 bit RSA-nøgler. Størrelsen for hemmeligheder i det diskrete logaritmeproblem skal ligeledes være 2048 bit. SHA-1 er valgt som hashalgoritme for at matche sikkerheden i resten af systemet. Desuden er en sikkerhedsstruktur for kommunikationspakker beskrevet.

Systemets brug fra brugerens side er skitseret vha. en række use-case beskrivelser. Disse støtter implementeringen og det resterende design ved at give overblik over, hvordan brugerens oplevelse skal være.

Systemets kommunikationsprotokoller er helt centrale for systemets funktionalitet og sikkerhed. Disse er forklaret og illustreret vha. sekvensdiagrammer. Desuden er programmets hovedstruktur og vigtigste tråde beskrevet, sådan at en implementering er let at påbegynde.

Implementeringen vil være med JXTA som peer-to-peer platform, da dette vil kunne spare meget grundlæggende arbejde.

5 IMPLEMENTERING

Dette kapitel omhandler implementeringen af en prototype af det foreslåede sikre og pålidelige backupsystem. Prototypen har til formål at vise, at det kan lade sig gøre at implementere det beskrevne design, sådan at de opstillede krav til systemet sikkerhed og funktionalitet overholdes. Prototypen skal således ikke ses som et færdigt produkt, der kan markedsføres uden videreudvikling. Prototypen vil dog kunne udføre den grundlæggende funktionalitet, der er beskrevet, men vil ikke indeholde alle tænkelige special-cases. Dette skyldes den begrænsede tid, der er sat af til projektet. Implementering af den grafiske brugergrænseflade er bl.a. kraftigt nedprioriteret på samme baggrund, da denne hverken er væsentlig for systemets funktionalitet eller sikkerhed.

Java er valgt som programmeringssprog, da dette tilbyder den ønskede platformuafhængighed. Desuden benyttes JXTA til at danne den grundlæggende peer-to-peer arkitektur.

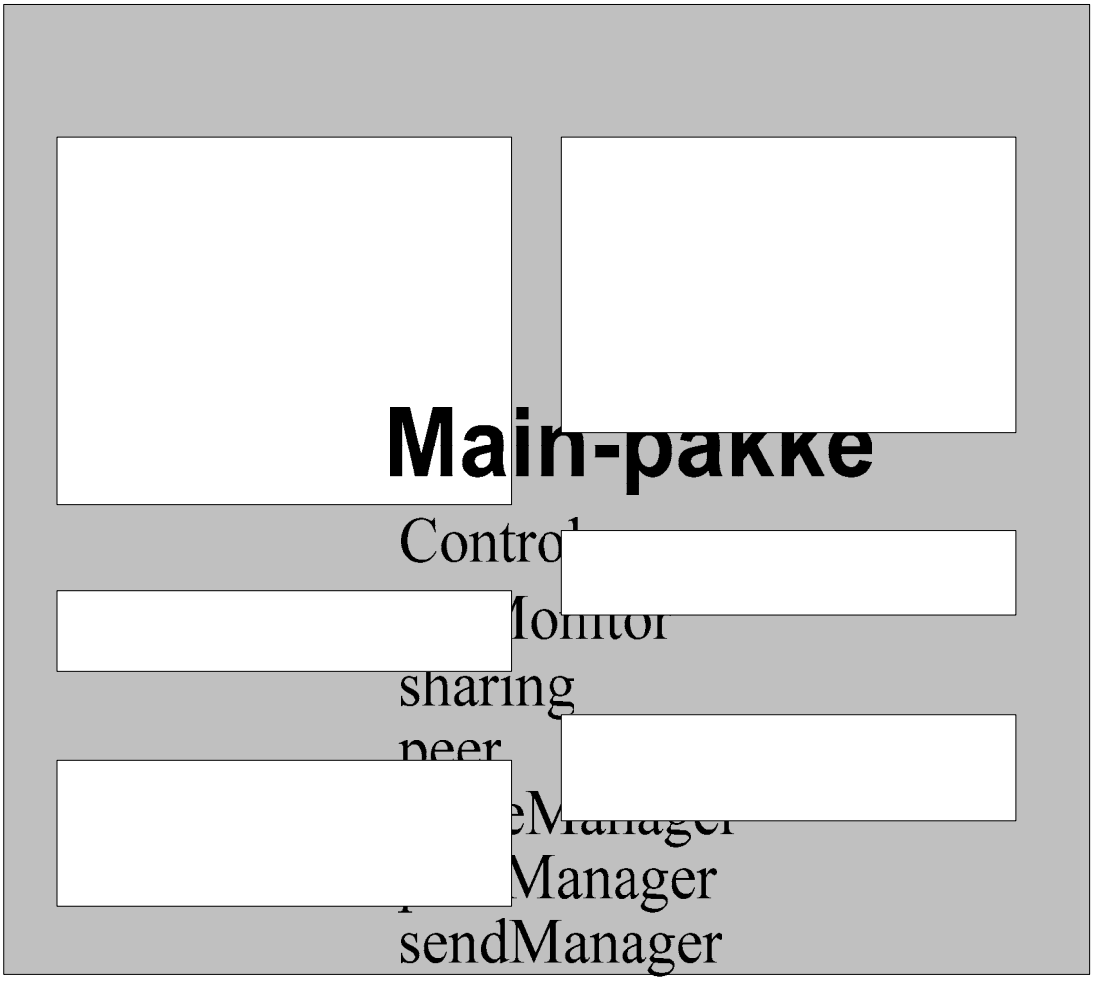
Backupsystemet har fået navnet Resilia[21]. Installation og brug af Resilia er beskrevet i Appendiks A.

5.1 OVERSIGT OVER APPLIKATIONEN

Dette afsnit giver et overblik over de vigtigste klasser i programmet. Klasserne er fordelt i 6 Java-pakker: Main, Objects, Send, Receive, JxtaSys og GUI. Pakkerne er som udgangspunkt dannet på baggrund af systemets tråde, jf. afsnit 4.2. Alle tråde har dog ikke fået deres egen pakke. Begge sende-tråde er således for overskuelighedens skyld placeret i samme pakke, ligesom modtager-trådene har fået deres egen pakke. Monitor-trådene har heller ikke fået deres egen pakke, da disse logisk placeres sammen med den tråd, som de agerer jobkø for. Objects-pakken indeholder kun data-klasser uden funktionalitet og har således slet ikke sin egen tråd. Dennes opgave er således at agerer databeholder for de andre. Denne opdeling er lavet for at øge overblikket over de mange klasser. Opdelingen giver desuden mulighed for at indkapsle trådenes klasser, sådan at metoder og variable til en hvis grad beskyttes fra uønsket tilgang.

På Figur 20 herunder ses programmets 6 pakker, samt hvilke klasser disse indeholder. Gennem resten af kapitlet, vil pakkernes og de enkelte klassers funktionalitet blive beskrevet yderligere. I oversigten er en række mindre vigtige klasser udeladt for at øge overblikket. Dette er bl.a. en række data-klasser og GUI-klasser.

[21] Resilia er afledt fra det engelske ord resilience, som bl.a. betyder robusthed og evne til at rette sig op, hvilket netop er en del af backupsystemets hovedformål. Der tænkes her især på systemets proaktive adfærd og distribuerede struktur.



Figur 20 – Oversigt over Resilias 6 pakker.

5.2 GRUNDLÆGGENDE DATASTRUKTURER

Programmet skal som udgangspunkt holde styr på to slags data. Den første slags er data, som omhandler en backup. Dette data modelleres vha. et sharing-objekt i Main-pakken (som første gang blev beskrevet under systemets design). Den anden slags data er information om de andre noder i systemet. Dette gemmes i et peer-objekt, ligeledes i Main-pakken. Udover disse datastrukturer, vil dette afsnit også beskrive dataklasserne fra Objects-pakken.

5.2.1 Sharing og shareManager

Et sharing-objekt repræsenterer en del af en backup hos en enkelt node. Hver gang en backup foretages, får hver deltagende node et sharing-objekt, som hører til denne backup. Objektet indeholder således de nødvendige variable for at systemets funktionalitet kan gennemføres, da

selve share-værdien fra Shamir's secret sharing langt fra er tilstrækkelig. Desuden indeholder objektet metoder til at udføre forskellige operationer på en backup.

Sharing-objektets primære variable er `share` og `serverNr`, hvor `share` er værdien fra Shamirs secret sharing. Da et share er på 2048 bit, jf. afsnit 4.4.3, repræsenteres værdien vha. en `java.math.BigInteger`, da denne klasse giver mulighed for at arbejde med meget store tal. `serverNr` angiver, hvilket servernummer det pågældende share har i denne sharing. Disse værdier vil således være forskellige i forskellige sharing-objekter, som alle er en del af den samme backup (på forskellige noder). Hvert sharing-objekt har ligeledes en række statusvariable og midlertidige datavariable, som ligeledes kan være forskellige på tværs af systemet (i den samme backup).

Der findes dog også en del data i et sharing-objekt, der er det samme i alle sharing-objekter, som repræsenterer den samme backup. Disse data er placeret i et sharingData-objekt, der er inkluderet i sharing-objektet. sharingData-objektet indeholder følgende variable:

```
BigInteger p,q,g,sharingID;
int k;
String ownerID,fileName,fileCheckSum;
long interval;
BigInteger nrOfUpdates,nrOfRecoveries;
Vector sharingPeers; // n = sharingPeers.size();
```

Værdierne p,q og g angiver værdierne fra Feldmans VSS. Værdien k angiver, hvor mange share-værdier, der er påkrævet for at gendanne den hemmelige krypteringsnøgle, sådan at backup-data kan genskabes. `sharingID` er hver sharings unikke id, som bruges til at adskille denne backup fra andre. Strengen `ownerID` angiver filejeren, som har ret til at genskabe data. `fileName` og `fileCheckSum` angiver backup-filens navn og checksum. `interval` angiver, hvor ofte et automatisk periodeskift skal foregå. `nrOfUpdates` og `nrOfRecoveries` angiver tidsperioden, som backup'en befinder sig i.

Vektoren sharingPeers indeholder en liste af sharingPeer-objekter, som repræsenterer samtlige medvirkende brugere. Et sharingPeer-objekt indeholder information om en enkelt medvirkende bruger, dvs. dennes serverNr, brugerens verificeringsværdi (repræsenteret ved en `java.math.BigInteger`) som bruges til at verificere dennes share samt brugerens identifikationsinformation, som stammer fra brugerens certifikat.

Ved at alle disse variable findes i det samme sharingData-objekt, er det let at sammenligne forskellige sharingData-objekter, hvilket er væsentligt, da disse som nævnt altid skal være ens i hele systemet.

Metoderne i sharing-objektet indeholder funktionalitet til at oprette opdateringsshares eller gendannelsesshares ud fra den pågældende sharing og verificere disse, når de modtages fra andre noder. Desuden findes metoder til at modificere en sharing som følge af vedligeholdelsesoperationerne samt en masse hjælpemetoder til at manipulere og returnere forskellige variable. Mere detaljeret information om sharing-objektet kan findes ved at betragte sharing.java i kildekoden i Appendiks B.

Hver applikation indeholder en liste af sharing-objekter, svarende til de backup'er, som applikationen medvirker i. Listen af sharing-objekter findes i shareManager-klassen. Denne klasse har til formål at styre denne liste og indeholder desuden metoder til at oprette en backup og gendanne en hemmelighed ud fra en række sharing-objekter. Klassen `shareManager` styrer således de operationer, der involverer flere sharing-objekter. Listen af sharing-objekter gemmes på harddisken i filen `sharing.ssb` i biblioteket `mySharings`. Dette gøres vha. filHandler-klassen hver gang ændringer foretages (opdateringer eller tilføjelser af nye sharing-objekter). Herved øges systemets robusthed, da en node ikke mister alle sine data, hvis den går ned.

Integriteten af alle delene i et sharing-objekt sikres af protokollerne for integritetscheck (se afsnit 4.5.7) og share-gendannelse (se afsnit 4.5.8).

5.2.2 Peer og peerManager

Et peer-objekt indeholder information om en anden node og dennes bruger. Informationen benyttes til at sende data til noden og til at afgøre, om noden opfører sig korrekt.

Først og fremmest indeholder peer-objektet brugerens navn, som benyttes til den grafiske repræsentation. Desuden haves brugerens certifikat-kæde, hvorfra brugerens offentlige nøgle og identifikationsinformation kan læses. For at kunne sende data til noden vha. JXTA-systemet indeholder peer-objektet desuden en `PeerAdvertisement` og en `PipeAdvertisement`. Disse er gemt fra den første kontakt med noden, sådan at JXTA's søgefunktioner efterfølgende ikke er nødvendige for at kommunikere med noden. Ydermere haves en variabel, der angiver, om noden er betragtet som kompromitteret eller ej. Sidst men ikke mindst holder objektet tiden for den sidst modtagne pakke fra noden. Dette er vel at mærke afsenderens lokale tid og benyttes til at forhindre replay-angreb, som beskrevet i 4.4.2.

Et node opretter et peer-objekt om en anden node, når denne er autentificeret. Objekterne gemmes i en `java.util.Vector` i peerManager-klassen, hvorfra de kan tilgås, når det ønskes. peerManager-klassen har således til formål at styre informationen om de forskellige noder.

De certifikater og asymmetriske krypteringsnøgler, som applikationen kender til, gemmes i en `java.security.KeyStore`, som ligeledes styres af peerManager-klassen. Denne `keyStore` tilgås vha. et password, som brugeren har defineret i forbindelse med oprettelsen af et nyt certifikat. Her dannes keyStore-databasen indeholdende brugerens nye nøglepar sammen med brugerens Certificate Signing Request, jf. afsnit 4.5.2. Det password som brugeren angiver sammen med sin personlige information til certifikatet, bruges til at beskytte keyStore-databasen og dermed den private nøgle. Dette password skal således benyttes, når brugeren skal identificere sig overfor systemet ved opstart. Systemet checker her, at brugerens certifikat er gyldigt og at der findes en brugbar privat nøgle. Denne opstartsverificering styres således af peerManager-klassen.

Peer-objekterne fra peerManageren gemmes ikke på harddisken ligesom sharing-objekterne. Dette skyldes, at informationen er let at indsamle, når en node tilslutter sig en gruppe, selvom data skulle være mistet. Da certifikaterne allerede er gemt i keyStore-databasen, behøver brugeren ikke deltage i autentificeringen af de samme medlemmer igen. Noderne vil således blive automatisk autentificeret.

I modsætning til sharing-objekterne er peer-objekternes integritet ikke sikret af systemets protokoller. Dette skyldes, at dette ikke er ligeså kritisk for systemets pålidelighed. Hvis modstanderen ødelægger peer-objekternes integritet, vil dette blot betyde, at noden ikke kan kommunikere med andre noder. Ved at genstarte noden bliver problemet løst, da den herved vil indsamle den korrekte information igen. Man vil dog forholdsvist let kunne udvide integritetsprotokollen til at omfatte peer-objekterne, hvis dette skulle ønskes. Dette kunne foregå ved, at noderne sammenligner peer-data på samme måde, som protokollen for integritetscheck sammenligner sharing-objekter.

5.2.3 Dataklasser i Objects-pakken

I dette afsnit vil Objects-pakkens dataklasser og disses funktioner blive gennemgået.

netPacket
Klasserne encryptedNetPacket og peerPacket nedarver begge fra netPacket-klassen. Dette skyldes, at encryptedNetPacket og peerPacket har den fælles egenskab, at de er pakker, der sendes mellem noder. Dette er f.eks. en fordel, hvis der skal tilføjes fælles variable eller metoder, da disse herved blot vil kunne tilføjes til netPacket-klassen. I den nuværende version er netPacket-klassen dog tom (udover en konstruktor og en toString-metode) og har derfor ingen reel betydning.

Klassen encryptedNetPacket indeholder pakkestrukturen fra afsnit 4.4.1. Der benyttes således et encryptedNetPacket-objekt, hver gang en krypteret netværkspakke skal sendes til en anden node, uanset hvilken slags data, den skal indeholde. De krypterede data, den krypterede symmetriske nøgle og datasignaturen er alle repræsenteret vha. byte-arrays. Et encryptedNetPacket-objekt generes vha. cryptoTools-klassen som beskrevet i afsnit 5.3.

Et peerPacket-objekt er det eneste objekt, der sendes til andre noder, uden at være krypteret (bortset fra JXTA-systemets discovery-kommunikation, som kun styres overordnet af dette program). Objektet benyttes til autentificering af noderne (som beskrevet i afsnit 4.5.2) samt til at udbrede information om noderne. Derfor indeholder det et peerID (som benyttes af JXTA-systemet), et navn på noden, som benyttes til den grafiske repræsentation samt en certifikat-kæde, hvorfra identifikationsinformation og en offentlig nøgle kan findes.

netData
Klassen netData er grundklassen i et hierarki af klasser, der kan indeholde den information, som skal sendes på netværket. Alle navne på disse klasser ender for overskuelighedens skyld på NetData, som f.eks. klassen sharingNetData. NetData-klasserne sendes altid vha. et encryptedNetPacket-objekt, der er beskrevet ovenfor, sådan at data beskyttes. Klassen netData indeholder selv variablene `type (int)` og `packetTime (long)`. Variablen packetTime har til formål at forhindre replay-angreb, jf. afsnit 4.4.2. Årsagen til at hvert netData-objekt ligeledes indeholder en type er, at dette giver mulighed for at genbruge de samme objekter (pakker) i forskellige sammenhænge, hvis dette skulle være ønsket.

