Kryptografisk adgangskontrol i peer-to-peer netværk

Søren Hjarlvig og Jesper Kampfeldt

LYNGBY 2003
EKSAMENSPROJEKT
NR. 72

IMM

Informatik og Matematisk Modellering
Danmarks Tekniske Universitet
Danmark

Forord

Denne rapport er resultatet af undertegnedes eksamensprojektarbejde udført ved institut for Informatik og Matematisk Modellering, Danmarks Tekniske Universitet under vejledning af Christian D. Jensen.

Projektet er gennemført i løbet af knapt 6 måneder og er afleveret til bedømmelse d. 19. december, 2003.

Vi vil gerne takke vores vejleder Christian D. Jensen for entusiasme, gode ideer og opbakning gennem hele forløbet.

Vi vil desuden gerne takke vores medstuderende og familie for opmuntring, kommentarer og tålmodighed.

Jens Thyge Kristensen, Peter Neergaard Henrichsen og Ellen Hjarlvig skal have tak for kommentarer og hjælp til korrekturlæsning.

Lyngby, december 2003.

Søren Hjarlvig (c971483)

Jesper Kampfeldt (c973663)

Resume

I dette eksamensprojekt er mulighederne for at indføre læse- og skriverettigheder, samt konfidentialitet, integritet og autentifikation i peer-to-peer fildelingsnetværk blevet undersøgt. Der er blevet udviklet en sikkerhedsmodel baseret på kryptografisk adgangskontrol, og denne er analyseret. På baggrund af analysen er der udarbejdet et design for et fildelingsnetværk med de ønskede egenskaber. Endelig er der blevet udviklet en prototype, der implementerer det foreslåede design. En gennemført funktionel test viser, at den udviklede prototype fungerer efter hensigten.
Den udviklede sikkerhedsmodel er bygget op omkring nøglesæt og nøgleringe. Alle filer, nøgleringe og brugere i systemet repræsenteres ved individuelle nøglesæt. Et nøglesæt indeholder tre nøgler: En offentlig, en symmetrisk og en privat, hvor den offentlige og den private nøgle udgør et asymmetrisk nøglepar. Nøglerne giver mulighed for tre niveauer af rettigheder: integritet, læserettigheder og skriverettigheder.
Sikkerhedsmodellen er ganske fleksibel, idet den kan benyttes både i RBAC-miljøer og i almindelige DAC-miljøer. Endelig kan den udviklede sikkerhedsmodel benyttes i forbindelse med PKI. Analysen af mulige trusler viser, at modellen giver god sikkerhed med hensyn til sikring af konfidentialitet og integritet. Men at der mangler beskyttelse mod trafikanalyse og oversvømmelsesangreb.

Nøgleord: Kryptografi, kryptografisk adgangskontrol, peer-to-peer, datasikkerhed, rettigheder, distribuerede systemer, distribution af kryptografiske nøgler.

Abstract

This Master's Thesis investigates the possibilities of providing read and write permissions as well as confidentiality, integrity and authentication in peer-to-peer filesharing networks. A security model based on cryptographic access control is developed and analyzed. A design based on the analysis and an investigation of existing peer-to-peer filesystems is created. Finally a prototype implementing the proposed design is developed and successfully tested.
The security model is based on the concept of key sets and key rings. All files, key rings and users are represented by individual key sets. A key set contains at most three keys: A public, a symmetric and a private key. The public and the private key is an asymmetric key pair. The keys correspond to three levels of permissions: Integrity, read permissions and write permissions.
The security model is quite flexible, since it is applicable in RBAC environments as well as in ordinary DAC environments. In addition it is possible to use the security model as part of a PKI. The thread analysis shows that the model provides good protection of confidentiality and integrity. However, possible attacks exist based on traffic analysis and denial of service (DoS).

Keywords: Cryptography, cryptographic access control, peer-to-peer, security, permissions, distributed systems, key management.

Indhold

Kapitel 1

Indledning

Indenfor de seneste år har peer-to-peer netværk vundet stadig større udbredelse. Deres popularitet skyldes bl.a. deres decentrale natur, der gør det muligt at distribuere store mængder digitalt indhold, uden at der opstår flaskehalse på samme måde, som der ville gøre, hvis distributionen skete fra enkeltstående servere.

Eksisterende peer-to-peer fildelingsnetværk er primært rettet mod at distribuere data så effektivt som muligt mellem medlemmer af netværket. Alle medlemmer i netværket har typisk de samme rettigheder i forhold til en bestemt fil. Oftest er det kun muligt at opdatere en fil ved, at gøre en ny selvstændig fil tilgængelig for de andre medlemmer af netværket. Både den nye og den gamle udgave af filen vil derfor være tilgængelig i netværket. Det er så op til det enkelte medlem at afgøre, hvilken udgave der er den nyeste.

Hvis man ønsker at benytte peer-to-peer filsystemer til kollaborative formål, er manglen på sikkerhed og den manglende mulighed for at opdatere filer ofte uhensigtsmæssig. Manglen på sikkerhed kan også være et problem, hvis man ønsker at benytte et peer-to-peer netværk til distribution af digitalt indhold, da de brugere, der henter indholdet, ikke har nogen garanti for, at det de henter er autentisk. Desuden kan det være en ulempe, hvis alle brugere af netværket automatisk har adgang til alt indhold, hvis man ønsker at tage betaling for indholdet, eller blot ønsker at registerere, hvem der henter det.

Formålet med dette eksamensprojekt er at undersøge mulighederne for at indføre læse- og skriverettigheder samt konfidentialitet, integritet og autentifikation i peer-to-peer fildelingsnetværk ved hjælp af kryptografisk adgangskontrol.

Resultatet af disse undersøgelser er et forslag til en sikkerhedsmodel for et fildelingsnetværk med de ønskede egenskaber. Denne sikkerhedsmodel analyseres med henblik på at fastslå mulige sikkerhedshuller. Desuden vil det blive undersøgt, hvilke sikkerhedspolitikker den udviklede model understøtter.

På baggrund af sikkerhedsmodellen, samt en gennemgang af state-of-the-art indenfor peer-to-peer filsystemer, udvikles et design for det foreslåede system. Der implementeres dernæst en prototype af systemet. Endelig testes og evalueres prototypen i forhold til de opstillede mål.

Ideen med kryptografisk adgangskontrol er at kryptere en fil således, at man skal være i besiddelse af en bestemt nøgle for at kunne dekryptere og dermed få noget ud af at læse filen. Ligeledes skal man være i besiddelse af en bestemt nøgle, for at kunne kryptere og signere en opdatering af en fil korrekt. Public-key kryptering kan benyttes til dette, da den private nøgle kan bruges til at kryptere (skrive), og den offentlige nøgle kan benyttes til at dekryptere (læse).

En del af projektet vil derfor være at undersøge, hvordan distributionen af disse kryptografiske nøgler kan foregå hensigtsmæssigt.

Resten af denne rapport er organiseret således:

State of the art: Dette kapitel indeholder en gennemgang og klassifikation af forskellige eksistende peer-to-peer fildelingssystemer og andre systemer indenfor distribueret lager. Der gives desuden introduktioner til JXTA platformen, moderne krypteringsalgoritmer samt kryptografisk adgangskontrol.

Analyse: På baggrund af forskellige brugsscenarier opstilles en kravspecifikation for det ønskede system. Dernæst udvikles en sikkerhedsmodel og denne analyseres.

Design: Med basis i de to foregående kapitler dannes i dette kapitel et design for det foreslåede system. En række vigtige beslutninger med hensyn til valg af netværksarkitektur og krypteringsalgoritmer sker således her.

Implementering: I dette kapitel beskrives implementeringen af en prototype af systemet. Dette sker naturligvis på baggrund af designet udviklet i det foregående kapitel.

Evaluering: Den udviklede prototype evalueres i forhold til den opstillede kravspecifikation og der udføres en funktionel test. Desuden vil den grundlæggende idé omkring at benytte kryptografisk adgangskontrol i peer-to-peer netværk blive evalueret. Endelig vil der blive givet forslag til videre arbejde indenfor området.

Konklusion: Dette kapitel indeholder en konklusion på det samlede arbejde.

Når man sætter sig for at designe et sikkert system, kommer man let på usikker grund. Bruce Schneiers ord får derfor lov til at konkludere denne indledning:

> *"An attacker needs to find one successful attack: one minor vulnerability that the defender forgot to close. A defender, on the other hand, needs to protect against every possible attack."*
>
> - ***Secrets and Lies*, Bruce Schneier**

Kapitel 2

State of the art

Udbredelsen af netværk og kraftige personlige computere har gjort det attraktivt at udvikle systemer, som gør det muligt at aggregere ressourcer koblet til netværket.
Som et eksempel på dette kan nævnes SETI@home projektet. Her udnyttes ledige computere til at søge efter mønstre i radiodata opfanget med kraftige radioteleskoper.
Der findes flere andre projekter der udnytter ledig processortid, f.eks. til at finde primtal (Mersenne), bryde kryptering (Distributed.Net) og simulere protein foldning (Folding@Home).
Lagerplads kan på samme måde som CPU-tid deles og aggregeres. Men der er et væsentligt problem, som skal løses, førend det distribuerede lagersystem bliver brugbart: Hvordan fordeles og genfindes de data, som man ønsker at lagre [BKK$^+$03]. Som det vil fremgå, kan dette problem løses på flere forskellige måder.

I følgende fremstilling vil ordet *node* blive brugt som en dansk betegnelse for det engelske ord *peer*. En *node* betegner derfor en enkelt maskine der indgår i et peer-to-peer netværk.

Et *peer-to-peer netværk* defineres som et netværk af noder, hvor alle noder i princippet fungerer på lige fod. En node kan derfor både optræde som forbruger af de ressourcer andre noder stiller til rådighed og stille ressourcer til rådighed for andre noder.

I dette kapitel vil der først blive præsenteret en række forskellige peer-to-peer systemer. Dernæst gennemgås en taxonomi, der kan benyttes til klassifikation af disse systemer. De præsenterede systemer vil derpå blive klassificeret efter taxonomien.

Systemerne præsenteres i tre grupper:

Systemer til distribueret lager: Disse systemer er karakteriseret ved at data flyttes ud i netværket, væk fra den node der har produceret dem. Desuden har systemerne fokus på tilgængelighed, således at data kan genfindes, selvom nogle noder skulle fejle.

Anonymiserende systemer: Disse systemer giver mulighed for anonymt at dele og tilgå data. Systemerne er desuden modstandsdygtige overfor forsøg på at foretage censur.

Fildelingssystemer: I fildelingssystemerne deler hver node sine egne data med de andre noder. Når en node har hentet data, stilles de normalt automatisk tilrådighed for andre noder. Populære data vil derfor hurtig blive udbredt i netværket.

Til sidst i kapitlet vil der blive givet en introduktion til forskellige moderne krypteringsalgoritmer og koncepter. Dette er en forudsætning for kapitlets sidste afsnit, hvor idéen bag kryptografisk adgangskontrol bliver forklaret.

2.1 Distribueret lager

Der findes en del forskellige distribuerede lagersystemer (eng.: *distributed storage*) bl.a. OceanStore [RWE+01], PAST [RD01], FarSite [ABC+02] og CFS [DKK+01]. Hver for sig adresserer disse systemer en delmængde af de problemer og anvendelsesmuligheder, som denne klasse af systemer rummer.

Distribuerede lagersystemer benytter typisk en distribueret hashtabel til at bestemme placeringen af data [BKK+03]. Den grundlæggende idé er at fordele udfaldsrummet af en hashfunktion mellem noderne således, at hver node bliver ansvarlig for en del af udfaldsrummet. Ofte benyttes SHA-1 hashfunktionen til at producere en hashværdi af data og metadata. På samme måde dannes der for hver node en hashværdi, f.eks. på baggrund af nodens offentlige nøgle. Den node hvis hashværdi ligger tættest på datas hashværdi bliver så ansvarlig for opbevaringen [RD01].

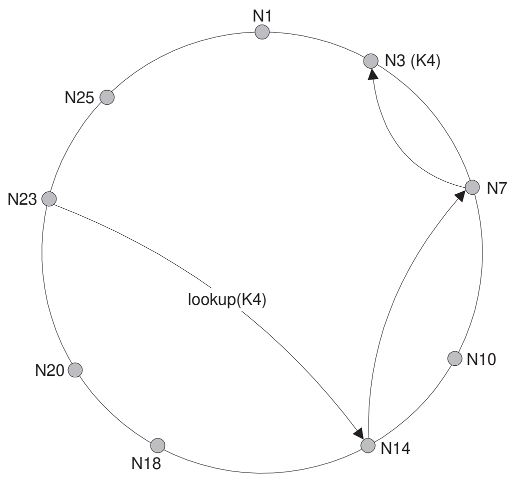
I figur 2.1 gives et eksempel på et opslag i en distribueret hashtabel. Node 3 er ansvarlig for data, der hasher til værdien 4. Der benyttes en routning, hvor opslaget for hvert router-hop kommer en bit tættere på målet.

I det følgende vil PAST blive beskrevet nærmere.

2.1.1 PAST

Hver node i et PAST-netværk bliver identificeret ved et 128 bit nodeID, dannet ved at hashe nodens offentlige nøgle. NodeID'et placerer noden i et cirkulært navnerum, der går fra 0 til $2^{128} - 1$.

PAST-klienter har mulighed for at foretage tre grundlæggende operationer:



Figur 2.1: Eksempel på opslag i en distribueret hashtabel.

- fileID = Insert(name, owner-credentials, k, file)
  Gemmer en fil på netværket på k forskellige noder. FileID bliver dannet som en 160 bit SHA-1 hash af filens navn, ejerens offentlige nøgle samt et tilfældigt valgt tal (eng.: *random seed*). Der dannes desuden et filcertifikat der signeres med nodens private nøgle. Certifikatet indeholder fileID'et, hashværdien af filen, k samt andre metadata. Noderne som skal lagre filen undersøger, om oplysningerne i certifikatet stemmer, førend filen accepteres.

- file = Lookup(fileID)
  Henter en kopi af filen på netværket. Hvis der findes flere kopier af filen på netværket, vil den "nærmeste" kopi blive hentet.

- Reclaim(fileID, owner-credentials)
  Fjerner en fil fra netværket. Efterfølgende kald af Lookup vil ikke med sikkerhed returnere filen. Der dannes et signeret reclaim-certifikat således, at de noder der lagrer filen, kan validere at det er ejeren, som forsøger at fjerne filen.

Lokalisering af data

PAST bygger ovenpå Pastry, som er en selvorganiserende, fejltolerant og skalerbar peer-to-peer router-protokol. Pastry gør det muligt at route beskeder ved højst $\lceil log_{2^b} N \rceil$ hop, hvor b normalt vil have værdien 4. Pastry benytter de 128 mest betydende bits af fileID'et til at route beskeder til den node, hvis 128 bit nodeID ligger numerisk nærmest.

Hver node vedligeholder følgende informationer om de andre noder i netværket:

- Router-tabel (eng.: *routing table*)

Denne tabel indeholder $\lceil log_{2^b} N \rceil$ niveauer med $2^b - 1$ pladser på hvert niveau. Tabellen er organiseret således at det n'te niveau indeholder adresser på noder, der deler n nodeID-cifre med den aktuelle node.

- Leaf-sæt
  Indeholder de næste $l/2$ noder, der har numerisk større nodeID end den aktuelle node, samt de foregående $l/2$ noder med numerisk lavere nodeID. l vil normalt have værdien 32.

- Neighborhood-sæt
  Indeholder adresserne på l noder, der ligger tæt på den aktuelle node. Ved "tæt" forstås her, at noderne er tæt forbundet i det underliggende fysiske net. Et mål for dette kan være ping-tiden mellem noderne.

Når en node modtager en besked, bliver den routet videre til en anden node, hvis nodeID har mindst et ciffer (svarende til b bits) mere tilfælles med beskedens fileID, end den aktuelle nodes nodeID. Hvis en sådan node ikke findes i router-tabellen, bliver beskeden i stedet sendt videre til den node fra leaf-sættet, som ligger numerisk tættest på fileID'et.
Når en fil er blevet lokaliseret, sendes den direkte til noden, der efterspørger den.

Nye noder

En ny node, der ønsker at tilslutte sig netværket, danner først et nodeID. Dette sendes til en eksisterende node i nærheden. Denne router det til den eksisterende node, hvis nodeID ligger numerisk tættest på det nye nodeID. Den nye node modtager nu leaf-sættet fra denne node og neighborhood-sættet fra den første node, der blev kontaktet. Router-tabellen dannes udfra den rute, den første besked blev sendt således, at den n'te node på ruten leverer den n'te række i router-tabellen.
Når en ny node bliver en del af netværket, bliver den måske samtidig ansvarlig for bestemte filer, hvis fileID ligger tæt på nodens nodeID. For at undgå, at alle disse filer skal overføres til noden, straks når den bliver en del af netværket, kan den nye node nøjes med at henvise til de noder, hvor filerne hidtil er blevet lagret. Filerne kan så siden overføres i baggrunden, når der er ledig kapacitet.

Hvis en node, der er udset til at lagre en fil, dvs. hvis dens nodeID er blandt de k numerisk nærmeste på fileID'et, men ikke har den fornødne lagerkapacitet, undersøger den, om en anden node i dens leaf-sæt har plads til at lagre filen. Hvis det er tilfældet, lagres filen på denne node og den første node gemmer blot en henvisning.
Hvis ingen andre noder i leaf-sættet har plads til filen, bliver der returneret en fejl til klienten. Klienten kan så generere et nyt fileID ved at vælge et andet tilfældigt tal som basis for hashfunktionen. Det nye fileID vil sandsynligvis ligge et andet sted i hash-udfaldsrummet, og noderne, som er ansvarlige for denne del af udfaldsrummet, vil måske have plads til filen.

Smartcards bruges til at sikre integriteten af de genererede fileID'er og nodeID'er

således, at en angriber f.eks. ikke kan komme til at kontrollere nabonoder eller styre filer i retning af bestemte dele af udfaldsrummet.
PAST understøtter desuden kvotaer således, at den enkelte brugers pladsforbrug kan reguleres.

2.2 Anonymiserende systemer

Distribuerede lagersystemer som Freenet [CSWH01] og Eternity [And96] fokuserer på at gøre udvekslingen af information anonym og modstandsdygtig overfor forsøg på censur. Ønsket om en høj grad af tilgængelighed er også centralt i disse systemer.

2.2.1 Freenet

I Freenet sikres anonymitet ved at route forespørgsler og svar gennem et antal noder. Den enkelte node på vejen kan ikke afgøre, om den foregående node er den oprindelige afsender af beskeden, eller om den næste node er den endelige modtager. Antallet af noder som beskederne routes igennem varierer tilfældigt. Kommunikationen mellem noderne er desuden krypteret således, at aflytning besværliggøres.

Nøgler

Freenet benytter primært to forskellige slags nøgler til at identificere indhold: Indholdsnøgler (eng.: *content-hash keys (CHK)*) og signerede navnerumsnøgler (eng.: *signed-subspace keys (SSK)*).
Der findes en tredje type nøgle, en såkaldt keyword-signed key (KSK). Den kan ses som en variant af SSK'en, hvor nøgleparret bliver genereret deterministisk udfra beskrivelsen. Den benyttes dog sjældent, da alle filbeskrivelser med denne type nøgle deler navnerum, og der ikke er nogen beskyttelse imod at samme beskrivelse benyttes om forskellige filer [CSWH01].

En CHK dannes ved at beregne en SHA-1 hash af filen, man ønsker at indsætte. Filer indsat med en CHK bliver krypteret med en tilfældig valgt krypteringsnøgle. For at få adgang til filen, skal man have CHK'en og krypteringsnøglen.

Fordelen ved at kryptere filerne med nøgler, der ikke kan udledes udfra filnøglerne er, at den enkelte node ikke umiddelbart kan vide hvilke data, den lagrer eller videresender.

SSK'er benyttes primært til at knytte beskrivelser til CHK'er. SSK'er dannes ved først at generere et tilfældigt offentligt/privat-nøglepar. Dernæst beregnes en hash af beskrivelsen. Denne sættes sammen med en hash af den offentlige nøgle, herefter hashes resultatet igen. Endelig signeres filen med den private nøgle. Filen krypteres desuden med en nøgle deterministisk genereret udfra be-

skrivelsen.
Forskellige filer hørende til det samme signerede navnerum, vil sandsynligvis have vidt forskellige nøgler, da beregningen af denne er afhængig af beskrivelsen. Dette medfører, at de typisk vil blive lagret forskellige steder i netværket, hvilket gør det mindre sandsynligt, at mange filer indenfor samme navnerum bliver utilgængelige samtidigt.

Mekanismen sikrer desuden, at kun den som er i besiddelse af den private nøgle, kan tilføje filer til et bestemt navnerum, da den private nøgle er nødvendig for at kunne signere filen korrekt. For at tilgå en SSK skal man således kun bruge den offentlige nøgle samt beskrivelsen.

En fil indsat med en SSK har ofte et indhold udelukkende bestående af en CHK. Dette gør det muligt at opdatere publicerede filer. Først indsættes den opdaterede fil under en ny CHK. Dernæst indsættes en opdateret SSK med den samme beskrivelse og offentlige nøgle, men med en henvisning til den nye CHK. Når en node modtager en opdateret SSK med en korrekt signatur, vil den erstatte den gamle SSK. Den gamle udgave af filen vil stadig kunne findes via den oprindelige CHK.

Lokalisering af data

Hver node vedligeholder en router-tabel, der kobler filnøgler med noder. Disse tabeller udveksles aldrig mellem noderne.
Når en node modtager en forespørgsel efter en bestemt filnøgle, undersøges det først om filen ligger i det lokale lager. Hvis ikke det er tilfældet, findes den nøgle som ligger tættest på i router-tabellen, og forespørgslen sendes videre til den tilhørende node. Dette fortsætter indtil data er lokaliseret eller forespørgslens TTL[1] bliver nul.
Hvis det første er tilfældet, returneres data ad samme vej som forespørgslen kom fra. Noderne på vejen gemmer en kopi af filen i deres lokale lager og opdaterer samtidigt deres router-tabel. Populære filer vil på den måde automatisk blive spredt yderligere i netværket, når de efterspørges.
For at bevare anonymiteten, kan en node vælge at sætte sig selv som kilde til filen, i de beskeder der returneres. Filen kan stadig lokaliseres korrekt, da nodens egen router-tabel stadig vil pege på den rigtige kilde.

Hvis data ikke kan lokaliseres eller en node ikke kan videresende forespørgslen, returneres en fejlbesked til den forudgående node. Denne sender så forespøgslen til den node, der er registret for den nøgle som ligger næst tættest på den efterspurgte. Resulterer dette også i en fejlbesked, prøves den tredje tætteste. Dette fortsætter indtil noden ikke har flere kandidater at prøve. Sker det, returneres en fejlbesked til nodens forudgåede node, der så prøver til sin næst tætteste, osv. Denne type søgning kaldes steepest-ascent hill-climbing med backtracking. Et lignende system bliver brugt, når data skal indsættes, blot bliver TTL værdien her et mål for, hvor mange noder data bliver lagret på.
Metoden gør, at bestemte noder efterhånden bliver specialiseret i at lagre fil-

[1]TTL - Time To Live, men i denne sammenhæng menes der egentlig “hops to live”.

nøgler indenfor bestemte dele af filnøgleudfaldsrummet. Det samme gælder ved søgninger, da nogle noder efterhånden vil få opbygget bedre router-tabeller indenfor bestemte dele af udfaldsrummet.

Sammenligner man Freenet-routning med routning i PAST systemet, kan man sige, at fordelingen af data udfra filnøgler i PAST er eksplicit (en node er specifikt ansvarlig for en del af udfaldsrummet). I Freenet er fordelingen mere implicit, da data har mulighed for at ende på mange forskellige noder, afhængigt af router-tabellerne i de noder data passerer. Jo mere specialiserede Freenet noderne bliver indenfor bestemte filnøgleområder, jo mere vil lagerstrukturen ligne den, man finder i et PAST netværk.

Hver Freenet-node stiller en vis mængde lager til rådighed for netværket. Efterhånden som filer indsættes og efterspørges, vil denne lagerplads blive brugt. Lageret virker som en LRU cache (Least Recently Used), idet de mindst efterspurgte filer vil blive overskrevet, hvis noden mangler lagerplads. Freenet garanterer altså ikke, at en fil vil forblive tilgængelig, hvis den ikke efterspørges i tilstrækkelig grad.

Nye noder

Når en ny node ønsker at tilslutte sig netværket, genereres et tilfældigt tal. En hash af dette sendes i en annonceringsbesked til en eksisterende node på netværket. Denne genererer et nyt tilfældigt tal, der XOR'es med den modtagne hash. Resultatet hashes igen, og sendes til en tilfældigt valgt node fra router-tabellen. Dette fortsætter indtil announcement-beskedens TTL bliver nul. Da afslører noderne deres tilfældigt valgte tal, og den nye nodes nøgle bliver nu dannet ved XOR mellem disse tal. Fremgangsmåden giver noderne mulighed for at validere, at de andre noder har afsløret de korrekte tal. Systemet sikrer, at ondsindede noder ikke kan manipulere med den nøgle, nye noder bliver registreret under i router-tabellerne.

Freenet i praksis

Siden den oprindelige Freenet artikel udkom [CSWH01], har Ian Clarke sammen med andre arbejdet på at videreudvikle systemet [Pro]. På nuværende tidspunkt er det muligt at hente referenceimplementationen og således køre en Freenet node på sin egen computer. En browserbaseret brugergrænseflade gør det muligt at tilgå indhold på Freenet på samme måde som almindelige hjemmesider. En foreløbig afprøvning afslører dog, at hastigheden, som det nok kunne forventes, lader en del tilbage at ønske. Desuden er softwaren plaget af diverse mere eller mindre graverende fejl. Dette har dog ikke afholdt folk fra at tage Freenet i brug, og således kan man finde alt fra politiske tekster og advarsler mod Scientology til piratkopieret software og MP3-filer.

2.3 Fildelingssystemer

Ideen bag fildelingssystemerne beskrevet i denne sektion er, at de enkelte brugere deler egne filer med andre brugere, der er tilsluttet netværket. Systemernes *reason d'entre* har typisk været ønsket om en effektiv og skalerbar distributionskanal til digitalt indhold, f.eks. programmer, musik eller film. Når en bruger har hentet en fil gøres den ofte automatisk tilgængelig for resten af netværket. Populære filer vil derfor hurtigt blive spredt i netværket og deres tilgængelighed således øget. Systemerne bliver i vid udstrækning brugt til ulovlig distribution af ophavsretligt beskyttet materiale.

2.3.1 Napster

Napster blev oprindeligt udviklet af Shawn Fanning, som et program hvor brugerne via internettet kunne dele mp3-sange. Programmet fik sin debut i 1999. Muligheden for at downloade sange i CD kvalitet gjorde Napster utrolig populært, og programmet havde på sit højdepunkt over 13,6 millioner brugere. Pladeselskaberne blev dog opmærksomme på, at brugerne primært delte ophavsretligt beskyttede sange. Dette førte til, at de sluttede sig sammen i søgsmål mod Napster. Dommen på søgsmålet kom i juli 2001, og beordrede Napster til at lukke sin service for at undgå flere krænkelser af ophavsretten. Napster indgik efterfølgende et forlig med pladeselskaberne, der gik ud på, at de skulle betale royalties for sange der fremover blev downloadet. Napster kunne dog ikke få deres forretning til at løbe rundt, og måtte 2. september 2002 indgive konkursbegæring.

Napster virkede ved, at brugerne havde en Napster-klient, der holdt styr på, hvilke sange man delte. Når en bruger ville downloade en ny sang, loggede Napster-klienten på en central Napster-server, der så formidlede forbindelsen til klienter, der havde den ønskede sang.

Der gives i det følgende en procedure for, hvordan download af filer med Napster fungerer. Da Napster er en lukket protokol bygger proceduren på OpenNap [BME], som er en åben udgave af Napster.

1. Der er to måder hvorpå man kan begynde en download i Napster, enten ved en *Search*- eller *Browse*-kommando. Ved en *Search*-kommando sender klienten en søgning til Napster-serveren, der returnerer en oversigt over de fundne sange til klienten. Ved en *Browse*-kommando spørger en klient en anden klient, hvilke filer den stiller til rådighed.
2. Når en bruger har fundet den sang, han vil downloade, enten vha. *Search* eller *Browse*, sender han en *get* kommando til serveren.
3. Serveren returnerer en *get ack*-kommando, som blandt andet indeholder IP-adresse og porten, som den afsendende klient lytter på.
4. Man opretter så en TCP-forbindelse på den specificerede port, og påbegynder downloaden.

5. I det øjeblik hvor downloaden går i gang, sender henholdsvis den modtagende og afsendende klient en *downloading file-* og *uploading file-*kommando til serveren. På tilsvarende måde sendes en *download complete* og *upload complete* til serveren, når downloaden er færdig.

På trods af Napsters korte levetid, havde systemet en betydelig indflydelse på udviklingen og udbredelsen af fildelingssystemer samt genoplivningen af peer-to-peer konceptet. Specielt Napsters retslige problemer har gjort, at stort set alle fildelingssystemer i dag fungerer uden en central server.

2.3.2 Gnutella

Gnutella blev opfundet og udviklet af Justin Frankel og Tom Pepper i marts 2000, og var tænkt som en protokol til udveksling af opskrifter. Det blev dog hurtigt opdaget, at Gnutella kunne bruges til udveksling af al slags data, og specielt ophavsretligt beskyttet materiale fandt sin vej til netværket. Dette betød dødsstødet for den officielle videreudvikling af Gnutella.

Vi vil i det følgende beskrive, hvordan Gnutella virker, med særligt fokus på, hvad der er specielt for Gnutella. Som nævnt tidligere er Gnutella en protokol, men samtidig er det også navnet på det program, som Justin Frankel og Tom Pepper udviklede for at vise, at protokollen virkede. I det følgende vil det fremgå, om der er tale om protokollen eller programmet.

