Diarienummer: 2005:197

KTH Teknik och hälsa

Övervakning av trafikkvalitet för SIP-baserad kommunikation

End-to-End Performance Monitoring of SIP-Based Communication

Examensarbete 20 poäng
Magisterutbildning Datornätverk med inriktning mot nätverksdesign

Författare: Emma Roos

Uppdragsgivare: KTH Syd

Examinator: Thomas Lindh, KTH Syd

Datum: 2006-05-13

KTH Syd Campus Haninge

Sammanfattning

Rapporten redovisar ett examensarbete som handlar om att hämta information från signaleringen vid ett IP-telefonisamtal och använda denna information för att få statistik om samtal samt konfigurera mätningar av dessa IP-telefonisamtals datatrafik. Trafiken skulle mätas med en s.k. flödesmätare, i detta fall NeTraMet.

I arbetet gjordes först en mindre laboration där, efter att en IP-telefoniserver satts upp, kördes olika IP-telefonisamtalstrafikfall och trafiken från dessa undersöktes i en s.k. sniffer och protokollanalysator. Vid denna laboration kom det fram vilken information som skulle kunna användas av den prototyp som skulle utvecklas för att konfigurera och starta mätningarna av samtalens datatrafik. I SIP-meddelandena som skickas vid ett samtal finns information om vilka parter som är involverade i samtalet samt information om den media som används och på vilken port median skickas. Genom att titta på de statusmeddelanden som skickas går det också att få information om ett samtal inte kan kopplas upp etc. och varför.

Flödesmätaren NeTraMet testades samt att olika metoder att generera monitoreringspaket också testades. Beslutet var att använda programmet *ngen* mot echo-porten för att generera monitoreringspaket.

En prototyp i Perl togs fram för att hämta information från SIP-meddelandena och konfigurera och starta mätningarna. Denna prototyp fanns i två versioner: en klientversion där IP-telefoniklienten som startade samtalet administrerade mätningen och en serverversion där SIP-proxyn administrerade mätningen. När serverversionen ska användas måste SIP-proxyn vara satt som ”record-route” för att få alla SIP-meddelanden att gå genom proxyn.

För att testa prototypen gjordes mätningar av IP-telefonisamtal mellan två datorer, den ena i Haninge och den andra i Karlskrona. I mätningarna togs information om förseningar, processeringstiden i mottagarnoden, förseningsvariationer och genomströmning fram för samtalsdatatrafiken. När det gäller processeringstiden hade denna en mycket stor påverkan på fördröjningen av paketen, detta troligtvis pga. att porten som monitoreringspaketen skickades till är nedprioriterad i datorsystemet. När det gäller genomströmningen låg denna på en jämn nivå, detta eftersom RTP-trafiken skickades med en jämn hastighet.

Abstract

This Masters Thesis is about how to gather information from IP telephony signaling and use this information to get statistics about calls and to configure measurements of these IP telephony calls data traffic. To measure the traffic a traffic flow meter called NeTraMet was used.

A laboratory test bed has been configured in order to development a measurement prototype. An IP telephony server was set up, different IP telephony traffic scenarios was run and the traffic from these was investigated with the help of a so called sniffer and protocol analyzer. The necessary information needed to configure and start the measurements of the calls data traffic was studied and decided.. In the SIP messages that were sent, there was information about what parties was involved in the call plus information about the media that was used and on what port the media was sent. By looking at the status messages that are sent, it's also possible to get information if a call can't be connected etc. and why.

The flow meter NeTraMet and different methods to generate the monitoring packets were tested. The monitoring packets were implemented by using the program *ngen* to generate packets from one client to the echo port of the other client.

A prototype was implemented in Perl to gather information from the SIP messages, configure and start the measurements. This prototype exists in two versions: one client version where the IP telephony client that started the call administrated the measurements and a server version where the SIP proxy administrates it. When the server version is used the SIP proxy must be set as record-route to make all the SIP messages pass through the proxy.

To test the prototype some measurements of IP telephony calls was done between two computers, one in Haninge and the other in Karlskrona. In the measurements information about delays, processing times in the destination node, delay variations and throughput was computed. Then it comes to the processing time, this time has a big impact on the delay of the packets, this probably because that the port that the monitoring packets was sent towards have a lower priority in the computer system. The throughput did not vary a lot, since the RTP traffic is sent at a constant speed.

Innehållsförteckning

1 Inledning

Målsättningen för examensarbetet denna rapport handlar om var att utforma en lösning för kvalitetsövervakning av SIP-baserad kommunikation med hjälp av en flödesmätare.

Utgångspunkten för arbetet är att en operatör driver en SIP-server som hanterar tal- och multimediatrafik för sina kunder. Operatören behöver ett system för övervakning av trafikkvaliteten för de tjänster där detta är relevant.

Tanken är att använda informationen från signaleringsmeddelanden (som är av protokolltypen SIP) och med hjälp av denna information kunna konfigurera en s.k. flödesmätare för att mäta kvalitetsparametrar såsom fördröjningar, fördröjningsvariationer, genomströmning och förluster. Från denna signaleringinformation kan det eventuellt också att gå att mäta antalet uppkopplingar som misslyckades eller blev avbrutna etc.

Som resultat av examensarbetet togs en prototyp fram som tar information från SIP-meddelandena och använder denna information till att göra kvalitetsmätningar med flödesmätaren NeTraMet. Denna prototyp testades i en mätning mellan två datorer som var involverade i ett IP-telefonisamtal.

Examensarbetet bestod av följande delar:

- Konfigurering av en SIP-server (som kör programvaran SIP Express Router) samt ett antal klienter.
- Utföra en laboration där samtal kopplas upp mellan två klienter via SIP-servern och signalerings- och datatrafiken spelas in och analyseras.
- Undersöka flödesmätaren NeTraMet och se hur denna fungerar samt hur den konfigureras.
- Utveckla en prototyp som tar information från signaleringsmeddelandena som skickas vid uppkopplingen och termineringen av ett IP-telefonisamtal och med hjälp av denna startar upp en kvalitetsmätning av samtalet med NeTraMet.
- Testa den utvecklade prototypen genom att med hjälp av den mäta IP-telefonisamtal.

2 Bakgrund

2.1 IP-telefoni

De senaste åren har det blivit vanligare att använda sig av de nu existerande snabba datornätverken till att ringa telefonsamtal mellan olika användare, s.k. IP-telefoni, i stället för att använda det vanliga telefonnätverket. Några av anledningarna varför detta görs är att med IP-telefoni går det att integrera tal-, video- och datatrafik på ett enkelt sätt samt att det är lättare att skapa nya funktioner. Det ses även som billigare än vanlig konventionell telefoni speciellt p.g.a. att i stället för att ha två nätverk (ett för data och ett annat för telefoni) behövs bara ett. En annan anledning är att det går att sitta var som helst i världen där det finns en Internet-uppkoppling och vara åtkomlig via sin vanliga IP-telefoniadress.

När det gäller standarderna för IP-telefoni finns två olika inriktningar, den ena inriktningen är kopplad till IETF medan den andra är kopplad till ITU-T. Detta arbete använder sig av IETF:s IP-telefonistandard, signaleringsprotokollet SIP. ITU-T:s signaleringsprotokoll heter H.232.

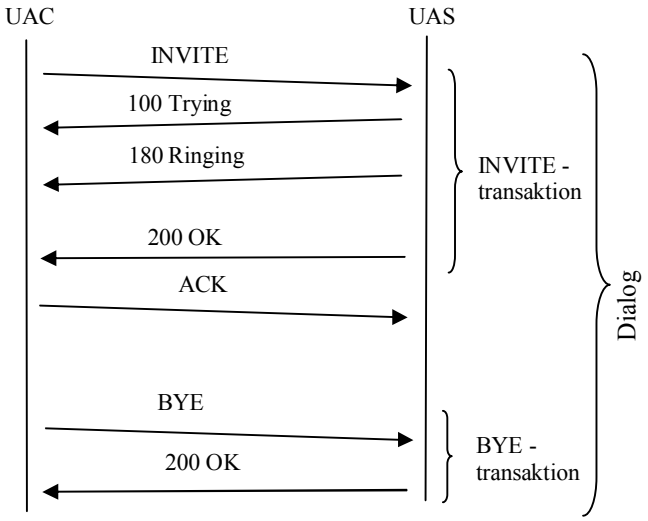
2.1.1 SIP - Session Initiation Protocol

SIP står för Session Initiation Protocol och är ett protokoll på applikationsnivå för signalering mellan en klient och en server. Protokollet utvecklades av IETF (Internet Engineering Task Force). SIP är i huvudsak till för att koppla upp och terminera multimediasessioner (t.ex. ett IP-telefonisamtal) mellan två ändpunkter över Internet samt informera om ändringar i en session. SIP protokollet inkorporerar element från Internet protokollen HTTP (Hyper Text Transport Protocol) och SMTP (Simple Mail Protocol). Från HTTP har SIP lånat en klient-serverdesign och användandet av URL (Uniform Resource Locator). Från SMTP har SIP lånat sättet att koda texten samt stilen på hur meddelandehuvudet ser ut. [1]

SIP bygger på klient-serverprincipen. En SIP-klient kallas för en User Agent (UA) och denna består både av en klientdel (UAC) och en serverdel (UAS). En ”User Agent” ska kunna ta instruktioner från en användare (vanligtvis en människa men även ett annat protokoll kan vara en användare) och sedan kunna koppla upp och terminera sessioner med en annan UA. En UA ska antingen kunna använda UDP som transportprotokoll eller TCP men helst ska den kunna klara av båda. UACn är till för att initiera förfrågningar (som t.ex. INVITE som skickas när ett samtal ska kopplas upp) medan UASen genererar svar på förfrågningar (t.ex. 180 Ringing som berättar att INVITE kommit in och att det ”ringer” hos mottagaren). [2]

En användares adress som används för att kunna få tag på denne består av ett URL. Detta URL liknar det ”mailto:”-URL som används i HTTP när det refererar till en e-mailadress. Ett SIP-URL ser vanligtvis ut som sip:användare@SIP-domän, t.ex. sip:emma@haninge.kth.se. Det kan finnas extra information i URLen såsom information om transportprotokoll, metod etc. samt taggar som är slumpmässiga nummer som används t.ex. om en användare är registrerad på flera olika ställen. Extra information i URLen brukar vanligtvis läggas till av programvaran i t.ex. proxyservrar under en mediasession.

I SIP finns något som kallas transaktioner. En transaktion består av en förfrågan samt de svarskoder etc. som följer. Figur 2.1 nedan visar hur SIP-meddelandeutbytet kan se ut vid en mediasession, i denna figur är också de olika transaktionerna markerade. Flera exempel på olika SIP-scenarion beskrivs i kapitel 3. Alla transaktionerna i SIP-trafiken som körs över en hel mediasession (t.ex. från första INVITE till “200 OK”-meddelandet som skickas som svar på BYE-meddelandet) kallas för en dialog. [3]



Figur 2.1: Exempel på SIP-meddelanden vid uppkoppling och terminering av en mediasession

När det gäller de olika förfrågningarna och deras svar består dessa meddelanden alltid av ett SIP-huvud, i detta huvud finns information om: vad för metod eller felkod meddelandet är, sändarens (To) och mottagarens (From) URL, en identifikationskod för vilken mediasession (Call-ID) och kommandosekvens (transaktion) (CSeq) SIP-meddelandet tillhör, vilken väg meddelandet gått (Via) samt information om det finns något innehåll i SIP-meddelandet och i så fall vilket protokoll detta är skrivet i och hur stort innehållet är.

Om det finns mera information än det i SIP-huvudet är denna information vanligtvis skriven i protokollet SDP (Session Description Protocol)[4]. Vanligtvis handlar denna SDP-information om mediaströmmens inställningar såsom mediatyp, mediaprotokoll, porten mediaströmmen använder, kodek m.m. De SIP-meddelanden som vanligtvis består av SDP-information är INVITE-meddelanden (om det inte finns information här vid starten av mediasessionen så finns den i ACK-meddelandet i stället) och “200 OK”-meddelanden.

2.1.1.1 SIP-servrar

I SIP finns olika sorters servrar, dessa är i huvudsak proxyservrar, omdirigeringsservrar och registrationsservrar. Dessa servrar accepterar SIP förfrågningar och svarar på dessa. Servertyperna är en sorts logiska grupperingar av funktioner och flera servertyper kan finnas i en SIP-server, t.ex. kan en server vara både en proxyserver och en registrationsserver.

En proxyserver är till för att ta emot SIP-förfrågningar och sedan vidarebefordra dessa till en mottagande UA eller annan proxyserver som finns på vägen till den mottagande UAn samt svara på förfrågningar. En proxyserver initierar aldrig en förfrågan. Den hanterar ej heller mediaströmmar. För att veta var en förfrågan ska skickas brukar proxyn ha en koppling med någon sorts lokaliseringsdatabas eller liknande, kopplingen mellan dessa är ej definierat i SIP.

En proxyserver kan antigen vara ”stateless” eller ”stateful”. En ”stateless” proxyserver hanterar inkommande förfrågningar bara genom att titta på förfrågans innehåll, när denna har hanterats tas all information bort om det meddelandet. En ”stateful” proxyserver sparar information om tidigare förfrågningar och deras svar och använder sedan denna information för att hantera framtida inkommande förfrågningar. Denna servertyp använder olika timers etc. för att se att t.ex. om en förfrågan vidarebefordrats, att det kommer ett svar på denna och om detta svar inte kommer inom en viss tid skickas förfrågan om, en ”stateless” server har inga timers. En ”stateful” proxyserver kan också

hantera autenticiering mellan ”User Agents”.

En annan serversort är omdirigeringsservrar. Denna sorts server svarar på SIP förfrågningar men vidarebefordrar inte dessa. I stället skickar servern t.ex. vid förfrågan om uppkoppling av samtal ett vidarebefodransmeddelande tillbaka som har information om var klienten ska skicka sina meddelanden för att koppla upp ett samtal med den mottagare uppringaren vill ha kontakt med.

Den tredje typen av SIP-server är en registrationsserver. Denna server tar emot s.k. registreringar från de olika SIP-klienterna i en adminstrativ domän. Denna information om registreringsservern får om klienterna är sedan åtkomliga för andra SIP-servrar (t.ex. en proxyserver) inom denna domän. Ofta måste klienten kunna autentifiera sig med registreringsservern för att bli registrerad.

SIP-servern som användes i projektet var en blandning av de tre olika typerna ovan (i huvudsak var det en proxy- och registreringsserver). Denna SIP-server använde sig av programvaran SIP Express Router och beskrivs i kapitel 2.1.3.

2.1.1.2 Metoder och svarskoder i SIP

SIP har ett antal metoder som används i signaleringen, de vanligaste av dessa är: INVITE, REGISTER, BYE, ACK, CANCEL och OPTIONS. Dessa metoder är till för att låta en UA fråga en annan UA eller en server om att utföra en specifik funktion. Det finns även andra metoder men dessa är inte lika vanliga som de ovan. En proxyserver behöver inte veta vad en metod gör utan den bara vidarebefordrar detta till den mottagande UAn.

När det gäller metoden INVITE har denna två huvudsakliga uppdrag. Det första är att starta uppkopplingsförfarandet av en mediasession och det andra är att ändra existerande mediasessions-inställningar (kallas i bland re-INVITE).

Metoden REGISTER är till för att informera t.ex. en registreringsserver om en klients IP-adress och URL. En klient behöver vara registrerad om den ska kunna ta emot inkommande samtal som går via en proxyserver.

BYE-metoden används när en mediasession ska kopplas ner. Detta sorts meddelande skickas bara av klienter som är involverade i en mediasession.

ACK-metoden används för att bekräfta de slutgiltiga svaren när en mediasession kopplas upp.

Metoden CANCEL används för att avbryta ett pågående försök att koppla upp en mediasession. Ett CANCEL-meddelande kan skickas av både en klient och en proxyserver.

Metoden OPTIONS används för att skicka förfrågningar om vilka olika möjligheter och inställningar en klient eller server klarar av.

Det finns sex klasser av svarskoder som kan skickas av en klients UAS eller en SIP-server vid en förfrågan från en UAC. Koderna består av ett tresiffrigt nummer samt en beskrivande text. De fem första klasserna är lånade från HTTPs svarskoder medan den sjätte är speciellt gjord för SIP [1]. Dessa sex klasser är:

- 1xx – Informerande svarskoder, dessa koder indikerar att mottagaren har blivit kontaktad och håller på att hantera den inkomna förfrågan men att ett svar på denna inte är klar än. Exempel på meddelanden är ”100 Trying” och ”180 Ringing”.

- 2xx – En förfrågan har lyckats. Dessa koder berättar för sändaren av förfrågan att denna har lyckats. Den vanligaste koden är ”200 OK”.
- 3xx – Omdirigeringskoder. Dessa koder informerar om att mottagaren finns någon annanstans samt informerar var den finns. Exempel på koder är ”301 Moved Permanently”, ”302 Moved Temporarily” och ”380 Alternative Service”.
- 4xx – Klientfel. Den mottagande klienten har inte lyckats hantera förfrågan och därför ska inte meddelandet skickas om, om inte t.ex. ändringar i meddelandet görs. Exempel på koder är ”401 Unauthorized”, “404 Not Found”, “408 Request Timeout”, “486 Busy Here” och “487 Request Terminated”.
- 5xx – Serverfel. Dessa koder informerar om att det har blivit något fel i servern. Exempel på koder är: ”500 Server Internal Error”, ”503 Service Unavailable” och ”513 Message Too Large”.
- 6xx – Globala fel. Dessa koder indikerar att en server (SIP-server eller UAS) har definitiv information om en specifik användare inte bara för den specifika förfrågan. Exempel på kod som är vanlig är ”603 Decline” [2].

2.1.2 RTP - Real-time Transport Protocol

RTP [5] står för Real-time Transport Protocol och är ett protokoll för transport av realtidstrafik över ett nätverk. RTP är precis som SIP och SDP utvecklat av IETF. RTP-paketen klarar av flera olika sorters media såsom ljud, video och data. RTP kör vanligtvis med UDP som transportprotokoll och det finns inget QoS (Quality of Service) inbyggt i protokollet utan detta ska i så fall hanteras av underliggande protokoll eller i ändpunkterna. Med hjälp av den information om datatyp, sekvensnumrering och tidsstämpling som finns i RTP [5] kan ändpunkterna upptäcka och hantera problem såsom tappade paket, variationer i förseningar i nätverket, paket som kommer i fel ordning och asymmetrisk routing [1].

RTP har också ett systerprotokoll som heter RTCP (RTP Control Protocol) detta protokoll (som vanligtvis använder RTP-strömmens portnummer plus ett vid en mediasession) används för att skicka kvalitetsrapporter mellan parterna i en mediasession. Denna rapport kommer inte att gå närmare in på detta protokoll utan kommer att undersöka en annan lösning på att få den information som skickas med RTCP.

När en mediasession etableras med SIP definieras transportprotokoll (UDP), kodekar och RTP-porten i den SDP-data som finns i de relevanta SIP-meddelandena. När det gäller kodekar etc. definieras dessa i en s.k. RTP Audio Video Profiler (RTP/AVP). I profilen definieras UDP som transportprotokoll, RTPs portnummer (med RTCP som nästa portnummer (udda siffra)), datatypen (kodek) m.m. När det gäller de olika datatyperna i RTP-paketen finns dessa fördefinierade och dessa identifieras med nummer. Datatyperna finns registrerade med IANA (Internet Assigned Number Association) och nya datatyper måste registreras med dem [1]. Exempel på några kodekar som kan användas är ITU G.711 PCM μ-law och A-law, GSM-kodning, MPEG video etc. [5][1]

2.1.3 SER – SIP Express Router

SER (SIP Express Router)[6] är en SIP-serverprogramvara som utvecklats av iptel.org som är en ”Voice over IP”-portal som finns för att lansera VoIP-teknologier. Denna portal initierades av ett tyskt forskningsinstitut som heter Fraunhofer Fokus [7].

En SER-server klarar av alla tre servertyperna som nämndes i kapitel 2.1.1 (proxy, registrering, omdirigering). SER har i grunden ganska enkla funktioner, dessa kan sedan byggas ut genom att ladda in olika moduler. En SER-server är mycket konfigurerbar och konfigurationen görs med hjälp av ett C-

liknande skriptsspråk (ett exempel på hur detta kan ser ut kan ses i bilaga B där konfigurationsfilen som användes i examensarbetet visas).

Bland de moduler som kan kopplas till en SER-server finns moduler för de olika SIP-servertyperna, användarhantering (bl.a. med en användardatabas i SQL), "record routing", om proxyservern ska vara "stateful" m.m. Förutom de mest logiska funktionerna finns också moduler för funktioner såsom applikationsserverinterface, support för ”presence”, SMS-gateway, RADIUS/syslog-taxering och auktorisering m.m. [6]

Det finns till SER också ett webbinterface för att bl.a. administrera användare, detta heter *serweb*. Ett annat verktyg till SER är *serctl.* Detta verktyg är ett hjälpmedel för att kunna göra de mest grundläggande administrativa uppgifterna i en SER-server såsom lägga till användare, ändra deras lösenord, titta på serverns status etc. Det finns också ett röstbrevlådesystem till SER. [8].

2.2 Trafikmätningar

Ju vanligare det blivit att datornätverk används, speciellt när det gäller realtidstrafik, desto mer behövs möjligheter att ta reda på hur ett nätverk mår t.ex. för att se om ett nätverk behövs uppgraderas. Det har bl.a. blivit mycket vanligt med kvalitetskrävande trafik, såsom IP-telefoni, videoströmmar och annan realtidstrafik på datornätverken, framför allt via Internet men även t.ex. inom ett företag. För att kvaliteten ska upprätthållas har det blivit vanligt att en operatör och en kund upprättar ett s.k. Service Level Agreement (SLA) som specificerar vilken kvalitet som kunden kan förvänta sig för det fördefinierade datornätverksutnyttjande kunden sagt att de ska ha. För att kunna upprätthålla SLAn måste bl.a. operatören kunna veta hur nätverket mår och om detta kan hantera den trafik som körs. För att kunna göra detta finns ett antal metoder för att kunna övervaka nätverket och mäta trafiken och dess kvalité. Detta kapitel beskriver kortfattat de metoder som finns samt att de två huvudsakliga verktyg som användes i projektet (Ethereal och NeTraMet). Detta arbete hanterar i huvudsak mätningar av kvalitén i trafiken i ändpunkterna.

2.2.1 Kvalitetsparametrar

Vanligtvis när det pratas om hur bra ett nätverk är handlar det om hastigheten på nätverket men det finns ett flertal andra kvalitetsparametrar som också är viktiga när det gäller ett nätverks kvalité, speciellt då det gäller realtidstrafik. De viktigaste av dessa parametrar är:

- Fördröjningar
- Fördröjningsvariationer (jitter)
- Paketförluster och
- Genomströmning

Den första viktiga parametern när det gäller kvalitén för realtidsapplikationer i nätverket är *fördröjningar* mellan ändpunkter (både envägs- och tur-och-returfördröjningar). Detta är tiden det tar för paketen att komma fram till sin destination (vid envägsfördröjningar) och vid tur-och-returfördröjningar är det tiden det tar för paketet att gå från källa till destination och för svaret på paketet att skickas tillbaka. Förseningarna ska helst vara så låga som möjligt. Fördröjningar skapas bl.a. av tiden det tar för trafiken att processas i nätverksutrustningen på vägen samt vilka kodnings- och kompressionsalgoritmer som används för data. Detta problem kan bl.a. lösas med hjälp av olika prioritet i nätverket samt att skicka mindre paket.

När det gäller envägsfördröjningar måste klockorna i mätpunkterna vara synkroniserade. De två vanligaste synkroniseringssätten är NTP (Network Time Protocol)[9] och GPS (Global Positioning

System)[10]. I GPS hämtas tiden från klockan på de GPS satelliter som finns. Klockorna i dessa satelliter är atomur. När det gäller NTP är detta ett protokoll som används av datorer för att ställa datorns klocka. NTP-servrarna finns i olika stratum där stratum 0 är en server som hämtar sin tid direkt från ett atomur eller via GPS. NTP-servrar hämtar vanligtvis tiden från en annan NTP-server och ju fler servrar det är mellan stratum 0 servern och servern som används, desto mindre exakt blir tiden. NTP kan också påverkas av ändringar av fördröjningen i nätverket. Detta betyder att NTP är sämre än GPS för synkroniseringen av klockorna som används vid mätning av fördröjningar. Tidsstämplingen är viktig vid envägsfördröjningar och klockorna ska helst vara så bra synkroniserade som möjligt. NTP räcker vanligtvis inte till för detta. Vid tur-och-retur-fördröjning är synkroniserade klockor inte lika viktigt. [11]

Den andra viktiga parametern är *jitter* eller *fördröjningsvariation*. Detta är variationen på paketens fördröjning. Med mycket jitter kommer trafiken in i en ojämn ström som kan försämra kvalitén på det som tas emot. För att lösa problemet med jitter är det vanligt att öka storleken på jitterbuffern som realtidsapplikationen eventuellt använder.

Den tredje viktiga parametern är *paketförluster*, vilket är hur många av ett visst antal sända paket som inte kommit fram till mottagaren. Detta inträffar när paketbuffrar i nätverkets utrustning flödar över eller på grund av bitfel som leder till att paketen kastas. Att tappa ett fåtal paket ställer vanligtvis inte till några problem men börjar paketförlusterna att gå över 1 % kommer kvalitén på tal- eller videoströmmen att degradera.

Den sista viktiga parametern är *genomströmning*. Denna parameter visar hur mycket data kom går genom nätverket under en viss tid.

2.2.2 Passiva och aktiva metoder och kombinering av dessa

Det finns två typer av mätningsmetoder: passiva och aktiva. De passiva metoderna består vanligtvis av att trafikinformation samlas in av olika nätverkskomponenter, användare och servrar. De aktiva metoderna består vanligtvis av att trafik genereras och mäts.

Med passiva metoder mäts befintlig trafik vilket gör att det går att vara säker att mätningen avspeglar verkligheten. Passiv mätning utförs vanligtvis av övervakningsapplikationer som finns på själva nätverkselementen i nätverket eller så mäts trafiken när den kommer in till ett visst interface på en dator. Ett problem med passiva metoder är att vid hög trafikmängd blir det mycket data som måste sparas. Vilket kan ge utrymmesbrist om stora delar av trafiken måste sparas samt att t.ex. nätverkskortet vid hög trafik inte kan hantera trafikmängden.