Herunder er de grundlæggende NetData-klasser og deres indhold listet:

- netData: `int type, long packetTime`
  - allVerifyValuesNetData: `verifyObject[] verifyObjects, String[] verifySignatures, BigInteger sharingID`
    - verifyObject: `Vector verifyPoly, Vector verifyUpdateShares, BigInteger sharingID`
  - deleteSharingNetData: `BigInteger sharingID`
  - fileReceivedNetData: `BigInteger sharingID, boolean ok`
  - fileRestoreNetData: `BigInteger sharingID`
  - integrityNetData: `sharing s`
  - leaveGroupNetData: `String ownerID`
  - recoverMyShareNetData: `BigInteger sharingID`
  - recoverSharingNetData: `sharing s`
  - requestSharingDataNetData: `-`
  - sharingDataNetData: `Vector sharingDatas`
  - sharingNetData: `sharing s`
  - sharingReceivedNetData: `BigInteger sharingID, boolean ok`
  - sharingUpdateNetData: `verifyObject verifyObj, String verifySign, BigInteger sharingID, BigInteger shareUpdateVal, sharingPeer sPeer`
  - startRecoverNetData: `BigInteger sharingID, int serverNr`
  - startSharingUpdateNetData: `BigInteger sharingID, boolean updateOperation`
  - updateValidationOkNetData: `BigInteger sharingID`

Der findes desuden NetData-klasser, som bruges til beskyldninger og til håndtering af special-cases, såsom manglende svar til dele af systemet fra en node, jf. afsnit 4.5.6. For et overblik over de forskellige objekters brug, henvises til sekvensdiagrammerne i afsnit 4.5.

Grunden til at alle netData-klasserne nedarver fra netData er, at databeholderne derved kan behandles, uden at den specifikke type kendes. De forskellige netData-objekter nedarver dog kun fra netData-objektet og ikke fra hinanden, da de ikke har nogen fællesegenskaber, der retfærdiggør dette, selvom de ofte indeholder nogle af de samme variable.

controlJob og sendJob

Klasserne controlJob og sendJob holder blot data hørende til et givent job til hhv. Control-tråden og en packetSender-tråd. Se kildekoden i Appendiks B for detaljer.

5.3 KRYPTERINGSVÆRKTØJER

Samtlige metoder til at kryptere og dekryptere data, både med asymmetrisk og symmetrisk kryptografi, er placeret i en klasse kaldet `cryptoTools` i Main-pakken. Denne indeholder desuden funktionalitet til at hashe en fil og til at lave signaturer og verificere disse.

Metoden `createEncryptedNetPacket` har en central rolle. Vha. klassens øvrige metoder opbygger denne metode den i afsnit 4.4.1 beskrevne pakkestruktur i et encryptedNetPacket-objekt. Først laves en signatur af data vha. afsendernodens private RSA-nøgle. Dernæst genereres en tilfældig 128 bit AES-nøgle, som bruges til at kryptere data, hvorefter AES-nøglen krypteres med modtagerens offentlige RSA-nøgle. Tilsammen giver disse en krypteret netværkspakke, som kun den tiltænkte modtager kan læse. Metoden `decryptEncryptedNetPacket` har en tilsvarende central rolle, da denne står for modtagelsen af en krypteret netværkspakke. Udover at dekryptere i den omvendte rækkefølge, verificeres signaturens korrekthed.

Metoderne `encryptFile` og `decryptFile` bruger en AES-nøgle til at kryptere eller dekryptere en fil fra den lokale maskine. Den resulterende fil gemmes direkte på harddisken. Når en backup oprettes eller gendannes er det således disse metoder, der krypterer og dekrypterer selve backup-data. AES-nøglen skal siden hen deles vha. secret sharing. Derfor laves denne ved at generere en tilfældig 2048 bit værdi modulo 128 bit. Dermed bliver AES-nøglen 128 bit som ønsket, mens den delte hemmelighed bliver hele 2048-bit-værdien, da det diskrete logaritmeproblem herved bliver svært at løse. Når hemmeligheden skal genskabes er det således en 2048 bit værdi, som genskabes, men ved atter at tage modulo 128 bit på denne, fås den anvendte AES-nøgle. Selve nøglegenereringen foregår i Control-klassen, sådan at metoderne i cryptoTools-klassen kun står for selve krypteringen og dekrypteringen.

Brugen af AES og RSA er i dette system baseret på en implementering leveret af BouncyCastle.org. Dette er en gratis implementering, der er valgt, fordi den nuværende version af Java ikke inkluderer en RSA-implementering. Dette betyder naturligvis, at sikkerheden i systemet også kommer til at afhænge af, at BouncyCastle er til at stole på. Brugen af BouncyCastle bør derfor tages op til overvejelse ved udarbejdelsen af en eventuel kommerciel version.

Det indbyggede java-program ”keytool” benyttes til at generere asymmetriske krypteringsnøgler og til at generere Certificate Signing Requests (CSR). Desuden benyttes keytool til at manipulere med keystore-databasen (importere certifikater). Denne del hører dog under opstart og autentificering og er således styret af peerManager-klassen.

5.4 IMPLEMENTERING AF PROGRAMMETS TRÅDE

Dette afsnit beskriver implementeringen af applikationens tråde. Trådenes funktionalitet vil blive skitseret og herved vil programmets vigtigste klasser blive berørt. Trådenes samarbejde er desuden skitseret i slutningen af afsnittet.

5.4.1 jobMonitor og sendJobMonitor

Klassen jobMonitor er en implementering af en jobkø. Køen er implementeret vha. en monitor, sådan at de kritiske variable (de forskellige jobs) beskyttes. Der findes to metoder. En metode (`void putJob(Object job)`) tilføjer et job, og en anden (`Object getJob()`) henter et job efter First-In First-Out (FIFO) princippet. FIFO-princippet er valgt, da en prioritering af jobs ikke umiddelbart giver nogen fordel. Man kunne dog siden hen vælge at prioritere jobs, hvis det viser sig, at der opstår flaskehalsproblemer og nogle jobs tydeligt er vigtigere end andre.

Klassen jobMonitor kan principielt agere jobkø for både Control og for packetSender-trådene. Det har dog vist sig praktisk at udvide den grundlæggende funktionalitet i jobkøen til packetSender-trådene, da jobkøen herved kan bortsortere nogle overflødige jobs og dermed mindske applikationens arbejde. Derfor er klassen sendJobMonitor konstrueret. Denne indeholder de samme metoder, som jobMonitor, men disses funktionalitet er udvidet, sådan at overflødige peerPacket-objekter bortsorteres. Disse objekter stammer fra Control-tråden, efter denne har fået input fra jxtaManager-tråden, som har fundet et nyt gruppemedlem. Hvis pakken til det fundne gruppemedlem allerede findes i jobkøen, ignoreres det nye job således.

5.4.2 jxtaManager

Tråden jxtaManager er placeret i pakken JxtaSys. Denne har til formål at initialisere nodens kommunikation med JXTA-systemet. Det er således denne, der sørger for at noden tilsluttes den fælles backupgruppe, der benyttes til at publisere information om egentlige backupgrupper, jf. afsnit 4.3. Når denne gruppe er tilsluttet, er det ligeledes jxtaManager-tråden, der sender discovery-forespørgsler til JXTA-systemet, for at finde brugbare backupgrupper.

Efter en egentlig backupgruppe er tilsluttet, sørger jxtaManager for at finde gruppens medlemmer, sådan at kommunikationen med disse kan påbegyndes. Dette sker ved at ændre discovery-forespørgslerne til at lede efter noder. Desuden ændres gruppen, hvori der ledes, da det nu er selve backupgruppen og ikke den fælles standardgruppe, der skal ledes i.

Forespørgslerne sendes hvert 15. sekund og svar fra JXTA-systemet modtages asynkront af jxtaManager-trådens dicoveryEvent-metode. Metoden sørger for at sende brugbar information til Control-tråden vha. dennes jobkø.

I jxtaManager-tråden findes metoderne createGroup, joinGroup og leaveGroup, som kan kaldes fra Control. Control kan således styre, hvilken gruppe applikationen skal være en del af. Metoderne benytter sig af hjælpeklassen peerGroupTool, som indeholder funktionalitet til at

oprette og tilslutte en JXTA-gruppe samt til at forlade den. Når en gruppe tilsluttes sker det i denne prototype uden autentificering, hvilket betyder, at grupperne ikke er password-beskyttede (selvom den grafiske brugergrænseflade understøtter dette). Dette vil dog kunne ændres ved at udvide peerGroupTool-klassen. Det bemærkes endnu engang, at dette ikke er kritisk, da systemets sikkerhed ikke afhænger af en gruppeautentificering.

5.4.3 GUI

Den grafiske brugergrænseflade (GUI) er bygget op til at understøtte brugen af systemet, som beskrevet i use case beskrivelserne i afsnit 4.1. Dette afsnit har til formål at give et overblik over de involverede klasser.

Grænsefladens tre primære opgaver er, at formidle:

- Indledende autentificering af en bruger overfor systemet.
- Indmeldelse i en backupgruppe.
- Systemets hovedfunktionalitet fra programmets hovedvindue.

GUI-pakkens klasser kan således inddeles i disse tre kategorier. Den første del, der håndterer den indledende autentificering, består af klasserne keyPassFrame og newCertFrame. Vinduet keyPassFrame er det centrale i denne operation og dette tager imod brugerens password. newCertFrame benyttes til at lave et nyt nøglepar og et certifikat (selve certifikatet udstedes af certifikatautoriteten).

Indmeldelse i backupgrupper sker vha. klasserne selectGroupFrame, newGroupFrame og joinGroupFrame. Vinduet selectGroupFrame viser tilgængelige backupgrupper, og indeholder knapper til at udføre operationens funktionalitet. newGroupFrame og joinGroupFrame håndterer hhv. oprettelsen af en ny backupgruppe og indmeldelse i en eksisterende.

Hovedvinduet og dets funktioner håndteres af klasserne mainWindow, theMenuBar, trustCertFrame og backupFileFrame. Her er mainWindow den centrale klasse, der indeholder selve hovedvinduet og dets funktioner. theMenuBar viser hovedvinduets menubar i toppen af vinduet. Når en bruger manuelt skal autentificere en anden, benyttes trustCertFrame, der viser den anden nodes certifikat samt muligheder for brugeren for enten at acceptere eller afvise. Vinduet backupFileFrame bruges til oprettelsen af en backup ved at modtage information fra brugeren om deltagere i backup'en og backupkonfigurationen (værdien k).

Tilovers er der nu en række fællesklasser, som benyttes flere forskellige steder. De tre vinduer okFrame, okCancelFrame og largeOkFrame er dialogvinduer, der benyttes mange forskellige steder i applikationen. Desuden findes klassen guiStandards. Klassen er den eneste i GUI-pakken, hvis navn ikke ender på "Frame" (bortset fra theMenuBar). Dette skyldes, at guiStandards ikke selv er et vindue, men i stedet indeholder information om forskellige komponenters udseende. Ved at alle Frame-klasser benytter disse standarder, kan applikationens udseende let ændres i hele applikationen, blot ved at ændre på standarderne. Denne vil ligesom resten af grænsefladen kunne videreudvikles i en evt. kommerciel version.

Skabelonerne til de fleste GUI-klasser er grundlagt vha. et lille gratis java-program kaldet javagui.exe. Vinduerne kan herved laves hurtigere, end hvis de skulle bygges fra bunden. Programmet indeholder simpel funktionalitet til at oprette et vindue og dets mest grundlæggende komponenter. Skabelonerne er herefter udbygget med den ønskede funktionalitet og det ønskede udseende.

5.4.4 packetSender og packetReceiver

Tråden packetSender findes i pakken Send. En applikation opretter en packetSender-tråd for hver node, der kommunikeres med i en backupgruppe. Tråden indeholder derfor information om modtagernoden, som benyttes til at sende en pakke via JXTA-systemet. Da de enkelte grupper antages ikke at have ret mange medlemmer, bør det ikke være et problem at håndtere mængden af tråde. Trådene styres af sendManager-klassen i Main-pakken og denne opretter en jobkø (sendJobMonitor) til hver packetSender-tråd.

Når der ikke sendes data til en packetSender-tråds modtager, er denne inaktiv. Når Control-tråden vælger at sende en pakke til modtageren laves et sendJob-objekt, som gives til den rette packetSender-tråds jobkø. packetSender-tråden vil hente et job ad gangen fra jobkøen og sørge for at sende data (et netPacket-objekt) til modtageren. Forbindelsen til modtagernodens packetReceiver-tråd oprettes vha. en net.jxta.socket.JxtaSocket. Hvis det data, der skal sendes er en krypteret netværkspakke, vil packetSender-tråden forsøge at sende denne tre gange (hvis det ikke lykkes første gang), før den opgiver. Ellers er det tale om en peerPacket, hvor et forsøg er tilstrækkeligt, da jxtaManageren blot vil finde den samme node igen, hvis ikke det går godt første gang. Der er således ingen grund til at blokere tråden længere end højest nødvendigt.

Der findes kun en packetReceiver-tråd i hver applikation. Denne modtager pakker fra alle gruppemedlemmer. De modtagne pakker er netPacket-objekter, som videresendes til Control-klassen som et controlJob-objekt vha. Control-trådens jobkø (jobMonitor). Forbindelsen til afsendernoden (og dennes packetSender-tråd) oprettes vha. en net.jxta.socket.JxtaServerSocket. Grundlæggende fungerer packetReceiver-tråden som en uendelig løkke, der opretter en forbindelse, modtager en pakke og lukker forbindelsen igen.

Forbindelserne oprettes vha. modtagerens peerID og pipeAdvertisement samt den PeerGroup, som både afsenderen og modtageren er medlem af. Hvis disse ikke matcher hos afsender og modtager, vil pakken ikke nå frem. Brugen af denne information skyldes JXTA's abstraktionslag, hvor man således ikke bruger IP-adresser og port-numre. JXTA finder selv frem til de korrekte noder, ud fra den givne information

5.4.5 fileSender og fileReceiver

Både fileSender-tråde og fileReceiver-tråde oprettes dynamisk, når en fil skal sendes. Klassen sendManager i Main-pakken holder referencen til fileSender-trådene, mens Control-klassen holder fileReceiver-trådene. Det er naturligt at placere referencen til fileReceiver-trådene i Control, idet de initialiseres i forbindelse med begivenheder i systemets protokoller, som ligeledes styres af Control.

Trådene fileReceiver og fileSender benytter net.jxta.socket.JxtaSocket og net.jxta.socket.JxtaServerSocket til at oprette forbindelsen på samme måde som packetReceiver-tråden og en packetSender-tråd. Når en forbindelse er oprettet sendes først størrelsen på filen, sådan at modtageren kan beregne, hvor lang tid det vil tage at hente denne. Denne funktionalitet udnyttes dog ikke i den nuværende prototype.

5.4.6 Control

Control-tråden er programmets hovedtråd. Den styrer således programmets øvrige tråde og alt input til applikation går gennem denne. Input kan komme fra brugeren vha. brugergrænsefladen, fra andre applikationer i form af netværkspakker, fra JXTA-systemet som discovery events og fra diverse timere, som er sat i gang i forbindelse med systemets protokoller.

Control-tråden bearbejder det modtagne input og styrer således programmets protokol. Dette gøres ikke alene af Control-klassen, men vha. alle hjælpeklasserne i Main-pakken, hvoraf klasserne shareManager og peerManager er allerede beskrevet i afsnit 5.2 sammen med peer-objektet og sharing-objektet. Når Control skal sende data til en anden node, bruges sendManager-klassen. Denne styrer sende-trådene og sørger for at kryptere data vha. cryptoTools-klassen. Det er således i sendManager-klassen, at listen af packetSender-objekter og fileSender-objekter findes. Den primære metode i sendManager-klassen er metoden sendPacket (`void sendPacket(netPacket np, PeerAdvertisement peerAdv, PipeAdvertisement pipeAdv, String netData)`), som sørger for at sende en pakke til modtageren ved at bruge den tilhørende packetSender. Tilsvarende sørger metoden sendFile (`void sendFile(File f, PeerAdvertisement peerAdv, PipeAdvertisement pipeAdv)`) for at oprette en fileSender-tråd, som sender den ønskede fil.