Lokalisering af data

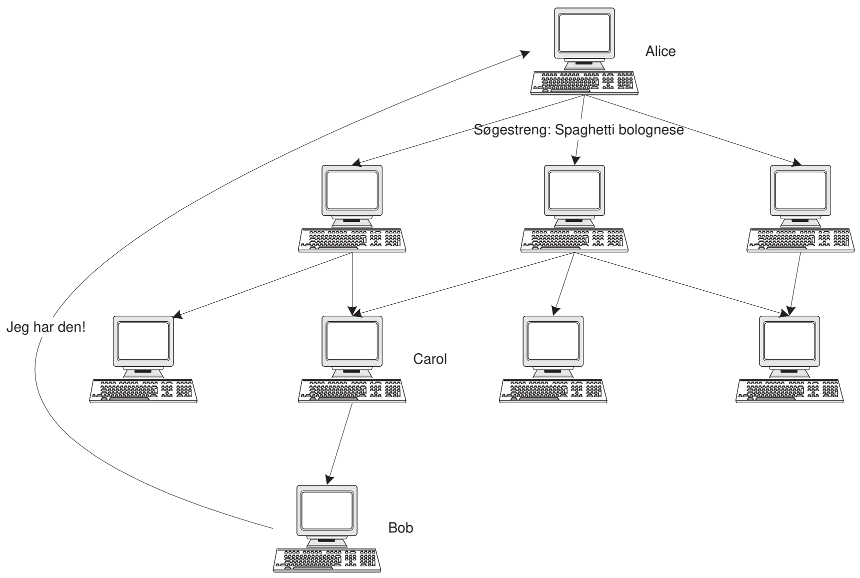
Gnutella er uafhængig af nogen form for central administration til at organisere netværket, udføre søgninger og formidle forbindelser mellem noder. Gnutella er altså en fuldstændig decentraliseret service.

Til at illustrere hvordan Gnutella virker, gives følgende eksempel:

1. Man starter med at sende et ping til en node, som man kender på netværket.
2. Ens ping bliver derefter broadcast'et til de nærliggende noder, der svarer med et pong. På denne måde opbygges en liste med noder omkring én og man er nu en del af netværket.
3. Hvis man f.eks. vil finde en opskrift på spaghetti bolognese, så spørger man de noder, man kender, om de har en.
4. Hvis en af de spurgte noder har opskiften, sender de et svar tilbage til én. Ligegyldig om en node har opskriften eller ej, sendes forespørgslen videre til de noder, som den spurgte node kender, i håb om, at de måske har en bedre eller anderledes opskrift. På den måde bliver forespørgslen langsomt udbredt til hele netværket.

5. Man har nu en liste over de opskrifter som passer til ens forespørgsel, og man kan vælge de opskrifter, man ønsker at downloade.

På figur 2.2 ses hvordan en søgeforespørgsel distribueres i et Gnutella-netværk. Pilene på figuren angiver afsendelsen af en søgeforespørgsel, og pilens retning hvem der modtager forespørgslen. Alice starter med at sende forespørgslen til dem hun kender, der så videresender den på samme måde. Bob sender en besked tilbage til Alice om, at han har en opskift, der matcher søgeforespørgslen.



Figur 2.2: Søgning i Gnutella.

Fri fortolkning

Læg mærke til, at der i ovenstående eksempel ikke står, hvordan den enkelte node matcher forespørgslen til konkrete filer på noden. Det er en af de unikke ting ved Gnutella, at der er fri fortolkning af forespørgsler. Til at illustrere dette byggede Gene Kan et al. [Ora01] en internetsøgemaskine, de kalder for InfraSearch, der bruger Gnutella-protokollen til at søge.

InfraSearch er som de fleste andre søgemaskiner en hjemmeside, hvor man kan indtaste sine søgekriterier. Når søgeordene er skrevet, sendes forespørgslen til et privat bagvedliggende Gnutella-netværk, hvor noderne svarer tilbage med html-sider, der passer til forespørgslen.

Som eksempel kunne man f.eks. forestille sig, at netværket bestod af Københavns Fondsbørs, Jyllands-Posten og en regnemaskine. Søger man efter "Mærsk" så vil Københavns Fondsbørs måske returnere en html-side med de nyeste aktiekurser, og Jyllands-Posten nogle nyheder omhandlende selskabet. Det vil sige, at hver node returnerer det der giver mening indenfor dens vidensområde. Dette bliver klart hvis man f.eks. søger på “1+1+16”. Dette vil sikkert ikke give mening for de fleste af noderne i netværket, på nær lommeregneren, der pludselig har fået noget, den forstår og returnerer 18.

Ydermere kunne denne form for søgemaskine også bruges til at søge på hjemmesider med dynamisk indhold, som traditionelle søgemaskiner har problemer med at indeksere, da det er de enkelte noder, der giver svarene og ikke hjemmesider, der parses. Søgningen vil desuden kunne foregå i realtid, i modsætning til “normale” søgemaskiner, der cacher og indekserer hjemmesiderne.

Arkitektur

Gnutella prøver at regulere netværket på en sådan måde, at de hurtigste noder kommer til at ligge som en inderkreds. Noder med mindre båndbredde lægger sig som ringe uden på inderkredsen, til man tilsidst har de langsomste yderst. På denne måde undgår man, at en langsom node forbinder to hurtige noder.

Al kommunikation i Gnutella-netværket foregår ved beskeder, som minder om pakker fra TCP-protokollen. Beskederne er alle forsynet med en universal unik identifikator (eng.: *Universally Unique IDentifier*) forkortet UUID [LR98]. Denne identifikator bruges til at undgå at beskeder sendes i ring i netværket. F.eks. kunne en node modtage en besked og sende den videre. Senere modtager noden måske beskeden endnu engang, via en anden rute, men fordi man cacher uuid'en, sendes beskeden ikke videre.

Det betyder at Carol på figur 2.2, der modtager den samme søgeforespørgsel to gange, kun sender den videre til Bob en gang.

Måden hvorpå Gnutella laver søgninger ved hjælp af broadcasts er meget ressourcekrævende og belastende for netværket. For at lette presset på netværket har hver besked et TTL felt, præcis som det kendes fra IP, hvor TTL-feltet angiver hvor mange hop, beskeden kan tage. Typisk sættes TTL til 7 for søgninger, hver gang en ny node modtager beskeden tælles TTL en ned, når TTL så bliver 0 videresendes beskeden ikke.

Skalerbarhed

Gnutella er blevet beskyldt for ikke at kunne skalere til et stort antal noder, hvilket både er rigtigt og forkert. Gnutella har aldrig været tænkt som et netværk, hvor alle kender alle på et vilkårligt tidspunkt. Så hvis man prøver at bygge et Gnutella-netværk, hvor enhver node kan “se” alle andre noder, så skalerer det ikke godt.

I Gnutella har man derfor et begreb, som man kalder for horisonten, der er et udtryk for hvor langt/meget man kan se. Når man står på stranden og skuer ud over vandet, vil ens udsyn være begrænset af jordens hældning. På samme måde begrænses antallet af noder man kan se f.eks. ved en TTL på 7 for alle beskeder. Denne grænse kaldes for horisonten. Ved at holde horisonten sådan, at noder maksimalt kan se omkring 10.000 andre, opnår man et velfungerende netværk [Ora01].

Udbuddet af filer, som hver enkelt node kan se, forandrer sig i takt med, at noder af- og tilgår den del af netværket, som den enkelte node kan se. Hvis man ikke er tilfreds med det man kan se, må man finde en anden indgang til netværket, hvilket vil ændre ens horisont.

Ideen med en horisont blev dog ødelagt, da folk begyndte at bruge host caches. En host cache er, som navnet antyder, en liste med alle de hosts som man er stødt på. Når denne liste vokser og vokser, så får man et netværk, der er mere og mere sammenvævet, hvilket gør at netværket overbebyrdes med søgebeskeder.

Host caches opstod da Gnutella-netværkets oprindelige indgang, findshit.gnutella.org, blev lukket. Det gjorde det besværligt at finde en indgang til netværket, og af frustration begyndte folk at gemme de hosts de fandt, hver gang de fik adgang til netværket.

Som et modtræk til host caches' store forbrug af båndbredde på netværket, har man udviklet specielle noder kaldet reflektorer. Disse noder fungerer ved at indeksere filerne, på de noder der er tilsluttet reflektoren. Når en søgebesked kommer til en reflektor, returnerer den de filer der passer til søgningen, men søgningen videresendes ikke. Dette giver en markant reduktion af antallet af søgebeskeder på netværket. Dog mister man nogle af de specielle ting ved Gnutella, nemlig fri fortolkning og søgninger i realtid.

Denne udvidelse har vi dog tilladt os at se bort fra i klassifikationen i afsnit 2.5.

2.3.3 FastTrack

FastTrack-protokollen benyttes af Kazaa, Grokster, IMesh programmerne og er ejet af firmaet Sharman Networks, der også står bag Kazaa klienten. Protokollen er lukket for offentligheden, men er i forbindelse med giFT-projektet [gIFT][gF] blevet analyseret og er på nuværende tidspunkt delvist dokumenteret i [Har].

Small world

I FastTrack-netværket er alle almindelige noder tilknyttet en supernode, der registrerer hvilke data noderne er i besiddelse af. Supernoderne udnævnes dynamisk. Forholdet mellem antallet af supernoder og almindelige noder er ifølge [Har] 1:100. Noderne, der udnævnes til supernoder, skal opfylde visse kriterier: De skal have en offentlig IP-adresse og en hurtig internetforbindelse, samt ledig processortid og tilstrækkeligt med ubrugt arbejdshukommelse.

Det har vist sig, at mange peer-to-peer netværk har en såkaldt "small world"-topologi [Mar02][Rip01]. Betegnelsen dækker over, at et lille antal af noderne har mange forbindelser til andre noder, men flertallet af noder har kun relativt få forbindelser. "Small world"-netværk er i høj grad tolerante overfor fejl. I Freenet kan 30% af noderne fejle, uden at netværkets ydelse påvirkes væsentligt [CSWH01]. Årsagen er, at hvis man forudsætter, at alle noder har den samme sandsynlighed for at fejle, vil fejl sandsynligvis ramme noder der har få forbindelser, da disse udgør majoriteten. Et målrettet angreb mod de velforbundne noder, vil selvfølgelig påvirke nettets ydelse i meget højere grad [CHM+02].

Ved at udnævne noder til supernoder udfra bestemte kriterier, forsøger FastTrack protokollen at styre hvilke noder, der bliver de velforbundne. På den måde kan det underliggende netværk udnyttes bedre.

Lokalisering af data

Ved søgninger sender noderne forespørgslen til deres supernode, der så søger lokalt og eventuelt sender søgningen videre til andre supernoder. Præcis hvilken information, der sendes rundt mellem supernoder, vides dog ikke. Men da antallet af beskeder som skal udveksles i forbindelse med søgninger på den måde bliver mindre, f.eks. i forhold til Gnutella, gør hierarkiet mellem noder og supernoder nettet mere skalerbart.
Når indhold er blevet lokaliseret, benyttes HTTP-protokollen til at overføre data direkte mellem noderne.

Nye noder

Når en node tilslutter sig netværket via en supernode, modtager den en liste med 200 andre supernoder. Denne information gemmes i en lokal supernodecache, der så kan benyttes næste gang noden skal tilsluttes netværket. Supernodecachen opdateres jævnligt så længe noden er forbundet til netværket. Problemet med at finde den initielle supernode løses formentlig ved at installere en supernodecache sammen med programmet. Supernoderne i denne cache kan så være noder kontrolleret af firmaet bag programmet. Når en node forsøger at tilslutte sig netværket, prøves de nærmeste supernoder først. Ved "nærmest" skal her forstås nærmest i det underliggende fysiske net.

2.4 JXTA

JXTA er et framework, der tilbyder en række nyttige primitiver i forbindelse med udvikling af peer-to-peer systemer. Motivationen for udviklingen af systemet var en erkendelse af det uhensigtsmæssige i, at genopfinde de samme grundlæggende mekanismer og protokoller, hver gang et nyt peer-to-peer skal udvikles.

JXTA projektet blev oprindeligt påbegyndt af SUN og den første referenceimplementation, JXTA 1.0, blev tilgængelig i april måned 2001. På nuværende tidspunkt foregår udviklingen i et åbent forum hvor alle kan bidrage. Denne gennemgang er baseret på JXTA 2.1 der blev tilgængelig i juni måned i år.

De grundlæggende elementer i JXTA er følgende:

- Noder (eng.: *peers*)
- Grupper (eng.: *peergroups*)
- Pipes
- Resolver
- Annonceringer (eng.: *advertisements*)

En gruppe er, som navnet indikerer, en gruppe af noder. En node kan være medlem af mange forskellige grupper. Medlemskabet kan beskyttes således, at kun bestemte noder har adgang til en gruppe.
Pipes er usikre, envejs, asynkrone kommunikationskanaler der benyttes til at sende beskeder mellem noderne.
Resolverens opgave er at knytte navne og logiske adresser til fysiske netværksadresser.
Alle Ressourcerne i JXTA identificeres med et unikt 128 bit JXTA ID. Annonceringer bliver brugt til at offentliggøre ressourcer, f.eks. noder, grupper, services eller indhold. Annonceringer benytter XML som dokumentformat og er således platformsuafhængige.

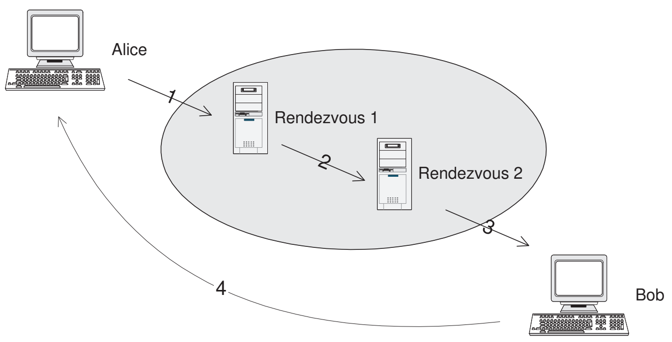
Lokalisering af data

Systemer som CFS (Chord) og PAST (Pastry) kan håndtere, at noder forlader og ankommer til netværket, men fungerer bedst, så længe dette ikke sker alt for ofte (lav “churn rate”). Jo højere nodernes udskiftningsrate er, jo flere ressourcer vil det kræve at vedligeholde den distribuerede hashtabels invarianter. Bliver udskiftningsraten høj nok, kan man risikere, at der bliver brugt flere ressourcer på at vedligeholde invarianterne, end på at foretage selve opslagene (“index trashing”) [TAA+03]. JXTA er især rettet mod peer-to-peer netværk hvor der er en høj udskiftningsgrad blandt noderne.

Rendezvousnoder (eng.: *rendezvous peers*) har en rolle, der ligner den supernoder og reflektornoder har i henholdvis FastTrack- og Gnutella-netværk. Almindelige noder, kaldet edge peers, har hver forbindelse til én rendezvousnode.

Hver rendezvousnode vedligeholder en liste over andre rendezvousnoder (Rendezvous Peer View (RPV)). Listen opdateres ved at hver node med jævne mellemrum, sender en kopi af listen til et antal tilfældigt udvalgte noder fra listen. Listen er ordnet efter ID, og hver node sender med jævne mellemrum en besked til noderne som ligger umiddelbart før og umiddelbart efter den selv på listen. Svarer disse noder ikke, bliver de slettet fra listen.
Hvis netværket af rendezvousnoder er forholdsvis stabilt, vil listerne hurtigt konvergere, og noderne får således et ens billede af indeksdistributionen [TAA+03].

Rendezvousnoder modtager annonceringer fra deres tilknyttede noder. Når en annoncering modtages, benyttes en hashfunktion til at beregne hvilken rendezvousnode fra listen, der skal have opdateret sit indeks. Typisk opdateres også nodens naboer på listen, af hensyn til tilgængeligheden hvis noden skulle fejle.



Figur 2.3: Eksempel på forespørgsel i JXTA.

Hvis en anden node ønsker at lokalisere annonceringen, sender den en forespørgsel til sin rendezvousnode. Denne beregner hashfunktionen og kontakter den relevante rendezvousnode, der kontakter den node, som oprindeligt foretog annonceringen. Endelig responderer noden, der foretog annonceringen, direkte til noden der forespugte. Se figur 2.3.

Hvis den relevante rendezvousnode ikke svarer, forsøges dennes nabonoder fra listen kontaktet. Hvis disse ikke kender annonceringen, videresender de forespørgslen til deres naboer i den respektive retning væk fra rendezvousnoden. Dette fortsætter indtil annonceringen er fundet eller forespørgslens TTL bliver nul.
Effekten af denne søgestrategi er, at de noder, som det er mest sandsynligt kender til annonceringen (de som er tættest på noden givet af hashfunktionen), vil blive spurgt først.

Den beskrevne indekseringsmekanisme benævnes ofte som *"Shared Resource Distributed Index"* eller bare *SRDI*.

Nye noder

Når en node skal tilsluttes netværket, tages en række metoder i brug for at finde en rendezvousnode.
Hvis noden har været tilsluttet før, prøves først rendezvousnoder kendt fra den foregående session. Lykkedes det ikke at etablere forbindelse til nogen af disse, forsøges multicast på det aktuelle netværk.
Hvis dette heller ikke lykkes, kontaktes særlige forudbestemte "seeding"-rendezvousnoder. Det forventes, at første gang en node skal tilsluttes netværket,

benyttes “seeding”-rendezvousnoderne, men efterfølgende vil noden kunne klare sig uden, hvilket mindsker afhængigheden af de faste “seeding”-rendezvousnoder.

Endelig skal det bemærkes, at JXTA kan håndtere forhindringer såsom firewalls og private netværk placeret bag disse. Dette sker ved hjælp af “relay”-noder. Er ens node placeret bag en firewall, således at man ikke kan modtage forbindelser fra eksterne noder, omgås dette ved at oprette en forbindelse til en “relay”-node, som er placeret på et åbent netværk. Eksterne noder sender nu deres forespørgsler til “relay”-noden, der så videresender dem.
Denne mekanisme vil virke, i de tilfælde hvor firewallen tillader forbindelser indefra og ud, men ikke forbindelser udefra og ind, hvilket er den mest normale sikkerhedspolitik.
JXTA giver desuden mulighed for at benytte HTTP som kommunikationsprotokol i stedet for TCP. JXTA kan altså fungere selv i de tilfælde, hvor en firewall kun tillader udgående HTTP trafik.

2.5 Klassifikation af peer-to-peer systemer

I [KIT02] opstilles fem kriterier, der kan benyttes til at klassificere peer-to-peer systemer. De fem kriterier er:

- Ressourcelokation (placering af data).
- Ressourcekontrol (placering af metadata).
- Brug af ressourcer.
- Global tilstandskontrol.
- Krav til QoS (Quality of Service).

Termen “ressource” skal her forstås bredt, som en betegnelse for hvad det er, netværket stiller til rådighed eller gør det muligt at dele; det kan f.eks. være processortid, lagerplads eller adgang til en distribueret database. Da de behandlede systemer alle benyttes til en form for fildeling eller lagring, vil “ressource” være identisk med “data” i det følgende.

2.5.1 Ressourcelokation

Placeringen af data kan være organiseret således, at alle noder i systemet ved hvor data befinder sig, eller placeringen kan være spredt og overladt til noderne selv.
Traditionelt serverbaseret lager, f.eks. NFS, kan ses som et ekstremt eksempel på organiseret placering af data. Her er data er placeret på en enkelt, global kendt lokation. Et andet eksempel på organiseret placering af data, er brugen af en distribueret hashfunktion til at beregne, hvor data skal placeres.
Ved spredt placering af data forstås, at det er den enkelte node som bestemmer placeringen af data, typisk ved at opbevare data selv.

2.5.2 Ressourcekontrol

Placering af metadata kan ligeledes være organiseret eller spredt. Hvis metadata findes på en eller flere globalt kendte noder, siges placeringen af metadata at være organiseret. Ved metadata forstås her de data, der er nødvendige for at finde frem til selve data. Afhængigt af systemet kan metadata også inkludere yderligere information om data, f.eks. en beskrivelse, eller angivelse af type og størrelse.

Det nok bedst kendte eksempel på organiseret placering af metadata er Napster, hvor centrale servere blev brugt til at indeksere de tilsluttede noders filer.
Hvis metadata derimod er spredt rundt på noderne, og man således ikke umiddelbart kan få et globalt overblik over metadata, siges metadata at være spredt.

2.5.3 Brug af ressourcer

Hvis brugen af en ressource kan foregå uden behov for andre ressourcer eller koordination med andre noder, siges brugen at være isoleret. Alternativt, hvis brugen af en ressource netop kræver dette, siges brugen at være kollaborativ.
I praksis vil mange systemer ligge et sted mellem disse to ekstremer. Henter man f.eks. en hjemmeside fra en webserver, kan man komme ud for, at kun selve teksten bliver hentet fra serveren. Billeder og reklamer der også optræder på siden, hentes fra andre servere, der er dedikeret til disse formål. De enkelte elementer kan måske i nogen grad bruges hver for sig, men alle elementerne skal helst være tilgængelige samtidigt, hvis brugeren skal opleve hjemmesiden som designeren har ønsket det.

2.5.4 Global tilstandskontrol

Hvis man har en distribueret database, kan det være nødvendigt med høj grad af tilstandskontrol (eng.: *state control*) mellem noderne, specielt i forbindelse med transaktioner, hvor man ønsker at sikre atomisitet, konsistens, isolation og persistens (ACID, af eng.: *Atomicity, Consistency, Isolation, Durability*).
I fildelingsnetværk vil der typisk ikke være så høje krav til den globale tilstandskontrol, men systemer, der benytter en distribueret hashfunktion til fordeling af data, vil ofte have brug for en grad af koordination mellem noderne, f.eks. når noder tilslutter sig eller forlader netværket.

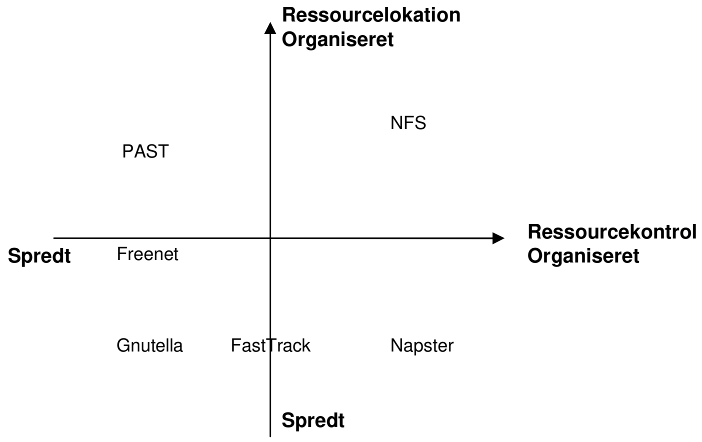
2.5.5 Quality of Service (QoS)

Involverer systemet direkte transmission af lyd, billed eller andre tidskritiske data, vil der være høje krav til QoS, både i kraft af kravet om et lavt antal fejl og kravet om lille forsinkelse på de transmitterede data. Men i de fleste tilfælde, hvor kommunikationen mellem noderne følger et query/response-mønster vil kravene til QoS være moderate.

Systemer der benyttes til at foretage videnskabelige beregninger, har ofte ret lave krav til QoS. Noderne kan returnere resultaterne, når det passer dem, og bliver et resultat ikke returneret, sendes opgaven blot til en anden node.

2.5.6 Klassifikation af eksisterende systemer

De fem klassifikationskriterier gør det muligt at placere peer-to-peer systemer i et fem-akset koordinatsystem. Men da alle de betragtede systemer har isoleret brug af ressourcer og moderate krav til QoS, er disse akser ikke medtaget i det følgende. Det tredimensionale klassifikationsrum er vist i figur 2.4 og figur 2.5 som to plane projektioner.



Figur 2.4: Ressourcelokation / Ressourcekontrol

I systemer som Napster, Gnutella og FastTrack befinder data sig initielt kun på en node, med mindre, selvfølgelig, at flere noder uafhængigt af hinanden vælger at gøre de samme data tilgængelige. Efterhånden som andre noder efterspørger og henter data, bliver data spredt i netværket. Disse tre systemer er derfor placeret ens på ressourcelokationsaksen (spredt ressourcelokation).

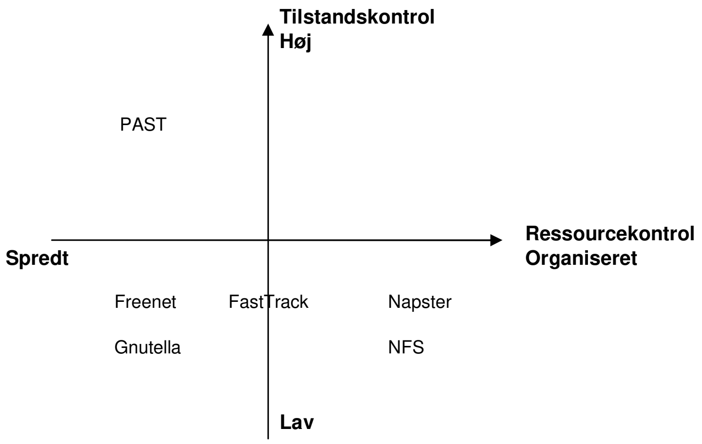
Da det således ikke på forhånd kan forudsiges, på hvilke noder data befinder sig, er det vanskeligt at lokalisere data. Napster løser dette problem ved hjælp af en central database (organiseret ressourcekontrol), Gnutella ved simpelthen at spørge alle noder, der befinder sig indenfor en vis afstand (spredt ressourcekontrol).

FastTrack benytter en løsning et sted i mellem disse to, hvor nogle noder udnævnes til supernoder, der så indekserer data og udfører søgninger. Da udnævnelsen af supernoder sker dynamisk og netværket således ikke er afhængigt af bestemte noder for at kunne fungere, er FastTrack klassificeret som en blanding af spredt og organiseret ressourcekontrol.

JXTA kan anvendes på mange forskellige måder og er derfor ikke medtaget i klassifikationen. Brugen af en distribueret hashfunktion til indeksering af tilgængelige ressourcer, samt brugen af rendezvousnoder, gør det dog fristende at placere JXTA på linje med FastTrack, midt mellem spredt og organiseret ressourcekontrol.

PAST benytter en hashfunktion til at beregne placeringen af data, ressourcelokationen er altså organiseret. Placeringen af data kan dog godt skifte efterhånden som noder tilslutter sig eller forlader systemet, og PAST er derfor placeret under NFS på ressourcelokationsaksen.
Som udgangspunkt er det op til klienterne selv at gemme metadata. Noderne der opbevarer data, opbevarer også visse metadata (filcertifikater), men disse bruges udelukkende til at validere filoperationer foretaget af klienten. Metadata betegnes derfor som værende spredt.

Freenet benytter en variant af denne teknik. Dog er placeringen af data ikke fuldt deterministisk, men vil afhænge af netværkets tilstand. Ressourcelokationen betegnes derfor som halvorganiseret. Det er stadig op til klienterne selv at gemme metadata; disse er derfor spredt. I praksis samler man ofte filnøgler, i filer der også distribueres via Freenet, f.eks. i form af html-formaterede hjemmesider, men da andre metadata stadigvæk skal eksistere lokalt, hvis man ikke ønsker at dele opdateringsretten til data, har vi i klassifikationen valgt at se bort fra denne overbygning.



Figur 2.5: Tilstandskontrol / Ressourcekontrol

For at routningen PAST kan fungere, kræves det, at de nærmeste noder opdaterer deres router-tabeller, når en node forlader eller tilslutter sig netværket. Nye noder skal hente kopier af de filer, som de er ansvarlige for, og hvis en node har forladt netværket, skal der måske oprettes ekstra kopier af de filer, som noden var ansvarlig for at opbevare.
PAST kræver derfor en høj grad af tilstandskontrol.

Gnutella-noder behøver kun at vedligeholde information om en eller flere nabonoder. Systemet har derfor meget lave krav til global tilstandskontrol. I Napsters tilfælde skal den centrale server holde styr på hvilke klienter, der er forbundet, samt hvilke data de er i besiddelse af. Desuden benyttes serveren til at formidle kontakt mellem to Napsterklienter, hvis den ene af dem er placeret bag en firewall. Napster kræver derfor nogen grad af global tilstandskontrol.

Freenet-noder kræver også en grad af tilstandskontrol, specielt i forbindelse med tilslutningen af nye noder. Desuden bliver netværket mere effektivt, efterhånden som noderne opbygger deres router-tabeller.

FastTrack er en lukket protokol, og klassifikationen af denne er derfor lidt usikker. Der er dog brug for en grad af koordinering i forbindelse med udnævnelsen af supernoder, samt ved søgninger og tilslutning af nye noder. FastTrack er derfor anbragt på linje med Napster og Freenet.

Endelig placeres NFS lavt på tilstandskontrolaksen, da denne protokol ikke kræver, at serveren vedligeholder tilstandsoplysninger om klienten.

2.6 Symmetrisk kryptering

Kryptering benyttes til at sikre konfidentialitet af de krypterede data. Symmetriske algoritmer kaldes således, fordi den samme nøgle benyttes både til kryptering og dekryptering. Moderne symmetriske krypteringsalgoritmer kan deles i to grupper: Stream-algoritmer og blokalgoritmer. I det første tilfælde lægges klarteksten sammen med strøm af pseudotilfældige data, genereret af krypteringsalgoritmen udfra nøglen, bit for bit, eller byte for byte. Blokalgoritmer krypterer klarteksten i blokke på typisk 64 eller 128 bit.

2.6.1 Stream-krypteringsalgoritmer

Sikkerheden i stream-algoritmer afhænger af hvor tilfældige de pseudotilfældige data der genereres er. Genereres der f.eks. kun 0'er, vil kryptoteksten være identisk med klarteksten. Genereres der derimod en fuldstændig tilfældig bitstrøm, vil sikkerheden være identisk med sikkerheden, der opnås ved anvendelse af en engangskodebog (eng.: emphone-time pad).

Den nok bedst kendte stream-algoritme er RC4, der blev udviklet af Ron Rivest for RSA i 1987. RC4 benyttes i vid udstrækning i forbindelse med sikker adgang til hjemmesider. RC4 er hurtig - cirka 10 gange hurtigere end DES - og der findes ingen kendte effektive angreb mod den. Algoritmen, der var hemmelig indtil 1994, hvor den blev postet anonymt til en mailing-liste, er desuden forbløffende enkel. RSA ejer fortsat rettighederne til RC4-navnet. Uafhængige implementationer kaldes typisk "alleged" (formodet) RC4. Planlægger man at sælge et produkt der indeholder en form for RC4, bør man derfor købe en licens hos RSA, for at være på juridisk sikker grund [Sch96].