När det gäller aktiva metoder går det bl.a. att låta några datorer i nätverket generera trafik på ett förkonfigurerat sätt. Dessa datorer kan också samla in statistik på de olika mätparametrarna och skickar detta sedan till ett centralt testcenter som analyserar resultaten. Den trafik som genereras i mätsyfte ska helst vara av samma typ som den trafik som vanligtvis kör på nätverket. Detta eftersom olika trafiktyper kan i bland hanteras olika av nätverksutrustningen i nätverket (t.ex. kan ICMP-trafik ha lägre prioritet än vanlig trafik eller att realtidstrafik har högre prioritet).

En metod är att generera så mycket som möjligt under en kortare period för att få information om hur mycket tal- och videotrafik som nätverket kan hantera innan kvalitén når en ej acceptabel nivå. Detta är en bra metod om det ska kontrolleras om ett nätverk kan hantera realtidstrafik om t.ex. ett beslut ska tas om att börja med VoIP.

En annan metod är att generera en mindre trafikström som kör hela tiden och därför blir en del av den vanliga trafiken. Denna metod kan vara bra om det ska finnas en möjlighet att få t.ex. larm när nätverket blir överbelastat eller för att kunna visa att de Service Level Agreements det kommits överens om följs.

Både de passiva och aktiva metoderna har olika för- och nackdelar och därför kan det vara en idé att kombinera dessa metoder. När det gäller det aktiva metoderna är fördelarna att dessa är ganska lätta att implementera och metoderna är flexibla. En annan fördel är att mätutrustningen inte behöver ha lika hög kapacitet, såsom nätverkskort som han hantera mycket trafik och hårddiskutrymme för att kunna lagra data som kommer in, som passiva metoder behöver. Ett problem med aktiva mätningar är att det kan fås snäva resultat om monitoreringspaketen inte är konfigurerade rätt eller att någonting oförutsatt händer i nätverket. En annan nackdel är att om monitoreringspaketen skickas för ofta kan dessa påverka resultatet (detta är inte ett lika stort problem idag eftersom de flesta nätverken idag är mycket snabba). En annan sak som måste tänkas på vid aktiva mätningar är att olika sorters trafik kan hanteras olika i nätverket, t.ex. att UDP- och TCP-trafik har högre prioritet än ICMP-trafik, detta betyder att monitoreringspaketen behöver vara av samma typ som den trafiktyp som är av intresse i mätningarna.

Det finns också ett antal mätparametrar som inte passar att använda aktiva metoder för att mäta, framförallt då genomströmning men även paketförluster där mätningarna kan bli mindre precisa om det är mycket trafik i nätverket [11].

När det gäller passiva metoder passar dessa metoder bäst till att mäta t.ex. genomströmning och tappade paket. De aktiva metoderna är bäst vid beräkning av fördröjningar och fördröjningsvariationer. Med de aktiva metoderna går det också att beräkna hur många paket som tappas mellan två monitoreringspaket (ett monitoreringsblock), detta för att se hur lång tid det är mellan perioder då inga paket tappas och hur många paket som försvinner då det är någon sorts blockering i nätverket.

I en kombinering av aktiva och passiva metoder mäts vanligtvis genomströmning och till viss del paketförluster passivt medan fördröjningar och fördröjningsvariationer mäts aktivt, paketförluster kan också använda sig av monitoreringsblocken, t.ex. för att undersöka om det finns perioder då det inte finns paketförluster samt perioder då ett stort antal paket tappas.

2.2.3 Ethereal

Ethereal [12] är en s.k. protokollanalysator och ett passivt mätverktyg. Programmet fångar in de paket som skickas över nätverket och avkodar sedan dessa så att användaren av programmet kan analysera trafiken som går över nätverket. För att spela in trafiken används drivrutinen libpcap i Linux- och WinPcap i Windows-versionen av programmet. Det går även att ladda in filer med data från tidigare sessioner då trafik spelats in för att titta på paketen från den sessionen. Programmet klarar av att avkoda flera hundra olika protokollsorter.

2.2.4 NeTraMet med monitoreringspaket

NeTraMet [13] är en skallad flödesmätare som mäter trafikflöden i ett nätverk. Med NeTraMet definieras vilka olika flöden som ska mätas med så kallade regler. Dessa regler kan skrivas med ett språk som heter SRL (Simple Ruleset Language)[14] som sedan kompileras till det format som NeTraMets regler är i. Ett exempel på en SRL-fil finns i bilaga C. Ett flöde kan definieras med tre olika adressattribut: ”Adjacent” (t.ex. MAC-adress), ”Peer” (t.ex. IP-adress) och ”Transport” (t.ex. protokolltyp (TCP, UDP etc.) och portnummer när det gäller ett IP-nät). Dessa adressattribut kan definieras för både sändar- och destinationsnoderna i flödet.

Datastrukturen i NeTraMet är definierad med SNMP MIBar. Data som fås från de olika definierade flödena kan hämtas med hjälp av vilket SNMP-program som helst men vanligtvis används NeTraMets ”manager”-program NeMaC till detta.

NeMaC både administrerar och hämtar data från de noder som kör NeTraMet. Detta program startas upp från ett kommandofönster och de administrativa inställningarna sätts antingen som flaggor vid kommandoraden, i en konfigurationsfil samt via en regelfil som definierar flödena som ska mätas. Med NeMaC går det att t.ex. definiera flödena, hur ofta data från dessa flöden ska hämtas, namnet på filen där data som hämtas ska sparas, vilken SNMP-community som används m.m. NeMaC generar två olika typer filer: en flödesfil som lagrar data som hämtats in om de olika flödena och en loggfil.

Den versionen av NeTraMet som användes i projektet var en s.k. OAM-version (OAM står för Operations Administration and Maintenance). Denna version av NeTraMet kombinerar det passiva verktyget NeTraMet med den aktiva metoden att ta hjälp av monitoreringspaket. I reglerna definieras en grupp av flöden som har ett grupp-id, till denna grupp finns ett definierat flöde som består av monitoreringspaket (som genereras på något vis (i projektet användes verktyget *ngen* [15])). Flödet med monitoreringspaketen identifieras med ett attribut som kallas MeasurementData. När ett monitoreringspaket kommer in till NeTraMet-agenten genereras mätningsdatainformation för alla flödena i gruppen, dessa kan sedan hämtas med NeMaC. För varje monitoreringspaket som kommer in sparas en tidstämpel som kan användas för att kalkylera fördröjningar och fördröjningsvariationer, även tappade paket kan beräknas. Denna information kan inte fås fram i den vanliga versionen av NeTraMet. Annan information som också fås från NeTraMet är antalet paket och bytes som skickats ut eller kommit in och med hjälp av denna information kan genomströmning beräknas, detta är vad som vanligtvis sparas i NeTraMet.

3 SIP

För att få en översikt hur SIP fungerar samt få idéer vilken signaleringsinformation som kan användas vid kvalitetsövervakning av SIP-baserad trafik gjordes en laboration där samtal kopplades upp mellan två klienter (och andra trafikfall) via en SIP-server. Alla datorerna som användes (klienter och server) satt på samma lokala nätverk, detta nätverk var switchat. Trafiken mellan datorerna spelades in med hjälp av programmet Ethereal.

Det som skulle komma fram från denna laboration var vilken signaleringsinformation som kan vara nyttig att använda för att informera flödesmätaren, som ska användas senare i examensarbetet, var den ska mäta trafikkvalitén. Samt om viss signaleringsinformation i SIP-meddelandena kan visa hur kvalitén är. Det var i huvudsak tre frågor som skulle besvaras:

- Vilken signaleringsinformation kan utväxlas?
- Hur kan signaleringsinformationen tas tillvara och användas för kvalitetsövervakning av tal-video-datatrafiken och
- Vilka parametrar från signaleringen kan användas för att mäta kvalitet för kommunikationen mellan klienter och server?

3.1 Upplägg

SIP-servern som användes i laborationen körde serverprogrammet SIP Express Router (SER) [6] på operativsystemet Fedora Core 2. Ett antal SIP användare, som skulle kunna registrera sig och ringa upp samtal genom servern, skrevs in i serverns användardatabas. Under större delen av laborationen var SIP-servern konfigurerad med record-route vilket betydde att all SIP trafik gick genom servern (även de meddelanden som vanligtvis inte gör detta).

Klienterna som användes var PC datorer som hade både Windows XP Professional och Linux distributionen Fedora Core 2 som operativsystem. I början användes bara klienter som kördes i Windows men även några Linux klienter testades (eftersom prototypen sedan behövdes utvecklas i Linux pga. av att det inte fanns en bra Windowsversion av NeTraMet). De IP-telefoniklienter som användes var X-Lite, Windows Messenger 5.1, SJ Phone, Linphone [16] och KPhone [17]. De två första klienterna kördes bara i Windows, den tredje i både Windows och Linux och de två sista är Linux klienter (dessa användes också senare i projektet när prototypen utvecklades och testades).

Samtalen mellan klienterna spelades in med hjälp av paketsniffern Ethereal. För att få med RTP-trafiken kördes Ethereal på en av klientdatorerna som var involverad i de olika samtalen. Detta eftersom RTP-trafiken går direkt mellan de två klienterna som är involverade i samtalet och inte via SIP-servern, samt att nätverket som datorerna är kopplade till är switchat och därför syns inte RTP trafiken för datorer som inte är med i samtalet. För att titta på SIP-trafiken som gick genom SIP-servern kördes också Ethereal på själva SIP-servern.

Ett antal vanliga scenarion kördes för att se hur signaleringen såg ut vid dessa. De scenarion som undersöktes var:

- Registrering av en klient med SIP-servern
- Samtal mellan två parter
- Uppkoppling av ett samtal där uppringaren lade på innan den uppringde hann svara.
- Uppkoppling av ett samtal där den uppringde sa till sin klient att inte svara på samtalet.

- Uppkoppling av ett samtal där den uppringde inte gjorde något (t.ex. inte ser att SIP-klienten ringer).
- Uppringaren försöker ringa någon som inte är registrerad med SIP-servern eller inte finns i SIP-serverns användardatabas.
- Samtal där någon av parterna pausar samtalet (Liphone klarade inte av detta).

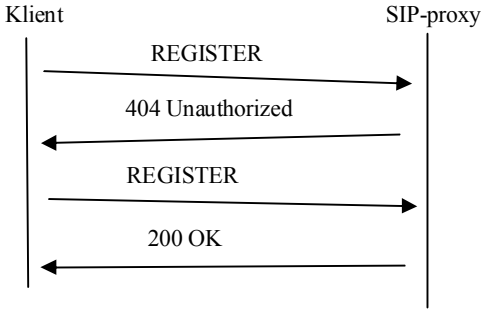
Det undersöktes också hur klienterna uppförde sig om en tredje part ringde upp en av de andra parterna under ett samtal. Vissa klienter hade också vissa inställningar för t.ex. om parten var upptagen och det undersöktes hur klienten informerade detta till uppringaren.

3.2 Resultat

3.2.1 Trafikfall

3.2.1.1 Registrering av klient

Innan en klient kan ringa till någon via proxyservern måste den registrera sig med proxyn. I figur 3.1 nedan visas vilka SIP-meddelanden som skickas vid registreringsscenariot.

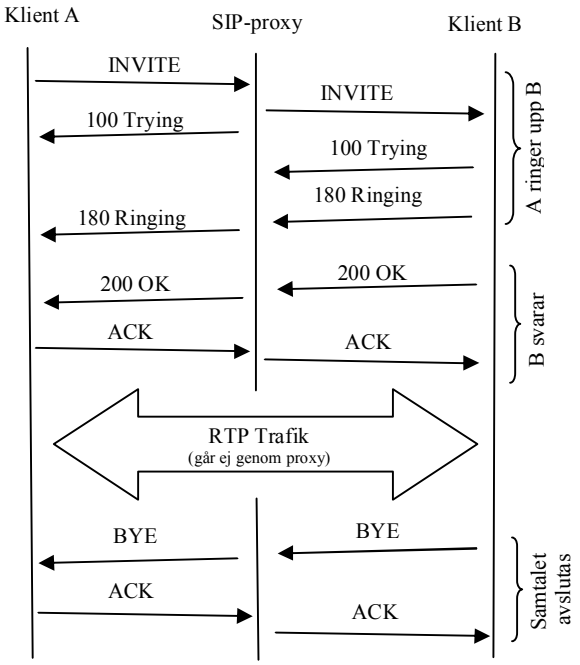


Figur 3.1: Registrering av klient

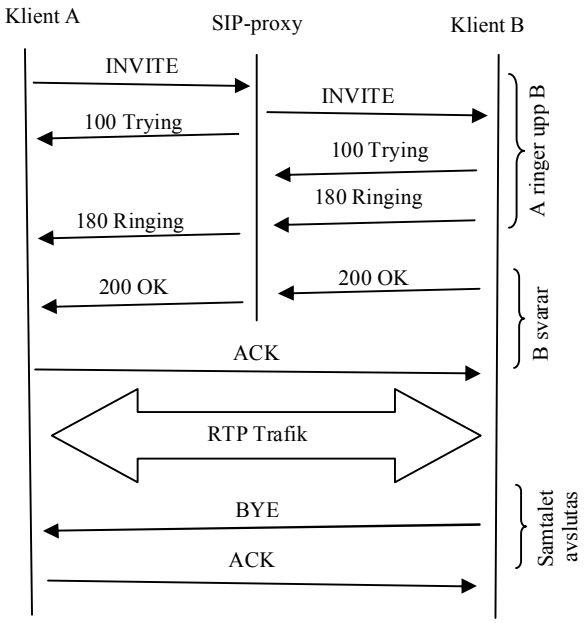
Klienten skickar ett REGISTER-meddelande till proxyn för att informera denna att klienten vill kunna koppla upp samtal via denna proxy. Eftersom proxyn i laborationen krävde att bara auktoriserade användare skulle kunna använda systemet skickades ett ”404 Unauthorized”-meddelande tillbaka. I detta meddelande finns en extra rad i SIP-huvudet för WWW- autentisering, detta meddelande består av en s.k. ”Digest realm” (SIP-nätverkets ID) och en s.k. ”nonce”. Klienten skickar ännu ett REGISTER-meddelande som svar till 404-meddelandet i detta finns en ”Authorization”-rad som har klientens svar på WWW- autentiseringen (användarnamn, realm, ”noncen”, SIP-serverns URI och ett svar på ”noncen” (skapas av en algoritm i klienten där svaret räknas ut med hjälp av noncen och användarens lösenord)). Om klientens användarnamn och lösenord är rätt blir klienten registrerad i proxyn och ett ”200 OK”-meddelande skickas tillbaka till klienten, om de inte är rätt skickas ett ”404 Unauthorized”-meddelande och klienten blir inte registrerad.

3.2.1.2 Samtal mellan två parter

I figur 3.2 och 3.3 nedan visas hur meddelandeutbytet vanligtvis ser ut vid ett samtal mellan två parter. Figur 3.2 visar hur det ser ut när proxyn har ”record route” satt medan figur 3.3 visar hur det ser ut när detta inte är satt.



Figur 3.2: Samtal mellan två parter med ”record route”.



Figur 3.3: Samtal mellan två parter utan ”record route”.

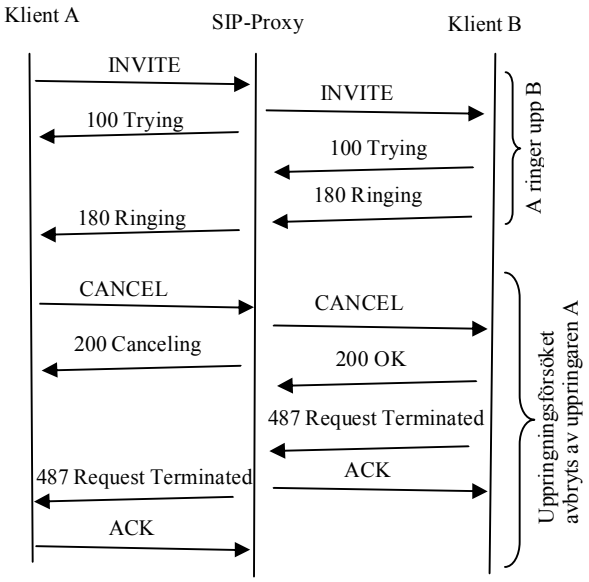
Som ses i figur 3.3 ovan går SIP-meddelandena direkt mellan part A och B från och med ACK-meddelandet som skickas när part B svarar på samtalet när ”record route” inte används.

Meddelandeutbytet kan skilja sig lite från det som visas i figurerna ovan beroende på vilken klient som används samt vilka inställningar dessa har. T.ex. har vissa klienter inställningar så att trafiken går som om ”record route” är satt även om detta inte är fallet. SJPhones grundinställning är att köra alla meddelanden genom proxyn, samma sak med Kphone. När det gällde den versionen av Kphone som användes under laborationen, uppförde denna klients signalering inte som det skulle om klienten var inställd att inte skicka alla sina signaleringsmeddelanden via servern.

När det gällde Liphone fanns det någon sorts bugg i programmet som gjorde att om A (uppringaren) är den som lägger på fungerar inte BYE-transaktionen som den skulle (BYE-meddelanden skickas om och om igen men B skickar inget ”200 OK”-meddelande). Linphone använder sig också av meddelandet ”101 Dialog Establishment” i INVITE-förfarandet samt att ett ”100 Trying”-meddelande skickas innan ”200 OK”-meddelandet när mottagaren av BYE-meddelandet mottagit detta meddelande.

3.2.1.3 Samtal som ej besvaras – uppringande part lägger på

I figur 3.4 nedan visas hur meddelandeutbytet ser ut vid ett uppkopplingsförsök av ett samtal där uppringaren bestämmer sig för att avbryta innan mottagaren hunnit svara.

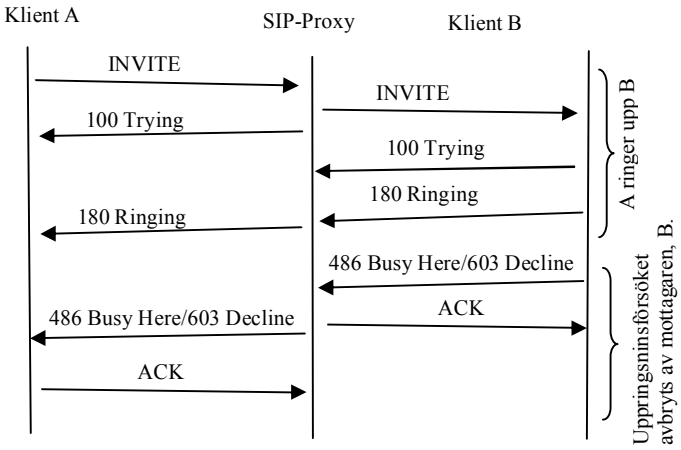


Figur 3.4: Uppkopplingsförsök av samtal där uppringaren avbryter försöket innan mottagaren svarar.

Uppringningsförfarandet är detsamma som beskrivits i kapitlet ovan fram till ”180 Ringing”-meddelandet. När uppringaren avbryter försöket skickas ett CANCEL-meddelande till proxyn. När CANCEL-meddelandet kommit in skickar denna ett 200-meddelande till uppringaren samt att den skickar vidare CANCEL-meddelandet till mottagaren. Mottagaren skickar sedan ett ”200 OK”-meddelande till proxyn samt när mottagarens klientprogram är klar med att avbryta uppringningsförsöket skickas också ett ”487 Request Terminated”-meddelande till proxyn. Proxyn skickar ett ACK på detta meddelande samt skickar vidare ”487 Request Terminated”-meddelandet till uppringaren som även den skickar ett ACK (i det här fallet till proxyn).

3.2.1.4 Samtal som ej besvaras – uppringd är upptagen eller avbryter (genom aktivt beslut)

Det kan hända att mottagaren är upptagen i ett annat samtal (och dess klient inte klarar av flera mottagare på en gång) eller att mottagaren inte vill svara och därför säger till sin klient att inte svara. I figur 3.5 nedan visas hur meddelandeutbytet kan se ut vid detta trafikfall.

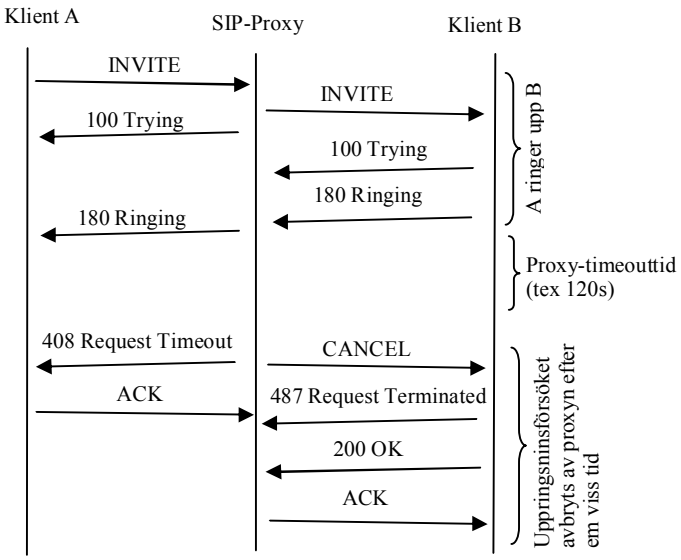


Figur 3.5: Uppkopplingsförsök av samtal där mottagaren avbryter med ett aktivt beslut eller är upptagen i samtal.

Meddelandeutbytet vid uppkopplingen av samtalet ser precis ut som vanligt fram till ”180 Ringing”-meddelandet. För att informera uppringaren att mottagaren är upptagen skickas antingen ett ”486 Busy Here”-meddelande eller ett ”603 Decline”-meddelande till proxyn för att sedan vidarebefordras till uppringaren. Vilket meddelande som skickas beror på vilken klientprogramvara mottagaren använder. När detta meddelande kommit in skickas ett ACK-meddelande tillbaka till den som skickade det.

3.2.1.5 Samtal som ej besvaras – uppringd gör ingenting

I figur 3.6 nedan visas hur meddelandeutbytet ser ut vid ett uppringningsförsök om mottagaren ej är upptagen men inte heller svarar.



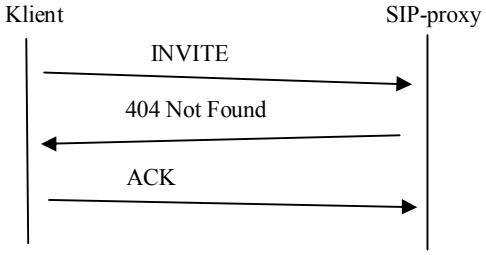
Figur 3.6: Uppringningsförsök där den uppringde parten inte gör någonting.

Hur länge uppringningsförsöket pågår beror vanligtvis på proxyn. I denna är det satt hur länge det får ta från att ”180 Ringing”-meddelandet skickats tills mottagaren reagerar på samtalsförsöket (t.ex. genom att svara då ett ”200 OK”-meddelande skickas). Om uppringningsförsöket pågått för länge ur proxyns synvinkel avbryter proxyn detta försök. Detta görs genom att proxyn skickat ett ”408 Request Timeout”-meddelande till uppringaren samt ett CANCEL-meddelande till mottagaren. Som svar på ”408 Request Timeout”-meddelandet skickar uppringande part ett ACK-meddelande. När det gäller mottagaren skickar denna ett ”487 Request Terminated”-meddelande samt ett ”200 OK”-meddelande som svar på proxyns meddelande. Proxyn svarar på detta med ett ACK-meddelande.

När det gällde en av klienterna som testades (SJPhone, Linux versionen) skickade denna klientprogramvara ett ”180 Ringing”-meddelande var 35:e sekund vilket gjorde att det inte blev någon timeout, utan uppringningsförsöket fortsatte även efter den tid proxyn hade definierat som det tid då försöket skulle avbrytas hade nåtts.

3.2.1.6 Försök att ringa part som inte är registrerad i SIP-servern.

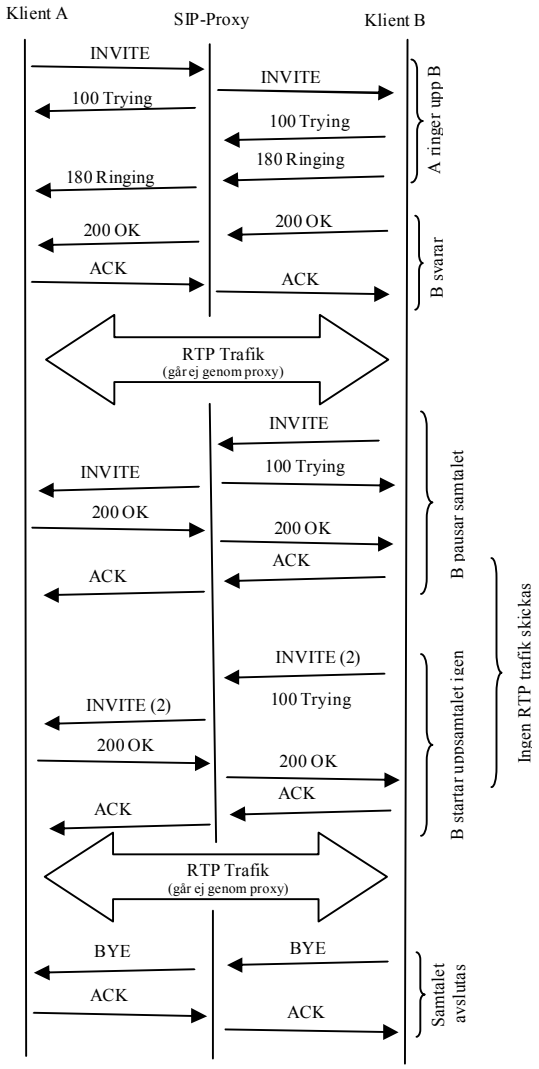
I figur 3.7 nedan visas hur meddelandeutbytet ser ut om den part som den uppringande parten försöker ringa inte är registrerad i SIP-proxyn. SIP-proxyn sänder ett ”404 Not Found”-meddelande som svar på sändarens INVITE-meddelande och denna svarar med ett ACK-meddelande.