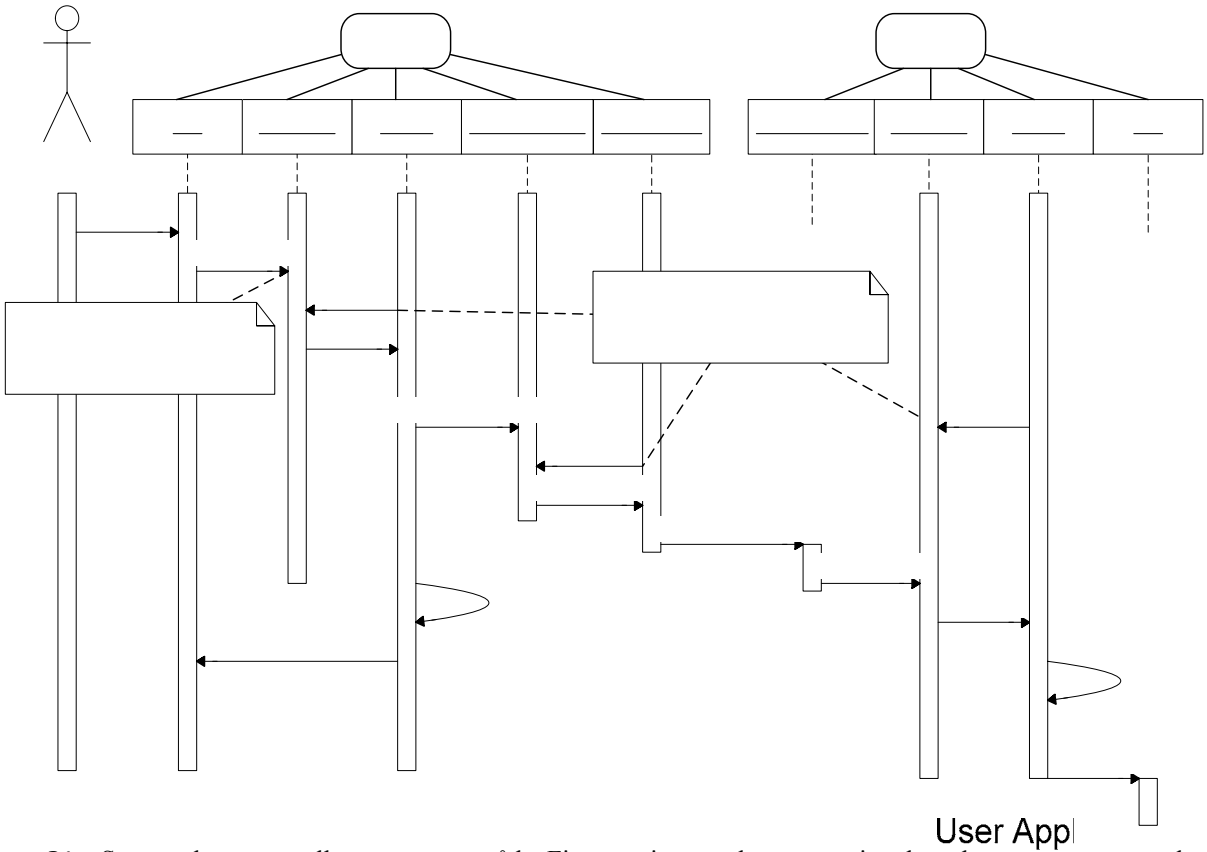
Control-klassen selv indeholder funktionalitet til at initialisere systemet og udføre dets grundlæggende funktioner og protokoller. Derudover indeholder Control-klassen de variable, som skal bruges til protokollerne, men som ikke findes i sharing-klassen eller peer-klassen. Dette er bl.a. lister over eksisterende gruppemedlemmer, som skal repræsenteres i brugergrænsefladen samt midlertidigt data til autentificering. For overskuelighedens skyld, er vedligeholdelsesprotokollerne (integritetscheck, opdatering og share-gendannelse) placeret i separate klasser (integrityProtocol og updateProtocol), da disse er omfangsrige. Klassen updateProtocol styrer både share-opdatering og share-gendannelse, da disse operationer minder meget om hinanden. Kald til vedligeholdelsesprotokollerne går dog stadig igennem Control-klassen.

Control-trådens interface er defineret vha. controlJob-klassen og kan kun kaldes vha. dette. Dette primære interface tilgås ved, at Control-klassen switcher på eksisterende controlJob-objekter i jobMonitoren. For at sende en opgave til Control, skal en anden tråd altså oprette et controlJob-objekt og sende dette til jobMonitor-tråden, som opbevarer jobbet, indtil Control spørger efter flere jobs. Control-klassen indeholder dog ligeledes et sekundært interface, som switcher på forskellige krypterede netværkspakker, efter disse er blevet dekrypteret. Hver gang en krypteret netværkspakke modtages i det primære interface sendes denne således videre til det sekundære interface, mens alt andet input håndteres af det primære interface. De vigtigste metoder fra de to

interfaces er indirekte beskrevet via protokol-beskrivelserne i afsnit 4.5. For yderligere information henvises til kildekoden i appendiks B.

5.4.7 Trådenes sammenhæng

For at illustrere trådenes sammenhæng gives her et kort eksempel på en af programmets operationer. Operationen er skitseret vha. et sekvensdiagram, hvor de medvirkende tråde er inkluderet. Den illustrerede operation er at slette en backup. Denne er valgt, fordi den er simpel men alligevel berører systemets væsentligste tråde. Sekvensdiagrammet vil primært beskrive trådenes samarbejde og vil således ikke inkludere andre detaljer om operationen.



Figur 21 – Sammenhængen mellem systemets tråde. Figuren viser en slette-operation, hvor brugeren starter med at give GUI'en en slette-kommando. Denne videregives til Control (vha. dennes monitor), som sender en kommando til en packetSender-tråd, vha. dennes monitor[22]. Herefter sendes kommandoen til modtagernoden, hvor packetReceiver-tråden modtager pakken og sender kommandoen til Control (igen vha. dennes monitor). Efter udførsel af kommandoen opdateres GUI'en og operationen afsluttes.

[22] Da der er en packetSender-tråd for hver modtagernode, gøres dette n-1 gange til n-1 forskellige packetSender-tråde.

De tråde, der ikke er inkluderet på Figur 21 er jxtaManager-tråden, fileSender-tråden og fileReceiver-tråden (samt diverse timere). Alle kommunikerer de med Control-tråden vha. jobMonitor-tråden, ligesom GUI-tråden og packetReceiver-tråden på Figur 21.

5.5 BRUGEN AF FELDMANS VSS

I afsnit 4.4.3 er det beskrevet, at de hemmelige nøgler, der skal deles vha. secret sharing, skal være på 2048 bit for at undgå angreb mod det diskrete logaritmeproblem. I sig selv er dette ikke noget problem, da det er hurtigt at lave en tilfældig værdi af denne størrelsesorden som siden hen kan bruges til at aflede en AES-nøgle. Dette betyder imidlertid også, at konstanterne fra Felmans VSS, p, q (og g) skal være af samme størrelsesorden (jf. afsnit 2.2.2), og da der er visse krav til disse, er de noget mere tidskrævende at beregne. Til gengæld skal det kun gøres en gang, hvorefter værdierne kan bruges til alle fremtidige backup'er, så længe nøglestørrelsen ikke øges.

Derfor laves et separat program, der udregner disse værdier kaldet *primes.java*. Som inputargument modtager programmet en minimums-bitstørrelse, dvs. størrelsen på den hemmelighed, der skal deles. Programmet skal derefter finde primtallene p og q, sådan at $p = mq + 1$ (siden hen findes g).

Programmet fungerer således:
Først findes et tilfældigt primtal p vha. `java.math.BigInteger`-klassens `probablePrime`-funktion. Derefter divideres p-1 med forskellige værdier af m, i dette tilfælde alle heltal fra 2 til 1000, og hvis divisionen ikke giver en rest, undersøges det, om tallet er et primtal vha. `isProbablePrime`-metoden[23]. Hvis dette er tilfældet, er q fundet. Hvis ikke dette lykkes, starter operationen forfra og finder et nyt tilfældigt primtal p.

Når p og q er fundet, findes g hurtigt. Dette gøres ved at teste mulige værdier for g i størrelsesordnen q, og finde ud af, om $g^q = 1 \pmod{p}$. Hvis dette er tilfældet, er en generator fundet.

Ved en test af programmet med 2048 som inputargument, tog det programmet ca. 1½ time at udregne nogle brugbare værdier på en 1400 MHz AMD processor med 512 Mb RAM. Dette skyldes primært, at tiden for generering af så store primtal gennemsnitlig ligger på ca. et halvt minut. Når først et p af denne størrelsesorden er fundet, tager det kun ca. 1 sekund at teste, om der findes et tilhørende q.

De fundne værdier hardkodes nu direkte i shareManager-klassen, sådan at denne har værdierne til rådighed, når nye sharing-objekter skal oprettes. Værdierne placeres her og gives videre som argument, når et sharing-objekt oprettes i stedet for at hardkode disse direkte i sharing-objektet. Herved vil en senere udvidelse af systemet kunne lave sharing-objekter med forskellige værdier af p, q og g, hvis dette skulle ønskes. Primes.java kan findes i kildekoden i Appendiks B.

[23] Da p på denne måde kan blive 1000 gange større end q, skal det sikres, at det tilfældige p er større end $m *$ minimumsbitstørrelse (dvs. $p > 1000 * 2048$ bit ≈ 2058 bit).

5.6 OPSUMMERING

Dette kapitel har beskrevet prototype-implementeringen af det designede system. Det resulterende program har fået titlen Resilia og installation samt brug af Resilia er beskrevet i Appendiks A.

Resilia er opdelt i 6 Java-pakker og disse er beskrevet kort og de tilhørende klasser er listet. Systemets primære datastrukturer er defineret vha. sharing-objektet og peer-objektet, som indeholder metoder og variable om hhv. en backup og en node.

Desuden er implementeringen af Resilias krypteringsfunktioner beskrevet. Bouncy Castles implementeringer af RSA og AES er benyttet her.

Til sidst er programmets tråde beskrevet, og deres samarbejde er illustreret. Control-tråden i Main-pakken indeholder systemets interface og dette kan bl.a. tilgås fra brugergrænsefladen, fra JXTA-systemet og fra andre noder (dvs. fra modtage-trådene).

Mere information om implementeringen kan fås ved at læse kildekoden til Resilia, som er placeret i Appendiks B.

6 EVALUERING

Dette kapitel har til formål at beskrive evalueringen Resilia. Dette sker på baggrund af en afprøvning af funktionaliteten samt en evaluering af systemets sikkerhed og ydelse. Desuden bliver JXTA-systemet evalueret separat.

6.1 VALG OG BESKRIVELSE AF TESTMETODE.

For at kunne sandsynliggøre at Resilia lever op til de specificerede krav, bør programmet testes på forskellige måder.

Først og fremmest er en funktionel test oplagt for at sandsynliggøre, at Resilia opfører sig som ventet, uden at der opstår fejl. Optimalt set skal denne test bevise, at der ikke er nogen fejl i programmet og at det er robust overfor alle tænkelige hændelser. Dette er dog oftest så godt som umuligt, da dette vil kræve, at man tester alle mulige hændelser i alle systemets mulige tilstande. I et distribueret system er dette endnu sværere, da det kan være meget svært (og til tider umuligt) at bestemme et distribueret systems globale tilstand. Afsnit 6.2. beskriver, hvordan Resilias funktionalitet er blevet afprøvet.

Hvis det tager 26 timer at lave den daglige backup, fordi backup-systemet er langsomt, er systemet naturligvis ubrugeligt. Derfor skal den sidste test evaluere Resilias ydelse og vurdere, om denne er acceptabel. Der skal således tages tid på systemets basale operationer. Dette er beskrevet i afsnit 6.3.

Et hovedformål med Resilia er at opnå sikkerhed i form af fortrolighed, integritet og tilgængelighed af data. Derfor er det ikke tilstrækkeligt, at systemet udadtil opfører sig som ventet, uden at der opstår fejl, da der stadig kan eksistere sikkerhedshuller. Det er altså oplagt at evaluere systemets sikkerhed. Dette er beskrevet i afsnit 6.4.

Tests gennemføres på et lokalnetværk og vha. flere noder på samme maskine. Dette vil naturligvis udelukke en række fejl, som vil kunne opstå, hvis testen blev gennemført på internettet. Desuden må man forvente hastighedsforringelser af systemet, når dette skal fungere på internettet pga. mindre båndbredde og mere komplicerede ruter mellem noderne. Disse fejl og hastighedsforringelser er dog centrerede omkring JXTA-systemet og bør således primært kræve tilpasning af JxtaSys-pakken. JXTA-systemet evalueres separat til sidst i kapitlet.

Begrænsningerne af testene er således foretaget for at fokusere på systemets protokoller, der betragtes som den mest vitale del af Resilia, både hvad angår sikkerhed, pålidelighed og funktionalitet. Desuden begrænses testene, fordi den afsatte tid til projektet er begrænset.

6.2 AFPRØVNING AF FUNKTIONALITET

Dette afsnit beskriver den funktionelle afprøvning af Resilia. Først beskrives overordnet, hvordan en funktionel test kunne se ud, hvis der var flere ressourcer til rådighed. Dernæst beskrives den egentlige afprøvning, der er foretaget. Årsagen til at den først beskrevne test ikke gennemføres, skyldes de tidsmæssige rammer for projektet. Den egentlige afprøvning af systemet har således ikke til formål at stille garantier for Resilias funktionalitet eller robusthed men er blot et forsøg på at sandsynliggøre, at systemet fungerer efter hensigten.

6.2.1 Testniveauer

En funktionel test bør under optimale forhold foretages på flere niveauer. Her gives kort et forslag til tre testniveauer, som vil kunne sørge for, at et system som Resilia, vil kunne testes grundigt.

På komponentniveau testes hver klasse for sig ved at give noget input og kontrollere, at det tilhørende output er som forventet. Ved en sådan test bør alle metoder testes og optimalt set skal alle dele af metoderne afprøves, sådan at alt kode er blevet gennemløbet. I praksis kan dette dog være yderst tidskrævende.

Det næste testniveau er en test af den samlede applikation. Her er de forskellige komponenter sat sammen, og den samlede applikation kan modtage input. Ved forskellige slags input kontrolleres det tilhørende output samt applikationens tilstand før og efter operationen. Denne test kan ligeledes være meget omfattende, især hvis man forsøger at teste applikationen med alt muligt input.

Det tredje og højeste niveau er en test af det samlede distribuerede system. Her testes de forskellige noders samspil ved systemets operationer samt hver enkelt nodes tilstand før og efter. Optimalt set bør alle mulige operationer foretages ved alle systemets tilstande, men dette er oftest umuligt pga. for mange mulige tilstande (som i øvrigt ikke altid kan bestemmes i det distribuerede system).

6.2.2 Afprøvning af Resilia

Under implementeringen af Resilia er de enkelte komponenter afprøvet undervejs. Dette er primært foregået ved udskrifter til terminalen, som kan vise, om komponenten opfører sig korrekt. Den ønskværdige systematik er dog ikke blevet benyttet og denne afprøvning vil således ikke blive dokumenteret.

Den egentlige afprøvning af systemet er således udelukkende foregået på et højt niveau, dvs. primært på det distribueret systems niveau. Test på applikationsniveau er kun foretaget i forbindelse med afprøvningen af det distribuerede system samt ved afprøvning af de funktioner, som ikke er distribuerede. Afprøvningen er desuden foregået uden at teste alle tænkelige special-cases, dels fordi nogle special-cases ikke er implementeret i denne prototype og dels pga.

projektets tidsmæssige rammer. Resilias vigtigste funktionalitet er dog blevet afprøvet, sådan at det kan sandsynliggøres, at systemet virker efter hensigten.
Systemet bør kunne klare, at optil *k*-1 noder på samme tid afviger fra systemets protokol. Afvigelserne kan enten foregå ved, at noderne sender forkert information, eller ved at de slet ikke svarer. Afvigelserne kan forekomme på vilkårlige tidspunkter under alle systemets operationer. Det er dog stort set umuligt, at bevise denne robusthed, da der er enormt mange forskellige tilstande og tilhørende operationer, som vil skulle testes.

Afprøvningen af Resilia er foretaget ved at afvikle systemets funktioner i nogle forskellige opsætninger og sikre, at der ikke opstår kritiske fejl. Det er således de mest tænkelige situationer, der er opstillet og afprøvet. Afprøvningen er opdelt i *opstart* og *hovedfunktioner* ligesom use case beskrivelserne i afsnit 4.1. Der er dog tilføjet væsentligt flere cases, for at komme flere dele af programmet igennem. Flere af disse har til formål at afprøve Resilias robusthed, når dele af systemet ikke følger protokollen. Dette er et forsøg på at sandsynliggøre, at Resilia er robust overfor forskellige fejl, men skal ikke ses som et forsøg på at bevise, at systemet er robust overfor alle mulige hændelser.

Afprøvningen af funktionaliteten består således af en række test-cases, som er sorteret i underafsnit efter de funktioner, som skal afprøves. I hvert underafsnit forklares operationernes formål kort. Derefter listes de afprøvede test-cases. Til sidst gives resultatet af de forskellige test-cases. Testene udføres ved at starte det opstillede systems noder op. Nogle noder manipuleres til at opføre sig, som de forskellige test-cases foreskriver (f.eks. skal nogle noder undlade at svare på givne operationer). De special-cases, som med vilje ikke er implementeret, vil naturligvis heller ikke blive testet.

I denne afprøvning er det primære succeskriterium, at backup'en kan genskabes efter operationerne (med undtagelse af "Slet"-operationen og opstartsoperationerne, som har sine egne succeskriterier). Hvis dette kan lade sig gøre, er det samtidig vist, at de involverede shares er korrekte, så derfor benyttes gendannelsesoperationen til at hjælpe med at afgøre, om de forskellige cases succeskriterier er opnået. Herudover skal systemet (alle noder) være i en tilstand, der tillader noderne at fortsætte med systemets operationer. Udover disse primære succeskriterier, har hver operation sine egne formål, som ligeledes skal opfyldes. Udskrifter til terminalen hjælper med at afgøre, hvordan en operation er forløbet. Alle større klasser har en boolsk variable kaldet *debugging*. Hvis denne tildeles værdien *true*, vil klassen udskrive relevant information til terminalen. Eksempler på udskrifter kan være fejlende verificering af share-værdier eller fejlende verificering af signaturer. Desuden giver udskrifterne til terminalen mulighed for at følge med i afviklingen af protokollerne. Ved hver test-case kontrolleres yderligere, at brugergrænsefladen opfører sig korrekt og ved operationer, der manipulerer med filer, kontrolleres det, at de forskellige filer er korrekte.