Der er desuden nogle faldgruber, som man skal være opmærksom på i forbindelse med brugen af stream-algoritmer. Man må aldrig genbruge en kodestrøm.

Gør man det, kan en modstander neutralisere krypteringen ved at XOR're de to kryptotekster med hinanden. Resultatet er en XOR mellem de to klartekster. Dette kan let brydes, hvis man i forvejen kender indholdet af en af de to beskeder. Stream-algoritmer har desuden nogle linearitetsegenskaber, der kan have uheldige effekter i forbindelse med sikring af integriteten af de krypterede data. Afsnit 3.3.2 behandler dette emne nærmere.

2.6.2 Blokkrypteringsalgoritmer

Moderne blokkrypteringsalgoritmer bygger på Shannons begreber om forvirring (eng.: *confusion*) og diffusion. Forvirringen forårsages af substitutioner, typisk ved hjælp af S-bokse. Diffusion sker ved ombytning af klartekstens indhold således, at dennes statistiske karakteristiska bliver sløret. Disse to operationer samt en operation hvori nøglen indgår, gentages ofte i et antal runder.

AES

AES[2] er resultatet af en konkurrence udskrevet af NIST[3]. Formålet var at finde en afløser for DES[4] algoritmen, der var resultatet af en tilsvarende standardiseringsproces, som fandt sted i 1970'erne. Problemet med DES er, at standardnøglen på 56 bit i dag er for kort i forhold til den hastighed, hvormed man kan foretage et brute-force angreb mod algoritmen. I 3DES benyttes nøgler på 112 bit, hvilket giver tilstrækkelig sikkerhed. 3DES er til gengæld temmelig langsom, da DES-algoritmen skal afvikles tre gange i stedet for en enkelt gang. DES er designet således, at den kan afvikles hurtigt på dedikeret hardware. Afviklingshastigheden på moderne 32 bit processorer er derimod ikke så hurtig, som man kunne ønske sig. Endelig er blokstørrelsen i DES på kun 64 bit, hvilket ikke er hensigtsmæssigt, når der skal behandles store datamængder.

AES benytter en blokstørrelse på 128 bit, hvor data arrangeres i en kvadratisk matrice, der er 4 bytes på hver led.

Før selve krypteringen kan starte, udvides den 16 byte lange nøgle til en længde på 176 byte. Dette giver mulighed for en nøgle til hver runde, samt en nøgle der benyttes initielt, før første runde.

For 128 bit nøgler benytter algoritmen 10 runder, hvor hver runde består af 4 operationer:

Først benyttes en S-boks til at foretage bytevis substitutioner af alle elementerne i matricen.
Dernæst rykkes elementerne i hver række på forskellig vis rundt indenfor samme række ("shift rows").
Den følgende operation kaldes "mix columns", og her gives hvert element i ma-

[2] AES - Advanced Encryption Standard
[3] NIST - National Institute of Standards and Technology
[4] DES - Data Encryption Standard

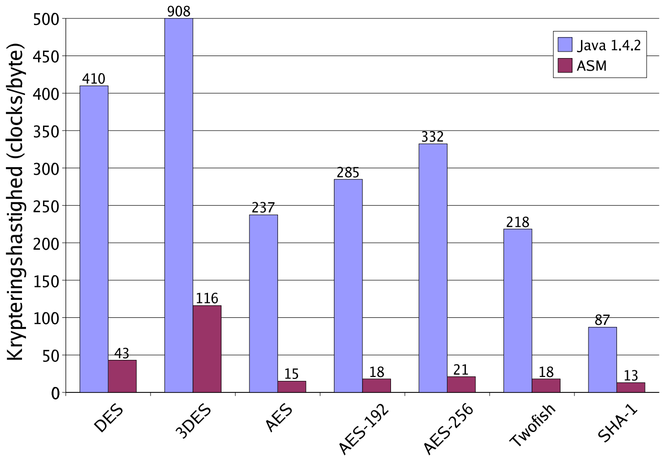
tricen en ny værdi, der er en funktion af alle fire bytes i den aktuelle søjle. Endelig lægges rundenøglen til. Det sker ved en XOR operation på alle elementer i matricen.

Formålet med de to midterste operationer, "shift rows" og "mix columns", er diffusion. Gentagelsen af disse operationer over mange runder sikrer, at alle data, der kommer ind, påvirker alle data, der forlader algoritmen.

AES har ikke, som f.eks. DES, en Feistel-struktur. Fordelen ved en Feistel-struktur er, at den samme algoritme kan bruges både til kryptering og dekyptering. Det er derfor nødvendigt at bruge inverse udgaver af ovenstående funktioner ved dekryptering. Det gælder selvfølgelig ikke den afsluttende XOR-operation, der er sin egen inverse.

For en mere detaljeret gennemgang af AES henvises til [Sta03].

Hastighed



Figur 2.6: Krypteringshastigheder for et udvalg af symmetriske krypteringsalgoritmer og hashfunktioner, programmeret i henholdvis assembler (ASM) og Java.

Assemblerhastighederne givet i figur 2.6 stammer fra [AL00]. De dækker over håndoptimerede assemblerimplementeringer af de forskellige algoritmer testet på en Pentium II processor. DES, 3DES og SHA-1 er dog optimeret og testet på en Pentium processor og vil sandsynligvis kunne køre lidt hurtigere på en Pentium II.

Javahastighederne er fremkommet ved at måle, hvor lang tid det tager at kryptere 1 Mb data under Java 1.4.2. I alle tilfælde, undtagen Twofish, er SUNs standardimplementering af algoritmerne blevet brugt (SunJCE 1.4.2). Twofish implementeringer stammer fra FlexiProvider [Fle]. Testen er foretaget på en Athlon XP 2000 (1.66 GHz) processor med 256 Mb ram under Redhat Linux 8.

AES og Twofish benytter 128 bit nøgler. AES-192 og AES-256 benytter, som navnene antyder, nøglelængder på henholdsvis 192 bit og 256 bit. Endelig benytter DES en nøglelængde på 56 bit og 3DES en nøglelængde på 112 bit.
Hastighederne for AES-192 og AES-256 er estimeret ved at lægge 20% henholdsvis 40% til AES. Dette skyldes, at AES normalt benytter 10 runder, AES-192 benytter 12 runder og AES-256 benytter 14 runder.
SHA-1 er ikke en krypteringsalgoritme, men en hashfunktion. Den er kun medtaget her for sammenligningens skyld.

Nøglelængder

Hvis man er i besiddelse af en blok klartekst og den tilsvarende kryptotekst, kan man lave et brute-force angreb, der består i at kryptere klarteksten med forskellige nøgler indtil resultatet svarer til kryptoteksten [Sch96]. Benyttes 56 bit nøgler er der 2^{56} ($\approx 10^{16}$) mulige nøgler. I gennemsnit vil det dog kun være nødvendigt at afprøve halvdelen før den korrekte nøgle bliver fundet.

Hvis man vil foretage et angreb af denne type mod AES, vil det kræve 237 clockcykler per nøgle at foretage selve krypteringen på en Pentium II processor [AL00]. Dertil kommer så sammenligningen mellem den resulterende og den originale kryptotekst, samt administration og klargøring af nøglerne, som skal afprøves. Hvis der ses bort fra dette, kan en 237 MHz Pentium II processor teste 10^6 nøgler i sekundet. Hvis man antager, at en moderne 2 GHz processor bruger samme antal clockcykler per nøgle, vil den kunne teste $8 \cdot 10^6$ nøgler i sekundet. Hvis man har en million af sådanne maskiner til rådighed, hvilket ikke forekommer urealistisk, set i lyset af, at f.eks. SETI@home projektet i øjeblikket har 4.6 millioner deltagere, fremkommer nedenstående tabel. Tallene for DES og 3DES er beregnet ud fra, at det tager hhv. 340 og 928 clockcykler at kryptere en 64 bit blok [AL00].

Nøglelængde (bits)	$T_{\frac{1}{2}}$	T
56 (DES)	1.7 timer	3.4 timer
112 (3DES)	$4 \cdot 10^{13}$ år	$8 \cdot 10^{13}$ år
128 (AES)	$6 \cdot 10^{17}$ år	$1 \cdot 10^{18}$ år
192 (AES)	$1 \cdot 10^{37}$ år	$3 \cdot 10^{37}$ år
256 (AES)	$3 \cdot 10^{56}$ år	$6 \cdot 10^{56}$ år

Universet er omkring 15 milliarder år gammelt (10^{10}) [Sch96], så 10^{17} år er rigtig lang tid. Der kan selvfølgelig fremkomme mere effektive angreb, ligesom hardwaren til stadighed bliver hurtigere. Det første er svært at gardere sig helt imod. Hvis man vil mindske risikoen for denne type angreb, er det sikreste at benytte krypteringsalgoritmer der har eksisteret længe og derfor har været udsat for de mest omfattende analyser. 3DES er her et oplagt valg [Sch96]. Prisen man betaler for denne sikkerhed, er den dårligere ydelse. Man kan dog konstatere, at AES allerede har været udsat for ganske omfattende sikkerhedsanalyser, uden at det er lykkedes at finde effektive angreb mod den komplette krypteringsalgoritme [FKL+00].

2.7 Public-key kryptering

Indtil public-key kryptering blev offentligt kendt i 1976 gennem Diffie og Hellmans epokegørende artikel [DH76], havde hemmelig kommunikation krævet, at en hemmelig nøgle på forhånd var udvekslet. I public-key kryptering skal der stadig udveksles nøgler, men disse behøver ikke at blive holdt hemmelige. Public-key kryptering kaldes også asymmetrisk kryptering, fordi det ikke er den samme nøgle, der benyttes til kryptering og dekryptering.

Hvis Alice og Bob ønsker at kommunikere sikkert sammen, genererer de hver en offentlig og en privat nøgle. Alice og Bob udveksler nu deres offentlige nøgler. Når Alice fremover ønsker at sende en hemmelig besked til Bob, gøres dette ved at kryptere beskeden med Bobs offentlige nøgle. Kun Bob der er i besiddelse af den tilsvarende private nøgle, kan dekryptere beskeden.
Systemet kan også bruges til digital signatur. Hvis Alice først krypterer beskeden til Bob med sin egen private nøgle, vil kun hendes offentlige nøgle kunne bruges til at dekryptere beskeden. Dette garanterer at kun Alice kunne have skrevet og signeret beskeden. De to lag af kryptering giver på den måde både integritet og konfidentialitet.

Det er uden betydning, om udvekslingen af offentlige nøgler bliver aflyttet, men det er vigtigt at nøglerne ikke kan modificeres under udvekslingen. Hvis Mallory f.eks. har held til at udskifte Alices nøgle med sin egen, vil hun fremover kunne læse Bobs beskeder til Alice. En sikker måde at foretage nøgleudvekslingen på, kunne være at Alice og Bob mødes og udveksler disketter eller lignende med nøglerne. Alternativt kan nøglerne udveksles via e-mail. Efterfølgende kan de verificeres ved at oplæse en hashværdi af nøglerne i telefonen (forudsat, selvfølgelig, at Mallory ikke kan imitere Alices eller Bobs stemme).

De vigtigste typer af public-key krypteringssystemer omfatter:

- RSA, der er baseret på vanskeligheden i at faktorisere store tal.
- El Gamal og andre systemer baseret på diskrete logaritmer, f.eks. DSA.
- Systemer baseret på elliptiske kurver, f.eks. ECDSA.

Af de disse tre er RSA langt det mest udbredte.

2.7.1 RSA

Sikkerheden i public-key systemer bygger på matematiske operationer, som er forholdsvis hurtige at udføre den ene vej, men kræver store beregningsmæssige ressourcer at invertere. Et eksempel på en sådan operation er multiplikation og faktorisering. Det er let at gange to store tal sammen, men krævende at faktorisere resultatet til de oprindelige operander. Dette udnyttes i RSA algoritmen, der i det følgende kort vil blive forklaret.

Kryptering og dekryptering udføres ved henholdsvis:

$$C = M^e \; mod \; n \tag{2.1}$$

og

$$M = C^d \; mod \; n = (M^e)^d \; mod \; n \tag{2.2}$$

Hvor M er en blok klartekst, og C er den tilsvarende blok kryptotekst.
En blok er k bit lang, hvor $2^k < n \leq 2^{k+1}$.

Både afsender og modtager skal kende værdien af n. Afsenderen skal desuden kende e, og modtageren skal kende d. Den offentlige nøgle bliver altså $KU = \{e, n\}$ og den private nøgle $KR = \{d, n\}$.

For at algoritmen kan virke, skal man altså kunne finde værdier af e, d og n således at:

$$M^{ed} = M \; mod \; n \tag{2.3}$$

Det kan udledes af Eulers theorem, at følgende forhold gælder [Sta03]:

$$m^{k\phi(n)+1} \equiv m \; mod \; n \tag{2.4}$$

Hvor $\phi(n)$ er Eulers totient-funktion, der angiver antallet af positive heltal, der er mindre end n og indbyrdes primiske med n. k er et arbitrært valgt heltal.

Hvis p og q er to primtal, der er valgt således at $n = pq$ og $0 < m < n$, fås endvidere:

$$m^{k\phi(n)+1} = m^{k(p-1)(q-1)+1} \equiv m \; mod \; n \tag{2.5}$$

Hvis algoritmen skal virke, skal det altså gælde at:

$$ed = k(p-1)(q-1) + 1 \tag{2.6}$$

Hvilket også kan skrives som:

$$\begin{aligned} ed &\equiv 1 \; mod \; \phi(n) \\ d &\equiv e^{-1} \; mod \; \phi(n) \end{aligned} \tag{2.7}$$

d og e er altså multiplikative inverse modulo $\phi(n)$, hvis d,e og $\phi(n)$ er indbyrdes primiske, dvs. deres største fælles divisor er 1.

Ønsker man at anvende RSA algoritmen skal man altså først vælge to primtal, p, q. Den bedste sikkerhed opnås hvis disse primtal har samme længde. n beregnes ved at multiplicere p og q: $n = pq$.

e vælges tilfældigt således, at e er primisk med $\phi(n)$ og mindre end $\phi(n)$, dvs. $gcd(\phi(n), e) = 1$ skal være opfyldt.

d kan nu beregnes ved: $d \equiv e^{-1}\ mod\ \phi(n)$
Euklids udvidede algoritme kan benyttes til dette.

Eksempel på beregning af RSA-nøglepar:

1. p og q vælges: $p = 11$ og $q = 13$.
2. n beregnes: $n = pq = 11 \cdot 13 = 143$.
3. Eulers totient-funktion beregnes:
   $\phi(n) = \phi(143) = (11 - 1)(13 - 1) = 10 \cdot 12 = 120$.
4. e vælges således, at e er primisk med $\phi(n) = 120$ og mindre end $\phi(n)$: $e = 7$.
5. d beregnes således, at $de \equiv 1\ mod\ 120$: $d = 103$, da $7 \cdot 103\ mod\ 120 = 1$.

Den offentlige nøgle bliver da $KU = \{e, n\} = \{7, 143\}$ og den private nøgle $KR = \{d, n\} = \{103, 143\}$.

Ønskes beskeden $M = 42$ krypteret, gøres følgende:

$$C = M^e\ mod\ n = 42^7\ mod\ 143 = 81 \tag{2.8}$$

$$M = C^d\ mod\ n = 81^{103}\ mod\ 143 = 42 \tag{2.9}$$

Ønsker man at bryde sikkerheden i RSA, er den mest oplagte mulighed at forsøge at faktorisere n til p og q. Dette muliggør beregning af $\phi(n)$ og dermed bestemmelse af d, da man allerede kender e, der er en del af den offentlige nøgle. En anden mulighed er at forsøge at bestemme $\phi(n)$ eller d direkte, men ingen af disse angreb er lettere at udføre end faktoriseringsangrebet [Sch96].

Til faktorisering benyttes typisk algoritmerne "Quadratic sieve" og "Generalized number field sieve". Sidstnævnte er hurtigst for store værdier af n. F.eks. kræver det $3 \cdot 10^{11}$ mips-år at faktorisere et 1024 bit tal hvis der benyttes "Quadratic sieve", mod $3 \cdot 10^7$ mips-år hvis der benyttes "Generalized number field sieve" [Sch96].

Firmaet RSA Laboratories har udskrevet en konkurrence, hvor de indbyder alle til at forsøge at faktorisere tal af forskellig størrelse. I 1999 lykkedes det at faktorisere et 512 bit langt tal, det næste nummer i rækken er 576 bit langt. Dette tal er netop blevet faktoriseret d. 3. december i år.

I systemer, hvor symmetrisk og public-key kryptering benyttes i kombination, er det vigtigt at der er balance mellem de benyttede nøglelængder således, at de to typer kryptering bliver lige vanskelige at bryde. Ifølge [Sch96] bør man, hvis der benyttes en 128 bit symmetrisk nøgle, bruge en 2304 bit RSA nøgle. Benytter man elliptisk kurve-kryptering (EC) bør man, ifølge [Use], benytte en privat nøgle, hvis længde er mindst det dobbelte af den symmetriske nøglelængde. Benyttes en 128 bit symmetrisk nøgle, bør man altså bruge en 256 bit EC nøgle. Ved elliptisk kurve-kryptering kan man altså bruge væsentligt kortere nøglelængder og opnå samme sikkerhed som ved RSA. De kortere nøgler fylder mindre, og kan genereres hurtigere. Kryptering og dekryptering kan ligeledes udføres hurtigere [Sta03].

2.8 Hashfunktioner

Til sikring af integritet af data benytter man typisk en hashfunktion. En hashfunktion er en funktion, der som input tager en meddelelse, typisk i form af en binær repræsentation af denne, og beregner en n bit stor værdi. Denne hashværdi har en $(1 - \frac{1}{2^n})$ sandsynlighed for at ændre sig, hvis meddelelsen ændrer sig. Den førnævnte sandsynlighed bygger på, at hashfunktionen afbilder meddelelser til hashværdier, der er fuldstændigt tilfældigt fordelt. Sikkerheden for integritet ved brug af hashfunktioner ligger altså i, at man i gennemsnit skal beregne hashværdier for 2^{n-1} forskellige meddelelser, for at få en kollision med en bestemt hashværdi. Man siger, at hashfunktionen har et sikkerhedsniveau (eng.: *security level*) på 2^{n-1} bit.

For at sikre, at en hashfunktion har denne sikkerhed på 2^n bit, stiller man typisk følgende krav til en hashfunktion:

Envejs: At en hashfunktion H er envejs (eng.: *preimage resistant*) betyder, at hvis man har en hashværdi z, så er det beregningsmæssigt svært at finde en meddelelse x, hvorom der gælder $H(x) = z$.

Svagt kollisionsfri: For en hashfunktion H der er svagt kollisionsfri (eng.: *2nd-preimage resistant*) gælder der, at for en specifik meddelelse x, er det beregningsmæssigt svært at finde en ny meddelelse $x' \neq x$ således at $H(x') = H(x)$.

Stærkt kollisionsfri: En hashfunktion H er stærkt kollisionsfri (eng.: *collision resistant*), når det er beregningsmæssigt svært at finde to meddelelser $x' = x$ sådan at $H(x') = H(x)$.

At en hashfunktion er stærkt kollisionsfri medfører, at den er svagt kollisionsfri, da det aldrig er sværere at finde en kollision til en vilkårlig hashværdi, frem for til en bestemt hashværdi. Derimod medfører en stærkt kollisionsfri hashfunktion ikke nødvendigvis, at den også er envejs [Pre99].

Man skelner mellem hashfunktioner, der tager et eller to input-argumenter. Hashfunktioner, der kun har et input i form af en meddelelse, kalder man for MDC'er (Manipulation Detection Codes). Dog bruger meget litteratur betegnelsen kryptografiske hashfunktioner eller bare hashfunktioner om denne klasse af hashfunktioner. Den anden klasse af hashfunktioner, der ud over beskeden også tager en nøgle som input, kalder man for MAC'er (Message Authentication Codes) eller "keyed hash functions".

Når man skal vælge en hashfunktion, bliver man nødt til at se på, hvor stor en hashværdi den giver, og hvor godt den opfylder hver af de tre krav. Vi vil nu for hver af de to klasser kigge på, hvor godt hashfunktionerne opfylder ovenstående tre krav. Eller sagt på en anden måde: Hvilke angreb skal man være opmærksom på for hashfunktioner af de to klasser.

En MDC tager, som tidligere nævnt, som input en meddelelse og returnerer en hashværdi, hvilket betyder, at alle kan checke om en MDC afbilder en meddelelse til en bestemt hashværdi. Dette fører til det såkaldte fødselsdagsangreb [Yuv79] (eng.: *birthday attack*), der bygger på fødselsdagsparadokset. Fødselsdagsparadokset siger at, hvis man har en gruppe på 23 personer, så er der lige over 50 % ($1 - \frac{365}{365} \cdot \frac{364}{365} \cdot \ldots \cdot \frac{343}{365} \approx 0,507$) sandsynlighed for, at to har fødselsdag samme dag. Dette er selvfølgelig ikke noget paradoks, men et ikke særligt intuitivt faktum. En n bit hashfunktion har derfor en sandsynlighed på

$$1 - e^{-\lambda} \tag{2.10}$$

for kollision, når der beregnes hashværdier for r forskellige meddelelser, hvor r $\approx \sqrt{\lambda \cdot 2^n} = \sqrt{\lambda} \cdot 2^{\frac{n}{2}}$ og 2^n er stor [TW02].

Fødselsdagsangrebet benytter sandsynligheden på følgende måde. Alice og Mallory er ved at indgå en ejendomshandel, hvor Mallory sælger en grund til Alice. Det foregår på den måde, at Mallory laver en salgskontrakt som Alice så signerer, og som efterfølgende er bindende. Som ved de fleste signeringssystemer beregnes der først en hashværdi af kontrakten, der så krypteres med Alices private nøgle. Mallory har udover den gode grund, som Alice forventer at købe, også en værdiløs sump.

Mallory prøver derfor at narre Alice til at købe sumpen ved at lave 2^{30} forskellige kontrakter, hvor Alice køber den gode grund og 2^{30} forskellige kontrakter hvor Alice køber sumpen. Kontrakterne behøver kun at være meget lidt forskellige f.eks. kan Mallory bruge dobbelte mellemrum eller bruge store bogstaver forskellige steder i kontrakterne.

Vi ved fra (2.10), at sandsynligheden for at der er en kollision i hver af de to grupper med hhv. gode og dårlige kontrakter er $1-e^{-\lambda}$. Men sandsynligheden for en kollision mellem de to grupper er ligeledes $1-e^{-\lambda}$, når r $\approx \sqrt{\lambda \cdot 2^n} = \sqrt{\lambda} \cdot 2^{\frac{n}{2}}$ er antallet af meddelelser i hver af de to grupper.

Hvis Mallory benytter en 2^{50} bit hashfunktion, er sandsynligheden for, at der er en kollision mellem de to grupper af kontrakter $1 - e^{-1024} \approx 1$ da $r =$

$\sqrt{\lambda \cdot 2^n} \Rightarrow 2^{30} = \sqrt{\lambda \cdot 2^{50}} \Rightarrow \lambda = 2^{10} = 1024$. Mallory tager nu henholdsvis den gode og dårlige kontrakt, der kolliderer, og giver Alice den gode som hun signerer. Mallory har nu en signatur, der gælder til begge kontrakter, og kan så påstå at Alice har købt sumpen. Hvis Alice ville have undgået dette angreb, skulle hun bare have lavet en lille ændring i kontrakten, inden hun signerede den.

Fødselsdagsangrebet kan være ganske hukommelseskrævende, hvis man f.eks. skal have de $2 \cdot 2^{30}$ hashværdier, fra forrige eksempel, i hukommelsen vil det kræve $\frac{2 \cdot 2^{30} \cdot 2^{50}}{8 \cdot 1024 \cdot 1024} = 6400$ Mb. Dette problem er dog blevet løst af Quisquater i [QD90], hvor han giver en algoritme, der gennemsnitligt skal evaluere $2 \cdot \sqrt{\frac{\pi}{2}} \cdot 2^{\frac{n}{2}}$ meddelelser før der er en kollision [Pre99], og som bruger en negligerbar mængde hukommelse. På grund af dette resultat er der generel tendens til at sige, at hashfunktioner kun leverer et sikkerhedsniveau på $2^{\frac{n}{2}}$ mod et fødselsdagsangreb. Som en direkte konsekvens af fødselsdagsangrebet, kan en MDC aldrig være stærkt kollisionsfri med et sikkerhedsniveau højere end ca. $2^{\frac{n}{2}}$.

Det er åbenlyst hvorfor man ønsker egenskaberne svagt og stærkt kollisionsfri i en hashfunktion, men måske er det uklart, hvorfor man også ønsker, at den skal være envejs. I mange sammenhænge kender man godt meddelelsen som hash'en er beregnet udfra, hvorfor der umiddelbart ingen skade kan ske, hvis man kan genskabe meddelelsen ud fra hash'en. Problemet er derimod, at hashfunktioner, der ikke er envejs, er åbne for mød-i-midten-angreb (eng.: *meet-in-the-middle attack*). Med et mød-i-midten-angreb kan man lave en kollision til en bestemt meddelelse ved ca. $2^{\frac{n}{2}}$ beregninger, hvilket gør at MDC'er der ikke er envejs, ikke kan være svagt kollisionsfri med et sikkerhedsniveau højere end ca. $2^{\frac{n}{2}}$.

En del af de tidlige hashfunktioner byggede på symmetrisk kryptering, hvor nøglen var kendt eller blev taget udfra meddelelsen, der skulle hashes. Da symmetrisk kryptering tydeligvis ikke er envejsfunktion, så kunne de brydes ved et mød-i-midten-angreb [Cop85]. En mere gennemskuelig grund til, at en hashfunktion skal være envejs er, at da en hashfunktion eller dens kompressorfunktion (eng.: *compression function*) typisk ikke er injektiv er det oplagt, at man ikke ønsker at kunne finde et pre-image, da det så kan være forskelligt fra det originale, hvilket betyder, at man har en kollision.

En MAC tager, som nævnt tidligere, udover meddelelsen, man vil hashe også en nøgle, hvormed meddelelsen hashes. Med andre ord, en MAC vil give to forskellige hashværdier for den samme meddelelse for to forskellige nøgler[5]. En MAC er derfor specielt nyttig til validering af ikke hemmeligholdte data. F.eks. kan Alice sende Bob en fil samt en MAC af filen, selv om Eve skulle sidde i midten og lytte. Eve kan ikke sende Bob en anden fil samt en MAC for den forfalskede fil, da hun ikke kender nøglen som Alice og Bob bruger til at beregne MAC'en.

Sikkerheden i en MAC er altså i høj grad baseret på en nøgle og selvfølgelig på også MAC'ens længde. En MAC har yderligere den fordel, at den er modstandsdygtig over for offline fødselsesdagsangreb.

[5]Da en MAC er en mange-til-en funktion, kan man selvfølgelig være meget heldig og ramme en kollision

En nem måde at gøre en MDC til en MAC er ved at kryptere den. På samme måde kan man gøre en MAC til en MDC ved at offentliggøre nøglen.

2.8.1 MD5 & SHA-1

De to mest udbredte og analyserede MDC'er er MD5 og SHA-1. MD5, der er udviklet af Ron Rivest, er ifølge [Dai] over dobbelt så hurtig som SHA-1. Målt på en Pentium 4 2.1 GHz processor, kan MD5 beregne hashværdier med en hastighed på 204.6 Mbps, målt under tilsvarende betingelser har SHA-1 en hastighed på 72.6 Mbps.

Hvor MD5 beregner en hashværdi for en meddelelse på 128 bit, bliver hashværdien for meddelelsen, beregnet med SHA-1, på 160 bit. Der er fundet en mindre svaghed i MD5s kompressorfunktion som gør, at man kan få den til at lave kollisioner. Dog er der ikke fundet en måde at udvide angrebene til MD5 som helhed [BB94, Dob96]. Der er ikke på samme måde fundet svagheder i SHA-1. Den eneste kritik SHA-1 har fået, er over de hemmeligholdte designkriterier, som NSA har haft i udviklingen af algoritmen. Man kan sige, at hvis det skal gå hurtigt så brug MD5. Hvis det skal være (helt) sikkert så brug SHA-1.

2.8.2 CBC-MAC & HMAC

Hvad MAC'er angår, er der tre specielt udbredte, som vi vil beskrive i det følgende. Den første vi vil kigge på er CBC-MAC, som benytter et vilkårligt blokchiffer i CBC-mode (eng.: *Cipher Block Chaining Mode*). CBC-MAC kan være svær at bruge rigtigt, da den kun virker på meddelelser af kendt længde, men til gengæld er der lavet teoretiske beviser for, at CBC-MAC er sikkert, så længe det underliggende blokchiffer er sikkert [BKR94].

Der findes måder at løse problemet med meddelelser af variabel længde i CBC-MAC, men mere lovende er dog nogle nye modes, som er blevet udviklet i forbindelse med udgivelsen af AES. Disse nye modes er XCBC og OMAC, som kan bruges på meddelelser af variabel længde, og hvorom man kan bevise samme egenskab som for CBC-MAC [IK02]. Men da disse modes er meget nye, bør man udvise en vis forsigtighed. Denne type MAC'er er typisk også en del langsommere end andre typer MAC'er, da de er baseret på et blokchiffer.

Den anden MAC vi vil kigge på, er den såkaldte HMAC, som er en generel måde at forvandle en MDC til MAC. En HMAC beregnes ud fra en MDC h samt to konstanter a og b, samt en nøgle K, på følgende måde:

$$h(K \oplus a \parallel h(K \oplus b) \parallel m) \tag{2.11}$$

Hvor m er meddelelsen der skal hashes. HMAC er enkel, effektiv og nem at bruge [FS03], og der er ingen angreb, så vidt vi ved, mod HMAC.

Den sidste mulighed vi vil beskrive, er kryptering af MDC'en med en krypteringsalgoritme. Kombinationen af MDC og krypteringsalgoritme kan ikke vælges ukritisk jf. 3.3.2; men når det er sagt, så vil sikkerheden i høj grad afhænge af krypteringsalgoritmens sikkerhed, hvorfor det er vigtigt at vælge en gennemprøvet og gennemanalyseret algoritme. Fordelen ved denne form for MAC er, at hvis man både vil sikre konfidentialitet og integritet, kan man kryptere data og MDC'en med samme algoritme og nøgle. Det er vigtigt i den sammenhæng at forstå, at man på den måde har bundet sikkerheden af konfidentialiten og integriteten sammen.