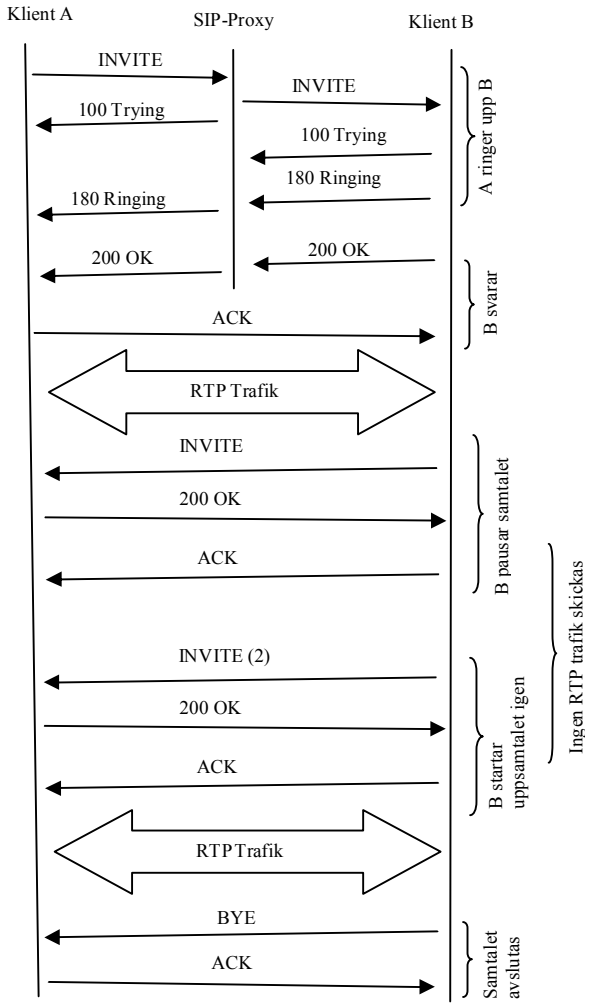
Figur 3.7: Uppringningsförsök där mottagaren inte finns registrerad i proxyn.

3.2.1.7 Samtal där någon av parterna pausar samtalet

I ett flertal IP-telefoniklienter går det pausa ett samtal och sedan återuppta det. I figurerna 3.8 och 3.9 nedan visas hur meddelande utbytet ser ut vid detta trafikfall, 3.8 är när ”record route” används medan 3.9 är när detta inte är satt.



Figur 3.8: Meddelandeutbyte vid samtal där samtalet pausas, med ”record route”.



Figur 3.9: Meddelandeutbyte vid samtal där samtalet pausas, utan ”record route”.

För att informera en part att den andra parten vill pausa ett pågående samtal skickar den part som vill pausa ett INVITE-meddelande till den andra parten. I detta INVITE-meddelande (som har samma Call-ID som alla andra SIP-meddelanden i samtalet) finns information i SDP-delen som informerar att detta är ett pausmeddelande. Vad för information som sätts är beroende på klienten. T.ex. i KPhone sätts både ”Connection adress” (under c) till 0.0.0.0 samt media porten till 0 i INVITE- samt ”200 OK”-meddelandet. När samtalet återupptas sätts ”Connection adress” tillbaka till klientens IP-nummer och media porten sätts till ett portnummer som inte är 0 (vanligtvis är inte detta portnummer samma port som innan pausningen av samtalet). I SJPhone sätts bara Connection adress till 0.0.0.0, media porten rörs ej.

Mottagaren av paus-INVITE-meddelandet skickar ett ”200 OK”-meddelande som svar på detta och ett ACK skickas tillbaka. För att återuppta samtalet skickas ännu ett INVITE-meddelande med information i SDP-delen som berättar att samtalet ska återupptas. Denna meddelandetransaktion uppför sig på ett liknande sätt som transaktionen som var vid pausningen av samtalet.

Om ”record route” används går alla meddelandena via proxyn (figur 3.8) annars går de direkt mellan samtalspartena (figur 3.9).

3.2.1.8 Tredje part ringer upp någon som redan är i samtal

Ibland kan en tredje part ringa upp en annan som redan är i samtal, det beror på vilken klient den mottagande parten har hur klienten uppför sig i detta fall.

I Kphone och SJPhone informerar klienten användaren om att ett samtal kommit in och personen vid klienten får ta ett beslut vad som ska göras. När det gäller KPhone måste parten pausa sitt första samtal för att svara på det inkommande samtalet, men efter detta kan parten starta upp det första samtalet igen och ha båda samtalen aktiva. När det gäller SJPhone pausas det första samtalet automatiskt om mottagaren bestämmer sig för att svara den tredje parten (det går sen precis som med KPhone sätta igång det första samtalet igen). Om det går att höra båda samtalen har ej testats.

Linphone klarar bara av ett samtal i taget och därför får den andra som ringer upp ett ”486 Busy Here”-meddelande som svar. Meddelandeutbytena för att starta samtal samt pausa dem är som det beskrivits i kapitlen ovan.

3.2.1.9 Andra scenarion och inställningar

Ett antal andra trafikfall som var mera klientspecifika testades också under projektet. Nedan beskrivs dessa scenarion och andra inställningar.

I Linphone går det att ställa in olika inställningar hur klienten ska uppföra sig vid inkommande samtal.

- Det går att ställa in klienten som Away/Do Not Disturb. Då skickas ett ”480 Temporarily Unavailble”-meddelande (via proxyn) till den som ringt upp. Detta meddelande svaras med ett ACK.
- Om klienten är inställd som ”Busy” skickas ett ”486 Busy Here”-meddelande.
- Inställningen ”Moved Temporarily” skickar meddelandet ”302 Moved Temporarily” till uppringaren.
- Det går att ställa in ”Alternative Service”. Då skickas meddelandet ”380 Alternative Service” med denna alternativa services URI i ”Contact”-raden i SIP-huvudet av meddelandet.

I KPhone går det att ställa in att samtalet ska skickas vidare till en annan klient. Detta görs under ”Call Forwading”. Med detta inställt skickas ett ”302 Moved Temporarily”-meddelande tillbaka med information on den andra mottagarens URI i ”Contact”-raden i SIP-huvudet. Om Kphone får in ett sådant meddelande kommer ett fönster upp som frågar om användaren vill använda denna adress i stället och om detta är fallet startas ett samtal upp mot denna nya adress.

I SJPhone finns inställningen ”Do Not Disturb”, då skickas ett ”486 Do Not Disturb”-meddelande (vanligtvis kallas 486 för ”Busy Here”) till uppringaren.

Det finns troligtvis en del andra scenarion etc. Men detta har inte undersökts utan trafikfallen ovan är det fokuserats på.

3.2.2 Information i signaleringsmeddelandena

Under laborationen undersöktes också den information som fanns i SIP-meddelandenas huvuden samt i deras SDP-data och vad av denna information som kan vara av intresse att använda. Nedan listas information som det kom fram till skulle kunna vara till nytta.

- Användare (användarnamn@sipserveradress.se) i ”From”- och ”To”-raderna i SIP-huvudet som finns i INVITE-meddelandet (och de andra SIP meddelandena)
- Klientadresser efter användarnamnet och @ tecknet i raden ”Contact” i huvudet. Uppringarens adress finns i INVITE-meddelandet medan den uppringda finns i ”180 Ringing”-meddelandet. Adresser kan också ses i VIA-raden.
- Raden ”Call-ID” i huvudet identifierar vilket samtal SIP-meddelandet tillhör.
- Raden ”CSeq” visar vilken transaktion ett meddelande tillhör.
- Media typ, protokoll och format (kodek) kan ses i SDP delen (m och a) av INVITE- och ”200 OK”-meddelandena. INVITE-meddelandet visar vad uppringaren vill/kan köra för mediatyper medan ”200 OK”-meddelandet visar vad den uppringda klarar av dessa mediatyper och vad som kommer att användas.
- I SDP-informationen finns också vilka portar (I m liksom ovan, detta visas i INVITE- och ”200 OK”-meddelandet) som används av RTP-trafiken.
- Vid pausning av ett samtal använder olika klienter olika sätt att berätta i paus-INVITE-meddelandet att ingen trafik ska skickas. Vissa klienter sätter portnumret till 0. När samtalet ska återupptas ändras porten igen. RTP-trafiken kör *inte* på samma portar före och efter att samtalet varit pausad om denna metod används. Andra klienter (t.ex. SJPhone) sätter ”Connection Information” (c) till 0.0.0.0 och när pausningen avslutas sätts c tillbaka till klientens IP-adress.

Lösningen som gjordes för att ta tillvara informationen för att kunna göra kvalitetsmätningar med NeTraMet redovisas i kapitel 4.

3.2.3 Parametrar från signaleringen som kan användas vid kvalitetsmätning

Vid kvalitetsmätning av IP-telefoni kan även vissa andra kvalitetsparametrar, förutom trafikstatistik såsom fördröjningar, fördröjningsvariationer, genomströmning och tappade paket vara av intresse. T.ex. kan det vara intressant att få information om samtal som inte lyckas kopplas upp (såsom spärr, upptaget etc.) och hur lång tid samtalen håller på.

När det gäller uppkopplingsstatisik kan detta tas reda på genom att titta på de olika inkommande status-SIP-meddelanden som skickas som svar i uppkopplingsfasen.

Det har inte lyckats simulera samtalsspärr i projektet men genom att titta på specifikationen för SIP [2] borde meddelandena som skickas antingen vara ett 4xx-meddelande eller vid serverfel ett 5xx-meddelande.

Vid upptaget skickas antingen ”486 Busy Here” eller ”603 Denied” beroende på den uppringde partens klientprogramvara.

Om uppringaren lagt på luren innan mottagaren svarat ses detta genom att statusmeddelandet ”487 Request Terminated” skickas.

Har det ringt för länge och uppkopplingsförsöket därför avbryts skickas statusmeddelandet ”408 Request Timeout” till uppringaren. Proxyn kör också en CANCEL-session med den uppringde parten.

Om mottagaren inte finns eller ej är registrerad i proxyn skickas statusmeddelandet ”404 Not Found”.

Ibland kan mottagaren vara temporärt oåtkomlig (detta kan t.ex. vara satt i klienten) eller att parten har temporärt flyttat. Detta informeras med meddelandena ”480 Temporarily Unavailble”, ”302 Moved Temporarily” (med eventuell information om en alternativ URI men detta är inte tvunget att ha) eller ”380 Alternative Service” (med information i Contact-raden i SIP-huvudet med denna alternativa services URI).

När det gäller att få information om de olika samtalens uppkopplingstid är en möjlig lösning att få denna information genom att subtrahera tiden då BYE-meddelandet skickades (när samtalet avslutades) med INVITE-transaktionens ACK-meddelande. En annan lösning som inte använder sig av SIP-meddelandena är att eventuellt titta på RTP-paketens tidsstämplar, men dessa paket tillhör inte signaleringtrafiken utan datatrafiken.

Ingenting av det som nämns ovan har tagits med i lösningen som beskrivs i kapitel 4 utan denna lösning mäter bara RTP-trafikstatistiken. Men det borde vara lätt att bygga ut de Perlskript som beskrivs där så att de även involverar dessa parametrar t.ex. genom att ha räknare som räknar hur många av de olika statusmeddelandena som kommer in. SER sparar också information om vilka och hur många statusmeddelanden som proxyservern sett sen senaste omstart av proxyn, detta kan ses med hjälp av verktyget *serctl*.

När det gäller lagring av information om uppkopplingstider kan detta lösas t.ex. genom att ha variabler som sparar information såsom minimum-, maximum- och medelvärde av de olika samtalens uppkopplingstider.

4 Utveckling av en prototyp för övervakning av SIP-baserad trafik med NeTraMet

För att kunna övervaka trafiken för IP-telefonisamtal som kopplas upp gjordes en prototyp. Denna prototyp är ett system som tar den information som behövs från SIP-meddelandena som skickas när ett samtal kopplas upp och sedan använder denna information för att skapa den SRL-fil som behövs för att starta NeMaC. När samtalet besvaras startas insamlandet av trafikstatistik. Eftersom RTP-trafiken kör på olika portar behövs en lösning att få information om porten som ska användas under samtalet på något sätt. Denna information finns i SIP-meddelandena som sätter upp samtalet som ska övervakas.

Nedan beskrivs de initiala testerna som gjordes för att få flödesmätaren NeTraMet att fungera samt att hitta ett sätt att generera de OAM-paket som behövs under mätningen. Därefter beskrivs hur informationen i SIP-meddelandena kan användas för att konfigurera mätningen av dataflödet samt att prototyperna, skrivna i Perl, för de två lösningarna som skapades beskrivs.

4.1 Flödesmätaren NeTraMet

Innan skapandet av prototypen undersöktes hur NeTraMet skulle användas för att kunna mäta RTP-trafik. NeTraMet installerades på alla klientdatorerna samt SIP-servern som användes under SIP-laborationen som beskrevs i kapitel 3. För att se hur NeTraMet och NeMaC fungerade användes ett SRL-skript som fåtts från ett annat examensarbete. Detta skript använde sig av ICMP-paket både för datatrafiken och för OAM-paketen, alla dessa genererades med *ping*. För att sedan få NeTraMet och NeMaC att mäta RTP-trafik skrevs SRL-filen om så att den hanterade UDP-trafik i stället. Först för bara datatrafiken men sedan även för OAM-paketen (efter att en undersökning hur dessa skulle genereras hade gjorts). I början definierades både sändardatorns och mottagardatorns IP-adresser och portar i SRL-skriptet, men det framkom under de initiala testerna att bara sändardatorns IP-adress och portar behövdes för att flödena som skulle mätas skulle definieras korrekt. Detta eftersom RTP-porten som används är reserverad för bara det pågående samtalets trafik så länge samtalet pågår och inga andra portar används. När det gäller OAM-paketen gäller samma sak så länge som de olika instanserna av udpgeneratorn använder olika sändarportar vid flera samtal från samma sändardator. Exempel på hur SRL-filen kan se ut (i detta fall genererad av prototypen som utvecklades) kan ses i appendix C.

För genereringen av datatrafiken, när det omgjorda SRL-skriptet testades, skapades denna trafik genom att ett samtal kopplades upp mellan två klienter. För att få en specifik RTP-port, eftersom denna port vanligtvis inte specificeras förrän samtalet kopplas upp, användes IP-telefoni klienterna Kphone och Linphone eftersom RTP-porten som ska användas kan definieras i förhand i dessa program. Hur OAM-paketen genererades beskrivs i kapitel 4.2 nedan.

I de första testerna som gjordes kördes bara NeTraMet och en instans av NeMaC tillsammans med udpgeneratorn på den uppringande partens dator, detta bara för att se att programmen fungerade ihop och genererade data. Därefter kördes även NeTraMet upp på mottagarens dator och för att få data från denna kördes det upp en andra instans av NeMaC på den uppringande partens dator som hämtade information från mottagarens NeTraMet.

4.2 Generering av OAM-paket

Eftersom RTP-trafiken som skulle övervakas är UDP-trafik behövde OAM-paketen vara UDP-paket med samma storlek som RTP-paketen. Detta för att datatrafiken och OAM-paketen inte ska hanteras annorlunda i nätverket vilket kan vara en risk om t.ex. ICMP-paket används som OAM-paket.

Tre olika program testades för att generera OAM-paketen:

Det första är ett program som heter UDP Ping Logger som utvecklats av Nerdlabs Consulting. Detta program skickar ett UDP-paket en gång i sekunden till mottagardatorns echo-port (port 7) och väntar sedan på svar [18]. Beroende på om paketet kommer fram eller inte skrivs detta med ett tecken i en loggfil. Problemet med detta program är att det inte går att definiera hur ofta UDP-paketen ska skickas, ej heller hur stora dessa ska vara.

Det andra programmet var ett Perl-skript som skrevs ihop speciellt för att testas i examensarbetet. Detta program använde sig av Perl-modulerna Net::Ping och Time::HiRes för att skicka UDP-ping mot port 7 på mottagardatorn. Problemet med detta program var att det inte gick att veta i förväg vilken port UDP-pingen skickades ut ifrån (detta går ej att definiera om Net::Ping används) vilket NeMaC behöver veta.

Det tredje programmet som testades, och som sedan användes, heter *ngen*. Detta program är del av en programsvit som heter NTools som utvecklats av Norbert Vegh på TeliaSonera AB R&B. NTools kan generera både UDP- och TCP-strömmar med olika konfigurerbara egenskaper som sedan på mottagarsidan kan tas om hand för att mäta olika trafikkvalitetsparametrar [15]. När OAM-paketen genererades användes bara generatorn *ngen* från detta programpaket. Generatorn sattes att generera 200 bytes UDP-paket (samma storlek som RTP-paketen) mot mottagardatorns echo-port (7) så att OAM-paketet skulle skickas tillbaka till sändardatorn. Dessa paket skickades med en datahastighet (rate) på 160 bps (1 paket per 10:de sekund) under testerna som gjordes för att se att NeTraMeT med det skrivna SRL-skriptet och *ngen* fungerade som det skulle. Under trafikmätningen mellan Karlskrona och Haninge som sedan gjordes för att testa prototypen som skapades (se kapitel 5) var hastigheten 1600 bps (1 paket i sekunden).

4.3 Konfigurering av trafikmätning med information från SIP-meddelandena

Den information som behövs för att skapa SRL-skriptet, som ger den regelfil de två instanserna av NeMaC ska använda, är den uppringande partens IP-nummer och RTP-port samt vilken port udpgeneratorn ska skicka ut OAM-paketen från. Som framkom under arbetet med att titta på SIP-meddelanden som gjordes innan finns denna information i det INVITE-meddelande som skickas av den uppringande parten när denne vill koppla upp ett samtal. IP-numret hittas på Contact-raden i SIP-huvudet medan RTP-porten hittas i den SDP-data som skickas med meddelandet. Udp-generatorns port definieras vid start av denna (*ngens* grundinställning för denna port är 8888). I utskriften av ett INVITE-meddelande nedan är den information som används av prototypen markerad med fetstil.

```
INVITE sip:test@7017ws05.haninge.kth.se SIP/2.0
Via: SIP/2.0/UDP
193.10.38.106;rport;branch=z9hG4bKc10a266a00000039430db1250000076100000293
Content-Length: 339
Contact: <sip:emma@193.10.38.106:5060>
Call-ID: 25A78A68-676D-4709-A2EF-716C5B3ED41A@193.10.38.106
Content-Type: application/sdp
CSeq: 1 INVITE
From: "Emma Roos"<sip:emma@193.10.38.105:5060>;tag=925268720508
Max-Forwards: 70
To: <sip:test@7017ws05.haninge.kth.se>
User-Agent: SJphone/1.60.289a (SJ Labs)

v=0
o=- 3333959589 3333959589 IN IP4 193.10.38.106
s=SJphone
c=IN IP4 193.10.38.106
```

```
t=0 0
a=direction:active
m=audio 49172 RTP/AVP 3 97 98 8 0 101
a=rtpmap:3 GSM/8000
a=rtpmap:97 iLBC/8000
a=rtpmap:98 iLBC/8000
a=fmtp:98 mode=20
a=rtpmap:8 PCMA/8000
a=rtpmap:0 PCMU/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-11,16
```

Udpgeneratorn behöver också veta till vilken dator den ska skicka UDP-paketen till, i detta fallet den uppringde partens IP-nummer. Detta IP-nummer finns i det ”180 Ringing”-meddelande som skickas när mottagaren tagit emot den uppringande partens INVITE-meddelande (och mottagaren ej redan är upptagen). IP-numret hittas även här i Contact-raden i SIP-huvudet på meddelandet. Nedan visas en utskrift av ”180 Ringing”-meddelandet som skickades som svar på INVITE-meddelandet ovan med den viktiga informationen prototypen använder sig av i fetstil.

```
SIP/2.0 180 Ringing
Via: SIP/2.0/UDP
193.10.38.106;rport=5060;branch=z9hG4bKc10a266a00000039430db1250000076100000293
Content-Length: 0
Contact: <sip:test@193.10.38.111:5060>
Call-ID: 25A78A68-676D-4709-A2EF-716C5B3ED41A@193.10.38.106
CSeq: 1 INVITE
From: "Emma Roos"<sip:emma@193.10.38.105:5060>;tag=925268720508
Record-Route: <sip:193.10.38.105;ftag=925268720508>
Server: SJphone/1.60.289a (SJ Labs)
To: "unknown"<sip:test@7017ws05.haninge.kth.se>;tag=12715031573
```

För att se att ett inkommande SIP-meddelande tillhör ett specifikt samtal kan även samtalets unika id-nummer, Call-ID, vara till nytta att veta. Detta finns i alla SIP-meddelandens SIP-huvud på Call-ID-raden. Detta Call-ID sätts av det allra första SIP-meddelandet som startar samtalet nämligen det INVITE-meddelande som den uppringande parten skickar för att starta hela processen att koppla upp ett samtal och har samma värde i alla SIP-meddelanden som tillhör det pågående samtalet. Även Call-ID är markerat med fetstil i utskrifterna av SIP-meddelandena som visas ovan.

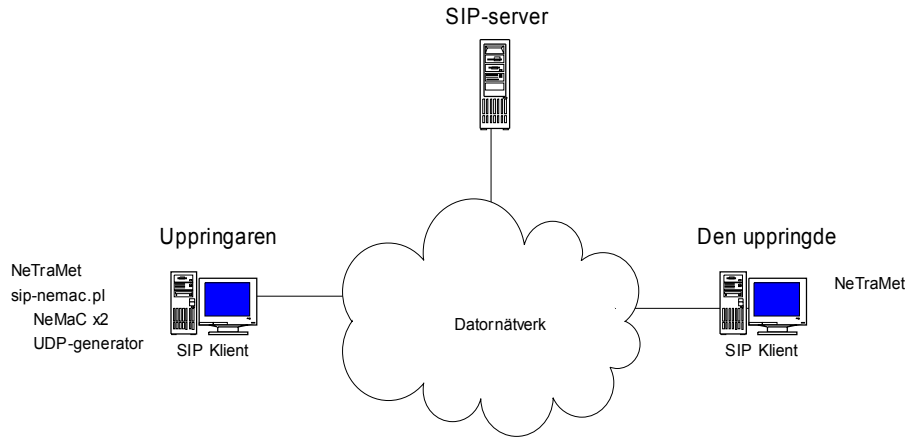
4.4 Prototypens uppbyggnad

För att kunna göra mätningarna måste på något sätt den datorn som ska göra själva trafikmätningen se uppkopplingen av samtalet för att sedan: ta informationen från signaleringsmeddelandena och skapa SRL-filen, kompilera denna fil, starta två instanser av NeMaC som pratar (via SNMP) med de NeTraMet-instanser som kör på datorerna som tillhör de två parterna i samtalet samt starta upp udpgeneratorn på en av datorerna som är involverad i samtalet (i prototypen blev detta den dator som tillhörde den uppringande parten).
För att kunna göra detta skapades ett program skrivet i skriptspråket Perl. Programmet använder sig av Perl-modulerna Net::PcapUtils [19] och NetPacket [20] för att titta på de paket som kommer in till datorn skriptet kör på och om paketet är ett SIP-paket hantera detta. Vid uppkopplingen av ett samtal skapar skriptet SRL-filen som ska användas från informationen som fås från SIP-meddelandena och kompilerar denna för att sedan om mottagaren svarar starta upp två instanser av NeMaC samt udpgeneratorn. Denna lösning i Perl gjordes i två varianter. Den första är ett skript som körs på alla klienter och där den klient som initierar samtalet är den som genererar OAM-paketen och samlar in trafikstatistiken. Den andra varianten är att klienten som initierar samtalet fortfarande genererar OAM-

paketen men det är SIP-servern som samlar in trafikstatistiken. I båda fallen kör alla klienterna NeTraMet.

4.4.1 Lösning 1: Klienten utför statistikinsamling

Den första lösningen är att ett program, prototypen *sip-nemac.pl*, samt NeTraMet körs på alla datorer vars IP-telefonitrafik ska mätas. Det är den part som initierar samtalet (uppringande parten) som är den som kommer att skicka ut OAM-paketen samt hämta in informationen med hjälp av NeMaC från de NeTraMet-instanser som kör hos både den uppringande parten och den uppringde parten (denne behöver bara köra NeTraMet för att mätningen ska fungera). Var programmen körs visas i figur 4.1.



Figur 4.1: Översikt över de program som körs och var de körs vid lösningen då klienten gör statistikinsamlingen.

Nedan kommer en nerbrytning av Perlskriptet (detta skripts kod kan ses i sin helhet i bilaga D).

Skriptet är uppdelat i ett antal delar och subrutiner:

- Importering av de Perlmoduler som ska användas.
- Definiering av globala variabler.
- Subrutinen *createconf()* som skapar SRL-filen samt kompilerar denna.
- Subrutinen *siphandler()* hanterar de SIP-meddelanden som kommer in.
- Subrutinen *get_callid()* returnerar ett SIP-meddelandes Call-ID. Denna subrutin används i huvudsak av *siphandler*-subrutinen.
- Subrutinen *got_a_packet()* kontrollerar om ett paket som kommit in är ett SIP-paket och om det är det anropas subrutinen *siphandler*.
- Huvudprogrammet öppnar upp nätverkskortet och startar en while-loop som hämtar in de UDP-paket som kommer in, dessa paket skickas till *got_a_packet*-subrutinen.

4.4.1.1 Import av Perlmoduler samt definiering av globala variabler

Det första som görs i Perl-skriptet är att de Perl-moduler som ska användas i skriptet importeras.

```
use Net::PcapUtils;
use NetPacket::Ethernet qw(:strip);
use NetPacket::IP;
use NetPacket::UDP;
```

Net::PcapUtils är en modul som är kopplad till modulen Net::Pcap som kopplar Perl mot libpcap. Net::PcapUtils består av ett antal funktioner som används för att enkelt kunna öppna upp ett nätverksinterface för att se de paket som passerar detta samt hantera denna information [19].

NetPacket är en grupp av moduler som kan användas för att avkoda olika paketsorter [20]. De moduler som används i Perlskriptet är NetPacket::Ethernet, NetPacket::IP och Netpacket::UDP.