Opstart

Opstart og brugerautentificering
Når Resilia startes op, skal brugeren identificere sig selv vha. et certifikat. Desuden skal brugerens nøglepar ligge i peer.keystore. Hvis dette er tilfældet benytter brugeren sit password (til den private nøgle) til at identificere sig overfor systemet. Ellers skal et nyt nøglepar og et nyt certifikat laves.

Følgende cases afprøves:
1. Brugeren starter op uden et certifikat og nøglepar. Han kan således ikke starte programmet, men kan i stedet lave et nyt nøglepar i en ny peer.keystore-fil og en tilhørende Certificate Signing Request (CSR).
2. Brugeren starter op med korrekt certifikat, nøglepar og password.
3. Brugeren starter op med korrekt certifikat og nøglepar men et forkert password. (kan ikke lade sig gøre).
4. Brugeren starter op med et certifikat og et nøglepar, der ikke matcher. (kan ikke lade sig gøre).

Testen forløb uden problemer og de forskellige cases gav de forventede resultater. Det blev dog opdaget, at peer.keystore-filen ikke allerede må eksistere, når et nyt nøglepar og en Certificate Signing Request laves vha. Register-metoden. Denne skal således p.t. slettes manuelt, hvilket naturligvis bør rettes.

Indmeldelse i grupper og autentificering af øvrige noder
Når brugeren har autentificeret sig overfor systemet, skal denne melde sig ind i en backupgruppe. I grupperne skal noderne autentificere hinanden.

Der afprøves følgende cases:
5. Det kan lade sig gøre at oprette en ny gruppe med et udvalgt gruppenavn.
6. Det kan lade sig gøre at blive medlem i en eksisterende gruppe (optil 5 medlemmer testes).
7. Når man er medlem i en gruppe bliver de resterende gruppemedlemmer fundet, sådan at programmets hovedfunktioner kan benyttes.
8. Når en node forlader gruppen, opdaterer de øvrige gruppemedlemmer deres medlemsliste.
9. Flere grupper kan eksistere på samme tid, uafhængigt af hinanden, sådan at medlemmer kun ser dem, fra deres egen gruppe.
10. Noder med certifikater fra samme CA som noden selv, autentificeres automatisk.
11. Noder med forskellige CA skal autentificere hinandens certifikater manuelt.
12. Når et certifikat er autentificeret manuelt, vil dette i fremtiden blive automatisk autentificeret, selvom programmet genstartes.

Disse test-cases blev gennemført med succes og gav de forventede resultater. Afprøvningen medførte dog en justering af, hvor ofte jxtaManageren søger efter andre noder i gruppen. Intervallet sættes til 30 sekunder i stedet for 15, da noderne lader til at bruge en del ressourcer på disse forespørgsler. Dette betyder naturligvis også, at det kan tage lidt længere tid for noderne at finde hinanden første gang, hvilket dog ikke er kritisk.

Hovedfunktioner

Backup og restore

Når en backup skal foretages, skal den krypterede fil sendes til de deltagende noder. Desuden skal sharing-objekterne fordeles korrekt. De deltagende noders brugergrænseflader skal opdateres med backup'ens filnavn. Gendannelsesoperationen (restore) skal bruge mindst *k* af systemets shares til at genskabe krypteringsnøglen, som derefter kan dekryptere data.

De cases, der er afprøvet er:

13. Lav backup i en gruppe m. 2 medlemmer (k=2).
14. Lav backup i en gruppe m. 3 medlemmer (k=2).
    a. Når alle medlemmer deltager.
    b. Når 1 medlem ikke svarer, selvom medlemmet er valgt til at deltage.
    c. 2 medlemmer laver backup på samme tid.
15. Lav backup i en gruppe m. 4 medlemmer (k=2).
    a. Når alle medlemmer deltager.
    b. Når 1 medlem ikke svarer, selvom medlemmet er valgt til at deltage.
16. Lav backup i en gruppe m. 5 medlemmer (k=3).
    a. Når alle medlemmer deltager.
    b. Når 1 medlem ikke svarer, selvom medlemmet er valgt til at deltage.
    c. Når 2 medlemmer ikke svarer, selvom disse er valgt til at deltage.
    d. Når kun 3 medlemmer er valgt til at deltage. (n=3, k=2)
17. Gendan backup i gruppe m. 2 medlemmer (k=2):
    a. Når 2 shares er til stede.
    b. Når 1 share er til stede (må ikke kunne lade sig gøre).
18. Gendan backup i gruppe m. 3 medlemmer (k=2):
    a. Når 3 shares er til stede.
    b. Når 2 shares er til stede.
    c. Når 1 share er til stede (må ikke kunne lade sig gøre).
    d. Når en node sender et forkert share.
19. Gendan backup i gruppe m. 4 medlemmer (k=2):
    a. Når 4 shares er til stede.
    b. Når 2 shares er til stede.
    c. Når 1 share er til stede (må ikke kunne lade sig gøre).
20. Gendan backup i gruppe m. 5 medlemmer (k=3):
    a. Når 5 shares er til stede.
    b. Når 3 shares er til stede.
    c. Når 2 shares er til stede. (må ikke kunne lade sig gøre).

De ovenstående cases blev gennemført med succes og gav de forventede resultater.

Slet backup

Når en backup skal slettes, skal både det krypterede data og de tilhørende sharing-objekter slettes på de deltagende noder. Desuden skal backup'en fjernes fra brugergrænsefladen på noderne.

De afprøvede cases er:

21. Slet en backup i en gruppe med 3 medlemmer.
22. Slet en backup i en gruppe med 5 medlemmer.
23. Slet en backup, som man ikke selv ejer (må ikke kunne lade sig gøre).

Disse cases gav de forventede resultater.

Opdater shares

Opdatering må kun finde sted, når mindst k noder er enige. Hvis for få noder gennemfører opdateringen vil backup-data ikke kunne genskabes. Det er ligeledes et succeskriterium for opdateringsoperationen, at share-værdierne er ændret, sådan gamle shares ikke kan deltage i gendannelsen af hemmeligheden.

Afprøvede cases er:

24. Opdater shares i gruppe m. 2 medlemmer (k=2)
25. Opdater shares i gruppe m. 3 medlemmer (k=2)
    a. Når alle medlemmer deltager.
    b. Når 1 medlem ikke svarer.
    c. Når 2 medlemmer ikke svarer (må ikke kunne lade sig gøre).
    d. Efter b. afprøves det, om det medlem, der ikke svarede kan bruge sit share til at deltage i genskabelsen af backup'en (må ikke kunne lade sig gøre).
    e. 2 medlemmer starter opdateringen på samme tid.
26. Opdater shares i gruppe m. 5 medlemmer (k=3)
    a. Når alle medlemmer deltager.
    b. Når 1 medlem ikke svarer.
    c. Når 2 medlemmer ikke svarer.
    d. Når 3 medlemmer ikke svarer (må ikke kunne lade sig gøre).

De ovenstående test-cases forløb som forventet efter et par mindre rettelser. Således blev alle shares opdateret efter hensigten med undtagelse af de cases, som ikke skal kunne lade sig gøre (25c og 26d).

Check integritet

Integritetskontrollen har til formål at sikre, at både backupinformation (sharing-objekter) og backup-filer er brugbare og korrekte. Hvis dette ikke er tilfældet, skal fejlene rettes. Det første succeskriterium er altså, at fejl opdages, hvis de findes. Det andet kriterium er, at systemet er i stand til at rette fejlene. De udvalgte test-cases er således valgt på denne baggrund.

De valgte cases er:

27. Check integrity i en gruppe m. 2 medlemmer (k=2)
    a. Når alt er i orden.
    b. Når 1 medlem mangler filen.
    c. Når 1 medlem har en modificeret fil.
    d. Når 1 medlem har et modificeret sharing-objekt (kan ikke lade sig gøre).
28. Check integrity i en gruppe m. 3 medlemmer (k=2)
    a. Når alt er i orden.
    b. Når 1 medlem mangler filen.
    c. Når 1 medlem har en modificeret fil.
    d. Når 1 medlems share-værdi er modificeret.
    e. Når 1 medlems sharing-objekt er modificeret på andre måder.
    f. Når 1 medlems mangler filen og har en ændret share-værdi.
    g. 2 medlemmer starter operationen på samme tid.
29. Check integrity i en gruppe m. 5 medlemmer (k=3)
    a. Når alt er i orden.
    b. Når 1 medlem mangler filen og 1 medlem mangler.
    c. Når 1 medlem mangler filen og 3 medlemmer mangler.
    d. Når 1 medlem har en modificeret fil og 3 medlemmer mangler.
    e. Når 1 medlems share-værdi er modificeret.
    f. Når 1 medlems share-værdi er modificeret og 1 medlem mangler.
    g. Når 1 medlems sharing-objekt er modificeret på andre måder.
    h. Når 1 medlems sharing-objekt er modificeret på andre måder og 1 medlem mangler.
    i. Når 1 medlem mangler filen og har en ændret share-værdi og 1 medlem mangler.
    j. Når 2 medlemmer har en modificeret fil.
    k. Når 2 medlemmers share-værdi er modificeret.
    l. Når 2 medlemmers sharing-objekt er modificeret på andre måder.
    m. Når 2 medlemmer har modificerede sharing-objekter (både share-værdier og andre dele af objektet) og mangler filen.

De ovenstående cases gav de forventede resultater. De forskellige konstruerede fejl blev således rettet i forbindelse med operationerne. Under afviklingen af 29m opstod dog nogle interne JXTA-problemer i forbindelse med filoverførslerne. Operationen blev dog afviklet med succes alligevel og filerne blev også overført i den forbindelse. Fejlen betød således kun, at enkelte fileSender-tråde fejlede, men da nye tråde automatisk blev oprettet i andet forsøg på at sende filerne, var der ikke noget egentligt problem.

Find forsvundet backupinformation

Når en node har mistet alt information (sharing-objekter), sådan at et integritetscheck ikke kan startes herfra, kan den forsvundne information findes ved at noderne udveksler ikke-hemmelig information (dvs. sharing-objekter uden share-værdier). Succeskriteriet for denne operation er således, at en node får den rette information til at starte en integritetskontrol (som derefter vil genskabe backup-filen og share-værdien). Dette kan både være i tilfældet af nedbrud med datatab eller hvis en bruger vælger at skifte til en anden node, som naturligvis ikke har brugerens data.

De afprøvede cases er:

30. Find forsvundet information i en gruppe m. 3 medlemmer (k=2)
    a. Når en node har slettet sine sharing-objekter.
    b. Når en node har slettet sine sharing-objekter og backup-filer.
    c. Når en bruger starter op på en ny node.
31. Find forsvundet information i en gruppe m. 5 medlemmer (k=3)
    a. Når en node har slettet sine sharing-objekter og backup-filer.
    b. Når en bruger starter op på en ny node mens 1 medlem ikke svarer.

Resultaterne af afprøvningerne var positive. Share-information blev genskabt, sådan at et integritetscheck kunne startes. Herved blev både sharing-objekter og filer genskabt.

6.2.3 Kendte fejl efter afprøvningen af Resilia

Da denne version af Resilia kun er en prototype, findes flere småfejl eller mangler, som primært skyldes manglende finpudsning og manglende implementering af forskellige special-cases. De afprøvede test-cases i forrige afsnit sandsynliggør dog, at systemets grundlæggende funktionalitet er i orden. Der findes således ingen kendte alvorlige funktionelle mangler, der går ud over Resilias beskrevne grundfunktionalitet. Der er dog for få cases og for lidt systematik i de afprøvede cases til at give et brugbart billede af systemets robusthed. Hvis robustheden skal sandsynliggøres, bør systemet testes grundigere.

Nogle af de funktionelle mangler i den nuværende version af Resilia er, at tilstande, der er uden for den grundlæggende model, ikke håndteres. F.eks. vil en distribuering af en backup fejle, hvis mindre end k noder opfører sig korrekt. Systemet vil dog ikke opdage dette, da hændelsen ligger uden for modellens antagelser om, at der hele tiden eksisterer k korrekte medlemmer. Noderne i systemet vil derfor havne i en tilstand, hvor de bliver ved med at tro, at distribueringen er i gang. I praksis er det dog ikke usandsynligt, at hændelsen kan forekomme, f.eks. hvis alle systemets noder endnu ikke er startet korrekt op. Systemet bør således tage højde for mulige hændelser, selvom disse ligger udenfor modellen, hvis de kan forekomme i praksis. Ved den beskrevne hændelse kunne man f.eks. implementere en timer, som kunne afbryde operationen og give en fejlmeddelelse til brugeren, hvis ikke operationen lykkes efter et givent stykke tid.

6.3 YDELSESTEST

Denne ydelsestest har til formål at afgøre, om Resilias ydelse er tilstrækkelig til, at systemet er praktisk anvendeligt. Testen vil gennemgå systemets vigtigste operationer og disses hastighed vil blive evalueret.

Testen foretages på et 100 Mbit lokalnetværk med to PC'ere. Den første er en 1400 MHz AMD processor med 512 Mb RAM. Denne håndterer fra 1-4 noder afhængig af, hvilken test der udføres. Den sidste node er placeret på en 450 MHz PIII processor med 392 Mb Ram. Der må således forventes hastighedsforringelser i forbindelse med flere noder på samme maskine. Til

gengæld må hastigheden forventes at værre en del større, end hvis testen blev foretaget over internettet. Der findes således betydelige usikkerheder i denne test. Alligevel vil testen kunne antyde, om systemets hastighed er indenfor rimelighedens grænser, sådan at Resilia er brugbart i praksis.

Ligesom afprøvningen af funktionaliteten, udføres forskellige cases, som vil blive evalueret. Tiderne måles fra det tidspunkt, hvor Control-tråden får kommandoerne til det tidspunkt, hvor Control-tråden betragter opgaven som udført. Da måleværdierne alligevel er behæftet med en betydelig usikkerhed, vil der kun blive foretaget to målinger af hver case, der skal skitsere størrelsesordenen af tidsforbruget. Opstart og autentificering af noderne kræver næsten ingen tid på et lokalnetværk. Dette er således ikke en del af testen, som i stedet vil koncentrere sig om systemets hovedoperationer. Ved kørsel over internettet, er det dog ikke usandsynligt, at det vil tage et stykke tid, at finde de andre noder, men dette afhænger af JXTA's effektivitet.

6.3.1 Backup og restore

En af systemets vigtigste operationer er dannelsen af en backup. Denne operation vil være en af de mest hyppigt anvendte, da man ofte vil tage backup af data, uden at få brug for at gendanne det.

For at undersøge Resilias tidsforbrug til den administrerende kommunikation, foretages to slags test. I den første test er backupfilen en tom fil, som således er hurtig at overføre mellem noderne. Dermed vil tidsforbruget på operationen afspejle Resilias kommunikation og det interne arbejde i noderne. I den anden test har backupfilen størrelsen 10 Mb.

Hastigheden måles for grupper af forskellig størrelse, for at vurdere, hvordan Resilias hastighed ændres, når systemet skaleres.

Ved distribuering af en backup bestående af en tom fil var resultaterne:
Gruppe m. 2 noder (k=2): 8,7 sek. 7,4 sek.
Gruppe m. 3 noder (k=2): 9,7 sek. 8,3 sek.
Gruppe m. 5 noder (k=3): 16,4 sek. 14,8 sek.

Ved distribueringen af en backup bestående af en fil på 10 Mb var resultaterne:
Gruppe m. 2 noder (k=2): 20,0 sek. 18,1 sek.
Gruppe m. 3 noder (k=2): 23,7 sek. 21,8 sek.
Gruppe m. 5 noder (k=3): 44,9 sek. 47,8 sek.

Systemets hastighed lader ikke til at være et problem i grupper af denne størrelsesorden. Tiden for at tage backup afhænger af antallet af deltagende noder samt af størrelsen på backup-filen, som deles. Systemets nuværende design antyder dog, at der vil opstå et flaskehalsproblem omkring filejeren, hvis systemet skaleres. Dette skyldes, at filejeren sender backupfilen n-1 gange. Sammenlignes distribueringen af en 10 Mb fil for 3 og 5 noder, ses også en fordobling af den målte tid. Så længe grupperne holdes små, lader problemet dog ikke til at være kritisk. Også ved distribueringen af en tom fil stiger tidsforbruget tilsyneladende med antallet af medlemmer. Dette er dog ikke underligt, da der her skal sendes flere kommunikationspakker rundt i systemet.

Tiden for restore af en nøgle afhænger af k (og ikke i lige så høj grad af n), da k er antallet af shares, som skal indsamles. Før en restore operation foretages, antages det, at systemet er korrekt (dvs. at et integritetscheck er foretaget, sådan at noden selv har den korrekte backupfil og sit eget share). Systemet bruger således kun tid på at samle nøglen og dekryptere data.