2.9 Kryptografisk adgangskontrol

Klassiske styresystemer som Unix og Windows NT benytter sig af en adgangsmatrice (eng.: *access matrix*) til at håndhæve rettigheder mellem brugere og objekter [Sta03], hvor objekter kan være alt fra filer til devices. Typisk implementeres dette ved enten adgangskontrollister (eng.: *Access Control Lists*) eller tilladelseslister (eng.: *capability lists*), som svarer til henholdsvis kolonnerne og rækkerne i adgangsmatricen. Unix implementerer ACL på filer ved de klassiske Read/Write/Execute rettigheder for owner/group/everyone. Der er dog ingen grænser for, hvilke rettigheder og brugere en ACL kan indeholde; det skal bare være understøttet i systemet. I Windows NT har man udvidet antallet af rettigheder, og man kan tildele rettigheder til både grupper og enkelte brugere. For at ACL'er virker, skal der være en referencemonitor, som håndhæver ACL'en. Dette er typisk styresystemet selv.

Det er vigtig at bemærke, at ACL kun giver sikkerhed, så længe objekterne ikke forlader systemets referencemonitor eller det domæne, som referencemonitoren overvåger. F.eks. vil man kunne tage en harddisk fra et system, hvor man ikke har adgang og sætte den i et system, hvor man har root/administrator adgang, og man vil have adgang til alle filer. Ydermere sikrer referencemonitoren kun filernes konfidentialitet og integritet, mens de er i systemet og ikke mens de bliver sendt over et netværk [HJ03].

Et alternativ til ACL, men som sikrer konfidentialitet og integritet af data både inden- og udenfor et bestemt domæne samt under transporten mellem domæner, er kryptografisk adgangskontrol (eng: *cryptographic access control*) [HJ03]. Kryptografisk adgangskontrol opnår dette ved at fjerne behovet for en referencemonitor, ved anvendelse af asymmetrisk kryptografi, når data bliver gemt på systemet. Kryptografisk adgangskontrol opererer med to rettigheder; læse og skrive. Til opretholdelse af læserettigheden krypteres data med den private nøgle, og læserettigheden begrænses til de brugere, der kender den offentlige nøgle. På samme måde opnår man skriverettighed, hvis man kender den private nøgle, for uden den kan man ikke skrive filer, som kan læses af brugere med den offentlige nøgle.

Den asymmetriske kryptografi sikrer konfidentialitet og autentifikation, men ikke integritet og tilgængelighed, hvis dele af eller hele filen overskrives. Fordi asymmetrisk kryptografi er flere gange langsommere end symmetrisk kryptografi, benyttes symmetrisk kryptografi til kryptering af selve filen, mens asymmetrisk kryptografi benyttes til at signere filen. Dette løser også integritetsproblemet, da der under signeringen bliver lavet en hash af filen, der senere kan bruges til verifikation. Der er nu tre nøgler, en symmetrisk nøgle og en offentlig og privat asymmetrisk nøgle. For at læse og verificere filen kræves den symmetriske nøgle samt den offentlige asymmetriske nøgle. For at skrive en fil kræves den symmetriske nøgle til selve krypteringen, samt den private asymmetriske nøgle til signeringen af filen.

Hvis en klient vil gemme en fil på en server, opretter klienten de tre nøgler, hvorefter filen krypteres med den symmetriske nøgle. Der beregnes en hash af den krypterede fil, der krypteres med den private nøgle. Så sendes den krypterede fil, den krypterede hash og den offentlige nøgle til serveren. Når serveren modtager dette, beregner den selv en hash af den krypterede fil, som den sammenligner med den hash, der blev sendt sammen med filen.
Med denne procedure sikrer serveren både autentifikation og integritet af filen. Alle klienter kan nu hente filen, men kun dem som kender den symmetriske nøgle kan læse filen. Har man også den offentlige nøgle, kan man kontrollere filens integritet.
Hvis en klient vil skrive til en fil, så sender klienten den nye version af filen til serveren, der udfører proceduren som for en helt ny fil. Hvis den er succesfuld, tjekker serveren om de to offentlige nøgler, for hver af de to versioner af filen, er ens. Hvis det er tilfældet accepteres den nye version.

2.10 Opsummering

I dette kapitel er en række peer-to-peer systemer blevet beskrevet. Systemerne har vist sig at være ganske forskellige.

Vi har for hvert af systemerne undersøgt den grundlæggende arkitektur, hvordan lokaliseringen af data foregår, samt hvordan systemet håndterer nye noder. Peer-to-peer systemerne er blevet klassificeret efter deres placering af ressourcer, placering af metadata, samt krav til tilstandskontrol.

Det viser sig at placeringen af metadata opdeler de betragtede systemer i tre grupper: I Napster og NFS er metadata placeret på centrale servere. PAST, Freenet og Gnutella overlader det helt til de enkelte noder at organisere metadata.FastTrack og JXTA repræsenterer en interessant klasse af hybridsystemer, hvor metadata fordeles på dynamisk udnævnte super- eller rendezvousnoder. Det gør det muligt at foretage globale søgninger, uden at man er afhængig af centrale servere. Desuden betyder hierarkiet blandt noderne, at systemerne har gode muligheder for at skalere til et stort antal noder.

Med hensyn til placering af data, kan systemerne deles i to grupper: I fildelingsystemerne opbevarer man data selv. Andre noder der er interessede i disse data, kan hente dem og vil herefter typisk gøre dem tilgængelige for resten af netværket. I disse systemer er der ingen garanti for, at data altid er tilgængelige, men populære data vil sandsynligvis være det.

I PAST, NFS og delvist i Freenet er data placeret på bestemte noder. Det stiller krav om en vis offervilje hos de enkelte noder. Man skal være villig til at stille lagerplads til rådighed for andre, og systemerne fungerer bedst, hvis noderne altid er tilsluttet netværket.

Til sidst i kapitlet har vi forsøgt at give et overblik over forskellige symmetriske og asymmetriske chifres sikkerhed og performance. Vi har kigget på, hvad hashfunktioner er, og hvilke krav man bør stille til dem. Elementære og udbredte angreb på hashfunktioner er forklaret. Ydermere har vi nævnt fordele og ulemper for udbredte hashfunktioner.
Endelig er det blevet forklaret, hvordan man ved hjælp af kryptografisk adgangskontrol, kan indføre læse- og skriverettigheder på filer, uden brug af en referencemonitor,

Kapitel 3

Analyse

I dette kapitel vil der på baggrund af forskellige scenarier blive opstillet en kravspecifikation. Denne kravspecifikation bliver brugt til at udvikle en sikkerhedsmodel, der så efterfølgende analyseres med henblik på at identificere eventuelle svagheder.

3.1 Applikationsmodel

I dette afsnit vil vi opstille en applikationsmodel for vores system, hvor modellen vil tjene som udgangspunkt og reference for det videre design af systemet. To scenarier vil blive beskrevet. Heraf udledes de opgaver, som vores system skal kunne løse.

3.1.1 Udgivelse

Udgivelse af digitalt materiale kan være en stor byrde, hvis antallet af modtagere er højt. Det skyldes primært de omkostninger, der er forbundet ved overførelsen af det digitale materiale. Som et eksempel kan nævnes Kazaa, verdens mest downloadede program, som er blevet downloadet mere end 230 millioner gange, og som dagligt bliver downloadet mere end 360.000 gange [Reu03]. Dette giver en daglig traffik på mere end 1.800 Gb data[1]. For at få en fornemmelse af denne størrelse, så svarer det til trafikken som ca. 700 ADSL linjer på 256 Kbit/s kan sende på 24 timer. Det er med andre ord ikke omkostningsfrit at sende datamængder af denne størrelsesorden.

Problemet kan løses ved at benytte sig af et peer-to-peer netværk, som f.eks. Kazaa eller Gnutella til distributionen. Disse peer-to-peer netværk har dog visse mangler:

[1] Kazaa fylder ca. 5 Mb.

Tilgængelighed: Når noget først er tilgængeligt på netværket, så er det tilgængeligt for alle. Dette er langt fra altid ønskværdigt set fra en udbyders synspunkt, der måske gerne vil have, at brugere skal registrere sig, før de kan benytte sig af udbyderens tjenester.

Opdatering: Der er ikke mulighed for automatisk annoncering af opdateringer eller nye versioner af udgivelsen, hvilket ville være naturligt for udbydere af software og andet digitalt materiale.

Autentifikation: Man kan ikke være sikker på, at det materiale man henter på netværket, kommer fra den oprindelige udbyder. F.eks. er det vigtigt, at brugere kan verificere, at et program kommer fra det officielle firma, og ikke er blevet modificeret til at indeholde bagdøre eller anden ondartet kode.

3.1.2 Kollaborativt arbejde

Brugen af samarbejdsværktøjer på tværs af organisationer eller mellem forskellige afdelinger indenfor samme organisation, kan ofte være problematisk, fordi værktøjerne tit kræver en central koordinerende server, eller ikke er designet med henblik på at skulle operere i et potentielt fjendtligt miljø. Disse problemer bliver specielt presserende i virtuelle organisationer, hvor der ikke er noget centralt koordinerende kontor.

Man kunne f.eks. forestille sig en gruppe af tekstforfattere, grafikere og layoutere samarbejde om at lave en publikation. Alle kommer fra forskellige organisationer. En fælles IT-platform som kan understøtte samarbejdet er således ikke tilstede. Løsningen er ofte, at parterne benytter e-mail til at sende filer til hinanden eller, at parterne giver hinanden adgang til deres respektive interne systemer.

Begge løsninger er problematiske; e-mail er en ineffektiv måde at transportere data på. Desuden er der ofte begrænsninger på den datamængde, det er muligt at sende. Man kan heller ikke være sikker på, at afsenderen husker at sende den opdaterede fil til alle de relevante modtagere. Der mangler således et gruppebegreb, der kan sikre, at alle i arbejdsgruppen har adgang til de samme ressourcer.

Mange organisationer har ikke mulighed for at give differentieret adgang til deres interne systemer og er derfor, forståeligt nok, tilbageholdende med at give eksterne samarbejdspartnere adgang. Desuden er det ofte forbundet med en del administration at oprette og fjerne brugerkonti.

Når flere brugere har mulighed for at arbejde med de samme filer, vil det ofte været nyttigt med en form for versionsstyring. På den måde kan man se, hvem der har foretaget ændringer og måske også, hvilke ændringer der er sket. Eksisterende serverbaserede systemer, såsom CVS [Sys] og SourceSafe [Sou], giver mulighed for at se en fils revisionshistorie, ligesom tidligere udgaver af filen også kan genskabes.

I SourceSafe skal en fil tjekkes ud, førend den kan redigeres. I den forbindelse bliver filen låst således, at andre brugere ikke også kan redigere filen. Denne metode sikrer, at flere brugere ikke opdaterer en fil samtidigt, men giver naturligvis problemer, hvis brugerne ikke tjekker filerne ind igen.

Et versionsstyringssystem forhindrer ikke nødvendigvis konflikter mellem samtidige opdateringer. I CVS låses filerne typisk ikke. I stedet forsøges brugernes ændringer flettet sammen med den udgave af filen, der måtte findes på serveren. Hvis der er konflikt mellem to ændringer, f.eks. hvis to brugere har rettet i den samme linje i samme fil, præsenteres den sidste bruger for konflikten, der så skal løses manuelt, førend den sidste brugers opdatering accepteres. Dette kan selvfølgelig kun lade sig gøre for tekstfiler, eller andre filtyper hvor en sammenfletning giver mening.
CVS tilbyder også en mellemløsning, hvor brugerne kan undersøge, om andre brugere er i gang med at redigere en bestemt fil. Dette forhindrer ikke brugeren i også at redigere filen, men problemet med opdateringer, der er i indbyrdes konflikt kan på den måde mindskes.

3.1.3 Kravspecifikation

Som en løsning på disse problemer foreslår vi et fildelingsværktøj baseret på peer-to-peer teknologi og kryptografisk adgangskontrol. I det følgende vil vi give en overordnet beskrivelse af funktionaliteten.

Det skal være muligt at dele filer. Til de delte filer knyttes læse- og skriverettigheder. Når filen deles, skal det være muligt for brugeren at angive hvilke andre brugere, som skal have rettigheder i forhold til filen.
Der skal være et gruppebegreb således, at det er muligt at tildele de samme rettigheder til alle medlemmer af en bestemt gruppe. Der skal dynamisk kunne oprettes grupper og tilføjes brugere til disse. En bruger skal kunne være medlem af flere forskellige grupper.
Grupperne skal beskyttes på samme måde som filer således, at brugere ikke kan rette i dem, uden at have de fornødne rettigheder.

For en bruger skal det være muligt at undersøge, hvilke filer andre brugere har givet ham adgang til. Når en fil er fundet, skal den kunne hentes fra en vilkårlig anden bruger som måtte have filen.

En bruger skal kunne checke autenticitet og integritet af filerne på netværket. Ydermere skal systemet til en vis grad sikre tilgængelighed af filerne på netværket.

Hvis en bruger ændrer en fil, som vedkommende har skriverettighed til, skal det være muligt at dele den opdaterede udgave af filen således, at andre brugere, der har adgang til filen, kan se, at en ny version af den oprindelige fil nu er tilgængelig. Det skal i den forbindelse være muligt at følge en fils revisionshistorie således, at man kan se hvornår, og måske også af hvem, filen er blevet redigeret.

Det skal være muligt for en bruger at tilgå netværket fra en vilkårlig node, og netværket skal kunne håndtere en høj grad af udskiftning blandt noderne.

Mængden af kontroldata, der sendes i forbindelse med distribution og søgninger, skal være begrænset af hensyn til brugere med langsomme netværksforbindelser.

3.2 Sikkerhedsmodel

Formålet med dette afsnit er at give en overordnet beskrivelse af den sikkerhedsmodel, der er blevet udviklet med henblik på at opfylde sikkerhedskravene opstillet i afsnit 3.1.3.

3.2.1 Filer & Nøgler

Til hver fil i systemet hører et offentligt/privat-nøglepar (KU/KR), samt en nøgle brugt til den symmetriske kryptering af filen (K_{sym}). Disse tre nøgler udgør et komplet nøglesæt. Man behøver ikke altid at være i besiddelse af alle tre nøgler. Afhængig af hvilke af nøglerne man har, er der tre forskellige niveauer af rettigheder:

- Integritet (KU)
- Integritet og læserettigheder (K_{sym}, (KU))
- Integritet, læse- og skriverettigheder (K_{sym}, (KU, KR))

I det følgende vil et mere eller mindre komplet nøglesæt blive benævnt som en filnøgle eller en brugernøgle afhængig af anvendelsen. Henvises der til specifikke nøgler på nøglesættet, vil disse blive benævnt som henholdsvis den offentlige nøgle (KU), den private nøgle (KR) og den symmetriske nøgle (K_{sym}).

Hver bruger af systemet har et personligt nøglesæt, der bl.a. benyttes til kryptering af en personlig nøglering. Den personlige nøglering benyttes til opbevaring af nøgler til de filer, brugeren har adgang til. Brugen af nøgleringe er beskrevet i afsnit 3.2.2.

Når en fil tilføjes, dannes der først en hash af filen ($H(file_n)$), denne krypteres sammen med selve filen med den symmetriske nøgle ($E_{K_{sym}}(file_n, H(file_n))$). Hashen benyttes ved dekryptering til at sikre, at filen er dekrypteret korrekt.

$$d_n = E_{K_{sym}}(file_n, H(file_n))$$

Tabel 3.1: Data

Logiske filer identificeres ved en hash af den offentlige filnøgle. Denne ændrer sig ikke på tværs af forskellige revisioner.
Bestemte revisioner af filen identificeres med en hash af de krypterede data, $H(d_n)$, hvor n er den pågældende version.

Til filen hører en log, der indeholder signaturer af filen. Formålet med revisionsloggen er at gøre det muligt at identificere, hvornår eventuelle konflikter (uafhængige opdateringer af samme revision) er opstået, samt at dokumentere filens revisionshistorie.

$$\left.\begin{array}{ll} Rev.\ 1: & H(d_1), S_{KR}(d_1), S_{U_{KR}}(d_1), U_{KU} \\ Rev.\ 2: & H(d_1), H(d_2), S_{KR}(d_2) \\ \ldots & \\ Rev.\ n: & H(d_{n-1}), H(d_n), S_{KR}(d_n), [S_{U_{KR}}(d_n), U_{KU}] \end{array}\right\} E_{KR}(\ldots, H(\ldots))$$

Tabel 3.2: Revisionslog

Når en ny fil oprettes eller en gammel opdateres, dannes der en tilføjelse til loggen. Dette gøres ved først at beregne en hash af den krypterede fil ($H(d_n)$), der så signeres med filens private nøgle ($S_{KR}(d_n)$). Hvis der er tale om en opdatering, inkluderes desuden hashen af den revision, som den nye revision bygger på ($H(d_{n-1})$).

Dernæst kan brugeren vælge at signere filen igen med sin private nøgle ($S_{U_{KR}}(d_n)$). På den måde kan filen blive signeret to gange. Først garanteres det, at ændringen er foretaget af en, som er i besiddelse af den private filnøgle, dernæst identificeres vedkommende, som har lavet opdateringen. I ovenstående eksempel er den første version af filen signeret med brugerens offentlige nøgle, den anden version er ikke. Da asymmetrisk kryptering er forholdsvis beregningstungt, kan det være en fordel for klienter med begrænsede processorressourcer at undlade at signere opdateringen med brugerens private nøgle.

Brugerens offentlige nøgle er desuden vedhæftet, så andre brugere, der ønsker at kontrollere brugersignaturen, kan gøre dette (U_{KU}). Hvis ikke man kender den offentlige nøgle i forvejen, kan man selvfølgelig ikke være sikker på identiteten af vedkommende, der har signeret opdateringen.

Hele revisionsloggen krypteres med filens private nøgle ($E_{KR}(\ldots, H(\ldots))$) således, at filens revisionshistorie ikke kan manipuleres af brugere, som kun har integritets- eller læserettigheder. Desuden kan brugere, der ikke har den offentlige nøgle ikke følge med i revisionshistorien, hvilket besværliggør trafikanalyse. Da public-key kryptering er meget beregningstungt i forhold til symmetrisk kryptering, benyttes public-key krypteringen udelukkende til at kryptere en symmetrisk nøgle, der så anvendes til at kryptere selve revisionsloggen. En nærmere forklaring af denne mekanisme er givet i afsnit 5.2.2.

3.2.2 Nøgleringe

En nøglering benyttes til at opbevare nøglesæt. Disse nøglesæt kan være filnøgler, brugernøgler eller nøgler til andre nøgleringe. En nøglering behandles på samme måde som en almindelig fil. Indholdet af filen er blot en liste af nøgler. For at kunne læse eller opdatere en nøglering, skal man derfor være i besiddelse af den relevante filnøgle.

Når man ønsker at give andre brugere adgang til en fil, gøres dette typisk ved at placere filnøglerne på en fælles nøglering. Det kan være en eksisterende nøglering, som man har skriverettigheder til, eller man kan oprette en ny. Hvis det sidste er tilfældet, vil det naturligvis være nødvendigt at distibuere filnøglerne til nøgleringen til de brugere, der skal have adgang.

Hvis man ønsker at give en gruppe af brugere læserettigheder samt at give en delmængde af disse skriverettigheder, vil det være nødvendigt at benytte to forskellige nøgleringe således, at de relevante filnøgler bliver krypteret med forskellige offentlige nøgler.

Som alternativ til nøgleringe kan man vælge at distribuere filnøgler enkeltvis gennem andre sikre kanaler, f.eks. krypteret e-mail. Modtager man en filnøgle på den måde, kan man overføre den til sin personlige nøglering. Filnøgler kan anvendes på samme måde, uanset hvilken nøglering de placeres på. Fremgangsmåden kan være en fordel, hvis der kun skal deles en enkelt fil, eller hvis man endnu ikke har etableret en fælles nøglering.

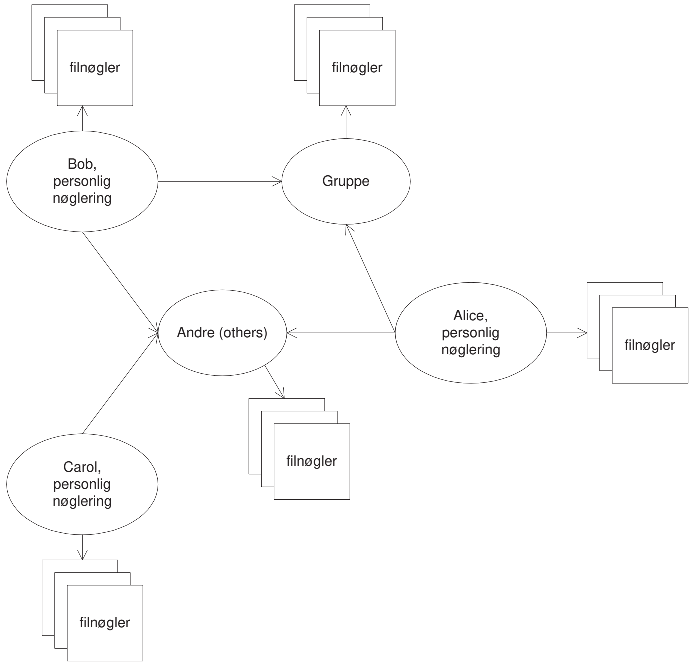
I eksemplet ovenfor kunne man f.eks. forestille sig, at gruppen af brugere, som skulle have skriveadgang var ganske lille, men gruppen, der kun skulle have læseadgang var stor. I det tilfælde kan det være en fordel at placere læsenøglerne på en fælles nøglering, men nøjes med at sende skrivenøglerne via krypteret e-mail.

På de følgende figurer svarer en cirkel til en nøglering og en firkant til en filnøgle eller en brugernøgle. En pil der starter på en nøglering betyder, at nøglesættet til det objekt der peges på, findes på pågældende nøglering.

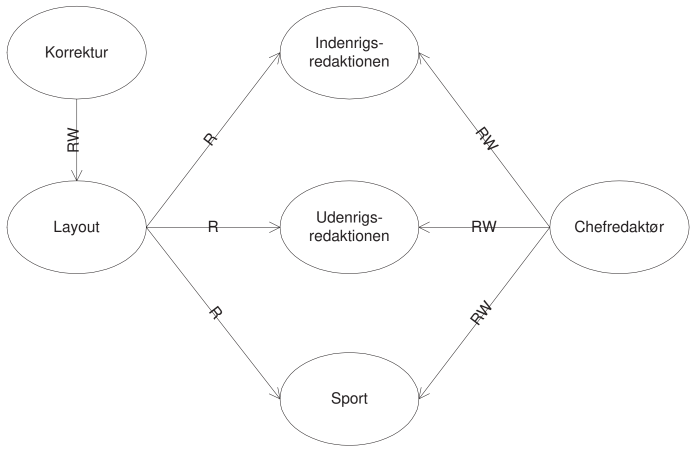
Filnøgler på nøgleringe kan frit kopieres til andre nøgleringe eller sendes til andre brugere. Det betyder, at selv om en filnøgle fjernes fra en nøglering, kan brugere, der har haft adgang til nøgleringen, stadig få adgang til filerne, enten via en ældre version af nøgleringen eller fordi de har kopieret filnøglen fra nøgleringen. Brugere der ikke har adgang til nøgleringen kan få adgang, hvis de får overdraget filnøglerne af andre brugere, der har adgang til nøgleringen.

På figur 3.2 har hver redaktion en nøglering. Chefredaktørens nøglering indeholder både læse- og skrivenøglerne til redaktionernes nøgleringe, mens layouterens nøglering kun indeholder læsenøglen. Korrekturlæseren har både læse- og skrivenøglerne til de layoutede artikler, og de placeres på layoutnøgleringen. I modsætning til normale RBAC-miljøer kan brugerne frit overdrage nøgler, og dermed roller, til hinanden.

Muligheden for at overdrage nøgler betyder, at sikkerhedsmodellen ikke vil være egnet i MAC-miljøer (MAC - Mandatory Access Control). Da disse systemer netop bruges hvis man vil forhindre, at en bruger videregiver klassificeret information til andre brugere med færre rettigheder.



Figur 3.1: Eksempel på brug af nøgleringe i et "Discretionary Access Control"-miljø (DAC). Alle har rettigheder til "others"-ringen, men kun Alice og Bob har adgang til "Gruppe"-ringen. Denne måde at fordele rettigheder på, svarer til hvad man finder i de fleste Unix systemer.



Figur 3.2: Eksempel på brug af nøgleringe i et "Role Based Access Control"-miljø (RBAC). Roller tildeles ved at give adgang til bestemte nøgleringe.

3.2.3 Tilladelser

Som det fremgår, kan filnøgler overdrages gennem nøgleringe eller alternative sikre kanaler. Et tredje alternativ er at oprette en *tilladelse*. Denne mulighed kan benyttes hvis man er i besiddelse af modtagerens offentlige nøgle. Er man det, krypteres filnøglen blot med modtagerens offentlige nøgle. Den resulterende fil kan uden risiko gøres offentligt tilgængelig, da det kun er brugeren ,som har den tilsvarende private nøgle, der kan dekryptere filnøglen. Tilladelser identificeres med en hash af brugerens offentlige nøgle, kombineret med en hash af den offentlige filnøgle. En bruger kan derfor opdage nye tilladelser til sig selv, ved at foretage en søgning efter en hash af sin offentlige nøgle.

Grunden til, at der benyttes en hash af den offentlige nøgle, i stedet for nøglen selv, er, at man på den måde undgår direkte at afsløre, hvem der har adgang til en bestemt filnøgle. Men nysgerrige personer kan naturligvis stadig undersøge, om der er givet en tilladelse til en bestemt offentlig nøgle.

Ulempen ved tilladelser er, at de ikke automatisk bliver spredt i netværket på samme måde som nøgleringe og almindelige filer. Årsagen er, at en tilladelse selvsagt kun er interessant for den bruger der kan dekryptere den. Til gengæld kan tilladelser godt distribures via usikre kanaler, f.eks. almindelig e-mail eller en hjemmeside. Så hvis man ønsker at sikre, at en tilladelse er tilgængelig, kan man uden konsekvenser for sikkerheden benytte en af disse metoder. Men man kan selvfølgelig også vælge at have tilladelsen liggende på en node, der altid er tilsluttet netværket.

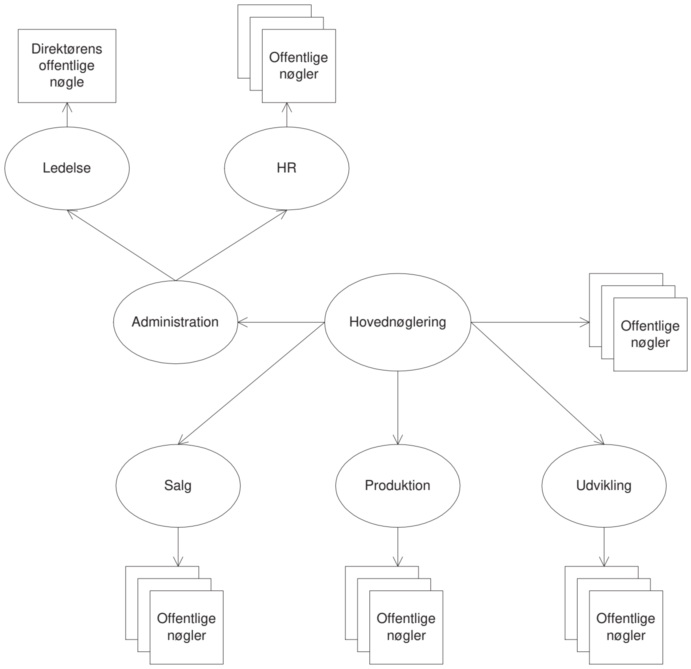
3.2.4 Nøgleringe som certifikatautoriteter

Nøgleringe kan som tidligere nævnt indeholde alle typer nøgler. Forestiller man sig en nøglering, der udelukkende indeholder brugernøgler, vil de brugere, der har skriveadgang til nøgleringen, kunne fungere som certifikatautoriteter. Nye brugere giver deres offentlige nøgle til en af de brugere, der har skriveadgang. Denne kontrollerer brugerens identitet efter en fastlagt sikkerhedspolitik. Kun hvis den nye bruger er den, som han giver sig ud for at være, tilføjes brugerens nøgle til nøgleringen.

Denne mekanisme kan fungere hierarkisk. F.eks. kan man forestille sig, at et firma opretter en hovednøglering.
Hver afdeling i firmaet opretter desuden en afdelingsnøglering. Når medarbejdere tilknyttes eller forlader afdelingen, opdateres afdelingens nøglering. Efterhånden som afdelingerne oprettes og nedlægges, tilføjes og fjernes afdelingernes nøgleringe fra hovednøgleringen.
Den enkelte medarbejder behøver kun at have nøglerne til hovednøgleringen for at få adgang til samtlige medarbejderes offentlige nøgler.

På figur 3.3 gives et eksempel med et firma med fire afdelinger: Salg, Produktion, Udvikling og Administration. Administrationsafdelingen består af to dele: Ledelse og Personale (HR - Human Resources). Skrivenøglerne til hovednøgleringen placeres f.eks. på Ledelses-nøgleringen.

Et lignende hierarki kan benyttes til distribution af nøgler til filer og almindelige nøgleringe.



Figur 3.3: Eksempel på brug af nøgleringe som certifikatautoriteter i et firma med forskellige afdelinger.

Disse hierarkier kan have vilkårlig dybde, men sikkerheden er naturligvis afhængig af, at nøgleringsadministratorerne, på hvert niveau, passer godt på den private nøgle til nøgleringen, og desuden er påpasselige med, hvilke offentlige nøgler de godkender. Dette betyder, at risikoen for at finde en falsk brugernøgle vil stige, jo længere ned i hierarkiet man bevæger sig - alt andet lige. Brugeren, der efterspørger en offentlig nøgle, må have tillid til hvert mellemliggende niveau i hierarkiet, for at han kan have tillid til de offentlige nøgler, han finder på et givent niveau.