Efter att modulerna importeras definieras globala variabler och variabler som ska kunna konfigureras av den som ska använda skriptet.

```
my $monitorpacketinterval = 1;
my $monitorpacketsize = 200;
my $collectinterval = 30;

my $myip = shift || "none";
my $sourceip;
my $sourceport;
my $destinationip;
my $nemacconf;
my $callid = "none";
```

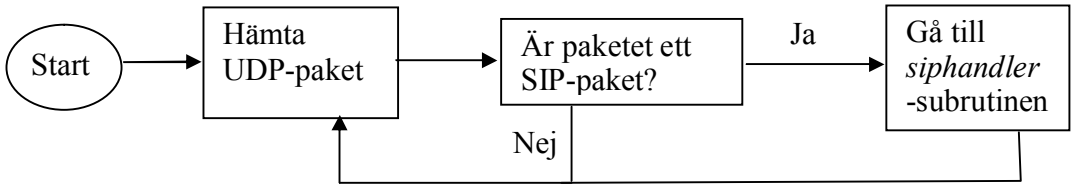
Variablerna *$monitorpacketinterval* och *$monitorpacketsize* används av *ngen*, den första variabeln är tiden mellan OAM-paketen medan den andra är storleken på OAM-paketen (exklusive länknivåhuvudet (Ethernet) på paketet). Variabeln *$collectinterval* definierar tiden mellan NeMaCs hämtningar av information från NeTraMet. Dessa tre variabler är definierade i början av skriptet eftersom dessa är variabler som lätt ska kunna ändras av användaren av skriptet för att passa de mätningar som görs. Resten av variablerna är globala variabler.

Variabeln *$myip* är IP-numret på den dator som trafiken för de IP-telefonisamtal som den startar (i detta fall den dator skriptet kör på) ska mätas. Detta IP-nummer definieras på kommandoraden när Perlskriptet startas upp. Det ska gå att byta ut värdet ”none” till datorns IP-nummer om skriptet inte ska användas till flera olika IP-nummer.

Variablerna *$sourceip*, *$sourceport*, *$destinationip*, *$nemacconf* och *$callid* sätts av subrutinen *siphandler()* om ett IP-telefonisamtal håller på att kopplas upp. De tre första variablerna är ganska självförklarande *$sourceip* och *$destinationip* är samtalets parters IP-nummer och *$sourceport* är den uppringande partens UDP-port. Variabeln *$nemacconf* används för att definiera namnet på de filer som skapas av programmet, nämligen SRL-, regel-, logg- och flödesfilerna som NeMaC använder sig av och genererar. Sist är det variabeln *$callid* som sätts till id-numret för det samtal vars trafik ska mätas, denna variabel är till för att se att de olika SIP-meddelandena som kommer in tillhör rätt samtal.

4.4.1.2 Huvudprogrammet och subrutinen got_a_packet()

Huvudprogrammet som finns skrivet sist i sip-nemac.pl beskrivs nedan. I figur 4.2 visas ett enkelt flödesdiagram vad huvudprogrammet och *got_a_packet* i stora drag ska göra.



Figur 4.2: Flödesdiagram över vad programmet i stora drag ska göra.

När programmet startas kontrolleras att ett IP-nummer är satt på kommandoraden (eller blivit definierat i *$myip* variabeln). Om detta inte är fallet avslutas programmet och ett meddelande skrivs ut om att ett IP-nummer måste definieras när Perlskriptet startas.

```
if ($myip eq "none") {
    print "Usage: perl sip-nemac.pl the_computers_ip.\n";
    exit;
}
```

Om *$myip* definierats skrivs information ut om när Perlskriptet startades och skanningen efter paket startar.

```
else {
print "\nProgram started ", `date`;
print  "************************************************************\n\n";

    $pktdesc = Net::PcapUtils::open(SNAPLEN => 1500, FILTER => 'udp', PROMISC =>
0);
    my ($next_packet, %next_header);

    while(1){
      ($next_packet, %next_header) =  Net::PcapUtils::next($pktdesc);
      got_a_packet($next_packet);
    }
}
```

Först öppnas nätverkskortet för att låta libpcap hantera inkommande UDP-paket. Detta görs med Net::PcapUtils *open*-funktion. Denna är satt till att ta in alla UDP-paket helt och hållet. Efter detta startas en *while*-loop som kommer att köra tills Perlskriptet avbryts (t.ex. med ctrl-c). Denna *while*-loop tittar efter inkommande UDP-paket och om ett paket kommer in skickas detta till subrutinen *got_a_packet()*.

```
sub got_a_packet {
    my $packet = shift;
    $ip_obj = NetPacket::IP->decode(eth_strip($packet));
    $udp_obj = NetPacket::UDP->decode($ip_obj->{data});
    # SIP message
    if ($udp_obj->{dest_port} == 5060){
      siphandler;
    }
}
```

Subrutinen *got_a_packet()* avkodar paketet, detta gör att viktig information om paketet läggs i olika variabler som kan användas, såsom sändarens och mottagarens IP-adress, sändar- och mottagarportnummer, UDP-data m.m. Därefter kontrollerar subrutinen om paketet är ett SIP-paket

genom att se om mottagarporten är 5060. Om detta är fallet skickas paketobjeket vidare till subrutinen *siphandler()*.

4.4.1.3 Subrutinen siphandler()

Subrutinen *siphandler()* är den del av Perlskriptet som hanterar de inkommande SIP-meddelandena. Denna subrutin känner igen ett flertal vanliga SIP-meddelanden och hanterar dessa. Vissa SIP-meddelanden är inlagda i hanteringsalgoritmen för SIP i informativt syfte, och om de dyker upp skrivs bara information ut om att de har kommit in. Dessa är inte alltför viktiga för själva övervakningen av samtalstrafiken men kan vara av intresse att veta. Eventuellt att meddelandena som är kopplade till REGISTER-transaktionerna kan vara av intresse i vidare utveckling av serverversionen av skriptet (som beskrivs i kapitel 4.4.2), för att se vilka datorer som inom den närmsta framtiden kan starta upp samtal och starta upp en instans av *sip-nemac-server.pl* för dessa nyligen registrerade datorer.
De SIP-meddelanden som är intressanta när datatrafiken för ett samtal ska mätas är de meddelanden som är kopplade till INVITE- och BYE-transaktionerna. I figur 4.3 visas ett flödesdiagram över hur programmet uppför sig vid ett inkommande samtal och resten av detta kapitel beskriver Perlskriptet noggrannare vad som görs med de olika SIP-meddelandena som kommer in.

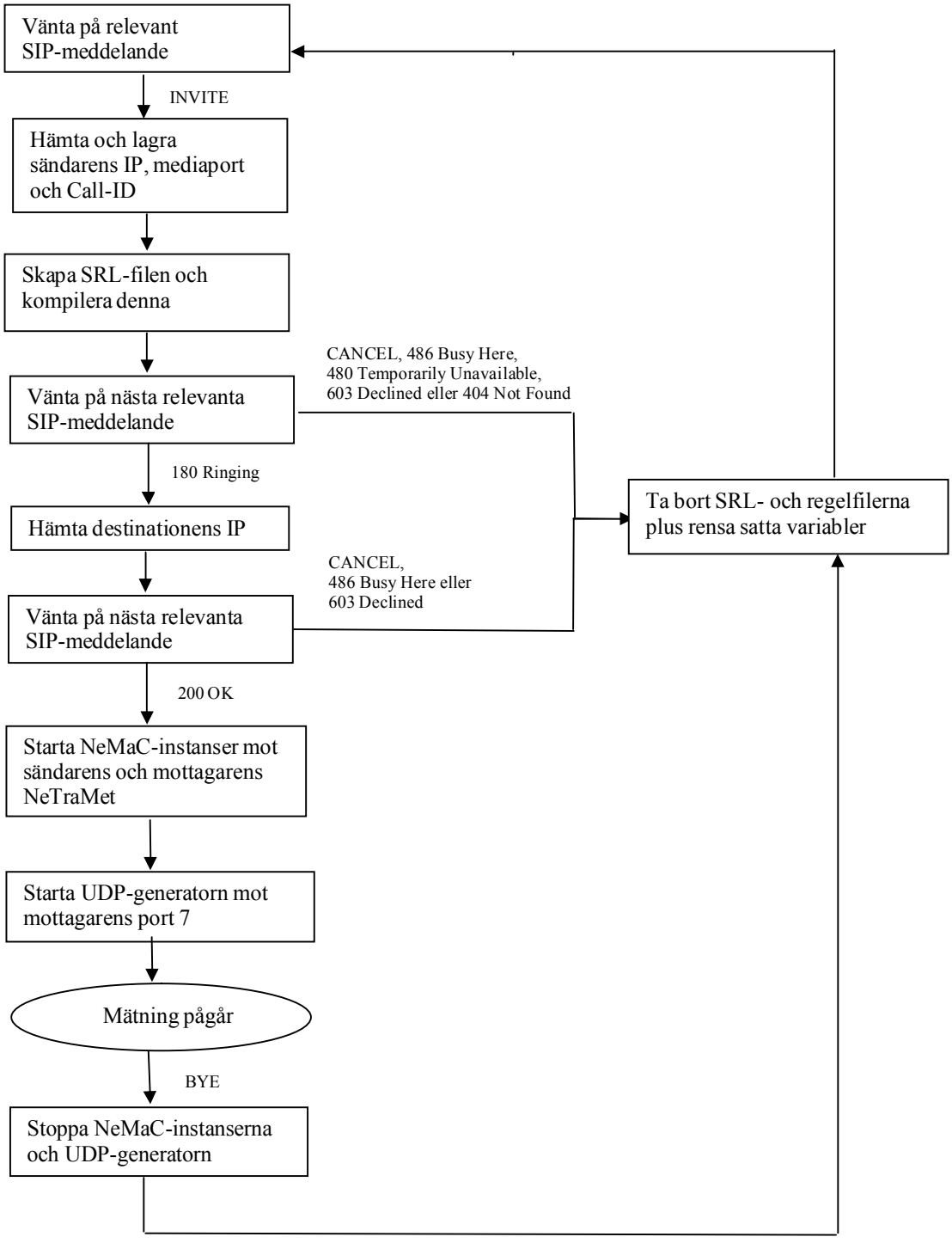
Att ett samtal ska kopplas upp märker Perlskriptet genom att den får in ett INVITE-meddelande. Programmet ser att ett paket är ett INVITE-meddelande genom att UDP-data som finns i den variabel *got_a_packet()* skapat börjar med ordet INVITE.

```
if ($udp_obj->{data} =~ /^INVITE/){
  print "Incoming INVITE ";
```

När programmet sett att meddelandet är ett INVITE kontrollerar skriptet att paketets IP-nummer är detsamma som IP-numret som definierades (*$myip*) vid start av programmet (om inte detta är fallet skrivs bara information ut om att ett INVITE-meddelande kommit in samt dess Call-ID men inget mer). Om IP-numret är detsamma som den IP-adress som är lagrad *$myip* görs en extra kontroll för att se att kontaktinformationen (IP-numret på Contact-raden i SIP-huvudet) också är IP-numret definierat i *$myip*, denna kontroll görs eftersom det kan vara en möjlighet att SIP-servern också har en SIP-klient körade på samma dator som servern och servern ser även SIP-meddelanden för samtal som den själv inte kopplar upp. Om även kontaktinformationens IP-nummer är *$myip* ska trafiken i samtalet som håller på att kopplas upp mätas.

```
if ($ip_obj->{src_ip} eq $myip){
  if ($udp_obj->{data} =~ /Contact:
  ([^@]+)@(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/){
    my $tempip = "$2";
    if ($tempip eq $myip) {
      print "from this computer.\n";
```

Därefter är det dags för programmet att ta den information den behöver från den UDP-data som sparats undan för att den ska kunna skapa SRL-filen som behövs. Denna data gås igenom och Call-ID, sändarens IP-nummer samt sändarens RTP-port hämtas.



Figur 4.3: Flödesdiagram över vad som händer i Perlskriptet på mottagardatorn vid ett inkommande samtal (kontroller av IP-nummer och om meddelandena tillhör rätt samtal borttaget för att förenkla).

För att få Call-ID körs subrutinen *get_callid()* och resultatet sparas i skalären *$callid*. Därefter används detta Call-ID för att sätta värdet i skalären *$nemacconf* som används för att definiera namnen på de filer som ska skapas för att mätningen ska kunna startas och de filer som genereras under själva mätningen. Denna variabel sätts till Call-ID för att filerna ska få unika namn, men för att slippa specialtecknet ”@” (det är vanligt att Call-ID avslutas med ett ”@” och sedan IP-numret för datorn som

initierade samtalet) i filnamnet kontrolleras det om detta finns med och om detta är fallet klipps @-tecknet och allt efter detta bort innan det sparas i *$nemacconf*-skalären.

```
$callid = get_callid();
if ($callid =~ /([^@]+)@(.+)/) {
   $nemacconf = "$1";
} else {
   $nemacconf = "$callid";
}
```

Sändarens IP-adress har redan lagts i en temporär variabel när ”Contact”-raden i SIP-huvudet kontrollerades. Denna temporära variabels värde läggs i skalären *$sourceip*.

```
$sourceip = $tempip;
print "The callers IP: ", $sourceip, "\n";
```

För att få reda på RTP-porten letar skriptet efter den rad i paketets UDP-data som börjar med ”m=audio” (denna information ligger i SDP-delen av SIP-meddelandet), numret som finns efter denna text är den RTP-port som kommer att användas, detta nummer läggs in i variabeln *$sourceport*. Eftersom skriptet bara letar efter texten ”m=audio” betyder detta att skriptet bara klarar trafik av mediatypen audio. I en framtida utveckling av skriptet kanske detta eventuellt också skulle ändras så att även andra mediatyper såsom data och video kan övervakas.

```
if ($udp_obj->{data} =~ /(m=audio )(\d+)/){
   $sourceport ="$2";
   print "The callers RTP-port: ", $sourceport, "\n\n";
}
```

När all information som ska tas från INVITE-meddelandet har fåtts körs subrutinen *createconf()* för att skapa SRL-filen samt kompilera den. Denna subrutin beskrivs i kapitel 4.4.1.4.

När mottagaren tagit emot det initiala INVITE-meddelandet (och denne har en IP-telefoniklient igång samt att denna klient inte är satt att ignorera samtal) svarar denna med ett ”180 Ringing”-meddelande. Perlskriptet ser att ett paket är ett ”180 Ringing” genom att se om variabeln med UDP-data börjar med texten ”SIP 2.0 180”. Om detta är fallet hämtar den det Call-ID som finns i SIP-huvudet på paketet och kontrollerar om detta är detsamma som Call-ID för det samtal som håller på att kopplas upp.

```
elsif ($udp_obj->{data} =~ /^SIP.2.0 180/) {
      print "It's ringing at the callee.\n";
      my $thiscallid = get_callid();
      if ($callid eq $thiscallid) {
          if ($udp_obj->{data} =~ /(Contact:
)([^@]+)@(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/){
            $destinationip = "$3";
            print "Mottagarens IP: ", $destinationip, "\n";
          }
      }
```

Om det är rätt Call-ID hämtas mottagarens IP-adress från ”Contact”-raden i SIP-huvudet och variabeln *$destiationip* sätts till denna.

Om mottagaren inte vill eller kan svara skickas ett statusmeddelande om detta tillbaka till den uppringde parten. De statusmeddelanden som Perlskriptet känner igen är ”486 Busy Here” och ”603 Decline” för att mottagaren inte vill svara samt ”480 Temporarily Unavailable” för om mottagaren ej är

åtkomlig för tillfället. För att se om SIP-meddelandet som kommit in är ett sådant statusmeddelande undersöks om paketets UDP-data börjar med ”SIP 2.0” och sedan numret på statusmeddelandet.

```
elsif ($udp_obj->{data} =~ /^SIP.2.0 (486|480|603)/){
      print "The recipient is unaccessible/busy.\n";
      my $thiscallid = get_callid();
      if ($callid eq $thiscallid) {
          system ("rm -f $nemacconf.srl $nemacconf.rules");
          $callid = "none";
      }
 }
```

Om SIP-paketet är ett sådant statusmeddelande kontrollerar skriptet om meddelandet tillhör det samtal som håller på att kopplas upp genom att kontrollera statusmeddelandets Call-ID mot samtalets Call-ID. Om statusmeddelandet tillhör samtalet tas SRL- och regelfilerna bort samt att skalären *$callid* sätts till grundvärdet ”none”. Samma sak ska göras om det visar sig att mottagaren inte är registrerad med SIP-servern (404 Not Found) samt om den uppringande parten avbryter samtalet innan mottagaren svarar (CANCEL).

Skillnaden är att med ”404 Not Found” kontrolleras också om meddelandet tillhör någon annan transaktion (t.ex. REGISTER) och då skriver den bara ut detta i konsolen. För att se om meddelandet är kopplat till en REGISTER-transaktion tittar Perlskriptet på CSeq-raden i SIP-huvudet. Denna rad visar vilken transaktion ett SIP-meddelande tillhör och har formatet: CSeq: nummer transaktionstyp (t.ex. CSeq: 1 INVITE eller CSeq: 1 REGISTER).

```
if ($udp_obj->{data} =~ /CSeq: \d+ REGISTER/) {
```

När det gäller CANCEL-meddelandet är skillnaden i hur detta hittas i paketets UDP-data. Här tittar Perlskriptet efter om meddelandet börjar med ett CANCEL.

```
elsif ($udp_obj->{data} =~ /(^CANCEL)/)
```

Om mottagaren svarar skickas ett ”200 OK”-meddelande till den uppringande parten. När detta kommer in ska själva insamlandet av information om datatrafikflödet startas. Perlskriptet ser att SIP-meddelandet är ett ”200 OK”-meddelande genom att titta om paketets UDP-data börjar med ”SIP 2.0 200”. För att sedan se om detta meddelande är ett svar på det initierande INVITE-meddelandet tittar skriptet på ”CSeq”-raden i SIP-meddelandet för att se att det tillhör en INVITE-transaktion samt att den ser om meddelandets Call-ID är detsamma som för det initierande INVITE-meddelandet. Skriptet räknar också ut, från informationen lagrad i *$monitorpacketsize* och *$monitorpacketinterval*, vilken ”rate” som ska sättas av *ngen* när detta program startas.

När skriptet vet att ”200 OK”-meddelandet som kommit in är kopplat till det initierande INVITE-meddelandet startas *NeMaC* och *ngen*. Dessa program startas med Perl-funktionen *open* för att kunna spara programinstansernas processid.

```
    elsif ($udp_obj->{data} =~ /^SIP.2.0 200/) {
        # The called party has "picked up". (Start NeMac and the UDP generator)
      if ($udp_obj->{data} =~ /CSeq: \d+ INVITE/) {
          print "Conversation has started.\n";
          my $thiscallid = get_callid();
          my $udpgenrate = ($monitorpacketsize * 8)/$monitorpacketinterval;
          if ($callid eq $thiscallid) {
            $nemacsrcpid = open(NEMACSRC, "NeMaC -c $collectinterval -g 120 -m
```

```
50000 -p -F $nemacconf.src.$sourceip.flows -L $nemacconf.log -r $nemacconf.rules
$sourceip write|");
            $nemacdstpid = open(NEMACDST, "NeMaC -c $collectinterval -g 120 -m
50000 -p -F $nemacconf.dst.$destinationip.flows -L $nemacconf.log -r
$nemacconf.rules $destinationip write|");
            $udpgenpid = open(UDPGEN, "ngen -proto udp -dstport 7 -rate
$udpgenrate -size $monitorpacketsize -dstip $destinationip|");
          }
      }
```

Två NeMaC-processer startas: den ena hämtar information från den uppringande partens dator (*$sourceip*) medan den andra hämtar från den uppringde partens dator (*$destinationip*). Båda programmen använder samma regelfil samt att de skriver till samma loggfil. När det gäller filerna där trafikflödenas data finns används olika filer. Alla filernas namn har värdet i skalären *$nemacconf* i början av filnamnet.

Det skickas också ett ”200 OK”-meddelande när ett BYE-meddelande skickats för att koppla ner samtalet. För att se om meddelandet tillhör BYE-transaktionen tittar skriptet i dess UDP-data för att se att värdet på CSeq-raden består av ett BYE samt att Call-ID är detsamma som det pågående samtalets Call-ID.

```
      elsif ($udp_obj->{data} =~ /CSeq: \d+ BYE/) {
          print "The conversation is ending.\n";
          my $thiscallid = get_callid();
          if ($callid eq $thiscallid) {
          }
```

När ”200 OK”–meddelandet för BYE kommer in är det meningen att NeMaC-instanserna och *ngen* ska stängas ner. Men p.g.a. att Liphone hade en bugg som ibland ledde till att inget ”200 OK”-svar skickades gör Perlskriptet detta när BYE meddelandet kommer in i stället. Detta betyder att ett fåtal RTP-paket inte mäts (men detta är så få (2-3 stycken) att det inte ska ha någon påverkan på resultatet av mätningen (i mätningarna som gjordes senare var ”hastigheten” ca 3000 paket/minut)).

BYE-meddelandet initierar nedkopplingen av samtalet och det är här som Perlskriptet avslutar trafikmätningen. Skriptet ser att SIP-meddelandet är ett BYE-meddelande genom att se om paketets UDP-data börjar med BYE. Därefter kontrollerar skriptet att SIP-meddelandet tillhör det pågående samtalet genom att jämföra Call-ID. När skriptet vet att det är rätt BYE-meddelande dödas NeMaC och *ngen*-processerna, skalären *$callid* sätts till ”none” samt att SRL- och regelfilerna raderas.

```
    elsif ($udp_obj->{data} =~ /^BYE/){
      print "BYE.\n";
      my $thiscallid = get_callid();
      if ($callid eq $thiscallid) {
          `kill $nemacsrcpid $nemacdstpid $udpgenpid`;
          close(NEMACSRC);
          close(NEMACDST);
          close(UDPGEN);

          $callid = "none";
          system ("rm -f $nemacconf.srl $nemacconf.rules");
      }
```

Subrutinen *siphandler()* hanterar även ett flertal andra statusmeddelanden som kan dyka upp. Dessa identifieras på samma sätt som statusmeddelandena som beskrivits ovan. Det enda som görs för dessa meddelanden är att information om att de kommit in skrivs ut i konsolen.

Om SIP-meddelandet som kommit in inte finns definierat i subrutinen skrivs paketets UDP-data (SIP-huvud och eventuell SDP-data) ut i konsolen.

```
    else {
      print "$udp_obj->{data}\n";
    }
```

För alla SIP-meddelanden som kommer in skrivs information om sändarens och mottagarens IP-nummer och port samt längden på paketets UDP-data ut i konsolen. Det skrivs också ut en separerare på konsolen i slutet av subrutinen för att det ska vara enkelt att särskilja de inkommande SIP-paketen.

4.4.1.4 Subrutinerna createconf() och getcallid()

Subrutinerna *createconf()* och *getcallid()* är kopplade till subrutinen *siphandler()*. Dessa delar bröts ut från den subrutinen eftersom *createconf()* tar stort utrymme medan funktionen *getcallid()* ger görs ett flertal gånger i algoritmen definierad i *siphandler()*.

Subrutinen *createconf()* skapar den SRL-fil som behövs för att kunna övervaka samtalet som kommit in samt kompilerar denna fil till en regelfil.

Det första som görs i subrutinen är att namnet på SRL-filen definieras. Detta namn är variabeln *$nemacconfs* värde med ett *".srl"* tillagt på slutet.

```
my $nemacsrl = "$nemacconf.srl";
```

Efter detta öppnas denna fil för skrivning och SRL-koden skrivs in i filen med de värden för sändardatorns IP-nummer och RTP-port inskrivna där de ska vara.

```
open CONFFILE,'>', $nemacsrl;

print CONFFILE "define OAM_SENDER = ", $sourceip, ";\n";
print CONFFILE "define OAM_PROTOCOL = udp;\n";
print CONFFILE "define OAM_SENDER_PORT = 8888;\n";
print CONFFILE "define DATA_SENDER_IP = ", $sourceip, ";\n";
print CONFFILE "define DATA_SENDER_PORT = ", $sourceport, ";\n\n";
...
```

När all SRL-kod skrivits in stängs SRL-filen och kompileras med *srl*-programmet så att den regelfil som NeMaC behöver skapas. Regelfilens filnamn består av värdet i variabeln *$nemacconf* med ett *".rules"* tillagt på slutet.

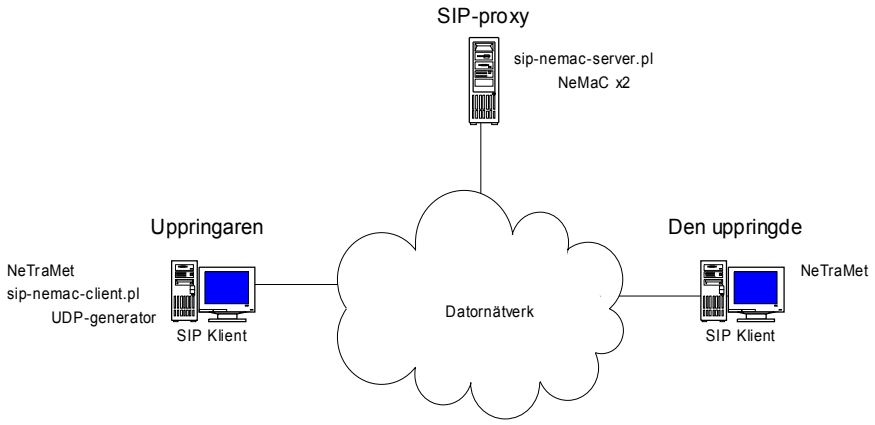
```
close CONFFILE;

system ("/usr/local/bin/srl", $nemacsrl);
```

När det gäller subrutinen *getcallid()* letar denna igenom paketets UDP-data efter samtalets Call-ID i SIP-paketet. När Call-ID hittats returneras detta.

```
sub get_callid {
    if ($udp_obj->{data} =~ /(Call-ID: )(.+)/){
      print "Call-ID: $2\n";
      return "$2";
    }
}
```

4.4.2 Lösning 2: SIP-servern utför statistikinsamling

Den andra lösningen är att SIP-servern är den dator som hämtar informationen från NeTraMet med NeMaC medan klienterna bara kör NeTraMet och udpgeneratorn. I denna lösning kör alla klienter, vars samtal klienten initierar ska mätas, en specifik klientversion av Perlskriptet (*sip-nemac-client.pl*) medan SIP-servern kör en instans av en serverversion av skriptet (*sip-nemac-server.pl*) för varje klient vars IP-telefonitrafik ska mätas, se i figur 4.4 för en översikt var programmen körs med denna lösning.