Ved restore af en backup var resultaterne:
Gruppe m. 2 noder (k=2): 5,0 sek. 4,8 sek.
Gruppe m. 3 noder (k=2): 4,1 sek. 4,4 sek.
Gruppe m. 5 noder (k=2): 6,1 sek. 5,3 sek.
Gruppe m. 5 noder (k=3): 9,4 sek. 8,3 sek.

Disse tider er ikke et problem for systemet.

6.3.2 Vedligeholdelse af systemet

Hastigheden af opdateringsoperationen og integritetscheck-operationen testes også. Det er ikke givet, hvor ofte systemet i praksis vil skulle vedligeholdes af disse operationer, men mere end en gang om dagen virker umiddelbart voldsomt. I de fleste systemer vil en gang om måneden blive betragtet som mere end rigeligt. Til gengæld skal operationerne foretages på samtlige af systemets backup'er.

Resultaterne ved opdatering af shares var:
Gruppe m. 2 noder (k=2): 17,8 sek. 14,7 sek.
Gruppe m. 3 noder (k=2): 26,9 sek. 26,1 sek.
Gruppe m. 3 noder, hvor 1 node mangler (k=2): 136 sek. 133 sek.
Gruppe m. 5 noder (k=3): 69,8 sek. 76,9 sek.
Gruppe m. 5 noder, hvor 1 node mangler (k=3): 232 sek. 237 sek.

Oprationer hvor medlemmer mangler, er generelt mere tidskrævende end operationer, hvor alle medlemmer deltager. Dette skyldes, at systemet her venter på timeouts, før protokollen fortsætter, sådan at alle noder har en chance for at nå at deltage. I den nuværende version afhænger længden af timeouts af antallet af gruppemedlemmer, da flere medlemmer skal have mere tid til at gennemføre kommunikationen. Det er bl.a. derfor, operationen tager længere tid i en gruppe med 5 medlemmer, end i en gruppe med 3 medlemmer, når grupperne mangler en node. Der er dog også en væsentlig tidsforøgelse fra 3 til 5 medlemmer i de grupper, der ikke mangler en node. Denne skyldes delvist den øgede kommunikation. Desuden tager operationen længere tid, fordi samlige noder med undtagelse af en enkelt, afvikles på samme maskine. Dette gør sig især gældende ved operationerne med 5 gruppemedlemmer.

Integritetscheck-operationen er i sig selv forholdsvis simpel. Den kompliceres dog, når fejl bliver fundet, som skal rettes. Især hvis en share-gendannelse skal sættes i gang, da denne er af samme størrelsesorden som opdateringsrutinen.

Resultaterne ved afvikling af integritetscheck rutinen var:
Gruppe m. 2 noder (k=2): 6,3 sek. 5,5 sek.
Gruppe m. 3 noder (k=2): 8,1 sek. 7,7 sek.
Gruppe m. 3 noder, hvor 1 node mangler (k=2): 46,8 sek. 50,3 sek.
Gruppe m. 3 noder, hvor 1 node har et forkert share (k=2): 39,0 sek. 36,7 sek.
Gruppe m. 5 noder (k=3): 16,5 sek. 13,8 sek.
Gruppe m. 5 noder, hvor 1 node mangler (k=3): 76,8 sek. 78,6 sek.
Gruppe m. 5 noder, hvor 1 node mangler og 1 node har et forkert share (k=3): 307 sek. 313 sek.

Igen ses det på resultaterne, at manglende medlemmer øger den tid, som operationen er om at afvikle. Desuden er gendannelsesoperationen tidskrævende. Yderligere viser resultaterne, at operationen ikke er særlig tidskrævende, medmindre systemet oplever forskellige fejl.

Generelt er vedligeholdelsesoperationernes tidsforbrug mærkbart, men slet ikke kritisk. Tidsforbruget vil således ikke forhindre Resilia i at virke efter hensigten i praksis. Systemets reaktion på en skalering af antallet af brugere er dog stadig ukendt. I praksis er det dog heller ikke forventet, at grupperne skal være særlig store, da mere end 20 kopier af backupdata virker overdrevet i stort set alle sammenhænge.

6.4 SIKKERHEDSTEST

Dette afsnit beskriver evalueringen af Resilias sikkerhed. Først evalueres sikkerheden generelt. Derefter beskrives kendte sikkerhedshuller i den nuværende prototype.

6.4.1 Generel evaluering af sikkerheden

Da Resilias hovedformål er at opnå sikkerhed og pålidelighed, er det oplagt, at evaluere sikkerheden i det udarbejdede system. Dette er dog ingen let opgave, da en test aldrig vil kunne bevise systemets sikkerhed og da et enkelt sikkerhedshul i værste fald kan give modstanderen en mulighed for at få mere information ud af noderne, end oprindeligt tiltænkt, sådan at systemets fortrolighed brydes. Alternativt vil et sikkerhedshul kunne give modstanderen mulighed for at ødelægge en backup's integritet, på en måde, så systemet ikke længere kan genskabe data. Begge disse udfald er kritiske og må ikke kunne forekomme.

Den mest oplagte måde, hvorpå sikkerheden kan testes, er ved at forsøge at bryde den. Oftest vil dette blive gjort ved at finde fejl eller mangler i systemets protokol og øvrige design, da dette er systemets grundlag. Det kan dog heller ikke udelukkes, at sikkerheden kan brydes som følge af fejl eller mangler i implementeringen. Det er naturligvis endnu sværere at finde disse fejl, når man selv har designet systemet, da fejl naturligvis ville blive rettet, hvis de var kendte. Derfor er det ofte en god ide at lade andre gennemgå systemets protokoller og lede efter svagheder. Dette kan f.eks. foregå vha. en formel analyse af protokollerne. Desuden kan man forsøge med diverse angreb, såsom modificeringsangreb eller man-in-the-middle-angreb. I Resilia er disse angreb dog håndteret af kryptografien og bør således ikke kunne lykkes, medmindre denne brydes eller

simple fejl opdages. Replay-angreb bør ligeledes være yderst svære at gennemføre, pga. håndteringen beskrevet i afsnit 4.4.2.

En af de mest oplagte metoder til at finde sikkerhedshuller i Resilia er derfor, at forsøge forskellige injektionsangreb fra en autentificeret node i systemet. Dette skyldes, at det er antaget, at noder kan kompromitteres, hvorved modstanderen kan få kontrol over autentificerede noder, som siden hen kan bruges til at sende forskellige pakker til resten af noderne. Pakkerne vil have en gyldig signatur (så længe systemet ikke har opdaget, at noden er kompromitteret) og giver således en oplagt mulighed for angreb, da kryptografien ikke kan beskytte mod disse. Dette betyder, at systemets protokoller og design er det eneste forsvar. Hvis protokollerne indeholder fejl eller mangler, vil sådanne angreb have gode chancer for at finde dem. Nogle af de ikke-implementerede special-cases vil f.eks. kunne give anledning til denne type fejl.

Pga. projektets tidsmæssige rammer, er Resilia ikke blevet udsat for angreb af denne type.

6.4.2 Kendte sikkerhedshuller

Hvis et sikkerheds hul opdages, skal det lukkes. Medmindre hullet er så kritisk, at større dele af systemets design skal revurderes, gøres dette vha. nogle special-cases, som er tilføjelser eller rettelser til de eksisterende protokoller. Da nogle sådanne sikkerhedshuller er fundet men ikke rettet pga. projektets tidsmæssige rammer, er det sårbarheder i den nuværende version af Resilia, som naturligvis bør rettes, før en eventuel kommerciel version. Der er dog ikke fundet sikkerhedshuller, som kræver større ændringer i det nuværende design.

De identificerede sikkerhedshuller handler om en kompromitteret node, som forsøger at ødelægge systemet ved at angribe systemets protokoller. Angreb på protokollerne kan deles i to kategorier: Den første kategori er injektionsangreb, som beskrevet ovenfor, hvor den kompromitterede node sender pakker, der ikke passer ind i protokollerne. Den anden kategori er angreb i form af manglende svar på udvalgte tidspunkter. Umiddelbare lyder dette simpelt, men det kan give systemet store problemer, da man ikke kan bevise, om den tavse node er kompromitteret, fejlet eller bare langsom.

De kendte mangler, som vil kunne rettes med diverse special-cases for at øge systemets sikkerhed er følgende:

- Ved opdatering eller genskabelse af shares findes et muligt problem: En kompromitteret node undlader at sende et share til alle. Dermed vil nogle noder tro, at noden opfører sig korrekt, mens andre ikke kan bevise, at dette ikke er tilfældet. Løsningen er skitseret i afsnit 4.5.6 og skitseret i implementeringen, der dog hverken er færdiggjort eller testet.
- Generelt mangler beskyldningsprotokoller (*accusations*), som kan hjælpe systemet med at afgøre, om en node er kompromitteret. Der kan findes mange af disse, og det bør overvejes, hvor store forseelser, der skal til, før en node erklæres for kompromitteret af resten af systemet, da det kan være besværligt for brugeren at skulle skaffe et nyt certifikat. Ved mindre forseelser, hvor noden muligvis er i uorden og ikke kompromitteret af en modstander, kan man vælge at lade noden være udelukket i et begrænset tidsrum, sådan at brugeren undgår certifikatfornyelser.

- Til brug for beskyldninger under vedligeholdelsesoperationerne, skal nodene kunne skeldne mellem nuværende og tidligere operationer, da korrekt men gammelt data ellers vil kunne bruges imod protokollen. Dette kan f.eks. gøres ved, at alle noder tæller, hvor mange gange operationen tidligere er udført. Tælleren kan dermed inkluderes i protokollens pakker, før data signeres, sådan at snyd umuliggøres.

Det skal understreges, at der muligvis findes flere af disse sikkerhedshuller, så protokollerne bør gennemtestes grundigt for de to typer angreb, før en endelig version er klar.

6.5 EVALUERING AF JXTA-SYSTEMET

Da JXTA er grundstenen for Resilias kommunikation, er det afgørende at systemet virker efter hensigten. Det var kendt på forhånd, at JXTA ville være svært at arbejde med pga. den ringe dokumentation og da det stadig er under gennemgribende udvikling, hvorfor visse dele måske ikke virker som forventet. Derfor er brugen af JXTA minimeret i Resilia, sådan at systemet stort set kun varetager de absolutte nødvendige opgaver, dvs. peer-discovery og kommunikation mellem noderne. I prototypen af Resilia har JXTA dog også fået til opgave at styre gruppeinddelingen af noder, sådan at flere forskellige backupgrupper kan eksistere samtidig.

JXTA's relay-noder og rendezvous-noderer er vigtige elementer for systemet, da disse definerer infrastrukturen. Relay-noder er essentielle, da de muliggør, at noder kan kommunikere med hinanden på internettet til trods for firewalls og NATs. Ideen er således, at kommunikationen automatisk videresendes vha. JXTAs protokoller (samt relay- og rendezvousnoder), uden at man skal bekymre sig om, hvordan noderne egentlig kommunikerer og finder hinanden.

Selvom grundideen er god, har erfaringerne fra dette projekt ikke været udelukkende positive. I den nuværende version fungerer Resilia fint på et lokalnetværk, hvor noder både kan finde diverse backupgrupper og hinanden. Kommunikationen mellem noderne sendes og modtages uden problemer og JXTA-systemet virker således efter hensigten i dette miljø.

Afprøvningen på internettet har ikke været lige så succesfuld. Ved den første afprøvning fandt noderne hverken hinanden eller de oprettede backupgrupper. Det skal dog bemærkes, at et test-miljø med to noder uden NAT og firewall ikke har været tilgængeligt. Derfor er afprøvningen på internettet udelukkende foregået med brug af relay-noder, hvilket komplicerer opsætningen og stiller højere krav til JXTAs funktionalitet. Det kan således ikke udelukkes, at problemerne helt eller delvist skyldes relay-noderne eller brugen af disse.

I forbindelse med afprøvningen viste det sig, at JXTAs peergruppe koncept muligvis var medvirkende til fejlene. Noderne var således ikke i stand til at finde nogen advertisements, som var publiceret i en oprettet JXTA-peergruppe (hvilket er tilfældet med alle Resilias advertisements), hvorimod disse blev fundet, hvis de var publiceret i JXTAs standardgruppe (som alle noder automatisk tilslutter sig, når de starter JXTA). Årsagen hertil er stadig ikke kendt. Dog er det muligt at rendezvous-noderne har noget med problemet at gøre. I den

benyttede version af JXTA (version 2.0) afbryder rendezvous-noderne tilsyneladende kontakten med andre noder, når de bliver medlem af en ikke-standard peergruppe.

I et forsøg på at omgå fejlene, forsøgtes Resilias brug af JXTA midlertidigt ændret til at undlade brugen af peergrupper. Dette betyder naturligvis en væsentlig forringelse af designet og vil ikke være ønskværdigt i en evt. kommerciel version. Ændringerne havde umiddelbart den ønskede effekt og noderne kunne med dette test-design finde hinanden og sende kommunikationspakker til hinanden. Filoverførsler lader dog ikke til at kunne lade sig gøre. Da adresseinformationen ved filoverførslerne er korrekt, lader problemet til at skyldes relay-noderne eller brugen af disse.

Problemet er ikke undersøgt yderligere i dette projekt pga. de tidsmæssige rammer. Før en eventuel kommerciel version af Resilia, er det således væsentligt, at JXTA-systemet kommer til at virke helt efter hensigten - også over internettet med firewalls og NATs. Derfor bør det undersøges, om der findes løsninger på det beskrevne problem. I denne sammenhæng kan det overvejes at opgradere til en nyere version af JXTA, da der hele tiden bliver rettet fejl i systemet. Alternativt kan det udskiftes med en anden peer-to-peer system. Dette vil være forholdsvist let i Resilia, da brugen af JXTA-systemet i Resilia er minimeret.

6.6 OPSUMMERING

Dette kapitel har beskrevet evalueringen af Resilia.

Resilias funktionalitet er afprøvet og ydelsen er undersøgt i de grundlæggende operationer. Desuden er systemets sikkerhed blevet evalueret. Yderligere er JXTA-systemet evalueret separat.

Afprøvningen af funktionaliteten gav ingen alvorlige fejl og sandsynliggør således, at den grundlæggende funktionalitet virker efter hensigten. Det er dog ikke muligt at udtale sig om systemets robusthed på denne baggrund.

Ydelsestesten viste, at systemets grundlæggende operationer ikke er så tidskrævende, at det vil give problemer i praksis. Man må dog forvente, at disse tider vil øges, når systemet skal fungere på internettet. Ud fra ydelsestesten er det dog forventet, at båndbredden mellem brugere ofte vil være den største flaskehals.

Evaluering af systemets sikkerhed konkluderede, at sikkerheden primært ligger i systemets protokoller. Sikkerheden kan ikke garanteres, men hvis protokollerne analyseres og testes grundigere, kan større tiltro til sikkerheden opbygges (og eventuelle fejl kan blive rettet). Desuden er nuværende kendte sikkerhedshuller beskrevet.

JXTA er stadig under udvikling og mange dele af systemet virker endnu ikke optimalt. Desuden er dokumentationen yderst mangelfuld, hvilket stadig gør systemet svært at arbejde med. Resilias brug af JXTA-systemet virker efter hensigten på et lokalnetværk men ikke gennem firewall og NAT, selvom JXTA's relay-noder benyttes.

7 KONKLUSION

I dette projekt er det undersøgt, hvordan et sikkert og pålideligt backupsystem kan laves ved en kombinationen af peer-to-peer teknologi og secret sharing kryptografi. Projektets hovedformål har således været at finde frem til en løsning, hvor både fortrolighed, integritet og tilgængelighed er sikret. Desuden har løsningens varighed været i fokus, idet hverken fortroligheden, integriteten eller tilgængeligheden må mindskes med tiden til trods for fejl og aktive angreb.

Resultatet af arbejdet er det distribuerede backupsystem Resilia, hvor n noder sammen sikrer fortrolighed, integritet og tilgængelighed af en backup. En grundlæggende model for systemet er opstillet og analyseret på baggrund af de grundlæggende teknologier. Trusler og risici ved modellen er ligeledes analyseret. Desuden er systemets design beskrevet og en prototype er implementeret og evalueret.

Prototypen af Resilia kan demonstrere systemets funktionalitet, men skal ikke betragtes som et færdigudviklet produkt. Formålet med prototypen er således at vise, at det kan lade sig gøre, at lave et system, der løser den oprindelige problemstilling, sådan at både fortrolighed, integritet og tilgængelighed sikres. Systemet vedligeholdes over tid vha. verificerbart og proaktivt secret sharing, sådan at systemets pålidelighed opretholdes og sikkerheden ikke mindskes mere end den altid aftagende sikkerhed på den tilgrundliggende kryptografi. Peer-to-peer arkitekturen afhjælper *single point of failure*-problematikken og er med til at sikre systemets robusthed ved fejl og aktive angreb eller skift i netværkstopologien. Der har dog været problemer med brugen af JXTA som peer-to-peer platform, hvorfor videre arbejde er nødvendigt, før systemet kan bruges over firewall og NAT. Evalueringen af Resilia antyder, at systemet i øvrigt virker efter hensigten. Resilias ydelse er brugbar og vil i de fleste tilfælde afhænge primært af de datamængder, der skal sendes mellem noderne under distribueringen af en backup og kun i ringe grad af Resilias administrerende kommunikation.