Den samme offentlige nøgle kan optræde på nøgleringe i forskellige hierarkier. Dette giver mulighed for at akkumulere tilliden til, at en bestemt offentlige nøgle tilhører en bestemt bruger, således kan der dannes et “netværk af tillid” (eng.: *web of trust*), som det kendes fra PGP [Sta03]. Beslutninger om hvilke nøgler og nøgleringe man kan have tillid til, er dog i vores system overladt til brugeren.

3.3 Trusler

Sikkerhedsmodellen opstillet i foregående afsnit vil i dette afsnit først blive vurderet i forhold til de klassiske krav om konfidentialitet og integritet. Dernæst vil mulige oversvømmelsesangreb (eng.: *Denial of Service (DoS)*), samt angreb baseret på trafikanalyse, blive gennemgået sammen med mulige forholdsregler mod disse angreb.

3.3.1 Trusler mod konfidentialitet

Hemmeligholdelse af den symmetriske nøgle er en forudsætning for bevarelsen af konfidentialitet. Hvis en node, der har filnøglerne til en bestemt fil, bliver kompromitteret, vil det være nødvendigt, at samtlige andre noder, der også har adgang til filen, udskifter deres nøgler. Det kan desuden være svært at fastslå, hvilken node der er blevet kompromitteret, og derfor ikke skal have adgang til den nye nøgle. Samme situation kan opstå, hvis en bruger videregiver en filnøgle til tredje part. Dette kan dog ikke betragtes som en sikkerhedsbrist, men snarere som en konsekvens af sikkerhedsmodellen, og den måde kryptografisk adgangskontrol fungerer på.

Med kompromiteret menes der, at der opnås adgang til den dekrypterede personlige nøglering. Dette kan f.eks. ske, hvis der installeres en keylogger eller andet skjult overvågningssoftware.

Hvis en filnøgle skal udskiftes, må man oprette en helt ny fil med ny offentlig nøgle. Den gamle fil kan sagtens fortsat være tilgængelig på netværket. Den genoprettede konfidentialitet gælder altså kun efterfølgende opdateringer af filen.

Da filnøgler som udgangspunkt ikke ændrer sig, er det vigtigt at vælge krypteringsalgoritme og nøglelængde således, at langvarende brute-force angreb har lille sandsynlighed for at lykkes.

3.3.2 Trusler mod integritet

Kryptering af en fil sammen med dens hashværdi, som beskrevet i afsnit 3.2.1, garanterer ikke nødvendigvis integritet. Seneste eksempel på dette er Wired Equivalent Privacy (WEP) protokollen, som er en del af 802.11 standarden [BGW01]. WEP bruges i 802.11 til at sikre konfidentialitet og integritet af pakkerne, der sendes i det trådløse netværk. Vi vil derfor kort beskrive, hvorfor det er muligt at kompromittere integriten i WEP, og hvad vi gør for at undgå samme type angreb i vores system.

WEP bruger RC4 til at kryptere beskeder M samt checksummen af beskeden $c(M)$. RC4 er et stream-chiffer, som genererer en bitstrøm, der XOR's på klarteksten, og dermed skaber kryptoteksten C. RC4 skal bruge en initialiseringsvektor v samt en hemmelig nøgle k for at generere bitstrømmen. Bitstrømmen, der skabes med initialiseringsvektoren v og den hemmelige nøgle k, noteres som RC4(v,k). En pakke, der sendes med 802.11, og som benytter WEP, vil derfor se således ud:

$$\langle v, C\rangle \tag{3.1}$$

Først kommer initialiseringsvektoren efterfulgt af kryptoteksten C dannet af beskeden samt checksummen XOR'et med bitstrømmen fra RC4(v,k). Kryptoteksten C er altså dannet som angivet i (3.2)

$$C = \mathrm{RC4}(v,k) \oplus \langle M, c(M)\rangle \tag{3.2}$$

WEP benytter sig af checksumsfunktionen CRC32 til at beregne checksummer til beskederne, som så skal sikre integriteten. Integriteten sikres ved, at en modtager dekrypterer pakken, beregner CRC32 af den dekrypterede besked og sammenligner med checksummen fra pakken. CRC32 har dog den uheldige egenskab, at den er lineær, hvilket vil sige, at der gælder følgende:

$$c(x \oplus y) = c(x) \oplus c(y) \tag{3.3}$$

Denne egenskab samt det, at RC4 er lineær gør, at man kan kompromittere integritet i WEP. Når vi siger, at RC4 er lineær mener vi, at det at lave en ændring (Δ) i klarteksten (M) og så kryptere med RC4, er det samme som at kryptere klarteksten med RC4 og lave den samme ændring i kryptoteksen. Med andre ord: $(M \oplus \Delta) \oplus \mathrm{RC4}(v,k) = (\mathrm{RC4}(v,k) \oplus M) \oplus \Delta$. Når man vil kompromittere integriteten laver man en ny kryptotekst C', der dekrypterer til $\langle M', c(M')\rangle$, hvor:

$$M' = M \oplus \Delta \tag{3.4}$$

altså den originale besked M XOR'et med en vilkårlig ændring Δ.

Der mangler så bare en forklaring på, hvordan man beregner C' udfra C, så den nye kryptotekst C' dekrypterer til $\langle M', c(M')\rangle$ i stedet for til $\langle M, c(M)\rangle$. For at få C' XOR's C med $\langle \Delta, c(\Delta)\rangle$. Vi giver nu et bevis fra [BGW01] på, at C' har den ønskede egenskab:

$$\begin{array}{rcll}
C' & = & C \oplus \langle \Delta, c(\Delta)\rangle & \text{Startantagelse} \\
 & = & \mathrm{RC4}(v,k) \oplus \langle M, c(M)\rangle \oplus \langle \Delta, c(\Delta)\rangle & 3.2 \\
 & = & \mathrm{RC4}(v,k) \oplus \langle M \oplus \Delta, c(M) \oplus c(\Delta)\rangle & \text{XOR egenskab} \\
 & = & \mathrm{RC4}(v,k) \oplus \langle M', c(M) \oplus c(\Delta)\rangle & 3.4 \\
 & = & \mathrm{RC4}(v,k) \oplus \langle M', c(M \oplus \Delta)\rangle & 3.3 \\
 & = & \mathrm{RC4}(v,k) \oplus \langle M', c(M')\rangle & 3.4
\end{array}$$

Vi kan nu se, at C' har den ønskede egenskab nemlig, at den dekrypterer til $\langle M', c(M')\rangle$:

$$\begin{aligned} C' \oplus \mathrm{RC4}(v,k) &= \mathrm{RC4}(v,k) \oplus \mathrm{RC4}(v,k) \oplus \langle M', c(M')\rangle \\ &= \langle M', c(M')\rangle \end{aligned}$$

Vi kan derfor lave vilkårlige ændringer i den originale besked M uden, at det opdages, hvilket er et klart brud på integriteten. Man skal dog være opmærksom på, at man ikke kan vide, hvad M' indeholder, kun hvilke ændringer der er i forhold til M. Et eksempel kunne være, at man vil "flippe" den første bit i M, man sætter derfor $\Delta = 100\ldots00$. Vi ved hverken, hvad der står i M eller M' kun, at den første bit i M er "flippet" i forhold til M'.

Hvis forfatterne til WEP havde valgt deres hashfunktion udfra kriterierne beskrevet i afsnit 2.8, ville de aldrig have valgt CRC32. Det skyldes, at CRC32 ikke opfylder kravene til at være svagt kollisionsfri eller stærkt kollisionsfri, når man bare kender en vilkårlig kollision. Det vil nu blive vist, at CRC32 hverken er svagt eller stærkt kollisionsfri, når en kollision er kendt:

Bevis 1 (CRC32 er ikke stærkt kollisionsfri)
forudsætning: Vi kender x og y hvorom der gælder $c(x) = c(y)$, hvor x $\neq$ y.

At bevise: Vi kan nu lave endnu en kollision ved at vælge en vilkårlig besked z og beregne to nye beskeder $x \oplus z$ og $y \oplus z$ der vil kollidere. Med andre ord skal vi bevise, at $c(y \oplus z) = c(x \oplus z)$ samt, at $(y \oplus z) \neq (x \oplus z)$.

Bevis:
Så længe $z \neq 0$ og $x \neq y$ vil det gælde at $(x \oplus z) \neq (y \oplus z)$, hvilket er en generel engenskab ved XOR.

$$\begin{aligned} c(y \oplus z) &= c(y) \oplus c(z) && 3.3 \\ &= c(x) \oplus c(z) && \text{forudsætning} \\ &= c(x \oplus z) && 3.3 \end{aligned}$$

Bevis 2 (CRC32 er ikke svagt kollisionsfri)
forudsætning: Vi kender x og y hvorom der gælder $c(x) = c(y)$, hvor x $\neq$ y.

At bevise: Vi kan nu beregne en kollion k til en bestemt besked z på følgende måde $k = x \oplus y \oplus z$. Med andre ord skal vi bevise, at $c(x \oplus y \oplus z) = c(z)$ samt, at $(x \oplus y \oplus z) \neq z$.

Bevis:
Så længe $z \neq 0$ og $x \neq y$ vil det gælde, at $(x \oplus y \oplus z) \neq z$, hvilket er en generel engenskab ved XOR.

$$\begin{aligned} c(x \oplus y \oplus z) &= c(x) \oplus c(y) \oplus c(z) && 3.3 \\ &= c(x) \oplus c(x) \oplus c(z) && \text{forudsætning} \\ &= 0 \oplus c(z) && \text{XOR egenskab} \\ &= c(z) && \text{XOR egenskab} \end{aligned}$$

For at undgå dette angreb er det nødvendigt at udskifte enten RC4 eller CRC32 med henholdsvis en ikke lineær krypteringsalgoritme eller en ikke lineær hashfunktion. Vi vil dog i vores system benytte både en ikke lineær krypteringsalgoritme og en ikke linæer hashfunktion, for at beskytte systemet mod eventuelle varianter af WEP angrebet.

3.3.3 Oversvømmelsesangreb

Et muligt oversvømmelsesangreb, der kan foretages mod den enkelte node, består blot i at skabe netværkstrafik ved at hente filer. På den måde kan man mætte nodens netværksforbindelse således, at der ikke er kapacitet til at servicere “rigtige” forespørgsler. Angriberen behøver ikke at kunne læse filerne, da det er tilstrækkeligt blot at kende hashværdien af filernes offentlige nøgler, for at kunne hente dem. Disse hashværdier kan f.eks. opsnappes ved at aflytte netværket, når noder, der er i besiddelse af filnøglerne, henter de tilhørende filer.

For at imødegå, at noder, uden de fornødne rettigheder, på den måde spilder netværkets kapacitet ved at hente filer, som de alligevel ikke kan dekryptere, kan følgende challenge/response-procedure benyttes ved filoverførsler:
Når en fil skal overføres, sender servernoden først et tilfældigt tal til klienten. Dette krypterer klienten med filens offentlige nøgle, og beregner derpå en hashværdi, der så sendes tilbage til servernoden. Servernoden foretager de samme beregninger og sammenligner sit eget resultat med det, som er modtaget fra klienten. Hvis de to resultater stemmer overens, sendes filen.

Grunden til, at det er en hashværdi af det krypterede tal og ikke tallet selv, der returneres er, at man på den måde undgår at sprede en masse klartekst/kryptotekst par, der måske kan udnyttes i et known-plaintext angreb.

Proceduren giver desværre mulighed for en ny type DoS-angreb, der går ud på at oversvømme servernoden med forespørgsler, og på den måde tvinge den til at foretage en masse beregningstunge krypteringsoperationer. I de fleste tilfælde vil båndbredden dog være en mere sparsom ressource end processortid. Derfor vil det give mening at spare båndbredde til gengæld for øget brug af processorressourcer. Desuden kan man i nogen grad beskytte sig mod denne type oversvømmelsesangreb, ved at lægge en pause ind mellem ekspeditionen af efterfølgende forespørgsler fra samme node. Dette beskytter selvfølgelig ikke mod DoS angreb rettet mod selve TCP-stakken på værtsoperativsystemet.

Et andet problem er muligheden for at foretage et man-in-middle-angreb: Mallory ønsker at hente en fil fra Alice, som hun ikke har rettigheder til. Hun modtager derfor et tilfældigt tal, som hun naturligvis ikke kan kryptere korrekt. Mallory laver derfor en falsk annoncering således, at andre noder tror, at Mallory er i besiddelse af filen. Hvis Bob så forsøger at hente filen fra Mallory, genererer Mallory ikke et nyt tilfældigt tal, men sender blot tallet fra Alice videre til Bob. Svaret fra Bob sender Mallory tilbage til Alice, der nu tror, at Mallory har rettigheder til filen.

På trods af denne mulighed, bliver det dog noget sværere for Mallory at udføre sit DoS-angreb mod Alice, da Bob eller andre noder skal gå i fælden, hver gang, Mallory anmoder Alice om filen.

Prototypeimplementeringen indeholder dog ikke, primært af tidsmæssige årsager, disse forholdsregler mod DoS angreb.

3.3.4 Trafikanalyse

Selvom revisionsloggen for en fil er krypteret, kan man stadig, alene ud fra størrelsen, gætte kvalificeret på antallet af revisioner. Det samme gælder selve filen. Ved at holde øje med hvordan filstørrelsen ændrer sig, kan man estimere antallet af opdateringer.

En mulig løsning på dette problem er at bruge padding, hvilket desværre nedsætter effektiviteten, da både de reelle data samt padding skal overføres, når filen hentes. Effekten ville også være begrænset, da man stadig kan skelne to forskellige versioner fra hinanden ved at sammenligne de to filer. En nysgerrig bruger kan altså holde øje med, hvor tit en fil bliver opdateret ved løbende at hente seneste version, og så sammenligne denne med de tidligere hentede filer. Så længe det er muligt at få fat i en kopi af filen, er det således svært at sikre sig mod trafikanalyse.

Ved at identificere filer med en hash af den offentlige nøgle, bliver det besværligt at hente filer, som man ikke har den offentlige nøgle til. Men man kan selvfølgelig stadig opsnappe hashværdien, når denne sendes over netværket, ligesom man vil kunne opsnappe selve filen, når denne sendes. Aflytning er dog besværligt i et peer-to-peer netværk, da man typisk må aflytte hver node individuelt, og ikke blot kan nøjes med at aflytte en servers forbindelse.

En løsning på problemet er at tilføje en tilfældig initialiseringsvektor i starten af en fil og så kryptere i CBC (eng.: *Cipher Block Chaining*) modus, hver gang filen skal sendes. På den måde vil filen se forskellig ud hver gang. Dette skal så kombineres med en form for padding således, at man ikke kan udlede noget udfra filens størrelse.

Ulempen ved denne løsning er selvfølgelig, at krypteringen skal ske, hver gang en fil overføres. Dette vil næppe være et problem for en moderne PC, men kan være det for PDA'er eller mobiltelefoner, eller hvis mange ønsker at hente filer fra den samme node samtidigt. Heldigvis kan en klient med begrænsede processorressourcer blot undlade at ændre initialiseringsvektoren og således undgå at skulle kryptere filen igen.

En anden ulempe ved at indføre en tilfældig initialiseringsvektor i starten af filen er, at det bliver umuligt at hente forskellige dele af filen fra forskellige noder, der har brugt forskellige initialiseringsvektorer. Den mest oplagte løsning på dette problem er at opdele filen i et antal segmenter, der hver begynder med en initialiseringsvektor. Disse segmenter kan så hentes individuelt fra forskellige noder.

De beskrevne forholdsregler mod trafikanalyse, er ikke medtaget i prototypeimplementeringen, dels på grund af de forskellige ulemper nævnt ovenfor, og dels fordi det i høj grad ville komplicere systemet.

3.4 Opsummering

Den opstillede kravspecifikation omfatter to hovedanvendelsesområder for det påtænkte system: Udgivelse af digitalt indhold samt kollaborativt arbejde. De vigtigste egenskaber i forhold til andre fildelingssystemer er sikringen af konfidentialitet og integritet for de delte data.

Sikkerhedsmodellen definerer to centrale begreber: Nøglesæt og nøgleringe. Alle brugere og filer i systemet udstyres med et nøglesæt bestående af tre nøgler: Et offentlig/privat nøglepar samt en symmetrisk nøgle. Den offentlige og den symmetriske nøgle giver læseadgang. Har man desuden den private nøgle, har man også skriveadgang. Nøglesæt kan samles på nøgleringe, der er beskyttet af nøglesæt på samme måde som andre filer i systemet. Nøgleringe kan deles mellem forskellige brugere. En arbejdsgruppe kan således oprette en fælles nøglering, der rummer nøglerne til de filer, gruppen ønsker at dele med hinanden.

Gennemgangen af mulige trusler afslører, at hvis der vælges tilstrækkeligt lange krypteringsnøgler, samt ikke lineære krypteringsalgoritmer eller hashfunktioner, vil systemet effektivt kunne opretholde konfidentialitet og integritet.

Til gengæld er der mulighed for at gennemføre DoS-angreb og angreb baseret på trafikanalyse. Det er muligt delvist at imødegå disse, hvis der træffes passende forholdsregler. Disse forholdsregler vil dog komplicere systemet temmelig meget, hvorfor de er fravalgt i prototypeimplementeringen.

Modstandere kan altså holde øje med, hvem der opbevarer bestemte filer, samt hvornår de opdateres. En sådan overvågningsaktivitet er dog både ganske krævende og samtidig svær at skjule. Hvorvidt disse angreb udgør et reelt problem, vil naturligvis afhænge af, hvilke sikkerhedskrav man stiller.

Det er vores vurdering, at det skitserede system primært vil finde anvendelse i situationer, hvor sikkerhedskravene er mindre til moderate. De beskrevne angreb vil derfor næppe udgøre nogen reel trussel.

Kapitel 4

Design

På baggrund af kravene opstillet i forrige kapitel, samt den viden vi har opnået i kraft af vores gennemgang af "state-of-the-art" indenfor de forskellige berørte områder, vil vi i dette kapitel beskrive et overordnet design for det påtænkte system.

4.1 Netværksarkitektur

Da det ønskede system, på linje med Gnutella, er tænkt som en fildelingsapplikation, virker det ikke urealistisk at antage en udskiftningsrate blandt noderne, der svarer til, hvad man finder i denne type systemer. Derfor er et af kravende til systemet, at det skal kunne fungere, når der er en høj grad af udskiftning blandt de tilsluttede noder. Dette krav er funderet i [RFI02], hvor det påvises, at i Gnutella netværket forlader 40% af noderne netværket i løbet af 4 timer.

Hvis data skulle distribueres efter en distribueret hashfunktion, som i PAST, kunne en node risikere at skulle replikere store datamængder, hver gang den tilsluttede sig netværket. Dette vil naturligvis være et problem, hvis noden ikke er placeret på en hurtig netværksforbindelse. Det ville også være ineffektivt, da de replikerede data sandsynligvis ikke ville være tilgængelige for andre noder særligt længe.

Dette er årsagen til, at vi har valgt at placere data på de noder, der producerer og deler dem. Efterhånden som filerne bliver hentet af andre noder, bliver de spredt i netværket. Denne opførsel svarer helt til hvad der kendes fra Napster, Gnutella og Kazaa.

Hvis man ønsker at kunne foretage effektive søgninger i netværket, skal metadata placeres andre steder end på de noder der opbevarer data. En centraliseret løsning er nem at implementere, men giver også et "single-point-of-failure" samt potentielle skaleringsproblemer. Hybridløsningerne der benyttes i Kazaa og JXTA virker derfor attraktive, da disse to problemer undgås.

Kazaa er en lukket protokol, der ikke umiddelbart er mulighed for at benytte eller bygge videre på. JXTA er derimod et åbent framework som det står frit for alle at benytte. Valget står altså mellem at benytte JXTA eller udvikle vores egen peer-to-peer hybridarkitektur. Da den tid der er til rådighed for udvikling er begrænset og vi ikke er tilhængere af at genopfinde den dybe tallerken, har vi valgt at benytte JXTA som grundlæggende peer-to-peer platform.

Når dette valg er taget er det oplagt, at delte filer skal publiceres ved hjælp af JXTAs annonceringskoncept, da dette muliggør globale søgninger. De enkelte noder skal derudover, kunne acceptere forespørgsler efter de filer som de deler.

Delte filer identificeres ved en hash af den offentlige filnøgle. Konkrete versioner identificeres desuden ved en hash af de krypterede data. Det er derfor nærliggende at lade JXTAs SRDI service indeksere annonceringerne efter disse værdier. Det vil gøre det muligt for noder der har den offentlige nøgle til en fil, at lokalisere de noder der udbyder revisionsloggen. Har man allerede revisionsloggen, kan de konkrete revisioner af filen findes udfra hashværdierne inddateret i denne.

Tilladelser identificeres ved en hash af den offentlige nøgle, som de er krypteret med, samt en hash af den offentlige filnøgle der gives adgang til. Hvis en bruger søger efter tilladelser til sig selv, ved han som regel ikke, hvilke filnøgler han har fået adgang til. Derfor søges der kun efter præfikset, dvs. en hash af brugerens offentlige nøgle.

Hvis tilladelsen indekseres efter hele navnet i den distribuerede hashtabel, vil søgningen fejle, da hele navnet vil hashe til en ganske anden værdi, end hashen af den offentlige nøgle alene. I de afprøvninger vi har foretaget af JXTA, har der ikke været problemer med at lokalisere tilladelser, selvom de var indekseret efter hele navnet og søgningen kun gjaldt prefikset. Dette skyldes formentligt det begrænsede antal noder i testopstillingen.

Problemet kan let løses ved at tilføje et ekstra felt i annonceringen, der kun indeholder hashen af den offentlige nøgle, og også indeksere annonceringen efter dette. Dette er dog undladt her af implementeringsmæssige grunde.

4.2 Kryptering

Når en ny fil skal deles, dannes der et nyt nøglesæt, bestående af et offentlig/privat-nøglepar samt en nøgle der skal bruges til den symmetriske kryptering. Filen krypteres med den symmetriske nøgle og der dannes en ny revisionslog, der indeholder en hash af de krypterede data samt signerede udgaver af disse.

Der er derfor brug for definitioner af filformater for de krypterede filer, revisionslog og nøgler. Desuden skal der være funktioner til at kryptere, dekryptere, samt læse og skrive disse filformater.

4.2.1 Asymmetrisk kryptering

Asymmetrisk kryptering benyttes i forbindelse med revisionsloggen, der dels krypteres med den private filnøgle og dels indeholder forskellige signaturer. Denne type kryptering benyttes også når der skal dannes en tilladelse, hvor en filnøgle skal krypteres med en brugers offentlige nøgle.

Vi har valgt at benytte RSA som asymmetrisk krypteringssystem. Vi havde overvejet at benytte elliptisk kurve-kryptering, for at undgå RSAs store nøgler, samt den noget langvarige nøglegenereringsproces.

Grunden til at elliptisk kurve-kryptering blev fravalgt er, at det på nuværende tidspunkt er småt med produktionsklare implementeringer. Således havde vi ikke held til finde en Java implementering med tilstrækkelig funktionalitet og dokumentation. Dette vil formentlig ændre sig med tiden, da EC stadig er et forholdsvist nyt område.

Da det ville være ineffektivt at benytte RSA til at kryptere hele revisionsloggen, benyttes en hybridløsning, hvor RSA kun bliver brugt til at kryptere en tilfældigt dannet symmetriske nøgle. Efterfølgende krypteres revisionsloggen så symmetrisk med denne nøgle. På den måde opnår man hurtigheden som det symmetriske chiffer giver, samtidig med at asymmetrien bevares. Ved implementering er det vigtigt at sikre, at der er sammenhæng mellem nøglelængderne for det asymmetriske og symmetriske chiffer, samt at nøglen til det symmetriske chiffer genereres tilfældigt.

4.2.2 Symmetrisk kryptering

Symmetrisk kryptering skal benyttes til at kryptere de delte filer samt det personlige nøglesæt. I det sidste tilfælde skal det være muligt at dekryptere nøglesættet ved hjælp af et kodeord, der indtastes når systemet startes. Krypteringsnøglen genereres her ved at beregne hashværdien af kodeordet sammenkædet med et tilfældigt tal. Det tilfældig valgte tal gemmes i klartekst sammen med den krypterede fil.

Grunden til at man sammenkæder kodeordet med et tilfældigt tal er for at undgå et ordbogsangreb (eng.: *dictionary attack*), hvor en angriber har beregnet hashværdier for alle ord i en ordbog og prøver dem som nøgler. Det har vist sig, at mennesker ofte vælger et almindeligt ord som deres kodeord, derfor er ordbogsangreb yderst effektive, hvis systemet ikke har været designet med dette angeb for øje. Ved at sammenkæde kodeordet med et tilfældigt tal, kan angriberen ikke bare benytte de på forhånd genererede nøgler, men er nødt til at beregne dem udfra det tilfældigt valgte tal.

Blokalgoritmerne fra hastighedssammenligningen i afsnit 2.6.2, betragtes alle - med undtagelse af almindelig DES - som værende meget sikre. Vi har derfor tilladt os at benytte algoritmernes hastighed, samt adgangen til modne og stabile implementeringer, som primære udvægelseskriterier.

Da AES med 128 bit nøgler er mellem de hurtigste blandt de betragtede algoritmer og desuden er udpeget af NIST[1] som standard, har vi valgt at benytte denne.

Det er dog værd at bemærke, at FlexiProviders Twofish implementation er den hurtigste blandt de testede javaimplementationer. Så hvis man ønskede den absolut hurtigste algoritme, med god sikkerhed under Java, skulle man vælge denne. Et argument imod dette valg er dog, at Twofish indtil nu ikke har modstået helt den samme mængde analyse som AES.

4.2.3 Nøglelængder

Som det fremgår af afsnit 2.7.1 er det vigtigt at nøglelængderne i systemer, hvor der benyttes både asymmetrisk og symmetrisk kryptering, er valgt således at de forskellige typer kryptering er lige svære at bryde. Den samlede sikkerhed bliver ikke bedre end sikkerheden i det svageste af systemerne. Hvis man f.eks. valgte at kombinere en 128 bit symmetrisk AES nøgle med en 512 bit asymmetrisk RSA nøgle, ville det være meget lettere at forsøge at faktorisere RSA nøglen, end at udføre et brute-force angreb mod AES nøglen.

I praktiske implementeringer af krypteringsalgoritmerne er kun et mindre antal forskellige nøglelængder understøttet. F.eks. understøtter Suns Java implementering af AES kun 128 bit nøgler. Tilsvarende er den højeste tilladte RSA nøglestørrelse på 2048 bit.

Da disse nøglestørrelser er de højest mulige i de betragtede implementeringer og desuden tilfredstiller moderate til høje sikkerhedskrav, har vi valgt at benytte disse.

Ulempen ved de store RSA nøgler er, at de tager lang tid at generere. Hvis de data man ønsker at beskytte, ikke behøver at være beskyttet specielt længe, eller man kan leve med ringere sikkerhed til gengæld for et hurtigere system, kunne man sagtens argumentere for at benytte 1024 eller 512 bit RSA nøgler.

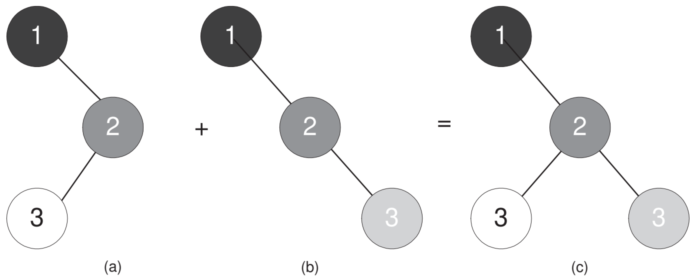
4.3 Revisionslog

Hver fil i systemet følges af en revisionslog. Denne indeholder signerede hashværdier af de forskellige revisioner. Dette gør det muligt at kontrollere at opdateringer er foretaget af brugere, som er i besiddelse af den private filnøgle. Revisionsloggen er desuden krypteret med den private filnøgle, således at man ikke kan følge opdateringshistorikken uden at være i besiddelse af den offentlige filnøgle. En anden fordel ved at revisionsloggen er krypteret, er at brugere uden den private filnøgle, ikke kan kan kryptere revisionsloggen korrekt. Det betyder at en falsk revisionslog kan detekteres straks, og altså ikke først når man har hentet en konkret revision af filen og konstateret, at signaturen ikke er korrekt.

[1] NIST - National Institute of Standards and Technology

4.3.1 Sammenfletning

Når en fil er delt mellem flere brugere er der mulighed for, at de uafhængigt af hinanden opdaterer filen. Hvis for eksempel to brugere opdaterer datafilen uafhængigt af hinanden, betyder det, at revisionsloggen hos hver af de to brugere er forskellig. Mere præcist så vil den seneste loginddatering (eng.: *logentry*) hos de to brugere være forskellig. En log vil altid indeholde mindst en revision (den initielle), og da vi ikke tillader at loginddateringer slettes, vil det altid være muligt at flette de forskellige logge sammen til et træ.



Figur 4.1: Sammenfletning af revisionslog.

På figur 4.1 ses et eksempel på en sammenfletning af to revisionslogge som er forskellige. På figuren repræsenterer cirklerne loginddateringer i revisionsloggen, og stregerne mellem cirklerne indikerer hvilken loginddatering, som en anden bygger på.

Den eneste loginddatering der ikke bygger på nogen anden er roden (den sorte cirkel). Da vi ikke tillader at loginddateringer slettes, må de som minimum have roden tilfælles og et nyt træ kan derfor bygges.
Det nye træ bygges ved at tage den/de undertræer, som er i den ene men ikke i den anden revisionslog, og forbinde dem på den anden revisionslog, som så udgør den sammenflettede revisionslog (se fig. 4.1).
Denne sammenfletning kan kun foregå hvis man har skriverettigheder, dvs. er i besiddelse af den private filnøgle.

Har man ikke skriverettigheder men f.eks. kun integritets- og læserettigheder, kan man ikke flette revisionslogge sammen. Når en sådan bruger søger efter, finder og henter en opdateret revisionslog, vil systemet checke om den nuværende revisionslog er indeholdt i den netop hentede revisionslog. Er den det, så vil systemet overskrive den nuværende log med den nye, ellers vil systemet beholde den nuværende.