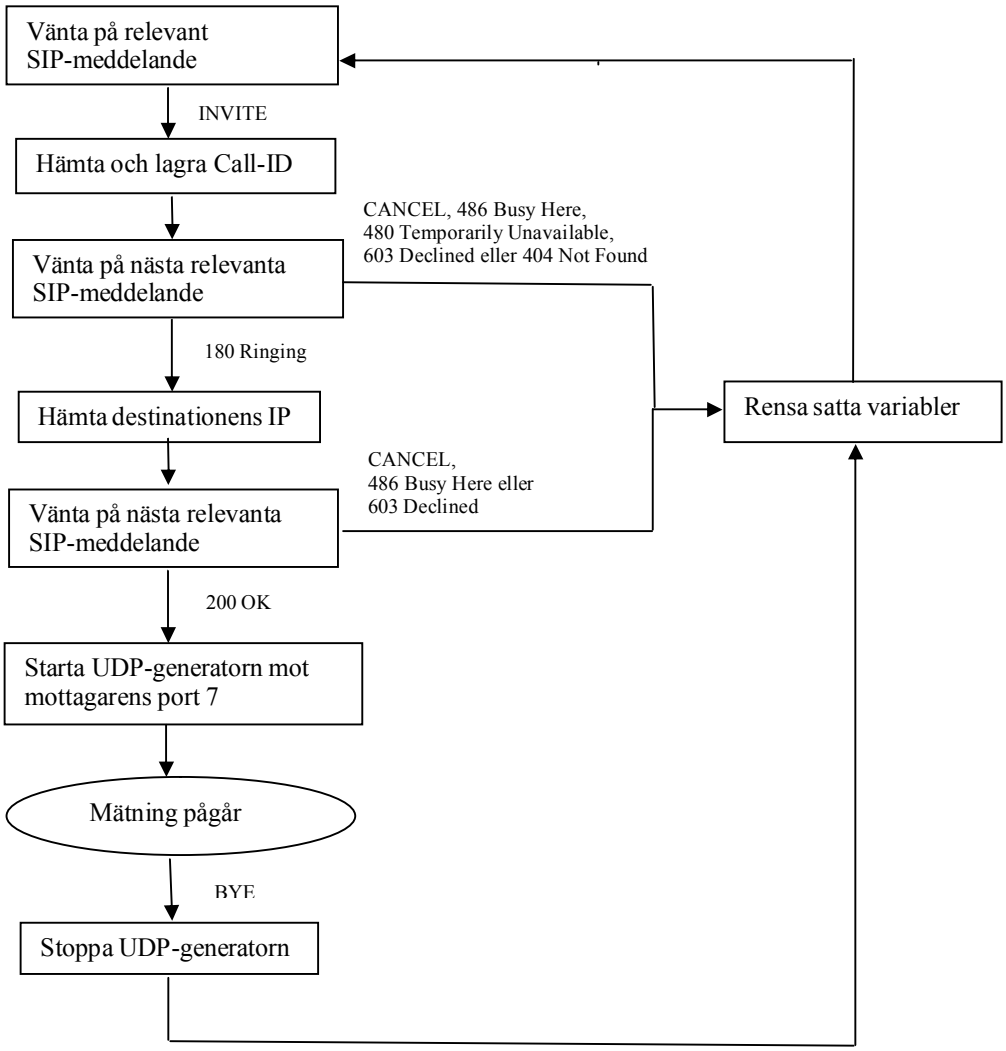


Figur 4.4: Översikt över de program som körs och var de körs vid lösningen då servern gör statistikinsamlingen.

Perlskripten som används i denna lösning bygger i stor del på den föregående lösningens skript. SIP-servern som kör serverversionen av programmet måste vara inställt på ”record-route”, vilket betyder att alla SIP-meddelanden går via servern. Om detta inte är fallet finns en risk att servern inte ser när samtalet som övervakas kopplas ner eftersom all SIP-trafik fr.o.m. ACK-meddelandet på INVITE-transaktionens ”200 OK”-meddelande, skickas mellan klienterna utan SIP-servern som mellanhand (om inte klientprogramvaran är inställd på att köra all SIP-trafik via servern).

4.4.2.1 Klientversionen: sip-nemac-client.pl

Klient versionen av skriptet, som heter *sip-nemac-client.pl*, kör på alla de SIP-klienter som kommer att initiera eventuella samtal. De stora skillnaderna i detta skript jämfört med *sip-nemac.pl* är att hela subrutinen *createconfig()* är borttagen samt att allt som har med NeMaC att göra också är borttaget. Det enda viktiga som detta skript ska göra är att om datorn initierar ett samtal ska den hämta samtalets Call-ID från INVITE-meddelandet som beskrevs ovan samt hämta mottagardatorns IP-nummer från ”180 Ringing”-meddelandet för att sedan starta udpgeneratorn när samtalet svaras samt stänga ner denna när samtalet avslutas. Figur 4.5 visar ett flödesdiagram över vad som händer när de olika SIP-meddelandena kommer in hos uppringardatorn när denna startar upp ett samtal.



Figur 4.5: Flödesdiagram över vad som händer på sändardatorn, när sip-nemac-client.pl används, vid start av ett samtal (kontroller av IP-nummer och om meddelandena tillhör rätt samtal borttaget för att förenkla).

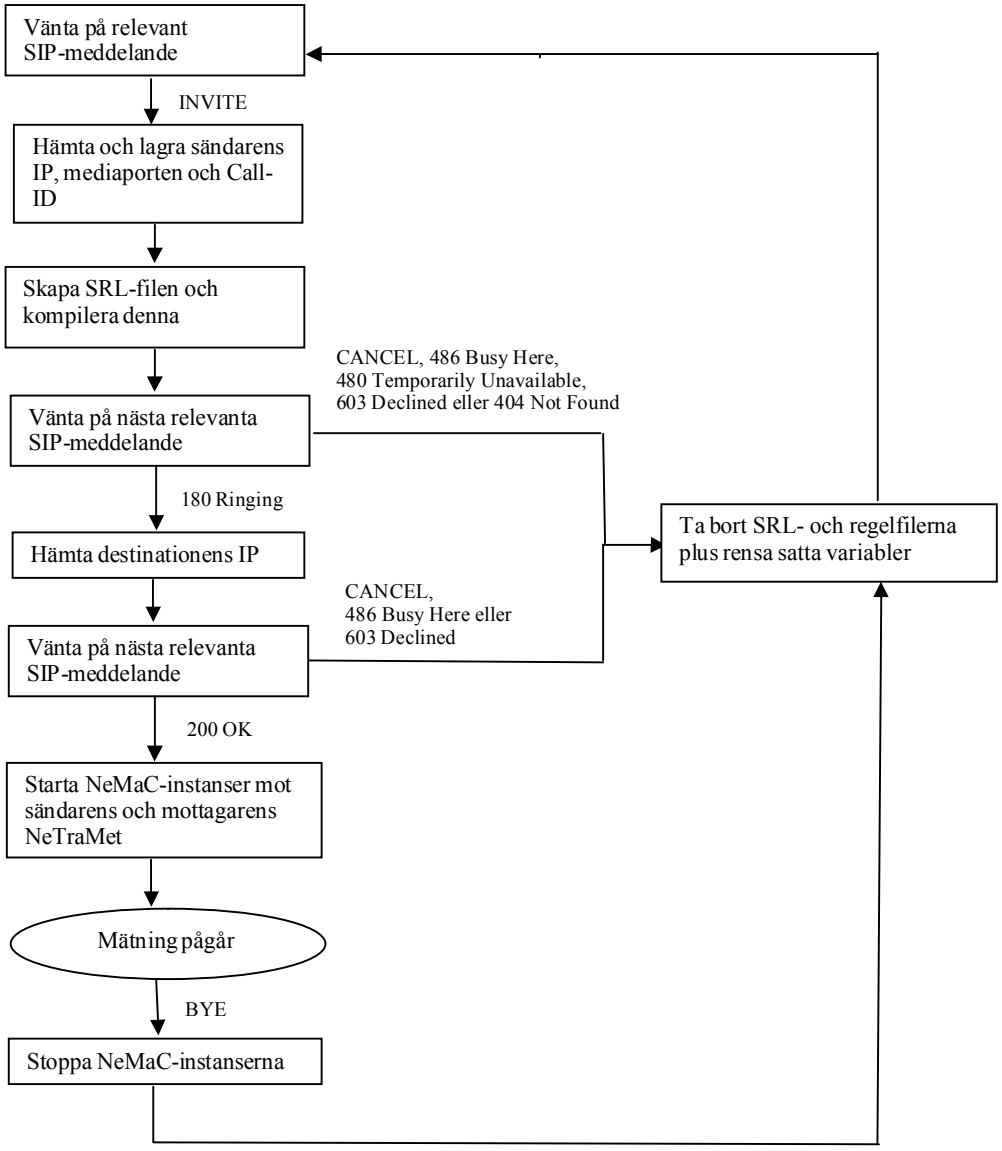
4.4.2.2 Serverversionen: sip-nemac-server.pl

På SIP-servern körs en instans av ett serverskript (*sip-nemac-server.pl*) för varje klientdator vars IP-telefonitrafik ska mätas. Detta program skapar och kompilerar den SRL-fil som behövs samt startar de två instanserna av NeMaC som ska startas om samtalet som initierats svarats.

Precis som med klientskriptet bygger serverskriptet på *sip-nemac.pl*. Här är den stora skillnaden att allt som har med udpgeneratorn att göra är borttaget, såsom uppstarten av den när ”200 OK”-meddelandet kommer in och skalären *$monitorpacketsize*. Ett tillägg i skriptet är att när ”200 OK”-meddelandet för INVITE-transaktionen kommer in till servern och NeMaC ska startas så har en extra kontroll lagts till, tillsammans med kontrollen av Call-ID, att paketets sändar-IP (*$ip_obj->{src_ip}*) är den uppringde partens IP-nummer (*$destinationip*).

```
if (($callid eq $thiscallid) && ($ip_obj->{src_ip} eq $destinationip)) {
      $nemacsrcpid = open(NEMACSRC, "NeMaC -c $collectinterval -g 120 -m 50000 -p -
F $nemacconf.src.$sourceip.flow -L $nemacconf.log -r $nemacconf.rules $sourceip
write|");
      $nemacdstpid = open(NEMACDST, "NeMaC -c $collectinterval -g 120 -m 50000 -p -
F $nemacconf.dst.$destinationip.flow -L $nemacconf.log -r $nemacconf.rules
$destinationip write|");
}
```

Detta görs eftersom SIP-servern ser två ”200 OK”-meddelanden, det som kommer in till SIP-servern från den uppringde parten samt det som skickas ut från SIP-servern till den uppringande parten, och för att skriptet inte ska starta två extra NeMaC-instanser triggas bara starten av NeMaC på det inkommande paketet. Figur 4.6 visar ett flödesdiagram över vad som händer på proxyservern när ett samtal startas när Perlskriptet används.

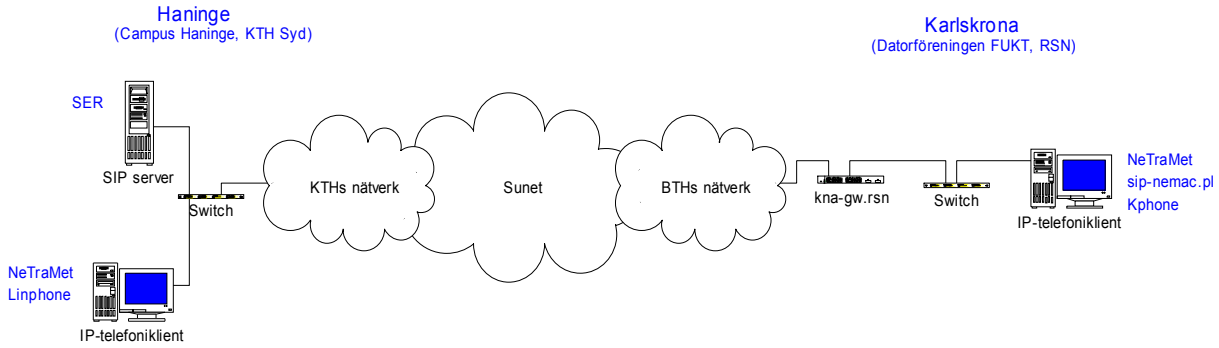


Figur 4.6: Flödesdiagram över vad som händer på proxyservern, när sip-nemac-server.pl används, vid start av ett samtal (kontroller av IP-nummer och om meddelandena tillhör rätt samtal borttaget för att förenkla).

5 Mätning av trafik mellan Karlskrona och Haninge

5.1 Konfiguration av mätningen

Mätningar av trafikkvalitén gjordes mellan Karlskrona och Haninge för att testa den utvecklade prototypen. Mätningen bestod av att ett IP-telefonisamtal kopplades upp av en dator, bilbo, på Datorföreningen FUKT (som har Internetuppkoppling genom Blekinge Tekniska Högskola) i Karlskrona mot en annan dator, 7017ws15, på KTH Syd, Campus Haninge. Den SIP-server som användes vid uppkopplingen, 7017ws16, fanns på samma lokala nätverk som klientdatorn i Haninge. Datorerna i Haninge körde operativsystemet Fedora Core 2 medan den i Karlskrona körde Debian 3.1. Den version av NeTraMet som användes på båda klientdatorerna var 5.1 Beta 11. Skriptet *sip-nemac.pl* kördes på klientdatorn i Karlskrona för att starta NeMaC samt udpgeneratorn *ngen* när denna kopplade upp samtal mot klientdatorn i Haninge. Klientdatorn i Haninge hade echo-porten (port 7) öppen för att kunna skicka tillbaka OAM-paketen som genererats av klientdatorn i Karlskrona. När det gällde IP-telefoniklienten i Karlskrona var denna Kphone medan den i Haninge var Liphone. I figur 5.1 nedan finns en översikt över de datorer och nätverk som var involverade i mätningen.



Figur 5.1: Översikt över datorer och nätverk som användes i mätningen.

Routingen av trafiken mellan orterna var inte helt symmetrisk vilket framgick om traceroute användes från en klient i ena orten till den andra och tvärtom.

Vägen som paketen från Karlskrona till Haninge, 13 hopp, går är:

1. kna-gw.rsn.bth.se (194.47.144.1)
2. kna-rsn-gw.net.bth.se (194.47.128.233)
3. 194.47.128.73 (194.47.128.73)
4. ronneby2-srp2.sunet.se (130.242.83.194)
5. kalmar1-POS0.sunet.se (130.242.81.113)
6. kalmar2-pos1.sunet.se (130.242.81.118)
7. stockholm2-POS3.sunet.se (130.242.81.122)
8. stockholm4-POS0.sunet.se (130.242.82.50)
9. kth2-srp1.sunet.se (130.242.85.68)
10. ea6-kth2-p2p.gw.kth.se (130.237.211.182)
11. cn5-ea6-p2p.gw.kth.se (130.237.211.101)
12. haninge-cn5-p2p.gw.kth.se (130.237.211.122)
13. h4112ws115.haninge.kth.se (193.10.38.115)

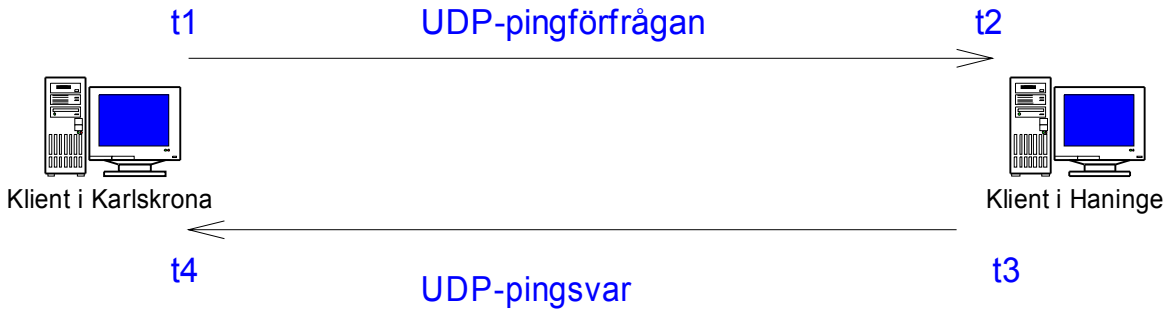
Vägen som paketen från Haninge till Karlskrona, 12 hopp, går är:

1. n83a (130.237.83.1)
2. cn5-haninge-p2p.gw.kth.se (130.237.211.121)
3. kth1-cn5-p2p.gw.kth.se (130.237.211.41)
4. stockholm4-SRP2.sunet.se (130.242.85.66)
5. stockholm2-POS8.sunet.se (130.242.82.49)
6. kalmar2-POS0.sunet.se (130.242.81.121)
7. kalmar1-POS1.sunet.se (130.242.81.117)
8. ronneby2-POS0.sunet.se (130.242.81.114)
9. bth1-SRP1.sunet.se (130.242.83.195)
10. 194.47.128.65 (194.47.128.65)
11. rsn-kna-gw.net.bth.se (194.47.128.234)
12. bilbo.fukt.bth.se (194.47.144.12)

Som framgår av ovanstående listor är vägen i stamnätet detsamma åt båda hållen medan vägen inom högskolenätverken skiljer sig åt till viss del.

Samtalen som mättes initierades av datorn i Karlskrona. Ett skäl till detta var svårigheter att öppna port 7 i Debian och ett annat var att SNMP-trafik ut från Karlskrona var stoppad. IP-telefoniklienterna startades och fönstret för den klient som startade på datorn i Karlskrona vidarebefordrades via X-fönstersystemet till datorn i Haninge. Eftersom *sip-nemac.pl* körde på bilbo startades NeMaC och udpgeneratorn när samtalet kopplades upp samt stoppades när samtalet terminerade där. Samtalet som kopplades upp och som sedan analyserades var ca 5 minuter långt.

Data som ficks av mätningarna sparades av NeMaC i flowfiler. Informationen i flowfilerna som var kopplade till OAM-paketens triggning av mätningen lades in i Excel och hanterades i detta program. Information om ett OAM-paket sparades för när paketet skickades ut från klientdatorn i Karlskrona (t1), kom in till klientdatorn i Haninge (t2), när svaret på OAM-paketet skickades ut (t3) samt när svaret togs emot av klientdatorn i Karlskrona (t4). Figur 5.2 nedan visualiserar detta.



Figur 5.2: De tidsstämplar som fås av de OAM-paket som skickas

I den OAM-data som sparades fanns följande information:

- OAM-paketets 16-bitars ID-värde. Denna information används för att se att OAM-paketen kommit fram i rätt ordning, samt att se vilken OAM-data som tillhör samma OAM-paket. Detta eftersom det finns fyra datainstanser per paket (när paketet skickats och tagits emot samt när svaret skickats och tagits emot)

- Tidstämpel för OAM-paketet. Detta används för att räkna ut tur-och-returtid och processeringstid för OAM-paketen samt vid uträkningen av genomströmningen eftersom OAM-paketen inte helt skickas ut med jämna mellanrum (kan vara en viss skillnad med någon millisekund).
- Antal bytes RTP-trafik som kommit in. Denna information användes för att räkna ut genomströmningen.
- Antal RTP-paket som kommit in. Denna information kan användas för att få en överblick över tappade paket.
- Antal bytes RTP-trafik som skickats ut och
- Antal RTP-paket som skickats ut, dessa två var alltid 0 eftersom denna information ej var definierad. För att se hur mycket trafik som skickats ut finns informationen i OAM-data för paketen som skickats ut.

Det som räknades ut för samtalet var:

- Tiden det tog för ett OAM-paket som skickats att komma tillbaka (RTT).
- Tiden det tog för mottagaren av OAM-paketet att skicka ett svar (processeringstiden).
- En ”exakt” RTT där processeringstiden subtraherades från RTT för att få ett bättre värde på tiden det tar att skicka paket över nätverket.
- Fördröjningsvariationen mellan paketen. Det tittades både på ITU-T:s och IETF:s definitioner av detta.
- RTP-trafikens genomströmning per övervakningsblock.
- Det tittades även på envägsfördröjningen av trafiken, men problem med ”drift” i klientdatorernas klockor (som använde NTP för att synkroniseras) gjorde att dessa värden ej kunde litas på. Det tittades ändå på dessa värden eftersom de fortfarande gav en viss bild över hur fördröjningen var spridd.

5.2 Resultat

Det samtal som kopplades upp och sedan analyserades var på 5:04 minuter och OAM-paketen skickades ut med ett intervall på 1 paket/s vilket gav 304 stycken OAM-paket.

5.2.1 Fördröjning

Fördröjningen räknades ut med hjälp av de tidsstämplar som ficks vid varje OAM-paket, figur 5.2 visar var dessa tider tas. Genom dessa tidstämplars värden går det att få fram både envägsfördröjning och tur-och-returfördröjning (RTT). När det gäller RTT går det att få ett mera verklighetstroget värde (”Exakt” RTT) genom att subtrahera processeringstiden i mottagarnoden. För att få användbara värden när det gäller envägsfördröjningar måste klockorna vara synkroniserade. Eftersom datorerna som användes vid mätningen bara använde NTP för klocksynkronisering är synkroniseringen för det mesta inte tillräckligt bra för beräkning av envägsfördröjning. De formler som används vid uträkning av de olika fördröjningstiderna är:

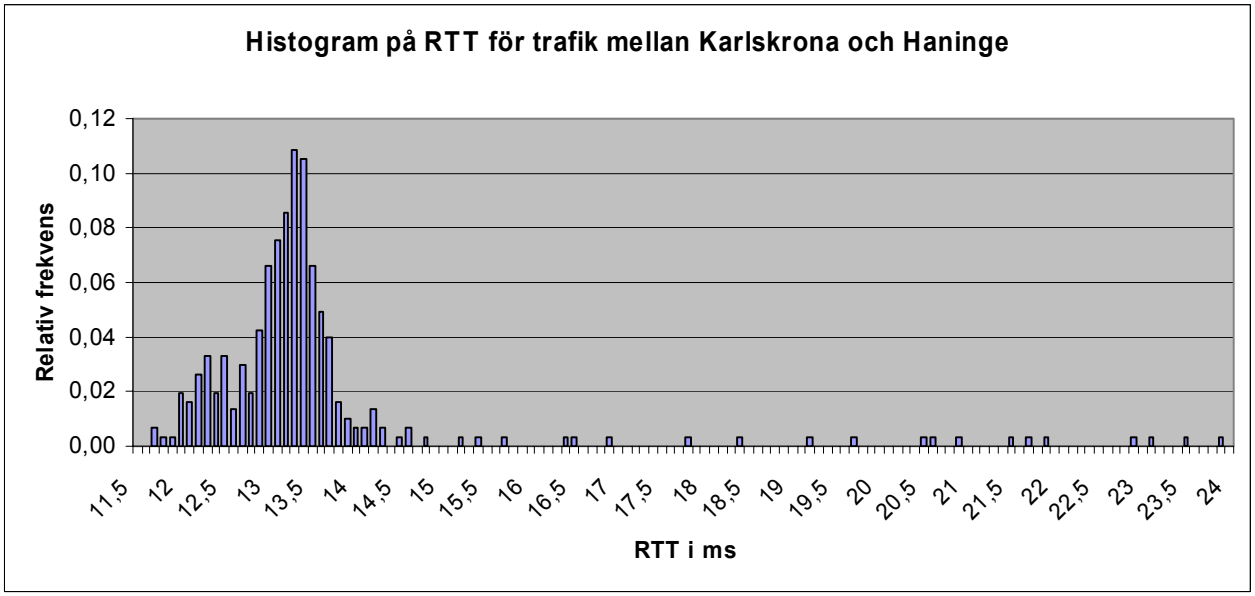
- Envägsfördröjning – sändare -> mottagare = t2-t1
- Envägsfördröjning – mottagare -> sändare = t4-t3
- RTT = t4-t1
- Processeringstid i mottagarnoden = t3-t2
- ”Exakt RTT” = t4-t1-(t3-t2)

Medelvärde, standardavvikelse, maxvärde, 99:de percentilen, 1:a percentilen och minimivärde för RTT, processeringstid samt ”Exakt” RTT redovisas nedan.

	RTT	Processeringstid	”Exakt” RTT
Medelvärde	13,56 ms	0,861 ms	12,699 ms
Standardavvikelse	1,835 ms	1,732 ms	0,533 ms
Maxvärde	23,89 ms	11,179 ms	14,946 ms
99 percentil	22,89 ms	9,611 ms	13,903 ms
1 percentil	11,901ms	0,359 ms	11,441 ms
Minvärde	11,694 ms	0,363 ms	11,266 ms

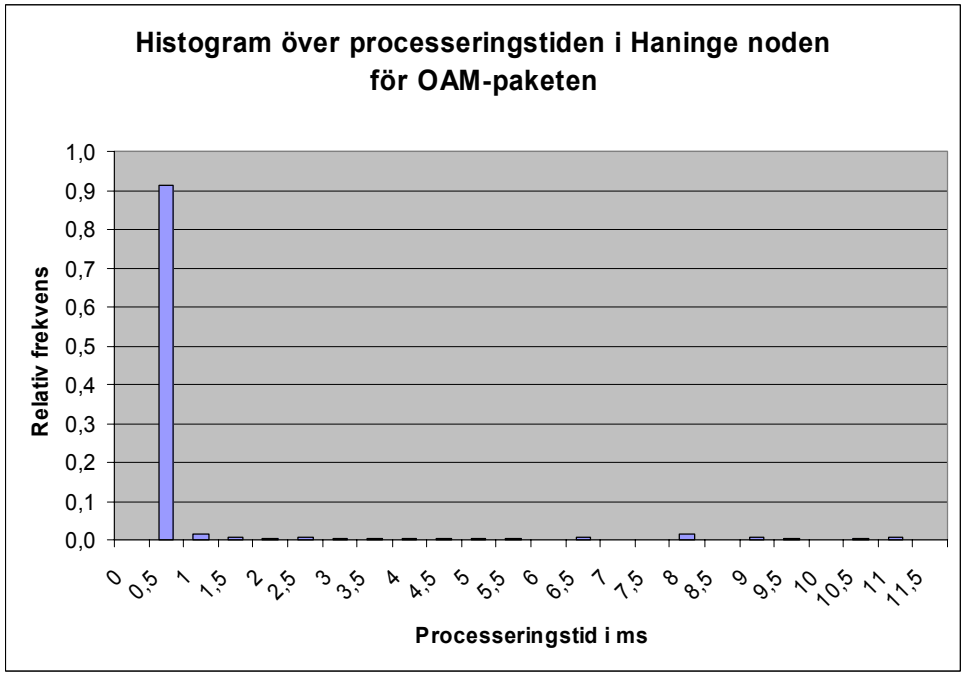
Tabell 5.1: Resultat på de olika fördröjningarna som ficks vid mätningen.