Resilia opfylder således projektets primære målsætninger. Så længe mindst k ud af n ($2k\text{-}1 \leq n$) noder er korrekte (dvs. følger systemets protokol og har umodificerede data) og mindre end k noder er kompromitterede af en modstander, kan systemet opretholde fortrolighed, integritet og tilgængelighed af en backup. Også ved et fuldkomment datatab på optil $n\text{-}k$ noder, hvor både data og krypteringsnøgler mistes, kan noderne genoprettes, sådan at systemet som helhed ikke svækkes. Udover den proaktive secret sharing er dette et resultat af brugen af certifikater til autentificering.

7.1 VIDERE ARBEJDE

Dette afsnit beskriver nogle oplagte områder, hvor Resilia kan forbedres i fremtiden.

7.1.1 Fordeling af data i systemet

Da Resilias ydelse i høj grad afhænger af mængden af backup-data, der skal fordeles i systemet, er det oplagt at undersøge, om dette kan gøres på en smartere måde. Især er Rabins informationsspredningsalgoritme interessant, da denne kan mindske systemets datamængde markant. Det skal dog først undersøges, om denne kan bruges i forbindelse med de samme proaktive vedligeholdelsesprocedurer, som er brugt i Resilia. Her tænkes især på gendannelsen af mistede shares, da systemet ellers vil miste sin varige tilgængelighed. Hvis ikke dette lykkes, kan man alternativt vælge, at mindske kravene til vedligeholdelse i systemet, hvis man mener, at pladsforbrug og båndbreddeforbrug er vigtigere. Dette vil naturligvis afhænge af, hvad systemet skal bruges til, og i dette projekt er den proaktive vedligeholdelse således vurderet som værende vigtigere.

7.1.2 Secret sharing modellen

Resilias brug af secret sharing kan udvides for at øge brugbarheden af systemet. Her tænkes især på indførsel af vægtning af shares, sådan at brugeren ikke nødvendigvis behøver at stole lige meget på alle noder. Da dette afspejler den virkelige verden, hvor vi heller ikke stoler lige meget på alle, vil det være en feature, der er yderst brugbar. Desuden vil det være forholdsvist let at implementere, da det ikke kræver store grundlæggende ændringer. En anden måde at videreudvikle secret sharing modellen kunne være, at indføre mulighed for konfigurationsskift i systemet. Dette vil kunne gøre Resilia endnu mere robust på langt sigt, da backupgrupper herved ikke skal beholde den samme størrelsesorden under hele backup'ens levetid. I den nuværende version, vil der opstå et problem i en (n,k)-konfiguration, hvis n-k medlemmer vælger, at de ikke længere ønsker at være en del af systemet. De resterende k medlemmer, vil således ikke længere kunne tolerere fejl, da systemets grundantagelser dermed vil være brudt. Her vil et konfigurationsskift f.eks. være nyttigt.

7.1.3 Øvrigt arbejde før Resilia tages i brug

Resilias funktionalitet bør øges, før en evt. kommerciel version. Her tænkes bl.a. på en bedre filhåndtering og flere indstillingsmuligheder for brugeren, såsom valgfri krypteringsalgoritmer og nøglestørrelser mm. I samme forbindelse kan brugervenligheden øges, sådan at systemet bliver mere intuitivt at bruge.

Før Resilia kan bruges over firewall og NAT skal brugen af JXTA-systemet (og muligvis JXTA-systemet selv) videreudvikles. Alternativ kan JXTA udskiftes helt, hvis en anden peer-to-peer platform foretrækkes, eller hvis man vælger at bygge funktionaliteten op fra grunden.

En implementering af elliptisk kurve kryptografi bør overvejes til at beskytte verificeringen af shares i stedet for det nuværende logaritmeproblem, da man herved kan øge applikationens ydelse ved at bruge mindre nøgler med samme anslåede sikkerhed.

Resilias protokoller skal færdigudvikles, sådan at alle kendte sikkerhedshuller eller funktionelle mangler fjernes. De beskrevne problemstillinger i afsnit 6.2.3 og 6.4.2 bør således overvejes. Desuden bør protokollen testes grundigere og analyseres med henblik på robusthed og sikkerhed. Dette vil være et stort arbejde og kræve meget tid, men er nødvendigt, hvis man skal være tilstrækkelig sikker på, at systemet er sikkert nok.

8 LITTERATUR

[1] Bruce Schneier, *Applied Cryptography*, second edition, John Wiley & Sons, Inc., 1996.

[2] Douglas R. Stinson, *Cryptography: Theory and Practice*, second edition, Chapman & Hall/CRC, 2002.

[3] C. P. Pfleeger, S. L. Pfleeger, *Security in Computing*, third edition, Prentice Hall, 2003.

[4] Joseph D. Gradecki, *Mastering JXTA: Building Java Peer-to-Peer Applications*, John Wiley & Sons, Inc., 2002.

[5] D. Brookshier, D. Govoni, N. Krishnan, *JXTA: Java P2P Programming*, Sams Publishing, 2002.

[6] M. O. Rabin, *Efficient dispersal of information for security, load balancing, and fault tolerance*, Journal of the ACM, 36(2):335-348, April 1989.

[7] A. Shamir, *How to share a secret,* Communications of the ACM, 22(11): 612-613, 1979.

[8] G. R. Blakley, *Safeguarding cryptographic keys*, Proceedings AFIPS 1979 National Computer Conference, pp. 313-317.

[9] P. Feldman. *A Practical Scheme for Non-interactive Verifable Secret Sharing*, In Proceedings of the 28th IEEE Symposium on the Foundations of Computer Science (FOCS), pages 427-437, 1987.

[10] T. P. Pedersen, *Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing*, Proc. Crypto'91 (LNCS 576), pp 129-140.

[11] A. Herzberg, S. Jarecki, H. Krawczyk and M. Yung, *Proactive secret sharing or: how to cope with perpetual leakage*, Proceedings of Crypto'95, LNCS 963, pp. 339-352, 1995.

[12] L. Zhou, Z. J. Haas, *Securing ad hoc networks*, IEEE Network Magazine, 13(6):24--30, November/December 1999.

Web referencer

D. Brookshier, *The Socket API in JXTA 2.0*:
http://java.sun.com/developer/technicalArticles/Networking/jxta2.0/ (Mar-2004)

Java keytool:
http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/keytool.html (Feb-2004)

Certifikatautoriteter/OpenSSL/Java keytool:
http://www.devx.com/Java/Article/10185/0/page/ (Feb-2004)

JXTA:
http://www.jxta.org

Java:
http://java.sun.com

OpenSSL:
http://www.openssl.org

9 APPENDIKS A - BRUGERVEJLEDNING

Programmet udviklet i forbindelse med dette projekt har fået navnet Resilia. Dette afsnit vil beskrive installationen og brugen af Resilia. Først forklares indholdet af den vedlagte program-cd.

9.1 INDHOLD AF PROGRAM CD

Den vedlagte cd indeholder følgende 6 mapper:

- Resilia
- ResiliaSrc
- Certificates
- OpenSSL
- BouncyCastle
- JavaJREInstall

Mappen *Resilia* indeholder den kompilerede kode af Resilia (placeret i filen Resilia.jar). Derudover findes mappen ”extlib”, som indeholder de benyttede JXTA-filer. *Resilia*-mappen indeholder desuden to certifikater og en keystore-fil (peer.keystore) samt tre mapper, som bruges af Resilia (encryptedFiles, restoredFiles og mySharings). Efter installationsvejledningen herunder er gennemført, vil denne mappe således indeholde alt, hvad der er nødvendigt, for at starte Resilia. Mappen skal dog kopieres til den lokale harddisk, før programmet kan afvikles. Når man befinder sig i mappen *Resilia* (på den lokale harddisk) og installationen i afsnit 9.2 er gennemført, kan programmet startes ved kommandoen:

```
java -classpath Resilia.jar Start
```

(På nogle systemer kan man blot dobbeltklikke på filen Resilia.jar i stedet.)
Det certifikat og nøglepar, som findes i Resilia-mappen har password’et: ”12341234”.

Mappen *ResiliaSrc* indeholder kildekoden til programmet. Denne er delt op i programmets 6 pakker. Desuden findes kildekoden til programmet primes.java, som blev beskrevet i afsnit 5.5. Kildekoden til Resilia kan kompileres vha. filen compile.bat. Denne findes ligeledes i *Resilia*-mappen. Kildekoden er bl.a dokumenteret vha. Javadoc.

Certificates-mappen indeholder 3 forskellige brugeres certifikater (både eget certifikat og certifikatautoritetens). Desuden findes tilhørende keystore-filer, hvor de tilhørende private nøgler er gemt. Det tilhørende password er: ”12341234” til alle tre certifikater. Certifikaterne er vedlagt for at give mulighed for at prøve Resilia med flere noder, uden først at skulle oprette en certifikatautoritet. Certifikaterne (og keystore-filerne) kan således blot udskiftes med dem fra *Resilia*-mappen, hvis et andet certifikat ønskes benyttet.

Mappen *OpenSSL* indeholder installationsfilen til openSSL til Windows. For andre versioner henvises til openSSL's hjemmeside (http://www.openssl.org/). Desuden findes en mappestruktur (hvor den øverste mappe hedder DemoCA), som skal bruges, hvis man vil oprette sin egen certifikatautoritet. Dette er beskrevet yderligere i afsnit 9.3.

Mappen *BouncyCastle* indeholder filen `bcprov-jdk14-120.jar`. Denne skal benyttes sammen med Resilia til at udføre kryptografiske operationer. Installationen af Bouncy Castle er beskrevet i afsnit 9.2.

Mappen *JavaJREInstall* indeholder filen `j2jre-1_4_2_05.exe`. Denne installerer Java JRE 1.4.2 på en Windows-maskine. Java kan også hentes fra Suns hjemmeside (http://java.sun.com/).

9.2 INSTALLATIONSVEJLEDNING

Før Resilia kan benyttes, skal flere forskellige komponenter installeres. De påkrævede installationsfiler kan findes på den vedlagte cd, som er beskrevet i forrige afsnit. Denne installationsvejledning er tilrettet Windows XP og vil muligvis skulle justeres, hvis programmet installeres på en anden platform. Det er dog de samme komponenter, der skal installeres på alle platforme. Det bør bemærkes, at der kan opstå problemer, hvis der benyttes forskellige softwareversioner på forskellige Resilia-noder.

Java
Da Resilia er et Java-program, er brugeren nødt til at installere Java Virtual Machine (JVM), før systemet kan bruges. Java JRE 1.4.2 er benyttet til udviklingen, og det anbefales derfor at benytte denne eller en nyere version. Installationsfilen (`j2jre-1_4_2_05.exe`) findes på den vedlagte cd under mappen *JavaJREInstall*. Filen køres og vejledningen følges for at gennemføre installationen.
Resilia skal have adgang til Javas keytool program. Dette gøres ved at tilføje stien `C:\JAVAPATH\j2re1.4.2_05\bin` til systemets sti (path), hvor `JAVAPATH` er stien til java-installationen (f.eks. `c:\Program Files\Java\`).

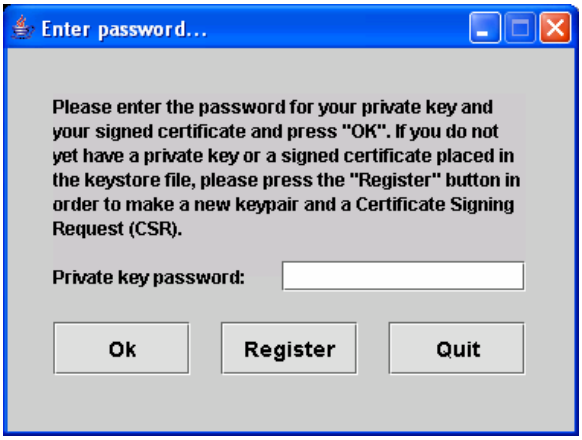
Bouncy Castle
Bouncy Castle skal installeres før systemet kan bruge de forskellige krypteringsalgoritmer. Dette gøres ved at kopiere filen `bcprov-jdk14-120.jar` fra program-cd'ens *BouncyCastle*-mappe til mappen `\lib\ext` i java-biblioteket (den fulde sti vil derfor typisk være noget i retning af `C:\Program Files\Java\j2re1.4.2_05\lib\ext`). Desuden åbnes filen `\lib\security\java.security` og linien:
"`security.provider.6=org.bouncycastle.jce.provider.BouncyCastleProvider`"
føjes til listen over security-providere, som findes lidt over midten af filen.

Efter ovenstående er gennemført, kan Resilia køres. Dette gøres ved at kopiere *Resilia*-mappen fra program-cd'en til et ønsket sted på harddisken og fra Resilia-mappen (i en terminal) skrive kommandoen:
```
java -classpath Resilia.jar Start
```
(På nogle systemer kan man blot dobbeltklikke på filen Resilia.jar i stedet.)

9.3 FØRSTE OPSTART

Når Resilia startes skal programmet kende til brugeren nøglepar og certifikat. Første gang programmet køres, er brugeren sandsynligvis ikke i besiddelse af disse, som derfor skal oprettes.

Ved opstart af Resilia vises følgende vindue:



Hvis brugeren har brug for et nyt certifikat og nøglepar trykkes på "Register". Herefter vil følgende vindue fremkomme:

Her indtastes informationen og der trykkes "Ok". Det indtastede password skal huskes, da dette skal bruges til at logge på systemet senere. Programmet vil nu lukke ned. Samtidig oprettes dog et nøglepar, som gemmes i filen peer.keystore. Desuden laves en Certificate Signing Request (CSR) i filen peer.csr. Begge filer er placeret i *Resilia*-mappen. Filen peer.csr sendes til Certifikatautoriteten, som svarer ved at udstede et certifikat (såfremt den rette autentificering er foregået).

Efterfølgende beskrives, hvordan man kan oprette sin egen certifikatautoritet vha. OpenSSL. Certifikatautoriteten kan oprette certifikater, der kan bruges sammen med Resilia.

Certifikatautoritet med OpenSSL
I denne prototype af Resilia er certifikater lavet vha. OpenSSL. Ved at eksekvere installationsfilen fra program-cd'en og følge vejledningen installeres OpenSSL. Desuden skal `openssl\bin\`-mappen tilføjes til systemets sti (*path*). I mappen `openssl\bin\` findes filen `openssl.cnf` (den er sandsynligvis skjult, men den eksisterer). Denne editeres og linien, hvor der står
`policy = policy_match` ændres til `policy = policy_anything`.

Herefter er OpenSSL klar til brug, og der skal nu laves en certifikatautoritet. Først laves følgende mappestruktur et vilkårligt sted på harddisken: (Mappestrukturen er lavet på forhånd under *OpenSSL*-mappen på program-cd'en og kan således blot kopieres over på harddisken)

```
demoCA            (mappe)
demoCA/private    (mappe)
demoCA/newcerts   (mappe)
demoCA/index.txt  (fil)
demoCA/serial     (fil)
```

Filen index.txt skal være tom, og filen serial skal indeholde strengen '01' (uden linieskift og uden '').

Certifikatautoriteten skal nu danne sit selv-signerede rod-certifikat som skal bruges til at udstede certifikaterne. Dette gøres vha. følgende kommando i en terminal fra demoCA-mappen (f.eks. fra C:\demoCA, hvis demoCA er placeret her):

```
C:\demoCA>openssl req -x509 -newkey rsa:2048 -keyout private/cakey.pem -out
cacert.pem
```

Herefter spørger programmet efter et password, som certifikatautoriteten siden hen skal bruge, når den signerer certifikater. Desuden skal information om certifikatet indtastes (navn, land, e-mail osv.). Derved fuldføres operationen og certifikatautoritetens rod-certifikat findes nu i filen `demoCA\cacert.pem`, mens den private nøgle findes i filen `demoCA\private\cakey.pem`. Dermed er certifikatautoriten klar til at udstede certifikater.

Når certifikatautoriteten skal udstede et certifikat, modtages en Certificate Signing Request (CSR) i form af filen `peer.csr` fra brugeren, som skal bruge certifikatet. Når certifikatautoriteten har autentificeret brugeren, placeres filen `peer.csr` i demoCA-mappen. Derefter bruges følgende kommando til at lave brugerens certifikat (Bemærk at kommandoen skrives fra C:\ og ikke fra C:\demoCA):

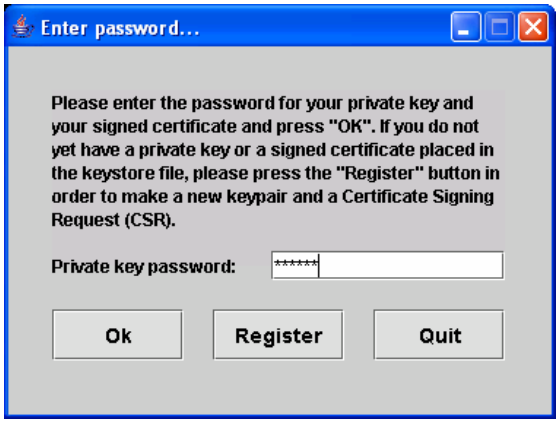
```
C:\>openssl ca -in demoCA/peer.csr -out demoCA/peer.crt -notext
```

(Argumentet `-notext` er nødvendigt for at certifikatet kan importeres af Resilia.)