Hvis systemet ukritisk overskrev loggen, så ville man kunne komme i en situation, hvor en bruger risikerer at miste loginddateringer han allerede havde. Hvis en bruger, der ikke har skriverettigheder, har revisionslog (a) på figur 4.1 og så finder revisionslog (b) på samme figur, og systemet så overskrev (a) med (b), ville brugeren miste den hvide loginddatering. Brugeren må derfor vente til en bruger med skriverettigheder sammenfletter revisionsloggene (a) og (b) til (c).

4.4 Brugergrænseflade

Hver bruger i systemet skal have sit eget personlige nøglesæt. Den offentlige nøgle kan gives til andre brugere, man ønsker at kunne kommunikere sikkert med. Den private nøgle benyttes til signaturer. Endelig benyttes den symmetriske nøgle til at kryptere den personlige nøglering.

På den personlige nøglering opbevares nøgler til de filer og nøgleringe som man har adgang til. Andre brugeres offentlige nøgler kan også opbevares på den personlige nøglering. Dette giver anledning til en træstruktur, hvor roden udgøres af det personlige nøglesæt.

Brugergrænsefladen opdeles i fire dele:

Filer og nøgler: Brugergrænsefladen skal give mulighed for at oprette nøgler og nøgleringe, samt at kopiere nøgler mellem nøgleringe. Det skal desuden være muligt gennem en revisionsoversigt at dekryptere og opdatere filer.

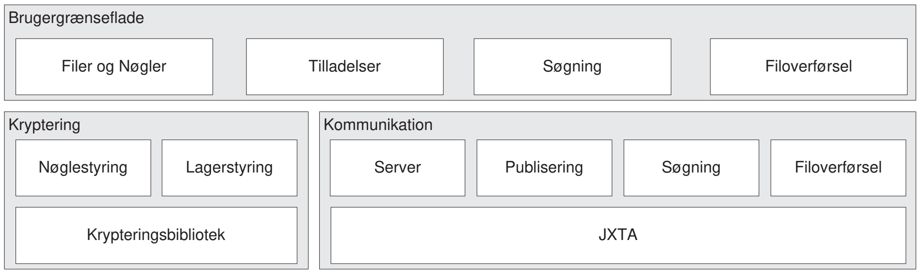
Tilladelser: Har man en filnøgle og en brugers offentlige nøgle, skal man kunne oprette en tilladelse. Dette sker ved at kryptere filnøglen med brugerens offentlige nøgle. Det skal desuden være muligt at eksportere offentlige nøgler i et ukrypteret format.

Søgning: Når man har en filnøgle, kan man søge efter den tilhørende revisionslog og eller konkrete revisioner af filen. Man kan også søge efter tilladelser andre brugere har givet en.

Filoverførsler: Når en fil er lokaliseret skal det være muligt at hente den. Der kan hentes flere filer samtidigt, og andre brugere kan hente filer fra en selv. Det er meningen at denne del af brugergrænsefladen skal give en oversigt over igangværende overførsler, samt mulighed for at afbryde disse.

4.5 Opsummering

I dette kapitel er en række grundlæggende komponenter i det påtænkte system blevet identificeret. Der er desuden truffet en række vigtige beslutninger for hver af disse komponenter.



Figur 4.2: Grundlæggende komponenter i systemet.

Kommunikation: Omfatter deling, søgning samt overførsel af filer. Vi har valgt at basere vores system på JXTA platformen, da vi på den måde får rådighed over et kraftfuldt peer-to-peer framework, der kan håndtere de nederste lag af kommunikationen for os.

Kryptering: Kryptering og dekryptering af filer og nøgler, samt generering af signaturer og verifikation af disse. Som symmetrisk krypteringsalgoritme benyttes AES med 128 bit nøgler. Som asymmetrisk krypteringssystem er valgt RSA med en nøglelængde på 2048 bit. Revisionsloggen krypteres i et hybridformat, således at den kan læses med den offentlige filnøgle, uden at den beregningskrævende asymmetriske kryptering skal benyttes til kryptering af hele loggen.

Brugergrænseflade: For at det påtænkte system kan blive praktisk anvendeligt, er det vigtigt at det kan benyttes på en brugervenlig måde. Brugergrænsefladen tænkes opdelt i 4 forskellige dele: Håndtering af filer og nøgler, tilladelser, søgning samt filoverførsler.

Kapitel 5

Implementering

Dette kapitel omhandler implementeringen af en prototype af det foreslåede system.

Vi har valgt at udvikle programmet i programmeringssproget Java samt at benytte Suns peer-to-peer framework JXTA som grundlæggende peer-to-peer arkitektur. Valget af disse to byggesten har gjort det muligt at udvikle en komplet prototype på den begrænsede tid, der har været tilrådighed.

Figur 5.1 giver et overblik over de vigtigste klasser i programmet. Af hensyn til overblikket er en række hjælpeklasser udeladt. Som det fremgår har vi forsøgt at benytte Model-View-Controller (MFC) programmeringsmønsteret til at strukturere programet. Formålet med denne opdeling er at separere brugergrænseflade og præsentation af data (view), fra logik (control) og data (model).

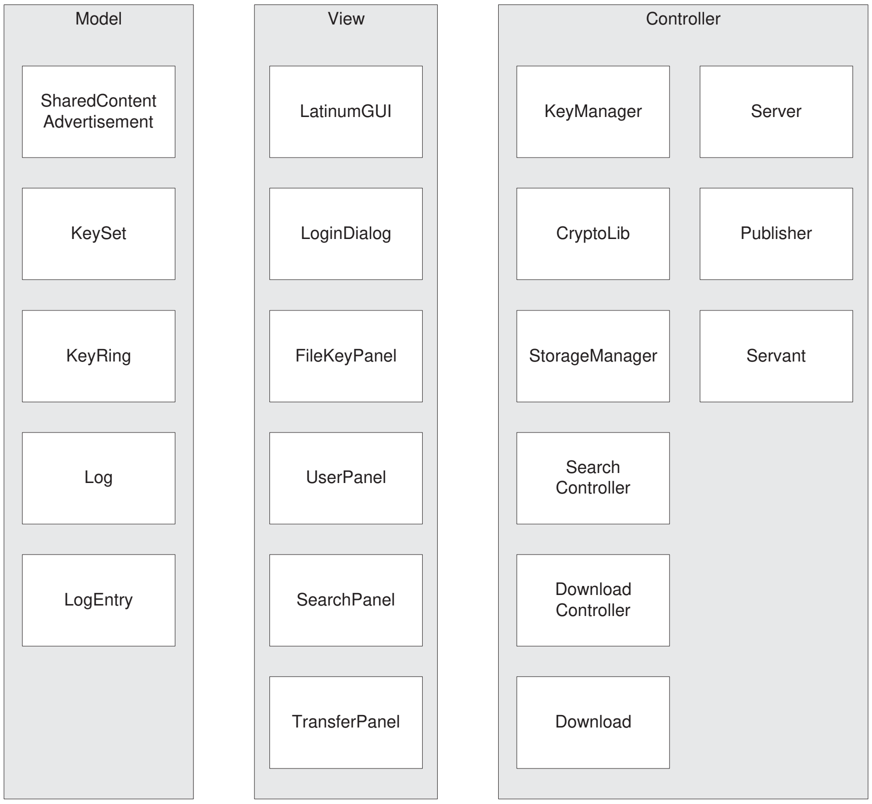
Vi har valgt at kalde programmet for Latinum[1].

I implementeringen er brugt Java version 1.4.2 og JXTA version 2.1.1.

5.1 Kommunikation

Serverdelen af programmet består af et publiceringsmodul og en egentlig server, der behandler indkomne forespørgsler efter filer.

[1] Navnet er afledt af det engelske ord for platin, platinum, og skal forstås som et symbol for noget værdifuldt og uforgængeligt, naturligvis med reference til de data, brugere forhåbentligt vil betro programmet.



Figur 5.1: Klasser i Latinum.

5.1.1 Annoncering af indhold

I JXTA offentliggøres delte ressourcer via annonceringer. En annoncering sendes og opbevares i XML format. Internt repræsenteres en annoncering af JXTA-klassen `Advertisement`.

Publiceringsmodulet overvåger hvilke filer der tilføjes og hentes i det lokale system og sørger for at publicere disse. Konkret sker dette ved med jævne mellemrum at læse et bestemt katalog igennem. For hver fil i kataloget undersøges det, om der allerede eksisterer en annoncering. Hvis det ikke er tilfældet dannes der en ny annoncering og denne publiceres. Publiceringen sker både til en lokal database og til andre noder. Det sidste sker dels ved at annonceringen publiceres til den aktuelle rendezvous node, dels ved at annonceringen broadcastes på det lokale netværk.

Hver node i systemet opretter en input pipe med et unikt ID, denne pipe annonceres overfor de andre noder via et `PipeAdvertisement`. ID'et gemmes og ændrer sig ikke på tværs af sessioner, dette er vigtigt, fordi filannonceringerne også indeholder pipe ID'et. Hvis noden skiftede pipe ID, hver gang man startede den, ville alle de annonceringer, noden hidtil havde publiseret blive ubrugelige.

Vi har udvidet `Advertisement` klassen ved at lave en ny klasse, `SharedContentAdvertisement`, der nedarver fra den.

Et `SharedContentAdvertisement` indeholder følgende felter:

- ID - Unikt ID der identificerer annonceringen.
- HKU - Den krypterede fils navn, dvs. en base64 kodet hash af filens offentlige nøgle, eventuelt kombineret med en base64 kodet hash af selve den krypterede fil, hvis der er tale om en konkret revision.
- Pipe - ID for den input pipe der lyttes på.
- Peername - Nodens navn.
- Linespeed - Hastigheden af nodens netforbindelse.
- Size - Filens størrelse.

Af disse felter benyttes kun HKU som indeksfelt for den distribuerede hashfunktion. Felterne er alle i klartekst og en ondsindet node kan frit manipulere indholdet. Man kan først verificere annonceringen når den tilhørende fil eller revisionslog er hentet.

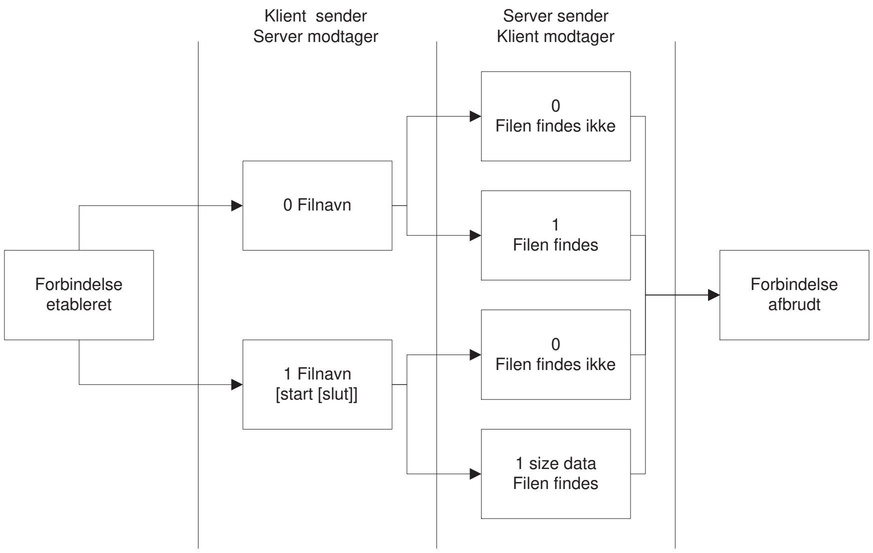
5.1.2 Søgning

Annonceringer findes ved at benytte JXTAs `DiscoveryService` klasse, hvor `getLocalAdvertisements` og `getRemoteAdvertisements` metoderne foretager søgninger i henholdsvis den lokale cache og på netværket. Lokale søgninger giver svar med det samme, mens svar fra netværket behandles asynkront og vises efterhånden som de ankommer. Svar modtaget fra netværket placeres desuden i den lokale database, således at de fremkommer straks, hvis søgningen gentages. Ulempen ved at bruge et svar der kommer fra den lokale database er, at man ikke umiddelbart kan vide, om noden der har foretaget annonceringen, stadig er tilsluttet netværket. Svar fra netværket kommer direkte fra noden, der har foretaget annonceringen. Man kan derfor være rimelig sikker på, at noden er tilsluttet netværket, og at der kan kommunikeres med den.

5.1.3 Server

Når en node ønsker at hente en fil, gøres dette ved at oprette en forbindelse til den annoncerede pipe. En pipe i JXTA er som udgangspunkt ikke pålidelig, dette er naturligvis uhensigtsmæssigt, når der skal overføres filer. Derfor benyttes `JxtaSocket`, der som navnet antyder emulerer en almindelig, pålidelig, socketforbindelse ovenpå JXTAs pipes.

På denne socketforbindelse benyttes en enkel request/response-orienteret protokol. Kommandoer og svar sendes som simple ASCII-kodede tekststrenge. Kommandoer afsluttes med et linjeskift.



Figur 5.2: Tilstandsdiagram for filtransportprotokollen.

Når en forbindelse er etableret, kan klientnoden sende to forskellige kommandoer til servernoden:

Kommando	**Argumenter**	**Beskrivelse**
0	Filnavn	Undersøger om en fil findes.
1	Filnavn [start [slut]]	Henter en fil.

Tabel 5.1: Kommandoer

Argumenterne `start` og `slut` giver mulighed for at hente en del af en fil. Den første byte der sendes, er den på position `start`, den sidste er den på position `slut-1`. Udelades argumenterne, eller hvis de er inkonsistente, f.eks. `start>slut`, sendes hele filen.

Kommando	**Svar**	**Beskrivelse**
0	0|1	1 hvis files findes, 0 hvis filen ikke findes.
1	(1 size data)|0	Returnerer filen hvis denne findes, ellers returneres 0.

Tabel 5.2: Svar

Hvis filen findes på servernoden, returneres “1” efterfulgt af det antal bytes der skal transmitteres. Dernæst følger selve data. Dette er nødvendigt for at klientnoden kan vide, hvornår overførslen er slut. En alternativ løsning på dette problem er, at benytte et end-of-file tegn til at markere afslutningen. Denne løsning blev fravalgt, fordi den ville kræve, at filen blev scannet igennem, når den blev sendt, således at tegnet kunne erstattes, hvis det forekom i filen.

Filoverførsler foregår i separate tråde. Når en node modtager en forespørgsel, startes der en ny tråd der kan betjene denne. Dette sker ved at oprette en ny instans af `Servant` klassen.

Ligeledes startes der en ny tråd på klientnoden når en bruger initierer en filoverførsel. Det betyder, at de enkelte noder uden problemer kan hente og sende flere filer simultant.

Klientdelen af protokollen er implementeret i `Download` klassen. Muligheden for at downloade en del af en fil, er ikke implementeret på klientsiden. Kommandoen, der kun undersøger om en fil findes, benyttes heller ikke, men er tiltænkt en fremtidig udvidelse, hvor klienten undersøger tilgængeligheden af en fil, før overførslen startes.

Når en klientnode forsøger at oprette en forbindelse til en servernode, forsøges dette i 2 minutter. Hvis det ikke lykkes at oprette en forbindelse indenfor dette tidsrum, terminerer `Download`-tråden med en fejlmeddelelse. Når først en forbindelse er etableret, gøres der ikke yderligere forsøg på at detektere timeouts. Hvis overførslen afbrydes fra en af siderne, detekteres dette. Men hvis en overførsel pludselig stopper, f.eks. på grund af netværksfejl, detekteres dette ikke automatisk.

5.2 Kryptering

Der er behov for en del kryptografiske algoritmer til at udføre symmetrisk kryptering, asymmetrisk kryptering, signering samt beregning af hashværdier. Da det ville være en stor opgave selv at programmere alle disse algoritmer, har vi undersøgt mulighederne for at bruge eksisterende krypteringsbiblioteker.

Sun har udviklet en udvidelse til Java, kaldet JCE - Java Cryptography Extension. JCE definerer standardmetoder for anvendelse af kryptografiske algoritmer. Det betyder at man kan bruge Suns indbyggede kryptografiske algoritmer og algoritmer fra tredjepart, en såkaldt "provider", på en ensartet måde.

Vi har valgt at bruge JCE, da det leverer den funktionalitet vi har brug for, og fordi det vil være nemt at udvide eller ændre de kryptografiske algoritmer i fremtiden.

Vi har bestræbt os på udelukkende at bruge Suns JCE for at lette installation og kørsel af programmet. Dette har dog ikke været muligt fuldt ud, da Sun ikke har en `Cipher` implementation af RSA, selv om man kunne få en anden opfattelse ved gennemlæsning af dokumentationen [Sun]. En `Cipher` implementation er nødvendig, hvis man ønsker at benytte RSA til egentlig kryptering, hvilket vi har behov for i forbindelse med revisionsloggen.

For at løse dette problem bruger vi en anden provider, Bouncy Castle [Cas], til RSA. Vi har implementeret forskellige statiske metoder i klassen `CryptoLib` til at kryptere og dekryptere de forskellige filformater.

5.2.1 Symmetrisk kryptering

Til at udføre symmetrisk kryptering har vi implementeret metoderne `encryptAESWithSHA` og `decryptAESWithSHA` i klassen `CryptoLib`. Som navnet antyder, krypterer `encryptAESWithSHA` data med AES, og beregner samtidig en SHA-1 hashværdi af data, som bruges til integritetscheck ved dekryptering med metoden `decryptAESWithSHA`.

Der findes to udgaver af `encryptAESWithSHA`. En der krypterer fra en fil til en anden, og en der krypterer fra hukommelsen til en fil. Ligeledes for `decryptAESWithSHA` findes der en version der dekrypterer fra en fil til en anden, og en der dekrypterer fra en fil til hukommelsen.

Kryptering og dekryptering fra en fil til en anden benyttes i forbindelse med datafiler. Kryptering og dekryptering til hukommelsen fra en fil og omvendt, benyttes i forbindelse med nøgler og nøgleringe.

Felt	**Beskrivelse**	**Type**
0	initialiseringsvektor (IV)	byte[] (32)
1	datalængde	long
2	data	byte[]
3	SHA-1 hash	byte[] (160)

Tabel 5.3: Filformat for filer krypteret med `encryptAESWithSHA`.

I tabel 5.3 ses filformatet for filer krypteret med `encryptAESWithSHA`. Først gemmes der en initialiseringsvektor på 32 bytes, så kommer længden af data efterfulgt af dataene og der afsluttes med SHA-1 hash af dataene. Felt 0 krypteres med 128 bit AES i ECB-mode og felterne 1-3 krypteres med 128 bit AES i CBC-mode med PKCS#5 padding.

Tabellen 5.3 er en overskuelig og hurtig måde at fremstille et filformat på, men den kan være tvetydig (eng.: *ambiguous*), da man ikke kan se om nogle af rækkerne må/skal repeteres. Derfor beskrives filformatet også ved den udvidede Backus-Naur Form [Fol], hvor Java primitiverne byte, short, long osv. bruges som termineringssymboler.

< IV >	::=	*byte**
< datalængde >	::=	*long*
< data >	::=	*byte**
< SHA1Hash >	::=	*byte**
< AESKryptoFormat >	::=	< IV >< datalængde >< data >< SHA1Hash >

Figur 5.3: Filformat for filer krypteret med `encryptAESWithSHA` på BNF.

5.2.2 Hybridkryptering

Hybridkryptering benyttes til kryptering af revisionsloggen. Til at udføre hybridkryptering har vi implementeret metoderne `encryptRSAHybWithSHA` og `decryptRSAHybWithSHA` i klassen `CryptoLib`. Som med `encryptAESWithSHA` og `decryptAESWithSHA` findes der to tilsvarende versioner af henholdsvis `encryptRSAHybWithSHA` og `decryptRSAHybWithSHA`.

Felt	Beskrivelse	Type
0	symmetrisk nøglelængde	short
1	symmetrisk nøgle	byte[]
2	initialiseringsvektor (IV)	byte[] (32)
3	datalængde	long
4	data	byte[]
5	SHA-1 hash	byte[] (160)

Tabel 5.4: Filformat for filer krypteret med `encryptRSAHybWithSHA`.

Tabellen 5.4 viser filformatet for filer krypteret med `encryptRSAHybWithSHA`. Først kommer længden af den symmetriske nøgle, denne er efterfulgt af selve den symmetriske nøgle. Så kommer en initialiseringsvektor, samt længden af de data der skal krypteres, efterfulgt af dataene og en SHA-1 hash af dataene.

Felt 0 er ukrypteret, felt 1-2 er krypteret med 2048 bit RSA og felt 3-5 er krypteret med 128 bit AES i CBC-mode med PKCS#5 padding. Felt 0 er ukrypteret for at lette implementeringen. Dette er uden betydning for sikkerheden, da denne ikke er baseret på, at angriberen ikke kender nøglelængden (jf. Kerckhoffs' antagelse, der siger at det antages, at angriberen ved, hvilken krypteringsmetode, der er i brug [Knu03]).

For at undgå tvetydighed, er filformatet fra tabel 5.4 gengivet på BNF.

< symLængde > ::= *short*
< symNøgle > ::= *byte**
< IV > ::= *byte**
< datalængde > ::= *long*
< data > ::= *byte**
< SHA1Hash > ::= *byte**
< HybKryptoFormat > ::= < symLængde >< symNøgle >< IV >
< datalængde >< data >< SHA1Hash >

Figur 5.4: Filformat for filer krypteret med `encryptRSAHybWithSHA` på BNF.

5.2.3 Kryptering med kodeord

Det personlige nøglesæt skal krypteres således, at det kan dekrypteres med et kodeord når programmet startes.

Til at kryptere og dekryptere personlige nøglesæt har vi lavet metoderne `encryptAES` og `decryptAES` i klassen `CryptoLib` som henholdsvis kan kryptere fra hukommelsen til en fil, og dekryptere en fil til hukommelsen, givet et kodeord.

Felt	**Beskrivelse**	**Type**
0	tilfældigt tal (salt)	long
1	datalængde	int
2	data	byte[]
3	hashlængde	int
4	SHA-1 hash	byte[] (160)

Tabel 5.5: Filformat for filer krypteret med `encryptAES`.

Filformatet for `encryptAES` kan ses i tabel 5.5. Først kommer det tilfældige tal, dernæst længden af det der skal krypteres, endelig følger selve dataene, længden af hashværdien der er beregnet for data og så hashværdien. Felt 0 er ukrypteret og felt 1-4 krypteret med 128 bit AES i ECB-mode.

ECB-mode betragtes normalt som mindre modstandsdygtigt overfor kodebogsangreb (eng.: *codebook attack*) end CBC-mode. Det mener vi dog ikke, udgør noget problem i dette tilfælde, da en det personlige nøglesæt har en beskeden størrelse og krypteringsnøglen kun bruges til dette ene nøglesæt.

For at undgå tvetydighed, er filformatet fra tabel 5.5 gengivet på BNF.

$<$ salt $>$	::=	*long*
$<$ datalængde $>$	::=	*int*
$<$ data $>$	::=	$byte^*$
$<$ Hashlængde $>$	::=	*int*
$<$ SHA1Hash $>$	::=	$byte^*$
$<$ KodeKryptoFormat $>$	::=	$<$ salt $><$ datalængde $><$ data $>$ $<$ Hashlængde $><$ SHA1 $>$

Figur 5.5: Filformat for filer krypteret med `encryptAES` på BNF.

5.3 Dataformater

Latinum gør brug af tre forskellige dataformater som de kryptografiske metoder, der er beskrevet i 5.2, krypterer til disken. Der vil i det følgende blive beskrevet, hvordan disse dataformater er opbygget og hvilken kryptografisk metode, der bruges til at gemme de enkelte formater på disken. Et dataformat kan altså betragtes som det filformat man ville få, hvis man gemte direkte til disken uden brug af de kryptografiske metoder.

Nøglefiler: Er filer der indeholder et nøglesæt. Af denne type filer er der tre forskellige slags: Nøglefiler med et ukrypteret nøglesæt, nøglefiler der indeholder ens personlige nøglesæt i krypteret form, og nøglefiler der indeholder et nøglesæt krypteret med en brugers offentlige nøgle, en såkaldt tilladelse.

Datafiler: Er de filer man ønsker at dele eller distribuere igennem netværket. Datafiler kan inddeles i to kategorier, datafiler der indeholder nøgleringe, og datafiler der indeholder brugerdata f.eks. programmer eller dokumenter.

Logfiler: Alle datafiler har en log associeret til sig. Loggen har til opgave at dokumentere en datafils udvikling. Loggen fungerer også som en “vejviser” i den forstand, at man med loggen kan finde og downloade revisioner af datafilen.

5.3.1 Nøgler

Til alle filer, der deles i Latinum, hører der et nøglesæt bestående af tre nøgler; en offentlig, en privat og en symmetrisk nøgle, jf. 3.2.1. Til at repræsentere et nøglesæt har vi implementeret klassen `KeySet`, som indeholder private variabler til at rumme henholdsvis den offentlige, private og symmetriske nøgle.

Nøglerne kan angives specifikt i konstruktoren, eller også kan de blive genereret tilfældigt ved at kalde null-konstruktoren. Da disse nøgler skal kunne gemmes, har vi lavet metoden `getBytes`, der returnerer nøglerne i et bytearray.

På den måde er det nemt at gemme et nøglesæt i krypteret form ved hjælp af de kryptografiske metoder i `CryptoLib`. Ligeledes har vi lavet en konstruktor i `KeySet`, der kan genskabe et nøglesæt ved hjælp af et sådant bytearray.

Felt	**Beskrivelse**	**Type**
0	priNøgleLængde	short
1	priNøgle	byte[]
2	pubNøgleLængde	short
3	pubNøgle	byte[]
4	symNøgleLængde	short
5	symNøgle	byte[]
6	nøgleSætType	boolean
7	betroet	boolean
8	beskrivelseLængde	short
9	beskrivelse	byte[]

Tabel 5.6: Dataformat for et nøglesæt, dannet af `getBytes` metoden.

I tabel 5.6 ses dataformatet for det bytearray som `getBytes` returnerer. Først kommer længden af den private nøgle efterfulgt af den private nøgle kodet i PKCS#8-formatet. Derefter kommer længden af den offentlige nøgle efterfulgt af den offentlige nøgle kodet i X509-formatet.

Så kommer længden af den symmetriske nøgle efterfulgt af den symmetriske nøgle som er ukodet. Derefter kommer en sandhedsværdi, der fortæller, om nøglesættet er nøglesæt for en nøglering eller ej.

Herefter kommer endnu en sandhedsværdi, der fortæller om nøglesættet er betroet (eng.: *trusted*), dvs. om man er sikker på hvem nøglesættet kommer fra. Dette er vigtigt, hvis nøglesættet indeholder en anden brugers offentlige nøgle. Til sidst kommer længden af en beskrivelse af nøglesættet efterfulgt af selve beskrivelsen.

Ukrypterede nøglesæt

Når man skal bytte/dele offentlige nøgler eller nøglesæt til fælles nøgleringe, er det praktisk, at man kan eksportere dem fra systemet i ukrypteret form. Det sker simpelthen ved, at dataformatet i tabel 5.6 skrives til en fil. Disse ukrypterede nøglesæt får filendelsen **ukey**. Det ukrypterede nøglesæt kan gemmes på en diskette og derefter overdrages personligt, eller man kan sende det via en anden sikker kanal.

Det personlige nøglesæt

Når man benytter Latinum for første gang, skal man oprette et personligt nøglesæt. Dette nøglesæt giver efterfølgende adgang til ens personlige nøglering, som indeholder nøgler til de filer, man har adgang til. Disse filer kan både være andre

< priLængde >	::=	*short*
< pri >	::=	*byte**
< pubLængde >	::=	*short*
< pub >	::=	*byte**
< symLængde >	::=	*short*
< sym >	::=	*byte**
< type >	::=	*boolean*
< betroet >	::=	*boolean*
< beskrLængde >	::=	*short*
< beskr >	::=	*byte**
< nøgleDataFormat >	::=	< priLængde >< pri >< pubLængde > < pub >< symLængde >< sym > < type >< betroet >< beskrLængde > < beskr >

Figur 5.6: BNF for dataformatet for et nøglesæt.

nøgleringe og almindelige filer. På den måde organiseres nøglesæt i hierarkisk struktur jf. 3.2.4, hvor ens personlige nøglesæt udgør roden. Det er derfor kun nødvendigt, at være i besiddelse af ens personlige nøglesæt for at få adgang til resten.

Et personligt nøglesæt bliver gemt ved at kalde `getBytes` og derefter `encryptAES` metoden i `CryptoLib` klassen. På den måde kan man få adgang til sin personlige nøglering ved et enkelt kodeord. Det er naturligvis vigtigt for sikkerheden at vælge kodeordet med omhu.
Personlige nøglesæt-filer har filendelsen **key** og er som standard døbt **perskey.key**, dette kan ændres i konfigurationsfilen **latinum.conf**.

Tilladelser

Hvis man har en brugers offentlige nøgle, og vil give ham rettigheder til en fil, man er i besiddelse af, kan det gøres ved en tilladelse. En tilladelse er et nøglesæt krypteret med en brugers offentlige nøgle. Mængden af nøgler i tilladelsen afgør hvilke rettigheder der overføres til brugeren, som man udsteder tilladelsen til jf. 3.2.3.

Tilladelser oprettes og gemmes ved at kalde `getBytes` for en kopi af det nøglesæt der er associeret til datafilen, hvor man har tilpasset antallet af nøgler til de rettigheder, man vil give videre. Derefter krypteres filnøglen med brugerens offentlige nøgle ved hjælp af `encryptRSAHybWithSHA` metoden fra `CryptoLib` klassen.

$$H(U_{KU})\_H(KU).\mathrm{pkey}$$

Figur 5.7: Filnavn for tilladelser.

Tilladelser får filnavne som defineret i figur 5.7, hvor $H(U_{KU})$ er en base64 kodet[2] SHA-1 hash af brugerens offentlige nøgle, og hvor $H(KU)$ er en base64 kodet SHA-1 hash af den offentlige nøgle i nøglesættet.

5.3.2 Data

Datafiler er de filer man ønsker at dele via Latinum. Vi skelner imellem datafiler der er nøgleringe og andre datafiler, som f.eks. kan være programmer eller dokumenter. Hvis datafilen *ikke* er en nøglering vil dataformatet selvfølgelig afhænge af, hvilken type fil der er tale om.