I figurerna 5.3, 5.4 och 5.5 visas histogram på de olika värdena som ficks. Som ses i figur 5.3 ligger det en top på RTT:n runt 13,5 ms men det finns också en liten uppgång runt 12,5 ms till vilket ingen förklaring hittats. Där finns också ett antal värden som är mycket högre än medelvärdet.



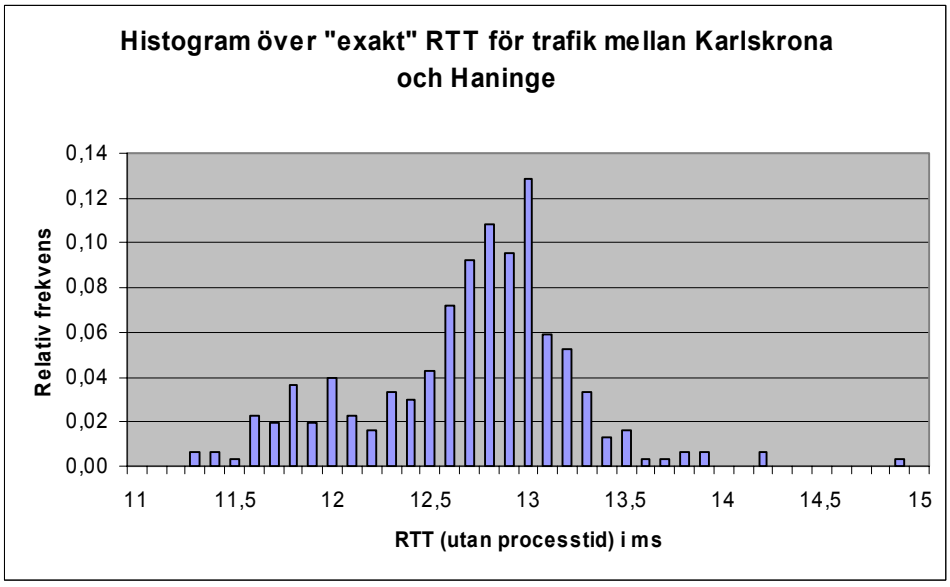
Figur 5.3: Histogram på fördröjningen (RTT) som ficks fram från OAM-paketen i mätningen

I histogrammet i figur 5.4 nedan visas värdena för processeringstiden i Haningenoden. Som ses ligger de flesta värdena på mellan 0,5 och 1 ms men det finns ett fåtal värden som är mycket högre än detta.



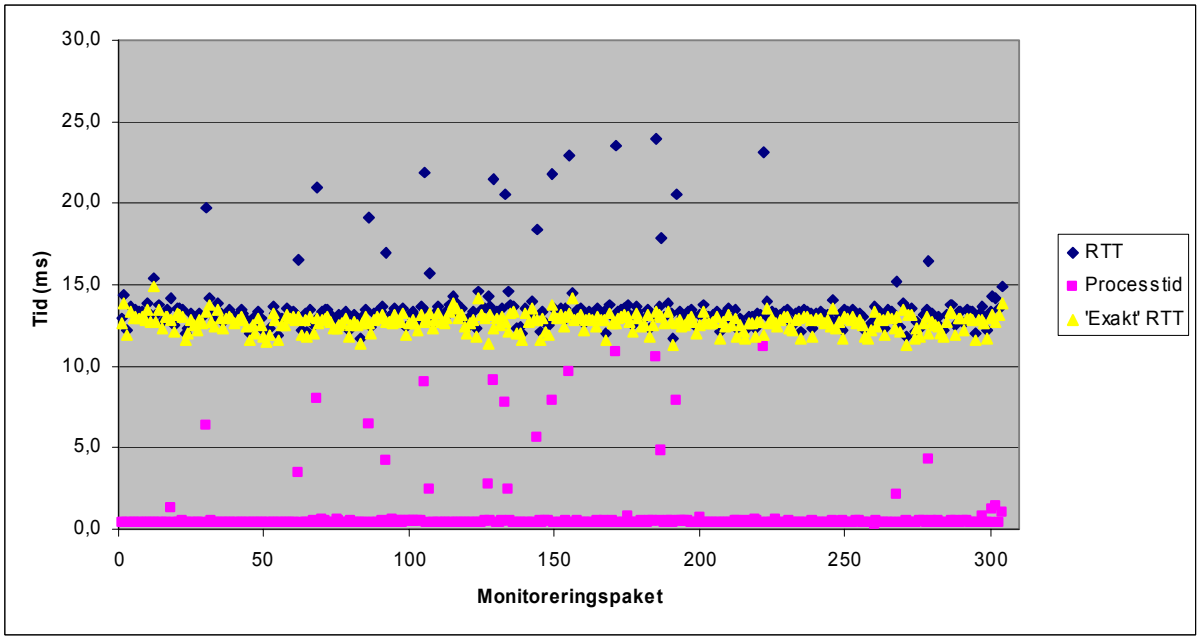
Figur 5.4: Histogram på processeringstiden i Haningenoden som ficks fram från OAM-paketen i mätningen

I figur 5.5 visar histogrammet att när processeringstiden tas bort från RTT:n blir spridningen på värdena mycket mindre. Toppen ligger nu på mellan 12,7 till 13,1 ms men fortfarande finns en annan top mellan 11,7 och 12,2 ms.



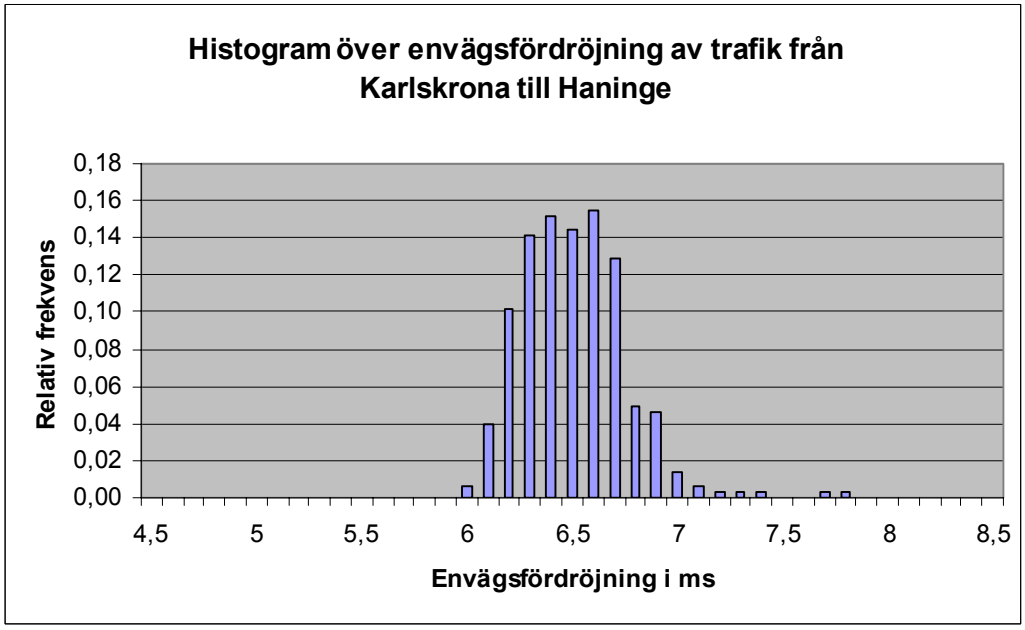
Figur 5.5: Histogram på den ”exakta” fördröjningen (RTT) som ficks fram från OAM-paketen i mätningen.

Som kan ses från de övre histogrammen kan processeringstiden ha en stor påverkan på resultatet. I grafen i figur 5.6 nedan visas RTT, processeringstid och ”Exakt” RTT för de OAM-paket som skickades under mätningen. Som ses i grafen går RTT upp med nästan lika mycket som processeringstiden när denna är högre än normalt.

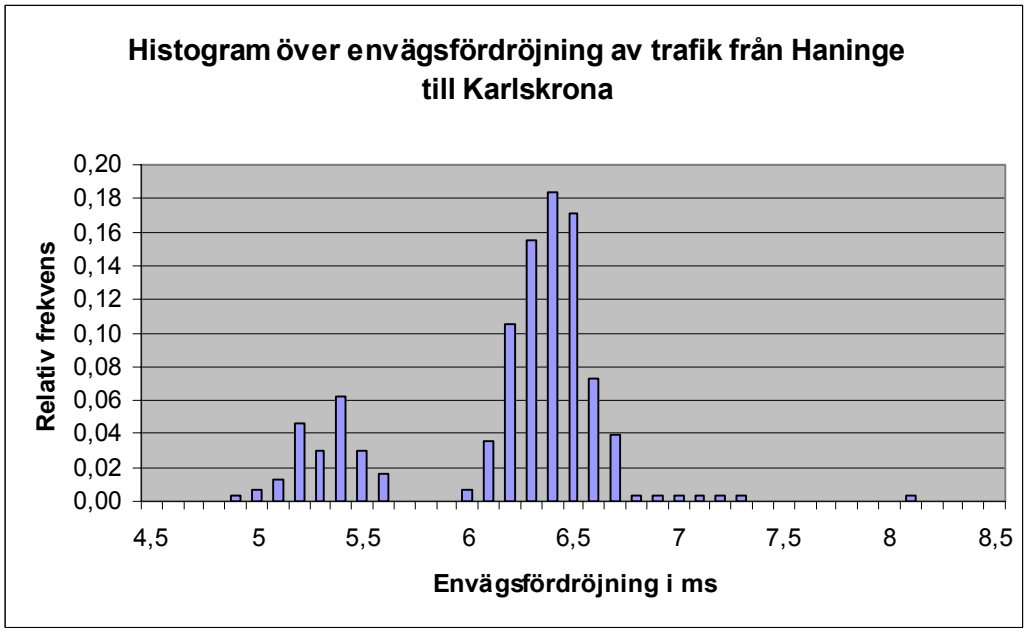


Figur 5.6: Graf över värdena på fördröjning, processeringstid samt "exakt" fördröjning som ficks vid mätningen Som ses har processeringstiden en stor påverkan på fördröjningen.

Eftersom trafiken åt ena hållet inte går riktigt samma väg som den som går åt andra hållet undersöktes även envägsfördröjningen. Dessa histogram finns i figur 5.7 och 5.8.



Figur 5.7: Envägsfördröjning för OAM-paketen från Karlskrona till Haninge



Figur 5.8: Envägsfördröjning för OAM-paketen från Haninge till Karlskrona

Som ses i histogrammen ovan dyker den lilla toppen upp för trafiken som går från Haninge till Karlskrona medan trafiken åt det andra hållet har en mera normal fördelning.

5.2.2 Fördröjningsvariation

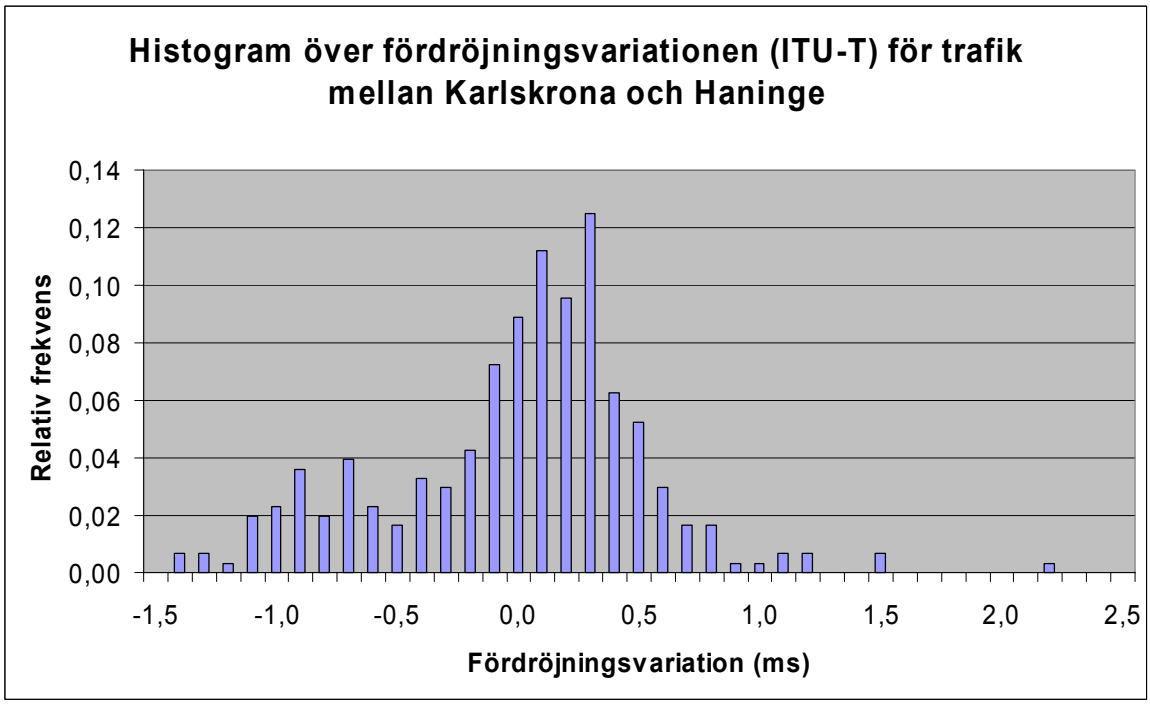
Fördröjningsvariationen som räknades ut bygger på värdena för den ”exakta” RTT:n. Det finns två sätt att räkna ut fördröjningsvariationen ITU-T:s metod och IETF:s metod. ITU-T räknar ut fördröjningsvariationen genom att subtrahera medel- eller minimivärdet för fördröjningen med det nuvarande paketets fördröjning (d(i)-medel(d) eller d(i)-min(d)) [21] medan IETF:s metod består i att förra paketets fördröjning subtraheras från det nuvarande paketets fördröjning (d(i)-d(i-1)) [22].

I tabell 5.2 nedan redovisas medelvärde, standardavvikelsen, maxvärdet, 99:de percentilen, 1:a percentilen och minimivärdet för fördröjningsvariationen (både ITU-T:s (som använder medelvärdet) och IETF:s uträkningsmetoder). Som ses ligger medelvärdet antingen på 0 eller mycket nära 0.

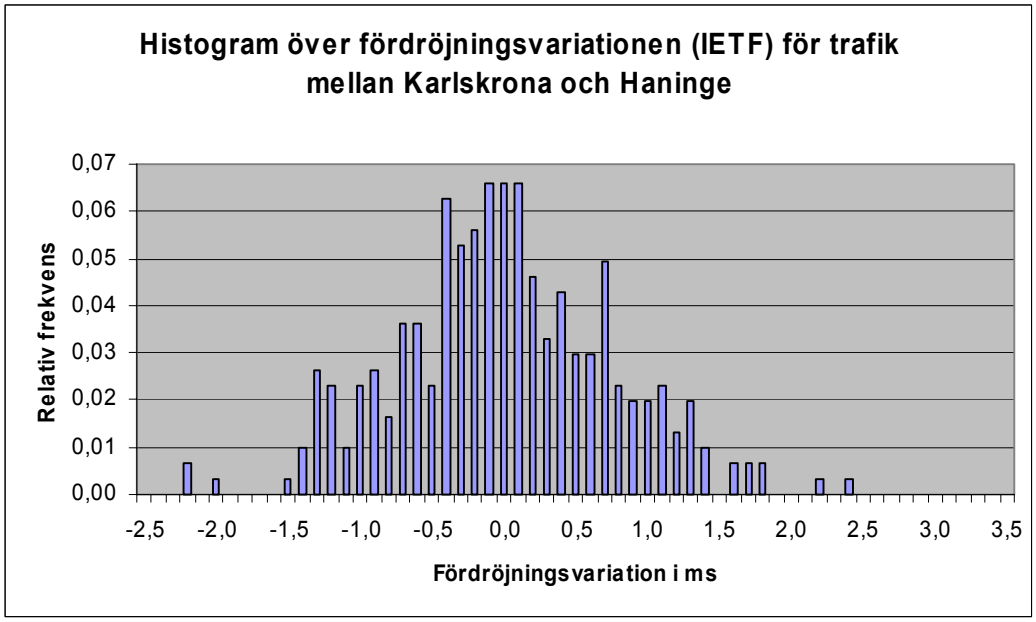
	Fördröjningsvariation, ITU-T	**Fördröjningsvariation, IETF**
Medelvärde	0 ms	0,004 ms
Standardavvikelse	0,533 ms	0,756 ms
Maxvärde	2,248 ms	2,352 ms
99 percentil	1,204 ms	1,833 ms
1 percentil	-1,257 ms	-1,538 ms
Minvärde	-1,433 ms	-2,225 ms

Tabell 5.2: Resultatet på de olika fördröjningsvariationerna som ficks från mätningen.

I figur 5.9 och 5.10 nedan visas histogram på fördröjningsvariationens värden. Som ses har fördröjningsvariationens, som räknas ut med ITU-T:s metod, sannolikhetsfördeling samma utseende som den ”exakta” RTT:ns medan histogrammet för värdena som fåtts från IETF:s metod är med jämnt fördelade runt medelvärdet.



Figur 5.9: Histogram på fördröjningsvariationen (för "exakt" RTT) som ficks fram från OAM-paketen i mätningen, ITU-T:s beräkningsmetod.



Figur 5.10: Histogram på fördröjningsvariationen (för "exakt" RTT) som ficks fram från OAM-paketen i mätningen, IETF:s beräkningsmetod.

5.2.3 Genomströmning

När det gäller genomströmningen undersöktes genomströmning per övervakningsblock. Detta räknas ut genom att ta antalet bitar (bytes * 8) som kommit in under ett övervakningsblock och dela detta med tiden mellan de två OAM-paketen som definierar övervakningsblockets start och slut (tid(i)-tid(i-1)).

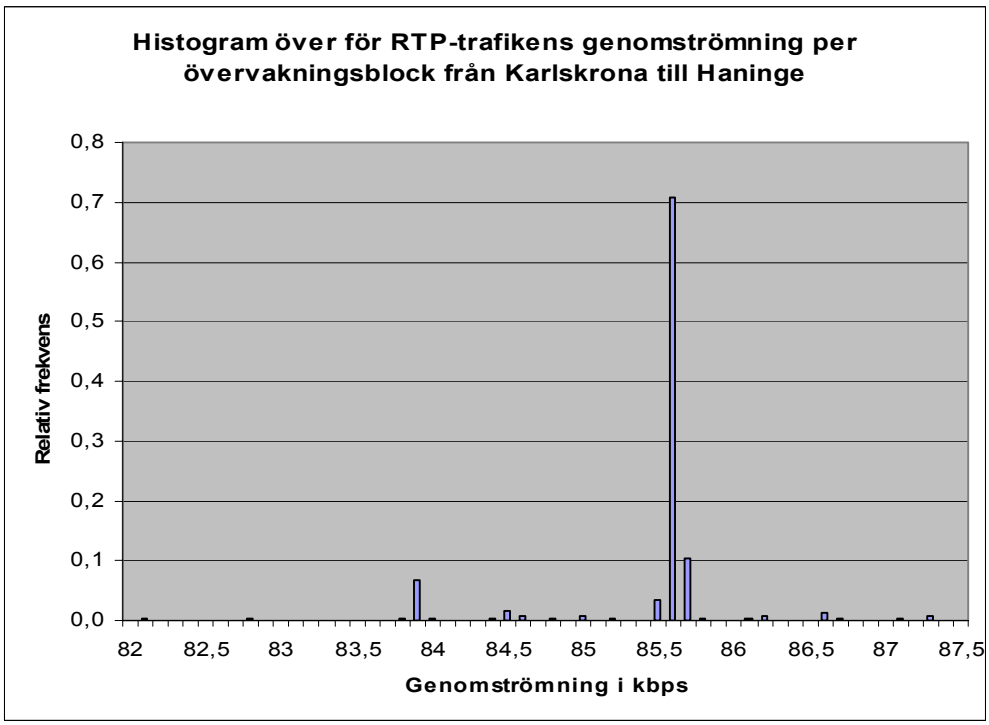
I tabell 5.3 nedan visas medelvärdet, standardavvikelsen, maxvärde, 99:de percentilen, 1:a percentilen och minimivärdet för genomströmningen in till Haninge noden och in till Karlskrona noden.

Genomströmning	Karlskrona -> Haninge	Haninge -> Karlskrona
Medelvärde	85,459 kbps	85,598 kbps
Standardavvikelse	0,585 kbps	1,258 kbps
Maxvärde	87,32 kbps	88,068 kbps
99 percentil	86,669 kbps	87,847 kbps
1 percentil	83,854 kbps	82,938 kbps
Minvärde	82,144 kbps	82,938 kbps

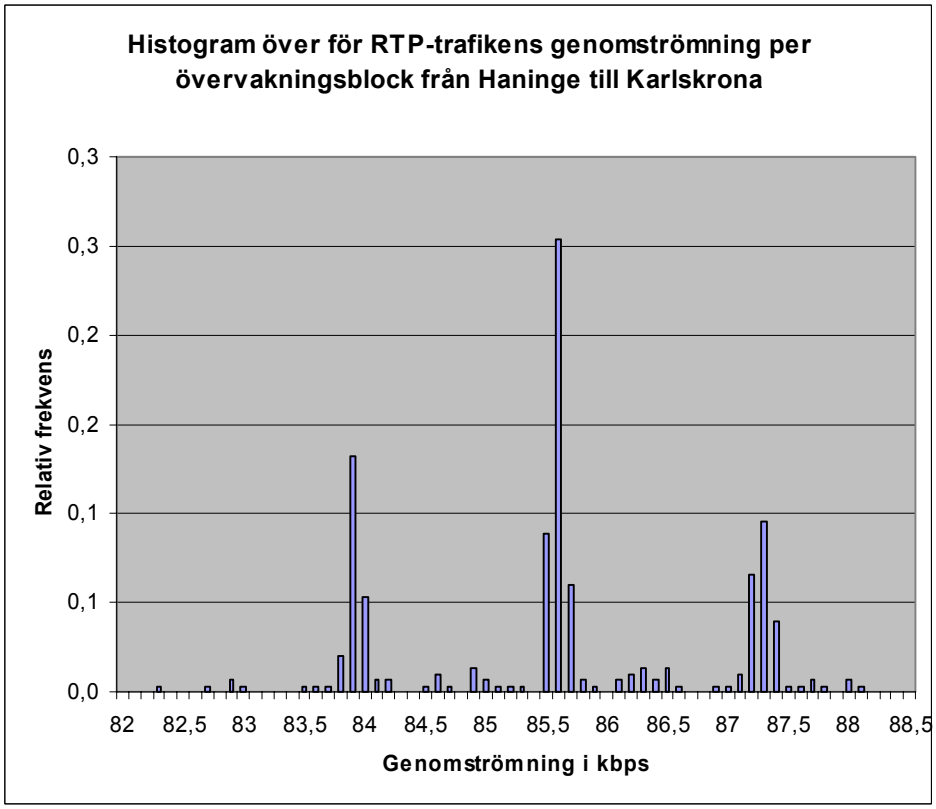
Tabell 5.3: Resultat för genomströmningen som ficks från mätningen.

Som ses är medelvärdena ganska lika (0,14 kbps) åt båda hållen. Skillnaden mellan minimum och maximum för genomströmningen är inte heller så stor (ca 5 kbps).

I figurerna 5.11 och 5.12 nedan visas histogram på genomströmningen. Som ses ligger det för det mesta på 85,6 (50 paket/s) för båda hållen. I genomströmningen för trafiken från Haninge till Karlskrona finns också två toppar på 83,9 (49 paket/s) och 87,3 (51 paket/s). De mindre staplarna i histogrammen dyker upp p.g.a. av att övervakningsblocken inte var precis 1 sekund långa.



Figur 5.11: Histogram på genomströmningen för RTP-paket från Karlskrona till Haninge, som ficks fram från OAM-paketen i mätningen.



Figur 5.12: Histogram på genomströmningen för RTP-paket Haninge till från Karlskrona, som ficks fram från OAM-paketen i mätningen.

5.3 Analys av resultat

När processeringstiden dras bort från RTT:n blir denna jämnare än med tiden i noden. ”Exakt” RTT ligger mellan 11 ms och 15 ms som är ett intervall på 4 ms. Som också ses påverkar processeringstiden RTT mycket. Med processeringstiden i noden ligger RTT på mellan 11,5 ms och 24 ms vid mätningen vilket är ett intervall på 12,5 ms vilket är 3 ggr högre än när processeringstiden tagits bort. Att processeringstiden är ojämn kan bero på att port 7 som är porten som tar emot OAM-paketen för att sedan skicka tillbaka ett svar på dessa troligtvis är nedprioriterat i datorsystemet eftersom det är en echo-port. För att få bättre resultat kan det därför kanske vara en idé att hitta eller utveckla en annan metod att generera och skicka tillbaka OAM-paketen, en metod som inte är nedprioriterad i operativsystemet. En annan orsak till att processeringstiden går upp kan också vara att datorn kör andra program som påverkar processeringstiden, både vanliga program och demoner.

Som ses i histogrammet för den ”exakta” fördröjningen (figur 5.5) ligger de flesta tiderna på runt 12,5 till 13 ms. Det finns också en topp mellan 11,7 och 12,1 ms, denna topp är 1/3 av huvudtoppen. Vad kan denna topp komma ifrån? Genom att titta på envägsfördröjningen i figur 5.8 ses det att denna topp dyker upp på tillbakavägen från Haninge till Karlskrona. Detta kan betyda att något händer på denna del av vägen, som inte är densamma som ditvägen (skillnaderna ligger i BTH- och KTH-delarna av nätverket, vägen genom Sunets stamnät är detsamma för trafiken åt båda hållen mellan Karlskrona och Haninge). Det har tyvärr inte kommit fram några direkta teorier varför detta är fallet. En tanke var att eventuell trafik som sänds vid olika intervall (t.ex. någon sorts mätning) skulle ha påverkat resultatet men i så fall skulle den mindre toppen ha legat efter den stora toppen vilket inte är fallet.

Som histogrammen visar är fördelningen av olika beroende om IETF:s eller ITU-T:s definition används. Fördelningen för fördröjningsvariation enligt ITU-T: definition överensstämmer med den ”exakta” RTT:n, alltså en liten topp före den dominerande toppen. Medan IETF:s fördelning består av bara en topp. Att fördelningarna är olika beror ju på att ITU-T:s definition visar hur värdena skiljer sig mot medelvärdet medan IETFs version visar hur ett värde skiljer sig från förra värdet.