Nu indtastes certifikatautoritetens password og der svares ja (y) til at udstede certifikatet. Herefter genereres certifikatet (filen `peer.crt`) i demoCA-mappen. Selve brugerens certifikat (`peer.crt`) sendes til brugeren sammen med certifikatautoritetens certifikat (`cacert.pem`). Brugeren placerer begge certifikater i *Resilia*-mappen. Dermed har brugeren et gyldigt certifikat, der kan bruges sammen med Resilia.

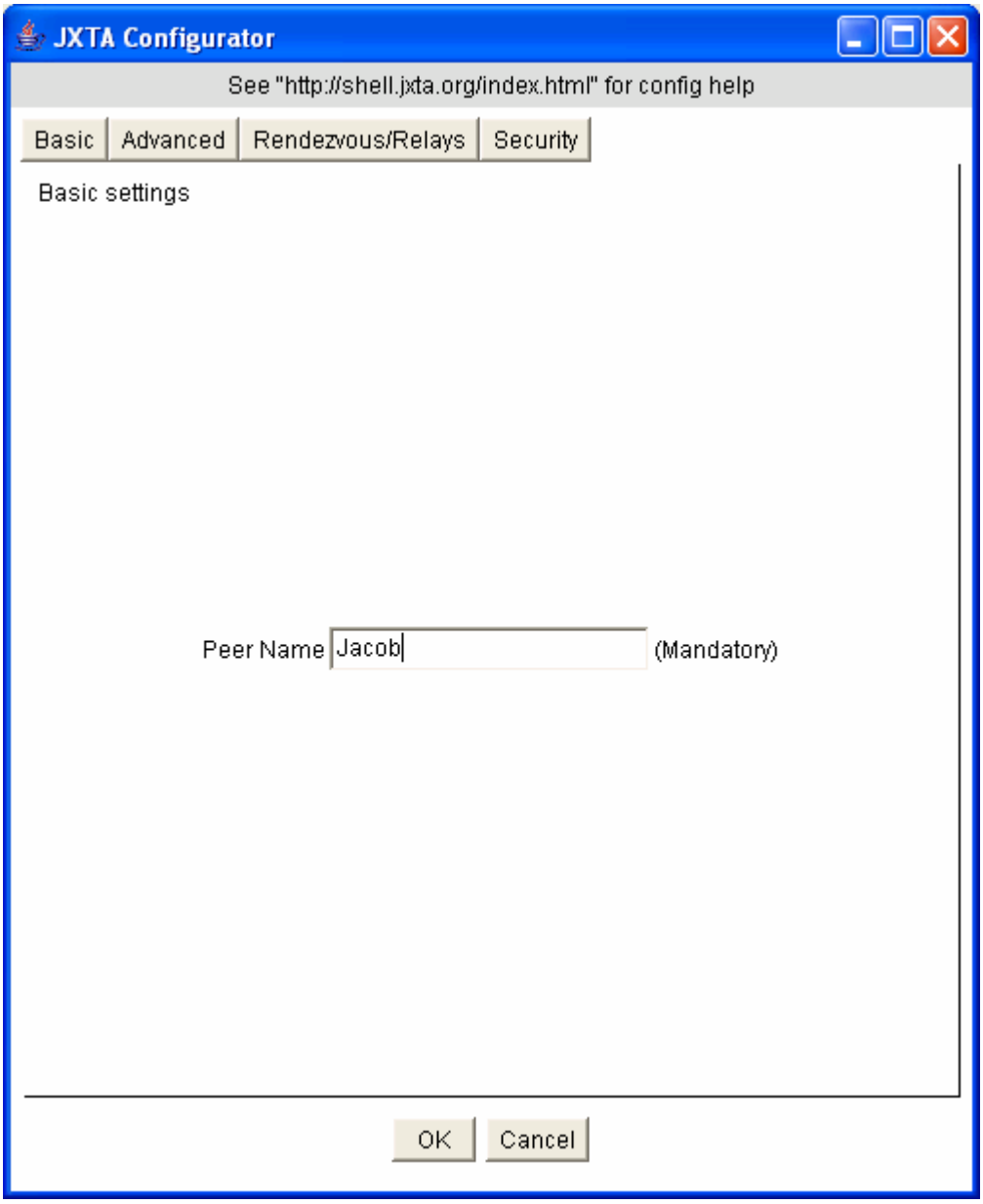
Opstart og konfiguration af JXTA

Nu er det gyldige certifikat oprettet og brugerens nøglepar findes i `peer.keystore`-filen. Dermed kan Resilia atter startes. Ved opstart vil password-vinduet fremkomme igen:



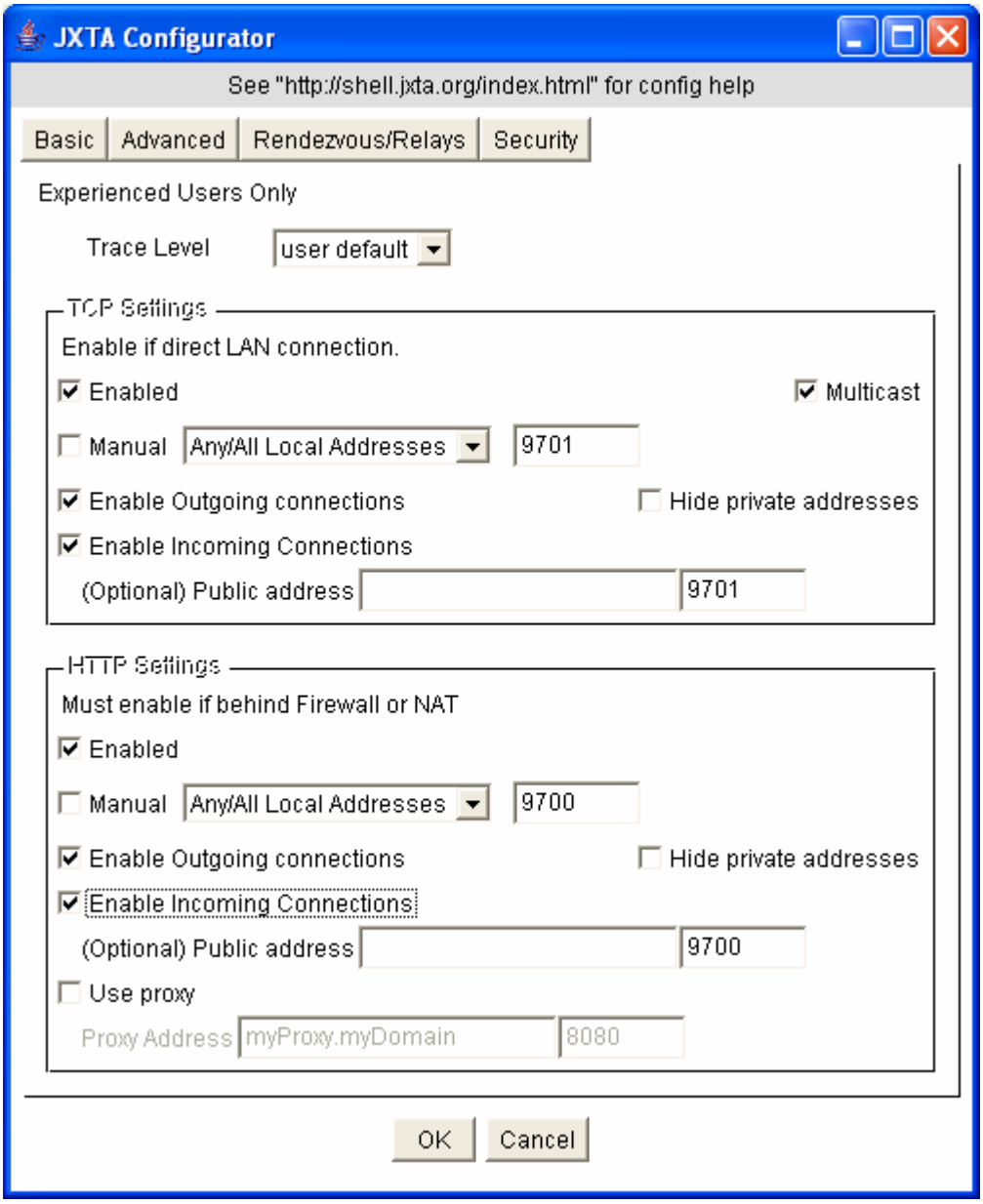
Denne gang skrives det password, som blev dannet, da brugeren indtastede sin information i vindetuet ”Create new certificate” ovenfor (Hvis certifikaterne fra program-cd’en benyttes er password’et: 12341234). Derefter vil programmet starte for første gang. Inden det er muligt at starte programmet, skal JXTA-systemet dog konfigureres. Denne konfiguration skal kun foretages første gang Resilia startes.

Derfor fremkommer JXTA-konfigurationsvinduet, som ser sådan ud:



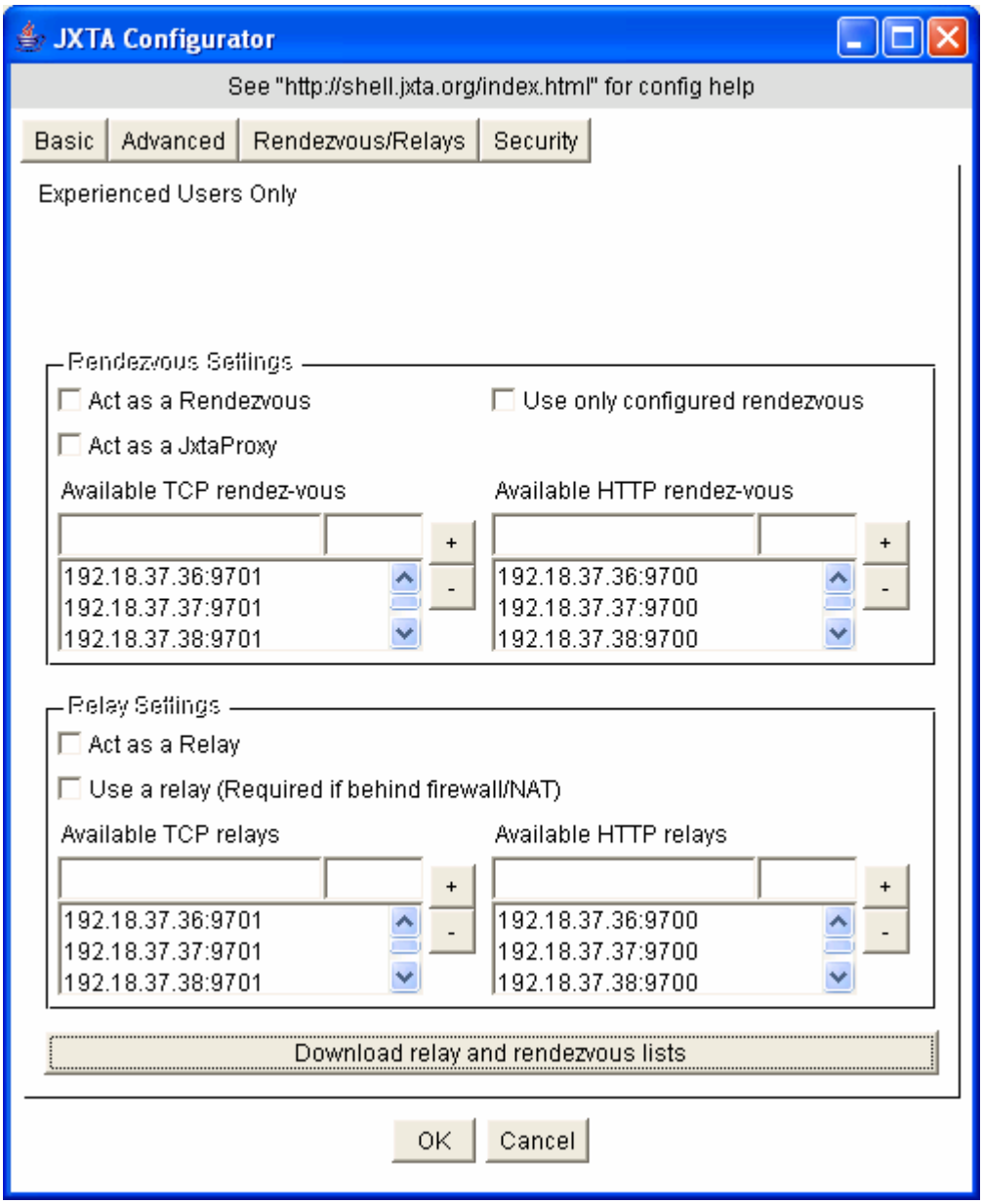
Konfigurationsvinduet har fire faneblade. Det første (*Basic*) er vist herover. Her skal blot indtastes et navn. Dette navn vil blive brugt af Resilia i programmets hovedvindue, jf. afsnit 9.4.2.

Konfigurationsvinduets andet faneblad (*Advanced*) ser ud som følger:



Her vælges hvilken kommunikation, der er tilgængelig for noden. Både TCP og HTTP er muligt (hvis begge slås til skal de dog have forskellige portnumre). Indadgående og/eller udadgående kommunikation skal vælges, så det passer til nodens netværksopsætning. Hvis flere noder skal kommunikere fra samme maskine, skal de ligeledes have forskellige portnumre.

Det tredje faneblad (*Rendezvous/Relays*) ser ud som følger:



Her kan noden vælge at agere rendezvous eller relay. Desuden kan den vælge, om den selv skal benytte andre relay-noder, og hvilke rendezvous-noder, der benyttes. Ved at trykke ”Download relay and rendezvous lists” kan nogle faste JXTA relaynoder og rendezvousnoder findes.

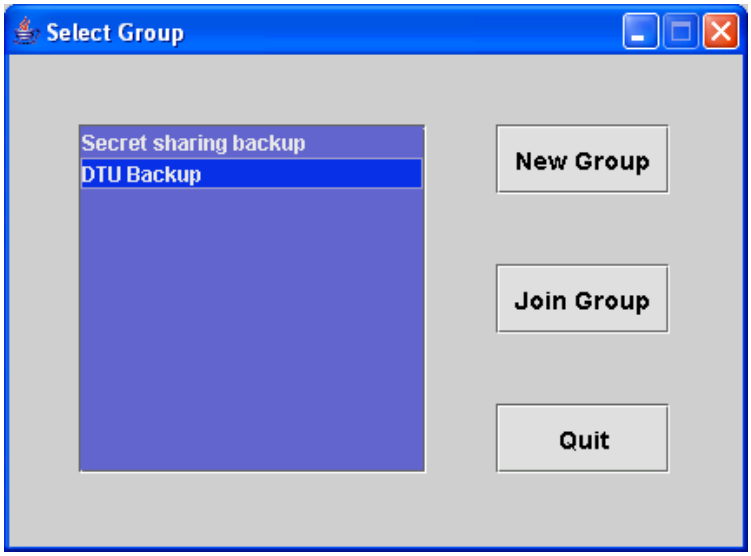
Herefter trykkes ”OK”. JXTA-konfigurationen er nu færdig og Resilia vil starte op. Det sidste faneblad (*security*) skal ikke udfyldes, da dette gøres automatisk af Resilia.

9.4 HOVEDFUNKTIONER

Dette afsnit beskriver brugen af Resilia. Først beskrives indmeldelse i backupgrupper. Derefter beskrives Resilias hovedvindue og dettes funktioner. Det er antaget, at Resilia er installeret som beskrevet i afsnit 9.2 og at et gyldigt certifikat og nøglepar (i filen `peer.keystore`) benyttes, sådan at Resilia er klar til brug.

9.4.1 Indmeldelse i en gruppe

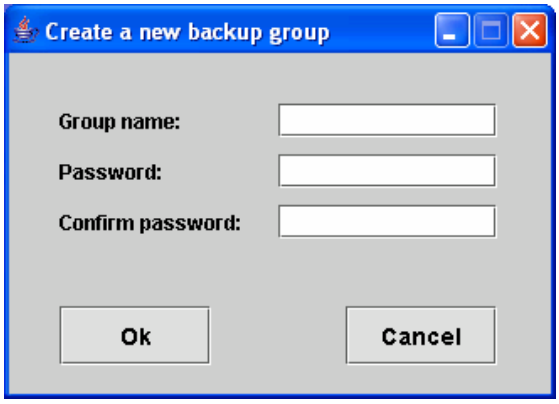
Når Resilia er installeret og konfigureret og når det første password er indtastet som beskrevet ovenfor, vil følgende vindue fremkomme:



Her vises de backupgrupper, som findes i systemet. Listen opdateres løbende, når nye grupper bliver fundet på netværket. P.t. er det således grupperne ”Secret sharing backup” og ”DTU Backup”, som er fundet. Desuden har vinduet følgende tre knapper:

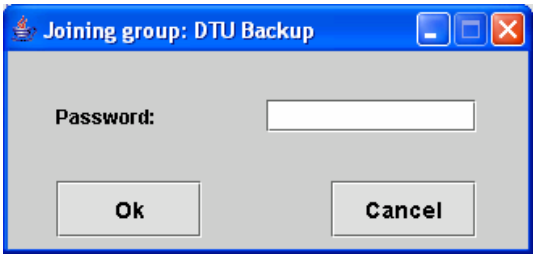
New Group – Starter en dialog for at oprette en ny backupgruppe (beskrevet herunder).
Join Group – Starter en dialog for at (beskrevet herunder).
Quit – Afslutter programmet.