Alle datafiler bliver krypteret med `encryptAESWithSHA` og får et filnavn som beskrevet i figur 5.8. Hvor $H(KU)$ er en base64 kodet SHA-1 hash af den offentlige nøgle for filen, og hvor $H(Data)$ er en base64 kodet SHA-1 hash af den krypterede fil.

$$H(KU)\_H(Data).\mathrm{file}$$

Figur 5.8: Filnavn for datafiler.

Nøgleringe

Da en nøglering er en samling af nøglesæt, er dataformatet for en nøglering enkelt at beskrive, ved hjælp af dataformatet for et nøglesæt, se tabel 5.7.

Felt	Beskrivelse	Type
0	nøgleSætLængde	short
1	nøgleSæt	byte[]

Tabel 5.7: Dataformat for en nøglering.

BNF'en for en nøglering kan ligeledes fremstilles ved hjælp af dataformatet for et nøglesæt, se nedenstående figur 5.9. Til at repræsentere en nøglering har vi implementeret klassen `KeyRing`, som indeholder objekter af `KeySet` klassen, svarende til de nøgler som er på nøgleringen. Dataformatet for en nøglering kan fås ved kalde metoden `getBytes` som returnerer et bytearray med det beskrevne dataformat.

$$\begin{array}{lcl}
<\text{nøgleSætLængde}> & ::= & short \\
<\text{nøgleSæt}> & ::= & <\text{nøgleDataFormat}> \\
<\text{nøgleringDataFormat}> & ::= & (<\text{nøgleSætLængde}><\text{nøgleSæt}>)^*
\end{array}$$

Figur 5.9: BNF for dataformatet for en nøglering.

[2]Resultatet af base64-kodningen modificeres ved at erstatte "/" med "-", for at undgå problemer med det underliggende filsystem.

Udover nøgleringenes specielle dataformat er der ikke forskel på nøgleringe og andre datafiler. Latinum benytter altså samme kryptografiske metode til at gemme dem og samme navnekonvention som for andre datafiler.

5.3.3 Revisionslog

Til hver datafil er der associeret en revisionslog. Revisionsloggen består af navnet på den oprindelige ukrypterede datafil samt en mængde loginddateringer (eng.: *logentry*) - en for hver opdatering der er lavet til filen.

I dataformatet for en revisionslog, som er beskrevet i tabel 5.8, er felt 2-16 en loginddatering, hvilket betyder, at felt 2-16 kan gentages flere gange jf. BNF'en på figur 5.10. Til at repræsentere henholdsvis en revisionslog og en loginddatering har vi implementeret klasserne `Log` og `LogEntry`.

Felt	**Beskrivelse**	**Type**
0	Filnavnlængde	short
1	Filnavn	byte[]
2	ByggerPåHashLængde	short
3	ByggerPåHash	byte[]
4	HashLængde	short
5	Hash	byte[]
6	Beskrivelseslængde	short
7	Beskrivelses	byte[]
8	FilNøgleSignaturLængde	short
9	FilNøgleSignaturType	byte
10	FilNøgleSignatur	byte[]
11	BrugerNøgleSignaturLængde	short
12	BrugerNøgleSignaturType	byte
13	BrugerNøgleSignatur	byte[]
14	BrugerNøgleLængde	short
15	BrugerNøgleType	byte
16	Brugernøgle	byte[]

Tabel 5.8: Dataformatet for en revisionslog.

I dataformatet for en revisionslog gives først længden af filnavnet for den oprindelige ukrypterede datafil efterfulgt af selve filnavnet. Så følger et antal loginddateringer, der er bygget op som beskrevet i tabel 5.8 fra felt 2-16.

I en loginddatering kommer først længden af hashværdien efterfulgt af selve hashværdien. Denne hashværdi (3) er en hashværdi af den forudgående krypterede datafil, som den nye datafil er en opdatering af. Så følger længden af endnu en hashværdi og selve hashværdien (5). Denne hash er en hash af den krypterede datafil, som loginddateringen repræsenterer.

Derefter kommer længden af beskrivelsen efterfulgt af selve beskrivelsen. Denne beskrivelse kan f.eks. sige noget om, hvad der er nyt ved denne opdatering.

Så kommer længden af signaturen, så en værdi for hvilken signaturalgoritme, der er brugt til at signere datafilen, og endelig selve signaturen. Den udviklede prototype understøtter kun SHA-1/RSA-signaturer.

De næste felter er valgfri for en loginddatering. Hvis en bruger har valgt at signere opdateringen vil felterne 12-16 også være tilstede, og i så fald kommer der en længde for en signatur, så en værdi for hvilken signaturalgoritme brugeren har brugt til at signere datafilen og selve signaturen. Dernæst kommer længden af nøglen, brugeren har brugt til at signere datafilen med, samt hvilken type det er, endelig følger brugerens offentlige nøgle.

< navnlængde >	::=	*short*
< navn >	::=	*byte**
< bpHashLængde >	::=	*short*
< bpHash >	::=	*byte**
< HashLængde >	::=	*short*
< Hash >	::=	*byte**
< besLængde >	::=	*short*
< beskrivelse >	::=	*byte**
< fnSigLængde >	::=	*short*
< fnSigType >	::=	*byte*
< fnSignaturen >	::=	*byte**
< bnSigLængde >	::=	*short*
< bnSigType >	::=	*byte*
< bnSignaturen >	::=	*byte**
< bnLængde >	::=	*short*
< bnType >	::=	*byte*
< bnøglen >	::=	*byte**
< logEntryNoUS >	::=	< bpHashLængde >< bpHash >< HashLængde > < Hash >< besLængde >< beskrivelse > < fnSigLængde >< fnSigType > < fnSignaturen >< bnSigLængde >
< logEntryUS >	::=	< logEntryNoUS >< bnSigType > < bnSignaturen >< bnLængde >< bnType > < bnøglen >
< logEntry >	::=	< logEntryNoUS > | < logEntryUS >
< logFilFormat >	::=	< navnlængde >< navn >< logEntry >*

Figur 5.10: BNF for dataformatet for en revisionslog.

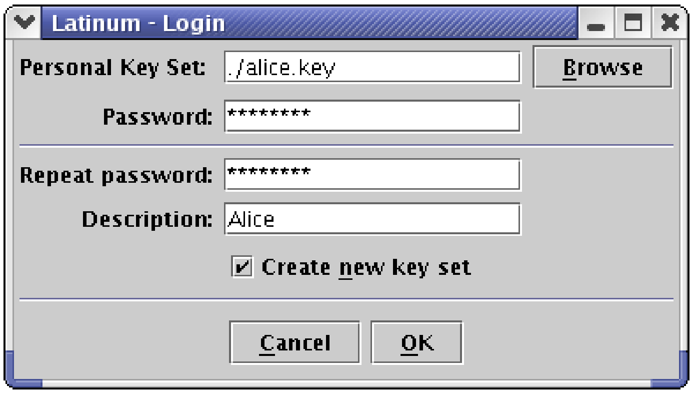
5.4 Brugergrænseflade

Brugergrænsefladen er blevet udviklet ved hjælp af Javas Swing komponenter. Det betyder at brugergrænsefladen vil optræde ens på tværs af platforme. De fire grundlæggende dele af brugergrænsefladen, identificeret under designfasen, er blevet implementeret som paneler placeret i en fanebladsstruktur. Klassen

`LatinumGUI` er ansvarlig for at instantiere fanebladsstrukturen og de fire paneler. Men før dette kan ske, skal det personlige nøglesæt dekrypteres.

5.4.1 Login

Når programmet startes vises en login-dialogboks. Her kan brugeren vælge en eksisterende personlig nøgle eller oprette en ny. Vælges der en eksisterende nøgle, forsøges denne nøgle dekrypteret med det indtastede kodeord. Lykkes dette, åbnes hovedskærmbilledet. Hvis den personlige nøglering er tilstede, dekrypteres den.



Figur 5.11: Login-dialog.

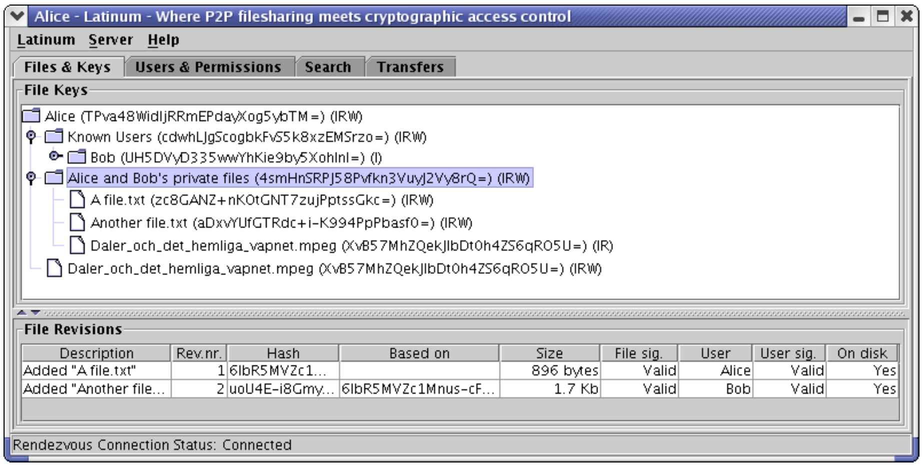
Denne funktionalitet ligger i `KeyManager` klassen samt `LoginDialog` klassen, der som navnet antyder er ansvarlig for at vise login-dialogen. Denne er vist i figur 5.11.

Hvis man ønsker at anvende Latinum på en anden computer, end den hvor man oprindeligt oprettede sin personlige nøgle, gøres dette ved medbringe nøglen, f.eks. på et hukommelseskort eller en diskette. Nøglen er krypteret som beskrevet i afsnit 5.2.3. Så skulle lagermediet blive stjålet, er det altså ingen katastrofe, så længe man har valgt et fornuftigt kodeord.

Hvis den personlige nøglering ikke findes på det lokale system, kan man foretage en søgning efter den, nøjagtigt som man kan søge efter andre filer, man har nøglerne til. Hvis man har ladet den computer man normalt benytter stå tændt, kan man hente nøgleringen fra denne. Alternativt kan man hente den fra andre brugere, der har været så venlige at replikere ens nøglering.

5.4.2 Filer & Nøgler

Dette faneblad er implementeret i `FileKeyPanel` klassen. Nøgler organiseres i en træstruktur, hvor den personlige nøgle udgør roden. Se figur 5.12.



Figur 5.12: Filer & Nøgler.

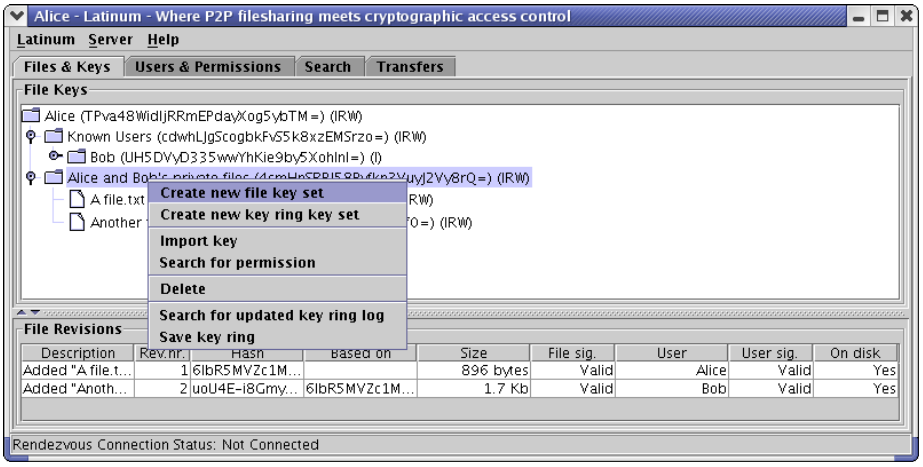
For hvert nøglesæt vises beskrivelsen først, i parentes bagefter vises en base64 kodet hash af den offentlige nøgle. Dette kan være nyttigt, hvis man har brug for at identificere de tilhørende krypterede filer. Endelig vises hvilke nøgler der er på nøglesættet: Den offentlige nøgle markeres med et I (Integrity), den symmetriske med et R (Read) og den private nøgle med et W (Write). Som det fremgår af figur 5.14, har man kun mulighed for at tilføje en ny revision af filen, hvis man har skriverettigheder (IRW). Tilsvarende kan man kun dekryptere en fil, hvis man har læserettigheder (IR).

Nye nøgler dannes ved at højreklikke og vælge "Create new file key set" eller "Create new key ring key set". Højreklikkes der på en eksisterende nøglering, tilføjes den nye nøgle til denne nøglering. Højreklikkes der på en filnøgle, tilføjes den nye nøgle på den samme nøglering, som den valgte filnøgle tilhører. Se figur 5.13.

Nøgleringe indlæses efter behov, første gang nøglen til nøgleringen ekspanderes. `KeyManager` klassen benyttes til at styre indlæsningen af nøgleringe og til tilføjelse af nøgler til nøgleringe. Den betjener sig af `KeyRing` klassen, der benyttes til intern repræsentation af en nøglering. Klassen indeholder bl.a. metoder til at gemme og indlæse nøgleringe.

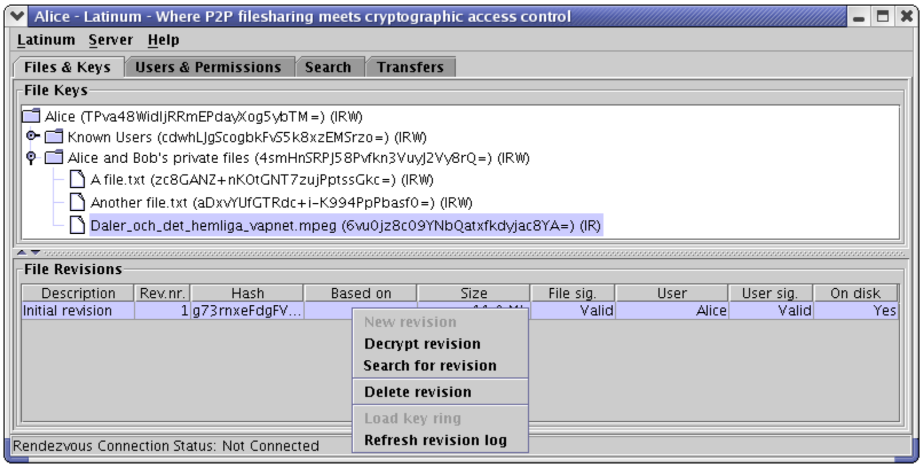
Nøgler kan kopieres til andre nøgleringe ved hjælp af "drag and drop". Dette fungerer ved først at vælge en nøgle med musen, dernæst holdes venstre museknap nede, mens musen føres hen til den ønskede nøglering, her slippes knappen og man kan nu vælge hvilke dele af nøglen, der skal kopieres. Denne mekanisme fungerer ved hjælp af `KeySetTransferHandler` klassen. Her oprettes der en kopi af nøglesættet med de valgte nøgler. Dette indsættes på den valgte plads i træet.

Når en filnøgle eller nøglering vælges, vises den tilhørende revisionslog i en tabel nedenfor. Første gang en revisionslog skal vises, indlæses den fra disken, fremover hentes den blot fra revisionslog-cachen. Denne cache er organiseret som



Figur 5.13: Filer & Nøgler, højrekliksmenu.

en hashtabel, der indeholder referencer til `Log` objekter. Hashværdien af filens offentlige nøgle benyttes som indeks i hashtabellen.



Figur 5.14: Filer & Nøgler, Filrevisioner, højrekliksmenu.

Hvis man ønsker at dekryptere en fil, sker dette ved at højreklikke på den ønskede revision og vælge "Decrypt revision". Tilsvarende kan man opdatere en fil ved at højreklikke på den revision man har arbejdet med og vælge "New revision". Se figur 5.14.

I `FileKeyPanel` klassen er der implementeret metoder, der håndterer disse kommandoer. `decrypt` metoden lader brugeren vælge hvor og under hvilket navn den dekrypterede fil skal gemmes. Dernæst kaldes `decrypt` metoden i `StorageManager` klassen. "New revision" fungerer tilsvarende, men her skal brugeren i stedet vælge hvilken fil der skal tilføjes til systemet. Efterfølgende kaldes `encrypt` metoden i `StorageManager` klassen.

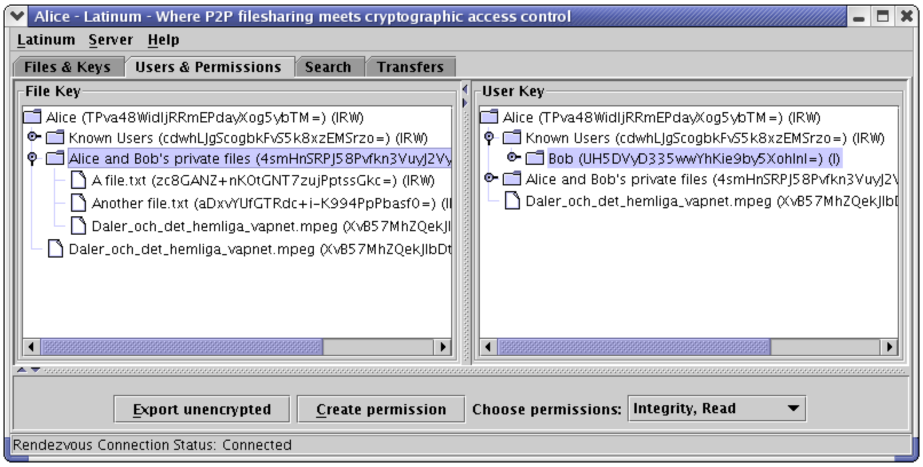
`StorageManager` klassen fungerer som grænseflade til de underliggende krypteringsfunktioner i `CryptoLib` klassen.

Felterne “Hash” og “Based on” gør det muligt, at holde øje med hvilke revisioner der er baseret på hinanden. Hvis der ikke forekommer konflikter vil en ny revision altid være baseret på den foregående. Hvis en bruger har baseret en opdatering på en gammel revision, vil det således fremgå. Systemet gør ikke noget forsøg på at sammenflette konfliktende opdateringer, men revisionsloggen forsøges flettet således at brugerne kan opdage og håndtere konflikten, se afsnit 4.3.1.

Gamle revisioner vil forblive tilgængelige, med mindre man eksplicit sletter dem.

5.4.3 Tilladelser

Hvis man ønsker at give en anden bruger adgang til en fil eller en nøglering, gøres dette ved at oprette en *tilladelse*. Først vælges den filnøgle man ønsker at give adgang til, dernæst vælges den offentlige nøgle for den bruger der skal have adgang. Der dannes nu en ny fil, hvis indhold består af filnøglen krypteret med den valgte offentlige nøgle, som beskrevet i afsnit 5.3.1. Denne fil deles automatisk således, at brugeren der har fået adgang, kan finde den.



Figur 5.15: Tilladelser.

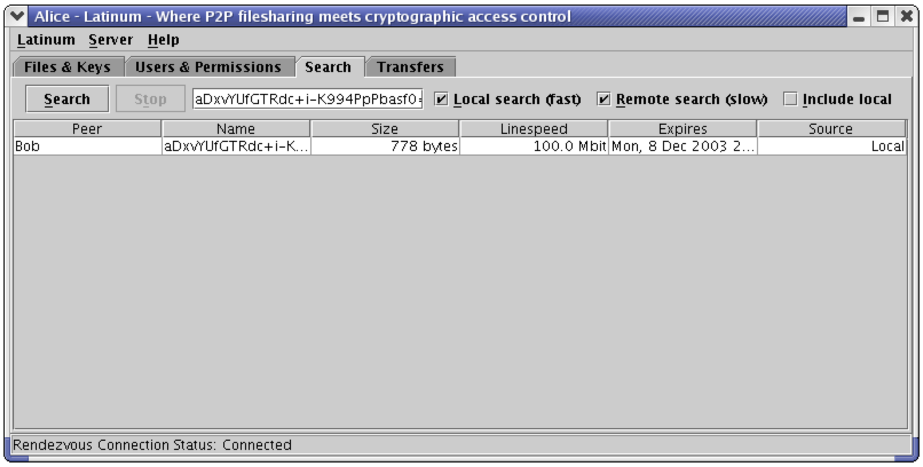
Man kan også vælge at eksportere en filnøgle ukrypteret, dette er især nyttigt, hvis andre skal have ens offentlige nøgle. I det tilfælde deles den ukrypterede nøgle ikke. I stedet fremkommer en dialogboks, der giver mulighed for at gemme den dannede fil, hvor man ønsker det.

I begge tilfælde kan man vælge hvilke rettigheder/nøgler, der skal gives tilladelse til eller eksporteres.

De to træstrukturer vist i figur 5.15 deler datamodel med træstrukturen under “Filer & Nøgler”. Ændringer i denne vil derfor straks afspejles.

5.4.4 Søgning

Figur 5.16 viser søgefanebladet. Søgninger kan startes ved at udfylde søgefeltet og trykke på “Search”-knappen. Men oftest vil søgninger være startet fra “Files & Keys”-fanebladet, hvor man ved højreklik kan starte en søgning efter en revisionslog eller en opdateret fil.



Figur 5.16: Søgning.

Selve søgningen foregår asynkront i `SearchController` klassen. Søgeresultaterne vil blive vist i tabellen efterhånden som de findes. Som beskrevet i afsnit 5.1.2 vil lokale søgninger (“Local search”) gå hurtigt, mens søgninger der sendes ud på netværket (“Remote search”), kan være længe om at give svar. Det kan sagtens forekomme, at den samme annoncering findes både i den lokale database og senere bliver modtaget som svar på en netværkssøgning. I disse tilfælde kasseres dubletter og den først fundne annoncering vises.

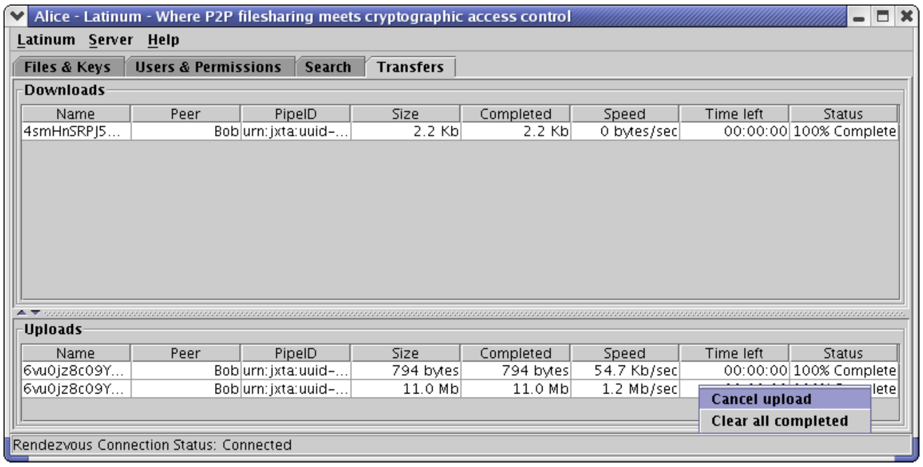
Er der valgt “Remote search”, fortsætter søgningen indtil brugeren vælger at stoppe den. Trykkes der på “Stop”-knappen notificeres `SearchController` klassen og søgningen stopper.

“Include local”-checkboksen benyttes til at afgøre om annonceringer, der stammer fra den lokale node, skal medtages i søgeresultatet. Dette er man normalt ikke interesseret i. Men til testformål, eller hvis man ønsker at kontrollere, at en fil er blevet korrekt publiceret, kan muligheden være nyttig.

Hvis man ønsker at hente en af de fundne filer, gøres dette ved at dobbeltklikke på den.

5.4.5 Filoverførsler

Når man har dobbeltklikket på et søgeresultat, oprettes der en ny instans af `Download` klassen. Dette håndteres af `DownloadControlleren`, der også sørger for at opdatere status for de igangværende filtransporter.



Figur 5.17: Filoverførsler.

Hvis man ønsker at afbryde en overførsel, gøres dette ved at højreklikke på overførslen og vælge "Cancel download" henholdvis "Cancel upload". Det forårsager at den `JxtaSocket` der bliver brugt til overførslen, bliver lukkket, hvilket også opdages af den node overførslen sker fra eller til. `Download` tråden og den tilsvarende `Servant` tråd vil herefter terminere.

5.5 Opsummering

I dette kapitel er det blevet beskrevet, hvordan det udviklede design er blevet implementeret.

Implementeringen omfatter fire dele:

Kommunikation: JXTA benyttes til at publicere og søge efter filer. Ovenpå en pålidelig JxtaSocket forbindelse, benyttes en enkel request/response-orienteret protokol til filoverførsler.

Kryptering: Suns JCE arkitektur benyttes til at generere nøgler, udføre selve krypteringen samt til signaturer. Udover Suns indbyggede krypteringsalgoritmer, benyttes Bouncy Castles RSA implementering til at udføre asymmetrisk kryptering.

Dataformater: Der er blevet defineret dataformater for nøgler, datafiler og den tilhørende revisionslog. Nøgler kan optræde i tre forskellige formater: Ukrypteret (udveksling af offentlige nøgler), krypteret med et kodeord (det personlige nøglesæt) og krypteret med brugerens offentlige nøgle (en tilladelse).

Brugergrænseflade: Brugergrænsefladen omfatter fire paneler placeret i en fanebladsstrukur samt en login-dialog. De fire faneblade opdeler betjeningen af programmet i følgende dele: Adminstration af filer og nøgler, tilladelser, søgning og filoverførsler.

Kapitel 6

Evaluering

I dette kapitel vil det udviklede system blive evalueret, med hensyn til hvordan kravene opstillet i afsnit 3.1 er blevet indfriet. Desuden gennemføres en funktionel test af programmet.

Dernæst anlægges en mere overordnet synsvinkel, og selve idéen om at indføre sikkerhed i peer-to-peer netværk, ved hjælp af kryptografisk adgangskontrol, forsøges evalueret på baggrund af de opnåede erfaringer.

I slutningen af kapitlet gives en række forslag til det videre arbejde indenfor området.

6.1 Evaluering af applikationen

En afprøvning af applikationen afslører, at det er let at oprette filnøgler og dele filer. Nøgleringe oprettes ligeledes uden problemer.

Selv med en hurtig processor tager det dog adskillige sekunder at generere de lange RSA nøgler. Det ville derfor have været en fordel, hvis man kunne vælge en nøglelængde, der passede til ens sikkerhedsbehov. Selv 512 bit nøgler, tager trods alt stadig en del tid at faktorisere, og ville således i mange sammenhænge give tilstrækkelig sikkerhed.

Til gengæld sker kryptering og dekryptering af filer ganske hurtigt. En fil der fylder omkring 10 Mb, krypteres således på et par sekunder, ganske som det kunne forventes ifølge benchmarken i afsnit 2.6.2.

Den mest konservative løsning, med hensyn til valg af symmetrisk krypteringsalgoritme, havde været at vælge 3DES frem for AES. Kryptering og dekryptering havde så tilgengæld været 4-5 gange langsommere.

Nøgler kan eksporteres ukrypteret. Denne funktionalitet kan benyttes hvis man ønsker at give sin offentlige nøgle til en anden bruger. Har man selv en anden brugers offentlige nøgle, kan man give vedkommende adgang til en fil eller en nøglering ved at kryptere filnøglen med brugerens offentlige nøgle.

Gruppebegrebet er understøttet ved hjælp af nøgleringe. De filer og nøgleringe, som gruppens medlemmer ønsker at dele, placeres på samme nøglering. Nøglen til nøgleringen overdrages derpå til hvert gruppemedlem. Dette kræver at man kender de enkelte gruppemedlemmers offentlige nøgler i forvejen. Hvis gruppens medlemmer i forvejen deler en nøglering, kan nøglerne til den nye nøglering blot placeres på den eksisterende nøglering.

Modtager man en nøgle fra en anden bruger, enten i krypteret eller ukrypteret form, importeres den let. For at få adgang til selve filen, skal man først søge efter noder, der tilbyder den tilhørende revisionslog. Denne skal så hentes. Når det er gjort, vælger man en konkret revision fra loggen, og søger efter noder, der tilbyder denne. Når en konkret version således er fundet og hentet, kan man dekryptere filen.

Denne procedure kan være ganske rimelig, hvis forskellige noder jævnlig opdaterer filen. Men i en del tilfælde deles data en gang og opdateres aldrig siden. Dette kunne f.eks. gælde et feriebillede eller en hjemmevideo. I disse tilfælde virker proceduren med revisionsloggen noget omfattende og kan ikke helt retfærdiggøres.

Arbejdsgangen kunne forkortes, hvis systemet automatisk undersøgte om der fandtes opdaterede revisionslogge, og hentede disse, hvis der gjorde. Ydermere kunne systemet også automatisk hente den seneste, og eventuelt eneste, version af filen.

Den beskrevne funktionalitet kunne sagtens implementeres, men findes altså ikke i prototypen.

Der gøres ikke noget forsøg på at forhindre samtidige opdateringer af den samme fil. Men disse vil fremgå af revisionsloggen, da det altid angives hvilken foregående revision, der ligger til grund for en given revision. Det overlades til brugerne af systemet at sammenflette eventuelt konfliktende opdateringer.

JXTAs annonceringsystem gør, at når man foretager søgninger, kan man modtage svar fra alle noder der i søgeøjeblikket er tilsluttet netværket. Den eftersøgte fil kan herefter hentes fra en vilkårlig af de disse noder.

Samlet set synes den udviklede applikation således at honorere de opstillede krav.

Vi har ikke udført afprøvning af systemet med et stort antal noder. Systemets evne til at skalere til et stort antal noder vil afhænge af hvor godt den underliggende JXTA arkitektur skalerer. Dog vil det udviklede program nok blive lidt besværligt at bruge, hvis man skal håndtere mange tusind brugere og filer. Dette er primært et spørgsmål om brugervenlighed, da den nuværende brugergrænseflade ikke er designet til at håndtere virkelig mange filer og brugere.

6.2 Afprøvning

Ved udvikling af et program er det nødvendigt at udføre en afprøvning af programmet. Afprøvningen har til opgave at sandsynliggøre, at programmet virker efter hensigten. Omfanget af afprøvingen er i høj grad bestemt af programmets funktion. Styrer programmet livsvigtige funktioner som f.eks. flytrafik, vil man nok lave en formel specifikation som kan verificeres og teste, om programmet opfylder specifikationen. Skal programmet styre moderat vigtige funktioner som f.eks. backup af data, vil man nok lave en strukturel og funktionel test som beskrevet i [Ses98].