Genomströmningen är relativt jämn, runt 8,2 kbps och 8,8 kbps åt båda hållen. Att den är jämn beror på att RTP skickar små paket (200 bytes) med en konstant hastighet (49-51 paket i sekunden) hela tiden mellan noderna, även om ingen pratar. Det tappas heller inga eller mycket få paket på vägen (under mätningen ovan tappades ett paket) vilket gör att genomströmningen inte heller går ner pga. av detta.

Paketförlusterna var mycket låga, oftast inga förluster alls. P.g.a. av att numreringen av monitoreringspaketen inte var detsamma för det skickade monitoreringspaketet jämfört med svaret på detta paket, samt en viss brist på synkronisering i mätningen, går det heller inte att beräkna antal paket per monitoreringsblock. Mätresultat för tappade paket kan endast erhållas för totalt antal tappade paket för mätperioden som helhet.

6 Diskussion och slutsatser

Serverversionen av prototypen som togs fram i detta arbete kan denna användas av en operatör för att mäta kvaliteten på användarnas IP-telefonitrafik. Antingen kan ett slumpmässigt antal samtal mätas för att kontrollera den allmänna kvaliteten, för att kunna göra detta måste någon sorts lösning tas fram för att göra detta urval. Ett annat sätt är att mäta trafiken för vissa kunder eller vissa samtal kunderna gör. För att operatören ska kunna använda serverversionen av prototypen måste ”record route” (som gör att all SIP-trafik går genom proxyn) vara satt. Om detta inte är fallet kommer inte servern att se när samtalet avslutas eller när samtalets inställningar ändras.

Klientversionen av prototypen kan också användas av en operatör. I detta fall krävs en metod för att hämta information från klienten som utför mätningen, samt att denna skickas till en central lagringsplats.

Metoden att generera monitoreringspaket som användes i projektet bör eventuellt ersättas av en annan metod hur dessa ska genereras och hur mottagaren ska svara. Detta p.g.a. i huvudsak två orsaker:

- När *ngen* används, var lösningen för att få svar på paketen att skicka dem mot port 7 på mottagardatorn. Problemet med port 7 och servicen ECHO som kör där är att, för det första är inte port 7 öppen vanligtvis pga. att denna port är en säkerhetsrisk och för det andra, om porten är öppen, är den vanligtvis nedprioriterad i systemet.
- Det andra problemet var att svarsmeddelandet inte fick samma ID-nummer som paketet som skickades initialt. Detta gör det svårare att analysera resultaten.

Vid mätningarna skickades monitoreringspaketen med en viss tid mellan varandra. En annan möjlighet, när det gäller när monitoreringspaketen ska skickas, är att ett specifikt antal paket skickas mellan varje monitoreringspaket. Men eftersom RTP skickar paket i en jämn ström skulle detta ge ett liknande resultat.

En svårighet med mätningar mellan ändpunkter av fördröjningar där inga precitionsklockor används är noggrannheten och tillförlitligheten av tidsstämplingen i ändpunkterna. När tidsstämpling görs i programmet i stället för på nätverkskortet, när paketet kommer in, kan leda till varierande och ej förutsägbara förseningar som det inte går kan skylla nätverket på. Ur operatörens synvinkel kan kompletterande mätningar mellan kantnoder (edge-to-edge) vara en lösning. Dessa mätningar kan ge en bättre översikt över vad i nätverket som påverkar fördröjningarna.

För synkronisering av tidsstämplingen av monitoreringspaketen användes NTP. Denna metod att synkronisera klockorna har stora svagheter, speciellt då envägsfördröjningar och envägsfördröjningsvariationer mäts. Detta beror på asynkrona förseningar i nätverket vid uppdatering av klockorna, samt drift i klockorna. Eftersom tanken är att mäta kvaliteten av en kunds IP-telefonitrafik är troligtvis GPS heller ingen bra idé eftersom det betyder att extra utrustning måste kopplas in. NTP är fortfarande bra nog då tur-och-retur-fördröjningar mäts och att mäta detta räcker vanligtvis.

En alternativ metod är att mäta RTP-trafikens kvalitet är att använda protokollet RTCP-XR [23]. Detta protokoll använder rapporter från RTCP som är ett protokoll kopplat till RTP. Metoden som utvecklades under detta arbete har fördelen att den inte är beroende av om RTP används eller inte utan kan även användas om ett annat protokoll används för mediaströmmen.

När det gäller detta arbete finns det ett antal saker som kan göras för att förbättra och bygga ut prototypen etc. Några idéer för att förbättra prototypen är:

- Skriva om prototypens källkod så att bara ett program behövs vare sig server- eller klientversionen ska användas. Bästa lösningen för detta är troligtvis definiering av olika flaggor som kan sättas på kommandoraden när programmet startas.
- Utveckla prototypen så att den kan hantera andra mediatyper än audio.
- Utveckla programmet så att den kan ta information från SIP-meddelandena som informerar om missade samtal etc. och använda dessa.
- Utveckla ett program som tar hand om resultaten i flödesfilerna och behandlar och redovisar dessa.
- Hitta ett bättre sätt att generera monitoreringspaketen för att lösa problemen med dessa som nämndes tidigare i detta kapitel.

Referenser

[1] Johnston, Alan B., *SIP – Understanding the Session Initiation Protocol*, Artech House Publishers, Norwood, USA, 2001, ISBN: 1-58053-168-7
[2] J. Rosenberg mfl, *SIP: Session Initiation Protocol*, RFC 3261, IETF, Juni 2002
[3] Jan Janak, *SIP Introduction*, FhG Fokus, 2003
[4] M. Handley, V. Jacobson, *SDP: Session Description Protocol*, RFC 2327, IETF, April 1998
[5] H. Schulzrinne mfl, *RTP: A Transport Protocol for Real-Time Applications,* RFC 3550, IETF, Juli 2003
[6] Fraunhofer Fokus, *iptel.org SIP Server: SIP Express Router*, http://www.iptel.org/ser/ (2006-03-06)
[7] Fraunhofer Fokus, *iptel.org – the IP Telephony Site*, http://www.iptel.org/ (2006-03-06)
[8] J. Kuthan, J. Janak, Y. Rebahi, *iptel.org SIP Express Router v0.11.0 -- Admin's Guide*, FhG Fokus, 2002
[9] The NTP Project (R&D), *Home of the Network Network Time Protocol*, http://www.ntp.org
[10] Trimble Navigation, *All about GPS*, http://www.trimble.com/gps
[11] T. Lindh, Performance *Monitoring in Communication Networks*, Doctoral Thesis, Stockholm, Sweden, 2004
[12] *Ethereal: A Network Protocol Analyzer*, http://www.ethereal.com/
[13] Nevil Brownlee, *NeTraMet - a Network Traffic Flow Measurement Tool*, http://www.caida.org/tools/measurement/netramet/
[14] N. Brownlee, *SRL: A Language for Describing Traffic Flows and Specifying Actions for Flow Groups*, RFC 2723, October 1999
[15] Norbert Vegh, *NTools*, 2004-06-01, http://www.omega.telia.net/ntools/ (2005-12-14)
[16] *Linphone – Telephony on Linux*, http://www.linphone.org/?lang=us&rubrique=1
[17] *Kphone*, http://sourceforge.net/projects/kphone
[18] Nerdlabs Consulting, *Uplog, An UDP based ping program with an "ASCII-graphical" log file*, 2003-06-18, http://www.nerdlabs.org/projects/uplog.php (2005-12-14)
[19] Tim Potter, *Net::PcapUtils - Utility routines for Net::Pcap module*, http://search.cpan.org/~timpotter/Net-PcapUtils-0.01/PcapUtils.pm (2005-12-15)
[20] Tim Potter & Stephanie Wehner, *NetPacket - modules to assemble/disassemble network packets at the protocol level*, http://search.cpan.org/~atrak/NetPacket-0.04/NetPacket.pm (2005-12-15)
[21] *Network Performance Objectives for IP-Based Service*, ITU-T Recommendation Y.1541, February 2002
[22] C. Demichelis, P. Chimento, *IP Packet Delay Variation Metric for IP Performance Metrics (IPPM)*, RFC 3393, November 2002
[23] T. Friedman m.fl, *RTP Control Protocol Extended Reports (RTCP XR)*, RFC 3611, November 2003

Bilaga A: Installationsinformation för SER i Fedora Core 2

1. Installera MySQL 4.0.
   a) Ladda ner rpm-filerna för ”Server”, ”Client” och ”Dynamic client libraries (including 3.23.x libraries)” från http://dev.mysql.com/downloads/mysql/4.0.html.
   b) Installera filerna med *rpm -i*. Antingen alla på en gång eller i ordningen bibliotek, klient och sist server. Följ eventuella hänvisningar som fås (ändring av mysql lösenord för root).
   c) Eventuellt ladda ner MySQL Administrator från http://dev.mysql.com/downloads/administrator/1.1.html. Ladda ner Linux (x86, glibc 2.3) versionen (en tar.gz fil).
   d) Installera MySQL Administrator i */opt*. Stå i denna katalog och kör kommandot: *tar -xzvf /katalog/med/nerladdad/fil/mysql-administrator-1.x.yy-linux.tar.gz*.
2. Installera SIP Express Router.
   a) Ladda ner Linux x86 tar.gz-filen från ftp://ftp.berlios.de/pub/ser/latest/bin/.
   b) Gå till rooten (katalogen ”/”) och packa upp den nerladdade filen: *tar -xvzf /katalog/med/nerladdad/fil/ser-0.x.y_linux_i386.tar.gz*.
   c) Öppna upp filen */usr/local/etc/ser/ser.cfg* i en editor och gör ändringar för att få SER att använda MySQL m.m. (se konfigureringsfilen i Bilaga B).
   d) Skapa en databas för SER med kommandot *ser_mysql.sh create*, Realm är datorns namn (t.ex. 7017ws16.haninge.kth.se).
   e) Öppna porten 5060 i brandväggen så att det går att komma åt servern utifrån: *Fedorameny -> Systeminställningar -> Säkerhetsnivå*. Under andra portar lägg till 5060:udp (Om där redan finns portar listade använd ”,” som separerare.).
   f) Starta SER demonen med kommandot *serctl start*.
3. Installera webserver.
   a) I *Fedorameny -> Systeminställningar -> Lägg till/ta bort program* markera boxen brevid webserver och tryck på ”Detaljer”.
   b) Markera boxarna brevid *mod_auth_mysql* och *php_mysql*. Stäng fönstret.
   c) Uppdatera.
4. Installera serweb.
   a) Ladda ner serweb från ftp://ftp.berlios.de/pub/ser/latest/contrib/.
   b) Gå till katalogen */var/www* och packa upp Serweb: *tar -xvzf /katalog/med/nerladdad/fil/serweb-0.x.y.tgz*.
   c) I */etc/http/conf/httpd.conf* lägg till ett alias ”Alias /serweb/ ”/var/www/serweb/html/” under Aliases.
   d) Editera *php.ini* och se till att följade värde är satt: *short_open_tag = on*
   e) Testa serweb genom att i en webbläsare gå till *http://<värddator>/serweb/*, <värddator> ska vara samma datornamn som satts som Realm när SER installerades.
   f) Logga in som admin. Lösenordet är default-lösenordet heslo.
   g) Om detta fungerar installera eventuella crontabs
      1. Öppna crontab för editering med *crontab –e*.
      2. För SIP-serverövervakningsinformation på webbsidan lägg till raden:
         * * * * * */usr/bin/php /var/www/serweb/scripts/cron_job/read_ser_moni.php*
      3. Avsluta editorn, detta installerar crontaben.

Bilaga B: SER konfigurationsfil (/usr/local/etc/ser.cfg)

```
#
# $Id: ser.cfg,v 1.25.2.1 2005/02/18 14:30:44 andrei Exp $
#
# simple quick-start config script
#

# ----------- global configuration parameters ------------------------

#debug=3         # debug level (cmd line: -dddddddddd)
#fork=yes
#log_stderror=no  # (cmd line: -E)

/* Uncomment these lines to enter debugging mode
fork=no
log_stderror=yes
*/

listen="7017ws16.haninge.kth.se"
check_via=no      # (cmd. line: -v)
dns=yes           # (cmd. line: -r)
rev_dns=no      # (cmd. line: -R)
port=5060
#children=4
fifo="/tmp/ser_fifo"
fifo_db_url="mysql://ser:heslo@localhost/ser"
fifo_mode=0666

# ------------------ module loading ----------------------------------

# Uncomment this if you want to use SQL database
loadmodule "/usr/local/lib/ser/modules/mysql.so"

loadmodule "/usr/local/lib/ser/modules/sl.so"
loadmodule "/usr/local/lib/ser/modules/tm.so"
loadmodule "/usr/local/lib/ser/modules/rr.so"
loadmodule "/usr/local/lib/ser/modules/maxfwd.so"
loadmodule "/usr/local/lib/ser/modules/usrloc.so"
loadmodule "/usr/local/lib/ser/modules/registrar.so"
loadmodule "/usr/local/lib/ser/modules/textops.so"
loadmodule "/usr/local/lib/ser/modules/uri_db.so"
loadmodule "/usr/local/lib/ser/modules/options.so"
loadmodule "/usr/local/lib/ser/modules/msilo.so"

# Uncomment this if you want digest authentication
# mysql.so must be loaded !
loadmodule "/usr/local/lib/ser/modules/auth.so"
loadmodule "/usr/local/lib/ser/modules/auth_db.so"
# ----------------- setting module-specific parameters ---------------

# -- usrloc params --

#modparam("usrloc", "db_mode",   0)

# Uncomment this if you want to use SQL database
# for persistent storage and comment the previous line
modparam("usrloc", "db_mode", 2)
```

```
modparam("usrloc", "use_domain", 1)

# -- auth params --
# Uncomment if you are using auth module
#
modparam("auth_db|uri_db|usrloc", "db_url", "mysql://ser:heslo@localhost/ser")
modparam("auth_db", "calculate_ha1", yes)

#
# If you set "calculate_ha1" parameter to yes (which true in this config),
# uncomment also the following parameter)
#
modparam("auth_db", "password_column", "password")

# -- rr params --
# add value to ;lr param to make some broken UAs happy
modparam("rr", "enable_full_lr", 1)

# -- option params --
modparam("options", "accept", "application/*")
modparam("options", "accept_encoding", "")
modparam("options", "accept_language", "en")
modparam("options", "support", "")

#För message store
#modparam("msilo", "db_url", "mysql://ser:heslo@localhost/ser")
#modparam("msilo", "registrar", "sip:admin@7017ws16.haninge.kth.se")
#modparam("msilo", "db_table", "silo")

#Registrar
modparam("registrar", "use_domain", 1)
modparam("registrar", "max_contacts", 1)

# -------------------------  request routing logic -------------------
# main routing logic

route{

      # initial sanity checks -- messages with
      # max_forwards==0, or excessively long requests

      if (!mf_process_maxfwd_header("10")) {
            sl_send_reply("483","Too Many Hops");
            break;
      };
      if (msg:len >=  2048 ) {
            sl_send_reply("513", "Message too big");
            break;
      };

      # we record-route all messages -- to make sure that
      # subsequent messages will go through our proxy; that's
      # particularly good if upstream and downstream entities
      # use different transport protocol
      if (!method=="REGISTER") record_route();

      # subsequent messages withing a dialog should take the
      # path determined by record-routing
      if (loose_route()) {
```

```
            # mark routing logic in request
            append_hf("P-hint: rr-enforced\r\n");
            route(1);
            break;
      };

      if (!uri==myself) {
            # mark routing logic in request
            append_hf("P-hint: outbound\r\n");
            route(1);
            break;
      };

      # if the request is for other domain use UsrLoc
      # (in case, it does not work, use the following command
      # with proper names and addresses in it)
      if (uri==myself) {

            if (method=="REGISTER") {

# Uncomment this if you want to use digest authentication
            if (!www_authorize("7017ws16.haninge.kth.se", "subscriber")) {
                        www_challenge("7017ws16.haninge.kth.se", "0");
                        break;
                  };

                  save("location");
                  break;
            };

            if ((method==OPTIONS) && (! uri=~"sip:.*[@]+.*")) {
                  options_reply();
                 };

            lookup("aliases");
            if (!uri==myself) {
                  append_hf("P-hint: outbound alias\r\n");
                  route(1);
                  break;
            };

            # native SIP destinations are handled using our USRLOC DB
            if (!lookup("location")) {
                  sl_send_reply("404", "Not Found");
                  break;
            };
      };
      append_hf("P-hint: usrloc applied\r\n");
      route(1);
}
route[1]
{
      # send it out now; use stateful forwarding as it works reliably
      # even for UDP2TCP
      if (!t_relay()) {
            sl_reply_error();
      };
}
```

Bilaga C: Exempel SRL-fil

```
define OAM_SENDER = 194.47.144.12;
define OAM_PROTOCOL = udp;
define OAM_SENDER_PORT = 8888;
define DATA_SENDER_IP = 194.47.144.12;
define DATA_SENDER_PORT = 5000;

if SourcePeerType == IPv4 save;
 else ignore;

if SourcePeerAddress == OAM_SENDER
&& SourceTransType == OAM_PROTOCOL
&& SourceTransAddress == OAM_SENDER_PORT
&& MatchingStoD == 1  {
      store FlowKind := 11;
      save SourcePeerAddress = 1.0/8;
      save OAMdata = 1.0.0!0 & 0.0.0!0;
      store OAMident := 1;
      count;
} else if DestTransType == OAM_PROTOCOL
&& DestTransAddress == OAM_SENDER_PORT
&& DestPeerAddress == OAM_SENDER
&& MatchingStoD == 1  {
      store FlowKind := 22;
      save SourcePeerAddress = 2.0/8;
      save OAMdata = 2.0.0!0 & 0.0.0!0;
      store OAMident := 2;
      count;
} else if SourceTransType == UDP
&& SourcePeerAddress == DATA_SENDER_IP
&& SourceTransAddress == DATA_SENDER_PORT save, {
      store FlowKind := 97;
      save SourceTransAddress;
      save DestTransAddress;
      save SourcePeerAddress;
      save DestPeerAddress;
      store OAMident := 1;
      count;
} else if DestTransType == UDP
&& DestPeerAddress == DATA_SENDER_IP
&& DestTransAddress == DATA_SENDER_PORT save, {
      store FlowKind := 98;
      save SourceTransAddress;
      save DestTransAddress;
      save SourcePeerAddress;
      save DestPeerAddress;
      store OAMident := 2;
      count;
} else ignore;

set rtp_test;
format
SourceKind DestKind FlowKind "  "
SourcePeerType SourcePeerAddress DestPeerAddress "  "
SourceTransType SourceTransAddress DestTransAddress
ToPDUs FromPDUs "  " ToOctets FromOctets "{{" OAMdata "}}";
```

Bilaga D: Prototypen

D.1 sip-nemac.pl

```
#!/usr/bin/perl -w
#
# sip-nemac.pl
# Emma Roos <hdm04ero@syd.kth.se>
#
# Program that scans the network for SIP messages and then uses
# information from those to configure and start up NeMaC to record
# traffic statistics from the different calls that are started.
#
# The recording of the traffic statistics does only start if the ip
# sent as an attribute to this program is the ip of the computer that
# starts the call.
#
# Start the program ./sip-nemac.pl <ip number> where the ip number is
# is the one for the computer the program is running on.
#
# *Don't forget to start up NeTraMet before you start this program.*
# *       (NeTraMet -i eth0 -rread -wwrite -r public -m 50000)     *
#

use Net::PcapUtils;
use NetPacket::Ethernet qw(:strip);
use NetPacket::IP;
use NetPacket::UDP;


my $monitorpacketinterval = 1; # The time between monitoring packets
                               # (in seconds).
my $monitorpacketsize = 200; # The packet size (IP) of the monitoring packets
                             # (in bytes). The size should be the same as the
                             # size of the RTP packets.
my $collectinterval = 30; # The interval between NeMaC collecting statistics

my $myip = shift || "none";
my $sourceip;
my $sourceport;
my $destinationip;
my $nemacconf;
my $callid = "none";

#
# Function to create the srl file with the current calls settings
#

sub createconf {

    my $nemacsrl = "$nemacconf.srl";

    #Create the SRL file

    open CONFFILE,'>', $nemacsrl;

    print CONFFILE "define OAM_SENDER = ", $sourceip, ";\n";
```

```
    print CONFFILE "define OAM_PROTOCOL = udp;\n";
    print CONFFILE "define OAM_SENDER_PORT = 8888;\n";
    print CONFFILE "define DATA_SENDER_IP = ", $sourceip, ";\n";
    print CONFFILE "define DATA_SENDER_PORT = ", $sourceport, ";\n\n";

    print CONFFILE "if SourcePeerType == IPv4 save;\n else ignore;\n\n";

    print CONFFILE "if SourcePeerAddress == OAM_SENDER\n";
    print CONFFILE "&& SourceTransType == OAM_PROTOCOL\n";
    print CONFFILE "&& SourceTransAddress == OAM_SENDER_PORT\n";
    print CONFFILE "&& MatchingStoD == 1  {\n";
    print CONFFILE "store FlowKind := 11;\n";
    print CONFFILE "save SourcePeerAddress = 1.0/8;\n";
    print CONFFILE "save OAMdata = 1.0.0!0 & 0.0.0!0;\n";
    print CONFFILE "store OAMident := 1;\n";
    print CONFFILE "count;\n }\n\n";

    print CONFFILE "else if DestTransType == OAM_PROTOCOL\n";
    print CONFFILE "&& DestTransAddress == OAM_SENDER_PORT\n";
    print CONFFILE "&& DestPeerAddress == OAM_SENDER\n";
    print CONFFILE "&& MatchingStoD == 1  {\n";
    print CONFFILE "store FlowKind := 22;\n";
    print CONFFILE "save SourcePeerAddress = 2.0/8;\n";
    print CONFFILE "save OAMdata = 2.0.0!0 & 0.0.0!0;\n";
    print CONFFILE "store OAMident := 2;\n";
    print CONFFILE "count;\n }\n";

    print CONFFILE "else if SourceTransType == UDP\n";
    print CONFFILE "&& SourcePeerAddress == DATA_SENDER_IP\n";
    print CONFFILE "&& SourceTransAddress == DATA_SENDER_PORT save, {\n";
    print CONFFILE "store FlowKind := 97;\n";
    print CONFFILE "save SourceTransAddress; \n";
    print CONFFILE "save DestTransAddress;  \n";
    print CONFFILE "save SourcePeerAddress;\n";
    print CONFFILE "save DestPeerAddress;\n";
    print CONFFILE "store OAMident := 1;\n";
    print CONFFILE "count; \n}\n";

    print CONFFILE "else if DestTransType == UDP\n";
    print CONFFILE "&& DestPeerAddress == DATA_SENDER_IP\n";
    print CONFFILE "&& DestTransAddress == DATA_SENDER_PORT save, {\n";
    print CONFFILE "store FlowKind := 98;\n";
    print CONFFILE "save SourceTransAddress;\n";
    print CONFFILE "save DestTransAddress;  \n";
    print CONFFILE "save SourcePeerAddress;\n";
    print CONFFILE "save DestPeerAddress;\n";
    print CONFFILE "store OAMident := 2;\n";
    print CONFFILE "count;\n }\n";

    print CONFFILE "else ignore;\n";

    print CONFFILE "set rtp_test;\n";
    print CONFFILE "format\n";
    print CONFFILE "SourceKind DestKind FlowKind \"  \"\n";
    print CONFFILE "SourcePeerType SourcePeerAddress DestPeerAddress \"  \"\n";
    print CONFFILE "SourceTransType SourceTransAddress DestTransAddress\n";
    print CONFFILE "ToPDUs FromPDUs \"  \" ToOctets FromOctets \"{{\" OAMdata
\"}}\";\n";
```

```
    close CONFFILE;

    #Compile the SRL file
    system ("/usr/local/bin/srl", $nemacsrl);
};

#
# Function that handles the incoming SIP messages
#

sub siphandler {
    print("$ip_obj->{src_ip}:$udp_obj->{src_port} -> ",
        "$ip_obj->{dest_ip}:$udp_obj->{dest_port} ",
        "$udp_obj->{len}\n");
    #
    # INVITE (Get the info needed for the NeMaC config file and create it)
    #
    if ($udp_obj->{data} =~ /^INVITE/){
      print "Incoming INVITE ";
      if ($ip_obj->{src_ip} eq $myip){
          if ($udp_obj->{data} =~ /Contact:
([^@]+)@(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/){
            my $tempip = "$2";
            if ($tempip eq $myip) {
                print "from this computer.\n";
                $callid = get_callid();
                if ($callid =~ /([^@]+)@(.+)/) {
                  $nemacconf = "$1";
                } else {
                  $nemacconf = "$callid";
                }
                $sourceip = $tempip;
                print "The callers IP: ", $sourceip, "\n";
                if ($udp_obj->{data} =~ /(m=audio )(\d+)/)
                {
                  $sourceport ="$2";
                  print "The callers RTP-port: ", $sourceport, "\n\n";
                }
                createconf;
            } else {
                print "from another computer (this is the SIP server).\n";
                get_callid();
            }
          }
      } else {
          print "from another computer.\n";
          get_callid();
      }
    }
    #
    # 180 Ringing
    #
    elsif ($udp_obj->{data} =~ /^SIP.2.0 180/) {
      print "It's ringing at the callee.\n";
      my $thiscallid = get_callid();
      if ($callid eq $thiscallid) {
          if ($udp_obj->{data} =~ /(Contact:
)([^@]+)@(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/){
            $destinationip = "$3";
```

```
            print "Mottagarens IP: ", $destinationip, "\n";
          }
      }
    }
    #
    # 200 OK
    #
    elsif ($udp_obj->{data} =~ /^SIP.2.0 200/) {
        # The called party has "picked up". (Start NeMac and the UDP generator)
      if ($udp_obj->{data} =~ /CSeq: \d+ INVITE/) {
          print "Conversation has started.\n";
          my $thiscallid = get_callid();
          my $udpgenrate = ($monitorpacketsize * 8)/$monitorpacketinterval;
          if ($callid eq $thiscallid) {
            $nemacsrcpid = open(NEMACSRC, "NeMaC -c $collectinterval -g 120 -m
50000 -p -F $nemacconf.src.$sourceip.flows -L $nemacconf.log -r $nemacconf.rules
$sourceip write|");
            $nemacdstpid = open(NEMACDST, "NeMaC -c $collectinterval -g 120 -m
50000 -p -F $nemacconf.dst.$destinationip.flows -L $nemacconf.log -r
$nemacconf.rules $destinationip write|");
            $udpgenpid = open(UDPGEN, "ngen -proto udp -dstport 7 -rate
$udpgenrate -size $monitorpacketsize -dstip $destinationip|");
          }
      }
        # The call had ended (message comes after the BYE message
      elsif ($udp_obj->{data} =~ /CSeq: \d+ BYE/) {
          print "The conversation is ending.\n";
          my $thiscallid = get_callid();
          if ($callid eq $thiscallid) {
            # This is where NeMaC and the udp generator should stop.
            # But because of a bug in Linphone I've moved this to the
            # handling of BYE message that initiate the tear down of the call.
          }
      }
      # Registering with server was successful
      elsif ($udp_obj->{data} =~ /CSeq: \d+ REGISTER/) {
          if ($udp_obj->{data} =~ /Expires: 0/) {
            print "User unregistered from server!\n";
          } else {
            print "User registered with server!\n";
          }
      }
      elsif ($udp_obj->{data} =~ /CSeq: \d+ CANCEL/) {
          print "Call cancelled successfully.\n";
          get_callid();
      } else {
          print "Another 200 OK\n";
      }
    }
    # The call is over. (Stop NeMaC and the UDP generator)
    elsif ($udp_obj->{data} =~ /^BYE/){
      print "BYE.\n";
      my $thiscallid = get_callid();
      if ($callid eq $thiscallid) {
          `kill $nemacsrcpid $nemacdstpid $udpgenpid`;
          close(NEMACSRC);
          close(NEMACDST);
          close(UDPGEN);
```

```
      $callid = "none";
      system ("rm -f $nemacconf.srl $nemacconf.rules");
  }
}
# 100 Trying
elsif ($udp_obj->{data} =~ /^SIP.2.0 100/){
  print "100 Trying.\n";
  get_callid();
}
# The called party doesn't want to or can't answer
elsif ($udp_obj->{data} =~ /^SIP.2.0 (486|480|603)/)
{
  print "The recipient is unaccessible/busy.\n";
  my $thiscallid = get_callid();
  if ($callid eq $thiscallid) {
      system ("rm -f $nemacconf.srl $nemacconf.rules");
      $callid = "none";
  }
}
# The caller "hangs up" before the call is established.
elsif ($udp_obj->{data} =~ /(^CANCEL)/)
{
  print "Call is cancelled by the caller before conversation had started.\n";
  my $thiscallid = get_callid();
  if ($callid eq $thiscallid) {
      system ("rm -f $nemacconf.srl $nemacconf.rules");
      $callid = "none";
  }
}
# 487 Request Cancelled
elsif ($udp_obj->{data} =~ /^SIP.2.0 487/) {
  print "487 Request Cancelled\n";
}
# ACK messages
elsif ($udp_obj->{data} =~ /^ACK/) {
  print "ACK.\n";
  get_callid();
}
# REGISTER messages
elsif ($udp_obj->{data} =~ /^REGISTER/) {
  if ($udp_obj->{data} =~ /Expires: 0/)
  {
      if ($udp_obj->{data} =~ /To\: \<sip\:([^\>]+)\>/){
        print "User $1 has closed their client program.\n";
      }

  } else {
      if ($udp_obj->{data} =~ /To\: \<sip\:([^\>]+)\>/) {
        print "User $1 is registering with server.\n";
      }
  }
}
# 401 Unauthorized
elsif ($udp_obj->{data} =~ /^SIP.2.0 (401)/) {
  if ($udp_obj->{data} =~ /CSeq: \d+ REGISTER/) {
      print "Password needed for registering with server!\n";
  } else {
      print "401 Unauthorized\n";
  }
```

```
    }
    # 404 Not Found
    elsif ($udp_obj->{data} =~ /^SIP.2.0 (404)/) {
      if ($udp_obj->{data} =~ /CSeq: \d+ REGISTER/) {
          print "User not found in server database!\n";
      } elsif ($udp_obj->{data} =~ /CSeq: \d+ INVITE/) {
          print "User not registered with server/Doesn't exist on server!\n";
          my $thiscallid = get_callid();
          if ($callid eq $thiscallid) {
            system ("rm -f $nemacconf.srl $nemacconf.rules");
            $callid = "none";
          }
      } else {
          print "404 Not Found\n";
      }
    }
    elsif ($udp_obj->{data} =~ /^SIP.2.0 (101)/) {
      print "101 Dialog Establishment\n";
    }
    # Other SIP messages
    else {
      print "$udp_obj->{data}\n";
    }
    print "\n**********************************************************\n\n";
}

sub get_callid {
    if ($udp_obj->{data} =~ /(Call-ID: )(.+)/){
      print "Call-ID: $2\n";
      return "$2";
    }
}

#
# Function that handles incoming packages and see if they are SIP messages
# or not.
#

sub got_a_packet {
    my $packet = shift;
    $ip_obj = NetPacket::IP->decode(eth_strip($packet));
    $udp_obj = NetPacket::UDP->decode($ip_obj->{data});
    # SIP message
    if ($udp_obj->{dest_port} == 5060){
      siphandler;
    }
}

#
# Main program
#

if ($myip eq "none") {
    print "Usage: perl sip-nemac.pl the_computers_ip.\n";
    exit;
} else {
print "\nProgram started ", `date`;
print  "**********************************************************\n\n";
```

```
$pktdesc = Net::PcapUtils::open(SNAPLEN => 1500, FILTER => 'udp', PROMISC => 0);
    my ($next_packet, %next_header);

    while(1){
      ($next_packet, %next_header) =  Net::PcapUtils::next($pktdesc);
      got_a_packet($next_packet);
    }
}
```