New Group dialogen
Efter New Group-knappen er trykket, vil følgende vindue fremkomme:



For at oprette en ny backup-gruppe skal et navn indtastes. Indtastning af password har ingen betydning i denne version af Resilia. Når informationen er indtastet, trykkes på "Ok"-knappen for at oprette gruppen. Antaget at informationen er indtastet, vil Resilias hovedvindue fremkomme. Ellers kan der trykkes på "Cancel"-knappen, for at komme tilbage.

Join Group dialogen
Hvis ”Join Group”-knappen trykkes, efter en gruppe er markeret, vil følgende vindue fremkomme (I dette tilfælde var gruppen ”DTU Backup” markeret):

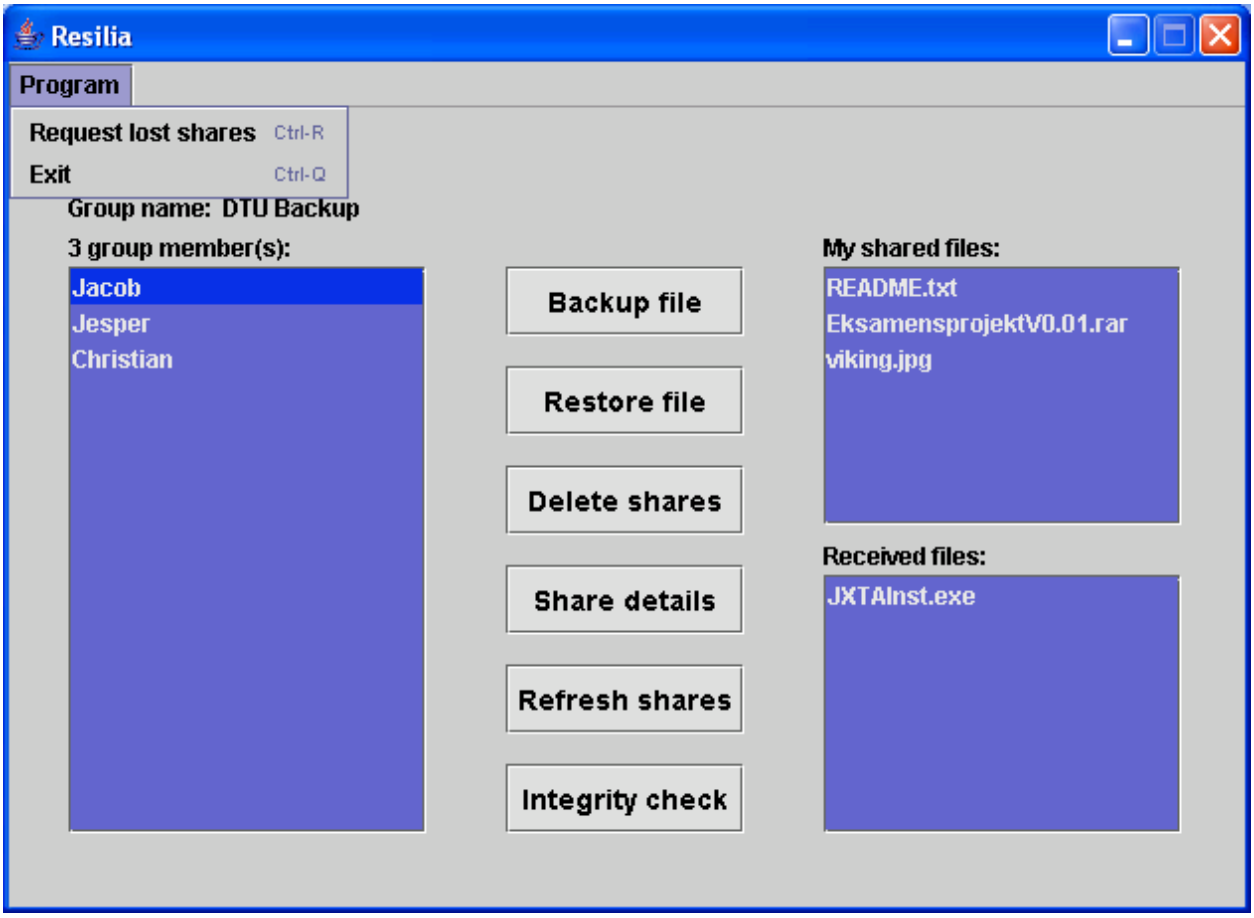


Øverst kan man se, hvilket gruppe man er ved at melde sig ind i. Da gruppe-passwordet i denne version af Resilia er uden betydning, kan man blot trykke ”Ok” for at melde sig ind i gruppen. Herved vil Resilias hovedvindue fremkomme. Ellers kan der trykkes på "Cancel"-knappen, for at komme tilbage.

9.4.2

Resilias hovedvindue

Når man er meldt ind i en gruppe vises Resilias hovedvindue. Dette ser ud som følger:

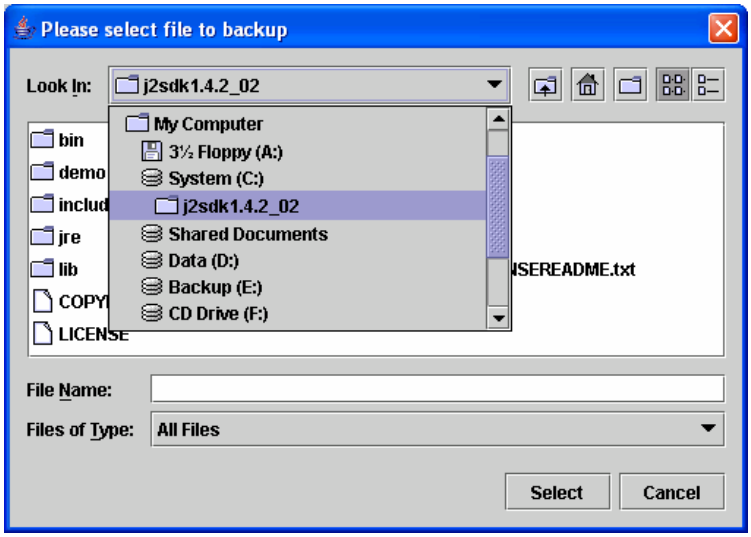


Vinduet består af tre lister. Den første (til venstre) indeholder navnene på gruppemedlemmerne. Brugeren af denne nodes navn står øverst. Den anden liste kaldet ”My shared files” (øverst til højre) indeholder filnavnene på de filer, som denne bruger har taget backup af vha. Resilias secret sharing backup. Den sidste liste kaldet ”Received files” (nederst til højre) indeholder filnavnene på de filer, som andre noder/brugere har taget backup af, hvor denne node er en del af backup’en.

Udover disse lister indeholder Resilias hovedvindue nogle knapper og en simpel menubar, hvorfra programmets funktioner kan tilgås. Disse funktioner gennemgås i det følgende.

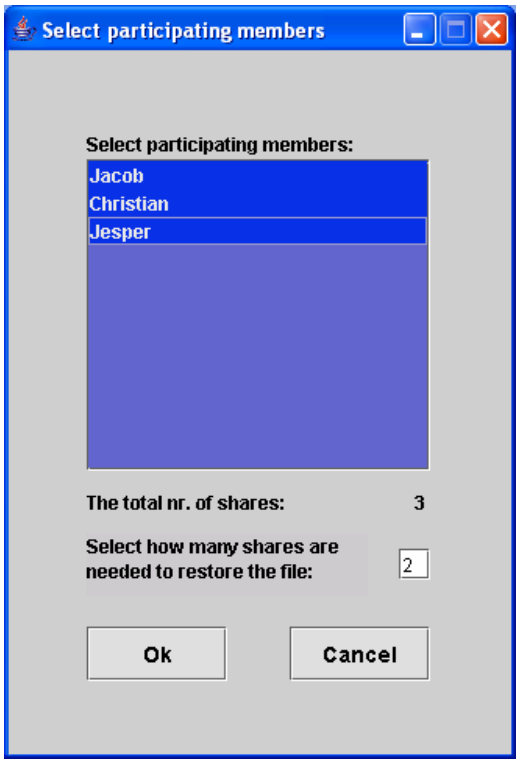
Backup file

Når en backup skal laves, trykkes på knappen ”Backup file”. Derefter vises en standard fil-browser, hvor man kan finde den fil, som man ønsker at tage backup af. Fil-browseren er vist herunder.



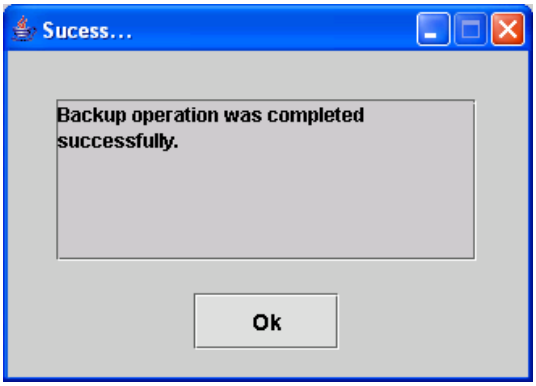
Når den ønskede fil er fundet, trykkes på ”Select” knappen. Operationen kan afbrydes ved at trykke ”Cancel”.

Efter en fil er valgt fremkommer følgende vindue:



Vinduet indeholder listen af eksisterende gruppemedlemmer samt et tekstfelt. I listen af medlemmer, vælges nu de medlemmer, som skal deltage i backup'en (flere medlemmer vælges ved at holde Ctrl-tasten nede). I tekstfeltet skrives et heltal, der definerer, hvor mange shares, der skal til at genskabe backup-filen. Når dette er gjort trykkes på "Ok"-knappen.

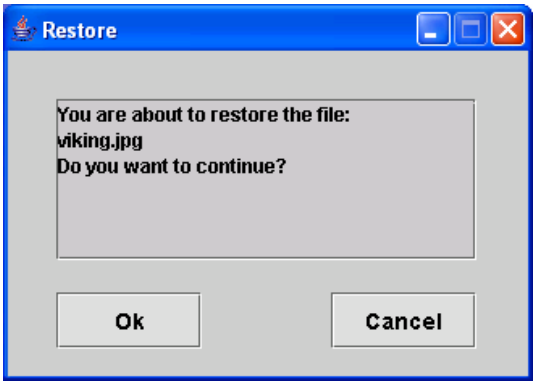
Herefter vil systemet distribuere backup'en. Når dette er gjort, vises en besked:



Desuden vises filenavnet i "My shared files"-listen i Resilias hovedvindue. Hos de deltagende gruppemedlemmer vil filnavnet blive vist i "Received files"-listen.

Backup'en er nu fuldført.

Restore file
Når en fil skal gendannes, vælges denne først i listen "My shared files" i hovedvinduet. Derefter trykkes på "Restore file"-knappen. Følgende vindue vil blive vist:



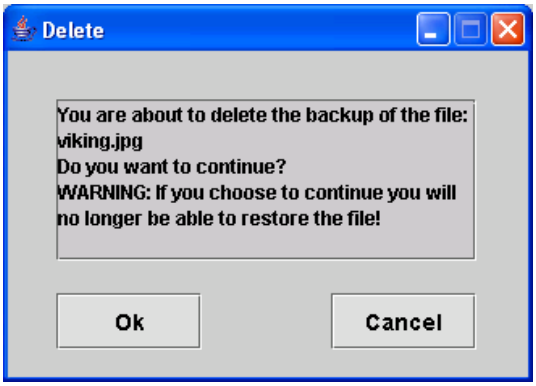
Hvis gendannelsen ønskes foretaget, trykkes "Ok". Ellers trykkes "Cancel".

Efter brugeren trykker "Ok", vil systemet gendanne backup-filen. Denne placeres under mappen "restoredFiles" i Resilia-mappen. Når operationen er færdig, vil følgende vindue blive vist:

Gendannelsen af backup-filen er nu fuldført.

Delete shares

Når en backup skal slettes, vælges denne først i listen ”My shared files” i hovedvinduet. Derefter trykkes på ”Delete shares”-knappen. Følgende vindue vil blive vist:



Hvis backup’en ønskes slettet trykkes ”Ok”. Ellers trykkes ”Cancel”.

Efter brugeren trykker ”Ok”, vil systemet slette backup-filen.

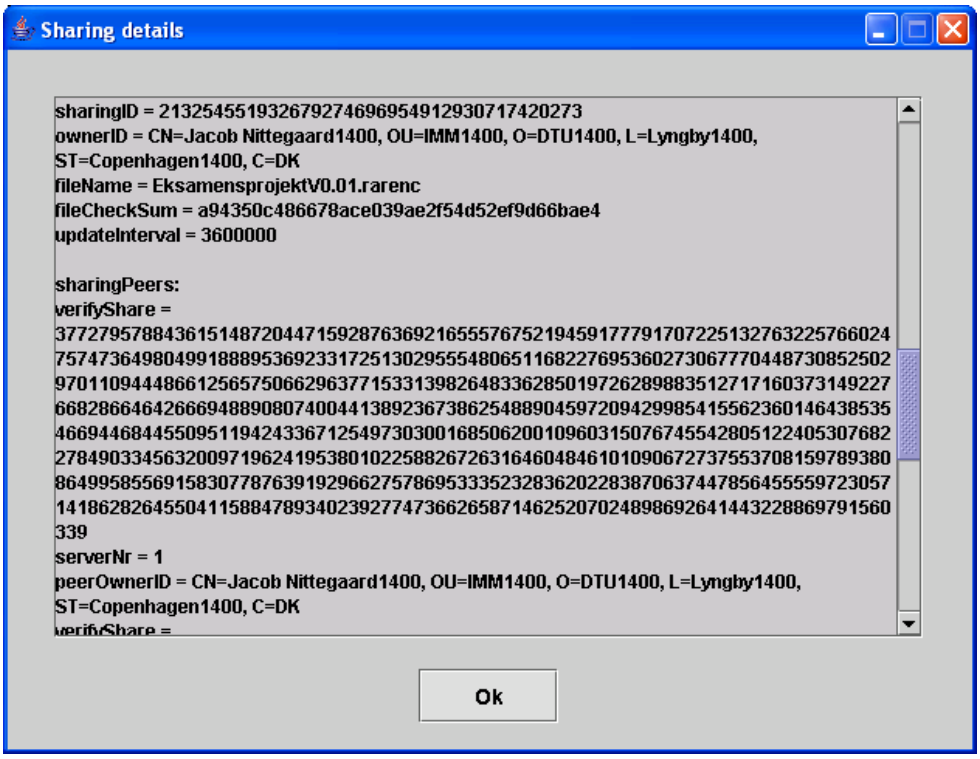
Filenavnet vil desuden blive fjernet fra ”My shared files”-listen i Resilias hovedvindue. Hos de deltagende gruppemedlemmer vil filnavnet blive fjernet fra ”Received files”-listen.

Herefter er operationen fuldendt.

Share details

Når brugeren ønsker detaljer om en backup, vælges denne først fra listen ”My shared files” eller fra listen ”Received files” i Resilias hovedvindue. Derefter trykkes på ”Share details”-knappen.

Programmet vil nu vise et vindue med information om backup’en. Denne inkluderer bl.a. programmets egen share-værdi og information om de andre noder. Et eksempel er vist herunder:



Scroll-baren til højre kan benyttes til at se alle informationerne. Når der trykkes ”Ok”, forsvinder vinduet og operationen er slut.

Refresh shares

Brugeren kan vælge at opdatere shares, hørende til en backup. I denne version kan opdateringen kun lade sig gøre en gang hvert 30. sekund.

Først vælges en backup fra listen ”My shared files” eller fra listen ”Received files” i Resilias hovedvindue. Derefter trykkes på ”Refresh shares”-knappen.

Systemet vil nu opdatere nodernes shares, tilhørende den valgte fil. Dermed bliver gamle ikke-opdaterede shares ubrugelige.

Integrity check
Brugeren kan checke integriteten af alle filer og shares hørende til en backup.

Først vælges en backup fra listen ”My shared files” eller fra listen ”Received files” i Resilias hovedvindue. Derefter trykkes på ”Integrity check”-knappen.

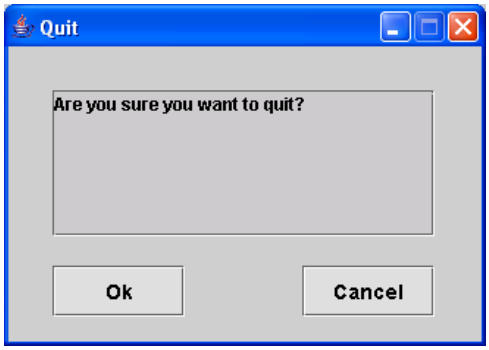
Systemet vil herefter checke alle shares og filers integritet og rette eventuelle fejl.

Request lost shares
Hvis alt brugerens viden om en eller flere backup’er er gået tabt, kan de andre gruppemedlemmer hjælpe med at genskabe denne.

Brugeren vælger ”Request lost shares” under ”program” i menubaren.

Derefter finder systemet brugerens shares og filer vha. de andre gruppemedlemmer.

Exit
Når Resilia ønskes afsluttet, vælges ”Exit” i menubaren. Derefter vil følgende dialogvindue blive vist:



Hvis brugeren trykker ”Ok”, forlades programmet. Hvis brugeren trykker ”Cancel”, lukkes dialogvinduet og programmet fortsætter.