En strukturel test har til opgave at checke, om alle dele af programmet kan køre. Man er derfor nødt til at lave testværdier som kommer ud i alle forgreninger af programmet. En funktionel test har til opgave at checke, om programmet opfører sig som forventet.

Man udarbejder derfor en række testcases med tilhørende beskrivelser af hvad man forventer at programmet gør. Når disse testcases afprøves i programmet, sammenlignes den aktuelle opførsel så med den forventede.

Når man vælger testcases til en funktionel test, er det vigtigt både at vælge cases som er typiske og cases som er ekstreme. Typiske cases er cases, der giver mening at bruge i programmet, hvorimod ekstreme cases er cases, der er unaturlige eller forkerte at bruge i programmet. Et klassisk eksempel på en ekstrem case er negative tal i f.eks. et ordresystem, da det ikke giver mening at bestille et negativt antal varer. Konsekvensen af dette kunne være, at kunder fik fratrukket varernes pris, i stedet for at få den lagt til, på deres regning.

En strukturel test for et program på flere tusinde linjer vil være meget tidskrævende. En strukturel test bruges i høj grad til at finde nytteslløse eller ulogiske dele af et program. En funktionel test bruges til, at finde ud af om programmet løser problemerne rigtigt [Ses98]. Vi har derfor valgt at fokusere på den funktionelle test.

Den funktionelle test består som førnævnt af en række forskellige testcases. Hver af disse testcases består af en beskrivelse af hvad der testes, hvad vi forventer vil ske, og om det faktiske udfald var identisk med, hvad vi forventede. Vores funktionelle test kan ses i bilag A.

Som det fremgår afslørede den funktionelle test ingen fejl. Det betyder ikke, at vores program ikke kan have fejl. Derimod har den funktionelle test sandsynliggjort, at den afprøvede funktionalitet virker som den skal.

6.3 Evaluering af sikkerhedsmodellen

Sikkerhedsmekanismerne i traditionelle filsystemer er baseret på tilstedeværelsen af en referencemonitor. Prøver man at overføre denne mekanisme til peer-to-peer netværk er problemet, at hver node har sin egen referencemonitor, og der er ingen garanti for, at disse referencemonitorer opfører sig ens, og håndhæver rettigheder på en ensartet måde.

Der er derfor brug for en mekanisme, der kan håndhæve rettigheder uden brug af en referencemonitor. Kryptografisk adgangskontrol giver netop mulighed for at indføre læse- og skriverettigheder og giver desuden konfidentialitet, integritet og autentifikation uden brug af en referencemonitor.

Dette projekt viser at mekanismen virker, og er praktisk anvendelig i et peer-to-peer fildelingssystem.

Læserettigheder håndhæves effektivt, da brugere uden den symmetriske nøgle ikke kan få noget ud af data.
Skriverettigheder håndhæves mere implicit, idet intet forhindrer en bruger uden skriverettigheder i at lave sin egen opdaterede udgave af filen. Sikkerheden består i, at andre brugere med adgang til filen, ikke accepterer opdateringer, der ikke er signeret med den private filnøgle.

Hvis man kun er i besiddelse af den offentlige nøgle fra et filsæt, kan man ikke dekryptere filen. Man har dog mulighed for at hente og dekryptere revisionsloggen. Dermed får man adgang til information om de konkrete revisioner af filen. Disse kan hentes og deres integritet valideres. Denne mekanisme kan udnyttes, hvis man f.eks. ønsker at benytte en maskine, som man ikke stoler på, til distribution af filerne. Dette kunne f.eks. være en server placeret hos en internetudbyder.

Kryptografisk adgangskontrol kan dog ikke umiddelbart anvendes i alle situationer. Da systemet er baseret på de to grundliggende operationer i public-key kryptering, er det vanskeligt at udvide systemet til at håndtere flere rettigheder end blot læse- og skriverettigheder.

Kryptografisk adgangskontrol er ikke specielt velegnet i systemer, hvor der benyttes MAC (MAC - Mandatory Access Control), da intet forhindrer brugerne i et overdrage nøgler til hinanden. Man kan dog argumentere for, at systemer baseret på kryptografisk adgangskontrol er mere robuste, da sikkerheden ikke står og falder med sikkerheden i en referencemonitor.

6.4 Videre arbejde

I forbindelse med udvikling af prototypen af Latinum er vi blevet opmærksomme på en række udvidelsesmuligheder, der ikke er blevet udforsket. Vi vil i dette afsnit kort beskrive disse udvidelsesmuligheder og give forslag til, hvordan de kunne implementeres.

Nogle af arbejdsprocesserne i Latinum kan med fordel forenkles. Hvis man f.eks. ønsker at hente en fil, man ikke tidligere har haft adgang til, skal man følgende skridt igennem:

Først skal man søge efter en tilladelse til filen og overføre den. Dernæst skal man søge efter og overføre revisionsloggen. Herefter kan man overføre de revisioner man er interesseret i.

Denne arbejdproces kan automatiseres ved f.eks. at tilføje et højrekliksmenupunkt, man kunne kalde for *search & download latest* under *File keys*. Dette kunne starte en funktion, som selv finder og overfører revisionsloggen, og dernæst overfører den seneste revision.

Den implementerede revisionsstyring betragter en opdatering af en fil som fuldstændig forskellig fra den version man opdaterer fra. Det betyder at hvis en bruger laver en lille ændring i en stor fil, så skal de andre brugere, der vil have den opdaterede revision, overføre hele filen. Dette er upraktisk, da mange af brugerne sikkert havde størstedelen af filen i forvejen.

Det ville derfor være mere effektivt, hvis man kun gemte ændringerne, så brugere, der havde den revision som opdateringen bygger på, kun behøvede at downloade ændringerne og flette dem ind den gamle version.

En anden løsningsmodel kunne være at segmentere filoverførslerne. På den måde behøvede brugerne kun at overføre de segmenter, der har ændret sig fra den revision de har, til den nye revision.

En umiddelbar positiv effekt ved begge disse løsninger er, at den vil øge raten hvormed filerne kan udbredes i netværket, da der skal overføres færre data.

Forskellige folk vil have forskellige behov for sikkerhed. Derfor kunne det være nyttigt, hvis brugerne selv kunne vælge nøglestørrelser og kryptografiske algoritmer. Det ville være specielt rart, at kunne vælge et alternativ til RSA og dens store nøgler. Det burde ikke være svært at udvide Latinum med denne funktionalitet, når der først kommer providers, som understøtter de kryptografiske algoritmer, man måtte ønske at anvende.

Kapitel 7

Konklusion

I dette eksamensprojekt er mulighederne for at indføre læse- og skriverettigheder, samt konfidentialitet, integritet og autentifikation i peer-to-peer fildelingsnetværk blevet undersøgt. Der er blevet udviklet en sikkerhedsmodel baseret på kryptografisk adgangskontrol, og denne er analyseret. På baggrund af analysen er der udarbejdet et design for et fildelingsnetværk med de ønskede egenskaber. Endelig er der blevet udviklet en prototype, der implementerer det foreslåede design. En gennemført funktionel test viser, at den udviklede prototype fungerer efter hensigten.

Rapporten indledes med en gennemgang og efterfølgende klassifikation af en række eksisterende peer-to-peer systemer. Klassifikationen afslører, at systemerne med hensyn til placering af metadata benytter tre forskellige strategier: Metadata kan placeres på centrale servere (f.eks. Napster), på dynamisk udnævnte supernoder (hybridmodellen, f.eks. Kazaa) eller fuldstændigt decentralt, på de samme noder som stiller data tilrådighed (f.eks. klassisk Gnutella). Hvis man ønsker den højeste grad af skalerbarhed og tilgængelighed, er hybridmodellen den mest attraktive. Men det er samtidig også den mest komplekse af de tre modeller.

Der er blevet opstillet en række brugsscenarier. Disse omfatter to primære anvendelser: Udgivelse af digitalt indhold og kollaborativt arbejde. Brugsscenarierne udmunder i en kravspecifikation, der er blevet benyttet under udvikling af sikkerhedsmodel og programdesign.

Den valgte netværksarkitektur er baseret på hybridmodellen, hvor nogle noder fungerer som indeksnoder for de øvrige. Indeksnoderne vedligeholder et distribueret katalog over de tilgængelige filer og varetager desuden søgninger i dette katalog. Løsningen er baseret på Suns JXTA framework, der tilbyder publicerings- og søgefunktionalitet samt de nødvendige kommunikationsprimitiver.

Selve de delte data placeres derimod *ikke* efter en distribueret hashfunktion, som det f.eks. kendes fra CFS og PAST. Dette valg skyldes, at udskiftningsraten blandt noderne i det udviklede system vil være for høj til, at det kan betale sig.

Data placeres således på de noder der deler og henter data. Dette svarer til, hvad der kendes fra eksisterende peer-to-peer fildelingssystemer, som f.eks. Napster og Gnutella.

Den udviklede sikkerhedsmodel er bygget op omkring nøglesæt og nøgleringe. Alle filer, nøgleringe og brugere i systemet repræsenteres ved individuelle nøglesæt. Et nøglesæt indeholder tre nøgler: En offentlig, en symmetrisk og en privat, hvor den offentlige og den private nøgle udgør et asymmetrisk nøglepar. Nøglerne giver mulighed for tre niveauer af rettigheder: integritet, læserettigheder og skriverettigheder. Hvis der er tale om en brugers offentlige nøgle, vil besiddelsen af denne gøre det muligt, at udstede tilladelser til denne bruger. Tilladelsen dannes ved at kryptere et nøglesæt med brugerens offentlige nøgle.

Nøglesæt kan samles på nøgleringe. Adgangen til en nøglering kontrolleres ligeledes af et nøglesæt. Hvis man er besiddelse af et nøglesæt med den offentlige og den symmetriske nøgle, og således har læseadgang, har man adgang til alle nøglerne på nøgleringen. Har man yderligere den private nøgle og dermed skriveadgang, kan man også tilføje nøgler til nøgleringen.

Sikkerhedsmodellen er ganske fleksibel, idet den kan benyttes både i RBAC-miljøer og i almindelige DAC-miljøer. I det første tilfælde svarer adgangen til en bestemt nøglering til at have en bestemt rolle. I det andet tilfælde benyttes en nøglering i forbindelse med grupper, ved at placere gruppens fælles ressourcer på samme nøglering. Gruppemedlemskab tildeles ved at overdrage nøglerne til gruppenøgleringen.

Endelig kan den udviklede sikkerhedsmodel benyttes i forbindelse med PKI. Her har kun certifikatautoriteten skriveadgang til en bestemt nøglering. Godkendte offentlige nøgler kan nu placeres på denne nøglering. Brugere med læseadgang til nøgleringen kan kontrollere offentlige nøgler ved at undersøge om disse eksisterer på nøgleringen.

Analysen af mulige trusler viser, at modellen giver god sikkerhed med hensyn til sikring af konfidentialitet og integritet. Men at der mangler beskyttelse mod trafikanalyse og oversvømmelsesangreb. I forbindelse med den påtænkte anvendelse af systemet, mener vi dog ikke, at muligheden for denne type angreb vil udgøre noget reelt problem.
Sikringen af konfidentialitet og integritet er afhængig af hvilke beslutninger, der træffes med hensyn til valg af krypteringsalgoritmer. Det er afgørende for sikring af integriteten, at en lineær symmetrisk krypteringsalgoritme ikke kombineres med en lineær hashfunktion. I den udviklede prototype er der taget højde for denne problematik, idet AES benyttes som krypteringsalgoritme og SHA-1 som hashfunktion. Begge disse algoritmer er således ikke lineære.

Signaturer og asymmetrisk kryptering håndteres i den udviklede prototype af RSA algoritmen. Denne er både gennemprøvet og sikker, men desværre ikke den hurtigste løsning. Dette skyldes at der skal benyttes store nøgler, hvis sikkerheden skal være af samme størrelsesorden, som den man finder i den benyttede symmetriske krypteringsalgoritme. Det vil derfor være en overvejelse værd, at benytte elliptisk kurve-kryptering når produktionsklare implementeringer på et tidspunkt bliver alment tilgængelige.

Bilag A

Funktionel test

I den nedenstående tabel ses de enkelte test som udgør den funktionelle test. Til hver test er der knyttet et testnummer, en beskrivelse af testen, en forventet opførelse og en sammenligning mellem forventet og aktuel opførelse. I den sidste kolonne indikerer et $\surd$, at testresultatet var identisk med det forventede. Mangler der et $\surd$ i den sidste kolonne betyder det, at der ikke er overenstemmelse mellem det forventede og det opnåede testresultat.

#	**Test beskrivelse**	**Forventet opførelse**	$\surd$
1	Start af Latinum.	Forventer at programmet starter uden nogle exceptions	$\surd$
2	Lukning af latinum.	Forventer at programmet lukker uden nogle exceptions.	$\surd$
3	Oprettelse af personligt nøglesæt.	Forventer at nøglesættes oprettes og kan bruges både i den aktuelle session og i fremtidige sessioner.	$\surd$
4	Brug af et prægenereret personligt nøglesæt	Forventer at nøglesættet kan bruges i den aktuelle session.	$\surd$
5	Tilslutning til rendezvous node	Forventer at programmet kan lave en forbindelse til en rendezvous node.	$\surd$
6	Oprettelse af en filnøgle	Forventer at nøglesættet for filen oprettes, filen krypteres og nøglesættet tilføjes til en nøglering.	$\surd$
7	Oprettelse af en nøglering	Forventer at nøglesættet for nøgleringen oprettes og tilføjes til ens personlig nøglering.	$\surd$
8	Kopiering af en filnøgle fra en nøglering til en anden	Forventer at nøglesættet for filen kopieres til den ønskede nøglering.	$\surd$
9	Kopiering af en nøglering til en anden nøglering	Forventer at nøglesættet for nøgleringen kopieres til den ønskede nøglering.	$\surd$

10	Lagring af en nøglering	Forventer at nøgleringen gemmes i krypteret form med de ændringer der er foretaget.	$\surd$
11	Sletning af filnøgle	Forventer af nøglesættet til filen slettes og fjernes fra nøgleringen.	$\surd$
12	Sletning af revisionsloggen for en fil	Forventer at revisionsloggen slettes.	$\surd$
13	Sletning af filnøgle + revisionslog.	Forventer det samme som i 11 + 12.	$\surd$
14	Sletning af filnøgle + revisionslog + fil.	Forventer det samme som i 13 samt at filen slettes.	$\surd$
15	Dekryptering af en fil man har læserettighed til.	Forventer at filen bliver dekrypteret rigtigt.	$\surd$
16	Dekryptering af en fil man ikke har læserettighed til.	Forventer at få en fejlmeddelelse.	$\surd$
17	Opdatering af en fil man har skriverettighed til.	Forventer at filen bliver krypteret og revisionsloggen opdateres.	$\surd$
18	Opdatering af en fil man ikke har skriverettighed til.	Forventer at programmet ikke tillader det.	$\surd$
19	Sletning af en revision af en fil.	Forventer at filen slettes fra disken.	$\surd$
20	Validering af filsignatur for en umodificeret fil man har på disken	Forventer at signaturen er sand og signaturfeltet ,"File sig.", derfor indeholder teksten "valid".	$\surd$
21	Validering af filsignatur for en umodificeret fil man ikke har på disken.	Forventer at signaturfeltet "File sig." er tomt.	$\surd$
22	Validering af filsignatur for en modificeret fil man har på disken	Forventer at signaturen er falsk og signaturfeltet "File sig." indeholder teksten "invalid".	$\surd$
23	Validering af brugersignatur for en umodificeret fil man har på disken	Forventer at signaturen er sand og signaturfeltet "User sig." inderholder teksten "valid".	$\surd$
24	Validering af brugersignatur for en umodificeret fil man ikke har på disken.	Forventer at signaturfeltet "User sig" er tomt.	$\surd$
25	Validering af brugersignatur for en modificeret fil man har på disken	Forventer at signaturen er falsk og signaturfeltet "User sig." indeholder teksten "invalid".	$\surd$
26	Eksport af et nøglesæt for en fil i ukrypteret form	Forventer at nøglesættet skrives til disken i ukrypteret form.	$\surd$
27	Eksport af et nøglesæt for en nøglering i ukrypteret form	Forventer at nøglesættet skrives til disken i ukrypteret form.	$\surd$

28	Import af et nøglesæt for en fil i ukrypteret form	Forventer at nøglesættet bliver tilføjet nøgleringen, som det er blevet impoteret til.	$\surd$
29	Import af et nøglesæt for en nøglering i ukrypteret form	Forventer at nøglesættet bliver tilføjet nøgleringen, som det er blevet impoteret til.	$\surd$
30	Søgning efter en opdateret revisionslog for en fil der er blevet opdateret	Forventer at loggen bliver fundet, hvis den er tilgængelig på netværket (begge tilfælde er testet).	$\surd$
31	Søgning efter en opdateret revisionslog for en nøglering der er blevet opdateret	Forventer at finde revisionsloggen, hvis den er tilgængelig på netværket (begge tilfælde er testet).	$\surd$
32	Søgning efter en specifik revision af en fil	Forventer at finde filen, hvis den er tilgængelig på netværket (begge tilfælde er testet).	$\surd$
33	Søgning efter en specifik revision af en nøglering	Forventer at finde nøgleringen, hvis tilgængelig på netværket (begge tilfælde er testet).	$\surd$
34	Overførsel af en fil	Forventer at filen bliver overført i hel tilstand.	$\surd$
35	Overførsel af en nøglering	Forventer at nøgleringen bliver overført i hel tilstand.	$\surd$
36	Overførsel af en revisionslog	Forventer at revisionsloggen bliver overført i hel tilstand.	$\surd$
37	Oprettelse af en tilladelse der giver integritetsret til en fil	Forventer at tilladelsen oprettes, gøres tilgængelig på netværket og kan findes af brugeren, og at tilladelsen kun giver integritetsret til den tiltænkte bruger.	$\surd$
38	Oprettelse af en tilladelse der giver integritets- og læseret til en fil	Forventer at tilladelsen oprettes, gøres tilgængelige på netværket og kan findes af brugeren, og at tilladelsen kun giver integritets- og læseret til den tiltænkte bruger.	$\surd$
39	Oprettelse af en tilladelse der giver integritets-,læse- og skriveret til en fil	Forventer at tilladelsen oprettes, gøres tilgængelige på netværket og kan findes af brugeren, og at tilladelsen giver integritets-, læse- og skriveret til den tiltænkte bruger.	$\surd$
40	Oprettelse af en tilladelse der giver integritetsret til en nøglering	Forventer at tilladelsen oprettes, gøres tilgængelige på netværket og kan findes af brugeren, og at tilladelsen kun giver integritetsret til den tiltænkte bruger.	$\surd$

41	Oprettelse af en tilladelse der giver integritets- og læseret til en nøglering	Forventer at tilladelsen oprettes, gøres tilgængelige på netværket og kan findes af brugeren, og at tilladelsen kun giver integritets- og læseret til den tiltænkte bruger.	$\surd$
42	Oprettelse af en tilladelse der giver integritets-,læse- og skriveret til en nøglering	Forventer at tilladelsen oprettes, gøres tilgængelige på netværket og kan findes af brugeren, og at tilladelsen kun giver integritets-, læse- og skriveret til den tiltænkte bruger.	$\surd$
43	Søgning efter en tilladelse til en nøglering	Forventer at finde tilladelsen, hvis den er tilgængelig på netværket (begge tilfælde er testet).	$\surd$
44	Søgning efter en tilladelse til en fil	Forventer at finde tilladelsen, hvis den er tilgængelig på netværket (begge tilfælde er testet).	$\surd$
45	Overførsel af en tilladelse til en nøglering	Forventer at tilladelsen bliver overført i hel tilstand og importeret på den valgte nøglering.	$\surd$
46	Overførsel af en tilladelse til en fil	Forventer at tilladelsen bliver overført i hel tilstand og importeret på den valgte nøglering.	$\surd$

Bilag B

Installationsvejledning

For at Latinum kan køre kræves det, at man har installeret en JCE provider fra Bouncy Castle og JXTA framework'et. Det vil i det følgende blive forklaret, hvor man kan hente disse, og hvordan man installerer dem.

Bouncy Castle kan downloades fra `http://www.bouncycastle.org/latest_releases.html`. Jar-filerne fra Bouncy Castle kopieres over i `$JAVA_HOME/jre/lib/ext`, hvor `$JAVA_HOME` er mappen hvor Java er installeret.

JXTA kan downloades fra `http://www.jxta.org/project/www/download.html`. Her findes også instruktioner i at installere JXTA.

Når Bouncy Castle og JXTA er installeret kan Latinum hentes fra `http://www.secureP2P.net`. Kørsel af Latinum sker ved at udføre kommandoen:

```
java -jar Latinum.jar
```

Når Latinum køres første gang oprettes der en konfigurationsfil, **latinum.conf**, som indeholder værdier for parametre, der har indflydelse på kørslen af Latinum. **latinum.conf** er en tekstfil som indeholde linjer på formen `<paramenter>=<værdi>` hvor `<værdi>` kan ændres efter ens behov.

Følgende parametre kan indstilles:

persKeyPath: Stien til ens personlig nøglesæt.

searchUpdateInterval: Intervallet hvormed at søgninger bliver gentaget.

linespeed: Hastigheden af ens netværksforbindelse.

tempDir: Stien til det katalog hvor filer midlertidigt opbevares mens de hentes.

encryptedFilesDir: Stien til det katalog hvor filer flyttes til når er overført eller tilføjet. Filer i dette katalog deles på netværket.

advUpdateInterval: Intervallet hvormed, at *encryptedFilesDir* scannes igennem for nye filer. Er der tilkommet nye filer publiceres disse.

Litteratur

[ABC⁺02] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002. Avail: `http://research.microsoft.com/~adya/pubs/osdi2002.pdf`.

[AL00] K. Aoki and H. Lipmaa. Fast implementations of aes candidates. In *The Third Advanced Encryption Standard Candidate Conference*, pages 106–120, April 2000. Avail: `http://citeseer.nj.nec.com/aoki00fast.html`.

[And96] R. Anderson. The eternity service. In *Proceedings of Pragocrypt '96*, 1996. Avail: `htpp://citeseer.nj.nec.com/anderson96eternity.html`.

[BB94] B. D. Boer and A. Bosselaers. Collisions for the compression function of MD-5. *Lecture Notes in Computer Science*, 765:293–??, 1994. Avail: `citeseer.nj.nec.com/denboer93collisions.html`.

[BGW01] N. Borisov, I. Goldberg, and D. Wagner. Intercepting mobile communications: The insecurity of 802.11. In *Proceedings of MOBICOM 2001*, 2001. Avail: `http://citeseer.nj.nec.com/borisov01intercepting.html`.

[BKK⁺03] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in p2p systems. *Communications of the ACM*, 46(2), February 2003. Avail: `http://portal.acm.org/citation.cfm?id=606299&coll=portal&dl=ACM`.

[BKR94] M. Bellare, J. Kilian, and P. Rogaway. The security of cipher block chaining. *Lecture Notes in Computer Science*, 839:341–358, 1994. Avail: `citeseer.nj.nec.com/bellare94security.html`.

[BME] B. Byer, E. Martin, and C. Edwards. Napster messages. Avail: `http://opennap.sourceforge.net/napster.txt`.

[Cas] Bouncy Castle. Bouncy castle 1.20 api specification. Avail: `http://www.bouncycastle.org/docs/docs1.4/`.

[CHM+02] I. Clarke, T. W. Hong, S. G. Miller, O. S., and B. Wiley. Protecting free expression online with freenet. *IEEE Internet Computing*, 6(1):40–49, 2002. Avail: http://citeseer.nj.nec.com/article/clarke02protecting.html.

[Cop85] D. Coppersmith. Another birthday attack. *Advances in Cryptology - CRYPTO '85: Proceedings*, 1985. Avail: http://www.springerlink.com/app/home/contribution.asp?wasp=b5uc1gxhlj4q%qy13xxb0&referrer=parent&backto=issue,2,44;journal,1358,1361;linkingpublicatio%nresults,id:105633,1.

[CSWH01] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001. Avail: http://citeseer.nj.nec.com/clarke00freenet.html.

[Dai] W. Dai. Crypto++ 5.1 benchmarks. Avail: http://www.eskimo.com/~weidai/benchmarks.html.

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976. Avail: citeseer.nj.nec.com/diffie76new.html.

[DKK+01] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001. Avail: http://citeseer.nj.nec.com/dabek01widearea.html.

[Dob96] H. Dobbertin. The status of MD-5 after a recent attack. *CryptoBytes*, 2, 1996. Avail: ftp://ftp.rsasecurity.com/pub/cryptobytes/crypto2n2.pdf.

[FKL+00] N. Ferguson, J. Kelsey, S. Lucks, B. Schneier, M. Stay, D. Wagner, and D. Whiting. Improved cryptanalysis of Rijndael. In *Seventh Fast Software Encryption Workshop*, 2000. Avail: citeseer.nj.nec.com/ferguson00improved.html.

[Fle] FlexiProvider. Avail: http://www.flexiprovider.de.

[Fol] Foldoc. Extended backus-naur form. Avail: http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?Extended+BNF.

[FS03] N. Ferguson and B. Schneier. *Practical Cryptography*. John Wiley & sons, 2003. ISBN: 0471223573.

[gF] giTF FastTrack. Avail: http://developer.berlios.de/projects/gift-fasttrack.

[gIFT] giTF: Internet File Transfer. Avail: http://gift.sourceforge.net.

[Har] T. Hargreaves. The fasttrack protocol. Avail: http://developer.berlios.de/projects/gift-fasttrack.

[HJ03] A. Harrington and C. D. Jensen. Cryptographic access control in a distributed file system. In *Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 158–165, 2003. Avail: `http://doi.acm.org/10.1145/775412.775432`.

[IK02] T. Iwata and K. Kurosawa. Omac: One-key cbc mac. *Cryptology ePrint Archive, Report 2002.*, 2002. Avail: `citeseer.nj.nec.com/iwata02omac.html`.

[KIT02] K. Kant, R. Iyer, and V. Tewari. A framework for classifying peer-to-peer technologies. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002. Avail: `http://csdl.computer.org/dl/proceedings/ccgrid/2002/1582/00/15820368.pdf`.

[Knu03] Lars R. Knudsen. Jeg har knækket en kode. *Mat Matilde*, pages 18–20, 2003.

[LR98] P. J. Leach and R.Salz. Uuids and guids. Technical report, IETF, 1998. Avail: `http://www.ics.uci.edu/~ejw/authoring/uuid-guid/draft-leach-uuids-guids-01.txt`.

[Mar02] E. P. Markatos. Tracing a large-scale peer to peer system: an hour in the life of gnutella. *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and The Grid*, 2002. Avail: `http://citeseer.nj.nec.com/markatos01tracing.html`.

[Ora01] A. Oram, editor. *Peer-to-Peer: Harness the Power of Disruptive Technologies.* O'Reilly & Associates, 2001. ISBN: 059600110X.

[Pre99] B. Preneel. The state of cryptographic hash functions. *Lecture Notes in Computer Science*, 1561, 1999. Avail: `http://www.springerlink.com/app/home/contribution.asp?wasp=16unyjmrmq6hd7j5nre7&referrer=parent&backto=issue,8,10;journal,1080,1268;linkingpublicationresults,id:105633,1`.

[Pro] The Free Network Project. Avail: `http://freenet.sourceforge.net`.

[QD90] J. J. Quisquater and J. P. Delescaille. How easy is collision search? application to des. In *Proceedings Eurocrypt'89*, pages 429–434, 1990. Avail: `http://www.springerlink.com/app/home/contribution.asp?wasp=49te8pxhyp4r%ufxhlbcl&referrer=parent&backto=issue,43,72;journal,1256,1268;linkingpublicati%onresults,id:105633,1`.

[RD01] A. I. T. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Symposium on Operating Systems Principles*, pages 188–201, 2001. Avail: `http://citeseer.nj.nec.com/rowstron01storage.html`.

[Reu03] Reuters. Kazaa nears download record, 2003. Avail: http://news.com.com/2100-1027_3-1009418.html?tag=fd_top&pid=12.

[RFI02] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), 2002. Avail: citeseer.nj.nec.com/ripeanu02mapping.html.

[Rip01] M. Ripeanu. Peer-to-peer architechture case study: Gnutella network. Technical report, University of Chicago, 2001. Avail: http://www.cs.uchicago.edu/research/publications/techreports/TR-2001-26.

[RWE+01] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance-free global data storage. *IEEE Internet Computing*, 5(5):40–49, 2001. Avail: http://citeseer.nj.nec.com/rhea01maintenancefree.html.

[Sch96] B. Schneier, editor. *Applied Cryptography, Second Edition*. John Wiley & sons, 1996. ISBN: 0471117099.

[Ses98] P. Sestoft. Systematic software test. Technical report, Royal Veterinary And Agricultural University, 1998. Avail: http://www.imm.dtu.dk/courses/02100/Notes/sestofttest.pdf.

[Sou] Microsoft Visual SourceSafe. Avail: http://msdn.microsoft.com/ssafe.

[Sta03] W. Stallings. *Cryptography and Network Security - Principles and Practices, Third Edition*. Prentice Hall, 2003. ISBN: 0131115022.

[Sun] Sun. Java cryptography extension (jce): Reference guide. Avail: http://java.sun.com/j2se/1.4.1/docs/guide/security/jce/JCERefGuide.html%.

[Sys] CVS Concurrent Versions System. Avail: http://www.cvshome.org.

[TAA+03] B. Traversat, A. Arora, M. Abdelaziz, M. Duigou, C. Haywood, J. Hugly, E. Pouyoul, and B. Yeager. Projext jxta 2.0 superpeer virtual network. Technical report, Sun microsystems inc., 2003. Avail: http://www.jxta.org/project/www/docs/JXTA2.0protocols1.pdf.

[TW02] W. Trappe and L. C. Washington, editors. *Introduction to Cryptography with Coding Theory*. Prentice Hall, 2002. ISBN: 0130618144.

[Use] Recommended Elliptic Curves For Federal Government Use. Avail: http://csrc.nist.gov/CryptoToolkit/dss/ecdsa/NISTReCur.pdf.

[Yuv79] G. Yuval. How to swindle rabin. *Cryptologia*, 3, 1979.