D.2 sip-nemac-client.pl

```
#!/usr/bin/perl -w
#
# sip-nemac-client.pl
# Emma Roos <hdm04ero@syd.kth.se>
#
# Client only version of sip-nemac.pl
#
# The program scans the network for SIP messages and when it discovers
# that a call is set up between this computer and a recipient it starts
# up the udp generator to generate monitor packets that are used by the
# server that is the party that runs the NeMaC clients
# (use sip-nemac-server.pl on the SIP server.)
#
# Start the program ./sip-nemac-client.pl <ip number> where the ip number
# is the one for the computer the program is running on.
#
# *Don't forget to start up NeTraMet before you start this program.*
# *       (NeTraMet -i eth0 -rread -wwrite -r public -m 50000)     *
#

use Net::PcapUtils;
use NetPacket::Ethernet qw(:strip);
use NetPacket::IP;
use NetPacket::UDP;


my $monitorpacketinterval = 1; # The time between monitoring packets
                               # (in seconds).
my $monitorpacketsize = 200; # The packet size (IP) of the monitoring packets
                             # (in bytes). The size should be the same as the
                             # size of the RTP packets.

my $myip = shift || "none";
my $destinationip;
my $callid = "none";

#
# Function that handles the incoming SIP messages
#

sub siphandler {
    print("$ip_obj->{src_ip}:$udp_obj->{src_port} -> ",
        "$ip_obj->{dest_ip}:$udp_obj->{dest_port} ",
        "$udp_obj->{len}\n");
    #
    # INVITE (Get the info needed for the NeMaC config file and create it)
    #
    if ($udp_obj->{data} =~ /^INVITE/){
```

```
      print "Incoming INVITE ";
      if ($ip_obj->{src_ip} eq $myip){
          if ($udp_obj->{data} =~ /Contact:
([^@]+)@(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/){
            my $tempip = "$2";
            if ($tempip eq $myip) {
                print "from this computer.\n";
                $callid = get_callid();
            } else {
                print "from another computer (this is the SIP server).\n";
                get_callid();
            }
          }
      } else {
          print "from another computer.\n";
          get_callid();
      }
    }
    #
    # 180 Ringing
    #
    elsif ($udp_obj->{data} =~ /^SIP.2.0 180/) {
      print "It's ringing at the callee.\n";
      my $thiscallid = get_callid();
      if ($callid eq $thiscallid) {
          if ($udp_obj->{data} =~ /(Contact:
)([^@]+)@(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/){
            $destinationip = "$3";
            print "Mottagarens IP: ", $destinationip, "\n";
          }
      }
    }
    #
    # 200 OK
    #
    elsif ($udp_obj->{data} =~ /^SIP.2.0 200/) {
        # The called party has "picked up". (Start NeMac and the UDP generator)
      if ($udp_obj->{data} =~ /CSeq: \d+ INVITE/) {
          print "Conversation has started.\n";
          my $thiscallid = get_callid();
          my $udpgenrate = ($monitorpacketsize * 8)/$monitorpacketinterval;
          if ($callid eq $thiscallid) {
            $udpgenpid = open(UDPGEN, "ngen -proto udp -dstport 7 -rate
$udpgenrate -size $monitorpacketsize -dstip $destinationip|");
          }
      }
        # The call had ended (message comes after the BYE message
      elsif ($udp_obj->{data} =~ /CSeq: \d+ BYE/) {
          print "The conversation is ending.\n";
          my $thiscallid = get_callid();
          if ($callid eq $thiscallid) {
            # This is where NeMaC and the udp generator should stop.
            # But because of a bug in Linphone I've moved this to the
            # handling of BYE message that initiate the tear down of the call.
          }
      }
      # Registering with server was successful
      elsif ($udp_obj->{data} =~ /CSeq: \d+ REGISTER/) {
          if ($udp_obj->{data} =~ /Expires: 0/) {
```

```
        print "User unregistered from server!\n";
      } else {
        print "User registered with server!\n";
      }
  }
  elsif ($udp_obj->{data} =~ /CSeq: \d+ CANCEL/) {
      print "Call cancelled successfully.\n";
      get_callid();
  } else {
      print "Another 200 OK\n";
  }
}
# The call is over. (Stop NeMaC and the UDP generator)
elsif ($udp_obj->{data} =~ /^BYE/){
  print "BYE.\n";
  my $thiscallid = get_callid();
  if ($callid eq $thiscallid) {
      `kill $udpgenpid`;
      close(UDPGEN);
      $callid = "none";
  }
}
# 100 Trying
elsif ($udp_obj->{data} =~ /^SIP.2.0 100/){
  print "100 Trying.\n";
  get_callid();
}
# The called party doesn't want to or can't answer
elsif ($udp_obj->{data} =~ /^SIP.2.0 (486|480|603)/)
{
  print "The recipient is unaccessible/busy.\n";
  get_callid();
}
# The caller "hangs up" before the call is established.
elsif ($udp_obj->{data} =~ /(^CANCEL)/)
{
  print "Call is cancelled by the caller before conversation had started.\n";
  get_callid();
}
# 487 Request Cancelled
elsif ($udp_obj->{data} =~ /^SIP.2.0 487/) {
  print "487 Request Cancelled\n";
  get_callid();
}
# ACK messages
elsif ($udp_obj->{data} =~ /^ACK/) {
  print "ACK.\n";
  get_callid();
}
# REGISTER messages
elsif ($udp_obj->{data} =~ /^REGISTER/) {
  if ($udp_obj->{data} =~ /Expires: 0/)
  {
      if ($udp_obj->{data} =~ /To\: \<sip\:([^\>]+)\>/){
        print "User $1 has closed their client program.\n";
      }

  } else {
      if ($udp_obj->{data} =~ /To\: \<sip\:([^\>]+)\>/) {
```

```
            print "User $1 is registering with server.\n";
          }
      }
    }
    # 401 Unauthorized
    elsif ($udp_obj->{data} =~ /^SIP.2.0 (401)/) {
      if ($udp_obj->{data} =~ /CSeq: \d+ REGISTER/) {
          print "Password needed for registering with server!\n";
      } else {
          print "401 Unauthorized\n";
      }
    }
    # 404 Not Found
    elsif ($udp_obj->{data} =~ /^SIP.2.0 (404)/) {
      if ($udp_obj->{data} =~ /CSeq: \d+ REGISTER/) {
          print "User not found in server database!\n";
      }
      elsif ($udp_obj->{data} =~ /CSeq: \d+ INVITE/) {
          print "User not registered with server/Doesn't exist on server!\n";
          get_callid();
      } else {
      print "404 Not Found\n";
        }
    }
    elsif ($udp_obj->{data} =~ /^SIP.2.0 (101)/) {
      print "101 Dialog Establishment\n";
    }
    # Other SIP messages
    else {
      print "$udp_obj->{data}\n";
    }
    print "\n**********************************************************\n\n";
}

sub get_callid {
    if ($udp_obj->{data} =~ /(Call-ID: )(.+)/){
      print "Call-ID: $2\n";
      return "$2";
    }
}

#
# Function that handles incoming packages and see if they are SIP messages
# or not.
#

sub got_a_packet {
    my $packet = shift;
    $ip_obj = NetPacket::IP->decode(eth_strip($packet));
    $udp_obj = NetPacket::UDP->decode($ip_obj->{data});
    # SIP message
    if ($udp_obj->{dest_port} == 5060){
      siphandler;
    }
}



#
```

```
# Main program
#

print "\nProgram started ", `date`;
print "************************************************************\n\n";

if ($myip eq "none") {
    print "Usage: perl sip-nemac-client.pl the_computers_ip.\n";
    exit;
} else {

    $pktdesc = Net::PcapUtils::open(SNAPLEN => 1500, FILTER => 'udp', PROMISC =>
0);
    my ($next_packet, %next_header);

    while(1){
      ($next_packet, %next_header) =  Net::PcapUtils::next($pktdesc);
      got_a_packet($next_packet);
    }
}
```

D.3 sip-nemac-server.pl

```
#!/usr/bin/perl -w
#
# sip-nemac-server.pl
# Emma Roos <hdm04ero@syd.kth.se>
#
# Server version of sip-nemac.pl
#
# This program scans the network for SIP messages and then uses
# information from those to configure and start up NeMaC to record
# traffic statistics from the different calls that are started.
#
# The recording of the traffic statistics does only start if the ip
# sent as an attribute to this program is the ip of the computer that
# starts the call. You must run one instance of this program per computer
# you want to record statistics for.
#
# Start the program ./sip-nemac-server.pl <ip number> where the ip number is
# is the one for the computer you want to record traffic statistics for.
# This computer must run sip-nemac-client.pl
#
# (*Don't forget to start up NeTraMet before you start this program.*
# *       (NeTraMet -i eth0 -rread -wwrite -r public -m 50000)     *)
#

use Net::PcapUtils;
use NetPacket::Ethernet qw(:strip);
use NetPacket::IP;
use NetPacket::UDP;


my $monitorpacketinterval = 1; # The time between monitoring packets
                               # (in seconds).
my $collectinterval = 30; # The interval between NeMaC collecting statistics

my $myip = shift || "none";
my $sourceip;
```

```
my $sourceport;
my $destinationip;
my $nemacconf;
my $callid = "none";

#
# Function to create the srl file with the current calls settings
#

sub createconf {

    my $nemacsrl = "$nemacconf.srl";

    #Create the SRL file

    open CONFFILE,'>', $nemacsrl;

    print CONFFILE "define OAM_SENDER = ", $sourceip, ";\n";
    print CONFFILE "define OAM_PROTOCOL = udp;\n";
    print CONFFILE "define OAM_SENDER_PORT = 8888;\n";
    print CONFFILE "define DATA_SENDER_IP = ", $sourceip, ";\n";
    print CONFFILE "define DATA_SENDER_PORT = ", $sourceport, ";\n\n";

    print CONFFILE "if SourcePeerType == IPv4 save;\n else ignore;\n\n";

    print CONFFILE "if SourcePeerAddress == OAM_SENDER\n";
    print CONFFILE "&& SourceTransType == OAM_PROTOCOL\n";
    print CONFFILE "&& SourceTransAddress == OAM_SENDER_PORT\n";
    print CONFFILE "&& MatchingStoD == 1  {\n";
    print CONFFILE "store FlowKind := 11;\n";
    print CONFFILE "save SourcePeerAddress = 1.0/8;\n";
    print CONFFILE "save OAMdata = 1.0.0!0 & 0.0.0!0;\n";
    print CONFFILE "store OAMident := 1;\n";
    print CONFFILE "count;\n }\n\n";

    print CONFFILE "else if DestTransType == OAM_PROTOCOL\n";
    print CONFFILE "&& DestTransAddress == OAM_SENDER_PORT\n";
    print CONFFILE "&& DestPeerAddress == OAM_SENDER\n";
    print CONFFILE "&& MatchingStoD == 1  {\n";
    print CONFFILE "store FlowKind := 22;\n";
    print CONFFILE "save SourcePeerAddress = 2.0/8;\n";
    print CONFFILE "save OAMdata = 2.0.0!0 & 0.0.0!0;\n";
    print CONFFILE "store OAMident := 2;\n";
    print CONFFILE "count;\n }\n";

    print CONFFILE "else if SourceTransType == UDP\n";
    print CONFFILE "&& SourcePeerAddress == DATA_SENDER_IP\n";
    print CONFFILE "&& SourceTransAddress == DATA_SENDER_PORT save, {\n";
    print CONFFILE "store FlowKind := 97;\n";
    print CONFFILE "save SourceTransAddress; \n";
    print CONFFILE "save DestTransAddress;  \n";
    print CONFFILE "save SourcePeerAddress;\n";
    print CONFFILE "save DestPeerAddress;\n";
    print CONFFILE "store OAMident := 1;\n";
    print CONFFILE "count; \n}\n";

    print CONFFILE "else if DestTransType == UDP\n";
    print CONFFILE "&& DestPeerAddress == DATA_SENDER_IP\n";
    print CONFFILE "&& DestTransAddress == DATA_SENDER_PORT save, {\n";
```

```
    print CONFFILE "store FlowKind := 98;\n";
    print CONFFILE "save SourceTransAddress;\n";
    print CONFFILE "save DestTransAddress;  \n";
    print CONFFILE "save SourcePeerAddress;\n";
    print CONFFILE "save DestPeerAddress;\n";
    print CONFFILE "store OAMident := 2;\n";
    print CONFFILE "count;\n }\n";

    print CONFFILE "else ignore;\n";

    print CONFFILE "set rtp_test;\n";
    print CONFFILE "format\n";
    print CONFFILE "SourceKind DestKind FlowKind \"  \"\n";
    print CONFFILE "SourcePeerType SourcePeerAddress DestPeerAddress \"  \"\n";
    print CONFFILE "SourceTransType SourceTransAddress DestTransAddress\n";
    print CONFFILE "ToPDUs FromPDUs \"  \" ToOctets FromOctets \"{{\" OAMdata
\"}}\";\n";

    close CONFFILE;

    #Compile the SRL file
    system ("/usr/local/bin/srl", $nemacsrl);
};

#
# Function that handles the incoming SIP messages
#

sub siphandler {
    print("$ip_obj->{src_ip}:$udp_obj->{src_port} -> ",
        "$ip_obj->{dest_ip}:$udp_obj->{dest_port} ",
        "$udp_obj->{len}\n");
    #
    # INVITE (Get the info needed for the NeMaC config file and create it)
    #
    if ($udp_obj->{data} =~ /^INVITE/){
      print "Incoming INVITE ";
      if ($ip_obj->{src_ip} eq $myip){
          if ($udp_obj->{data} =~ /Contact:
([^@]+)@(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/){
            my $tempip = "$2";
            if ($tempip eq $myip) {
                print "from this computer.\n";
                $callid = get_callid();
                if ($callid =~ /([^@]+)@(.+)/) {
                  $nemacconf = "$1";
                } else {
                  $nemacconf = "$callid";
                }
                $sourceip = $tempip;
                print "The callers IP: ", $sourceip, "\n";
                if ($udp_obj->{data} =~ /(m=audio )(\d+)/)
                {
                  $sourceport ="$2";
                  print "The callers RTP-port: ", $sourceport, "\n\n";
                }
                createconf;
            } else {
                print "from another computer (this is the SIP server).\n";
```

```
                get_callid();
              }
            }
      } else {
          print "from another computer.\n";
          get_callid();
      }
    }
    #
    # 180 Ringing
    #
    elsif ($udp_obj->{data} =~ /^SIP.2.0 180/) {
      print "It's ringing at the callee.\n";
      my $thiscallid = get_callid();
      if ($callid eq $thiscallid) {
          if ($udp_obj->{data} =~ /(Contact:
)([^@]+)@(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/){
            $destinationip = "$3";
            print "Mottagarens IP: ", $destinationip, "\n";
          }
      }
    }
    #
    # 200 OK
    #
    elsif ($udp_obj->{data} =~ /^SIP.2.0 200/) {
        # The called party has "picked up". (Start NeMac and the UDP generator)
      if ($udp_obj->{data} =~ /CSeq: \d+ INVITE/) {
          print "Conversation has started.\n";
          my $thiscallid = get_callid();
          if (($callid eq $thiscallid) && ($ip_obj->{src_ip} eq $destinationip)) {
            $nemacsrcpid = open(NEMACSRC, "NeMaC -c $collectinterval -g 120 -m
50000 -p -F $nemacconf.src.$sourceip.flow -L $nemacconf.log -r $nemacconf.rules
$sourceip write|");
            $nemacdstpid = open(NEMACDST, "NeMaC -c $collectinterval -g 120 -m
50000 -p -F $nemacconf.dst.$destinationip.flow -L $nemacconf.log -r
$nemacconf.rules $destinationip write|");
          }
      }
        # The call had ended (message comes after the BYE message
      elsif ($udp_obj->{data} =~ /CSeq: \d+ BYE/) {
          print "The conversation is ending.\n";
          my $thiscallid = get_callid();
          if ($callid eq $thiscallid) {
            # This is where NeMaC and the udp generator should stop.
            # But because of a bug in Linphone I've moved this to the
            # handling of BYE message that initiate the tear down of the call.
          }
      }
      # Registering with server was successful
      elsif ($udp_obj->{data} =~ /CSeq: \d+ REGISTER/) {
          if ($udp_obj->{data} =~ /Expires: 0/) {
            print "User unregistered from server!\n";
          } else {
            print "User registered with server!\n";
          }
      }
      elsif ($udp_obj->{data} =~ /CSeq: \d+ CANCEL/) {
          print "Call cancelled successfully.\n";
```

```
      get_callid();
  } else {
      print "Another 200 OK\n";
  }
}
# The call is over. (Stop NeMaC and the UDP generator)
elsif ($udp_obj->{data} =~ /^BYE/){
  print "BYE.\n";
  my $thiscallid = get_callid();
  if ($callid eq $thiscallid) {
      `kill $nemacsrcpid $nemacdstpid`;
      close(NEMACSRC);
      close(NEMACDST);

      $callid = "none";
      system ("rm -f $nemacconf.srl $nemacconf.rules");
  }
}
# 100 Trying
elsif ($udp_obj->{data} =~ /^SIP.2.0 100/){
  print "100 Trying.\n";
  get_callid();
}
# The called party doesn't want to or can't answer
elsif ($udp_obj->{data} =~ /^SIP.2.0 (486|480|603)/)
{
  print "The recipient is unaccessible/busy.\n";
  my $thiscallid = get_callid();
  if ($callid eq $thiscallid) {
      system ("rm -f $nemacconf.srl $nemacconf.rules");
      $callid = "none";
  }
}
# The caller "hangs up" before the call is established.
elsif ($udp_obj->{data} =~ /(^CANCEL)/)
{
  print "Call is cancelled by the caller before conversation had started.\n";
  my $thiscallid = get_callid();
  if ($callid eq $thiscallid) {
      system ("rm -f $nemacconf.srl $nemacconf.rules");
      $callid = "none";
  }
}
# 487 Request Cancelled
elsif ($udp_obj->{data} =~ /^SIP.2.0 487/) {
  print "487 Request Cancelled\n";
}
# ACK messages
elsif ($udp_obj->{data} =~ /^ACK/) {
  print "ACK.\n";
  get_callid();
}
# REGISTER messages
elsif ($udp_obj->{data} =~ /^REGISTER/) {
  if ($udp_obj->{data} =~ /Expires: 0/)
  {
      if ($udp_obj->{data} =~ /To\: \<sip\:([^\>]+)\>/){
        print "User $1 has closed their client program.\n";
      }
```

```
      } else {
          if ($udp_obj->{data} =~ /To\: \<sip\:([^\>]+)\>/) {
            print "User $1 is registering with server.\n";
          }
      }
    }
    # 401 Unauthorized
    elsif ($udp_obj->{data} =~ /^SIP.2.0 (401)/) {
      if ($udp_obj->{data} =~ /CSeq: \d+ REGISTER/) {
          print "Password needed for registering with server!\n";
      } else {
          print "401 Unauthorized\n";
      }
    }
    # 404 Not Found
    elsif ($udp_obj->{data} =~ /^SIP.2.0 (404)/) {
      if ($udp_obj->{data} =~ /CSeq: \d+ REGISTER/) {
          print "User not found in server database!\n";
      } elsif ($udp_obj->{data} =~ /CSeq: \d+ INVITE/) {
          print "User not registered with server/Doesn't exist on server!\n";
          my $thiscallid = get_callid();
          if ($callid eq $thiscallid) {
            system ("rm -f $nemacconf.srl $nemacconf.rules");
            $callid = "none";
          }
      } else {
          print "404 Not Found\n";
      }
    }
    elsif ($udp_obj->{data} =~ /^SIP.2.0 (101)/) {
      print "101 Dialog Establishment\n";
    }
    # Other SIP messages
    else {
      print "$udp_obj->{data}\n";
    }
    print "\n**************************************************************\n\n";
}

sub get_callid {
    if ($udp_obj->{data} =~ /(Call-ID: )(.+)/){
      print "Call-ID: $2\n";
      return "$2";
    }
}

#
# Function that handles incoming packages and see if they are SIP messages
# or not.
#

sub got_a_packet {
    my $packet = shift;
    $ip_obj = NetPacket::IP->decode(eth_strip($packet));
    $udp_obj = NetPacket::UDP->decode($ip_obj->{data});
    # SIP message
    if ($udp_obj->{dest_port} == 5060){
      siphandler;
```

```
    }
}

#
# Main program
#

print "\nProgram started ", `date`;
print  "************************************************************\n\n";

if ($myip eq "none") {
    print "Usage: perl sip-nemac.pl the_computers_ip.\n";
    exit;
} else {

    $pktdesc = Net::PcapUtils::open(SNAPLEN => 1500, FILTER => 'udp', PROMISC =>
1);
    my ($next_packet, %next_header);

    while(1){
      ($next_packet, %next_header) =  Net::PcapUtils::next($pktdesc);
      got_a_packet($next_packet);
    }
}
```